DISTRIBUTED SYSTEMS FOR SYSTEM ARCHITECTS

ADVANCES IN DISTRIBUTED COMPUTING AND MIDDLEWARE

Consulting Editors

Prof. John A. Stankovic
University of Virginia
Dept of Computer Science
Charlottesville, VA 22903-2442
stankovic@cs.virginia.edu

Dr. Richard E. Schantz
Principal Scientist
BBN Technologies
Cambridge, MA 02138
schantz@bbn.com

DISTRIBUTED SYSTEMS FOR SYSTEM ARCHITECTS

by

Paulo Veríssimo
University of Lisboa, Portugal

Luís Rodrigues
University of Lisboa, Portugal

KLUWER ACADEMIC PUBLISHERS
Boston / Dordrecht / London

Distributors for North, Central and South America:
Kluwer Academic Publishers
101 Philip Drive
Assinippi Park
Norwell, Massachusetts 02061 USA
Telephone (781) 871-6600
Fax (781) 681-9045
E-Mail <kluwer@wkap.com>

Distributors for all other countries:
Kluwer Academic Publishers Group
Distribution Centre
Post Office Box 322
3300 AH Dordrecht, THE NETHERLANDS
Telephone 31 78 6392 392
Fax 31 78 6392 254
E-Mail <services@wkap.nl>


Electronic Services <http://www.wkap.nl>

Library of Congress Cataloging-in-Publication Data

Veríssimo, Paulo, 1956-
Distributed systems for system architects / Paulo Veríssimo, Luís Rodrigues.
p. cm.--(Advances in distributed computing and middleware; dist1)
Includes bibliographical references and index.
ISBN 0-7923-7266-2 (alk. paper)
1. Electronic data processin--Distributed processing. I. Rodrigues, Luís, 1963- II. Title. III. Series.

QA76.9.D5 V45 2000
005'.36—dc21 00-052178

Printed on acid-free paper.

Printed in the United States of America

À Luísa, ao Tiago e ao Vasco,
por esse tempo,
em que o tempo era largo

To my parents, Vasco and Lurdes,
my sister, Elsa,
my wife, Ana,
and my children, Hugo and Sara

Contents

The primary audience for this book are advanced undergraduate students and graduate students. Computer architecture, as it happened in other fields such as electronics, evolved from the small to the large, that is, it left the realm of low-level hardware constructs, and gained new dimensions, as distributed systems became the keyword for system implementation. As such, the *system architect*, today, assembles pieces of hardware that are at least as large as a computer or a network router or a LAN hub, and assigns pieces of software that are self-contained, such as client or server programs, Java applets or protocol modules, to those hardware components. The freedom she/he now has, is tremendously challenging. The problems alas, have increased too. What was before mastered and tested carefully before a fully-fledged mainframe or a closely-coupled computer cluster came out on the market, is today left to the responsibility of computer engineers and scientists invested in the role of system architects, who fulfil this role on behalf of software vendors and integrators, add-value system developers, R&D institutes, and final users. As system complexity, size and diversity grow, so increases the probability of inconsistency, unreliability, non responsiveness and insecurity, not to mention the management overhead.

What System Architects Need to Know

The insight such an architect must have includes but goes well beyond, the functional properties of distributed systems. Most of the problems in configuring, deploying and managing a distributed system come not from what the system *does* that we have not understood, but from what the system *is not* that we have overlooked, that is, from inappropriate non-functional properties: unreliability, lack of responsiveness, insecurity. In other words, *fault tolerance, real-time, security*, are fundamental but sometimes neglected attributes of distributed systems. The mastery of the relevant concepts and techniques is as important as that of the issues related with distribution itself. Finally, it is necessary to understand the *management* of systems with this complexity and versatility. Together, these issues form the body of knowledge that this book

intends to pass on to future **distributed system architects**. A book covering all these issues risks being either too long or too shallow. That would happen if the subjects were treated as if the book were a collection of smaller books dedicated to each topic. Most of the existing books on distributed systems are addressed to system programmers, or operating system designers. However, the knowledge a system architect must have is different.

The system architect must first understand the *fundamental concepts* and the most important *paradigms* concerned with the problem of distribution. Then, once presented with the main *models* of distributed systems and with the problems posed by them, such as how to implement a given feature, or how to overcome a certain limitation, she/he will have the background to understand the architecture of the solutions, in a logical composition of building blocks, structure, techniques and algorithms. Specific *systems* or sub-systems whose architecture, protocols and modules will be discussed, provide the pretext for the student to integrate all the material she/he has been exposed to. Finally, the architect has the opportunity to create her/his own architecture, in a *case study* that develops along the book.

The next thing that singles out this book's structure is the way that the **fault tolerance, real-time, security** and **management** parts are treated, once more addressing architects of distributed systems. Most of the existing books specializing on each of the above topics do so in a thorough but horizontal way. Here, the student will see a continuity in the style of addressing each of these matters, and an integration with distribution. In fact, each of the following parts is also organized as: concepts; paradigms; models; and systems. Besides, the contents refer to the basic matters given in the Distributed Systems part in a problem-oriented manner.

This is further emphasized by the case study, an imaginary wine company with facilities spread through the country, whose information system, the *VintagePort'63 System (VP'63)*, must adapt to the modern times. The case study is methodically addressed at the end of each part, so that we progressively make VP'63: (1) modular, distributed and interactive; (2) dependable; (3) timely; (4) secure, and (5) manageable.

Finally, at the end of each part there is a repository of web URL's linking to most of the systems discussed. All URLs were functional at the time of print. This is a risky endeavour for a printed book, in such a fast changing world. However, we took the risk, we believe our effort will save a lot of precious time to many students and readers, and will provide them with a useful database of over 250 contacts that they can themselves improve and update as time goes by.

Student information

The book provides solutions to the following general problems:

- Advanced undergraduate students need be exposed to all these subjects, but one cannot generally afford one course per theme at this level, so this book provides the teacher with an integrated and homogeneous textbook.

- Graduate students (MSc) may either take advantage from having a single book for several thematic introductory courses, and/or from using the book as a bootstrap text for more in-depth, focused courses, complemented with research papers.

Parts of this book will thoroughly cover the subjects needed for an advanced undergraduate course on distributed system architecture, to be preceded by an introductory course on computer networks and distributed operating systems. It may be used to teach introductory courses on any combination of *"Introduction to-"* fault tolerance, real-time, or security, both at advanced undergraduate or graduate level (e.g. *Fault-tolerant Real-time Systems*). It may be used to teach more focused graduate courses on distributed systems, fault-tolerance, real-time, security, or management, complemented with dedicated books or research papers (e.g. *Secure and Reliable Web Systems*, *Configuration and Management of Distributed Systems*).

How to use the book

Undergraduate Teaching

Part I provides the support for teaching distributed system architectures, admitting the students had an introductory operating systems course, whose depth will dictate how much of Chapter 1 is given. Some topics from Chapter 2 may be omitted (addressing all sections is however suggested). Selected parts of Chapters 3 and 4 consolidate the basic notions. The sections on Client-Server, WWW and Group-oriented in Chapter 3, and DCE in Chapter 4 are strongly suggested. Chapter 5 is an excellent pretext and inspiration for assignments. If enough credits are devoted to this area, the teacher may expand the example course just suggested, or split it, by further addressing Chapters 6, 16, 21 and 11, in order to provide complementary introductory notions of fault-tolerance, security, management and real-time, by order of priority.

Postgraduate teaching

Part I in its entirety provides a thorough understanding of distributed system architecture, admitting the students had previous introductory notions in the area (for example the undergraduate course just suggested), so that they can go directly to the in-depth issues of Chapters 2 through 5. Each of Parts II to V, or combinations thereof, may be used to teach courses on the relevant themes. It is advisable to start with a review of selected parts of Chapter 2, and start the case study with Chapter 5, for completeness.

Self-study

The book may be of assistance for support of advanced research studies, both as a broad body of reference in the disciplines of distributed systems, and as a pointer to deeper study by means of the bibliography of each part.

Support

The book readers, students and teachers will find some support in the book web page, www.navigators.di.fc.ul.pt/dssa. Web copies of the URL tables will be kept as up-to-date as possible. Electronic versions of all figures will be made available to teachers by specific request. A mailbox is available on the page for requests, suggestions and questions.

Guided Tour of the Book

Part I, Distribution, addresses the fundamental issues concerning distribution, and it is the largest part of the book. It contains a comprehensive set of notions that will develop in the reader a thorough understanding of distributed system architecture, from concepts and paradigms, to models and example systems. Chapter 1, Distributed Systems Foundations, discusses the foundations of distributed systems, and is intended as a review of the basic subjects regarding distribution, such as computer networks, distributed operating systems and services, complemented with a few formal notions, useful for a more elaborate treatment of some subjects. Distributed system architectures are given from a evolutionary perspective, from remote login to mobile computing, so that the reader, further to understanding what the several architectural models are, captures why they appeared or mutated, and what needs each one serves. Inasmuch as History is paramount to Architecture, so is the knowledge of computing systems evolution to the system architect. Chapter 2, Distributed System Paradigms, presents the most important paradigms in distributed systems, in a problem-oriented manner, purposely addressed to to-be architects. That is, rather than being exposed to the subjects in a paradigm-centric manner, enveloped in some formal description, the reader is faced with a problem or a need, then with a solution in the form of a paradigm, and when appropriate, with details about relevant mechanisms or algorithms. And finally, should it be the case, the limitations of that paradigm may also be pointed out, so that another paradigm, solving the problem, is motivated, and so forth. Namely, the chapter addresses: message passing, remote operations, group communication, naming and addressing, time and clocks, ordering, synchrony, coordination, concurrency, and consistency. Chapter 3, Models of Distributed Computing, discusses the main models used nowadays in distributed systems, that is: what are the main classes of distributed activities; why different models serve different needs, and how we design the software architecture and structure the run-time environment of distributed applications. The chapter explains clearly the main reasons for the known debate between the synchronous and asynchronous frameworks for distributed computing. Then, it addresses known models such as: client-server with RPC, group-oriented, World-Wide Web, distributed shared memory, message-buses. Chapter 4, Distributed Systems and Platforms, consolidates the notions learnt along the previous chapters, in the form of examples of enabling technologies, toolboxes, platforms and systems. The last two chapters do not address system-call details or system in-

ternals, since the scope of the book is designing and building systems, rather than programming them. Chapter 5 starts a case study: The VP'63 (Vintageport'63) Large-Scale Information System. An imaginary Portuguese wine company, with facilities spread through the country, has a traditional information system, the VintagePort'63 (VP'63), that must adapt to the modern times. Centralized, mainframe-based, little interactivity, proprietary, it must adapt to the distributed nature of the company and its distribution network, and to the business evolution. The case study is methodically addressed at the end of each part, so that we progressively solve the above-mentioned problems, making VP'63: modular, distributed and interactive; dependable; timely; and secure.

Part II, Fault-Tolerance, addresses dependability of distributed systems, that is, how to ensure that they keep running correctly. It contains the fundamental notions concerning dependability, such as the trilogy fault-error-failure and provides a comprehensive treatment of distributed fault-tolerance. Chapter 6, Fundamental Concepts of Fault-Tolerance, starts with the generic notion of dependability and its associated concepts, and ends with the introduction of distributed fault-tolerance. In fact, distribution and fault-tolerance go hand in hand, since the former requires the latter to keep reliability at an acceptable level, and the latter is made easier by some qualities of the former, such as independence of failure of individual machines. Chapter 7, Paradigms for Distributed Fault-Tolerance, discusses the main paradigms of this discipline. After introductory concepts and notions about fault-tolerant communication, it addresses issues such as: replication management, resiliency and voting, and recovery. Chapters 8 and 9, Models of Distributed Fault-Tolerant Computing and Dependable Systems and Platforms, show how to incorporate fault-tolerance in distributed systems. Explaining the main strategies for the diverse fault models, its materialization in discussed for remote operation, diffusion and transactional computing models. Finally, examples of relevant systems are given. Chapter 10 continues the case study: Making the VP'63 System Dependable.

Part III, Real-Time, takes the same explanatory approach of Part II, and discusses how to ensure that systems are timely. It contains the fundamental notions concerning real-time, and provides a comprehensive treatment of the problem of real-time in distributed systems. Chapters 11 and 12, Fundamental Concepts of Real-Time and Paradigms for Real-Time, address the fundamental notions and misconceptions about real-time, in a distributed context. The main paradigms are presented, in a comparative manner when applicable, such as synchronism versus asynchronism, or event- versus time-triggered operation. Chapter 12 further addresses issues such as: real-time networks, real-time processing, real-time communication, clock synchronization, and input-output. Chapters 13 and 14, Models of Distributed Real-Time Computing and Real-Time Systems and Platforms show how to achieve timeliness of distributed systems, in its several forms, from the hard, soft or best-effort real-time classes, to the time-triggered and event-triggered models. Chapter 14 gives examples

of distributed real-time systems in several settings. Chapter 15 continues the case study: Making the VP'63 System Timely.

Part IV, Security, addresses security of distributed systems, that is, how to ensure that they resist intruders. Security is paramount to the recognition of open distributed systems as the key technology in today's global communication and processing scenario. This part contains the fundamental notions concerning security, and provides a comprehensive treatment of the problem of security in distributed systems. Chapter 16, Fundamental Concepts of Security, discusses the fundamental principles, such as the notions of risk, threat and vulnerability, and the properties of confidentiality, authenticity, integrity and availability. Chapter 17, Fundamental Security Paradigms, treats the most important paradigms, such as: cryptography, digital signature and payment, secure networks and communication, protection and access control, firewalls, auditing. Chapters 18 and 19, Models of Distributed Secure Computing, and Secure Systems and Platforms, consolidate the notions of the previous chapters, in the form of models and systems for building and achieving: information security, authentication, electronic transactions, secure channels, remote operations and messaging, intranets and firewall systems, extranets and virtual private networks. Chapter 20 continues the case study, this time: Making the VP'63 Secure.

Last but not least, Part V on Management, because distributed systems are too complex to be managed ad-hoc. In essence, there is a contradiction in the nature of the problem. Distributed systems are geographically spread. They have a large number of visible— and thus manageable— components, from computers, routers, modems, and network media, to programs, operating systems, protocols, etc. However, whilst some of the components can be self or locally managed, thus in a distributed fashion, system management is human-centric, and by nature centralized. The book does not intend to give a magic recipe for this problem, which is still an active area of research, but will give the reader the ability to understand it and become aware of the existing solutions for it. Chapters 21 and 22, Fundamental Concepts of Management, and Paradigms for Distributed Systems Management, give insight on the fundamental concepts, architectures and paradigms concerning network and distributed systems configuration and management. Chapter 22 presents the main management functions: fault, configuration, accounting, performance, security, quality-of-service, name and directories, and monitoring. Chapters 23 and 24, Models of Network and Distributed Systems Management and Management Systems and Platforms, discuss the main models, such as centralized, decentralized, integrated, and domain-oriented management, and point to examples of tools, systems and architectures. Chapter 25 finalizes the case study: Managing the VP'63 System.

Acknowledgments

A number of people, some not knowingly, contributed to this book. It was from lecturing advanced undergraduate and graduate courses and working together

with system architects in international distributed systems projects for the past few years, that we came to figure out what a distributed system architect should know, clearly enough (so we hope) to cast it in a book. Students and colleagues at the Technical University of Lisboa (IST) and more recently at the University of Lisboa (FCUL), and projects with the Navigators group, such as the ESPRIT DELTA-4, DINAS or BROADCAST were marvellous thinking tanks.

Nevertheless, there are some people who, for their direct influence on the book, deserve a very warm acknowledgment. In alphabetic order, and hoping not to forget anyone: Lorenzo Alvisi, Alan Burns, António Casimiro, Miguel Correia, Yves Deswarte, Elmootazbellah Elnozahy, Matti Hiltunen, Joerg Kaiser, Sacha Krakowiak, Jean-Claude Laprie, Jeff Magee, Pedro Martins, Roger Needham, Nuno Ferreira Neves, Nuno Miguel Neves, Guevara Noubir, David Powell, Brian Randell, Peter Ryan, Rick Schlichting, Robert Stroud, Morris Sloman, Neeraj Suri, Irfan Zakiuddin. This passable piece of text would be unintelligible without their help. The comments and feedback from the reviewers were also an enormous help and incentive. Our students at FCUL detected many typos and several less clear parts.

To some fellow architects goes the final word of appreciation for several years of many chats, discussions and common projects: Ken Birman, Flaviu Cristian, Hermann Kopetz, David Powell, and Rick Schlichting.

PAULO VERÍSSIMO, LUÍS RODRIGUES

LISBOA, AUGUST 2000

Foreword

Developers tasked with architecting a functional and dependable system out of a collection of machines connected by a communications network—i.e., a distributed system—face enormous challenges. Largely because of a loosely coupled hardware architecture with no physically shared memory, many things that are straightforward in centralized systems are difficult in distributed systems. For example, synchronizing processes with separate threads of control typically uses shared variables on a single machine, but must be done with message passing in a distributed system. The extra time delay associated with sending messages over a network increases the asynchrony of the processes and necessitates the use of special protocols to coordinate their respective actions.

Perhaps the most serious open issue in building such systems relates to ensuring what are sometimes called non-functional attributes: reliability, availability, timeliness, security. For example, providing just the first two attributes requires systems architects to deal not just with normal operation, but also failures that may have arbitrary effects on only parts of the system for an unpredictable duration. Timeliness and security are similarly challenging. To provide the predictable timing behavior needed to build a real-time distributed system requires controlling literally every part of the system, from the hardware through the system software to the application. To guarantee a secure computing environment that ensures confidentiality, integrity and privacy requires, among other things, sophisticated mathematical cryptographic techniques and the ability to do subtle analysis. Even worse, for systems that need *all* of these attributes—as is increasingly the case—the challenges are combinatorial, not additive. The designer can take techniques to guarantee availability and combine them with techniques to guarantee security and easily end up with a system that provides neither.

In this book, Paulo Veríssimo and Luís Rodrigues provide a comprehensive and timely treatment of all these challenges. In a clear and consistent way, they address the fundamental characteristics of such systems and tackle the issues involved in providing service that is fault tolerant, can meet real-time guarantees, and is secure. They also address management issues, which are a

non-trivial and often neglected part of the problem. In each case, they focus on the fundamental paradigms associated with that area, describe the building blocks—both conceptual and practical—needed to address the issues, and give concrete examples of existing systems that incorporate state-of-the-art solutions. The presentation in each section of an evolving case study involving a hypothetical Portuguese wine producer makes the book even more valuable by providing a consistent context for discussing issues and for demonstrating the subtleties that arise when systems have to ensure multiple attributes.

In writing this book, the authors bring to bear a wealth of expertise and experience, not just in research aspects of the problem, but also as practical systems architects. Given the challenges involved, no one is better equipped to guide the reader through the intricacies of the issues and the details of the techniques needed to realize the vision of highly dependable distributed systems.

Rick Schlichting
AT&T Labs - Research,
Florham Park, New Jersey, USA
22 September 2000

I Distribution

A distributed system is the one that prevents you from working because of the failure of a machine that you had never heard of.

— Leslie Lamport

Contents

Overview

Part I addresses the fundamental issues concerning distribution, providing a generic set of notions about the architecture of distributed systems. Chapter 1, Distributed Systems Foundations, discusses the basic subjects regarding distribution, and reviews desirable background such as computer networks and distributed operating systems. The evolution of distributed system architectures is presented, from remote login to mobile computing. Chapter 2, Distributed System Paradigms, presents the most important paradigms in distributed systems, in a problem-oriented manner: the reader is faced with a problem or a need, then with a solution in the form of a paradigm. Chapter 3, Models of Distributed Computing, discusses the main models used nowadays in distributed systems. Chapter 4, Distributed Systems and Platforms, consolidates the notions learnt along the previous chapters, in the form of examples of enabling technologies, toolboxes, platforms and systems. Chapter 5 starts a case study: the obsolete information system of an imaginary wine company with facilities spread through the country must adapt to the modern times. VP'63 (VintagePort'63) is the name of the project that we will develop throughout the book, with the help of the reader.

1 DISTRIBUTED SYSTEMS FOUNDATIONS

This chapter discusses the foundations of distributed systems. It begins with defining distributed systems, and performing a review of the basic subjects regarding distribution, such as computer networks, distributed operating systems and services. It introduces some generic formal notation to be used throughout the book in more elaborate treatments of some subjects. Distributed system architectures are discussed, namely: remote access; file and memory distribution; client-server; thin clients and network computers; portable and mobile code; message-based architectures; mobile computing.

1.1 A DEFINITION OF DISTRIBUTED SYSTEMS

Distributed systems have many different facets which are very hard to capture by a single definition. It is much easier to talk about distributed systems by referring to specific characteristics, or symptoms, of distribution. One such characteristic is the presence of a computer network. However, as we will see, this characteristic is not *per se* enough to define a distributed system. Distribution also comes hand in hand with disciplines such as fault tolerance, real-time, security, and systems management. Because of the shortcomings of technology, all these issues must be taken into account to build distributed systems that are efficient, reliable, timely, secure, predictable, stable, and able to adapt to changes in the environment and in the organization.

P. Verissimo et al. *Distributed Systems for System Architects*

1.1.1 What is a Distributed System?

A computer network is not a distributed system.

This is our first attempt at defining a distributed system, by clarifying the most common misunderstanding with this respect. A **computer network** is an infrastructure serving a set of computers interconnected through communication links of possibly diverse media and topology, and using a common set of communication protocols. A **distributed system** is a system composed of several computers which communicate through a computer network, hosting processes that use a common set of distributed protocols to assist the coherent execution of distributed activities. The Internet for example, is a huge computer network, as a matter of fact the most important network today. It uses TCP/IP as the common protocol suite. However, despite offering a few applicational services as tradition, such as e-mail and telnet, it is not a distributed system. A lot of distributed systems are however built on top of the Internet or using Internet technologies, such as enterprise intranets and extranets, large-scale distributed file systems and databases, virtual enterprise systems, home banking and electronic commerce systems, groupware systems, etc. One important difference is that computer processes in a distributed system *share* some common state and cooperate to achieve some common goal (e.g., they cooperatively run a distributed application). In contrast, computers in a computer network may never interact at all, or simply receive or send occasional messages (e.g., e-mail).

A distributed system is the one that prevents you from working because of the failure of a machine that you had never heard of.

The famous quote from Leslie Lamport which opened this part illustrates an important facet of distributed systems that most of us have faced in our own hard way. Although a lot of what is written on distributed systems starts with emphasizing the benefits from distribution, it is very important to realize that depending on a collection of machines connected by a network has its pitfalls. In fact each individual machine has a finite reliability, that is, the probability of not failing. Even assuming that machines fail independently, they communicate through networks that are often unreliable and exhibit unpredictable delays, and as such they influence each other, both when they work and, most importantly, when they fail. The simple fact of assembling a system with a collection of these components only makes the situation worse. This is explained very easily: the **reliability** of a system, R_s, composed of n components of reliability R, is $R_s = R^n$. Now, suppose a "distributed system" with 10 computers with individual reliability $R = 0.9$, which is quite good. If the failure of a single computer can disturb the operation of the complete system, Lamport's irony is fulfilled: the resulting system will have a reliability of $R_s = (0.9)^{10} = 0.35$, which is quite bad. Of course, such a system exhibits an inadequate architecture. In order to have a useful distributed system, we should overcome the problems of dependability caused by distribution, with the adequate architecture and protocols.

Is my multicomputer a distributed system?

There is a thin border between multicomputers and distributed systems. A **multiprocessor** is generically considered to be a machine composed of several *tightly-coupled* processors, sharing common resources, such as central memory, secondary storage (disk), and input/output through a common backplane bus. A **multicomputer**, on the other hand, is generically defined as a set of *closely-coupled* fully-fledged computers with their own basic resources such as central memory and basic input/output. Closely-coupled (Kronenberg et al., 1987) is understood as a weaker definition of tightly-coupled that allows for either backplane or short-range, fast-network interconnection media. Computers may or not share other resources such as disk, but do so on a computer-to-computer basis. Here lies the main difference to distributed systems, characterized with this regard by the *loosely-coupled* aspect of a network, with significant and uncertain transmission delays (vis a vis execution delays). Current Internet and Web-based distributed architectures lie on this side of the spectrum, whereas the trendy LAN-based **cluster** architectures (Pfister, 1998) lie towards the multicomputer side of the spectrum.

Generally speaking, and paraphrasing Schroeder in (Mullender, 1993), we should say that we are in the presence of a distributed system if the following symptoms are present:

- multiple computers
- interconnected by a network
- sharing state

These symptoms imply a few relevant characteristics. Machines are decoupled and separated enough that they have *independent failure* probabilities. Potentially, *communication* is *unreliable*, has *variable delays*, and *speed* and *bandwidth* are moderate, when compared with intra-computer communication. Investment costs are often lower than for centralized mainframe-based systems, for the same computing power. Of course, total cost of ownership may include costlier terms for distributed systems, such as management costs. Finally, in a distributed system there is an intrinsic difficulty in determining the order of events and in assessing the global state of the system from inside of it. This is due to the fact that sites can only know about each other through the exchange of messages. Note that communication has variable delays and is significantly slower than the pace at which events take place inside each site. In consequence, two participants at different sites may have a different perception of the evolution of the system: we say there is a partial order of events. Since the state of the system is split among the several sites and also influenced by messages in transit, it is not guaranteed that we can accurately determine the *global state* of the system in all situations.

Table 1.1 enumerates a few of the differences between centralized and distributed systems. Centralized systems have a natural accessibility to resources and information because they are local. Distributed systems (DSs), on the other hand, have a potentially very wide geographical scope, given the possibil-

Table 1.1. Centralized versus Distributed Systems

Centralized Systems	Distributed Systems
accessibility	geographical scope
homogeneity	heterogeneity
manageability	modularity
	scalability
consistency	sharing
	graceful degradation
security	security
	low cost factor

ity of remote operation and access. Homogeneity of technologies and procedures is a characteristic of centralized systems, whereas DSs should (and normally do) support heterogeneity, that is, sites of different makes and operating systems. This difference simplifies the management of centralized systems, but in turn, allied to modularity, renders DSs incrementally expansible. Note however that recent mainframe architectures are also extremely modular despite centralized, and can almost achieve the incremental expandability of DSs. However, scalability is achieved like in no other system with distributed system architectures, which can reach extremely high scales, both in number of sites and in geographical span. Consistency is easier to maintain in single-site centralized applications than in distributed ones, where a snapshot of the global state of the system is more difficult to capture. On the other hand, distributed state has its positive side, because it means information sharing among several sites, remote execution, distributed parallel processing, and so forth. Graceful degradation is the property of a system that continues to operate, possibly in a progressively degraded manner, in the measure that its components fail, but does not fail abruptly because of one such failure. This characteristic underpins the great potential of DSs for achieving reliability and availability (availability is the measure in which a system is up and working, during its useful life). It is yielded by modularity and geographical separation, through redundancy and reconfiguration techniques aiming at achieving fault tolerance. Security is easily achieved in centralized systems by physical access control and isolation, leading to a reduction of the level of threat, that is, of the probability of the system falling within the reach of an intruder. This is not feasible in DSs, since the potential for reducing threats in open and public networks and systems with anonymous users is more limited. However, it has lately been shown that distributed systems can attain a high level of security, provided that it is achieved more at the cost of reducing the effect of intrusions, than of reducing threats. In conclusion, we see that there are pros and cons, but distributed sys-

tems have significant advantages over centralized ones, if one makes the right decision about when to distribute.

1.1.2 When to Distribute

Why do we need a distributed system to solve this problem? If you do not need a distributed system, do not distribute. An architect should always be able to answer the question above. In response, the architect must consider that informatics[1] systems are distributed essentially for three reasons: when the problem has a decentralized nature; when distribution techniques are useful artifacts of the solution; when the problem consists in adapting to changes and evolution in the activity and location of organizations.

When *the problem has a decentralized nature*, it is not natural for the locus of control or the state repository to be centralized. For example: a manufacturing enterprise network performing concurrent engineering activities from remote locations; teleconference or other computer supported cooperative work (CSCW) activities; an industrial automation shop floor with multiple manufacturing cells, and so forth. Distribution, when used for these applications, allows the best possible fit between the problem and the computational models.

Other problems exist where there is no obvious decentralization. At the most, remote access is desired to a central facility where control and state are centralized, such as a transactional bank database. In other cases, although the business model is centralized the company has distributed facilities, such as a commercial company running several shopping malls across the country. In these cases, *distributed techniques come in help of more efficient, efficient or robust solutions.* For example, a central bank database where all account records are consistent at all times with the real client accounts, is a desirable model of a banking process. Whilst we wish to retain this view, why not split the database in several fragments, by geographical regions, located at the main agencies in each region, so that accesses are faster and not so dependent on network availability? Going a bit further, why not replicate these fragments in two or more regions, so that in case of failure, the database service is always accessible? Distributed techniques would be helpful here, for several purposes: to direct requests to the adequate fragment; to maintain consistency of the fragments as a whole database; to maintain mutual consistency of each set of replicas. However, in this case, good techniques would be those that hide the fact that the system is distributed. This property is well-known in distributed systems, and is called *distribution transparency.*

Finally, for organizational reasons, an enterprise might opt to base its infrastructure on distributed system technologies. Here, regardless of the nature of current applications, the driving force are the applications-to-be, that is, the *adaptability to a quickly changing business scenario.* The advantages brought by

[1] "Informatics" is a word of european origin getting increased acceptance in the community of computer users and developers. It is used to denote in general terms all that is related with use of computers and networks in information processing, access and manipulation.

distributed systems in modularity, expandability and scalability may outweigh the increased burden in management. For example, a fast growing commercial distribution company may base its central information system on a networked cluster of co-located servers, instead of a single large mainframe. This setting is bound to adapt to a number of possible management decisions in the future: creation of shop divisions in several places of the country; specialization and autonomy of several functions in larger departments; drastic increase of the Internet electronic commerce front-ends. An initially modular solution survives these growth crisis with less pain. For example, some of the servers may easily migrate to other locations and remain connected as they were initially, through *virtual private networks*, a distributed technology that extends local area networks through the Internet. Or else, the initially modest but modular commerce server can be enhanced by adding, as needed, additional web server front-ends for load sharing.

1.1.3 Downsizing, Rightsizing and Other Stories

There was a time when *downsizing* was the keyword for restructuring companies, informatics included. A somewhat unnecessarily literal interpretation of the term lead to large systems featuring one or more heavyweight mainframes suddenly being shrunk to hundreds of small PCs, or at the most to many dozens of RISC servers. This caused enormous organizational trouble. The problem with downsizing is that it took the wrong perspective. That is of course known nowadays but probably it is still not clear *why*, at least judging from the reflux caused by this disappointment, which lead to repositioning several informatic systems back around centralized and closed mainframes. A keyword appeared in the meantime, trying to stress one of the facets of the question: *rightsizing*. We might introduce yet another one: *rightplacing*. In fact, the problem has to do with how to modularize our system, i.e., how large are the chunks, and at what level of granularity the architect works. But it also has to do with where those modules are placed, i.e., how networks are laid out, where servers and clients are placed, and how software modules are distributed among the former. If this is understood, the architect will certainly be able to do an adequate job at laying out the system architecture, probably (but not necessarily) distributed, and certainly using whatever fits *right*, be it mainframes, or RISC servers, or PCs. There is nothing wrong with a mainframe, but rather with the way it is used. Old mainframes were closed proprietary systems, with little versatility. Modern mainframes are modular and expansible inside, and open to other systems and to the Internet. In a sense, they should be seen as very big servers, and treated as yet another building block in the solution. What is wrong with a distributed system of mainframe servers?

1.1.4 Evolution of Distribution

Distribution has been fast evolving and maturing since the mid seventies. It was not always clear what should and would be distributed, in terms of the

main resources of a computing system: applications, files, memory, processing. It all started with the need to share the (still scarce) resources of computing systems.

The first stage of evolution of distribution techniques was the sharing of files through *file transfer protocols* (`ftp`) and the sharing of access to other machines' applications via *remote session* or *remote login protocols* (`rlogin`). Transferring files back and forth soon became a nuisance, and file sharing through *distributed file systems* was the next and obvious step. The idea seems simple now, but it revolutionized the panorama of distributed computing: a program should be able to open a file (`fopen`) resident in a remote machine, and should do it pretty much in the same way as a local file open. Furthermore, users should see a directory tree that looked unique, regardless of where the files were resident, that is, a distributed virtual file system.

Table 1.2. Major Milestones in Distributed Computing

1972	ARPANET– genesis of the Internet architecture (Postel, 1978)
1976	Ethernet– first widespread local area network (Metcalfe and Boggs, 1976)
1978	OSI– Open Systems Interconnect Reference Model (Zimmermann, 1980)
1980	Internet– first widespread computer network (Leiner et al., 1997)
1984	RPC– remote procedure call paradigm (Birrell and Nelson, 1984)
1985	NFS– distributed file system (Sandberg, 1985)
1985	AFS– large-scale distrib. file system (Morris/Satyanarayanan et al., 1986)
1987	Apollo Domain– distributed file/memory system (Levine, 1987)
1987	ODP– Open Distributed Processing Ref. Model (ODP, 1987)
1987	Vax Cluster– networked cluster multicomputer (Kronenberg et al., 1987)
1987	Camelot/Encina– distributed transactional system (Spector, 1987)
1990	DCE– Distributed Computing Environment (Lockhart Jr., 1994)
1990	WWW– World-Wide Web (Berners-Lee and Cailliau, 1990)
1994	CORBA– object broker distributed architecture (OMG, 1997b)

Machines became more powerful, networked systems more frequent, to the point of being ripe for more sophisticated forms of distribution. *Distributed concurrent processing* consists of running simultaneously, in several machines, several concurrent processes that communicate and synchronize themselves through variables and structures resident in shared memory. The body of techniques that made this possible as a programming paradigm are collectively called *distributed shared memory (DSM)*: memory distribution by remote paging. Another distributed processing paradigm is *remote execution*, which consists in having a local process invoke the execution of a function or a procedure on a remote machine. The local and remote processes are for that reason called respectively client and server, and the programming paradigm using this principle is thus named *client-server*. The most distinct representative of this paradigm is a technique that caused a second revolution in distributed processing, the *remote procedure call* or *RPC*. Briefly, it consists of allowing a client

process to make a procedure call that looks like a normal one but is in fact executed on a remote machine.

We conclude this brief portrait of the evolution of distribution techniques by presenting a few of the major milestones in this evolution over the past few years, in Table 1.2.

Table 1.3. Generic Distributed System Services

Name Service	Based on a replicated and distributed database, supplies the global names and addresses of users, services and resources
Registration Authentication and Authorization Services	Registers users and services, performs runtime authentication of users and control of their access to services and resources
File Service	Provides the abstraction of a unique file system, globally accessible, made of distributed repositories, eventually replicated for performance or availability
Networking Service	Provides access by users and programs to the basic networking and communication facilities (e.g. sockets over TCP/IP on LAN, dial-up, Internet)
Remote Invocation Service	Provides for remote operation client-server invocation
Brokerage Service	Performs trading and binding of services and users in a heterogeneous environment (e.g., Object Request Broker)
Time Service	Supplies and keeps synchronized a global time reference, normally made of local clocks
Administration Service	Performs tactical management tasks, in order to manage users and keep the system resources and services operating correctly

1.2 SERVICES OF DISTRIBUTED SYSTEMS

Modern distributed systems feature a set of basic services, which can be complemented with specific services depending on the objectives of the system. The most relevant generic services of a distributed system are summarized in Table 1.3. In recent systems, the security aspects have been considerably reinforced, leading to the consolidation of security-related functions— obviously including any basic registration, authentication and authorization services— in a global *Security Service.* High quality-of-service (QoS) communication has

also become increasingly important due the need to achieve greater dependability, timeliness and security of communications on open systems. This is stressed everyday by the emergence of demanding applications, in the electronic business, cooperative (CSCW, teleconference), and multimedia rendering domains (games and movies). It would not be surprising to see certain *Advanced Communication Services* appear bundled in distributed system support packages, such as: reliable group communication; real-time communication; cryptographic communication. The following chapters, amongst other things, are going to show how to build the services presented in this section.

1.3 DISTRIBUTED SYSTEM ARCHITECTURES

This section gives an overview of the main distributed systems architectures. Distributed system architecture has evolved due to several factors, including the change of user requirements, infrastructure modifications, and technology advances. We describe the several architectural styles that developed accordingly to that evolution: remote access; file and memory distribution; 2- and 3-tier client-server; mobile; message-based.

1.3.1 Remote Access

Remote access is the primordial form of distribution. The purpose of such an architecture is to provide distributed access to central facilities. Its main facets are represented in Figure 1.1. Figure 1.1a shows a primitive form of remote *terminal access* through leased or switched telephone lines (analogue, plain digital, or ISDN) normally used to access central mainframes from terminals. However, the simplest genuine distributed access forms are those represented by Figure 1.1b. In the bottom, we have **remote session** through a data network to a central server, from terminal servers or PCs and workstations. Next up is **file transfer**, a form of remote access to files. In the top of the figure, we have remote access by **dial-up** through the telephone network to a **network service provider**, which then bridges the circuit through the data network, up to the server.

1.3.2 File and Memory Distribution

The advent of workstations allowed a democratization of computing power, with the appearance of facilities containing many workstations collectively owning a significant amount of resources. Sharing of the resources available in these workstations on a peer basis becomes thus desirable. File and memory distribution architectures, as represented in Figure 1.2, pursue that objective. These architectures have similarities. Figure 1.2a portrays the principle of file distribution, whereby a set of machines contribute with their local volumes to form a global file system. Individual volumes are made available (mounted) as branches of a global logical tree. Figure 1.2b suggests the principle of distributed memory, whereby virtual pages of the address space of a process can

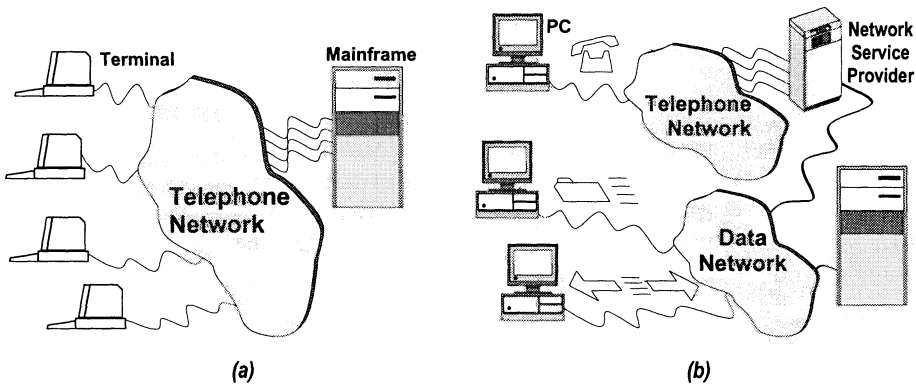

Figure 1.1. Remote Access Architectures: (a) Plain Telephone Line; (b) Data Network

be mapped in any of the system's sites and paged back to a common secondary storage. Depending on the memory distribution policy, these pages can be replicated in different sites, or migrated from site to site in order to be shared by different threads. Alternatively, a master copy may reside at one site and be cached in other sites.

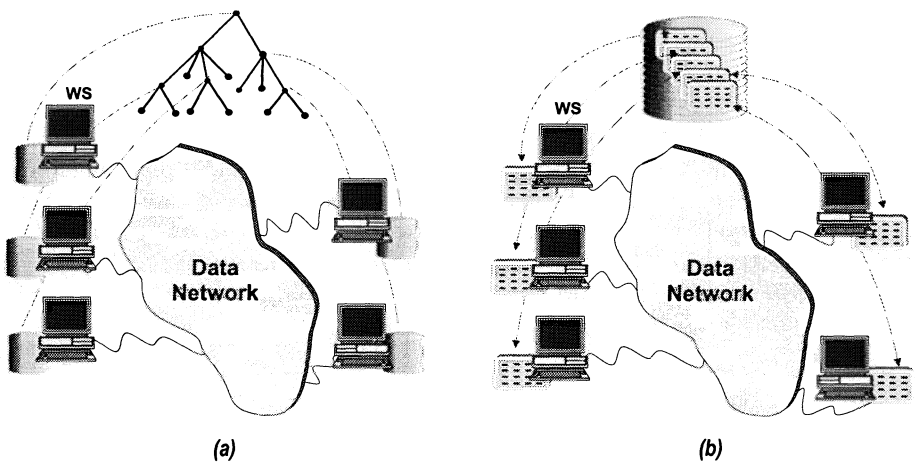

Figure 1.2. Distributed File (a) and Memory (b) Architectures

1.3.3 Remote Access II

In this phase, resources are still expensive, and sometimes not perfectly shared. An effective solution consists of reducing individual workstation's resources and basing the main services and resources on departmental servers. Figure 1.3 depicts the two main architectures that represent what can be considered the first

reflux in the evolution of distributed architectures towards decentralization. Figure 1.3a shows **diskless** workstations, whereby the file system is concentrated in one or a few central servers, and files are loaded through the network directly to the main memory of the workstation. Workstations become obviously simpler, less expensive, and easier to manage. Figure 1.3b shows yet another step back, materialized by **X-terminal** architectures. These architectures are based on machines whose only computing power is used to run the graphical PMI (person-machine interface) of the application, typically the X-Windows client-server environment, whereas the file storage and CPU power lie with departmental servers. Except for the PMI, all programs and applications execute on the remote servers. In a sense, these models bring us back to remote access of files and of computing power, though in a more sophisticated way.

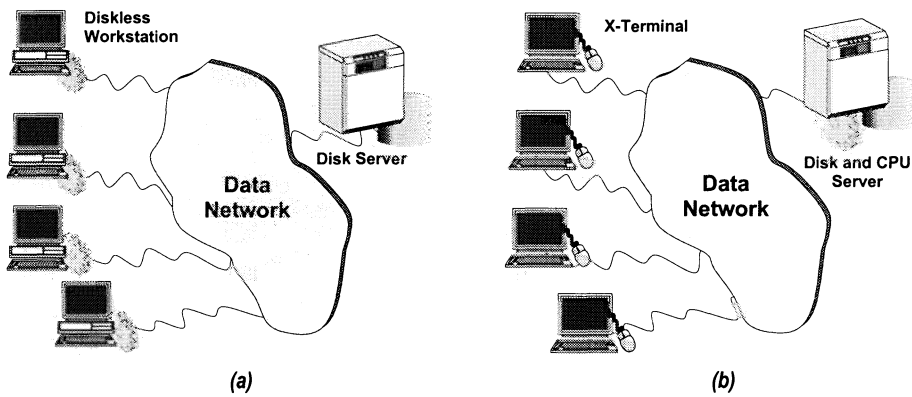

Figure 1.3. Diskless and X-Terminal Architectures

1.3.4 Client-server Architectures

The use of fully-fledged **client-server** architectures, as represented in Figure 1.4a, kept increasing at a steady pace, with the downsizing era. Client-server architectures are among the most deployed today, and are characterized by having services residing at central servers, shared by the client computers. However, clients also have local activity: they run autonomous programs— client computations— and call the remote servers whenever necessary.

One problem with this architecture is that clients end-up getting cluttered with too much code and files, which turn the overhead of garbage-collecting obsolete files and updating programs difficult to manage, requiring ever increasing performance and resources. This is called in informatics lingo the **fat client** syndrome (Figure 1.4b).

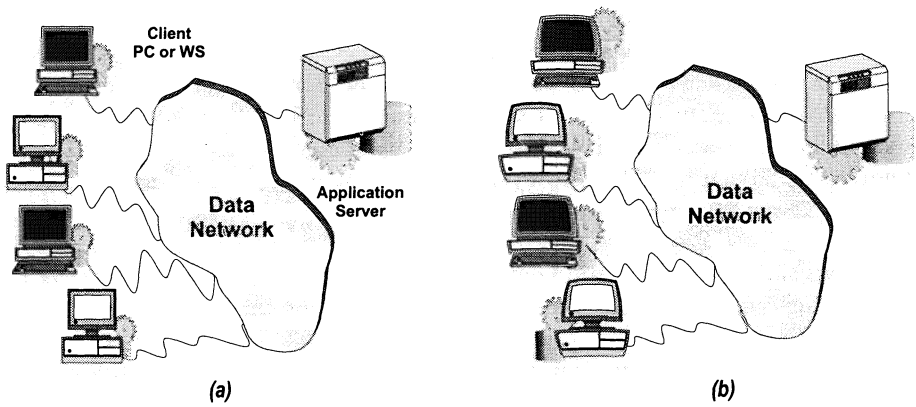

Figure 1.4. Client-Server Architectures

1.3.5 3-tier Client-server Architectures

Because of the the problems of the client-server architecture described above, a new reflux took place, and **thin-clients** made their appearance. A simple explanation of the rationale for this evolution is depicted in Figure 1.5a, where we see most of the resources, including disk and processing power, migrate to central servers. As such, they free the client machines, such that resources, files and programs become centrally managed. At a first glance, one may wonder what is the difference to the X-terminal architecture depicted in Figure 1.3b. The fact is that architecture evolved indeed, and this reflux was materialized by a new concept, the **3-tier client-server** architecture. The first tier corresponds to the person-machine interface of the application, the multimedia components that typically execute on a PMI server. The second tier corresponds to the application server, where the core of the application executes. Finally the third tier corresponds to databases and legacy systems where the data is persistently stored.

In classic client-server computing, the client shared part of the application code and ran the PMI code. The sharp separation of functions of 3-tier computing made it easy to locate all three tier services away from the client, which only retained the multimedia client functionality. Looking in retrospective, the WWW was the catalyst of this evolution, because it brought the missing links for this architecture to work. This architecture is also called **network computing**, since most computing is performed by servers located in the "network", and little code is left resident on the thin clients or *network computers*, whose hardware can in consequence be drastically simplified.

Servers can still be configured in a very modular, and perhaps tier-specific way, even if centrally located. However, since department servers become more loaded, power- and storage-hungry, a variant of this architecture consists of consolidating all services in a mainframe, instead of several servers. This is depicted in Figure 1.5b, and although it simplifies setting-up of an initial con-

figuration and its subsequent management, all benefits of distribution on the server side are lost: availability, modularity, expandability, heterogeneity.

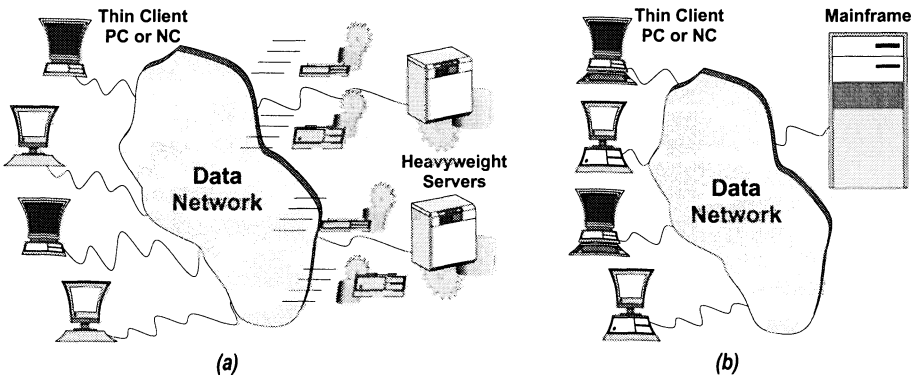

Figure 1.5. 3-tier Client-Server or Thin-Client Architectures

1.3.6 Mobile-code Architectures

It was soon realized that thin clients were too poor in functionality to perform useful work. One way to overcome the problem was with add-ons for installing code back in the client, such as *plugins* and *JavaScripts* running in the WWW environment, on which network computing is mainly based. Environments such as Java provide the basis for genuine **portable and mobile code** architectures, since they are capable of shipping code modules (applets) to heterogeneous systems, where they run in protective environments called sandboxes. Represented in Figure 1.6a is a refinement of the thin client architecture where the client, though not having resident code, can import in runtime the code it needs to perform its task. In other words, architectures based on mobile code try to bring computing power back to the client, without the penalties of classic client-server architectures that we discussed earlier. Generalizing, mobile code can migrate to machines other than NCs, such as PCs and workstations.

1.3.7 Mobile Site Architectures

We have addressed mobile code architectures, where code modules traverse the network from site to site, but sites are fixed. **Mobile site** architectures are those where sites themselves move from one place of the network to another. The generic architecture allows both for clients and servers to be mobile. It is currently deployed in very few applications, such as emergency networks and military applications (command, control and communications). Variants simplify the problem, for example by allowing only the clients to move, as deployed in mobile cellular phone technology. Another variant consists in allowing sites to move only while off-line, and re-appear at another location. This is called

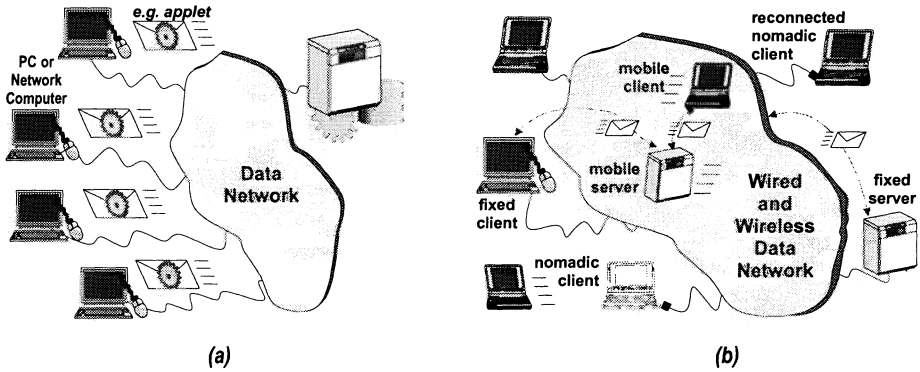

Figure 1.6. Mobile Code Architectures: (a) Portable and Mobile Code; (b) Mobile Nodes

nomadic, and drastically simplifies the problem posed by genuinely mobile sites that "travel" through the network while in operation. Nomadic sites are the most current variant in mobile computing today, materialized by notebooks and PDAs (Personal Digital Assistants) that are plugged in different networks by traveling users, but once plugged stay there for some time. These several scenarios are illustrated in Figure 1.6b.

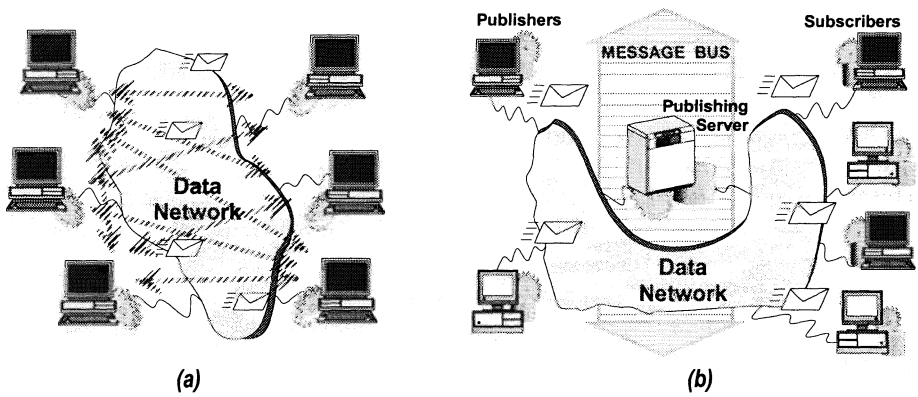

Figure 1.7. Event-based Architectures: (a) Multipeer; (b) Publisher-subscriber

1.3.8 Event-based Architectures

The client-server paradigm does not represent all forms of distributed computing. Applications such as bulletin boards, teleconferencing or cooperative work require a peer nature, where no participant is anyone's server or client. Moreover, they require spontaneity, or handling of unsolicited *events*. These needs are met by **event-based** architectures, also called *message-based* architectures.

In an advanced event-based architecture, communication normally has a multicast or group nature, and offers several embedded ordering and reliability attributes. Participants can run distributed **multipeer** interactions, also called *conversations* (Peterson et al., 1989) on top of this support, that is, directly sending messages to one another, as depicted in Figure 1.7a. An asymmetric variant of event-based architectures is *producer-consumer*. Simple examples of such applications are e-mail and news. The fashionable web-based *information-push* technologies fall into this classification. A useful enhancement is to allow the producers (or publishers) to post their messages without the consumers being on-line at the moment, providing support for the latter to *pull* the messages later, when they connect again. Furthermore, consumers should be able to specify (or subscribe) to given types of messages, and receive only messages of those types. This variant is called a **message-bus** or **publisher-subscriber** architecture, represented in Figure 1.7b. The message bus is implemented by a network publishing server, acting as a transparent persistent buffer storage for the messages posted by the publishers, and as a forwarding agent to disseminate them by the subscribers, according to their subscription specification.

1.4 FORMAL NOTIONS

This section introduces a few fundamental concepts and notation conventions that will assist us in more elaborate treatments of some subjects.

1.4.1 Modeling Distributed Systems

Distributed systems are normally modeled as a set of N processes or participants p that live on M processors or sites s. Sites are interconnected by network links or channels that may have several topologies (e.g., point-to-point, broadcast). Certain models ignore the site-participant layering and only consider processes interconnected by point-to-point links. The evolution of the system can be modeled by a succession of **events** e_p^i, for the i_{th} event in the timeline of each process p. Superscripts or subscripts may be selectively omitted when there is no risk of ambiguity. Whenever necessary, we can associate physical timestamps to events: $t(e)$ denotes the real time instant at which e took place, and is thus the timestamp of e as defined by an omniscient external observer. We can also denote references to real time instants in the timeline as $t_0, t_a, t_b, ...$ The **state** of a process, S, is modified upon the occurrence of each event. We model this evolution as a **history**, H, which is an ordered set of tuples. Each tuple is composed of an event e and relevant state information related to the occurrence of event e. A **run** is an ordered set of events in a process execution, described by a history. A **distributed run** is a partially ordered set of events in the execution of several processes.

1.4.2 Representing Distributed Computations

Events in a process computation can be: **execution** events, for executing actions internal to the process; **send** events, for sending messages to other processes; **receive** events, denoting the reception of messages from other processes.

In distributed systems algorithms and protocols, it is usual to distinguish between receive, already defined, and **deliver**, denoting the delivery of a message to the upper layer. For example, consider a protocol layer that exchanges several messages with corresponding entities in other sites, such as a reliable communication protocol. Several *receive* events take place from lower layers (e.g., the network), and at the end of a successful execution, the protocol performs a deliver action (of the message) to the end process.

Message exchanges in distributed algorithms, which occur along the timeline from and to different sites of the system, are best represented by **space-time** diagrams, such as depicted on the left of Figure 1.8. For example, execution events a and b occur in sequence in process p_1. At p_2, send event c sends message m to p_3 where its reception generates receive event d. Note that event e took place after event c in real time. However, can p_2 or p_3 know about that? In fact no, unless there is an additional artifact, a global clock that gives the same time to all processes, and events are **timestamped** as they happen.

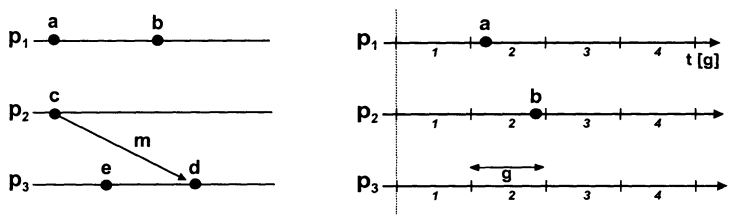

Figure 1.8. Space-Time and Lattice Diagrams

A timestamp is of the form $T(e) = c(t(e))$, meaning that the timestamp $T(e)$ takes the value of the clock c at the time when event e happened. The granularity g of a digital timestamping system (a clock) is the minimum increment of time between two consecutive timestamps, and determines how finely it measures time. Granular timelines, or **time lattices** are often used in distributed systems to account for the granularity of the timestamping systems. Events between two consecutive timestamps in the lattice are considered simultaneous. For example, as shown on the right of Figure 1.8, events a and b receive the same timestamp.

Timing variable notations are also important in distributed systems. In general terms, we use lowercase to denote instantaneous values, such as the passage of real time or the value of a clock. We use uppercase to denote time intervals, such as message delivery times, or constants and static variables, such as timeout values, timestamps, delivery delay bounds. For example, upper and lower message delivery time bounds are normally denoted by $T_{D_{max}}$, and by $T_{D_{min}}$. However, the current time of a clock would be $c(t_{now})$.

1.4.3 *Global States*

We are sometimes interested in getting the global picture of a distributed system. The **global state** of a distributed system at a given point in real time is a vector composed by the individual states of its n processes, $\mathcal{S} = [S_i...S_n]$, at that time. Under this viewpoint, the **interleaving view**, the system goes through a succession of states. Another viewpoint focuses on events, the **space-time view**, whereby the system evolves as a partially ordered set of events occurring in the several processes of the system. A **cut** in the space-time diagram is a segment intersecting the timelines of all processes. Consider a snapshot consisting of the processes' states at each intersection c_{ij} of a cut C_i with process p_j (*see* Figure 1.9): the snapshot yielded by a cut does not always provide a valid representation of the global state (GS) of the system. There are in consequence three types of cuts, illustrated in Figure 1.9: *inconsistent cut* – the snapshot gives an invalid picture of the GS of the system; *consistent cut* – the snapshot gives a correct but possibly incomplete picture of the GS of the system (for example, it ignores messages in transit); and *strongly consistent cut* – the snapshot faithfully represents an actual GS of the system.

Note that in the strongly consistent cut C_1 in Figure 1.9, there are no messages in transit. We can retrieve the states of the individual processes atomically at each c_{1j}, and get a valid global state. In the consistent cut C_3, the system has messages in transit. If we perform the same operation as above, reading state at all c_{3j}, we will get a correct though incomplete picture of the global state, where only the messages in transit (m_3 and m_4) are missing. However, note that by analyzing c_{33} and c_{34}, we are told that those messages were sent. With an adequate protocol we can wait a little longer for them to arrive, and in consequence, given this waiting time, a consistent cut is as good as a strongly consistent one. Finally, there are cuts that yield nothing valid, such as inconsistent cut C_2. Why? Observe that although traversed by messages in transit as in the consistent cut, there is something wrong with message m_1. What? It seems to be traversing the cut in the wrong direction (backwards in time). Note that this is so, because in the state information, c_{23} tells us that m_1 arrived, but there is no record of it being sent (that will be recorded later than c_{21}). This contradicts the fundamental cause-effect relation. Consistent cuts are important constructs in distributed systems, and the protocols used to obtain them are called **snapshot** protocols (Chandy and Lamport, 1985).

1.4.4 *Safety, Liveness, and Timeliness*

We can specify a system in a formal manner, in terms of high-level properties written in a formal language, including formulas containing logic (and, or), temporal (eventually, always) and time (until/from) operators. Two generic classes in which system properties can be divided are: safety and liveness. Informally, **safety** properties specify that wrong events never take place, whereas **liveness** properties specify that good events eventually take place. A liveness property specifies that a predicate $\mathcal{P}$ *will eventually be true*. A safety property

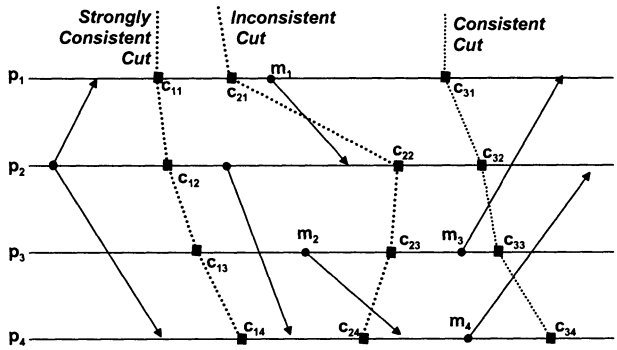

Figure 1.9. Cuts and Global States

specifies that a predicate $\mathcal{P}$ *is always true* (Alpern and Schneider, 1987; Manna and Pnueli, 1992). For example, "any delivered message is delivered to all correct participants" is a safety property. If it is not secured, the system becomes incorrect. However, it does not impose that messages are delivered at all. Property "any message sent is delivered to at least one participant" is a liveness property. If it is not secured, the system may not progress (messages are not delivered). Liveness and safety properties complement themselves. A particular class of property is **timeliness**, which specifies that a predicate $\mathcal{P}$ *will be true at a given instant of real time*. Observe property "any transaction completes until T_t from the start": it is a timeliness property. For it to be secured, all transactions must execute within T_t time units. We can specify the properties of any program or protocol in terms of safety, liveness and/or timeliness.

1.5 SUMMARY AND FURTHER READING

This introductory chapter discussed the fundamental concepts concerning distribution, defining distributed systems and introducing issues such as the difference between centralized and distributed systems, when to distribute, and how distributed systems have evolved. The fundamental distributed system services were introduced, and the most common distributed system architectures were presented. The chapter ended by presenting a few formal notions and notation for more elaborate treatments, namely on: modeling distributed systems, representing distributed computations and global states, and specifying properties (safety, liveness, and timeliness). The following chapters will discuss these introductory concepts in greater depth. For more introductory level material, the reader may consult the books of (Tanenbaum, 1996; Tanenbaum, 1995; Silberschatz et al., 2000). An elaborate treatment of formal specification of program properties can be found in (Manna and Pnueli, 1992). A discussion of the problem of obtaining consistent global states is given in (Babaoğlu and Marzullo, 1993).

2 DISTRIBUTED SYSTEM PARADIGMS

This chapter presents the most important paradigms in distributed systems, in a problem-oriented manner, purposely addressed to to-be architects. Namely, the chapter addresses: naming and addressing; message passing; remote operations; group communication; time and clocks; synchrony; ordering; coordination; consistency; concurrency; and atomicity. Paradigms are motivated by showing their problem-solving potential and also their limitations.

2.1 NAMING AND ADDRESSING

Humans associate *names* with entities, objects and resources, in order to refer to and to communicate with them. Computers are no exception, thus names are given to computers, printers, files, mailboxes, etc. Name management is thus a fundamental component of a computing system and, in particular, of a distributed computing systems.

Names alone can be used to identify any object or entity in the system if they are *unique*, i.e., if we know that no two similar objects or entities can have the same name. For instance, the name of the father and mother, along with gender, birth data and location of birth can be used to uniquely identify most human beings with a few exceptions (for instance, when twins are born). Humans do have some other *attributes* that can be used as *unique identifiers* such as, for instance, the pattern of their eye's iris. Of course, the iris pattern is too complex to be described verbally, and cannot be used as a textual name

P. Verissimo et al. *Distributed Systems for System Architects*

in daily life (although it can be used to identify a credit card holder, if an iris reading device is available). As a result we tend to assign more practical names to things and beings.

Soon after we are born, our parents give us a name. In distributed systems, we call the act of associating a name with an object, *binding*. This name does not need to be unique, since it is often used in a certain *context* that restricts the set of object in can be associated with. Names can be *pure*, in which case they can be seen merely as a pattern that can only be used to compare with other similar patterns; no information about the object can be extracted from the name alone. Names can also be *impure*, in which case their structure and format yields additional information, such as the internet name of a mail server "`mail.di.fc.ul.pt`", that allows us to extract its logical location ("`.pt`" for Portugal, etc).

Names are useful if they can later be used to obtain attributes of the named entity. Assume that you want to contact one of the authors of this book to provide us some feedback. If you want to send us a letter, you would like to use our name to obtain our mailing address. If you want to make us a phone call, you would like to obtain our phone number. If you just want to see our picture, you may just want to obtain the address of our home page. All these attributes are called *addresses* as they can be used to interact with the entity the name refers to. Obtaining an attribute from a given name, usually an address, is called in distributed systems *resolving* the name.

2.1.1 Addressing types

Addresses are also names that have a special meaning to a given communication protocol. Some addresses can be *primitive* names, i.e., names that cannot be further translated into other names. For instance, a mail address consisting of the name of a country, city, street and the door number is a primitive name. It is used directly to route letters to a specific mailbox in that given physical location.

The more complex and powerful the protocol, the less likely the address is a primitive name. For instance, in order to send an e-mail, several protocol layers need to be traversed, implying several corresponding name-to-address *resolutions* to be performed. From an e-mail address such as `dssa@di.fc.ul.pt`, one first obtains the name of a mail server for that domain, like `mail@di.fc.ul.pt`. This name needs in turn to be resolved into an IP address used to establish a TCP connection to that server. At some point, at the lower layers, the IP address will be translated into an Ethernet address, in order to send individual packets to a specific network connection.

Addresses that allow a pair of objects to interact are often called *point-to-point* addresses. However, names can be also given to groups of objects that can be managed and accessed as a whole. When a group name is resolved we can obtain a list of point-to-point addresses that allow us to contact each group member individually or even better, a *logical group address* that allows us to interact with the group as a whole, without needing to know the individual

addresses. Group addresses are an abstraction that requires the use of *group communication protocols*, able to recognize these addresses.

More powerful protocols hide details from the application code that would make it complex and difficult to port to other environments. For instance, when using IP, an application does not need to be concerned with details such as which type of architecture or operative system is used on the remote machine, what type of network that machine is connected to, etc. Now, consider a mobile machine, moving between networks: an IP address is bound to a given network, thus basic IP cannot be used to address it. On the other hand, if mobile-IP is used, the application no longer needs to be concerned about the location of the target machine, since the system reroutes packets to the appropriate location. Moreover, if multicast-IP is used, the sender does not need to be concerned about how many recipients are active. Unfortunately, the more complex the protocols the more expensive in performance they usually are, thus it is interesting to have different alternatives available to the application designer.

Sometimes, the application code is shielded from the myriad of low level communication protocols by some middleware layer, which is responsible for selecting the most appropriate protocol to contact the desired resource. In this context, the low level names recognized by the middleware are often called *references*, to distinguish them from protocol-specific addresses.

2.1.2 Name to Address Translation

Referring to object and resources using names instead of addresses has several advantages. To start with, names are often textual and given using intuitive words. Thus, names are much easier to remember by humans than addresses. Using a single name is also much more convenient than using the set of possible addresses required by the several alternative protocols to access a given object. Additionally, most low-level communication protocols do not provide location transparency, i.e, the address of an object depends on its physical location. This happens because for practical reasons low-level addresses are not pure, they incorporate information about the "location" of the object. If we want to have the ability to re-configure the system, by re-location of the objects, then it is mandatory that clients use names instead of addresses, since addresses will have a short-term validity.

Of course, at some point in time a specific protocol must be selected to interact with the object and a concrete address needs to be obtained. A mechanism is required that dynamically obtains an address given a name, also known as name resolution. This can be done by having the client broadcast a request to find out what is the address of the desired server (Figure 2.1a), which is normally replied by the server itself. This approach is usable in LANs, but not in larger scale systems. Using a modular approach, we can encapsulate this functionality in a dedicated service, that we call a *name service*. The most important and used primitive of a name service is a *lookup* primitive: it accepts a name and returns an address, as exemplified in Figure 2.1b. Additionally,

we may want to add new associations dynamically to the name service. A *bind* primitive allows an application to register an association between a name and an address with the name service. Since it is interesting to allow dynamic reconfiguration of the system, we may also add an *unbind* primitive, that allows a previous association to be canceled. Of course, once we have a name service available we may want to extend it to store other additional attributes associated with a name, instead of just storing a single address. The number and types of attributes may vary with the type of objects the name is associated with. For instance, sensible attributes to associate with a machine name are: IP address, other addresses associated with different protocols, operating system, name and contact of the system administrator, etc. Some name services also support *reverse resolutions*, i.e. the ability to obtain a name given a set of attributes (e.g., address).

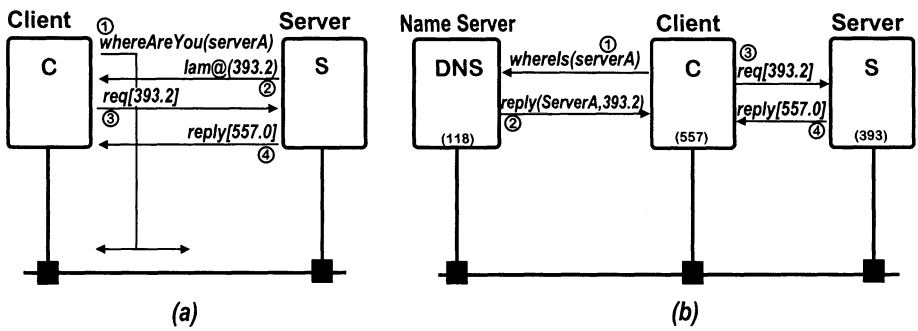

Figure 2.1. Name to Address Translation: (a) Broadcast; (b) Name Server

If the functionality of a name service is easy to understand, implementing it in a distributed system in a scalable way is a significant challenge. Since components refer to each other by names and need to resolve these names in order to interact, the availability of the name service is critical to overall system availability. Furthermore, the service must be available from every node of the system, since all nodes need to perform name resolutions. A naive approach to implement a name service is to replicate the name service state in every node, such that name resolution can be performed locally (by searching a file or querying a database). This solution is impractical. Replicating the name service data at every machine can be a significant waste of resources and makes updates to name service complex and inefficient (since all replicas would need to be updated). Additionally, in very large-scale systems, the information stored in the name service may be huge, requiring significant storage and computing resources. In many cases, like the global Internet, is not feasible to store all the name service state in a single machine.

As mentioned before, name inquiries can be performed by a broadcast communication protocol. A pure broadcast approach is not used in general, because it easily overloads all nodes with name server inquiries. However, the approach

is used for specific goals, such as the Internet Address Resolution Protocol (ARP), to obtain the Ethernet address of a node given its IP address.

Most existing implementations of a name service use several servers that cooperate among each other to preserve the global name service state and to provide a highly available and efficient resolution service to the other nodes of the distributed system. The *(distributed) name server* approach is going to be the topic of our next section.

2.1.3 Name Server Approach

A scalable approach to the implementation of a name service is to use a set of cooperating name servers. Each name server stores a portion of the name service data. The division of the name space among the servers can be made according to several criteria such as geographical locality or, more usually, administrative boundaries. A host using the name service must exchange messages with one or several servers to obtain the desired service.

At this point, the reader is probably wondering how does a host obtain the address of the name server. The answer is that the addresses of name servers are *well-known addresses*. In fact, the address of the name server is the only address that needs to be well-known, since in principle the address of every other server can be obtained through the name service. However, for both historical and practical reasons, it is common to find other popular services running on well-known addresses as well.

Let us focus on the name service. We assume that if an application wants to resolve a name, it forwards the request to a local *name service agent*. The purpose of the local agent is to hide the interaction with the name servers from the application. Two alternatives are now available for resolving the name: (i) the agent contacts a single server, which is then responsible for contacting other servers if needed; (ii) or the agent is responsible for polling all known servers until a resolution is obtained. In the latter case, the name server contacted just has to search its local database and return the requested attribute if a matching name is found, or return error otherwise. In the former approach, it is up to the name server to contact other servers. Again, two alternatives are available. The name server can interactively inquire all the remaining servers, or a recursive procedure can be executed.

A key issue for the efficiency of a name service (and of many other distributed services) is a wise use of *caching*. A cache is a copy of a frequently used piece of information that is kept in a transparent way near to its users. Caching is used extensively in the implementation of name services because some popular names are likely to be resolved several times in a short period of time by the same or related applications. Caches of recent name-to-address resolutions are kept both at the name servers and at the agents. The use of caches has minor disadvantages, the most relevant being that it delays the propagation of a new binding when the address associated with a given name changes. Because of this problem, many name services allow the clients to request an *authoritative*

name resolution, that bypasses all caches and obtains a fresh translation from the name server responsible for managing the concerned name.

2.2 MESSAGE PASSING

The most basic form of interaction in distributed systems is message exchange. In order to exchange messages, two components must select a protocol and obtain the address of each other. The components must also agree on the format of the messages exchanged. The messages are usually structured as a sequence of fields: the number, meaning, and format of each field must be understood by both participants in order for the exchange of information to be successful. We recall that the same logical values can be represented using different bit patterns in different architectures. Thus, at sending time, the representation of values is normally converted from the format internal to the machine to the format agreed for the messages (and later converted again at the recipient). These steps can be omitted if the participants know *a priori* that the same format is used at both ends. A possible message format is illustrated in Figure 2.2. The message includes the name of the source, a sequence number to identify the request, the identification of the service being requested, and the parameters required for that request.

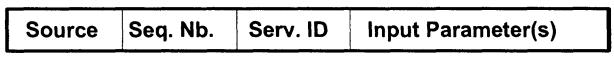

Figure 2.2. Possible Message Format

2.2.1 *Send-Receive-Acknowledge Protocol*

To support message passing two primitives are needed from the communication system. A *send* primitive, used to request the transmission of messages, which accepts a destination address and the message contents. A *receive* primitive, used to collect messages sent by others, which returns a message when available (Figure 2.3a). Typically, one cannot guarantee that every message sent is received by the intended recipient. Most communication protocols are unreliable, in the sense that they can occasionally drop messages. Additionally, nodes are also unreliable, they can crash and become unable to collect messages. Due to this reason, application designers normally make use of a slightly more sophisticated protocol, able to generate an acknowledgement back to the sender every time the recipient successfully receives a message. Thus, an *acknowledged-send* primitive can also be used to support message exchange (Figure 2.3b).

2.2.2 *Interface Styles*

The previous interface is deceptively simple, almost trivial. It should thus not be surprising that some subtle design decisions need to made when implementing the interface. Let us consider the acknowledged-send primitive again. The

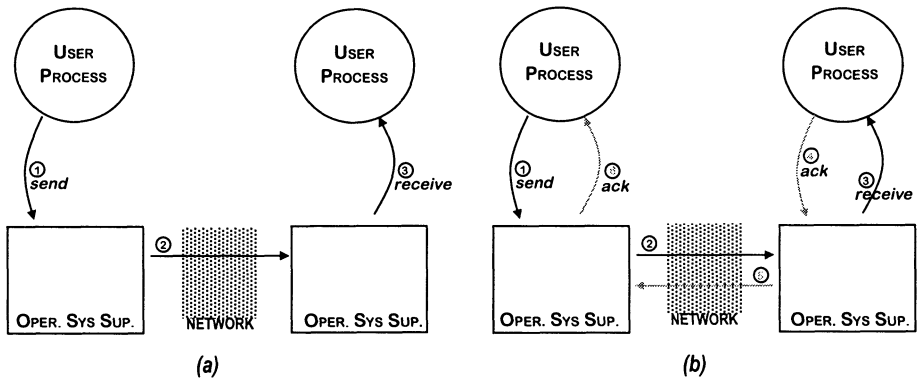

Figure 2.3. Message Passing Protocols: (a) Send-Receive;(b) Acknowledged-Send

question that the implementor needs to answer is: should the primitive block the client until an acknowledgment is returned? At first sight it may look like the answer is a definitive yes, since it seems reasonable to force the application to be sure that the message was properly received before executing the next step of the algorithm. Of course, this hides another subtle but fundamental problem: for how long should the client be blocked if an acknowledgement is not received? We will return to the issue of timing in distributed systems later in the chapter.

Even if the acknowledgements are always received, it may be useful to allow the client to proceed at least a bit further without waiting for the acknowledgment. In some applications, a client with a single thread of control may want to send several messages in order to start the parallel execution of different requests. If the network delay is significant (an recall that there are physical limits to the optimization of latency), forcing the client to wait for an acknowledgment before it can send another message incurs in a significant performance penalty. An alternative design consists in providing non-blocking primitives and supplying additional primitives to allow the application to check later if an acknowledgment was received.

On the other hand, one should be aware that the use of non-acknowledged send primitives does not necessarily imply that the *send* primitive returns immediately. In fact, the message may have to be copied to buffers of the protocol stack before the primitive is allowed to return. If no free buffer space is available, the application may be temporarily blocked until memory is freed. Also, in some high-performance interfaces, the message may be copied directly from the application address space to the network, without requiring an additional copy to intermediate buffers. In this case the application can be temporarily blocked until network access is granted to the node.

Sometimes, a remote node needs to send information to the client or user in an unsolicited, spontaneous manner. It does it in an unidirectional manner and is not interested in receiving any response. Such a message exchange is called a

notification. Notifications play an important role, since they are the adequate way of conveying information about *events*. Event-based programming is a fashionable style today in distributed systems.

2.2.3 Quality of Service of Message Passing

We have just mentioned that there is no way to guarantee that a message is received, in particular because we cannot prevent the recipient from failing. We will discuss high reliability protocols in the Fault Tolerance part of the book. However, as long as both participants remain active and the network remains connected, plain connection-oriented transport protocols do a good job at ensuring that all messages sent are actually received (by using low-level acknowledgments and managing retransmission of lost packets). By adding sequence numbers to the headers of each message, one can also ensure without much difficulty that the messages are received in the order they were sent. The resulting semantics of information flow is also known as a First-In-First-Out (FIFO) channel. This semantics is very convenient and most protocol families offer it at the transport level, such that it is frequently taken as a given in distributed systems research.

2.3 REMOTE OPERATIONS

A simple send primitive, even if acknowledged, is insufficient when the objective is to invoke some remote operation and get the result or confirmation back. The acknowledgment only confirms that the request was received, not that the associated action was completed with success (nor does it send back the results of that operation). The client-server model mentioned in the previous chapter is probably the most popular model to structure interactions between components in distributed systems. In this model, the client invokes a remote operation by making a *request* to the remote server and expects to receive a *reply* carrying the results of the requested operation.

2.3.1 Request-Reply Protocol

The request-reply protocol can be constructed using the send-receive protocol described previously (Figure 2.4a). The client sends a request to the server that, in turn, receives and processes the request. When the request has been completed, the server builds a reply that is sent back to the client.

Acknowledged send-receive has a limited interest for the implementation of receive-reply protocols, because it is useful for the acknowledgement to be produced only in the end, to achieve a closed-loop semantics: the reply confirms that the associated request was not only received but also processed. However, when unreliable channels are combined with requests that take a long time to process, it may be difficult for the client to distinguish the case where a request was dropped from the case where a request is taking a very long time to process. In these extreme cases, immediate acknowledgments become useful, to inform

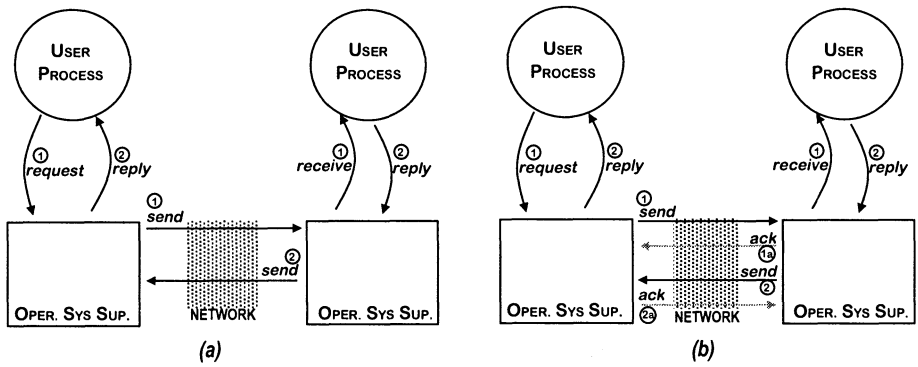

Figure 2.4. Remote Operation Protocols: (a) Plain Request-Reply; (b) Acknowledged

the client that the server has received the relevant message (Figure 2.4b). The reply itself can be acknowledged to tell the server the result was received. Intermediate acknowledgements ("I am alive") may also be used by the server, to reassure the client that the former is still active processing the request.

2.3.2 *Interface Styles*

As with send-receive-acknowledgement, request-reply interactions can also be constructed using blocking or non-blocking interfaces. Blocking interfaces are consistent with the remote operations model (Figure 2.5a). However, the arguments in favor of non-blocking interfaces assume some significance, since a request may take a long time to complete. If the client is allowed to execute while waiting for the reply, there is an associated increase in performance. Non-blocking interfaces are also called 'asynchronous' in some operating systems work, although this designation should be reserved to specify the lack of a notion of time bounds, or of synchronization in terms of time (*see Asynchronous Models* in Chapter 3).

Unfortunately, non-blocking interfaces involve some degree of complexity. New primitives that allow the client to collect the replies must be available. Since several replies can be pending, there must be a way to match the replies with the associated requests. At some point, the client may want to block until a specific reply arrives, or just until one reply from a set of useful replies is available. The client may also not want to be blocked waiting for replies in any case; or it may want to be notified of the reply as it arrives. Finally, there is a risk to program correctness, associated with executing some actions further down the code, which might implicitly depend on the result of the pending request.

An alternative allowing parallelism to be exploited is to rely on blocking interfaces and implement multi-threaded clients. When the client wants to make several requests in parallel it simply forks as many threads as needed. Each individual thread performs a single blocking request. This substantially

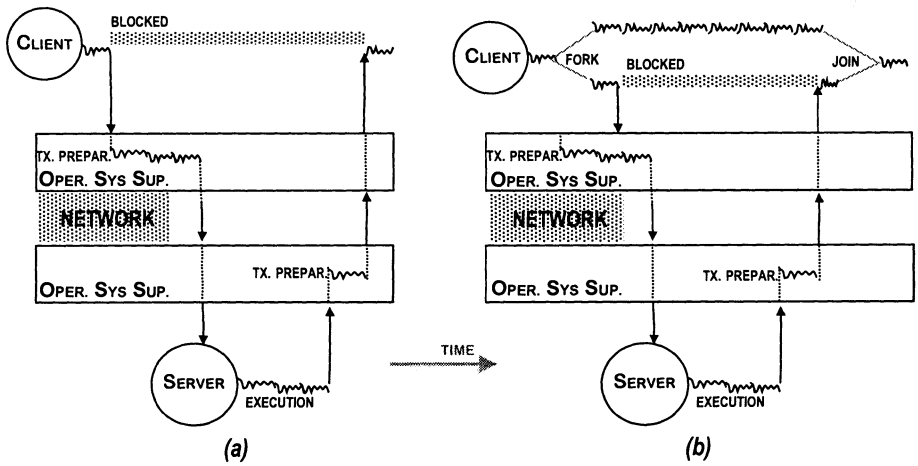

Figure 2.5. Remote Operation Interfaces: (a) Blocking; (b) Non-Blocking

reduces the complexity of the interface and of the client code. Figure 2.5b exemplifies this mechanism. Correctness problems deriving from order inversion may still occur, the responsibility of avoiding them lying with the programmer.

2.3.3 Quality of Service of Remote Operations

As we have noted above, the reception of the reply reassures the sender that the request was processed. The information contained in the reply usually includes the results of the service (if any) or the cause of error if the service could not be provided.

The absence of a reply may also indicate a fault: it may be due to the failure of the server or to losses in the communication link. In both cases the failure can occur before or after the request was serviced, so it may happen that the request was executed but the reply lost. In the part of the book dedicated to Fault Tolerance we discuss the effect of failures in request-reply systems (*see Fault-Tolerant Remote Operations* in Chapter 8).

When request and reply messages are short and can be sent in a single datagram, a connectionless datagram service can be used to support request-reply communication. If the sender waits for the reply to the previous request before doing another request, FIFO order is straightforwardly implemented. However, when request or reply messages are very long, a connection-oriented transport layer simplifies the implementation of the request-reply, taking care of fragmentation and reassembly of the messages according to the characteristics of network.

2.4 GROUP COMMUNICATION

Point-to-point communication is just a particular case of a more general pattern of *multipoint* communication. There are many examples of distributed constructs based on the notion of a group of participants. These constructs can obviously benefit from a support to multipoint communication, also called *multicast*. Take for instance the implementation of the name service discussed in Section 2.2: multicasting the name look-up request can speedup name resolution, since several servers will look the name up in parallel, and the first hit will be used. Another striking example of an application where multicast is extremely relevant is video and audio diffusion, where a stream of data is sent in parallel to a group of registered recipients.

Note that a multicast is different from a *broadcast*, where all participants in the system are addressed. Multicast is selective in the sense that only a selected *group* of participants are addressed. The efficiency of this addressing method depends on the implementation: it should be noted that sending one multicast message to n participants is quite different from sending n point-to-point messages, one for each participant. If the protocol supports multicast, it can optimize the message diffusion tree by sharing resources. For instance, if hardware multicast is available (e.g., LANs), the message can be delivered to all recipients using the channel only once. If no hardware multicast is available, significant savings can still be obtained by avoiding that a message traverses the same link more than once, as illustrated in Figure 2.6: the optimization here would have consisted of sending only one message in the A-B hop, and so forth (this is implemented e.g. by multicast-IP routing).

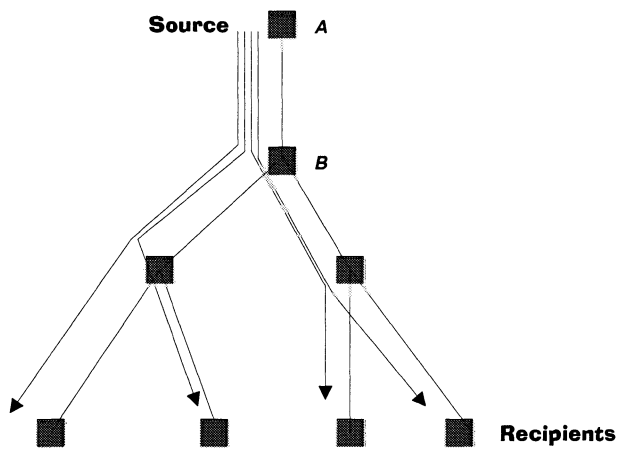

Figure 2.6. Multicast Tree

Cooperative applications, such as decision support tools, allow the interaction among several users that exchange messages, consult and update shared white-boards, etc. These applications require not only multicast support, but

also up-to-date information about which participants are active, i.e., information about the *group membership*.

In the previous examples the notion of *group of processes* is explicitly used in support of the functionality of the application. In this case, we say the groups are *visible*. Groups can also be a powerful tool to obtain non-functional requisites. For instance, a group can be used to collectively refer to a set of replicas of the same component executing the same action. (This is called *replication*, a fault tolerance technique to ensure continuity of service despite crashes of individual components, which will be deeply studied in the Fault Tolerance part of the book). The idea is for the application to address the component transparently of the existence of replicas, with the help of group communication. In this case, we say that groups are *invisible*.

2.4.1 Groups and Views

Generically, a *group membership service* provides two functions to participants: the ability to explicitly create and become member of groups, keeping that information in what is called the *group membership*; and the provision of updated information about current mutual reachability, which is called the *group view*. In a similar manner, a *group communication service* allows group members to exchange information, offering reliability and ordering properties which may vary with the quality of service selected. A protocol suite that offers both group membership and group communication services is called a *group platform*.

The main purpose of a membership service is to dynamically provide *group views*, that is, lists of unique identifiers of the processes that are mutually reachable (it is assumed that processes have a unique identifier and that there is a total order on these identifiers). Most group membership services offer primitives that allow a process to join (or leave) a group. Likewise, other ancillary services such as failure detectors provide raw information about reachability of members. Each time a change occurs, a new view is delivered to all members, so that all agree on the new state. The membership services are distinguished by the guarantees provided on the delivery order of views and on the delivery order of messages with respect to view changes. A membership service should provide two fundamental properties (Hiltunen and Schlichting, 1994), which are very hard to achieve in the presence of faults, as we will discuss later in the book:

Accuracy - the information provided reflects the physical scenario

Consistency - the information provided is consistent at all processes

Group communication services can be defined as services that allow to multicast a message to all (or to a subset) of the group members. From the point of view of the service provided, there are two main aspects:

- *Reliability aspects* – regarding the message delivery guarantees.
- *Ordering aspects* – regarding the message ordering guarantees.

Many different group membership and communication semantics can be defined, depending on the reliability and ordering properties of messages, among

each other and with regard to views. It is also useful to distinguish *closed* group models where just the members of the group are allowed to send messages to the group, from *open* group models where a participant does not need to belong to a group in order to send messages to it. In group communication services using a closed model, all members are peers, in the sense that all members can both send and receive messages to and from the group, respectively. Open models tend to distinguish different *roles*, where full *members* are entitled to receive messages and views, whereas *senders* are allowed to send messages to the group but are not necessarily aware of its membership, nor allowed to receive group messages.

2.4.2 Multicast Protocol

A multicast protocol is responsible to deliver a message to all group members. The main components of a multicast protocol are:

- *routing*, responsible for selecting the message path from its source to the addressees;
- *omission tolerance*, responsible for coping with messages that are lost or corrupted in the physical infra-structure, by redundant transmission or retransmission;
- *flow-control*, responsible for minimizing the loss of data caused by lack of buffer space at the addressees (or at intermediate routers);
- *ordering*, according to some policy;
- *failure recovery*, responsible for enforcing predefined ordering and reliability criteria in relation to view changes.

The first three aspects are addressed by a *multicast transport service*. Ordering of messages with regard to each other is offered by *ordering protocols*. Delivery guarantees in case of failure of the sender, and ordering of messages with regard to group views are usually offered by the *membership services*. The last two aspects will be dealt with in later sections.

The routing procedure consists in finding a path that minimizes both the number of messages exchanged and the multicast latency. In order to meet the first requirement, hardware multicast should be used whenever possible. To satisfy the second requirement, the path should follow a Minimal Cost Steiner Tree. Although some reliable multicast protocols address this aspect directly (Schneider et al., 1984; Garcia-Molina and Spauster, 1991), the trend is to delegate the routing procedure to standard protocols (Deering, 1989).

Network omissions are normally tolerated by using acknowledgments to detect errors, and retransmitting lost messages. These acknowledgments can be sent back whenever a message is received (*positive acknowledgment*) or only when the loss of a message is detected (*negative acknowledgment*). The former method offers a faster failure detection (even with sporadic traffic) while the later minimizes network traffic.

Finally, several techniques exist to implement flow control (Macedo et al., 1995), including usage of credits (Powell, 1991), sliding-windows (Tanenbaum, 1996), rate-based control (XTP, 1998), etc.

2.4.3 Interface Styles

Concerning multipoint interactions, it is possible to draw some parallels with the already studied send-receive-acknowledgment and request-reply interface styles. Like the point-to-point send-receive primitive, multicast-send can also be acknowledged or not-acknowledged. If it is acknowledged, the primitive only returns when the system is in the condition of ensuring that the desired quality of service can be guaranteed. Consider for instance a primitive with the following reliability definition:

> A message must be delivered to all correct group members as long as the sender remains correct during the execution of the protocol.

This is a relatively week definition of reliability, since it gives no guarantees about the outcome of the send primitive when the sender fails. In order to provide the guarantees stated above, the protocol may require each recipient to send back an acknowledgement. As soon as an acknowledgment is received from every group member the primitive may return. On the other hand, if some acknowledgements are missing, the sender must retransmit the message. To prevent retransmitting the message indefinitely in case some member crashes, such a protocol requires the assistance of a *failure detector* mechanism: an oracle that tells the protocol that it should no longer wait for replies from this recipient, since it is failed, and as such is no longer a "correct group member" as per the definition made above. Obtaining reliability when the sender fails, and building failure detectors, are rich topics that will be discussed later, in the Fault Tolerance part of this book.

When the sender is a group member, it usually receives its own messages (we say that the system offers *inclusive* multicast). Alternatively, some systems deliver the message to all members but the sender (the system is then said to offer *exclusive* multicast). The reader may wonder why some systems bother to deliver a message to its own sender, since the sender is aware of what it sent anyway! This policy is mandatory whenever messages have to be collectively ordered by some discipline. In fact the sender is only aware of the relative order of its own messages with regard to messages sent by other members if the former (or references to them) come integrated in the received flow. This unidirectionality of group information flow may strongly simplify the application design, and will be addressed again in Chapter 3.

The equivalent to request-reply semantics can also be defined for multipoint communication. This type of semantics is usually offered in architectures where group members provide service to clients that do not belong to the group (and thus, do not receive the messages sent to the group). In this case, clients send a multicast to the group and expect a reply from one or all of the group members. One possible use of this model is to support replication as mentioned

before. Another use of the model is to support *load balancing.* Load balancing is achieved by sending the request to all members of the group which, implicitly or explicitly, coordinate themselves to distribute the load (by agreeing on which member processes each request). In these two models, although the sender multicasts the request to all group members, it is interested in obtaining a single reply. Such a multipoint request-reply primitive blocks until the first reply is obtained.

Finally, groups can also be used to parallelize work. Consider for instance the problem of image processing using a technique known as ray-tracing (Watt and Watt, 1992). This technique has the property that each pixel of the final image is computed independently from the others. Thus, a ray-tracer can be easily parallelized, by making different servers be responsible for creating a portion of the picture. In such a setting, the client would send a request to the group but would block until all replies came, before proceeding.

We have illustrated the need for *at-least-n* ($n \geq 0$) and *all* semantics for collecting replies in multipoint request-reply primitives. Variants can be considered, such as: majority of members, as many as possible until a deadline, etc.

2.4.4 *Quality of Service of Group Communication*

We have briefly addressed the aspect of reliability in multicast communication. Other important quality of service criteria in group communication are the ordering policy that is enforced on the message flow, and timeliness, translated into synchronism properties. These two issues are generic bodies of research rich enough to deserve sections on their own. In consequence, we spend the next few sections dealing with them, first discussing the use of time in distributed systems.

2.5 TIME AND CLOCKS

Time is a very useful artifact to represent the ordering of events in any system. It plays a very important role in human life: try and picture one day in your life without looking at a watch or even thinking about time! However, strange as it may seem, time in the sense of a global reference, has been neglected for long in distributed systems. Several reasons explain this: systems and networks were unpredictable with regard to time, so most of the models used in distributed systems did not rely on time, they were what we call asynchronous, or time-free; decentralized and distributed algorithms requiring the synchronization of interactions with multiple sites were not in current use; interactive applications either with humans or with devices in the environment were not that common.

The situation changed: infrastructures improved their timeliness, yielding a growing acceptance of time-related models that address problems unsolved by asynchronous models; real-time architectures pervaded the arena of generic distributed systems, in areas such as telecommunication intelligent network architectures, multimedia, on-line distributed transactions, large-scale file sys-

tems, industrial information systems. All this evolution created a growing need for time-dependent protocols.

This section discusses the role of time and clocks in distributed computing systems and how a consistent view of time can be obtained in such systems.

2.5.1 The Role of Time

The role of time is intimately related to ordering, sequencing, synchronizing. The passage of time is marked by an abstract monotonically increasing continuous function, which people agreed to call **real time**[1].

By convention, this function increases at a rate equal to 9192631770 *times the period of the radiation emitted by the transition between two hyperfine levels of the ground state atomic cesium* 133, a time unit which people have agreed to call **second**. Along history, people have represented time in a number of ways and the 'second' itself has had other less precise representations, such as being a sub-multiple of the solar day. We can graphically represent these time units as a sequence of points over a straight line, called a *timeline*. We can reference what we do and what we observe (events) to points over the timeline, and extract conclusions thereof, such as cause-effect relations. That makes our life easier. The use of time in computer systems has to do with two aspects:

- recording and observing the place of events in the timeline
- enforcing the future positioning of events in the timeline

In distributed systems, the first is concerned with the distributed recording of events. The second is concerned with the synchronization of the concurrent progress of the system.

For instance, we can record the order of all the events that happened during a working day, in order to establish what are the most recent versions of our working files. However, if instead we **timestamp** each file, that is, associate it to a *point in the timeline* at the moment we close it, we simply have to compare timestamps to discover the version containing the most recent update. Imagine another problem, measuring the performance of two disk controllers (benchmarking), assessing which of them reads a large file faster. We may set up our test so that they receive the read command at exactly the same time, and observe which ends first. Alternatively, we may just run our tests separately (even in different days), and measure the **duration** of each test as an *interval in the timeline*, between the start and end points. Generalizing, a duration is a *time chain* composed of several added intervals.

We can also quote examples of the use of time to coordinate actions. Consider that we need to make a number of computers perform some actions at a pre-defined time. For instance you may want to switch on the oven half-an-hour before you get home and switch on the microwave twenty five minutes later.

[1] As opposed to "clock time", the way we mimic real time. "Real-Time" (with slash) is still another convention that refers to the research area concerned with building timely systems.

Clearly, this is easy if your system can schedule actions for future points in the timeline, or after the end of intervals in that same timeline. As a more concrete example, suppose you want to trigger both a change of points and a change of lights in a railway crossing. The specified time may be absolute, or it may be relative: the controller may be informed of the change of lights at 5:03:25 and then specify "change points at 5:03:35" or else specify a priori "change points after 10sec of change of lights".

The use of time references such as timestamps and durations (points and intervals in time) is current in computers, through *timers* and *local clocks*, devices that implement the timeline abstraction. Computers have used these devices locally since long. However, the correct use of time gets complicated in distributed systems, in particular those that interact with the environment or with humans.

To start with, if a file f_B was updated later than a file f_A, then we assume f_B's timestamp is going to be greater than f_A's timestamp, since time increases monotonically. If file f_A is updated at site A and file f_B at site B, this assumption must still be valid. In other words, we want to be able to timestamp *distributed events*, that is, related events that take place in different sites. If this property does not hold, many things can go wrong. For instance, programs that create binaries by compiling just the files that have been updated after the last compilation, such as the popular Unix `make` program, may give erroneous results if files are timestamped inconsistently.

In another example, assume we want to measure the message delivery delay associated with a given link between sites A and B. This duration corresponds to the interval between the send request and the delivery notification instants, in a conceptual timeline. In other words, we want to be able to measure *distributed durations*, that is, durations whose time chain links may develop across more than one site.

In our third example, we want to measure the round-trip delay between A and B (ever tried the Unix `ping` command?). A request-reply message exchange is performed and the time elapsed between the sending of the request and the reception of the reply is measured on the sender's site. This duration has a particular nature: it is a *closed-chain* or *round-trip* distributed duration, that is, its time chain starts and ends at the same site, after traversing other sites.

These examples require the existence of a timeline on which the relevant events and durations are mapped. *But which timeline?* A's? B's? No, it has to be a global timeline, a timeline that any event at any site can be mapped on. This notion of system-wide **global time** implements the abstraction of a universal time, the same everywhere in the system, also called *newtonian time*, and is currently implemented through a *global clock*, a clock that provides the same time to all participants in a distributed system.

This may prompt another question: *Which global time?* Lisboa's? New York's? Tokyo's? We tend to assume that the timestamp of the file update can be related to the time in our watch. But can we? The world is an intricate mesh of interdependent systems, human- and computer-based, and instead of

a single global clock, there are multiple clocks, some of them global only inside their subsystem, that is, global *internal time* references. In order to be able to coordinate these several players, not only computer systems, but also human's watches and clocks allover the world, a precious artifact was created, even before computers: *absolute time*. Absolute time references are universally agreed standards, made available as sources of *external time* to which any clock and any internal global clock can synchronize. Only if this is done can we reply to the question made in the beginning of the paragraph.

2.5.2 Local Clocks

The most common way to provide a source of time in a process is to use a *local physical clock* (*pc*). The clock at a correct process k can then be viewed as implementing, in hardware, an increasing (monotonic) discrete function pc_k that maps real time t into a clock time $pc_k(t)$. In other words, a device that materializes the timeline. Local physical clocks are typically based on oscillators such as quartz and are imperfect for two reasons that portray their main characteristics, namely their *granularity* (g) and their *rate of drift* (ρ), described with more detail in Table 2.1.

Table 2.1. Properties of a Physical Clock

• **Physical Clock Granularity** – physical clocks are granular, that is, they *tick* advancing a unit at each tick t_{tk}, which corresponds to a discrete amount of time g, the **granularity** of the clock $$pc_k^{tk+1} - pc_k^{tk} = g$$
• **Physical Clock Rate** – physical clocks drift from real time, that is, there is a positive constant ρ_p, the *rate of drift*, which depends not only on the quality of the clock but also on environmental conditions such as temperature, such that the rate of advance of the clock is not exactly real time, but rather $$0 \leq 1 - \rho_p \leq \frac{pc_k(t_{tk+1}) - pc_k(t_{tk})}{g} \leq 1 + \rho_p \quad \text{for} \quad 0 \leq t_{tk} < t_{tk+1}$$

Local clocks can be used to timestamp local events and measure local durations. The error caused by drift is normally insignificant for small durations. Computer clock drift rates are normally around several *parts per million* (ppm), that is, they can drift several microseconds per second ($\rho_p \simeq 10^{-5}$). It is also common to use a local clock as a timer, to set *timeouts*. Timeouts play an important role in send-acknowledgement protocols, since a network error is assumed if an acknowledgment is not received within a specified period of time.

Timeouts prevent the sender from waiting indefinitely and trigger a retransmission, an abort, or some other corrective measure.

Local clocks can also be used to measure round-trip distributed durations. For example, round-trip communication delays between the local site and another site and back. Generically, a locally measured duration between events a and b, $t(b) > t(a)$, given timestamps Ta and Tb from a clock of granularity g, ignoring ρ, is given by:

$$Tb - Ta = t(b) - t(a) \pm \varepsilon \quad \text{for } 0 \leq \varepsilon \leq g$$

2.5.3 Global Clocks

A global clock in a distributed system is built by *synchronizing* all local clocks to the same initial value. More appropriately, what is done is create from the physical clock at each process p, a *virtual clock* (vc_p). The initial value of the virtual clocks is set such that for all p, $vc_p(t_{init})$ are as close as possible. Since physical hardware clocks can be permanently drifting from each other, some effort must be made to *re-synchronize* the clocks periodically. Typical rates of drift of $\rho_p \simeq 10^{-5}$ seem very small, but note that they can make the accumulated error of a clock after 60 minutes exceed 30 milliseconds. In essence, what is done is to bring them back again as close as possible. Both the initial synchronization and the periodical re-synchronization of the local virtual clocks are made by a clock synchronization algorithm (*see* Clock Synchronization in Section 12.8). The set of virtual clocks under the control of the algorithm forms a global clock, whose properties, given in Table 2.2, are maintained over time.

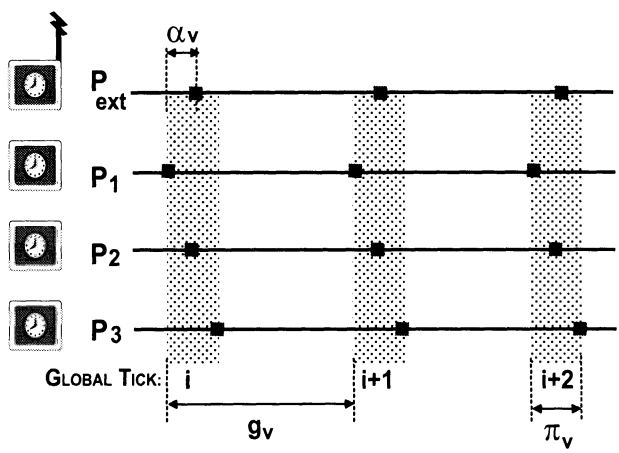

Figure 2.7. Properties of a Global Clock

Figure 2.7 depicts the main operational parameters of a global clock system: *granularity* (g_v), *precision* (π_v), and *accuracy* (α_v). Note that precision can be seen as the maximum deviation among equivalent clock ticks at each clock (*see* for example tick $i + 2$). An interesting consequence of the definitions of

Table 2.2. Properties of a Global Clock

• **Convergence** (δ_v) – characterizes how close virtual clocks are to each other immediately after the synchronization algorithm terminates. It determines how good the algorithm is:

$$|t(vc_k^0) - t(vc_l^0)| \leq \delta_v$$

• **Precision** (π_v) – characterizes how closely virtual clocks remain synchronized to each other at any time. Within limits, this is user defined, but: it depends on how fast the clocks drift; it cannot be better than convergence; it must not imply too many re-synchronizations. It is defined by:

$$|t(vc_k^{tk}) - t(vc_l^{tk})| \leq \pi_v, \text{ for all} tk \geq 0$$

• **Rate** (ρ_v) – is the instantaneous rate of drift of virtual clocks. Defined by:

$$1 - \rho_v \leq \frac{vc_k(t_{tk+1}) - vc_k(t_{tk})}{g} \leq 1 + \rho_v \quad \text{for} \quad 0 \leq t_{tk} < t_{tk+1}$$

• **Envelope Rate** (ρ_α) – the long-term, or average rate of drift, defined by:

$$1 - \rho_\alpha \leq \frac{vc_k(t) - vc_k(0)}{t} \leq 1 + \rho_\alpha, \text{ for } 0 \leq t$$

• **Accuracy** (α_v) – characterizes how closely virtual clocks are synchronized to an absolute real time reference, provided externally. It is defined by:

$$|t(vc^{tk}) - t_{tk}| \leq \alpha_v, \text{ for all} tk \geq 0$$

---

precision and accuracy is that in a set of clocks with accuracy α_v, precision is at least as good as $\delta_v = 2\alpha_v$.

Global clocks are required to solve distributed event timestamping, and distributed duration measurement. Recall that these are the remaining of the measurement problems we enumerated. Generically, a distributed duration between events a and b, $t(b) > t(a)$, given timestamps Ta and Tb measured by a global clock of granularity g and precision π, ignoring ρ, is given by:

$$Tb - Ta = t(b) - t(a) \pm \varepsilon \quad for \quad 0 \leq \varepsilon \leq \pi + g$$

Accuracy makes sense only when there is synchronization to an external source of absolute time that represents real time, called *external synchronization* as opposed to *internal synchronization*, where clocks only achieve precision

in terms of internal time. In Figure 2.7 the external source is represented by P_{ext} with a receiver, for example of GPS (*see* below). Obviously, the external time reference must be taken into account at each re-synchronization as the time to synchronize from.

The main international time standards are the *Universal Time Coordinated, UTC*, a political time reference carrying all the properties of date and time as we use them currently, such as leap second insertion, leap days, etc., and the *Temps Atomic International, TAI*, a chronoscopic reference, that is, a monotonically increasing function at a constant rate, without any discontinuity. TAI is generated from atomic cesium clocks, the devices that currently provide the most accurate and stable abstraction of the 'second'. Several institutions have these clocks, and there are several time source methods. We will just address what we consider to be the simplest and most effective way to get TAI or UTC (it can be derived from TAI): a GPS satellite signal.

The NavStar Global Positioning System, GPS (Parkinson and Gilbert, 1983), is a network of 21 satellites covering the earth surface in a very complete way, so that normally at least 4 of them are above the horizon. Although used mainly for positioning and navigation, the feature of interest here is that they provide an extremely good source of absolute time from their cesium atomic clocks, with a stability in the order of $\rho_g \simeq 10^{-14}$, that is, 1 s in 3 000 000 years. Satellite clocks are monitored and corrected periodically in conditions which ensure an accuracy on ground of $\alpha_g \leq 100ns$ for the GPS-receiver clocks, which may be installed in computers. GPS receivers are currently cheap, and the availability of signal reception is very high. They offer several interfaces, and most commercial devices provide UTC. The only caveat is that the GPS receiver antenna must be under the light cone of the satellites it is receiving from, that is, the antenna must be placed externally, and reasonably clear of building walls.

2.5.4 *Round-trip duration measurement*

With additional algorithmic support, certain distributed durations can be measured without the explicit existence of global clocks, but just assuming that local clocks have the bounded rate of drift property. Although the notion of using time chains to prove or extract time-domain properties has been around in several works, this paradigm was first formalized and later refined by Flaviu Cristian and his team (Cristian, 1989; Fetzer and Cristian, 1996) in the context of time and clocks. The basic principle is illustrated in Figure 2.8a. When a message m_1 is sent from p to q, its delivery delay can be measured with a bounded and known error, provided that there is a sufficiently fresh previous message (m_0) from q to p (shown dashed in the figure) that closes the time chain between p and q. *Why fresh?* Because the error caused by the drift of each of the clocks at p and q is proportional to the magnitude of the rate of drift itself but most importantly, to the separation between both messages. *How does it work?* Observe Figure 2.8a (recall that we want to measure $t_D(m_1)$): q knows $T_{q1} - T_{q0}$ by its clock. If it is told about $T_{p1} - T_{p0}$ (sent in m_1), then all it

needs to know is $t_D(m_0)$. This is impossible, but a sure lower bound is T_{Dmin}. So one can define basic algorithmic guidelines to be followed in any protocol or application using this method:

- ensure that there are regular messages exchanged between the relevant sites;
- ensure that timestamps of message transmissions and deliveries are are also exchanged between the relevant sites.

The delay of m_1, $t_D(m_1)$ in Figure 2.8a, is measured by the following expression, which ignores rate of drift for the sake of simplicity. Since the dominant term here is the message delivery variance, we also ignore g:

$$t_D(m_1) \leq (T_{q1} - T_{q0}) - (T_{p1} - T_{p0}) - T_{Dmin}$$

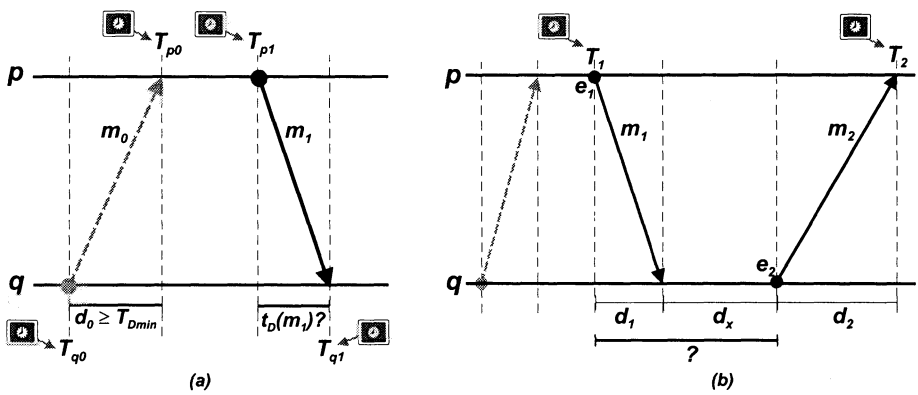

Figure 2.8. Round-trip Duration Measurement: (a) Message delay; (b) Distributed Duration

The method we describe next consists of a refinement of the round-trip duration measurement paradigm, whereby sites send or use extra messages to create round-trip loops, closing otherwise open-loop distributed time chains, so that the latter can be measured. Suppose we want to measure the duration between distributed events e_1 and e_2 shown in Figure 2.8b. The specific guideline we need to follow is:

- send a message immediately after the start and end events, e_1 and e_2 (for simplicity we assume $t(e_1) = t(m_1)$ and the same for (e_2, m_2))

Participant p logs the timestamp of e_1, $T_1 = c(t(e_1))$. Observe that q, after receiving m_2, can compute its delivery delay d_1. When event e_2 takes place, timestamped T_2, after local duration d_x measured since the timestamp of delivery of m_1, $(T(m_1))$, a message is immediately sent back to p. The latter, after receiving m_2, timestamps that moment as $T(m_2) = c(t(m_2))$. Participant p also computes the delay of m_2, as d_2. The duration between events e_1 and e_2 can be computed by both p and q as follows:

- at p: $d_{12} = T(m_2) - T_1 - d_2$

- at q: $d_{12} = d_1 + T_2 - T(m_1) = -(T(m_1) - T_2 - d_1)$

Round-trip measurement with local clocks can measure distributed durations. Generically, a distributed duration between events e_1 and e_2, $t(e_2) > t(e_1)$, measured by round-trip at the site where the interval starts, ignoring ρ and g, and considering a message delivery delay variance of $\Gamma = T_{Dmax} - T_{Dmin}$, is given by the following expression (m_1 and m_2 are respectively the start-of-interval and the end-of-interval messages):

$$T(m_2) - d_2 - T_1 = t(e_2) - t(e_1) + \varepsilon \quad for \;\; 0 \leq \varepsilon \leq \Gamma$$

The error of this method is not negligible compared to using a high quality global clock. Besides the drift factor which we ignored, note that in the basic message delay measurement mechanism (Figure 2.8a), short of knowing the delay of m_0, we stipulated its lowest bound, T_{Dmin}. The difference between T_{Dmin} and the current delay of m_0 accounts for an additional error. The method has the additional disadvantage of not being transparent to user algorithms or applications. On the other hand, is does without having to explicitly establish a global clock in the system. If the interval is long enough that the drift is no longer negligible, then even global internally synchronized clocks are not enough, they have to be externally synchronized.

2.6 SYNCHRONY

The terms "*synchronous*" and "*asynchronous*" are used in the context of distributed systems with many different meanings. In the context of send-receive and request-reply interfaces, which we have already discussed in this chapter, they are often used to denote *blocking* (synchronous) and *non-blocking* (asynchronous) primitives. In groupware systems, the terms are used to distinguish respectively between *same-time* and *different-time* interactions, that is, when all participant interact simultaneously, or better said, *synchronized*, or when the participants interact in a deferred way, or *non-synchronized.* In digital systems and many systems that extend or mimic hardware implementations, synchronous usually means *clock-driven*.

In this section we focus on yet another meaning of synchrony, the one commonly used by the distributed systems community. In this context synchrony refers to the nature of executions that assume worst-case times for local and distributed actions.

2.6.1 Synchronism

We say that an algorithm or protocol is *synchronous* if it is possible to bound its action delays (processing and network). Synchronism properties are important because they allow decisions to be taken based on the passage of time. For instance, if a component is expecting a message at a given moment, and the message has not arrived past that moment, it can be immediately assumed that some fault has occurred. This cannot be done in *asynchronous* settings, where

the passage of time provides no information, since participants and networks can be arbitrarily slow.

Synchronism can assume different forms that we enumerate from weaker to stronger: bounds do not exist (asynchronous); bounds exist but are not known, or they exist and are known but only hold at times (partially synchronous); bounds exist and are known (synchronous). The latter prefigures the synchronous framework in systems design. Synchronism is expressed in terms of *timeliness* properties which, as we studied in Section 1.4, specify behavior with relation to time constraints. For the sake of example, a common definition of synchronism for message delivery is:

Time-Bounded Delivery – Any message delivered is delivered **within** a known bound T_{Dmax} **from** the time of send request

2.6.2 Steadiness and Tightness

There are grades of synchronism in distributed algorithms and protocols. Consider a distributed execution that starts in one site and ends in the same or another site. How synchronous is it? An obvious example of such an execution is message delivery. In consequence, let us define the following:

Delivery Time ($t_D^p(m)$) – interval between the $send(m)$ event of message m, and the $deliver_p(m)$ event at p, i.e. $t_D^p(m) = t(deliver_p(m)) - t(send(m))$

How synchronous a protocol is can be assessed by two metrics: how steady (constant) is the delivery delay as seen by one participant; and how tight (simultaneous) is a delivery to multiple participants.

Steadiness (σ) – is the greatest difference between the maximum (T_{Dmax}^p) and minimum (T_{Dmin}^p) delivery times observed at any participant p: $\sigma = \max_p \ (T_{Dmax}^p - T_{Dmin}^p)$

Tightness (τ) – is the greatest difference, for any messages m, between $t_D^p(m)$ and $t_D^q(m)$, for any p, q: $\tau = \max_{m,p,q} \ (t_D^p(m) - t_D^q(m))$

These definitions are exemplified in Figure 2.9. Delivery time at p is shown in Figure 2.9a. Steadiness is shown in Figure 2.9b, where message **x** yields the maximum delay and **y** the minimum delay, both at p. Tightness, in the same figure, is shown with the execution of **y**.

2.6.3 Achieving Synchrony

If synchronism is important, why not just make all systems synchronous? The problem is that synchrony is very difficult to achieve, as it is often in conflict with other important goals: resource sharing, scale, openess, and interactivity. Consider for instance the important resource that constitutes the network. Most networks have shared medium, and their operation is highly unpredictable.

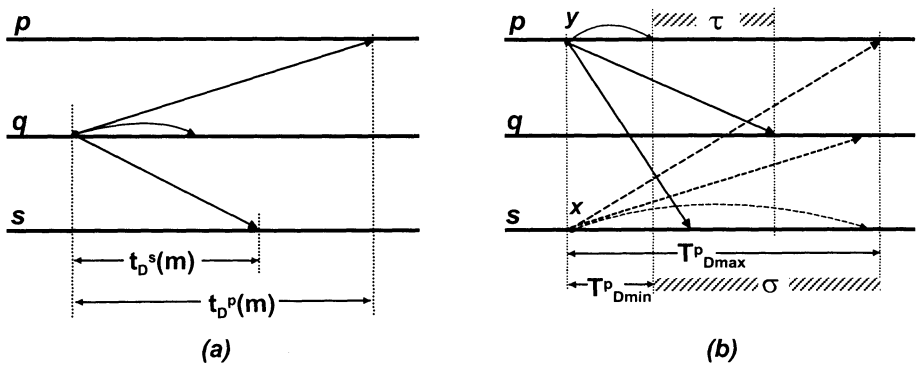

Figure 2.9. Synchronism Metrics: (a) Delivery Time (t_D); (b) Steadiness (σ), Tightness (τ)

Achieving synchrony in a system means securing timeliness properties, that is, the capacity to execute actions tied to pre-specified time instants or intervals, specified by constructs such as "at", "within", "until", "every", or "after". There are a number of informal ways of specifying such behaviors: "task T must execute with a period of Tp"; "any message is delivered within a delay Td"; "any transaction must complete within Tt from the start"; "action A must be triggered at clock time Tc"; "action B must be triggered after delay Td from now". This implies both *infrastructure* and *algorithmics*.

It is not enough to wish for a process to execute an action in "100ms from now". There has to be enough processing power, and the process has to be scheduled in time. It is pointless to demand a packet delivery delay of 100μs, if the sheer transmission delay of that packet amounts to 1ms for that network's throughput, or if packets are frequently lost in transit. Infrastructure is necessary to ensure at the lower levels that the system has some self-determinacy with regard to time. For example: having network packets reach their destinations within some delay; scheduling processes when needed; providing clocks with the necessary granularity, precision and accuracy; reading clocks in a timely manner. These issues pertain technically to real-time system operation, and will be addressed in the Real-Time Part of this book.

However, infrastructure alone is not enough to achieve synchrony. For example, a LAN does not achieve bounded delivery delay *per se*. In a Token-ring LAN, while the token rotates, it assesses which is the highest priority frame waiting, and schedules transmission of that frame for the next rotation. This seems a very elegant native mechanism to achieve bounded delay for priority frames. However, if a high priority request arrives just after a very long low priority frame starts to be transmitted, it will have to wait a long time before the low priority transmission ends, perhaps violating the desired time boundedness. Some scheduling and/or load control algorithmics must be applied to solve this problem. A real-time LAN has tight transmission, because frames

arrive almost at the same time everywhere, but it is fairly unsteady, since frame delays vary with load. And when faults occur, even tightness is lost. Clock-driven algorithms (i.e., using global clocks) and error recovery mechanisms are important contributors to achieving steady and tight transmission.

2.6.4 *Implementing Steadiness and Tightness*

Recall that steadiness, defined for message delivery, measures the variance of the relevant delay observed by each participant. Recall also that tightness, in the same context, measures the simultaneity of delivery instants at the several recipients of a multicasted message.

The simplest real-time protocols have a *timer-driven* structure, that is, without global clocks, at most using timers. Delivery is potentially unsteady and untight, but still it can be time-bounded delivery as we defined it earlier. Figure 2.10 shows a systematic method for achieving synchronous message delivery with timer-driven protocols. The basic assumptions are: maximum and minimum frame delivery latencies of δ_{mx} and δ_{mn}; a maximum number of consecutive transmission errors (omissions) k; a retransmission timeout of T_{tout}; and n participants. The approach uses closed time chains to enforce timeliness and detect errors, and consists of:

- structuring the protocol in a bounded number p of clearly delimited phases, for modularity (the number of phases depends on the protocol reliability and order properties, one phase being enough for reliable delivery);
- structuring each phase as a bounded series of up to $k + 1$ round-trip (send-ack) transmission rounds, to recover from omission errors (the maximum number of rounds depends on the desired error resilience; in absence of errors, one round is enough).

Each protocol execution can be represented by a chain in time, bounded to known values T_{Dmax} and T_{Dmin}. The maximum message delivery occurs when the network is slowest and faults occur in all round-trip transmissions of all phases, yielding the following simplified expression for an upper bound: $T_{Dmax} \leq p(k+1)T_{tout}$. The minimum message delivery delivery occurs when the network is fastest and no faults occur, each phase being implemented by one round of one transmission plus $n - 1$ replies of minimum delay (the last phase delivers the message), yielding the following simplified expression for a lower bound: $T_{Dmin} \geq (p-1)n\,\delta_{mn} + \delta_{mn}$.

Steadier and tighter protocols can be built with the help of good global clocks. For that reason, these protocols are also called *clock-driven*. One such method is depicted in Figure 2.11. The basic assumptions are: maximum frame delivery latencies of δ_{mx}; reliable frame channels; global time from synchronized clocks with granularity g and precision π. The approach relies on the existence of a reliable transport of low-level frames, and of a global clock both to timestamp frames and to coordinate message delivery. It consists of the following steps:

- sender timestamps m to be sent with the value of its local clock ($c(m)$);

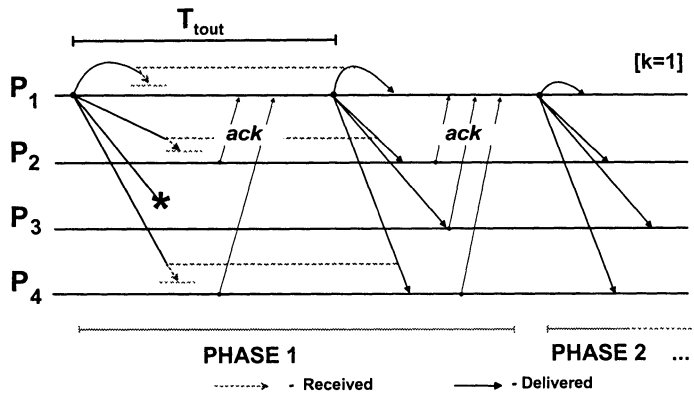

Figure 2.10. Unsteady and Untight Synchronous Protocol

- m is reliably transmitted and arrives everywhere by δ_{mx};
- recipients keep m awaiting, and all deliver m at $T_D(m) = c(m)+\Delta$, measured by their clocks.

The waiting time is a system-wide constant. For that reason, the protocols of this class are also called Δ-*protocols*. The value of Δ depends on the uses of the protocol, for example for ordering (*see* Section 2.7). Note that the actual message delivery latency is fairly constant, lying somewhere in the interval $[\Delta; \Delta - \pi - g]$. That is, steadiness is $\sigma = \pi + g$. Furthermore, tightness is also very good, since all recipients deliver each message when their clocks have the same value $T_D(m)$, which by definition of precision implies that tightness is $\tau = \pi$, as shown in the figure.

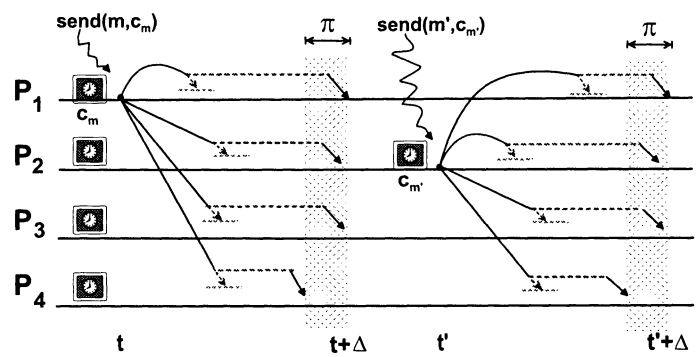

Figure 2.11. Steady and Tight Δ-protocol

With a global clock it is straightforward to build a *Time Division Multiple Access* protocol (*TDMA*) in a distributed system. Local clocks schedule transmission exactly during that site's slot, and message dissemination periods

occur in regular succession, also called in a *time-triggered* way, meaning that it is triggered at pre-defined instants from a clock, as shown in Figure 2.12. The basic assumptions are: n participants; maximum frame delivery latencies of δ_{mx}; transmission period duration of T; reliable frame channels; global time from synchronized clocks with granularity g and precision π. The approach relies on the existence of a reliable transport of low-level frames, and of a global clock to coordinate message delivery. It consists of the following steps:

- the timeline is organized as a lattice divided in slots longer than δ_{mx};
- each period over the lattice occupies n slots, as many as the participants;
- sites are ranked 1 to n and in a period, each site transmits in its slot;
- at the beginning of each slot (tick of the lattice), one site transmits its frame to all others, of maximum duration δ_{mx};
- all frames from all participants are processed at the end of each period.

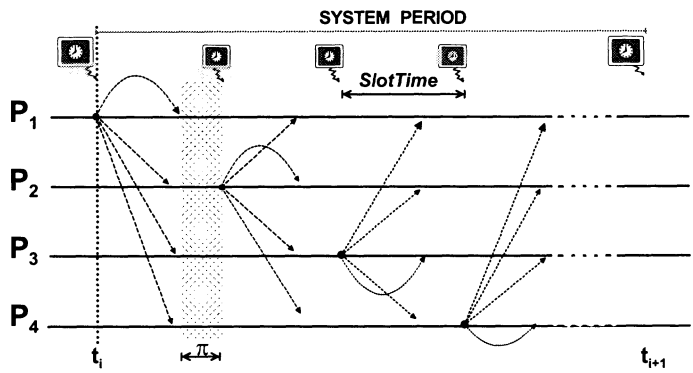

Figure 2.12. Steady and Tight TDMA Protocol

Macroscopically, since a transmission round is triggered at the beginning of a period and messages take effect at the end of that period, it is as if all sites sent their frames in the beginning of a period and delivered them at the end of the period, by their local clocks. This analysis yields a stable delivery delay, approximately of one period of the lattice, T, with an error given by steadiness, of $\sigma = \pi$. Furthermore, tightness is also very good, since all recipients deliver each message when their clocks have the same value, the end of the period, yielding a tightness of $\tau = \pi$.

Microscopically, the timing error of each site in entering the medium may be π. In order not to overrun the next slot, there must be a guard interval of at least π in each slot time, as shown in Figure 2.12 (*see* the transmission of P_1). Then, the period T can be extracted from the expression $T \geq n(\delta_{mx} + \pi)$.

2.7 ORDERING

The notion of *order* of events appears quite naturally when describing distributed computations. As a matter of fact, it is a fundamental paradigm. To understand why, recall the timelines we discussed back in Section 2.5. Now imagine you take the magnitudes of time out of the timeline. What remains is *order*, a local sequence of events, where each one happens before the other. For instance, when describing the send-receive protocol we have mentioned the use of FIFO order, which ensures that messages are received at the sending site in the order the send requests happened in the sending site. In this section we discuss the role of order and mechanisms that can be used to order events and messages in distributed systems.

2.7.1 The Role of Order

In many distributed applications there is a need to order events. Ordering assumes two facets. The first one has to do with determining *a posteriori* the order in which events happened. This allows us to understand which events occurred first and to assume or exclude *cause-effect* relations among them. Note that our understanding about the universe, and in consequence about computational systems, wanders about this fundamental relation: message A "caused" the sending of message B; command C "caused" the execution of processing step S, and so forth. A typical computational application of a posteriori ordering is the following: if we log the order by which events occurred, we can replay a non-deterministic computation. This feature is precious in distributed debugging. The second use of ordering is to ensure that events take place according to some pre-defined ordering policy, which must be enforced *a priori*. This is achieved by ordered delivery protocols. For instance, in order to ensure FIFO delivery, one needs to be able to order messages before delivering them, in the same order they were sent, despite delays or losses.

The most intuitive notion of order is *physical order*, i.e., the order by which events occur in a real time timeline as seen by an omniscient observer. There is a strong reason for that: for event a to cause event b, it must take place before b. In consequence, we say physical order is a *potential causal order*, i.e., it orders all events that *may* be causally related. This order can be captured if all events are timestamped with the value of a global clock. However, quite a few events will be ordered unnecessarily with this approach.

A tighter potential causal relation is *precedence*, or "happened before", introduced in (Lamport, 1978b). Observe Figure 2.13a. It is clear that event a precedes event b, since they occur in sequence in the same process. This is the first condition defining precedence, also denoted as ($\rightarrow$). Now note that event c took place before event e in physical order. However, does it precede e? No, because it could not 'cause' e. In contrast, if c is the sending of a message, and d its reception at another site, then $c \rightarrow d$. This is the second condition for precedence. The third is the transitive closure of $\rightarrow$. Events that do not

depend on each other are said to be *concurrent*, and they can be ordered either way.

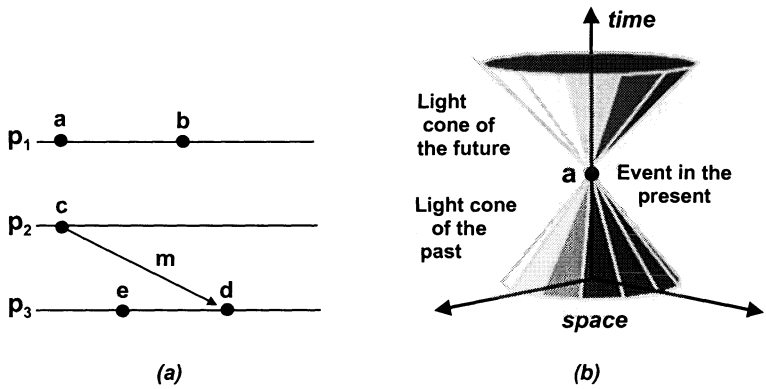

Figure 2.13. Precedence: (a) Space-time View; (b) Relativistic Light Cone

This example shows that not all messages have to be ordered by the physical order of the *send* requests. As a matter of fact, if m is the fastest possible transmission, then d is the earliest event that can be caused by c, because no other information departing from p_2 could reach p_3 earlier. That is, c only precedes d because they are *time-like* separated by more than the time it takes to overcome their *space-like* separation[2]. Out of curiosity, precedence in distributed systems is, on a smaller scale, among the phenomena explained by the Relativity Theory, because of the significant duration of message propagation vis-a-vis duration of local executions. In Figure 2.13b we see a three-dimensional space-time diagram where a occupies the vertex of an inverted light cone disposed along the time axis. The cone delimits the fastest speed of propagation. For a to be said to precede b, b must be inside the cone (Hawking, 1988).

2.7.2 *FIFO Order*

First-In-First-Out (FIFO) order reflects the potential causal order generated by a single process.

FIFO Order – Any two messages sent by the same participant, and delivered to any participant, are delivered in the order sent

Assume that channels are unordered. FIFO order can be recovered from arriving messages simply by timestamping each message sent with a local sequence number. A FIFO ordering can thus be implemented by making the recipient deliver the messages by the order of their sequence numbers. In or-

[2] "Time-like" is measured in the time coordinate, "space-like" concerns the space coordinates, in Relativity jargon.

der to do so, the recipient may have to temporarily buffer messages that are received out-of-order, and/or request the retransmission of missing messages.

Consider the following example depicted in Figure 2.14a: Paul at site r is solving the first phase of a problem by executing three modules in sequence. He disseminates the intermediate results through messages m_1, m_2, and m_3, to Mary and John, respectively at s and q, who perform the second phase operations, which depend on the respective order. John has just been delivered message m_1 at q, with sequence number 10, and the protocol has just received message m_3 with sequence number 12. The protocol simply waits for message m_2 with number 11, or requests its retransmission if lost, and only then it delivers messages m_2 and m_3 in that sequence.

When is FIFO insufficient? Observe the example in Figure 2.14b: the problem was complex, so Paul decided to distribute his part of the job, asking Mary to perform step 2 after he performed step 1, which he would obviously signal with m_1. Step 2 is executed by Mary, resident at site s, only after m_1 arrives, after which message m_2 leaves s. This looks correct, the problem is that the FIFO protocol ignores any relation between sites r and s, and since m_1 got a bit delayed, it will be delivered to John at q after m_2. Since John is waiting for the messages in the order they were issued to perform the second phase, this contradicts the application semantics. What went wrong is that FIFO cannot be used if competing senders to a site also exchange messages among themselves.

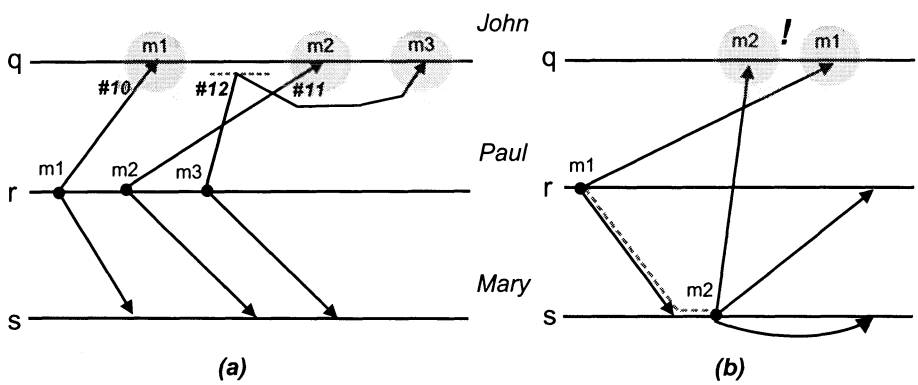

Figure 2.14. FIFO Order: (a) An Example; (b) FIFO Insufficient

2.7.3 Causal Order

Consider the scenario of Figure 2.15a: the problem with FIFO order that we have just analyzed is solved. The protocol used secures potential causality *across sites.* It ensures that messages obey *causal delivery* or *causal order.*

Causal Delivery – For any two messages m_1, m_2 sent by p, resp q, to the same destination participant r, if $send_p(m_1) \rightarrow send_q(m_2)$ then $deliver_r(m_1) \rightarrow deliver_r(m_2)$, i.e. m_1 is delivered to r before m_2

In consequence, delivery of m_2 is delayed until m_1 arrives (we can see the dashed causal chain). How is causal order implemented? When participants communicate solely by exchanging messages, there are simpler and more accurate ways of capturing potential causality and implementing causal delivery than by using physical clocks. They consist of tracing precedence in message interchanges, since they are the only way of developing causal relations among sites. This class of protocols secures what we may call *logical order*. As said before, this order is *potential* because it captures sequences that *may* be causally related.

Logical Order – A message m_1 logically precedes ($\stackrel{l}{\rightarrow}$) m_2, **iff**: m_1 is sent before m_2, by the same participant **or** m_1 is delivered to the sender of m_2 before it sends m_2 **or** there exists m_3 s.t. $m_1 \stackrel{l}{\rightarrow} m_3$ and $m_3 \stackrel{l}{\rightarrow} m_2$

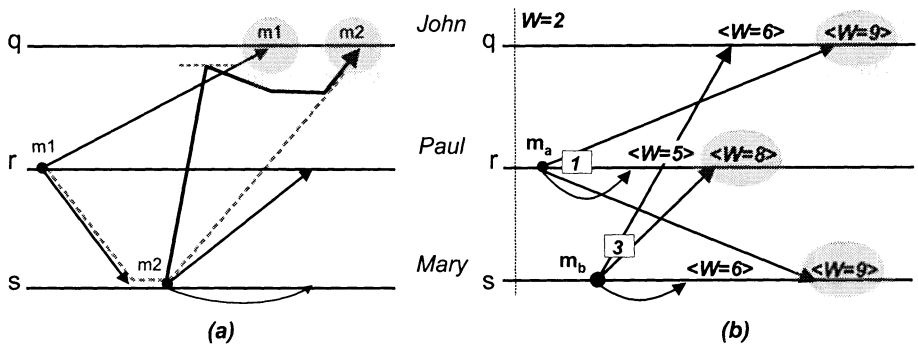

Figure 2.15. Causal Order: (a) An Example; (b) Causal Insufficient

The reader might question the utility of causal order protocols, since most applications today do not use them and still work correctly. The fact is that the semantics of most applications is client-server or producer-consumer with little or no peer interactions at all, in which case FIFO order is enough. However, to cite just a few useful cases, there are many emerging applications oriented to peer interactions, such as teleconferencing and interactive multimedia, and some system support packages for distributed debugging and distributed shared memory.

Still, causal order does not ensure correct behavior in all situations. The work lead by Paul implies some computations, whose result is accumulated in working variable W. W is updated by comparing its previous state to each new result, taking the greatest and adding 3. There have been some errors in previous work, so Paul decides that all steps will be done in parallel by him, Mary and John, and the results disseminated to all, so that they maintain a

replica of W and compare the results. Any one finishing a step simply posts a result to all including himself, in causal order. If everybody is doing the same steps, it is expected for W to be the same everywhere. Let us study the run in Figure 2.15b, considering an initial value of $W = 2$.

Paul at r and Mary at s disseminate their results. Since these two requests are concurrent, the causal order protocol makes no effort to order them. In consequence, $m_a = \langle 1 \rangle$ is received first at r, the previous value $W = 2$ is kept, and W evaluates to 5. Later, it is incremented to 8 after $m_b = \langle 3 \rangle$ is received. At q and s however, request m_b is received first: W evaluates to 6, being later incremented to 9, after reception of m_a. This violates Paul's assumption of a replicated computation at all sites. Of course, subsequent steps depending on the value of W will not be consistent.

2.7.4 *Total Order*

Causal order lets concurrent events happen without ordering them. This is usually a positive feature, because it allows parallel computations to progress without unnecessary constraints. However, the last example of the previous section has shown that in some cases it is useful to order concurrent events. Figure 2.16 points to the solution of the problem that we have identified in Figure 2.15b, by using total order, which can be defined as follows:

Total Order – Any two messages delivered to any pair of participants are delivered in the same order to both participants

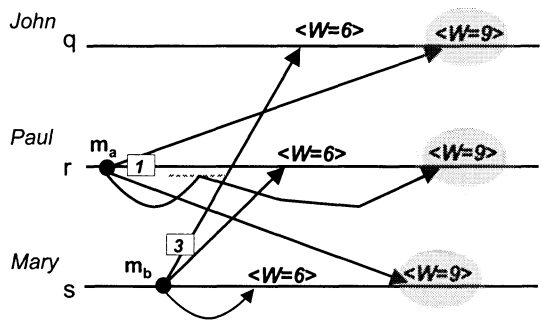

Figure 2.16. Total Order

The need for total order is felt in scenarios such as: achieving determinism of replicated executions in different processes of a distributed system; or ensuring that different participants get the same perception of system evolution and state (same messages in the same order). The latter has also been called *common knowledge* (Halpern and Moses, 1987). The kind of replicated server executing well-defined commands used in the example prefigures a strategy known as the *replicated state-machine* approach (*see State Machine* in Chapter 7). A precondition for all state-machine replicas to behave identically is to execute the same command inputs in the same order. In consequence, a protocol providing

total order may be a helpful device. However, note that total order is orthogonal to causal order, i.e., there may exist combinations of the two paradigms, such as FIFO, causal or non-causal total order protocols.

2.7.5 *Temporal Order*

Logical order is based on a simple observation: if participants only exchange information by sending and receiving messages, they can only define causality relations via those messages. However, participants can interact without necessarily exchanging messages through a given logical order protocol:

- by exchanging messages via a protocol other than the ordering protocol;
- by interacting via the outside world.

In both cases there are *hidden channels*, that is, information flowing between participants which is not controlled by the ordering discipline, so to speak, taking place in a clandestine manner. These messages subvert causal delivery, since they are not subjected to the ordering discipline (Veríssimo, 1994). The first anomaly is well-known, and its most common example is the mixed use of a protocol for logical order and another protocol, for example, low-level O.S. protocols, such as RPC, distributed file system, or shared memory protocols. The problem is exemplified in figure 2.17a: m_2 is issued because of the RPC that the top sender executed, so in fact it is preceded by m_1 (note the dashed causal chain); nevertheless, for the protocol they are concurrent and m_2 happens to be delivered before m_1; m_3 may carry the undesirable effects of this order violation.

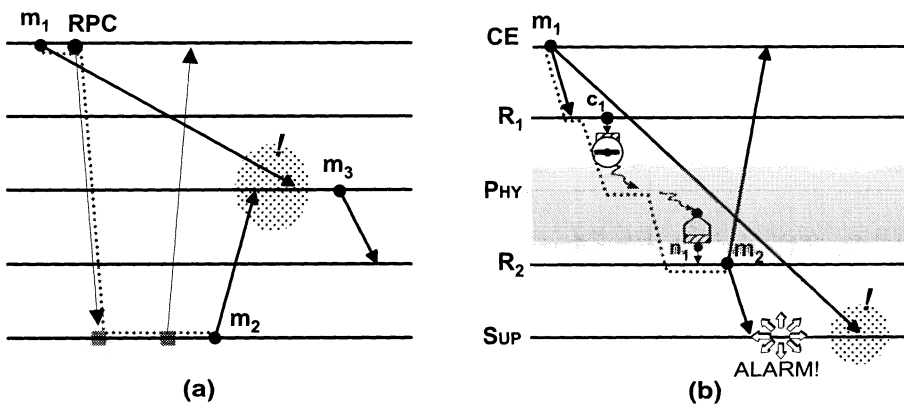

Figure 2.17. Hidden Channel Examples: (a) Other Protocols; (b) Physical Feedback

Less known are hidden channels developed by means of feedback paths through the environment. This can happen with any device, but is common in process control. Again, logical order implementations cannot possibly know about these paths. Figure 2.17b presents such an example. A physical process PHY is under the control of CE. SUP is a supervision unit which detects anoma-

lies and handles alarms. CE issues an output command message m_1 to valve controller R_1, copied to SUP. R_1 issues the physical actuation command (c_1). As a consequence of feedback through PHY, a fluid detection sensor notifies (n_1) its controller R_2, which signals the fact to CE and SUP through m_2. In the example, m_1 arrives later than m_2 to SUP, because the protocol does not know about c_1 or n_1. If you follow the dashed causal chain, it is obvious that there is a causality violation. The consequence in this application is that SUP issues a `leakage!` alarm, whereas the system is functioning perfectly.

How is this problem solved? Inasmuch as it is undesirable to have a discrimination of physical order for events separated by unnecessarily small intervals, it should be possible to evaluate the minimum interval that is relevant to define potential causality. In a distributed computer system or in a physical process, it takes a finite amount of time for an input event (e.g., deliver) to *cause* an output event (e.g., send). For example, the time for an information to travel from one site to the other; the execution time of a computer process; the feedback time of a control loop in a physical process. Supposing δ_t is that minimum time for a given system, we can call it δ_t-**precedence**, to mean that two events have a potential causal relation only if they are separated by more than δ_t. This definition is more accurate than a mere physical order. As a result, we can formulate a useful definition of temporal order:

Temporal Order – A message m_1 is said to temporally precede ($\overset{\delta_t}{\rightarrow}$) m_2 **iff**: m_1 is sent before m_2 by more than δ_t, i.e.,
$t(send(m_2)) - t(send(m_1)) > \delta_t$

According to the definition of δ_t-precedence and to the definition above, a protocol delivering messages in temporal order secures causal delivery even if there are hidden channels, which is not guaranteed by logical order protocols.

2.7.6 *Ordering Algorithms*

There are many ordering algorithms both to enforce causal order and to enforce total order. We will address both types of protocols in the following sections.

Causal Order algorithms The purpose of a causal order algorithm is to ensure that messages are delivered to the application in ways that respect causal order, i.e., if m_1 and m_2 are to be delivered to the same process, and $m_1 \rightarrow m_2$, then m_2 is delivered after m_1. One of the most intuitive ways of enforcing causal order is to make every message carry its own causal past. In order to do so, we need to keep at each process p a list of messages that we will call $past_p$. This list is used as follows:

- When a message is sent, it carries the *past* of its sender in a control field. Note that this field can be much larger than the data field itself, since it may contain several messages.
- After sending a message m, the sender adds m to its *past* list.

- When a message m is received, its $past_m$ field is checked. Messages in $past_m$ that have not yet been delivered, are delivered to the application. These messages are added to the *past* list of the recipient. Then, the received message m itself is delivered to the application. Also, the message is added to the *past* of the recipient.

It can be proved that if $m_1 \rightarrow m_2$ then m_1 is in the *past* list of the sender of m_2, and thus it will be delivered before m_2 according to the rules described before. A variation of this protocol was actually used in one of the earlier implementations of causal order, by Birman and Joseph (Birman and Joseph, 1987). It exhibits the very positive feature of never delaying message delivery. Since a message m carries its own past, messages in m's past can be delivered as soon as m is received. The negative side is, of course, the very large size of the control field. Besides, the simple protocol as described is impractical because the *past* list (and fields) grows indefinitely. It has to be augmented with a fundamental mechanism of any practical causal order protocol: some way to purge obsolete information from the *past*. For instance, a message that has been delivered to all intended recipients can be discarded.

In today's networks, where message omissions are infrequent, to send the whole message in the *past* field is an overkill. It is very likely that in most of the runs all messages in $past_m$ will have been received and delivered when m is received. One way to reduce the size of the control information is to store and exchange only message identifiers in *past*, instead of complete messages. This approach assumes that a third-party component is responsible for providing guaranteed delivery, i.e, if a message is lost it is somehow automatically retransmitted until delivered to all intended recipients. The rules regarding sending and receiving messages must be changed to fit this approach:

- When a message is sent, it carries the *past* of its sender in a control field. Note that this field contains only message identifiers. However, it can still be much larger than the data field itself, since it may contain many identifiers.
- After sending a message m, the sender adds m's identifier to its *past* list.
- When a message m is received, its $past_m$ field is checked. If $past_m$ contains messages that have not been delivered yet, the message is put on hold, until these messages arrive and are delivered.
- When all messages in $past_m$ have been delivered, the message m is delivered to the application. Also, the message identifier is added to the *past* structure of the recipient.

Note that now messages are forced to wait until all messages in its past are received and delivered. Note also that the approach mitigates the problem of large control fields, but does not solve it completely, i.e., we still need a way to remove obsolete information from *past*. The question is: even if we remove all obsolete information from *past*, what is the worst-case size of the control structure needed to ensure causal order *without* enforcing additional synchronization in the system? It can be proven that, in the general case, the control information required has n^2 size, where n is the number of processes

in the system. This means that at least one message identifier needs to be maintained for each pair of communicating processes.

Consider a system of N processes, p_1 to p_n. The following method can be used to code and store causal information. Each message is identified by the identity of its sender and a sequence number local to each process p. This sequence number can be seen as a local clock that counts send events; it is not synchronized with the local clocks of other processes. Each process logs the sequence number of the last message that it has sent to each of the remaining processes. This information is kept in an array named $SENT$. For instance, if we have four processes, and the $SENT$ array at p_1 is $SENT1 = [0, 2, 4, 2]$, we can infer that the last message sent by p_1 was message number $(p_1, 4)$ that was sent to p_3. As we have seen before, the causal past of a process is made not only of the messages that the process has sent, but also of the causal past of messages delivered by that process. Thus, in order to capture its own causal past, each process has to log his own knowledge about the messages that the other processes have sent. In other words, its own causal past is made of its own $SENT$ array and of an approximation of the $SENT$ arrays of other processes. This resulting control information is often represented as a matrix, also called a *matrix clock*, where each line represents one of the $SENT$ arrays just described. Thus, the element $MATRIX_k[i, j]$ of the matrix keeps the sequence number of the last message sent by process i to process j, as known by process k.

It is worth to note that even a matrix of size n^2 is an excessive amount of control information in systems where messages are short, and can thus represent a significant overhead. Thus, much effort has been made in order to reduce the size of the control information that needs to be kept and exchanged. For instance, if processes only send multicast messages (*i.e.*, all messages are sent to all processes), all elements in the same line of the matrix have the same value, thus the matrix reduces to a *vector* of size n, called a *vector clock*.

Consider the example of Figure 2.18a. When process p_1 sends m_1 its clock is updated to $[1, 0, 0]$. Processes p_2 and p_3 update the same entry of their clocks when they deliver m_1. The same reasoning applies to messages m_2 and m_3. Figure 2.18b shows how the vector clocks can be used to enforce causal delivery. Message m_5 is sent by p_2 after the delivery of m_4, so it is timestamped with a clock value of $[2, 1, 1]$; this means that m_5 should be delivered after m_2, m_3 and m_4 (and, transitively, m_1). Thus, if because of network delays m_5 is received before m_4 at p_3, its delivery is delayed until causal order can be ensured.

Another way to reduce the amount of control information is to decrease the degree of concurrency in the system. In fact, the n^2 bound only applies to systems where we want to keep an exact track of causal dependencies, i.e., where one can always say, by comparing two matrix timestamps, if two messages are causally related. If we are ready to accept more imprecise information, we can reduce the size quite effectively. Consider the following scheme to implement causal order, which only requires a single integer value to be kept and exchanged (this idea was proposed originally by Leslie Lamport, so the logical clock is often called a Lamport clock):

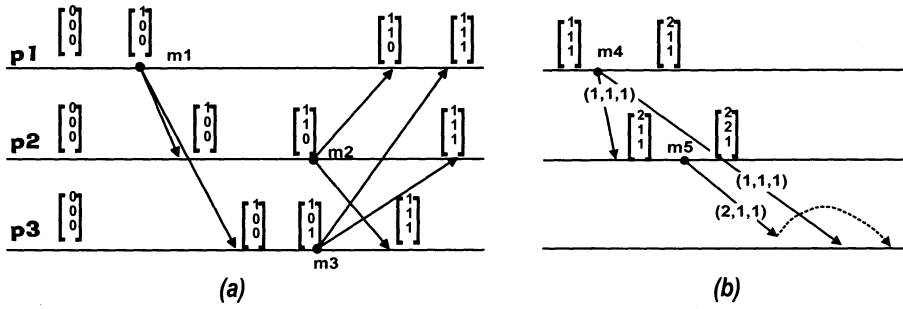

Figure 2.18. Vector Clocks: (a) Principle; (b) Practical Use

- Each process keeps a single integer called *logical clock*, or simply *lclock*.
- When a message is sent, it carries its sender *lclock* in a control field. The *lclock* is incremented.
- Messages are exchanged using FIFO channels, thus two messages from the same sender to the same destination are received in the order they were sent.
- When a message is received, it is placed in a waiting queue, ordered according to its *lclock* (the sender identifier is used to order messages with the same *lclock*). The message is kept in waiting state until a message with equal or greater *lclock* is received from *every* sender in the system. By the time this condition becomes true, because of FIFO channels, all messages with smaller timestamp have also been received. The message becomes *deliverable*.
- A message can be delivered if it is in the *deliverable* state and it is at the head of the waiting queue. When m is delivered, the recipient $lclock_p$ is updated according to the following rule: $lclock_p = \max(lclock_p, lclock_m)$

According to this scheme, if $m \rightarrow n$ then $lclock_n > lclock_m$. However, the opposite is not true, i.e., if $lclock_p > lclock_o$ this does not necessarily mean that $o \rightarrow p$. Thus, the protocol orders more messages than those actually needed to be ordered in order to preserve causal relations. In this scheme, the delivery latency is bounded by the slowest sending rate in the system. Finally, it is worth noting that it also possible to exploit knowledge about the topology of the communication patters to further reduce the amount of information exchanged (Stephenson, 1991; Rodrigues and Veríssimo, 1995).

Total order algorithms The goal of a total order algorithm is to ensure that all messages are delivered to all recipients in the same order. *How to achieve this goal?* Actually, in the previous section we have just described one way to do it. Let us look again at the algorithm used to implement causal order. Messages are delivered according to the order of their timestamps, but a message is only delivered when all messages with lower *lclock* values have been received and delivered. Now assume that all messages are sent to all participants and that a deterministic rule is used to order messages that have

the same logical clock (for instance, according to the lexical order of their senders). Then, since the ordering criteria are the same everywhere, messages are delivered in the same order at every process. Algorithms in this class are called *symmetric* algorithms, since all processes execute the same steps.

Symmetric algorithms are interesting for several reasons. To start with they are simple to implement and rely on a single mechanism (logical clocks) to enforce both causal and total order. Also, when all processes are sending messages, total order can be established without any additional exchange of control messages, so their overhead is relatively small. However, since a message needs to be received from every process to ensure total order, the latency of message delivery is limited by the rate of the slowest process in the system. It is possible to alleviate this problem by making the delivery condition depend on just a majority of processes in the system (Dolev et al., 1993) but the best results with symmetric algorithms are obtained when all processes are sending messages at a fast pace and have their clocks synchronized (Rodrigues et al., 1996).

Note that if synchronized clocks are available, they can be used instead of logical clocks to timestamp messages. In this case, the protocol is able to deliver the messages according to their "birth" time. Additionally, if the system is synchronous and it is possible to assume a worst-case message delivery time Δ, one can use the passage of time as a mechanism to be sure that no message with a lower timestamp is going to be received. This is implemented by the Δ-*protocols*. *See* again the example of Figure 2.11 in Section 2.6: let $c(m)$ be the timestamp of m, and let $\Delta = \delta_{mx} + \pi$, δ_{mx} the maximum delivery delay and π the precision of clocks. This implies that by $c(m) + \Delta$, all messages sent at or before m must have reached their destination. Thus, by that time one can safely deliver m and obtain a total order, e.g., by delivering messages everywhere in timestamp order, and messages with the same timestamp in lexicographic order of senders. This overcomes the above-mentioned problem of having to wait for a message with a higher timestamp from every other process but unfortunately, unless specialized high-performance architectures are used, the value of Δ can be quite large. Note that the protocol also enforces a causal order.

A completely different approach consists in selecting a special process in the system and assigning it the task of ordering all messages. This process works as a *sequencer* of all messages and is often called the *token* site. The protocol works as follows: all senders send their messages to the sequencer; the sequencer assigns a unique sequence number to all messages; and it retransmits them back to all recipients. The total order is the order by which the sequencer processes the incoming messages. This scheme is illustrated in Figure 2.19a. A variant is illustrated by Figure 2.19b, in which case the messages are sent in multicast to all processes and the sequencer just disseminates the sequence number assigned to each message.

This scheme can be quite fast in systems where the network latency is small as illustrated by the work of Kaashoek et. al (Kaashoek and Tanenbaum, 1991). Of course, this scheme offers best results for those messages sent by the sequencer itself. Due to this reason, some systems dynamically move the

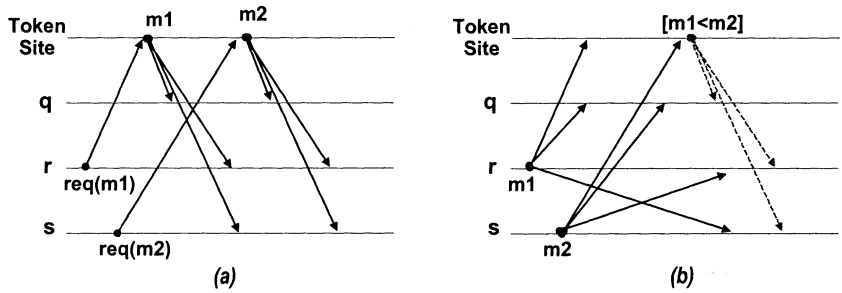

Figure 2.19. Total Ordering with Sequencer: (a) Point-to-Point Send; (b) Broadcast

sequencer to the node that is producing more messages (Birman et al., 1991a). If all processes are producing many messages, one can simply rotate the role of sequencer among all nodes, using a token-passing scheme (Amir et al., 1993b). In these protocols, the failure of the sequencer (or token holder) poses a problem of unavailability and may even require complex recovery procedures to secure a correct ordering. One way to preserve the ordering information established by the sequencer in case of failure is by requiring the sequence numbers to be known by a quorum of nodes before delivering the messages (Chang and Maxemchuck, 1984). There are other alternatives to implement total order. For instance, the best of the two previous approaches can be combined using a hybrid protocol (Rodrigues et al., 1996). It is also possible to use properties of specific networks, for instance by using a shared medium as a physical sequencer, as suggested in (Veríssimo et al., 1989; Cart et al., 1987; Rufino et al., 1999) or properties of particular message dissemination strategies, such as a shared spanning tree (Schneider et al., 1984; Garcia-Molina and Spauster, 1991).

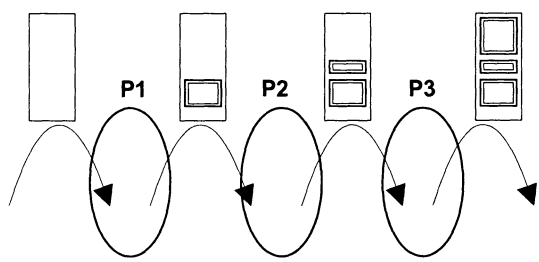

Figure 2.20. Assembly Line of Producer-Consumer

2.8 COORDINATION

There are many ways different processes can interact among each other: they can explicitly exchange messages, access shared regions of memory or access shared resources, including physical devices such as printers, plotters, etc.

PRODUCER

```
00  while (1) {
01        item = produce();
02        while (n==MAX)
03              ;
04
05
06        buffer[in] = item;
07        in = (in+1)% MAX;
08        n++;
09
10  }
```

CONSUMER

```
20  while (1) {
21        while (!n)
22              ;
23
24
25        item = buffer[out];
26        out = (out+1)% MAX;
27        n--;
28
29        consume(item);
30  }
```

Figure 2.21. Naive Producer-Consumer

Consider for instance a classical producer-consumer relationship. In this pattern of activity, a process designated the *producer* creates items that need to be processed by another process called the *consumer*. Note that the consumer may in turn produce a result to be processed by another consumer, creating an "assembly-line" of processes, as illustrated in Figure 2.20. This is one of the possible strategies to parallelize a task, using functional decomposition. Each element of the chain, sometimes called a *filter*, is specialized in a specific processing step, and n items can be processed concurrently in a chain of n filters.

Note that several coordination issues appear in this simple example. In order to implement this structure, the producer has to handout the item to the consumer. Assume that the item is exchanged by placing it in some shared device, with a limited amount of storage; to simplify the example, assume that the device just has space for a single item. Clearly, the producer and the consumer have to coordinate the access to the device. The producer can only put the item in the device when the device is empty; the consumer can only pick the item when the device is full. If there are several producers, we must also prevent more than one producer from trying to put an item simultaneously on the device. In consequence, the processes must coordinate in order to ensure that when a process is accessing the device other processes are excluded from doing so. This is known as the *mutual exclusion* problem.

2.8.1 Basics of Synchronization

As we have already mentioned, there are two main paradigms to support communication and synchronization in concurrent programs: shared memory and message passing. In this section we start by discussing some important issues regarding shared memory systems.

Let us go back to our producer example, assuming that the consumer and the producer exchange items using a portion of shared memory, in our example

PRODUCER

```
00  while (1) {
01        item = produce();
02        // there is still a
03        // bug here
04        while (n==MAX)
05              ;
06
07        begin-mutual-exclusion;
08        buffer[in] = item;
09        in = (in+1) % MAX;
10        n++;
11        end-mutual-exclusion;
12  }
```

CONSUMER

```
20  while (1) {
21        // there is still a
22        // bug here
23        while (!n)
24              ;
25
26        begin-mutual-exclusion;
27        item = buffer[out];
28        out = (out+1) % MAX;
29        n-;
30        end-mutual-exclusion;
31        consume(item);
32  }
```

Figure 2.22. Naive Producer-Consumer with Mutual Exclusion

an array of size `MAX`. The code for the consumer and the producer may be something like depicted in Figure 2.21. The reader does not have to be too familiar with concurrent programming to check that the program does not work! We may not predict the order by which processes execute operations, and in fact the interleaving of the executions depicted may yield unexpected results. Consider that: the buffer has a size `MAX=10`; it is empty (`N==0`), the first free position being `IN=1`; and two producers p_1 and p_2 execute the code to add an item to the array. Since the array is empty, they both pass the guard at line 02. Now imagine the following interleave of operations: p_1 executes line 06, p_2 also executes line 06, p_1 executes lines $07-08$ and p_2 also executes lines $07-08$. *What happens?* We will have `IN==3` and `N==2` which is correct, but p_2 will place its item in position 1, the same position previously used by p_2, and in consequence no item will be placed in position 2.

This is again a *mutual exclusion* problem, now in the access of a shared data structure. The program is designed to work correctly only if processes access the shared array (and its associated control variables, `N` and `IN`) in isolation. To achieve this goal, one could add explicit instructions in the code to mark the beginning and end of the code that must be executed in mutual exclusion, also known as a *critical region*, as illustrated in Figure 2.22.

The purpose of the `begin-mutual-exclusion` and `end-mutual-exclusion` guards is to make sure that at most one process is executing the code between the guards at any given moment. When access is granted to a given process, other processes must wait until the critical region is released. But how to implement these primitives? One can use the concept of a *lock*, a mechanism that can have two states: opened and closed. In order to access a critical region, a process must find the lock open, even if it has to wait for that, and then close it. In order to leave the critical region, the process merely releases the lock.

NAIVE LOCK IMPLEMENTATION

```
00  shared:
01      LOCK lock = OPEN;
02
03  begin-mutual-exclusion is
04      while (lock==CLOSED)
05          ;
06      lock = CLOSED;
07  end;
08
09  end-mutual-exclusion is
10      lock = OPEN;
11  end;
```

LOCK USING TEST-AND-SET

```
00  shared:
01      LOCK lock = OPEN;
02
03  begin-mutual-exclusion is
04      while (test-and-set(lock))
05          ;
06  end;
07
08
09  end-mutual-exclusion is
10      lock = OPEN;
11  end;
```

Figure 2.23. Lock Implementation: (a) Naive; (b) Using test-and-set

The algorithm is simple but only works if we have a lock, so now we need to implement one. Maybe a boolean variable can do the job, as is illustrated in Figure 2.23a?

Unfortunately, this code suffers from exactly the same concurrency problems of the previous example. When the lock is released, it is possible to have several processes evaluate the guard of line 04 and enter the critical region before the `lock` is set to `CLOSED`. If you are a beginner in concurrent programming, you may feel discouraged right now. Even a simple boolean lock is hard to get right! Are there solutions to the problem at all?

The answer is yes, of course. The crucial problem with our naive lock implementation is that it cannot not ensure the indivisibility of the test-lock and set-lock sequence in lines 04 – 06. Although there are other approaches, a lock is normally implemented through an atomic *test-and-set* CPU instruction, enforced by hardware. Using such an instruction, the code can be simply re-written as illustrated in Figure 2.23b, which works correctly.

Although correct, the previous code is inefficient, since a waiting process consumes processor cycles until the lock is released. This behavior is often called *busy-waiting* and a lock implemented this way is also called a *spinlock*. Spinlocks are efficient when the critical region is known to be short (thus, the process will not wait for many cycles), but they only work when the concerned processes run in parallel, such as multiprocessors, or uniprocessors with co-processors (e.g. I/O). Otherwise, while the waiting process loops there is no chance for the lock-holding process to run and release it.

Since the implementation of the *lock* and *unlock* primitives may involve the use of resource unfriendly busy waiting procedures (or the access to other critical resources, like temporarily disabling interrupts), these services are usually supported by the operating system kernel itself. This allows the implemen-

```
PRODUCER                                CONSUMER

00  while (1) {                        20  while (1) {
01      item = produce();               21      wait(sem-items);
02      wait (sem-free, 1);             22      wait (mutex);
03      wait (mutex);                   25      item = buffer[out];
04      buffer[in] = item;              26      out = (out+1) % MAX;
05      in = (in+1)% MAX;               27      signal (mutex);
06      signal(mutex);                  28      signal (sem-free);
07      signal(sem-items);              29      consume(item);
08  }                                   30  }
```

Figure 2.24. Producer-consumer with semaphores

tation of obvious optimizations such as preventing the scheduling of blocked processes until the resource is released.

Djisktra proposed an abstraction for synchronization that is more powerful than locks. This mechanism, called *semaphore*, exports two operations called `wait` and `signal` respectively. The semaphore holds a number of units, which is defined when the semaphore is created. The `wait(n)` primitive decrements `n` units from the semaphore. It is blocking if the semaphore does not hold enough units to satisfy the request, in which case the calling process waits until enough units are available. The `signal` primitive is always non-blocking, and increments the number of units available in the semaphore. A *mut*ual *ex*clusion lock, also called a *mutex*, can be implemented using a one-unit semaphore. The `lock` operation decrements one unit and the `unlock` increments one unit. However, semaphores can be used to express semantically richer concepts, such as the number of occupied or free entries in the shared array of our producer-consumer example. The code of our example, re-written to use semaphores, is presented in Figure 2.24.

Other similar synchronization constructs, such as *conditional regions*, *sequencers and event counters*, or *barriers* have been proposed and supported by several systems. However, these mechanisms are often considered too low-level and difficult to master directly. Thus, a significant body of research exists on mechanisms to support the programming of concurrent applications, including languages with explicit support for concurrency (Ada is a good example).

2.8.2 Distributed Mutual Exclusion

The shared memory model is not trivial to implement in distributed systems, specially when a distributed system is defined as a collection of processes that communicate solely by exchanging messages. Although a *distributed shared memory* abstraction can be implemented on top of a message passing system, it is easier to start by discussing how synchronization problems can be solved using message passing alone.

Let us start with the problem of mutual exclusion in distributed systems. This is an interesting problem and the quest for a solution is an excellent way to get acquainted with the subtleties of distributed algorithms. As a general rule, it is simpler to start by considering a distributed solution that relies on centralized control. This can be achieved by putting most of the logic on a central server with which the other processes exchange messages.

For the mutual exclusion problem, we can for example build a *lock server*, which keeps the identity of the current holder of the critical region and manages a queue of clients waiting for their turn, as illustrated in Figure 2.25a. The server and clients execute a simple algorithm. When a process wants to access the critical region, it sends a LOCK request to the server. When the server receives the LOCK request it immediately sends back a LOCK-GRANTED reply if the region is free; otherwise, it inserts the client request at the end of the waiting queue. To release the critical region, a process simply sends an UNLOCK request to the server. The server picks the first request in the waiting queue, if any, and sends the LOCK-GRANTED reply to the associated client.

This solution has a number of drawbacks that are worth being enumerated. To start with, the lock server is a single point-of-failure. If it crashes, information on who was holding the lock and on the relative order of waiting processes is lost. The failure of a client that holds the resource also wrecks the algorithm, since no UNLOCK message will ever be sent. Finally, the lock server may be a bottleneck for system performance. However, we leave these juicy fault tolerance aspects for Part II of the book and focus on the functional aspects of the algorithm.

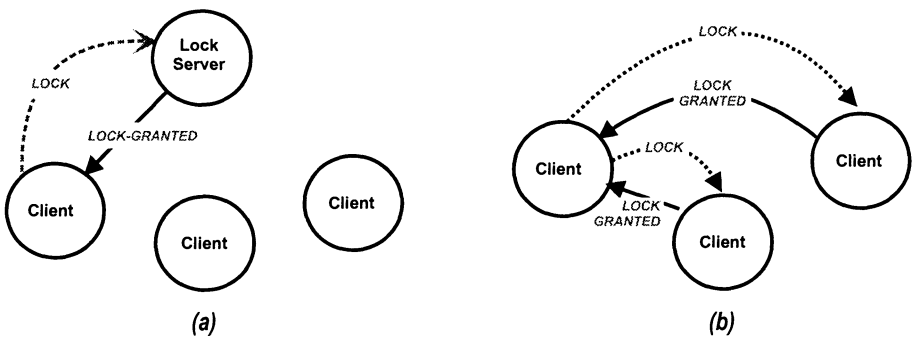

Figure 2.25. Distributed Mutual Exclusion Control: (a) Centralized; (b) Distributed

Following our piecemeal approach, we now try to make the previous algorithm fully decentralized. One possible way is to replicate the state that was kept in the central server, in every system process. In this sense, we would change our interaction style from client-server to multipeer, where all processes interact in a conversational manner, or if you prefer, where all processes are clients and servers at the same time. To keep the same design, a process requesting access to the critical region would send a LOCK request to every other

process, and would wait for a LOCK-GRANTED reply from every other process as well, as illustrated in Figure 2.25b. Finally, when releasing the critical region, it would send an UNLOCK request to all other processes.

Unfortunately, things are not that easy! Suppose the following scenario: there are four processes p_1, p_2, p_3, and p_4; the critical region is free; and p_1 and p_2 try to acquire the lock at the same time. Assume that p_1's request is received first at p_3, and p_3 grants access to p_1. Now suppose that p_2's request is received first at p_4: p_4 will grant access to p_2, a decision inconsistent with the decision of p_3. Worst: actually this scenario would result in *deadlock*, since neither p_1 or p_2 would obtain the lock but would prevent each other and other processes from obtaining it.

However, we have met this problem before and the solution is easy. This is a replicated computation requiring totally ordered delivery to all replicas— of the LOCK request messages in this case— so that they remain consistent (*see Total Order* in Section 2.7). This can be achieved using a totally ordered multicast protocol. Another approach consists in merging the total order and the mutual exclusion algorithms. The following algorithm, proposed by Lamport (Lamport, 1978b), achieves this goal.

The algorithm can be seen as an extension of our bogus decentralized algorithm. It relies on FIFO channels between processes and on logical clocks. All messages are timestamped with the logical clock of the sender, and clocks updated whenever a message is sent or received (*see Ordering Algorithms* in Section 2.7). When a request is received it is inserted in the waiting list, which is now organized in the order of the request timestamps. The receiving process sends every other process an ACK message. As soon as an ACK has been received from every other process, the request is marked as *stable*. There is no need to send explicit LOCK-GRANTED messages, since a process enters the critical section when: the resource is marked as free; its own request is at the head of the waiting list; *and* the request is stable.

Ricart and Agrawala (Ricart and Agrawala, 1981) have proposed an optimization of the previous algorithm based on the observation that no process can enter the critical region until its request is fully acknowledged. Thus, the process that holds the resource can simply defer all acknowledgments until it releases the section. This optimization avoids the exchange of explicit UNLOCK messages.

2.8.3 Leader Election

As we have seen, it is often simpler to solve a distributed problem using an approach that relies on centralized control, materialized in a control server. However, this has the drawback that the system becomes unavailable when the server crashes. A possible strategy is to allow any process to assume the role of the centralized server, and to use a *distributed leader election* algorithm to select, in run-time, which process should play this role. This allows the system to survive failures of the server.

Leader election has similarities with mutual exclusion. In some sense, the mutual-exclusion algorithm "elects" which process is granted access to the critical region. Using this observation, we can design a leader-election algorithm based on the distributed mutual-exclusion algorithm presented in the previous section.

The algorithm works as follows: every process in the system requests the lock; the first process to be granted access becomes the leader. Although this algorithm works, it is possible to make optimizations based on the specific requirements of the leader election problem. For instance, in leader election, a process can abort the execution of the algorithm as soon as a leader is elected (whereas in mutual exclusion, each requesting process eventually wants to access the resource). Furthermore, if a process receives a LOCK request from another process p it may support p's election, instead of competing to become the leader.

Let us see how the previous approach works when all processes try to become leaders at the same time. Each process sends a LOCK request message. Since messages are concurrent, all carry the same timestamp value (say, 1) and would be ordered using the order of process identifiers. In this case, the process with the lowest identifier would always be granted the resource, i.e., it would always be elected leader. This is a simple and acceptable outcome in most applications. Such an algorithm was proposed by Garcia-Molina (Garcia-Molina, 1982) and it is known as the *bully algorithm*, as it always elects the strongest active candidate, i.e., the process with the lowest identifier.

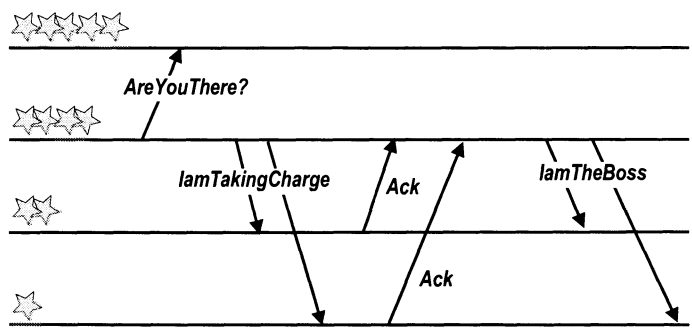

Figure 2.26. Leader Election

Since the idea of a lower process identifier being stronger than a higher process identifier is somehow counter-intuitive, we explain the algorithm in terms of *rank*. Each process has a rank, and the active process with the highest rank wins the election. The algorithm, illustrated in Figure 2.26, works as follows. A given process p knows *a priori* that only a processes with higher ranks can be elected. Thus, instead of sending a message to every process, it just sends a polite ARE-YOU-THERE? message to the higher levels of the hierarchy. If someone replies, p silently gives up its attempt to become the leader, and respectfully waits for one of the processes with higher rank to become the new

PRODUCER

```
00  while (1) {
01      item = produce();
02      wait (mutex);
03      wait (sem-free, 1);
04      buffer[in] = item;
05      in = (in+1) % MAX;
06      signal(mutex);
07      signal(sem-items);
08  }
```

CONSUMER

```
20  while (1) {
21      wait(sem-items);
22      wait (mutex);
25      item = buffer[out];
26      out = (out+1) % MAX;
27      signal (mutex);
28      signal (sem-free);
29      consume(item);
30  }
```

Figure 2.27. Producer-consumer with semaphores (and bug)

leader. If nobody replies, process p attempts to become the leader by making sure that processes with lower rank know that it is there. This goal is achieved by having p send an I-AM-TAKING-CHARGE message to processes with lower rank and waiting for an acknowledgment from each of these processes. When these acknowledgments have been received (or a timeout occurred, since some of the processes with lower rank may have crashed) p assumes the leadership by sending an I-AM-THE-BOSS message to all processes.

2.8.4 *Deadlock*

In Section 2.8.2, while searching for a solution for the distributed mutual exclusion problem, we were faced with a scenario that caused *deadlock*. Deadlock occurs when two or more processes are waiting for each other in a configuration from which no progress can be made.

Deadlock can also occur in centralized systems, when processes need to synchronize. Consider for instance the solution for the producer/consumer problem depicted in Figure 2.27. The code is almost equal to that of Figure 2.24, but was deliberately changed to allow deadlock to occur. Actually, the error is subtle and often made by beginners in concurrent programming: the producer waits for free space in the shared buffer while holding the `mutex` lock.

The deadlock occurs in the following scenario: a producer obtains the mutex and finds that the shared array is full; it is thus blocked in the `sem_free` semaphore; unfortunately, consumers need to obtain the mutex in order to release space in the shared buffers. The producer cannot make progress (and does not release the mutex) until space is freed and `sem_free` is incremented; the consumers and other producers cannot make progress until the mutex is released.

Deadlock may occur in this case, and in fact in any other case, as long as the following four necessary conditions hold:

- *Mutual exclusion*: some resources are not sharable, and can be held only by one process at a time (in the example, the non-sharable resources are the mutex and the "empty" slots on the shared array).
- *Hold-and-wait*: at least one process waits for additional resources while holding non-sharable resources (in our case, the producer waits for free slots while holding the mutex).
- *No-preemption*: a process that holds a resource is allowed to keep it until it is ready to release it (in our case, the producer is not coded to release the mutex before the item is placed in the shared array).
- *Circular-wait*: There is a circular chain of n processes, where p_0 is waiting for a resource held by p_1, which in turn is waiting for a resource held by p_2, ..., and p_{n-1} is waiting for a resource hold by p_0 (in our case, the producer is waiting for any consumer and consumers are waiting for the producer).

Given that these four conditions must hold for deadlocks to occur, we might think that deadlocks could be *prevented* just by eliminating one of these conditions. And indeed they can, but unfortunately it turns out that eliminating any of these conditions is not a trivial task. Let us understand why.

- Mutual exclusion could be eliminated by having only sharable resources, but this is too restrictive a condition for most applications (e.g., shared data structures are simply non-sharable).
- Hold-and-wait may be eliminated if processes just hold one resource at a time, but again this is too restrictive for most applications (for instance, if a process is doing a bank transfer, it needs to update two accounts). Alternatively, one might force a process to request all the resources it needs a priori, but this may be very inefficient, since some resources are locked long before they are actually needed.
- No-preemption may be eliminated by... allowing preemption, i.e., by forcing a process to release its resources. Although this might sound simple, it is actually complex to implement in practice, since a process that holds a resource is likely to have read or updated the latter, and the correctness of the algorithm it is executing may depend on the state of the resource it is holding. Thus, a process that is forced to release a resource may be forced to *rollback* to a previous point of the algorithm.
- Finally, the circular-wait condition can be prevented by imposing a total order on all resources and forcing processes to acquire all resources by the same order. This suffers from the same drawback as holding all resources simultaneously. The total order defined for the resources may not be the order in which the process needs to access the shared resources.

Alternatives to deadlock prevention are deadlock *detection* and *resolution* or deadlock *avoidance*. Deadlock detection consists in automating the process of detecting deadlocks, usually by detecting existing circular-wait conditions. Deadlock resolution consists in automating the process of breaking the deadlock, usually by aborting one or several of the processes involved. Finally

deadlock avoidance consists in making checks before a resource is given to a process, to make sure that the conditions for deadlock are not met. Distributed algorithms for deadlock detection or avoidance are much harder than their centralized counterparts. The problem with distributed deadlock detection is that the algorithm needs to capture the global state of the computation, in a system where lock requests can be "in transit", i.e., in messages exchanged among nodes. The issue of obtaining a consistent global state will be the subject of the next section.

It is worth mentioning that most systems do not include any built-in mechanisms to prevent, detect or resolve deadlocks and leave this task to the application designer. In fact, the mechanisms described above may introduce a non negligible amount of run-time overhead. Thus, their use should be reserved to applications where deadlock-free code cannot be obtained trivially by careful programming.

2.9 CONSISTENCY

Informally, we can say that the system state is consistent if it does not violate some integrity constraint imposed upon its specification. In this section we will briefly discuss some mechanisms that can be used to enforce consistency and to verify that the system state is consistent.

2.9.1 Consistent Global States

In an active distributed computation, sometimes one needs to assess that some global property is verified, for instance, that there is still a token rotating, or that deadlock was not reached. In order to obtain such information, one needs to take a *snapshot* of the global system state. The global state is a set of local states taken at given points of the execution of each process. Due to this reason, the global state is also called a *cut* (*see Formal Notions* in Section 1.4). The main difficulty is that this snapshot can only be obtained by machines within the system. Additionally, it is not easy to obtain a global state in a dynamic system, where the state if each individual process is continuously changing.

To give an example in a non computer-related scenario, imagine that you go to a Star-Trek convention, where a large number of fans are trading "collector" cards with the pictures of the popular characters from the series (for those that do not live in this planet, Star-Trek is an old sci-fi TV serial; due to some obscure reason, people that spend too much time in front of a computer tend to be attracted by this show). Your job is to obtain the exact number of cards circulating in the convention hall (the global state). You can easily imagine yourself overwhelmed by such an outstanding task, wandering around among dozens of fans wearing funny outfits and constantly permuting cards, asking yourself how many times did you count the same card.

The more computer-related example of finding the sum of two accounts, A and B, will guide us through the problems in getting consistent global states. You know that the sum must be 700$, but the user can freely transfer money

from one account to the other just by exchanging messages between the nodes holding the accounts. This is illustrated in Figure 2.28, where 50$ are sent from node A to node B. The figure illustrates the succession of local states at each node. Assume that you want to take a system snapshot and try to do so from a node C, not depicted in the figure, solely by exchanging messages with the target nodes A and B.

Consider the case of cut 1, where the state of node A is taken after the transfer has been sent but the state of node B is taken before the same transfer has been received. The global state would sum 650$ because the amount in transit has not been taken into account. Cut 2 is more awkward. The state of node A is taken before the transfer message has been sent and the state of node B is taken after the transfer message has been received and processed. In this case the (inconsistent) global state would show more money than there really is in the system. So, the figure illustrates how easy it is to end-up with a global state that is *inconsistent* with the system operation.

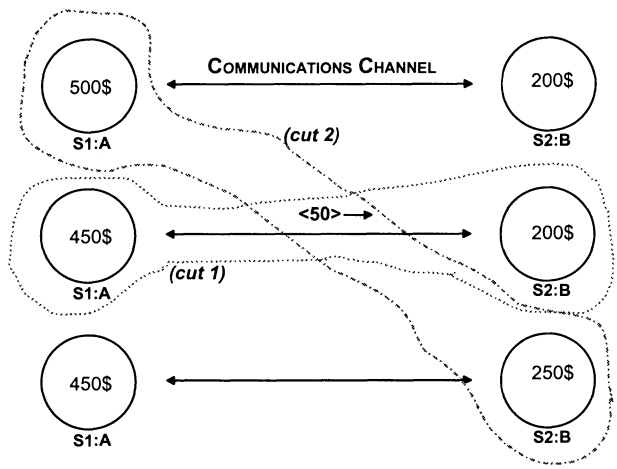

Figure 2.28. Ad-hoc State Snapshots

It seems that the problem consists in not taking into account the messages in transit. Let us change the previous approach, including the messages received and sent by each node in the snapshot. In cut 1, the state of node A contains the value of the local account (450$) and a record of having sent 50$ over the wire; the state of node B contains the value of the local account (200$) but has no record of having received the transfer. Is this cut consistent? If the goal of the global state is to assess the global amount of money in the system, it is quite obvious that the cut should not contain any messages in transit. A cut with such property is named *strongly consistent*. Therefore, cut 1 is not strongly consistent. However, if the application that is going to analyze the snapshop is able to take into account that messages can be in transit, cut 1 does not contradict the expected system behavior: the total amount is less than 700$

because there is some money in transit; it may even wait for those messages to arrive and update the state. Since this global state still makes sense, we call cut 1 *weakly consistent* or just consistent. Consider now cut 2. The state of node A has no record of sending any message. However, the state of node B includes a transfer sent by A. Cut 2 is clearly *inconsistent* with the expectable system evolution: it should not be possible for B to receive a message from A when A has no record of having sent such messages. Thus, the snapshot resulting from cut 2 is of little practical use.

Figure 2.29 shows the problem of inconsistency in an intuitive way. On the left, we show the history of processes, and a cut obtained ad-hoc. It can be proven that a cut is only consistent if all events in the cut are concurrent (i.e., if there is no causal relation between any two events of the same cut). Now, observe that there is a causal chain between events c_{21} and c_{22}, created by m_1, which tells us that the cut depicted in the figure is not consistent. A visually intuitive manner of checking if a cut is inconsistent is to stretch the line that represents the cut such that it becomes a straight vertical line, as represented on the right of the figure. The cut is inconsistent if there is a message that crosses the cut from right to left.

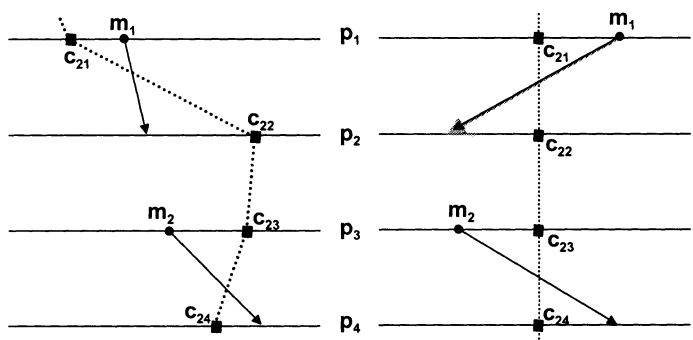

Figure 2.29. Inconsistent Cut

From the previous examples it should be clear that trying to obtain a global state simply by contacting, one by one, each of the target processes without any additional coordination may lead to inconsistent cuts. Short of a better solution, we have to stop the system to obtain a consistent cut, as done in a number of commercial settings. The interesting question is how to devise a protocol that obtains consistent cuts while the system is active. To illustrate the problem, we will describe one of the first *snapshot protocols*, by Chandy and Lamport (Chandy and Lamport, 1985).

The protocol assumes FIFO channels connecting the processes and uses a special control message, called a MARKER, to distinguish messages sent *before* the snapshot from those sent *after* the snapshot. The global snapshot includes the local state of each process and the state of each channel. Each process is responsible for capturing the state of all its incoming channels. The algorithm

is initiated by some process. That process saves its state and then sends a MARKER through all its outgoing channels. It then continues its normal operation and waits until it receives a marker from all its incoming channels. The state of those channels is captured as "containing" all messages received *after* the local state of the process has been saved and *before* the marker is received.

Other processes behave in a similar manner. When they receive the marker for the first time they immediately save the state of the channel from which the marker is received as *empty*. Then they save their own state and send markers through all outgoing channels. Finally, they wait for markers to be received from the remaining incoming channels. When a marker is received through some channel and the local state has already been saved, the corresponding channel state can be recorded. The algorithm terminates when states from all processes and all channels have been captured.

2.9.2 Distributed Consensus

Consensus is a fundamental problem in distributed computing, abstracting a wide class of related problems that appear in the design of distributed applications. In an informal way, the goal of consensus is to make a set of processes agree on a single value that depends on the initial values of each of the participants (this excludes the trivial solution, where all agree on some pre-defined fixed value).

Consider the following example: there is a manufacturing cell where items of different sizes and forms are packed. Items arrive to the cell through a belt, one at a time. To speedup processing, a set of packing machines are available to serve requests. When two or more machines are available, one has to decide which machine picks the next item. One solution to this problem would be to create a centralized dispatcher, in charge of selecting the packing machine for each item. Can this problem be solved in a decentralized way by executing some protocol among the machines themselves? The answer is yes, if you have a consensus module. This problem can be expressed in the following way: for each item, the machines must reach consensus about which one picks the item.

Let us devise a simple protocol to solve this problem. Assume that all machines are numbered. If a machine is free when a new item arrives, it proposes its own number to serve the item. If a machine is busy when an item arrives, it simply waits for: *i*) becoming free to serve that item; or *ii*) receiving some proposal from another machine; in the latter case it supports the proposal of the other machine. Once a machine has proposed some value, it does not change its mind. For instance, if machine number 2 has voted in favor of machine number 1, it will not propose another machine for the same item. Since several machines can be free or become free in a concurrent way, different proposals can be sent. Consensus can be reached simply by collecting a proposal from every machine, discarding the duplicates, ordering the proposals according to the machine number and picking the first proposal in that set. Since all machines receive exactly the same set of proposals, and the selection algorithm is deterministic, consensus is reached.

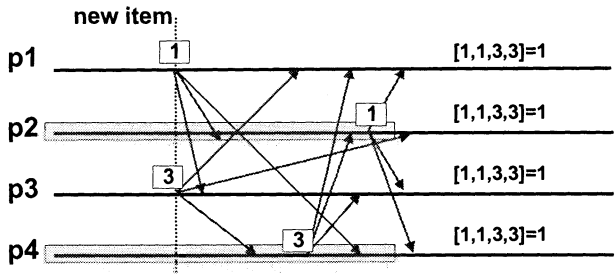

Figure 2.30. Simple Consensus Protocol

The operation of the protocol is illustrated by Figure 2.30. In the example, when a new item arrives processes p_2 and p_4 are busy, so only p_1 and p_3 offer to pick the job. Process p_2 receives the proposal from p_1 in the first place, thus it supports p_1, while for the same reason p_4 supports p_3. When all the votes are collected, all nodes receive two votes on p_1 and two votes on p_3. Using a deterministic function the item is assigned to p_1. From this example, it looks like the solution to the consensus problem is deceptively simple. Actually, *it is* deceptively simple if all processes remain correct, in the same measure as it gets complex in the presence of failures. Consider the algorithm described above. Imagine that a machine crashes in the step of disseminating its own proposal. It may happen that some of the remaining machines receive the proposal and others do not. Since the sets of collected proposals are no longer the same, the decision is no longer deterministic. In the Fault Tolerance part of the book we will discuss these issues in detail.

2.9.3 Agreement on Membership

Very often, a set of processes cooperate in a tight fashion to achieve some common goal. For instance, processes can cooperate to distribute load, each picking a different request or even partitioning the work of a single request by all servers. Other examples include cooperative applications, such as teleconferencing, multiuser chat services, etc. Many of these groups of processes are not static. New processes can join and old process can leave at any time. A *membership service* is responsible for providing each member and possibly users, with information about who is participating in the computation and who is not. The list of active participants at a given time is called a *view*. In order to be useful, such service must provide consistent information to all members. In other words, all members of the group need to reach consensus about the current membership.

In absence of failures, consensus on the membership can be achieved using an algorithm similar to the one described in the previous section. Consider that every time there is a membership change, a new view is delivered to all participants. For sake of clarity, consider also that views are totally ordered, i.e, after delivering view V^i to all processes, the service delivers the view V^{i+1},

and so on. The membership service could work as follows: when a new process wants to join the group, it sends a message to all members of view V^i. Each members makes its own proposal for the membership of view V^{i+1}, according to the requests it has received. Different members can receive a different number of join requests and, therefore, can propose different views. Members and candidates collect all proposals from view V^i, pick one using some deterministic rule (for instance, the view with the largest number of members), and deliver the selected view as view V^{i+1}. This simple approach works in absence of failures but, naturally, a fault-tolerant membership service is much more useful (if a members crashes, a new view should exclude the failed member from the membership). This issue will be addressed the Fault Tolerance part of the book.

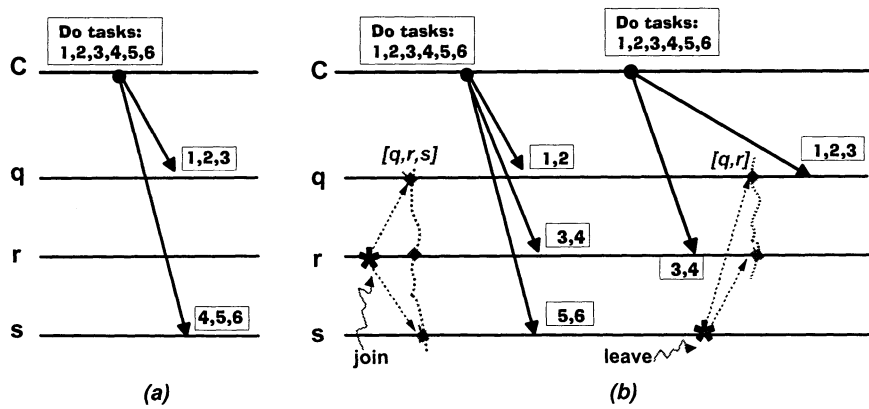

Figure 2.31. Ad-Hoc View Change

It should be noted that agreement on membership, by itself, may not be enough to make the design of group-based applications simple. It is often relevant to understand how membership information is ordered with regard to the message flow. Let us consider the case of load balancing through a group of participants, illustrated by Figure 2.31.

In this example a team of servers collect requests from clients (e.g., C), and divide the work associated with each request in a decentralized and uniform manner, based on membership information. If the workers have consistent information about the membership at all times, they can use a deterministic algorithm to distribute the load, without being required to explicitly send messages to each other. In order to do this the workers must simply be aware of their *number* and their respective *rank* in the group. For instance (*see* Figure 2.31a), if the client request includes tasks $1 \ldots 6$ and there are two workers available (q and s), each worker performs three of these tasks. After finishing, they consolidate the results. The team may be dynamic: whenever some worker enters or leaves it notifies all others of the fact, as shown on the left of Figure 2.31b, where after r joins, the team is $[q, r, s]$ (i.e., number= 3, rank(q)= 1;

rank(r)= 2; rank(s)= 3). The first job works well. Since there are 3 workers, q picks tasks 1 and 2, r picks tasks 3 and 4, and so forth. Now process s leaves the group and notifies the others, but the notification crosses the second job request, leading to an inconsistent perception of the team when dividing that job. The request from client C is delivered to q after the new view and thus q, aware that s is no longer in the group, picks half of the tasks (1,2,3). However, the request is delivered to r before the new view and thus, since r is not aware that s has left when it receives the request, it just picks a third of the tasks (3,4). The decisions of q and r are not consistent, and thus task 3 is performed twice, and tasks 5 and 6 are not executed.

What was wrong with the second scenario? Requests were received in different views by different participants. To prevent this problem, one needs to order view changes with regard to messages, so that any message is received by all in the *same* view.

2.9.4 View Synchrony

As we have seen in the previous section, there are cases where it is useful that membership information is ordered with regard to the message flow. Let us try to be a bit more precise about this concept (without delving into formal specifications).

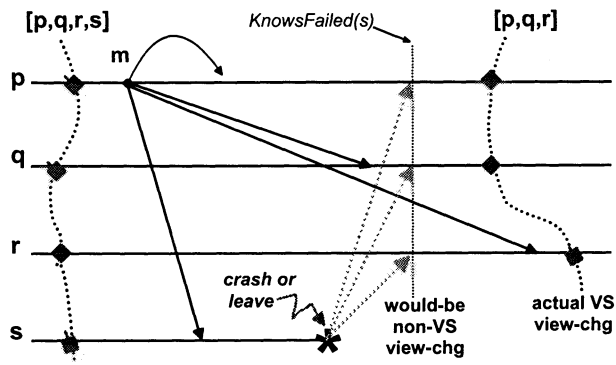

Figure 2.32. View-Synchronous View Change

As before, we assume that the membership service provides a *linear* sequence of views to all processes. Also, we will not consider the effects of faults. We say that a message m delivered to a process p after the delivery if view V^i and before the delivery of view V^{i+1} is *delivered in* view V^i. The view-synchrony ordering requirement, initially coined *virtual synchrony* by Birman and Joseph (Birman and Joseph, 1987), can be expressed as follows: if a message m is delivered to a process p *in* view V^i, then for all $q \in V^i$, m is also delivered to q *in* view V^i.

How to ensure that all processes deliver the same messages in the same view? One way to achieve this goal is, again, using the consensus as a building block. Consensus can be used in the following way: when installing a new view, say V^{i+1}, all processes in view V^i must reach agreement, not only on the membership of V^{i+1}, but also on the set of messages to be delivered in view V^i. A simple non fault-tolerant algorithm to implement view synchrony could work as follows. Assume that all processes are operating in view V^i. Messages are sent in multicast to all processes. When a process p receives a message m, it immediately delivers that message. When its time to install a new view, process p stops sending and delivering more messages. It then sends to every other process in the group the list of messages already delivered in that view (this is inefficient, but we are trying to make it simple). All processes collect the lists sent by every other process. As soon as all lists have been collected, every process executes the following steps: *i)* all messages in the collected lists are delivered; *ii)* a new view V^{i+1} is installed and; *iii)* the processes start accepting and delivering new messages (now delivered in the new view). Note that this algorithm artificially delays the delivery of the new view until all messages from the previous view have been delivered, solving the problem explained in the previous section, as illustrated in Figure 2.32. The fault tolerance aspects of this problem are also challenging, and will be discussed in the relevant part of the book.

2.9.5 *Atomic Broadcast*

In the example motivating the need for view synchrony, the way each request was processed depended upon the last view delivered. In some cases, the response to a given request may depend not only on the membership but also on previous requests. In these cases, messages need to be ordered not only with regard to views but also with regard to other messages in a total way (*see Ordering Algorithms* in Section 2.7).

An *atomic broadcast* protocol ensures that all messages are received by all members in exactly the same order. In other words, atomic broadcast combines reliable broadcast with total order. Practical protocols will rather provide atomic multicast, that is, to a *group* of participants. Originally, atomic broadcast was introduced in the context of fault-tolerant systems (and is still used mostly in that context), and owes its name to the notion of indivisibility with regard to faults, i.e., the broadcast is either delivered to all correct participants or to none.

Atomic broadcast can also be expressed as a consensus problem. All the participants must agree on: *i)* whether they delivered the message; *ii)* the order of that message with regard to other messages. To privilege the understanding of the reader as we did in the previous sections, we will provide a simple solution, even if with a deplorable performance. The algorithm follows the same lines of the previous algorithms, and also assumes the availability of a linear membership service. The algorithm works as follows. Messages are sent to all participants. Participants *do not deliver* these messages; instead they keep

them in a bag of unordered messages. They do this with every message until a view needs to be delivered (we have warned you, this *is not* efficient). When a new view is about to be installed, processes exchange their bags of unordered messages. All bags are mixed into a big set of messages to be delivered in the previous view, ordered using some deterministic rule, and delivered by that order to the application. Note that this simple algorithm does not deliver any messages if the view never changes. However, this can be fixed by running the consensus periodically, just as if a new view was about to be delivered. Actually, there is a class of total order algorithms that work exactly this way (Chandra and Toueg, 1996).

2.9.6 Replica Determinism

Informally, replication consists of maintaining identical copies of the same process or data in several locations. Replication is a fundamental technique to achieve fault tolerance: even if one of the replicas fails the others will be available for providing service. Replication can also be used to improve the performance of a system by placing replicas of a service or data close to their clients. A particular example of the use of replication for better performance are *caches*, local copies of a remote item to speedup data access.

Although the notion of maintaining exactly identical copies is intuitive, it turns out to be much more difficult to implement (and even define) than it looks at first sight. Actually, the only way to achieve fully identical behavior is to execute all replicas in *lock-step*, i.e., to ensure that replicas are synchronized at the instruction level. When a replica executes a given instruction, all replicas execute that instruction, and so on. In this way, the state of each replica can be compared at any point in time, and all replicas consume inputs and produce outputs at approximately the same real time instant. To ensure that replicas maintain the same state one must also ensure that all replicas receive exactly the same inputs and that their code responds in exactly the same way to the same input (programs that have this property are said to be deterministic). Of course, this level of consistency can only be achieved in the small scale, using hardware to keep the replicas in lock-step.

Given that the level of synchronization achieved by executing the replicas in lock-step is not scalable, one needs to use a more generic definition of replication that can be used in a broader range of systems. The notion of *replica determinism* states that two replicas, departing from the same initial state and subject to a same sequence of inputs should reach the same final state and produce the same sequence of outputs. The simpler way of achieving replica determinism is to use deterministic programs and to rely on an atomic broadcast protocol to disseminate the inputs (to ensure that all replicas receive exactly the same sequence of inputs). This is the basis for one of the most intuitive forms of replicated processing, called the *replicated state-machine* approach (*see State Machine* in Chapter 7).

Note that if the replicas are executed in different nodes, and inputs are exchanged using multicast messages, it is natural that some degree of de-

synchronization occurs. Due to network delays, omissions and retransmissions, the inputs can be delivered to different replicas at different moments. Even if the input is delivered at exactly the same instant, one replica may progress faster and produce the output sooner than others because of differences in load or hardware. The amount of de-synchronization allowed between two replicas depends on factors related with the timing constraints of the application.

It should also be noted that the combination of atomic broadcast and deterministic programs is not the only way to ensure replica determinism. Different replication strategies use different techniques to coordinate the replicas. For instance, one of the replicas may be elected to decide the order by which input messages should be processed. This and other techniques will be discussed in depth in the Fault Tolerance part of the book.

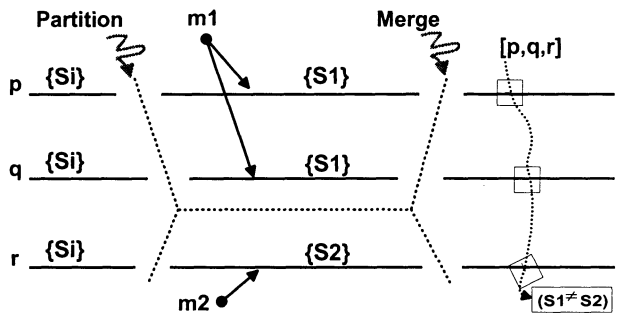

Figure 2.33. State Divergence with Partitioning

2.9.7 Primary Partition

In order to keep the replicas with a consistent state, one needs to ensure that communication is possible such that replicas can coordinate. Unfortunately full connectivity cannot be always ensured in complex networks. Sometimes, because of link failures or router crashes, the network is split into two or more *partitions*. When this happens, nodes in the same partition can communicate with each other but are isolated from nodes in other partitions.

Unfortunately, there is no way for a process within the system to distinguish network partitioning from the crash of one or more processes. When replicas of the same service are separated by network partitioning it may happen that the replicas in one partition assume that the replicas in the other partition have failed, continuing to operate in isolation (and vice-versa). Being unable to coordinate with one another, the state of replicas in different partitions is likely to *diverge*. If partitioning is *healed* later on, it may be impossible to *reconcile* the state of replicas in different partitions into a single consistent state.

Consider the example of Figure 2.33. All the three processes start with the same state Si. Network partitioning leaves p and q connected and r in another

partition. Message m_1 is processed by p and q whose state becomes $S1$. In the other partition, m_2 is processed by r changing its state to $S2$. When the two partitions merge, neither $S1$ or $S2$ are a prefix of the other, thus the replicas have diverged. In the general case, automatic reconciliation will not be possible and human intervention will have to be requested.

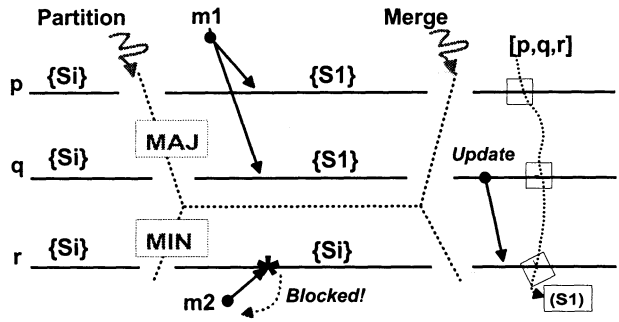

Figure 2.34. Primary Partition

Since network partitioning cannot be prevented in large-scale complex networks (e.g., Internet) a way to prevent divergence needs to be found. A simple technique consists of preventing two partitions from being active at the same time. This can be done by selecting a single partition, called the *primary partition*, to remain active and by blocking the activity in all other partitions. Of course, the criteria for selecting the primary partition must be such that each partition is able to locally evaluate them without communicating with the other partitions. One such criterion is to select the partition with the majority of nodes as the primary partition. By definition, at most one partition will have the majority of nodes, so no divergence will ever occur.

Consider the example of Figure 2.34. When the partitioning occurs all replicas are in state Si. Replicas p and q form the primary partition, thus they are allowed to continue processing messages and they move to state $S1$. However r is blocked so it does not process message m_2 and remains in state Si. Now the merger: Si is in the past (is a prefix) of whatever state the primary partition (p and q) reaches when the partition is healed ($S1$ in this case). In consequence, that state represents the current state of the replicated component and can be safely copied to the out-of-date component r during recovery. The down side of primary partition is that it is possible that the network is partitioned in such a way that no partition has a majority of nodes. Techniques to alleviate this problem will be discussed later (*see Replication Management in Partitionable Networks* in Chapter 7).

2.9.8 Weak Consistency

Maintaining strong consistency or, in other words, avoiding inconsistency, simplifies distributed concurrent programming. However, the reader has certainly

perceived that it often has non-negligible costs in performance and/or availability, such as preventing progress in one or more partitions, as seen in the last section. In some applications, the ability to make progress is more relevant than ensuring that things never diverge. This *consistency/availability* tradeoff becomes particularly relevant in environments where partitions are frequent or even enforced such as mobile computing, where a user may disconnect her machine from the network but still want to continue working.

It may also happen that the probability of inconsistent updates is very small, even if progress is allowed in several partitions. Consider a mobile computing environment: most of the files updated by users are personal files and thus, it is very unlikely that a file copy is updated in the office while the owner of that file is updating it on her portable computer. In this case, it is clearly advantageous to allow progress to be done in any partition. On the other hand, if the same file happens to be updated concurrently by more than one user, it will be very hard to merge both updates in a new file in a completely automatic manner. Manual user intervention is then required. For instance, the system may automatically preserve both copies of the file and prompt the user to integrate them by hand.

It is worth mentioning that even if partitioning never occurs, ensuring strong replica consistency is inherently expensive, since all updates to replicated data need to be totally ordered. For efficiency reasons, one may want to support weaker consistency models that do not require all updates to be ordered in a total manner. We will discuss these models when addressing the issue of implementing distributed shared memory systems (*see* Section 3.8 in Chapter 3).

2.10 CONCURRENCY

We have already discussed the need for mutual exclusion when different threads of control access shared data structures. Mutual exclusion is a particular case of *concurrency control*, the body of mechanisms that ensure the consistency of data despite concurrent access by several threads.

2.10.1 Atomic and Sequential Consistency

Before discussing the notion of consistency applied to sequence of operations, let us approach the notion of consistency applied to individual pieces of data when programs read and write from memory positions. In this context, the intuitive notion of consistency is immediately derived from the behavior of a single physical memory, what is called *atomic consistency.* Consider, for sake of clarity, that each individual access to memory is a single atomic operation. Atomic consistency can be described as follows: writes and reads are ordered according to the physical (real time) order. Naturally, writes issued at time t are immediately observable by all reads issued at time $t' > t$.

However, for efficiency reasons, in current architectures we do not have just a single central memory. Several levels of cache exist between the processor core and main memory. If several processors are available, the same memory position can be replicated in more than one cache, which raises the problem of

cache coherence. Of course, the most tempting approach is to hide the existence of caches from the users and ensure that the resulting system behaves exactly as a system with no caches at all, i.e., ensuring atomic consistency. However, this may be very hard and inefficient to enforce, since it requires the serialization of all memory operations.

A slightly weaker model called *sequential consistency* allows processes to read data from their caches as long the resulting sequence of memory operations is equivalent to some serial execution of the same memory operations. Note that this does not force writes and reads to obey physical order. For instance, a process is allowed to read some outdated value in its cache even if a write has already been issued by other processes, as long as that read can be serialized in the past of the memory update. In this model, the behavior is no longer that of a single, non-replicated, memory. However, if the processes communicate exclusively through memory operations they cannot detect the difference to an atomic execution.

The advantage of a sequentially consistent memory is that it can be implemented without serializing all memory operations. In fact, only write operations need to be serialized and reads must only be ordered with regard to writes. The model also allows for non-conflicting accesses to different data structures to proceed in parallel.

2.10.2 Serializability

A good understanding of the memory consistency model is paramount to building correct programs. However, it is not enough. As we have seen when discussing the need for mutual exclusion, one is often concerned with the execution of a *sequence* of memory operations in a consistent way. Mutual exclusion is one of the solutions for the problem. By locking all the variables the process wants to access during the *critical region*, one is sure that no other correctly coded process (i.e., that checks the lock at the appropriate locations) is allowed to update these variables.

Mutual exclusion works by enforcing a serial order on the execution of a sequence of operations. A process obtains the mutual exclusion lock, accesses the data and releases the lock. Then another process is allowed to obtain the lock on the data, and so on. Mutual exclusion works well for small sequences of operations, where the programmer can easily associate locks with shared data structures. For programming in the large, mutual exclusion becomes cumbersome and inefficient.

Just consider a large database with thousands of data items and a large number of complex programs, written by different people, that access different pieces of the database. It is clearly very difficult, if not impossible, to identify what pieces constitute critical sections. The conservative approach of accessing the database in exclusive mode would be unacceptable either, for it would impose an enormous latency in database access. In most cases, locking the whole database is an overkill since it is likely that many of the database accesses are to be performed on unrelated items.

```
DATABASE
    int A, B, C, D;

transaction mvAtoB (int x) is
    A := A-x;
    B := B+x;
end,

transaction mvCtoD (int x) is
    C := C-x;
    D := D+x;
end;

transaction sumAll is
    x := 0;
    x := x + A;
    x := x + B;
    x := x + C;
    x := x + D;
    print x;
end;
```

Figure 2.35. Simple Transactions

What is needed is a mechanism to ensure that these sequences of operations, that we will call *transactions*, execute in such a way that their outcome is equivalent to the outcome of their execution in *some* serial order without forcing the transactions to actually execute in a serial order. In particular, if two transactions access unrelated items, they should be allowed to progress in parallel, since any interleaving would result in an outcome equivalent to a serial one. This correctness criteria is know as *serializability*.

2.10.3 Concurrency Control

Consider a database composed of four integers, A, B, C, and D, which are accessed by the transactions depicted in Figure 2.35. Given that the transactions updating the database only move quantities from one variable to another, the total sum of all variables should be a constant in the system. However, if the transactions are executed by concurrent threads, one needs to add synchronization primitives to ensure that correct results are obtained (*see Basics of Synchronization* in Section 2.8).

As we have noted in the previous section, mutual exclusion is the most straightforward, but not necessarily the most efficient, way of implementing concurrency control. One way to obtain mutual exclusion would be to associate a global mutual exclusion semaphore to the complete database. In this case, each transaction would perform a *wait* operation on the semaphore before accessing any variable and a *signal* operation after the last access, as illustrated in Figure 2.36. Although the global mutex serializes all accesses (thus, enforcing a serializable execution) it introduces much more synchronization that strictly needed. For instance, *mvAtoB* and *mvCtoD* should be allowed to execute in parallel since they access different data items.

One way to increase the degree of concurrency in the system while still ensuring that the resulting execution of concurrent transactions is serializable, is to associate a mutual exclusion semaphore with each database variable and

```
DATABASE
    int A, B, C, D;                          semaphore db-lock;

transaction mvAtoB (int x) is                transaction sumAll begin
    wait (db-lock);                              x := 0;
    A := A-x;                                    wait (db-lock);
    B := B+x;                                    x := x + A;
    signal (db-lock);                            x := x + B;
end;                                             x := x + C;
                                                 x := x + D;
transaction mvCtoD (int x) begin                 signal (db-lock);
    wait (db-lock);                              print x;
    C := C-x;                                end;
    D := D+x;
    signal (db-lock);
end;
```

Figure 2.36. Global Concurrency Control

to let each transaction lock just the variable it accesses. The resulting code is illustrated in Figure 2.37. Note that while there is an active update the transaction *sumAll* is blocked, since it needs to obtain a lock on every variable. On the other hand, *mvAtoB* and *mvCtoD* can proceed in parallel since they use different semaphores. The concurrency control strategy illustrated in Figure 2.37 represents the simplest form of *locking*, a technique that is widely used in database systems, with several optimizations. In the Fault Tolerance Part of the book we will discuss different ways to optimize the locking scheme introduced here.

2.10.4 One-copy Serializability

We have previously discussed the notion of memory consistency in systems where several copies of the same memory item exist. We recall that the goal was then to make memory look, as much as possible, like a single centralized memory. The question now is to define an equivalent criterion for sequences of operations, or transactions, which access data that is replicated. The criterion is *one-copy equivalence*, i.e., the set of replicas should behave like a single copy. Combining one copy equivalence with the serializability criteria described above, one gets *one-copy serializability.*

Note that if all copies are available and mutually reachable, they can synchronize with each other to ensure that the consistency criteria is not violated. However, if networks partitions occur, and different copies become located in non-connected partitions, synchronization becomes impossible. To ensure one-copy serializability it is necessary to guarantee that updates can occur at most in one partition, but never in concurrent partitions. Techniques to achieve one

```
DATABASE
    int A, B, C, D;

transaction mvAtoB (int x) is
    wait (lock-A); wait (lock-B);
    A := A-x;
    B := B+x;
    signal (lock-B); signal (lock-A);
end;

transaction mvCtoD (int x) is
    wait (lock-C); wait (lock-D);
    C := C-x;
    D := D+x;
    signal (lock-D); signal (lock-C);
end;
```

```
semaphore lock-A, lock-B;
semaphore lock-C, lock-D;

transaction sumAll is
    x := 0;
    wait (lock-A); wait (lock-B);
    wait (lock-C); wait (lock-D);
    x := x + A;
    x := x + B;
    x := x + C;
    x := x + D;
    signal (lock-D); signal (lock-C);
    signal (lock-B); signal (lock-A);
    print x;
end;
```

Figure 2.37. A Lock Associated with each Variable

copy equivalence will be discussed in the Fault Tolerance part of this book (*see Transactions and Replicated Data* in Chapter 7).

2.11 ATOMICITY

Intuitively, an atomic operation is an indivisible operation. In other words, an atomic operation has no intermediate visible steps. It is very frequent to find sequences of operations in programs that one would like to make atomic in the sense described above. Consider for instance that you are booking a sequence of plane tickets to go from Austin Texas (USA) to Porto (Portugal). You will probably need to by a ticket from Austin to Houston, from there to New York, then to Lisboa and finally to Porto. You need all these tickets together and you probably do not want just part of them, so you would like to make the sequence of bookings an atomic sequence.

2.11.1 Transactional Atomicity

Atomic transactions are a paradigm that allows arbitrary sequences of operations on data items to be transformed into atomic operations. In terms of programming interface, the desired sequence of operations needs to be bounded by a pair of *begin transaction* and *end transaction* directives. A transaction that successfully terminates is said to *commit*. A transaction that, for some reason, cannot terminate is said to *abort*.

Two main components are needed to implement atomic transactions. First one needs a recovery mechanism, i.e., some mechanism to ensure that in the case a transaction aborts, the system state is left as it was before that transaction was initiated. Second, we need to ensure that intermediate results from a

transaction are not made visible unless the transaction commits. This is usually achieved by the concurrency control mechanisms.

To implement recovery, transactional systems need some form of *persistent store*, i.e., some (physical or logical) device where data can be safely stored without being lost. Assuming that such basic component is available (in its simpler form, it is merely a disk), recovery can be implemented by saving, in the persistent store, a copy of the original value of all data items updated by the transaction. These values are kept in a data structure called the transaction *log*. If the transaction aborts, the original state can be recovered from the log.

Transactional atomicity also applies with regard to failures. The transaction outcome is also recorded in the transaction log: before the transaction is confirmed to the user, an entry stating that the transaction was committed must be saved in the log. Additionally, a transaction cannot be committed unless all updates have been saved in the persistent store. If the node crashes, the log is parsed upon recovery. If a commit record is in the log, the transaction results are left unchanged. If no such record exists, or if an abort record is found, the original state is recovered from the log. The reader should be aware that alternative forms of logging exist, but we will not discuss them in detail at this point.

2.11.2 Distributed Atomic Commitment

In distributed systems, a transaction may access data items on different nodes. Such a transaction is called *distributed transaction*. Like a centralized transaction, a distributed transaction should also exhibit the atomicity property. This means that all nodes involved in the transaction must agree if the transaction should be aborted or committed. A protocol that ensures this sort of agreement is a *distributed atomic commitment* protocol.

In absence of failures, distributed atomic commitment can be solved as illustrated by Figure 2.38. One of the participants in the transaction is designated as the coordinator of the protocol. The coordinator sends a PREPARE message to all other participants, implicitly asking all processes if they are ready to commit the transaction. If a node detects that some error prevents the transaction from being committed, it replies NOTOK to the coordinator. If the node is ready to commit the transaction (this usually means that all the updates are stored in non-volatile memory and will not be lost) it replies OK to the coordinator. This constitutes the first phase of the protocol. In the second phase, if the coordinator receives OK from all the participants it sends a COMMIT to all participants. If it receives at least one NOTOK, it sends an ABORT. In any case, it coordinator awaits an *acknowledgment*. To ensure a fast dissemination of the transaction's outcome, the coordinator retransmits the decision if some acknowledgements are missing. Because of its structure, this protocol is known as a *Two-Phase Atomic Commitment* protocol. The disadvantage of this protocol is that it may block if the coordinator fails in some of the protocol steps, because the remaining processes may be unable to assess which decision (commit or abort) if any was issued by the coordinator. In such cases, the system

must halt until the coordinator recovers. Protocols that exhibit higher availability will be discussed in the Fault Tolerance part of the book (*see Atomic Commitment and the Window of Vulnerability* in Chapter 7).

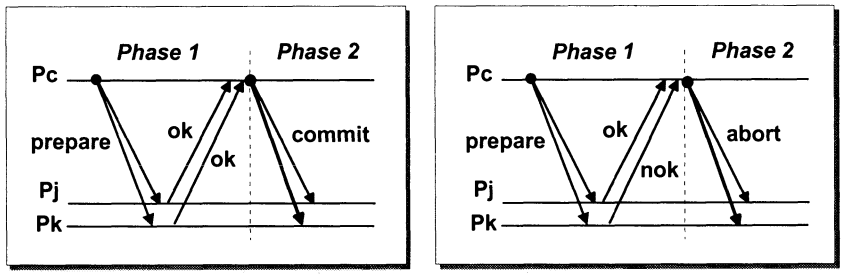

Figure 2.38. Two-phase Commit Protocol: (a) Commit; (b) Abort

2.12 SUMMARY AND FURTHER READING

In this chapter we have presented the main distributed system paradigms. Starting from the problem of naming and addressing we have visited the main interaction styles, including message passing, remote operations and group communication. Then we have discussed the importance of time, clocks and the issue of system synchrony. We discussed the issue of ordering distributed events and from there departed to discuss the main distributed coordination paradigms.

For further reading, notes about GPS can be found in (Dana, 1996). Clock synchronization deserves a detailed treatment in Section 12.8 of the Real-Time Part. There are two related facets of group communication services: those designed towards dependability, such as (Birman and Joseph, 1987; Cristian, 1990; Moser et al., 1994), and those designed to disseminate multimedia information in the Internet, such as DVMRP (Deering, 1989), MOSPF (Moy, 1994), PIM (Deering et al., 1996) and RMTP (Lin and Paul, 1996) among others. For a good book on multicasting on the Internet see (Paul, 1998).

For different approaches to enforce causal order see (Birman and Joseph, 1987; Peterson et al., 1989; Ladin et al., 1992; Raynal et al., 1991; Rodrigues and Veríssimo, 1995; Ezhilchelvan et al., 1995). The problem of anomalous behavior was first identified in general by (Lamport, 1978b), and later pointed out for real-time control systems (Veríssimo et al., 1991). It was reported in (Cheriton and Skeen, 1993) as a limitation of causal ordering, which it is not: in fact it is a limitation of 'logical implementations of causal ordering'. There are also many published works on total order. The approaches of (Peterson et al., 1989; Amir et al., 1993a; Melliar-Smith et al., 1990; Dolev et al., 1993) are examples of symmetric protocols. Examples of sequencer based protocols are (Chang and Maxemchuck, 1984; Kaashoek and Tanenbaum, 1991; Birman et al., 1991b; Amir et al., 1993b). A hybrid approach is given in (Rodrigues et al., 1996). Protocols based on consensus that operate in asynchronous systems augmented

with failure detectors can be found in (Guerraoui and Schiper, 1997; Rodrigues et al., 1998a; Fritzke Jr. et al., 1998; Rodrigues and Raynal, 2000). For a deeper discussion on temporal order see (Veríssimo, 1996; Veríssimo and Raynal, 2000).

There are many interesting books on concurrent programming and synchronization constructs. The reader will find further information about readers/writers, rendez-vous, sequencers, event counters and so forth. The book of Ben-Ari (Ben-Ari, 1990) includes several interesting examples of concurrent and distributed programming models. A book that also covers the real-time aspects of concurrent programming and synchronization is (Burns and Wellings, 1996). The more recent work of (Lea, 1997) presents interesting patterns for concurrent programming in Java.

Ricart and Agrawala (Ricart and Agrawala, 1981) proposed an optimization of the mutual exclusion algorithm of Lamport that reduces message complexity. A treatment of consistent global states is presented in (Babaoğlu and Marzullo, 1993) and a very interesting survey on checkpoint protocols can be found in (Elnozahy et al., 1999). The problem of distributed consensus in different system models and under different failure assumptions has been discussed in several papers. Some interesting references are (Dolev et al., 1983; Fischer et al., 1985; Dwork et al., 1988; Chandra and Toueg, 1996; Aguilera et al., 1998; Guerraoui et al., 2000).

3 MODELS OF DISTRIBUTED COMPUTING

This chapter discusses the main distributed systems models. As an introduction, it sets the context by addressing the main facets of the problem. Frameworks clarify what can be done given different assumptions on failures and synchronism, explaining that we can structure distributing computing along different vectors serving different needs. Strategies help the architect reason about the available ways to go in order to serve her requirements and objectives. Then two more fundamental issues are addressed before delving into the system models: explaining the main differences between the synchronous and asynchronous formal frameworks for distributed computing; and presenting the primitive classes of distributed activities and their overall scheme of operation, for understanding the purposes of distribution. Finally, the chapter presents known models such as: client-server with RPC, group-oriented, distributed shared memory, message buses.

3.1 DISTRIBUTED SYSTEMS FRAMEWORKS

In this section we analyze the several frameworks on which an architect can base the construction of a distributed system. Some are complementary, others orthogonal. Some are functional, others are formal. The former are concerned with specification of infrastructure, architecture and interfacing, in order to fulfill a certain functionality. The latter address the formalization of hypothesis and properties, some of which non-functional, and their implementation

P. Verissimo et al. *Distributed Systems for System Architects*

and validation through the adequate paradigms and algorithmics. We intend this section to give the reader a picture of the several vectors along which the architectural work on distributed systems is developed, namely: infrastructure; semantics; organization of distributed activities and services; distribution of information repositories; access to distributed services. This material introduces the basic classes of distributed activities and the main models of distributed computing, to be addressed later in the chapter, such as: asynchronous; synchronous; coordination; sharing; replication; client-server; groups; distributed shared memory; message buses.

3.1.1 Infrastructure

Infrastructure issues are concerned with the hardware, networking and operating system support. They constitute a basic framework through which the architect provides the system with enabling functionality for the higher level activity.

Distributed systems today, looked from afar, are best represented by a large number of organization sites, with mutual access and access to services and resources inside the organization (*intranet*), and having access to the several interconnected public networks worldwide (*internet*). Besides, it is today common that these organizations have *facilities*, which are interconnected in a closely-coupled way across different cities or even countries through highly efficient connections called tunnels, and allow remote access from the internet both to collaborators and outsiders (e.g., e-commerce). This is implemented by the so-called *extranets*.

Users have been presented with ever-increasing power and functionality in their local machines. RISC MIPs in "normal" workstations are today (2000) rating beyond the many hundreds. Users have already experienced successful high-performance distributed applications on a LAN scope. They expect to maintain and improve this status-quo over the "global network", that is, to cooperate with geographically separated participants and to access remotely placed services, as if they were inside their intra-organization network.

In order to build architectures satisfying these needs, one cannot ignore the infrastructure on which the system will rely. What we discuss below are basic building blocks that should be given consideration in any open distributed system architecture.

Tightly versus Loosely Coupled Distributed Systems A closer look into the inside of organizations, shows machines normally plugged to local area networks. The technological potential of LANs and similar technologies is considerable: high bandwidth, low and known error rates, multicast, time boundedness, reliability vis-a-vis partitions. Users are concentrated on a geographically small area. Applications based on tightly-coupled systems such as backplane multiprocessors or closely-coupled networked multicomputers are easily deployed and have attractive performance.

However, most applications require distribution over wider areas, meaning connectivity via internetworks such as the Internet, which exhibit weaker attributes: low bandwidth, higher and unpredictable error rates, point-to-point, asynchrony, susceptibility to partitioning.

It is highly desirable to integrate these two styles of computing seamlessly. As a first step towards integration of tightly and loosely coupled computing, there are two architectural principles to follow:

- *use of common protocols* from the network level (e.g., TCP/IP) all the way up to the applications (e.g., CORBA), both in the intranet and extranet, and through the Internet;
- *extension of this principle* to the internal structuring of cluster multicomputer modules.

However, this integration requires a few additional architectural devices, both at the network and operating system support levels.

Large-Scale Infrastructures Certain infrastructure issues become more important in the measure that distributed systems grow in span, complexity and number of nodes. The implications of scale on the structure of distributed systems can be read under several facets: the computation participants; the communication system.

Scale affects computation participants in several ways. The most obvious aspect of scale concerns the number of participants in the computation. The number of entities simultaneously involved in a computation varies, according to the type of interaction concerned. Additionally, that number is often significantly smaller than the number of nodes in the network. For the sake of simplifying our forthcoming analysis, we propose to consider a coarse-grain scale metric, of three "levels": very-large-scale— order of millions and up; large-scale— order of the thousand up to the million; small-scale— order of hundreds down.

The communication system characteristics have a fundamental impact on the scale of computations, since they make it extremely difficult, and sometimes even impossible, to reproduce in large-scale the operating conditions that are otherwise found in small-scale systems. In consequence, there is a need for structuring applications in ways that allow reasonably efficient operation. Hierarchical organization and clustering according to the topology of the infrastructure, are paradigms addressing this particular issue. In fact, clustering seems one of the most promising structuring techniques to cope with large scale, providing the means to implement effective divide-and-conquer strategies.

A large-scale network such as the Internet forms what we might call a *global network*, for the purpose of a large-scale computing platform. A number of nodes in the order of 10^7, and growing, puts it in the very-large-scale level, with a number of *structural* characteristics dictated by its scale and limited technology: sparse connectivity; limited diffusion capabilities; weak reliability and timeliness; globe-wide distances; public-domain or standard protocols.

In face of these characteristics, we can extract a set of functional communication properties, the most important of which are listed below, deriving both from sheer scale and from technology shortcomings: large communication delay variance; asynchronism; partitioning (e.g. set $\mathcal{M}$ of nodes reach each other and set $\mathcal{N}$ of nodes reach each other, but do not reach the other set); non-transitivity (e.g. A reaches B, B reaches C, but A cannot reach C); non-symmetry (e.g. A reaches B, but B cannot reach A).

On the other hand, inside what we might call *local networks*, a moderate number of nodes puts local networks in the small- to large-scale level. The infrastructure takes significantly different characteristics that should not be ignored: availability of LAN or MAN technology (including the foreseen role of ATM); dense connectivity (normally broadcast-level); good reliability and timeliness; private operation, enterprise-oriented. Such significant differences should not be ignored by a large-scale architecture.

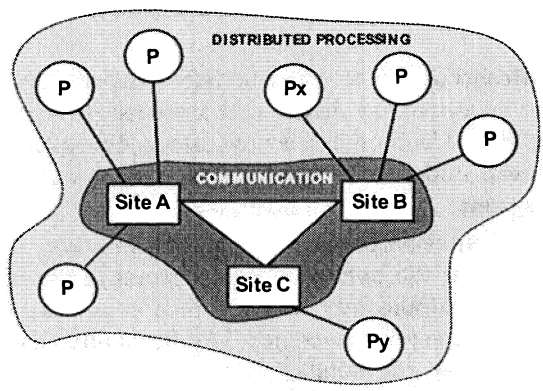

Figure 3.1. Decoupling Communication from Processing with Site-Participant Clustering

Site-Participant Clustering Distributed processing involves interactions among entities in different hosts (e.g., processes, tasks, etc.). We call them generically **participants**, denoted P in Figure 3.1. However, it is desirable to separate these functions from communication functions, highly specialized. The figure suggests the organization of the host into a **site** part, which connects to the network and takes care of all inter-host operations, i.e., communication, and a participant part, which takes care of all distributed activities and relies on the services provided by the site-part modules. Participants can be senders or recipients of information, or both, and interact via the respective site part, which handles all communication aspects on behalf of the former. This construct also introduces a first level of *clustering*, the site as a cluster of participants, an important architectural construct for two reasons:

- it allows protocols to take advantage of a multiplying factor between the number of sites and the (sometimes large) number of participants that are active in communication and distributed processing;

- it lets 'sites' concentrate on communication and frees 'participants' to concentrate on distributed processing activities (algorithms, applications, etc.)

This distinction between sites and participants is normally realized by a *communication subsystem* or site-level *communication server* approach to structuring the operating system's support for the machine's networking.

WANs of LANs Current large-scale computing infrastructures retain a clear duality, which is materialized by several aspects, from administration to technology, in what appears to be a logical 2-tier infrastructure, a WAN-of-LANs structure, as depicted in Figure 3.2: pools of sites with privately managed high connectivity links, such as LANs or MANs or ATM fabrics, which we call generically **local networks**, interconnected in the upper tier by a publicly managed point-to-point **global network** (e.g., the Internet). The global network is public and runs standard protocols; each local network is run by a single, private, entity (e.g., the set of LANs of a university campus, MAN of a large industrial complex, ATM structure of a regional company department), and can thus *run specific protocols* alongside with or in complement to standard ones (i.e., TCP/IP).

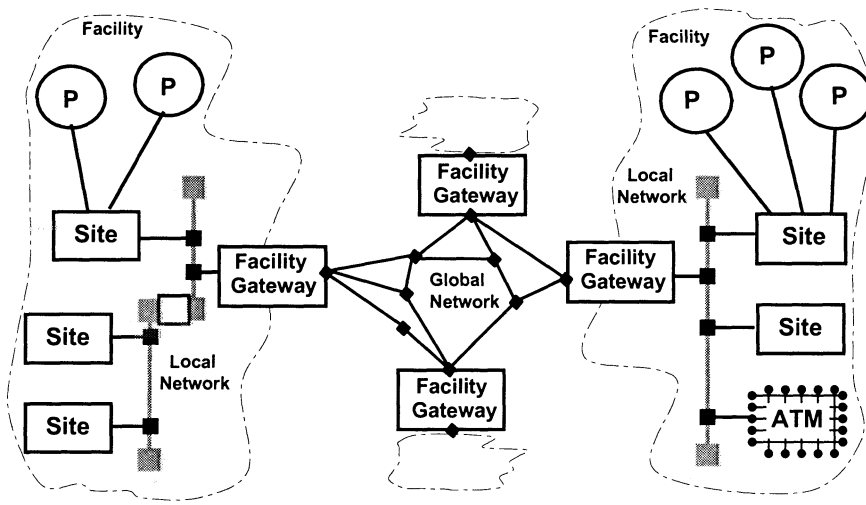

Figure 3.2. 2-tier WAN-of-LANs

This second level of clustering, of sites that coexist in the same local network, can simplify inter-network addressing, communication and administration. These sites are hidden behind a single logical entry-point, a **facility gateway**, which represents the local network members for the global network, performs routing, firewalling, etc. Ironically, it was because of security that we started observing ad hoc implementations of this architectural principle: the concept of closed intranet; firewall routers and gateways; protocol proxying at the facility gateway; network address translation (*see* Chapter 18).

3.1.2 Semantics

The growth of networked and distributed systems in several application domains has been explosive in the past few years. This has changed the way we reason about distributed systems in many ways. One issue of definitive importance is the following: *Which model has the most appropriate semantics for distributed applications?*

One important aspect is the *time-related semantics*, addressed by *timeliness* specifications, as we studied in Section 1.4. A traditional trend when large-scale, unpredictable and unreliable infrastructures are at stake (e.g. Internet) has been to use the so-called *asynchronous* models. Remember that we have already addressed 'synchrony' in Section 2.6. As a systems framework, 'asynchronous', in order to be simple at this point, means that there are not bounds on essential timing variables, such as processing speed or communication delay. This model, that we address in Section 3.3, ignores timeliness, and as such it has served well a large number of applications where uncertainty about the provision of service was tolerated.

However, a large part of the services we see emerging has interactivity or mission-criticality requirements, that is, there is some expectancy that: the service is indeed provided; it is provided with a reasonably narrow delay variance; if it is not provided or it is provided too late, some loss will arise for the user. We see thus that these requirements arise from a mixture of real-time and dependability constraints on one hand (e.g. air traffic control, telecommunications intelligent network architectures), and user-dictated quality-of-service requirements on the other (e.g. network transaction servers, multimedia rendering, synchronized groupware). This behavior requires the fulfillment of *timeliness* specifications, which in essence call for *synchronous* system models, which we study in Section 3.4. Under the 'synchronous' framework there are known bounds for timeliness variables. Some mechanisms used for securing synchronous behavior were studied in Section 2.6.

Also of great importance is *failure semantics*. We are going to study in depth how systems fail and what can be done about it in Part II, but we need not be building fault-tolerant systems to give enough concern to failures. We have already addressed the behavior of basic protocols in the presence of failures in Chapter 2, such as the remote operations and group communication protocols. The semantics of failure is also related to the synchrony of the system, as we are going to discuss in Sections 3.3 and 3.4.

The *semantics of system support* is important. The underlying layers of the infrastructure can supply increasingly stronger abstractions. The stronger they are, the more complex that system support must be, and this is not always desired. On the other hand, the weaker the support is, the more complex applications must be, and it is not always desired to put this burden on the application programmers' hands. Let us give an example based on *consistency*, a very important paradigm in distributed systems, which is at the root of many a distributed application. Consider an application that must guarantee consistency of actions performed in several sites, as well as consistency of replicated

data held in those sites. This application may be built on top of the raw multicast networking facilities, such as multicast-IP sockets and ancillary protocols of the same level of abstraction. In terms of consistency, this goes as much as ensuring a *best-effort* multicast message delivery to the group of sites involved in the application. However, in Chapter 2 we studied paradigms that give better consistency guarantees. Namely, *view synchrony* (*see* Section 2.9) specifies reliable and consistent message delivery to a group of sites in the presence of faults and site failures. The view-synchrony layer would be built on top of the best-effort layer, and our application could be simpler if it did not have to cope with the problems solved by view synchrony. Still, our application involves replication, and it would be desirable to guarantee replica consistency despite failures and network partitioning. The *primary partition* paradigm could be built on top of view synchrony, serving as a basic support on top of which replica determinism could be enforced (*see* also Section 2.9).

3.1.3 Organization of Distributed Activities and Services

The way distributed processing activities and services are organized deserves great attention. This framework addresses the core structure of the system (protocol suites, services and servers) to be accessed by the users. It is concerned with the understanding of a few basic informal classes of distributed activities, such as coordination, sharing and replication, and of the methods to compound them. *Coordination* is essential to any decentralized and/or cooperative activity, such as those performed by parallel processes, concurrent engineering (e.g., CAD) tool managers, fragmented database managers, or distributed transaction managers. *Sharing* classifies the generic activities that imply a set of participants competing over a resource. Examples of sharing are concurrent accesses of distributed participants to a database repository, to a file system, to a critical region of server code, or to a resource spooler. *Replication* is concerned with performing the same sequence of actions or maintaining a set of replicated data items. Replicated processing or management of replicated files or databases prefigure this class of activity.

We discuss the rationale and the main issues related with each class of distributed activity in Section 3.5. Actual distributed processing models materialize combinations of these primitive classes. We are going to address several of them throughout the rest of the chapter, such as client-server with RPC, groups, distributed shared memory and message buses. *Client-server with RPC* materializes the classic centralized service, whose server hosts all processing elements that clients may invoke. Concurrency among the competing clients only exists if allowed by the server, e.g., by multithreading. *Group-oriented* processing is based on group communication, and allows highly decentralized and parallel operation. Its open-loop nature easily supports the coordination of decentralized or replicated activities. The *distributed shared memory* (DSM) model can be highly concurrent and parallel, depending on the DSM coherence model. Coordination is done indirectly, through the distributed memory shared by the clients. *Message buses* support event-based operation, such as publisher-

subscriber or information-push. They have a producer-consumer flavor, where publishing is managed through shared write accesses to the message bus. Coordination and replication may may have relevance if the bus is replicated and subscriber dissemination (push) distributed by the bus replicas. Certain activities are organized as centralized services, such as file systems or databases, whereas others are organized in decentralized ways, whereby participants communicate directly with each other, such as voice over IP or group meeting support packages.

3.1.4 Distribution of Information Repositories

Complementary to organizing activities is organizing the way information is stored, retrieved and disseminated, in a distributed way. Information *storage* may be performed in a distributed form in essentially two ways: *fragmented* and *replicated*. These ways can obviously be combined to serve the architect's strategy, for example, fragmenting for locality of accesses to a database, while replicating the fragments to increase availability. The distribution aspects of *retrieval* are normally equated on a client-server basis, and make use of distributed *caching* to decrease latency. *Dissemination* can be seen as a form of automatic retrieval that implies a contract or expression of interest in a matter, or *subscription*. It relies on reliable *multipoint delivery* protocols for efficient dissemination to communities of subscribers.

This framework is intimately related to the previous one, since certain services imply both processing and storage. However, it is good systems practice to try and separate these systems issues whenever possible, and distributed systems, of all systems, make this separation relatively easy, besides desirable. Take the example of a database: one can separate the data managers from the transaction managers, and apply different policies to them. Another example: a transactional file system deployed through web servers has different information storage hierarchies, from the file system, through the web server and proxy caches, to the client caches. There is benefit in taking a data-oriented viewpoint at this problem. We will be addressing distributed file systems and web-based systems in Section 4.2. We also study message-bus models in Section 3.9, a model oriented to information dissemination.

3.1.5 Access to Distributed Services

Last but not least, is the way users access services in distributed systems. This framework discusses how activities, services and information, are made available to the users. Several distributed applications are non-interactive, such as a batch scientific calculation job on a pool of distributed parallel processes. However, most of the access to applications of distributed systems today is *interactive*. Moreover, if the advantages of distribution are in fact used, a considerable part of that interactive access is *remote*, that is, it is not made at a console physically connected to the site providing the service. This creates

problems to be solved, such as the need for responsiveness at a human pace; the need for reliability; the need for security.

The architect has available several mechanisms. Some are compounded in the computing models, such as the RPC client-server, where the client accesses the central server, but has a certain local processing autonomy integrated in the computing model. Client-server access has several facets, from raw RPC to HTTP requests, or group-based or object-based client-server interactions. Other access mechanisms are generic, such as the serial line virtual terminal, the simplest way to access a remote computer, e.g., over a leased or dial-up line. Remote session protocols allow a user to establish an interactive session over the network and work off a protocol such as IP (or PPP for serial lines). We are going to discuss remote access a propos the client-server and group-oriented models presented in Sections 3.6 and 3.7 in this Chapter, and we discuss web access technologies in Section 4.5.

3.2 STRATEGIES FOR DISTRIBUTED SYSTEMS

The adequate strategy for the design of a distributed system depends, as any strategy, on subjective factors, that is, what the architect wants to do in face of the requirements given to her, and on objective factors, that is, what she can do in face of constraints of the environment, of cost, etc. The latter present the architect with a number of tradeoffs. The former depend on the imagination of the architect. We discuss below the strategies we feel most important: distribution for information and resource sharing; distribution for availability and performance; distribution for modularity; distribution for decentralization; distribution for security. After equating her strategy, the architect will develop the system along the frameworks for the design of distributed systems that we have just presented.

3.2.1 Fundamental Tradeoffs

To begin with, there are a few fundamental tradeoffs to be made, depending on the particular application to be deployed or the problem to be solved:

- centralized versus decentralized control
- sequential versus concurrent distribution
- visible versus invisible distribution
- function versus data shipping
- service versus server
- scale versus performance
- openess versus determinacy
- synchrony versus asynchrony

Centralized control of applications renders them easier to build and manage. However, certain problems of a decentralized and distributed nature are best addressed through decentralized applications. The best tradeoff should

be devised for each case, since there is a spectrum of intermediate architectures available, for example closely coupled, or transparently distributed but centrally controlled.

The concurrency allowed by each model of distributed processing represents another tradeoff that has important implications. Let us motivate the issue with a simple and informal example of a system with N interacting participants. We consider a distributed model to be highly *concurrent* if the ratio of *activity_periods/elapsed_time* of each participant is high. This is the case of the DSM or the group-oriented models, which by virtue of the model allow to run several activities simultaneously in several places in a concurrent way. On the other hand, if that ratio is low, we say we have essentially *sequential* distribution. For example, in remote operation based models such as RPC, upon a remote execution request from a client the thread of control is passed to the server, and so on if calls are recursive. That is, by virtue of the model, control goes sequentially from one place to another. We say "by virtue of the model" because: even with concurrent models we may have highly sequential applications; but sequential models do not allow concurrency even when desired. The tradeoff lies in the fact that sequential distribution models are simpler and more intuitive to program with.

Another viewpoint at distribution is whether it is visible, or is concealed by some artifact of the model. Invisible or *transparent* distribution is achieved by models that hide distribution, sometimes from the programming model, such as the RPC model, or even from the architecture, such as the DSM model, where processes synchronize and communicate through a shared memory, as in a single-box mono- or multiprocessor, or object models inspired by the ODP model, such as CORBA, that we discuss in Section 4.4. The distributed system becomes a huge virtual machine. *Visible* distribution is carried out by models that preserve some visibility of message passing, such as the group-oriented and the message-bus models. The tradeoff relates to whether distribution is also part of the problem or only part of the solution. In the former, maybe visibility should be sought, whereas in the latter there is advantage in transparency.

What is it that we distribute? Data or code? Essentially, we distribute data, or processing functions, or both, and the tradeoff is mainly equated in terms of the balance of computing and networking resources involved, and in the end, of performance. We may invoke a remote operation on remote data, which causes the least overhead at our site (as with database searches). We may invoke a remote operation on local data that we ship (*data shipping*) to the server, or have remote data shipped back to our site and processed (as with file system operations). Both use some network bandwidth, and the latter also uses local computing resources. Alternatively, we may have code shipped back to our site (*code shipping*), to execute on local data (as with applets).

A common but pernicious misunderstanding is the often made confusion between *service* and *server*. In an environment where modularity is a keyword, nothing could be more wrong than to equate them as equal. However, it is common to hear "the name server is down", without considering asking a few

questions, such as: "did the name service go down, but the hosting server is still up?"; or, "did the server hosting the name service go down?"; in which case, "can the name service migrate to another hosting server?". The recognition of this difference opens perspectives for a tradeoff between number of servers and number and location of services on the former, dictated by parameters such as per service overhead, load balancing, location, criticality, etc. A physical server can and should host several services, but these services should not be "glued" to the machine, but rather be configured and set up to be modular and portable between the pool of servers of the facility (*see* also Configuration in Part V).

Scale is normally detrimental to *performance*, and this tradeoff is a delicate one. A good architecture should exhibit scalability, that is, the ability to expand in number of components and geographical span, without the performance being linearly affected by that growth. *Openess* is often desired, specially in large-scale distributed systems. However, an open system has less chances of being controlled and exhibit *determinacy*, if compared to smaller scale systems. Finally, we have the synchrony versus asynchrony tradeoff. *Asynchrony* means simple but time-uncertain systems, whereas *synchrony* means more complexity but ability to secure timeliness specifications.

In conclusion, a lot of these tradeoffs end up being equated together, since very often the question is put between:

- **distribution in the small** – the development of small-scale distributed systems in closed environments, typically of homogeneous nature, over LANs or high-speed networks, where controlled behavior can be achieved, for applications requiring reliability and synchrony (e.g., real-time);
- **distribution in the large** – the development of large-scale distributed systems in open environments, typically of heterogeneous nature, over WANs-of-LANs, where behavior is uncertain, and applications can live with asynchrony (e.g., Internet).

3.2.2 Distribution for Information and Resource Sharing

This is how distributed systems started, and is still the strategic objective of the most part of distributed settings, from public-oriented Internet-based sites and servers, to private organizational facilities, such as intranets. The technologies that are concerned with implementing this strategy have to do with remote access to central servers: remote session protocols, client-server computing, HTTP thin-client and network computing. They also have to do with classic information dissemination technologies, such as e-mail, news and bulletin boards. In implementing this strategy, it is advisable to concentrate on the users side, that is, to guarantee access to the information and resources residing on central services, assuming the servers have perfect availability and performance. Assuring this with the quality of service desired requires looking at: user management (registration, naming, administration); session security (authentication, authorization, access control policing); communication band-

width and reliability (network planning); client-side software (system support, protocols, applications).

3.2.3 *Distribution for Availability and Performance*

Assume that the strategy of distribution for information and resource sharing has been laid down in terms of *logical servers*, in a divide-and-conquer approach, as we suggested. However, servers are not perfect. They do not have infinite performance, nor are they always up. Several strategies may be laid down for the mapping from logical to physical servers. The baseline approach relies on a *single server*, multiprocessor if need be, or even a mainframe. This is simple, but it is a single point of failure and it scales hardly. A central server providing to a community of users should not be unavailable or overloaded. Unfortunately, this is what we see too many often, e.g., in Web sites.

So, can better be done? Recall that distribution has attributes that may be useful here: geographic separation, and failure independence. Allocating services wisely to several servers reduces the probability of total failure, achieving *graceful degradation*. Distributing a service through several servers (*load balancing*) improves performance. There is a link here between parallelism and distribution. Mechanisms for *coarse grain parallelism* may be relevant to enhance load balancing. These measures grant an entry level of availability and performance improvement that may be just enough for most settings.

Introduction of replication, whose methods we will study in detail (*see* Chapter 8), enhances our strategy further. By hosting replicas of the services in different servers, the probability that the service is ever down can be drastically minimized. Incidentally, service duplication (two replicas) is normally enough to guarantee a reliability above 0.99.

Finally, if replicas are located near the users, or groups of users, performance and availability may be further improved. This is specially true if the system scale is considerable, and networking is slow and prone to partitioning. There are two facets to this strategy: (a) replicating the server, e.g., a distributed transactional database manager that has replicas located near relevant groups of users; (b) replicating data, e.g., the *cache* hierarchies of the web that replicate pages in proxy servers located near relevant groups of users and even in the client browsers.

3.2.4 *Distribution for Modularity*

Imagine a central computing facility of a company, where all services are hosted, serving the company's headquarters, co-located with the facility. Despite the fact that services and users are nearby, distribution may still have a relevant role as a structuring artifact, through a powerful attribute: modularity.

The strategy is materialized by organizing the architecture *modularly*, as a distributed system, using distributed systems techniques. The cost of this approach is that management of such a setting is more complex than its integrated counterpart. What is to be gained is explained by two orders of reason.

The first is concerned with a greater ability for *managing the uncertainty* in the evolution of the organization and its activity (geography, scale, re-engineering, reorientation, markets). The second is concerned with leveraging investments in distributed systems technologies aimed at improving performance and availability through distribution, combining two strategies and thus killing two birds with one stone.

3.2.5 Distribution for Decentralization

There are a number of human-driven activities that are decentralized in their nature. To this decentralization corresponds a certain local autonomy of means and procedures, controlled by coordination points with the rest of the structure. Before distributed systems made their appearance, informatic support of these activities took only two possible forms, whatever their degree of decentralization. Either everything was based on centralized computing facilities, with virtual terminal connections to human operators, or there were several, independent computing centers or *islands*, that would transfer information among each other off-line.

Decentralization occurs in various degrees: highly decentralized operation, e.g., teleconference, concurrent engineering; moderately decentralized operation, e.g., automated manufacturing cells; little decentralized operation, e.g., client-server with client based processing autonomy. Distribution serves to adapt the activity model to the computing infrastructure. The strategy is *decentralizing* control, by placing it where it is required, while retaining the necessary degree of *integration* and *coordination* between the several sites. In consequence, it develops along two axes: to provide the several loci of control with a consistent view of the state evolution of the system (distributed state dissemination); to enforce coordination of actions according to the degree of decentralization desired (distributed algorithmics).

3.2.6 Distribution for Security

We are reaching a changing point, where systems security is no longer assured *in spite of* distribution, but rather, *with the help of* distribution. This is emerging in several areas of security, such as protection and cryptography. Authentication systems based on distributed voting servers provide more robust operation if compared to single-site systems. Threshold cryptography based on quorums among groups of distributed participants is a useful paradigm in emerging areas. Archival systems based on file fragmentation and scattering through distributed file servers increase resilience if compared to single-site whole-file hosting.

3.3 ASYNCHRONOUS MODELS

Fully asynchronous distributed systems are those systems where time does not count. That is, the applications we run on those systems should be satisfied

by guarantees of the liveness kind, such as "a message is eventually delivered", or "the execution eventually terminates". The properties of the asynchronous system model are defined in Table 3.1. In essence, they show that the system is free from temporal constraints[1], or *time-free*. Note that it is implied that processing and message delivery can take an arbitrarily long time, and that clocks are not useful in their role of providing time references (as we have defined it in Section 2.5), not even locally, let alone in a distributed way.

Table 3.1. Asynchronous Model Properties

• Processing delays are unbounded or unknown
• Message delivery delays are unbounded or unknown
• Rate of drift of local clocks is unbounded or unknown
• Difference between local clocks is unbounded or unknown

Such a model is simple, and obviously adapted to environments that give very little guarantees. However, a fully asynchronous model has limitations that compromise its usefulness for practical systems. If faults occur (even as simple as machines stopping), it is impossible to guarantee the deterministic solution of basic problems such as consensus, a statement which became known as the *FLP impossibility result* (Fischer et al., 1985). Moreover, when using certain paradigms, for example when managing replicas under primary partition consistency, we cannot even reconfigure the system deterministically (Chandra et al., 1996). Last but not least, we cannot guarantee the slightest time bound for the duration of an execution. In consequence, what practical systems do is attempt at relaxing the full asynchronism assumptions. Two main tacks or combinations thereof have been taken:

- considering that the system is not *always* asynchronous
- considering that the system is not asynchronous *everywhere*

The main problem haunting correctness of applications in the asynchronous model is the impossibility of telling a slow site or participant, from a failed one. **Failure detection** is a paradigm addressing the correct detection of several types of failures in distributed systems components. *Crash* failure detectors are used in asynchronous systems to detect crashes, that is, sites or participants that fail suddenly by stopping. The trustworthiness of their decisions is obviously constrained by the difficulty of telling genuine failures from unpredictable delays. However, if the system is not *always* asynchronous, it will have periods where this detection can be made reliably. Certain systems rely on

[1]The last two are essentially equivalent: since a local clock in a time-free system is nothing more than a sequence counter, synchronized clocks are also impossible in an asynchronous system. However, they are listed for a better comparison with synchronous and partially synchronous models.

this hypothesis, for example, the *asynchronous systems with failure detectors* (Chandra and Toueg, 1996). Another approach is to consider that for some of the properties, bounds do exist, which is equivalent to saying that part of the system structure is not fully asynchronous, or in other words, that the system is not asynchronous *everywhere*. For example, that clocks have bounded rate drifts, and/or can be synchronized, or that there are bounded message delivery delays, even if large. Certain systems rely on this hypothesis, for example, the *timed asynchronous* (Cristian and Fetzer, 1998), or the *quasi-synchronous* (Veríssimo and Almeida, 1995) systems.

3.4 SYNCHRONOUS MODELS

An asynchronous model expects very little from the environment. It may wait an undefined amount of time for the completion of a problem, but nothing bad happens during that period. From that sense, it is a very safe model, but not live, and from a system's builder and user perspective, availability and continuity of service provision are mandatory requirements. Speaking more formally, we need to solve problems in bounded time and the alternative approach seems to be a synchronous model. A synchronous model is one where known bounds exist for execution duration, message delivery delay, etc., such as described in Table 3.2. In essence, the table shows that the speed of system evolution has a known lower bound, and furthermore, that clocks can be used to determine timing variables, such as position of events in the timeline, or measurement of durations[2].

Table 3.2. Synchronous Model Properties

• Processing delays have a known bound
• Message delivery delays have a known bound
• Rate of drift of local clocks has a known bound
• Difference between local clocks has a known bound

The main problem with the synchronous model is when either the synchrony of the environment or the worst-case load scenarios of the application are difficult to determine. In these cases, the system may function incorrectly. In the first case, because the bounds may be violated. For example, our application relied on the assumption that it took at most $100ms$ to deliver a message— if some messages take longer, something may go wrong, such as messages being delivered out of order, or ignored. In the second case, because the bounds may become insufficient. For example, we assumed requests arrive at a server

[2]The last property specifies the existence of synchronized clocks. Whilst not required of every synchronous system, the first three properties make it possible.

at a rate of 100 per second, which is quite alright for a processing bound per request of $10ms$— if there are peak bursts of arrival, say 100 requests in a half-second interval, some requests may be delayed, or even overrun. This problem only has to do with the difficulty of ensuring that the assumptions made about the system and the environment hold. This can measured by a probability called **assumption coverage** (*see* Section 6.2 in Chapter 6). Of course, partially synchronous models such as suggested to handle the shortcomings of asynchronous models may also prove effective to handle the problem of lack of coverage. Seen from the viewpoint of synchrony now, these models withstand some uncertainty in system behavior, that is, they tolerate that the system is not always synchronous, or is not synchronous everywhere.

3.5 CLASSES OF DISTRIBUTED ACTIVITIES

For the sake of a better understanding of the several models of distributed processing, it is useful to informally divide distributed activities into three primitive classes: coordination, sharing, and replication.

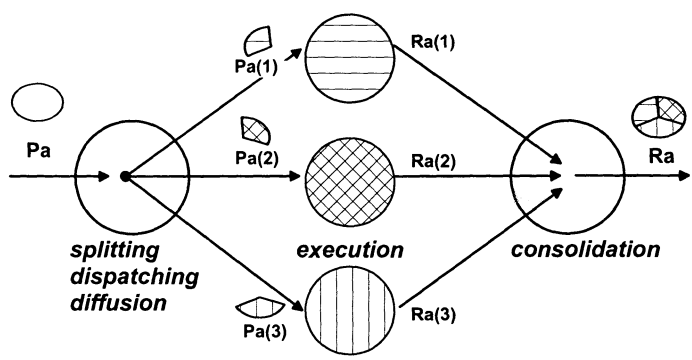

Figure 3.3. Flow Diagram of Coordination Activities

3.5.1 Coordination

Coordination concerns the necessary steps for the execution of actions in several sites that contribute towards a common goal. The generic information flow diagram is depicted in Figure 3.3: requests $P_a, P_b, ...$ are to be executed by a set of participants, each doing part of the job. The generic scheme involves the following phases: splitting, dispatching, diffusion, execution, consolidation. *Splitting* consists in dividing each job, say P_a, into several tasks $P_a(1), P_a(2), ...$ to be performed by some or all of the participants. *Dispatching* consists in allocating the tasks to participants, in adequate number and capabilities, and in the order required by the application. *Diffusion* consists in getting the partial tasks to the relevant participants. Splitting and dispatching can be done at the source issuing the request, as is done by a parallelizing compiler that splits a job and dispatches the parallel tasks by the several available processors in

the distributed system, and then disseminates the task requests as appropriate. Alternatively, the whole job request can be disseminated to the working participants, in order that splitting and dispatching are done at the destination, in a decentralized manner. It requires that the participants have an agreed algorithm that deterministically splits the job and extracts their task at each site. If the participant team may vary dynamically, paradigms such as view synchrony and membership are useful tools to develop these algorithms (*see* again the example of Figure 2.31 in Section 2.9). Ordered diffusion may or may not be necessary, depending on whether the split/dispatch rules depend on the past history of the system. Certain coordinated decentralized activities, for example in distributed control, require all participants to have a *common knowledge* about system evolution. This is best achieved through a totally and causally ordered message diffusion flow. *Execution* takes place at the relevant participants, and the task results $R_a(1), R_a(2), ...$ are then consolidated to form the job result R_a. Once again, consolidation may occur at the source or at the destination. *Consolidation* at the destination is simpler, but implies knowledge by the destination on how to consolidate $R_a(1), R_a(2), ...$ into R_a. Consolidation at the source requires the working participants to run an algorithm, after which the result is returned.

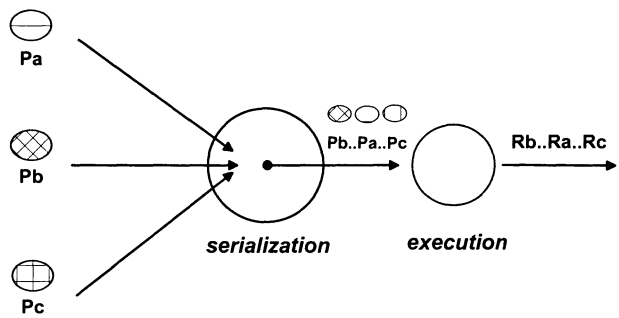

Figure 3.4. Flow Diagram of Sharing Activities

3.5.2 Sharing

Sharing concerns the necessary steps to ensure the correct execution of actions directed at shared resources. As depicted in the flow diagram of Figure 3.4, several action requests $P_a, P_b, ...$ directed at a common resource are *serialized* before execution. This can be done at the destination, or within the protocol that directs the requests to the resource, or both. The ordering discipline depends on the semantics of the activity. For example, if all possible relations between the sending participants must be traced, then causal order of the requests must be ensured, for proper serialization. If participants do not interact, then FIFO order is enough to secure correct serialization (*see Ordering* in Chapter 2). But if shared actions on the resource are commutative (e.g., cast-

ing votes in an electronic ballot), or if serialization is done at the destination, then the protocol can be unordered.

3.5.3 Replication

Replication concerns the necessary steps for the execution of the same set of actions in several sites, such as to produce the same results. The generic information flow diagram is depicted in Figure 3.5. The same request P_a is executed by a set of participants, in a process that involves the following phases: diffusion, execution, consolidation. *Diffusion* consists in disseminating the request to the relevant participants. Depending on the model of replication management, the protocol may or not require that all messages are reliably delivered and totally ordered, in order to enforce replica determinism (*see Replica Determinism* in Chapter 2). *Execution* takes place at the relevant participants, and once more according to the replication model. For example, in what is called *active replication*, all participants execute the same set of requests in the same order. However, in *passive replication*, a primary replica is the only to execute the requests, whereas the backup replicas simply log them, and receive state updates from the primary, at specific points in the computation called *checkpoints*. The execution results are then *consolidated*, in order that the correct result is delivered. For example, if the fault model is omissive, that is, if replication is only used for availability, then it suffices to deliver one of the results, $R_a = R_a(i)$, perhaps the first to be produced. However, if the fault model includes value faults, then majority voting is desired amongst the several replica results, that is, $R_a = vote(R_a(1), ..., R_a(n))$ (*see Replication Management*, and *Resilience and Voting*, in Chapter 7). This form of consolidation requires the working participants to run an algorithm, after which the result is returned. Alternatively, consolidation can be performed at the destination, that is, all $R_a(i)$ results are delivered and acted upon by the recipient. Less frequent, this form of consolidation is however very efficient, specially for cascaded or recursive replicated computations.

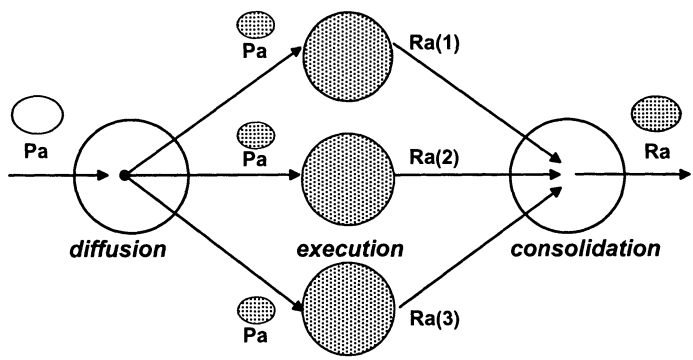

Figure 3.5. Flow Diagram of Replication Activities

3.5.4 *Combining Activities*

Conceptually, it is helpful to see distributed activities as a combination of coordination, sharing, and replication. To get an idea, imagine a distributed data repository (e.g., a database), made of fragments in several sites that are also replicated, and try and analyze it under the light of the activity classes we discussed in this section. The repository is shared by several users that access it competitively. Access to the fragments has to be coordinated, so that each request is handled by the competent fragment manager. Finally, each fragment is replicated, so that the logical database is always accessible. Figure 3.6 exemplifies the set-up. There are three fragments, each replicated twice, and the replicas are located by pairs, in three sites S_1 to S_3, so that each pair of replicas in different sites. The user requests, $P_a, P_b, ...$, are serialized by a multicast protocol, which ensures that the repository as a whole receives competitive requests in the desired order (e.g. FIFO). Splitting and dispatching are done at the destination. In consequence, the request multicasts are disseminated to the several sites. The architect made the multicast reliable and totally ordered, to ensure that all fragment replicas receive all requests in the same order, and can thus take deterministic decisions in a decentralized way. For example, a request concerning fragment a_2 will be processed only by the relevant fragment manager. Likewise, if active replication is used, a request concerning a_2 will be processed by both replica managers, that is, at sites S_2 and S_3. On the other hand, in a complex transaction, operations may take place in more than one fragment. The architect selected consolidation at the source, so the partial results $R_a(i)$ are consolidated and the final result R_a delivered to the requester.

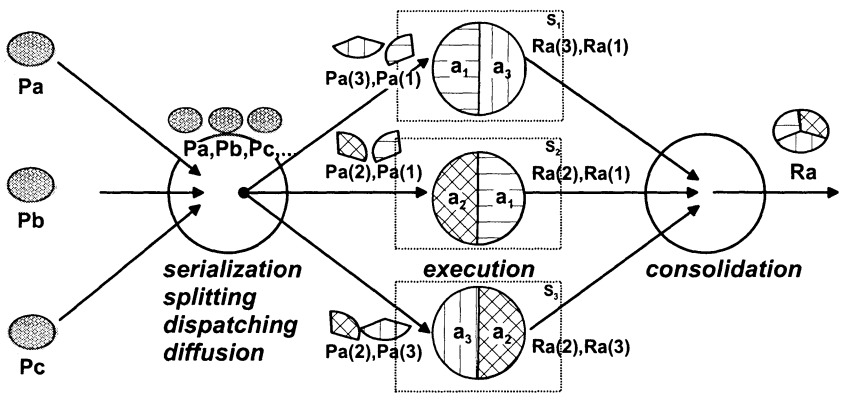

Figure 3.6. Combining Activities

The identification of primitive classes of distributed activity is mainly an analysis method. For performance and efficiency reasons, the design of real toolboxes and applications is often not that modular and neat. Still, the architect has only to gain if the first blueprints of the software architecture of a distributed system are well structured, even if naive as in the example of

the figure. There is time for compacting and optimizing in the later phases of design.

3.6 CLIENT-SERVER WITH RPC

The client-server model is the most used model in distributed computing. As the name implies, it consists of structuring the application around the notion of *servers*, which provide services, and *clients*, which use those services. To obtain a service, the client sends a *request* to server which, in turn, sends back a *reply* with the results or just a confirmation that the desired service was (or was not) provided. The approach can be applied to a large number of distributed applications. In fact, many highly successful applications such as the pervasive WWW, for instance, have been built around the concept.

In terms of programming model, it is possible to establish a parallel between requesting remote services and requesting local services. Traditionally, when a program requests a local service (to read a local file for instance) it does so through a function call. In most modern operating systems this is implemented as call to a library function that, in turn, performs a system call. Thus, programmers of non-distributed applications are already familiar with the concept of requesting a service by calling a function. One can preserve this paradigm when providing support for requesting service from remote servers. The idea is to organize the software in such a way that the client continues to make a function call as in the non-distributed case. Instead of just trapping the kernel, the library function must perform the request-reply protocol already discussed, ensuring the the service is provide by the remote server (actually, to implement the request-reply protocol, the libary function may have to trap the local kernel more than once). For the client application, everything looks like if it was able to invoke a function on the server to obtain the service. When this approach is used, we say that client-server interactions are structured as *Remote Procedure Calls* (RPC).

3.6.1 RPC Architecture

The RPC concept is very simple and intuitive. To make its use practical and efficient, a software infrastructure is required that provides tools and mechanisms to help the programmer of distributed applications. This is not as simple as it may look at first glance. A complete package to support RPC includes communication primitives, mechanisms to name and locate servers, mechanisms to perform data format conversions, and so on. The motivate the need for all these things, let us describe the interactions with more detail. Assume that you want to build a server that provides the following procedure as a remote service:

```
int dosomething (int param);
```

This is an extremely simple procedure that takes as input a single parameter (an integer) and returns a single result (also an integer). The goal of the RPC system is to allow a client to call `dosomething` at one machine (the

client machine) and allow the computation of the result, based on the input parameter, to occur at another machine (the *server* machine). Furthermore, for convenience of the client code, the call to `dosomething` should similar to a local function call. To achieve this goal, the trick is to let the client locally call a fake `dosomething` that looks like the real `dosomething` (which is going to be executed in the server). This fake function has the same signature, i.e, the same name, parameters and results, so it really looks like the original. This function is usually called the *client stub* or the service *proxy* (M. Shapiro, 1986).

The sequence of actions concerning the execution of an RPC are illustrated in Figure 3.7. The client stub is responsible for sending a request to the server and waiting for a reply. To send the request, the stub must first create the corresponding message. In order to do so, the stub converts the input parameters in a format suitable for being transmitted over the wire and recognizable by the server. This process is known as *linearization* or *marshaling*. After being formatted formatted, the message is sent to the server through some *session level* protocol using the communication system. That protocol is responsible for retransmitting the request, waiting for replies, discarding duplicate or obsolete replies, etc.

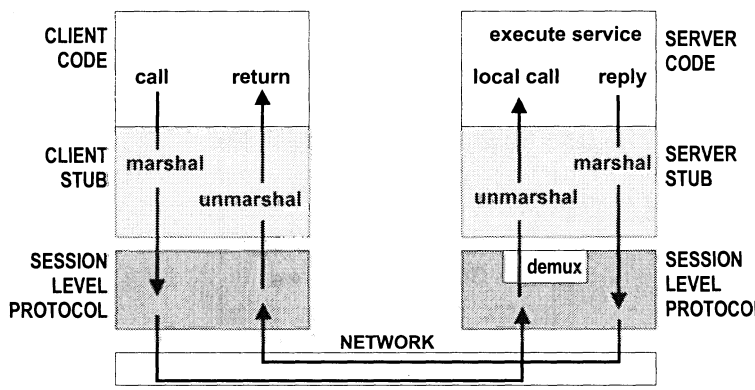

Figure 3.7. RPC in Action

On the server side, the message follows a symmetric path. It is received and processed by the session protocol, which identifies the target service and delivers it to the relevant *server stub*. The server stub extracts the parameters from the message (this task is called *unmarshaling*) and does a local call to the real `dosomething` that does the work. When this function returns, the server stub creates a reply message and hands it to the session protocol in order to be returned to the client, following the same steps as before, now in the opposite direction. On the client side, the client stub finally returns the original call to the client, with any relevant results.

The RPC functionality is cast into a few major building blocks which make up the architecture of an RPC system, as illustrated in Figure 3.8. The user package is a library that includes functions that help the application designer to

deal with the aspects that cannot be hidden by the client proxy. For instance, the client application may be required to explicitly invoke a naming service to locate the server. In this case, procedures that perform these functions would be available in the user package. The client and server stubs are responsible for marshaling and unmarshaling the procedure parameters. Finally the communication protocol is responsible for making sure that the messages are delivered to their recipients. It should be noted that at least a portion of the communication protocol is usually executed by the operating system kernel. Thus, in order to execute a remote procedure call one needs to perform at least the following calls: a local function call to the client stub, a system call to the kernel to send the message, to cross the network, to commute from the server kernel to the server stub, finally a local call to the function that implements the service (plus the same sequence for the reply).

Comparing the cost of the sequence with the cost of a local function call it becomes clear that there is a significant overhead in the RPC. However, for coarse grain services, this overhead may represent a small percentage of the time required by the server to complete the service. Actually, for certain services such as file systems, one can achieve performances which are comparable to the local case or even, in extreme cases, outperform the local calls, by avoiding other costs, such as context switches.

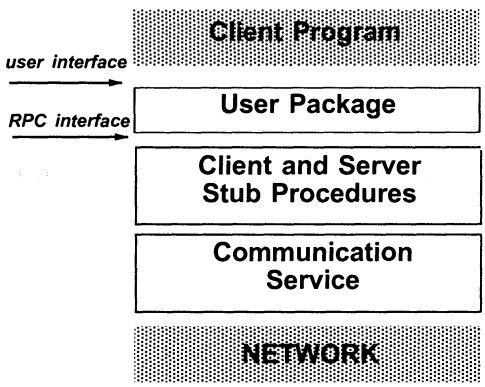

Figure 3.8. RPC Architecture

3.6.2 Handling exceptions

We have stated that the goal of a RPC system is to make remote calls look like local calls. Unfortunately, in a distributed system this is impossible to achieve because the client and the server have independent failure modes. In a local call, if there is a failure, the whole process crashes. In a remote call, the server can crash and the client remain active. Thus, new errors, that do not appear in the local case, can occur when a service is provided by a remote server. This

also means that it is very hard to take a client written to make just local calls and, in a fully transparent manner, make it operate using RPCs.

In languages that support exception handling (such as Java), the RPC may raise new exceptions. If the client is not prepared to handle those exceptions, it may be forced to crash when some anomaly occurs. If the programming language has no exception mechanism, error codes have to be returned to the client as an output parameter of the procedure call. However, this forces the original signature of the procedure to be changed to include either the error parameter, or the new error codes if the parameter is already there.

3.6.3 RPC protocols

A fundamental part of the RPC architecture is the session level protocol that is used to exchange requests and replies between the client and server. The protocol defines the steps required to exchange requests and replies, the format of the messages, how and when these are retransmitted, etc.

Naturally, the session level protocol needs a transport level protocol to send and receive packets. Two alternatives are available for this purpose: to use a connection-oriented byte-stream protocol, such as TCP, or to use a connectionless protocol such as UDP. In the former case, issues such as reliable delivery, fragmentation and reassembly, retransmission management, failure detection and so forth are handled by the transport protocol itself. This makes the life of the RPC designer much simpler. Unfortunately, establishing a connection is a costly procedure that introduces a non-negligible latency in the first RPC. In systems where the interactions between clients and servers are limited to a small number of calls (often, just one) this overhead can be the primary cause of RPC latency.

Because of the performance limitation of connection-oriented protocols, one may be tempted to rely on a connectionless transport service instead. However, this approach requires the session level protocol to address explicitly problems such as the need to fragment messages and the need to ensure reliable delivery. One reason to follow this path is that it is possible to exploit specific RPC semantics to build a "tailored" protocol that can address these problems in a more efficient manner than a generic reliable byte-stream protocol.

There are some intuitive arguments in favor of using a tailored transport protocol. For instance, reliable transport protocols usually require the exchange of acknowledgments. In an RPC protocol, an explicit acknowledgment to confirm the reception of the request can be redundant, since it may be soon followed by a reply. On the other hand, if the acknowledgment is suppressed, it is hard for the client to distinguish the case where the request is lost from the case where the server is taking a long time to process it. Then, an aggressive approach (retransmitting the request too soon) wastes resources, whereas a conservative approach (waiting for a long timeout) increases the latency in the case a real omission has occurred. In those cases it may be preferable to send the acknowledgement back to the client anyway. Unfortunately, when an acknowledgement is received but no reply follows, the client is still in trouble: does this mean

that the server has crashed while executing the request? Additional "are-you-still-there" requests (followed by a "yes-i'm-just-busy" response) may have to exchanged to keep the client waiting for the reply. No matter how we improve the protocol, there are fundamental limits to this reliability–performance tradeoff that we address in the Fault Tolerance part (*see Fault-Tolerant Remote Operations* in Chapter 8).

If heterogeneous machines are to be supported, then in addition to the transport protocol the RPC architecture requires both ends to agree on a common format to code the procedure parameters. Naturally, the marshaling and unmarshaling procedures can also consume a reasonable amount of time, specially if type checking is performed in run-time. For high-performance RPC between machines with the same architecture it may be worth to disable marshaling altogether and to send the data structures exactly as they are stored in the local memory. The reader should be aware that this trick only works with flat data structures, such as records or arrays. Complex structures such as linked-lists, trees, etc, need to be linearized regardless of the architecture in use.

3.6.4 Building Client-Server Systems

In the previous section we have described how an RPC works. In this section we discuss what code needs to be provided by the programmer when building a distributed application structured around RPCs. To start with, the application programmer needs to develop the function that provides the service to be executed in the server. If she is building the complete system, she will also need to develop the client that invokes the service. But how about the communication protocol, the client stub and the server stub? The good news is that there are many platforms that ease the task of the programmer with this regard.

The RPC protocol can be provided as a library to be linked with both the client and server code (or as a mixture of library and kernel code). Thus, it does not need to be re-written by the application programmer. On the other hand, each service function has its own signature, and the code that performs the marshaling depends on its specific parameters and result values. This means that specific client and server stubs need to be coded for each procedure. Implementing these procedures manually is an extremely tedious job. In fact, we have already discussed the need for using some pre-agreed format to linearize the procedure parameters, thus there is no room for creativity here. The solution is to rely on a tool that creates the stubs automatically, based on a description of the service interface. This tool is called a *stub compiler*. In order to implement a stub compiler one needs:

- an *Interface Definition Language* (IDL) to declare the procedure interface(s);
- a target programming language in which the stubs should be produced;
- mappings to translate the IDL types into the corresponding types of the target programming language;
- the pre-agreed format for coding the parameters in the RPC messages.

The use of an IDL is fundamental to support heterogeneity. As long as appropriate mappings are defined for several programming languages one can build a client-server application where the client is implemented in one language (say, Java) and the server in another (for instance, 'C++' for performance reasons). However, it is a bit unfortunate that in order to ensure interoperability between components written in two different languages, the programmer has to learn yet another language, IDL, before she can start implementing her RPC-based client-server application. Why not just pick some "popular" programming language and use it to define the interfaces? To start with, some languages are appropriate for some purposes and completely inadequate for others. Ideally, an RPC package should not favor some types of application in detriment of others. Additionally, the most widely used languages are not designed for building distributed systems. For example, they support certain data types, such as pointers, which make impossible the task of automating stub creation. Consider for instance, the following function definition in the 'C' programming language:

```
int foo (char* p);
```

This definition is clearly ambiguous from a stub generator point of view. To start with, there is no automatic way of extracting the size of the buffer being passed as a parameter. For instance, it can be a block of memory whose size is implicitly agreed between the caller and the callee but not explicitly declared. It can also be a string, in which case the size can be computed at run time by parsing the string until a terminator is found. Even if the size is known, there is no automated manner of discovering if the parameter is an input parameter, an output parameter (i.e, if the caller just provides the pointer and expects the callee to fill the buffer) or an in-out parameter (the caller provides a buffer that is altered by the function). Thus, a stub compiler would have to always include the buffer in the request and in the reply message, which could be a waste of resources.

With a powerful, yet unambiguous IDL language, the programmer can focus on the application design and leave the tedious task of building the client and server stubs to the stub compiler. In addition to the marshaling and unmarshaling of parameters, the stubs can also automate the process of establishing the communication channel between the client and the server. This process is known as *binding*. Note that the complexity of the binding procedure depends on the type of communication protocol used: for instance, a connection-oriented protocol may require a connection to be established.

The simplest form of binding is *static binding*, i.e., each service is provided at a pre-defined address (in the IP world, at a pre-defined address and port). This approach is seldom used, since it does not provide any support for dynamic system configuration. A slightly better approach is to have the node of the service provider fixed but allow the port to be defined at run-time. In such case, the port must be discovered dynamically, for example by letting the client make an RPC to a small local name server at the target machine. This local

name server, called the *port-mapper*, maintains an association between ports and services (the port-mapper itself runs at a well-defined port). The fully-fledged solution is, of course, to rely on an external name service. Thus, before establishing the communication channel, the client must inquire the name service to obtain the address of the server. A cache with the addresses of recently used servers may speed up this step. Naturally, during its initialization step, the server has to register its own address on the name server.

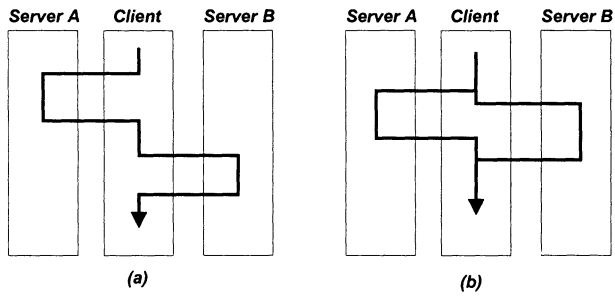

Figure 3.9. Client Threads: (a) Single-threaded; (b) Multi-threaded

The last issue related with the implementation of RPC systems we want to discuss is the use of multiple threads, both in the client and in the servers, leading to different programming styles as illustrated in Figures 3.9 and 3.10.

Let us first discuss the need for the use of multiple threads in the client. When an RPC is executed, the client is usually blocked until a reply is received. If the reply takes a long time, either because there is a high network latency, or because the service takes a long time to execute, it may be worth to allow processing to proceed in the client. Also, as we have discussed earlier, the client may want to perform several RPCs in parallel to avoid the serialization of network latencies depicted in Figure 3.9a. In these cases, a new thread may be created to execute each RPC and terminated when the RPC completes, as illustrated in Figure 3.9b.

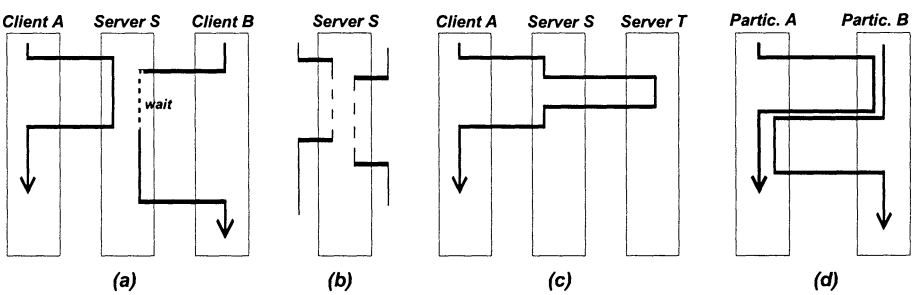

Figure 3.10. Server Threads: (a) Single-threaded; (b) Multi-threaded; (c) Multi-tiered; (d) Conversational

On the server side, the use of multiple threads is of paramount importance to increase throughput, namely when the execution of the service requires the server to execute I/O operations. For example, consider the case of a server that has to access the disk in order to execute a service (such as a file system server). If a single threaded server is used, the server will be blocked on the I/O operation and unable to process requests from other clients, as illustrated in Figure 3.10a. A multi-threaded server allows different requests to be processed in parallel by different threads, as illustrated in the detail of Figure 3.10b. If the execution of a request forces a thread to block on an I/O operation, another thread is scheduled to run and a request from another client is processed. Naturally, the gains in performance come at the expense of more complex servers, since the different threads must synchronize to ensure that shared data is updated in a consistent manner. Additionally, it may happen that requests end-up being executed in an order different from the order they were delivered by the communication protocol. In many cases, request are unrelated and this makes no difference, but in cases where causal dependencies exist between requests, the servers threads must also synchronize to respect these order relations.

It should be noted that, in order to provide a service, the server itself may have to invoke remote servers, as illustrated in Figure 3.10c. Thus, in logical terms, we can describe the application as executing a distributed flow of control that is propagated from client to server according to the sequence of RPCs. This style of interactions is also called multi-tiered. Note that in this case, the intermediate processes act both as a client and as a server.

In an extreme case, we have a scenario where all processes are both clients and servers, and perform *multi-peer* or conversational interactions with an extremely free discipline, as shown in Figure 3.10d. Multi-peer interactions do not fit very well in the asymmetric nature of client-server. However, it is easy to find simple scenarios where it is useful to let the client also play the role of a server. One of the most intuitive examples is the case where a client wants to be informed of the change of some variable in the server. One way to achieve this goal is to let the client periodically check the status of the server. This method, known as *polling*, is not efficient. Another way is to let the client register a *callback* procedure in the server and wait for the server to call this procedure when the value changes. Clearly, in this case, the client must be able to perform the role of server.

3.7 GROUP-ORIENTED

Despite its enormous utility, there are some limitations of the RPC client-server model: it is blocking (for the client); it is based on point-to-point interactions (client to server); and it is asymmetric (only the client initiates the interaction). Many current applications require non-blocking, multi-point, and multi-peer interactions. The group-oriented model is one versatile model capable of meeting those requirements, and implementing several styles of distributed programming. Group-oriented programming consists basically of representing the actors and the targets of distributed activities as groups of participants, which

communicate through diffusion, or multicast protocols, with well-defined but varying order, reliability and synchrony semantics.

This model can thus be used in applications where the notion of grouping is inherent, such as: cooperative document editing; teleconferencing; communication in multi-tool CAD environments; flexible manufacturing cells; distributed parallel processing. However, groups may also be useful structuring devices in system software design, by representing sets of objects that must be referenced together, often in a transparent way, such as: replicated process groups; replicated databases; and replicated sensors or actuators. Other examples are: grouping internet routers for selective table updates; grouping pools of system resources for consistent and decentralized allocation; grouping sites of distributed parallel computations.

3.7.1 Group-Oriented Architecture

The basic building blocks of group oriented systems are illustrated by the architecture depicted in Figure 3.11. Note that it is possible to build group oriented systems using different architectures, but it is not the purpose of this section to discuss deeply all the subtle differences among the existing systems. This particular architecture is based on a hierarchical model, that decouples the modules in two major classes: modules that manage interaction among *sites*, and modules that manage interactions among *participants* that execute in those sites.

At the site level, a basic building block is the *Site Failure Detector*, responsible for detecting the failure of other sites. This module is used to provide a *Site Membership Service*, responsible for maintaining membership information about the sites that are participating in the group communication. Site failure detection is also used by the *Multicast Networking* layer, responsible for supporting unreliable message multicast services. The *Group Communication* module is responsible for providing reliability and ordering guarantees to all messages exchanged among sites. Group communication uses the Multicast Networking module to exchange messages and the Site Membership module to obtain the list of active sites.

Participant-level modules are built on top of the site-level modules. The *Participant Membership* module provides membership information at the participant level. When a participant fails, it is marked as unreachable by the *Participant Failure Detector*. The same happens with all participants at a site that is detected unreachable by the Site Membership Service. The *Activity Support* module provides support for several common distributed paradigms, such as, for instance, replication management.

In terms of interaction among participants, a typical sequence of events is illustrated by Figure 3.12. A participant becomes member of a group in response to a `join` request. If the join succeeds, the participant will receive the membership and view of the group; the membership information is also updated at all the other members. After becoming a member of the group, a participant can multicast to and receive from the group. Messages are sent

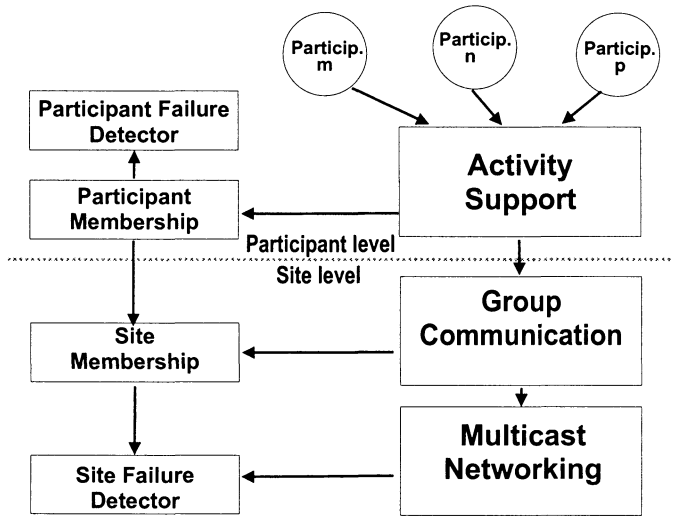

Figure 3.11. Group-Oriented Architecture

with a specified quality of service (in terms of reliability and ordering). At the sender site messages traverse the protocol stack downwards until they reach the network and, at all participants, they traverse the protocol stack upwards and are delivered to the group members with the quality of service requested.

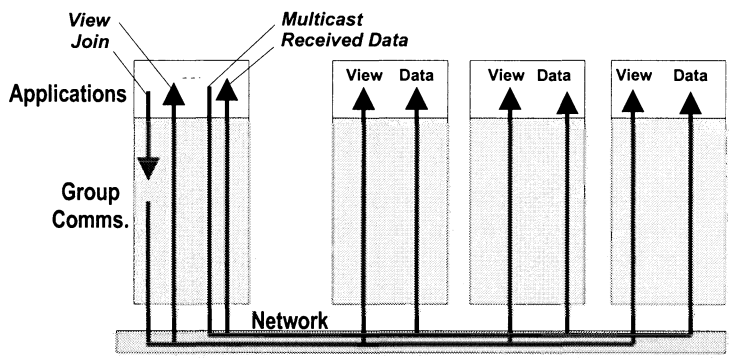

Figure 3.12. Groups in Action

3.7.2 Design Issues

When building an infrastructure to support the development of group oriented distributed applications one is faced with many alternative designs. In fact, many different architectures have been built and can be found in the rich bibliography on the subject.

The most important design issue is to understand what type of service needs to be provided. Nothing has more impact on the system architecture than the user requirements. Is reliability more important than throughput or vice-versa? Are there any real-time requirements? The behavior of the service with regard to faults is also extremely important. Must continuity of service be assured in face of network partitions? What is the appropriate consistency/availability tradeoff for the target application?

Additionally, it is important to understand the properties of the network infrastructure. The reader must be aware that there is always a tradeoff between generality and performance, when designing communication protocols. Generic approaches make few assumptions about the underlying network. The resulting protocols are easy to port to different network structures but often exhibit poor performance. On the other hand, tailored solutions exploit particular features of a given class of networks in order to achieve better performance. However, tailoring has also its disadvantages: if the features exploited are peculiar only to one or two networks, it may be difficult, if not impossible, to port the resulting protocols to other networks that do not own these characteristics. The successful design must capture the right balance between generality and performance (Rodrigues and Veríssimo, 2000).

By matching the application requirements with the properties of the underlying network, the system architect can select the most appropriate protocol for his own goal. It is important to emphasize that, for each specific facet of group communication, different protocols exist that exhibit their best performance under different scenarios. Consider for example the problem of total order. It has been shown (Rodrigues et al., 1996) that some total order protocols are more efficient when the network latency is small and others are more efficient when network latency is large. Then, depending on the system usage, it is preferable to use the former, the latter or some hybrid approach (Rodrigues et al., 1996).

Given that several alternatives are possible, for different usage patterns, it may be wise to use a configurable group communication service. A number of recent systems have been built using a modular approach, where semantically rich services are constructed by combining several small specialized protocols (Hiltunen and Schlichting, 1993; Hayden, 1998). This approach, called the micro-protocol approach, allows the application designer to select the communication stack that precisely matches the application needs. Some of these architectures also provide support to change the protocol configuration in runtime (Hayden, 1998).

Like any other system service that has strong performance requirements, the way the group communication subsystem interacts with the operating system kernel makes a difference. One possible solution is to implement the group communications package in the kernel itself. This introduces some performance gains (Vogels et al., 1992) at the expense of a more complex installation procedure. This approach also simplifies the implementation of models that distin-

guish between *sites* and *participants*, such as the architecture described in the previous section, since the kernel supports all the application processes.

To simplify the installation, the code that would otherwise run in the kernel can be executed in a dedicated server. Some operating systems built using the *micro-kernel* approach are optimized for this sort of configuration. However, in most kernels, the context switch overhead introduced in the message path is non negligible.

To avoid additional context switching, it is possible to run the group communications package in the address space of the application process (as a library). This approach is also extremely simple to install. However, in order to be efficient it requires the use of several threads of control or else requires the application to be driven by a main thread controlled by the communication package (this leads to an event driven programming style that may be awkward in some cases).

3.7.3 Building Group-Oriented Systems

A group communication system offers membership services and group communication services. Group membership allows processes to become members of groups and receive membership information. Group communication allows processes to send messages to groups and receive messages sent to the groups they belong to.

Like any other message-passing interface, multicast interfaces have advantages and disadvantages. On the negative side, a message passing interface may be viewed as a low-level construct, not rich enough to be useful for building complex applications. It lacks the higher-level feel of remote procedure call, even though a group remote procedure call mechanism can be constructed on top of a multicast message passing interface. Of course, even remote procedure call may be considered too low-level if what an application really needs is transaction support. On the other hand, a multicast send/receive interface is very versatile and efficient, and does not restrain participants to play fixed roles such as client or server that do not fit well in multi-participant interactions.

In fact, many of the applications using group communication are better served by a multi-peer style of interaction (supported by a multicast message-passing system) than by any other higher-level interface. For instance, many real-time systems interact with the environment through event notifications and group communication is perfectly adjusted to disseminate events to many processes (these systems are often called *responsive* systems). Other examples of applications for which message passing is particularly well suited are applications in the area of Computer Supported Collaborative Work (CSCW), where a number of users may interact in a multi-peer fashion, often in an unstructured way.

In group communication it is useful to distinguish the role of sender and recipient. The recipients of a message are typically all the group members (although some systems allow just a subset of the group to be addressed). When several application-level groups have similar membership, it is possible to imple-

ment a form of resource sharing called *light-weight groups* (Guo and Rodrigues, 1997). This technique consists in mapping several participant-level groups in a single site-level group. The advantage of such mapping is that site-failure detection and recovery can be shared by all participant-level groups. This optimization is illustrated by Figure 3.13. Figure 3.13a shows two participant-level groups P_i and P_k, each supported by a different site-level group SG_i and SG_k. Figure 3.13b illustrates the same participant-level (light-weight) groups supported by a single site-level group.

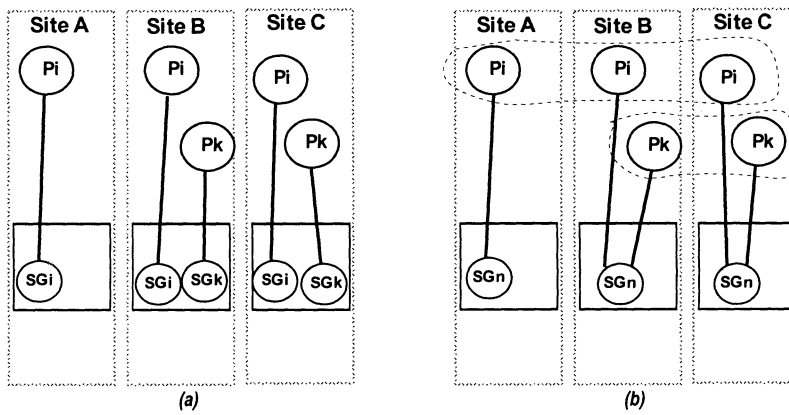

Figure 3.13. Group Access Methods: (a) Normal; (b) Lightweight

Senders to a group can be either members of the group or processes outside the group. A system that only allows group members to send to the group follows a *closed* group model. A system that allows non-members to send to the group follows an *open* group model. The task of sending to a group without being member of the group is made difficult by the fact that the sender must first obtain an *approximation* of the group membership (if not the up-to-date membership) in order to decide where to send the message to. Due to this reason, some systems distinguish between senders that, whilst not being members, keep some form of binding to the group and are informed of group membership changes (called *attached senders* in the discussion below), and senders that just keep a soft connection (called *detached senders*). Detached senders must interact with some proxy that is either a member or an attached sender to a group, as illustrated in Figure 3.13c.

These roles in group communication can be used to structure the application according to several group programming models, such as the client-server model, the dissemination model and the multi-peer model, as illustrated in Figure 3.14.

The client group-server model is an extension of the point-to-point client server model. In this model, instead of having a single server one has a group of servers that coordinate to provide service to one or more clients. The group-server can be used for fault tolerance, for load-balancing or just to exploit

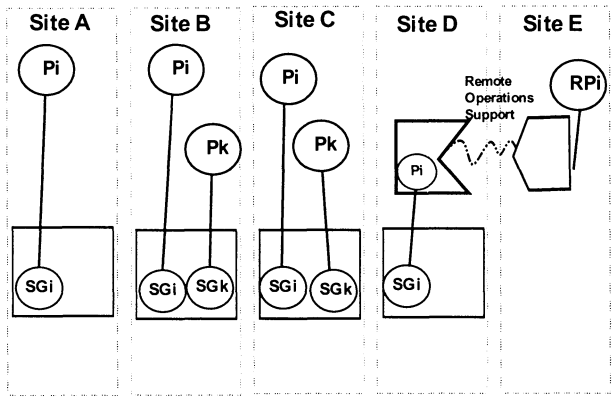

Figure 3.13 *(continued).* Group Access Methods: (c) Remote

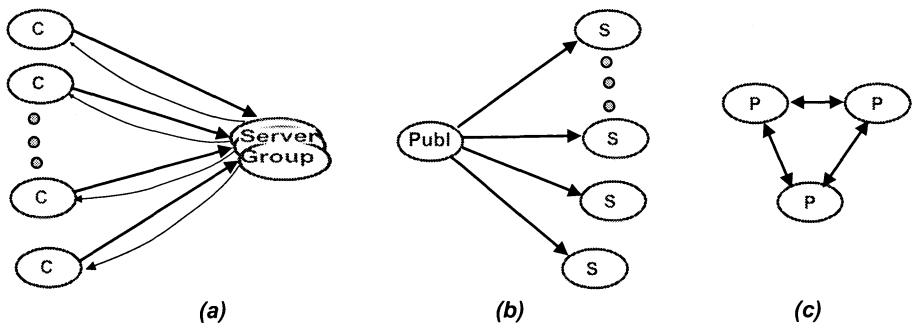

Figure 3.14. Basic Group Programming Models: (a) Client-Server; (b) Dissemination; (c) Multipeer

locality. The key advantage of group communication is that clients do not need to be aware of the number or location of servers; they can simply send a message to the group and way for the first reply, as illustrated in Figure 3.14a. Of course, it is also possible to build clients that are prepared to collect different replies, one from each server, and combine them in a final result. In this model one typically has a small set of servers and a possibly large number of clients.

In the dissemination model one has a producer of information that multicasts messages to a large number of information consumers, as illustrated in Figure 3.14b. The advantage of group communication in this model is that the data producer does not need to be aware of the number/location of recipients and does not need to explicitly send many point-to-point messages. This model is commonly used to disseminate multimedia streams on the Internet and is supported by reliable multicast protocols that use IP multicast underneath. This sort of applications have usually weaker requirements than fault-tolerant applications. Thus, they rely on efficient protocols that, instead

of offering strong guarantees such as virtual synchrony, have better scalability properties.

Finally, in multi-peer interactions, illustrated by Figure 3.14c, all members play a similar role. In this model all participants are allowed to send messages to the group and receive messages sent to the group (including their own messages). Multi-peer interactions are most effective for a small number of participants, and allow the easy implementation of strong quality-of-service specifications (e.g., reliability and ordering).

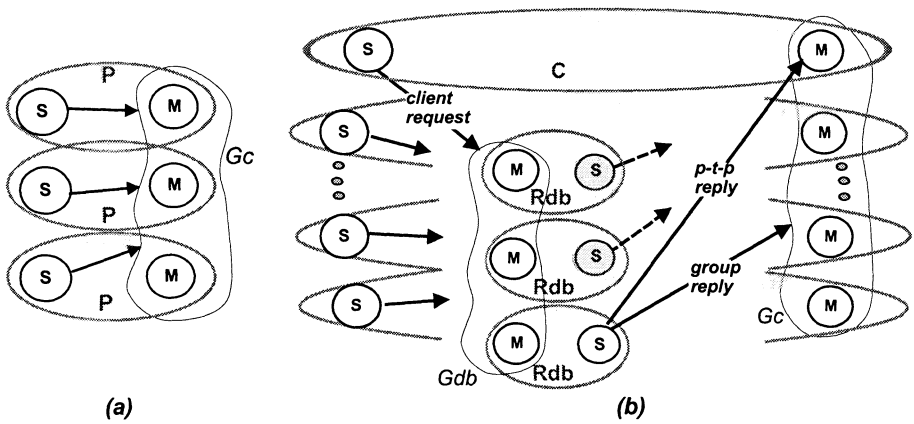

Figure 3.15. Implementing Group Models: (a) Multipeer; (b) Reliable Client-Server

Figure 3.15 illustrates how the sender and receiver roles described previously can be used to implement some group programming models. For instance, a multi-peer model can be easily implemented by having each participant P be both a sender (S) to and a receiver member (M) of the multi-peer group (Figure 3.15a).

On the other hand, a reliable client-server model can be implemented by replicating the server. A group (G_{db}) is associated to the set of server replicas, as illustrated with a database server (R_{db}) in Figure 3.15b. Each individual client will play the role of sender to G_{db}, and each individual server will be a member of the group. In the particular configuration illustrated in Figure 3.15b, a group (G_c) is created to receive the replies. Individual servers are senders to G_c, and individual clients are members of G_c. Of course, one or several client groups can be created, and nothing prevents a reply from going to a single client, as shown in the figure. Note also that the S and M handles of both the servers and the clients belong to different groups, unlike the multi-peer example! Finally, the example suggests that only the bottom server replica is active sending replies, with the others as backups. In fact, a choice of alternatives exists: they all send copies of the reply; or they share the load between them.

3.8 DISTRIBUTED SHARED MEMORY

Distributed shared memory (DSM) is an intuitive paradigm that emulates the execution environment of a shared memory multiprocessor in a distributed system. The model is intuitive in the sense that the paradigms that are valid for concurrent programming in shared memory systems, remain valid in distributed systems (with some limitations that we discuss below). An application area where distributed shared memory paradigms are useful is the area of high-performance computing. Using DSM, parallel programs built for shared-memory multiprocessor systems can be ported to a cluster of workstations.

The ultimate goal of a DSM system is to give the illusion of a centralized shared memory system, both in terms of semantics and in terms of performance. In other words, memory distribution should be *transparent* to the application designer. As it will be seen, this goal not is easy to attain, in particular the performance aspects (the semantic aspects can be easily satisfied if efficiency is not an issue). A fundamental aspect of DSM systems is that the adopted algorithms should try to minimize the number of messages exchanged among nodes, since network throughput can be a system bottleneck. Another fundamental aspect, even more important, is the minimization of latency. In particular, scenarios where a node is forced to wait for a message from another node in order for a memory operation to make progress should be avoided. Another way to express this requirement is to say that remote accesses should be hidden from the programmer.

Of course, in order to implement the abstraction of a global shared memory, the DSM system *has* to exchange messages. These messages are needed to propagate updates performed to the global memory by the several participants involved in the computation. In the management of these message exchanges two conflicting goals emerge. From the message size point of view, if bigger messages are used, less messages are exchanged and this minimizes network overhead. Consider for instance that a process makes several sequential updates to a given page. Instead of sending a different message for each of these updates, a single message can be sent at the end with the updated page. Additionally, managing the state of the distributed memory at a coarser granularity level (say virtual memory pages), reduces the overhead caused by the maintenance of control information. On the other hand, the bigger the pages the higher the chance of running into a scenario known as *false sharing*. False sharing occurs when two nodes update unrelated variables that, by coincidence, are located in the same page. Although at the logical level the nodes are not updating the same data, from the point of view of the DSM system they are sharing the same page. As a result, the page may have to be moved back and forward between both nodes, a phenomenon often called *thrashing*.

3.8.1 DSM Architectures

The most intuitive behavior for the memory is the one of a centralized system where all operations are atomic. This model is called *atomic consistency* or *lin-*

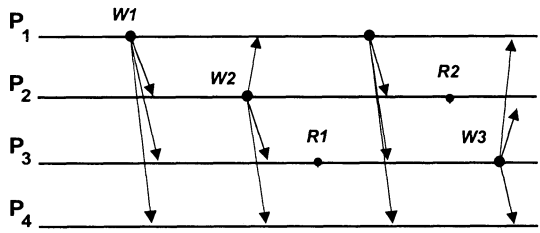

Figure 3.16. Strong Consistency

earizability (Herlihy and Wing, 1990): all write operations are totally ordered (obeying the operations' real time order) and read operations always return the last value written into the memory. The intuition is given in Figure 3.16. If the temporal order of operations does not need to be respected, the model is named *sequential consistency* (Lamport, 1979). The former models of memory consistency are usually labeled as *strong consistency* models. Although these models accurately reflect the programmer's intuitive view of the (single) memory behavior, they are very expensive to implement in a distributed system as they require a total order to be enforced on memory operations. To understand why this is not a trivial task, let us survey the main architectures to support distributed shared memory, depicted in Figure 3.17.

The simplest implementation, actually quite naive, is illustrated by Figure 3.17a. In this architecture, a central server holds the memory pages and all data accesses by the clients are performed through a remote invocation to the server. The server serializes all requests, thus ensuring the total order needed to preserve strong consistency. The system behaves just like if a single central memory was available. In fact, this architecture relies on having a single central memory on the server.

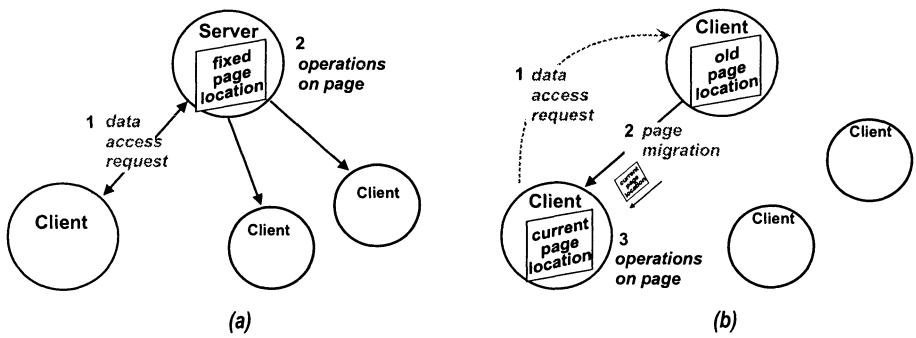

Figure 3.17. DSM architectures: (a) Centralized; (b) Migration

If all memory accesses had to be remote, like in the previous architecture, the performance would be deplorable. Fortunately, programs exhibit some degree of locality in data accesses. Locality means that if a given memory position

is read or written, then it is likely that other adjacent memory positions will also be accessed. Instead of forcing all accesses to execute remotely, one can simply migrate a chunk of memory (say a page) from the server to the client, as illustrated in Figure 3.17b. Subsequent accesses to that same page can then be performed locally by the client. If another client later wants to access the same page, the page is migrated again. Since a given page is at a single location at any given point in time, consistency is also ensured.

The previous architecture performs much better than a centralized one since it allows most accesses to be executed locally, on the client's machine. However, it does not allow two clients to access the same page at the same time. The page has always to be migrated before it can be accessed by another site. Given that reads are often much more common than writes, and given that many applications rely on shared structures that can be read in parallel by many different threads, it is wise to allow several identical copies of the same page to be replicated in the system. The variant illustrated in Figure 3.17c has read-only replicas that are read concurrently. Each time a client requests, a new replica is created. Write requests are coordinated by the server. Naturally, if several replicas of the same page exist, one has to guarantee that they behave like a single page. Two approaches can be used to ensure that replicas are kept with similar contents upon a write: one is to propagate the update to all replicas; the other is to invalidate outdated copies and keep a single copy updated. Figure 3.17d depicts the full replication approach, where read-write page replicas can be held in several clients. Clients can perform competitive access requests, which are coordinated by a sequencer. The interleave of local operations and commands from the sequencer depends on the DSM consistency semantics. The sequencer function can be distributed, for example performed by a decentralized protocol run by all concerned clients.

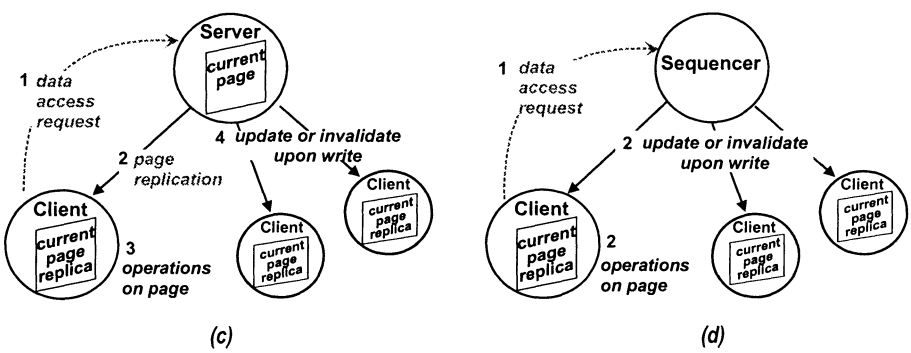

Figure 3.17 *(continued)*
DSM arch.: (c) Read-only Replication; (d) Read-Write Replication

3.8.2 Securing DSM Consistency

A possible strategy to implement distributed shared memory support is as follows. At the initialization time, each page is located at a single node in the system. That node is called the *owner* of the page. When another node wants to write to the page, it must first obtain control of the page. This requires the implementation of some form of location service, from which the client can obtain the current owner of a given page. The location service is, by itself, a complex component. If a centralized server is used, then it may become a bottleneck in the whole DSM operation. So, decentralized versions should be used (for instance, making each node responsible for keeping track of the location of a subset of the pages). After locating the current owner, the client requests the migration of the page, and becomes the new owner of that page. Since the client wants to write to the page, there is no point in keeping a copy of the page at the previous owner (this copy would become obsolete after the write).

Assume now that another node wants to read the same page. As before, the node must first find the current owner of the page and then request a copy of the page. If the owner is not actively writing to the page, it replicates the page in the reader's memory. To make sure replica consistency is enforced, both copies of the page become write protected. Thus when one of the nodes tries to update the page, an exception is generated and the remaining copies are invalidated before the writer becomes the owner of the unique valid copy of the page.

Consider now that a page is read very often by a large number of nodes but written very infrequently. Following to the steps described in the previous paragraphs, immediately after an update the writer is the owner of the single copy of the page. When another node issues a read, the page is replicated on that node's memory. Since many nodes read the page, many copies are created on demand, one-by-one, and the same page crosses the network several times. In those cases, if support for multicast is provided at the network level, it may be more efficient to send the update to all nodes as soon as the writer finishes is job. This not only saves duplicate transfers but also ensures that readers find a valid copy of the page in their caches when the read is issued (and are not forced to wait for the page to be transferred). This approach, illustrated back in Figure 3.16, is known as *eager update*. The effectiveness of the eager update approach greatly depends on the data access pattern. If many readers are able to benefit from a single multicast update, the eager approach is effective. On the other hand, if the pages are updated frequently, a new update might have to be disseminated before the previous update is read. In such case, multicasts represent a non-productive waste of network resources.

Additionally, in order for the eager approach to be used effectively, the DSM system has to have a mechanism to detect that the program "has finished" a batch of updates to a given page. Actually, this may be extremely hard or even impossible to determine, unless the programmer itself explicitly adds to the application code directives that define the boundaries of access to shared

data. For instance, some models require data accesses to be bounded by special synchronization primitives, called respectively, *acquire* and *release*. When a process issues an *acquire* request, it informs the system that it needs to obtain an updated copy of the memory pages. When a process issues a *release*, it indicates that it has finished updating the shared memory. DSM systems that use this approach are often said to implement *weak* models of memory consistency, which are then labeled according to the strategy used for propagating the updates. For instance, if updates are disseminated when the *release* operation is issued as illustrated in Figure 3.18, the system is said to implement *release consistency*. If after a *release* the updates are only disseminated when another node performs an *acquire*, the system is said to implement *lazy release consistency* (see Figure 3.19).

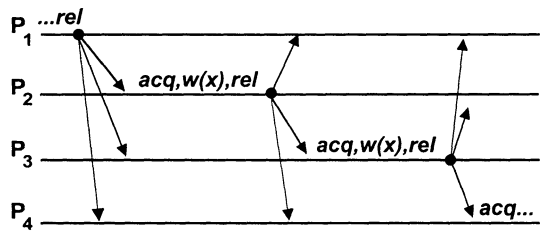

Figure 3.18. Release Consistency

Another way of bounding memory accesses is to use language constructs. For instance, in object-oriented programming languages, data is encapsulated and can only be accessed by invoking the object's methods. Also, if the object is shared by several threads of control, one usually has a synchronization mechanism associated with that object (for instance, the mutual exclusion lock associated with synchronized objects in Java). These language constructs can be used to automatically insert shared memory primitives in the application program, such as the acquire and release primitives described above, and to implement distributed shared memory in software.

It should be noted that although the previous models are often said to implement *weak consistency*, this label is a bit misleading, according to the definition we made in Chapter 2. In fact, if the programmer does not use synchronization primitives (such as the acquire and release) in a correct way, the memory will be weakly consistent indeed. However, the whole purpose of these mechanisms is to make distributed memory sequentially consistent for those programmers that use the synchronization primitives in a correct way. In some sense, these models can be better described as clever implementations of strong consistency, that use the application programmer knowledge about data access patterns to optimize the DSM implementation.

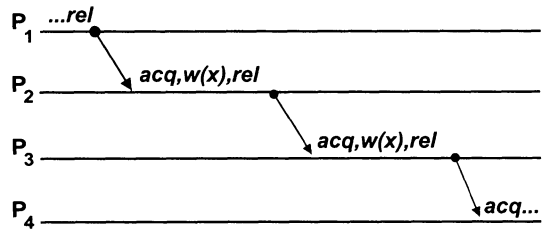

Figure 3.19. Lazy Release Consistency

3.8.3 Building DSM Systems

How can we build a system that makes use of distributed shared memory? In many different ways, depending on the programming language used and on the performance requirements. Let us discuss two different alternatives.

One way of using DSM is for implementing high-performance parallel applications. Consider for instance that one needs to implement an iterative algorithm that requires two large matrices to be multiplied a very large number of times (until some value converges). If several nodes are available to perform the computation, the work can be split by making each node responsible for computing a portion of the matrices. The parallel version of the program would work like this: the input matrices for round i are divided into n equal pieces (assuming that all nodes have equivalent processing power); each node computes its portion of the resulting matrix; when the full matrix is computed, if the algorithm has not converged, it is taken as input for round $i + 1$. Note that all threads have to synchronize at the end of each round, to make sure the computation is finished before starting a new round. DSM may be a viable paradigm to implement this sort of algorithm since it spares the application the burden of exchanging the pieces of the matrix explicitly. If a message-passing model were used, pieces of the matrix would have to be exchanged explicitly among nodes.

Note that similar algorithms have been coded for shared memory multiprocessors. What changes have to be made to multiprocessor code, in order to re-use it in a DSM environment? To start with, some interface is needed to specify where each thread should be launched. If a strong consistency model is supported by the DSM system, maybe nothing else is needed. It may happen however that the resulting performance on the parallel system turns out not to be as good as expected. To circumvent the performance problem, weaker memory consistency models may have to be used. However, in that case, the original code will have to be changed to introduce the synchronization primitives required to preserve memory consistency.

Distributed shared memory techniques can also be used to manage the consistency of object caches. Assume that an object in a client-server system is accessed by several clients simultaneously with a pattern where reads are much more frequent than writes. Instead of implementing all operations as remote

procedure calls, one can build smart client stubs that are able to synchronize directly among themselves to maintain consistent replicas of the object.

3.9 MESSAGE BUSES

A *message bus* is an abstraction that allows processes to exchange messages indirectly, through an intermediate component, called the *bus*. In this model, some processes called *publishers* produce messages to the bus, whereas other processes called *subscribers* consume messages from the bus. Most message buses allow a message to be produced at one moment but only be consumed later on. The message bus maintains the message in a non-volatile store until it is consumed. Message passing is a much lower level abstraction than the remote procedure call or the distributed shared memory abstractions described in previous sections. Given the availability of other semantically richer alternatives, what can be the appeal of a message bus?

To start with, the publish-subscribe paradigm is simple and easy to understand. It remains to be seen if this paradigm is powerful enough to build all sorts of complex distributed applications, but it is certainly a good tool to solve simple problems, even by the non-expert programmer. Additionally, it does not require the producer and the consumer to be active at the same time. Thus, it supports what is often called asynchronous, or better said, non-synchronized interaction. This type of support is particularly useful when the participants have low or sporadic connectivity. For instance, a producer can publish messages during the day and the system can propagate all these messages in batch at night, such that they are consumed in the next morning by a machine located in a remote office. This can be a clever way to make two components, located in different facilities, to interact without requiring permanent connectivity. Finally, since the application components are not directly coupled, but coupled via the bus instead, it is easier to reconfigure the system. One can change the number, identity or location of the subscribers without changing the publishers and vice-versa.

3.9.1 Message-bus Architecture

The abstract architecture of a message bus is very simple. One has publishers and subscribers connected to common bus, as illustrated by Figure 3.20. The bus can be *volatile*, i.e., messages cross the bus and can be consumed at that time and else vanish in the ether, much like radio broadcast, or *persistent*, in the case where messages are stored *in* the bus until consumed.

Two types of addressing schemes can be used in message buses. The most simple one allows messages to be deposited on *mailboxes*. The publisher specifies the name of the mailbox(es) where it wants to place the message and the subscriber specifies the name of the mailbox(es) from where it wants to collect messages. Mailboxes serve as repository of messages, and can have (and usually do) a maximum storage capacity. If the mailbox capacity is exceeded, the publisher may obtain an error or be requested to block when producing a

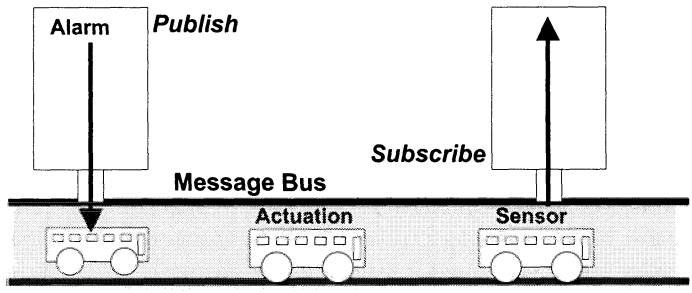

Figure 3.20. Message-bus Architecture

new message. The subscribers may have the choice of removing the messages from the mailboxes, or simply obtain a copy of the message, such that several subscribers can consume the same message.

An alternative addressing scheme, but also more difficult to implement, is to use *subject-based addressing*. When messages are published in the bus, they are labeled with a subject field, including one or several keywords. The subscriber registers its interest in receiving messages that contain one or more keywords in their subject fields. Efficient matching of subscriber requests with message subjects in a large message space can be a complex task.

There are similarities between the message bus abstraction and the Linda tuple-space (Carriero and Gelertner, 1986). The tuple space is a programming paradigm where threads communicate and are synchronized through a global shared repository of *tuples*. Processes can publish a tuple using an *out* primitive and consume tuples using an *in* primitive. Tuple consumption uses pattern matching on the contents of the tuple, what is also named *content-based addressing*.

3.9.2 Building Publisher-Subscriber Systems

To exemplify how the publisher-subscriber model can be implemented we use the simplest architecture of all, where the bus abstraction is materialized in a central server, as illustrated in Figure 3.21.

In this case, the publish activity is very simple. The publisher just sends a message to the server that stores it in non-volatile memory. Message subscription can be implemented using two different alternatives: the *push* strategy or the *pull* strategy. In the push strategy the subscribers just register with the server the interest in receiving a certain class of messages, and the server is responsible for disseminating these messages to the interested subscribers. Multicast communication can be used to disseminate messages in a efficient way, when many subscribers are interested in the same messages. In the pull strategy, it is up to the subscriber to contact the server periodically to fetch the messages. This second scheme may look less efficient, but has its advantages

in systems where the subscribers are not permanently connected and want to collect the messages in a deferred way (non-synchronized).

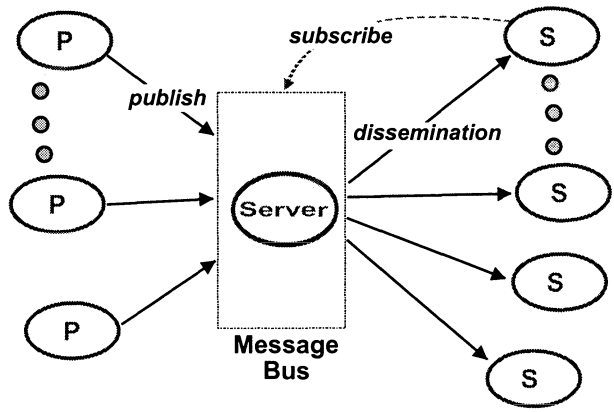

Figure 3.21. Implementing a Publish-Subscriber System

The problem with the architecture exemplified in Figure 3.21 is the centralized server, which is both a performance bottleneck and a single point of failure. By using replication, both performance and reliability can be improved (*see Event services* in Chapter 8).

3.10 SUMMARY AND FURTHER READING

In this chapter, the main distributed computing models were discussed. The objectives of this chapter were the following: to make clear to the system architect what are the main frameworks to build distributed systems; which strategies are best fit for the several problems requiring the assistance of distributed solutions; which models implement the several strategies. The discussion was lead in a problem-solving manner, and backtracking to the paradigms introduced in Chapter 2, whenever appropriate.

Following the original work of Birrell and Nelson (Birrell and Nelson, 1984), many systems were built using Remote Procedure Calls. A survey can be found in (Ananda et al., 1992). A discussion of different implementation alternatives is given in (Chung et al., 1989). A way to detect and terminate orphans is discussed in (Panzieri and Shrivastava, 1988).

The use of group communication to build distributed applications has many interesting examples. The V system (Cheriton and Zwaenepoel, 1985) was one of the first to use the process group approach. Birman (Birman and van Renesse, 1994) provides many examples of different group interaction styles. Systems that integrate the remote Procedure Call with replication and group communication are discussed in (Cooper, 1985; Ladin et al., 1992; Elnozahy and Zwaenepoel, 1992b; Wood, 1993; Rodrigues et al., 1994).

The work of Li and Hudak (Li and Hudak, 1989) addresses the tradeoffs involved in enforcing strong consistency shared memory with hardware support. Weaker memory models are proposed in (Goodman, 1989; Gharachorloo et al., 1990; Ahamad et al., 1991; Bershad and Zekauskas, 1991). Systems such as Munin (Carter et al., 1991) support different strategies to implement shared memory. Object oriented distributed shared memory models are described in (Bal and Tanenbaum, 1988; Guedes and Castro, 1993).

There are several sources of information on messaging systems, including the book of (Miller, 1999). A book that gives a good description of the underlying communication protocols is (Paul, 1998).

4 DISTRIBUTED SYSTEMS AND PLATFORMS

This chapter consolidates the matters discussed in the previous chapters, in the form of examples of enabling technologies, toolboxes, platforms and systems. Namely, we discuss: name and directory services; distributed file systems; the Distributed Computing Environment (DCE); object-oriented environments (CORBA); the World-Wide Web; groupware systems.

4.1 NAME AND DIRECTORY SERVICES

We recall that a *name service* is responsible for storing associations between *names* and *addresses*. More generally, the name service stores associations between names and *attributes*. We have also seen that the name service is usually provided by a set of cooperative name servers.

A naive implementation of a name service can be trivially derived using a centralized name server that keeps associations between names and attributes in main memory or in a file. However, such a simple solution does not address the true challenges of building a real-life name server, namely, the scalability and administrative issues related with the maintenance of a very large name space.

Scalability is a challenge, since the name service is used very often by a large number of applications. Administrative issues are also a challenge, because it is impractical and undesirable to have all the name servers managed by a single central entity. Instead, the management of a distributed name service is itself

P. Verissimo et al. *Distributed Systems for System Architects*

usually distributed, each organization being responsible for managing its own servers.

4.1.1 DNS

The Domain Name Server (DNS) is the main name service used in the Internet. The keys to the scalability of DNS are a hierarchical partitioning of the name service database, careful use of replication (also for higher availability) and intensive use of caching.

The name format used in the DNS is well know today, mainly because of the widespread use of the Internet and the World-Wide Web (WWW). DNS names consist of a sequence of *labels* separated by dots, such as for instance "`www.di.fc.ul.pt`". The name reflects the hierarchical structure of the DNS architecture. The domain "`pt`" is a top-level national domain, in this case Portugal. Similar top level domains exist for most countries connected to the Internet. Other top-level domains, include "`com`" for commercial organizations, "`edu`" for educational organizations, "`gov`" for governmental agencies, and some others. Below the "`pt`" domain, is the "`ul`" domain, which stands for University of Lisboa, divided in several faculties. The next domains are "`fc`" for Faculty of Sciences and "`di`" for Department of Informatics, the faculty and department of the authors. Finally "`www`" is an alias for another name, the machine that holds the Department's Web server. This hierarchical structure is illustrated in Figure 4.1.

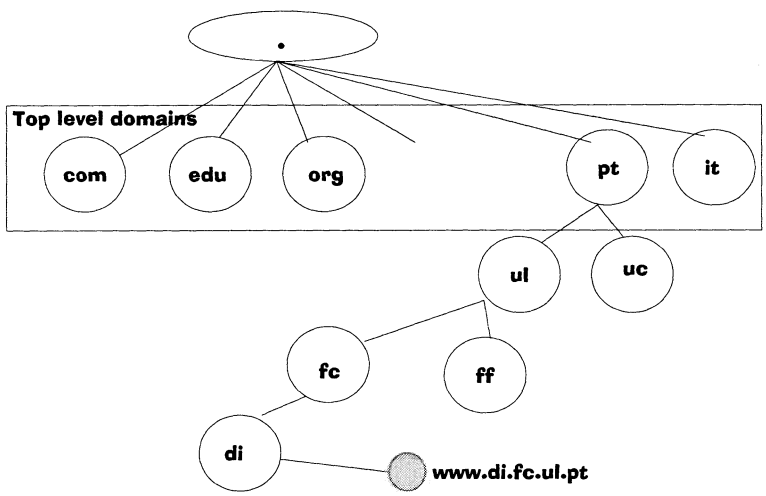

Figure 4.1. Hierarchical DNS Name Space

Although in the previous example there is some coincidence between logical a geographical proximity, this is not mandated by the DNS and it is not even a general rule. The machines from a given domain can be spread through many different geographical locations, such as, for instance, the sub-domains of the

```
$ORIGIN fc.ul.pt.
        86400           IN      NS      dns.di.fc.ul.pt.
        86400           IN      MX      10 mail.di.fc.ul.pt.
$ORIGIN di.fc.ul.pt.
titanic 86400           IN      CNAME   mail.di.fc.ul.pt.
navigators      86400   IN      CNAME   navserver.di.fc.ul.pt.
ORIGIN navigators.di.fc.ul.pt.
www     86400           IN      CNAME   formiga.di.fc.ul.pt.
```

Figure 4.2. DNS Configuration File

"com" (a.k.a. dotcom) top-level domain that can be placed anywhere in the world.

When mapping the logical structure onto physical name servers, the DNS does not enforce a one-to-one mapping. For load balancing and higher availability, more than one server may manage the data for a single domain, and several small domains can be managed by the same server. The mapping is performed using the notion of *zones*, subsets of the address space that are managed by a server. Several servers may hold information about the same zone, some of these are designated *authoritative*, i.e, they are the source of the most up-to-date bindings for names in that zone.

The purpose of the DNS name servers is to maintain information about names and to provide this information to clients in response to DNS queries. The most common DNS queries are host name resolutions and mail host location queries. The first type of request is used to obtain the IP address of a machine given its name. The second type of request returns the IP addresses of machines willing to accept mail for a given domain. Less used queries are reverse address resolutions (obtain the name given the IP address). Name servers obtain the information required to answer these queries by reading a configuration file, whose format is illustrated in Figure 4.2 (some lines were deleted on purpose).

In run-time, the name servers are accessed though a *resolver*, a function provided as a library and linked with the application code. The resolver is responsible for contacting one or more servers in order to resolve a name. The list of name servers to be contacted, sorted by order of preference, is configured in a file of the client's machine. The servers themselves can be classified in primary servers, secondary servers and caching-only servers, according to their role in the hierarchy.

The consistency of the information stored in the DNS configuration files is, of course, crucial for the correct operation of the Internet. It is easy to create bugs by incorrectly filling-in these files, such as omitting trailing dots in domain names, use of invalid characters in IP addresses, missing fields in records, etc. (Beertema, 1993). Some tools can help the system administrators verify the contents of DNS information (Romão, 1994).

4.1.2 GNS

One characteristic of the DNS is that it is extremely difficult to reorganize the logical hierarchy. All names are either local (when just the machine name is given) or absolute names. There is no way of moving an organization' s name space from one domain to another. For instance, if Universidade of Lisboa was to be registered under the "edu" domain, all the absolute names would have to be changed (the department's Web server would have to be named "`www.di.fc.ul.edu`") and the previous names would become invalid.

The Global Name Service (GNS), developed at the DEC Systems Research Center (now owned by Compaq) was designed to accommodate change. Therefore, it includes mechanisms to support the name space re-organization. In the GNS, the root name of each organization is assigned a globally valid unique identifier. Each name must explicitly carry the unique identifier of the organization's root, which remains valid even if the location of the root in the global tree is changed.

The root directory of the global hierarchy must hold the location of all organization roots. Thus, according to the GNS architecture, currently the Universidade de Lisboa root would have an entry in the root directory, placing it under "`PT/UL`", as illustrated in Figure 4.3. In our example, the name of our Web server could be "`⟨#456/FC/DI/,WWW⟩`". If our name space would move under "`EDU`", this entry would be changed to place it under the new domain. In its previous location, a forwarding pointer would redirect all requests using (outdated) full pathnames.

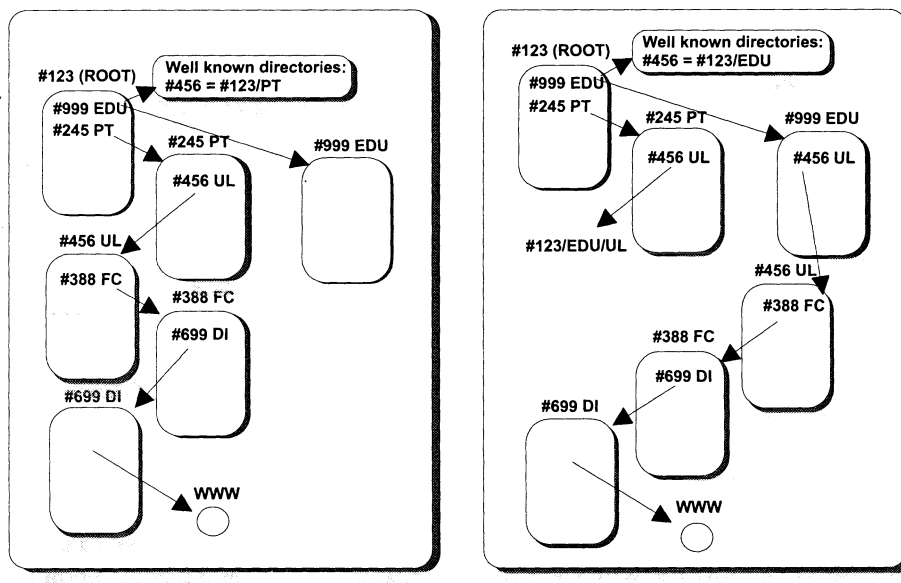

Figure 4.3. Hierarchical GNS Name Space

4.1.3 X.500

The X.500 Directory Service is a CCITT and ISO standard that provides a more general service than the name servers previously described. X.500 allows names to be associated with arbitrary attributes and supports queries based on combinations of attributes. For instance, the directory service could contain information about the faculty and students, with attributes such as research areas, hobbies, contact information, etc. A query to the directory service could ask for students interested in "neural networks" and "routing algorithms" or faculty members whose hobby is "sailing". Naturally, performing this sort of queries in large portions of the name space can be extremely expensive.

The X.500 Directory Service structure is depicted in Figure 4.4. The tree is named *Directory Information Tree (DIT)*, and it connects *Directory System Agents (DSA)* hierarchically. We see that each of these agents controls a dashed part which is a portion of the whole tree. The local information under the agents' control resides in the *Directory Information Bases (DIB)* located with each of them. To withstand large scale, but at the same time avoid inconsistencies, the name must both be *composite* and *unique*. A complete X.500 name is called *Distinguished Name (DN)*, and it is an ordered sequence of *Relative Distinguished Names (RDN)*. An RDN is an unordered sequence of *attributes* with well-defined *types*. Besides this structure, attributes of names that have specific scopes should be defined from well-known type descriptions, if possible standards. For example, country attribute types (i.e., country names) should be chosen from the two-digit country code standard ISO 3166 (FR-France, PT-Portugal, USA-United States of America,etc.). Names are generally of the form:

Attributes:
C (country); O (organization); U (org. unit); L (location); CN (common name)
Relative DNs:
C=PT, O=ULisboa; U=FacSciences; L=DptInformatics; CN=Luís
DN of Luís at the U.L.:
⟨*C=PT/O=ULisboa/U=FacSciences/L=DptInformatics/CN=Luís*⟩

The name scheme of X.500 is extremely powerful, but for that reason it sometimes looks a bit user unfriendly. However, DNs can be represented in a much more intuitive way, similar to DNS names, if attributes are hidden, leaving only the types. Luís, in the example above, would become: ⟨*PT,ULisboa,FacSciences,DptInformatics,Luís*⟩.

Distinguished Names (DNs) can be organized in a global tree, if they maintain the desired attributes that we postulated earlier on: being composite and unique. Let us look at a possible distinguished name tree, shown in Figure 4.5. The tree shows a hierarchy of university name structures, in two countries, representing several faculties (or colleges). This tree only makes sense if it is well-formed, which happens if the following rules are followed: RDNs are assigned by hierarchically organized agents; and each agent ensures that all names it assigns are unambiguous and unique in the realm under its control. That organization relies on the X.500 Directory Service structure depicted in

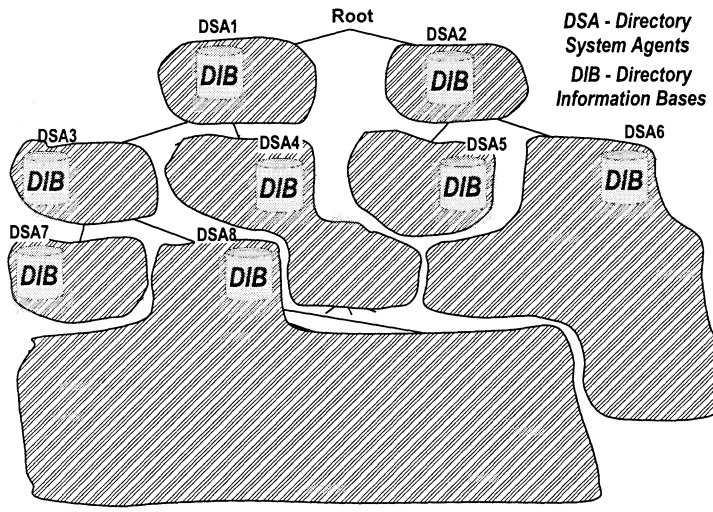

Figure 4.4. An X.500 Directory Structure

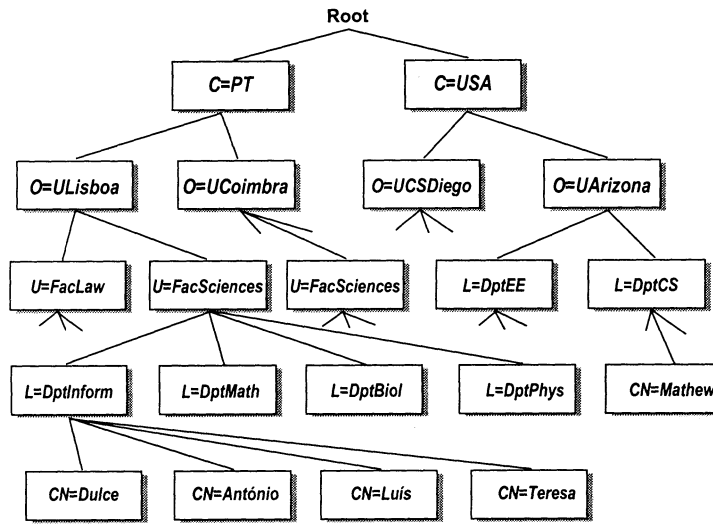

Figure 4.5. An X.500 Distinguished Name Tree

Figure 4.4. Of course, it would make sense for the distinguished name tree presented in Figure 4.5 to have been created under the auspices of the structure of Figure 4.4: we can observe the complete picture in Figure 4.6.

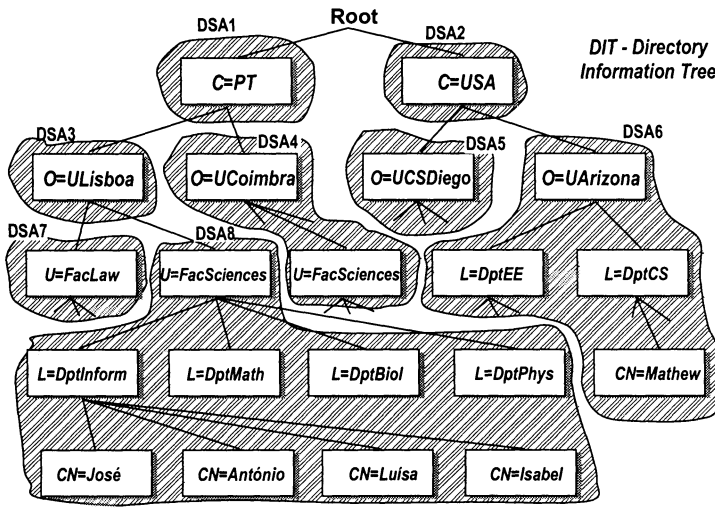

Figure 4.6. Name Tree under the X.500 Directory Service

4.2 DISTRIBUTED FILE SYSTEMS

Distributed file systems try to offer the same services as centralized file systems do. Thus, before going in the details of how to build the distributed version let we do a brief review of some elementary file systems concepts.

The main purpose of a file system is to provide support for storage of data on a non-volatile medium, typically a hard-disk. Data is stored on *files*, which have one or more names and other attributes, such as protection information, dates of last access and update, size, etc. Thus, every file system has, implicitly or explicitly, a directory service attached. In fact, a file system can be described as a two-layer architecture, as illustrated in Figure 4.7. At the bottom layer, we have a flat-file service, where each file is identified by some *unique file identifier*. The layer on top implements the directory service, storing the associations between textual names and unique file identifiers. On many file systems, the directory service data is stored on one or more files of the flat file system.

It is also important to recall how programs use a file system. In order to access a file, programs must first *open* it. When opening a file, the application specifies the file name and the desired access mode (for read, write or both). The file system uses the directory service to obtain the unique file identifier, checks if the user has the rights to perform the desired operation on the file and, in case of success, returns a *file handle* to the application. The handle acts as a *capability*: its possession entitles the owner to access the file (capabilities will be discussed in detail in Chapter 18). In order to *read* or *write* the file, the application makes a request containing: the handle, a pointer to the buffer where the data should be copied to/from, and the amount of data to be transferred. Typically, the system internally keeps a *file pointer*, the position within the file where the next read/write operation is going to take place. Finally, when the application

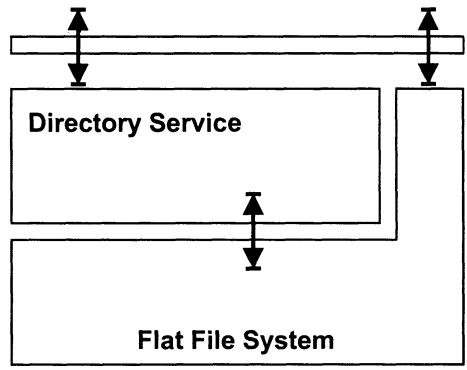

Figure 4.7. File System Architecture

is done with the file it should *close* it. Other functions commonly found in the file system interface are functions to set the file pointer to a given absolute or relative position, to truncate the file, to delete the file, to read the file attributes and to manipulate directory information (for instance, rename the file). Finally, it is worth mentioning that in order to provide good performance, centralized file systems rely on caching algorithms to minimize I/O operations and keep in-use data in main memory.

What does it mean to distribute a file system? To start with, a distributed file system allows data to be stored on a given machine and to be accessed from other machines. In order to do so, the system can be configured using a client-server model: files are stored on a server and accessed by many different remote clients. The server part can itself be distributed, i.e., instead of forcing all the files to be in the same server, different portions of the file system can be stored on different servers. This makes the system scalable, since more storage and processing can be added to the system by plugging additional servers.

Although the general idea of implementing a distributed file system based on a client server model is quite simple, many interesting questions need to be answered in order to develop an efficient implementation. How does a client discover which server stores the file it is looking for? Should the directory service be also distributed, or centralized instead? How many bytes should be sent at once from the server to the client? Should the cache be located only in the server, only in the client or in both? If more than one client are allowed to cache the same file, how is cache consistency maintained? Should the server keep information about which clients did open a given file?

Naturally, there is not a single answer to all these problems. Figure 4.8 illustrates two extreme alternative designs. In Figure 4.8a the client uses a download/upload model, retrieving and storing the whole file from/to the file server. In the model of Figure 4.8b clients cache file blocks that they request from the server, and contact the latter to validate their caches and get new blocks. Files remain remote on the server. In the next few paragraphs we

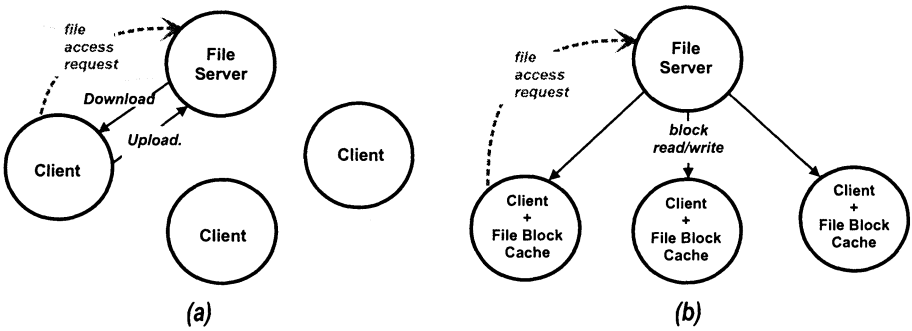

Figure 4.8. Distributed File Models: (a) Download-Upload; (b) Remote Access

are going to briefly review some successful systems, that have answered the previous questions in different ways.

4.2.1 NFS

The Network File System (NFS), developed by Sun Microsystems in 1984, was the first commercial distributed file system. It is a striking example of good distributed systems design.

The NFS is built using a client-server approach. Servers store files and respond to requests from remote applications that need to access those files. However, the applications are not requested to contact the server directly. Instead, file system calls from the application are redirected to a proxy of the remote service that executes on the client's machine. This proxy is simply called the *NFS client*, and it takes care of the interaction with the remote server. This interaction is performed using SUN's remote procedure call service. The design decision of using RPC to access the server, along with the strategic option of making the server's interface public, was one of the reasons that made NFS so popular. Since the interfaces and the format of the associated messages were known, it was possible for different companies to develop NFS components for many different architectures and operating systems.

The use of the client-server approach is made transparent to the client application through the addition of a Virtual File System (VFS) layer to the Unix kernel. The layer introduces an additional level of indirection in the file system calls. Instead of calling a specific file system primitive, the application calls the VFS interface that, in turn, calls the NFS client primitives. The VFS allows the kernel to support several file systems simultaneously, including the NFS and the native file system.

The NFS server uses a *stateless* approach, i.e., servers do not keep any state about open files. Also, servers keep no information about the number and state of their clients. This means that each request must be self-contained, *i.e.*, the server must be able to process the request just looking at its parameters, without any knowledge from past requests. Since the server remembers nothing,

it does not lose any relevant information when it crashes. Thus, a stateless approach has some advantages from the fault tolerance point of view: a server may crash, recover, pick a new request and continue as if nothing happened (we postpone a further discussion about fault tolerance issues to the next part of the book). On the other hand, since the server does not keep state, it is unable to check if a file is being accessed by one or more clients. This, as we will see, makes the task of preserving the Unix semantics of file sharing impossible. Due to this reason, NFS does not support the *open* primitive (open semantics require the system to "remember" that the file has been opened). Instead, a *lookup* primitive, that provides a handle for a file name is provided.

A partial list of the NFS server interface is presented in Table 4.1. The reader will notice that *read* and *write* calls include an *offset* parameter. This is required because the server, being stateless, does not store the file pointer on behalf of the client. Instead, the file pointer must be stored by the NFS client and sent explicitly on read and write calls. The *cookie* parameter in the `readdir` call plays a similar role, allowing a client to read a large directory in pieces (the cookie points to the next directory entry to be read). The *lookup* primitive returns a *file handle* that consists of a pair: a unique *file system* identifier, and a unique *file* identifier. The unique file identifier is made of the Unix inode[1] of the file and a generation number, which is incremented whenever the inode is re-used. This ensures that identifiers are not reusable, and that the identifier of a new file cannot be confused with the identifier of a previous file on the same file system.

Table 4.1. NFS Interface (partial)

Name (parameters)	Returns
lookup (dirfh, name)	(fh, attr)
create (dirfh, name , attr)	(newfh, attr)
remove (dirfh, name)	(status)
read (fh, offset, count)	(attr, data)
write (fh, offset, count, data)	(attr)
mkdir (dirfh, name, attr)	(newfh, attr)
readdir (dirfh, cookie, count)	(direntries)

Consider now an application that reads a file, reading only one byte at a time. If a remote procedure call was to be performed for each byte read, the performance of the system would be unacceptable. This problem is not specific of distributed systems, in a centralized system one also avoids performing an I/O operation for each byte read by caching one or more file blocks in main

[1] In the Unix file system, files are uniquely identified by a index node, or simply, inode.

memory. In the NFS a similar approach can be followed, by caching the block both in the server's memory and in the client's memory. Unfortunately, having the same data cached in different machines introduces the problem of cache coherence. In a centralized system, the cache is physically shared by all processes. Since a single copy of the cached data exists, the sharing semantics of a centralized Unix file system is the following: if a process does a write, the results become immediately visible to all other processes. Clearly this is very expensive to obtain in a distributed system, since propagating an update requires at least one remote procedure call. The NFS approach implements a weaker consistency model that tries to balance consistency with performance requirements.

When reading a file, the NFS client reads a complete disk block (which is 8 kilobytes in the Unix BSD 4.2 Fast File System). The block is cached in the client's memory and is considered valid for some amount of time (typically, 3 seconds for files). Thus, subsequent reads that fall into the same block do not require a remote access to the server. After this period, a new access must first check with the server if the cache is still valid. For this purpose, the client also remembers the "version" it has cached, more precisely, the last time the data has been updated in the server. If the cache is still valid, it is assumed valid for another 3 seconds period; if not, the new version of the block is fetched again from the server.

Writes are executed in a similar manner. Instead of contacting the server each time a byte is written, the client caches all the writes for some time. The cache has to be flushed if the file is closed or if a *sync* call is performed by the application. Otherwise, updates are sent asynchronously to the server, using periods of low activity in the client. This task is performed by a Unix *daemon*, called the *block io daemon* (or simply the *bio-daemon*). The daemon can also try to optimize reads by performing *read-ahead*, i.e., requesting in anticipation the next block of a file being read by an application. To ensure that writes are guaranteed to be stored on disk when the remote procedure call returns, the cache on the server operates in *write-through* mode, *i.e.*, writes are immediately forced to disk.

So far, we have presented the interaction between a client and a server. We have not discussed how the client finds the appropriate server in the first place. The name of the servers storing remote files is configured at each client using an extension to the Unix *mount* facility. The mount mechanism allows a file system to be "attached" to another file system at a given directory, called the mount point. The NFS mount procedure allows a sub-tree of the server's file system to be mounted on a specific directory of the client machine (if the client has no disk, it can be mounted on the root directory of the client machine). When performing the mount operation the client contacts the server, which checks access rights and, in case of success, returns to the client a file handle to the mounted directory. If when translating the textual file name into a file handle, the NFS client traverses a mount point, it uses the file handle returned by the mount operation to perform the subsequent lookups.

4.2.2 AFS

The Andrew File System (AFS) was originally developed at Carnegie-Mellon University and later became a product of Transarc. The major design goal of AFS was to achieve scalability in terms of number of clients. Many of the design decisions behind the development of AFS aimed at overcoming known limitations of previous systems. For instance, with regard to the NFS file system described above, it was observed that the cache validation procedure, where clients inquire the servers about the validity of cached data, could easily overload the server with too many requests. Furthermore, it was noticed that most of these requests were unnecessary, given that the majority of files are not shared and thus, caches are usually valid.

To support their design decisions, the AFS development team made extensive measurements of file usage on their academic environment. The observations made at that time and in those environments have shown interesting facts: files were usually small; reads were much more common than writes and typically sequential; most files were updated by a single user; and when one file was accessed it was likely to be accessed again in the near future.

To address these access patterns, and assuming that local disks were available at client machines, a file system based on *whole file caching* was proposed, i.e., in AFS clients cache complete files (this approach has later been relaxed to accommodate very large files, by supporting caching of file portions). Once a file is cached, all read and write accesses are purely local and require no synchronization with the server. Once closed, the file remains in cache. When re-opened, the local cache is used whenever possible.

Enough disk space should be reserved in the clients' cache to hold the files needed for the typical operation of most applications. A daemon process in the client, called *Venus* is responsible for managing the AFS cache and for transferring files from and back to the server. The counterpart of Venus on the server is called *Vice*. To the clients, the file system appears seamless, though some files are local and others shared through Venus. This architecture is illustrated in Figure 4.9.

AFS supports read-write files accessed in competition by clients, but a RW file can have several read-only copies hosted at more than one Vice. Whenever the master file is updated, a *release* command makes sure that the RO copies are also updated. The above characteristics make AFS well-suited for a few classes of applications over file systems:

- shared read-only repositories, with occasional updates (e.g., news, price lists)– typically using RO copies for wider availability to many readers, single-client updates;
- shared read-write repositories, with infrequent updates (e.g., cooperative editing)– competitive few-writer activity, local caches remain valid for long;
- non-shared repositories (e.g., personal files)– single-writer activity, local cache remains valid wherever user is.

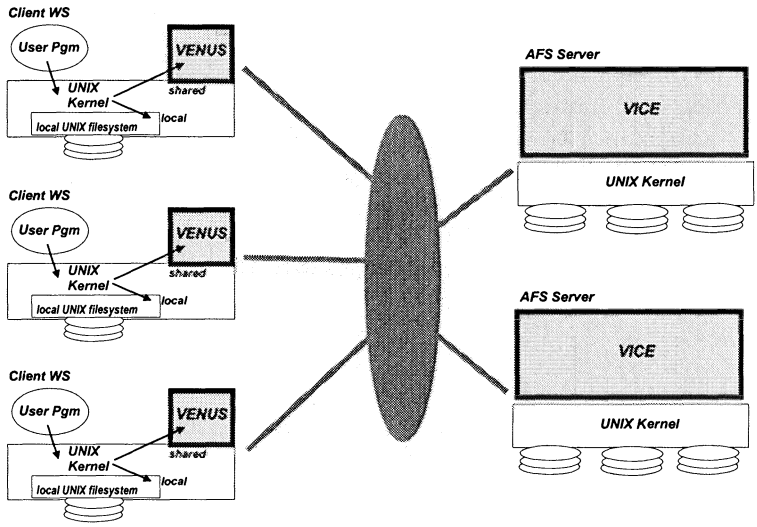

Figure 4.9. AFS Architecture

The goal of whole file caching is to relieve the server from unnecessary load. But, if the client should try to limit the interaction with a server to a minimum, how can it check the validity of data in its cache? The AFS approach consists in delegating on the server the responsibility for invalidating the client's cache when some other client updates the same file (actually, this optimization was only implemented in the second version of AFS).

When Vice gives Venus a copy of a file, it also makes a *callback promise*, or CBP. A CBP remains valid until hearing otherwise from Vice. When a file is changed at Vice, it calls back all clients holding valid CBPs for that file, so that they cancel the CBP, which invalidates the file cache. When Venus opens a file, it analyzes the cache. If the file is not in cache, it requests a copy to Vice. If the file is in cache, it checks the CBP. If the CBP is valid, it opens the local copy; if not, it requests a current copy to Vice.

The reader will note that there is a window during which a local copy may be opened that is not the most recent one[2]. In order to reduce this risk, an expiration mechanism exists that supersedes the algorithm described above: a file is only opened locally if it is less than T since the local Venus has last heard from Vice concerning this file. That is, if a file is opened after T, the latest version is downloaded from Vice. A typical value for T is 10 minutes. This mechanism also recovers from the loss of callback messages.

If the client crashes it may miss one or several callbacks from the server. Thus, when a client recovers it has to contact the server and determine the

[2]Erlier versions of AFS checked directly with Vice before opening, instead of the CBP, but this did not scale well.

status of all the files it holds, by checking the timestamps of the relevant CBPs with the file information on the server. To prevent the server from storing callback information indefinitely, access rights (also called the *file tokens*) are only valid for some limited period.

The semantics of AFS is not exactly one-copy. When more than one client open a file concurrently, the server will hold the state of the last file to be closed. This form of consistency is however adequate for the example classes we have enumerated earlier. Furthermore, applications can always superimpose their synchronization on top of these basic mechanisms.

4.2.3 Coda

The Coda file system is a follow-up of the AFS project at CMU lead by some members of the AFS development team. The main goals of Coda were to improve reliability and availability vis-a-vis partitioning, and to support nomadic and mobile computing. This was achieved by *whole volume replication*, and by *disconnected operation*. Coda supports the use of portable computers as file system clients, and tries to offer what the authors call *constant data availability*.

Coda can be in one of three states (Figure 4.10). The whole file caching approach of AFS allows the client to cache in his local disk the files that he will need while disconnected. Caching files that are going to be needed in the future is called *hoarding*. Manual hoarding is possible but the authors have studied techniques to automate the task of selecting which files to cache. When the portable computer disconnects from the network, Coda is in the *emulation* phase: the user can work on the files cached in the local disk.

The servers containing replicas of a file form its *volume storage group* or VSG. Often, only part of the replicas are available (partitioning, disconnection), the *available VSG* or AVSG. Opening a file consists of reading it from one of the AVSG replicas and caching it locally. When the file is closed, it remains valid at the client, and a copy (reconciliation) is made to all AVSG servers.

Naturally, while disconnected the client is unable to receive any callbacks from the servers, and this presents an opportunity for conflicting updates on the same file. When the portable is later reconnected to the network, an automatic file system *reintegration* procedure is used, as illustrated in Figure 4.10. The procedure compares the versions of the client files with those of the server files and checks for conflicts. When no conflicts are found, the system automatically reconciles both versions of the file system. When conflicts are found, two versions of the conflicting files are stored and manual intervention of the user is requested.

4.3 DISTRIBUTED COMPUTING ENVIRONMENT (DCE)

In this part of the book we have referred to a number of technologies that help in the design and implementation of distributed applications and systems. Examples of these technologies are remote procedure call services, directory services, time services, distributed file systems, security systems, etc. For each

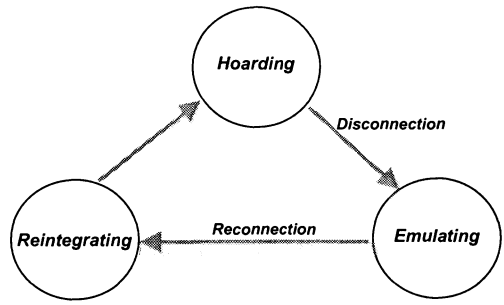

Figure 4.10. Coda Operation

of these services, several commercial products have emerged on the market. The diversity of technologies, often incompatible, makes the task of integrating applications made by different developers extremely hard.

The Distributed Computing Environment (DCE) is a standard endorsed by a consortium of several companies, including major players such as IBM, former DEC, and Hewlett Packard, under the name of Open System Foundation (OSF). DCE selects a particular technology for each of the services previously listed and offers them in an integrated package. The package was initially supported on Unix but was later ported to other operating systems as well.

Conceptually, there is not much really exciting in DCE, even though some of its services represent excellent technical solutions. This does not come as a surprise since DCE adopted standards and technologies that had already proven their value, some of which are addressed in this book. For instance, the directory service is based on X.500, the file system is derived from AFS, the time service is derived from NTP. The merit of DCE is to provide all these technologies as a coherent set.

Another feature that makes it hard to describe DCE in a few lines, is that it contains components that operate at different levels of abstraction. DCE is independent of the platform and operating system, thus these two layers are somehow outside of the scope of the DCE package. On top of the native operating system, DCE adds a dedicated threads interface. The availability of threads was felt to be a fundamental requirement for the efficient implementation of distributed applications (in particular servers), so DCE includes its own thread package. The *thread package*, like most of DCE components, has plenty of options and operational modes, enough to make almost everybody happy. For instance, three different scheduling policies are supported, priority based, round robin within the same priority and time-sliced round-robin; three types of mutexes are available; and so on.

Using the operating system (augmented with the DCE thread interface), the DCE *remote procedure call* package is implemented. The main computational model supported by DCE is client-server and RPC is a fundamental building block for the remaining services. Like almost every RPC package, DCE RPC allows server interfaces to be written in an Interface Definition Language and

provides the compiler to automatically generate client and server stubs from these interfaces. Each service is identified by a unique identifier. To help programmers to obtain unique identifiers that are really unique, a unique identifier generator is also provided (which encodes the location and date of generation). The RPC package provides optional authentication and encryption (*see Secure Client-Server with RPC* in Chapter 18). On top of the DCE RPC service, the *time service*, the *directory service* and the *security* service are implemented.

The Distributed Time Service (DTS) is an evolution of NTP (*see Network Time Protocol* in Chapter 14). Its role is, of course, to keep local clocks synchronized. The service is of paramount importance for many other services. Among other applications, the global notion of time is used by the file system to timestamp updates and compare file versions. It is also used by the security service to check the validity of a credential. An interesting feature of the DCE time service is that the user is informed of the actual accuracy of the value provided. DTS uses this information when comparing two dates, to check that the timestamping error is small enough that they are comparable.

The directory service (names and structure are inspired by X.500, *see X.500* in this chapter) is organized as a set of cooperative *cells*. Each cell manages its own name space and has a local Cell Directory Server (CDS). To "glue" different cells, two global directory servers can be used: the DCE Global Directory Server or the Internet DNS. Cell directory servers interact with the global service to a Global Directory Agent, that shields the CDS from the details of the GDS or DNS.

The security server of DCE is based on the Kerberos *security server* (*see Kerberos* in Chapter 19). It manages access rights based on Access Control Lists and implements authenticated RPCs.

Finally, we can find the DCE file system, called the Distributed File System (DFS). It contains two main components: a local component, called *Episode*, and a global component based on AFS (*see AFS* in this chapter). In interesting feature of the DFS is that the file naming service is integrated with the directory service CDS, so files can be relocated just by updating directory data.

4.4 OBJECT-ORIENTED ENVIRONMENTS (CORBA)

In some sense, CORBA, the *Common Object Request Broker Architecture* is an object-oriented DCE. It has also been proposed by a consortium of major industry companies, the Object Management Group (OMG). Essentially, CORBA also follows a client server approach but at a higher level of abstraction. Instead of having client processes interacting with server processes, CORBA provides the basis for having objects interacting with other objects.

The state of a CORBA object is encapsulated by a well-defined interface. Like in an RPC system, object interfaces are written in an Interface Definition Language, in this case in CORBA IDL, whose grammar is a subset of C++. Following object-oriented principles, CORBA IDL supports inheritance, thus new interfaces can be built by extending previously defined interfaces.

OMG started by defining the architecture illustrated in Figure 4.11. The core of the architecture is the *Object Request Broker*, an abstraction that supports interaction among objects. The broker is responsible for making sure that an object can invoke another object, implementing the required protocols to send the requests and receive the replies. Of course, application programs need an interface to the broker in order to instantiate objects, create references to remote objects, issue object invocations and so on. The first CORBA 1.1 specification defined the CORBA IDL, the IDL mappings to common programming languages, and the application programming interfaces to the ORB. This allowed to develop application code that was more or less portable through ORBs from different vendors. The "catch" was that some vendors did include some non-standard features in their ORBs. These features were added to enhance the standard ORB functionality, but in practice, these proprietary enhancements prevented seamless portability. Another catch was that protocols and message formats were not part of the standard. The idea was to give room for each vendor to pick the most appropriate solution for their target market and architectures. The less positive aspect of that decision was that ORBs from different vendors did not inter-operate. This problem was eventually fixed with the release of CORBA 2.0, that defined a common protocol to be supported by every ORB, the Internet Inter-ORB Protocol, or simply IIOP (actually, to be more precise, IIOP is an implementation of a more *General Inter-ORB Protocol* (GIOP) over the TCP/IP protocol suite).

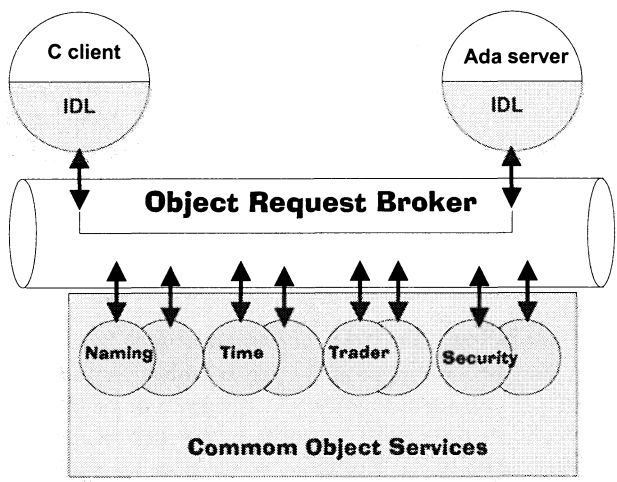

Figure 4.11. CORBA Architecture

If you have read the previous section on DCE, you already know that in order to build useful complex distributed applications you need more than remote invocations. CORBA has defined an extensive set of services, characterized by their standard CORBA IDL interfaces. No less than 15 services have been defined. We can describe some of the most relevant:

- *Naming Service:* as in any other system, it keeps associations among names and object references.
- *Persistence Service:* defines an interface to store the object state on *storage servers*, which can be implemented using traditional file systems or advanced database systems.
- *Concurrency Control Service:* provides a lock manager that can be used in the context of transactions to enforce concurrency control.
- *Transaction Service:* supports transactions, offering atomic commitment services.
- *Event Service:* allows some components to produce events that are distributed through an *event channel* to all interested *subscribers*.
- *Time Service:* provides a common time frame in the distributed system.
- *Trader Service:* allows objects to register their properties and clients to search for appropriate servers using this information.

Other services include the *Life Cycle Service*, the *Relationship Service*, the *Externalization Service*, the *Query Service*, the *Licensing Service*, and the *Collection Service*. In addition to these general purpose services, many interfaces have been standardized for specific business domains.

Of course, it is possible to build applications using just a few of these services. In fact, none of them is mandatory (but it is hard to build something useful without the naming service). The basic Corba functionality is pretty "conventional" when compared with an RPC system. The programmer writes the object interface in IDL. The IDL specification is compiled and a description of the interface is stored on the *Interface Repository*. From the IDL specification both client and server stubs are created for the target programming language (support for at least C++ and Java is now fairly common). The application programmer still has to write the actual object code, which is linked with the server stub and with an *Object Adapter*. The adapter supports the interface between the ORB and the object, providing the functionality to register the object within the ORB, to dispatch requests to the appropriate objects, and to send back replies.

The most straightforward way to activate an object is to execute it in the context of a dedicated process (this approach is called the *unshared server* approach). This process can be started when the system boots or just when an invocation is received by the ORB. As long as this process remains active, all invocations are forwarded to it. However, other policies can be implemented. For instance, it is possible to create a different process to execute each method. This approach, called the *server-per-method* approach is more suitable for stateless objects, where no shared state needs to be preserved on volatile memory across invocations.

Corba can be used to develop new applications from scratch. As with DCE, one advantage of using Corba is that many of the annoying details related with the implementation of RPC, server and client instantiation, etc, are handled by the ORB. An application built this way will be able to inter-operate

with any other application adhering to the Corba standards. Additionally, "*transformer*" objects can be built to wrap legacy applications, adding Corba-compliant interfaces to old code. This type of architecture is known as a "Corba 3-tier client-server architecture" and is illustrated in Figure 4.12. Using this 3-tier architecture and Corba IIOP, it is also possible to build powerful applications for the World-Wide Web, but this is the topic of our next section.

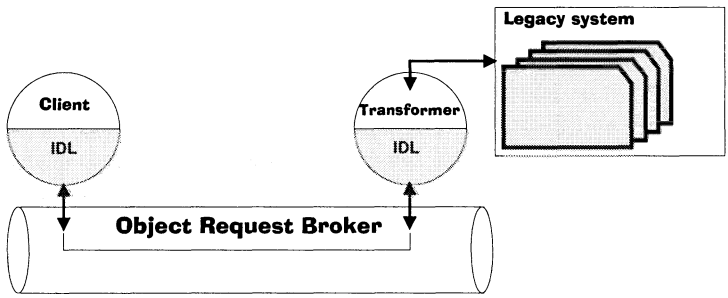

Figure 4.12. Corba 3-tier Architecture

4.5 WORLD-WIDE WEB

In the late 80's, despite the enormous potential and relative maturity of distributed systems technologies, it was felt among researchers that a "killer application" was lacking, one that could show the benefits of distributed systems in an obvious and indisputable way. The *World-Wide Web* (known simply by WWW or *the web*) was *the* killer application of the nineties.

It is interesting to observe that the application that had such a big impact in industry and society was in its inception relatively simple in terms of distributed systems concepts. It is basically a client-server application: clients, known as *browsers*, make requests to WWW servers in order to fetch documents and launch the execution of commands.

The WWW was created by Tim Berners-Lee and Robert Cailliau, working at CERN, with the goal of supporting information sharing among physics scientists. We can now say that the system was extremely successful in that task; it was the genesis of a global infrastructure that supports sharing and dissemination of information at a scale never seen before. The key for this success relies on the simplicity of its interface. Using a browser, remote information can be accessed just by clicking a button. Previously, cumbersome and often arcane sequences of commands had to be issued to achieve the same goal.

Documents in the WWW have a structured impure name called the *Uniform Resource Locator* (URL).
The URL has the following format: `<protocol>://<serveraddr>{/<path>}` where: the first field specifies which protocol must be used to interact with the server (several protocols are supported, being HTTP the most common); `serveraddr` is the address of the server to be contacted; and `path` indicates

which document should be fetched (if no name is specified, a document called `index.html` is read). Documents may be stored in several formats, from simple text to audio and video. Some of these formats are recognized and interpreted by the browser itself. Others are interpreted by companion applications that can launched by the browser in order to display the document.

Having a simple way to name and access documents is already a major contribution of the WWW architecture. However, if users were required to memorize the URL of all documents they were interested in, WWW would not have been such a big success. The other key factor of success was the use of *hypermedia* documents, which include *links* to other documents. The browser is able to interpret and display hypermedia documents in the *HyperText Markup Language* (HTML). Additionally, the browser allows the user to activate a link (typically, by clicking a mouse button) and automatically fetches the document whose URL is associated with the link. In this way, the user just has to remember the URL of the main page of an information repository or *site*, also known as the *home page*, which in turn holds the links for all other relevant documents (actually, most browsers have a way to store URLs, so that users do not even need to memorize the URLs of home pages). Today, it is a major business to provide pages, known as *portals* with links to useful information on the web, shops, advertising and much more. From a major portal, and just by clicking, the user is able to navigate through a huge net of documents, an often addicting activity also known as *surfing the web.*

The infrastructure we have just described is extremely useful and efficient, but it is somehow limited since it only supports the flow of information from the server to the client. Often, we also want the clients to send information to the server and request the execution of remote actions. For instance, the user may want to perform a query on a database, or issue an order when buying some goods. Thus, the first step to make the web more interactive was to allow clients to request the execution of programs in the servers. To support that type of interaction, servers were extended in several ways, the most common of which is an interface called the *Common Gateway Interface* (CGI). The CGI specifies how the browser indicates which programs should be executed and how parameters are sent to that program (and results sent back to the browser). According to this interface, WWW servers are able to launch programs upon request, which are executed as a separate process in the server machine. These programs can be binaries written in any programming language. However, CGIs are often interpreted programs written in popular script languages such as *perl.* CGIs can be fully-fledged applications or mere interfaces to other remote systems, such as database systems, forming a three-tier architecture as the one illustrated in Figure 4.13.

Typically, CGIs send results back to the clients in the form of HTML documents. Thus the CGI architecture allows to create web pages in run time. Pushed to the limit, this same concept allows to create a site where all the pages are created dynamically used information about format and contents stored in a database. The advantage of such system is that it simplifies the maintenance

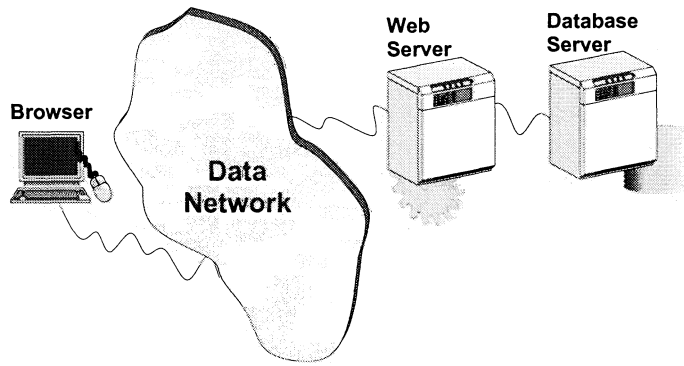

Figure 4.13. Three-tier WWW Architecture

and update of the pages, and allows pages to be customized according to the user profile.

Although the CGI concept is very versatile, it is not very efficient, because creating a new process is an expensive system operation. Often, the task performed by the CGI is extremely simple, for instance adding the time of day or the number of visitors to an otherwise static page. To avoid the need to start a new a process to execute these simple actions, most web servers offer extensions that allow some operations to be performed by the server itself, before sending the page to the client. The degree of complexity of the tasks performed by the server depends on the expressiveness of the language used to specify them. These extensions are known by several names, such as *Server Side Includes* (SSI), *Servlets* (which are Java applets executed by the server), or *active server pages* (Microsoft proprietary extensions). Some of these extensions support standard interfaces specially designed to support efficient access to database systems, such as Open DataBase Connectivity (ODBC) and Java DataBase Connectivity (JDBC)inxxJava DataBase Connectivity (JDBC).

We have just seen how to expand the processing capabilities of the server. We will now discuss how to expand the capabilities of the client browsers. It should be noted that today's browsers already have an impressive range of built-in facilities. However, it is impractical to include in the browser all the code to handle every possible format of multimedia documents. A way to make the browsers extensible is to allow new functionality to be attached dynamically to the browser. These extensions are known as *plugins*. Plugins have to be loaded onto the client machine and explicitly installed. To relieve the users from having to explicitly download and install plugins, the browser can be extended to accept code that is shipped together with a page.

One way to send code to the client is to send binaries, but this raises several problems. To start with, the server must have binaries for the different architectures where clients execute— an overwhelming task, given the myriad of architectures where browsers can run. This solution also poses problems to

the clients— accepting and executing binaries from someone which may be not fully trusted raises severe security concerns.

An alternative its to allow the client to execute just a small number of commands, which are included in the page in the form of a script. Using this approach, the browser has a built-in interpreter of the scripting language. When parsing a page, the browser looks for scripts and executes them. Since the script can only execute the commands recognized by the browser, this solution is much more secure than accepting arbitrary binary code. The problem with scripting languages is that they are generally limited in the range of functionality they provide.

The *Java* language emerged has a excellent tradeoff between pure binary code and simpler scripting languages. By embedding a Java virtual machine in the browser, one can insert Java code in a web page. Being a very powerful language, Java supports the development of complex applications. On the other hand, the Java virtual machine ensures that the downloaded code does not violate safety or security constraints. For example, the browser can download smart graphic interfaces that serve as front-ends to other applications (that are executed in the server) or even download fully-fledged applications to be executed locally.

The combination of client side and server side processing turns the WWW into an environment where a wide range of distributed applications can be executed.

4.6 GROUPWARE SYSTEMS

The purpose of a groupware system is to support collaboration activities, including support for document sharing and exchange, in various formats and through diverse protocols. Users can be located in different physical locations and access data using different types of devices, including portable computers. A key feature of successful groupware systems is the *integration* of different data sharing mechanisms, like databases, electronic mail services, shared editors, replication managers, etc.

A groupware system often plays the role of a central information repository, a place where all the information required for a business process is held and can be manipulated. Typically, a groupware system also includes a workflow engine that keeps track of the document life-cycle. For instance, if a product order must first be validated by the section manager, the system may ensure that the order is not issued before being approved. Additionally, the system may also be configured to raise an exception if the order is not processed by the manager within 24 hours.

A notable groupware system is Lotus Notes. The major component of Notes is a hypertext document database that can store and link data coming from several sources including web pages, mail messages, images, etc. Users can query for documents using a full text search engine. Versioning capabilities allow users to update documents and keep track of changes. It is possible to exchange documents that contain links to other documents. For instance, a

user can send a mail message to a set of colleagues and include a pointer to a working version of a document; by following that link, the colleagues will have access to the most updated version of the document.

Issues such as security, concurrency control, and replication management become particularly relevant in a context where sharing is the primary concern. Several security mechanisms are integrated in the Notes system, such as: mutual authentication between users and servers; powerful access control lists based on roles (like "author", "editor", "designer", among others); encryption of documents and communication channels; and the ability to sign documents.

Lotus Notes uses replication to place documents near their users. If an enterprise has several offices, it is possible to create replicas of a database in each site. Documents can be updated concurrently in different sites and the system supports automatic synchronization of replicas, using versioning control to detect which documents were changed and need to be copied. Clients can also replicate a portion of the database in their private machine, just to access data more efficiently or because they need to disconnect. Again, replica synchronization is performed when connectivity is regained.

An application programming interface allows using the basic Lotus mechanisms in the development of complex distributed applications. A set of development tools help programmers in this task. The tools include a dedicated scripting language (LotusScript), a formula language to build *filters* that can process documents, and design elements that ease the task of building graphical user-interfaces. As any other successful commercial product, Lotus also includes a myriad of interfaces to legacy and third-party database and systems, including, of course, the web.

Lotus can be seen as a good example of what is called *different-time-different-place* collaboration. This is probably the most frequent collaboration pattern. However, some other collaborative activities require a more tightly-coupled interaction, sometimes known as *same-time-different-place* or *synchronous interaction.* Tools for synchronous interaction include dissemination of audio and video, telepointers (a mechanisms where an user can point to a location in a document and the other users see the location pointed at), chat windows, shared drawing tools, etc. Such tools have the same sort of concurrency control and security requirements than the previous systems but exhibit much more stringent requirements in terms of connectivity.

4.7 SUMMARY AND FURTHER READING

This chapter gave examples of distributed systems and platforms. We started with the discussion of distributed name services and distributed file systems. Then, integrated platforms that include these and other services, such as DCE and CORBA, were presented. Finally, we briefly surveyed web and collaborative technologies.

More information on the implementation of DNS can be found in (Solomon et al., 1982; Bloom and Dunlap, 1986). An analysis of the DNS traffic is given in (Danzig et al., 1992). Alternative name server designs can be found

in (Cheriton and Mann, 1989; Guy et al., 1990). A survey by Satyanarayanan on distributed file systems can be found in (Mullender, 1993).

There are several very complete books on DCE and CORBA. The book of (Shirley et al., 1994) provides a good introduction to the several DCE components. Interesting books on Corba are (Baker, 1997) and (Henning and Vinoski, 1999). A comparison between COM and Corba in given in citePitchard:99. Naturally, a large number of books about the Internet and the WWW are available, including the good introductions of (Comer, 1997; Abrams, 1998). For a short description of many WWW technologies, see (Spainhour and Quercia, 1996) and for a more detailed treatment see (Deitel and Deitel, 2000). Many books on Lotus notes exist, such as (Lamb and Lew, 1996) and (Haberman et al., 2000).

Table 4.2 gives a few pointers to information about some of the systems described in this chapter. Some of the sites are extremely complete repositories of distributed systems related software.

Table 4.2. Pointers to Information about Distributed Systems and Platforms

System Class	*System*	*Pointers*
Internet	**IETF**	www.ietf.org
	RIPE	www.ripe.net
	ISOC	info.isoc.org
	CIX	www.cix.org
	ICANN	www.icann.org
	IANA	www.iana.org
	Internet2	www.internet2.edu
	Internic	www.internic.net
	NGI	www.ngi.gov
	ISC	www.isc.org
Networking	**Cisco**	www.cisco.com
	Lucent	www.lucent.com
	***x*-Kernel**	www.cs.arizona.edu/xkernel/
	Triad	www-dsg.stanford.edu/triad/index.html
	Spinglass	www.cs.cornell.edu/Info/Projects/Spinglass/index.html
WWW	**WWW**	www.w3.org
	Apache	www.apache.com
	Netscape	www.netscape.com
	MS-Explorer	www.microsoft.com
Parallel Computing and Shared Memory	**MPI**	www-unix.mcs.anl.gov/mpi
	PVM	www.csm.ornl.gov/pvm/
	Parallel Tools	www.ptools.org/
	Top500	www.top500.org
	Spec	www.spec.org/
	ThreadMarks	www.cs.rice.edu/~willy/TreadMarks/overview.html
	Alewife	cag-www.lcs.mit.edu/alewife
	Beowolf	dune.mcs.kent.edu/~farrell/equip/beowolf
	Avalanche	www.cs.utah.edu/avalanche/
DCE	**Transarc**	www.transarc.com
Databases & Transacs.	**Open Group**	www.opengroup.org
	THOR	www.pmg.lcs.mit.edu/Thor.html
	TPC	www.tpc.org
	Arjuna	arjuna.ncl.ac.uk
Information Dissemin.	**Ninja**	ninja.cs.berkeley.edu
	Salamander	www.eecs.umich.edu/~farnam/projects/collab.html
	Globe	www.cs.vu.nl/~steen/globe
	W3Objects	arjuna.ncl.ac.uk/W3Objects/index.html
	Infospheres	www.infospheres.caltech.edu

Table 4.2 *(continued)*
Pointers to Information about Distributed Systems and Platforms

System Class	*System*	*Pointers*
ORBs and other Object Environms.	**OMG**	www.omg.org
	IONA	www.iona.com
	TAO	www.cs.wustl.edu/~schmidt/TAO.html
	Eternal	beta.ece.ucsb.edu/eternal/Eternal.html
	ORBacus	www.ooc.com
	Java IDL	www.javasoft.com/products/jdk/idl/index.html
	omniORB	www.uk.research.att.com/omniORB/omniORB.html
DCOM		www.microsoft.com
Multi-user Applics.	**DiamondPark**	www.merl.com/projects/dp/tour/index.html
	Spline	www.merl.com/projects/spline
	Sametime	www.lotus.com/home.nsf/welcome/sametime
	NetMeeting	www.microsoft.com
	MRObjects	www.cs.ualberta.ca/~graphics/mrobjects
	Maverik	aig.cs.man.ac.uk/systems/Maverik/index.html
Distrib. file Systems	**AFS**	www.angelfire.com/hi/plutonic/afs-faq.html
	NFS	www.ietf.org/rfc/rfc1813.txt
	Ficus	ficus-www.cs.ucla.edu/ficus
	Coda	www.coda.cs.cmu.edu
Distrib. O.S.	**Amoeba**	www.cs.vu.nl/pub/amoeba/
	QNX	www.qnx.com
	Mach	www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public
	Sprite	www.CS.Berkeley.EDU/projects/sprite
	EROS	www.eros-os.org
	PLAN 9	plan9.bell-labs.com/plan9dist
	GUIDE	www-bi.imag.fr/GUIDE/presguide.html
	Alpha	www.realtime-os.com/alpha.html
	Angel	www.soi.city.ac.uk/research/sarc/angel
	Inferno	www.vitanuova.com
	Grasshopper	www.gh.cs.su.oz.au/Grasshopper/index.html
Webcasting	**Marimba**	www.marimba.com
	Pointcast	www.pointcast.com
	TIBCO	www.tibco.com
	InfoBus	www.trl.ibm.co.jp/projects/ibr/index_e.htm
	iBus	www.softwired-inc.com/

5 CASE STUDY: VP'63– THE VINTAGEPORT'63 LARGE-SCALE INFORMATION SYSTEM

This chapter starts a case study that we carry throughout the book: The VP'63 (VintagePort'63) Large-Scale Information System. An imaginary wine company owning a traditional and obsolete information system starts a project aiming at its modernization. The case study is methodically addressed at the end of each part, so that we progressively improve VP'63. In this part, we start by making it: modular, distributed and interactive.

5.1 INTRODUCTION

We start our running case study, which we will develop throughout the book. At the end of each part, we apply the concepts addressed in that part. The purpose of the case study is to exercise the skills of the architect in developing an architecture, and ultimately assessing how well the notions discussed in the book were assimilated. In consequence, we will use the style of a dialogue inside a team of system architects, and will intentionally not define or refer to the places where the terms and concepts used have been previously treated in the book.

An imaginary traditional Portuguese wine company owns an obsolete information system. The company management has devised an ambitious strategy for enhancement of the system in support of current and emerging business, and has contracted a team to develop an architectural project for that development. The corporate strategy makers traced the following objectives:

P. Verissimo et al. *Distributed Systems for System Architects*

- Seamless business information support system, from shop floor, through offices, to higher management, enabling applications to give coherent and up-to-date information about the state of the business.
- Coherent document management support system, allowing the design of simple applications that reliably disseminate persistent information created by several producers to the whole or groups of enterprise collaborators.
- One-PC-per-employee strategy, adapted to the real circumstances (e.g., rural workers) ie., at least one-PC-per-employee-group.
- Improved automation of the wine processing facilities, aimed at a better quality of the core process (wine making) whilst retaining the traditional ways, and at more effective handling of the ancillary processes (bottling, corking, labeling, etc.).
- Integrated industrial management support system, allowing the design and/or installation of multiparty interoperative applications, a prompt, real-time perception of the shop-floor state from several places in the company, and an integration with the business information system.
- On-line transactions in two facets: a Web portal oriented towards the wine culture, featuring historical and informative contents and a virtual wine shop supporting direct customer-to-business commerce; and a virtual enterprise network connecting the company to its suppliers and downstream wholesale clients, supporting business-to-business commerce.

The project received the code name of *VintagePort'63 Large-Scale Information System*, VP'63 in short, and the team will be composed by the authors and the reader. The case study is methodically addressed at the end of each part, so that we progressively improve VP'63. Of course, this strict sequence relates to the book structure. In a real project the architect had better tackle all the facets concurrently— distribution, fault tolerance, real-time, security, management— in a spiral of development that clarifies their interdependencies and eliminates conflicting objectives, until the final architecture.

5.2 INITIAL SYSTEM AND FIRST STEPS

The company has several vineyards in the country, with local offices and processing plants in some of them. The central offices are in Porto, the capital of the famous Port Wine. The information system, like many others of the earlier generations, is mainframe-based, centralized, without Internet access. Remote facilities access it through remote login, via virtual terminals on PCs, connected to the mainframe through leased lines, as exemplified in Figure 5.1a.

More recently, some of the larger offices, co-located with the processing plants, have augmented their computing needs, such that ad-hoc solutions were put in place. This mostly consisted of installing mid-range servers with local databases and hosting local support services, both office and production management. However, this evolution created a potential for inconsistency with the mainframe system, which in principle must hold a coherent state of the

business. In fact, this situation can be described as having semi-autonomous subsystems or *islands*, now detailed in Figure 5.1b. These islands must perform periodical explicit state reconciliations with the central mainframe system. It had been decided that these operations take place during the night, every day, since they involve stopping the system, making a series of file transfers, and running consolidating transactions.

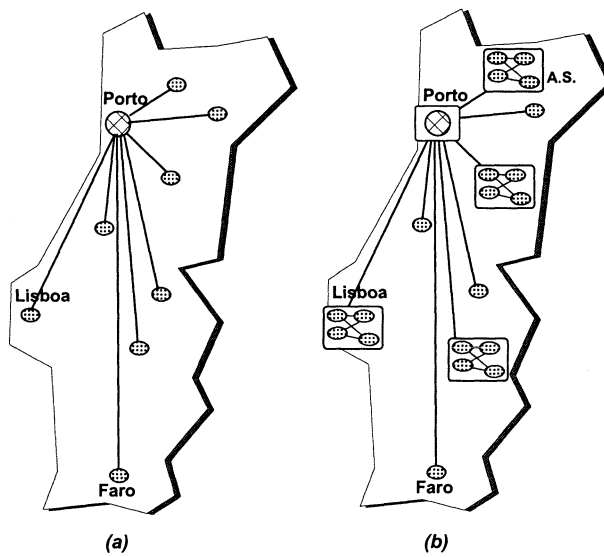

Figure 5.1. (a) Centralized Mainframe (rlogin); (b) Autonomous Subsystems/Islands (ftp)

5.3 DISTRIBUTED COMPUTING APPROACHES

In what concerned distribution, the team agreed that the strategy for the system evolution should be traced along the following lines:

- maintaining centralized business control, whilst allowing the deployment of distributed services;
- achieving openess, through the use of COTS systems (e.g., Linux and WNT), protocols (e.g., TCP/IP, HTTP), and infrastructures (e.g., Internet);
- distributing processing activities, for modularity in face of fast changing business configurations;
- distribution of data repositories for information and resource sharing;
- enhancement with proprietary middleware when applicable;
- distribution should have in mind availability and performance enhancement, to be addressed in later stages.

Q.1. 1 *What should be the evolution in terms of networking infrastructure?*

The networking setting of the company migrated to the Internet. The autonomous islands were already networked internally through local area networks running TCP/IP, that connected via the leased lines to a special purpose gateway from TCP/IP to the mainframe's native communication architecture. In consequence, all islands have been fitted with routers connected to an ISP via an adequate link in terms of speed and throughput.

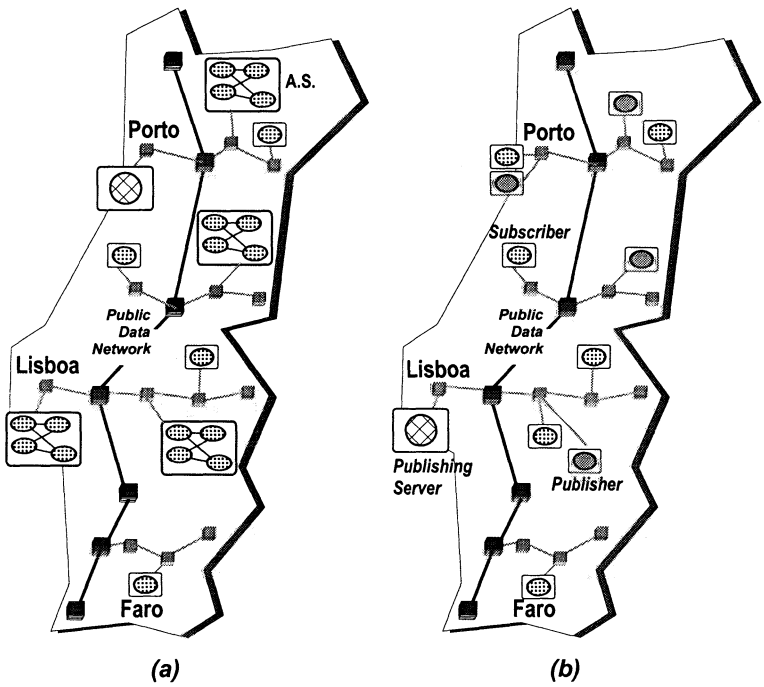

Figure 5.2. (a) Client-Server; (b) Publisher-Subscriber

On the other hand, the PCs hosting the isolated remote terminals were converted to fully working client units capable of performing local computing, and connecting through the Internet to local services (to be defined), services on the main system, and other local units. The connection was designed using a small ISDN dial-up router that allows a small-scale expansion of the number of remote clients at a facility. The new networking layout can already be perceived in Figure 5.2a. The application of the WAN-of-LANs principle was deferred to a later phase, when the installation of a secure VPN will be considered. After this phase there will be a desirable homogeneity of the architecture amongst what are now islands and isolated remote client positions. This will be beneficial both in terms of architectural coherence and modularity, and potential for reconfiguration of the company's information system layout. This was a requirement from the corporate strategy makers.

Q.1. 2 What are the adequate paradigms in what concerns the organization of distributed activities and services in such a company?

The team reflected on the business objectives present and future, traced for the company by the corporate strategy makers. Two paradigms were seen as enabling the installation of applications serving the outlined strategy:

- Client-server, enabling: access of remote facilities to central services; access to services in the autonomous subsystems by the local clients; transactional access of Web clients, both internal (employees) and external (e-commerce clients).
- Publisher-subscriber, enabling: event-based handling of generic management information in a content-sensitive manner; event-based handling of time sensitive production information, for seamless integration between the production management and the general management systems.

Q.1. 3 How should client-server and publisher-subscriber subsystems be set up?

Figure 5.2a illustrates the client-server set-up for the whole company. The main database remains at the headquarters in Porto, but the database engine is provided with a transactional front-end supporting remote queries and updates from the clients residing at the several company facilities in the country. The operations are essentially the same as supported previously by the closed mainframe. However, there is a potential for adapting old applications and writing new applications to take advantage of local client processing power, instead of loading eveything onto the mainframe. Furthermore, this opens the way to 3-tier computing from thin Web clients at a later stage.

Figure 5.2b exemplifies how a publisher-subscriber subsystem should be set up. This concerns support for applications handling the part of the documental information that requires a `push` treatment (billboards, memos, and any changes requiring prompt attention in regulations, price lists, production information such as stock/orders, etc.). In consequence, the architecture must allow for several publishers that may be in different facilities. The publishing engine resides in Lisboa, where the company has important administrative staff offices with a powerful server, which will be adapted to be the publishing engine. This is a persistent repository which holds the publication rules for the subscribers. For example, new price lists are disseminated to commercial department heads, memos are disseminated through the relevant subscription list, finished production batches are made available as new stock, etc. At least part of this information can also be made available through front-ends such as news readers.

5.4 DISTRIBUTION OF DATA REPOSITORIES

The former set-up is still based on a central mainframe database. Given the increased dependency on the information system, situations of bottleneck on the

central database may arrive. Besides, parts of the central database were already shipped every night to the autonomous islands, since their operation was too intensive to support remote accesses for every operation. This creates a daily inconsistency that twists the business-centric computational model applications are supposed to comply with, by definition of the enterprise model. The team identified a number of potential problems deriving from the analysis above:

- peak situations will become frequent when the main database in Porto will be overloaded;
- when the main database server in Porto or the connection to it fail, the whole enterprise operation stops;
- situations of network partitioning in long-haul connections from distant facilities are more probable;
- there is a potential for conflicting operations over the day between different autonomous islands, and between the latter and remote clients.

The increased informatics content of the company's information flow shrinks reaction times and increases the frequency of transactions. In consequence, this situation may assume a dramatic proportion if nothing is done to address it.

Q.1. 4 *How can the performance, availability and consistency problems created by a centralized database and ad-hoc caches be solved?*

All this points to the distribution of the information repository, by means of a distributed database management system. Fragmentation of the central database and its distribution by several of the main facilities of the company is a mandatory step, depicted in Figure 5.3a. This is easy to do since all main facilites were planned to have high quality access to the Internet. Criteria for fragmentation should pay attention to data importance, functionality and locality. Criteria for distribution should match the criteria for fragmentation, placing fragments with local information in the relevant areas, functional fragments with the corresponding department locations, critical fragments in highly-protected and/or highly-accessible locations, depending on the perspective of criticality being integrity or availability.

Fragments whose definition deserves special attention are those of the autonomous islands. For performance reasons, information accessed often should be near each location. Instead of this being done through loosely consistent caches, the persistent information repositories of the islands should be redefined as fragments of the main database, as illustrated in Figure 5.3b. The team devoted special attention to the definition of these fragments. If properly designed, most of the transactions on an island will exhibit the property of locality, not burdening the main system. A refinement was considered important though: part of the information resident in the islands databases was read-only information copied from the database and shipped to and cached in all islands on a daily basis. That information cannot stay in a single fragment, except the one at the main database site. The team identified two solutions: (a) in anticipation to the reformulation of the information flow, part of this infor-

mation will circulate through the information dissemination subsystems, i.e., publisher-subscriber and distributed Web-based file system (to be defined); (b) in anticipation to the measures to achieve fault-tolerance, where database replication will be foreseen, at this stage through read-only replicas for performance reasons.

that the database is depicted whole at the headquarters location in Figure 5.3b, despite being fragmented elsewhere.

It was also decided to keep a copy of the complete database at the headquarters location, as depicted in Figure 5.3b. The consolidated copy is achieved by periodically reconciling all the external fragments. This was planned in order to facilitate the transparent operation of strategic packages such as data warehousing, data mining or executive information systems, which require a full image of the business information system. Some of them are resource and power hungry, and making them operate on a distributed database would have a negative effect on performance. This way they can operate on a central database image without disturbing the performance of the rest of the system.

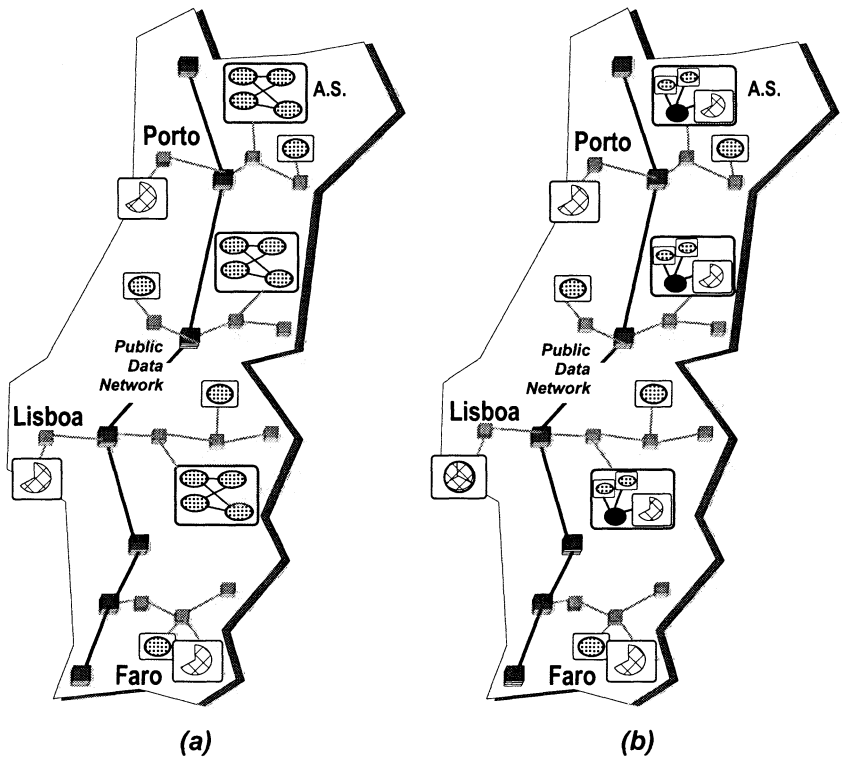

Figure 5.3. (a) Fragmentation of Central DB; (b) Consolidation of the Island's DB Fragments

5.5 DISTRIBUTED FILE SYSTEM ACCESS

Amongst the distributed access to services, the file service access is of paramount importance. It enables the transparent distribution of file-based applications and is in itself a reliable way of disseminating information that does not change too often.

Q.1. 5 *What kind of distributed file service is adequate for a geographically large-scale setting such as VP'63?*

The team specified a large-scale distributed file system (DFS) architecture with facilities for setting-up read-write (RW) and read-only (RO) files or volumes, distributed by several servers. The file system model, exemplified in Figure 5.4a, should be of the upload-download type, with server files cached in local client disks. This overcomes the delay and instability of WAN communication. Transactional file access to both RO and RW volumes makes it easy for human or computer clients to use the system as a file-based information dissemination/archival infrastructure, which can be combined with the event-based publisher-subscriber mechanisms already described. Replicated RO volumes support long-term publishing: a single writer sporadically modifies files, and releases them onto all RO copies at the distributed company sites (e.g., billboards, general regulations). Alternatively, shared RW volumes support moderately frequent single- or multiple-writer updates (e.g., global company phone directory, multiple source FAQs, or price lists, etc.). All of them should be considered as a logical part of the global information system, as suggested by the way the main file server is depicted.

Human users, mainly non computer-literate users, should be given access through the Web in as many situations as possible, given its simplicity. In addition to the existing 3-tier solutions for database access currently provided by all DB-engine manufacturers, the team studied the enhancement of the file access to provide 3-tier Web-based access to the large-scale DFS. This set-up, depicted in Figure 5.4b, has an enormous scability, but deceiving simplicity: the core infrastructure of file system servers fuels the DFS client's caches located strategically in the company infrastructure, as the second level of the hierarchy. Both servers and caches are the local file systems on which HTTP servers run, serving pages to the third level of the hierarchy, the HTTP client browsers. The set-up implies that contents to be disseminated and later browsed from, be edited in the desired language/format: ascii, html, scripting languages, etc.

Further Issues

The project needs now some refinement, and the reader was assigned the study of a few questions that were still left to be solved:

Q.1. 6 *What routing policies should be used in the application of the WAN-of-LANs principle to the networking infrastructure of VP'63?*

Q.1. 7 *What kind of protocol architecture/stack should be used for enabling client-server RPC access to the database server?*

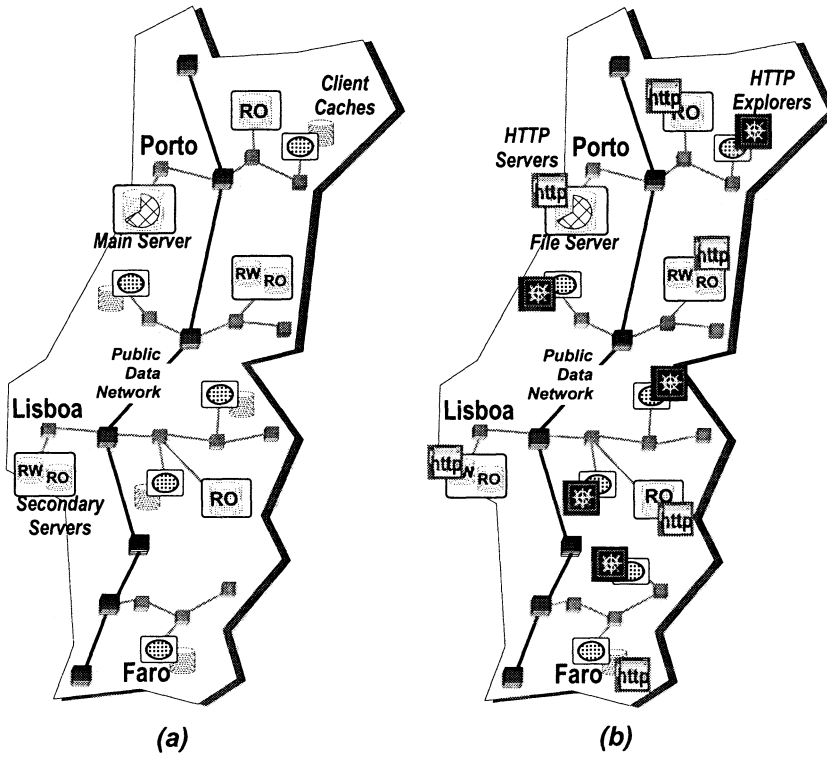

Figure 5.4. (a) Distributed File System; (b) Web-based File Access

Q.1. 8 *Design the detailed architecture of the publisher-subscriber subsystem using group communication.*

Q.1. 9 *Define criteria for fragmentation of the database and location of the relevant fragments, based on a generic classification of information criticality and locality.*

Q.1. 10 *Given that it was once possible to have the autonomous systems working for a full day completely disconnected, will it be possible to configure the system so that they are connected to the central system only a few times a day, reducing the networking costs? Determine a policy for that and its implications on consistency.*

Q.1. 11 *The same question above applies to the DFS. Define a policy for use of a weakly consistent file system that supports disconnected operation, and its implications on consistency.*

Q.1. 12 *Consider the transactional DFS studied: (a) would it easily support a low read/write ratio with single writer?; (b) would it get worse with highly*

competitive multiple writers with high read/write ratios?; (c) and with low read/write ratios?

Q.1. 13 *Design the detailed architecture of the 3-tier DFS-Web infrastructure, including layout and update mechanisms. Suppose that the core DFS service is hosted in three main servers. Optimize both the freshness of information updates and the response time of Web page requests from end clients.*

6 FAULT-TOLERANT SYSTEMS FOUNDATIONS

This chapter addresses the fundamental concepts concerning fault tolerance. It starts by introducing the notion of dependability and discussing why it is difficult to build dependable systems. The evolution of fault-tolerant computing is reviewed, from hardware fault tolerance to distributed software-based fault tolerance. Finally, the chapter introduces the most relevant architectures for fault-tolerant communication and processing, that are later described in detail in the subsequent chapters of this part.

6.1 A DEFINITION OF DEPENDABILITY

Compared with simple but nevertheless useful instruments (such as a hammer, for instance) computer systems seem to be rather fragile artifacts: they often do not behave as we expect them to, and usually decide to do it at the most inconvenient moment. This undesired behavior has two main causes. The first is that computers are complex systems, made of many different hardware and software components. These components interact with each other in ways often not predicted by the system designer. It is a challenging task to create the appropriate architectural constructs to address the mismatches caused by this complexity (the hammer always seems to work fine, even when everything starts looking like a nail). The second reason stems from an old rule of engineering, known as Murphy's Law. Put simply, this law states that if we neglect the possibility of hazards occurring, they tend to occur in the worst possible manner

P. Verissimo et al. *Distributed Systems for System Architects*

at the worst possible moment. This chapter presents the fundamental notions required to build dependable computing systems, i.e., computing systems that behave like their users expect.

We say that a system is *dependable* if it exhibits a high probability of behaving according to its specification. This rather simple statement hides a number of subtle issues. To start with, it assumes that a comprehensive specification of the system behavior is available. This requirement is sometimes overlooked: it is not simple to derive a complete and unambiguous specification of the system from user requirements that are often fuzzy or implicit. Additionally, a complete specification should not be limited to what the system does but must also specify the environmental conditions required for the system to provide the desired service. Most people realize that a personal computer is not water-proof but few people realize what exactly happens when the computer is exposed to high temperature or heavy dust.

Another ambiguous issue in our definition of dependable system is the notion of high probability. How high is high? This naturally depends on the purpose of the target computer system: the requirements of a life-supporting system and of a video-game console are quite different. The consequences of a failure are much more dramatic in life-supporting systems than in a gaming machine (even though the resistance of an arcade console to physical damage needs to be probably higher than that of a medical instrument). The knowledge of the required degree of dependability is important because, as you may expect, dependability does not come for free. Paraphrasing Laprie (Laprie, 1992), dependability is then:

Dependability - the measure in which reliance can justifiably be placed on the service delivered by a system

Is there a systematic way to achieve such reliance justifiably? To start with, we must understand what are the *impairments* to dependability, i.e., the potential causes for incorrect behavior. Then, we must learn about the *means* to achieve dependability, i.e., the techniques that allow us to achieve correct behavior despite the impairments. Finally, we must devise a way to express the level of dependability desired and assess whether it was achieved, by defining dependability *attributes*. Each of these issues will now be addressed in turn.

6.1.1 Fault, Error and Failure

The impairments to dependability assume three facets: fault, error, and failure. When the system behavior violates its service specification we say that a *failure* occurs. Building dependable systems is about preventing failures from occurring. However, to be successful, we must understand the process that leads to failure, which starts with an internal or external cause, called *fault*. The fault may remain *dormant* for a while, until it is activated. For example, a defect in a file system disk record is a fault that may remain unnoticed until the record is read. The corrupted record is an *error* in the state of the system that will lead to the failure of the file service when the disk is read. The failure

is thus the externally observable effect of the error. It should be noted that similar failures can be derived from quite different errors. A screen filled with strange characters can be the visible result of either a defective video card or a flawed operating system routine. On the other hand, errors are sometimes not immediately visible at the system interface. In the disk example, a long time may elapse before that particular record is read. Such errors are in the *latent* state until they are detected and/or they produce a failure. As with failures, the same error can be caused by different faults. For instance, the disk error may be due to a physical fault in the disk surface (bad record), but it may also be due to a misalignment of the disk head.

There is a wealth of fault types, which can be classified along several axes or viewpoints. There is a fundamental distinction of the phenomenological origin of faults: *physical*, generated by physical (hardware) causes; *design*, when introduced during the design phase; *interaction*, when occurring at the interfaces between system components, or at the interfaces with the outside world. Design faults, and some interaction faults, are caused by humans. Faults may also be classified according to the nature (accidental or intentional, malicious or not), the phase of creation in the system's life (development or operation), the locus (internal or external), the persistence (permanent or temporary). A classification is proposed in (Laprie, 1992). Most relevant to distributed systems are interaction faults, since they target interactions between distributed components. Amongst them, temporary faults assume special relevance: *transient* (external) faults mainly affect communication (for instance, electromagnetic noise due to a spark); *intermittent* (internal) faults mainly affect concurrent programs, so typical of distributed systems (for example, races and deadlocks).

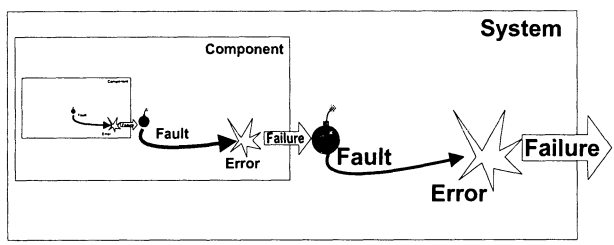

Figure 6.1. Fault, Error and Failure

The fault, error, and failure definitions can be applied recursively when a system is decomposed into several components:

fault → error → failure → fault → error → failure ...

That is, an error inside the system is often caused by the failure of one of its components, which at system level should be seen as a fault. Figure 6.1 illustrates this recursion. Although we should avoid ambiguity when addressing the mechanisms of failure, it is thus possible to address the same phenomenon as a (component) failure or a (system) fault, depending on the viewpoint. Likewise, interactions between components may fail in several ways (e.g., timing failure)

constituting system-level faults that lead to an erroneous state (e.g., timing error). Faults may cause other faults. An error may give rise to other errors, by propagation, and in fact, a failure may be at the end of a chain of propagated errors.

6.1.2 *Achieving Dependability*

As we have seen, there is usually a cause-effect chain from a fault to a failure. To achieve dependability one should break this chain by applying methods that act at any point in the chain to prevent the failure from occurring. The source of the chain, the faults, are thus the natural targets of several means to achieve dependability. These means can be used in isolation or, preferably, in combination.

The first approach we study is called *fault removal.* It consists of detecting faults and removing them before they have the chance of causing an error. Targets of fault-removal include software bugs, defective hardware, and so forth. *Fault forecasting* is the set of methods aiming at the estimation of the probability of faults occurring, or remaining in the system. Some classes of faults are easier to detect and remove than others. In consequence, fault forecasting can be seen as complementing fault removal, by predicting the amount of residual faults in the system.

Fault prevention, as its name implies, consists of preventing the causes of errors by eliminating the conditions that make fault occurrence probable during system operation. For instance, using high quality components, components with internal redundancy, rigorous design techniques, etc. The combination of fault prevention and removal is sometimes called *fault avoidance*, i.e., aiming at a fault-free system.

Of course, not all faults can be prevented from occurring during system operation, whereas other faults may even be present from the beginning of operation, having eluded fault removal. In consequence, one must create complementary mechanisms that block the effect of the fault before it generates a failure. In such case, we say that the system is capable of providing correct service despite the occurrence of one or more faults or, in other words, that the system is *fault-tolerant* (FT). Fault tolerance acts at the stage of error production, through mechanisms that are designated by *error processing*. Upon identification of an error-producing fault, it is desirable to eliminate it, as soon as possible, in what is called *fault treatment*.

6.1.3 *Measuring and Validating Dependability*

Fault avoidance and fault tolerance are strategies that can help the system architect to build dependable systems. But how to assess the degree of success of these strategies, i.e., how to *measure* and *validate* the degree of dependability attained by a system? This is expressed through the following *attributes*:

Reliability	the measure of the continuous delivery of correct service
Maintainability	the measure of the time to restoration of correct service
Availability	the measure of the delivery of correct service with respect to the alternation between correct and incorrect service
Safety	the degree to which a system, upon failing, does so in a non-catastrophic manner

Recall that we have attached a probability to the notion of dependability as provision of correct service. Several of these metrics of dependability can be expressed as probabilistic functions. *Reliability* can be equated with the probability that the system does not fail during the period of mission of the system (e.g., a flight). For continuous mission systems (e.g., web servers), reliability can also be expressed by the *mean time to failure* (MTTF) or by the *mean time between failures* (MTBF). Finally, reliability can be expressed as a failure rate probability, for example, 10^{-9} *failures per hour*, a typical figure for safety-critical systems. Another metric of dependability is *availability*, the probability of the system being operational at any given time, when it alternates with failed states. *Maintainability* is the attribute defining the time needed for the system to recover from a failure. Given reliability MTBF and maintainability MTTR (mean time to repair) of a system, availability can be expressed as *MTBF/(MTBF+MTTR)*. For the sake of example, Table 6.1 shows the corresponding downtime per year for several availability figures. Worthwhile noting is the capability of defining a continuum between full service and complete interruption of service, expressed as *performability* (Meyer, 1992). This attribute quantifies the capacity of graceful degradation of a system. In other words, it offers a combined metrics of performance and dependability, which can be seen as a 'dependability' view of *quality of service* (*see Quality-of-Service Models* in Chapter 13). Another important attribute is *safety*, which is equated with the conditional probability of, given a failure, it not being catastrophic. Other attributes that can be considered of dependability are those concerning *security*, such as the preservation of confidentiality or integrity. Security is discussed in Part III of this book.

Given a specification in terms of system dependability attributes, it is important to *validate* whether the latter are attained. The distinction is often made between *building the right system*— validation in general terms— and *building the system rightly*— verification of the design and implementation— as two facets of this process. This couple is often referred to as *Validation & Verification*, or simply *V & V* (Boehm, 1988). One cannot simply put the system into operation and measure how often it fails; usually the desired probability of failure is so low that this approach is infeasible. In the early design phases, the functional and design specifications of the system should be subjected to *design validation*, with the aim of assessing whether the proposed system satisfies the requirements. Later, the implementation specifications (e.g., algorithms and protocols) and the implementation itself (e.g., code) should be subjected to

Table 6.1. Downtime per Year for Several Availability Figures

Availability	Down-time/year	Example Component
90%	> 1 month	Unattended PC
99%	≃ 4 days	Maintained PC
99.9%	≃ 9 hours	Cluster
99.99%	≃ 1 hour	Multicomputer
99.999%	≃ 5 minutes	Embedded System (PC tech.)
99.9999%	≃ 30 seconds	Embedded System (special HW)
99.99999%	≃ 3 seconds	Embedded System (special HW)

implementation validation, to check whether the system is correctly built. For instance, an automatic tool can check a specification (made in a formal language), against system properties (also formally specified): this is called *formal verification*. This sort of verification can also be made in implementation code, by tools that perform exhaustive code walks. Another important class of validation techniques is *dependability evaluation*: techniques in this class allow to quantify the dependability of a system based on the dependability of its components. Examples of such validation techniques commonly used in dependability work are *fault-injection* and *software reliability* evaluation. Fault-injection, as the name implies, consists in artificially generating faults in a target system and observing the resulting behavior. Software reliability evaluation can be done using techniques such as statistical trend analysis.

In conclusion, two relevant techniques to achieve dependability are fault removal and fault forecasting which complement each other. The resulting architecture can then be subject to dependability validation to assess in what extend the dependability attributes are met. Table 6.2 summarizes the impairments, means and attributes of dependability that we have just discussed.

Table 6.2. Main Dependability-Related Concepts

Impairments	Means	Attributes
Faults	Fault Removal	Reliability
Errors	Fault Forecasting	Maintainability
Failures	Fault Prevention	Availability
	Fault Tolerance	Safety

6.1.4 Fault Assumptions and Coverage

In order to avoid or tolerate faults we need to understand how, how often, and for how long they occur. Thus, the first step to building a fault-tolerant system

is to define the *fault model*, i.e., the number and classes of faults that need to be tolerated. Obviously, it is possible to classify faults according to many different criteria. In this book we are mainly concerned with system faults in *interactions* between components, or in other words, faults concerning actions whose result is observable outside the component, since these are the most relevant in distributed architectures (network messages, input-output observations and actuation, clock readings, disk reads and writes, etc.).

The most benign classes of faults belong to the *omissive* fault group. These faults are characterized by the component not performing some interaction when specified to. *Crash* faults occur when a given component permanently stops operating. *Omission* faults occur when a given component omits an action from time to time. *Timing* faults occur when a component is late performing an action. The delay between the specified instant for the action to take place and the actual instant when the action is observed is called the *lateness degree*. Note that an omission fault can be seen as a particular case of timing fault, that exhibits an infinite lateness degree (the action never takes place). Likewise, if *omission degree* is the number of successive omission faults, then a crash fault is a particular case of omission fault, with an infinite omission degree (all actions after a certain point are omitted).

The *assertive* fault group is characterized by the component performing some interaction in a manner not specified. Assertive faults are further divided into *syntactic* faults, when the construction of the interaction is incorrect, and *semantic* faults, when the meaning conveyed by the interaction is incorrect. A syntactic fault is a semantic fault where the construction is also incorrect. Consider a room temperature sensor interacting with a controller. The output of the sensor is defined to be a signal sign (+ or−) followed by two numeric digits. Readings such as "+ab" or "*24" can be marked as erroneous by a syntactic analyzer (an error detector in fact) while a reading of "-99" can only be detected as incorrect by using the application semantics (for instance, if you know that the target can *never* reach that temperature) or through comparison with redundant information (held by the user, or from other correct components).

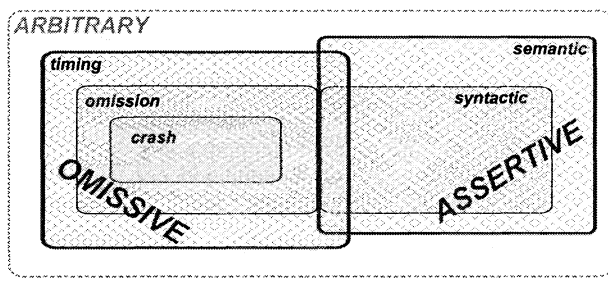

Figure 6.2. Classes of Interaction Faults

Figure 6.2 illustrates the relationship between the fault classes just described. When interactions are multi-component (e.g., a multicast transmission), faults

may affect all concerned components, in which case they are said to be *consistent*. Otherwise, they are *inconsistent* (e.g., omission faults at just some of the recipients).

Omissive faults occur essentially in the time domain. Assertive faults occur in the value domain. Inconsistency introduces the space dimension. When these dimensions can combine, we have *arbitrary* faults. The arbitrary fault class is used when one does not wish or cannot make any assumptions on the behavior of faulty components. Obviously, this should be understood in the context of the universe of "possible" faults of the concerned operation mode of the component. We recall that we are interested in interaction faults. Practical systems based on this baseline assumption normally specify quantitative bounds on the number of faults, or at least quantify the tradeoffs between the resilience of their solutions and the number of faults eventually produced (Babaoğlu, 1987).

Of course, this behavior seems overly pessimistic, and needless to say, expensive to tolerate. An arbitrary fault can be caused by an improbable but possible sequence of accidental events, which for example may lead to a catastrophic failure of a safety-critical system. It can also be caused by the meticulous action of an intentional and malicious component (an intruder) that deliberately tries to defeat the system protections. Both cases (safety and security) may justify the adoption of such a restrictive fault model. A well-known subset of arbitrary faults are the *Byzantine* faults after a paper by (Lamport et al., 1982), that denotes inconsistent semantic faults (e.g., sending different messages to different recipients). A particular case of arbitrary fault occurs in an interaction that takes place before being expected. Although it might be called an early timing fault, it is akin to a forged interaction (Powell, 1992). In this book, timing faults always refer to (omissive) late timing faults.

System interaction faults are immaterial: they take place in the context of protocols running between components. In fact, they are very appropriately *component failures*, in the viewpoint of the components suffering them. Throughout this book, we will use either term— system fault or component failure— depending on the viewpoint. For example, we will talk about an omission or timing failure in a communication network component, which leads to an erroneous state of the communication system that we generically call an omission or timing error, and may discuss how to make protocols timing or omission fault-tolerant.

A system tolerating two arbitrary faults is not necessarily better than one tolerating two omissive faults, and this one not necessarily better than a third system tolerating one omissive fault. Recall that our purpose is to minimize the number and the probability of occurrence of failures. If in worst case, the probability of the system doing more than a single omissive fault was negligible, then all the three systems would be just as good. In fact, the first two systems would be over-dimensioned. On the other hand, if we had assumed 'one omissive fault' and built the third system based on this assumption, it could fail if two omissive faults or one arbitrary fault occurred. The problematic we are addressing is generically called **coverage** of the fault tolerance mechanisms.

Coverage - given a fault in the system, coverage is the probability that it will be tolerated

It our context, it all boils down to **assumption coverage**. The assumption/coverage binomial is a quite important issue in the design of distributed fault-tolerant systems. When we assume that predicate $\mathcal{P}$ will hold with a coverage Pr, we say that we are confident that $\mathcal{P}$ has a probability Pr of holding. There is an important *separation of concerns* to be made in system design (Powell, 1992):

- **environmental assumptions** – the assumptions concerning the behavior of the environment where the system will run (which includes the infrastructure, networks, hardware, etc.), namely its faulty behavior;
- **operational assumptions** – the assumptions concerning the behavior of the system proper, or how the system will run (which includes programs, algorithms, protocols, etc.), under a given set of environmental assumptions.

The *environmental assumption coverage* (Pr_e) is then concerned with the conditional probability of a set of assumptions ($\mathcal{H}$) holding— such as clock rate of drift, network datagram delivery delay, omission error degree, number of component failures— given any occurrence of a fault f. Likewise, *operational assumption coverage* (Pr_o) is related with the probability that a given algorithm ($\mathcal{A}$) solves a problem, given the assumed set of environmental assumptions. Note that in fault-tolerant algorithms, this denotes the coverage of the error-processing mechanisms. Normally, if the algorithm and its implementation are proven correct, we expect a coverage $Pr_o = 1$. Pr_e is always an upper bound on total coverage. These are called deterministic algorithms, in contrast to probabilistic ones, where $Pr_o < 1$. In consequence, given any fault f:

$$\textit{if } Pr_e = Pr\{\mathcal{H}|f\} \text{ and } Pr_o = Pr\{\mathcal{A}|\mathcal{H}\} \text{ then } \textit{total coverage} \text{ is } Pr_o \times Pr_e = Pr\{\mathcal{A}|f\}$$

For example, we make a set of assumptions $\mathcal{H}$ about the environment, say:

H1- during a reference interval T, at most k omissions occur

H2- the participants are not subjected to partitioning

Then, we design a deterministic reliable multicast protocol $\mathcal{A}$, which has several properties, for example:

A1- any message delivered to a correct participant is delivered to all correct participants

A2- any message sent by a correct participant is delivered to at least one participant

The architect should follow two complementary tracks in designing the system: (i) proving that the protocol, based on the set $\mathcal{H}$, ensures that every Ai holds with coverage one; (ii) determining the coverage of every Hi of $\mathcal{H}$; (iii) checking whether the total coverage, which equals the environmental assumption coverage Pr_e, is satisfactory with regard to the requirements.

6.1.5 How do computers fail?

An important and useful way of discovering which faults are relevant is to gather empiric data from systems already in operation. A study on the causes of system failures in large information systems conducted by Tandem Computers has been reported by (Gray, 1986). The results are quite interesting, and still relevant today. In this study, the smallest contributor to system failures were hardware faults. Most faults (42%) were caused by incorrect system administration or human operators (as we said, users do not always understand the system). Software faults were the second cause of failure (25%). The third contribution came from environmental causes: mainly power outages, but also a fire and a flood (disasters do happen!). Other studies confirm these numbers (Gray and Reuter, 1993; Pfister, 1998).

Several lessons can be extracted from these numbers, and these lessons can help the system architect in the task of building dependable systems. The first lesson is that system dependability can be increased by using appropriate administrative and system operating procedures. The second lesson is that software development methodologies that promote fault prevention and removal can also significantly increase system reliability. The third lesson is that software fault tolerance is a critical aspect in dependable computing. Although this book presents many useful abstractions for the development of distributed software, it is not specifically targeted at software engineering. However, the interested reader will find that many of the techniques described in these chapters can be used to tolerate both hardware and software faults.

6.2 FAULT-TOLERANT COMPUTING

As we have just seen, given the impossibility of avoiding all faults, one has to tolerate them in order to achieve dependable computing. Fault-tolerant computing refers to the techniques that can be used to prevent faults from generating failures. As you can imagine, there is no single technique that solves all problems: the most suitable set of techniques needs to be chosen depending on the classes of faults to be tolerated and the service requirements.

If the end user can easily tolerate a small down-time period, fault-tolerant techniques must prevent faults from creating an erroneous system response and support a quick repair of the failed components. Often, the easiest way of preventing a wrong result from being produced is to shutdown the system as soon as a safe state is reached. For instance, in a train control system a safe state can normally be reached simply by stopping all trains. In many other cases, continuity of service must be provided even in the presence of faults: an airplane cannot be stopped before landing; the downtime of a Web server may cause unacceptable loss of revenue. In all cases fault tolerance requires the use of redundant resources.

6.2.1 *Space, Time and Value Redundancy*

The use of *redundancy* is fundamental to fault tolerance. Redundancy assumes several facets, qualitative and quantitative. Since drivers do not want to be stopped on account of a flat tire most cars carry a spare one. Raiders will not want to be trapped in the middle of the Sahara Desert and so they will carry not just one but a few spare tires. Professional trucks will mount twin wheels, because the damaging of a tire while riding is itself unacceptable. In computer systems, redundancy can be applied in the space, time and value domains, and as much of it as needed.

Space redundancy consists of having several copies of the same component. The same information can be stored in several disks, tolerating the loss of one disk (if disks are placed very far apart, we can even tolerate events such as floods or fires). Different nodes can compute the same result in parallel to ensure that, even when one of them crashes, the result is available on time (active replication). In a distributed system, information can be disseminated along different network paths, to survive physical media damage.

Time redundancy consists in doing the same thing more than once, in the same or in different ways, until the desired effect is achieved. A simple example of time redundancy is the retransmission of a message in order to tolerate omissions due to electromagnetic noise or temporary receiver overflow. More sophisticated examples consist in repeating computations that have aborted because of temporary software faults (overload, particular interleaving of operations causing deadlock, etc).

Value redundancy consists in adding extra information about the value of the data being stored or sent. This extra information is normally control data in the form of codes that allow the detection, or even the correction, of integrity errors in the data being stored or transmitted. For instance, a parity bit or an error correcting code can be added to memory chips or to disk structures, respectively to detect or detect/correct data corruption. Frame check sequences or cyclic redundancy checks can be added to the data being transmitted in order to detect multi-bit corruption by noise. Cryptographic message signatures do the same in the presence of malicious errors.

6.2.2 *Error Processing*

Error processing has three facets: error detection, error recovery, and error masking. *Detection* is the first step at avoiding failure. Detection can be performed by several mechanisms, depending on the type of error: hardware bit-by-bit comparison; error detecting codes and signatures; timeouts or watchdogs; syntactic or semantic checks, and so forth. Once detected, an error can be confined, such that it does not propagate. If there is not enough redundancy in the system to recover from the error, the component or the system can at least be shut down, confining the behavior of the system to crash failures. For example, this characterizes the way self-checking components operate (Carter and Schneider, 1968; Wakerly, 1978).

A more effective approach is *error recovery*, which requires the system to have enough redundancy to carry on operating despite the error. There are two main approaches to error recovery. One consists in going back to a correct state and try to restart the computation from there. It is called *backward* error recovery. A crude version of this approach is emblematic of computer professionals: many jokes exist on the common belief that turning the computer off and on is *the* solution to most problems (unfortunately, things such as memory leaks tend to reinforce this belief). Fortunately, backward recovery actions need not be so drastic. For instance, detecting a frame corruption using the Cyclic-Redundancy-Check (CRC) and requesting a retransmission is a simple, common, and effective form of backward recovery. Another example consists in performing a computation and checking the result according to some pre-defined assertions; if the result is incorrect, it is ignored and an alternative algorithm is tried (this approach is known as recovery blocks (Randell, 1975)). However, as some of us have already discovered by personal experience, going back to a correct state is not always as trivial as it may look: (a) the error may have propagated to other components, making recovery hard if not impossible; (b) going back may require undoing (aborting) intermediate computations that may in turn have affected other computations; (c) the computation may have produced effects outside the system, that cannot be undone by the system alone. If these problems are not properly addressed, they may leave the system in an inconsistent state. One common form of backward error recovery involves progressing by steps in the computation, and storing the system state at the end of each step. The saved state is called a *checkpoint*. When an error occurs, the computation resumes from the latest checkpoint, after re-storing the relevant state.

The alternative to backward recovery is naturally *forward* recovery, which consists in taking corrective measures that cancel or alleviate the effects of the error. Forward error recovery is often a necessity. In some cases there is not enough time to go back to a correct state and to restart from there. For instance, if a message with a sensor reading is lost, it is preferable to wait for the next reading than to request the retransmission of the reading, which gets out of date. External actions that are impossible to undo also require some form of forward recovery when errors occur. For example, if the cash dispenser breaks just after the notes are handed to the customer but before the transaction is completed, it can no longer be rolled-back (that would be backward recovery). Instead, some form of mechanism must allow the customer and transaction records to be updated later. One form of recovering from crash failures in components is by reconfiguration, for example, *switching over* to a spare component. For example, in a dual-bus LAN, when one medium is detected failed, message exchange switches over to the other.

Inasmuch as cars have single wheels but trucks have twin wheels, having a pair of everything in a computer system is only cost-effective for special-purpose cases where interruption of service, even for brief moments, is not acceptable, and may even lead to damages far greater than the cost of the replica(s)— note

Table 6.3. Error Processing Techniques

error detection	detecting the error after it occurs aims at: confining it to avoid propagation; triggering error recovery mechanisms; triggering fault treatment mechanisms
error recovery	recovering from the error aims at: providing correct service despite the error
backward recovery:	the system goes back to a previous state known as correct and resumes
forward recovery:	the system proceeds forward to a state where correct provision of service can still be ensured
error masking	the system state has enough redundancy that the correct service can be provided without any noticeable glitch

that a pair of something only handles a single omissive fault; other types of faults require even more redundancy. Anyway, if enough redundancy is added to the system, errors can be automatically masked and never become visible, because there will always be a way of providing the correct result. This is *error masking* (also called *error compensation*). For instance, assume that you have a communication medium that exhibits omission faults, and you want to build a system that tolerates a single omission. If you can afford doubling the bandwidth and transmit every message twice, the error will be always masked. A similar approach can be used to handle crash failures of a single component: just use two components and pick the result that is available. If your car has twin wheels, one of the tires can tear and you can still continue to drive happily without even noticing the error (the "happy" epithet only applies to readers not living in the suburbs of a big city).

Summarizing these concepts in Table 6.3, we conclude with a few remarks:

- in some systems, error detection is just followed by isolation of the failed component, which implies that the system either gracefully degrades if the component is not vital, or is shut down if otherwise— self-checking components fall into this category;
- error recovery requires error detection, and the two make up the most used combination of error processing techniques;
- error masking does not require detection because it is applied systematically;
- however, error detection should always be used, such that the faulty component can endure *fault treatment*: isolation, removal, repair.

6.2.3 *Evolution of Fault-Tolerant Computing*

A non-exhaustive list of the major milestones in the evolution of fault tolerance (FT) in the past few years is given in Table 6.4. The list is necessarily incomplete, and tries to refer to the works more related to distributed systems. Fault-tolerant computing is historically associated with control applications. Its foundations lie in fault-tolerant electro-mechanic and digital design. Progressively, fault tolerance has occupied its place as a prominent design concept to improve dependability of computing systems in general, and of distributed systems in particular. *Hardware-based* fault tolerance was a pioneering concept, whose basic ideas were introduced by Von Neumann. It relied on *hardware component replication* techniques, which consist in using duplicate or triplicate components that operate in lock-step and whose results are filtered by (hardware) voting components. The idea behind such techniques is that the voting element, given its simplicity, can be made by design much more robust than the component being replicated. By construction, replicated components are usually tightly-coupled, often in the same physical board. Hardware-based fault tolerance can be an effective way of ensuring that some system component has a controlled failure mode (for instance, that it only fails by crashing or that it never exhibits Byzantine behavior). One of the first implementations of the concept in a computational system was the FTMP (Hopkins et al., 1978), used in a flight control system, where components were triplets working in lock-step, what was called triple-modular redundancy (TMR). However, the approach also has some obvious disadvantages:

- it does not provide the answer to faults that affect all replicas, such as catastrophes (e.g., floods or fire) or design errors;
- specialized hardware is not cost-effective and hard to update at the same pace of Commercial Off-The-Shelf (COTS) components;
- finally, hardware faults are just a small portion of the faults a system is subjected to, and this portion is becoming less and less significant given the complexity of today's software.

With the increment in the use of fault tolerance in computer systems in general, an obvious evolution consisted in resorting to *software component replication*. The use of such techniques forms what is called *software-based* fault tolerance, whose simplest form mimics the hardware FT approach. Instead of replicating hardware components, software components are replicated and their results consolidated by a voter component also built in software. Software-based fault tolerance can be more cost-effective than hardware-based FT. It is also simpler to add or remove software replicas than hardware replicas.

Software replicas can have varying granularity, from whole programs (e.g., database) to functions (e.g., cryptographic algorithm), and can provide varying levels of resilience within the same system, achieving what is called *incremental* fault tolerance. Besides, they can be executed in the same or separate hardware modules, which may co-reside (e.g., multiprocessors) or be distributed. This prefigures a new style of system design that may be called *modular* fault tol-

Table 6.4. · Major Milestones in Fault-Tolerant Computing

1967	Diagnosability in computer systems (Preparata et al., 1967)
1968	Self-Checking component (Carter and Schneider, 1968)
1975	Recovery blocks for software FT (Randell, 1975)
1976	Transactions (Eswaran et al., 1976)
1978	Distributed two-phase atomic commitment (Gray, 1978)
1978	TMR based computer system – FTMP (Hopkins et al., 1978)
1978	N-version programming (Chen and Avizienis, 1978)
1978	Byzantine agreement for FT communic. (Lamport et al., 1982)
1979	Weighted voting for replicated data management (Gifford, 1979)
1978	Distributed software-based FT – SIFT (Wensley et al., 1978)
1978	State machine FT programming model (Lamport, 1978a)
1981	Distributed Atomic Transactions (Lampson, 1981)
1985	Group-oriented FT programming model – ISIS (Birman and Joseph, 1987), AAS (Cristian et al., 1985)
1985	FT distributed-system fieldbus – MARS (Kopetz et al., 1989a)
1985	Commercial FT computers – Stratus (Wilson, 1985)
1986	Commercial FT computers – Tandem (Bartlett et al., 1987)
1986	Commercial LAN with medium FT – FDDI (FDDI, 1986)
1987	Distributed and incremental FT (Powell et al., 1995)

erance: system architects break down the system in software modules, decide the different levels of replication of each module, and foresee the number and placement of the necessary hardware modules where software modules will be installed.

The most obvious use for this modularity is *distributed* fault tolerance, which leverages the advantages of modular fault tolerance through the failure independence given by geographical dispersion: each replica of a given component is located in a different node of a distributed system. Modular and/or distributed FT are the most used approaches today in dependable system design. The emphasis on component interaction explains why algorithms and protocols are so relevant in this class of systems, and also why the focus lies on interaction faults, as discussed earlier. In addition to these important arguments, there is today a fundamental case in using distributed fault tolerance: existing computing systems are distributed and need to be made dependable. Distributed FT provides the ground to tolerate several classes of faults:

- hardware faults, since the results of the several replicas are consolidated such that a correct value is returned, despite the failure of hardware components;
- transient software faults (also called Heisenbugs from the Heisenberg uncertainty principle), since they occur sporadically and are thus normally masked by redundant execution in different environments;
- disasters, since the geographic dispersion of nodes supported by distributed FT provides the basis to survive them.

In the case of design faults, simple replication provides little help: errors will systematically occur in all replicas. In consequence, what is needed is

that replicas are designed diversely, different system architectures are used, or execution results are tested against assertions about the desired outcome. For example, each replica of a given component can be designed and developed by a different team of programmers. This is one of the main techniques to achieve *software fault tolerance*. Software design diversity is rather expensive (most software products already cost too much, even when a single development team is involved) and as such it is only employed when the stakes are very high, such as in safety-critical systems. On the other hand, a characteristic of distributed fault tolerance is that replicas of a same component can reside in different hardware and/or operating system architectures, and execute at different moments and in different contexts. This implicit "diversity" is enough to tolerate many design faults, especially those hardware or software faults that lead to an intermittent behavior. It follows that (external) transient faults can also be tolerated this way.

6.3 DISTRIBUTED FAULT TOLERANCE

When characterizing distributed systems in general (and not necessarily fault-tolerant) we have mentioned a number of important properties, such as modularity, support for heterogeneous hardware, incremental growth, etc. These properties derive from the distributed nature of the system and are not specific to fault tolerance. However, they are also extremely important when applied to fault tolerance! We will illustrate this fact with a couple of examples.

An important property for fault tolerance is *modularity*. Distributed fault tolerant systems are built of nodes, networks and software components. Failure assumptions are mainly concerned with the interactions between these hardware and software components. Construction of these systems is based on modular hardware and software units. A separation of concerns is sought by attempting at decoupling software units from the hardware units where they execute, as suggested by Figure 6.3. Fault tolerance is to a large extent achieved through adequate protocols to govern the interactions between components in the presence of the assumed failure modes.

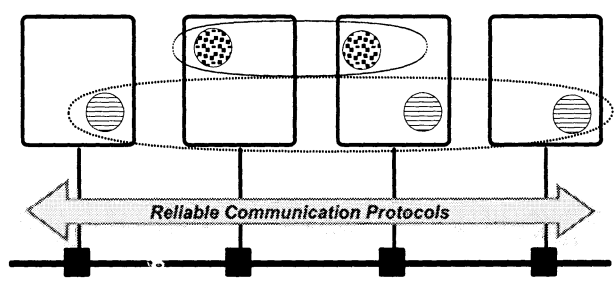

Figure 6.3. The Principle of Distributed Fault Tolerance

If the appropriate design techniques are used, either hardware or software components can be replaced without changing the complete architecture. In

result of this modularity, different dependability levels can be achieved using different combinations of components and protocols. A distributed architecture can thus provide *incremental dependability*, i.e., higher dependability features can be obtained in an incremental way: by enhancing the more fragile components (fault prevention); and/or incrementing the number of replicas of each component or making them resilient to more severe faults (fault tolerance). Another advantage of modularity lies in making it easier to build systems that exhibit *graceful degradation*: when some components fail, instead of collapsing, the system continues providing a lower level of service. It is possible to progressively decrease the degree of dependability of an application, or to preserve some applications in detriment of others.

Support for *heterogeneity* is also a key element of distributed fault tolerance. As we have mentioned before, it allows the use of components with diverse design and this can help in tolerating design faults. Support for different hardware also simplifies the evolution of the system, puts less constraints when buying new hardware, and allows to minimize the financial costs associated with the use of redundancy. On the other hand, it enhances *maintainability*, through the possibility of re-instantiating failed software units in available hardware units of different makes.

It is also easy to extend some of the fundamental properties of distributed systems to make them useful for fault tolerance. We will just give one example using the *encapsulation* property. Distributed systems allow designers to assume a glass-box view of interconnected black-box components. Remote object invocation is a simple way of hiding the internal structure of the server from clients. This property allows the system architect to build modular, possibly distributed, fault-tolerant subsystems, which she may recursively use as black-box components with resilient properties at a higher level of abstraction. For example, a closely-coupled multicomputer may be built with distributed fault tolerance techniques, and then used as a resilient black-box component of a wider fault-tolerant distributed system.

We must end this section with a word of caution. Despite all these advantages, the system architect should never forget that complexity is itself a potential cause of faults. We have started this chapter by stating that many faults are due to unexpected or not fully understood interactions between different components of the computing system. Modular and distributed systems do not make these interactions any simpler. The KISS rule should be considered as a rule of thumb by every architect: "Keep It Simple, Stupid!" On the other hand, rigorous design principles must be followed when building a dependable distributed system. Chapters 7 and 8 will survey the main paradigms and models than can help the system architect in this task.

6.4 FAULT-TOLERANT NETWORKS

We have just mentioned several advantages of distributed fault tolerance. Before you go over-enthusiastic about this strategy, remember that there is usually no such thing as a free lunch. The "catch" here is that distributed components

need a communication network to interact with each other. Naturally, the communication network should not be a single point of failure itself. Thus, we need to build *fault-tolerant networks* too!

The interaction failures that we studied earlier are an adequate representation of what may go wrong in networks: omissions due to lost messages (messages can be lost because of noise, lack of clock synchronization, overflow at the recipient or in a router, etc); timing failures, specially when a single channel is shared by several nodes; assertive failures, when data is corrupted along the transmission path.

A belt-and-suspenders approach consists in constructing a fully space-redundant architecture. Each node is connected to the other nodes by more than one link in parallel. The example illustrated in Figure 6.4a features a duplicated broadcast bus LAN, where we can see that the complete network is replicated, including the physical channel and the communication boards. This architecture tolerates one omissive fault. The actual number of network replicas will depend on the number and type of assumed faults. Variants may exist depending on the criticality. For example, the buses may be unidirectional (single transmitter, multiple receivers), one per node, so that no one node can disrupt communication by jabbering the channel. This approach would require four channels for the example in Figure 6.4a. Alternatively, the network may have the topology of a completely or partially-connected graph of point-to-point links, where reliability is achieved through store-and-forward transmission through the several alternative routes, as exemplified Figure 6.4b. Hypercubes are a common topology in this kind of networks. The fully space-redundant architecture provides a basis to achieve tolerance of any class of fault. However, it is very expensive and is only used in extreme cases.

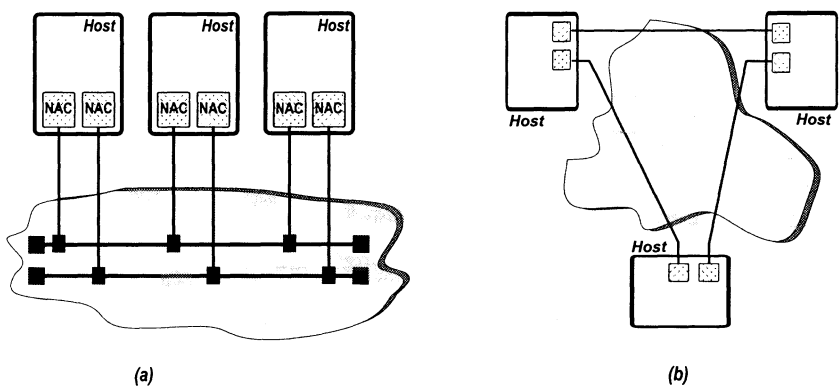

Figure 6.4. Space-Redundant Network Architecture: (a) Bus; (b) Point-to-Point

For omissive faults, there is also the alternative of using time redundancy, i.e., to send the same message several times through a simplex (non-replicated) channel. However, even in this simple case, if components fail permanently (e.g., crash of an Ethernet hub), or exhibit a large number of errors (e.g., frame

omission errors due to environmental noise), time redundancy is not enough. An intermediate approach is depicted in Figure 6.5. The medium-redundant architecture aims at achieving a non-stop medium and provides an extremely cost-effective solution to the fault-tolerant communication problem. Replication is done only for the physical layer components of the architecture (cabling, modems, codecs, transceivers). The upper layers do not even known that such redundancy exists, and so any protocols for simplex networks can be used transparently. Figure 6.5a presents a dual- medium bus architecture, where transmission is done on both media, but reception is switched over to the alternative medium, upon detection of a medium failure, on a per recipient basis, a shown. On the other hand, Figure 6.5b presents a dual ring architecture. Communication takes place in the primary ring (outer one), which reconfigures if any of its parts fails and interrupts the ring, by wrapping around the secondary ring, as shown in the figure.

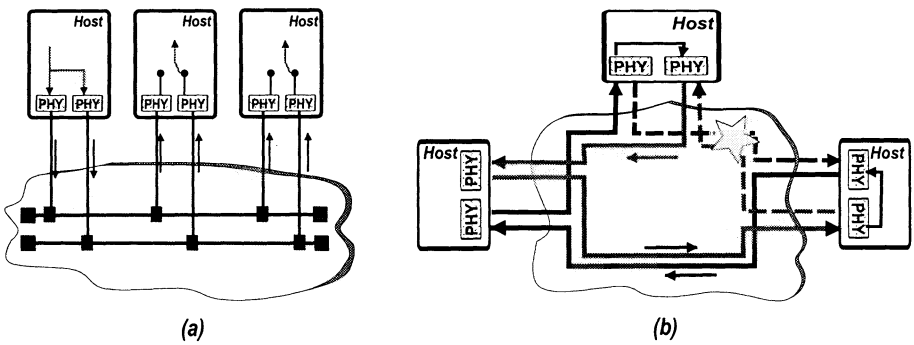

Figure 6.5. Medium-Redundant Network Architecture: (a) Bus; (b) Ring

6.5 FAULT-TOLERANT ARCHITECTURES

We have been discussing several concepts related to building fault-tolerant systems. Let us illustrate how these concepts can be put into practice, by doing an overview of the main fault-tolerant computing architectures. The detailed paradigms and mechanisms that make these architectures work will be discussed in the subsequent chapters.

Figure 6.6 exemplifies two basic architectural approaches for achieving local availability. Figure 6.6a exemplifies *redundant storage*, which achieves availability through disk replication, for example by means of RAIDs (Redundant Arrays of Inexpensive Disks), redundant disks that provide several levels of reliability. The highest level can guarantee virtually non-stop operation. However, such architectures do not solve the problem of processor failures. Figure 6.6b depicts the *redundant processor* approach, whereby the computer can be provided with more than one processor module or board, so that it remains available in the case of one or more processor failures. A dual processor architecture

provides a comfortable level of availability, and is often seen combined with the RAID-based redundant storage approach to deploy highly-available servers.

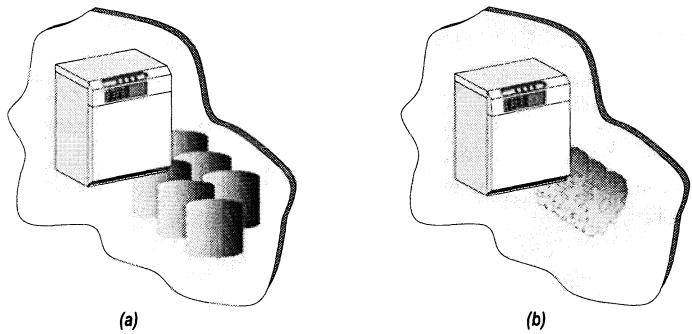

Figure 6.6. Redundant Architectural Modules: (a) Storage; (b) Processors

These basic architectures remedy the problem of crashing components, but say nothing about their possible misbehavior. Figure 6.7a illustrates the *self-checking* architectural approach for achieving local reliability vs. a wide set of faults, even assertive ones. The idea underlying the approach is that there is a component (the checker) which performs surveillance of the unit (which by the way can be a whole processor). The checker analyzes all operations and immediately stops the unit when it detects an error, before an erroneous result is propagated. A unit that is not allowed to do any errors whatsoever while functioning is said to be *fail-silent*: it behaves correctly or else fails by crashing. Now we have guaranteed correctness, but lost availability. Figure 6.7b depicts a well-known approach to achieve reliable and available processing. *N-modular redundancy*, or *NMR*, is a concept whereby several modules execute identical steps, in a tightly synchronized (lock-step) manner, so that results can be compared on a bit-by-bit basis by a voter. As long as there are less than half failed modules, the unit executes correctly, whether components crash or produce incorrect results. The example in the figure is a triple-modular redundant (TMR) architecture, which tolerates one faulty module. Note that a self-checking unit can also be built out of a 2-MR unit.

These basic concepts can be combined and applied in a broader and distributed context. For example, the lock-step model is too constraining. However, the same concept can be applied to *distributed replicated processing*, as depicted in Figure 6.8. The foundations of replication as a distributed activity have been discussed earlier (*see Replication* in Chapter 3). Based on distributed fault tolerance, these architectures provide great versatility. As exemplified in the figure, a number of replicas residing in different sites perform replicated computations, and interact among themselves through protocols that ensure resilience to the assumed fault classes (e.g., timing, value, arbitrary, etc.). The number of replicas required varies according to the fault model: class and number of faults assumed. For example, in order to tolerate one crash fault, two replicas are needed and the first result is taken. Tolerating one value fault

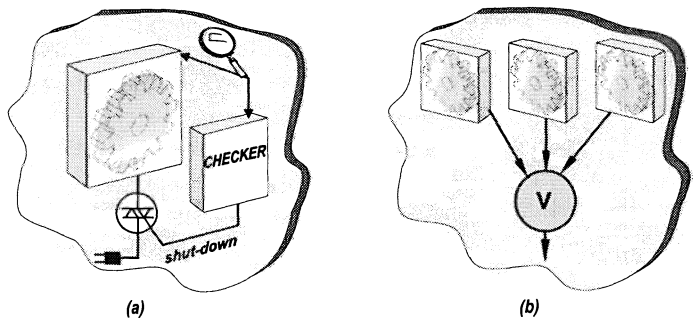

Figure 6.7. Redundant Architectural Modules: (a) Self Checking; (b) NMR

requires a majority vote between three replicas. If faults are arbitrary, then at least four replicas are required. Replicas receive inputs, execute processing steps and produce outputs in the form of messages. This is also called a *state machine* model. Unlike lock-step execution, replicas can execute at slightly different times, and in different ways, if the hosts are heterogeneous. However, they should produce the same outputs, for a same sequence of inputs.

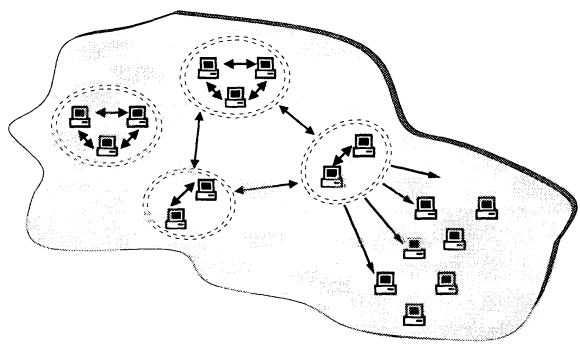

Figure 6.8. Distributed Replicated Processing Architecture

As suggested in Figure 6.8, different replica groups may communicate among themselves in order to construct more elaborate distributed computing architectures. Some architectures studied in Part I of this book (*see Distributed System Architectures* in Chapter 1) can benefit from such combinations of distributed fault tolerance techniques in order to become dependable. Figure 6.9 gives two important examples: client-server and publisher-subscriber. Figure 6.9a exemplifies how to render a service dependable. The underlying *logical* server is in fact made of two or more replicas. The principle of operation is that client requests are addressed to all replicas, executed by all, and a consolidated reply is given back to the client. The client need not (should not) even known that the servers are replicated. This is called *replication transparency.* The important notion is that the service remains available despite failing servers.

Figure 6.9b illustrates a dependable publisher-subscriber system. Recall that the architecture relies on a network publishing server. This server is a single point of failure, which is rather awkward given the notion of "bus" purported by the underlying model. That can be avoided if the server is replicated. The more widely it is replicated, the more it resembles a bus, in the sense of making information reach everywhere and not depending on the failure of a single component. Information is published to all replicas, for example through multicast. In absence of failures, subscribers may get their information from different servers that share the publishing load.

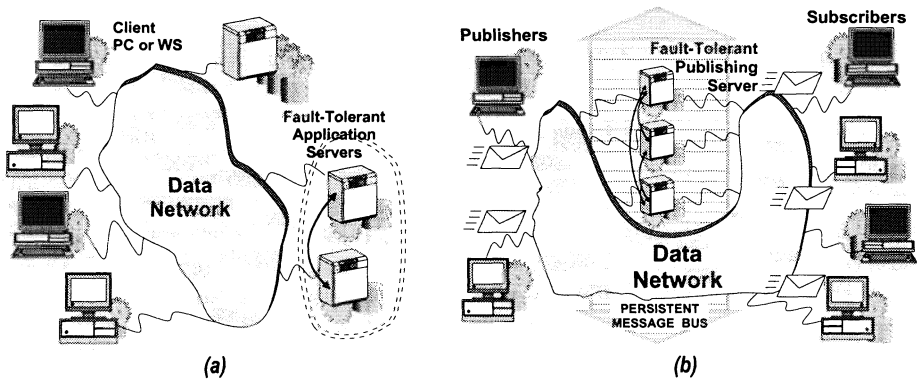

Figure 6.9. Distributed Fault-Tolerant Architectures: (a) Client-Server; (b) Publisher-Subscriber

6.6 SUMMARY AND FURTHER READING

Fault tolerance is fundamental to building dependable systems, i.e., systems that we can depend upon. This chapter has reviewed the basic concepts and terminology underlying dependability, and introduced the fundamental approaches to fault tolerance. Finally, we have characterized distributed fault tolerance and addressed the motivations for using the approach, which encompass both making distributed systems dependable and making dependable systems using distribution.

The seminal work on dependability concepts of Laprie, in the context of the IFIP WG10.4 on Fault Tolerance, is a must read for everyone working in fault tolerance (Laprie, 1987; Laprie, 1992). Other relevant works include the good survey of early fault-tolerant systems provided by Rennels (Rennels, 1984), or the more recent surveys of (Cristian, 1994) and (Mishra and Schlichting, 1992; Powell, 1994). A discussion on failure mode assumptions and assumption coverage is given in (Powell, 1992). For some works giving a comprehensive treatment of other aspects of dependability besides distributed fault tolerance, see (Lee and Anderson, 1990; Jalote, 1994; Laprie, 1998).

II Fault Tolerance

Why do computers stop and what can be done about it?

— Jim Gray, 1986

Contents

Overview

Part II, Fault tolerance, addresses dependability of distributed systems, that is, how to ensure that they keep running correctly. It contains the fundamental notions concerning dependability, such as the triad fault-error-failure and provides a comprehensive treatment of distributed fault tolerance. Chapter 6, Fundamental Concepts of Fault tolerance, starts with the generic notion of dependability and its associated concepts, and ends with the introduction of distributed fault tolerance. In fact, distribution and fault tolerance go hand in hand, since the former requires the latter to keep reliability at an acceptable level, and the latter is made easier by some qualities of the former, such as independence of failure of individual machines. Chapter 7, Paradigms for Distributed Fault Tolerance, discusses the main paradigms of this discipline. After introductory concepts and notions about fault-tolerant communication, it addresses issues such as: replication management, resilience and voting, and recovery. Chapters 8 and 9, Models of Distributed Fault-Tolerant Computing and Dependable Systems and Platforms, show how to incorporate fault tolerance in distributed systems. Explaining the main strategies for the diverse fault models, its materialization is discussed for remote operation, diffusion and transactional computing models. Finally, concrete system examples are given. Chapter 10 continues the case study: making the VP'63 System dependable.

7 PARADIGMS FOR DISTRIBUTED FAULT TOLERANCE

This chapter discusses the main paradigms concerning fault tolerance in distributed systems. Namely, the chapter addresses: failure detection, membership, fault-tolerant communication, replication management, resilience and recovery. The paradigms are explained in practical terms, by exemplifying the problems they solve, as well as their limitations.

7.1 FAILURE DETECTION

We have seen previously that one approach to build dependable systems is to detect an error and later recover from it. Since an error arises from a failure occurring at component level, component *failure detection* is fundamental to fault tolerance. Even if the system is able to mask the error, failure detection pins down the affected component. The failed component can then be disconnected or repaired, and the desired level of redundancy restored in the system. Failure detection is also important from a performance viewpoint: if a component is known to be failed there is no point in wasting resources trying to communicate with it. This section discusses the aspects related to failure detection in modular and distributed systems. As it will be seen, distributed failure detection is harder than it might look at first glance. Additionally, accurate failure detection is impossible in systems where the network can partition.

In order to detect the failure of a given component (the target) we need another component (the failure detector). We also need some channel between

P. Verissimo et al. *Distributed Systems for System Architects*

the failure detector and the target component such that the behavior of the latter can be monitored. Thus, in order to detect the failure of a component we need to add two more components to the system, and these components may also fail!

One of the possible outputs of a faulty failure detector consists in marking a correct target component as failed. That is why failure detection is a complex issue. Usually, the failure detector should be constructed in a way such that it exhibits a much higher reliability than the observed component (it should be much simpler, or be made of better hardware, or both). Additionally, the system should be constructed in such a way that the consequences of erroneous failure detection are less severe than the absence of failure detection. For instance, consider that a signaling system is introduced in a railway system to prevent train collisions. If because of benign failures in the signaling system the trains are forced to a brief halt once in a while, this is still a reasonable price to pay to avoid the loss of human lives (after all, better to arrive later than never).

Depending on its properties, the channel between the failure detector and the target can also complicate the task of achieving reliable failure detection. If the channel is not perfect (i.e., it may lose or delay information), then it may be difficult, if not impossible, to distinguish the failure of the observed component from the imperfect behavior of the channel.

7.1.1 Local Failure Detection

Let us start with local failure detection. By "local" we mean failure detection in an environment where the detector and the target are "close" enough to establish a "perfect" observing channel. For instance, the failure detector component may be in the same machine or even in the same board of the target component so that reliable communication mechanisms exist (O.S. or HW) to establish the observation channel.

Examples of local failure detection are *self-checking* routines, implemented in software or hardware such as parity checks (in memory, disks or buses). *Guardian* components check the validity of the outputs produced by the observed component: for instance, they can check if memory accesses are performed within some pre-defined allowed range. This type of failure detection is reliable, given that the channel can be considered perfect and the failure detector is quite simple. Other examples of local failure detectors are *watchdog* components. They test whether a computation progresses within a certain pace. They can be implemented in software or hardware. A HW watchdog is a down counter fed by a clock. It is loaded with the equivalent of a time interval, and the observed process has to reset it before it expires, otherwise a failure signal is produced. A SW watchdog may be implemented by the O.S. For instance, a process is instrumented to periodically set a memory position at certain points in the computation, to show it is making progress. The O.S. periodically verifies the position and resets it. Whenever it finds the position not set, it produces a failure indication (the process is late or lost). If the O.S.

is capable of controlling its timeliness, these failure detectors can be considered reliable.

However, even in a local environment failure detection may be harder than it looks. Consider the software watchdog we have just mentioned. If the monitoring is not performed by the kernel itself but by another user-level task instead, that task may not have enough information about the system scheduling decisions to distinguish the failure of a process from a load problem: the machine may be overloaded and the target process may have been swapped-out and not scheduled for some time (therefore, unable to update the variable) even though it is still in a correct state.

7.1.2 System Diagnosis

The previous model considered two different types of system components: the target components and the failure detectors. One can generalize this model by considering all components alike: each component plays a dual role in the system, providing service and testing other components. *System diagnosis* consists in identifying which system components are faulty based on the results of tests that components mutually perform on each other. Let us start with the assumption that the outcome of tests reported by correct components can be trusted. On the other hand, faulty components may incorrectly report correct components as faulty (or faulty components as correct). The difficulty here is that there is no *a priori* knowledge of which components are faulty, so the diagnosis has to be performed by analyzing the reports from all components.

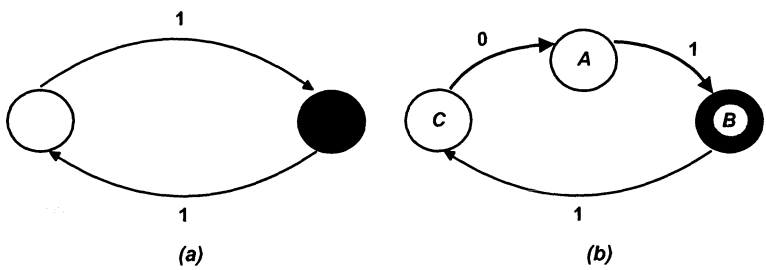

Figure 7.1. System Diagnosis: (a) Symmetric Detection (one faulty); (b) Diagnosis Ring

One way to represent a system for diagnosis purposes is to use a directed graph, where nodes represent components and edges link observers to observed nodes, in this direction. The label on the edge is either correct (0) or faulty (1). Faulty nodes are represented in black and correct nodes in white. Consider the simple example of Figure 7.1a with just two nodes that observe each other. In the example both nodes have marked the peer as faulty! Since the arcs in the Figure are symmetric, without *a priori* knowledge it is impossible to assess which is the faulty component. In fact, it has been shown (Preparata et al., 1967) that in a system where f components may be faulty we need: $n \geq 2f + 1$

processes to diagnose the fault; and that each component is tested by at least f other components.

Consider now the example of Figure 7.1b, where alternatively the diagnosis graph is organized in a ring. If there is at most one faulty process, it can always be identified, since there are three nodes. The faulty process will have: (i) a converging edge marked 'faulty'; and (ii) the source node of that edge marked 'correct'. B obeys these conditions, C is still considered correct despite marked faulty by B (because the latter does not obey (ii)). In order to perform the diagnosis, the failure detection reports must be collected and analyzed by a (logically or physically) centralized component, usually a supervisor external to the system.

7.1.3 Distributed Failure Detection

Distributed failure detection is harder than local failure detection, basically because it has to rely on message exchange (no shared memory or local interconnection buses are available). Thus, the communication channel between the observer and the target may not be perfect.

In order to focus on the difficulties introduced by distribution we will now assume that we are attempting to detect the failure of processes. To make things even simpler, we assume that processes can only fail by crashing and that the system is synchronous (i.e., delays are bounded). Thus, a process is correct as long as it provides evidence of activity (by sending or replying to messages). How this activity is monitored depends on the detection protocol. It may expect that the observed component periodically sends messages on the channel, usually called "I'm alive messages" or *heartbeats*. When these messages are missing the component is considered failed. Heartbeats are sent spontaneously, but an alternative form is for the monitored component to wait for a *probe* message coming from the failure detector and then reply with the "I'm alive" message. During periods of activity, message exchanges on behalf of the computation can be used to perform failure detection; special messages only need to be sent when the component is in an idle state. Practical systems can further optimize this process. For instance, if a broadcast channel is available, heartbeats can be sent in broadcast mode. Alternatively, the processes can be organized in a logical ring, and exchange heartbeats with their neighbors (this requires some re-organization in case of failure but minimizes the network traffic).

To simplify this even further, let us also assume that the network provides full connectivity, i.e., any process can send (and receive) messages directly to (and from) any other process. This is not always the case but allows us to abstract from the way nodes are connected at the network level. Using this topology, any process plays the role of an observer (to monitor the activity of other processes) and a target (i.e., it is monitored by all the other processes). Thus, instead of a one-to-one relation we have a many-to-many relation. Ideally, failure detection should be consistent; for instance, if a process fails, it should be detected as failed by all correct processes in the system. In a seminal work,

Chandra and Toueg (Chandra and Toueg, 1996) have defined two properties that help formalizing this intuitive notion of consistency of distributed failure detection:

Strong Accuracy - a safety requirement, specifying that no correct process is ever considered failed

Strong Completeness - a liveness requirement, specifying that a failure must be eventually detected by every correct process

If perfect channels are available, heartbeat exchanges meet strong accuracy and strong completeness. Such a failure detector is called *perfect*. If a node crashes all correct nodes will note the absence of the heartbeat and will detect the failure.

What happens when the channel that interconnects the processes is not perfect? One must distinguish the case where the imperfection can be fixed by some simple protocol, from the case where the imperfection is impossible to overcome.

The first case is simpler to discuss. Assume that the communication channel is not perfect but has some mild imperfection, for instance, it makes a small number k of omissions. The solution to this problem is to transform the imperfect channel into a perfect channel using one of the redundancy approaches described in the previous chapter. For instance, each heartbeat can be retransmitted $k + 1$ times, effectively ensuring that it is observed by all correct processes.

7.1.4 When Failure Detection is Imperfect

Perfect failure detectors are obviously very convenient. When a process goes down all the other processes know about it and can coordinate their actions to implement corrective measures. Unfortunately, it is not always possible to implement perfect failure detection, and this is the harder case.

There are two major adversaries of perfect failure detection. One is the lack of bounds on the number and type of faults the communication channel may give. Imagine that the number of omission faults of a channel between two processes cannot be bounded. If a process does not receive any heartbeat message from the other process this may be because the other process is failed or because the channel has dropped all heartbeats sent so far. Actually, no type of coordination can be achieved between two processes if the behavior of the cannel is not restricted in some way. At least, the channel should not always drop all messages (or all messages of a certain type). Thus, it is usually assumed that the channels are *fair*, i.e., if a message is sent infinitely often by a process then it is received infinitely often by its receiver (Lynch, 1996).

A particular case of link failure that prevents any sort of failure detection occurs when the link crashes and one or several processes become disconnected from the rest of the network. In this case we say that *network partitioning* has occurred and it makes more sense to use the words *reachability detection*

than failure detection. Note that consistency of reachability information is as relevant as the consistency of failure detection.

The other adversary of perfect failure detection is the lack of bounds for the timely behavior of system components (processes or links). If a link can delay a message arbitrarily, or if a process can take an arbitrary amount of time to make a processing step, there is no way to distinguish a missing heartbeat from an "extremely slow" heartbeat. This means that if the system is asynchronous, perfect failure detection cannot be implemented (*see Asynchronous Models* in Chapter 3).

This is a not a comforting conclusion. Distributed systems are complex enough even with perfect failure detection. Without it, most problems become much harder. Let us consider for instance the problem of having process A send a message m to process B over a lossy link. Assume that you define reliable communication in the following way: as long as A and B remain correct, A will succeed in sending m to B. A simple positive acknowledgement protocol can be implemented: A will re-transmit m until it gets an acknowledgement from B or until B crashes. However, without perfect failure detection, A will never be sure that B has in fact failed and, in order to meet the specification, it will have to store and retransmit m forever!

The impossibility of perfect failure detection raises another question: is there a middle term between perfect failure detection and no failure detection at all? Chandra and Toueg (Chandra and Toueg, 1996) have defined weaker forms of the accuracy and completeness properties:

Weak Accuracy - at least one correct process is never considered failed by all correct processes

Weak Completeness - a failure must be eventually detected by at least one correct process

Different classes of failure detectors can be defined combining weak and strong accuracy and completeness properties. Are all these classes useful? Let us discuss this issue in the context of pure asynchronous systems.

7.1.5 Asynchronous Failure Detection

We have just seen that the asynchrony of the system makes perfect failure detection impossible. Actually, it has been shown that several other problems requiring some form of coordination, such as consensus, atomic broadcast or atomic commitment have no deterministic solution in asynchronous systems subjected to failures. This is known in academia as the *FLP result*, after a famous paper, by Fischer, Lynch and Paterson that demonstrated the impossibility result for the consensus problem (Fischer et al., 1985).

Nevertheless, given that it is often impossible to impose a bound on message or processing delays, solutions for distributed coordination problems in asynchronous systems have been sought by several researchers. This effort lead to the following interesting question: what are exactly the minimum synchrony requirements to solve problems such as consensus? Chandra and Toueg showed

that consensus can be solved in asynchronous systems augmented with failure detectors and that the weakest failure detector to solve the consensus problem has the following two properties (the solution also requires that a majority of processes are correct, and that no partitioning occurs):

Eventual Weak Accuracy - there is a time after which some correct process is never suspected by any correct process

Weak Completeness - a failure must be eventually detected by at least one correct process

A failure detector with these properties is called *eventually weak*. The reader should note that this definition only requires weak accuracy to be satisfied at some point in time. The intuition behind this requirement is that consensus can be solved if a period of stability is preserved long enough to allow coordination among the processes. Stability is defined in terms of having at least one correct process that is not suspected by any of the other correct processes. The intuition behind this is that during the stability period, this process can act as a coordinator that supports the establishment of consensus.

Naturally, the impossibility results still holds. This means that even a eventually weak failure detector cannot be implemented in a pure asynchronous system. This fact rose several doubts about the practical utility of the model. On the other hand, most of the algorithms having this model in mind, make so little assumptions about the system behavior that they never risk violating safety conditions even when the failure detector does not satisfy the above properties. For instance, in the algorithm proposed by Chandra and Toueg (Chandra and Toueg, 1996), if the failure detector does not satisfy its properties consensus may never be reached, but on the other hand the processes never take inconsistent decisions.

7.1.6 The Problem of Partitioning

Partitioning is caused by the crash of one or more links that split the network in disjoint subsets, or *partitions*. Processes within the same partition are able to communicate among themselves but unable to communicate with processes in other partitions. Network partitioning is a serious problem in distributed systems because it prevents processes in different partitions from coordinating their activities.

There are two main approaches to address the problem of partitioning. One is to allow uncoordinated progress in different partitions. When partitioning is *healed*, in other words, when partitions merge, processes have to *reconcile* their state. Automatic reconciliation is usually very difficult (or even impossible) in the general case. An alternative approach is to allow progress in one partition exclusively, the so-called *primary* partition. The primary partition approach prevents divergence (*see Primary Partition* in Chapter 2), but in turn it blocks all the other system partitions. Several different criteria can be applied to select the primary partition, one of the simplest being a majority criterion, as illustrated in Figure 7.2. It should be noted that the network can be partitioned

in such a way that no primary partition can be identified and the system be forced to block until the partitions merge.

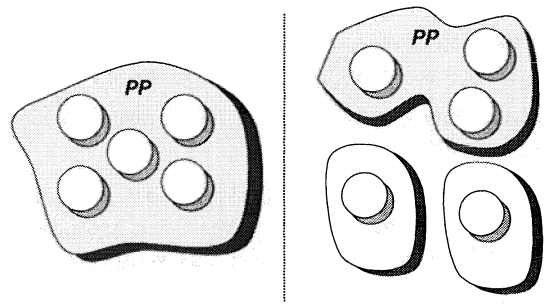

Figure 7.2. Partitioned Network with Primary Partition (PP)

Link failures are usually detected by timeouts (note however that some particular types of networks can provide accurate information about the state of links). This means that partitions can only be detected accurately if there are bounds on processing/communication delays and on the number of omissions tolerated. Otherwise, it may be impossible to distinguish a failed link from an extremely noisy or extremely slow link. It follows that partitions cannot be accurately detected on pure asynchronous systems. Inaccurate detections may create what is called a *virtual partition.* Participating processes behave as if the system was split, while in fact the links are just slow.

Conceptually, there is little difference between a virtual and a physical partition, since from the point of view of the participating processes, the two types of events cannot usually be distinguished. In practical systems, there is usually a huge difference. Virtual partitions tend to have a short duty cycle, regenerating rapidly and spontaneously, and may be repetitive. For instance a link can be temporarily overloaded because of an ongoing bulk-data transfer and regain its normal latency as soon as the data transfer finishes. However, a router may be on the verge of thrashing, holding long queues, discarding packets, recovering, degrading again, and so forth. The problem is that this may generate periods of *instability* where links switch from down to up state, forcing the system to be in permanent reconfiguration.

The application itself can cause this unstable behavior. Consider that for some reason, a link is so slow that it seems to be down. This creates a virtual partition in a distributed computation. When the link recovers, the partition is healed. Processes from either side start a state-transfer procedure to become mutually consistent. The state being transferred overloads the link and the virtual partition occurs again, aborting the state-transfer. This decreases the load on the link, causing the link to recover once more. The state transfer restarts, re-initiating the cycle. A behavior very close to this has been observed in early implementations of the BGP Internet routing protocol (Stewart, 1999).

The problem has no solution, but practical systems can alleviate these symptoms, in essentially two ways. Firstly, by making a best effort to tell physical partitions from virtual ones, so as to delay failure detection decisions to when there is a high certainty of actual failure. Quality of service failure detection is a possible approach along this line (*see* Section 13.11 in Chapter 13). Secondly, by devising programming models that accommodate partitioning. This will be discussed in subsequent sections of this chapter.

7.2 FAULT-TOLERANT CONSENSUS

We have introduced the *consensus* problem in the Distributed Systems Part of this book (*see Distributed Consensus* in Chapter 2). We recall here the definition of the consensus problem: each process proposes an initial value to the others, and, despite failures, all correct processes have to agree on a common value (called decision value), which has to be one of the proposed values. This problem is of paramount importance in distributed systems, particularly in fault-tolerant distributed systems, since many problems can be solved using consensus as a building block, for instance: membership (agreement on who are the members of a group), ordering of messages (agreement on sequence numbers), atomic commitment (agreement on the outcome of a transaction, etc). It is worth noting that the system diagnosis problem that we have addressed previously can also be considered a consensus problem, where correct processes must agree on which processes are faulty (for a deeper survey relating system diagnosis with the consensus problem see (Barborak et al., 1993)).

The solution to the consensus problem in a system where no faults occur is deceptively simple. For instance, consider the following solution: the processes with the lowest identifier is statically selected as the coordinator and sends its initial value to all the remaining processes; this value is the outcome of the consensus! In the absence of faults, this trivial protocol clearly solves the problem, since every process decides on a value and that value is one of the initial values. Curiously, it is extremely difficult to "extend" this solution to tolerate faults, even if a perfect failure detector is available. Let us see why.

To start with, it should be obvious that this solution is inherently non fault-tolerant since it relies on a single and fixed coordinator. If the coordinator crashes, the algorithm blocks. Note also that if the coordinator crashes while disseminating its value, some process may decide while others may remain blocked. It is tempting to believe that the problem can be simply fixed by selecting another coordinator in case the first one crashes. The updated protocol could work like this. When the failure of the current coordinator is signalled to the next process down the line (there is a total order on process identifiers), this process assumes the role of the coordinator and sends its initial value to every other process. When the same notification is received by some process other than the next coordinator, it simply waits for the value from the new coordinator. Unfortunately, this algorithm only works if the first coordinator does not crash during the dissemination of its value. Otherwise, some process may receive the value from the first coordinator (and decide on that value)

and others from the next coordinator (and decide differently), which violates consensus. The trick is that a protocol to solve consensus has to prevent a process from deciding a value while there is no guarantee that this is the only value that can be decided by other processes, even if faults occur. When one is sure that a given value is going to be decided (even if not every process has decided yet), the value is said to be *locked.*

How can a value be locked? Consider that we add the following rules: when a process receives the initial value from the coordinator, it changes its initial value to that of the coordinator. Thus, if that same process later assumes the role of coordinator it proposes the value it has received from the previous coordinator (note that this does not prevent a process from proposing its initial value in the case it did not receive any value from previous coordinator(s)). The protocol can then be improved as follows. The coordinator sends its value to every other process. These processes do not immediately decide the value; instead, they update their initial value and send an acknowledgment back to the coordinator. When the coordinator receives an acknowledgment from every non-crashed process, it knows the value is locked. Even if it crashes, the new coordinator (one of the processes that have acknowledged) will also propose that same value, as illustrated in Figure 7.3. Of course, at this point only the coordinator knows that the value has been decided, so it disseminates a special DECIDED message to inform the remaining processes of that fact. When DECIDED is received from the coordinator, a process can safely decide on that value.

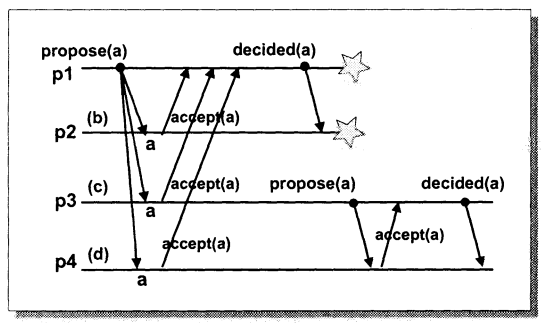

Figure 7.3. Fault-Tolerant Consensus (Perfect Failure Detector)

Note that the previous solution only works if failure detection is reliable. As we have discussed in the previous section on failure detection, in systems where failure detection is unreliable, namely in fully asynchronous systems, there is no deterministic solution to the problem (Fischer et al., 1985). We have also noted that consensus can be solved in an asynchronous system augmented with an eventually weak failure detector, as long as a majority of processes do not crash. We give a brief intuition of a possible solution. The protocol is quite similar to the protocol we have described above. However, the coordinator simply waits for a majority of acknowledgments to lock a value. This allows the system to

make progress as long as a majority of processes can communicate, no matter whether the remaining processes are crashed or simply slow. On the other hand, when another process decides to become the coordinator its task becomes more complicated, since the previous coordinator may have locked a value without the intervention of the new coordinator. To avoid proposing an inconsistent value, the new coordinator has to contact a majority of processes in order to "check" whether a previously locked value exists.

7.3 UNIFORMITY

The fault-tolerant consensus problem can be defined with two distinct flavors: the *uniform* consensus and the *non-uniform* consensus. The uniform consensus definition states that if two processes decide, they decide the same value. Note that no distinction is made between faulty and correct process (the property applies to all processes in a uniform manner). For instance, if a process decides on a value and later crashes, all the correct processes must also decide on that same value. The non-uniform flavor is a weaker form of consensus stating that if two *correct* processes decide, they decide the same value. This allows processes that remain correct to decide on a different value than that decided by a crashed process.

Note the example of non-uniformity in Figure 7.4: suppose p and q crashed or partitioned after q received m, but before r and s did. This may have undesirable effects, if processes have stable storage or partitioning may occur: when q recovers or merges, its state S_m diverges from the state of r and s, S_k.

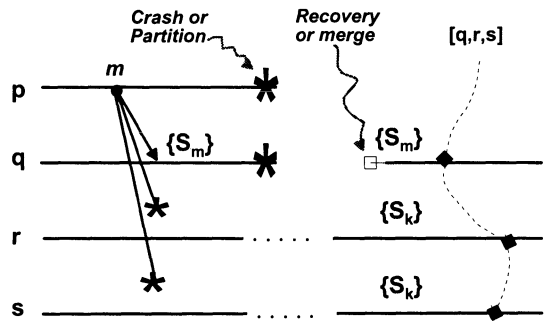

Figure 7.4. Non-Uniformity

You may ask yourself why should someone bother with these subtle differences instead of just using the stronger form of consensus in all cases. The issue is that protocols that solve non-uniform consensus can be more efficient. Consider for instance the following protocol that relies on perfect failure detectors.

As in the previous section, we assume that there is a total order on process identifiers and that the process with the smallest identifier is the coordinator for consensus. The coordinator sends its value to every process in the system and, when this value is received, a process decides immediately. If the coordinator

remains correct, eventually every process receives and decides the same value. If the coordinator crashes, the next process in the line assumes the role of the coordinator. The new coordinator asks every other correct process if they have decided. If at least one correct process has decided, the new coordinator forwards the value that has been previously decided to every other correct process. Otherwise, if no decided value is known by the processes that remain correct, the coordinator decides and disseminates its own initial value (note that the previous coordinator as well as other crashed processes may have decided differently).

Now compare the protocol described above with the protocol that we have presented in the previous section. While in the non-uniform version a process may decide as soon as it receives a proposal from the coordinator, in the uniform version, this proposal has to be acknowledged by (at least) a majority of correct processes before being decided. Thus, in all applications where the state/actions of crashed processes cannot compromise the correctness of the system, the non-uniform version of the protocol provides much better performance.

7.4 MEMBERSHIP

We have already seen many examples of distributed activities where several processes cooperate to achieve a common goal. We can refer to the set of collaborating processes as a *process group*, which has a membership. We have introduced the notion of group membership under *Consistency*, in Chapter 2. Here, we review the problem and analyze the requirements of membership protocols in the presence of failures and/or partitioning.

The membership of the group is the set of processes belonging to the group at a given point in time. A *membership service* keeps track of the group membership and provides this information to the group members in the form of a *group view*, the subset of members that are mutually reachable at a given point. The group membership is often dynamic: in response to user demand or changes in the runtime environment (load, failures, etc) the group can grow, by letting new processes *join* the group, or shrink, by letting members *leave* the group. Processes may also become involuntarily unreachable because of failures.

7.4.1 Group Membership

At first glance, the task of providing participants with information about who belongs to the group may seem rather simple. However, group membership is as a form of distributed agreement (all group members must agree on what is the group view) and we know distributed agreement in the presence of faults is an intrinsically complex problem. Actually, it turns out that even *defining* the properties of a group membership service is a difficult task, and several different types of membership service have been proposed in the literature. We will discuss the different alternatives in the next few paragraphs. For now,

let us just state the informal and intuitive definition of consistent membership information: if the membership of the group remains unchanged and there are no link failures, all members should obtain the same group view.

Usually, the agreement service is required to remove processes that have failed from the group view. So the membership information needs to be consistent and *accurate*. However, as we have seen before, accurate failure detection can be hard or even impossible to perform. In this case, how should the membership service behave? Assume that a group member p is suspected (it shows no activity). If the membership service does not remove p from the group, the application trusts p to work properly. For instance, the application might be using a work distribution algorithm where every process contributes with its share of work. While p remains in the view, other processes will expect it to do its job.

On the other hand, if the process is removed from the group it will not participate in the progress of the application and will eventually become desynchronized with respect to the other (active) processes. In practice it will become unable to contribute to the group unless some resynchronization procedure is executed to update its state. Recovery is something we will discuss later in this chapter.

The order by which the information is provided to the users is also relevant. Application code can be made simpler if changes in the group membership are received in the same order by all members. For instance, if a process is notified that p has failed, and later that q has also failed, then all other correct processes should be notified of the failure of p and q in that same order. Note that membership heavily relies on failure detection. Inaccurate failure detection may cause membership to have erratic behavior.

7.4.2 Linear Membership

A linear membership service is characterized by enforcing a total order on all views, i.e., all correct processes receive exactly the same sequence of views, as illustrated in Figure 7.5. As with the consensus problem, the linear membership service can be uniform or non-uniform. If it is defined as uniform, the history of views delivered to a crashed processes is necessarily a prefix of the history of views delivered to the correct processes.

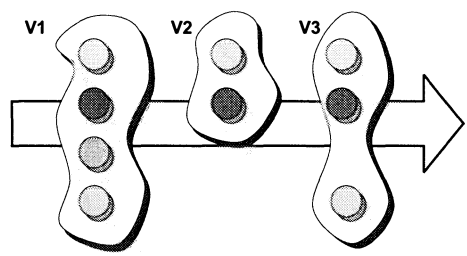

Figure 7.5. Linear Membership

In a synchronous system without partitions, linear membership can be easily enforced. Processes are either correct or crashed. All correct processes can detect the crashes and can communicate among themselves. Membership just requires agreement on the content of each view.

In a system subject to network partitions, or in an asynchronous system where failure detection is unreliable, ensuring a linear membership is more difficult. For instance, if the system is split in two non-communicating partitions, only one of these partitions, called the primary, may continue to deliver views. Because of this constraint, the linear membership is also called *primary-partition* membership.

What happens to the processes that are not in the primary partition? Three alternatives are possible. One is to block these processes while the partition persists and let them catch-up with the others as soon as the partition is healed. The other is to force these process to crash. Later they can be activated as new processes and join the system again. The third alternative is to implement a partial membership service, as described next.

7.4.3 Partial Membership

The primary-partition model is quite intuitive but too restrictive for certain type of applications. It is often interesting to keep delivering views in both partitions. Both sides continue to operate and when the partition is healed, the state is reconciled (in a manner that is usually application-dependent). Thus the group splits and merges in response to changes in the network connectivity. Views are no longer totally ordered, instead a *partial* order of views is provided. It is possible to define different types of partial membership according to the amount of overlap that is allowed among views delivered in different partitions. Of all the possible definitions of partial membership, the *strong* partial, illustrated in Figure 7.6, is the most intuitive. According to this model, concurrent views never intersect. In other words, V_2 and V_2' correspond to two completely disjoint partitions, which later merge again into V_3. Strong partial membership supports the virtual synchrony paradigm defined in Chapter 2 (*see Consistency*).

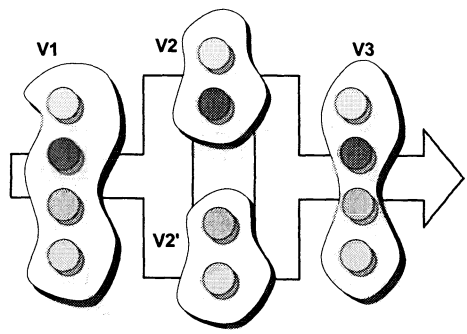

Figure 7.6. Strong Partial Membership

7.5 FAULT-TOLERANT COMMUNICATION

Fault-tolerant communication is about making sure that two or more processes can exchange information despite the occurrence of failures in the communication link or in some of the participating processes. As we have seen before, the major types of failures that need to be handled are: timing, omission, and value failures (message corruption). Malicious links can also spontaneously generate messages. In this section we discuss how these failures can be addressed, both for point-to-point and for multicast communication.

7.5.1 Reliable Delivery

We will start our study with omissions. From the previous chapter, the reader should already have a good idea of how communication can be made reliable. Two main alternatives are available: error masking or error recovery, represented in Figure 7.7. Error masking can be based on spatial or temporal redundancy. Spatial redundancy consists in deploying several links connecting the communicating processes (Figure 7.7a). In order to mask k omissions, $k+1$ links should be used. Of course, each message should be sent through all the links. Temporal redundancy consists in sending the same message several times. Again, in order to mask k omissions, the message should be sent $k+1$ times (Figure 7.7b). In both cases, duplicates should be discarded at the recipient. Error masking is appropriate when the assumed number of successive omissions k, also called the *omission degree*, is small and the need for fast recovery compensates the waste of bandwidth.

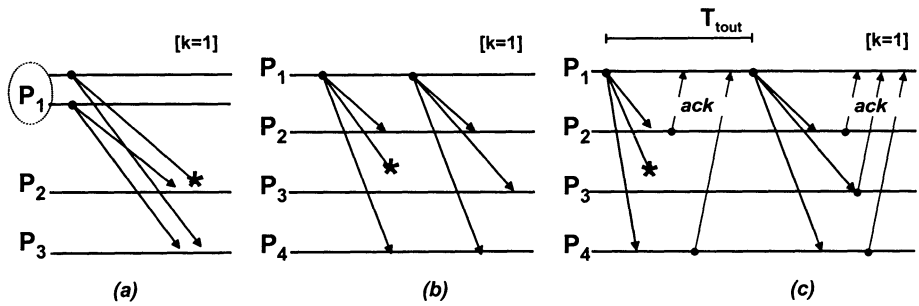

Figure 7.7. Communication Error Processing: (a) Masking (Spatial); (b) Masking (Temporal); (c) Detection/Recovery

Error detection and recovery is based on acknowledgments and timeouts. Acknowledgments can be sent whenever a message is received (*positive acknowledgment*, as in Figure 7.7c) or only when the loss of a message is detected by the recipient (*negative acknowledgment*). In the positive acknowledgment scheme, a message is retransmitted if a confirmation is not received by the sender before a timeout occurs. In the negative acknowledgement scheme, a retransmission is requested by the recipient by sending a negative acknowledg-

ment back to the sender. The former method offers a faster failure detection with sporadic traffic. The latter minimizes network traffic but requires one of two things: (a) a stream of numbered messages from the sender to the recipient, such that the recipient detects the lack of a message if it receives message i but not message $i - 1$; or (b) a time-triggered lattice such that the recipient knows it should receive a message at a given time. Error detection and recovery offers a better usage of network resources (in particular, the negative acknowledgment scheme) since retransmissions just happen when an omission actually occurs (and in most modern networks, with the notable exception of wireless communication, omissions are relatively rare).

In most practical systems, a given message is not retransmitted an infinite number of times: after a pre-defined number of retransmissions, the communication is aborted in the assumption that either the recipient or the link have crashed. This is equivalent to implicitly assuming that the system has some degree of synchrony, but this assumption is not always substantiated. The necessary steps to implement error/failure detection this way are embodied in a technique called *bounded omission degree* (*see Real-Time Communication Models* in Chapter 13).

7.5.2 Resilience to Sender Failure

We have just seen how to deal with network omissions. We now discuss the problem of sender failure. In the case of point-to-point communication this is relatively easy since it basically is a problem of failure detection. Note that if the sender crashes before successfully transmitting the message, the message is lost. Failure detection comes into play just to prevent the receiver from waiting forever for the missing message.

In multicast communication the failure of the sender is more problematic. If there are omissions in the link it is possible for a message to be received just by a subset of the participants. Usually, the sender of the message has the responsibility for retransmitting it. In such case, the failure of the sender may leave the system in an inconsistent state. At this point, it is worth to distinguish three levels of reliability in multicast, illustrated in Figure 7.8:

1. *Unreliable multicast.* The weakest form of multicast, where no effort is made to overcome link failures. Basically, multicast is as reliable as the link and the sender are (if one of these components exhibits a fault, some or all recipients may lose the message).

2. *Best-effort multicast.* A multicast where the sender takes some steps to ensure the delivery of the message, like retrying or repeating. However, if the sender fails, no reliability can be guaranteed.

3. *Reliable multicast.* A multicast where the participants coordinate to ensure that the message is delivered to all correct recipients (as long as it is delivered to at least one correct recipient). For instance, a recipient takes over a failed

sender. As with consensus, there is a stronger definition of reliable multicast, *uniform multicast* that we will discuss later in this section.

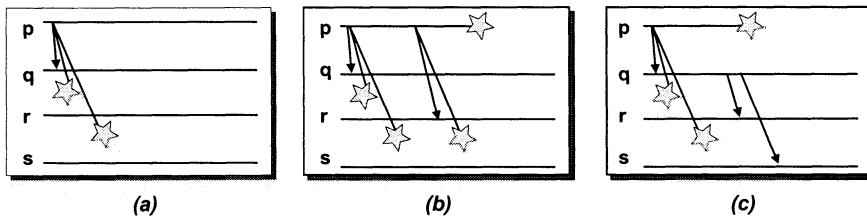

Figure 7.8. Multicast Reliability: (a) Unreliable; (b) Best-effort; (c) Reliable

How can reliable multicast be obtained? Well, we are back to error masking and error recovery. In an error masking approach, all recipients retransmit a message as soon as they receive it. Thus, even if the sender fails, each recipient makes sure the every other recipient also receives the message. If an error recovery approach is used, recipients keep a copy of the message but only retransmit the message when the failure of the sender is detected. Of course, message copies cannot be kept forever since they consume precious memory resources. Thus, some mechanism must detect when a message has been delivered to all intended recipients such that its copies can be discarded. Such a mechanism is usually called a *stability tracking protocol.*

The error masking approach is more appropriate for systems where accurate failure detection is impossible (for instance, asynchronous systems). However, unless channels are assumed to be reliable (an abstraction), the message has to be retransmitted forever. As we have noted, most systems give up after a certain number of retransmissions, which implicitly means the failure of the recipient. Error recovery also assumes that failures can be detected. As a matter of fact, the multicast reliability is strongly related with the issue of membership (failure detection and notification).

7.5.3 Tolerating Value Faults

So far, we have been discussing how to tolerate omission faults. However, messages can also be corrupted during the transmission, so it is also necessary to tolerate value faults. Fortunately, most of these faults can be detected using checksums or signatures. Messages corrupted are then simply discarded, and the value fault transformed into an omission fault, which we know how to handle.

Checksums do not cope with all the possible value faults. Value faults can be caused by a faulty sender that produces an error before the checksum is computed. Semantic faults of this type can only be tolerated using space redundancy, i.e., by comparing the values produced by different sources of the same logical value. If we have several recipients, it is important to ensure that, for consistency, the result of the comparison is the same at all correct processes.

In other words, we need to ensure that correct recipients can agree on the outcome of the comparison. To achieve such goal, a consensus algorithm must be executed among the recipients.

7.5.4 Tolerating Arbitrary Faults

Arbitrary faults are harder to tolerate. A faulty sender may send conflicting information to different recipients. Also, if the link exhibits malicious behavior, it can spontaneously generate a message that is syntactically correct, impersonating a legitimate sender. The problem of reaching consensus in the presence of arbitrary faults is called *Byzantine Agreement* after a paper by Lamport, Shostak, and Pease (Lamport et al., 1982).

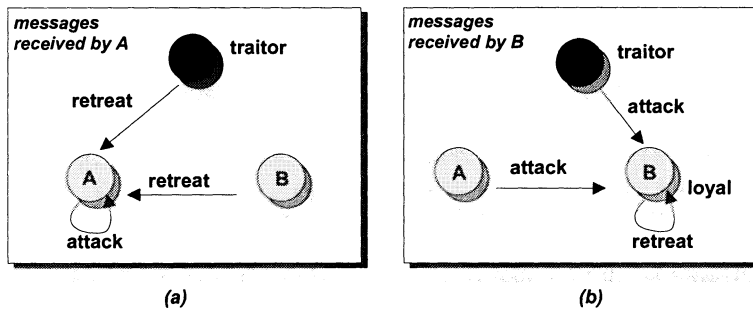

Figure 7.9. Traitor Sends Conflicting Information

The Byzantine Agreement problem can be formulated as follows. A number of generals, in face of an enemy army, must decide whether to attack or to retreat. Most of these generals are loyal to each other (correct) but some are traitors (faulty). In the presence of favorable conditions, the combined force of the loyal armies can defeat the enemy. However, unless all loyal generals attack together, their troops will be defeated. The problem is that loyal generals must agree on a single binary value (attack/retreat) despite the presence of traitors that will, maliciously, try to prevent agreement from being reached.

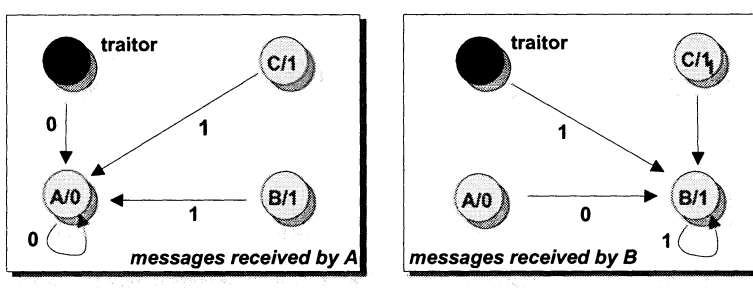

Figure 7.10. One Round of Byzantine Agreement (attack:1; retreat:0)

Assume that the system is synchronous and that the agreement protocol operates in rounds. In each round, generals send messages to every other general. However, traitors are free to omit some or all messages and to send conflicting messages to different generals. The initial value proposed by each loyal general consists of his own assessment of the correct decision: to attack or retreat. How many traitors are sufficient to prevent agreement? Consider a scenario with three generals, two of them loyal as illustrated in Figure 7.9. Given the intuitive notion of majority vote, a correct decision should be possible. We will see that it is not so. One of the loyal generals, A, wishes to attack, whereas the other, B, believes that the armies must retreat. Assume that A receives two retreat messages, one from B and one from the traitor (Figure 7.9a). Since there is a majority of retreat messages can A safely decide? Unfortunately, the traitor can send a conflicting message to B supporting A's proposal to attack (Figure 7.9b). A simple majority would force A to retreat and B to attack! Another interesting finding is that additional rounds of message exchange do not provide any help. In fact, it can be proven that at least $3f + 1$ processes are needed to tolerate f Byzantine faults so, in our previous example, it is impossible to ensure that loyal generals always reach agreement.

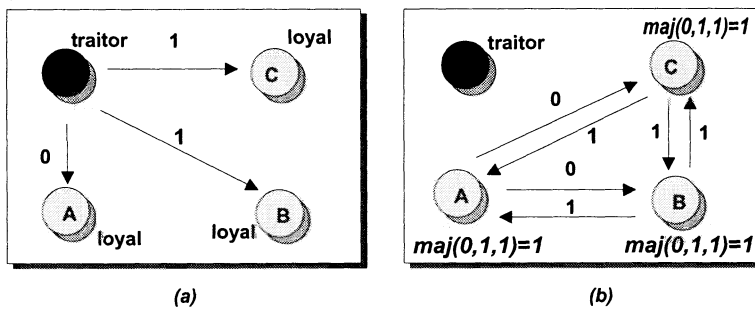

Figure 7.11. (a) First and (b) Second Rounds of Byzantine Agreement (partial view, faulty sender)

What happens if we add an additional loyal general to the system? Figure 7.10 shows a scenario after the first round of messages. Assume that the loyal generals have pre-agreed that they should follow the majority and, in case of ties, retreat. As the figure shows, consensus cannot be reached in one round of messages, even with more processes in the system. In the example, two loyal generals (B and C) want to attack and another loyal general (A) wants to retreat (from now on, we replace 'retreat' by 0 and 'attack' by 1 to simplify the figures). As before, by sending an attack vote to B and a retreat vote to A, the traitor can force A and B to disagree. However, we now have enough redundancy in the system to mask the influence of the traitor with an additional round of messages. For each sender p, the other three remaining processes exchange the values they have received from p to agree on the value

sent by p. Let us see why, in this case, an additional round of messages (for each value proposed) solves the problem. There are two cases:

- *The sender of the value is faulty.* In this case p can send inconsistent votes to the loyal processes, as illustrated in Figure 7.11. However, since all the remaining generals are loyal, after exchanging among them the value received from the traitor they have exactly the same set of values. If they use a majority criteria, all correct processes will select the same final value.
- *The sender is correct.* In this case, p disseminates exactly the same value to all processes, as illustrated in Figure 7.12. When the remaining three processes exchange the value received, the two correct processes forward the value originally sent by p and the faulty process forwards some arbitrary value. Still, since there is a majority of correct values, the value originally sent by p will be chosen.

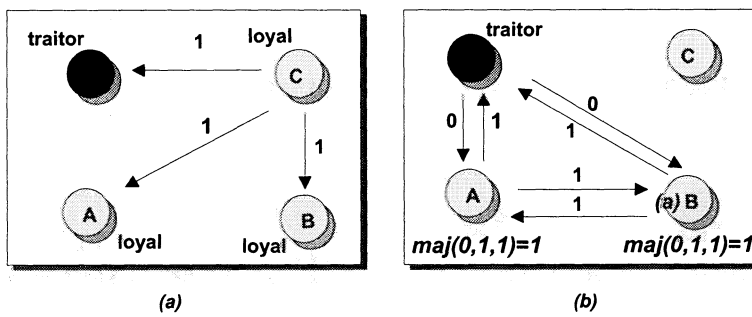

Figure 7.12. (a) First and (b) Second Rounds of Byzantine Agreement (partial view, correct sender)

Since after this second round of message exchanges every correct process has obtained exactly the same input from every other process in the system, correct processes can use a majority function to select the final decision of the Byzantine agreement. The interesting feature of this solution is that in order for the correct processes to agree on a common decision, they have first to agree on the input value provided by each of the processes in the system (correct or faulty). This recursive approach works because in the second round there is one less degree of freedom in the system.

Actually, this recursive approach can be extended to tolerate additional faulty processes. Let us call the complete protocol described above *Byzantine(1)* and the protocol executed in the second round *Byzantine(0)*. The protocol *Byzantine(1)* can be described as follows: first, for each process p, the remaining processes execute *Byzantine(0)* to agree on the value proposed by p, then they use a majority function on the agreed values to select the final decision. A protocol *Byzantine(f)* to tolerate f faulty processes (in a system with at least $3f+1$ processes) could be described using the same recursion: first, for each process p, the remaining processes execute *Byzantine(f-1)* to agree on the

value proposed by p, then they use a majority function on the agreed values to select the final value.

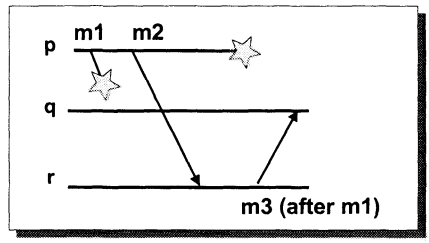

Figure 7.13. Causal Hole

7.5.5 Securing Causal Order

In Part I of this book, we have discussed a number of ways to enforce causal order (*see Ordering* in Chapter 2). Crash faults can affect these protocols in different ways. Consider the following example, illustrated by Figure 7.13. Process p sends a point-to-point message m_1 to process q, immediately after it sends another message m_2 to process r and then crashes. Message m_1 is lost but m_2 is received. Assume that there are no other messages involved. Can r deliver m_2? If local criteria are used, the answer is yes. Although $m_1 \rightarrow m_2$, m_1 was not sent to r, thus delivering m_2 does not violates causal order. However, if r delivers m_2 it will be *contaminated*: its state will causally depend on a message that cannot be recovered. A *hole* in the causal history will be created. Any subsequent message from r to q will not be delivered (it causally depends on m_1, which cannot be recovered).

To avoid contamination, before delivering m_2, a process must be sure that all preceding messages have been delivered or, at least, that there are enough copies of the message in the system to ensure its future delivery. Note that if $k + 1$ processes have a copy of the message, the message can be recovered even if k processes fail. Also note that this is the default behavior of a causal protocol when all the messages are sent to the same set of processes. Stability tracking protocols are useful to reduce the history to be kept.

7.5.6 Securing Total Order

Enforcing total order requires some form of agreement among the participating processes. In fact, the problem can be expressed in terms of having the processes agree by which order the messages should be delivered. As in the consensus problem, non-uniform and uniform versions of total order can be defined. There is more than similarity between these two concepts. In fact, it has been shown that uniform consensus and uniform atomic broadcast are equivalent problems in different classes of distributed systems. In other words, you can build a

```
Sender p:
    Initialization
        let S be the sequencer;
    procedure multicast(m) begin
        send UNORDERED(m) to S

Sequencer S:
    Initialization
        let sn := 0; // sequence number
    upon receive UNORDERED(m) from p
        sn := sn+1;
        send ORDERED(sn, m) to all recipients

Recipient q:
    upon receive ORDERED(sn, m) from S
        deliver m;
```

Figure 7.14. Simple sequencer based total order algorithm

fault-tolerant total order primitive based on a uniform consensus primitive but you can also build a consensus primitive based on total order.

This last reduction is actually very simple: all processes atomically broadcast their values and all processes agree on the first value delivered by the underlying atomic broadcast layer. Alternatively, a deterministic function is applied to the set of all process values, in order to extract the same value everywhere.

We have already discussed the challenges posed by failures in the agreement problem. These difficulties also apply to the total order protocols. Consider for instance the simple sequencer-based algorithm to enforce total order whose pseudo-code is shown in Figure 7.14. We recall that, in its simpler form, all processes send their messages to a centralized sequencer site, which then forwards them to all recipients. The delivery order is defined as the order by which messages are sent by the sequencer. When the sequencer fails, a new sequencer must be elected. However, some messages may have been received by some processes but not by others. For instance, in Figure 7.15, messages $m1$ and $m4$ are delivered to q and r but not to p. The new sequencer has to retransmit these messages in the same order to p, otherwise the system will be contaminated. In order to obtain ordering information, the elected sequencer must gather information about which messages have been delivered and in which order (assume that p would be elected as the new sequencer— it could learn about $m1$ and $m4$ through q and r). Since crashed processes cannot be inquired, the order established by the new sequencer may be different from the order of delivery in processes that have crashed. This also means that a simple sequencer algorithm cannot ensure uniform total order.

Enforcing total order using consensus as a building block can be done as illustrated in Figure 7.16 (we use the algorithm presented in (Chandra and Toueg, 1996)). Messages are disseminated using an unordered primitive guaranteeing that all correct processes will eventually receive all messages sent by

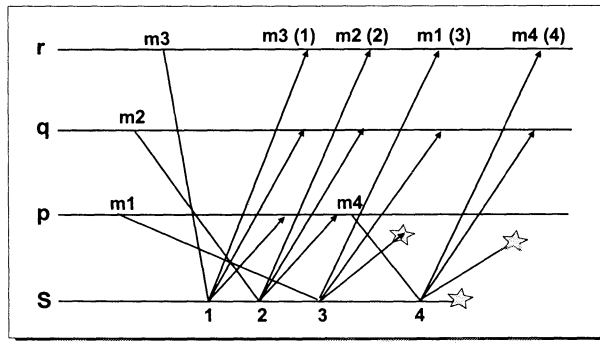

Figure 7.15. Sequencer-Based Total Order

```
Sender p:
    Initialization
        unordered ← ∅ ;
        ordered ← ∅ ;
        k ← 0 ;
    procedure total-order-broadcast(m) begin
        broadcast UNORDERED(m)

    when receive UNORDERED(m) from p
        unordered ← unordered ∪{m}
    when (unordered-ordered) ≠ ∅ begin
        k ← k + 1 ;
        propose (k, unordered-ordered);
        wait until decide (k, msg_set);
        ordered ← ordered ∪ msg_set; // deliver msg_set in some deterministic order
```

Figure 7.16. Chandra and Toueg's total order algorithm

correct processes (this can be obtained by letting each recipient forward each message to all other processes). Messages received this way are saved in a bag of unordered messages. To order messages, each process executes a sequence of consensus, each consensus orders a set of messages. For consensus number i, a process p proposes its bag of unordered messages. The result of the i^{th} consensus is the set of messages which all correct processes agree to assign sequence number i to. These messages are removed from the bag of unordered messages and delivered to the user according to some deterministic rule before a new round of consensus is started.

The fault tolerance aspects of some previously studied ordering protocols (*see Ordering* in Chapter 2) also worthwhile mentioning. The token-site total ordering method survives f failures by rotating the token and copying relevant ordering information $f + 1$ times, before considering a message stable. In Δ-protocols ordering information can be evaluated locally at every recipient. So

FT concentrates on two aspects: ensuring timely delivery; making clock synchronization fault-tolerant. Exceeding the assumed delivery delay or precision bounds can make the protocol fall apart.

The reader should also note that protocols providing nonuniform total order may also cause the contamination of the system. Consider for instance a sequencer-based total order algorithm. The sequencer receives two multicasts m_1 and m_2 and delivers them by that order. It subsequently multicasts a new message m_3 whose contents depends on the delivery order of m_1 and m_2. The sequencer then crashes before forwarding the delivery order to the remaining processes. The surviving processes receive m_1, m_2 and m_3 and decide to deliver those messages. In a nonuniform algorithm, the new sequencer is free to assign any order to these messages, for instance: $m_2 < m_1 < m_3$. Unfortunately, the contents of m_3 is now inconsistent with the selected delivery order. A uniform total order protocol prevents this case from occurring (but at a greater cost).

7.6 REPLICATION MANAGEMENT IN PARTITION-FREE NETWORKS

In the next few sections we will address a basic technique to provide continuity of service and/or availability of data: spatial redundancy in the form of replication. We begin by making strong assumptions about the system: the network is not subjected to partitions and the processes only fail by crashing.

7.6.1 State Machine

Before proceeding, it is useful to distinguish the behavior of components from the determinism point-of-view. The state and outputs of a *deterministic* component depend exclusively on its initial state and on the history of *commands* that it has processed. A deterministic component, also called a *state machine*, has been characterized as follows (Schneider, 1993):

> **Semantic Characterization of a State Machine**– Outputs of a state machine are completely determined by the sequence of requests it processes, independent of time and any other activity in the system.

Figure 7.17a depicts the principle of the state machine. If two components, executing the same state machine: are started with the same initial state; and execute exactly the same (totally ordered) sequence of commands; then they always exhibit the same behavior (as long as they remain non-faulty). It is also useful to distinguish *write* commands, that cause the state of the component to change, from *read* commands, that do not cause a state update.

On the other hand, the state and behavior of a *non-deterministic* component depend not only on the sequence of commands it executes but also on local parameters that cannot be controlled. Unfortunately, there are many mechanisms that can cause a non-deterministic behavior: non-deterministic constructs in programming languages such as the Ada select statement; scheduling decisions; resource sharing with other processes; readings from clocks or random number

generators; etc. The state of two non-deterministic replicas is likely to diverge even when they execute same sequence of inputs.

Between these two extremes it is also useful to define *piecewise deterministic components* (Strom and Yemini, 1985). The execution of these components can be decomposed into several deterministic sequences of instructions, intertwined with non-deterministic steps, such as the processing of a sporadic event or message. A state machine where command processing order depends on non-deterministic events such as urgent message arrivals or internal scheduling decisions is a piecewise deterministic component, where the granularity of the sequence is the command.

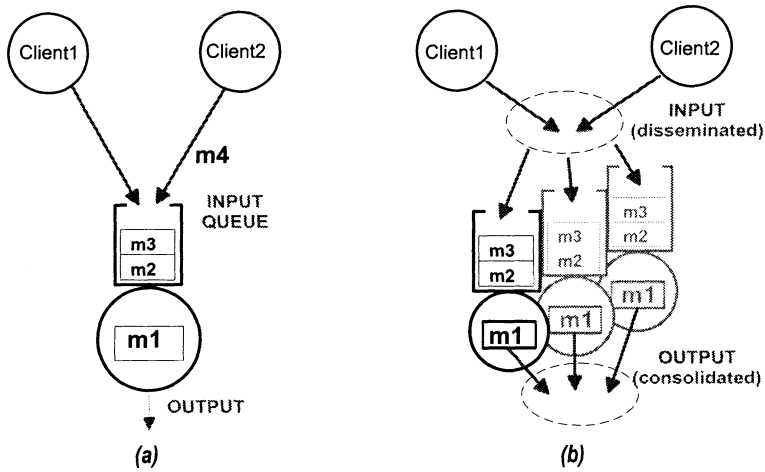

Figure 7.17. State Machine: (a) Simplex (b) Active Replication

7.6.2 Active Replication

Active replication is an intuitive technique that can be applied to state machines. It consists of having several replicas of the same state machine executed by different processes. In order to ensure that the state of these replicas is kept consistent, an atomic multicast protocol must be used to disseminate the state machine commands. The atomic multicast primitive ensures that all replicas receive the same commands in the same order, as illustrated in Figure 7.17b. When the state machine produces an output, all replicas produce the same result. The consolidation is simple in an omissive fault model: any of the individual results can be used (remember, no value faults yet).

When active replication is used, replica consistency is preserved implicitly by the atomic multicast protocol used to distribute commands. No explicit recovery procedure is required when one of the replicas fails: the remaining replicas will continue to provide service. Of course, the approach has its disadvantages. It is resource demanding, because each replica requires a full set of resources

to operate. Also, it requires the use of total order, which is intrinsically less efficient than weaker communication primitives. With regard to this last point, it is worth mentioning that only write commands need to be totally ordered: read commands can simply be causally ordered.

7.6.3 Semi-Active Replication

Active replication can only by applied to state machines. Semi-active replication is a variant of active replication that can also be applied to piecewise-deterministic components. Also called *leader-follower*, the idea is that all replicas execute the commands but a single replica (the leader) is responsible for making all non-deterministic scheduling decisions and provide this information to the other replicas (the followers) as illustrated in Figure 7.18a.

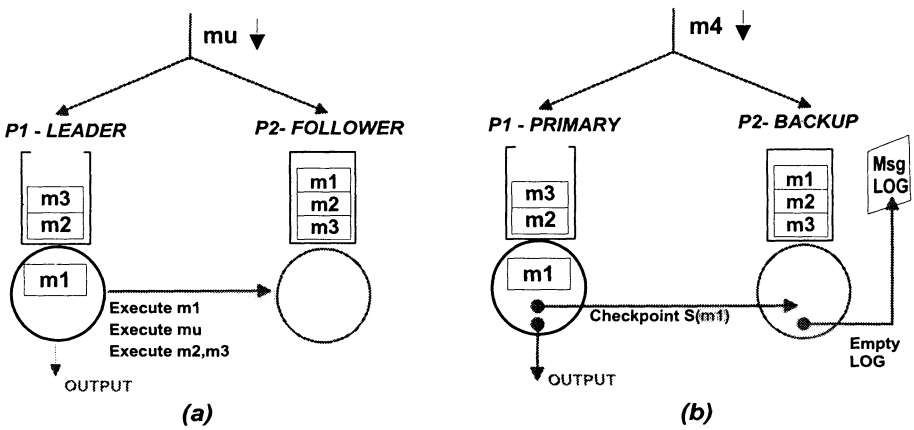

Figure 7.18. (a) Semi-Active Replication; (b) Passive Replication

Since the order of execution is defined by the leader, this technique does not require the use of totally ordered broadcast. Unordered unreliable broadcast can be used instead: note that the queues of the two replicas have the same messages, in different orders. Follower replicas log all messages but delay the processing until the leader sends them the result of its non-deterministic decisions, namely the execution order. For example, note that the leader sends instruction $\langle executem_1 \rangle$, then an urgent message m_u arrives and preempts the foreseen execution sequence (m_2, m_3). The next instructions from the leader will thus be to execute m_u, and then m_2, m_3. To avoid inconsistency, every non-deterministic decision taken by the leader must be disseminated immediately to the followers. So there is a tradeoff among the degree of consistency, the degree of non-determinism, and the cost of leader-follower synchronization.

7.6.4 *Passive Replication*

Both active and semi-active replication require the replicas to execute commands, and this doubles the resource requirements. Passive replication is more economic, since only one replica (called the *primary*) executes commands. The other replicas, called *secondary* or *backup*, remain in idle state, although they receive and log all commands. Periodically, the primary replica *checkpoints* its state in a location that can be read by the backups: it can send its state directly to to the backup replica(s) (using messages, as shown in in Figure 7.18b) or save it on a shared repository. All backup replicas clear their logs after a checkpoint. When the primary fails, one of the backup replicas is elected as a new primary and resumes operation from the last *checkpoint*, executing from the log to catch-up with the state of the primary just before failing. Depending upon the size of the replica state and upon the frequency of checkpointing, each checkpoint can include the complete state of the replica or just the updates from the last checkpoint (*incremental checkpointing*). The reader should note that there is room for tradeoffs between: size of checkpoint, size of log, frequency of checkpoints, recovery glitch.

7.6.5 *Lazy Replication*

Lazy replication (Ladin et al., 1992) can be seen as a hybrid scheme that mixes active and semi-active replication. The approach uses semantic knowledge to distinguish the operations that need to be executed with total order from those that can be executed in different orders in different replicas.

Clients forward their request to one of the replicas. If the request is an update that needs to be totally ordered, the replica synchronizes with the other replicas to establish a total order for that request; all replicas will apply the update in the same order and the contacted replica will reply to the client. The reply carries a vector clock that must be returned by the client in subsequent requests (this ensures that the client sees the updates in an order consistent with causality, even if it contacts different replicas). Requests that do not need to be totally ordered are executed in an order that respects causality by the replica that receives the request. A reply is sent to the client and the request is forwarded to the other replicas in background (thus, the name of lazy replication).

7.7 REPLICATION MANAGEMENT IN PARTITIONABLE NETWORKS

All of the replication schemes just discussed assume reliable failure detection. Besides, replica divergence is not avoided in the case of network partitioning. We will now present replication schemes that ensure consistent service even when some replicas become mutually unreachable. These techniques work in networks where partitions can occur and in systems where replicas can crash and later recover.

7.7.1 Static Voting

The idea behind voting is that any given operation should only be allowed to proceed if a minimum *quorum* of replicas, or copies, can perform it. Quorums must be defined in such a way that conflicting operations always intersect in at least one replica. This common replica is able to make the outcome of the previous operation available to the replicas executing the new operation. The most current state can be identified by having each replica maintain a *version number* that is incremented every time the data are updated.

The simplest example of a quorum algorithm for managing replicated data is one where read operations are allowed to read any single copy, and write operations are required to write all replicas of the object. This *read-one write-all* algorithm provides read operations with a high degree of availability at a very low cost. On the other hand, it severely restricts the availability of write operations since they cannot be executed after the failure of a single copy.

Weighted voting An extension to the above-mentioned scheme consists in assigning each copy a number of *votes*. Quorums are defined based on the number of votes instead of the number of replicas and the condition that guarantees overlapping consists in requiring that the sum of *quorums* for conflicting operations on an item should exceed the total number of votes for that item. This technique is the basis for a set of algorithms named *majority voting* (Thomas, 1979) and *weighted voting* (Gifford, 1979). Weighted voting is based on the following: given n the total number of votes for an item, and r and w the quorums required for read and write operations respectively, then it should be $2w > n$, and $w + r > n$. The intuition behind this can be explained by an example. Suppose $n = 7$, $r = 4$ and $w = 4$. Then, if a partition containing replicas summing at least 4 votes is written, only 3 votes are left, not enough to write divergently in other partitions (first condition). The second condition is similar, ensuring that one read and one write cannot be made concurrently (in two partitions), but are serialized regardless of how partitions develop. Namely, if the write occurs first, then the read is sure to include at least one of the replicas that have seen the previous write. This replica can update the others, ensuring sequential consistency of the history of operations. The separation between number of replicas and number of votes is the key point: a careful vote assignment taking into account the properties of each individual replica may yield improved results on the availability of the system. In the example above, suppose that replica A stored in a node with better reliability and/or connectivity is given 3 votes, and all other four replicas are given 1 vote: a partition with A and any other replica secures the majority of votes, allowing system operation to proceed even if a majority of replicas fail or partition.

Coteries An alternative approach to describing quorums, in particular weighted quorums, is to use an explicit set of processes, or *quorum groups*. The collection of quorum groups used by an operation is called the *quorum set*. To ensure overlapping, each group in the quorum set must overlap with every other group

in that set. A quorum set with such characteristics is called a *coterie* (Garcia-Molina and Barbara, 1985) and it can be used to achieve mutual exclusion. The reason for explicitly invoking an operation on a quorum group is that there are quorum sets which cannot be defined by voting algorithms. Although quorum sets are the most generic way to describe coteries, they are expensive: all quorum groups must be stored locally, and a search for the quorum group must be performed upon every operation. This is significantly more expensive than voting, where only the local vote and quorum need to be stored (and checked) at each process. Additionally, the number of quorum set alternatives is so vast that it is difficult, if not impossible, to choose the most appropriate quorum set for a given system. Due to this reason, other representations of quorum sets that are comparable to coteries have been searched.

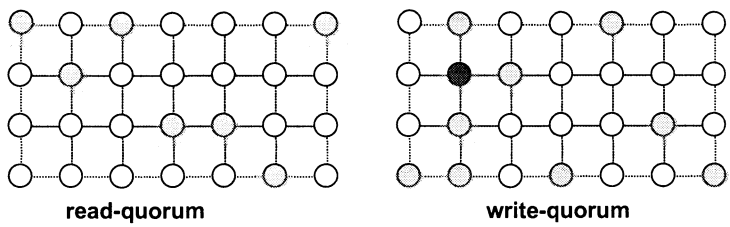

Figure 7.19. Grid Read and Write Quorums

Structural representations One of the methods to describe quorum sets is to use logical structures such as a tree, a grid, etc. Let us give some examples. The *tree quorum* algorithm (Agrawal and El-Abbadi, 1991) organizes replicas in a logical binary tree. The algorithm tries to find a quorum group by selecting a path from the root of the tree to any one leaf. If no such path can be found due to the inaccessibility of a copy c, that copy must be replaced by two paths, both starting at children of c (the algorithm is recursive). One problem with this algorithm is the probability of overloading the root node, since it belongs to all (failure-free) quorums. Another example is the *grid algorithm* (Cheung et al., 1990), that organizes the replicas in a logical rectangular grid: a read quorum must contain a node from each column and a write quorum must contain a read quorum and all the nodes in a column of the grid. In this way conflicting operations are guaranteed to overlap, as shown in Figure 7.19.

Byzantine Quorum Systems The systems described above are designed to tolerate benign faults such as crashes or partitions. Thus, the intersection of quorums for conflicting operations is required to have at least one process. It is possible to tolerate additional types of faults by using larger intersections. For instance, quorum systems that are able to mask Byzantine faults can be constructed by ensuring that quorums contain a majority of correct processes (Malkhi and Reiter, 1998). For instance, in a read operation, a client can accept

a value that is returned by at least $f + 1$ servers (and ignore arbitrary values returned by faulty servers).

7.7.2 Dynamic Voting

In the algorithms described above, quorum sets are chosen *a priori* at system design time. At run time, a node simply determines the reachable set of sites before performing an operation, and checks if this group belongs to the quorum set designated for the operation. In contrast, in a dynamic scheme, quorum groups can be changed during runtime. The idea is to use information about the current system configuration (such as which nodes are down) to adapt the algorithm in order to maximize some performance criteria. Typically, dynamic schemes attempt to make the system operate with small read quorums, since read operations are usually predominant. Consequently, the corresponding write quorums require a larger number of replicas and may become hard to reach when failures occur. In this case, after a failure the system should switch to another, more favorable, read and write quorum sets.

Most of the algorithms described in the previous section can be extended to take reachability information into account. Dynamic variants of voting algorithms are based on similar principles: partitions are identified in some form and updates are performed on replicas that are in the same partition. For instance, the weighted dynamic voting algorithm (Davcev, 1989) changes the majority criteria whenever a network partition is suspected. Whenever this criterion is changed, the version number of each copy at that point is stored (this information is called a *partition vector*). This allows to keep track of which replicas have participated in the most recent majority; the majority criteria can only be changed when a majority of up-to-date replicas are reachable. When communication is re-established, partitions are merged using the partition vector information. Usually, to avoid degrading the performance of read operations, a new partition is only installed during write operations (Agrawal and El-Abbadi, 1990).

7.8 RESILIENCE

The degree of resilience assumes at least two facets. The first is qualitative, and concerns the kind of faults to be tolerated, for example: whether or not the system can partition; or whether time- or value-domain faults are assumed. The second is quantitative, concerning the number of faults to be tolerated.

7.8.1 When to Compare Results?

Voting and comparison are common activities in fault tolerance paradigms. In the last section, we studied the use of voting to assure a tradeoff between consistency and availability: assessing the existence of progress conditions, such as a quorum or a majority of replicas or votes, in order to let a computation proceed without the risk of inconsistency.

In order to tolerate value faults, different sources of the same "logical" value must be available in the system, so that their values can be compared, or voted upon. Of all the techniques discussed so far, only those involving space redundancy, such as active replication, are able to provide tolerance of value faults, since the outputs are computed by each replica with complete independence from the other replicas. The correct sources will produce a valid value and the faulty sources an incorrect value. Voting on these results yields the correct value by masking the error, or at least allows detecting the error, depending on the amount of redundancy.

Voting is very simple when all correct values must be exactly the same and can be compared bitwise. For example, in the case of a single assumed fault and given a vector of values to be compared, a two-element vector (two replicas) allows detecting an error when the two entries are different, but there is no recovery, since it is a no-winner situation. A three-element vector allows masking one error, by deciding for a majority (at least two) of equal values.

7.8.2 Exact and Inexact Agreement

We have discussed how to pick a correct value from a set of values produced by different replicas. In many fault-tolerant architectures, the consumer of the value is also replicated, and distributed. This means that, in order to preserve consistency, all consumer replicas need to select not only a correct value but also the same correct value. In other words, the consumer replicas need to *agree* on the correct value. In omissive failure systems, if all producer replicas atomically broadcast their values, consumer replicas end up with the same vector of producer replica values, and thus can do a deterministic comparison, arriving at the same value. As a matter of fact, this is another way of reaching consensus through atomic broadcast (*see Securing Total Order* in Section 7.5).

This task can be further complicated if a faulty source can send different values to different replicas (byzantine faults). Note that we have already discussed a means of overcoming this problem. Byzantine Agreement (BA) allows a value to be reliably distributed through a set of consumer replicas under such a failure mode (*see Tolerating Arbitrary Faults* in Section 7.5). If all producer replicas execute BA, then an Interactive Consistency (IC) vector is built (Pease et al., 1980), with exactly the same content at every consumer replica. A deterministic comparison on the IC vector will yield the same result at all replicas.

Exact (bitwise) agreement cannot be reached when two correct replicas can produce different values. How can this happen? Consider, for instance, that the vector of values to be compared is the result of analog sensor readings. Then, any two correct sensors can read values that are slightly different and thus not comparable bitwise. In consequence, one has to use some "convergence function" performed on the whole of the vector, in order to pick the "right" value, which may be neither of the initial values. Besides, in the case of replicated consumers and value faults, the result of the function may be slightly different from replica to replica. This is called *inexact agreement.* Clock synchroniza-

tion, that we discuss in the Real-Time part of the book, is another example of inexact agreement.

Several convergence functions exist, and some may be sophisticated and highly dependent on the application semantics, for instance, when the value to be produced is a captured image. For the sake of example, we will describe a couple of simple convergence functions for real numbers, that will tolerate up to f faulty values in the vector:

Fault-tolerant Midpoint - selects the midpoint of the values collected after discarding the f highest and f lowest values. Requires at least $2f + 1$ values, $3f + 1$ with byzantine faults

Fault-tolerant Average - selects the average of the values collected after discarding the f highest and f lowest values. Requires at least $2f + 1$ values, $3f + 1$ with byzantine faults

7.8.3 How Many Replicas or Spares?

It is probably worthwhile making a point of the situation in terms of the number of replicas that are really needed to achieve fault tolerance. This number depends on the number and type of faults that need to be tolerated. In order to tolerate f omissive faults, $f + 1$ replicas are needed. To tolerate f value faults, $2f + 1$ replicas are needed, since a majority vote must be made on the value to return. This number is still valid for distributed replicas with value faults, provided that the communication subsystem only does omissive faults. However, it goes up to $3f + 1$, in order to tolerate Byzantine faults.

These numbers represent the minimum number of replicas required. The question now is the following: should we use the bare minimum or should we use additional redundancy?

7.8.4 The Point of Diminishing Returns

Given available resources, one may be tempted to introduce more redundancy than strictly required. This may have several benefits. To start with, it may increase the system reliability, since additional redundancy allows the system to tolerate additional faults. Also, in several replication schemes, additional redundancy may provide load balancing, for instance, by executing reads in parallel on different replicas.

However, additional redundancy also means additional costs and complexity to ensure consistency. For instance, the algorithm proposed in (Lamport et al., 1982) is optimal in the number of rounds $(f + 1)$ but requires an exponential number of messages to reach Byzantine agreement $(O(n^f))$. Even load balancing may not be a universal advantage, since it may penalize some classes of operations: in a read-one/write-all strategy to replicated data, the availability of the write operation decreases with the number of replicas.

On the other hand, adding more components means increasing the probability of having a component failed at a given point in time. Thus, after a certain point the introduction of an additional replica may not produce an significant

increase in the overall system reliability. It is also important to note that, as we have seen previously, adding additional components is not the only way to increase the system reliability. The other alternative is to use better components. Using analytical models it is possible to find the point where additional redundancy is no longer cost effective and components with higher coverage should be used. This has been called the *point of diminishing returns* (Stiffler, 1978).

7.9 RECOVERY

Most of the techniques that we have discussed so far try to ensure the availability of a correct result. In many systems with less stringent requirements, it is enough to ensure that after a failure the components are able to restart the computation from a consistent state (ideally, not far from the crashing point). Even when availability is a primary concern, it is interesting to ensure that not all of the state is lost in the case where all or some components crash.

Recovery from crashes requires the application state to be saved in *stable storage*. Stable storage entails both the notion of persistence, i.e., surviving the entity creating it, and of reliability, i.e., exhibiting very low probability of losing or corrupting information. Stable storage is usually implemented on non-volatile media (disks, or combinations thereof). However, note that a set of replicated volatile memory repositories can act as a stable storage subsystem. Resilience (persistence and reliability) is secured by ensuring that at least one of those replicas remains operational at all times. The reader should note that passive replication can be seen as a form of "saving state to stable storage".

7.9.1 Stable Storage

Stable storage built from volatile memory is simple and solves several problems, namely as a medium-term repository, for example for long computations, or publisher-subscriber message buses. However, it is not the general approach. This is because truly non-volatile media have a better coverage against unexpected events, such as power breakdowns, and other common-mode failures. How should we build a stable storage device? It is time to apply some of the concepts we have discussed so far, namely the hierarchical nature of fault-tolerant systems, now at the disk level.

The simplest form of stable storage would be a disk. But there are a lot of failures that can affect the information on that disk. Value and space redundancy can be used in order to preserve the consistency of information despite disk errors. For example, the same information can be stored into two different disk blocks and a checksum added at the end of each disk block. If a crash occurs in the middle of a write operation, or if a block is somehow corrupted, the error can be detected from the invalid checksum, and the "last" value can be recovered from the other block. The crash of the whole disk can be tolerated again with space redundancy: the information can be written in two separate disks (this approach is also called "mirrored disks").

An interesting technique that increases both the availability and the performance of the stable storage consists in using a Redundant Array of Inexpensive Disks (RAID) (Patterson et al., 1988). According to this approach, the stable storage container is made of n storage disks and a parity disk. The information is scattered among the storage disks and the parity of the n storage blocks is saved in a correspondent block in the parity disk. If one of the storage disks fails, its content can be recovered by resorting to the parity information. In the current implementations, the parity blocks are also scattered uniformly among all the disks to prevent the parity disk from becoming a bottleneck (this disk would have to participate in all writes).

7.9.2 Checkpointing

The stable store we just studied can be used to save the state of the components that need to survive a crash. The state that is saved in stable store is also called a *checkpoint*. Upon recovery after having crashed, the component reads the last checkpoint and resumes operation from there: this operation is called a *rollback*. As you can see, the basic idea behind this technique, called *checkpoint-based rollback-recovery*, is quite simple, almost too good to be true. However, there are some difficulties that need to be surmounted to render this technique useful.

A very important point that needs to be taken into account is that components do not operate in isolation in a distributed system. They interact with other components by exchanging messages with them. To the recovering process, any messages it sent between the last checkpoint and the crash instant simply do not exist! In some sense, the recovering process will behave like someone who has lost his memory after a car accident and is unable to remember his own wife and children. However, any other component having received those messages will then be inconsistent, since "apparently" no one sent them.

So what is the solution? Fortunately, there is something that can be done with a computer program that cannot be done with humans: travel back in time. Consider the crash and recovery of a component. If all components have periodically checkpointed their state, we can force them to rollback in such a way that some past *consistent global state* is re-established which includes the recovering process. Thus, recovery requires the search for the earliest consistent set of checkpoints, called *recovery line*.

The extensive literature in this area can be classified into three main approaches for taking checkpoints. The first, called *coordinated* checkpointing, consists in having the processes coordinate before taking the checkpoint. The purpose of coordination is to ensure that the set of checkpoints is consistent (*see Consistent Global States* in Chapter 2). The advantage of always taking consistent checkpoints is that, in the case of crashes, the system only needs to rollback to the last checkpoint. On the other hand, coordination itself may introduce some delays in the execution of the application.

Uncoordinated checkpointing avoids coordination costs by allowing processes to take checkpoints independently from each other. Unfortunately, with uncoordinated checkpointing there is no guarantee that a set of consistent check-

points will exist. We recall that two checkpoints C_1 at process p_1 and C_2 at process p_2 are mutually inconsistent if C_1 contains the message m sent by p_1 to p_2 but C_2 has no record of sending this message. It is easy to see that this scenario may repeat itself: the rollback of one process may force the rollback of another process, which in turn forces the first process to rollback again, and so on. This phenomenon, called *domino effect*, may cause the system to rollback to its initial state (Randell, 1975). The effect is shown in Figure 7.20: p_1 fails, and then recovers, rolling back to checkpoint c_a. Evidence of sending message m_i no longer exists, and so, p_2 is forced to rollback to checkpoint c_b. However, this "unsends" message $m_j$and thus p_3 is forced to pull back to c_c. We leave it to the reader to confirm that *rollback propagation* will bring the system back to the initial state.

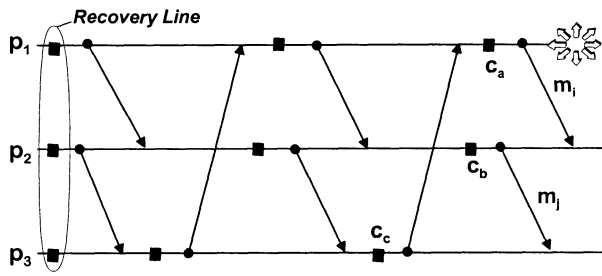

Figure 7.20. Domino Effect

A third technique, also not requiring coordination, is called *communication-induced* checkpointing. It avoids the domino effect by requiring components to checkpoint upon receiving and prior to processing certain messages that may induce conflicts.

Finally, it is worthwhile mentioning that if the checkpoint is made not to a local store, but to one or several additional component replicas in different sites, then it is not necessary to wait for the component to recover after a crash: recovery may start in one of the replicas immediately the crash is detected. This is the mechanism underlying passive replication (*see* Section 7.6).

7.9.3 Logging

Many applications could live with a system that simply checkpoints periodically, often enough that the rollback delay does not become too large. Nevertheless, there are some problems that limit the effectiveness of rollback. The most obvious is when the computation is not fully deterministic: upon recovery the component is not guaranteed to reproduce exactly the same steps and results it has produced before the crash. Also, certain actions done since the last checkpoint may have left a trace outside the subsystem of components involved in the checkpointed computation. Actuations on the environment or messages sent out are good examples of what we may call *real actions*: these cannot be undone, and rolling back and repeating them may cause inconsistent behavior.

This problem could be avoided by systematically checkpointing whenever such an action is performed. However, one of the problems with checkpointing is that it consumes time. If the component holds a large state, or if the above-mentioned actions are too frequent, the performance overhead of checkpointing may become unbearable. Checkpoints may be optimized of course, namely by selecting the state variables to checkpoint, e.g., by using application semantics and/or compressing the data to be stored.

However, if the computation is piecewise deterministic (*see* Section 7.6), there is a systematic way to minimize the number of required checkpoints, by *logging* all non-deterministic events between consecutive checkpoints. The state of the component can then be reconstructed from the most recent checkpoint and the log: the state is first recovered from the checkpoint and then all events from the log are replayed in the same order as before the crash. This technique is called *log-based rollback-recovery*. This may allow the system to recover without forcing other components (or the outside world) to rollback. Three major approaches to log-based rollback-recovery have been proposed: pessimistic logging, optimistic logging, and causal logging.

Pessimistic logging systems make sure that information about each non-deterministic event is logged before the event affects the computation. As a result, when a process crashes and recovers it is guaranteed to execute the same sequence of events, reaching a state that is consistent with the (internal or external) actions performed prior to the crash. Unfortunately, the number of log operations that are inserted in the process path may represent a significant performance overhead. To avoid these costs, it is possible to log information about non-deterministic events asynchronously. This means that the computation proceeds and the logging is performed later. The idea is that since faults are not very frequent, all log operations succeed in most cases. Thus, these protocols are also called optimistic protocols. Unfortunately, once in a while a process may crash before logging all non-deterministic events. Upon a failure, the maximum consistent state must be recovered from the last global consistent checkpoint and from a sub-set of events recorded in the local logs. To achieve this state, other processes may be forced to rollback to obtain a consistent global state. In consequence, interaction with the outside world requires explicit synchronization among processes. The third alternative, causal logging, keeps track of causal relations among events, retaining most of the advantages of optimistic logging without making optimistic assumptions. The description of causal logging is outside the scope of this book (for a survey, see (Elnozahy et al., 1999)).

Finally, garbage collection is also an important matter, since both logs and checkpoints claim resources. Uncoordinated checkpoint protocols identify the recovery line and dispose of all checkpoints past it. Coordinated checkpoint protocols dispose of all checkpoints but the most recent. Logs are deleted immediately a new checkpoint is made.

7.9.4 *Atomic Commitment and the Window of Vulnerability*

The last sections have discussed recovery approaches based on actions that can be rolled back individually. A step forward would be to encapsulate these actions in sequences that cannot be undone individually, and have the system automatically guarantee this. Atomic transactions achieve this effect: if a transaction commits, its effects are durable. A protocol that ensures such properties is called an *atomic commitment protocol* (*see Distributed Atomic Commitment* in Chapter 2), and what we discuss here is how much we can rely on such a protocol, in the event of process crashes and recoveries.

Recapitulating, the processes participating in a distributed transaction must coordinate their actions (and checkpoints) to ensure that a committed transaction is not erroneously aborted upon recovery. The *two-phase commit* protocol is one of the most used atomic commit protocols (Figure 7.21). We revisit the algorithm with more detail. This protocol is coordinated by one of the participants. In the first phase, the coordinator sends a PREPARE message to all other participants. Upon reception of a PREPARE, a process checks if it is ready to commit the transaction. If the answer is affirmative, it ensures that all the results are logged in stable storage, appends a *prepared* entry to the log and replies OK to the coordinator. Otherwise, it aborts the transaction and replies with NOTOK. If the coordinator receives an OK from all participants it commits the transaction. Otherwise the transaction is aborted. A *committed/aborted* entry is added to the coordinator log and the log is forced to stable storage. The coordinator then initiates the second phase, sending a COMMIT/ABORT message to everybody. Upon reception of the COMMIT/ABORT, each participant commits/aborts the transaction and sends an acknowledgement back to the coordinator. To ensure a fast dissemination of the transaction's outcome, the coordinator retransmits the decision if some acknowledgements are missing.

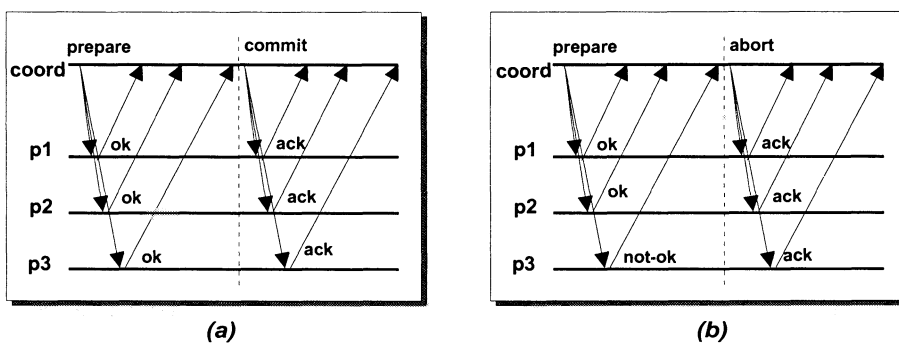

Figure 7.21. Two-phase Atomic Commitment Protocol: (a) commit; (b) abort

The two-phase commit protocol is quite simple and efficient. However, it suffers from a major drawback: if the coordinator fails between the PREPARE and the COMMIT/ABORT, the remaining participants will be blocked waiting for the decision! They cannot abort the transaction either, because the coordinator

might have said COMMIT before failing, and a subset of the participants might have committed and then failed. Only when the coordinator recovers can a safe decision be taken. These failure scenarios may also take place if the system partitions.

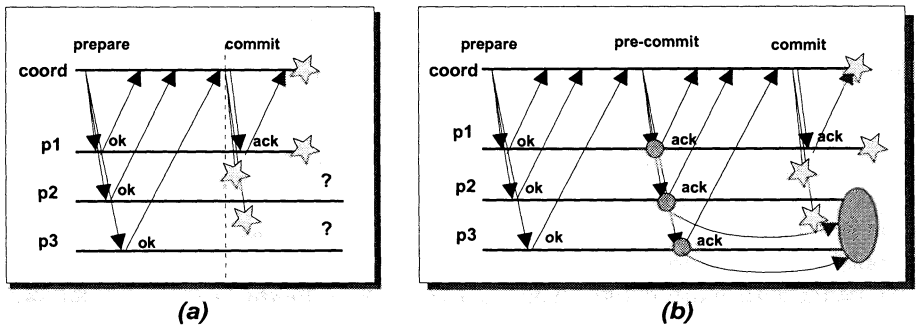

Figure 7.22. (a) Blocking of Two-phase Commit; (b) Three-phase Commit

A non-blocking solution to the problem exists, and is known as *three-phase commit protocol*, exemplified in Figure 7.22b. The idea behind three-phase commit is to delay the final decision until enough processes "know" which decision is about to be taken. Namely, between the two initial phases the coordinator sends a PRE-COMMIT message to all processes and waits for an additional round of acknowledgements. Only then the COMMIT is sent. The advantage of this scheme is that even if the coordinator fails before issuing the commit, the remaining processes may resume the operation since they have received the PRE-COMMIT message. Three-phase commit is much more resilient than two-phase commit, at the cost of performance.

The reader might have already noticed that there are similarities between the atomic commitment problem and the agreement problem. If fact, atomic commitment can be seen as a form of agreement with some restrictions: all participants must agree on the outcome of the transaction, with the proviso that the outcome can only be commit if all participants are ready. Actually, since we want to prevent failed processes from disagreeing with active processes, atomic commitment is a variant of uniform agreement. Non-blocking solutions to the problem of atomic commitment were known before the problem of uniform agreement was well understood. As we have already discussed, there are no deterministic solutions to the (uniform) agreement problem in asynchronous systems. Thus non-blocking actually means: 'non-blocking as long as a majority of processes remain correct'.

Actually, an elegant way of describing an atomic commitment protocol is to use consensus as a building block. The protocol can then be rephrased as follows. The first phase proceeds as before with a minor difference: one participant is responsible for sending a PREPARE message but responses are multicasted to all participants. The subsequent phase is decentralized and based on the execution of consensus. In order to reach a decision about the outcome of the

transaction, all participants propose an initial value to consensus. Participants that collect OK from all other participants propose to COMMIT the transaction. Participants receiving a NOTOK or suspecting that some other participant has failed propose to ABORT the transaction. The consensus itself guarantees that all correct participants agree on the same outcome.

7.9.5 State Transfer

We now consider the case when the component recovers and can be re-integrated in the replica group in order to re-establish the original number of replicas. The state of the recovering process has to catch-up with the state of the remaining replicas. This problem requires specialized protocols, called *state-transfer* protocols, whereby one of the active replicas transfers its state to the recovering one. State transfer poses both practical and conceptual problems.

The more practical problems are concerned with the identification, capture and physical transfer of the state. For the sake of performance, one should try to transmit only the variables relevant to the recovering replica. If the replica only has volatile storage, everything must be transferred. However, it may be able to load code and read-only tables from disk, and re-initialize the computation. This minimizes the amount of data to be transferred. Application-specific state-transfer opens further opportunities for optimization, since the programmer knows better than anyone else what to transfer. For example, suppose that the state of a replica depends exclusively on the last n commands it executed: for instance a component that keeps an average of the last k sensor readings. In this case, replica integration can be performed just by waiting (after k sensor readings, the state of the replica is updated).

If replicas checkpoint/log to stable storage, the recovering replica can implement incremental transfer, i.e., just transfer the results of the changes made during the crashed period. This requires the recovering replica to recover first from the last checkpoint/log in stable storage, and then from the active replicas. These have to take the necessary steps to keep a history of changes since the failure of a replica is detected, until it recovers.

The conceptual problem is that ideally state-transfer should be performed with minimal interference with the behavior of the remaining replicas. That is, active replicas should not be prevented from providing service to clients while the state is being transferred. The problem with this approach is that the state is a moving target: it is being changed at the same time it is being transferred! On the one hand, this may entail incorrect system state transfer. On the other hand, if the active replicas change their state faster than they are able to transfer it, the joining replica will never be able to catch-up.

The former problem may be solved by giving priority to state transfer in detriment of replica computations, which may momentarily slow the computation down. As for the latter problem, we now describe a simple way to perform state-transfer in systems where total order is used. The protocol is exemplified in Figure 7.23. The recovering replica (p_3) initiates the procedure by resuming communication with the replica set. At this point, if the set was using some

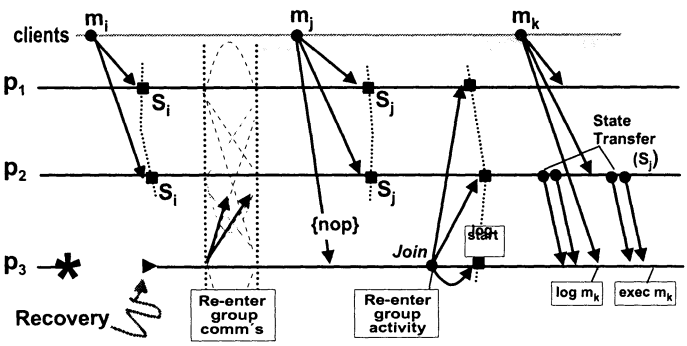

Figure 7.23. Recovery with State Transfer

form of group communication, it starts receiving all messages, but still discards them. Next, it multicasts a request to JOIN the replica group activity, which triggers a state-transfer operation. This message is delivered in total order to all replicas, including the joining replica, marking a cut in the global system state: one of the active replicas (p_2) checkpoints its state at this point (S_j), and sends it to p_3 in one or more STATE messages; p_3 starts logging any messages that arrive after the cut, since they follow S_j. New requests from clients now arrive at all replicas (e.g., m_k), and can continue to be processed by all replicas except the joining one. The joining replica logs all client requests until it receives the last STATE message. Then, it consumes all pending requests in the log, in the order by which they were received. At this point, state transfer is complete, and all replicas are consistent. It should be clearer now why it may be of interest to slow down the computation a bit during state transfer: if the log is enormous after state transfer, the recovering replica may take very long to catch-up. Why is this a problem? Remember that the whole purpose of recovery was to re-establish the degree of replication, which is only achieved when transfer is complete.

7.9.6 Last Process to Fail

Assume a replicated computation following an optimistic replication strategy. As the name implies, availability is the primary concern in this approach, and so as long as one replica is active the service is not interrupted. Replicas have some persistent state that survives failures. As replicas start failing, their persistent state will be made obsolete by the progress of the surviving replicas. If all fail before anyone of them has the chance to recover, the last replica to fail has the most up-to-date state. Upon recovery, operation cannot be resumed before this replica recovers, otherwise the last updates would be lost.

The question is: when a process recovers how can it know that it was the last process to fail? Let us assume that each replica keeps a *version* number that keeps track of the number of changes performed that replica state. If all

recovering processes compare their version numbers, the one with the highest version number knows it was the last replica to fail. The problem with this approach is that all replicas need to recover in order to perform the comparison. It would be more interesting to resume the operation as soon as the last replica to fail recovers. How to do this?

A solution exists to this problem (Skeen, 1985). Consider that each process is able to detect the failure of other processes. When process i detects that some other process j has failed, it adds this information to a local "obituary" log which is saved in stable storage. The last process to fail is the process that has registered the death of every other process in its own obituary log.

7.10 SUMMARY AND FURTHER READING

This chapter has discussed the main challenges in achieving correct operation of a distributed system in the presence of faults. Several basic paradigms were addressed, such as: failure detection, partitioning, membership and consensus, agreement and order of message delivery. If these problems are solved, one can tolerate faults through the use of replication. Replication has several coordination aspects, which depend on the underlying system model: partition-free or partitionable; crash or crash-recovery. Resilience and recovery finalized the paradigms addressed in this chapter.

For further study, a fundamental reference on the problem of failure detection in asynchronous systems is (Chandra and Toueg, 1996). The QoS aspects of failure detectors are discussed further in (Veríssimo and Raynal, 2000; Chen et al., 2000). For partitionable programming models, see (Amir et al., 1993a; Cosquer et al., 1996; Babaoğlu et al., 2000). Partitioning in synchronous systems is further discussed in *Real-Time Communication Models*, Chapter 13. More information on consensus can be found in (Fischer et al., 1985; Turek and Shasha, 1992). The problem of group membership has been addressed under several system models, namely (Birman and Joseph, 1987; Cristian, 1988; Kopetz et al., 1989b; Jahanian and MoranJr, 1992), or (Amir et al., 1992; Golding, 1992; Rodrigues et al., 1993).

There is also a huge amount of literature on fault-tolerant communications, in particular, on fault-tolerant group communication, such as (Amir et al., 1993a; Birman and Joseph, 1987; Birman et al., 1991a; Chang and Maxemchuck, 1984; Dolev et al., 1993), or (Kaashoek and Tanenbaum, 1991; Ladin et al., 1992; Moser et al., 1995; Rodrigues et al., 1996; Rodrigues et al., 1998a). Deeper study on fault-tolerant communication paradigms can be found in (Hadzilacos and Toueg, 1994).

About the state-machine approach, the excellent tutorial of Schneider is recommended (Schneider, 1993) and for other examples of active and semi-active replication see (Barrett et al., 1990; Chereque et al., 1992). Examples of the primary-backup approach are given in (Speirs and Barrett, 1989; Budhiraja et al., 1993). Improved tree and grid quorum voting algorithms are discussed in (Kumar and Cheung, 1991). Multidimensional voting techniques have also been studied as alternative representations for quorum sets (Ahamad and Am-

mar, 1991). For dynamic variants of grid algorithms, see (Paris and Sloope, 1992), which allows the grid to be re-organized based on reachability information. Concerning checkpointing, see (Speirs and Barrett, 1989; Kim and You, 1990; Wang and Fuchs, 1992; Silva and Silva, 1992), or (Elnozahy and Zwaenepoel, 1992a; Alvisi and Marzullo, 1993; Alvisi et al., 1999) for further study. A portable checkpoint protocol implementation tool, based on MPI, is described in (Neves and Fuchs, 1998). An excellent and comprehensive survey can be found in (Elnozahy et al., 1999).

8 MODELS OF DISTRIBUTED FAULT-TOLERANT COMPUTING

This chapter illustrates how the paradigms discussed in the previous chapter can be applied and combined to achieve fault tolerance in an application-oriented way. The chapter starts by introducing classes of fault-tolerant systems that make different assumptions about the system properties, from arbitrary to crash and from asynchronous to synchronous. Then, it discusses strategies for the several approaches to building a fault-tolerant architecture. The main models for building fault-tolerant systems are then presented: remote operations, event services and transactions.

8.1 CLASSES OF FAILURE SEMANTICS

We have discussed in Chapter 6 that failures can be classified according to many different criteria. We have seen in Chapter 7 that the solution for a given problem, for instance distributed agreement, strongly depends on the class of failures the system is subjected to. In this chapter we consider the four main broad classes of failure semantics (or failure modes): arbitrary, crash, omissive, and crash-recovery. More refined classes may be derived from the former to satisfy particular assumption requirements.

8.1.1 Arbitrary Failures or No Assumptions

If you build a system under the assumption that components can fail in an arbitrary manner, you are walking on the safe side of the road. Of course, you

P. Verissimo et al. *Distributed Systems for System Architects*

still have to forecast how many failures can occur, in order to determine how much redundancy to use. Likewise, some environmental and/or infrastructural assumptions are still likely to be made, such as drift of clocks, for example. Nevertheless, the arbitrary assumption approach strongly reduces the chances of unexpected failures, because the coverage of the general system tends to one. Unfortunately, a safer road is also a longer road. Just recall the high number of nodes, messages and rounds required to solve Byzantine agreement (*see Tolerating Arbitrary Faults* in Chapter 7).

On the other hand, assuming arbitrary failures leads to architectures with the lowest performance/price ratio. *Then, when should we choose this model?* To start with, when failures can be catastrophic, and this usually means when human lives or huge amounts of money may be at stake. The highest possible coverage is desired for these systems. Secondly, when the system will be under the threat of malicious attacks. The intruder may manipulate some components in an unpredictable way, attempting to defeat the systems defenses. The arbitrary failure mode implies that rather than trying to guess the intruder's moves, the system is prepared to tolerate any type of behavior.

8.1.2 Fail-Silence or Crash

On the other extreme of the spectrum, the fail-silence mode, often called crash mode, assumes that a component works perfectly until the moment when it suddenly dies, or *crashes*. It does nothing, good or bad, after crashing. This is a widely used failure semantics, specially applicable to components such as processes or processors, which tend to fail in this way. Note that this does not mean that erroneous events cannot happen internally to the component before it crashes. It only means that we expect the internal erroneous behavior associated with the fault never to become visible at the component interface(s). For instance, the address space of a process may be corrupted in an arbitrary manner just before a crash due to an address violation exception; it still looks like a crash if the process does not produce any erroneous output.

The fail-silence assumption simplifies the design of fault-tolerant protocols (*see Fault-Tolerant Communication* in Chapter 7), but should be used with care. Many designers are tempted to postulate that off-the-shelf components are fail-silent, just because they do look as if they were (most of the times). However, experiments have shown that the coverage of this assumption is not very high. In consequence, the system architect should not expect too much from the dependability of systems built to this assumption, unless specific fault-prevention measures are embedded in the design of the system components (e.g., self-checking, wrapping). The level of sophistication of these measures depends of the desired coverage: from hardware (e.g., parity checks, watchdogs) to programming (e.g., systematic validation of types, bounds, message fields, etc.).

8.1.3 Weak Fail-Silence or Omissive

The fail-silence assumption is a bit too strong since it does not consider frequent failures in the omissive class, such as omissions or timing, which are extremely common for certain components in distributed systems, and certainly unavoidable in simplex (non-replicated) networks. The *weak fail-silence* mode establishes the following behavior for a component: (a) it is correct, or else fails by crashing; (b) 'correct' is a non-ambiguous specification of controlled omissive failures. For instance, a weak fail-silent network component specification could be:"during a reference interval, it does at most k omission failures or else it crashes". This could be read as: "components fail by crashing, but while alive they may give up to k successive omissions", which happens to be very appropriate to describe the behavior of most networking components of a distributed system.

Techniques to tolerate omissive faults are easy to implement, and in consequence, it is little more complex to design weak fail-silent systems than it is to design their pure fail-silent counterparts. However, the coverage of properly chosen weak fail-silence assumptions is bound to be higher, for the same system complexity. Many practical systems assume a mixture of fail-silent and weak fail-silent components. Generalizing to less benign failures, whenever a component exhibits a definable failure mode that is lesser than arbitrary, we say it is a *fail-controlled* component.

8.1.4 Crash-Recovery

Most systems assume that components recover sooner or later. With some notable exceptions (such as probes sent to remote locations of the solar system), one can repair or replace failed components. Even in spatial vehicles, some components can reset and recover, such as in the Mars Rover story (*see Resource Conflicts and Priority Inversion* in Chapter 12). The difference between the crash and the crash-recovery modes is that in the first model, the recovering component has no memory of its past existence and has to start anew, whereas in the crash-recovery model the recovering process preserves some of its old state (e.g., up to the last checkpoint made).

Algorithms for the crash (no-recovery) model are concerned with keeping the consistency of processes that remain alive and rely on state-transfer mechanisms to re-integrate recovering components. Algorithms for the crash-recovery model extend consistency requirements to failed processes. Thus, the later tend to require the use of the "uniform" versions of distributed algorithms (e.g., uniform atomic broadcast or uniform consensus, *see Consistency* in Chapter 2).

8.1.5 Synchronous and Asynchronous Models

Timing assumptions are extremely important in a distributed system because they are directly related with the problem of accurate failure detection. With this regard, distributed systems have been classified into synchronous or asynchronous systems, depending on whether or not known bounds on processing,

communication delays and clock rate of drift exist (*see* Sections 3.3 and 3.4 in Chapter 3).

The main issue concerning the choice of model with regard to fault tolerance is coverage. Designers frequently postulate a synchrony model (e.g., fully synchronous) and then go about making their design, focusing on the failure assumptions and algorithms to handle them. Very often, the design fails not because some severe fault occurred or too many faults took place, but simply because the synchrony assumptions of the model were violated and that caused the algorithm to misbehave (e.g., misuse of timeouts in the fully asynchronous model, which is in essence time-free; or neglecting timing failures, in the fully synchronous model). Partial synchrony models are in between the former two, trying to take the best of both worlds (*see Between Synchronism and Asynchronism* in Chapter 13). They seem to have an instrument that neither of the others have: they assume *timing failures* in the model. This allows to integrate the synchrony and failure models, making possible a seamless design for fault tolerance.

8.2 BASIC FAULT TOLERANCE FRAMEWORKS

Fault tolerance frameworks provide the grounds for the architect to start the construction of a fault-tolerant system. This section discusses the several vectors along which the architectural work on distributed system fault tolerance may develop. Namely: hardware FT; software-based hardware FT; software FT; communication. In the course of the discussion, it will naturally establish pointers between the paradigms discussed in the previous chapter, and the models to be discussed further ahead.

8.2.1 Hardware Fault Tolerance

The aim of hardware fault tolerance is to tolerate hardware faults using hardware mechanisms. It usually consists of having low-level mechanisms to detect and recover or mask errors. The simplest one is self-checking, which we introduced earlier in this part in Figure 6.7a. Examples of self-checking mechanisms that can be implemented in an efficient way by hardware are parity bits, verification of assertions, checking ranges of memory addresses, etc. When a component is detected to be failed it is stopped. In result, either the system is halted or the system has enough redundancy to continue operating. One approach is to keep a set of spares aside, and enhance the system with a new component, the *supervisor*, which performs a *switchover* from the failed component to a spare, in an automatic manner. When the spare is only started after the failure of the original component is detected, this is called "cold standby", "hot standby" if otherwise. When the technique is applied to stateless components, the cold standby can provide service as soon as it is started. For stateful components, the standby must first "catch-up" with the state of the failed component at the moment of the failure, the *takeover* process, which may entail a glitch in the provision of service.

A natural evolution of the previous approach consists in having two or more component replicas operating in parallel. For instance, if three replicas are used, and their results compared, the unit that compares the results can mask a single error without any service interruption (when the results differ, the majority is assumed to be right). This type of architecture, which we have illustrated earlier in Figure 6.7b, is called Triple-Modular-Redundancy, or simply TMR, and in this context the comparison unit is called a voter. Voting is usually performed at the bit level, which also means that the replicated components operate in lock-step. The TMR architecture can be trivially extended to tolerate more faults by increasing the number of replicas (N-Modular Redundancy, or simply NMR).

In a distributed systems context, hardware fault tolerance today should rather be seen as a means to construct *fail-controlled* components, in other words, components that are prevented from producing certain classes of failures, and then to use these improved components to achieve more efficient fault-tolerant systems.

8.2.2 Software-Based Hardware Fault Tolerance

Software-based fault tolerance aims at tolerating hardware faults using software techniques. Recall that it is the basis of modular FT, which underpins the main paradigms of distributed fault tolerance. In consequence, it is not surprising that all paradigms discussed in the previous chapter are pertinent to this framework. The fault-tolerant computing models that we are going to address in this chapter do an intensive use of software-based fault tolerance. The main players are software modules, whose number and location in several sites of the system depends not from construction constraints, but from the dependability goals to be achieved.

As we studied in Chapter 7 (*see* Sections 7.6 and 7.7), a great deal of the redundancy management policies for software-based fault tolerance inherit basic concepts of hardware fault tolerance, duly generalized, expanded and adapted. Detecting that a component failed and stopping its operation can also be done remotely in distributed systems (*see* Section 7.1). One possible way to detect a failure is to test the component periodically: this may not prevent the failed component from producing erroneous results, but it limits the duration of the abnormal behavior.

Another solution is to force all component outputs through a filter able to assess the correctness of the produced results: if an incorrect output is detected the component is halted. Spare components and replicas can be adequately placed in the system to recover from errors. Secondary spares may be called into operation when the primary fails. Replicas may operate in parallel, in what is called active replication. Besides providing glitch-free operation in terms of availability with regard to crash failures, if replicas diverge this means that an error occurred. A dual configuration may work as a self-checking (albeit distributed) component, whereas a triple or more can provide continuity of service with regard to value failures. When components exhibit fail-silent behavior and

there is a need for sparing resources, computation may be based on a primary replica, whereas the backups will be in a warm standby state, lagging the state of the replica. This is passive replication, and the lag is bounded by having the passive replicas be updated from time to time with the state of the primary.

From what was said in the last section, it is evident that software-based and hardware-based fault tolerance are not incompatible design frameworks. As studied in the previous chapter, distributed algorithms that tolerate arbitrary faults are expensive in both resources and time. For efficiency reasons, the use of hardware components with enforced controlled failure modes is often advisable, as a means for providing an infrastructure where protocols resilient to more benign failures can be used.

8.2.3 Software Fault Tolerance

Software fault tolerance aims at tolerating software faults. It is worth to distinguish three types of software faults:

- Faults due to particular sequences of events that are difficult to reproduce (the Heisenbugs). This type of faults can be tolerated by executing the same code more than once in the same or in different machines.
- Those design faults that may lead all replicas to fail in exactly the same way. These faults can only be tolerated by using design diversity. Naturally, the redundancy can be applied in the time or space dimensions, i.e., different versions of the same program can be executed sequentially on the same machine or in parallel on different machines.
- Faults that are specific to some peculiar hardware or configuration parameter (e.g., O.S.). This type of faults can be tolerated by executing the same program on different machines/environments. This case is not very common since it only has advantages if the software just does environment-specific errors.

Extensive bibliography has been published on the subject of software fault tolerance since the pioneering works described in (Randell, 1975; Chen and Avizienis, 1978). See for example in (Kim and You, 1990; Issarny, 1993; Xu et al., 1995), or the survey in (Lyu, 1995).

8.2.4 Fault-Tolerant Communication

Communication is so specific of distributed systems that it prefigures a framework of its own. Several techniques assist the design of fault-tolerant communication networks, as we saw in Section 7.5. Their choice depends on the answer to the following question: *What are the classes of failures of communication network components?* This is akin to establishing the failure mode assumptions, and leads to the selection of the type of redundancy: space, time, or value.

Assertive errors can be detected using value redundancy, in the form of message consistency checks, such as CRCs or signatures. Corrupted messages

can then be dropped, transforming an assertive fault into an omission fault. If enough redundancy is provided, the errors can be not only detected but also corrected: the combination with time redundancy, by repetition, recovers or masks most errors. Some semantic errors require space redundancy in order to be masked, namely because they can be made before the insertion of the consistency check.

Omissive errors are extremely common and they form the bulk of the body of research in FT communication. Timing errors are mainly addressed through three schools of thought: the one that seeks at preventing their visibility at system level, normally related with hard real-time systems design; the one that attempts at recovering from or masking them, sometimes with application help, akin to mission-critical systems design; and the one that gets rid of them, by assuming that the system is time-free, related with asynchronous systems design (*see Real-Time Communication* in Chapter 12 for the former two). Omission errors are classically addressed by time redundancy. Space redundancy by full replication is only used when either the glitch associated to recovery or the bandwidth overhead generated are not desired.

Finally, some network components can crash permanently. This may occur at the interface between the node and the medium (for instance, a malfunctioning Ethernet card) or at the medium itself (a broken cable, a broken repeater, etc). The effect of a component crash on the overall network heavily depends on the topology of the network, technology used, medium access protocol, location of the error, etc. It may disconnect a node, a partition, or the complete network. As for omissions, the chosen policy depends on the desired quality of service: full redundancy by duplication or n-plication of all network components; or medium-only redundancy, either by reconfiguring or replicated medium components.

8.3 FAULT TOLERANCE STRATEGIES

There are several factors affecting the choice of strategy to design a fault-tolerant system, such as: classes of failures (i.e., aggressiveness of environment); cost of failure (i.e., limits to the assumed risk); performance/price ratio; available technology (i.e., limits on the FT constructs available). We line up below the strategies we feel as most important. Once a strategy defined, design should progress along the guidelines suggested by the several fault-tolerant design frameworks just presented. Strategic issues are: redundancy policies between fault prevention and fault tolerance; types of faults tolerated (i.e., hardware or software faults); level of service provided (i.e., glitch-free or recoverable operation). We are going to discuss the main strategic issues concerned with fault tolerance in distributed systems: fault tolerance vs. fault avoidance; tolerance of design faults; perfect non-stop operation; reconfigurable operation; recoverable (fail-fast) operation; fail safe vs. fail operational.

8.3.1 Fault Tolerance versus Fault Avoidance

Fault avoidance, in its facets of fault prevention and fault removal, is an important part of resilient system design, aiming at amplifying component reliability. Given the complementary nature of fault tolerance (FT) and fault avoidance (FA), how much emphasis should be put on each in relative terms? Is it better to use highly reliable (but also highly expensive) components? Or should we use several copies of less reliable components and pay the overhead of complex fault-tolerant protocols?

The good mix of FT and FA has to do with two issues: the number, and the severity of faults. In terms of the number, the answer lies basically on the expected length of the mission versus the expected reliability of the components used, for a given amount of redundancy. For a given reliability, the number of spares should not go beyond the point of diminishing returns, and this dictates the balance. The issue is also relevant when space, power-consumption or performance reasons discourage the use of large amounts of replication.

However, the touchstone of the FT/FA balance lies in the nature of faults, vis-a-vis the efficiency of the associated replica consistency protocols. We have seen in Chapter 7 how costly protocols tolerating arbitrary faults can be, with regard to protocols tolerating omissive or even certain kinds of assertive faults. Based on the quality of service to be provided by the system, and on a preliminary elaboration of failure mode assumptions about the environment and infrastructure, the architect should lay down her strategy, starting with the definition of the architecture and the choice of components. Then, the evaluation of the coverage of assumptions may dictate successive iterations through this process until a balance between coverage and effectiveness is obtained. A cost-effective approach has been to rely on Commercial Off-The-Shelf (COTS) components. Very often, COTS do not offer the desired level of confidence in their behavior. In such a case, fault tolerance may recursively be applied at sub-system level, enhancing those COTS components in order that a controlled failure mode is obtained of the final ensemble, with high enough coverage. This recursive use of fault tolerance to build the component effectively classifies as fault prevention at the interface of the component, when seen at the outer system level.

8.3.2 Tolerating Design Faults

Most of the design faults in mature systems, software or hardware, are transient faults. As such, many fault-tolerant systems that use replication or time redundancy (repetition or re-execution) become tolerant of design faults, since subtle design faults tend to be activated only in some scenarios which are hard to reproduce. This is a valid strategy for all but highly-critical systems. In this case, one has to rely on design diversity which is naturally extremely expensive. In the limit, the architect will point to the development of N code versions, each produced by a different team, which will execute in as many distinct hardware platforms, in active replication. Note that it is usually harder to build a sys-

tem using diverse components, because of problems such as conversion between different machine representations of equivalent values.

8.3.3 Perfect Non-stop Operation?

Is there such thing as perfect non-stop operation? This notion is in the eye of the beholder, that is, the service user, and so a careful answer will be "in principle, yes".

The ideal fault-tolerant system masks all errors in such a way that the user is never aware of their effect. Note that the masking approach is expensive, and is eligible when the cost of fault tolerance is negligible compared to the cost of service interruption. It is thus the strategy for life- or money-critical systems, such as flight control or air traffic control, on the safety and availability facets, respectively. Fault tolerance is something like virtual memory: it requires better hardware and slows down the execution a bit, but it is hard to live without it.

Network omissions and some software Heisenbugs can be masked using time redundancy. The most straightforward approach to mask crash failures is to use active replication, for example under the diffusion or event-based model, which we address in Section 8.5. It should be noted that in some cases even backward recovery is adequate, if the recovery glitch is short enough to go unnoticed. Sometimes when we use the Web, our network connection is so slow that we wouldn't notice the server crashing and recovering.

In partitionable or pure asynchronous systems, perfect non-stop operation is not achievable unless in some very specific cases where the application semantics imposes little consistency requirements (recall the FLP impossibility result!). Thus, if non-stop operation is required it is necessary to invest on redundant network architectures (that minimize the probability of partition) and time-controlled environments (to ensure the required synchrony).

8.3.4 Reconfigurable Operation

Active replication is expensive and as such many services resort to cheaper redundancy management schemes, based on error recovery instead of error masking. This alternative approach can be characterized by the existence of a visible glitch. The underlying strategy, which we may call reconfigurable operation, is normally addressed at availability-oriented services, such as transactional databases, web servers, etc., normally accessed through remote operation primitives, such as studied in Section 8.4.

Several techniques may be used, and the typical replication management schemes fall in the semi-active and passive classes (*see* Section 7.6). These techniques imply an omissive failure mode, and are normally used to handle crash failures of components. The failure of a component triggers a reconfiguration procedure that automatically replaces the failed component by a correct component. During reconfiguration the service may be temporarily unavailable or suffer some performance degradation, whose duration depends on the policy

used. For example, the take-over of a passive backup when the primary fails produces a glitch composed of the detection and the reconfiguration latencies. Reconfigurable operation is very complex in asynchronous systems, because it is impossible to have accurate failure detection and one risks to have multiple components playing the role of "primary", if the adequate algorithms are not used (*see* Section 7.7).

8.3.5 Recoverable Operation

Consider that a component sometimes crashes but recovers after some time. A fault-tolerant design can be obtained under these circumstances, if a set of preconditions hold. Firstly, the duration of recovery must be known and bounded, and short enough for the application's needs. Secondly, the crash must not give rise to incorrect computations. This may achieved through several techniques, amongst which we name checkpointing into stable storage and logging executions past the last checkpoint, as studied in Section 7.9. In distributed computations, NVRAM may provide the support for logging last moment state and achieving consistency of recoverable remote operations (*see* Section 8.4). Recoverable exactly-once operation can be achieved with atomic transactions, which we address in Section 8.6.

This strategy concerns applications where at the cost of a noticeable temporary service outage, the least amount of redundancy is used. Architectures have evolved in order to grant a reduced recovery latency, in what are also called *fail-fast* systems. The strategy also serves long-running applications, such as scientific computations, where availability or reliability concerns are not as demanding as in interactive applications. However, the duration of the computation is such that the probability of a failure jeopardizing the whole computation requires attention.

8.3.6 Fail-Safe or Fail-Operational

In certain situations, it is necessary to provide for an emergency action to be performed in case the redundancy within the system is no longer enough (spare exhaustion) or is not adequate for the faults occurring (assumption coverage). In this case, rather than letting the system evolve to a potentially incorrect situation, which in critical systems may entail a catastrophic failure, it is preferable to shut the system down at once, what is called a *fail-safe* behavior.

This strategy, important to safety-critical systems, may complement other strategies concerning the regular behavior of the system. As a variation, instead of halting at once certain applications may require that the system performs an orderly shut down routine before halting. In this case, it is required that no matter the situation that causes the need for shut down, the system have the capability of performing the orderly shut down routine, as part of its nature of being a fail-safe system.

In special cases, the fail-safe nature of a system is not obtained by stopping it at all. Indeed, in some cases, that would be a catastrophic outcome, as in the

case of a flying airplane with engine problems. In these situations, there must be contingency plans for the system to be put in a mode where it continues operating despite having failed the provision of its intended service. This should be achieved at least until a safe stop is possible. We call these systems *fail-operational*.

8.4 FAULT-TOLERANT REMOTE OPERATIONS

Remote Procedure Calls (RPCs) or Remote Method Invocations (RMIs) are a well established method of providing a reasonable degree of distribution transparency to the application programmer. Basically, RPCs are based on a client-server architecture. Communication is supported by a simple message exchange protocol that operates in a request-reply fashion. In this section we discuss the fault-tolerant issues related with the implementation of fault-tolerant RPC services.

8.4.1 Reliability of Remote Operations

An RPC architecture has mainly three macroscopic components, namely: the client, the network and the server. Of course, each of these components can fail. To illustrate the main issues regarding fault tolerance in RPC systems we first assume a relatively benign (and common) failure mode: clients and servers can fail by crashing and the network can crash and/or lose messages.

What is desired of the server is that it executes requests once, and only once. This desirable semantics is called *exactly-once*, but we will see that it is unreachable in the case of remote operations with RPC in the presence of failures, and we will have to accept weaker semantics. The several failure scenarios and possible remedies are illustrated in Figure 8.1. We will refer to them in the discussion below.

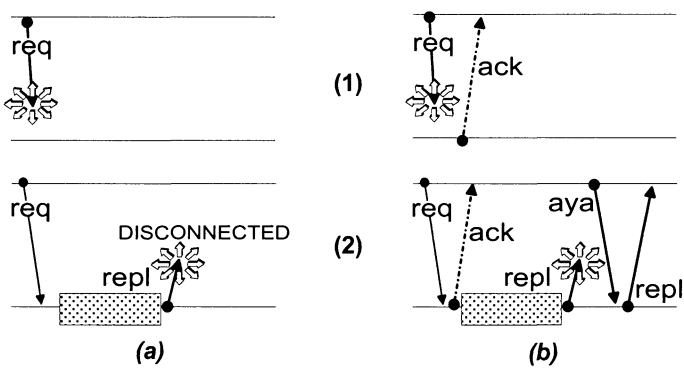

Figure 8.1. Remote Operations: (a) Network Failures; (b) Remedies

Network omissions may affect either the request sent from the client to the server or the reply sent back by the server to the client (Figures 8.1.1a

and 8.1.2a). These two faults are indistinguishable from the client point of view; in any case, what it can observe is the absence of the desired reply. Well, this protocol was too simple, and thus we consider enhancing it, in the way shown in Figures 8.1.1b and 8.1.2b: the request is acknowledged, and this remedies the first problem, since the client can timeout on the acknowledgment and repeat the request; for the second problem, the client periodically enquires the server with an "are you alive?" heartbeat message, which the server replies with the RPC result if it had already finished.

So far so good, but the server may fail. See Figures 8.1.1c and 8.1.2c, depicting two distinct situations: the server fails *before* executing the service in the former, and *after* executing it in the latter. Although the enhanced protocol detects server failure through the *aya* heartbeat, the two situations are indistinguishable from the client viewpoint. Worse, it may even happen that the request arrives, it is executed, the *ack* is lost and then the server fails: the client still thinks that the request did not make it to the server. In short, the client may identify similar syndromes for different failures. Is this a problem?

Intuitively, if the client does not have a result, it makes sense for it to re-issue the request until a reply is received or until some pre-defined number of retries has been exceeded. However, this uncertainty leaves the client in suspense: "Did my Pizza order succeed or should I buy a Burger?". The problem of having the client re-issue the request is that the server may receive several copies of it. In the case of the Pizza order, the client may not be willing to pay for four Pizzas when she did order just one (no pizza is *that* good), so discarding the duplicate requests to ensure only one execution seems like a wise decision.

However, enforcing this behavior introduces a non-negligible amount of overhead. Requests must be stamped with some unique identifier and the server must store the identification of previous requests. Additionally, the server must also store the replies it has produced in the past, in order to send them back to the clients when requested. Even if clients make their requests one at a time (which means that the server just has to keep the reply to the last request from each client) this may still be a significant overhead when a server has many clients.

An additional problem will definitely drive us away from exactly-once semantics: all that was said above works while the server does not fail. Consider a simple server, with no stable storage: it is amnesic after it recovers, and thus it cannot know which requests it has executed (recall that the client does not know either). There are only two alternatives: either execute request repetitions, or do nothing.

If by all means re-execution must be avoided, then a recovering server cannot execute request repetitions. This discipline combined with duplicate elimination happens to be the strongest semantics that can be achieved with RPC remote operations, called *at-most-once*: it does what its name implies, and thus, maybe nothing ends-up being executed, or worse, something is partially

executed. Some researchers further distinguish the first situation, calling it *zero-or-once* behavior.

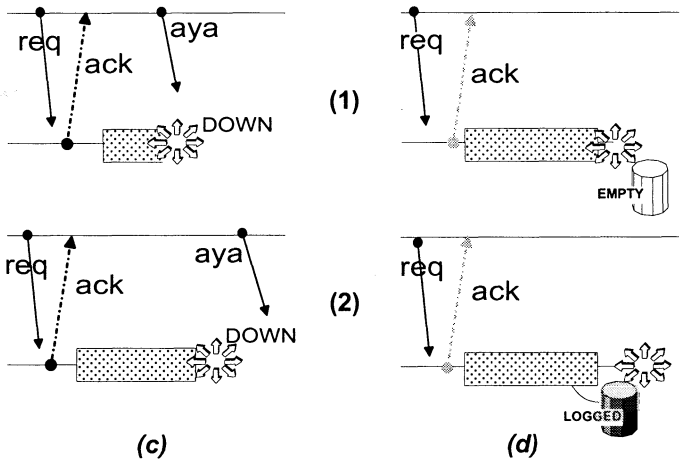

Figure 8.1 *(continued)*
. Remote Operations: (a) Server Failures; (b) Logging to Non-Volatile Store

The effectiveness of at-most-once semantics can be enhanced by making servers state-full. If they preserve their state in persistent memory, they can reconstruct it upon recovery before resuming service. This has severe performance impact, and thus a practical implementation consists in using NVRAM[1] rather than disk, just to keep request identifiers and their execution state. This reduces the window of ambiguity concerning request execution state.

Yes, we said reduce, not cancel. See Figures 8.1.1d and 8.1.2d, depicting two very distinct situations taking place after the server executes the request: it fails *before* logging it in the former, and *after* logging it but before informing the client, in the latter. We leave it to the reader to understand that there is no way out of this ambiguity, except by using another paradigm.

State-full servers and at-most-once behavior not only consume more resources but may also exhibit poor performance. This overhead should be avoided whenever there is no significant disadvantage in executing the same request more than once. For instance, consider a server that provides clients with the value of the room temperature. Executing the same request several times has no negative effect other than consuming resources on the server machine.

Requests that can be executed several times and produce equivalent results are said to be *idempotent*, and the corresponding semantics is called *at-least-once*. This is so convenient that is often worthwhile designing the application in such a way that all requests have this property. In this case, servers are

[1]Non-Volatile RAM.

much simpler and faster, since they are not required to keep any state about past requests. In consequence, they have no state to lose when they crash. For this reason, they are said to be *stateless*. Stateless servers can recover after a crash and continue providing service as if nothing happened.

Client failure is less critical from the point of view of data consistency, but still deserves some comments. If a client fails, this is only a problem if the server has given the client the exclusive right to use some resources, in the course of execution of a previous request. In a fault-tolerant system, to assign resources to a component that may fail and whose failure cannot be reliably detected is generally a bad idea. A more conservative approach is to *lease* (Gray and Cheriton, 1989) the resources for some fixed amount of time and require to client to re-acquire the resources periodically. Another downside of having clients whose crash goes unnoticed is that their last wishes may keep the servers busy computing responses that nobody will ever read. These computations are called *orphans*, and the problem may be solved with a facility called orphan detection (Panzieri and Shrivastava, 1988).

8.4.2 Building Reliable Client-Server Systems

So far we have discussed the issue of data consistency and operation correctness in Remote Procedure Call systems, in the presence of failures. We often want more than that: we also want availability. If modular fault tolerance is used, the issues concerned with building a reliable client-server system are incremental to the principles of building basic C-S systems discussed in *Client-Server with RPC*, Chapter 3.

It is time to apply the replication techniques that we have studied in previous chapters. A very intuitive way of offering high availability in an RPC system consists in constructing a fault-tolerant server using the replicated state-machine approach. The client broadcasts the reply to the group of servers using a primitive that enforces agreement and order. An atomic multicast protocol is particularly well suited for the job. All replicas process the request and send back a reply to client. According to the types of faults being tolerated, the client can pick the first reply or wait for a majority of similar results. The servers can be themselves clients of other servers. Since the server is replicated, different copies of the same request will be produced which need to be combined in a single correct request (again, using voting) before being processed by the servers.

The actively replicated state-machine approach is the most intuitive way of building reliable client-server systems using replicated RPCs. However, it is not the only one. All the other schemes discussed before, namely the semi-active and passive replication schemes can be used (and have been used) to build fault-tolerant C-S servers.

8.5 FAULT-TOLERANT EVENT SERVICES

Sometimes, it is more convenient to express an application in terms of event notifications than through request-reply interactions. As we have seen in Part I of this book, event-based systems, like their RPC based counterparts, have three main components: the event producers, or *publishers*, the communication media, or *channel*, and the event consumer or *subscriber*. Two variants of the architecture can be found: volatile channel systems (where the messages can be consumed as they cross the channel and discarded afterwards) and persistent channel systems (where the messages are kept by the system until consumed by their recipients). Fault-tolerant techniques can be applied to both systems.

8.5.1 Volatile Channel Architectures

In event based systems, when a notification is produced it is usually with the aim of being processed by some other component interested in the event. The processing associated with the event may be rather simple, such as storing a sensor reading in a log, or complex, such as shutting down the system in a graceful manner after an alarm has been fired. In such systems, fault tolerance means: (i) ensuring that the desired event is produced; (ii) that it is reliably delivered to the consumer; and (iii) ensuring that there is a consumer ready to process the event.

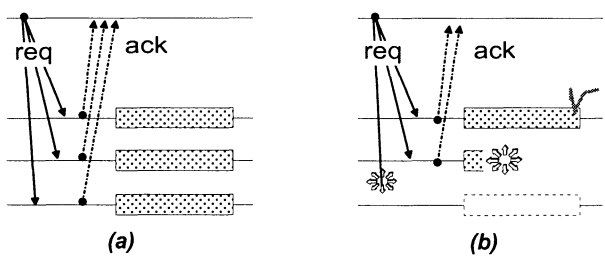

Figure 8.2. Replicated Channel: (a) No Failures; (b) Exactly-once Execution

In order to ensure that the event is produced, replication should be used. For instance, if a sensor reading must be available, one has to replicate the sensor. In order to ensure that the event is processed, one can also replicate the subscribers. As before, to keep the replicas consistent, the replicated-state machine approach can be used. Naturally, this requires the use of group communication in order to ensure the all events are delivered in total order to all consumers. In terms of failure semantics, note that it is quite straightforward to achieve *exactly-once* behavior with the state machine model fed by reliable group communication. Observe Figure 8.2: in the no-failure situation we have all three replicas execute the request; three replicas survive 2 failures and deliver the service exactly-once, as shown.

8.5.2 *Persistent Channel Architectures*

Persistent channel publisher-subscriber architectures are similar in many respects to volatile channel architectures. However, since the channel stores the events for later retrieval, the architecture does not require the publishers and the subscribers to be active at the same time in order to exchange notifications. On the other hand, the channel is much more than just a communication media, it is a storage media that needs to preserve the messages in a reliable way.

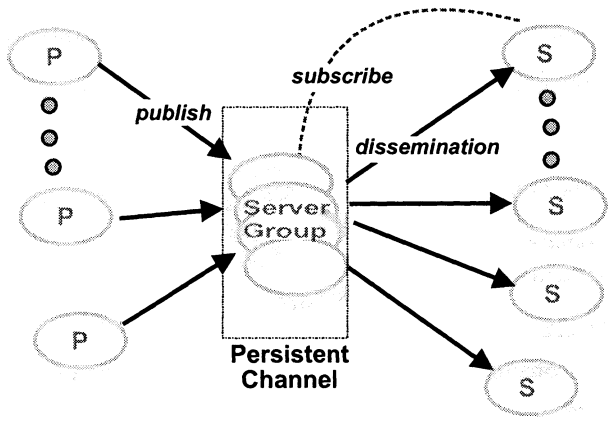

Figure 8.3. Fault-Tolerant Publisher-Subscriber

One way to describe the persistent channel abstraction is to consider the channel as a fault-tolerant server that can be accessed by publishers and subscribers. In some sense, the "channel server" implements some form of stable storage that is fault-tolerant, namely to crashes. The channel can be implemented as a replicated set of servers, using techniques described earlier. This can be done either through non-volatile storage, or through replicated volatile storage (*see Recovery* in this Chapter). Figure 8.3 exemplifies the principle: the message channel is at least capable of recoverable operation without losing data. Since it is replicated, it can also achieve non-stop operation given a sufficient degree of replication. Publishers and subscribers can access the server using one or several IPC primitives (message passing, RPC, group communication).

8.6 TRANSACTIONS

This section describes a number of relevant issues regarding the implementation of transactional systems. The concept of concurrent transaction has been introduced in Part I of the book (*see Concurrency* and *Atomicity* in Chapter 2). We begin by summarizing the most important notions, before we delve into the implementation decisions concerning transactional systems.

Most applications have sequences of operations that are only useful if executed as a whole. Consider for example a bank transfer that withdraws money from one account and deposits the same amount in another account. If this

sequence is interrupted by some reason (for instance, a failure) someone will lose money (either the money is withdrawn and never deposited or vice-versa). At the application level, recovery can be greatly simplified if some underlying mechanism guarantees the *atomicity of failure* of a sequence of operations. Atomic transactions are such *indivisible* sequences of operations: either all operations in the sequence are successfully executed, or none is executed. We say that a transaction *commits* its results when it successfully terminates; otherwise we say that the transaction *aborts*. Transactional systems must have some way to delay the effects of the transaction until the transaction commits (i.e., they keep the intermediate results in a log, also called the *redo log*). Alternatively, operations may be allowed to take effect immediately, as long as the system has a way to *undo* these effects, in the case the transaction aborts (what is called an *undo log*). Note that if another transaction reads a value produced by a transaction that is later aborted, what is called a *dirty read*, the second transaction must abort too (it has "seen" results that, for all practical effects, "did not happen"!).

Clearly, transactions are easier to implement in systems where the semantics of the operations are simple and well understood. For instance, if all operations are *reads* and *writes* in data items, it is almost straightforward to construct a redo log (containing the new data values) and/or an undo log (containing the old data values). In fact, databases are the ideal field of application of transactions. In systems that perform external actions, also called real actions, enforcing transactional semantics is much more complex and sometimes impossible.

The atomicity and indivisibility of the transactions also have another facet: concurrency in the access to the same data by different transactions should be "hidden" from the transaction programmer. This preserves the intuitive notion that the transaction is indivisible and executes as a single atomic instruction. Also, it is well known that concurrent programming is by no means a trivial task, which is handled automatically by the transactional system. Getting *concurrency control* out of the way of the application programmer is by itself an effective way of improving code reliability and guaranteeing data *consistency*. Finally, the effects of the transactions should not be lost, or in other words, must be *durable*. Of course, how much durable depends on the applications needs. There is a range of progressively more fault-tolerant approaches: the results are stored on disk; on redundant disk arrays; or in several different disks on different locations.

Collectively, the "**A**tomicity, **C**onsistency, **I**solation, and **D**urability" properties are also known as the ACID properties of transactions.

8.6.1 Transaction System Architecture

Transaction systems are generally composed of three components, namely: the *Transaction Manager*, the *Scheduler* and the *Data Manager*. A generic block diagram of a transaction system architecture is depicted in Figure 8.4, showing the interaction between the several modules.

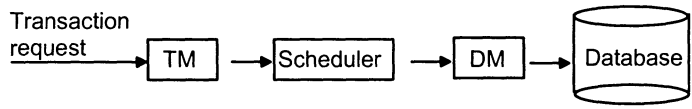

Figure 8.4. Transaction System Architecture

```
procedure Deposit (inout Account a, in Amount v)
begin
    Amount x;

    x = a.read ();
    x = x + v;
    a.write (x);
end Add;
```

Figure 8.5. Deposit transaction

The Transaction Manager ensures the interface with the application code, adding transaction identifiers to the application requests and forwarding them to the appropriate node. The Scheduler is responsible for concurrency control; it executes, rejects or delays the operations to enforce concurrency control. The Data Manager is itself composed of two sub-components: the *Recovery Manager* and the *Cache Manager*. The Recovery Manager is responsible for the resilience of the data accessed by the transaction; it manages the physical media, the recovery logs and applies the recovery procedures. The Cache Manager is responsible for moving data between the stable storage and the faster volatile memory.

8.6.2 Concurrent Transactions

Those that have experience with concurrent programming are quite familiar with the problems that can occur when several threads of control concurrently access a shared data structure. Transactions are no exception and also require the use of some form of *concurrency control*.

Let us give a couple of simple examples of problems that may occur if concurrency control is not enforced. Consider the transaction of Figure 8.5 that adds a certain amount to a given account. Consider now that the target account "my_account" has an initial value of zero and that two concurrent transactions T1 and T2 execute `deposit(my_account,10)` and `deposit(my_account,20)` respectively. If transactions are executed one at a time, the final result will be 30 in `my_account` has you would probably expect. However, we want transactions to execute concurrently, but unfortunately, the interleaving of instructions illustrated by Figure 8.6 results in a final value of 20, as if only one of the transactions was executed. This particular problem is known as the "lost update" problem.

```
T1 x1 = my_account.read (); // reads 0
T2 x2 = my_account.read (); // reads 0
T1 x1 = x1 + 10;      // x1 = 10
T2 x2 = x1 + 20;      // x2 = 20
T1 my_account.write (x1); // my_account = 10
T1 my_account.write (x2); // my_account = 20
```

Figure 8.6. Lost Update Problem

Other similar problems can occur. Consider the same example. A transaction withdraws a given amount from one account and deposits the same amount in another account. If another transaction tries to compute the sum of both accounts, and accesses the first account *after* the withdrawal and the second account *before* the deposit has been made it will find that some money is missing. This problem is known as the *inconsistent retrieval* problem.

Enough for the problems! Something needs to be done to prevent these scenarios from occurring, and this something is called concurrency control. The goal should be to allow as much concurrency as possible. Of course, we could prevent the concurrent execution of transactions by using a global lock on all data (for instance, on the complete database). The resulting performance of the system would be worse than deplorable. Typically, many transactions access unrelated data items and can be executed concurrently without any type of restrictions. What is needed is a mechanism that allows transactions to execute concurrently as long as they produce the same results has if they were executed in (some) serial order. This correctness criterion is known as *serializability*.

A huge body of theory exists on concurrency control for databases and, in particular, on enforcing serializability. Here we will just address one of the simplest (and popular) mechanisms: *locking*. Locking uses two types of locks: *shared* locks and *exclusive* locks. Whenever a transaction accesses a data item it locks that item: if the transaction accesses the item for reading it acquires a shared lock (other read locks can be granted); if the transaction accesses the item for writing it acquires an exclusive lock (no one else can grab this item). Only shared locks are compatible. If at least one of the locks is an exclusive lock the locks are incompatible, as illustrated in Table 8.1. If the item is already locked by another transaction, the transaction must check first if the locks are compatible; if they are not compatible, the transaction must wait until the previous lock is released in order to acquire its lock.

Locks cannot be released as soon as the operation completes; this would be almost as good as not having locks at all (just try to add an acquire immediately before and a release immediately after every item access on Figure 8.6). It has been shown that if a transaction does not release any lock before acquiring all the locks it needs, serializability can be enforced. This is known as *two-phase locking* (2PL), because the locks are first acquired (this phase is also called the *growing* phase) and later released (this phase is called the *shrinking* phase).

Table 8.1. Lock Compatibility

Lock Types	Shared	Exclusive
shared	compatible	incompatible
exclusive	incompatible	incompatible

We have already noted that if a result from a transaction that later aborts is seen by another transaction, that second transaction also needs to abort. This rule can easily be applied in a recursive manner. If a third transaction sees an intermediate result from the second transaction, the third transaction will have to abort too. This phenomenon is known as *cascading aborts* or sometimes, by the more visual name of *domino effect.* Cascading aborts are not a positive feature, since they require work to be undone and this means poor resource usage. A way to prevent cascading aborts is to prevent intermediate results from being visible before the transaction terminates. When locking is used, this can be achieved simply by holding all the locks until the transaction commits or aborts; this restriction to the two-phase locking rule is so popular that deserves a name of its own: *strict two-phase locking.*

Locking is simple but not exempt from disadvantages. One of the problems with locking is that it may cause *deadlocks.* Consider for example a transaction T1 that transfers an amount from account A to account B and another transaction T2 that transfer an amount from account B to account A. If T1 locks the account A and T2 locks the account B, none of the transactions will be able to make progress. Many different strategies can be used to prevent, avoid and detect deadlocks (*see Coordination* in Chapter 2). The most simple strategy is to define a maximum waiting time for a lock to be released. If this timeout expires a deadlock is assumed and the transaction is aborted.

8.6.3 Distributed Transactions

Distributed transactions are simply transactions that access items on different nodes of a distributed system. Distributed transactions can be built using the mechanisms developed for centralized transactions. A generic block diagram of a distributed transaction system architecture is depicted in Figure 8.7, showing the interaction between all TM modules, which competitively launch transactions, and local Scheduler modules, which must ensure that the distributed transaction is scheduled correctly at every site.

In fact, a distributed transaction can be described as a collection of several sub-transactions, each individual sub-transaction being initiated on each node visited by the distributed transaction. The local transactional mechanisms on each site will guarantee that each individual sub-transaction either executes completely or aborts. Since locking is a *local* concurrency control mechanism,

i.e., locks are associated with individual data items, it can be used to enforce serializability of distributed transactions.

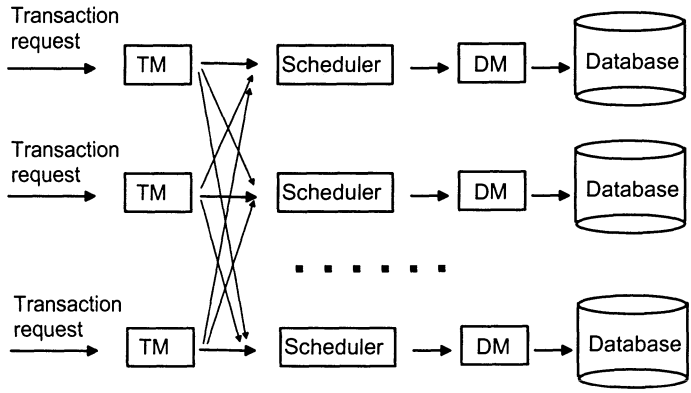

Figure 8.7. Distributed Transaction System Architecture

Distributed transactions have nevertheless a unique feature that is not present in centralized transactions. All the participating nodes need to agree on the outcome of the transaction. Thus, nodes participating in a distributed transaction need to execute an *atomic commitment* protocol. These protocols have been discussed earlier (*see Recovery* in Chapter 7).

8.6.4 Transactions and Replicated Data

Replication can improve the availability of data and can also improve the system performance, by putting data closer (in terms of communication delays) to its user. A transaction system that supports access to replicated data should support replication transparency. This means that the several replicas of the data should look just like a single copy to their users. This is called *one-copy equivalence* and the corresponding correctness criteria is called *one-copy serializability*.

One intuitive way of making several replicas look like a single copy is to keep all the replicas with exactly the same value. In other words, when one replica is updated, all replicas are updated. Since all the replicas always have the same value, when a read is performed you just need to read a single replica (this is also called the *write-all, read-one* approach). At this point, you probably recall the state machine of Section 7.6.2 as a general technique to maintain consistent replicas. We will now ask you to put aside the state-machine approach for some moments, and to think of how to maintain replica using exclusively transactional mechanisms. In terms of transactions, updating all the replicas means including the writes to different replicas within the same transaction. Thanks to the all-or-nothing property of transactions, this will ensure that either all replicas are updated (if the transaction commits) or none is (the transaction is aborted).

In a system where replicas never fail, the write-all approach works fine, at least from a consistency point of view. Of course, the performance of this approach is highly dependent on the number of replicas and on the communication efficiency. However, if a replica fails the system comes to a halt, since not all replicas can be updated. Fortunately, we have already presented solutions for this problem (*see Replication Management in Partitionable Networks* in Chapter 7). Let us just recall some of the paradigms that can be applied to this problem: weighted voting, coteries, structural representations and dynamic voting. These solutions will preserve one-copy serializability even in the presence of partitions. Note that write quorums should always intersect among themselves and with read-quorums. This means that conflicting operations always intersect in at least one replica. This replica can detect the conflict using a local concurrency-control mechanism (such as locking).

Suppose now that you are in an environment where network partitions (real or virtual) are very unlikely. In such cases, when a replica does not respond this means that the replica is crashed. Voting can be too penalizing, since in order to increase the availability of writes it reduces the performance of reads (which can no longer be done from a single replica). Why not just try to keep all available copies updated and forget about the crashed replicas? This approach is known as *write-all-available* copies approach. Of course, one has to be careful with recovering replicas because these replicas are out-of-date. Thus, recovered replicas cannot be read until they are brought up-to-date.

Does the available replicas approach really work? The answer is yes if special care is taken in dealing with crashes. Consider the following scenario. There are two data items A and B with different initial values, say $A = 50$ and $B = 100$. Consider that a transaction T_1 sets $B := A$ and a transaction T_2 sets $A := B$. The final outcome depends on the order by which T_1 and T_2 are serialized. If T_1 is serialized before T_2 both A and B will be set to 50. If T_2 is serialized before T_1 both A and B will be set to 100. In any case, the final value of A should be the same of B. Suppose now that there are two copies of each account, denoted A_1, A_2, B_1 and B_2 and that A_1 and B_2 fail during the concurrent execution of T_1 and T_2. Now, let us look at what happens if transactions execute as follows:

1. *Transaction* T_1 *reads* A_1
2. *Transaction* T_2 *reads* B_2
3. A_1 *and* B_2 *crash*
4. *Transaction* T_1 *updates just* B_1 *because* B_2 *has crashed*
5. *Transaction* T_2 *updates just* A_2 *because* A_1 *has crashed*

The final amount of the remaining replicas will be different (in fact, this execution swaps the values of A and B)! What went wrong? The problem is that the transactions were operating with different sets of available replicas: T_1 did read A_1 but T_2 only wrote A_2. In order for available replicas to work properly, crashes need to be serialized with data accesses. This should not come as a surprise since we have already discussed the concept on virtual synchrony (*see Consistency* in Chapter 7) and the relevance of ordering failure information with regard to application messages.

This brings us back to the state-machine approach. Are transactions and state machines incompatible methods of managing replicated data? Not really. Actually, they can be seen as complementary methods. Transactions guarantee the atomicity of sequences of operations; this is an aspect that is not addressed by the state-machine approach. Replicated state machines put emphasis on replica consistency; although transactional mechanisms can also address this issue they can be improved with the lessons learned from building state machines.

Imagine that two concurrent transactions and T_1 and T_2 try to lock a replicated data item. Unless special care is taken, the lock requests may arrive in different orders to the two replicas. Thus, one replica can be locked by T_1 and the other by T_2. This would result in deadlock. The replicated state-machine solution to this problem would be to use an atomic multicast primitive, ensuring that both replicas receive the same lock request in the same order, preventing the deadlock from occurring. Recent work has shown that the use of ordered group communication primitives can improve the performance of replicated database management systems (Pedone et al., 1998; Kemme et al., 1999).

8.6.5 Building Transactional Systems

Systems where the users submit transactions and wait for the outcome of the transaction are called *On-Line Transaction Processing* systems (OLTP for short). Transactional systems can be used to build OLTP applications but also to build applications that operate off line or in batch mode.

An example of a batch transaction processing system is the system that processes check payments. Checks issued by bank A and deposited in bank B during the day are sent in batch to the issuer at the end of the day (usually, through some third party clearinghouse to avoid the need for each bank to contact every other bank directly). Bank A will then debit the appropriate accounts and/or register exceptions such as lack of funds. Again, the results of the transactions are sent back in batch to bank A (usually, one or more days after) which in turn will credit the accounts where the checks were deposited.

On the other hand, some operations performed with a debit card on an automatic teller machine require OLTP support. Operations such as reading the account balance require the execution of a distribution transaction that, ultimately, needs to contact the computing system of the issuing bank. Building OLTP systems raises many challenges which are not limited to fault tolerance. These systems are usually very large in terms of users and volume of data and must be able to withstand a very large number of transactions per second. This requires a clever design in terms of operating system constructs (how the transactional system is decomposed into processes and threads), memory management (the hierarchy from disk to CPU cache) and communications (to ensure that enough bandwidth is available). Obviously, these systems also pose enormous challenges in terms of security but these issues will be dealt with in Part IV of this book.

Disaster recovery is often considered as a must-do item in a large enterprise information system. The idea is to have contingency plans, should a major disaster occur that would severely degrade the operational capability of the information system, namely its public presence, mainly through the OLTP operations. However, there is a misunderstanding: what disasters are we talking about— informatics[2] disasters (computer blowing-up, disks failing, backups lost)? Or *real* disasters (major floods, massive power outages, large-scale fires, earthquakes)?

Many companies devote (expensive) contingency plans to both kinds. However, it seems that the first ones can and should be avoided by technical, runtime measures. Fault tolerance is failure avoidance. Tolerance of faults that can yield catastrophic effects is *disaster avoidance.* Distributed fault tolerance is just about the paradigm to handle serious faults that can have a geographical dependency. In reality, it only makes sense to make contingency plans for the events that cannot be avoided, such as environmental disasters with global proportions. Very often, it is better to prevent than to remedy.

8.7 SUMMARY AND FURTHER READING

This chapter discussed how to apply the paradigms presented in Chapter 7 to build fault-tolerant systems. It departs from a systematization of failure classes and how these impact the approaches to fault tolerance, both in terms of techniques chosen and in terms of the service provided. Detailed evaluation of failure assumption coverage of hardware and software components can be found in (Iyer and Joshi, 1985; Madeira and Silva, 1994; Arlat et al., 1990; Carreira et al., 1998; Maxion and Olszewski, 1998). See (Kopetz et al., 1989a; Powell, 1991) for systems built to the strong and weak fail-silence assumptions.

Then, we have focused on three of the major constructs to build reliable distributed applications: remote operations, event based systems and transaction systems. We discussed the application of the paradigms discussed eralier to build these types of systems. Additional readings in reliable remote invocation systems can be found in (Cooper, 1985; Liskov et al., 1987; Panzieri and Shrivastava, 1988; Rodrigues et al., 1994). Fault-tolerant event-channels have been presented in (Oki et al., 1993; Felber et al., 1997). Naturally, there is a huge bibliography on transactions, from which two books emerge as fundamental references: (Bernstein et al., 1987; Gray and Reuter, 1993).

[2] "Informatics" is a word of european origin getting increased acceptance in the community of computer users and developers. It is used to denote in general terms all that is related with use of computers and networks in information processing, access and manipulation.

9 DEPENDABLE SYSTEMS AND PLATFORMS

This chapter gives some examples of dependable systems and platforms. Whenever possible, paradigms and models previously studied are pointed to the reader. The overview of each system is concise and the selection is subjective but tries to illustrate each class of approaches using concrete case studies that we find representative of that class. Namely, we discuss: distributed fault-tolerant systems, transactional systems, cluster architectures, and how to make legacy systems dependable. In each section, we will mention several examples in a summarized form, and then will describe one or two the most relevant in detail. Table 9.1 at the end of the chapter gives a few URL pointers to where information about most of these systems can be found.

9.1 DISTRIBUTED FAULT-TOLERANT SYSTEMS

9.1.1 Tandem and Stratus

Tandem (now belonging to Compaq) and Stratus are two companies that have specialized in building fault-tolerant systems. Solutions offered by these companies include a combination of hardware and software fault tolerance that offer reliability and continuous availability. The products that achieve higher degree of reliability use proprietary hardware and software (including specialized operating systems). We will focus on some of their most innovative architectures.

Stratus was the first company to introduce a processor architecture that used duplication at the level of CPU, I/O and communication hardware. In this

P. Verissimo et al. *Distributed Systems for System Architects*

architecture, a CPU server with crash failure semantics is obtained using a pair of microprocessors that execute in lock-step and whose results are compared. To offer availability, CPU servers are replicated using the state-machine approach; in this way a pair of CPU servers may mask the crash of single server. At the memory level, a two-bit detection/ one-bit correction coding is used to build a memory unit with crash failure semantics.

Tandem developed a set of products for high-reliability transaction processing called the Guardian 90 system. The set includes a single operating system that offers transactions in the kernel and supports the parallel execution of *process pairs* on duplicated hardware. In the process pair approach, each active process has a backup process associated to it. All actions of the active process are checkpointed to the backup such that, in the case of failure of the active, the backup can continue the operation. In terms of hardware, the Guardian executes on top of a fully duplicated architecture. The hardware is configured to ensure the existence of two disjoint paths from terminals to servers. Today, Tandem offers a wide range of solutions combining the process-pair approach and hardware redundancy (lock-stepped microprocessors).

The pressure for competitive solutions even if offering lower levels of reliability lead many companies to explore an alternative track, applying their expertise to enhance commercial off-the-shelf components (COTS), using modular (software-based) fault tolerance techniques. Target markets are high-reliability, high-availability versions of servers based on commercial operating system "standards" such as Unix and WindowsNT/2K.

9.1.2 The Isis family

Isis (Birman and van Renesse, 1994) started as a research project at Cornell University that focused on the use of process groups to build reliable distributed applications. The core of the Isis system was an innovative set of group membership and reliable communication primitives, enforcing different ordering properties including causal and totally ordered multicast. Isis was also the first system to introduce the important concept of virtual synchrony. The main services offered by ISIS were: ISIS Toolkit (basic protocols); Reliable Distributed Objects (object interface to groups); Distributed News (a publisher-subscriber facility); Reliable NFS (non-stop NFS); Distributed Resource Manager (load balancing and coarse-grain distributed parallelism). The architecture of ISIS is depicted in Figure 9.1.

Isis was later supported commercially by a startup company called Isis Distributed Systems (IDS), later acquired by Stratus. The combination of a professional support and a competitive pricing to universities helped to disseminate the project results and made Isis a reference system in the area of group communication.

The launch of IDS did not terminate the research on group communication at Cornell. Different generations of protocols that improved previous work have been developed. The design of Isis involved with the experience gained with the usage of the system in large-scale "real-life" scenarios. Eventually, the system

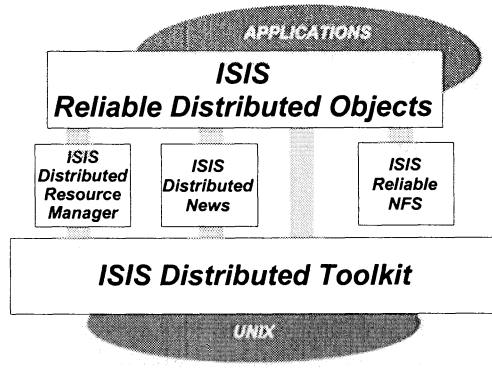

Figure 9.1. ISIS Architecture

was completely redesigned, leading to the development of Horus (van Renesse et al., 1996), a system with the same aims but with a completely different architecture, with emphasis on modularity, protocol composition, operation in partitionable environments, and security. The latest generation of the system is called Ensemble, a group communication system written in the ML programming language with a number of innovative features, such as support for formal validation of protocols (Hayden, 1998).

9.1.3 Arjuna

Arjuna (Shrivastava et al., 1991) is a distributed fault-tolerant system that integrates several software-based fault tolerance techniques. Originally developed at the University of Newcastle upon Tyne (UK), it is now commercially supported by Arjuna Solutions Ltd. Arjuna is a distributed platform, consisting of libraries, stub compilers, run-time mechanisms and services (protocols, servers) that support the development of fault-tolerant applications using programming languages such as C++ and Java. The project was started in 1987 and the first public release of Arjuna code was distributed in 1991.

Maybe the most important fault-tolerant mechanism supported by Arjuna is nested transactions. Using Arjuna, the programmer can build objects that are persistent and distributed. Object methods are invoked using remote procedure calls in the context of transactions. The system manages the state of the object, bringing the object to main memory (what is called activating the object) and later saving its new state when the transaction commits. Concurrency control is managed by Arjuna to ensure that the execution of concurrent transactions is serializable. Today, the system includes a CORBA Object Transaction Service. Object replication was also added to Arjuna, allowing different replicas of a given object to be maintained in different servers. To simplify the management of replica consistency, a multicast communication layer and group management services were added to the system.

Arjuna is an example of a research project that was able to incorporate ideas from previous transactional systems, such as Argus (Liskov, 1985) and Avalon (Eppinger and Spector, 1991), and group oriented systems such as Isis (Birman and Joseph, 1987) in a comprehensive prototype that reached the maturity to migrate to the commercial arena.

9.1.4 Delta-4

The Delta-4 (Powell, 1991) architecture was aimed at the development of fault-tolerant distributed systems, offering a set of support services implemented using a group-oriented approach. An object-oriented application support environment provides separation of concerns: it allows building applications with incremental levels of fault tolerance, while the non-functional properties concerned with achieving dependability are secured transparently from the applications programmer. Black-box commercial applications can also be rendered fault-tolerant without change, through special transformer objects (wrappers). The architecture was designed and developed by a consortium of several European corporations, institutes and universities, under the ESPRIT research programme. Delta-4 stands for "**D**efinition and **D**esign of an Open **D**ependable **D**istributed Architecture". To the authors' knowledge, Delta-4 was one of the first architectures to integrate, at all levels, modular and distributed fault tolerance concepts, namely relying on the emerging group communication and membership technologies. An excellent perspective on Delta-4 is offered by Powell (Powell, 1994).

A Delta-4 system consists of a number of computers (possibly heterogeneous) connected by a reliable communications system. The application programs consist of software components distributed among the system's nodes. A given component may be replicated, its copies being executed on different machines. Each machine consists of a *hosting node* and a *Network Attachment Controller* (NAC). NACs are dedicated communication boards where a reliable communication system is executed, supporting reliable multicast communication among computational entities. The NACs are the single hardware component specific of the Delta-4 architecture: collectively, they implement a reliable group communication abstraction under a fail-silent assumption. On top of it, hosting nodes may have any behavior, even fail-uncontrolled (i.e., arbitrary). The combination of the hosting node with the NAC forms a network node. The Local Executives[1] (LEXs) of the host machines can also be heterogeneous. Each NAC runs a real-time kernel as the execution environment. The host may run any operating system, e.g., UNIX. It may also run a real-time O.S. in the real time version of Delta-4, the XPA (*see Dynamic Systems* in Chapter 14).

The modular architecture of Delta-4 is represented in Figure 9.2. The distributed software running on the nodes can be classified as follows:

[1] Name given to the local execution environment, usually an operating system or a real-time kernel.

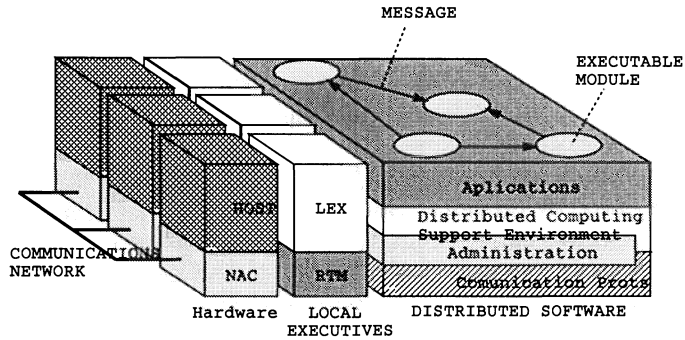

Figure 9.2. Delta-4 Architecture

- The communications software executed on the NACs; the communications system of the architecture is named the *Multicast Communication System*, offering multi-point reliable connections.
- The system administration software, managing the computational elements (including the communication system), which is executed partially in the host computer and partially in the NAC.
- The *Applications Support Environment*, named DELTASE, that supports the development of distributed applications. DELTASE was designed according to emergent standards such as the ODP model[2] (ODP, 1987) and closely followed the work of other organizations involved in the standardization process, namely the ANSA (ANSA, 1987) architecture[3].
- The user software, composed of several components, potentially developed using different programming languages, that interact through *Remote Service Requests.*

The *Multicast Communications System* (MCS) was designed using an OSI[4] like layered architecture. Delta4-XPA used a simplified high-performance version (*see Dynamic Systems* in Chapter 14). The main layers of the communication architecture are:

- *Membership and Multicast* local-area communication.
- *Reliable Transport Service*, offering reliable multi-point connections.
- *Replication Management*, a key element for implementing the fault tolerance techniques of the architecture. This level coordinates the communication among replicated access points, guaranteeing the message delivered to all addressed points and filtering duplicated messages. This component can execute error detection and error compensation protocols when needed.

[2]Open Distributed Processing.
[3]Advanced Networked Systems Architecture.
[4]Open Systems Interconnection.

- *Session* and *Presentation* multi-point services.

The services of the Multicast Communication System are used to support three fault tolerance techniques: Active replication; Passive replication; Semi-Active replication.

Active replication as implemented by MCS supports both fail-silent and fail-uncontrolled hosts, by providing non-voting or voting algorithms, respectively. Failure detection is also passed on to System Administration (for example, for cloning the replica in another node). In the second technique, *passive replication*, a component may be replicated but only one copy remains active, periodically sending a snapshot of its own state to the remaining copies. This approach can only be applied to fail-silent components. Finally, *semi-active replication* leaves the initiative to a designated replica, the *leader*, which processes input messages, generates replies, and instructs the *followers* to execute the same steps in the same order, without producing outputs. This technique allows fast error recovery, and accommodates some non-determinism not allowed by active replication.

In the Delta-4 architecture, the complexity of all fault-tolerant mechanisms is hidden from the application programmer by the support environment DELTASE, in what constituted a pioneering object-oriented transparently fault-tolerant middleware, featuring characteristics such as the provision of incremental and configurable degrees of replication and fault resilience. On the other hand, these techniques are complemented by a powerful system administration component, responsible for diagnosing system faults and triggering the appropriate fault treatment operations, such as automatic reconfiguration, cloning, and so forth.

9.1.5 The Information Bus

The Information Bus (TIB) (Oki et al., 1993) is a commercial product developed by Teknekron Sotfware Systems, Inc, a company co-founded by Dale Skeen. It consists of a distributed environment for the development of fault-tolerant applications based on the publisher-subscriber model.

The Information Bus supports an event-driven communication model, where publishers inject data objects on the bus tagged with a *subject* string. Subscribers register the subjects they are interested in with the system, and subsequently receive a copy of data objects associated with these subjects. Events can be published with different quality-of-service requirements. The weaker semantics is *reliable* delivery, which guarantees that messages are delivered only once in FIFO order as long as the network does not suffers a partition and both the publisher and the subscriber do not crash. The stronger *guaranteed* delivery ensures delivery regardless of failures; in this case the message is logged in non-volatile storage before being sent and retransmitted until a reply is received.

The Information Bus also offers an RPC mechanism with *at-most-once* semantics in the presence of failures. To establish the binding between the client

and the server, the underlying publisher-subscriber service is used. The client publishes an inquiry searching for servers. The relevant servers publish an advertisement with address information to allow the client to establish a point-to-point connection.

In order to integrate legacy applications with the Information Bus, one has to build dedicated software modules, called *adapters*, which are able to convert the data objects used in the Bus into representations understood by those applications. Today, Teknekron Software Systems is called TIBCO and offers different products that exploit the concept of reliable publish-subscribe interaction.

9.2 TRANSACTIONAL SYSTEMS

9.2.1 CICS

The Customer Information Control System (CICS), was originally developed by IBM in 1968 to support the interaction of terminals with mainframe computers. In the original CICS model, each terminal sends input messages to a server on the mainframe, which invokes a program to process the message. The server was actually a single operating system process in whose address space both CICS and all its applications execute. Making CICS execute in user space, along with that fact that CICS makes no use of peculiar operating system features, made the system extremely portable. On the other hand, executing all services in the same address space also made the system very vulnerable to software faults in any of the involved applications.

CICS integrates a wide range of services, including presentation services (that take care of data-format translations), session services (that allows programs to open, and later close, channels to other programs or devices), storage services (with different semantics), file services, transaction management, journal management, recovery management (for transaction abort, shutdown, restart and recovery from tape), program management (linking, loading and execution), thread management, authorization and authentication.

Distributed transaction processing is also supported by CICS since transaction submitted to a given CICS system can be routed to another CICS system or invoke remote operations on other systems. The interacting CICS systems can be located on the same machine or in different machines. The reason for having more than one CICS in the same machine is to improve fault-containment. In this case, remote invocations are performed efficiently using shared memory. If CICS are on different machines, communication can be performed using IBM's transactional communication service LU6.2 which offers several qualities of service, the stronger of which supports ACID properties across distributed transactions.

9.2.2 Encina

Encina is an OLTP System originally built by Transarc, a company founded in 1989 by Alfred Spector. Encina is an evolution of previous work by Spector at Carnegie Mellon University on transactional systems (Eppinger and Spector, 1991) and was designed in the framework of the Distributed Computing Environment (DCE) of the Open Software Foundation. Encina uses other DCE services as building blocks, such as the DCE RPC for implementing remote procedure calls and Kerberos for authentication and encryption. Transarc became a subsidiary of IBM in 1994, and Encina is now part of the IBM's transaction processing products, TXSeries[tm], that also includes CICS and the distributed file system AFS.

To simplify the task of building distributed applications with transactional semantics using the C programming language, Encina includes a specialized programming environment called Transactional-C. This environment includes a number of extensions to the C programming language, such as directives to express concurrency (supported in run-time by a thread package) allowing C programmers to launch concurrent sub-transactions in an elegant way. With the advent of the OMG architecture, Encina was enhanced to support the development of C++ servers and C++ or Java clients running on the Orbix object request broker.

9.3 CLUSTER ARCHITECTURES

Cluster is a designation that has been used to describe a family of systems with quite diverse characteristics. In general terms, a cluster is an integrated collection of machines that, to some extent, can be managed has a whole and, sometimes, can be interfaced by the external nodes as a single machine. Clusters have been built to offer high processing power (as a cheap alternative to expensive specialized multiprocessor architectures) or to offer higher availability, usually by providing a convenient way to restart a service on a different cluster node when a crash occurs. Most hardware and software vendors have cluster products, including Sun, Compac, HP, IBM, Microsoft, etc. A complete book can be written and has been written just about clusters (Pfister, 1998) so we will concentrate on features related to fault tolerance.

The key aspect of clusters, from the availability point-of-view, is that they are made of several processing nodes and several storage nodes interconnected by some high speed link. It is possible to have each storage node statically assigned to each processing node. However, storage nodes are frequently shared among processing nodes, either by making the storage nodes interface the network or by using dual-port disks. These alternatives are illustrated in Figure 9.3. Alternative (c) has received enormous interest recently, given the need for huge amounts of storage to be offered in a shared and distributed way. Products emerging under this technology are called *Storage Area Networks (SAN).*

It is possible to develop applications that make use of the existing redundancy to run in non-stop or reconfigurable modes. Examples of the latter are

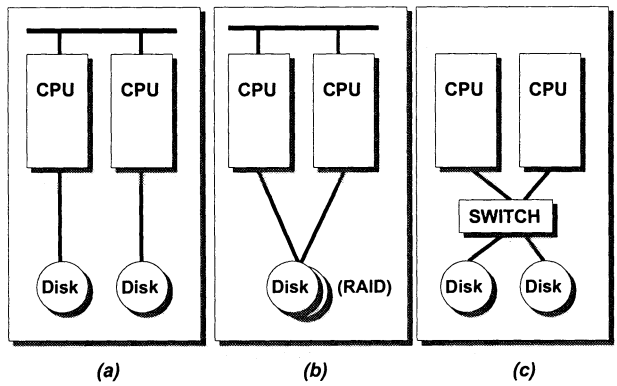

Figure 9.3. Storage in Cluster Architectures: (a) Non-shared; (b) Local sharing; (c) Network sharing

fault-tolerant versions of file systems, such as the Highly Available NFS (HA-NFS) (Bhide et al., 1991), or of database servers such as the ones available from vendors such as Oracle or Informix. However, clusters are often used in recoverable mode. Nodes are used for load sharing, individually running "standard" applications unaware of cluster facilities. The cluster software allows the application to be launched on any computing node. If a crash occurs, the application goes down, and cluster management facilities are able to launch the application in another node of the cluster.

Services provided by the cluster software and hardware usually include failure detection services, reliable communication services, event managers that are able to provide each member of the cluster with information about the cluster status (for instance, to distribute load information), configuration managers (where administrators may specify dependencies among applications and location policies), load balancing managers, fail-over managers, etc. With regard to failure detection it is worth noting that the use of dedicated networks to connect the cluster nodes increases the synchrony of the system and, therefore, renders failure detection easier. Nevertheless, most clusters that use a primary-backup approach exploit specific hardware interfaces, like the SCSI challenge-defense protocol, to ensure that two nodes cannot simultaneously consider themselves as primary.

9.4 MAKING LEGACY SYSTEMS DEPENDABLE

Most of the examples described in the previous sections used the approach of constructing a fault-tolerant system or toolkit from scratch. Clearly, this is the most effective approach to optimize the efficiency of solutions. However, the system architect is often challenged with the task of making legacy systems dependable without re-implementing existing applications (a task that may not be feasible because of timing or cost constraints).

One of the most successful approaches to deal with legacy systems consists in building a new interface to the legacy component. The new interface is responsible for intercepting inputs and outputs and manipulating these interactions to make them suitable to implement a given fault-tolerant strategy. Consider for instance that you have a legacy system that behaves like a state machine. In this case, the system can be made fault-tolerant by replicating the component. This means that the interface must intercept the inputs and multicast them in total order to all replicas. The outputs must also be intercepted and combined in a single output. The approach is illustrated in Figure 9.4.

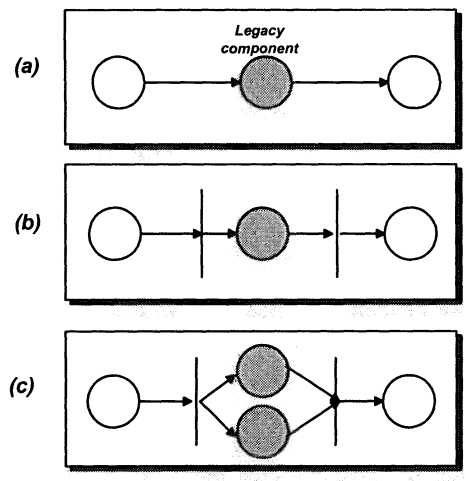

Figure 9.4. Interfacing Legacy Systems

This approach has been used successfully in many systems under different names, such as *transformers* in the Delta-4 project, *adapters* in the TIB approach, *wrappers* (Birman, 1996), or *metaobjects* (Fabre et al., 1995). There are many ways of implementing wrappers. If the component interacts with other components through message passing, it obviously becomes easier to make the interposition. If all interactions are performed using some standard RPC-like mechanism it becomes even simpler. For instance, CORBA (OMG, 1997a) components can be wrapped by generic components using dynamic invocation facilities (Felber et al., 1996). Otherwise some other operating system or language-specific techniques may be used. For instance, specialized dynamically linked libraries may be provided to intercept all the system calls made by a given component. Alternatively, specialized language libraries or specialized language run-time environments can be provided to support the component execution.

It should be noted that sometimes it is feasible to alter the way the clients interact with the servers. In such case fault-tolerant features can be embedded in the client protocol, for instance enhancing clients with the ability to multicast requests to a group of servers. At first glance this may look a bit intrusive but

there is a general tendency to make legacy applications available through new interfaces, such as web browsers, even when no fault tolerance concerns are involved. Additionally, technologies such as Jini (Waldo, 1999) allow the client to dynamically download the right interface from the name server.

On the other hand, if it is not affordable to change clients, the wrapper must make the fault-tolerant features fully transparent for the client protocol. Consider for instance the case where the client uses the UDP protocol to communicate with the server and that a primary-backup scheme is selected to make the server fault-tolerant. In case of failure of the primary, the backup must be able to impersonate the IP address of the primary such that the client continues to use exactly the same address as before. Of course, migrating an TCP connection without disruption requires a bit more work.

9.5 SUMMARY AND FURTHER READING

This chapter has given concrete examples of research and commercial fault-tolerant systems. The number of systems cited was necessarily small and many excellent systems were not mentioned. For access to downloadable software or further reading, Table 9.1 gives a few pointers to information about some of the systems described in this chapter.

In terms of fault-tolerant systems coming from industry, documentation on the concepts underlying the Advanced Automation System prototype (Cristian, 1994; Cristian et al., 1996) and the Parallel Sysplex Cluster (Bowen et al., 1997a; Bowen et al., 1997b) are definitely worth reading. For detailed material on cluster architectures, see (Pfister, 1998).

Group communication systems have been a field of very reach research. In addition to the work at Cornell, many other groups have developed long and solid work on the area. We list some of the most relevant systems: Transis (Amir et al., 1993a), Totem (Moser et al., 1995) and the associated concept of extended virtual synchrony (Moser et al., 1994), Psync (Peterson et al., 1989) and Consul (Mishra et al., 1993), NavTech (Rodrigues and Veríssimo, 1995; Rodrigues et al., 1996), Relacs (Babaoğlu et al., 1994) and RMP (Callahan and Montgomery, 1996).

For further work on object-oriented fault-tolerant middleware, see (Fabre et al., 1995). Other relevant systems are Manetho (Elnozahy and Zwaenepoel, 1992b; Elnozahy and Zwaenepoel, 1992a) and Harp (Liskov et al., 1992). One of the pioneer works on transaction systems was Argus (Liskov, 1985) developed at MIT by Barbara Liskov, whose most recent creation is THOR (Liskov et al., 1999). Fault-tolerant transactional systems based on coordinated atomic actions are described in (Xu et al., 1999).

Table 9.1. Pointers to Information about Fault Tolerant Systems and Platforms

Class of System	*System*	*Pointers*
IETF RFCs		www.rfc-editor.org
OMG	(FT-CORBA)	www.omg.org
IEEE-IFIP	(Dependab.)	www.dependability.org
Message Buses		
	TIBCO	www.tibco.com/
	iBus	www.softwired-inc.com/
Group Communication	**ISIS**	www.cs.cornell.edu/Info/Projects/ISIS
	HORUS	www.cs.cornell.edu/Info/Projects/Horus
	Ensemble	www.cs.cornell.edu/Info/Projects/Ensemble
	Relacs	www.cs.unibo.it/projects/relacs
	Transis	www.cs.huji.ac.il/labs/transis
	Totem	beta.ece.ucsb.edu/totem.html
	Phoenix	lsewww.epfl.ch/projets/phoenix
	xAMp	www.navigators.di.fc.ul.pt/
	Spread	www.spread.org
Checkpointing		
	Manetho	www.cs.cmu.edu/~mootaz/manetho.html
	Libckpt	www.cs.utk.edu/~plank/plank/www/libckpt.html
	Egida	www.cs.utexas.edu/users/lorenzo/lft.html
Transactional systems	**Argus**	www.pmg.lcs.mit.edu
	THOR	www.pmg.lcs.mit.edu/Thor.html
	Arjuna	arjuna.ncl.ac.uk
Fault-tolerant Systems	**Eternal**	beta.ece.ucsb.edu/eternal/Eternal.html
	Electra	www.softwired-inc.com/people/maffeis/electra.html
	Cactus	www.cs.arizona.edu/cactus
	OGS	lsewww.epfl.ch/OGS
	Filterfresh	www1.bell-labs.com/org/11356/
Validation and Verification Tools	**LAAS**	www.laas.fr
	Critical	www.criticalsoftware.com
	Ballista	www.cs.cmu.edu/~koopman/ballista
	ULTRASAN	www.crhc.uiuc.edu/PERFORM
	Orchestra	www.eecs.umich.edu/RTCL/projects/orchestra
	Kronos	www-verimag.imag.fr
	PVS	pvs.csl.sri.com
Clusters and Commercial Platforms	**IBM**	www.research.ibm.com
	TANDEM	www.tandem.com
	STRATUS	www.stratus.com
	Microsoft	research.microsoft.com
	Compaq	www.compaq.com/enterprise/highavailability.html

III Real-Time

I want a real-time system where I can log on.

— Alan Burns

Contents

Overview

Part III, Real-Time, discusses how to ensure that systems are timely, under a number of circumstances, including faults, overload, uncertainty. It is especially concerned with real-time in distributed systems. Chapters 11 and 12, Fundamental Concepts of Real-Time and Paradigms for Real-Time, address the fundamental notions and misconceptions about real-time, in a distributed context. The main paradigms are presented, in a comparative manner when applicable, such as synchronism versus asynchronism, or event- versus time-triggered operation. Chapter 12 further addresses issues such as: real-time networks, real-time processing, real-time communication, clock synchronization, and input-output. Chapters 13 and 14, Models of Distributed Real-Time Computing and Real-Time Systems and Platforms, show how to achieve timeliness of distributed systems in the several real-time classes— hard, soft or mission-critical— and models— time-triggered and event-triggered. Chapter 14 gives examples of distributed real-time systems in several settings. Chapter 15 continues the case study, this time about making the VP'63 System timely.

10 CASE STUDY: MAKING VP'63 DEPENDABLE

This chapter takes the next step in our case study: making the VP'63 (VintagePort'63) Large-Scale Information System dependable. Increased reliance on computers for day-to-day operation on the one hand, and greater geographical dispersion of the system on the other, have raised concerns about the impact of service outages or even severe failures on the business results. In consequence, part of the study concerns the enhancement of the reliability and availability of the VP'63 system.

10.1 FIRST STEPS TOWARDS FAULT TOLERANCE

The reader should recall that this is the next step of a project implementing a strategic plan for the modernization of VP'63, started in Chapter 5, and continued in the Case-Study *chapters of each part of this book. The reader may wish to review the previous part, in order to get in context with the project.*

The development strategy laid out will impose a growing dependence of the business on the computing infrastructure. The most important facet of the desired dependability of the services provided by VP'63 is thus availability. Whilst it is desirable that the system does not fail often (reliability), it should also exhibit small glitches, remaining operational for a very high percentage of its life time.

The client-server front-end to the database is already fragmented. This is a first step at independence of failure, as recalled in Figure 10.1a: local

P. Verissimo et al. *Distributed Systems for System Architects*

transactions will not be affected by the failure of servers of the other fragments. However this is not enough to guarantee overall availability, since operation related with that fragment stalls.

On the other hand, the publisher-subscriber infrastructure is based on a single-location server (Lisboa) which, once down, stalls the whole service, as depicted in Figure 10.1b. This is a severe availability impairment.

The 3-tier DFS-Web architecture has already a few aspects enhancing availability: RO file or volume replicas are accessible by any client, and so the initial objective of performance also serves availability, since upon failure of the local RO copy, the client can fetch a remote one. This may eventually be extended to replication of RW files or volumes, with a DFS that support this feature.

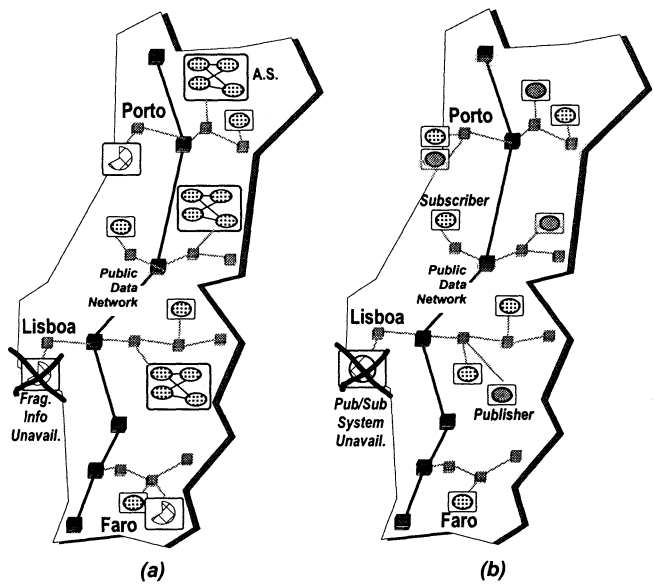

Figure 10.1. Failure Scenarios: (a) Fragmented DB; (b) Pub-Sub System

10.2 FAULT-TOLERANT CLIENT-SERVER DATABASE

Q.2. 1 What can be done to improve the availability of the database server?

One possible answer is to go for a replicated distributed database server. All fragments are now replicated in other sites. The team decided to use a level of replication of $n = 2$ for the first prototype, since in normal environments, with a fair maintainability, known statistics show that availabilities in excess of two nines can be achieved. Since the database is fragmented by enough sites (*see* the initial situation in Figure 5.3a), it is not necessary to allocate extra machines to achieve fault tolerance.

The modular fault tolerance principle is used, cross-allocating fragment replicas to sites containing other fragments, as shown in Figure 10.2a. However, tests

have shown that the risk of network partitioning is non-negligible in the links off the main inter-city connections. This can make the replica pairs diverge.

Q.2. 2 What can be done to address partitioning of the database replicas?

If fragments are replicated at least in triplets, the primary-partition consistency criterion can be used, preventing divergence by allowing progress only in a partition with the majority of replicas.

Q.2. 3 Considering that the DBMS used allowed modular fragmentation and replication, will there be situations where it makes sense to allocate different redundancy levels to different fragments?

In the course of this observation, it was also decided to group crucial data in a main fragment (MF) and replicate it in the main site and all islands, increasing the level of redundancy of this data. This configuration study will later be extended to other data.

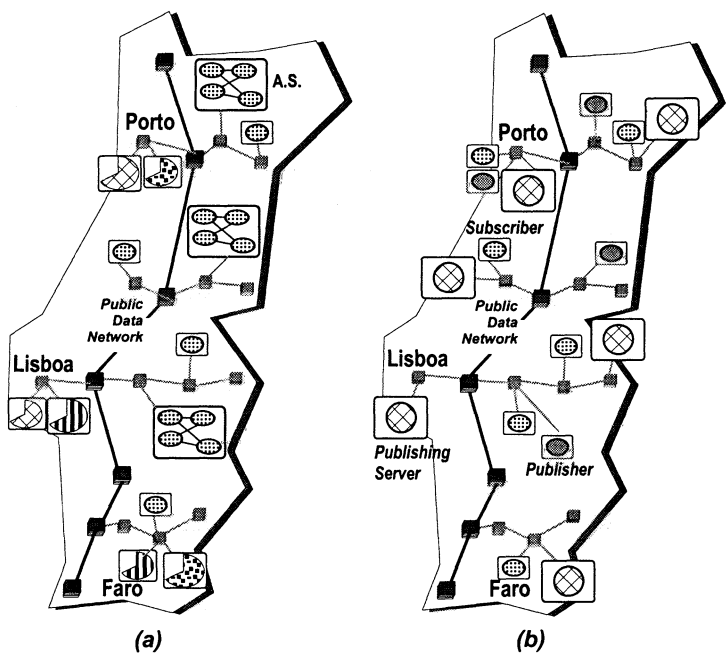

Figure 10.2. Fault Tolerance: (a) Client-Server Database; (b) Publisher-Subscriber

10.3 FAULT-TOLERANT DATA DISSEMINATION

Q.2. 4 What can be done to improve the availability of the pub-sub server?

With a simplex server, the initial situation depicted in Figure 5.2b, the whole dissemination system stops when the server is down. Plus, data may be lost,

unless there is a transactional interface between publishers and server, and the latter has persistent storage.

A possible solution is to set up a replicated publisher-subscriber server. This may also have the benefit of improving the performance of the scheme since subscribers get in general nearer the information bus materialized by the publishing server. Figure 10.2b shows the architecture: data are published through all replicas, subscribers get data from one of them. In fact, the load of dissemination to the subscribers can be distributed by the publishing servers.

10.4 FAULT TOLERANCE OF LOCAL SERVERS

The efforts described so far were aimed at achieving fault tolerance of the global services, taking advantage of replication at other facilities. This entails two consequences: (a) undesirable service degradation as seen locally (e.g., when clients in an island have to fetch from a remote database copy due to failure of the local server); (b) unacceptable local execution glitches during reconfiguration (e.g., in a production process controller).

Two instances of the problem were identified: the main publishing/subscribing server in Lisboa, and the production control and management server in the main wine processing facility. The idea is to increase the reliability figures for the related servers, that is, taking steps to prevent failure of these components. The team has considered the utilization of either or combinations of highly-available units and clustered computers. Highly-available units are classic approaches, with embedded multiprocessors, redundant power supplies and RAID disks. Clustered computers over a fast LAN offer the advantage of modularity and use of COTS components, being in turn more complex to manage.

Further Issues

These issues need some refinement now, and the reader was assigned the study of a few questions that were still left to be solved:

Q.2. 5 *Study concrete measures based on highly-available or clustered computers to achieve "non-stop" operation of the Lisboa pub/sub server.*

Q.2. 6 *Consider an island DB fragment: study a scheme for doing local replication of that fragment, for local availability in case of one failure.*

Q.2. 7 *Study a quorum replication scheme that ensures the continued operation of the main DB fragment when both Porto and Lisboa are up and connected.*

Q.2. 8 *Study a replication scheme guaranteeing that a copy of the whole database exists in Porto. The extra fragment copies: can be read-only; need not be updated as timely as the others.*

Q.2. 9 *Define and justify the necessary communication semantics (e.g. ordering, reliability) between: publishers and publishing server replicas; server and subscriber groups.*

11 REAL-TIME SYSTEMS FOUNDATIONS

This chapter addresses the fundamental concepts concerning real-time, starting with the definition real-time and clarifying a few current misconceptions. It traces the evolution of real-time computing towards distribution and discusses its relation with fault-tolerance. Finally, the most relevant architectural approaches to real-time in networks and distributed systems are introduced, to be detailed in the subsequent chapters of this part.

11.1 A DEFINITION OF REAL-TIME

Intuitively, real-time systems are systems that deal with time. This probably means "doing things on time". But what is time anyway? And... which time exactly? The systems' time? The users' watch time? In fact, one of the fundamental problems in real-time systems design is how to relate the several dimensions of time in a distributed system: time amongst the several sites; time between a site and its users (human and other); time between the system and the environment; time between a controller and the controlled system. When thinking about how to address these problems, we face important issues concerning real-time and distributed systems:

- modeling the interaction between the computer and the real world (an oven is not a computer);
- maintaining temporal accuracy of measurements (the temperature of the oven 30 minutes ago may be useless now);

P. Verissimo et al. *Distributed Systems for System Architects*

- accommodating important load versus finite resources (a 10Mbaud real-time network is useless for 20Mbaud loads);
- recognizing deadlines and urgency (an urgent message to be delivered within the next 10 milliseconds should not stay at the tail of a long queue of non-urgent messages);
- tolerating faults (no deadlines were ever met by a crashed RT computer).

When the role of time is misunderstood, a few misconceptions may arise about what a real-time system is. We will address and clarify the most common ones, after attempting at defining a real-time system.

For the treatment of the Real-Time Part, it is assumed that the reader is familiar with most of the distributed systems paradigms discussed in Chapter 2, and definitely with the material developed in Sections 2.5, 2.6, and 2.7 of that chapter. We will be using the acronym RT to mean 'real-time' in this part.

11.1.1 What is a Real-Time System?

Let us try and answer this question in a precise way. What dictates the 'real-time problem' is the need to synchronize our actions with the environment. The environment has its own pace, and thus we have to make our system adapt to this pace, and react according to the evolution of the environment. This proves more difficult than it looks, and justifies *real-time* as a research area of its own right. By convention, **real-time** (with slash) is the keyword chosen to designate this area. Do not confuse with the abstraction **real time**, the (unobservable) universal time reference, the same everywhere in the system, also called newtonian time, that marks the passage of time (*see Times and Clocks* in Chapter 2). We find as examples of real-time systems: oven controller; manufacturing cell; fly-by-wire controller; traffic lights control system; air traffic control system; multimedia computer game system; command, control and communication systems.

Real-time System - system whose progression is specified in terms of *timeliness* requirements dictated by the environment

This generic definition implies several corollaries that have been used as alternative definitions of real-time system. For example, a real-time system is a system where *the correctness of a computation is defined both in terms of the logical results and the time at which they are provided.* On the other hand, timeliness requires synchrony (*see* Section 2.6): a real-time system can be seen as a system that *has the capacity of executing actions within pre-specified intervals.* In fact, real-time can be seen as the body of principles and techniques for specifying and building *synchronous* systems. As a final corollary, a real-time system *is a system that provides at least one real-time service.* That is, although RT and non-RT services may coexist in the same system, one single RT service is all it takes for requiring the system to have real-time behavior. Examples of real-time services are: to read a sensor cyclically; to activate a valve at a precise instant; to reply to a request or deliver a message in bounded time; or to execute a task within a given interval.

There are several **classes** of real-time systems, because the kind of constraints put by matching the environment to the timeliness requirements vary. The existence of the classes is justified by the fact that different architectures and paradigms address different problems in each class, and it is difficult if not impossible, to address all of them with a single architecture. In consequence, real-time system architects are used to distinguishing between:

- *hard* real-time systems, where timing failures are to be avoided
  — example: on-board flight control system (*fly-by-wire*);
- *soft* real-time systems, where occasional timing failures are accepted
  — example: on-line flight reservation system;
- *mission-critical* real-time systems, where timing failures should be avoided and occasional failures are handled as exceptional events
  — example: air-traffic control system.

How is timeliness specified in real-time systems? We learned that timeliness is essentially about bounding delays. In this part, we are going to see that this may have several facets. Namely, the real-time systems community often uses terms with a specific meaning: deadline, liveline, targetline, release time, slack, laxity, jitter, latency, delay, etc. Whenever it is the case, we will try to establish the mapping onto corresponding terms in distributed systems.

A **distributed real-time system** is a particular instance of real-time system, with two major characteristics:

- timeliness guarantees have to be provided over a system of sites interconnected by a network— which is harder to do;
- once provided, timeliness guarantees can be associated to other attributes, such as modularity, geographical separation, failure independence, load balancing, and others discussed in Chapter 1— which is an additional advantage.

Some issues concerning the architecture and functionality of distributed systems assume new dimensions under a real-time perspective, which we address throughout this part:

- time-related aspects of fundamental concepts and paradigms (e.g., synchrony, time and order across multiple sites);
- models enforcing timeliness (e.g., time-triggered or event-triggered);
- scheduling (e.g., distributed and dynamic);
- communications (e.g., real-time networks and protocols);
- input-output (e.g., multiple sensor and actuator handling);
- dependability (e.g., replicated sensors, actuators and controllers).

A class of systems intimately related with real-time are the **embedded systems**, also called application-specific systems, where individual components have specific tasks or assignments. It is essentially a black-box system, made on purpose for an application. Embedded systems can be specially made from scratch, or they can be integrated with modular components such as OEM

board families, or even from COTS[1] components such as PC CPU boards and peripherals. Although some authors consider them so, embedded systems are not always real-time systems. Nevertheless, they are real-time or at least interactive for their great majority, and as such a great deal of what is written about embedded systems concerns real-time.

11.1.2 Misconceptions about Real-Time

Despite the evolution and visibility of real-time (RT) computing over the past few years, a number of misconceptions about real-time still persist. There is a very complete compilation of them in (Stankovic, 1988), of which we extract the main examples:

- RT is ad-hoc design, assembly programming, interrupts, and so forth;
- RT systems are automata, pre-programmed and static;
- RT is about having enough speed, and ever-increasing MIPs and Mbauds will solve all "performance" problems;
- RT deadlines do not make sense, since they will be missed because failures occur, messages get lost, software has bugs, etc.

Real-time is currently much more than a collection of clever hardware and firmware engineering principles. If final system enhancements may benefit from ad-hoc tuning, the fact is that real-time systems design obeys a systematic that goes from architecture to programming languages, and algorithms.

The realm of static RT systems represented by PLCs, PID controllers and similar devices is but one of the facets of today's real-time systems. The "real-time systems where one can log on", many of them distributed, have pervaded the scene of interactive systems, simply because user demand has required more predictable systems in the time domain.

Many people still equate 'real-time' with 'performance'. However, real-time *is not* about performance, but about **predictability**. It is about "within 100 seconds" and not about "as fast as possible". If all actions are executed within 99 seconds each by system Slow, it performs excellently, though on average more than 100 times slower than system Fast, which executes most actions below the second. But we had not asked for that had we? Furthermore, speed will not solve all problems. The more resources we have, the more we will spend. The Wintel[2] saga is the best example available today. The point is about **scheduling** resources correctly so that *deadlines* are met. If system Fast executes one thousand actions below one second, and then takes 101 seconds to execute just one action, it will have failed.

Finally, a word about reliability: all systems fail, so the fact that measures are taken to secure timeliness specifications in normal operation does not

[1] Commercial Off The Shelf.

[2] Acronym to denote the folkloric notion that the more powerful Intel PCs become, the more power-hungry Microsoft software gets.

exempt the designer from either designing for the worst-case situation, or endowing the system with the necessary mechanisms to tolerate faults when they happen.

11.1.3 Evolution of Real-Time Computing

A few of the major milestones in the evolution of real-time systems in the past few years are enumerated in Table 11.1. Real-time in scientific terms probably started with the exploratory transatlantic navigations in the fifteenth century, with the combined use of existing knowledge on astronomy, cartography, and mechanics. Determining coarse points in the sun's movement, such as noon, evolved to measuring the minute accurately, around 400 years ago, followed later by the second, with the appearance and evolution of the chronograph (Boorstin, 1983). More recently atomic clocks, e.g. in the GPS NavStar navigation system (Parkinson and Gilbert, 1983), yield clock accuracies of the 10^{-7}th of a second.

Real-time in the realm of electronics started with control systems, made of hard-wired relay or digital systems. Then, early real-time computing systems appeared, in the form of dedicated computers for fixed-base applications, most of them military, such as SAGE or Whirlwind (Redmond and Smith, 1980). Operating systems evolved in order to provide support for development of generic real-time applications.

The advent of microprocessors opened the way for small and cheap embedded control units, modularly built around families of standard form-factor cards, relying on firmware-based real-time multitasking kernels (*see* Section 14.1). Microprocessors also gave a push to the development of black boxes such as PLCs (programmable logic controllers) and PID (proportional, integral, differential) process controllers, aimed at replacing relay and analog electronics with more versatile (programmable and settable) modules. The PLC normally polls the sensors and issues commands to the actuators based on some control program.

Distribution appeared in the form of interconnection of the above-mentioned components over real-time LANs, such as Token-bus (Token Bus, 1985), or the so-called *field buses*, simplified LANs which are a kind of digital system over a long wire, such as (CAN, 1993). MAP, the Manufacturing Automation Protocol (MAP, 1985), despite its shortcomings, was a major cultural breakthrough, bringing real-time networking into the *shop-floor*. In its first steps, distribution in computerized control was mostly concerned with replacing point-to-point cabling, through field buses: the central unit executed an automaton which read information from and sent commands to remote units on a polled, synchronized basis. More recently, we have witnessed an evolution towards using field buses as support for distributed control systems, supported by paradigms such as client-server, state-machine, or producer-consumer.

Distribution penetrated in the real-time arena for several reasons:

- geographical separation— the nature of most real-time control problems is distributed;

Table 11.1. Major Milestones in Real-Time Computing

1947	Early RT systems (Whirlwind) (Redmond and Smith, 1980)
1957	Early RT systems (SAGE) (Redmond and Smith, 1980)
1973	Rate Monotonic Scheduling (Liu and Layland, 1973)
1982	Early COTS multitasking executives (VRTX,RMX)
1983	NavStar GPS (Parkinson and Gilbert, 1983)
1983	Ada Programming Language (ADA 83, 1983)
1984	Basic Imprecision of clock synch. (Lundelius and Lynch, 1984b)
1985	Early RT LANs (Token Bus, 1985)
1985	Manufacturing Automation Protocol (MAP, 1985)
1987	Priority Inheritance (Cornhill et al., 1987)
1987	Real-Time Databases (Son, 1987)
1988	Early Field Buses (MIL-STD-1553B, 1988)
1990	Early Field Buses (FIP, 1990)
1989	Round-trip Clock Synchronization (Cristian, 1989)
1989	Network Time Protocol (Mills, RFC1119)
1991	Non-centralized Field Buses (Profibus, 1991)
1993	Non-centralized Field Buses (CAN, 1993)
1993	Deterministic (DCR) Ethernet (Le Lann and Rivière, 1993)

- decentralization— many of these problems involve interacting clusters with local autonomy;
- parallelism, load balancing, replication— desirable system characteristics come as artifacts of distribution.

Distributed real-time systems evolved along three main axes. *Embedded control and field-bus* distributed systems are mainly devoted to computerized control, and essentially try to use distributed systems techniques in low-level networks and simplified nodes. *Large-scale mission-critical* systems resort to dynamic techniques in order to secure deadlines under the uncertain circumstances encountered in the complex environments they normally address. *Soft real-time* systems represent today a great part of the interactive systems, namely multimedia rendering and conferencing systems on the Internet, and try to ensure timeliness specifications in a probabilistic way, namely through QoS negotiation and adaptation techniques.

11.1.4 Real-Time and Fault Tolerance

It is hard to think seriously about real-time without considering fault tolerance. We defined real-time system as one whose progression is specified in terms of timeliness requirements dictated by the environment, because the real world does not wait. However, it does not wait either in normal situations or in faulty situations. In consequence, ensuring timeliness in fault-free situations is complementary to ensuring it under faulty situations.

For example, suppose a worst-case message delivery time analysis for a time-critical LAN that only takes into account medium access schedulability, that is, how frame transmissions are distributed in time among the several nodes.

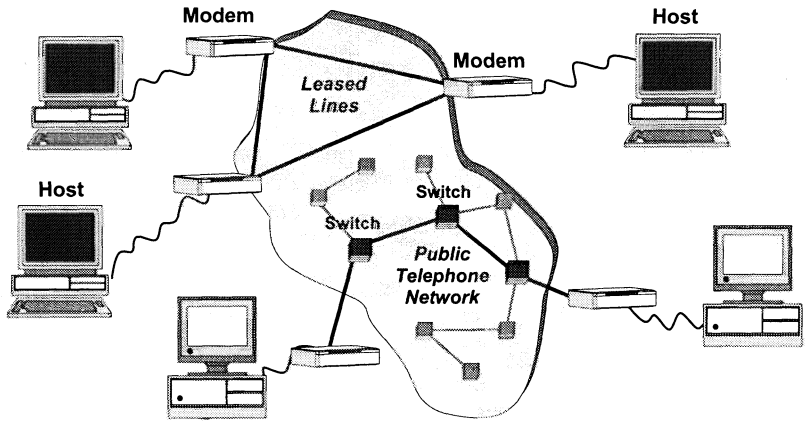

Figure 11.1. Physical Circuit

If omission faults are not taken into account, then the calculated bound will be exceeded when faults occur. If those faults are not tolerated, it will even be impossible to calculate the real bound. Even under a soft real-time perspective, we need availability to fulfill probabilistic deadline assurances. "As soon as the computer is up again" may not be compatible with the specification "any transaction should terminate within 10 seconds for at least 90% of the times, and within 1 minute in 100% of the times" with which we exemplified soft real-time systems requirements earlier. In dependability terminology, *reliable real-time* means *tolerating or preventing timing faults*, according to a pre-defined *fault model.* During this part, we will have the opportunity to see a few examples of the symbiosis between real-time and fault tolerance. It is thus expected that the reader has been exposed to relevant materials in the Fault Tolerance Part, for example, Chapter 6 and the first part of Chapter 8.

11.2 REAL-TIME NETWORKS

The first step towards setting up a distributed real-time architecture is providing the system with adequate networking structure, capable of exhibiting real-time behavior. Several network architectures serve this purpose.

A primordial structure with real-time properties is the **physical circuit** type of network. Real-time behavior, as depicted in Figure 11.1, is achieved by guaranteeing a physical path between any two endpoints, normally a combination of wires, repeaters, and switches. For example, the capabilities of telecommunications circuits for carrying real-time data are often forgotten : permanent leased lines or switched circuits such as digital telephone, ISDN or GSM. This avoids the sharing problems, such that the latency is virtually constant and equal to the propagation delay between the two points. A complementary problem is ensuring that the individual load does not exceed the throughput of the link. Physical circuits have been used to structure early dis-

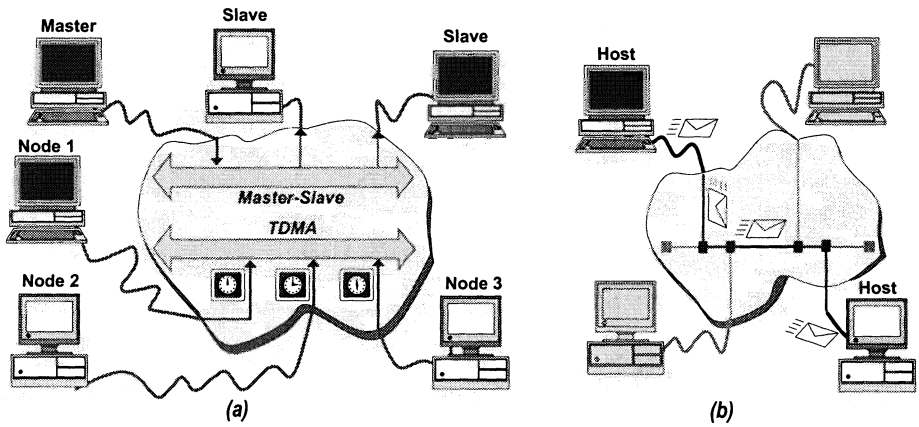

Figure 11.2. (a) Digital Bus; (b) Virtual Circuit

tributed real-time systems, namely in the fault-tolerant area (*see* Section 6.4), often resorting to special-purpose networks such as graphs of point-to-point links, or single-sender broadcast links. On the other hand, there are examples of remote control and interactive telemedicine systems built over ISDN-based computer telephony.

Another structure with real-time properties is what we might call **digital bus** type of network. Its philosophy was inherited from the centralized digital systems era, when the need arose to extend the geographical reach of control systems. Typical instantiations of the digital bus network structure are early instrumentation or field buses based on the master-slave principle, such as GPIB (IEEE-488.2), MIL-STD (MIL-STD-1553B, 1988), or FIP (FIP, 1990). TDMA-based structures are also examples of digital-bus networks (Kopetz and Grunsteidl, 1993). As suggested in Figure 11.2a, real-time behavior is ensured because although the medium is shared there is no uncertainty-causing contention, since it is shared on a pre-determined basis: a master polls all slaves; or a global clock determines the transmission slot for every node. Again, it is necessary to ensure that the global load does not exceed the throughput of the medium.

In the past few years, a number of real-time LANs have appeared, some of them standardized such as Token-bus (Token Bus, 1985), others proprietary such as DCR-Ethernet (Le Lann and Rivière, 1993). Field buses have also evolved toward a decentralized nature, emulating existing fully-fledged LANs, such as Profibus (Profibus, 1991), or CAN (CAN, 1993). These networks exhibit ring or bus shared media topologies, decentralized clocking and control, and prefigure the most common real-time network structure today, the **virtual circuit**, of the frame switching type. Real-time behavior must be achieved through a sharing policy, implemented by *medium access control* mechanisms, which ensure that a virtual path is established for each packet transmission, as suggested in Figure 11.2b. Unlike non real-time LANs, which privilege fair-

ness of medium sharing, these mechanisms aim at letting through the most important (with higher priority) frames ahead of the others, controlling the maximum amount of time each node transmits, and/or bounding the interval between successive transmission opportunities for each node. These mechanisms aim at controlling the **latency** observed by each frame. Complementary to this, it is always necessary to match the total load offered by all nodes to the maximum **throughput** achievable by the medium.

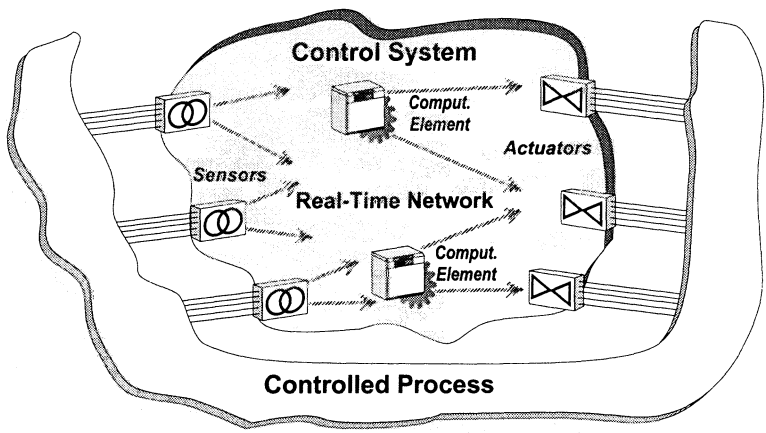

Figure 11.3. Real-Time Control Architecture

11.3 DISTRIBUTED REAL-TIME ARCHITECTURES

This section gives an overview of the main architectures for distributed real-time systems. One such architecture is the one supporting **real-time control** (real-time started with control systems). The main components of the architecture, as depicted in Figure 11.3, are the controlling system, which is the computational part, and the controlled system, the physical system under control, which we will call *environment*, for simplicity. It is very important to understand the behavior of the controlled system, since several communication and interaction paths go through it, as shown in the picture. These are called *feedback* paths. The points of contact between the controlling system and the environment are the sensors and the actuators. The control activity takes place by having the *sensors* acquire the state of the environment (e.g., the temperature of an oven, or the flow in a pipe). That information is sent to the *computing elements* of the distributed control system where it is processed, and then the reply is issued in the form of commands to the *actuators* (e.g., open the burner throttle to increase the temperature, close a valve to decrease the flow). The environment reacts to these stimuli and provides feedback as if it sent 'messages' to the sensors, closing the loop we see in the figure. Timeliness requirements are expressed differently, depending on the kind of control. Discrete control requires individual actions to be taken in bounded

time, in response to events from the environment or from the controlling system (e.g., taking a robot's arm from under a 1 ton press before it comes down). Continuous control requires that a variable is maintained within an allowed value interval, although this translates, other aspects solved (such as sensing errors), to timeliness requirements: correcting it periodically, often enough to compensate for slow steady state variations; correcting it sporadically, quickly enough after disturbances (e.g., maintaining the temperature of an oven, both in steady state and after doors are opened and new, cold materials inserted). Most hard real-time systems are devoted to control and thus follow this architecture, where the main timeliness issue is to ensure that all the control loops are served frequently enough and/or sporadics are treated quickly enough.

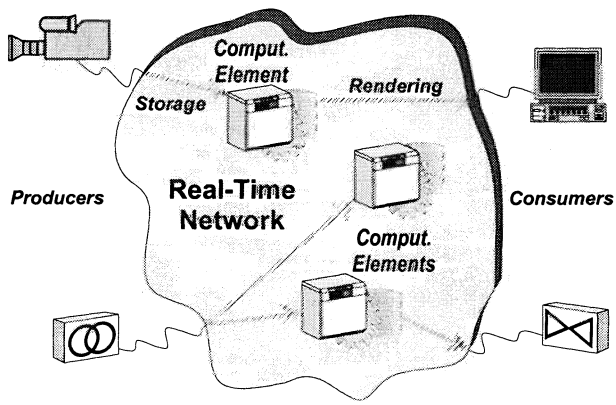

Figure 11.4. Real-Time Producer-Consumer Architecture

Another relevant architecture is the one supporting **real-time producer-consumer**. It is concerned with supplying information (data, messages) from one or more producers to one or more consumers, such that the throughput, or the latency, or both, are kept within pre-specified bounds. As depicted in Figure 11.4, data is originated in the *producers* (e.g., images from a digital camera), then processed by the distributed *computing elements* of the architecture (e.g., compressed and encoded, and/or stored), and finally delivered to the *consumers*, which absorb it. By absorption, we mean that there is no special timeliness concern associated with the information once arrived at an end consumer, that is, we might consider the consumer to be an infinite sink. Furthermore, note that in this example, data not only has to arrive at an average minimum rate (throughput), but also at a steady pace, which implies an upper bound for the instantaneous delivery latency of each message. Competing producers sending to the same set of processing nodes further complicate the issue, since it is necessary to ensure the throughput and latency properties for the whole flow. Multiple consumers just require the ability of the system to multicast the information to them in real-time. Observe that for certain applications, such as video-on-demand, the producer-consumer path is only relevant from the storage server to the consumer client, since the contents have been

generated and recorded beforehand. There are hard real-time applications constructed on this kind of architecture, which normally imply totally deterministic flows from producer to consumer, such as in embedded multimedia processing systems. There are also soft real-time instantiations of the producer-consumer architecture, where constraints can be relaxed, such as Internet sound and video rendering.

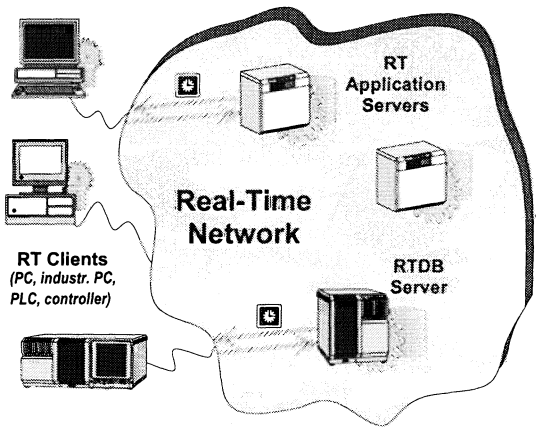

Figure 11.5. Real-Time Client-Server Architecture

Finally we analyze the **real-time client-server** architecture. Several real-time applications require the ability to execute remote commands or services and eventually return results, all of this in bounded time. A classical example of operations on this kind of architecture are transactions on *real-time database*. As shown in Figure 11.5, clients issue competitive requests to the server (e.g., dial-time request of the translation from a free-toll number to the actual subscriber number). Both the scheduling of the request for execution and the execution itself should take place in bounded time (e.g., the successful telecommunications database lookup). The result (e.g., the called party number) is returned to the client. Soft real-time applications are readily built on a client-server architecture. The more demanding hard real-time or mission-critical classes may face difficulties with traditional client-server technologies: to the lack of determinism of the arrival pattern of competing client requests, one has to add the potential lack of determinism of most multi-threaded programming approaches on the server side (*see* Section 3.6 on the basic characteristics of the client-server model). As such, real-time client-server architectures (and namely RT databases) are normally designed in a way that constrains these sources of non-determinism.

11.4 SUMMARY AND FURTHER READING

This introductory chapter discussed the fundamental concepts concerning real-time in distributed systems. Concepts and terms were introduced, such as:

real-time system; hard, soft and mission-critical real-time. The main real-time network and distributed system architectures were introduced, to be further debated in the following chapters. Namely, we divided real-time networks into three large type groups: the physical-circuit, digital-bus, and virtual-circuit groups. Finally, we discussed the real-time control, producer-consumer, and client-server real-time architectures. Relevant surveys on research in the area can be found in (IEEE-RT, 1994).

12 PARADIGMS FOR REAL-TIME

This chapter discusses the main paradigms concerning real-time in distributed systems, in the viewpoint of the system architect. Namely, the chapter addresses: specifications for describing timeliness; timing failure detection; the real-time entity-representative relation; the time-value duality of real-time entities; real-time communication; flow control; scheduling; clock synchronization; input-output. We explain these paradigms in practical terms, giving examples of the problems they solve and of their limitations.

12.1 TEMPORAL SPECIFICATIONS

The representation of temporal specifications is of extreme importance in real-time systems. Several things have to be described, specified, or quantified, when dealing with real-time paradigms: the timing of events related with the execution of real-time communication and computation actions (e.g., deadlines); the patterns of arrival of events (e.g., sporadic); the definition of triggering conditions (e.g., time lattices). We assume the reader to be familiar with basic notions about time and synchrony (*see Time and Clocks* and *Synchrony* in Chapter 2).

12.1.1 Timing of Events

Real-time researchers and developers name several timing variables in particular ways that sometimes depend on the context. Before learning these specific

P. Verissimo et al. *Distributed Systems for System Architects*

terms, let us analyze what we need to specify and to measure, *in general terms*, in real-time systems. Real-time systems are in essence *reactive* or *responsive*. Most of what they do is related with responding to events produced by the environment and by human users.

Response Time - the interval that mediates between the occurrence of an input event and the occurrence of the first related output event

For example, the interval between the arrival of a `train-passing` sensor reading and the output of the `close-gates` actuator command. The maximum and the minimum response times are relevant variables of a real-time system. The first, because it measures the capability of handling the dynamics of controlled processes: slow systems cannot control fast changing processes adequately. The second, because together with the first, it measures the variance of the speed of the computer response: quality of control is affected by the latter (Kopetz, 1997).

Real-time systems must respond according to pre-specified timings. The basic thing about timing is being able to specify action *durations* and event *positions* or *timestamps* in the timeline. For example: *within* T_A *from* t_A to mean that an action will be performed with a maximum duration of T_A, starting at t_A. In general terms, let us call them specifications of timed actions.

Timed Action - the execution of some operation, such that its termination event should take place *within* an interval T_A *from* a reference real time instant t_A

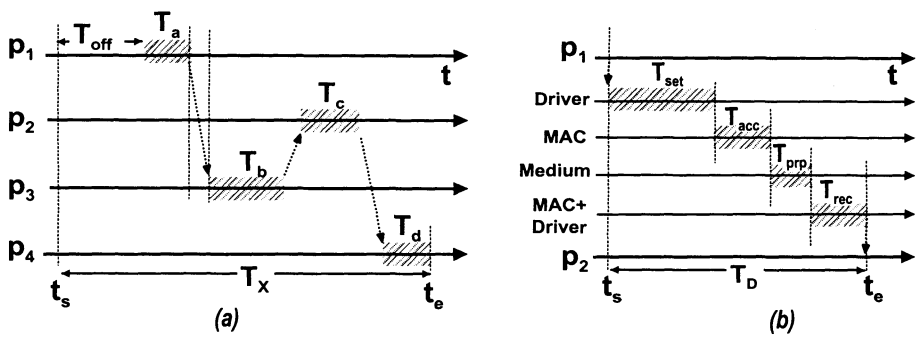

Figure 12.1. Termination Time: (a) Computations; (b) Communication

We start by analyzing the *termination time*, that is, the duration of completion of an action. Figure 12.1a shows the execution of computations. Note that the execution of an action may be complex, as depicted in the diagram of the figure: made of several elements or *tasks* that take place in several sites. Tasks are invoked for execution and run when scheduled. The essential timing specifications are: *deferral time* (T_{off}), the delay introduced before the execution is requested, also called *offset*; *termination time* (T_X), the difference between the termination (or end) and request (or start) event timestamps (resp. t_e and

t_s) in the timeline; *execution time* (T_{ET}), which accounts for the duration of the computation in continuous execution (may be shorter than the termination time), in this case the sum of durations T_a, T_b, T_c, and T_d. Figure 12.1b on the other hand shows the execution of a communication action. The figure depicts the split into the four parts that account for the termination time of a transmission: the *set-up* time (T_{set}) is spent preparing the frame for transmission, from the time it is handed from the user buffer to the operating system; *access* time (T_{acc}) is the time the frame spends waiting to be transmitted; the time spent by the frame in transit is the *propagation* time (T_{prp}); finally, the *reception* time (T_{rec}) is the time spent in transferring the frame to the recipient buffer. The delivery time T_D is the difference $t_e - t_s$ in the timeline, and is computed from the sum of the components described above.

Let us consider now how to position the *instant of completion* of an action on a desired point of the timeline, say t_A. In the case of those actions that consist of nothing else but generating an output event, this may be specified by scheduling the execution of a short-lived, negligible duration action *at* t_A, with the help of a clock. In order to achieve the same for an action that has a non-negligible and constant duration T_A, it must start at $(t_A - T_A)$, which we specify by requiring that the action terminates *within* $-T_A$ *from* t_A.

There are natural timing *errors* in the execution of timed actions. These errors are normally called *jitter* in real-time lingo, and derive from fundamental limitations of the support infrastructure, such as non-determinism of the software, scheduling, faults, clocks, etc.

Jitter - the uncertainty about the instant of completion of a timed action, taking the form of *variance* in the duration of its execution or of *imprecision* in the positioning of its termination event

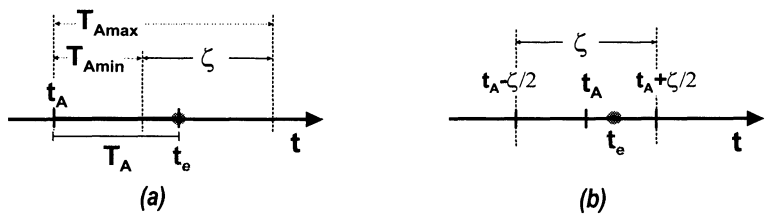

Figure 12.2. Jitter: (a) Variance; (b) Imprecision

Consider the duration specification that an action terminates *exactly after* T_A *from* t_A, that is, assuming a constant delay T_A (e.g. if it takes exactly 2 seconds for a labeling machine head to come down right over an arriving part, the former may be commanded to start coming down 2 seconds before the part arrives, for speed improvement). Assume that the execution of the action has some jitter, which shows up in this case as a *variance* in the termination time (e.g., the head may arrive too late or too soon sometimes, causing incorrect labeling) depicted in Figure 12.2a. The correct specification to recognize this

fact should be *"t_e within ζ from $t_A + T_{Amin}$"*, where ζ is the action jitter, such that $T_{Amax} = T_{Amin} + \zeta$. Then, either the labeling system accommodates the jitter, or another solution must be found.

Consider now the positioning specification of an action completion event e_A in the timeline *"e_A at t_A"*. As exemplified by Figure 12.2b, jitter shows-up as an *imprecision* in the positioning of the event. In fact, in order to be correctly equated, the specification should read *"e_A within $\pm\frac{\zeta}{2}$ from t_A"*, where ζ is the action jitter. Then, either this imprecision is supposedly small enough to be acceptable, or else another solution must be found.

12.1.2 Triggering Timed Actions

There are essentially two ways of triggering timed actions in real-time systems. The **event-triggered** approach makes the system react upon the occurrence of an input event. As Figure 12.3a exemplifies, reaction of the system is immediately triggered by the event arrival, and the subsequent response issued. The system reacts as the input is made, and with the timing given by the speed of response. In the **time-triggered** approach, depicted in Figure 12.3b, the system reacts upon the command of a clock. Regardless of when an input event arrives, it is processed at the next input point dictated by the clock. Likewise, outputs are also synchronized by the clock, as shown in the figure, no matter how fast they are produced. Timed actions of different kinds can be combined. For example, input events served in an event-triggered manner, but outputs synchronized by a clock. Note that in consequence, the output jitter of an event-triggered action is given by the execution time variance, whereas in a time-triggered action it is given by the imprecision of the clock.

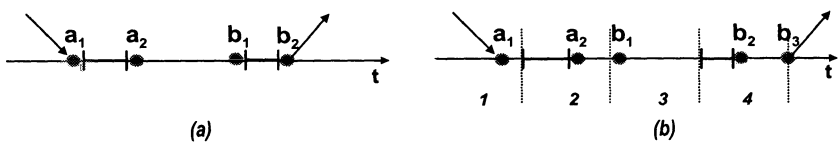

Figure 12.3. Triggering Timed Actions ($a_i \rightarrow a_{i+1}$, $b_i \rightarrow b_{i+1}$): (a) Event-triggered; (b) Time-triggered

Time lattices (*see* Figure 1.8 in Chapter 1) are powerful constructs to synchronize the triggering of simultaneous timed actions, to facilitate the measurement of their duration, and to tell the ordering of distributed events. Controlled by clocks, the same physical tick of the clock at every site marks what we call a *microtick* (Kopetz, 1997), a global tick of the lattice. Recall that events are ordered by the physical granularity (g_p) of the clock, which is often so fine that it orders events that have no causal relation (*see Temporal Order* in Chapter 2). This problem is attenuated if we construct the lattice around a global clock with an artificially coarser virtual granularity (g_v). Time is now marked in *macroticks* spaced by g_v, and all events in a g_v interval between two ticks are considered concurrent.

12.1.3 Arrival Distributions

We said that the evolution of a real-time system is dictated by the environment. This means that the system receives *inputs* from the environment, and then has to react according to what is expected, and when expected. We spent the last sections analyzing how to specify and trigger this response.

But what about the inputs themselves? Can we process an undefined amount of information per time unit? Of course not, systems have a limited processing capability. In consequence, all variables that we have analyzed, such as response time or termination time, depend on the load on the system. Can we predict and/or bound the amount of information that arrives at the system input? Only in some cases. This is why we have defined classes of real-time systems: (a) the determinism of the system with regard to time depends on the predictability of the inputs received from the environment; (b) the predictability (or determinism) of the environment depends on the class of application. Systems handling regular and deterministic arrival patterns are simpler, but have to cope with the potential lack of coverage of those assumptions. Systems accepting irregular and uncertain distributions are closer to physical reality, but are more difficult to design and prove correct.

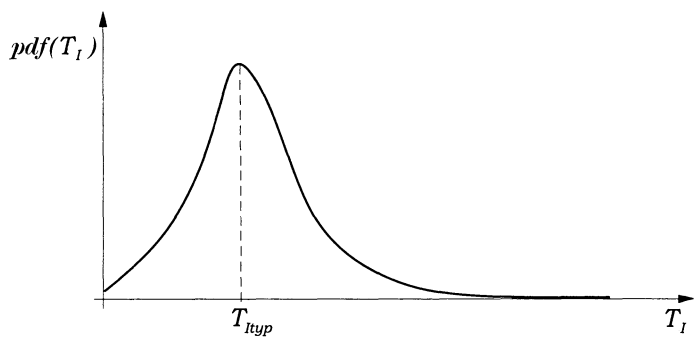

Figure 12.4. Example Probability Density Function of Event Inter-arrival Times T_I

In Figure 12.4 we depict a probability density function that could model the inter-arrival intervals of events produced repetitively by a source. Such a distribution has a zone (in the center) where events are distributed more or less every T_{Ityp}. However, there is no limit to the amount and frequency of information that may arrive to the system on certain occasions, since the distribution intersects the Y axis. We call such distributions **aperiodic**. It is not hard to see that aperiodic distributions are not ideal to achieve real-time behavior because response time is conditioned to the amount of information input per time unit, of which aperiodic distributions offer no guarantees.

On the other hand, if events arrive in a known maximum amount at known points of the timeline, i.e., in a regular fashion, it is easy for the system to behave deterministically, and for response and termination times to be precisely

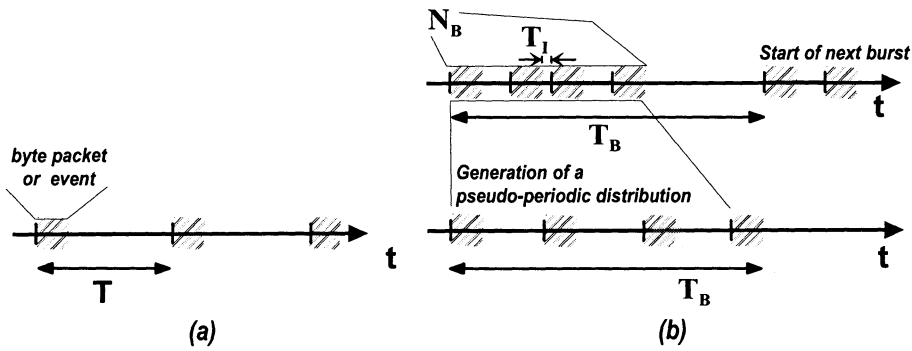

Figure 12.5. Discrete Arrival Distributions: (a) Periodic; (b) Sporadic

determined. In Figure 12.5a we depict such a distribution, with the obvious name of **periodic**, of period T.

Sometimes the designer must achieve deterministic behavior, but she cannot model the environment so regularly, because the latter is rather unpredictable, with irregular arrival patterns, such as event-triggered message deliveries and task execution requests from sensor events. A more sophisticated discrete arrival pattern represents this behavior, the **sporadic** pattern, which assumes that input information arrives irregularly in bursts, but has a few crucial bounds. Figure 12.5b (top) graphically shows the several parameters involved:

- **burst period** (T_B)– minimum delay between the start of two consecutive bursts, a known lower bound
- **burst length** (N_B)– maximum amount of information submitted in one burst (e.g., number of events, number of bytes), a known upper bound
- **inter-arrival time** (T_I)– minimum separation between two consecutive events, a known lower bound

A sporadic arrival distribution is normally treated under a short-term perspective: sporadic requests have to be served within the inter-arrival time (T_I), and an amount up to N_B of such requests must be handled without resource disruption. This is adequate for example for emergency events. However, a sporadic distribution is one that has long-term regularity, and as such it can be mapped onto a periodic distribution. This is adequate for treating sporadic events that are generated by periodic tasks, and also the so-called *event showers*. As a matter of fact, under these conditions N_B/T_B gives the rate of a periodical distribution where we would have spread the event arrivals throughout the period interval, as the bottom of Figure 12.5b suggests.

In conclusion, the real-time system architect should not forget that arrival distributions are mere artifacts to represent the environment behavior. They serve to separate concerns, and prove that given a problem and an event distribution, there are paradigms solving the problem for that distribution. However, the importance of the other part of the job should not be minimized, as it is

sometimes the hardest one: to prove that the environment is faithful to the distribution we chose to model it. Nothing could go more wrong than a wrong set of assumptions (*see Failure Assumptions and Coverage* in Chapter 6).

12.1.4 Utilization Factor

Another way of determining whether *time is enough* is in relative terms.

Utilization Factor - measure of the percentage of useful work time of a resource, over elapsed time

For example, an utilization factor of 70% is often pointed out as a good CPU time-loading figure for microprocessor based control systems (Laplante, 1997). It means that during a period of 100ms, the processor is executing useful instructions during 70ms. Put the other way around, it also measures whether the CPU has enough power to handle a set of tasks: if 3 tasks each with a duration of 40ms on a given CPU have to complete within 100ms, the CPU utilization factor becomes 120%, which means it is *overloaded*.

Communication resources are also measured in terms of channel utilization factor. An utilization factor of 85% on a 10Mb/s Token-bus LAN means that during the period of a second, the channel is letting 8.5 megabits through. The maximum utilization factor (10 megabit per second in this case) is also called maximum *throughput*.

12.2 TIMING FAILURE DETECTION

Note that hard real-time systems are designed in terms of preventing timing failures. However, controlled timing failures are allowed in mission critical or soft real-time systems. In consequence, the measures to be taken by real-time systems in response to timing failures vary according to the class of operation: orderly fail-safe shutdown; recovery or compensation; reconfiguration by adaptation to less stringent deadlines. Whatever the solution, timing failures should be detected. In fact, we are concerned with *late timing failures* (e.g., the missed deadline or late message syndrome) which are the most general type of omissive failure (*see Fault Assumptions and Coverage* in Section 6.1). So, more precisely, a timing failure is:

Timing Failure - Given the execution of a timed action specified to terminate until real time instant t_e, timing failure is the occurrence of the termination event at a real time instant $t_e^{'}$, $t_e < t_e^{'} \leq \infty$. The amount of delay, $Ld = t_e^{'} - t_e$, is the lateness degree

What do we need to know about a timing failure in a real-time system? We need to detect it in bounded time, i.e., in a *timely* manner. We must detect all relevant late actions as timing failures, i.e., in a *complete* manner. We must avoid detecting timely actions as failures, i.e. in an *accurate* manner. Recall that we have already discussed *crash failure* detectors (*see* Section 7.1), having characterized the quality of detection according to two essential properties, completeness and accuracy. We should be able to characterize timing failure

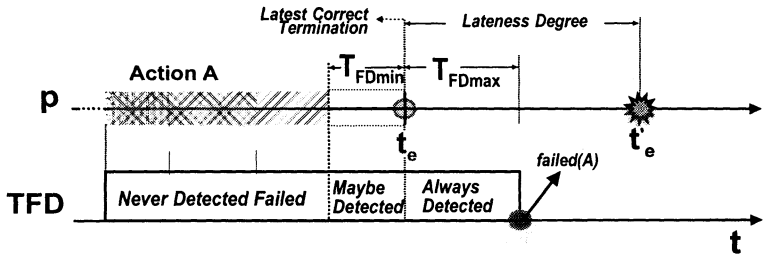

Figure 12.6. Timing Failure Detection in Action

detectors according to a similar reasoning. We must however introduce the time dimension, in order to define both the failure and when it is detected. The following defines the desirable properties of a **timing failure detector** (TFD), assuming that any timed action has an observable termination event e, specified to occur until real time instant t_e:

Timed Strong Completeness - There exists $T_{TFD_{max}}$ such that a timing failure in any timed action is detected within $T_{TFD_{max}}$ from t_e

Timed Strong Accuracy - There exists $T_{TFD_{min}}$ such that any timed action that terminates at p before $t_e - T_{TFD_{min}}$ is considered timely

The properties of the failure detector are illustrated in Figure 12.6. Note that 'timed' in both properties specifies that there is an upper bound ($T_{TFD_{max}}$) on detection latency, and a lower bound ($T_{TFD_{min}}$) on detection accuracy.

12.3 ENTITIES AND REPRESENTATIVES

There is a fundamental paradigm that has to do with the relation between elements of the environment and their computational representation, the *entity-representative* paradigm (Kopetz and Veríssimo, 1993). This relation is more important than meets the eye, and if neglected, the possibility of separation of concerns in the conception of parts of the real-time systems is diminished. That would be a bad architectural principle.

A **real-time entity** (RTe) is an element of the environment, such as a fluid valve or the temperature of an oven, with a behavior which may be time-dependent and a state the system is supposed to acquire or modify, described by continuous or discrete values, e.g.: 150 meter/sec; open/closed. The state of RTe's can be read or written to, but not both. Consider the example of a valve. We should define a write-only RTe `valve_actuator` representing the valve actuator, whose state can be positioned to one of `open|closed`, by actuator-dependent procedures (e.g., commands, control registers, etc.). Then, if we would at the same time wish to monitor the state of the valve, we should define a `valve_sensor` read-only RTe, whose state might take one of the values `open|closed`.

The **representative** (RTr) of a real-time entity is an element of the computational system through which the latter *observes* or *acts* on the state of

the real-time entity in the environment. Representatives offer an interface tractable by the other elements of the computational system, and resort to sensors and actuators as a means of handling the RTe's they represent. They can assume several forms: an integrated intelligent sensor controller; the driver of a stepping motor; a computer process controlling PC I/O boards; an A/D (analog-to-digital) conversion acquisition module.

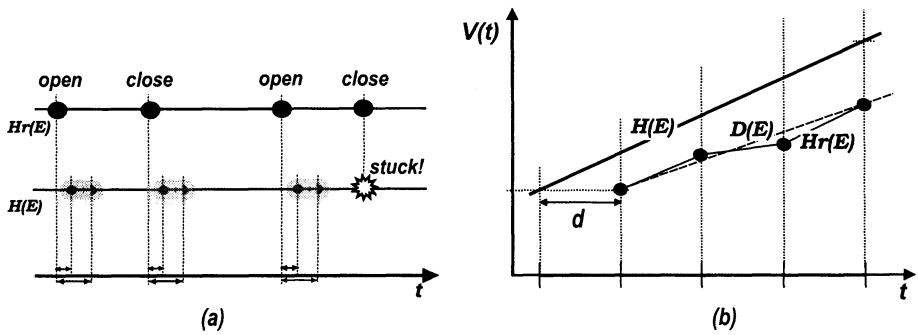

Figure 12.7. Real-Time Entity-Representative Relationship: (a) Discrete Entities; (b) Continuous Entities

Note that the state of a real-time entity is not accurately reflected in its representative at all times during system evolution: the temperature of an oven and its measure at a sensor representative differ with sensor *accuracy* and, more importantly, with *time*; a valve actuator representative may have been instructed to shut, but the valve itself may remain open because of a *failure*. The problem is amplified by *distribution* and *replication*, since one can get divergent readings of the same RTe by different sites. A correct design should address these problems (*see* Chapter 13).

For the sake of understanding the relation of an entity E with its representative $r(E)$, consider the evolution of the state of E with time as a non-observable sequence of states along the timeline. Then, consider the evolution of the state of $r(E)$, composed of a sequence of observations or actuations, depending on whether it is a read-only or a write-only RTe, respectively. The real-time entity-representative relation is represented in Figure 12.7, where the curves of $E(t)$ and $r(E)(t)$ are represented respectively as $\mathcal{H}(E)$ and $\mathcal{H}_r(E)$. In essence, a representative *emulates* its real-time entity with an error in the value of state, or in the time of state changes, or in both.

Figure 12.7a illustrates $\mathcal{H}(E)$ and $\mathcal{H}_r(E)$ for a write-only boolean RTe (e.g. valve). The exemplified window of discrepancy between the RTr inside the computer system ($\mathcal{H}_r(E)$) and the actual RTe ($\mathcal{H}(E)$) has a fixed part, the actuation delay, and a variable part (in gray) because of the jitter of positioning the command. In faulty situations, the discrepancy may be so large that the relation is no longer valid, unless measures to ensure fault tolerance are taken: in the last actuation, the RTr says `open`, whereas the valve got stuck at `closed`. Figure 12.7b illustrates an analog read-only RTe. Against the curve of $\mathcal{H}(E)$,

we depict $\mathcal{H}_r(E)$, which exemplifies the discrepancy between the RTe and its representative. What happens may be explained by three factors: a delay in reading the state of E (execution time); the jitter of that reading (variance in execution time); and the magnitude error of the reading itself (sensor inaccuracy). Dashed line $\mathcal{D}(E)$ shows what would be the situation if only a fixed delay error component d existed.

The entity-representative relation is an important architectural paradigm. Representatives hide the complexity of the physical reality, transforming it into representations tractable by computers. For the real-time computer system, $r(E)$ *is* E. This transformation can be extremely useful, provided that the resulting errors are definable and/or bounded. The paradigm allows a separation of concerns in the design of real-time systems:

- *definition of the real-time entities* – response to the environment part of the requirements specification
- *definition of the representatives* – specification of the computational entities representing the environment;
- *definition of the input/output* – specification of the reliable and timely observation of, and actuation on, the state of RTe's;
- *definition of the control* – specification of the reliable and timely processing of the information supplied by input representatives and production of responses to output representatives.

12.4 TIME-VALUE DUALITY

Consider the specification of an action: "at real time instant T_A, produce a result of value V_A". Upon execution, the action will produce a value v_A at real time t_A, for storage, use, or other. For example, delivering a message, or storing an observation. Producing a late correct value yields a *timing error*. Likewise, producing a timely incorrect value yields a *value error* (*see* Section 6.2).

Now assume that the specification concerns the state of a real-time entity E, *whose value depends on time*, $V = E(t)$. The relevant action specification becomes: "at real time instant T_A, produce a result of value $V_A = E(T_A)$". For example, determining the position of an engine crankshaft. Producing $v_A \neq V_A$ at $t_A = T_A$ yields a *value error* $|v_A - V_A|$, on time (e.g., an erroneous position of the crankshaft at time T_A is returned). Likewise, producing $v_A = V_A$ at $t_A > T_A$ would just yield a *timing error*.

Would it? Recall that by specification the value returned at t_A must be $E(t_A)$. However, what was returned was $E(T_A)$ (e.g., the value of a past position of the crankshaft). So, in this situation, the timing error causes a value error, since the "correct" value returned concerns the value of the entity in the past. If the timing error is $|t_A - T_A|$, the corresponding value error is $|E(t_A) - E(T_A)|$.

Essentially, a **time-value entity** is such that there are actions on it whose time-domain and value-domain correctness are inter-dependent. This paradigm consolidates notions addressed in the context of a number of problems in different areas of computing, such as computer control, I/O sensing, real-time

databases, or clock synchronization (Kopetz, 1997; Poledna, 1995; Marzullo, 1990; Ramamritham, 1995; Lamport and Melliar-Smith, 1985). There are two problems to solve if correct operation of a system using time-value entities is sought:

- ensuring the correct observation of both an instantaneous value of the entity and its positioning in the timeline, that is, the *time of a value*, which assumes two facets:
  observing the value at a given time – e.g., the angle 50μsecs past the lower position of the crankshaft, a switch position at 5:00
  observing the time at which a given value occurs – e.g., when is the crankshaft at 5 degrees to the top position, whenever a switch closes
- ensuring the correct use of such an observation past the time it is made, that is, the *value over time*

The first problem is equated with the error in observing the time of a value, i.e., the observation error. For an observation $\langle r(E_i)(t_i), T_i\rangle$ of the value of an RTe E_i at t_i receiving timestamp T_i, the **observation error** in the *value domain* is given by $\nu_i = |(E_i(T_i) - E_i(t_i)) + (E_i(t_i) - r(E_i)(t_i))|$. The first term in parentheses is the effect of the timing error in positioning the observation, supposedly at T_i but in fact made at t_i. The second is caused by the error of the observation apparatus (sensor module). Simplifying, we get an intuitive $\nu_i = |E_i(T_i) - r(E_i)(t_i)|$: we expect the value of E_i at T_i, but we get an approximation of the value $(r(E_i))$, measured approximately (t_i) at T_i. For observing the time at which a given value occurs, and in fact the sensible way to observe discrete entities (i.e. ones whose value jumps abruptly, say from logical one to zero), we should use a time domain metrics. The observation error in the *time domain* would then be $\zeta_i = |T_i - t_i|$: E_i assumed a given value at t_i, but the system logs it as having happened at T_i. The error accounts for the positioning error (jitter), and the time-domain effect of the sensor value error. Since determining the exact error of each observation is not feasible, we may instead work with an upper bound:

- *Given a known $\mathcal{V}_o$, we say that an observation $\langle r(E_i)(t_i), T_i\rangle$ is* **consistent** *in the* **value** *domain, if and only if $\nu_i \leq \mathcal{V}_o$*
- *Likewise, given a known $\mathcal{T}_o$, we say that the observation is* **consistent** *in the* **time** *domain, if and only if $\zeta_i \leq \mathcal{T}_o$*

The problem can be generalized to the values of a set of RTe's *at a given instant*, e.g., needed for performing computations to derive a composite variable. If the relevant observations all have bounded errors referred to that instant, the total error of the computation is bounded:

- *Given a known $\mathcal{V}_m$ and a set of observations of RTe's, we say they are* **mutually consistent** *in the value domain, if and only if there is an instant t_m such that each observation $\langle r(E_i)(t_m), T_i\rangle$ is consistent w.r.t. $\mathcal{V}_m$*

It is a sufficient condition for a set of observations to be mutually consistent, that they are consistent, and that the timestamps of all observations fall within a known interval $\mathcal{T}_m$ (i.e., $\forall i, j\ |T_i - T_j| \leq \mathcal{T}_m$). The latter is also called the *relative validity interval* in the context of databases (Ramamritham, 1995; Song

and Liu, 1992), and is also related with the notion of *accuracy interval* in the context of replicated sensor observations (Marzullo, 1990).

The second problem we stated is concerned with using a value while it is still valid. In order to solve this problem, we must bound the response time or the termination time of the action that follows the observation (unexpected or programmed) of a time-value entity. The action can be performed later than the observation time, but not too late or unpredictably later, because this may imply performing an incorrect action. Assume bound $\mathcal{V}_a$ for the maximum acceptable error accumulated by an observation over time, depending on the application in view and on the dynamics of the observed time-value entity (in order to separate concerns, we neglect the observation error and define T_i as the instant of reference):

- *Given a known $\mathcal{V}_a$, we say that an observation $\langle r(E_i), T_i \rangle$ is* **temporally consistent at** $t_a \geq T_i$ *if and only if* $|E_i(t_a) - E_i(T_i)| \leq \mathcal{V}_a$

In complement to the "instantaneous" consistency property of the observation instant, this property captures the evolution of consistency with time, a characteristic of time-value entities. In fact, temporal consistency can be secured if an interval $\mathcal{T}_a$ can be defined such that the variation of the value of the RTe within that interval is at most $\mathcal{V}_a$. In other words, an observation is temporally consistent *within $\mathcal{T}_a$ from T_i*. This interval is also called *absolute validity interval* for databases (Ramamritham, 1995; Song and Liu, 1992), or *temporal accuracy interval* for control (Kopetz, 1997).

The time-value paradigm is at the heart of practically all real-time designs. It gives a common explanation to phenomena that have been addressed separately, such as temporal constraints in R/T databases, dynamics of computer control, and even clocks. In fact, clocks are interesting time-value entities: the value of a clock, $c(t)$, is a value established at a given time, which represents time itself. As an exercise, the reader may wish to find out the analogies between the properties of time-value entities and those of clock systems.

12.5 REAL-TIME COMMUNICATION

Real-time communication is related with achieving a few fundamental attributes:

- known and bounded message delivery delay
- deterministic time-domain behavior
- recognition of urgency classes in the overall traffic
- reliability of medium connectivity

Depending on the type of architecture, these are achieved in different ways, but whatever the techniques, the real-time communication paradigm can be expressed in generic terms:

Real-Time Communication - the achievement of bounded and known message delivery delays, in the presence of disturbing factors such as other real-time traffic, variable load, or faults

This generic definition suggests a few things. Firstly, that it is necessary to transmit frames (network-level information packets) in bounded time, given

the interference of other traffic. Some of it will be competing for the channel on an equal foot, but clearly not all frames have the same importance, or *urgency* class, a parameter that allows some frames to get through ahead of others. Practical communication systems hardware (e.g., LANs) provides this distinction through priorities. Besides, channel scheduling should encompass the fact that load may not be steady, which means that the *latency* bounds should be achieved under variable throughput, known to be a difficult goal.

Another issue suggested by the definition is that it is necessary to get the message (user-level information packet) through despite faults. Real-time touches reliability in this point. It is of little use guaranteeing schedulability of individual frames on the network without thinking what is the final time budget to get a message across. This encompasses the use of mechanisms studied in Chapter 7, to detect, recover or mask transmission errors.

The discussion above suggests one of possible strategies for realizing the real-communication paradigm, a divide-and-conquer strategy breaking down a solution in a number of conditions to be fulfilled:

1. **computing the load budget and defining urgency classes**– defining urgency classes and allocating worst-case load patterns to each

2. **ensuring connectivity**– providing the adequate measures to ensure reliability of the network medium, and control partitioning

3. **preventing timing faults**– enforcing a bounded time from request to actual transmission of a single frame, given the worst-case load conditions assumed, in absence of faults

4. **tolerating omission faults**– ensuring that a message is delivered despite the occurrence of omissions

5. **controlling the flow of information**– ensuring that the offered load is such that the desired throughput and latency are secured

The conditions presented above are sufficient to achieve the real-time communication requirement. Condition 1 makes the basic assumptions about the environment. Condition 2 encompasses the initial discussion about how to achieve medium reliability. Both conditions have to meet the requirements of the application in mind. Condition 3 makes sure that any frame is sent within a known time bound, even if it does not arrive. Condition 4 ensures that a message is delivered in the presence of omission faults. A time bound as per Condition 4 is calculated as a function of the assumed maximum number of omissions during the protocol execution, the use or not of space redundancy, as per Condition 2, and the time bound on individual transmissions as per Condition 3, if time redundancy is used. Condition 5 addresses the need for matching the actual load presented by the environment, assumed initially by Condition 1, with the capabilities of the system. A real-time communication model implementing this strategy in presented in *Real-Time Communication Models* in Chapter 13.

12.6 FLOW CONTROL

Flow control is a fundamental paradigm of distributed real-time. In fact, given that bandwidth and computational power are finite, it is impossible to guarantee timeliness if the load flow on the system is not controlled. The role of *real-time flow control* is to regulate the global load flow of the system and to throttle the instantaneous flow of individual source classes in terms of their arrival pattern (rate and amount of data), so as to preserve the capacity of the system to exhibit bounded response times.

The control of a periodic flow is a simple task. A sporadic flow, such as the one produced by sensors of discrete real-time entities, is harder to treat. In Section 12.1 we have discussed the possibility of a bursty sporadic arrival pattern being smoothened over a longer interval (*see* Figure 12.5b), as long as this is allowed by the service latency requirements of the arriving requests, and the interval is not greater than the burst period, if one exists. The mechanism that makes this possible is called rate-based flow control, or *rate control*, and helps balance system load without glitches. *Credit control* is a load control scheme based on allocating a certain amount of credit units (for example octets) to sinks (e.g. recipients), per flow of information coming from a source (e.g. sender or a group of senders). When the credit is over, the recipient refuses to accept more information. Credit complements rate control in case of sporadic real-time operation, whenever it is necessary to perform resource reservation for significant amounts of information of bursty nature.

Flow control is of little use if the average load is misadjusted in the first place. As such, there are measures which can be considered of implicit flow control, such as compacting information at the sensor representatives before sending it to the core of the system, or eliminating redundant messages corresponding to a same event (e.g. fire alarm), that otherwise generate event-message showers.

12.7 SCHEDULING

Real-time is not about having a lot of bandwidth and computational power, and even if it were, we would always find ways of exhausting it. Alternatively, we may think we solve the problem by letting the critical task always get the processor when needed. But the processor power may have to be shared by several critical tasks and the problem surfaces again. Even if we think we have enough power to guarantee the timely execution of our critical task(s), chances are we are using it up at the wrong moment. Remember La Fontaine's Fable of the Hare and the Turtle? Figure 12.8 illustrates this famous tale adapted to real-time scheduling[1].

System TURTLE is a slow system, with relative speed s=1, context switch delay c=1, scheduling first the task that must finish earlier (this is called earliest deadline first, and is a sensible real-time scheduling policy that we are going to study). System HARE is a very fast system, with relative speed s=10, almost

[1] We owe this example to Gerard LeLann.

negligible context switch delay c=0.01, scheduling first the task that comes first (this is called first-come-first-served, and is a not so sensible scheduling policy for real-time). The work presented to either system at time t is: two task execution requests arrive at approximately the same time t, task A before task B; task A, execution time Xa=270 relative time units, deadline Da=t+290; task B, execution time Xb=15, deadline Db=t+28.

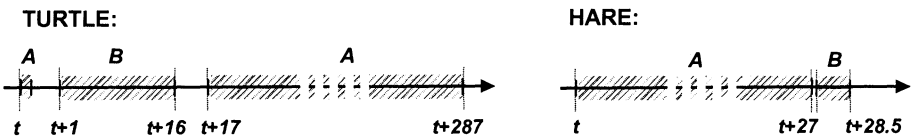

Figure 12.8. The Hare and the Turtle Schedulers

Let us analyze how system TURTLE serves the job. Observing the figure, we see that it starts by releasing A, then switches right after to B, because it has the earliest deadline, B executes during 15 time units, terminating before the deadline. Then, it switches to A again, finishing after 270 time units, before the deadline.

What happens when system HARE serves the job? We see that it starts serving A immediately it arrives, and A will run to completion, according to the first-come-first-served policy. The rationale behind the HARE approach is that the system is so fast that it serves any request "quickly" enough to get ready for the next. However, A executes in 27 time units (speed of 10), the context switches in almost negligible time to B, B executes in 1.5 time units, and ... it misses the deadline by half a time unit!

The lesson to be learned is that real-time is about determinism and guarantees, rather than speed, and in this context the **scheduling** paradigm is concerned with using the available resources in the right way in order to help the system (its programs, its algorithms) achieve timeliness guarantees.

12.7.1 Types of Scheduling

Scheduling assumes several facets, according to the objectives of the system, and to the problems posed, such as the load assumptions. Most non real-time scheduling policies aim at *fairness* and reasonable *performance*, in the access to resources by the several users. Real-time systems, on the other hand, aim at fulfilling timeliness requirements, if need be in detriment of the performance of less important tasks. Let us define some terminology before advancing further. Table 12.1 introduces some generic parameters specifying instants and intervals (that is, events and durations) related with task execution timing. Figure 12.9 depicts a task execution.

The main classes of real-time tasks are: aperiodic, periodic, and sporadic, named according to their main pattern of execution request arrivals (*see Arrival Distributions* in Section 12.1). Aperiodic tasks are impossible to treat deterministically, the best that can be done is service on a best-effort man-

Table 12.1. Task Execution Timing Parameters

Not.	Designation	Description
T_{trg}	trigger instant	arrival instant of event causing the execution
T_{off}	deferral time	delay introduced before execution request **(offset)**
T_{req}	request instant	instant of execution request **(release)**
T_{Rmin}	min. inter-req. time	minimum interval between any two consecutive requests (equals request period T_R, for periodic tasks)
T_{Xmin}	min. termin. time	minimum elapsed time from request to termin. event
T_{Xmax}	max. termin. time	worst-case elapsed time from request to termin. event
T_{WCET}	worst-case exec. time	maximum task duration in continuous execution
T_{lax}	laxity	slack time available for execution ($T_{Xmax} - T_{WCET}$)
T_{live}	earliest term. instant	earliest that task may complete, also called **liveline**
T_{trgt}	typ. termin. instant	*desired* instant of completion **(targetline)**
T_{dead}	latest term. instant	latest that task may complete, also called **deadline**
T_{int}	max. interfer. time	max. time task can be suspended by higher pri. tasks
T_{blk}	max. blocking time	max. time task can be blocked by lower pri. tasks
P	priority	importance of task w.r.t timing (highest is often 0)
U	max. utilization factor	max. percent. of CPU utilization (T_{WCET}/T_{Xmax})

ner. Periodic tasks are the workhorse of static schedule design. Sporadic tasks serve applications that do not have a regular behavior (request arrival is not periodic).

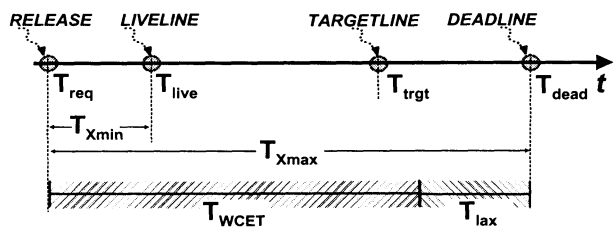

Figure 12.9. Task Execution Timings

We say scheduling is *static*, if all the scheduling plans or scheduling conditions are elaborated beforehand. Static scheduling is also called *off-line* scheduling, as it assumes predicting timing variables such as execution times, request times, resource conflicts, and so forth, and/or assigning static levels of importance to tasks (e.g., fixed priorities). On the other hand, *dynamic* scheduling, also called *on-line*, computes the schedule at run-time, based on the analysis of a list of tasks ready for execution.

Fixed-priority scheduling was very common in earlier real-time operating systems, and is still widely used. With *dynamic-priority* scheduling, priority may change during execution, to reflect the varying importance of tasks, e.g. as deadlines approach.

When the scheduler can interrupt the execution of a task in order to schedule a new one, normally of higher priority, we say scheduling is *preemptive*.

Otherwise, if tasks once scheduled run to completion, we say the scheduler is *non-preemptive*.

Centralized scheduling is performed at a central point, which is normal in single-processor systems. In *distributed* systems scheduling , this point would also become a single point of failure. In consequence, *decentralized* scheduling is preferred for distributed systems.

12.7.2 Schedulability

The usual scheduling scenario is that there is a set of N tasks with several deadlines, to be executed in the processor, over a maximum interval T_{Xmax}, corresponding to the latest deadline. In a periodic task set, the interval corresponds to the least common multiple of the periods. This situation is presented to the designer or design tool at design time for off-line schedulers, or to the scheduler itself for on-line or dynamic schedulers. One has to determine if the schedule is *feasible*. This action is termed **schedulability testing** and is an important step of scheduling. There are three classes of schedulability tests:

- **sufficient**– passing it indicates that the task set is schedulable
- **necessary**– failing it indicates that the task set is not schedulable
- **exact**– passing indicates schedulability; failing indicates non-schedulability

We say a scheduler is *optimal*, when it always finds a feasible schedule if one exists. Sufficient or necessary schedulability tests have an error margin but are simpler than exact ones, and sometimes the only reasonable solution. Obvious such tests are checking that $T_{Xmax} - T_{WCET} \geq 0$, or $T_{Rmin} - T_{Xmax} \geq 0$.

The simplest tests are *utilization-based* schedulability tests, which fail if the schedule will be using the CPU more than a certain percentage. Consider a set of N periodic tasks, each with an individual utilization factor of T_{WCET}/T_R: the schedulability test expression for this set is $\sum_{i=1}^{N}(T_{WCET}/T_R) \leq U_{max}$, where U_{max} depends on the algorithm being used (ultimately, the CPU cannot be used more than 100%!).

Utilization-based schedulability tests are go-no-go sufficient tests. An alternative approach is the *response-time-based* test, explained as follows. The individual WCET of each task is computed. Then, it is used to derive the actual worst-case termination (or response) time, by adding to the WCET the total time the task must yield to all other tasks (e.g. higher priority ones) on account of the scheduling policy, i.e., the *interference* time (T_{int}). The process is repeated for each task. The schedulability test finalizes by simply comparing the computed with the desired maximum termination times. The response-time-based analysis has the advantage of being an exact test, and of giving a quantitative output.

12.7.3 Static Scheduling

Static or off-line scheduling has to take into account worst-case execution times, and urgency, resources, causal precedence, synchronization and deadline requirements of all tasks involved. The schedule is computed by determining

the exact points in time where each task or code module should be launched to meet its deadline. Schedulability testing here means: can the schedule be constructed? Once constructed, the schedule is executed repeatedly, i.e., in a periodic way.

With static schedules, a way of improving schedulability is by using *mode changes*. A mode is a well-contained phase of the system operation (e.g., take-off, cruise, landing in a plane) such that only the necessary tasks, resources and respective requirements are considered for scheduling.

A widely known algorithm for static scheduling of independent periodic tasks in a single processor is **rate-monotonic** (Liu and Layland, 1973). It is a pre-emptive algorithm based on fixed or static priorities, with the following additional characteristics:

- *optimal when maximum termination time equals period ($T_{Xmax} = T_R$)*
- *all worst-case execution times (T_{WCET}) are known*
- *priorities are assigned in inverse order of the period*

The scheduler, exemplified in Figure 12.10, wakes-up at every start of period, and schedules the highest priority ready task, preempting the running task if necessary. In the example, we have: task T_1, period $T_R = 1$, duration $T_{WCET} = 0.5$; task T_2, $T_R = 5$, $T_{WCET} = 1$; and task T_3, $T_R = 10$, $T_{WCET} = 3$. When all periods are multiples of the smallest, $U_{max} = 100\%$, which is the case of the example, so the schedulability test goes: $\sum_{i=1}^{3}(T_{WCET}/T_R) = 0.5/1 + 1/5 + 3/10 = 1$, which means it is OK.

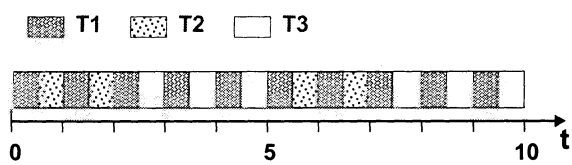

Figure 12.10. Rate Monotonic Scheduler in Action

12.7.4 Scheduling of Sporadic Tasks

Testing schedulability is easy for periodic tasks, since the arrival pattern of the future (period) is known. Attaining and analyzing the schedulability of task sets where periodic tasks coexist with sporadic tasks is a difficult "task" per se, because: (a) we may no longer make decisions completely off-line; (b) it may be hard problem, even when done on-line. Of course, we may consider that a sporadic task is a *pseudo-periodic* task whose period is the minimum inter-arrival time ($T_B \gg T_I$). This has the consequence that most of the service periods are empty, leading to a very low processor utilization.

Another approach is the *sporadic server*, for patterns where $T_{Xmax} \ll T_I$, that is, single, rare, but very urgent sporadics (i.e., burst length is $N_B = 1$, and $T_B \simeq T_I$). The server task is a periodic task with high priority, scheduled

in competition with all the other tasks of the system. When it runs, it serves any pending sporadic request until exhausting its allocated execution time.

The *dynamic scheduling* approach applies well to task sets containing sporadic tasks. There is an algorithm based on dynamic priorities which is adequate for sporadics, and optimal for periodic tasks. It is called **earliest-deadline-first** (EDF). When the scheduler wakes-up it evaluates the time-to-deadline of every task, and orders their priorities by the inverse of that value: the task that has the earliest deadline receives the highest priority. This algorithm is very elegant and intuitive, and can achieve a maximum utilization of $U_{max} = 100\%$.

Dynamic deadline-oriented algorithms, such as EDF, are very adequate for scheduling of sporadics, since they adapt the priorities of the task set to newly requested sporadic tasks.

12.7.5 Resource Conflicts and Priority Inversion

If two or more tasks compete for resources other than the processor itself, such as a mutual exclusion semaphore or critical section, they become interdependent. This happens frequently in distributed systems, so we give a bit of attention to this problem. In most of these cases the exact schedulability test becomes an NP-complete problem, that is, it exhibits a computationally infeasible complexity. However, the computational power×time concerned with finding a schedule should be much lower than the power×time required to run the tasks themselves. In consequence, on-line scheduling resorts to simpler but inexact tests that sometimes have consequences.

For example, the scheduling of periodic tasks with or without sporadics may suffer what is known as *priority inversion*: a task loses the processor during a non-negligible and non-desirable amount of time, called *blocking time* (T_{blk}), blocked on a resource held by lower priority tasks. Consider the following example of a communications and telemetry system:

- A scheduler provides preemptive scheduling based on fixed priorities. The system has three tasks. Two of them compete for an information channel, a critical resource accessed in mutual exclusion (mutex semaphor).
- The highest priority task is an information dispatcher task, which takes care of dispatching all data to and from the channel, so that it does not overrun.
- The lowest priority task is a meteorological data gathering task, running infrequently with low priority, to publish data on the channel.
- The middle priority task is a communications task, handling system's communications. It does not compete for the information channel.

Now the "scene of the crime", depicted in Figure 12.11a:

- *1. The meteo task (L) acquires the mutex and publishes its information.*
- *2. The dispatcher task (H) runs: H preempts L, tries to acquire the mutex and blocks on it, awaiting for the meteo task to release it. L runs again.*

- *3. However, in the meantime, the communications task (M) requests service: M preempts L, and runs to completion.*
- *Conclusion: H was blocked, first by L then by M through L.*

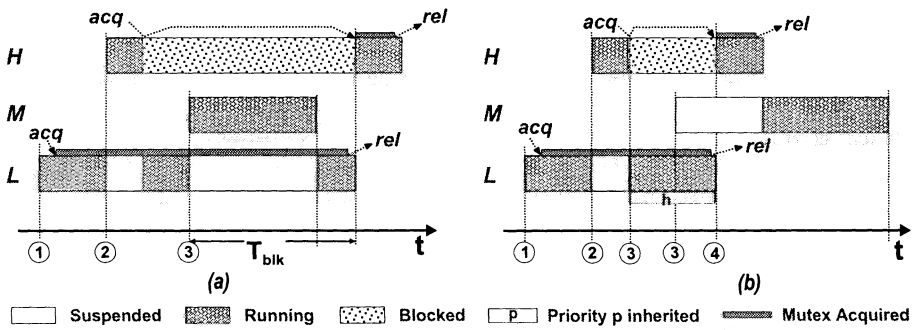

Figure 12.11. Resource Conflicts: (a) Priority Inversion; (b) Priority Inheritance

Is this example realistic? You should have replied yes, because this is what happened with the on-board computer of the NASA Mars Pathfinder probe that landed on that planet in 1997. Shortly after the Sojourner Rover— the small autonomous guided vehicle (AGV) released from the probe— started collecting data, the spacecraft began experiencing total system resets. These were caused by a watchdog mechanism that reacted to absence of activity from the (blocked) dispatcher task, and the system was re-initialized. Fortunately, this recovery strategy was acceptable at that point of the mission, otherwise, it could have been a catastrophe.

What happened was a classical case of priority inversion, a syndrome identified long ago (Lauer and Satterwaite, 1979), which also occurs with networking (Peden and Weaver, 1988). Solutions for it were first addressed in (Cornhill et al., 1987; Sha et al., 1990). In order to remedy the problem, the authors proposed a mechanism which introduces dynamic priority setting to a set of tasks with initial fixed priorities, called **priority inheritance**:

- *the dynamic priority of a process is the maximum of its initial priority and the priorities of any process blocked on account of it*

With priority inheritance, the Mars Pathfinder scene would be scheduled as in Figure 12.11b:

- *1. The meteo task (L) runs with priority l, acquires the mutex and publishes*
- *2. The dispatcher task (H) runs with priority h: H preempts L, tries to acquire the mutex and blocks on it, awaiting for the meteo task L, which runs again inheriting H's priority, h.*
- *3. The communications task (M), with priority m, becomes ready for execution: M waits for L, since h (current pri. of L) is higher than m.*
- *4. L finishes and releases the mutex unblocking H, which grabs the processor ($h > m$), acquires the mutex, and runs to completion. Then, M runs.*
- *Conclusion: H had the minimum blocking possible: waiting for L to finish.*

12.7.6 Scheduling in Distributed Systems

Some single processor algorithms behave well in distributed systems, but others are inadequate or have to be enhanced for distributed operation. Processors are separated by communication links, which themselves are shared and thus have to be scheduled as well. It is difficult to perform scheduling of these distributed resources. Some approaches have relied on heuristics for the cooperation of the distributed system nodes in finding a schedule (Ramamritham et al., 1989). Holistic approaches to the problem have also been considered, where a global scheduling complying with system-wide constraints is constructed from the timing analysis of each module. This has been attempted namely in the embedded systems area, relying on simplifying assumptions on the communications infrastructure, and assuming a periodic behavior of the system (Tindell et al., 1995; Kopetz et al., 1989a). On the other hand, the best known examples of distributed scheduling are the local area network medium access control algorithms, e.g., FDDI, Token Bus, Token Ring, CAN. Some of them exhibit real-time operation, such as the timed-token protocol used in the Token Bus and FDDI networks, or the priority scheduling based on the frame identifier of CAN.

Scheduling in distributed systems is still a subject of research. However, systems are built every day, and apparently, two main approaches can be taken to scheduling in distributed systems with the current state-of-the-art:

- constructing distributed systems whose hardware works in a lock-step fashion, synchronized with the network subsystem also in lock-step (e.g., TDMA) or bit-synchronized (e.g., CAN), and scheduled in a globally periodic manner (e.g., time-triggered cyclic schedules running over TDMA or CAN-like channels). These systems are normally small-scale hard real-time.
- constructing distributed systems whose hardware choices are dictated by the availability of existing COTS (commercial off-the-shelf components), and whose scale and dynamics are dictated by the problem being solved. These systems have better be designed such that tighter (hard real-time) schedules are ensured inside each microscopic component (e.g., nodes, network), and that the cooperative scheduling of the macroscopic system ensures the prosecution of the system's timeliness requirements with the best possible coverage. These systems are normally medium-scale mission-critical real-time.

12.8 CLOCK SYNCHRONIZATION

Observe Figure 12.12a: in the center (dashed line), it depicts a perfect clock, one that always represents real time. However, it also depicts real hardware clocks that are not perfect, i.e., they deviate from real time by a certain amount each second, called rate of drift. We studied the properties of global clocks made of local hardware clocks (*see Time and Clocks* in Chapter 2) and saw that if nothing is done, individual clocks will drift apart with the passing of time. The reader is referred to that section for all basic definitions concerning clocks.

Observe how clocks deviate from the perfect clock in Figure 12.12a: the outside thick dashed lines represent the bound on the rate of drift, a fundamental assumption for deterministic clock synchronization, since it allows us to predict the maximum deviation after a given interval. The amount of deviation between any clock and the perfect clock (which follows real time) at a given time, e.g. at tick t_k, is given by drawing a horizontal straight line passing through t_k in the clock time axis: the length of the line segment connecting the two clock timelines, measured in the real time axis, is the deviation. The current *accuracy* of the clock set is given by the maximum such deviation. As we see, this difference increases as time passes. If the desired accuracy of the clock set is α, no clock may drift to the outside of the grey band, of width α to either side of the perfect clock timeline, as depicted in Figure 12.12a. However, we see that the slowest clock (C_s) will leave the band from tick t_k on (point A), so it should have been re-synchronized before that.

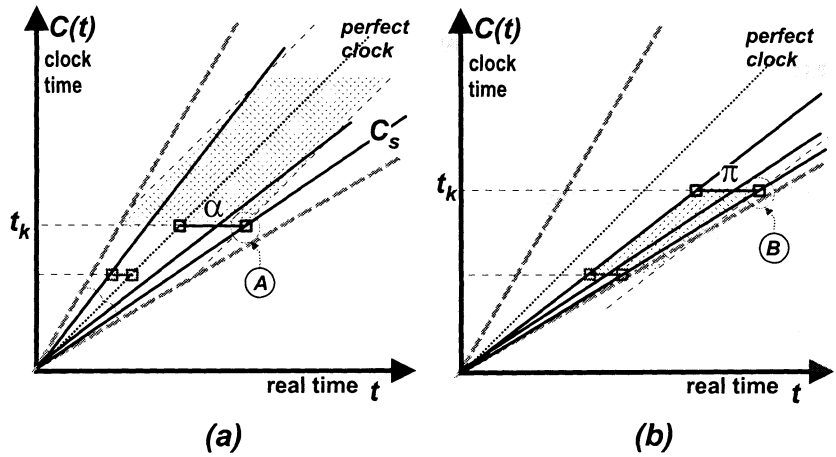

Figure 12.12. Behavior of a Clock with Time: (a) Accuracy Drift; (b) Precision Drift

Figure 12.12b represents the relative deviation among a set of clocks. The amount of relative deviation between the same tick t_k at any two clocks is found by drawing a horizontal straight line passing through t_k in the clock time axis: the length of the line segment connecting the clock timelines, measured in the real time axis, is the deviation. The current *precision* of the clock set is given by the deviation between the two outmost clocks. This difference increases as time passes as well. However, note that their absolute deviation from real time is irrelevant for determining precision: in the example, the set of clocks deviated considerably from the perfect clock, nevertheless they kept within the desired precision π until tick t_k, where the slowest clock got out of the π envelope (point B), as depicted in Figure 12.12b. Obviously, they should have been re-synchronized before that.

The process of maintaining the properties of Precision, Rate, Envelope Rate and Accuracy of a clock set is called *clock (re)synchronization*. Precision is se-

cured by **internal** synchronization, whereas **external** synchronization secures both accuracy and precision, since by securing accuracy, it guarantees that precision remains within $\pi = 2\alpha$. Internal clock synchronization is normally based on convergence functions, whereby the several clocks attempt to converge to a same value at the end of the re-synchronization run. External synchronization is normally based on having all local clocks periodically read from and adjust to one or more clocks containing an absolute time reference. Figure 12.13 exemplifies the latter: clocks are brought together and never deviate more than α from the perfect clock. With either internal or external clock synchronization, it often important to maintain the clock set within the envelope of drift from real time (the thick dashed lines in Figure 12.13). This has to do with securing the Envelope Rate property. The Rate property will be discussed ahead.

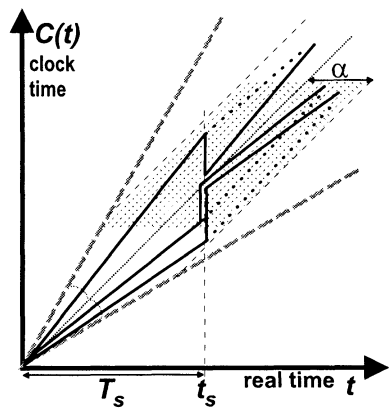

Figure 12.13. Clock Synchronization

Clock-Reading Error The precision achieved immediately after the synchronization, which we have called *Convergence*, δ_v, is a very important property of clock synchronization algorithms, since the quality of an algorithm is measured, amongst other things, by how close it brings the clocks together. Unfortunately, δ_v cannot be made arbitrarily small, and this was defined by a fundamental result in clock synchronization (Lundelius and Lynch, 1984b):

> **Basic Clock Imprecision -** Given n clock processors on a network with maximum and minimum message delivery delays T_{Dmax} and T_{Dmin}, the convergence of any synchronization function is $\delta_v \geq (T_{Dmax} - T_{Dmin})(1 - 1/n)$

The number of processors n is normally large enough that a good working bound for δ_v is $\Delta\Gamma = T_{Dmax} - T_{Dmin}$. The intuition behind this result is that in order to synchronize their clocks, processes need to exchange messages. Unfortunately, the variance in message-passing delays introduces a remote **clock-reading error**, that is, we never know whether the clock value we have just received concerns $t_{now} - T_{Dmax}$ or $t_{now} - T_{Dmin}$.

A second order effect on the quality of synchronization that we neglected in the expressions above is that after reading and while the synchronization is in course, clocks continue to drift apart at a rate ρ_p. If the synchronization algorithm duration is Γ_s, then the additional error in precision caused by this phenomenon is $2\rho_p\Gamma_s$ (observed between by the fastest and slowest hardware clocks). This can normally be neglected. However, we advise the architect to always double check this term before deciding to do it.

Reducing the Clock Reading Error We cannot contradict the Basic Clock Imprecision result, but we can create conditions for achieving a reduced variance in communication delays *during* the critical steps of the execution of the algorithm. In essence, trying to get to a $\Delta\Gamma' = T'_{Dmax} - T'_{Dmin} \ll \Delta\Gamma$, such that $\delta_v \geq \Delta\Gamma'$ (instead of $\delta_v \geq \Delta\Gamma$). To ways of achieving this rely on: trying synchronization enough times until obtaining a run where the variance of the messages involved is small (Cristian, 1989; Mills, 1991); canceling the message delay terms that exhibit greater variance, either algorithmically (Halpern and Suzuki, 1991; Drummond and Babaoğlu, 1993; Veríssimo and Rodrigues, 1992), or through special hardware support (Kopetz and Ochsenreiter, 1987).

12.8.1 Clock Synchronization in Action

A clock synchronization algorithm has the following tasks:

- generating a periodic resynchronization event
- providing each correct process with a value to adjust the virtual clocks

The time interval between successive synchronizations is called the **resynchronization interval**, denoted T_s. At the end of synchronization, clocks are adjusted so that they become separated by at most δ_v. For the sake of convenience, the clock adjustment is usually modeled by the start of a new virtual clock upon each resynchronization event. It can be applied instantaneously, see the instant t_s in Figure 12.13, where the clocks are brought suddenly nearer, and then start drifting again in the next interval. However, this neatly violates the Rate property. One clock is even brought backwards, which is unacceptable. So the alternative is to spread the adjustment over a time interval, by simulating a clock with a slightly different rate, as exemplified by the dotted timelines of the clocks after t_s: the fast clocks become slower, the slow clocks become faster, so that they converge. This is called **amortization** and it is the way synchronization is usually applied: instead of changing the clock, a rate correction factor is applied in software when it is read. For a desired precision π, the value of the resynchronization interval T_s can be extracted from the expression $\pi = \delta_v + 2\rho_p T_s$. Adjusting clocks by changing their values is called state synchronization. It can be combined with another method, *rate synchronization*, which consists of adjusting the rate at which the hardware clock ticks, and even adjusting the instant (phase) of the ticks. These methods are employed when ultra accurate synchronization is desired.

Is clock synchronization a difficult task? The answer is yes, but. Designing clock synchronization algorithms in the presence of communication delay variance and faults is a complex task. However, once deployed, the mission of the architect is selecting the adequate protocol, and simply using time services supported by those algorithms. However, she should understand the limitations of the use of time and timestamps that have been discussed earlier.

12.8.2 Internal Synchronization

Internal clock synchronization algorithms are normally cooperative, where each process reads the values of every other process, and applies a *convergence function* to the set of remote readings. At the end, there is agreement on the adjustment. In what follows, we will use 'clock' and 'processor' interchangeably. Known internal clock synchronization algorithms are of the **agreement** class, and fall essentially into one of three types:

- averaging (AVG)
- non-averaging (NAV)
- hybrid averaging-non-averaging (ANA)

Averaging Clock Synchronization The principle of averaging algorithms, of which there are many examples (Lamport and Melliar-Smith, 1985; Lundelius and Lynch, 1984a) is shown in Figure 12.14a. Clocks start a synchronization run when a number of them reach the end of the period, or resynchronization interval, before the current precision π^{i-1} exceeds the allowed worst-case precision. Each clock disseminates its value to all others. The values received form a *clock-readings vector*, used as input to a convergence function, which computes the value to be applied to the new virtual clock launched in the next period. Assuming that clocks receive the same vector, the same value is applied at the end of this process to each clock, and the initial convergence or precision enhancement of the next period, measured by δ^i in the figure, is dictated by the jitter with which this action is performed.

Both forming the clock-readings vector and reaching agreement on the new clock value can be done in a local manner, or may involve a minimum number of interactions among a minimum number of clock processors. This depends on the fault assumptions of clocks, processors and network, with regard to type (e.g., crash, omissions, value) and number (e.g., how many clocks may fail). The convergence function itself also depends on these fault assumptions, and may be based on known functions such as the *fault-tolerant average*, or the *fault-tolerant midpoint* as exemplified in the figure, which consists of selecting the middle value in the ordered clock-readings vector (*see* also *Resilience* in Chapter 7). As a final note, the precision enhancement of this class of algorithms is directly affected by: the variance of communication delays; the variance in the execution duration of the algorithm.

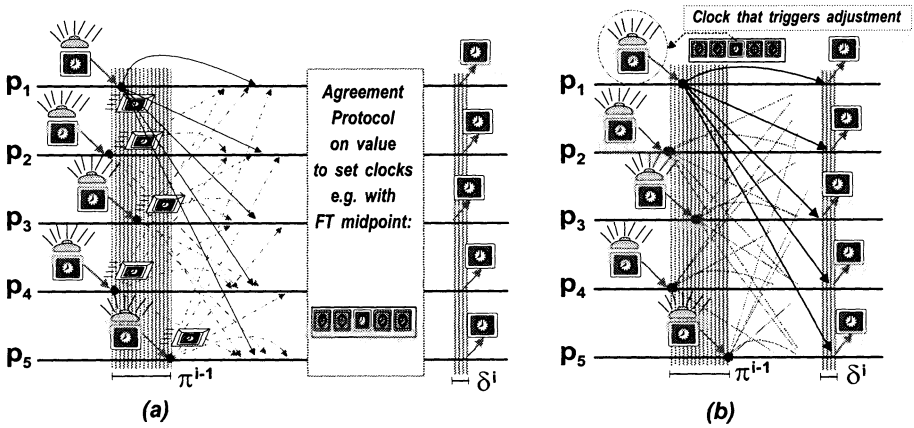

Figure 12.14. Clock Synchronization Algorithms: (a) Averaging; (b) Non-Averaging

Non-Averaging Clock Synchronization Non-averaging algorithms rely on a different principle: instead of disseminating the value of the clock, each clock processor disseminates a control message to signal the end of a period. Figure 12.14b illustrates the principle. The difference is that in the averaging class, the message just sends the clock value as input to a function that performs agreement on the value to adjust the clock with, at some later time. In the non-averaging class, the message does not carry a value, but itself positions an event in the timeline meaning that 'it is the start of period i now', e.g. 5:00, on the sender's clock. All the other processors will do the same. Since some clocks may fail and for example start too early, processors wait for the k^{th} clock to signal the event, in order to get sufficient evidence that it is 5:00. The number k is dictated by the failure assumptions (we leave it to the reader to understand why the example in the figure is resilient to $f = 1$ failures, and thence $k = 2f + 1 = 3$, p_1's clock). After seeing this k^{th} time marker message, processors simply adjust their clocks to 5:00, instead of agreeing on a value, as in the averaging case. Precision enhancement, measured by δ^i in the figure, depends on the difference between the instants when the several processors adjust their clocks, dictated by the magnitude and variance of the delivery delay of the time marker messages and on the assumed faults. Examples of protocols relying on this principle are (Halpern et al., 1984; Srikanth, 1987; Drummond and Babaoğlu, 1993).

Averaging-Non-Averaging Clock Synchronization Hybrid averaging-non-averaging (ANA) clock synchronization algorithms combine the advantages of averaging and non-averaging classes. The principle, depicted in Figure 12.15, consists of reaching agreement on two issues: (i) the clock that triggers the adjustment (the NAV type of agreement); (ii) the adjustment value to load the clocks with (the AVG type of agreement). The algorithm starts by having each clock processor disseminate a message both to stand as time marker of the end

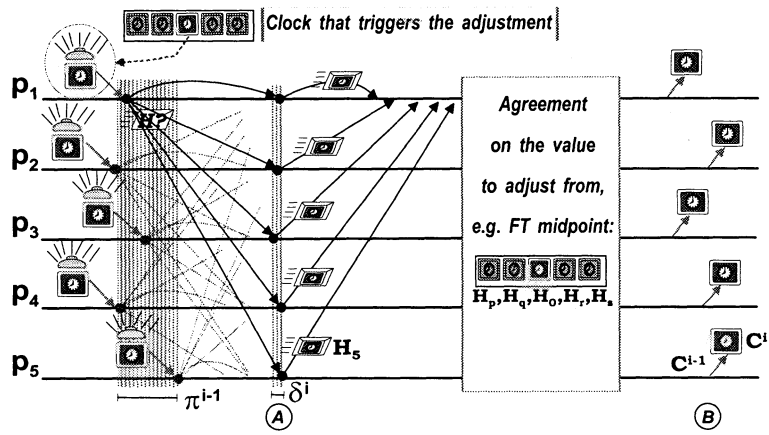

Figure 12.15. Hybrid Clock Synchronization Algorithms (clock p adjustment – $\Delta_p = H_0 - H_p$; new period clock value – $C^i = C^{i-1} + \Delta_p$)

of period $i - 1$, and to act as a remote clock-reading request message to all clocks ($H?$). Each message marks a point in all processors' timelines, as in non-averaging algorithms, and at the k^{th} message, all agree that this is a valid synchronization point. The figure exemplifies p_1 as triggering the adjustment. Clock readings may be sent to the requester, as in the figure, or broadcast to all. This depends on how the agreement on the adjustment value is performed. Suppose an FT midpoint function is applied to the clock-readings vector, and thus H_0 is selected as the new clock value. Instead of setting the clocks with this value, adjustments to each individual clock p are determined by computing $\Delta_p = H_0 - H_p$. Note that the clock-readings vector contains the value of all clocks when the time marker message $\langle H? \rangle$ was received. In consequence, when adjusting the value of each clock for the new period, $C^i = C^{i-1} + \Delta_p$, we are referring the adjustment to the time marker reception instants. The precision enhancement δ^i, is indeed determined at that moment, as signalled in the figure (A), and thus independent of the moment when the adjustment is applied at each clock (B), and largely insensitive to the rate of drift of the hardware clocks during the agreement interval, which may be neglected for most situations. Examples of protocols relying on this principle are (Veríssimo and Rodrigues, 1992; Clegg and Marzullo, 1996).

12.8.3 External Synchronization

External clock synchronization aims at injecting the time of an external reference, the *master* clock, into all *slave* clocks of the system. In that sense, clocks synchronize themselves individually from that reference, rather than agreeing among each other. They either must trust the master, or find fault-tolerant configurations where several masters can be consulted. Known external clock synchronization algorithms are of the **master-slave** class, and the simplest

type relies on *dissemination* of time by the master. This can be achieved through radio broadcasting (spread-spectrum like in GPS, or long-wave beacons), and is for example the method used to synchronize GPS receiver units. However, it is seldom convenient, and sometimes not even possible, that all nodes of a distributed system have a radio receiver.

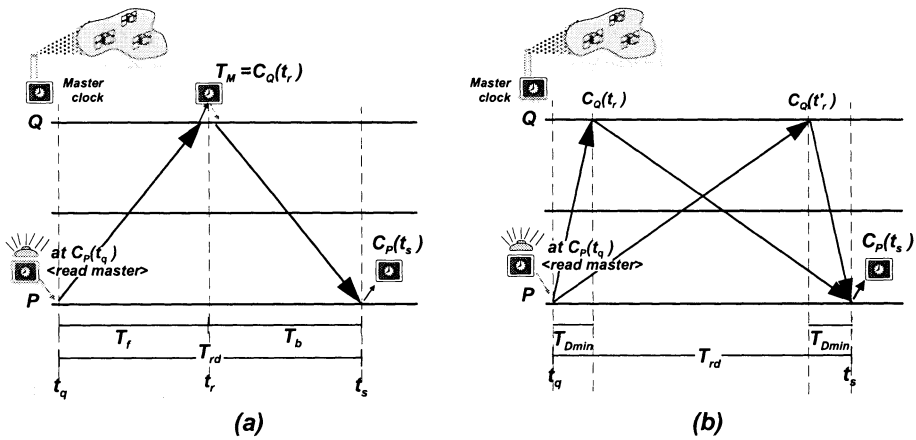

Figure 12.16. Round-Trip External Clock Sync.: (a) Zero-Error; (b) Measuring the Error

Round-Trip Clock Synchronization Most known external clock synchronization algorithms are of the round-trip type, whereby they take the initiative of performing a remote read of the master, receiving its time back and adjusting their own clocks at that time. Figure 12.16 illustrates the technique for estimating the master clock time value at t_s. This involves knowing t_r. Short of knowing T_f or T_b, we estimate t_r to be at the midpoint, or half the round-trip time T_{rd}. Then, $C_P(t_s) = C_Q(t_r) + T_{rd}/2$. Here we can see that only a symmetric round-trip yields zero error. However, the only run that P can detect as symmetric is a minimum delay run, where the request and reply messages have T_{Dmin} duration. P can find this out if it measures $T_{rd} = 2T_{Dmin}$. As a matter of fact, P can do better, it can determine a bound on the error of any remote clock read, given by: $\epsilon = \pm(T_{rd}/2 - T_{Dmin})$. The scenario is depicted in Figure 12.16b. Cristian proposed a well-known *probabilistic* clock synchronization protocol, in which, given a target accuracy, the slave makes several request-reply attempts trying to get a fast enough round-trip that yields the desired reading error (Cristian, 1989). The protocol is probabilistic, because the chances of success depend on the distribution of the network delay. This protocol inspired the Network Time Protocol (NTP) that we discuss in Chapter 14 (Mills, 1991).

12.9 INPUT/OUTPUT

Input/Output is concerned with all that crosses the computational border of a system. In real-time systems, this means talking about the perception or **observation** of, and the **actuation** on, the environment. These are performed, respectively, by **sensors** and **actuators**, devices through which the control system (the real-time computer system) captures and modifies the state of the controlled system (the physical process system). Besides the functional aspects of I/O, there is a general concern in industrial systems with making I/O reliable. A typical example of that concern is a 'trusted' valve used in critical fluid control, a quad compound represented in Figure 12.17. Distribution is as desirable for I/O as it is for computation, either for fault-tolerance or because of the often distributed nature of processes. In this section, we discuss the input/output paradigm under the functionality, distribution and fault-tolerance viewpoints.

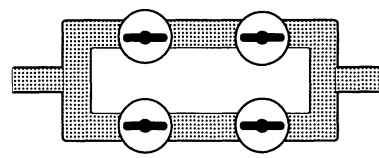

Figure 12.17. A Classical Mechanical Quad Valve

12.9.1 Observation

Observation is: *the act of acquiring and eventually pre-processing the state of a real-time entity, through one or more sensors.* Observations can be made through several techniques, generally falling into: sampling; polling; latching; interrupt.

We use *sampling* when it is possible to decide when to make an observation. Sampling is normally used to observe continuous entities, which exhibit more or less predictable variation curves. The frequency of observation is determined by known control rules. A basic rule-of-thumb is that it should never be less than twice the frequency of the highest harmonic of the entity's waveform. When entities exhibit sudden changes, information may be lost. With *polling* we can observe the entity in infinite loop. This should only be done during a short period where some significant changes are expected (this is analogous to the spinlock in programming), since it ties the processor up. It can be done normally if the sensor has a dedicated micro-controller (what are called intelligent sensors). Sensing can also be performed by *latching* significant changes. This is very appropriate for discontinuous entities, such as, but not only, digital values. Latches are memory elements that register one or more state changes. They can be combined with other techniques, such as sampling or interrupts. It is convenient to use *interrupts* when entities are sporadic: their state changes sel-

dom and at undetermined times. The sensor interrupts the computer system, and thus triggers the observation at the appropriate moment.

There are a set of *pre-processing* functions that improve the quality of observations. Amongst them: *low-pass filters* cancel glitches (e.g., switch debouncing); *likelihood* or *plausibility* tests filter out implausible deviations (e.g., abnormal air temperature readings); *correction curves* linearize sensor outputs (e.g., temperature transducers); *composite variables* are computed from several samples or sources (e.g., rates; averages; approximate convergence functions using sensor replicas or sensors of different physical magnitudes).

12.9.2 Sensor Types

Sensors measure a number of physical magnitudes, through several principles. A few examples of magnitudes: motion; speed; acceleration; pressure; temperature; humidity. Digital sensors are generally on-off: object passing; door open-shut; end-of-range. Some are sophisticated: pattern; color; shape. Several principles are used: magnetic field; photoelectric; hall effect; piezoelectric; image; light (visible, UV, infrared).

When there are not special reliability concerns, the sensor implementation is *simplex single-access*, that is, a single chain of elements, from the real-time entity through the sensor to the computing element. However, if the computing element fails, the sensor is lost. When the process is critical, use of some redundancy may be considered, in order to achieve reliability and availability. Figure 12.18a depicts an actively replicated sensor set, feeding a set of computing elements. This *fully-redundant* sensor configuration is characterized by having each sensor connected to a different representative, desirably in a different node. The computing elements receive the same set of values from the replicated representatives and perform consensus on the final value to be used.

12.9.3 Actuation

Actuation is: *the act of issuing and eventually post-processing a command to change the state of a real-time entity, through one or more actuators.* An actuation can be triggered in several ways: immediate; deferred; periodic.

Immediate actuation is the normal actuation as soon as issued. *Deferred* actuation is invoked to be issued after a specified delay, or at a specified time. *Periodic* actuation is invoked to be repeated at every T. The last two assume some processing capability by the actuator.

There are a set of **post-processing** functions that improve the quality of actuations. The output of actuators can be corrected, through *correction curves*. A command may be issued in its final form to a set of replicated actuators. In this case, *replica control* functions make sure that every replica receives the adequate stimulus. For example, making sure that the individual actuators of the quad valve of Figure 12.17 receive the adequate commands.

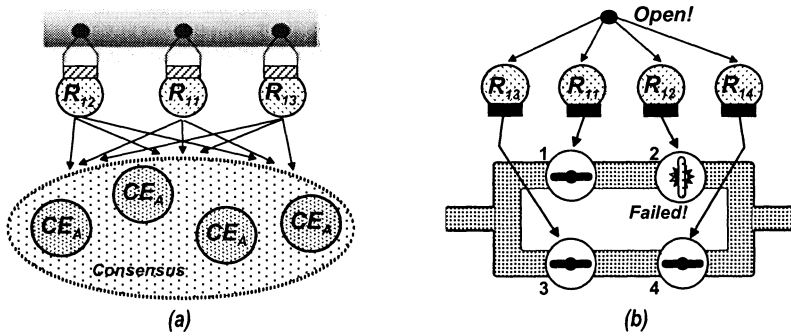

Figure 12.18. (a) A Reliable Sensor; (b) A Reliable Actuator

12.9.4 Actuator Types

Actuators are of different types, depending on the target of actuation. We may be talking of such different things as closing a switch, raising a digital boolean value, opening a valve, or driving a stepping motor. Actuators act on physical magnitudes of the environment, by means of an electric command. Examples of actuator driving principles are: electro-mechanical; electro-pneumatic; electro-hydraulic; electronic.

Similar to sensors, the simplest form is the *simplex single-drive* actuator. It is used when actuators have outstanding mechanical reliability, or when the loss of the actuator is not crucial for system operation. Otherwise, this is an unreliable combination. The *fully-redundant* actuator is the most dependable configuration. In the example shown in Figure 12.18b, there are 4 actuators, one computing element per actuator, residing in different nodes. In this particular example, the configuration controlling the quad valve ensures that an open or a close command always work, in the presence of arbitrary behavior of at most one controller-valve pair. We leave it to the reader to confirm this assertion.

12.9.5 Distributed and Fault-Tolerant I/O

Distributed and fault-tolerant input-output uses some of the techniques for communication and replication management presented in previous parts of this book, and the requirements and properties discussed in those contexts apply to I/O as well. Synchronization of inputs and outputs is most important. That was left clear in Section 12.4. Firstly, we have the problem of positioning the time of the observations and actuations at different nodes in the timeline. Secondly, we have the problem of replication. In multi-access or replicated inputs, the meaning of two samples of the same real-time entity separated by an even small interval may be very different. This is also valid for multi-drive or replicated outputs, for analogous reasons.

The *synchrony* of distributed observation and actuation can influence the quality and even the correctness of I/O operations on real-time (RTe) enti-

ties. *Steadiness* measures the jitter of distributed observations and actuations. *Tightness* measures the degree of simultaneity of sampling, and the allowed skew of replicated actuation commands.

12.10 SUMMARY AND FURTHER READING

This chapter addressed the main paradigms concerning real-time. We began by introducing ways of specifying timing constraints and of detecting their violation. Then, the relation between the real-time entity and its computational representative was defined, namely the duality between time and value. Real-time communication, flow control, scheduling, clock synchronization, and input-output, complete the set of paradigms studied in this chapter.

For further study on failure detection in distributed systems, a formal treatment is introduced in (Chandra and Toueg, 1996) for time-free systems. Timing failure detection is addressed in (Veríssimo and Raynal, 2000). On the formal embodiments of temporal specifications and synchronization, communication by time, temporal order, real-time causal message delivery, and time lattices, please read (Lamport, 1984; Kopetz, 1992; Suri et al., 1994; Veríssimo, 1996).

Response-time-based schedulability analysis is detailed in (Sha et al., 1990), or (Burns and Wellings, 1996; Audsley, 1993). In (Leung and J., 1982) the deadline-monotonic algorithm is described, optimal for maximum termination times smaller than the period. Scheduling of sporadics is further treated in (Mok, 1983; Sprunt et al., 1989; Jeffay et al., 1991). Priority inheritance blocking is a syndrome avoided by priority ceiling and immediate ceiling protocols, thoroughly described in (Cornhill et al., 1987; Sha et al., 1990; Burns and Wellings, 1996). Further material on scheduling can be found in (Ramamritham, 1996a; Burns and Welling, 1996; Audsley, 1993; Buttazzo, 1997) and (Ramamritham et al., 1989; Fohler, 1995; Tindell, 1994; Mossé et al., 1994) namely for distributed settings.

Formal aspects of clock synchronization are surveyed in (Schneider, 1987). The special issue published in (RTS Journal, 1997) gives a good account of current approaches to clock synchronization in open systems. Examples of a hybrid class of clock synchronization algorithms combining internal and external clock synchronization are featured in (Veríssimo et al., 1997; Fetzer and Cristian, 1997b). Further notes about the use of GPS in clock synchronization can be found in (Dana, 1996). Clock synchronization on embedded systems and field buses presents unusual problems, addressed in (Ramanathan et al., 1990; Kopetz and Ochsenreiter, 1987; Rodrigues et al., 1998b; Schossmaier et al., 1997). Interval clock synchronization, based on the notion of interval clocks, is discussed in (Marzullo, 1983; Schmid and Schossmaier, 1997).

A generic treatment of sensors and actuators in reactive systems is presented in (Wood, 1991). Paradigms for fault-tolerant sensors are discussed in (Marzullo, 1990; Kopetz and Veríssimo, 1993). The I/O paradigm may be generalized and extrapolated to generic devices (e.g., gateways, management information bases), a viewpoint taken in (Powell, 1991; Wood, 1991).

13 MODELS OF DISTRIBUTED REAL-TIME COMPUTING

In this chapter, we aim at providing a global view of what is timely behavior of a distributed system architecture. The chapter starts by introducing classes of real-time systems with different timeliness guarantees, setting the stage for introducing the several frameworks for structuring real-time systems. Then, it discusses strategies for the several approaches to building an architecture, where the paradigms presented in the last chapter show their usefulness. The main models of real-time systems are then presented. From the viewpoint of timing: partial synchronism; time-triggered; and event-triggered models. From a functional viewpoint: real-time communication; real-time control; real-time and active databases; and quality-of-service.

13.1 CLASSES OF TIMELINESS GUARANTEES

It is difficult, if not impossible, to build a real-time system satisfying all sorts of requirements concerning timeliness. In consequence, we normally devise our models and systems for specific classes of requirements: hard, soft, and mission-critical real-time.

Hard Real-time System - system where any failure to meet *timeliness* requirements may have a high cost associated

The hard real-time class specifies that the system *must always* be timely, in order to avoid costly timing failures (e.g., a slowed down module of a tomato processing unit causes all tomatoes to be smashed against each other; a slow

P. Verissimo et al. *Distributed Systems for System Architects*

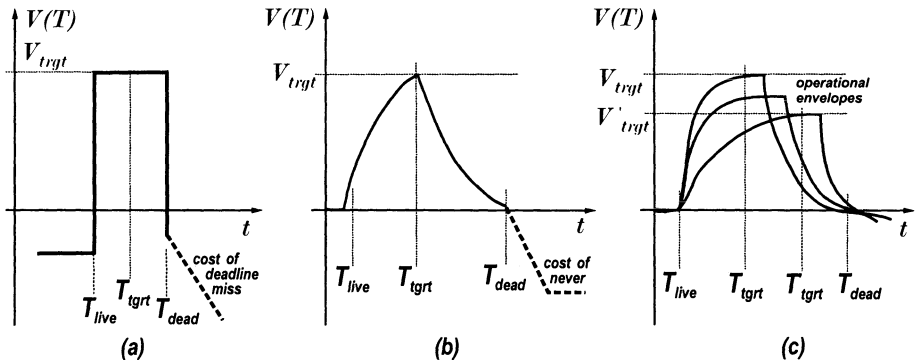

Figure 13.1. Time-Utility Curves for Several Real-time Classes: (a) Hard; (b) Soft; (c) Mission-Critical

robot arm may get stuck under a one-ton press). These systems are also called **time-critical**. Figure 13.1a shows a graphical way of defining criticality, through *time-utility* or *time-value* functions (Jensen and Northcutt, 1990; Burns and Wellings, 1996). These functions portray the value of the service as a function of the time at which it is provided, w.r.t. the time at which it should be provided. We see that not providing the service within the [liveline;deadline] interval carries a cost. The criticality of the system is given by the cost of failure versus the benefit of normal operation, measured by how deep the hatched line goes down (e.g., a stalled engine controller may destroy valves and cylinder pistons; or a late change in rail crossing points can cause a train to de-rail).

Soft Real-time System - system where occasional failure to meet *timeliness* requirements is acceptable

The soft real-time class specifies that the system *should often enough* be timely, that is, it can fail to meet timeliness specifications provided that failures do not occur with too high a probability, or too great a deviation or lateness degree (*see* Section 6.2). This is depicted in Figure 13.1b, where we see the value curve evolving smoothly to and from the targetline as time passes. For example, the specification of a real-time ticket reservation system might look like this: any transaction should terminate within 10 seconds for at least 90% of the times, and within 1 minute in 100% of the times (the latter defines the allowed lateness degree). Whenever there is a cost associated to an extremely long delay or omission of the service, this can be represented, as shown in the hatched line of the figure. Many *interactive systems* are non real-time systems, but the truth is that a fair number of them should indeed have been designed as soft real-time systems, if user requirements were to be respected. How many times did we have to wait in a long bank counter line, only to hear that "It's the computer system's fault"?.

Mission-Critical Real-time System - system where any failure to meet *timeliness* requirements may have a cost associated, and the occasional failure to meet those requirements is considered an exception

The class of mission-critical (also called *best-effort*) systems is designed in order to guarantee that timeliness requirements are systematically met. However, these systems are normally complex and/or the environment behavior is not totally specified, such that timeliness cannot be fully guaranteed. The occasional occurrence of timing faults is tolerated, but should be considered exceptional. If the service risks being provided near or after the original T_{dead} recurrently, because of timing failures, considerable degradation may take place. The system should provide some means for reconfiguration to a less demanding operational envelope where the service, despite having less value (V'_{trgt}), may be provided *predictably later*, thus avoiding timing failures. Figure 13.1c introduces this notion of dynamics in the time-value function of the service. Examples of systems of this class are air traffic control systems, weapons control systems, telecommunications intelligent network architectures.

13.2 REAL-TIME FRAMEWORKS

We spend this section analyzing the main frameworks at the disposal of the architect to build, or build-in, real-time capabilities in distributed systems. This discussion will refer to material presented in the previous chapters, and will introduce some of the models we will discuss later in this chapter. Some are concerned with the synchronism and timing, such as partial synchronism, time-triggered, event-triggered. Others are concerned with the functional models, such as real-time communication, real-time control, and real-time and active databases.

13.2.1 Budgeting

All starts by equating the budget of the time-related variables of the system to be designed. The average load budget derives from the system requirements (number of real-time entities, individual debit of information to and from the system, computation and communication costs). Then, it is essential to make assumptions on the arrival patterns, that is, on how the several load flows are distributed in the time domain: periodic, sporadic, aperiodic. In complement, the timeliness requirements are introduced: when, at what pace, how fast, and with what guarantees, is this load to be processed, and output. We have discussed a few of these issues in Section 12.1. These requirements are introduced by the necessary match between the environment where the system will run, and the type of service desired from it. This budget dictates a first approximation on the power of the computational and networking resources (MIPS, throughput, latency), that can be fine-tuned through subsequent design phases.

13.2.2 Synchronism and Timing

Satisfying the timeliness requirements is concerned with choosing the adequate system model. Whilst the hard-real time class is normally equated with a full-synchrony model (*see Synchronous Models* in Chapter 3), more adequate partial synchrony models have emerged that represent well the soft and mission-critical classes, as we are going to discuss in Section 13.4. On the other hand, choosing time-triggered or event-triggered frameworks implies many subsequent decisions about the system support, with regard to implementing timeliness, as we will see in Sections 13.5, 13.6, and 13.7 (e.g., information flow control, scheduling). As discussed in Section 12.7, periodic scheduling is more easily analyzed and tested, and is the workhorse of time-triggered system models, that we address in Section 13.7. However, when the environment does not comply with periodical operation, the combined scheduling of periodic and sporadic processes is necessary, as accommodated by the event-triggered model, studied in Section 13.6.

13.2.3 Networking

Distributed scheduling is still a subject of research, because of its complexity. For that reason, it is common to separate concerns between network scheduling and local scheduling, at least in larger scale systems. This justifies the importance of real-time networking, as the framework of building and configuring networks and achieving real-time communication. Most networks have their own distributed scheduling policies, on top of which communication protocols can be built. Section 12.5 has introduced the real-time communication paradigm, whose notions will be applied to a real-time communication model discussed in Section 13.8. For certain applications, it may be necessary to extend the bounded delay requirement on reliable multicast message delivery to richer paradigms, such as causal or total order.

13.2.4 Input-Output

Input-output, in complement to processing, is the framework dealing with the boundaries of the system. It is specially concerned with interfacing the environment, whatever architecture it applies to: control, producer-consumer, client-server. Section 12.9 described the main functional principles and techniques of input-output that should guide any implementation. For a matter of separation of concerns, I/O should be seen as a framework separated from the processing activities in the core of the system. In the generic real-time system model that we are going to discuss in Section 13.5, we show that this separation, desirably equated under the entity-representative paradigm presented in Section 12.3, is extremely convenient, for correctness, functional, and implementation reasons. The time-value paradigm (*see* Section 12.4) establishes the correctness conditions for dealing with representations of real-time entities, such as the time of their values, or their value over time.

13.2.5 Programming

Real-time systems programming is a rich and still evolving framework, since it currently combines a wealth of notions, such as: concurrency; robustness; timeliness. Real-time programming is by nature concurrent, and lies on essentially two approaches: language or system support. Language support is granted through language annotations or specialized languages, like Ada (ADA 83, 1983), whose current version is Ada 95 (Bodilsen, 1994), which offer primitives that support concurrency under timeliness constraints. These are enforced by run-time support environments built on top of a real-time kernel. The system support approach uses programming languages not necessarily designed with real-time in mind, which may lack built-in concurrency or timeliness enforcing constructs. These are granted through system libraries supplied as subsystems of a core real-time kernel. This is the track of the so-called *real-time multitasking executives*, so frequently used as a COTS basis to build real-time systems. The virtues of objects for structuring programs are also useful in a real-time context (Forestier et al., 1989). The *real-time object* paradigm serves as an encapsulation element. It can be used to represent real-time entities and computing elements, and provide functional as well as temporal containment, making it easier to reason in terms of the time-domain correctness of complex tasks.

13.2.6 System Support

Last but not least is the question of system support. Besides quantitative issues (e.g. of computational power), the architect must be concerned with qualitative issues. To begin with, there is the balance between specially developed and COTS components. Many critical embedded systems are purposely built. However, the growing trend for using COTS components applies to real-time systems as well, specially when the number of non-critical, complex application-specific systems in the soft real-time and mission-critical classes increases. Under this perspective, the real-time kernel should accommodate the scheduling needs equated during the early design stages, not only the scheduling policy, but also the thread dispatching issues. This reasoning applies to the choice of a standard LAN or field bus as networking support, which must comply not only with the raw data throughput and reliability requirements, but also with the goals for individual frame transmission delay in several urgency classes. This may require additional software-based mechanisms complementing the native priority schemes of the chosen network.

13.3 STRATEGIES FOR REAL-TIME OPERATION

The possible strategies to be followed by the architect in real-time system design are conditioned by several factors, such as: class of operation, price, performance, available technology. The latter present the architect with a number of tradeoffs. The main strategies line-up according to the main target of the system being considered: money-critical, safety-critical, complex large-scale,

quality-of-service. Once agreed on a strategy, the system is conceived along the guidelines suggested by the frameworks for real-time design just presented.

13.3.1 Money-Critical

We designate a system *money-critical*, when the stakes involved in the case of timing failures are high. For example, the control of a process factory, where failures or errors or delays may cause the malfunction or destruction of the process line, entailing considerable loss of product or estate. Having in mind the need for fault tolerance (we have made this statement repeatedly throughout this part), the real-time specific strategies imposed by this target class of systems foresee highly regular designs, and belonging to the hard real-time class of system. Systems should be as simple as possible, even at the cost of versatility, in order to be verifiable. The operation models should be as regular and predictable as possible, which is achievable when the operation of the controlled system can be put within pre-specified boundaries (e.g., a batch process line). This will be further debated in Section 13.9, on real-time control models. Of course, this strategy suggests choices such as: static scheduling; periodic processing and communication; sampling I/O. These are materialized for example in models of the time-triggered type, that we study in Section 13.7.

13.3.2 Safety-Critical

A money-critical system becomes *safety-critical* when the cost involved in case of catastrophic failure is incommensurate to the value of the service in normal conditions. This is related with but not limited to the potential to jeopardize human lives. These systems belong to the area of control (e.g., nuclear power, fly-by-wire, drive-by-wire, etc.). The strategy for building safety-critical real-time systems concerns the use of the most demanding combinations of real-time and fault tolerance techniques, and of structured and formal design and verification methods (Burns and Wellings, 1995; Sinha and Suri, 1999), bringing the discussion of the last section to an even higher level. The main difference to money-critical systems is that safety-critical systems have necessarily to pass a certification process against standards describing several types of requirements. The latter imposes such stringent criteria for all phases of system commissioning, from design to implementation, that it may overshadow decisions based on technical features. In the sense that catastrophic failure of money-critical systems also has a lack of safety aspect, even if in moderate terms, the former are also called *safety-related*.

13.3.3 Embedded

Today's embedded real-time systems have assumed a distributed nature. In the measure that embedded distributed real-time applications start being built on top of virtual-circuit like structures, such as LANs or field buses, the gap between the desired 'distributed systems' model of the applications and the

underlying 'networking' model becomes apparent. One such problem is the effect of the network *jitter* in the application timings. If analyzed superficially, it would seem the problem would disappear with more rigid interconnecting structures, such as physical-circuit or digital-bus ones, such as time-triggered TDMA networks. However, there is more to an architecture than the network. The strategy for modern embedded system design calls for looking at the complete architecture, what we might call a **field-bus distributed system**, where networking, middleware support and applications are seamlessly integrated and thus no gap exists. If an architectural approach to the problem is followed, this can be done in most of the existing field buses. Applications must see an adequate distributed systems support environment in the underlying infrastructure. The latter must be built with building blocks exhibiting adequate properties. Finally, the adequate algorithms must be used to interconnect these blocks, in order to achieve the desirable reliable real-time behavior of the whole.

13.3.4 Complex Large-Scale

Whenever a system is complex and/or has a considerable scale, the enforcement of timeliness requirements may conflict of the lack of assumptions restricting the behavior of the environment— because of the complexity of the process— and with weaknesses of the infrastructure— because of its scale. In this case, one cannot expect to fully predict the evolution of the environment, and as such, the design strategy of the system cannot be turned to static policies, and regular behavior. These systems fall to a great extent into the mission-critical of soft real-time classes. Some will be money-critical in nature, but cannot be designed by the strategies we proposed above. As a compromise, the strategy to design these systems suggests choices such as: timing error detection and recovery; dynamic scheduling; combined scheduling of sporadic and periodic processing and communication; interrupt-driven I/O. These choices are materialized for example in models of the event-triggered type, that we study in Section 13.6. Many architectures of this kind are client-server, which introduces a further ingredient of non-determinism. The real-time database model, addressed in Section 13.10, is a representative design strategy for this kind of systems.

13.3.5 Quality-of-Service

An emerging class of real-time systems is quality-of-service oriented. For example, video-on-demand, voice-over-IP, visualization and other multimedia services, of the soft real-time class. These systems are normally structured around the producer-consumer or client-server architectures. Less important than avoiding timing failures at all cost, is the limitation of their effect and production, through adaptation to uncertain timeliness of the environment (varying operation conditions) and uncertain utilization patterns (varying number of users and of flows). As such, a strategy for this kind of systems should be

based on: dynamic scheduling; combined scheduling of sporadic and aperiodic events; QoS failure detection and adaptation.

13.4 SYNCHRONISM MODELS REVISITED

We have addressed synchronism in Section 2.6. We have learned that it is expressed in terms of timeliness properties, and it underlies any mechanism for securing real-time behavior, since synchronism stipulates that it is possible to bound action delays (processing and network). There is an obvious relation between the *time* and *synchronism* paradigms: if time is expressed in durations and positions in the timeline, synchronism bounds the variance and imprecision with which those durations and positions are established, i.e., the *jitter* as defined in Section 12.1. There is also a relationship between the *synchronism* and *order* paradigms, established by the definition of temporal order made in Section 2.7: if synchronism expresses the steadiness of a protocol, the latter dictates how fine a temporal order the protocol can implement, that is, how small the δ_t-*precedence* potential causality threshold (Veríssimo, 1996).

13.4.1 Timeliness versus Liveness

We saw in Chapter 1 that it is convenient to formally specify what we wish of a system in terms of high-level safety and liveness properties. Safety properties specify that wrong events never take place, whereas liveness properties specify that good events eventually take place. Take the example specification we used then:

P1- any delivered message is delivered to all correct participants

P2- any message sent is delivered to at least one participant

P2 is a liveness property. We know it will happen, but we do not know when. This is not compatible with our expectations about real-time systems, where we decided to use time as artifact to guarantee synchronization with the environment, and all players involved: the user that produces inputs and receives outputs, the sensor that is read, the actuator that produces an output, the processor that has to finish a task, the network that has to deliver a message, the failure detector that detects that something was not done as and when it should. We learned that this is done by defining *timeliness* properties: whatever happens is only correct if done by T, at T, every T, etc. That is, we are introducing a safety condition. Timeliness properties have in fact a safety facet, specified by means of time operators or time-bounded versions of temporal logic operators (Koymans, 1990; Lamport, 1994). In order to turn our example specification into a real-time specification, we would augment it with a timeliness property:

P3- any delivered message is delivered within T_{Dmax} from the time of send request

13.4.2 Partial Synchrony Models

Real-time systems behavior is materialized by *timeliness* specifications, which in essence call for a synchronous system model. However, large-scale, unpredictable and unreliable infrastructures are not adequate environments for synchronous models, since it is difficult to enforce timeliness assumptions. Violation of assumptions causes incorrect system behavior. In alternative, the asynchronous model is a well-studied framework, appropriate for these environments. This status quo leaves us with a problem: fully asynchronous models do not satisfy our needs, because they do not allow timeliness specifications; on the other hand, correct operation under fully synchronous models is very difficult to achieve (if at all possible) in large-scale infrastructures, typical, for example, of mission-critical systems, since they have poor baseline timeliness properties.

What system model to use for applications with timing requirements running on environments with uncertain timeliness? The question probably has more than one answer. Recently, approaches have emerged that tolerate *partial synchrony* of the system while securing timeliness properties, or else detecting their absence, for example, the *timed asynchronous* (Cristian and Fetzer, 1998), or the *quasi-synchronous* (Veríssimo and Almeida, 1995) models. These models rely on the observation of two aspects of real-life environments:

- synchronism is not a homogeneous system property;
- worst-case termination times or delays are much larger than normal ones.

That is, parts of the system or epochs of its operational life can be reliably considered synchronous. As such, bounds on response time and other variables can be defined that hold on a subset of the system, or during limited periods of its operation.

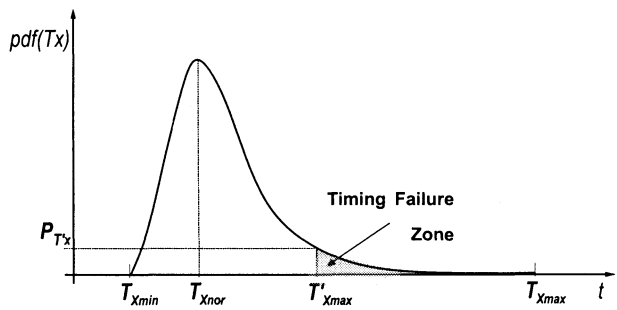

Figure 13.2. Distribution of termination times

Observe Figure 13.2: in real environments, the probability density function of the time it takes for an activity to complete (e.g., message delivery), rather than being a step, has a shape similar to the one represented in the figure. In settings with uncertain timeliness, such as large-scale systems, the worst-case termination time, T_{Xmax}, if it exists, is much higher than the average case (T_{Xnor}), such that it becomes useless. The assumption of a shorter, artificial

bound T'_{Xmax}, increases the expected responsiveness. In contrast, it increases the probability of timing failures, since T'_{Xmax} has a non-zero probability of not holding $(1 - P_{T'_X})$, yielding a timing failure. However, if timing failures are detected when they occur (*see* Section 12.2), a reliable system can be built which works synchronously when the environment allows, and reacts in order to preserve correctness in the presence of timing failures.

Table 13.1. Partially Synchronous Model Properties

• Some of the system properties— processing or message delivery delays, rate of drift of local clocks, difference between local clocks— have a known bound • For the others, a known bound may not exist or may be too large

The generic properties of this model are listed in Table 13.1 (*see* also Table 3.2 in Chapter 3). Partially synchronous systems offer support for building mission-critical or soft real-time applications that are dependable, in the sense that they offer resilience to failure of timing assumptions.

13.5 A GENERIC REAL-TIME SYSTEM MODEL

We present a generic real-time system model that will help us understand other, specialized models introduced later in this chapter (Kopetz and Veríssimo, 1993). The model is depicted in Figure 13.3, and it relies on the separation between input-output, communication, and computing.

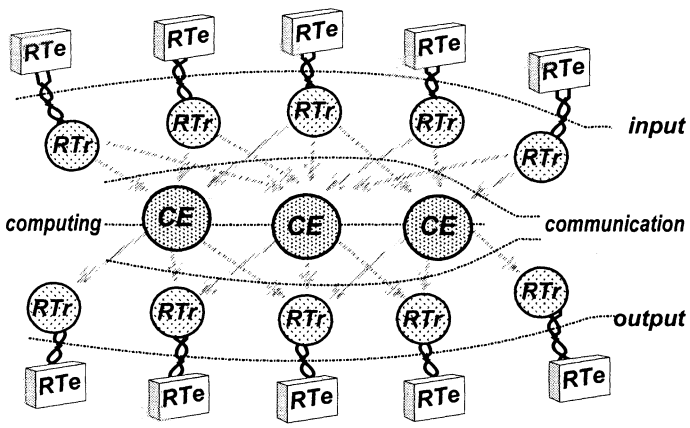

Figure 13.3. Generic Real-Time System Model: RTe- real-time entity; RTr- representative; CE- computing element

Computing is performed by *computing elements* (CE), computational entities which process observations of RT entities (RTe), modify the internal state

of the system, and eventually trigger actuations on other RTe's, to modify the state of the environment. *Input-output* is performed between the RTe's and their *representatives* (RTr) (*see Entities and Representatives* in Chapter 12). Representatives assume a crucial role of containment between the physical reality and the computer world. RTr's are to RTe's what device drivers are to normal computers: they hide the complexity and idiosyncrasy of sensors and actuators, and translate that physical reality into variables and structures, which computers can understand. Once defined the rules and boundaries for the relation between an RTe and its representing RTr, the latter can be assumed to "be the RTe", inside the computing system. *Communication* ensures the timely flow between the computing elements and the input and output representatives, since we are reasoning in terms of distributed systems.

The value of an RTe E at any instant t, $S = E(t)$, is represented *inside the computer* by $S^r = r(E)(t)$. The implications of using S^r instead of S have been studied in Sections 12.3 and 12.4. This separation of concerns simplifies system design. One step is to ensure that the RTr faithfully represents its RTe in all situations including assumed faults in the I/O area. Once this secured, the architect can devote his effort to ensuring that the computing mechanisms, given correct representations of all input and output RTe's in terms of computing abstractions, produce correct results. Last but not least, and given the needs dictated by the above steps, he must see to it that the communication mechanisms ensure that information flows in a timely manner between CE's and RTr's.

Particular models may implement specializations of this generic model. The producer-consumer may be concerned with the observation of real-time data and its timely processing and storage at one time, and its recovery and timely rendering at another, in an open-loop fashion. On the other hand, in real-time control the loop is closed, from observation of the environment, to computation and actuation, and feedback to the observed input, through the environment. This model can be activated in a time-triggered fashion, or in an event-triggered fashion. It is very useful to analyze these apparently contradictory schools under the same generic model, as we do next.

13.6 THE EVENT-TRIGGERED APPROACH

Figure 13.4 illustrates the architecture and information flow of an event-triggered (ET) system. The *information flow* retains an event-like nature up to the center of the system. This approach adapts well to overload and unexpected events, either from the same or different representatives.

Event-triggered system - one that reacts to significant events directly and immediately

13.6.1 Event-Triggered Architecture

In any real-time architecture, it is important to regulate the flow of information from the periphery (representatives) to the core of the system (computing

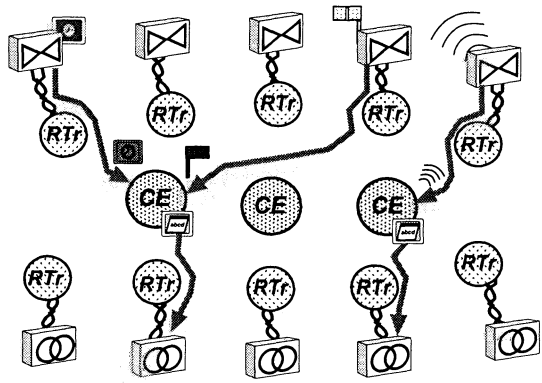

Figure 13.4. Event-triggered Architecture

elements). This is preferably done using policies adapted to real-time, such as *rate-based flow control*, which act at the input representatives, rather than at the computing elements (*see Flow Control* in Chapter 12).

The definition of the operational envelope of ET systems (the set of assumptions about the behavior of the environment to control) is by nature not rigid. A beneficial consequence is that they can admit situations where they are stressed beyond the design-time worst-case workload without falling apart. Average *responsiveness* of ET systems is the best possible in general, since an event gets through and is processed just depending on preemption speed and communications medium access time. Given the bursty nature of most ET traffic, priorities play an important role in letting urgent events get through in a message or event queue.

The dynamics of ET systems is at the same time their weakness and their strength. The irregular event distributions (sporadics) make *predictability* hard to ensure. It is an advantage however, when system complexity and lack of environment knowledge cannot ensure predictability but require versatility, as in mission-critical operation. Given their dynamic characteristics, the scheduling of ET systems is not decided *a priori*: it must be done *on-line*, will be *preemptive* in most cases, and will be prepared to schedule a computation serving a sporadic event of extreme importance, immediately it arrives (*see Scheduling Sporadic Tasks* in Chapter 12).

The ET approach is also valid under the perspective of critical applications. Imagine that such a system is processing critical hard real-time tasks, but sometimes enters overload periods: an ET system can be designed in order to withstand this extra load, exhibiting what is called *graceful degradation.* This prefigures the *mission-critical* class of operation, which ET systems are very apt for. In the impossibility of meeting all timeliness requirements, they will attempt at reducing the cost involved in failing to meet some, for example by selecting those that would, if not met, lead to catastrophic failure.

The extra complexity put into the design principles of ET systems, a disadvantage already pointed out, becomes an advantage once the system is designed and it is necessary to extend it. Because of the provisions to support dynamic operation, this turns out to be easier. The event-based approach allows selective dissemination of information and allocation of resources.

13.6.2 Event-Triggered Protocols

ET systems are specially devoted to treating *sporadic* events, which accurately represent the environment encountered in most real-time problems (*see Temporal Specifications* in Chapter 12). Event-triggered systems, as shown in Figure 13.5, are so to speak 'idle', waiting for something to happen. When an external event or burst of events occurs (e.g., objects passing under a detector beam), it is transformed into an *event message* or message interrupt, which is sent to the interior of the system, where one or several computing elements process it, modify the system state, and eventually produce outputs. Note that messages are processed as they arrive, and so are outputs. This is strict ET behavior.

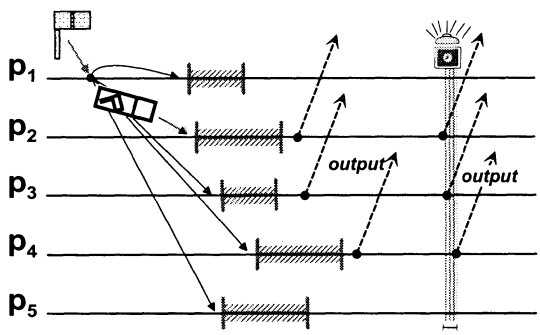

Figure 13.5. Timing Issues in an Event-triggered System

ET systems are said to be *susceptible to event showers*. However, this would be a simplistic way of looking at the problem. Note that our model establishes a barrier at the representatives, which can establish measures for filtering the redundant and spurious information out of the flow to the computing elements, neutralizing the 'event shower' effect. In other words, if e_1 is the leading event carrying information about an alarm situation, the information carried by the subsequent event shower tail contributes little to the information already brought by e_1. The system architect can thus create rules to: *compact* successive instantiations of the same alarm at the representatives; *discard* redundant events, either at the representative or upon arrival at the computing element (by a pre-processor); *reserve* communication and computing resources for the forthcoming shower; *smoothen* the transmission of event bursts to the computing elements, by using rate-based flow control to space them along the interval between bursts, when there is enough laxity.

Another common idea is that ET systems are *unsteady and untight*, or in other words exhibit a high jitter, making them inadequate for more demanding real-time control applications. This derives from the fact that processing and outputs are basically event-driven. However, with the assistance of synchronized clocks it is very easy to superimpose time-triggered behavior on top of an event-triggered communications or processing system. This is exemplified on the far right of Figure 13.5: actuations can be triggered by the clocks, at a steady (periodic) pace, and tightly at all nodes. In fact, timing of an event triggered system may be purely timer-driven or clock-less (*see* Figure 2.10 in Chapter 2), but it may also be clock-driven, without necessarily being periodic (*see* Figure 2.11 in Chapter 2).

13.7 THE TIME-TRIGGERED APPROACH

The architecture and information flow of a time-triggered (TT) system is shown in Figure 13.6. In TT systems, the *information flow* is throttled in the periphery of the system, as shown in the figure, because there is a preliminary transformation from event to state (state of the RTe).

Time-triggered system - one that reacts to significant events at prespecified instants in time

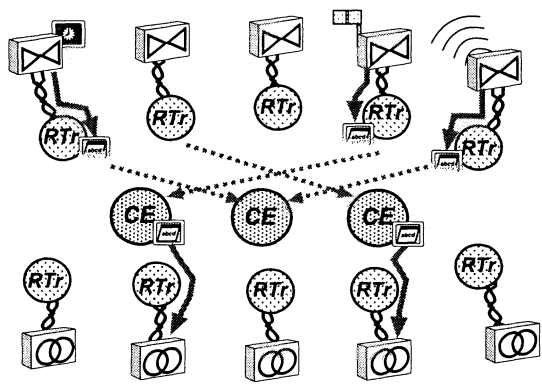

Figure 13.6. Time-triggered Architecture

13.7.1 Time-Triggered Architecture

By assumption, there is no such thing as *overload* in TT architectures. However, one cannot exert 'flow-control' on the environment. This is universally true, that is, the situation is no different from ET systems in this respect. So, *flow control* in TT systems is performed between the environment and the representative, which only accommodates the information it is prepared to recognize and transform in one period. In order that no overflow occurs at the representative, the event arrival distribution must be well known. Otherwise, unexpected but important events may be filtered out. As such, the bounded

demand assumption can be a dangerous one if the environment is not thoroughly described: a requirement of TT system design. Take the (unfortunate but realistic) example of a weapons-control system: a TT system designed for a maximum of 50 incoming enemies will be blind or puzzled when a 51st enemy arrives.

The crux of the TT approach is to create the conditions to make the system appear well-behaved enough that events occur in synchrony with the system clock, and simple enough that only the assumed event distributions occur. This works for a number of applications, namely in continuous process control. Once the I/O problem is solved, from then on the system is quite predictable. If we look at Figure 13.6, we see that events are transformed into state at the representatives. As a matter of fact, it is as if every representative holds a piece of the global state, all of which are disseminated to all computing elements when the start-of-period ticks, in the form of *state messages*. These messages contain structured data, the whole of which forms the global state or system context. In consequence, they are not consumed; instead, each overwrites its previous instantiation in the global state, like a piece in a puzzle. Going back to Figure 13.6, the flow between representatives and computing elements pictured there is periodic and static, and it is always the same amount of information: the objective is the cooperative refreshment of the global state. In consequence, *resource reservation* rather than flow-control is necessary.

TT systems are built as periodic automata, and as such are the perfect match for *periodic* events, mostly generated by artificial processes, but that is the case in a lot of process control settings (discrete or continuous). TT *response* is thus cyclical, occurring at pre-specified instants in time. Responsiveness depends on the system period. Given that the environment is asynchronous with regard to the system, an event may have a waiting time of one cycle in the worst-case, half-cycle on average. So, when a very urgent alarm arrives it may have to wait that long to be served. This is only relevant when the expected service delay for these urgent sporadic events is of the order of magnitude of the system period or shorter.

TT systems have other advantages. Given their cyclical and lock-step (in pulses) evolution they are simpler to test and show correct in the case of, for example, critical applications. Design for *predictability* is easily achieved in TT systems. In consequence, they are excellent for small closed systems controlling static environments and repetitive processes, like some manufacturing cells. Predictability and testability are very important factors of choice when reliability and safety figures have to be very high. In consequence, it is not surprising to see critical problems (nuclear, fly-by-wire or drive-by-wire control, train control, etc.) addressed by TT systems. On the other hand, they may be harder to commission for large-scale or often-varying settings.

Scheduling is calculated *off-line* and it is *static*. The system still has to treat several tasks, so the automaton we mentioned before is a multi-tasking one. However, since we know the evolution of the system *a priori*, we can also determine how long the processing steps last, and combine their interleaving

in order that all tasks perform their work by the end of a period. Once this schedulability exercise is done, the schedule may be cast into the system executive (*see Static Scheduling* in Chapter 12). Obviously, increasing the number of nodes, or adding new tasks will require a total redesign of the node slots, communications, schedule, etc. Mission-critical systems are thus not a field of preference for TT systems, which cannot handle unexpected events, and can only handle sporadic bursts if considered at design time, by means of several predefined *operating modes.* This means as many pre-tested schedules, and still leaves open the problem of falling outside the 'outer' operating mode.

13.7.2 Time-Triggered Protocols

Activity in TT is triggered at environment-independent instants, a shown in Figure 13.7. The period is short enough to match the rate of evolution of the environment and long enough for the duration of processing.

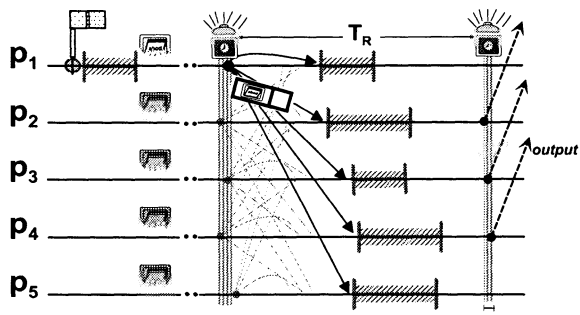

Figure 13.7. Timing Issues in a Time-triggered System

There is no event shower effect in TT systems. To comply with the cyclical operation style, events are collected and pre-processed in the periphery of the system, between periods, as shown in the figure. Note that observations are made by sampling, and pre-processed so that whatever relevant happened since the last period at each representative is included in the partial state information to be sent in *state messages* to the computational part. At given moments, these messages are disseminated to the computing elements, which assemble them in a complete state image of the system. The necessary tasks are then deterministically scheduled at each node, in order to perform the necessary state transformations and eventually schedule results to be output to representatives. The are also output at a pre-determined instant, which coincides with the end of the period. Actuation can be made as steady and tight as allowed by the precision of the synchronized clocks (mandatory in TT systems).

Flow control would also seem to be a non-problem. Note two things however: (1) the system is always working as in full load— be the environment 'quiet', or going through worst-case alarm situations, the information flowing is the same; and (2) the representative must handle any showers of alarm situations, and the maximum rate of event arrivals. The messages transactioned between the latter

and the computing elements must always carry enough octets to represent the maximum I/O information that can be generated in a period in any situation. The efficiency of use of resources is thus compromised in favor of determinacy of operation.

Time-triggered communication is mandatory to support the kind of operation depicted in Figure 13.7 for a TT system. In fact, the whole architecture, including the network, works in pulses. For example, the MARS system, a pure TT system (Kopetz et al., 1989a), accesses the network through a TDMA protocol called TTP, using a slotted access method where each node knows exactly when and for how long to transmit (*see* Figure 2.12 in Chapter 2).

13.8 REAL-TIME COMMUNICATION MODELS

When speaking of real-time networks, there are several fields of interest: the virtual circuit type of network, the realm of medium to large-scale settings; the physical circuit type, the realm of control networks; and the digital bus type, the realm of lock-step systems. These types were presented in Chapter 11, in increasing order of tightness of expectations about real-time behavior. The timescales involved also span an increasingly finer range, from the second to the microsecond. The use of specialized protocols and hardware is more related with the digital-bus end of the spectrum, in contrast with the standard components available for virtual circuit types over open networks.

We are going to develop a real-time communication model based on the definition of the paradigm made in Section 12.5. The steps are generic enough to be useful for design of any of the network types considered. The architect will have to specialize the implementation of the model according to the expectations of a given type. We discuss structure and configuration (load budget, urgency classes, network properties), connectivity enforcement (medium reliability, redundancy measures), and algorithmics (preventing timing faults, tolerating omission faults).

13.8.1 Configuration and Structure

In order to *configure* the network, a few points must be taken into account:

- traffic patterns;
- latency classes;
- network sizing and parameterizing.

The designer must be able to model the *traffic patterns* offered to the network by each individual node, falling in the types defined in Section 12.1: aperiodic, periodic or sporadic traffic. The next step is to perform some traffic separation in *latency classes*, corresponding to the classes of urgency in the system, which translate into a range of successively higher transmission time bounds. As a general rule, urgent data (critical) and protocol control frames should be mapped onto the highest network priority. The other traffic should

be distributed by the remaining priorities, according to their relative urgency and/or importance.

Regardless of the network technology used as the basis for a reliable real-time network, one important point is not to get tied to a particular implementation, i.e., achieve portability. *Abstracting* from physical particularities of a network is thus a good architectural principle. Take a LAN for example. LANs have a set of common properties, some never stated, which can be extremely helpful for the construction and proof of correctness of some distributed systems paradigms. For the sake of example, we describe a set of useful abstract properties of LAN-type networks, including field buses, in Table 13.2.

Table 13.2. Abstract LAN Properties

Broadcast	correct nodes receiving an uncorrupted frame transmission, receive the same frame
Error Detection	correct nodes detect any corruption done by the network in a locally received frame
Network Order	any two frames received at any two correct nodes, are received in the same order at both nodes
Full Duplex	frames transmitted are also received at the sending node
Bounded Omission Degree	in a known time interval T_{rd}, omission failures may occur in at most k transmissions
Tightness	correct nodes receiving an uncorrupted frame transmission, receive it at real time instants that differ, at most, by a known interval T_{tight}
Bounded Transmis. Delay	any frame is transmitted by the network within a bounded delay T_{TXmax} from the send request

The *Broadcast* and *Error Detection* properties impose detection of errors in the value domain, in a broadcast (e.g., the CRC protection mechanism). Frames not passing the test are simply discarded, usually by the MAC VLSI. *Network Order* is the physical order imposed by the mutual exclusion on the communication medium. The *Full Duplex* property is only directly supported by some LAN chipsets, such as the Token-Ring, and the switched Ethernet. It is also supported by the CAN chipset. The *Tightness* property measures the maximum interval between reception instants in different nodes, a function of the variances of the end-to-end propagation delay and the interrupt processing time for a frame. Omission errors normally have a physical origin: mechanical defects in the cable, EMI corruption of a passing frame, modem synchrony loss, receiver overrun, transmitter underrun, etc. As such, it is possible to make probabilistic assumptions about the occurrence of omission errors dur-

ing an arbitrary interval of concern T_{rd}, which boils down to a number k of successive transmissions hit by omission errors, as per the *Bounded Omission Degree* property. The *Bounded Transmission Delay* property specifies a maximum transmission delay, which is T_{TXmax} in the absence of faults. It is not a self-contained property of networks, depending on the particular network, its sizing, parameterizing and loading conditions.

13.8.2 Maintaining Connectivity

Medium reliability is the crucial issue to secure connectivity, and it involves fault tolerance measures in the networking infrastructure (*see Fault-Tolerant Networks* in Chapter 6). Of course, if the whole network is replicated, we have n paths to each destination, and can mask $n-1$ medium failures (*see* the examples of Figure 6.4 in Chapter 6). However, many real-time networks are simplex infrastructures that can also be provided of medium connectivity measures. In essence, we can count on two strategies to maintain connectivity (*see* the examples of Figure 6.5 in the same Chapter): space-redundant medium, where several media are available in parallel; reconfiguring medium, where the medium breaks-up and reconfigures to a new correct state.

13.8.3 Achieving Bounded Transmission Delay

Enforcing a bounded and known transmission time bound T_{TXmax} (*Bounded Transmission Delay*), is not guaranteed per se in a LAN. The load budget and the separation in latency classes allocated to LAN priorities must be equated with network throughput, to achieve figures for the transmission delay numbers. This process is analogous to the discussion of the schedulability tests and maximum termination time done in Section 12.7.

Each class is scheduled according to the available policy of the network, either a dedicated scheduling mechanism, such as in special purpose TDMA networks, or the available fixed priority mechanisms of standard LANs and field buses, which work by guaranteeing some precedence order or a certain amount of the channel bandwidth to fulfill latency requirements. Hybrid mechanisms have been the most successful: building improved schedulers on top of standard LAN or field bus priority mechanisms.

The network should be *parameterized* in order to reply to these requirements. Some LANs, such as the Token Bus, require a setting of several timers in order to correspond to the load budget and latency class definition just discussed (Janetzky and Watson, 1986; Gorur and Weaver, 1988). It may sometimes be discovered at this point that the latency aimed for is not achievable with the amount of load offered. Then, the load has to be adjusted, in an iterative procedure. The process is LAN dependent and should not be neglected for a successful design.

13.8.4 Achieving Bounded Delivery Delay

The *bounded omission degree* assumption introduced by the *Omission Degree* property is paramount for achieving a bounded delivery delay, since it dictates the level of redundancy needed, either time redundancy (number of repetitions), or space redundancy (number of network channels or media). Note that without these reliability assumptions, it is impossible to put a bound on the duration of the action of getting a message through.

In order to pursue the divide-and-conquer strategy, the architect must now consider the implementation of an omission-free network infrastructure, for an omission degree of k, through a choice of fault-tolerance mechanisms. The algorithms must also be aware of the measures taken at hardware configuration time about medium reliability, discussed in the previous section. In Chapter 7 we studied the basic error processing protocols, which point to several alternatives: (1) space redundancy with error masking; (2) time redundancy with error masking; (3) time redundancy with error detection/recovery.

Let us assume that up to k errors may occur. The time budget for alternative 1 is transparent to omissions, since at least one frame arrives on one of the media, in each redundant transmission on all of the $(k+1)$-plicated networks. The time budget for alternative 2 has a constant overhead of transmitting not one but $(k+1)$ copies of each frame. The time budget of alternative 3 has to take into account the accumulated durations of the error detection timeouts in the worst-case runs involving $k+1$ retries. Note that if a bounded transmission delay T_{TXmax} is ensured, then by the *Bounded Omission Degree* property either of mechanisms 1-3 achieves message delivery in bounded time despite omission errors, and in absence of partitioning.

13.8.5 Controlling Partitioning

Let a network be partitioned when there are subsets of the nodes such that nodes from different subsets cannot communicate with each other[1]. *Physical partitioning* may occur in a real-time network on account of physical defects: bus medium failure (cable or tap defect), transmitter or receiver defects, etc. On the other hand, there can be *virtual partitioning*, where networks exhibit glitches in their operation, e.g. *congestion* in wide-area networks, or *inaccessibility* in LANs or field buses (Veríssimo, 1993).

We can prevent physical partitioning (alternative (1) in the last section), or we can have a glitch of variable duration with medium-only space redundancy (the solution for alternatives (2) and (3)), or we can reduce but not avoid virtual partitioning. We had rather address all forms of partitioning in the same way, and talk about 'controlling', rather than 'preventing'. In consequence, we say partitioning is acceptable in a real-time network if it is *controlled*: duration is bounded and suitably short for the service requirements.

[1]The subsets may have a single element. When the network is completely down, *all* partitions have a single element, since each node can communicate with no one else.

13.8.6 Implementing Flow Control

Sometimes it is necessary to exert flow control on the communication load. Traditional flow control mechanisms normally used in non real-time communication, such as the sliding-window scheme used in TCP (Comer, 1991), can delay transmissions for long periods or even arbitrarily. Besides, they are not appropriate for multicast communication.

Rate control is a real-time flow control policy implementing a rhythmic operation that is equally suited for periodic and sporadic traffic. It aims at matching the sender's average debit with the recipient's capabilities, without discontinuities in traffic flow. The Xpress Transport Protocol (XTP) is a typical example of protocol using rate control, capable of real-time behavior (XTP, 1998). Credit and rate control may be combined in XTP.

13.9 REAL-TIME CONTROL

Control is historically the main application of real-time systems. As shown in Figure 11.3 in Chapter 11, it is concerned with observing selected real-time entities from a *controlled process*, and computing whatever corrections and actions necessary to maintain the process within the pre-specified operational envelope. The meaning of operational envelope may vary depending on the type of control:

continuous control	applies for example to batch processing, and its correct operational envelope is defined by ensuring that each of a certain number of variables stays within a maximum error from their pre-defined values, the *set-points*
discrete control	applies for example to manufacturing, and its correct operational envelope is defined by ensuring that a pre-defined sequence of actions takes place at the appropriate times, dictated by either a pre-defined static schedule, or in reaction to external events, or both

13.9.1 Architecture of Distributed Control Systems

What is called control loop in computer control is: observing the value of a read-only real-time entity RTe_1 at a given time; computing the necessary action; acting on a write-only RTe_2 in response to the input; physical feedback from RTe_2 through the environment, eventually changing the state of RTe_1, which will be read again, and so forth. A distributed real-time control architecture has the following building blocks:

- input and output representatives
- computing elements
- reliable multicast or broadcast communication subsystem

Representatives are connected to the real sensors and actuators. A representative is a driver or task that handles the respective sensor or actuator. In centralized control, sensors and actuators are normally connected to a single

computing element, or *controller* in this case, and in consequence representatives co-reside with the computing elements. In distributed control, the representatives of input and output real-time entities may reside at different nodes, where they may or not coexist with controllers.

Anyway, it is essential to disseminate the information from sensors to the several controllers, so that they acquire a common knowledge about the image of the system and of any events occurring. This is easily achieved through the synchronous reliable multicast or broadcast protocols that we have studied in Chapter 2, either under the initiative of representatives, or of controllers. Supposing that each controller schedules its tasks correctly in reaction to that information, some controllers will produce outputs to representatives connected to actuators. Here again, reliable multicast protocols may be used in case several actions must take place at different actuators in a synchronized way, for example, in the case of the quad valve studied in Section 12.9.

In *algorithmic* terms, real-time control implies the solution of essentially three problems:

- timely and correct observation of real-time entities of the environment
- timely and correct computation of the action to be executed next
- timely and correct actuation on real-time entities of the environment

The first problem has to do with the way observations are performed. Observations are essentially related with determining a value at a given time or determining the time at which a given value occurs. Considering that the information gathered from the environment is as accurate as needed in the value and time domains, the second problem has three facets: the control algorithms must be logically adequate to the problem; observations must be used correctly over time, since the state of the RTe may vary after being read; scheduling of the necessary tasks must ensure the production of results within the required response times.

The third problem concerns the correct implementation of those results when they imply actuations, and has to do with the ability to position an action at a given point in the timeline (*see* Sections 12.4 and 12.9).

Distribution and *replication* in control introduce additional problems:

- synchronizing and disseminating observations made at different nodes
- splitting and allocating control tasks to different nodes
- synchronizing actuations made at different nodes

Models that allow splitting tasks of a global computation through different nodes have been studied in Chapter 3. One can use global time for triggering a synchronized acquisition or actuation. Clock-driven observations allow synchronizing the acquisition of data from different or replicated sensors. This way one can determine the order of external events, even if acquired in different nodes. Clock-driven actuation serves the purpose of accurately positioning actions on the environment in the timeline, with cooperative or replicated actuators. The steadiness and tightness of the operations influence the results (*see Distributed and Fault-Tolerant I/O* in Section 12.9). In general, the time-

liness properties of control actions influence the quality of control, and even its correctness.

13.9.2 Quality of Control

Logical aspects of the algorithmics notwithstanding, the quality of control is affected by two time-domain factors: response time and jitter. The *response time* affects quality for two reasons: (a) if an RTe varies significantly during a single control cycle (observing/computing/actuating), when the actuation is issued the RTe no longer has the value read, producing an error; (b) if the response time exceeds an absolute bound, for example for discrete actions, a serious failure may occur. Fixed delays can sometimes be compensated for by prediction or extrapolation. The remaining error caused by the accumulated *jitter* in the several phases of the control cycle cannot: jitter of the observation; communication and execution time jitter; positioning jitter of the actuation. We are going to base our analysis of these errors using the time-value paradigm (*see* Section 12.4).

Time of a Value Observe Figure 13.8a, depicting a curve made of sensor observations ($\mathcal{H}_r(E)$) of the value of a real-time entity ($\mathcal{H}(E)$) along time. On the left half we depict the problem of observing of a value at a predetermined time T_{obs}. In the example, we assume a sensor amplitude error bounded by ν_s, and a jitter bounded by ζ_o. Recalling Section 12.4, a *consistent* observation of a value at a given time has a value domain error bounded by a known $\mathcal{V}_o$. In this case, $\mathcal{V}_o = \nu_s + \nu_o$, ν_o being the maximum equivalent value error caused by ζ_o. This situation is depicted in the figure: the real value is E_a at T_{obs}; the reading suffers the error ν_s and yields E_a'; finally, the jitter ζ_o positions the reading action too early, yielding E_a'' read at $T_{obs} - \zeta_o$, as if it were read at T_{obs} with an additional value error of ν_o. The alternative perspective of determining the time at which a predetermined value V_{obs} occurs may also suffer a maximum error that is shown on the right of Figure 13.8a: the real event is E_b at T_{val}; however, its observation is delayed by ζ_s, to when $\mathcal{H}_r(E)$ reaches the V_{obs} threshold (E_b'), given the error of ν_s; the observation receives a later timestamp, $T_{val} + \zeta_s + \zeta_o$, given the positioning jitter ζ_o. The error in determining the time at which a given value occurs is thus bounded by $\mathcal{T}_o = \zeta_s + \zeta_o$. For discrete entities like booleans, the effect of the sensor threshold error (the equivalent timing error ζ_s) can be neglected in the expression of $\mathcal{T}_o$.

Value over Time We take two control-related examples to illustrate this issue: the bearing of a ship; the angle of the crankshaft of an engine piston. There is a vast body of research on computer control algorithmics that this book does not intend to replace. The derivations discussed here are very simple, and aim at illustrating the generic problem of use of time-value entities in control, and its implications in a computerized and distributed context.

Consider an observation $\langle v_i, t_i \rangle$ of a time-value entity E (for now consider it perfect, that is, $v_i = E(t_i)$, neglecting any observation errors). This observation

will be used for some computation that depends on the value of E, e.g., a slight correction in the helm by the autopilot, or the computation of the ignition point by the engine controller.

When the result is produced, at some later time t_j, a non-negligible amount of time may have elapsed: the time spent in effectively delivering the observation (T_D), plus the execution time of the action (T_X) (see the figure). This *delay* error affects the accuracy of the control action, since it is supposed to concern the value of E at t_j, but was based on its value at t_i. Additionally, the error itself is uncertain, because of *jitter*: both the delivery time and the termination time have a variance, and in consequence, t_j cannot be determined accurately. Let us denote the maximum total error due to the execution of the computation (observation delivery and adjustment computation) as $\mathcal{T}_x$.

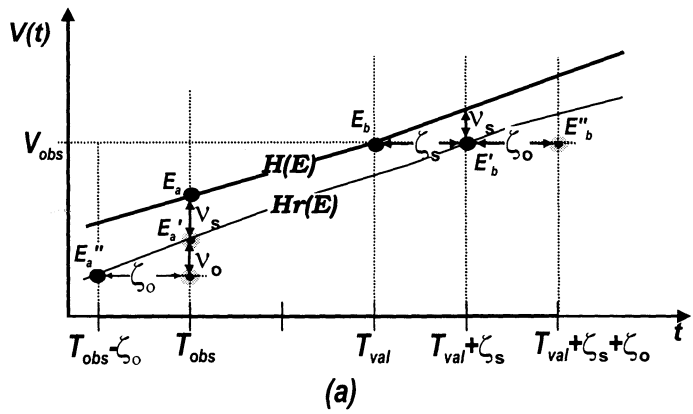

Figure 13.8. Time-Value Entities: (a) Time of a Value

If $E(t)$ is not predictable, either because it cannot be represented analytically (e.g., discrete, or too complex), or because the cost of computing the prediction would be too high, the original observation value must be used in the computation of the control adjustment. Then, the total timing error is bounded by the maximum response time of the system, $\mathcal{T}_x = T_{Dmax} + T_{Xmax}$. However, $E(t)$ is often a predictable function with a small *extrapolation* error for some near future time. This happens with most continuous variables, and opens the way to several optimizations that we do not discuss in detail. In general terms, it consists of computing a prediction $\langle v_k, t_k \rangle$ of the value of E for $t_k > t_i$ and using it instead of $\langle v_i, t_i \rangle$ as input for the computation. This adjustment cancels part of the $\mathcal{T}_x$ error. In order to capture the intuition, note the simplest of the approximations: to cancel the error due to the fixed part of the delays. As a first shot, consider making $t_k = t_i + T_{Dmin} + T_{Xmin}$. The result will be produced at an instant t_j anywhere between t_k and $t_k + \zeta_d + \zeta_t$, respectively the variances of the delivery time and the termination time. In consequence, the effect of the error is reduced to the jitter terms. Consider further adding the average jitter, e.g., $t_k = t_i + T_{Dmin} + \zeta_d/2 + T_{Xmin} + \zeta_t/2$:

this reduces the effect of the jitter to half, i.e. the total timing error becomes $\mathcal{T}_x = \pm(\zeta_d + \zeta_t)/2$. Since $t_k - \mathcal{T}_x \leq t_j \leq t_k + \mathcal{T}_x$, if we compute the adjustment for t_k, then neglecting the extrapolation error, the value error $\mathcal{V}_x$ will be bounded by the maximum deviation during $\mathcal{T}_x$.

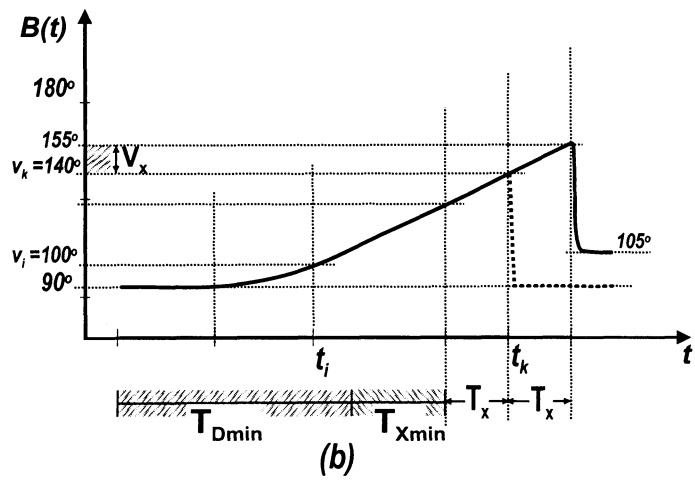

Figure 13.8 *(continued).* Time-Value Entities: (b) Value over Time

Coming back to our examples, we illustrate the first in Figure 13.8b (bearing of a ship), and leave it to the reader to elaborate on the second. Given the continuous nature of the real-time entity `bearing` ($B(t)$), timing errors translate into value errors in the helm correction. Observe the figure: the ship had a steady bearing of 90° (East) but starts to drift. The current bearing is read at t_i to be 100°, but the adjustment is computed using an extrapolation to time t_k using average delays. If the execution finishes at t_k, then the bearing correction is accurate. However, the termination event may take place at t_j anywhere between $t_k \pm \mathcal{T}_x$. In the extremes, the maximum error in bearing correction is attained ($\pm\mathcal{V}_x$): the autopilot applies a correction to make the ship come back from a bearing of 140° to 90°; however, for the situation shown in the figure, the ship will under-correct to 105°, because it is already aiming at 155° when the correction is applied ($t_k + \mathcal{T}_x$), and vice-versa for the other extreme of the interval.

If the response time gets to be of such magnitude that some boundary condition may be crossed before termination, more serious problems than mere control inaccuracy may occur. An excessive response time compared with the rate of drift will make the ship successively under-correct and over-correct, following an unstable route. Likewise, if the computation of the ignition point finishes after the crankshaft reaches the earliest possible ignition point, the ignition control may miss the ignition point, and even damage the engine.

To conclude, we now introduce the observation-related errors discussed in the previous section. The maximum timing error of the observation itself is

$\mathcal{T}_o = \zeta_s + \zeta_o$, with a corresponding value domain error bound of $\mathcal{V}_o$. If the observation was read from a representative, or a real-time database, it concerns a past state of the RTe. Recalling Section 12.4, the correctness condition is *temporal consistency*: an observation stored at a representative must not be older than a stipulated bound $\mathcal{T}_a$, called the absolute validity interval. This interval introduces a value domain error bounded by a known $\mathcal{V}_a$. The total timing error of the control loop in our example, from the instant of observation to the instant of actuation, is thus bounded by:

$$\mathcal{T}_{ctl} = \mathcal{T}_o + \mathcal{T}_a + \mathcal{T}_x$$

This bound on accumulated timing errors translates, for continuous variables, into a predictable equivalent value error bound:

$$\mathcal{V}_{ctl} = \mathcal{V}_o + \mathcal{V}_a + \mathcal{V}_x$$

Wrapping-up, quality of control implies securing the following, for all RTe's:

- the value stored at the representative must be consistent – this implies bounding the observation error by a known $\mathcal{T}_o$, and must be secured by the I/O instrumentation, which must be good enough;
- the value stored at the representative must be temporally consistent at all times, since it gets obsolete with time – this implies bounding the absolute validity interval of the representative value to a known $\mathcal{T}_a$, and must be secured by the I/O interface, which must refresh often enough;
- a reading from the representative must remain temporally consistent until used – this implies bounding the response time of the action where the reading is used, such that the associated timing error $\mathcal{T}_x$ is bounded, and must be handled inside the communications and computational parts. The error term $\mathcal{T}_x$ may be further reduced by using prediction functions;
- when computations concern several RTe's, the concerned observations must also be mutually consistent, i.e., the timestamps of all observations should fall within a known interval $\mathcal{T}_m$ (*see* Section 12.4).

13.9.3 Distributed Control

The ability to order related input events and to position related output events in the timeline is crucial for distributed real-time control, since now sensors and actuators are connected to several controllers with a network in the middle.

Consider the problem of analyzing a stream of events of a failure situation in an electrical power distribution network. Measurements are made by distributed representatives, which send them to the controllers. The causality relations between events will dictate the correct reconfiguration procedure. However, if two events that are causally related are inversely ordered, this will disturb the recovery procedure, and in some cases may worsen the problem.

Likewise, consider the problem of distributed control of the semaphores of a road crossing. Actuation of the several lights must be synchronized: light must go green on one side just after the light went red on the other side; pedestrian lights must change accordingly. The lack of tightness in positioning each related set of actuation events can cause incorrect behavior, for example, causing both lights to be green during some time. As for replicated actuators, they should

be exercised at approximately the same time, otherwise either a voting fails, or it looks like more than one actuation was made.

Distributed Observation Distributed observations receive timestamps from the local clocks. The meaning of the timestamps versus the effective separation of the observations depends, as we know, on the granularity (g) and the precision (π) of the clocks (*see Time and Clocks* in Chapter 2). The two relevant problems introduced by distribution are (Veríssimo, 1994): (a) what is the minimum separation of observation events that can be ordered by a system of global timestamps; (b) what is the minimum difference between timestamps of two distributed observations so that their mutual order can be determined.

Observe that these questions are extremely important, since up to now, we have been able to derive correctness conditions for control considering a conceptual timestamping facility common to all entities. Only if we know how to relate timestamps produced by different clocks at different sites, can we apply these derivations to distributed control. In consequence, the system *must* have a clock subsystem with good enough precision and granularity to address the application requirements in determining: δ_t-precedence— the minimum interval to generate causal relations (*see Temporal Order* in Chapter 2); jitter— the error in positioning distributed events in the timeline or in measuring distributed duration variance (*see Timing of Events* in Chapter 12).

Consider that the virtual granularity of the clocks is made $g \geq \pi$, that is, coarser than their physical granularity. This *granularity condition* (Kopetz, 1992) ensures that distributed timestamps of the same event are at most one tick apart, which looks like a sensible measure. With this approximation, one can draw interesting conclusions about distributed observations (Veríssimo, 1994), listed in Table 13.3.

Table 13.3. Separation, Timestamping and Ordering of Events

• any two events separated by at least $2g$ are correctly ordered
• any two events separated by at least g but less than $2g$ are never inversely ordered, but may receive the same timestamp
• any two events separated by less than g may receive the same timestamp or be arbitrarily ordered with consecutive timestamps
• the same event observed at two sites may receive the same timestamp or be arbitrarily ordered with consecutive timestamps
• events with the same timestamp are always concurrent (not $2g$-precedent)
• only events with timestamps separated by at least 2 are guaranteed to be in their physical order
• only events with timestamps separated by at least 4 are guaranteed to be $2g$-precedent

Distributed Actuation Distributed actuation is normally concerned with synchronizing actions in two situations: actuators in different locations; replicated actuators. For example, the lack of tightness of a replicated actuation can transform an exactly-once actuation into several actuations with at-least-once semantics, and that may be undesirable. Consider *actuator granularity*, g_a, the interval from command input to completion of the action or until the actuator is ready for a new command, whichever is longer. During that interval, the actuator does not respond to further commands (e.g., a discharge laser is unable to fire again until it recharges). If the actuation points of the replicas are separated by g_a or more, they will be perceived as more than one actuation. In consequence: *the tightness of actuation must be better than actuator granularity*, $\tau < g_a$. However, it need not be much tighter than g_a, so, on the other hand, this condition removes the general belief that replicated output must be tightly synchronized. Take the example of the valve quad of Figure 12.17 in Section 12.9: many electric valves have actuation times in the order of the many hundreds of milliseconds, so replicated valves can support untightness of this order of magnitude.

13.10 REAL-TIME DATABASES

Real-time databases (RTDB) were born form the need to access, manipulate and update data with temporal constraints in a structured manner. That is, essentially what regular databases are about, except that the items of an RTDB have a *time-value* nature, and as such, their correctness "degrades" with time. For an introductory study on time-value entities that helps understanding this discussion, *see Time-Value Duality* in Chapter 12.

It is as if we were comparing the storage of bricks (non real-time database) with the storage of food (RTDB), for sale. If the bricks are intact they maintain their value practically forever, and we can sell and use them at any time. Food items decay with time, and as time goes by, they are worth less, until eventually becoming useless, if not sold in the meantime.

In complement to RTDBs, *active databases* were born from the need to detect and react to significant changes in the internal state of the database, triggering an action in consequence (Berndtsson and Hansson, 1995). This sequence is called *event-condition-action* or ECA.

13.10.1 Architecture of RTDB Systems

RTDBs are used: for recovery of values needed for a computation or an operation, in bounded time; for computing on sets of values with temporal validity (time-value entities), such as sensor observations; for combining these values with previously stored values in bounded time; for updating values cyclically, with pre-determined periods. A real-time database architecture has mostly the same building blocks as a non real-time one:

- transaction manager
- scheduler

- data manager
- data

The differences lie in the way the blocks work. The semantics of transactions, the rules for scheduling transactions, and the mechanisms for data management are oriented to fulfill time constraints, as well as logical. RTDBs offer transactions that aim at retaining the ACID properties (*see Transactions* in Chapter 8): atomicity; consistency; isolation; durability. Obviously, in a real-time context, this requires re-qualifying some of these definitions with a few constraining assumptions (Ramamritham, 1996b), in the context of what have been called **real-time transactions**. For example, consistency is different in a temporal context: when we abort a transaction so that the database remains logically consistent, it may no longer be temporally consistent, if some items lost their validity because of delay. In essence, an RTDB item is a *representative* ($r(E_i)$) of a *real-time entity* (E_i) (*see Entities and Representatives* in Chapter 12). The logical correctness of a database item depends on the observation originating it being *consistent* (*see Time-Value Duality* in Chapter 12). In temporal correctness terms, real-time databases additionally require the solution of two problems:

- keeping the value of each individual item consistent with its RTe;
- keeping the values of a collection of items mutually consistent.

The first is achieved by maintaining items *temporally consistent*. The second is secured by ensuring that the relevant collection of observations is *mutually consistent*. *How is this checked?* (a) By writing the observation timestamp and the absolute validity interval together with the item when an update is done ($\langle r(E_i), T_i, \mathcal{T}_a \rangle$. Then, when the item is used, T_{now} must be *within* $\mathcal{T}_a$ *from* T_i, for temporal consistency. (b) By checking that all timestamps of the collection of items about to be used fall within the relevant relative validity interval $\mathcal{T}_m$, for mutual consistency. *How is this enforced?* By ensuring that two conditions are simultaneously met: refreshing each item before it loses validity, that is, at most by the end of the absolute validity interval; ensuring that the update instants of all items of a collection always fall within an interval not greater than the relative validity interval.

In conclusion, an RTDB is consistent at time t, if and only if its items are consistent. The RTDB must be updated in accordance to the validity requirements of the several items. Note that enforcing mutual consistency (often called relative consistency in an RTDB context) may imply tightening update intervals for individual items as would be defined for enforcing absolute consistency alone.

13.10.2 Real-Time Transactions

In a client/server style of operation such as provided by RTDBs, there is a problem with guaranteeing all temporal specifications, if no assumptions are made about when and how often requests can be issued by clients to the database. In that sense, RTDBs can be designed having in mind the same classes of real-

time discussed earlier: hard, mission-critical, soft, and even non-real-time, as far as the coverage of those guarantees is concerned. This has implications on the internal organization of the transaction manager and scheduler modules. In functional terms, real-time transactions invoked on RTDBs can also be classified in several types, depending on the action performed:

write-only	an observation is written on the RTDB, possibly overwriting an existing item with a fresh value (e.g., a temperature)
update	an item is read, updated, written again (e.g. event counter)
read-only	an item is read, possibly for being used in computations, and/or to be output to an actuator

Real-time transactions have logical as well as temporal correctness criteria. Let us exemplify the latter. Consider the following RTe characterizations: E_1, E_2 : $\langle$*temperature*,$\mathcal{T}_T = 50\rangle$; E_3, E_4 : $\langle$*pressure*,$\mathcal{T}_P = 10\rangle$ (the absolute validity intervals $\mathcal{T}_T$ and $\mathcal{T}_P$ were computed based on the shortest time needed to attain a variation of ν_T and ν_P respectively, in the value of a temperature and a pressure). Consider further the set $S = \langle\{E_1, E_2, E_3, E_4\}, \mathcal{T}_m = 5\rangle$, for which a relative validity interval $\mathcal{T}_m$ is specified. Database items are specified as $R_i = \langle V_i, T_i, \mathcal{T}_i\rangle$. Now, consider the following situation at instant $T = 1800$ time units, w.r.t. stored items: $R_1 = \langle 147, 1752, 50\rangle$; $R_2 = \langle 162, 1755, 50\rangle$; $R_3 = \langle 1088, 1751, 110\rangle$; $R_4 = \langle 1114, 1750, 110\rangle$. The RTDB is absolutely consistent, because none of the items has lost its validity, and it is mutually (or relatively) consistent, because current timestamps are not separated by more than 5 units of time. Nevertheless, R_1 must be updated until $T = 1802$, in order to remain absolutely consistent. However, at the moment this is done, the set loses its mutual consistency. This shows the interdependence of both kinds of consistency. The remedy lies in bringing the update times of all variables in set S to within $\mathcal{T}_m$, while ensuring that the update period is not greater than the shortest absolute validity interval. When items belong to different sets bound by relative validity specifications, the smallest of the relative validity intervals should be used for the mechanism above.

13.11 QUALITY-OF-SERVICE MODELS

Quality-of-service models are mainly used to support soft or mission-critical real-time applications, when there is a need for ensuring an *end-to-end data flow* with real-time and reliability guarantees during a mission (radar data capture) or a session (remote video rendering), but unlike hard real-time systems:

- these guarantees must be *negotiated* on a need basis, in competition with other flows, in what forms a *contract*;
- what is granted by the infrastructure may be less than what was asked for;
- the contracted guarantees may vary during the mission/session, by initiative of the infrastructure or of the contractor.

The QoS model finds applicability in producer-consumer applications, such as those found in real-time instrumentation or telemetry, and in multimedia

capture and rendering. In particular, the concept of QoS adaptation, which means reviewing the QoS contract as a means to maintain some stability in the coverage of the received versus expected guarantees, has a generic application in the field of mission critical real-time systems. It may form the basis for the formal reasoning about mechanisms such as cost-value functions and operational envelopes (*see* Sections 13.1 and 14.5).

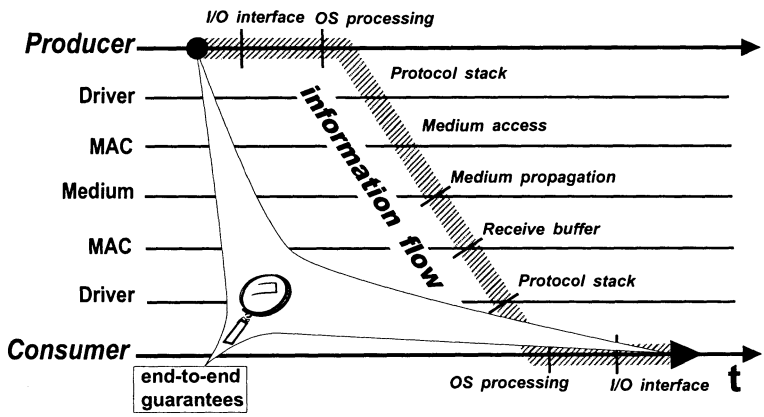

Figure 13.9. The Quality-of-Service Model

The main characteristics of the QoS model in distributed settings (Aurrecoechea et al., 1998) are suggested in Figure 13.9:

- end-to-end establishment of QoS guarantees for a given flow– timeliness, performance, reliability;
- piecewise enforcement of guarantees throughout the path– device, OS and communications driver scheduling, latency of access and medium throughput, receive buffer handling, flow control and error processing;
- maintenance of guarantees– surveillance of the flow QoS.

The steps towards implementing a QoS model concern QoS specification and management, and have been typified in (Hutchison et al., 1994):

- specification and mapping
- negotiation and resource allocation
- admission control
- maintenance, monitoring and policing
- renegotiation and adaptation

QoS specifications are made in terms of parameters falling into different classes or categories, such as timeliness, or volume, or reliability. Within each class, QoS is specified with metrics adequate to the magnitudes concerned. For example: `latency`, `tightness`, or `jitter` specify *timeliness*; whereas `throughput` or `BurstLength` specify *volume*; and `BER` (bit error rate) or `MTBF` (mean time between failures), or `MTTR` (mean time to repair) specify *reliability*.

A **QoS specification** consists of a list of parameters and values and intervals of validity for them, indicating the conditions, the goals and the importance of each parameter P. A possible specification may be like:

Sampling Period (TS)- the interval over which the value of the parameter is acquired. For example, if P is a round-trip time, a distribution function (pdf) is computed with the samples.

Threshold (TH)- the upper or lower acceptable bound on P. A typical upper bound is for a round-trip duration. A typical lower bound is for a bandwidth.

Weight (WT)- a measure of the relative importance of parameter P. A value of 0 would mean that the parameter would not be taken into account.

A specification is made in abstract terms, which must be mapped on the infrastructure. When the *mapping* takes place, the specification may unfold onto several more detailed sub-specifications. For example, a `BER` (bit error rate) requires error counters and local interval clocks (timers). A `latency` in message delivery is divided in the partial delays across the message path.

Through *negotiation* a *contract* is established with the infrastructure, and thus the several components involved, in order that there is a guarantee that the specification will be respected. After a service QoS specification is presented, components declare the best QoS they can supply for the desired service. Depending on that response, the management 'contracts' a given level of QoS or instead reports the end-user (e.g., an application) that the desired QoS cannot be obtained.

After negotiation, the components try to perform *resource allocation* in order to secure the promised support. For example, to reserve a certain amount of buffers, or to allocate a given level of priority to a given flow, or to redefine the O.S. schedule of processes. A successful negotiation should be seen as a promise rather than a firm commitment, which only takes place when *admission control* is passed, that is, when the system remains stable after the allocation of resources to the pending request. Namely, the QoS of other services should not be affected.

Parameters may have an associated assumed *coverage*, that establishes the desired level of guarantee for the service. This is for most applications a *statistical* rather than an *instantaneous* problem. In other words, it is necessary that the QoS holds over an integral of time, regardless of small glitches. In real-time language, this could be described in terms of mission-critical or soft real-time behavior. This means that maintaining QoS is subject to typical statistical control problems such as inertia, histeresis and instability.

A special kind of timing failure detectors should be used that understand this problem. We call them **QoS failure detectors**, QoS-FD, and they are designed to evaluate a number of relevant operational parameters, over an interval of time (Veríssimo and Raynal, 2000). The detector is configured through the QoS specification issued by the application, which instructs it how to perform *QoS tests*. The detector tests the end-to-end QoS seen in the flow from the

producer to all consumers involved in a distributed application. The value V of each parameter P of the QoS specification is evaluated by the QoS-FD over the sampling period TS. The sampled values are tested against the threshold TH. A variable *threshold exceeded*, TE, accounts for the percentage of the sampling periods of P where it fell beyond the bound TH during TS. Besides providing analog information about individual parameters, the Qos-FD may provide a boolean notion of *QoS failure* at a given node, akin to what is provided by crash failure detectors. This indication may be constructed by first computing a consolidated analog indicator, for each node, which is an average of the TE's of all parameters P, weighted by the respective WT variables. Then, define a global threshold for this analog indicator, which when passed generates a boolean failure indication. These individual failure indications can be consolidated in vectors, which form the opinion of one node about the others, or in a matrix, if nodes exchange their vector information, as crash failure detectors do (Veríssimo and Raynal, 2000). The specification may also contain the indication of the exception processing to be taken when the specified level of service is not attained. When the QoS-FD reports a failure, it can forward useful information to the application or middleware support, such as, for example, the lateness degree of timing failures, or the current point of the desired timeliness bound in the variable's pdf (probability density function). This information is precious for the higher layers to decide what to do. Depending on the system's ability to handle the situation, reaction to a QoS failure may have several outcomes:

termination- the activity cannot withstand a lower QoS; we say that the activity is not adaptable;

adaptation- the activity may reduce its requirements on the system, expecting that the offered QoS recovers (e.g., by reducing the refreshing rate of the scene in a teleconferencing application, or by reducing the refreshing rate of a radar trace); alternatively, the activity may live with the degraded QoS, by resorting to adaptation mechanisms using the own application heuristics (e.g., by passing from color to B&W rendering in the same teleconferencing application, or by using less sophisticated target trajectory prediction algorithms);

renegotiation- the activity triggers a renegotiation procedure, trying to obtain a new but satisfactory balance on QoS, normally by reducing its demand on the affected dimension but trying to recover in an alternative one (e.g., if bandwidth is stressed but there is spare computing power, augmenting the compression factor of the information transmitted from the remote sites relieves communication at the cost of more MIPS, but information significance— such as image quality, or sensor data accuracy— is maintained)

13.12 SUMMARY AND FURTHER READING

In this chapter, we discussed the main models of real-time distributed computing. The first part of the chapter was concerned with providing the system architect with a comprehensive view about the available architectural

options, frameworks and strategies. The rest of the chapter was devoted to models of distributed real-time computing addressing concrete functional and non-functional aspects: event-triggered and time-triggered operation; communication; control; databases; and quality-of-service. We advise the following works as further reading. A detailed study of real-time programming issues is done in (Burns and Wellings, 1996). Specifically on real-time object computing, we suggest (Kaiser and Livani, 1998; Yau and Karim, 2000; Becker et al., 2000; Siqueira and Cahill, 2000), or (Jensen, 2000; Kalogeraki et al., 2000).

On the formal specification and verification of the timeliness aspects of real-time designs, we suggest (Koymans, 1990; Lamport, 1994; Sinha and Suri, 1999; Graw et al., 2000; Bozga et al., 1998). The relation between order and synchronism specifications is explained in (Veríssimo, 1996). Some researchers have studied the partial synchrony of systems (Dolev et al., 1983; Dwork et al., 1988) under a time-free perspective. For further study on timed partially synchronous models see (Cristian and Fetzer, 1998; Veríssimo and Raynal, 2000). On programming with partial synchrony see (Almeida and Veríssimo, 1996; Fetzer and Cristian, 1997a; Veríssimo et al., 2000).

For further study of real-time networks see: (Tanenbaum, 1996; Halsall, 1994) about most common standard LANs; or (Pimentel, 1990; Pleinevaux and Decotignie, 1988) for field buses. Practical details on the definition and use of abstract network properties in the construction of reliable real-time communication protocols are given in (Veríssimo and Marques, 1990; Rufino et al., 1998). Studies about network scheduling can further be found in (Tindell and Burns, 1994; Tindell et al., 1994; Zuberi and Shin, 1995), or (Tindell et al., 1995; Lehoczky, 1998; Tovar et al., 1999). On medium reliability of real-time networks, see (Veríssimo, 1988; Rufino et al., 1999) for buses, or (FDDI, 1986; Rom, 1988) for rings. Likewise, a few space- and time-redundant approaches for fault tolerance in real-time communication are described in (Babaoğlu et al., 1986; Zheng and Shin, 1992; Cristian, 1990; Veríssimo and Marques, 1990; Cristian et al., 1985). Studies and measurements on the performability and inaccessibility of networks are described in (Meyer et al., 1989; Rufino and Veríssimo, 1992; Prodromides and Sanders, 1993; Veríssimo et al., 1997).

Concepts and design of distributed computer control systems are discussed in (Steusloff, 1981; Fisher, 1990; Bauer et al., 1991; Wikander, 1998).

A discussion on the issue of fulfilling temporal constraints on RTDBs can be found in (Ramamritham, 1995; Song and Liu, 1992). For further study on real-time and active databases, the reader is referred to (Ramamritham, 1996b; Berndtsson and Hansson, 1995; Purimetla et al., 1995; Korth et al., 1996; Ozden et al., 1996).

In (Jeffay, 1993; Rajkumar et al., 1995; Kaiser and Mock, 1999) the real-time producer-consumer model is discussed. For further study on QoS architectures, the reader is pointed to (Abdelzaher and Shin, 1998; Leslie et al., 1996; Volg et al., 1996). Discussions on the construction of adaptive applications are made in (Cosquer et al., 1996; Friday et al., 1999).

14 DISTRIBUTED REAL-TIME SYSTEMS AND PLATFORMS

This chapter gives examples of systems and platforms for real-time computing, consolidating the matters discussed in the previous chapters. Namely, it discusses: operating systems; real-time LANs and field buses; time services; embedded systems; dynamic mission-critical systems; real-time over the Internet. In each section, we will mention several examples in a summarized form, and then will describe one or two the most relevant in detail. Table 14.1 at the end of the chapter gives a few URL pointers to where information about most of these systems can be found. The table also points to the IETF Request for Comments site, where any RFCs cited can also be found.

14.1 OPERATING SYSTEMS

Several real-time operating systems have been deployed in the recent years. Many assume the form of a *real-time multitasking executive*, a simplified and highly-modular kernel, normally working with preemptive scheduling based on priorities. Most of these systems use fixed priorities. Concurrency is normally based on synchronization primitives offered by the system support. Typical functions offered through systems calls are: process management (creation, deletion, blocking, suspension, scheduling); memory management; synchronization and inter-process communication (queues, semaphores, mailboxes). Interrupt management is often a sophisticated two-level structure: front-end interrupt handler for immediate processing of an interrupt event; and interrupt

P. Verissimo et al. *Distributed Systems for System Architects*

task, for more complete processing, scheduled in competition with all others, with a priority matched with the interrupt level. Amongst the examples of such kernels, one can enumerate: RTEMS, VxWorks, VRTX32; OS9. Some of these solutions have fully-fledged user-oriented subsystems, sometimes offering a UNIX-like interface along with the real-time functionality, such as QNX or LynxOS.

There are a number of experimental research real-time operating systems, such as RT-Mach and Spring. The Real-Time Mach (Tokuda et al., 1990) incorporated ideas from a previous project, the ARTS distributed real-time operating system. ARTS is an object-oriented system. A real-time object in the ARTS context is associated a maximum termination time (time fence), and an exception handler, for when the fence is jumped over. RT-Mach is called a resource kernel, i.e. one providing resource-centric services that can be used to satisfy end-to-end QoS requirements. These are normally handled by a QoS manager sitting on top of RT-Mach, which can make adaptive adjustments to resources allocated to applications.

The Spring kernel (Stankovic and Ramamritham, 1991) is the operating system of an experimental distributed real-time system addressing mission-critical and soft real-time applications. The architecture comprises a network of multiprocessor nodes (SpringNet). All system calls have a bounded WCET, and tasks perform resource reservation before execution, so that a task, when it executes, has a predictable WCET. Spring schedules according to a combined notion of timeliness and importance: critical, essential and unessential tasks. Spring gives a-priori guarantees to the critical tasks, while dynamically guaranteeing deadlines of the other arriving tasks.

Other operating systems are not made from scratch, but are rather evolutions of standard ones, such as the several variations of real-time UNIX (e.g., Solaris RT), or the real-time Linux (RTLinux). These variants modify UNIX in areas such as kernel scheduling (turned preemptive and re-entrant), IPC (improved synchronization), memory and disk management (swap prevention), and I/O subsystem, in order to achieve timeliness guarantees from an otherwise time-sharing system. Next, we analyze RTLinux with more detail.

14.1.1 Real-Time Linux

RTLinux was developed in the New Mexico Institute of Mining and Technology, USA, and presented in (Barabanov and Yodaiken, 1997). It is designed as a real-time kernel on the bare machine, on top of which several real-time and non real-time tasks may run. Linux itself runs as one of the latter kind. Linux runs at a non real-time level, with the lowest priority, and it can be preempted at any time by higher-priority tasks. The basic RTLinux scheduler is preemptive amongst tasks with fixed priorities. Alternative EDF and rate-monotonic schedulers are also available. It handles sporadic as well as periodic tasks. RTLinux provides low-level task creation, interrupt handlers, and IPC through shared memory and queues, for communication between interrupt han-

dlers and tasks, and Linux processes. RTLinux recursively uses Linux services when appropriate.

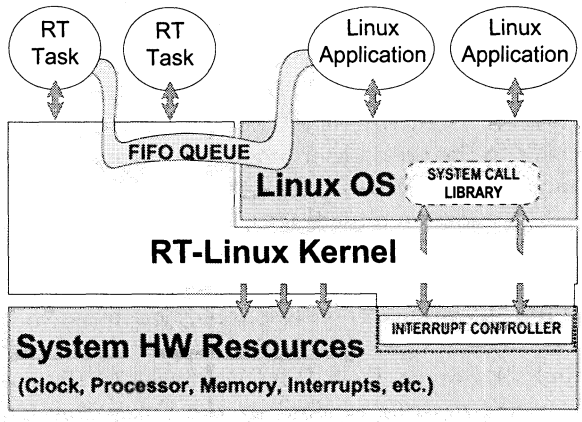

Figure 14.1. RTLinux Architecture

The RTLinux offers simple primitives. The `rt_task_init` primitive creates and launches a task, with a given priority. A task may be rendered periodic, with a pre-defined period, through `rt_task_make_periodic`. The `rt_task_wait` puts a task to sleep. IPC between RTLinux tasks and Linux processes takes place through FIFO queues, created by `rtf_create`. Primitives `rtf_put` and `rtf_get` enqueue and dequeue data respectively. However, since real-time tasks can communicate with Linux processes, it enjoys all the functionality of the latter for the (non-real-time) activities concerned with person-machine interfacing, storage, Internet access, etc. However, all communication is non-blocking, so that RTLinux is never delayed by synchronization or resource contention with a non real-time (low priority) process. RTLinux traps all external system interrupts and enable/disable interrupt instructions, so that Linux cannot disturb the real-time guarantees, for example by disabling interrupts (which it does very often).

14.2 REAL-TIME LANS AND FIELD BUSES

We have discussed the role of LANs and field buses in distributed real-time architectures, in Section 11, and addressed some LAN and field bus scheduling mechanisms in Section 12.7. The Token Bus LAN has been the most promising *real-time LAN*. Its timed-token scheduling alows the definition of one hard real-time and several soft real-time latency classes. However, the tuning of timers and other parameters is a complex operation, requiring considerable expertise. If not done properly, it can significantly degrade the operation of the network, and this was probably one of the reasons for the decline in the use of Token Bus. Several attempts have been made to achieve real-time operation on Ethernet. Earlier on, a modified Ethernet has been proposed, DCR Eth-

ernet, featuring deterministic collision resolution (Le Lann and Rivière, 1993). Recent full-duplex switched Ethernet networks open new perspectives for real-time operation, since collisions are avoided, and thus real-time communication protocols can be designed for this new physical reality of Ethernet.

Field buses are assuming increasing importance in distributed real-time systems. Early field buses such as MIL-STD (MIL-STD-1553B, 1988) or FIP (FIP, 1990) had a centralized structure. FIP (Factory Instrumentation Protocol) is a field bus oriented to the centralized produced-consumer paradigm, currently a european standard (EN-50170-3,1996) called WorldFIP. WorldFIP performs transactions of data buffers and messages, addressed point-to-point, in multicast or in broadcast. Data is limited to 128 bytes per frame. In fact, the initial purpose was to extend I/O cabling from a controller unit (the master node). It was not until the appearance of decentralized field buses such as Profibus or CAN that they started emerging as potential support for embedded distributed systems. Profibus, or Process Field Bus (Profibus, 1991) is a field bus with a MAC (medium access control) inspired by the Token Bus LAN, currently a european standard (EN-50170-2,1996). Although Profibus has a decentralized nature, it recognizes master and slave stations. Master stations are normally controller nodes, the ones that compete for bus access. Slave stations are passive, normally corresponding to I/O nodes.

14.2.1 CAN – Controller Area Network

CAN is inspired by the Ethernet, and is in fact a *carrier sense multi-access with deterministic collision resolution* network. Arbitration of medium access is done by direct bit-by-bit comparison of the 11-bit frame identifiers of the transmitting stations (*see* Figure 14.2). Bits can be *recessive* (zero) or *dominant* (one), and if a recessive and a dominant bit are transmitted simultaneously on the bus, the latter imposes itself to the former on the bus: while transmitting a frame identifier, each station monitors the bus; if it transmits a recessive bit, and a dominant bit is monitored, the station gives up transmitting and starts to receive incoming data; the station transmitting the lowest identifier goes through and gets the bus. Note that each identifier defines a *priority* (in inverse order of the identifier value), and frame transmission scheduling in CAN is thus by highest priority, on a per station basis.

Bit-by-bit arbitration works because the network works in quasi-stationary mode, that is, the signal phase is the same throughout the bus length. However, this requires a small length/rate product. The bus length and data rate are related, and typically, we have 40 meter @ 1Mbps, or 100 meter @ 50Kbps. Data is limited to 8 bytes per frame. CAN performs detection and recovers from a number of errors, such as bit errors, CRC errors, and missing acknowledgments. When an error in transmission is detected, an error flag is broadcast by recipients, the frame is re-transmitted immediately and the previous transmission discarded. This achieves an almost atomic broadcast behavior, except if very rare failure scenarios occur.

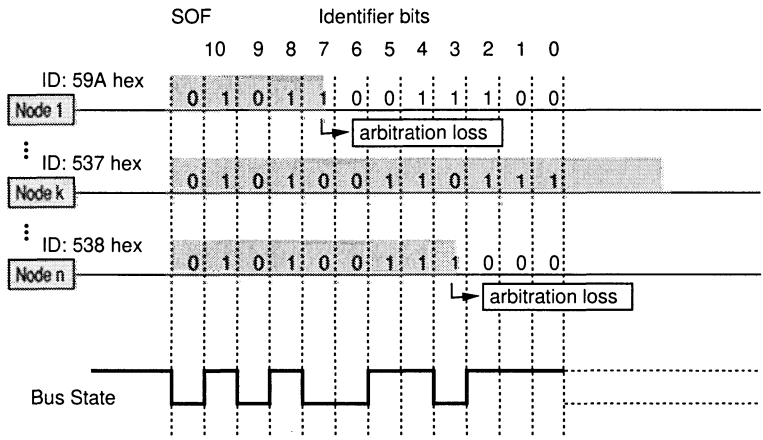

Figure 14.2. CAN Arbitration

14.3 TIME SERVICES

Systems have a basic local clock service, offered through a `getTime` primitive. This monotonic up-counter allows measuring interval durations. Most systems offer a timer or alarm service, which counts down from an initial value, normally provided through a `startTimer(timeout)` primitive. The timer can be canceled before it expires (`killTimer`), otherwise it raises an alarm. In some systems, the timer launch event can be linked to the automatic execution of a function in response to the timeout (`startTimer(timeout,function)`). This is the most precise way of taking an action upon a timeout, since there are no spurious delays in between.

Increasingly more often, local clocks in a distributed system are synchronized (*see Clock Synchronization* in Chapter 12), which gives the `getTime` primitive a global meaning: it returns the time at any site in the system, with a difference bounded by precision π. When clocks are externally synchronized to a standard source, such as UTC, `getTime` returns a time that is comparable with time in other external sources (e.g., our wrist watch), with a difference bounded by accuracy α. This also guarantees precision, since $\pi = 2\alpha$. A facility common in real-time operating systems is the ability to schedule the execution of functions at a given future time, such as `execAt(hour,function)`.

For large-scale systems, several global time services have been deployed through the recent years. The Berkeley TEMPO system (Gusella and Zatti, 1989) was developed in the context of the UNIX Berkeley effort, for internal synchronization. TEMPO is based on a hybrid agreement master-slave algorithm. An external hybrid algorithm combining master-slave round-trip and averaging agreement is presented (Fetzer and Cristian, 1997b). CESIUMSPRAY is another system that provides external synchronization, based on a hybrid agreement master-slave synchronization approach (Veríssimo et al., 1997). The master is the GPS NavStar (Parkinson and Gilbert, 1983) system of satel-

lites, which disseminates its time reference through the slave nodes with GPS receivers on ground, at least one per LAN. All nodes in each LAN participate in an averaging-non-averaging clock synchronization algorithm that injects the GPS time in all other clocks.

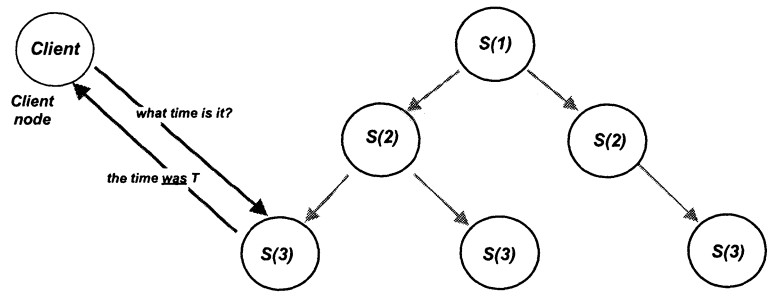

Figure 14.3. The NTP System with Server Hierarchy ($S(i)$ - Stratum i), and detail of Client Synchronization from a Remote Server

14.3.1 Network Time Protocol

The Network Time Protocol (RFC 1119) is the most widely deployed time service and protocol (Mills, 1991). It has been serving as the time reference for Internet nodes. The NTP Time Synchronization Service is a complex hybrid structure, inspired by Cristian's master-slave round-trip synchronization protocol (*see* Figure 12.16 in Chapter 12), which combines master-slave dissemination and round-trip with AVG agreement for synchronization of the server tree, with master-slave round-trip for client synchronization. The system is implemented as a hierarchical structure of masters spread across the Internet, as depicted in Figure 14.3. The masters supply UTC time, and synchronize the slaves (clients) with an accuracy that is probabilistic and depends on the stability of transmission times on the Internet, to and from the master. The NTP protocol currently offers some protection against spoofing of time datagrams, through authentication. It also offers interfaces for correcting the rate of drift of hardware clocks, if such mechanisms exist on the client or server computers.

Master clock servers are organized in descending order of intrinsic accuracy in the hierarchy. The **stratum** 1 servers are always directly connected to a UTC source of known accuracy, such as a GPS receiver. The *strata* below, 2 to n, synchronize themselves to servers of the immediately higher stratum. The inter-server synchronization can be round-trip, but when good quality accuracy is desired, servers can perform *symmetric* message exchanges, whereby servers of the same stratum or adjoining strata improve their synchronization through agreement-based adjustments. A third and simple scheme is available, obtaining good accuracies when servers are connected through a high-speed, low-delay link, by simply *multicasting* the time and doing a fixed correction on the (small) delay. At the time of this writing, the NTP system is reported to

have around 175,000 active hosts, of which over two hundred servers belong to the public infrastructure of NTP on the Internet (Strata 1 and 2), and has been reported to grant clients average accuracies in the order of the several tens of milliseconds (Minar, 1999).

The Distributed Time Service (DTS) of the DCE platform (*see DCE* in Chapter 4) is inspired by the same philosophy of the NTP system. The DTS is both available for users and for internal use of other services. For example, the protocols of the Kerberos authentication service of DCE rely on the existence of global time. The DTS format is : `1999-12-31-23:59:30.456+01:00I000.125` The first field indicates date, followed by the time, down to the millisecond (`23:59:30.456`). The *time differential factor* field (`+01:00`) indicates the deviation of the relevant time zone from the Greenwich meridian zone (longitude zero). DTS (such as NTP) is capable of estimating how accurate it is running, this is given by the *inaccuracy term*, `I000.125`, which would indicate in this case a worst-case accuracy of ± 125 milliseconds.

14.4 EMBEDDED SYSTEMS

Embedded systems normally have a static character, are small-scale, and are used for hard real-time applications. They are often time-triggered.

A number of experimental systems have been developed in the past few years. Maruti-II (Saksena et al., 1995) is a distributed time-triggered real-time system. The computational model of Maruti is organized around elementary programming units that are connected to one another in acyclical unidirectional graphs, through a configuration language, forming execution threads. Schedulability of the threads is analyzed off-line. MAFT (Kieckhafer et al., 1988) and FTPP (Harper et al., 1988) are fault-tolerant real-time systems specially designed for safety critical embedded applications. They are based on communication systems resilient to arbitrary failures while securing timeliness requirements, and thus take a fully synchronous byzantine agreement approach and cyclic scheduling. GUARDS (Wellings et al., 1998; Powell et al., 1999) is a generic fault-tolerant computer architecture based on commercial off-the-shelf (COTS) hardware and software components. The architecture is configurable in two axes: in terms of different fault-tolerance strategies based on modular physical redundancy; and in terms of different integrity levels, allowing the co-existence of sub-systems of different criticality.

14.4.1 Mechatronic Architectures

Mechatronics is the integration of electronic and computational technologies on devices that were originally mechanical. Since they contribute to component integration, they are often used in small embedded systems. Examples are: the control devices of an automatic photographic or video camera; active car breaking or suspension devices; "intelligent actuators", such as robot manipulators. The advantages of mechatronic are manifold:

- reliability– integration reduces the weak points of interconnections

- simplicity– less moving/mechanical parts
- economy– lower price
- miniaturization– computer/electronics replacing cumbersome apparatuses;
- quality– better parametric quality, such as precision, responsiveness, etc.

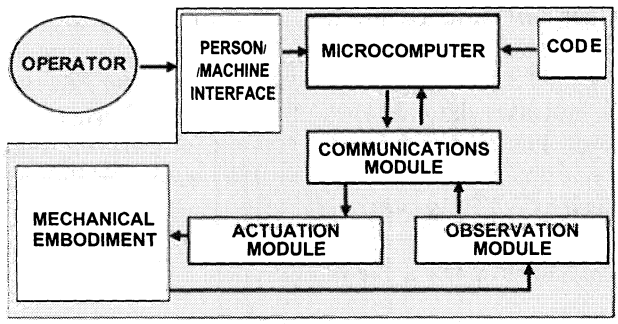

Figure 14.4. Mechatronic Architecture

The architecture of a typical mechatronic system is depicted in Figure 14.4. Note the integration between the computational, the I/O and the physical parts. The mechanical embodiment comprises all the mechanical parts, functional and encapsulation, and is connected to the control part through sensors and actuators, organized in two modules (observation and actuation). The communications module implements the abstraction of a miniature field bus interconnecting the sensors and actuators to the microcomputer assembly. Finally, the microcomputer can be hooked to a person-machine interface, when one exists.

14.4.2 The MARS System

MARS (MAintainable Real-time System) (Kopetz et al., 1989a) is an experimental distributed fault-tolerant hard real-time system for critical applications. Recently, an industrial prototype version called TTA (Time Triggered Architecture) has been implemented, with functionality incorporated in silicon with a view of applications in mass production areas such as automotive electronics.

The MARS architecture is oriented to testability of the proper operation of the system in the design phase. In order to help achieve this objective, MARS is based on the principle of resource adequacy, and follows a time-triggered (TT) activation discipline. Communication is performed by a specialized TDMA protocol called TTP (Time Triggered Protocol) (Kopetz and Grunsteidl, 1993). I/O pre-processors ensure that input events are transformed into state messages that are forwarded to the computing elements and instantiated as fresh copies of system state, overwriting previous ones. Periodic tasks then act at pre-defined moments on this newly updated state, proceeding without any external communication or synchronization until the production of output state

messages to tasks in other components, or command messages to the actuators. MARS follows a systematic software design methodology based on static (off-line) scheduling. Since there are no synchronization points, the maximum execution time of the tasks is easily determined by off-line code analysis.

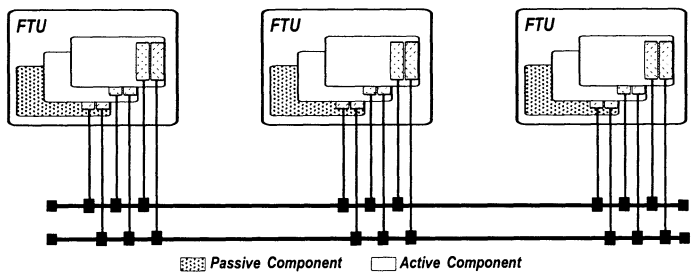

Figure 14.5. The MARS Architecture

The MARS architecture, shown in Figure 14.5, is organized around the cluster concept. A cluster can be decomposed into a set of Fault Tolerant Units (FTUs) interconnected by a replicated real-time communication channel that runs the TTP protocol. A Fault Tolerant Unit is normally composed of two replicated computer elements operating in active redundancy under the fail-silent model, backed-up by a shadow component: as long as any one of the components of a FTU is operational, the FTU is considered operational; the shadow acts as a hot-swap spare. A fault-tolerant clock synchronization service and a fault-tolerant membership service are integrated with the TTP protocol.

14.5 DYNAMIC SYSTEMS

Dynamic systems normally reflect the need for versatility and adaptability of large and complex real-time systems. Scheduling is dynamic, and the systems normally address mission-critical or soft real-time applications. They are often event-triggered.

HARTS (Shin, 1991) is an experimental real-time distributed system, based on a multicomputer cluster, interconnected by point-to-point links. Its operating system HARTOS allows hard and soft operation to coexist. HARTOS schedules groups of periodic tasks with the same priority, and schedules preemptively groups of different priorities. Processes can change priorities dynamically. HARTOS relies on reliable real-time communication and global time services for interacting with the other nodes. The ALPHA system (Jensen and Northcutt, 1990) also features dynamic priorities based on the *time-utility* principle. ALPHA is specially devoted to mission-critical systems, where scheduling decisions must comply with the uncertainty about the environment, yet provide the best possible timeliness assurances.

14.5.1 The Advanced Automation System

The Advanced Automation System (AAS) was developed by IBM for air traffic control (Cristian et al., 1996). The system architecture, comprises several modules or clusters, whose components are networked by replicated Token-Ring LANs. Figure 14.6 depicts the structure of an Area Control Computer Complex (ACCC), which supports the air traffic control functions. Clusters are interconnected to each other through a replicated backbone Token Ring, to each they connect by a Communication Gateway Subsystem, composed of replicated bridges. Radar sensor data is time-sensitive and must get to the central processors with real-time requirements. These must process the information, display it and help the operator make decisions. All these operations are time-critical.

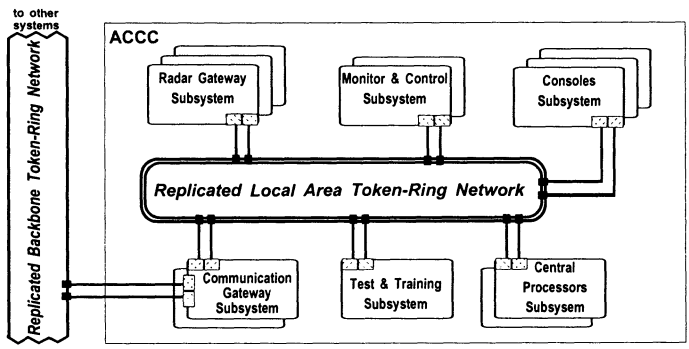

Figure 14.6. Advanced Automation System – Snapshot of a Cluster Network

Besides the functional modules, several services help achieve non-functional properties of AAS: topology and route managers, maintaining an accurate image of the current state of the network and its links; service availability manager, controlling the availability of the crucial AAS services, by means of failure detection, promoting passive replicas, re-inserting recovered units, etc.; global availability manager, controlling the state of all hardware and software components of the system. These services depend on a suite of lower-level fault-tolerant services: atomic broadcast; group membership; and synchronized clocks. The AAS protocols are clock-driven, and triggered either by time or events. The atomic broadcast and group membership protocols are of the Δ-protocol class, where messages are delivered after a Δ time on the local clock. The communication system is diffusion based, with space channel redundancy. Different levels of channel redundancy (up to four) are used in the different parts of AAS.

14.5.2 The Delta4-XPA System

Delta-4 is a system based on distributed fault tolerance (Powell, 1991). It has a modular architecture relying on software based fault tolerance, where components interact via reliable group communication and replication management

protocols (*see Distributed Fault-Tolerant Systems* in Chapter 9). Delta4-XPA, the Extra Performance Architecture of Delta-4, also described in (Powell, 1991), addresses the realm of mission-critical applications. XPA aims at implementing hard and soft real-time designs under an unbounded-demand perspective, that is, recognizing that a complete definition of the operational envelope is not possible, or that the definition imposed by design has a limited coverage. This requires the architecture to have some form of graceful degradation ability. In other words, while the system would normally act as a hard real-time one in the presence of the "foreseen" environmental and computational constraints, it would adapt to "unforeseen" situations — working in modes that are progressively less effective, precise, reliable, etc. — without abruptly falling apart. XPA can thus be described as a *hard real-time system with graceful degradation.*

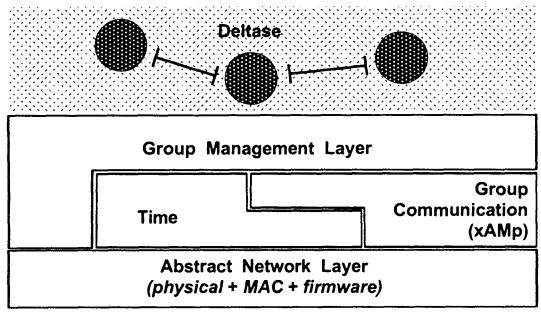

Figure 14.7. The Delta4-XPA Architecture

The architecture of XPA, depicted in Figure 14.7, comprises nodes interconnected through a LAN-based real-time communication subsystem, providing a suite of reliable group communication primitives called xAMp (Rodrigues and Veríssimo, 1992), and a fault-tolerant global timebase of synchronized clocks (Veríssimo and Rodrigues, 1992). The main policy for handling replication at the group management layer is *leader-follower*, a semi-active replication management mechanism that allows consistent replica preemption. XPA schedules under a combined notion of *importance*, the measure of the consequences of a computation not meeting its deadline, and *urgency*, the notion of approaching deadline. Scheduling is thus *priority-based* among classes of tasks ranked by importance, and then *earliest-deadline-first* inside each class.

14.6 REAL-TIME OVER THE INTERNET

The emergence of the multimedia era in Internet has encouraged a lot of research with the objective of achieving real-time communication guarantees on that uncertain infrastructure. Multimedia is intimately related to dissemination and conferencing, and as such the IP multicast subset of the Internet, the Mbone, was the field for early experiments on real-time over the Internet. Real-Time over the Internet is anyway associated with the notion of QoS specification, and of resource reservation. Ferrari does an extensive analysis of ways

of specifying client requirements for real-time communication quality of service, on open networks (RFC1193).

Efficient stream protocols appeared, such as ST2 (RFC1819), aiming at achieving end-to-end real-time communication through resource reservation. ST2 works at the same level of, and in complement to IP, but is connection-oriented. ST2 application areas are real-time transport of multimedia data, such as: digital audio and video packet streams, and distributed simulation and gaming. The ST2 communication model includes: a flow specification to express user real-time requirements; a setup protocol to establish real-time streams based on the flow specification; a data transfer protocol for the transmission of real-time data over those streams. ST2 is composed of two protocols: ST for the data transport; and SCMP, the Stream Control Message Protocol, for all control functions. Though ST2 supports multicast, it does not support the notion of group communication and management, which helps the construction of multiparty multimedia applications, such as in (Panzieri and Roccetti, 2000).

The Resource ReSerVation Protocol, RSVP, is a resource reservation setup protocol for Internet (RFC2205). RSVP is an Internet control protocol that provides receiver-initiated setup of resource reservations for multicast or unicast data flows, to be later used by transport protocols such as RTP. The RSVP protocol is used to ensure a specified QoS over a unidirectional data flow: it lets the sending host request the necessary resources, and it lets routers propagate the QoS requests along the route to the destination. QoS reservation is made by the recipients, and is propagated in the reverse data path back to the sender, which consolidates the requests of the several recipients.

RTP, the Real-Time Transport Protocol (RFC1889), provides end-to-end network transport of real-time data. It is accompanied by a control protocol (RTCP) to allow monitoring of the data delivery. RTP and RTCP are designed to be independent of the underlying transport and network layers. Applications typically run RTP on top of UDP. RTP can rely on low-level protocols such as ST2 to provide timeliness guarantees. RTP is complemented with companion standards defining how to specify the profiles and encoding rules for the several types of payload real-time data. With regard to security of the data flows, namely its confidentiality, RTP will use underlying services such as IPsec (*see Extranets and Virtual Private Networks* in Chapter 19).

14.7 SUMMARY AND FURTHER READING

This chapter gave examples of systems and platforms for distributed real-time computing. The objective of the chapter was to relate the notions learned throughout this Part with existing products and systems. We reviewed operating systems, real-time LANs and field buses, time services, embedded systems, and dynamic mission-critical systems. We finalized with an overview of protocols related with real-time on the Internet.

For further reading, Table 14.1 gives a few pointers to information about some of the systems described in this chapter. Detailed insight on the practical

design and SW/HW integration of real-time systems is given in (Laplante, 1997). Thorough discussions on real-time UNIX and Linux design issues are done in (Furht et al., 1991) and in the RTLinux links of Table 14.1. Real-time CORBA is an emerging OMG technology at the time of this writing, comprising: fixed priority scheduling, control over ORB resources for end-to-end predictability, and flexible communications. Real-time Java is receiving a great deal of interest, because of its potential for embedded applications. POSIX defines an important standard for a UNIX-based real-time programming API (POSIX, 1995).

On CAN scheduling, see further (Tindell and Burns, 1994; Tindell et al., 1994). On subtle CAN failure modes and CAN atomic broadcast, see (Rufino et al., 1998). CANopen is a set of technologies and recommendations for interoperability of CAN designs.

A recent survey on the NTP Time Service gives useful details on its current set-up and performance (Minar, 1999). Details on DCE-DTS can be found in (Lockhart Jr., 1994). Further material on AAS, Delta4-XPA, and MARS/TTA can be found in (Cristian, 1994; Speirs and Barrett, 1989; Kopetz, 1997). Amongst other interesting practical real-time systems we suggest looking at (Hachiga, 1992; Hedenetz, 1998; Heiner and Thurner, 1998).

Mechatronics is treated comprehensively in (Bradley et al., 1991; Wikander, 1998). Material on distributed industrial systems, such as Computer Integrated Manufacturing (CIM) and Supervisory Control and Acquisition (SCADA), can be found in (Beekmann, 1989; Veríssimo et al., 1996), and some of the pointers in Table 14.1. In (MAP, 1985; CNMA, 1993; MMS, 1990; ISODE, 1993) the main relevant standards are described.

Table 14.1. Pointers to Information about Real-Time Systems and Platforms

Sys. Class	*System*	*Pointers*
RFCs	(IETF)	www.rfc-editor.org/
OMG	(RT-CORBA)	www.omg.org
RTJ	(Sun RT-JAVA)	www.rtj.org
JC-RTJ	(J-Consort.)	www.j-consortium.org
RT-INFO	(RT Encyclopaedia)	www.realtime-info.be/encyc
IEEE-CS	(RT Systems)	cs-www.bu.edu/pub/ieee-rts
On Time		www.britannica.com/clockworks
		www.ubr.com/clocks
		www.newscientist.com/nsplus/insight/time
and		www.real-time.org
		tycho.usno.navy.mil/ctime.html
Real-Time		www.auto.tuwien.ac.at/Projects/SynUTC/time.html
	RTEMS	www.rtems.com
Real-Time	**RTLinux**	www.realtimelinux.org
Executives		www.rtlinux.org
and O.S.'s		www.realtimelinux.org/CRAN
		www.aero.polimi.it/projects/rtai
(experim.)		www.ittc.ukans.edu/kurt
	Alpha	www.realtime-os.com/alpha.html
	RT-Mach	www.cs.cmu.edu/afs/cs/project/art-6/www
	Spring	www-ccs.cs.umass.edu/rts/spring.html
Real-Time	**VxWorks**	www.vxworks.com
Executives	**QNX**	www.qnx.com
and O.S.'s	**LynxOS**	www.lynx.com
(commerc.)	**VRTX32**	www.mentorg.com/embedded/vrtxos
	OS-9	www.microware.com
	GPIB	www.ni.com/gpib
Real-Time	**PROFIBUS**	www.profibus.com
Networks	**WorldFIP**	www.worldfip.org
and	**CAN**	www.canopen.com
F. Buses		www.ems-wuensche.com
	XTP	www.ca.sandia.gov/xtp/forum.html
		dancer.ca.sandia.gov/pub/xtp4.0
	Time Services	tycho.usno.navy.mil/ctime.html
		www.boulder.nist.gov/timefreq
	Time Sync SW	www.eecis.udel.edu/~ntp/software.html
Time		www.ubr.com/clocks/timesw/timesw.html
and	**Time Sync HW**	www.eecis.udel.edu/~ntp/hardware.html
Clock	**NTP**	www.eecis.udel.edu/~ntp
Services	NTP CookBook	www.umich.edu/~rsug/services/ntp.html
	NTP Public Servers	www.eecis.udel.edu/~mills/ntp/servers.htm
	GPS	www.gpsworld.com
		www.gpsy.com/gpsinfo
RT DB's	**RTDB**	www.eng.uci.edu/ece/rtdb/rtdb.html
	Maruti-II	www.cs.umd.edu/projects/maruti
Real-Time	**HARTS**	www.eecs.umich.edu/RTCL/harts
Platforms	**GUARDS**	www.cs.york.ac.uk/rts/guards
	ACQUA	www.crhc.uiuc.edu/PERFORM
	MARS/TTA	www.vmars.tuwien.ac.at
	MAP/MMS	icawww.epfl.ch/MMS
Industrial	**FactoryLink**	www.USDATA.com/solution/factorylink.html
Platforms	**FactorySuite**	www.Wonderware.com/products/factorysuite71.htm
	LabView	www.ni.com

IV Security

If you think cryptography is the solution to your problem, then you do not understand cryptography, and... you do not understand your problem.
— Roger Needham

Contents

Overview

Part IV – Security, addresses security of distributed systems, that is, how to ensure that they resist intruders. Security is paramount to the recognition of open distributed systems as the key technology in today's global communication and processing scenario. This part contains the fundamental notions concerning security, and provides a comprehensive treatment of the problem of security in distributed systems. Chapter 16, Fundamental Security Concepts, discusses the fundamental principles, such as the notions of risk, threat and vulnerability, and the properties of confidentiality, authenticity, integrity and availability. Chapter 17, Security Paradigms, treats the most important paradigms, such as: cryptography, digital signature and payment, secure networks and communication, protection and access control, firewalls, auditing. Chapters 18 and 19, Models of Distributed Secure Computing, and Secure Systems and Platforms, consolidate the notions of the previous chapters, in the form of models and systems for building and achieving: information security, authentication, electronic transactions, secure channels, remote operations and messaging, intranets and firewall systems, extranets and virtual private networks. Chapter 20 continues the case study, this time: making the VP'63 System secure.

15 CASE STUDY: MAKING VP'63 TIMELY

This chapter continues the case study that we have been carrying throughout the book: The VP'63 (VintagePort'63) Large-Scale Information System. The wine company has planned to automate some of its industrial facilities and needs to guarantee two objectives: to implement distributed real-time control and automation of some units such as wine processing and bottling/corking; to incorporate the real-time supervisory, control and acquisition (SCADA) into the global enterprise resource planning and information system, under a CIM perspective.

15.1 FIRST STEPS TOWARDS CONTROL AND AUTOMATION

The reader should recall that this is the next step of a project implementing a strategic plan for the modernization of VP'63, started in Chapter 5, and continued in the Case-Study *chapters of each part of this book. The reader may wish to review the previous parts, in order to get in context with the project.*

Generalized networking does not exist, and the situation of the company's industrial facilities can be described as being in the stage of *islands of automation.* That is, even when there is some local networking inside a cell, there is not a seamless interconnection and information flow in the whole industrial plant. In the factory floor, and specially inside the islands of automation, the company has introduced ad-hoc automation during the course of the years, to solve localized problems or improve certain processes. Devices are wired di-

P. Verissimo et al. *Distributed Systems for System Architects*

rectly to the controllers, except for a few more recent units, where devices are wired to the PLC through a proprietary field bus. The degree of computerized control still lies on PLC (programmable logic controller) technology and relay logic programming, as exemplified in a bottling/corking cell in Figure 15.1a.

15.2 DISTRIBUTED SHOP-FLOOR CONTROL

There are essentially two areas in the factory or shop floor: the continuous control part of the problem, which concerns wine processing; the discrete control part, which concerns activities like bottling and corking and labeling. The team identified the following problems in the current setting:

- imprecise continuous control leads to lesser quality batches, and also makes it impossible to regularize crops with different attributes in order to achieve predictable characteristics of the company's blends;
- imprecise discrete control leads to a high percentage of rejects in quality control (e.g. imperfect filling, labeling and corking);
- generally, but more so in the discrete processes, the rigid hard-wired and hard-coded systems make it difficult and slow to change anything in the process, although the requirement for flexibility is ever growing.

Q.3. 1 How to evolve towards a flexible support architecture for distributed real-time shop-floor control?

The team devised a strategic development plan consisting of designing a pilot distributed control system, to test and assess the new technologies. After a sucessful pilot phase, the results will be extended to the whole of the industrial facilities. A bottling and corking cell was elected for this sub-project.

Essentially, the cell consists of three modules: `conveyor`, `filler` and `corker`. The labeling module was omitted for simplicity. The conveyor belt brings empty bottles into the cell, passes them in succession under the filler, under the corker and then out of the cell to the packaging cell. The filler fills the bottle with wine from a tap connected through a pipe system to the wine tanks. The corker consists of a cork insertion press, with its own cork feeding subsystem, which presses the compressed cork into the bottle. The system has a few sensors: `bottle-arrived-at-filler`, `bottle-arrived-at-corker`, `filler-flow-meter`. Additionally, it has the following actuators: `belt` (start, stop), `tap` (open, close), `press` (down,up).

The prototype cell architecture is depicted in Figure 15.1, where the team applied known notions on distributed real-time architectures. One computing element per module (conveyor, filler, corker) processes the relevant state evolution. The sensor and actuator representatives are materialized by drivers that handle the relevant controllers. Given the cell simplicity, computing elements and representatives were distributed by three physical nodes. Each node hosts a computing element. Sensors and actuators relevant to a module were placed with the node hosting the relevant computing element. This is mainly for the sake of wiring organization, since the information from sensors is disseminated

to all computing elements, so that they all have a common knowledge of the system state. Outputs to actuator representatives are also disseminated. This way further changes in topology do not require a reconfiguration of the IPC. Protocol choice and configuration should obey the hard real-time needs of this type of discrete control operation.

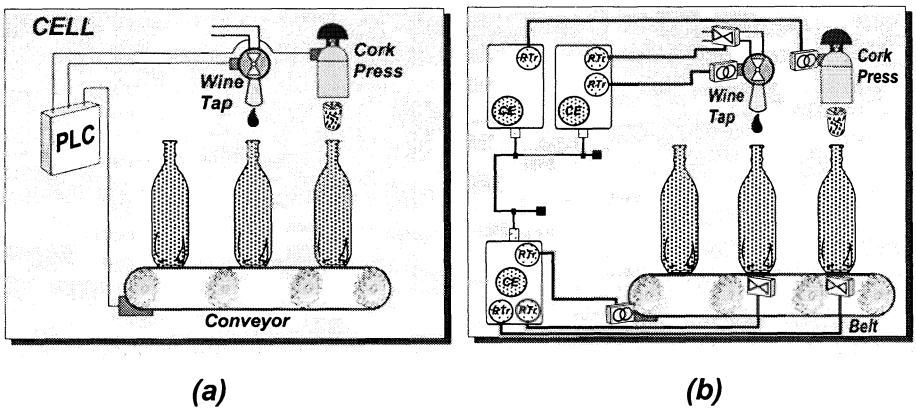

Figure 15.1. (a) Ad-hoc Computer Control; (b) Distributed Control of a Flexible Manufacturing Cell

15.3 INTEGRATION OF THE INDUSTRIAL SYSTEM

There are essentially two points to address in response to the corporate strategic plan:

- achieving a seamless industrial information flow between production management and shop-floor devices, in both directions (commands down, state image up)— this is now impossible because there is no global networking nor distributed information support;
- swifter interconnection of the industrial production information and the business information— achieving the first objective is a pre-condition to this one.

Q.3. 2 How to evolve towards an integrated industrial information system architecture?

Islands are not interconnected. Recipes and manufacturing orders have often to be inserted by hand in the shop-floor controllers.

Supervisory (SCADA) systems for different processes are proprietary, and having their own closed databases, making it impossible to dialogue with other subsystems and re-use their information. On the other hand, high-level applications (production management and control, maintenance management, quality control) run isolated, without an integrated connection to the business management system. In consequence, the business image of the complete enterprise has normally outdated information concerning the industrial facilities.

The team devised a plan for re-organizing the industrial facilities according to the CIM paradigm:

- design of an integrating networking infrastructure based on open protocols, TCP/IP for the matter, interconnecting all islands, with communication gateways to the office-level networking infrastructure;
- selection and installation of an open distributed SCADA platform providing a global image of the distributed processes to be shared by the high-level applications, and provision of a standard means of conveying commands and instructions from the latter to the shop-floor units;
- adaptation of existing applications or selection/installation of new applications capable of hooking to the open platform;
- design of a mechanism supporting the information flow between the industrial information system and the business information system.

The system architecture of a production unit is based on integrated networking between all cells, and a connection to the enterprise information system, which depends on the technology of the latter. Each cell is provided with a *cell controller*, where an instance of the distributed SCADA platform resides. Communication between devices and cell controllers, and between the latter and high level applications is ensured by the MMS factory communication standard, so that the VMD (virtual manufacturing device) paradigm can be used.

Cell controllers hold up-to-date copies of the cell state, and can replicate these images in other instances of the platform, namely those accessed by industrial applications. Conceptually, there is a seamless hierarchy of state freshness, from what may be called a *real-time image*, to a *historical image*. Real-time images are captured through the VMD event or polling interfaces, and held in volatile state in the cell controller. Their correct capture involves maintaining a real-time QoS stream between devices and the VMD. Historical images serve archival purposes, and hold the statistical memory of the process in persistent memory. They do not have real-time constraints, although they are often stored together with the timestamp of the sampling instant.

The other facet, that is, how orders, commands and instructions come from high-level applications to the shop-floor, is implemented through the VMD client-server interfaces. Devices, considered as VMDs, essentially dialogue with upper layer applications by acting as servers receiving commands in the form of client-server RPCs invoked by the industrial applications (e.g., modification of set-points in controllers, loading of new discrete control programs, loading of new continuous control recipes, loading of batch manufacturing orders etc.).

Further Issues

These issues need some refinement now, and the reader was assigned the study of a few questions that were still left unsolved:

Q.3. 3 *Would model would be more adequate for the continuous and for the discrete control cells respectively, time-triggered or event-triggered?*

Q.3. 4 *What measures should be taken to ensure the availability of the bottling/corking process implemented by the cell architecture just studied?*

Q.3. 5 *Are there safety issues in any of the production cells? If yes, money-critical or safety-critical?*

Q.3. 6 *Consider the remote applications visualizing shop-floor state images: in what way are they QoS-sensitive?*

Q.3. 7 *Could the distributed CIM platform take advantage from the generic distributed services put in place during the first phases (distribution and fault tolerance) of the project? How and which services?*

16 FUNDAMENTAL SECURITY CONCEPTS

This chapter addresses the fundamental concepts concerning security. It starts by defining what security is: the reasons leading to insecurity, the types of computer misuse, and the evaluation of the risk associated with both the vulnerabilities of computers and the threats to which they are exposed. Then, it explains the foundations of secure computing, and traces the relationship between distribution and security, on the one hand, and fault tolerance and security, on the other hand. Next, it analyzes the behavior of the intruder, in an attempt to illustrate to the reader both the motivations for attack, and the techniques and procedures normally used to perform that attack. Finally, the most relevant architectural and technological approaches to security in networks and distributed systems are introduced, to be detailed in the subsequent chapters of this part.

16.1 A DEFINITION OF SECURITY

In order to understand security, we must first understand what are the impairments to security. As with fault tolerance, we can only assure correct behavior of systems, if we understand why and how they fail in the first place. Two of the things that make this field a challenging one are that: the direct causes of failure are faults (attacks) maliciously made by humans; and those are very often made possible by unintentional faults (vulnerabilities) made by designers, who are also humans... and humans are unpredictable.

P. Verissimo et al. *Distributed Systems for System Architects*

16.1.1 Insecurity, People and Computers

People and computers have a curious relationship: from an initial militant mistrust, at work and at home, people tend to evolve to an almost blind trust on the machines' abilities to solve problems, and to an overwhelming dependency on computers to hold important information and perform important tasks. Alas, this trust and dependency are frequently not supported on any technical evidence. We have seen that this may have serious consequences when accidental faults occur (*see Fault-Tolerant Computing* in Chapter 6). It can have even more serious ones in result of malicious faults caused by other people— the intruders— on the computer systems people— the legitimate users— so haphazardly use. We will spend sometime analyzing the situation from the human viewpoint, and will extract a few morale quotes, that we emphasize in italics.

Perhaps most of the problems with computer insecurity can be attributed to social factors, more precisely, to the fact that informatics evolved so fast that social rules have not accompanied it. To give an example, people tend to minimize the importance of the information that computers hold: they rarely think of preserving the privacy of their own information, and tend to do the same with information entrusted to them, e.g., professionally. However, are these people irresponsible? Not necessarily: many of them methodically lock everything inside their desk's drawers and office lockers, while leaving the whole disk of their PC within the reach of a few thumbscrews. This happens because their behavior codes with regard to computer-based information are still imperfect, if compared to other media.

Insecurity is as concerned with technical deficiencies as with people's attitudes.

On the other hand, most of the hackers are juveniles who consider that invading another person's computer, stealing a password, or eavesdropping on computer networks is part of an innocent game, and in fact a proof of their intellectual superiority over other youths or grown-ups, who for instance have such naive passwords as `snoopy`, or do not have their servers protected with the latest patch against the latest discovered vulnerability. In fact, most of these hackers have never thought that there is little difference, in ethical terms, between what they do and: invading another person's house, even if the door is not locked; stealing a key to a personal drawer; listening to private phone conversations, and so forth. The fact is that the ethics of informatics is not part of the basic education. That is, the rights to integrity and privacy of property and personal information are recognized in terms of the traditional icons (real estate, cars, lockers, telephones, etc.), but not so much with computers, disk files, and network bits.

However, the irresponsibility of the young and sometimes inept hackers does not make their actions less serious. The amount of information available today on the Net, for example, whole Web pages full of recipes for exploiting vulnerabilities in computer systems of all makes and brands, spreads the base of potential hackers. It is not too paranoid to speculate that behind an everyday thicker curtain of amateur hackers, people with a real interest in severing computer security— computer criminals, radical groups, terrorists— can act with an increased degree of impunity.

Since this book is mainly devoted to university students, we cannot refrain from addressing two words at the informaticians-to-be whose passion by computers is addressed at other people's computers. The first is that most employers will not be very eager to hire a computer science or engineering graduate with a record for computer-related crimes. The second is that there is a fantastic difference between a mechanical engineer, who is able to design and build a car, and a car thief, who is only able to break into the car. Of course, the latter is an expert on alarms, car door and steering wheel locks, which make, say, one hundredth of the important parts of a car. Guess who has the most comprehensive and creative activity?

Insecurity is quantitatively caused to a great extent by the actions of people who must be educated about the seriousness of their deeds.

Security costs money. If a security case is well made, it should normally cost less than the losses arising from rare but devastating incidents. However, such as with fault tolerance, it is very difficult to persuade people to invest on a system attribute whose effect they will rarely or never see. It is much easier to authorize an investment on making the system "perform faster", because that is palpable, than on making the system "be secure". (This message is dedicated to chief information or technology officers and other people with similar decision power). If you pay a security guard (or have an insurance), and there is never an attempt to rob your house, after a few years you start wondering why you are throwing all that money away. After you fire the guard (or cancel the insurance), you are finally robbed, and you lose ten times more than what you had paid until now, but that was a human reaction, do not be demoralized.

It is better to invest than to spend.

The general insecurity reaction to the first serious incident is no less human: close everything, resort to very restrictive emergency policies, like for example disconnecting from the network, and invade the place with all sorts of new bureaucratic rules. Again, an old saying in action: "locking the stable door after the horse has bolted". This is very frequently complemented with a bit of burying-head-in-the-sand, such as pretending nothing happened, not making a complaint, not requiring the services of computer security specialists. Moreover, since the pressure to be on-line is enormous these days, it can be decided, many often in the worst possible way, to buy "something to take care of the pirates", preferably "something that has cryptography inside". In 90% of the cases, the innocent target of this frenzy is a firewall, that mysterious panacea for all security incidents. Shortly after, and with great surprise, the system is attacked again. Sometimes because unprotected direct modem dial-up connections from behind the firewall remained active, sometimes because the intruders were already behind the firewall: they worked there. No technology works per se. Organization security requires a systemic approach, as many other architectural problems in computer systems do: analyze the problem; establish the requirements and/or the policy; specify the functionality; select the enabling technologies.

Cryptocracy (a form of technocracy) is enemy of good systems practice.

To end with, if there was a special advice we would give to beginners in this domain, it would be: in any security case, always use your knowledge on how

humans behave (the good and the bad). There is a well-known saying about the fundamental problems being those that remain no matter how much hardware or software you put or take: in security, what remains are humans and their values.

16.1.2 Vulnerability, Attack and Intrusion

Hackers exploit weaknesses in operating systems, applications, network software, and so forth. Reckless users or administrators may introduce other weaknesses, because of the way they configure or run the systems. Illustrative examples of weaknesses are: world-accessible files; accounts with default password or without one; easily guessed passwords; stack overflow and other low-level bugs; windows of vulnerability in the execution of certain system calls; cleartext remote logins; unprotected programs with root privileges; forgotten protocol ports; lack of authentication of most communication protocols.

Hackers also do social engineering, which is a way of exploiting human weaknesses. Social engineering is the art of extracting secrets from people with their unwitting consent, and it can do more for a hacker in five minutes than a fortnight of methodic probing. An example of social engineering that became folkloric is the following: a hacker phones a company's system operator pretending he works for the Telecom company, alleging that there is a problem with one of the company's lines that connects a given server to the outside; he says that in order to perform tests he will need a login and password into the server; he even asks the operator to remain on-line and report if the operation was successful. It is surprising to find out how fashionable variants of this story are.

Vulnerability - non-malicious fault or weakness in a computing or communication system that can be exploited with malicious intention

It is interesting to define vulnerability in terms of dependability: vulnerability is a non-malicious design or configuration fault that can be activated to introduce malicious faults and/or errors in that system. It is constructive to establish a parallel between fault tolerance and security, since after all both are facets of the same objective of dependability: achieving justified reliance in the operation of systems vis a vis the occurrence of faults, be they malicious (security) or not (fault tolerance). The resulting cross-fertilization may result in new ways of looking at either field.

Threat- potential of attack on computing or communication systems

The closest analogy to threat in dependability is the definition of hostility of the environment, that is, the potential of the environment to introduce faults (e.g., electromagnetic radiation).

Attack - malicious intentional fault introduced in a computing or communication system, with the intent of exploiting a vulnerability in that system

An attack may be successful or unsuccessful. The latter can happen because the system was not vulnerable to that attack (or, what is equivalent, the attack was perpetrated by an inept hacker). When successful, an attack may generate errors in that system. Attacks on computer systems (hacking) assume several forms, more or less disruptive or destructive, such as attempting to: penetrate a firewall or a computer, introduce viruses, damage resources, eavesdrop for information theft or privacy violation.

Intrusion - erroneous state resulting from a successful attack on a computing or communication system

If nothing is done, the intrusion will result in the failure of the security mechanisms. Alternatively, the erroneous state characterizing the intrusion may be detected and fought back with countermeasures, or it may be masked if the system has enough defenses to resist the intrusion. In fact, these are underlying techniques of what we may call intrusion tolerance. The process of security failure, depicted in Figure 16.1, has an obvious analogy with the fault-error-failure sequence studied in the Fault Tolerance Part of the book.

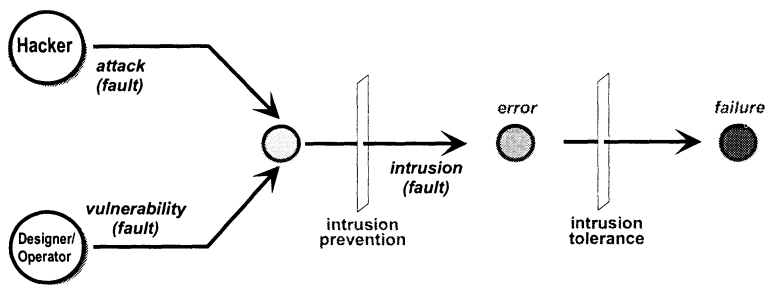

Figure 16.1. Vulnerability, Attack and Intrusion

The effects of intrusions, depending on the intention (and the skill) of the attackers, can take several forms, such as: interception and fraud in an electronic payment or home banking system; forging of electronic banking transfers; penetration of a company's information system for industrial espionage; or of a hospital's database, to disclose sensitive information about patients or use it for blackmail; destruction of a government's database, by a group of radicals; "bombarding" of a company's commercial server, in order to bring it down, for sabotage; fraud in the GSM mobile phone system, for free calling; breaking of an electronic wallet system, for counterfeiting digital money; and so forth.

Either the level of threat or the degree of vulnerability alone do not measure how secure a system is. In fact, consider the following two example systems, system Vault and system Sieve, whose degree of vulnerability and level of threat we have quantified for the sake of example. System Vault has a degree of vulnerability $v_{vault} = 0.1$, and since it has such high resilience, its designers have put it to serve anonymous requests in the Internet, with no control whatsoever to whoever tries to access it, that is, subject to a high level of threat, $t_{vault} = 100$.

System Sieve, on the other hand, is a vulnerable system, $v_{sieve} = 10$, and in consequence, its designers have safeguarded it, installing it behind a firewall, and controlling the accesses made to it, in what can be translated to a level of threat of $t_{sieve} = 1$. Now consider the product *threat* × *vulnerability*: with the imaginary values we attributed to each system, it equates to the same value in both systems (10), although system Vault is a hundred times less vulnerable than system Sieve. The correct measure of how secure a system is depends as much on the number and severity of the flaws of the system (vulnerability) as on the potential of the attacks it may be subjected to (threat). This measure is called risk, and it is very important since it explains that one cannot ascertain the level of security of system or product from the catalogue alone.

Risk - combined measure of the level of threat to which a computing or communication system is exposed, and of the degree of vulnerability it possesses

16.1.3 Security Properties

We have discussed until now what makes systems insecure, how they can be attacked, and what is the effect of insecurity. Now, we are ready to understand what we would like of a system, in order to consider it secure. We would like it to preserve our information from the eyes of intruders. We would like to be sure that whoever is dialoguing with us at the other end of the line is really who she says she is. We would like to avoid that someone changes or destroys our information. Finally, we would like to avoid that someone brings our system down on purpose. We may not always need all these attributes simultaneously, and some of them are not even achievable with the same techniques. These are good reasons for understanding as precisely as possible what we mean by security:

Confidentiality	the measure in which a service or piece of information is protected from unauthorized disclosure
Integrity	the measure in which a service or piece of information is protected from illegitimate and/or undetected modification
Authenticity	the measure in which a service or piece of information is genuine, and thus protected from personification or forgery
Availability	the measure in which a service or piece of information is protected from denial of authorized provision or access

Confidentiality is concerned with protecting information and computing and communication services from the eyes of intruders. Privacy is the default form of confidentiality for private information and communication. The simplest approach to ensure confidentiality is physical isolation, for example, locking a computer, or using networks which make tapping difficult, such as fiber optic media. However, this is not always possible or convenient, so we are going to study methods based on *encryption*, which ensures that although the intruder has access to the information, it is unintelligible to him.

Authenticity is concerned with guaranteeing the origin of a service request, a piece of data or a message, or the identity of a service provider or the creator of a piece of information. Intruders may wish to pretend they are someone else, as part of the attack on a system, for example for getting access to other people's information, or to forge the identity of the creator of a document. This is avoided by *authentication*. Malicious users may also pretend they did not do something they did do, for example, denying they signed a check with which they paid some goods. Avoiding it is called *non-repudiation*. For a piece of data, authenticity is recognizing the creator's signature, even if he denies. For a recipient, authenticity is being certain that a message signed by a sender was really sent by him, and that the sender cannot deny to have sent it. Key to ensuring authenticity is a technique that we will study called *digital signature*. Authenticity is not considered as a first order property by several authors, and is defined by some as being the 'integrity' of some meta-information concerning the identity of the object.

Integrity is concerned with avoiding or detecting the modification of information or messages, including service interactions, with malicious intent. The easiest way to secure integrity is to use some form of checksum, although with cryptographic resilience, to detect modification. We will study techniques to perform these checks based on *secure hashes* and *message digests*. However, sometimes we also require prevention of modification, if we fear a radical attack trying to erase our whole information system. Short of using redundancy techniques such as the ones we studied in the Fault Tolerance part of this book, the best solution, and one widely used in security, is not letting the intruders get anywhere near the information. We will also study *protection* techniques based on *access control*.

Availability is concerned with ensuring that information and computing or communication services remain accessible to authorized users, despite what are normally called denial-of-service attacks. These attacks may be performed for reasons such as sabotage, vandalism, terrorism and politics. Availability, such as integrity, can be procured with techniques based on protection. However, static access control has very little effect on denial-of-service attacks, as recognized in (Lampson, 1993). The reason is that an attack may come by the same channels as authorized users, such as bombing a Web server with legitimate requests in order to bring it down. Reactive (dynamic) access control based on intrusion detection, such as yielded by some firewall systems, may help mitigate the problem, for example by selectively blocking requests originating from a suspicious machine.

Availability may also be achieved by means of techniques based on redundancy management, that is, fault tolerance. The idea is to replicate the service, possibly in several places, making the intruder's life more difficult, since he has to attack all replicas in order to bring the service down. However, these techniques are not such a definite solution as they are for accidental faults. The reason is that for most of the latter it is possible to define a *fault model*, bounded both in type of behavior and maximum number of faults supposed to occur, and

build systems that can provably be available while that fault model holds. In the case of malicious intentional faults, we do not know yet how to put a bound on the type and number of attacks, since they are human-driven and depend on non-technological factors such as the persistence, time, and intelligence of a hacker[1].

16.1.4 Evolution of Secure Computing

Security is a very old activity. Before computers, people would already use ciphers to send secret messages. Among the first, we can count the **Caesar cipher**, used by the Romans. It was a *substitution* cipher, which worked by replacing each letter of the message with the letter 3 positions ahead in the alphabet, wrapping around from Z to A. Very naive, it worked well since there were not many cryptanalysts around at that time. A *transposition* or *permutation* cipher is yet another technique that works by shuffling the order of letters. A common transposition method would be to write the plaintext line by line, in a sheet of fixed character length per line, and then get the ciphertext by reading it column by column. Doing this twice would significantly increase robustness. It was reportedly used in World War II by the resistance, who would encode and decode their messages by hand.

Shannon launched the basis for cryptography, enunciating two fundamental principles: *confusion* and *diffusion*. Confusion is the property whereby it becomes extremely difficult for an intruder to find out the relationship between the plaintext and the ciphertext. The rationale behind it is to eliminate redundancies and statistical patterns. Substitution generally contributes well to create confusion. Diffusion is the property that dissipates the information patterns throughout the text, so that a single bit change should reflect itself in many places of the ciphertext. Current systems are a mix of the application of these two principles.

World War II brought the intensive use automated encryption/decryption, by means of mechanical devices, or **rotor machines**, such as the Enigma machine of the Germans. The principle of operation was based on substitutions, and permutations implemented by a mechanical wheel. One machine might have several rotors, the output of one wired to the input of the next. The Enigma was broken by the Allies, using computing resources. This is a historical example of a principle that is still true and haunts every cryptographic system: no matter how good a system is, it takes every year less time to break it by brute-force. Very important milestones for secure computing, still valid today, were erected after the war. We enumerate a few in Table 16.1.

The public key principle was one of the most important discoveries in cryptography, generally attributed to Diffie and Hellman. Merkle did contemporary work that also ranks him among the asymmetric cryptography pioneers. A recent announcement credits John Ellis, a cryptographer working for the British

[1] In fact, this may also prove difficult for software design faults.

Table 16.1. Major Milestones in Secure Computing

1972	Reference monitor protection model (Lampson, 1974; Anderson, 1972)
1973	BeLa Formal security model (Bell and LaPadula, 1973)
1975	DES- First widely used symmetric crypto algorithm (DES, 1977)
1976	Public key cryptography principle (Diffie and Hellman, 1976)
1978	RSA- First widely used asymmetric crypto algorithm (Rivest et al., 1978)
1978	Mediated authentication/key distribution (Needham and Schroeder, 1978)
1982	Blind signature for non-traceable digital cash (Chaum, 1983)
1988	Kerberos- Widely used KDC authentication service (Steiner et al., 1988)
1990	PGP- Public domain strong cryptography system (Zimmermann, 1995)
1994	ECash- First commercial non-traceable digital cash sys. (DigiCash, 1994)

government, for the invention of the public key cryptography principle a few years before Diffie and Hellman. This work was kept secret and never published until now.

Security in informatics[2] has long left the exclusive realm of the military and governmental institutions. Nowadays, networking and distribution on open systems have become the major paradigms for supporting information processing. Confronted with the need to work under the level of threat of environments such as the Internet, designers, vendors and users have become aware that the way to go can no longer be by reducing the threats to systems. Instead, the design of systems must be improved and adequate techniques incorporated, in order to reduce their vulnerabilities and achieve an acceptable risk of operation. This is why expressions such as 'public key cryptography', 'DES', 'digital cash', have become vox populi.

Generalized use of cryptography has made some governments nervous about the possibility of people having completely confidential communications and files, unlike what exists today with telephone communications and hard paper files, that can be disclosed under a warrant. In trying to preserve this metaphor, **key escrow** encryption systems were proposed. The system is a normal cryptographic system, except that the key is also deposited with a government agency. In special cases, namely a suspicion of crime, the key can be obtained from the agency by the competent authorities, which are then able to tap communications or decrypt files.

The use of cryptography has few *restrictions* in most countries, with the notable exceptions of France in Europe, and the prohibition of exporting strong cryptography products in the U.S.A. (Koops, 1999), partially lifted in the end of 1999. The near future will witness the increment of awareness by private users about the security risks of open distributed computing. This will give a great push for the proliferation of commercial systems incorporating cryptography.

[2] "Informatics" is a word of european origin getting increased acceptance in the community of computer users and developers. It is used to denote in general terms all that is related with use of computers and networks in information processing, access and manipulation.

Most probably, it will contribute to an evolution of the attitude of authorities with regard to security.

16.1.5 Distribution and Security

One of the oldest rules of security is to distribute the power and the knowledge about crucial issues, so that no single person can control and misuse that power or knowledge. We all remember the tales about the doors with three locks, and the secret map that was cut into several pieces. With the advent of computers, and since they were essentially centralized machines, the computer, as a unit of intrusion, became a single point of failure. Since it was not convenient to tear a mainframe in pieces, that ability of splitting the control of a secret was lost for a while, until distributed systems came into play.

Distributed systems, on the other hand, have presented some shortcomings in the transition from centralized ones. First, instead of being held in a central physical point, critical data was spread by several sites. Secondly, these sites were forced to exchange information through networks whose physical access cannot be controlled completely. These are detrimental factors to the baseline security of distributed systems.

However, distributed systems bring back the possibility of distributing control and knowledge. The fact that data reside in several sites may be used in one's favor, if spread in a way that the intruder must break into several sites to retrieve useful information. Cryptography has taught us how to securely send information over insecure networks. In conclusion, with the adequate techniques, distribution presents the systems architect with a powerful framework to achieve very high levels of security, and materialize back the metaphors with which we started this section.

16.1.6 Fault Tolerance and Security

Dependability is the justifiable reliance on the operation of a system. From that viewpoint, achieving reliance on the presence of accidental or of malicious faults are two faces of a same coin. Vulnerabilities of the system are non-malicious design or configuration faults that the intruder exploits to induce other faults of malicious nature, or attacks. This combination aims at achieving an intrusion on the system. The resulting erroneous state, if not handled, may lead to a security failure of the service that the system is supposed to provide (e.g., communication confidentiality, database integrity, etc.). Besides the conceptual analogy, there is a chance for using techniques of either field in a complementary way. In essence, intrusion detection and recovery, or intrusion masking, can be seen as techniques for achieving *intrusion tolerance*. Some fault-tolerant protocols use cryptographic signatures as a means for tolerating value faults. Byzantine agreement techniques, used in ultra-dependable systems to mask arbitrary errors, also have applicability in security.

16.2 WHAT MOTIVATES THE INTRUDER

As we have seen, behind the several possible security-related failures in a computer system, there is a human brain. It is difficult to establish a regular model for malicious faults, as we have done for example with accidental faults when studying dependability (*see Fault-Tolerant Computing* in Chapter 6). It is also not obvious how we award probability distributions to human decisions, for example, it would be surprising to discover that hackers' successful attempts at installing exploits in an operating system follow a Poisson curve. Short of such a formal framework, we may nevertheless try to understand what motivates the intruder: the *hacker* of computer systems, or the *phreaker* of telecommunication systems, in security lingo.

The first interesting thing to learn about intruders is the "why". The motivations of hackers vary: curiosity; collecting trophies; free access to computational and communication resources; bridging to other machines in a distributed system; damaging or sabotaging systems, for criminal, mercenary or political reasons; stealing information for own use or for sale, such as software, commercial or industrial secrets, etc. The "how" depends on the hacker's ability and on the system to be penetrated. Generally, he takes the following steps:

Exploration of vulnerabilities	Finding weak points in the computer system (e.g., accounts without password; vulnerable configurations or drivers present; accessible password file)
Access to the system	Making a plan of attack (e.g., matching dictionary words to password file entries; attack system driver with malicious program to get an account; eavesdrop on login/password pairs on the network)
Control of the system	Controlling all system resources, by becoming *root* user after attacking the system's security mechanisms with malicious programs (e.g., malicious script securing root access; racing a penetration program against a legitimate system call)
Deletion of traces	Concealing activity during intrusion campaign (e.g., disguising himself while logged in, erasing system logs after logout)
Continued stealth access	Perpetuating access to the machine in a stealthy manner (e.g., Trojan horses or backdoor programs that can be activated by special codes or sequences, abandoned accounts)
Exploration of new targets	Looking for new trophies (e.g., channels leading to other machines; access information in personal files; eavesdropping from the intruded machine)

Talking about the tools used, it is typical to find a hacker sitting on a good PC, behind a fast modem. Many hackers use blue-boxes and other "colored" boxes, which are devices to confuse the billing electronics of telephones and allow the hackers to call for free. "Software tools", in the form of attack programs, called "exploits" in hacker lingo, abound in the Internet, exchanged in Internet Relay Chat systems (IRC) or published in Web pages.

16.3 SECURE NETWORKS

Security in networks starts with physical protection. One of the major threats today is eavesdropping on networks. Technology can give a hand here, since for example, fiber optic cables are very difficult to tap without notice and in practical terms they do not radiate.

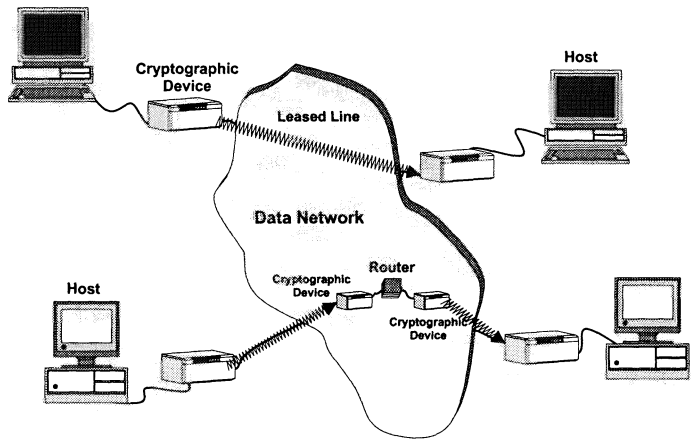

Figure 16.2. Physical Circuit Encryption

The latter attribute is important, since today it is possible to detect energy radiated by the passing bits on network cables, at considerable distances. However, while measures can be taken inside premises to make physical tapping difficult, wiretapping in large-scale networks is almost impossible to avoid. In consequence, we had better look elsewhere for network security: communication encryption. There are different levels at which encryption can be performed, in terms of the OSI model. Encryption can be done at the lowest layers, Physical and Data Link, or the higher layers, such as Network (e.g., Internet IP), Transport, Presentation, Application, or even by the user.

Physical circuit encryption, also called link encryption, is adopted when the idea is to encrypt all traffic passing through a link between two points. As shown in Figure 16.2, a physical link is connected by a pair of cryptographic devices, and all cleartext (non-encrypted data) going through that route is encrypted, that is, converted to ciphertext (denoted by zigzag lines), which is decrypted by the corresponding device at the other end. Encryption is done at the Physical or Data Link layers, either in software or in hardware, but this approach is most convenient for hardware-based encryption. As shown in the top of Figure 16.2, this approach is specially suited for linking hosts via long-haul leased lines. Physical circuit encryption applies strictly to a link between two points. In consequence, if there are routers in the middle, then the data has to be decrypted, so that the router can analyze the routing information, and then routed via another link, where it is encrypted again, as shown in the bottom of the figure. Typically, each pair of devices over a link shares one key.

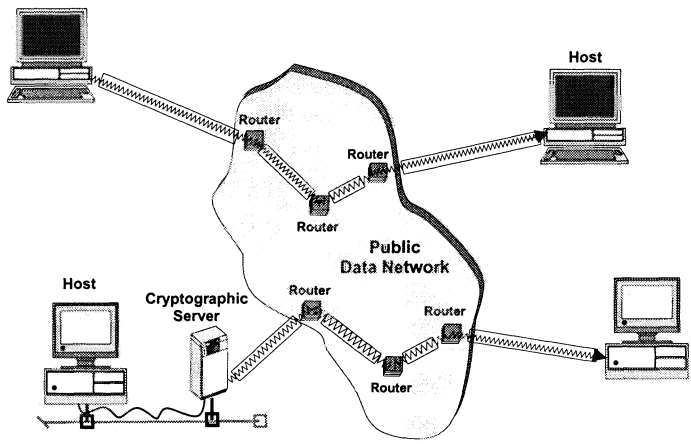

Figure 16.3. Virtual Circuit Encryption

Physical circuit encryption requires a large number of encryption devices, and a large number of encryption/decryption operations in routes with many hops. Alternatively, one can resort to *virtual circuit encryption*, also called end-to-end encryption, where encryption is performed on a per traffic flow basis, at the higher layers of the architecture. Not only can encryption be selective, because of the multiplexing existing at these layers, but it can also be preserved until the final destination. This is because the headers of the lower communication layers are appended *to* the encrypted data, instead of being embedded *in* it, like it is done in physical circuit encryption. This is exemplified in the top of Figure 16.3, where the encapsulation of encrypted data is shown by a rectangle enveloping the zigzag line. Encapsulation is removed and inserted again in the course of normal processing of communication functions at each hop, but the zigzag line remains intact. For example, encryption can be done at the Network or Transport layers, and only for certain destination domains, addresses or ports. The Network protocol header (e.g., IP) is inserted after encryption, and in consequence, the data can remain encrypted throughout its way. Virtual circuit encryption can also be performed higher up, such as the Presentation or Application layers, or even by the user process itself. Email communication, for example, has provisions for encryption. Virtual circuit encryption is the most used approach, and is normally performed in software. However, nothing prevents it from being performed in hardware between the user and the host, as is the case with applications relying on intelligent smart cards. Physical and virtual circuit encryption can be combined in order to achieve higher levels of security. The principle exemplified in Figure 16.3 can be used to build an encrypted link between two facilities of a same organization. The respective endpoints are connected by a cryptographic stream encapsulated in a network protocol, with the sole purpose of carrying (tunneling) the data through the network between these two endpoints. This principle can be extended to sev-

eral endpoints in a pair-wise manner, to create what are called **virtual private networks**, or VPNs, in a secure way.

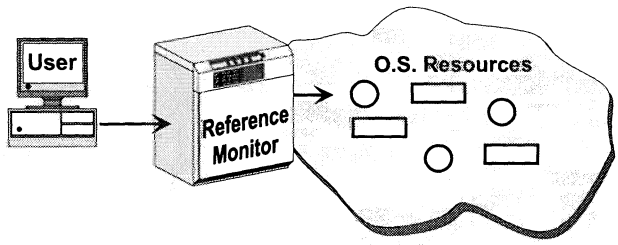

Figure 16.4. Reference Monitor

16.4 SECURE DISTRIBUTED ARCHITECTURES

This section gives an overview of the main architectures for distributed systems security. Conventional operating systems have protection mechanisms based on facilities implemented in hardware by the underlying microprocessors. Normally these provide for a number of layers of privilege for executing instructions and accessing resources, and also virtual and separate address spaces, or even full virtual machines executing in complete isolation over the same hardware (Tanenbaum, 1992). Operating systems then offer a higher-level of protection centered in their resources, such as files and network devices. Users have different rights of access to different resources, and may even have private resources.

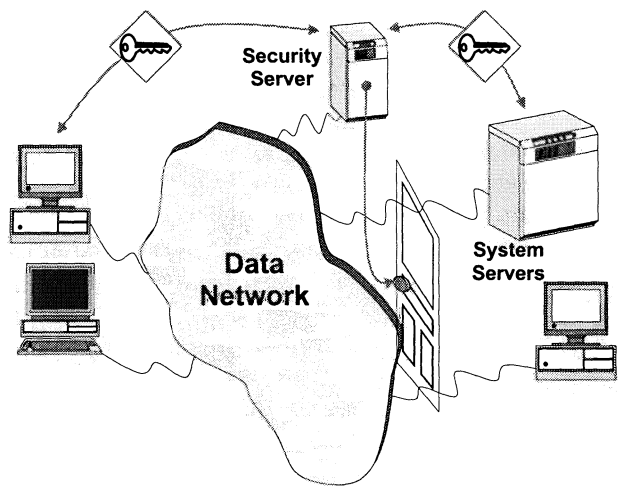

Figure 16.5. Security Server

In secure systems, the concern about protection is taken one step further: it is delegated on an entity, called *reference monitor*, such that all requests to

protected resources are addressed for authorization to that entity (Figure 16.4). The reference monitor is normally a secure part of the operating system kernel. In distributed systems, an *authentication and authorization server*, or *security server*, must perform that task for interactions between machines accessing resources that are distributed in a network. As a consequence, the gatekeeper function of the reference monitor is done in a virtual and distributed way, as illustrated in Figure 16.5. Although all machines we see may be physically interconnected by the same network, logically the clients on the left side can only access the system servers on the right side after granted permission by the security server. They have to authenticate themselves and get an authorization to access a given resource. The reason why this "virtual gate" works is that this dialog is cryptographic, and the authorizations take the form of *cryptographic credentials* that the system servers analyze.

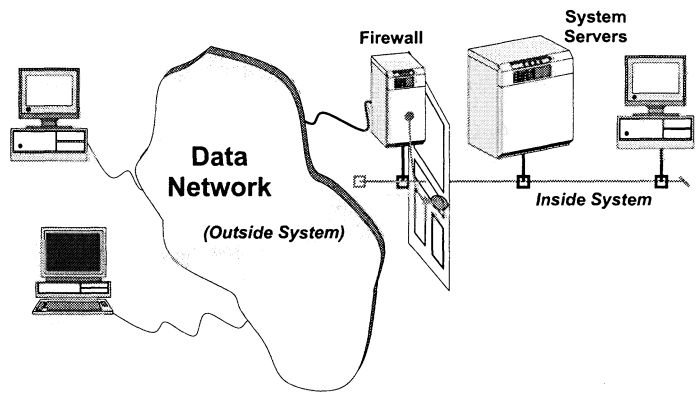

Figure 16.6. Firewall

A simpler form of protection that in a way extends the principle of the reference monitor for centralized systems is the *firewall.* It acts by physically interposing a barrier between an outside system and an inside system. Although the protected inside system may be a distributed system, for the firewall it is a hard-core in terms of security, that is, a *security perimeter* (all the bad guys are out and the good guys are in). In consequence, all requests coming from the outside system to resources in the inside system must pass through the firewall, as depicted in Figure 16.6. This model is weaker than the reference monitor model, since its access rules are less precise and normally directed to communication protocols (block or allow traffic to given ports, addresses, protocols, etc.). A special form of firewall function called *application gateway* can implement more sophisticated protection, but in an application-dependent form. On the other hand, in terms of network traffic, the firewall is more versatile, since it also allows protection to traffic from the inside to the outside.

Implementing secure remote operations is fundamental in open systems. The problem, as exemplified in the top of Figure 16.7, is achieving security of interactions between a client terminal and a server. A **sniffer**, a machine that

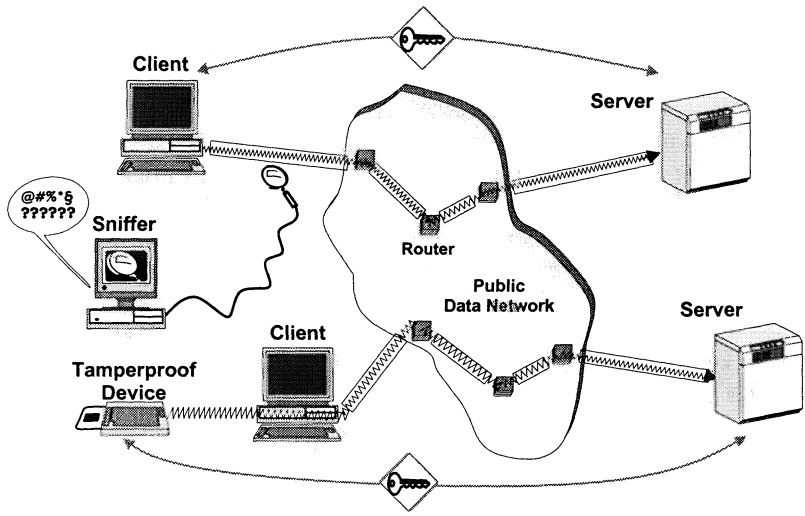

Figure 16.7. Secure Remote Operations: Normal, and with Tamperproof Devices

listens to everything that passes on a network, may read all the information, including passwords and sensitive information, for later misuse. As we see in the figure, secure remote operation architectures include some form of virtual circuit encryption (*see* Figure 16.3), which encrypts either all the communication, or just sensitive parts, in an end-to-end manner, so that eavesdropping does not succeed. These architectures are used in a variety of situations over insecure environments like the Internet, such as plain remote logins, client-server computations, Web HTTP server accesses, or electronic commerce. In this latter application, the level of security may be increased if, as exemplified in the bottom of Figure 16.7, the cryptographic channel is established between the server and a **tamperproof device** representing the user. The latter represents a good solution when the client machine is not trusted. These devices are for example: boxes with secure micro-controllers that read a card and execute cryptographic protocols, or intelligent smart cards themselves able to execute cryptographic protocols.

There are several variants of electronic payment architectures. The most promising are the ones centered around digital cash. As Figure 16.8 exemplifies, the relevant parts of the architecture are: a banking network; and tamperproof devices, also called wallets or purses. Digital cash is generated by the client's bank and loaded into the client's smart card (a tamperproof device of its own right) by means of a cryptographic protocol ran between the smart card and the bank server. This can be done in an ATM machine, for example. What is interesting in this architecture, in contrast to the ones presented before, is that not everything happens inside a connected network. The client takes her card to a merchant and pays completely off-line, again by running a cryptographic protocol that unloads digital cash to the merchant's terminal,

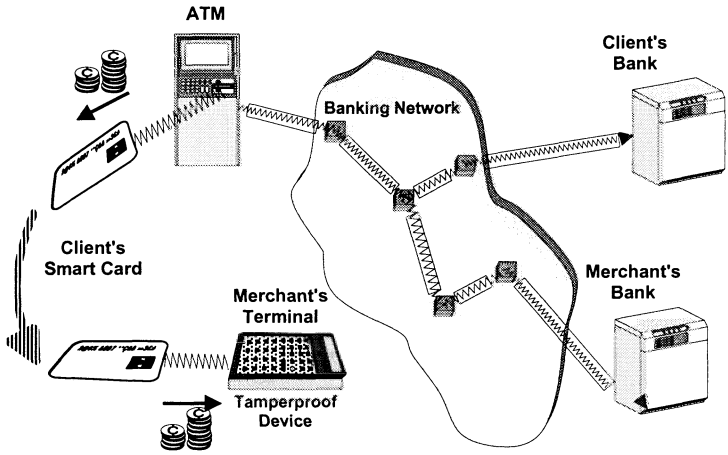

Figure 16.8. Electronic Payment

a tamperproof device that is not connected permanently as well. Later, the merchant deposits the money in his bank. The role of the banking network is to consolidate these electronic transactions, making the money flow between the several participants— in this example, from the client's account to the merchant's account.

16.5 SUMMARY AND FURTHER READING

This introductory chapter discussed the fundamental concepts concerning security, and introduced terms such as: vulnerability, threat, intrusion, security hazard, confidentiality, integrity, authenticity, availability, sniffer, physical and virtual circuit encryption, secure tunnel, reference monitor, security server, firewall, tamperproof device. During the following chapters, we will discuss them in greater depth. For more introductory level material, the reader should see the introductions of (Pfleeger, 1996; Kaufman et al., 1995). Pfleeger also does an extensive discussion on legal and ethic issues (Pfleeger, 1996). Kaufman et al. (Kaufman et al., 1995) and Abrams et al. (Abrams et al., 1995) give fairly complete glossaries of security terms. Stallings provides an interesting reading on security in networked and distributed systems (Stallings, 1999).

17 SECURITY PARADIGMS

This chapter discusses the main paradigms concerning security. An exhaustive description of all the major algorithms concerned would require a whole book's length in order not to be shallow. Instead, we will motivate each paradigm in a problem-solving manner, exemplified when applicable with one or two chosen algorithms that are analyzed with the necessary detail. Throughout the chapter, we are going to discuss: trusted computing bases, cryptography, digital signature, digital cash, authentication, protection, and secure communication.

17.1 TRUSTED COMPUTING BASE

Most of the paradigms to achieve security that we describe below require some form of computation. One might ask *where* is this computation supposed to take place, in order for it to be immune to hackers in the first place. The **Trusted Computing Base (TCB)** is the paradigm whereby it is possible to build a computing nucleus that is immune to intrusion, even if submersed in hackers attacks, or even if incorporated in a system built by hackers. A few other building blocks are based on its existence: reference monitors, firewalls, and authentication servers.

17.1.1 Specification of a TCB

A Trusted Computing Base (TCB) is that part of the system, comprising hardware, firmware and software, which is responsible for supporting the security

P. Verissimo et al. *Distributed Systems for System Architects*

mechanisms used to protect the system, such as authentication, access control, auditing. The specification of a TCB must secure a few desirable properties:

Interposition	the TCB position is such that no direct access to protected resources can be made bypassing the TCB
Shielding	the TCB construction is such that it itself is protected from unauthorized access
Validation	the TCB functionality is such that it allows the implementation of verifiable security policies

The TCB is a subset of the operating system, it has total control of the hardware it runs on, and no other software runs in a more privileged level. In consequence, *Interposition* means that nothing can detour the TCB to access system resources. Note that we are not saying that everything *has to go* through the TCB, but that the access to any resource *can* be controlled by the TCB, so that it can implement whatever functions necessary to control access. These functions depend on what sort of *security policy* is being enforced by mechanisms built on top of the TCB, what sort of attacks it is trying to resist, etc.

Shielding means that nothing should intrude the TCB. By construction, the TCB must be impervious to accesses to its own structures, other than by the authorized users, normally the *security administrators*. Again, all depends on the proper design of the TCB.

Finally, *Validation* means that the TCB functionality should be simple enough in order for the security mechanisms materializing the security policy in the computer system to be formally specifiable and verifiable.

The above-mentioned properties do not imply that a TCB cannot be fooled into doing wrong things: you should note the *separation of concerns* between the enabling architecture for protection (the TCB) and the protection mechanisms themselves (authentication, authorization, auditing). If the latter are improperly designed or else follow an improper security policy (or the absence of one), then security hazards can happen in spite of a correct TCB.

17.2 BASIC CRYPTOGRAPHY

Cryptography is a mandatory paradigm in security. Modern computer cryptography relies on a *cryptographic algorithm* or **cipher**, and a **key** or pair of related keys. The original data, *cleartext* or *plaintext*, passes through the algorithm and generates *ciphertext*. This is called **encryption**, and it is a function of an encryption key. The role of the key is to parameterize the algorithm, such that for the same cleartext, encrypting it with two even slightly different keys yields drastically different ciphertexts. The reverse operation, called **decryption**, recovers the original cleartext from the ciphertext, by running the algorithm again, with a decryption key. In some systems, the encryption and decryption keys are equal.

We will use the following notation: $E_{K1_a}(M)$ to mean "encrypt M with encryption key $K1$ related to a" (a may denote the key owner, or a session, or a message). Similarly for decryption: $D_{K2_a}(C)$, where $K2$ is the decryption key. If there is no ambiguity, we will omit indices, for example $E_{K_a}(M)$. As in all security textbooks, we will use the classic players for our metaphors: Alice and Bob, the good guys, helped when needed by Carol and Dave. Mallory and Eve, the bad guys. Trent, a mediator, or trusted third party. It is usual to use **principal** to designate any *participant* in a security protocol. We will use both words interchangeably.

Table 17.1. Basic Attributes of a Cryptosystem

• given a pair of encryption/decryption keys $K1$ and $K2$, if $E_{K1}(M) = C$, then $D_{K2}(C) = M$ and thus $D_{K2}(E_{K1}(M)) = M$
• given $E_{K1}(M)$, without $K2$ it is infeasible to recover M
• given M and $E_{K1}(M)$, it is infeasible to recover $K1$
• given $K1$, it is infeasible to recover $K2$ and vice-versa
• if keys are equal, $K1 = K2 = K$, first three statements still apply

K1=K2=K, the cryptosystem is symmetric and the above statements still apply . if K1=/=K2, the cryptosystem is assymmetric, and given K1 it is infeasible to obtain K2, and vice-versa.

The strength of the **cryptosystem** defined as above lies not on the algorithm's secrecy, but on the secrecy and quality of the keys. The algorithm can and should be public domain. A proprietary secret algorithm does not give users any guarantees about its robustness or seriousness, since the principles on which it relies are not known. Furthermore, if put in public domain, it will be exhaustively tested by the academic community, so that as years go by, either it ends up being broken (we say it is cryptanalyzed) or the confidence in it increases. However, the algorithm must be such that if an intruder does not have the key, he cannot obtain useful information from the cyphertext. In fact, this can be put in the form of more concrete attributes, listed in Table 17.1.

Infeasible does not mean impossible. Actually, by testing all possible combinations of keys, plaintexts, ciphertexts, etc., one may impair some of the premises above. This is a **brute-force attack**, and keys should be made long enough that the computational cost and/or time involved make this attack impractical. A word of caution about the cryptographic protocols based on an algorithm: the way the algorithm is used should not introduce vulnerabilities. Unfortunately, this may sometimes happen.

There are two major classes of algorithms in modern cryptography: symmetric and asymmetric. The names derive from the fact that the former relies on a secret key used in both operations of encryption and decryption, whereas the latter relies on two different keys, one used for encryption and the other for decryption. Another relevant building block is secure hashing, which has the

property that it cannot be tampered with, and interesting uses, such as creating unique "fingerprints" of texts or messages. In what follows, we discuss these three building blocks with more detail, emphasizing the aspects that matter to a systems architect. For a deeper study of cryptography, the reader should refer to the remarkably clear and complete book of Schneier (Schneier, 1996).

17.3 SYMMETRIC CRYPTOGRAPHY

Symmetric cryptography is characterized by the fact that there is only one key, which is kept secret, and instances of it have to be shared between both ends of a channel. In general: $K1 = K2 = K$, and thus if $E_K(M) = C$, then $D_K(C) = M$ and thus $D_K(E_K(M)) = M$. If a key is shared by two participants, like A and B depicted in Figure 17.1, it is also usual to denote it by K_{ab}.

The security of this approach relies on K being kept secret by both participants. This presents several problems. One is of self-containment: a new key must go through some secure channel to both participants (or at least from one to the other). Practical solutions involve alternative channels such as person-to-person or paper mail. However, the latter are not interesting if keys need to be exchanged frequently, in which case fast and secure means to distribute and exchange keys must be used. For example, by combining cryptosystems, we can use one to deliver the keys of the other. The other problem is of robustness: a key compromise at either end compromises the whole channel. A third problem is related with scale: key management becomes worrying for large systems. For example, since one key is needed for each pair of participants, in order to support arbitrary communication among 10 participants 45 keys are required, and this number goes up to almost 5000, for 100 participants. On the bright side, symmetric encryption algorithms are relatively fast.

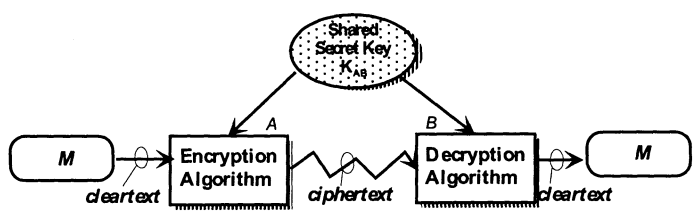

Figure 17.1. Symmetric Cryptography

There are two types of symmetric ciphers: *block ciphers* and *stream ciphers*. The block algorithms encrypt one block of data at a time (e.g., 64 bits). The stream algorithms process the cleartext one bit (or byte) at a time. A widely used block cipher is *DES*, the *Data Encryption Standard* (DES, 1977).

17.3.1 Data Encryption Standard

The DES was developed for the U.S. government and standardized in 1977. The DES algorithm is based on an iterative application of substitution and permutation functions, for 16 cycles. Substitutions achieve Shannon's paradigm of *confusion*, whereas the permutations, or transpositions, provide for another Shannon paradigm, *diffusion*.

The algorithm is quite simple and elegant. It is a block cipher, of 64-bit length. The key is 56 bits long. It works as outlined in Figure 17.2. The input is permuted initially. Then 16 equal iterations follow. Each round receives the result of the previous as input. The block is divided in two halves, one of the halves (R_{i-1}) is combined with a 48-bit sub-key generated from the 56-bit DES key (one different sub-key is generated per round), and then XORed with the other half (L_{i-1}), yielding R_i. R_i is concatenated with L_i (equal to R_{i-1}, giving the 64-bit result of the round. After the 16 rounds, a final permutation is performed, which is the inverse of the initial permutation. An interesting property of DES yields a very simple decryption procedure: if the ciphertext is run through DES in the same way as encryption is done, with the same key, the cleartext is obtained. The only condition is that the sub-keys are used in the reverse order.

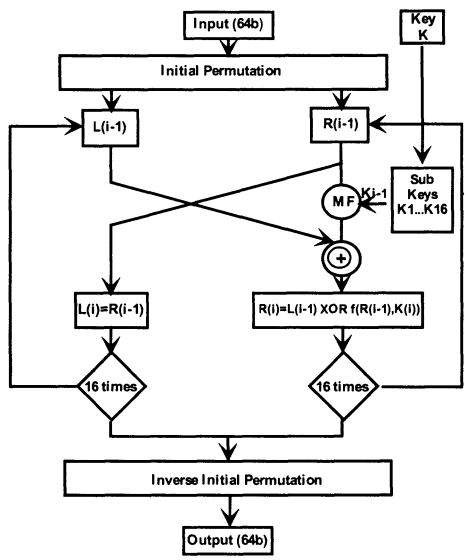

Figure 17.2. DES - Data Encryption Standard

DES is quite fast, and specially amenable to hardware implementations. It is one of the most used algorithms today, and no fundamental weakness was discovered so far. However, the 2^{56} search space of the 56-bit key length is a current source of worry. It is believed to have been chosen because it corresponded, around 1977, to a computational power within the reach of the

U.S. government security agency (NSA), but no one else, to break the cipher by brute force. However, advances in computational power have currently placed that power in the hands of too many organizations and people. Despite this proviso, DES is a very robust algorithm, if used in the adequate mode.

17.3.2 One-Time Pads

A *one-time pad* is the best representative of the stream cipher type of symmetric encryption. It is the only really unbreakable cipher. It is based on having a truly random, never-ending sequence of characters, bytes or bits, depending on what we want to encrypt, which we combine, one-by-one, with our plaintext stream. The original one-time pad idea applied to characters (Kahn, 1967): the pad was a stripe of truly random characters that were added modulo 26 to the plaintext stream of characters. In computers, the pad is binary, and it is XORed with the plaintext in transmission. In reception, it is XORed again with the ciphertext, yielding the original text. Since the pad is random and used only once, there is no information for the cryptanalist to withdraw. One-time pads can be a very useful building block, so they deserve a few comments:

- security relies on the secrecy of the pad, which must, as with any symmetric cipher, be distributed to both ends of the channel;
- security relies on the randomness and uniqueness of the pad: non-random sequences and reuse introduce weaknesses;

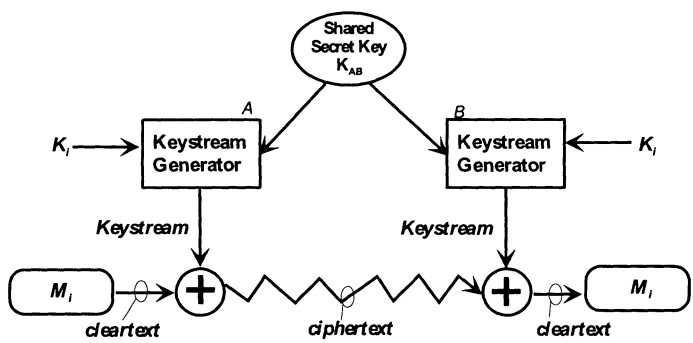

Figure 17.3. Stream Cipher

Real-life one-time pads are only possible if generated and distributed in advance. This has its uses, but for encryption of computer communication links, stream ciphers must be produced in real time, in the way shown in Figure 17.3. Hardware box A produces a stream (called *keystream*) that *looks* like random, and XORs it with the incoming plaintext stream. However, it cannot be random, since box B must produce *exactly* the same sequence, in the same phase, so that, when XORed again with the ciphertext, it recovers the plaintext. Of course, such a device will always produce the same sequence when turned on, becoming susceptible to attacks. In consequence, keystream

generators have keys, and the output is a function of the key, as shown in the figure. Stream ciphers are susceptible to bit errors that desynchronize the ciphertext stream. The quality of such a system is dictated by how much it resembles a random-number generator.

17.4 ASYMMETRIC CRYPTOGRAPHY

In asymmetric cryptography there is a pair of keys, the *public key* and the *private key.* Because of this fact, it is also called **public key cryptography**. Each participant owns such a pair of keys. Only the private key need be secret, the public key is handed away to anybody wishing to send the participant an encrypted message. As depicted in Figure 17.1, participant B gave participant A her key, or published it in a name server (or in a newspaper ad, why not?). A uses B's public key Ku_b to encrypt M and send it. Only the pair of the public key, private key Kr_b, can decrypt the message. In consequence, in our notation: if $E_{Ku_b}(M) = C$, then $D_{Kr_b}(C) = M$ and thus $D_{Kr_b}(E_{Ku_b}(M)) = M$.

The security of this approach relies on Kr being kept secret. This presents two advantages with regard to symmetric cryptography. The first is that there is no need for key exchange with secrecy: every participant generates keys and publishes the public one. The second is that a channel can only be compromised in one end, that is, the end of the key owner. The scale of key management is also better than for symmetric cryptography, since one public key is needed per participant. In consequence, for 100 participants we need only 100 keys. The down side is that asymmetric cryptography is 1000 (HW implementations) to 100 (SW implementations) times slower than symmetric. Key distribution deserves a word of caution: if a public key is not received in first hand from its owner, then it must be ensured that it is authentic. A key repository may be tampered with, and when fetching Alice's key, we might unwittingly be getting Mallory's key, who had in the meantime penetrated the key/name server as part of an attack against our interaction with Alice (*see* more on this in Section 18.6).

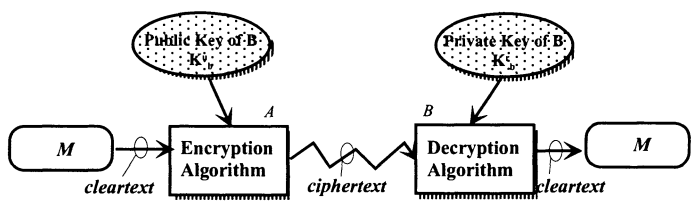

Figure 17.4. Asymmetric Cryptography

17.4.1 Diffie-Hellman

The first asymmetric algorithm published was the secret number computationDiffie-Hellman algorithm (Diffie and Hellman, 1976). It owes its security to the difficulty of calculating discrete logarithms to invert exponentiations in a finite field, or the difficulty of factoring numbers that result from the product

of large primes. The objective is to arrive at a shared secret number without ever passing it over a communication medium. As depicted in Figure 17.5, Alice (A) and Bob (B) only do public communication. To arrive at the shared secret number K, they execute the same procedure. They both agree on public numbers n, a large prime (e.g., 512 bits), and m, which can be small, and whose properties with regard to n are omitted here (*see* (Diffie and Hellman, 1976) or (Schneier, 1996) for details). Then, Alice generates a *secret* large random number x_a, performs $y_a = m^{x_a} \bmod n$, and sends y_a to Bob. Bob does the analogous computation, using x_b and sending y_b to Alice. Finally, they both compute the same number K, since

$$K = y_b^{x_a} \mod n = y_a^{x_b} \mod n = m^{x_b \cdot x_a} \mod n$$

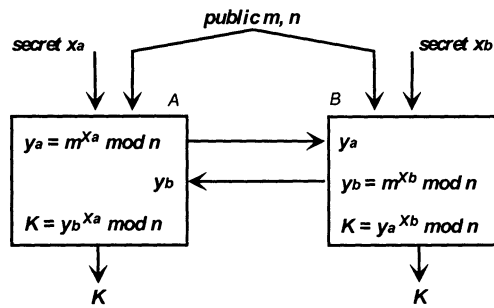

Figure 17.5. Diffie-Hellman

Diffie-Hellman does neither encryption nor authentication. D-H's obvious utility is to create shared secret keys to be used in symmetric cryptography (*see* more on this in Sections 17 and 17.11), and it is very effective at doing it.

17.4.2 RSA

RSA, published in 1978, owes the name to its inventors, Rivest, Shamir and Adleman (Rivest et al., 1978). It was the first widely used asymmetric encryption algorithm. Its security relies on the *trapdoor one-way function* concept, that is, a function that is not reversible (one-way) unless a secret is known (trapdoor). In the approach followed, this boils down to the difficulty of factoring numbers that result from the product of large primes, such as in the Diffie-Hellman algorithm.

A prior step consists in generating the key pair. The key length is variable (a typical size is 1024 bits). Alice selects two *secret* random large prime numbers p and q, of equal length, and computes $n = p.q$. Then she selects a random encryption key e, such that e and $(p-1)(q-1)$ are relatively prime. Finally, she computes the decryption key $d = 1/e \mod ((p-1)(q-1))$. For the user, $Kr_b = \langle d, n \rangle$ is the *private key*, which must be kept secret, and $Ku_b = \langle e, n \rangle$ is the *public key*, which Alice publishes or sends to people.

RSA is not a block cipher, but cleartext M is divided in blocks of size smaller than n that are encrypted one at a time. The ciphertext is the concatenation of the encrypted blocks. The algorithm itself is quite easy to understand. The encryption and decryption of one block is outlined in Figure 17.6. To encrypt a message for Bob (B), Alice (A) fetches Bob's public key $\langle e, n \rangle$, and performs $c_i = m_i^e \mod n$. Decryption is similar: Bob, using his private key , $\langle d, n \rangle$, computes $m_i = c_i^d \mod n$.

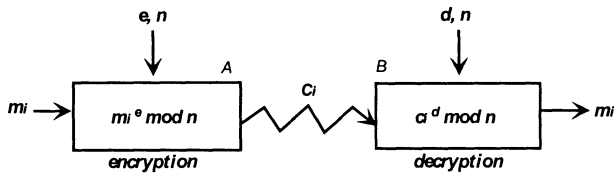

Figure 17.6. RSA– Rivest-Shamir-Adleman

RSA is slow compared to DES, but it is very secure. The variable key length allows it to accompany technology evolution. A 1024-bit key is believed to be extremely secure, given the enormous search space, but nothing prevents anyone from encrypting with a much longer key (and much slower...), say for extremely sensitive documents.

There are no known serious or feasible vulnerabilities to RSA itself. However, protocol flaws such as poorly chosen encoding of messages may open the door to some kinds of attacks. RSA Data Security proposed a standard to address this issue. It is called PKCS (Kaliski, 1993), and it is a set of documents with guidelines for encoding and setting-up structures for using RSA correctly and without vulnerabilities.

17.5 SECURE HASHES AND MESSAGE DIGESTS

Hash functions are compression functions. One-way functions are non-reversible functions. A very useful building block in cryptography is a **one-way hash function**, a function that is easy to compute, compressing a text to a block of fixed length (typically 128 bits), but very difficult to reverse. In that sense, it is also called a *secure hash* or **Message Digest (MD)** algorithm. We use the following terminology: the *digest* or *hash value* of M is $h_m = H(M)$. The security of a message digest lies on the fact that it is not reversible. We are more specific about the necessary properties in Table 17.2.

The first property is helpful for using hashes as representatives of documents without revealing those documents. The second and third are useful in signing texts: they state that a hash uniquely represents a given text, and no one can produce another text that hashes to the same value and say it was the previous text instead. They are also useful to checksum messages, in order to check if they were tampered with: if no one can produce a message that hashes to the same value as the original one, then any changes in a message after a digest was performed are detectable.

Table 17.2. Basic Attributes of a Secure Hash

• given h_m, it is infeasible to recover M such that $h_m = H(M)$
• given M, it is infeasible to find M' such that $H(M) = H(M')$
• it is infeasible to find a pair (M, M') such that $H(M) = H(M')$

Most secure hash algorithms work in the way exemplified in Figure 17.7: they digest a message recursively block by block, using the digest of the previous block as input to the next round. Secure hashes or message digests have a number of uses, for example, to fingerprint a text maintaining its privacy, to compress a text that is going to be signed, or to check the integrity of a block of data.

17.5.1 MD5

MD5, for Message Digest, was invented by Rivest (Rivest, 1992), who produced the whole series of MDi's. It is the direct successor of MD4, and is more secure and a bit slower than its predecessor. MD5 processes the text in 512-bit blocks (64 bytes), and produces a fixed-length output of 128 bits (16 bytes). MD5 operation is exemplified in Figure 17.7. The text is padded to a multiple of 512 bits, with a length field inserted in the pad. The algorithm makes four passes at each block, taking as inputs the 128-bit digest of the previous block and the current 512-bit block, and mangling them in different ways.

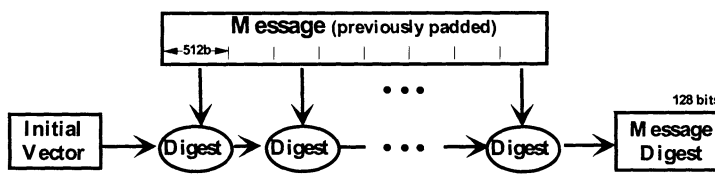

Figure 17.7. Message Digest Algorithm

17.6 DIGITAL SIGNATURE

In this section, we discuss several forms of signatures in latus sensus: message authentication codes, message integrity checks and digital signatures. Let us introduce some terminology: given a pair of keys $K1$ and $K2$ belonging to principal A, **signature** of A is $S_{K1}(M)$ and **verification** of A's signature is $V_{K2}(M)$. When there is no ambiguity, we may use $S_a(M)$ and $V_a(M)$. If a symmetric approach is used, then $K1 = K2 = K$ but the signature process becomes more complex. The obvious utility of signatures is more or less the same as in real life, but let us be a bit more formal and characterize what is a correct digital signature, that is, a well-formed and not-revoked signature:

Authenticity	a correct signature uniquely identifies a principal, and only that principal
Unforgeability	a correct signature was made by its owner deliberately
Integrity	a correct signature on a document ensures that it cannot be changed without that being noticed
Non-reutilization	the whole or part of the signed document cannot be reused in another document
Non-repudiation	a correct signature cannot be denied by the owner of the signature (key)

These properties emulate what we would desire of paper signatures. Actually, some of these properties can be violated in hand-made signatures. In computers, we have to be even more careful, since a computer file is vaporware compared with the hardness of a paper signature. *Authenticity* stipulates that the signature unmistakably identifies a given principal. That is, people can recognize s_a as being Alice's signature. However, Mallory might imitate Alice's signature, so it must also have the property of *unforgeability*, which ensures that if we are facing Alice's signature, it was really made by her deliberately, because no one else could forge it. Once a document is signed, it cannot be changed, at least without that being noticed. That property is called *integrity*. Suppose Mallory did cut-and-paste of Alice's signature from a legitimate document file to a document forged by him? This would be quite easy with ASCII files. In consequence, we know that it must be avoided by stipulating the *non-reutilization* property. In addition, now we know that digital signatures are not made in ASCII. Sometimes, our problem is not with Mallory forging a signature, but with Mona, who disguises herself, buys expensive jewelry, but later denies saying that someone stole her check wallet on that day. *Non-repudiation* is the property that ensures that the signer cannot deny having put her signature on a particular document. The fact that the signer can accept this property follows from the previous properties. As an exercise, deduce that a signed document obeying the first four properties *must* be a legitimate, unaltered document deliberately signed by the signature owner.

17.6.1 Cryptographic Checksums

Checksums are small fixed-length strings that are used to check the integrity of messages or files. Hashes give good checksums, like network packet CRC (cyclic redundancy check), but unlike checking against accidental bit errors, a mere hash is not enough against a deliberate attack: Mallory changes the message and then recomputes the hash (which is public). A *cryptographic checksum (CC)* is a non-forgeable hash, in a sense, a form of signature of a message, that depends on a key. We use the following terminology to differentiate from plain hashes: the *CC* of M is $H_K(M)$. CCs make sense when a message does not require encryption but integrity must be safeguarded. They may also authenticate messages exchanged between two users. They separate protection

from encryption: when a message is encrypted, it is naturally protected, but after decryption at the recipient, the protection is lost. Having CCs of sensitive files stored in disk would for example foil virus or Trojan horse invasion.

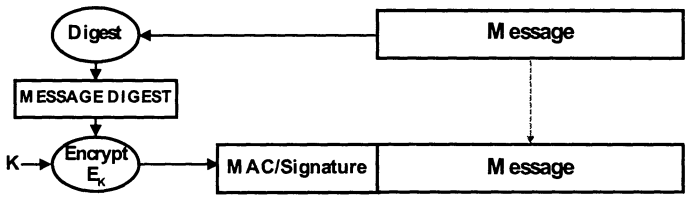

Figure 17.8. Generating and Appending a Signature or MAC to a Message

Cryptographic checksums may be implemented in several ways, and take several names: *Message Integrity Code* or message integrity check (Linn, 1993), or *Message Authentication Code (MAC)*, depending on whether they are securing integrity or authenticity, respectively. A simple way of generating a MAC is the following: Bob hashes the message, with say MD5, and then encrypts the hash with any symmetric algorithm, say DES, obtaining a MAC. The block diagram of this scheme is explained in Figure 17.8, considering that E_K is a symmetric block cipher with secret key K.

An alternative and very simple method is based on hashing only, dispensing with encryption. It only depends on Alice and Bob sharing a secret key K (not an encryption key, just a secret key): Bob computes the length L of message M, and concatenates L, M, and K; then he computes the message digest, obtaining $H_K(L, M, K)$. The approach is very fast and it is secure with a resilient message digest algorithm.

Cryptographic checksums secure the unforgeability and integrity properties. The above methods have the key distribution problem typical of symmetric approaches. By using public key cryptography the key distribution problem is minimized, while also providing an elegant method for true digital signature.

17.6.2 Signing and Verifying

A very interesting additional result of asymmetric cryptography is that *encrypting with a private key* or *decrypting first* (then encrypting) is equivalent to *signing*. Although there are public key signature algorithms, if an encryption algorithm is used for signing, then: *signature* by A is $S_a(M) \equiv D_{Kr_a}(M)$ and *verification* of A's signature is $V_a(M) \equiv E_{Ku_a}(M)$. The principle is depicted in Figure 17.9.

There is no problem in doing the operations in reverse order. Note that when Alice encrypts M with her private key getting S, she produces something unique and unforgeable, since her key is secret. On the other hand, anyone can verify Alice's signature: when Bob receives S supposedly signed by Alice, he fetches Alice's public key, and decrypts S, which could only have come from Alice. It is not as simple as that though: if Bob does not know what he is

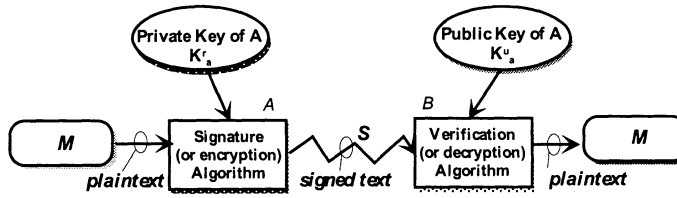

Figure 17.9. Asymmetric Digital Signature

expecting to verify, he can be fooled. Consider the following: Mallory can send a piece of data to Bob, pretending it is a signed document coming from Alice; under certain circumstances, Mallory may even construct a ciphertext that makes some sense when Bob verifies it; this is forging, and it is undesirable. Any encryption algorithm can be used to sign, but there are reasons, of both efficiency and security, to be careful about the structure of the signed document and/or to use specialized signature algorithms.

It is not efficient to run an algorithm on a whole text just to sign it. Imagine it is a 10 MByte contract! The message digests that we have studied solve the problem, yielding a *message-digest public-key signature* protocol. The principle can be understood by looking again at Figure 17.8, but considering now that E_K is a public key (asymmetric) cipher, and K is the private key of the signer. The protocol is explained in Figure 17.10, where Alice wants to send a signed text M to Bob.

		Action	Description
1	A	$h_m = H(M)$ $s_m = S_a(h_m)$	Alice computes the message digest and signs the 128-bit digest with her key
2	A → B	$\langle M, s_m \rangle$	She sends both the text and the signature to Bob
3	B	$h_m = H(M)$ $v_m = V_a(s_m)$ $(v_m = h_m)?$	Bob verifies the signature using the following procedure: he hashes the message; then he verifies the signature using Alice's public key; if the result is equal, then M is ok, and was signed by Alice

Figure 17.10. Message-digest Public-key Signature

Let us discuss now the effectiveness of digital signatures in real-life applications. For example, Bob cannot *reuse* the signature to append in another document, but he can reuse the whole document (it is a file!). This is not convenient if it is a bank check or digital note. In order to completely assure the non-reutilization property when the whole document cannot be reused, the document should include unique sequence numbers, timestamps and/or expiry

dates before it is signed. The verification process will include checking if the sequence number exists, if the expiry date was not exceeded, etc.

There is another problem, concerning **repudiation**, that also exists with hand-made signatures and negotiation protocols (e.g., credit cards). It is explained as follows: Bob signs a document at 3:00pm. Then he later complains to the police that someone early that morning stole the diskette where he held a copy of the private key, and gives forged evidence that this might have happened as early as 12:00am. In consequence, he denies all signatures that he made since 12:00am, including the document in question. There will always be a window of uncertainty in these operations. In other words, it is a fundamental problem: it can be reduced, but it cannot be eliminated, unless the model is changed. In fact, we can eliminate it if we timestamp and certify all transactions, but that requires on-line access to a mediator.

The security of public-key signature with one-way hashing lies on two facts besides the resilience of the signature and hashing algorithms: the secrecy of the signer's key, both physical and in terms of length (e.g., 1024 bits yield an enormous search space); unfeasibility that another message has the same hash as the original one (e.g., 128 bits yields a probability of 2^{-128}). This signature approach has the authenticity, unforgeability, integrity, non-reutilization and non-repudiation properties. In that sense it is a fully-fledged signature.

There is an additional advantage in the message-digest public-key signature scheme explained above, which concerns *multiple signature*. Suppose that the contract of our last example was to be signed by n principals. Instead of doing n whole-text signatures, each principal signs a copy of h_m, and the signed text is $\langle M, s_m^1, ..., s_m^n \rangle$. Each signature can be verified separately.

Still another advantage is for *notary* purposes: Alice wants to archive a document with a notary, who dates it and certifies that Alice produced it before that date. In the classical procedure, the notary would have to see and copy the document, and that can be inconvenient. With this method, the notary only certifies the digest of the document, and so Alice can keep the privacy of her document and only reveal it if it is ever necessary to prove that the certification corresponds to it. The probabilities that there is another meaningful text that hashes to the same value are negligible.

Signing with symmetric cryptography is worthwhile mentioning. Since a symmetric key would have to be shared between two principals that do not trust each other (the signer and the verifier), this cannot be done directly, but rather through an **arbiter**. This is obviously bothersome.

17.6.3 DSA

The *Digital Signature Algorithm (DSA)* (DSS, 1994) is an asymmetric algorithm for signing. Most of the market used RSA for signature until the *Digital Signature Standard (DSS)*, featuring DSA, came out in 1994.

We will give an overview of the protocol, skipping the details. The signer has a long-term pair of keys: a private key Kr chosen at random and a public key Ku computed from some public numbers and the private key. The key

length is variable from 512 to 1024. For each signature, a new pair of keys is generated, let us call them single-use keys: private Kr_s chosen at random and public Ku_s computed from the public numbers and the private key. A text M is previously hashed and then signed with a function using Kr_s, Ku_s, and Kr. The signed text is $\langle M, s_m, Ku_s \rangle$, where s_m is the signature, and Ku_s the public single-use key for that text. The security of DSA relies on: generation of "good" public numbers; Kr being kept secret; and Kr_s not being reused.

17.6.4 Blind Signature

Blind signatures were invented by Chaum (Chaum, 1983), with the purpose of authenticating an object without revealing the identity or whereabouts of its owner. If the object is digital money, this provides for **untraceability**, a desirable property that real money has, but is very difficult to achieve with digital money, without impairing other properties (*see* Section 17.7).

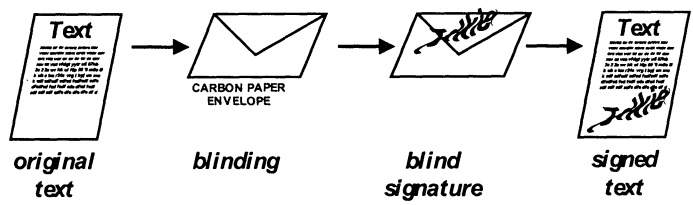

Figure 17.11. Generating a Blind Signature

The metaphor behind the blind signature concept is illustrated in Figure 17.11. The problem is the following: Alice has a document to be signed by Trent, but she does not want Trent to read it. She puts the document inside an envelope with carbon paper lining, and asks Trent to sign the envelope on the outside. The signature prints on the document as well because of the carbon paper. Alice opens the envelope and has the document signed by Trent. The algorithm is described in Figure 17.12. Alice wants Trent to sign document M for her. Trent has public and private keys u and r, and public modulus n. It works because the blinding and signing operations are commutative.

This completely blind signature is uncomfortable for Trent, since he does not know if Alice is giving him something nasty to sign (such as "I owe Alice a million EURO"). A protocol combining blind signature with a cut-and-choose technique introduces faiirness in the process. The protocol, shown in Figure 17.13, is a generalization of the protocol of Figure 17.12. The cut-and-choose metaphor is explained as follows:

Cut-and-Choose - Alice and Bob have a bucket full of fish that they caught, to divide in half. Alice divides the fish in two piles (cutting) and Bob blindly picks one of them (choosing). Bob thinks this is fair because his odds of getting the best pile are 50%. On the other hand, Alice should make the division fair, since she has no advantage

		Action	Description
1	A	$\widehat{M} = Mk^u \bmod n$	Alice prepares document M, and blinds it, obtaining $\widehat{M}$, by multiplying by a quantity depending on the blinding factor (random number k) and Trent's public key (u)
2	T	$\widehat{S(M)} = S(\widehat{M}) = (Mk^u)^r \bmod n = M^r k \bmod n$	Trent signs the blinded document $\widehat{M}$
3	A	$S(M) = \widehat{S(M)}/k = M^r \bmod n$	Alice unblinds $S(\widehat{M})$, obtaining document M signed by Trent with r

Figure 17.12. Blind Signature Protocol

in cheating by making one pile bigger than the other— her odds are 50% as well.

	Action	Description
10	A → B	Alice prepares p blinded forms of notes of value V, blinds each of them with a different blinding factor and sends them to Bob, the banker
20	B	Bob selects $p - 1$ forms, asks Alice for their blinding factors, removes the blinding factors from the forms, and verifies they request V
30	B → A	Bob signs the remaining note form, N, hands it to Alice and debits V from Alice's account
40	A	Alice removes the blinding factor from note N, but Bob's signature remains on it. She can spend it in a shop now

Figure 17.13. Basic Digital Cash Minting Protocol

The chances that the last copy of the note form contains something different are 1 in p, and of course, this probability can be made as small as wished, since it is dictated by the number of blinded copies Alice has to generate. By the properties of digital signatures, Bob can later recognize his signature and is bound to it, even if he has never seen the text that he signed.

17.7 DIGITAL CASH

Digital cash is the materialization of the "money" metaphor onto computer and digital systems. It is an enabling paradigm for emerging technologies that will certainly play an extremely important role in the near future, such as digital

payment systems, electronic transactions and electronic commerce. Digital cash depends on a number of cryptographic principles that we have already discussed, namely digital signatures, and very specially blind signatures.

17.7.1 Properties of Digital Money

Perhaps the two most frightening nightmares about digital money are: for the user, that it evaporates somewhere inside a network or computer; for the authorities, that someone discovers how to counterfeit it. This is nothing that could not happen with real money though: it could burn under the mattress in a house fire or evaporate in the bank under a depression; it can be counterfeited with several degrees of perfection.

It is extremely important however, to formalize a set of properties for the digital money concept. Something against which algorithms, protocol and system designs can validated. The following desirable properties, almost sic from (Okamoto and Ohta, 1992), comprehensively define digital money:

Independence	properties of digital money do not depend on its location
Uniqueness	digital money items cannot be copied or reused
Untraceability	digital money items cannot be traced
Off-line Validity	digital money items have standalone value
Transferability	digital money items can be transferred between users
Divisibility	digital money items can be subdivided

The independence property stipulates that digital money does not get less secure when pieces get out of the user's wallet and into the merchant's terminal, for example. Uniqueness stipulates that you cannot counterfeit money by copying an item or by reusing it. Untraceability is a very important property, though many existing systems do not provide it: it guarantees, such as with real money, that one cannot trace where the money was spent or who spent it. Off-line validity assures that money can be spent without need for connection to any central system. Transferability and divisibility ensure that users can pay things or give money to each other, instead of only to the merchant, and that pieces can be subdivided into smaller ones.

Existing digital cash systems fulfill only some of the properties, while an experimental system proposed in (Okamoto and Ohta, 1992) satisfies them all. The need for particular properties and the impact of their absence will be discussed further in Section 18.11.

17.7.2 Generating and Using Digital Cash

To introduce a digital cash payment system, we will use as our point of departure, the cash minting protocol using blind signatures and cut-and-choose that we studied in Figure 17.13. After Alice has note N (step 40 of Figure 17.13)

she goes on and spends it at Mike's shop. The basic protocol for payment with digital cash is shown in Figure 17.14.

	Action	Description
50	A → M	Alice spends N in Mike's shop
60	M	Mike the merchant checks Bob's signature
70	M → B	If everything is Ok, he accepts payment and sends N to Bob
80	B	Bob checks the signature and credits V to Mike's account

Figure 17.14. Basic Digital Cash Payment Protocol

This protocol lets Alice or Mike *reuse* the note, although they *cannot forge* it. This is called *double spending*, and a simple modification addresses the problem. It consists of having Alice concatenate each form of note with a random *uniqueness string (Un)*. Bob now also verifies that the Un strings are all different, when unblinding the forms. The note finally gets back to Bob, after Alice bought her merchandise. Bob, besides checking the signature, also checks that Un does not exist yet in his "spent" database list. If it does, then there has been double spending. There is a final problem to be solved: double spending is indeed detected, but the guilty may either be Alice or Mike, that is, there is an *imperfect detection*. One possible remedy would be for Mike to ask Alice to write a random non-erasable *identity string (Id)* on N. Then, Bob compares the identity string in the database record with the one in the note: if it is the same, Mike is the guilty one, if not, it is Alice. However, Alice could have forged the Un random number generation, giving the same identity string in the second time, to frame Mike. In consequence, this remedy and the previous one are only safe for *on-line* spending.

Off-line operation requires more sophisticated techniques to create Id, shown in the final protocol in Figure 17.15, where we consolidate the protocols of Figures 17.13 and 17.14 and show the modified or added lines in bold. Recall that Un is a uniqueness string randomly generated, long enough that there are no two notes with the same Un. Before we proceed, let us introduce two more cryptographic operations:

Secret Splitting - division of an item M of data in two parts such that either of them alone reveals no knowledge about M; joining of the two is a public operation that reveals M

Bit Commitment - processing of an item M of data such that M can no longer be changed and the result reveals no knowledge about M; M can be publicly revealed when the owner reveals a secret

Detail on the two can be found in (Schneier, 1996). The preparation of the Id strings in step 10 is the following: Id is an ASCII string revealing Alice's

	Action	Description
10	A → B	**Alice prepares p blinded forms of notes of value V, concatenates each form with one *uniqueness string (Un)* and p *identity strings (Id)*, such that for each one, $Id_j = Id_{j_L} \vert Id_{j_R}$, and sends them to Bob, the banker**
20	B	Bob unblinds $p-1$ forms and verifies that they request V
21	B	**Bob also verifies that the Un strings are all different and that the Id strings identify Alice**
30	B → A	Bob signs the remaining form, N, hands it to Alice and debits V from Alice's account
40	A	Alice removes the blinding factor from note N
50	A → M	Alice spends N in Mike's shop
60	M	Mike the merchant checks Bob's signature
52	M,A	**Mike gives Alice a binary *selection string (Ss)* of length p and requests Alice to reveal either Id_{j_L} or Id_{j_R} of each of the p Id strings of N, depending of the value (0 or 1) of Ss in that position**
70	M → B	If everything is Ok, he accepts payment and sends N to Bob
80	B	**Bob checks the signature, checks that the Un string does not exist yet in his "spent" database list, inserts the Un and Id strings in the list and credits V to Mike's account**

Figure 17.15. Robust Digital Cash Payment Protocol

identity completely; Id is secret splitted in two halves Id_{j_L} and Id_{j_R}; each one is bit committed. This ensures that: Alice cannot change them; only the two halves reveal Alice's Id. If the Un string exists, there is a problem. Then Bob checks the Id string: if it is the same, Mike is guilty, if not, it is Alice. Let us understand why:

If they are the same, it can only be because either Mike copied the note, or Alice forged two identical notes. However, if Alice spends the note again, the new merchant will give her a different selection string Ss. It is almost impossible that the two notes look the same (probability 2^{-p}), even if Alice wanted to, trying to frame Mike. On the other hand, if they look different, could it be Mike trying to forge a slightly different note and reuse it, framing Alice? That is impossible, since only Alice can reveal other sections of the Id string. In consequence, if they are different, definitely Alice is cheating. Furthermore, Alice must have revealed Id_{j_L} to one merchant and Id_{j_R} to another, in at least one of the p Id strings, and can thus be identified by Bob.

So finally, we have a protocol whereby Alice can *anonymously* spend her digital cash *off-line* without any fear of tracing, unless she cheats. However,

the protocol provides adequate safeguards to the merchant and the bank, by *detection* of fraud. Recall that the security of this protocol lies in the blind signature privacy, and in the very low probabilities: of Alice cheating the bank ($1/p$, for p initial note forms); of there existing two equal Un strings (2^{-L}, for an L-bit length); and of there existing two identical selection strings (2^{-p}, for a p-bit length).

17.7.3 Payment with Tamperproof Devices

There are two problems with the previous protocols: people can still cheat, although they are detected and prosecuted; eavesdroppers and spoofers can defraud the scheme if they have physical access to the devices involved.

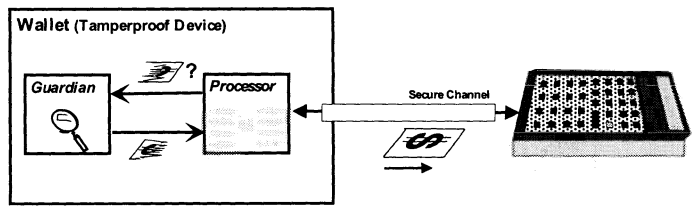

Figure 17.16. The Guardian Concept

The obvious way to avoid this problem is to give Alice and the merchant *tamperproof devicestamperproof device* where they store bank notes, and which are capable of running a secure end-to-end protocol. This way, Alice or Mike cannot tamper with the devices, and an intruder cannot penetrate the secure channel. It is possible to conceive a secure off-line payment system this way. Let us sketch a simple protocol for payment with tamperproof devices (e.g., smart-cards):

1. *Bob the banker has a public/private key pair ($Term_u, Term_r$) and a public/private key pair ($Note_u, Note_r$) to generate 1 EURO notes; Bob produces Alice's wallet and Mike's terminal, and stores the $Note_u$ and $Term_r$ keys in the terminal; he also stores $Term_u$ in Alice's wallet;*
2. *Alice loads her wallet with a sum in notes produced and signed by Bob with $Note_r$, say at an ATM (another tamperproof device);*
3. *Alice connects her wallet to Mike's terminal, and wants to pay a purchase;*
4. *The wallet authenticates the terminal with the $Term_u$ key and both establish a secure channel; the wallet transfers a sum V in notes to the terminal, which the latter validates with the $Note_u$ key;*
5. *The terminal later uploads the notes to the bank, and Bob checks his signature, verifies the note number against a "spent" list, and credits Mike.*

This is secure because the devices are trusted by the bank not to cheat, are supposedly tamperproof, and only connect to an alien device after authentication. Note that step 5 is a double check for an eventual fraud, but it implies that Bob recorded the note number upon its generation and most probably, Alice's identity as having purchased the note. When the note comes back, it

may have recorded the place where it was spent, and so Alice's steps can be traced. If the wallet, for example a smart-card, is trusted only by the bank who issued it, Alice cannot be sure that her privacy will be safeguarded. If on the other hand, it is trusted by Alice alone, then the bank or the merchant may question its security.

17.7.4 Payment with Guardians

The solution lies in the *guardian* or *observer* concept (Chaum, 1992): the **electronic wallet** consists of a tamperproof device whose processor is trusted by the user. The wallet processor executes the protocols, but needs the cooperation of another element, the *guardian*, trusted by the bank. Their relative position and interaction, as illustrated in Figure 17.16, is such that: the guardian cannot tamper with the protocol execution; the guardian cannot communicate with the outside; the processor cannot progress with the protocol execution without the repeated assistance of the guardian. Alice trusts the processor and knows that the guardian cannot disturb its operation, nor reveal secrets to the outside without the processor noticing. The bank trusts the guardian and only accepts operations in which the guardian has partaken. This is called *multi-party* security. We can make our protocol sketch evolve to payment using tamperproof devices with guardians:

1. *Alice has a wallet with Gus, the guardian from Bob the banker;*
2. *Alice loads her wallet in the bank with notes of value V;*
3. *Gus loads a down counter with V, which it decrements at each payment;*
4. *Bob generates a pair of public-private signature keys, hands the private one to the care of Gus, and gives the public one to Alice's wallet;*
5. *When Alice spends a note N, her wallet has Gus sign the note, after which it decrements the counter;*
6. *Alice hands the note N to Mike the merchant (Mike has Gus's public key). Mike checks the signature;*
7. *Mike sends N to Bob, who checks the signature again and pays Mike.*

The protocol above is just a sketch: it is too simple and naive, since anyone can generate a pair of keys and simulate Gus operation near Mike. Alice can do double spending. Besides, the protocol is sensitive to Gus being broken into. The solution lies in having Alice and Gus cooperate to generate a blinded signature book containing totally anonymous signature key certificates called **digital pseudonyms**, each of which will serve to validate one and only one spending. A full discussion, and working protocols can be found in (Brands, 1995) and (Chaum, 1992).

17.8 OTHER CRYPTOGRAPHIC ALGORITHMS AND PARADIGMS

Encryption and Digest Algorithms

IDEA, International Data Encryption Algorithm (Lai, 1992) deserves our attention because it is currently one of the most promising symmetric algorithms,

it is widely used and has been extensively analyzed during the past few years, without any fundamental weaknesses discovered. The most relevant protocol using it is PGP (*see* Section 19.1). It is patented and can be licensed for commercial applications. It is a block cipher of 64-bit blocks.

RC4 is a symmetric stream cipher with variable key size, widely used in several protocols. One example is the SSL protocol (*see* Section 19.1). Although it is proprietary (RSA Data Security), its sources became public domain on the Usenet years ago. RC4 has a special export status if it uses a reduced key length, which was up to 40 bits long during many years. Although becoming fragile, this was attractive to U.S. companies willing to export their secure systems. They only have to modify the protocol to use a reduced key length. What is strange is why this would be attractive to buyers in countries where there are few or no restrictions to cryptography, in detriment of alternative cryptographically-stronger products. Certainly, a different attitude would help liberalize cryptography. Merkle's Knapsack was really the first asymmetric encryption algorithm, but suffered a series of cryptanalysis that compromised its success (Merkle, 1978). El Gamal (ElGamal, 1985) is an asymmetric encryption and signature algorithm which inspired DSA. The main differences are in performance, DSA being significantly faster. The Secure Hash Algorithm (SHA), was proposed as a standard to be use together with the DSA signature standard. It produces a 160-bit hash, and makes five passes over each block, so it is in principle more secure against attacks than MD5, which has a 128-bit output and makes four passes. It is however slower than MD5.

Random Number Generators

Random numbers are a very important building block in cryptography. Operating systems sometimes do not have really random number generators: they have a period and other deterministic characteristics, and although good enough for most of our uses, they do not resist cryptanalysis. This is worrying if the security of a given algorithm depends on true randomness. Some applications, like physical circuit or link encryption (*see* Section 16.3), require that both ends have devices such as shown in Figure 17.3: they cannot be really random otherwise they could not be synchronized, but they have to look like producing random sequences. These are called *cryptographically secure pseudo-random number*generators. A good quick test of a secure sequence is that it should not be compressible. Techniques for generating good randoms are detailed in (Schneier, 1996).

Steganography

Steganography is an old paradigm for concealing data. It existed before computers, but it can assume very sophisticated forms when computer technology is available. It is based on hiding information under an apparently normal or innocent piece of data. Historical examples include: invisible ink; markings on paper, only visible under a certain light angle; tiny punctures on letters of

a printed sheet; and so forth. Steganography is not alternative to cryptography. Cryptography obscures the content of a message, but not the message, which reveals a notion of importance or secrecy to outsiders, because of being encrypted. Steganography conceals the existence of the very message, in order that outsiders do not even know that communication is taking place. In contrast, when the coding is discovered, both the existence of the message and its contents are revealed.

Key Escrow

Free use of strong cryptography has raised fears that underground forces such as terrorists, organized crime and so forth could make use of it to conceal their activities. In the U.S.A., this lead to a proposal of an **escrow** encryption system for use by the general public, where the keys of the users would be copied, split between two state agencies and safely stored in databases. The system would normally preserve confidentiality, but under a court warrant requiring the cooperation of both agencies, the key could be surrendered to the authorities, which would then be able to tap communications or decrypt files. These rules were defined in an Escrowed Encryption Standard (EES) (EES, 1994). The algorithm initially proposed for the system was a symmetric algorithm called Skipjack whose structure remained secret, and which would only be available through tamperproof hardware devices, of which at least two prototypes were produced, called Clipper and Capstone. Later, a more reasonable proposal took form, in what was called *fair cryptosystem* by its inventor (Micali, 1993). The basic ideas of the method are the following: it relies on resilient public-key cryptography; it lets users generate their own keys pairs; the private key is split in n parts and handed to *several* official agencies, with an algorithm that guarantees that the key can only be reconstructed by having at least $k \leq n$ parts. The latter condition prefigures what is called **threshold cryptography**.

17.9 AUTHENTICATION

Authentication is the process of proving the identity of a principal A, or that of proving that B acts on behalf of A. Real systems solve several facets, stronger or weaker, of this problem. For example, Alice may prove her identity as creator of a document by signing it, or as an authorized user of a service through a password. A machine Threepeeo may prove its identity to another machine by its address. These are examples of *one-way authentication*. What if Alice does not trust the server? Then, the server must also authenticate itself to Alice. This is called *mutual authentication*.

When Alice sits on Threepeeo, she delegates the authentication process on it, to execute the login program on the server. What happens if Threepeeo has been tampered with, or someone stole Alice's password? Or, what if some other machine impersonates Threepeeo's address? How can Alice or Threepeeo unequivocally prove their identity? Alice could use her cryptographic signature in the process of authentication. A PIN-protected tamperproof smart card Ar-

toodeetoo, personal to Alice, may prove its identity and indirectly that of Alice. The relationship established between Alice and Threeepeeo or Artoodeetoo are forms of what is called *delegation*. However, what if Alice lends Artoodeetoo to someone, or loses it together with the PIN? Or else, she lets her private key be stolen?

This section will discuss answers to these questions.

17.9.1 Types of Authentication

There are three basic *types of authentication*, depicted in Figure 17.17:

Unilateral	authentication is based on principal A authenticating itself to principal B
Mutual	authentication is based on principals A and B mutually authenticating themselves
Mediated	authentication is based on principal A being authenticated to principal B by a mediator T, whom they both trust

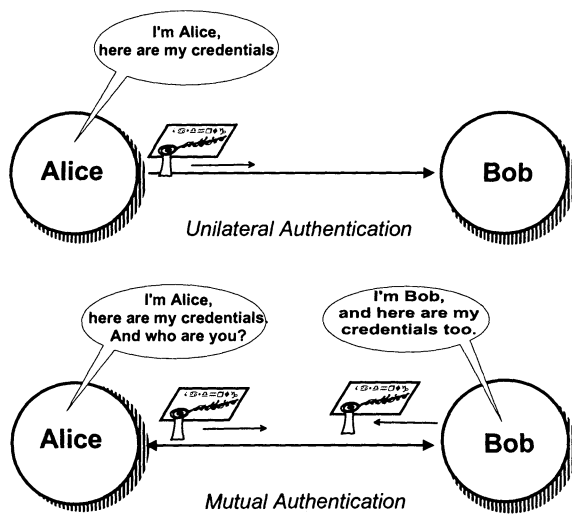

Figure 17.17. Authentication Types: (a) Unilateral; (b) Mutual

Unilateral authentication is the simplest form of authentication. A principal A (Alice in Figure 17.17a) has to produce credentials that accredit her with principal B (Bob). The term **credential** is used here in a free manner. In fact, it may take several forms, as we will see in Section 18.5. It can be a password, a signature or a cryptographic seal on one or more messages, or the proof of knowledge of any of those. Unilateral authentication is characterized by the fact that at the end of the protocol, Bob (often a server) believes it is Alice who is dialoguing with it, whereas Alice (often a user) can never be sure that

she is really talking to Bob. In certain situations, this is inappropriate, and mutual authentication should be used. Essentially, as Figure 17.17b suggests, both principals follow similar steps to persuade one another of their identities. At the end of the process, both Alice and Bob are mutually sure that they are talking to one another.

In other cases, namely in distributed systems, principals are capable of performing pair-wise mutual authentication, as defined in the previous paragraph. More precisely, all principals are capable of performing mutual authentication with a distinguished principal, a mediator. Each newcomer in the system must "learn" how to authenticate to the mediator. In turn, the mediator is capable of performing authentication (unilateral or mutual) between any two principals in the system. This scheme is obviously attractive for high-level authentication in open distributed systems. As suggested in Figure 17.17c, Trent already knows Alice and Bob, but they do not necessarily know, or trust, each other. Alice requests Trent to introduce her to Bob. Trent hands credentials to both, that will allow Alice and Bob to perform an exchange similar to that of Figure 17.17b, and get authenticated.

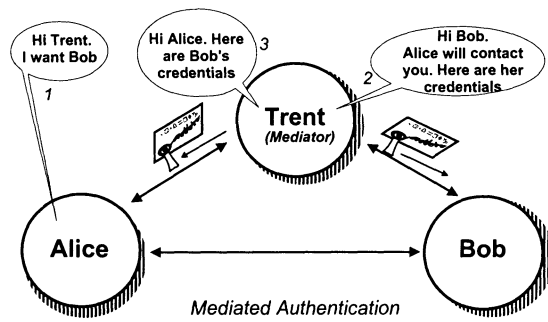

Figure 17.17 *(continued).* Authentication Types: (c) Mediated

17.9.2 Delegation

Another challenge to security related with distributed systems, is that they are modular, and geographically dispersed. In consequence, an end-user needs to have a few devices perform operations, many often in remote machines, on her behalf. The principle to achieve this correctly, that is, to authenticate other devices to act on one's behalf, is called *delegation.*

Consider the following, picking again our example from the beginning of this section: Alice is trying to login from her PC Threepeeo onto a department server S. Most of the time, all Alice does is type in her username and password. Threepeeo takes care to perform all the necessary calculations and run the necessary protocol steps. So, when Alice sits on Threepeeo, she actually *delegates* the authentication process on it, to execute the login program on the server.

The first problem is: what if Threepeeo has been tampered with? Threepeeo will do all sorts of inconvenient things using Alice's name and account, and it is hard for Alice to say she did not. This problem is caused by the fact that Alice delegated her full privileges in the organization on her computer account, *without restriction and forever.* For example, Alice is organically untitled to access the department's CD-ROM recorder, but technically, it is Alice's account on Threepeeo that has that right, always. If Alice would delegate properly, by issuing specific authorizations for specific actions, during a specific time, then she would have a powerful mechanism for securely having devices do actions on her behalf, with a much narrower window of vulnerability. Moreover, they could forward the permission to other devices, if the latter would still respect the conditions of the delegation. This is the principle of *specific delegation*, and has to do with *what* Alice authorizes others to do.

The second problem is: What if someone or something impersonates Alice or Threepeeo and issues (forges) or forwards a delegation to the wrong place, with a wrong content? This may cause trouble and will be problematic for Alice. This problem happens when the delegation techniques used by Alice do not allow her to unequivocally prove her identity, that is, authenticate herself. If Alice would authenticate properly, by issuing delegation certificates that have standalone validity, that is, remain as secure during their useful life as they were when they left Alice's hands, this problem would be highly mitigated. Use of cryptographic authentication protocols is paramount here. This is the principle of *authentication forwarding*, and has to do with ensuring that *it is* Alice who authorizes others to do things. It is the only secure way to forward a delegation.

A final problem remains, and it is: What if someone is still capable of forging the original certificate? This may cause a lot of trouble and it will be even harder for Alice, since as mechanisms get more sophisticated, Alice will have more trouble proving that they have been tampered with. The first explanation would be that the cryptography involved had been broken. The problem is however more complex than that, it has to do with what we might call the *end-to-end authentication* problem, and may occur even under the doubtful assumption that the cryptographic certificate were absolutely secure after issued. The certificate might have been forged because someone stole Alice's keys or passwords, and there is little one can do about it. Alternatively, the forgery might have taken place somewhere between Alice's keyboard and the certificate file writing, because someone might have tampered with Alice's machine in order to perform a sophisticated impersonation attack. The situation can be improved by reducing the vulnerabilities of that tiny end path from Alice, the person, and her first delegation, the device that issues certificates. The protected tamperproof device Artoodeetoo where Alice enters a PIN or a password and which issues the certificate or cooperates with her machine for that purpose, is a good advance. It is practically end to end, so nothing can get in the middle. Of course, Alice could lose Artoodeetoo, with her PIN written on the back. Well, then we would need serious end-to-end stuff. Short of implant-

ing a computer chip into Alice's brain, some advanced biometrics technology such as iris scanning can help, because it is still difficult for Alice to lend or lose her eyes. However, there is no definitive "cryptocratic" solution, because you should not forget the comments in Section 16.1: more than technology, the behavior of humans is central to achieving security.

17.9.3 Key Distribution

These are the basic authentication paradigms. Before concluding, let us introduce a problem related with authentication: key distribution. Cryptographic protocols need keys to operate, genuine keys have to get to the participants, and sometimes these keys are established just for one session. In some cases, it is convenient to combine key exchange with authentication, to avoid impersonation and forgery attacks.

Key distribution in general is a paradigm crucial to any cryptosystem, the has two facets. In fact, in all the protocols we have just discussed, we have considered that *long-term key distribution* had already taken place, that is, the keys were already with their legitimate owners and users. This includes long-term secret keys shared between two principals or public/private key pairs used for the purpose of signature and authentication. Another facet of the key distribution problem is what is called *short-term key exchange*, this particular facet of key distribution being understood as the ad hoc, frequent and on-line exchange of keys for temporary use, e.g., with the purpose of setting-up a temporary session or communication channel. These keys are called **session keys**. This is obviously a sensitive issue, since, recalling our introductory notions on *risk*, it is an operation subject to a very high level of threat. In consequence, any vulnerabilities in the protocols will be highly exposed.

The long-term key distribution problem is the bootstrap problem of most cryptographic systems, from authentication to communication systems. All must start with the first shared key or pair of asymmetric keys. Long-term public keys may be distributed by the owner to the people she interacts with, via some trusted off-line mechanism. After a few keys are in place, they can be used to sign new key files, which we call certificates, in a transitive signing chain of mutual trust. This is used for example in PGP (*see* Section 19.1). Public key certificates can travel securely over the network. Exchanging shared secret keys for symmetric cryptography is a harder problem. Unlike public key distribution, secret keys in transit not only risk corruption but also disclosure. The "primordial secret" has to be sent by some other means, for example a combination of mail, telephone, and fax. After that, principals can exchange new keys under the cover of this first key.

17.10 ACCESS CONTROL

Access control is concerned with validating the access rights of users to resources of the system. It can be seen at different levels. One may control the access of end users to database records, control the access of users and programs to

files, or control the access of program instructions to segments and pages of a computer system. The first type of control is implemented by the database management system, the second by the operating system and the third by the microprocessor. There are several approaches to access control, and systems in general possess some form of such control. The advances in computer security have contributed to a sophistication of access control mechanisms and protection models. In what follows we present the main *access control mechanisms*: *access control lists*; *capabilities*; *access control matrices*. Mechanisms are used to implement *access control models*, which essentially reflect a specification of how principals should access computer resources.

A *protection domain* is the set of resources that lie under the realm of an access control mechanism. They could for example be the O.S. resources depicted in Figure 16.4. The several ways resources can be accessed are called *access rights*: read, write, execute, lookup, create, delete, truncate, append, insert. Resources are generically called *objects*: pages, files, processes, devices. The entities trying to access are called *subjects*: humans, programs.

17.10.1 Canonical Access Control Mechanisms

The most popular access control mechanism, used by many operating systems, is the **access control list** or **ACL**. The ACL mechanism is defined by the following:

- each *object* has a list (ACL) of the subjects that may access it;
- each element of the list is a pair ⟨*subject,rights*⟩, where: *subject* is an *Id* of a user, a process, or a group; and *rights* is the enumeration of the access rights granted to this subject on that object, usually a bit mask (e.g., `xrwd` for UNIX, i.e. execute, read, write, delete);
- the ACL is protected by the system against unauthorized modification.

Among the advantages of the ACL approach, we note that the access control of an object is centralized in a single structure. Besides, when many subjects share the object, they are simply grouped in specific or generic groups (e.g., `world` in UNIX is "all subjects"). However, in security terms, grouping can present a vulnerability, since it hides access rights of individual subjects and is thus prone to granting access rights to the wrong subject, for example when increasing the access rights of a group. On the other hand, discriminating every subject's rights would be impracticable, since it would soon overload the ACL. Another well-known access control mechanism is the *capability*. The capability mechanism takes a different approach from the ACL one:

- each *subject* has a list of the objects of a protection domain which it may access;
- each entry of the list is a capability: a pair ⟨*object,rights*⟩, where: *object* is the *Id* of a resource; and *rights* is the enumeration of the access rights granted to that subject on this object, usually a bit mask;
- the capability is protected by cryptography against unauthorized modification or forging.

Compared with ACLs, capabilities are oriented to subjects rather than to objects. For that reason, they do not exhibit the problems of ambiguity of subject access rights and of overloading: a subject must have a capability for any object it wishes to access; and only has capabilities for the objects it accesses. Besides, since a capability is protected, it can be securely transferred or copied to other subjects, yielding delegation of access rights. Note that capabilities do not reside permanently with the subjects, they are requested to a control entity when necessary. This entity has a central list of subjects versus access rights to objects, which may be an ACL, or an access control matrix, that we discuss below.

17.10.2 Access Control Matrix

A formal model of access control (Lampson, 1974) requires a more complete statement of access rights, in the form of an *access control matrix* or *ACM*. The ACM mechanism is defined by the following:

- each *subject* has a list of the objects of a protection domain which it may access;
- each *object* of a protection domain has a list of subjects that may access it;
- these two lists form a matrix where each entry is a triple ⟨*subject,object,rights*⟩, where: *subject* is an *Id* of a user or process; *object* is the *Id* of a resource; and *rights* is the enumeration of the access rights granted to the subject on the object;
- the ACM is protected by the system against unauthorized modification.

An access control matrix will normally be sparse, since subjects have on average access to few objects. A simple example is shown in Table 17.3. Objects are the financial, the personnel, the stock and production files of the information system of an organization. The general manager only has read access to all files. Although she is the most important officer, she does not need to have greater rights in order to perform her function. This feature is very important, and is called the *least privilege* rule (Saltzer and Schroeder, 1975). Another very important rule in protection of information is the *need-to-know* rule: regardless of his position in the organization hierarchy, a subject should only be given access to the information needed to perform his work. The example follows this rule in general: the managers only have rights over the objects related to their functions. In consequence, a subject should be given *the least amount of rights to the least number of objects* possible in order to perform her job.

Searching a column of the matrix of the figure is equivalent to searching the ACL of an object for the rights of a given subject. Searching a row of the matrix is equivalent to finding the capability of a subject for a given object.

17.10.3 Discretionary Access Control

Until now, we said nothing about the form that the ACL, capability, or ACM may take. Who defines what a subject can do with the objects she creates or owns? Most systems do not impose any a priori restriction on such definitions.

Table 17.3. The Access Control Matrix

	Objects			
Subjects	finance	personnel	stock	production
General Mgr.	r	r	r	r
Finance Mgr.	rwcd	—	r	r
Production Mgr.	—	—	rw	rw

A subject can give whatever access rights she decides to the objects she creates or controls. We say these systems where control is subject-based follow a *discretionary access control (DACC)* policy, which is essentially a lack of policy for that matter. Note that this does not imply any lack of quality or effectiveness of the access control mechanisms just discussed, we are now at a slightly higher level of discussion: what are the rules to fill in the access control tables, that is, to decide which objects subjects or groups of subjects may access, and in what manner. The set of these rules is called the *access control policy.*

Consider subject s and object o. In a DACC policy, the access rights r granted to s on o solely depend on an ad hoc strategy F of the administrator or the owner of o, towards s, decided on a case-by-case manner:

$$DACC: \quad \langle s, o, r \rangle = F(s, o)$$

The DACC policy as defined by F can change dynamically, and can vary at the administrator's or owner's will. Discretionary access control is obviously the policy followed by most commercial-grade operating systems and database management systems.

17.10.4 Mandatory Access Control

Discretionary access control makes it impossible to enforce an access control policy, since access rights may change dynamically according to the current rights and the will of the users. If there is no formal and automatic way to check access control of authorized users, there is also no way to control malicious software, such as Trojan horses. A more restrictive policy is required, where users are not allowed to change access rights of objects, even if they own them, if that change is against some highly defined access control policy. We say these systems follow a *mandatory access control (MACC)* policy[1].

In order to follow a MACC policy, each subject or object must have a static security *class* or *label* (also called classification or clearance). Consider subject s with security class $C(s)$, and object o, with security class $C(o)$. The access

[1]The acronyms used are normally DAC and MAC, but we wish to avoid any confusion with other terms such as Message Authentication Code, MAC.

rights r granted to s on o solely depend on a static strategy F, which is a deterministic function of the security classes of both, and not of the particular subject or object. In consequence, the owner of o cannot change the access rights for s, if that violates the access control policy rules:

$$MACC : \langle s, o, r \rangle = F(C(s), C(o))$$

Now that you have understood the difference in level of abstraction between access control mechanism and access control policy, we may introduce an even higher level of abstraction: the *security policy.* A security policy is the top, human level, set of rules to enforce security in an organization. We are obviously interested in computer-supported organizations. A security policy dictates, amongst other things, the access control policy, that is, the rules to form F, which in turn will be implemented by the access control mechanisms. The access control policy is either a MACC or a DACC policy, or a MACC policy complemented by a DACC policy. This whole strategy is concerned with protection models, that will be studied in Section 18.7.

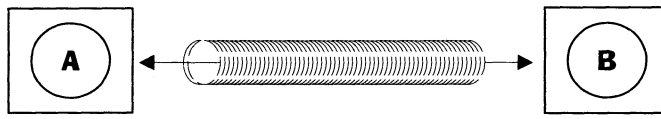

Figure 17.18. Secure Channel

17.11 SECURE COMMUNICATION

In an analogy with reliable communication, secure communication means ensuring that two or more principals (users, machines, protocols), communicate with security, despite the occurrence of malicious faults (attacks). There are essentially two ways of achieving *secure communication* over insecure media: secure channels and secure envelopes.

Indeed, a sufficient condition to achieve secure communication, is that principals are capable of building a *secure channel* between them. The channel is an abstraction that can be realized physically (*see* Figure 16.2) or virtually (*see* Figure 16.3). Observe Figure 17.18. Suppose A and B are connected by a pipe as shown, so that the pipe goes directly from A's mouth to B's ear, with no bifurcation. The pipe is completely opaque, so that no one can see what it carries. Finally, the pipe's material is also hard, so that no tool can penetrate it, inject or suck things from the inside, without that being noticed. That's a secure channel, and in fact the attributes we have just exemplified can be translated into the following properties:

authenticity - what B receives was sent by A, who cannot deny;

confidentiality - only B reads what A sent;

integrity - what A sent cannot be altered without detection.

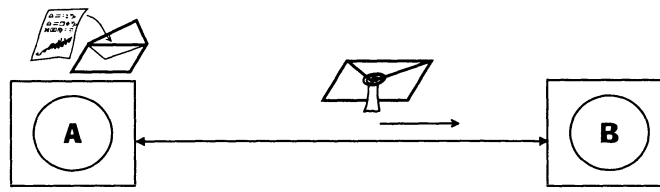

Figure 17.19. Secure Envelope

A secure channel is so to speak a form of *immediate* communication, and applies to the traditional forms of computer communication or telecommunication links, such as local area network protocols, TCP/IP, X.25, an so forth. By contrast, we identify a form of *deferred* communication, where it is desired to secure messages whose delivery is sporadic and deferred in time. Electronic mail is a perfect example of this. In that case, messages must have standalone security, that is, be wrapped in a *secure envelope*, which should enjoy the same security properties of the secure channel above: authenticity; confidentiality; integrity. Figure 17.19 depicts the principle of the secure envelope. A message is signed, put inside an opaque envelope, and the envelope is sealed. The whole packet has *standalone security* while traveling through an insecure medium: it cannot be changed without that being noticed (integrity); it cannot be read (confidentiality); and once opened, the signature can be verified (authenticity).

17.12 SUMMARY AND FURTHER READING

This chapter discussed the main paradigms concerning security. It addressed basic concepts of security, such as the trusted computing base and the foundations of modern cryptography. A detailed study was made of the main cryptographic paradigms: symmetric and asymmetric cryptography; secure hashes and message digests; digital signature; and digital cash. Authentication, access control, and secure communication complete the set of paradigms studied in this chapter. A remarkable text on secure channels, simultaneously simple and comprehensive, is Needham's chapter in (Needham, 1993). Additional cryptographic protocols can be found in (Schneier, 1996; Ryan et al., 2000; Menezes et al., 1999). A reference publication for real random numbers is the Rand Corporation million-number table (RAND, 1955). For an in depth discussion on digital cash properties, see (Okamoto and Ohta, 1992). Digital cash and guardians are detailed in (Brands, 1995), (Chaum, 1992), and (Boly et al., 1994). For further material about steganography, see (Wayner, 1993). A comprehensive treatment of authentication is found in (Kaufman et al., 1995). (Pfleeger, 1996) and (Abrams et al., 1995) do a thorough study of protection and access control mechanisms and security policies. The principles of intrusion tolerance, as opposed to prevention, are laid down in (Deswarte et al., 1991). For discussions on secure group communication see (Schneier, 1996; Reiter, 1996). Another challenging problem is maintaining keys in a group communication system, where members enter and leave (Steiner et al., 1998).

18 MODELS OF DISTRIBUTED SECURE COMPUTING

This chapter aims at providing the architect with a global view of the problem of security, by showing where the paradigms learned in the previous chapter, fit in the several models of distributed secure computing. It starts by discussing the main classes of malicious faults and errors expected in computer systems, and in distributed systems in particular— that is, attacks and intrusions. Then, it equates the main frameworks and strategies for building secure systems— authentication, secure channels and envelopes, protection and authorization, and auditing— as a form of bridging from the detail of paradigms to the global view provided by models. Finally, specific models for distributed secure computing are presented.

18.1 CLASSES OF ATTACKS AND INTRUSIONS

18.1.1 Computer Misuse

Not all types of security-related incidents are perpetrated by hackers (nonusers) through sophisticated techniques. We have already discussed the importance of negligence or occasional abuse of authorized users. *Computer misuse* designates actions and attitudes, by users and nonusers, aimed at impairing the security of computer systems. Abrams et al. introduce a comprehensive classification of computer misuse with increasing degree of severity and sophistication (Abrams et al., 1995):

P. Verissimo et al. *Distributed Systems for System Architects*

Human error	Accidental human mistake of an authorized user causing a vulnerability that may lead to intrusion. For example, giving world read/write permissions to a confidential file, or creating an account without password
Abuse of authority	Intentional action of an authorized user abusing the authority granted by his activity. For example, a bank teller setting-up schemes of bogus transactions that leak a few cents each time to his personal account
Direct probing	Attack made by an unauthorized user (or nonuser) to a system by means of passively exploiting existing vulnerabilities with the aim of intrusion. E.g., entering through a forgotten account with default password, or using a stolen or guessed password
Software probing	Attack made by a nonuser to a system by means of passively exploiting existing vulnerabilities with the help of specially built malicious software, with the aim of intrusion. For example, use of a *Trojan horse* that pretends to be the login program, logging inadvertent users while it steals all their passwords
Penetration	Attack made by a nonuser to a system by means of actively exploiting existing vulnerabilities in the protection mechanisms of the system, normally with the help of specially built malicious software, with the aim of intrusion. E.g., sending a malicious HTTP request that confuses the HTTP server, leading it into giving an anonymous intruder total (root) control
Subversion	Attack made to a system, either at design or runtime, by designers, or by authorized or unauthorized users, by means of covert and methodical undermining of the protection mechanisms of the system, with the aim of continued intrusion. For example, by modifying operating system programs in order to introduce *trapdoor* that covertly perpetuate the intrusion

Human error and *abuse of authority* have little to do with computer technology, since a computer can hardly tell whether a legitimate user is using or abusing it. In fact, current solutions to this kind of misuse lie with ensuring users behave as they should, by means of inspection and auditing. *Probing*, whether manual or automated with the help of software, involves the discovery and/or direct use of vulnerabilities of the system. The attacker passively exploits the way certain functions are wrongly configured, or turned off, or else uses information obtained by other means (this is akin to a burglar entering a facility with a stolen key, pretending to be a legitimate user). *Penetration* bypasses the protection mechanisms, questioning their effectiveness. In consequence, it is an active and very aggressive attack against the implementation of the system's security policy (this is akin to the same burglar entering the facility, disabling the alarm, and guessing the safe combination). *Subversion* is the ultimate attack. It can be made either at design time— by one of the engineers, or at distribution time— by inserting malicious patches in down-

loadable code, or at runtime— after a successful intrusion. For example, the latter can be done by replacing crucial operating system programs— such as login, remote login, password change, auditing and monitoring, command line editor— with a complete kit of malicious programs with trapdoors and Trojan horses. Very often, a computer system is threatened at different times and in a pre-planned sequence. In generic terms, this is an **intrusion campaign**, and its consequence for the targeted organization is a *security hazard.* In general terms, intruders use a combination of the described types of computer misuse, combining human related misuse with attacks exploiting different vulnerabilities. Technically, it is usual to separate attacks between active and passive, with regard to the kind of interaction between the intruder and the target system.

18.1.2 Active Attacks

Active attacks are characterized by aggressive attempts to penetrate the system, disrupt its operation, and/or steal, modify or destroy data. Attacks may be directed at networks, machines, services or information.

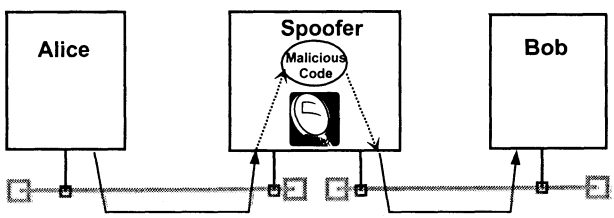

Figure 18.1. Spoofing Attack

For example, penetration attacks may be directed at an internal network by breaking through a firewall, or at a host, by defrauding its access control mechanisms, and stealing information. Penetration may instead be directed at a specific protocol, to gain control of the interaction, such as a home banking transaction. **Spoofing** is a specific form of such an attack, whereby a malicious host intercepts communications between two participants, changing its contents dynamically. The modification may take several forms: insertion/deletion or replay of whole messages, on-the-fly modification of message content. This last form is depicted in Figure 18.1. This attack can be used to penetrate cryptographic communication, or to append malicious software to downloaded programs. A well-known attack on Internet protocols is address spoofing, whereby source addresses of IP packets are modified to make them look like they are coming from somewhere else. The attack may be prepared by a previous attack on a DNS server that modifies a whole set of $\langle name, address \rangle$ pairs.

Disruption attacks are addressed at resources, for example communications jamming or bringing an operating system down. These attacks may be directed at specific services, hence the name **denial-of-service** attacks. A typical such attack is email spamming, whereby email servers are flooded with phony mes-

sages, to the point of being brought down. Modification attacks concern the contents of information repositories, e.g., databases, by modifying or destroying them. Specific Web-based services have also been attacked lately as a form of sabotage. These are either disruptive attacks, or modification attacks addressed at the content of the pages.

Attacks can be made either directly or through malicious software, such as viruses, worms, bombs, trojan horses and trapdoors. A **virus** is a software module that is appended to genuine programs. When the program executes, the virus is activated and performs the planned attack, often a mix of penetration, modification and disruption. Viruses, besides attacking the system, try to reproduce themselves, by infecting other programs in the machine.

A **worm**, unlike a virus, is an autonomous program which in general performs penetration attacks. Worms, once arrived at a host, install themselves taking advantage of vulnerabilities of the victim O.S., and prepare the assault of the next host. Besides migrating, they can copy themselves like viruses do.

A **bomb** is a disruption attack consisting of inserting a malicious software module inside a program or a system that perform some useful function, such as a database server. When activated, it will do something destructive, from blocking the server to deleting the database. The detonator is built by modifying the original program or O.S. configuration, so that the bomb is fired by a logical condition (logic bomb) or at a given date (time bomb). Bombs have been used very often for blackmailing companies.

A **Trojan horse**, or trojan for short, is a program that replaces legitimate system or user programs, and performs attacks on the system. Its name derives from the fact that it also executes the function that the original program was supposed to, in order not to be detected. A **trapdoor** is a special kind of trojan, normally a software module inserted inside a system, either during or after design, in order to subvert the access control mechanisms and let non-authorized users in.

A **trapdoor** is a piece of code inserted in a software module, either during or after design, providing a means of accessing a system other than the usual access procedure. A special kind of trojan consists of an access control mechanism with a trapdoor inserted, in order to let non-authorized users in.

A **covert channel** is a subtle form of information-theft attack. It consists of an indirect communication channel outside the reach of the access control mechanisms, which can be used to disclose information to an unauthorised user through a non-detectable means. A primitive channel would consist of having the leaking program encode the leaked information inside digital pictures served by a Web server on the same host (we briefly mentioned the underlying technique, called steganography, in Section 17.8). These channels are sometimes extremely subtle and strange, such as having the leaked information encoded as a pattern of disk usage (rythm of access, pattern of allocation), to be read and decoded by any non-privileged program in the same host.

18.1.3 Passive Attacks

In general, we term passive attacks those which, unlike active attacks, do not require an explicit action against the security mechanisms of systems or the integrity of their information. The hacker merely uses what is within reach, sometimes materialized or made possible by human error, or abuse of authority, such as reading files incorrectly open to the world, or logging in with a guessed password. Typical passive attacks are reading attacks, often aimed at gathering information for planning an active attack, for example, following the execution of a protocol, or logging login/password pairs passing on a network.

Probing, to explore basic system vulnerabilities, is the basis of many passive attack techniques. Internet IP discovery and port scanning is the basic technique to unveil vulnerabilities of networked installations. Password guessing is the basic technique against hosts using password authentication. The same idea applies to key guessing, for cryptographic protocols. These attacks are normally addressed at an encrypted form of the password or key, e.g., such as found in a password file, by exhaustively trying off-line the possible combinations, until a match is found with the encrypted item. This is called *off-line guessing*, and practical attacks are made with programs that test combinations of a glossary of probable words to be used by the password or key owner. Such a **dictionary attack** includes dictionaries of the language, and personal information that may be grabbed, for example, from the subject's Web page. These programs are as useful to hackers as they are to system managers, who may detect poor passwords and instruct the owners to change them. A popular such tool is the `crack` for UNIX.

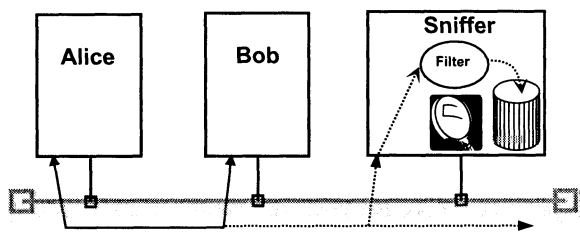

Figure 18.2. Sniffing Attack

Sniffing is the typical passive attack on a network, whereby the intruder exploits the broadcast nature of propagation of a medium, such as wireless communications or local area networks. The attack over a LAN is depicted in Figure 18.2, where the sniffer is a machine whose LAN adapter is configured to receive all the passing frames (promiscuous reception). Information received is filtered and stored in disk, to be used later. Sniffers are normally network management and debugging tools, such as `tcpdump` for the Internet. However, if misused, they become very nasty artifacts, because of their stealth nature: it is very hard to trace a sniffer, since it is absolutely passive.

File or database **snooping** is a passive attack directed at stored data, very frequently enabled by erroneous configurations and other vulnerabilities such as default passwords or wrong access permissions, easily discovered by methodical but discreet probing, also called doorknob rattling.

Although less destructive than active attacks in appearance, passive attacks are much more difficult to trace, exactly because of their frequently stealth nature, and are very often the first line of a final active attack.

18.1.4 Intrusions

In terms of effect, the results of successful attacks to computer and communication system components fall into one of the following general *intrusion categories*:

Resource Theft	Unauthorized use of computational or communication resources
Resource Disruption	Malicious disturbance of the service provided by computational or communication resources
Information Theft	Unauthorized interception, disclosure and/or use of information such as data and/or software, in computing or communication systems
Information Modification	Unauthorized disturbance of the content of information in computing or communication systems, by alteration, deletion, and/or forging

This classification emphasizes two important issues: the separation between computing or communication *resource* security, addressed by the first two, and *information* security, addressed by the last two; and the separation between *stealing* and *disturbing*.

These distinctions are important in defining strategies, since different organizations may have different policies about the relative importance of information versus resource security, or confidentiality versus integrity and availability. It is also important because different techniques apply to each category of intrusion.

18.1.5 Attacking a Cryptographic System

Let us analyze now specific attacks to cryptographic systems, which can take several forms: attacks on the algorithm; attacks on the messages; attacks on the keys; attacks on the protocol.

Any or all of these attacks can obviously be combined in an intrusion campaign against a cryptographic system. The algorithm is a crucial component of the crypto system. One of the main reasons why algorithms should be public is in order for cryptologists to know its structure, test its security, and eventually build trust on it, if it is not broken, or **cryptanalyzed**. Attacks on algorithms normally follow a few known patterns, or a combination thereof, based on what the cryptanalyst knows. In a *cyphertext-only* attack, the attacker only has ac-

cess to encrypted material. This is the basic attack, but yields very little to work with for resilient algorithms. The attacker needs more information to be effective, for example, he can manage to get hold of cyphertext blocks of which he knows the plaintext (for example, by acting as a legitimate user). This is known as a *known-plaintext* attack. Getting more sophisticated, he may try to manipulate the cryptosystem such that it encrypts selected blocks of data, and then collect the corresponding cyphertext. This is called a *chosen-plaintext* attack. This material will serve to look for vulnerabilities that invalidate one or more of the rule-of-thumb properties of a good cryptosystem listed in Table 17.1 of Section 17.2.

Attacks on messages have to do with the attacker playing with cipher blocks in transit in a secure channel or envelope, by changing, replaying or reordering whole blocks. The reuse of a signed text or digital cheque, or the forging of a financial transaction using blocks of genuine transaction, are examples of such attacks. Ciphers should be strengthened with information used only once, that is called **nonce**, such as time, sequence numbers and random quantities.

Attacks on keys have two facets. The first addresses the fundamental limit of computational ciphers, which is the feasibility of making a brute-force attack, by testing all possible combinations. The second facet has to do with guessing attacks. Long keys may render the first attack infeasible, but a cryptosystem can fall long before that limit is reached, given a poor choice of keys. Keys are susceptible to dictionary attacks as are passwords. Even brute-force attacks may become fast enough, if the key owner uses a subset of the character space (e.g., lowercase, alphabet/only), since the search space is decreased.

A crypto system is materialized by a protocol. This protocol may have logic vulnerabilities, that have nothing to do with the algorithm but with the way it is implemented in a real system. The attacker studies the protocol, and performs active attacks with the aim of confusing it and cause it to take wrong decisions, such as surrendering a shared secret key, or letting the attacker into a server without completing his authentication. For example, a spoofing attack can undermine an otherwise correct crypto system. The types of attacks described show that a cryptosystem does not live just on a good algorithm. In fact, a system using an unbreakable algorithm may fall under naive message formats, poorly chosen keys, or protocol bugs. In consequence, an architect should always have a systemic view towards solving security problems.

18.2 SECURITY FRAMEWORKS

After discussing the possible threats to system security, we spend this section analyzing the main frameworks with which the architect can work in order to build secure systems, introducing some of the models addressed later in this chapter, such as authentication and key distribution with key distribution centers and certification authorities, hybrid cryptography for secure channels and envelopes, or protection via discretionary or mandatory access control policies.

18.2.1 Secure Channels and Envelopes

Secure channels and secure envelopes are basic paradigms of secure communication studied in Section 17.11. They form a framework for setting-up distributed processing services with security, such as remote sessions, file transfer, RPC, Web servers, email, messaging services, on-line transactions, and so forth.

Secure channels are normally set up for regular communications between principals, or communications that last long enough for the concept of connection to make sense. For instance, file transfers or remote sessions, among the ones just mentioned. They live on a resilience/speed tradeoff, because they are on-line, and may use combinations of physical and virtual encryption. Secure channels adopt per-session security, and normally use symmetric communication encryption, which provides the best performance, but requires attention in terms of resilience. Practical protocols should protect their keys, and reveal as little as possible about the communication channel that may be useful for a cryptanalist.

Secure envelopes are used mainly for sporadic transmissions, such as email. They resort to per-message security and may make use of a combination of symmetric and asymmetric cryptography (also called hybrid) as a form of improving performance, especially for large message bodies. However, they are not so susceptible to the resilience/speed tradeoff, given their deferred nature. This is relevant for ultra-sensitive communication, where longer keys and asymmetric encryption may be used, to increase resilience of the ciphertext.

Section 18.4 is going to discuss practical issues of cryptographic protocols. Section 18.10 addresses the establishment of secure channels and the generation of secure envelopes, and continues building on top of that, presenting models for remote sessions, remote client-server with procedure call, and electronic mail.

18.2.2 Authentication

There are three forms of authentication paradigm, as studied in Section 17.9, and they make sense in different situations. Essentially, unilateral or mutual authentication are chosen depending on how important it is, for each principal, to identify its peer with certainty. There will be systems where authentication is compounded with the service itself, such as a centralized system with remote and distributed access. Unilateral authentication by password is the most used. However, when threats are expected, some form of cryptography should be envisaged. It is then common to provide users with a secret key shared with the server, so that they can execute such a protocol, performing either unilateral or mutual authentication. When there are several services and many users, this becomes impractical. Then, mediated authentication is one solution, where a distinguished service is created just for the purpose of authentication: a key distribution center (KDC). Now, users only share secrets with the KDC, and when they wish to authenticate to other services or users, they use its mediation. In large systems, an alternative may be to provide users and services with public/private key pairs, so that every public key can be found in a specialized

server, a certification authority (CA), that provides key certificates signed by it. Powerful asymmetric signature and encryption protocols can then be used for authentication, key distribution and long-term encryption.

A remaining issue is the significance of a given authentication process: who is being authenticated? When we request a call-back number in a computer dial-up, we believe that the number represents a user. When we use link encryption, we just trust the secure channel between the link extremities, not what is beyond the link. However, when we use virtual circuit encryption, we trust an end-to-end secure channel, for example between a client program and a server program. When we provide the user with authentication gadgets, such as smart cards or biometric devices, this is because we do not trust the client program to stand for the user at all times: the host can be intruded, an intruder can logon impersonating the user. We have just discussed delegation, that we will address together with authentication and key distribution in Sections 18.5 and 18.6.

18.2.3 Protection and Authorization

Protection, a fundamental framework in secure computing, is about restricting the access to and use of information and programs, to authorized users. As such, the central paradigm is *authorization.* The access control and trusted computer base (TCB) concepts form the underlying basis on which to build protection models. Access control, that we studied in Section 17.10, is performed by assessing the rights of subjects to access objects, those rights being defined by access control lists or matrices, or by capabilities. Normal operating systems however, do not provide the adequate guarantees of trustworthiness to run sensitive access control mechanisms. The TCB, that we introduced in Section 17.1, provides the necessary notion of secure, tamperproof base.

One fundamental model for protection is the reference monitor (RM) model, based on running all access control functions on a TCB, and obliging all requests for system resources to go through the RM. The RM does not necessarily stipulate how access control to objects should be given to subjects. However, in order to fulfill high-level security policies in a verifiable way, access control mechanisms should not be configured by users or administrators at their own will, which is how they are often implemented, but instead dictated by access control policies (sets of rules), such as the statement that "an unclassified subject cannot read from a top-secret object". The former and the latter policies are called respectively discretionary and mandatory access control policies.

Despite what we just said, protection (taken in a broad sense) starts with good architectural procedures. Physical separation by architecture and topology of the network is the first step to protecting a distributed system. Secondly, machine protection, further reducing the level of threat by reducing the exposure of services, for example, by putting restrictions on remote access, dial-up, or the way administrators perform sensitive services. Firewalls come next, as a means to enforce both physical and logical separation, since containment is achieved at both architectural and protocol levels. These measures can be further refined through the separation of concerns between machine protection

and data protection. It should still be difficult for an intruder that breaks into a machine, to further get to users' or services' data. This has to do with the logical architecture of services and applications, and the adequate use of distribution, fragmentation, replication and cryptography. All these notions will be expanded in Sections 18.7, 18.8 and 18.9, where we will study models for protection of systems, from architecture, including firewalls, to formal models.

18.2.4 Auditing and Intrusion Detection

Logging system actions and events, or in other words, performing an **audit trail** of the system, is a good management procedure, and is routinely done in several operating systems. It allows a posteriori diagnosis of problems and their causes, by analysis of the logs. Audit trails are a crucial framework in security. Not only for technical, but also for accountability reasons, it is very important to be able to trace back the events and actions associated with a given time interval, subject, object, service, or resource. Furthermore, it is crucial that all activity can be audited, instead of just a few resources. Finally, the granularity with which auditing is done should be related with the granularity of possible attacks on the system. Since logs may be tampered with by intruders in order to delete their own traces, logs should be tamperproof, which they are not in many operating systems. A broader perspective on this subject is provided by **intrusion detection systems (IDS)** (Debar et al., 1999). An IDS system is a supervision system that follows and logs system activity, in order to detect and react against attacks and intrusions, preferably in real-time, that is, with low latency.

18.3 STRATEGIES FOR SECURE OPERATION

Strategy is conditioned by several factors, such as: type of operation, acceptable risk, price, performance, available technology. Technically, besides a few fundamental tradeoffs that should always be made in any design, the strategy of the architect for the design of a secure system develops along a few main lines that we discuss in this section.

18.3.1 Fundamental Tradeoffs

There are a few fundamental tradeoffs to be considered by the systems architect:

- cost vs. effectiveness
- performance vs. security
- robustness vs. lifetime
- degree of vulnerability vs. level of threat
- cost of securing vs. cost of intruding
- prevention vs. tolerance of attacks
- prevention vs. detection of modification
- detection/recovery vs. prevention of fraud

- cost of security vs. cost of breaking

The cost versus effectiveness tradeoff may be patent for example in how far goes the effort to implement a real TCB, since there is a spectrum of alternatives from purposely made, high-coverage security kernels, to adapted, and necessarily more fragile, commercial operating systems. Security can mean adverse performance in several instances. For example, robustness of a cryptographic channel depends on the key length, but the longer the key, the slower the channel. Asymmetric cryptography is deemed more robust than its symmetric counterpart, however it is much slower, by approximately three orders of magnitude (*see* Table 18.1). The cost versus effectiveness tradeoff yields to the need for performance when both speed and robustness are required, implying the use of cryptographic hardware, versus software-based cryptography, inexpensive and robust, but much slower (*see* Table 18.1 for typical speed-up factors). On the other hand, robustness of a channel is inversely proportional to its lifetime, since more material can be revealed to an attacker. Long-lasting channels should refresh their context regularly (e.g. keys).

The degree of vulnerability of system components cannot be reduced arbitrarily. In other words, *vulnerability removal* must be balanced with the reduction of the level of threat the system is subjected to, in order that the risk of operation remains acceptable. Reducing the level of threat is the path of *preventing attacks*, whereby the system gets protected from certain attacks. Attack prevention is advised for long-term secrets, that is, information that must remain confidential for a long time. In this case, even a cyphertext-only attack presents a risk if one is not certain that it will resist a brute-force analysis during the lifetime of the confidential information. However, most of the time acceptable risk does not imply absolute intrusion-free operation, the cost of which may be overwhelming. Instead, it is acceptable for the cost of intruding to be significantly higher than the value of the service being offered. This may justify the use of weaker but faster, cheaper or simpler cryptosystems, for example. However, this approach has its limitations: it is impossible to guarantee that all attacks are prevented, given the unpredictable nature of attackers; it is sometimes not possible to prevent attacks at all, given the open nature of some applications. So, we better seek for an alternative to vulnerability removal plus attack prevention, which we may call *intrusion prevention*. Essentially, this means admitting that attacks will take place and lead to intrusions, what we might call *intrusion tolerance*. The latter consists of letting the system be attacked and intruded upon, but providing it with the means to resist and avoid failure of the security properties.

Which properties to secure represents another class of tradeoffs. For example, integrity can mean prevention of modification, or its detection. The latter is ensured with cryptographic techniques. However, the former requires protection, and sometimes redundancy. Computer fraud has legal implications that make the tradeoff between detection/recovery and prevention a delicate one. Fraud prevention may reveal itself cumbersome (more complex and slower

Table 18.1. Figures of Merit of Several Cryptosystems

	Variants	
Algorithm	**SW: Speed(Pr/Key)**	**HW: Speed(Pr/Key)**
DES	1.2Mb/s (66M4/56b)	512Mb/s (32M/56b)
DES	9.3Mb/s (100M5/56b)	640Mb/s (-/56b)
3DES	0.4Mb/s (66M4/56b)	214Mb/s (-/56b)
IDEA	2.4Mb/s (66M4/128b)	177Mb/s (25M/128b)
RSA encr	5Kb/s (66M4/512b)	220Kb/s (-/512b)
RSA decr	320Kb/s (66M4/512b)	
RSA sign	0.16s (S2/512b)	
RSA ver	0.02s (S2/512b)	
DSA sign	0.20s (S2/512b)	
DSA ver	0.35s (S2/512b)	
MD5	5.9Mb/s (66M4/-)	315Mb/s (-/-)
SHA	2.6M/s (66M4/-)	253Mb/s (-/-)

protocols), whereas fraud detection is followed by a recovery process which normally takes place outside the system and is thus lengthy.

Table 18.1 gives some useful insight on what is behind some fundamental tradeoffs just discussed. Speeds of cryptographic material are presented (either in Mb/s or sec.), differentiating between types of algorithms, and hardware and software implementations for several key lengths (main sources: (Garfinkel and Spafford, 1997; Lampson, 1993; Schneier, 1996)). Processor codes are in the form xMy, where x is speed in MHz and y is: 4- i486; 5-iPentium; S2- SUN Sparc II. Key lengths in bits. We have tried as much as possible to harmonize figures, and ended up using the today obsolete 66MHz i486 (66M4). However, we compare with DES figures for the 100MHz Pentium (100M5), to give you an idea of the evolution. We would have to review the book every 6 months to cope with technical progress. It should be easy to extrapolate to faster machines, if you have good comparative benchmarks at hand. Look for computation intensive ones. RISC architectures, for example, do remarkably well because of pipelining and internal parallelism: note that the speed-up in DES from 66M4 to 100M5 is almost 8 times, quite a bit higher than the clock speed-up.

18.3.2 On Keys and Passwords

Keys and passwords are the most fragile components of a secure system, and we know a system breaks by its weakest link. So, this section gives a few hints on their correct use. We will use 'key' to denote both key and password here, unless a distinction is necessary.

A brute-force attack is the ultimate barrier on a key or password. A pool of 200 hardware chips featuring 256M encryptions/sec. will break a 56-bit key (e.g. DES) block cipher in 2 weeks. To make things worse, hardware speed and power are going up at an incredible pace. However, that time goes up to

an unfeasible 2×10^{20} years, if the key length is 128 bits (e.g. IDEA). Long-term asymmetric keys provide even more credible protection. For example, a 1024-bit key cipher is un-attackable just by brute-force if the key is random.

A key is weak if it has redundant information. This has the effect of reducing its equivalent length. For example, a 56-bit key using only lowercase letters and digits "shrinks" to around 40 bits. The attack exemplified above would succeed after a dangerously short 20 seconds. If keys have lexical content, they become amenable to dictionary attacks, which can make an attack even faster. (Please, do not use "dowjones" or "dragonball" for a key or password). Incidentally, 40 bits is the key length of several U.S. commercial encryption products (e.g., the export version of U.S.-made SSL), derived from the export restrictions. These were partially lifted in the end of 1999 (e.g., full Netscape and PGP are now freely exportable). The restrictions have been partially lifted. For example, SSL can now work with 128-bit cryptography all over the world.

How to generate a good key or password then? Good cryptographic systems have resorted to the **passphrase** stratagem. A passphrase is a long, intelligible sentence, known only to the user. When the user enters it, the system applies a cryptographic checksum to it, generating a high quality key or password. A rule of thumb for practically random content is one passphrase letter per password bit, but shorter phrases are normally enough. For example,

"My dear friends, who on earth would believe this is my passphrase?"
might yield after hashing something like the 64-bit hexadecimal quantity
`E6C1 0A9B 894E 03AF`
a good albeit hard to memorize password

Keys can be strengthened by combination. Long asymmetric keys may protect shorter but faster symmetric keys, the former being called **key-encrypting keys**. Practical protocols will perform *key rollover* as a routine, that is, change session keys often, even during a session. The robustness of fixed key-length ciphers (e.g. DES, IDEA) can be increased to approximately twice by running the protocol three times, which is known as *encrypt-decrypt-encrypt*. Finally, keys should be made both to the measure of the attacks they must withstand and to the lifespan of the data they are to protect. Keys may be long- or short-term. Data may be long- or short-lived. A key should be planned to resist an attack that can be made during its lifetime. There are a few exceptions: a short-term key protecting long-lived data inherits the lifetime of the data; the lifetime of a long-term key protecting long-lived data, or protecting short-term keys that protect long-lived data, must be an upper bound of all data lifetimes. Some of these issues will be detailed in Section 18.4.

18.3.3 Vulnerability versus Threat

A four-digit PIN is not a weak password if it is to be keyed in by hand at an ATM, validated by a token such as a card. A brute-force attack on the 10^4 possible combinations is practically infeasible with this technology: it would take too long; it would be too conspicuous; the machine would confiscate the card after a number of attempts. However, suppose that the technology was

directly transposed to an environment where the card would be read in a PC and the card Id together with the PIN would be submitted via a network to the bank. Then we would have an extremely fragile system, because: an automated program would be able to test all the combinations very fast; and do it remotely, under the cover of anonymity, without any possibility of physical action on the hacker or the card.

What did we want to show with this example? It showed that vulnerabilities are sensitive to the level of threat. It also stressed a more fundamental issue that we had not mentioned yet. Security existed long before the widespread use of computers and communication facilities. There is a risk that procedures that were perfectly safe in a manual, electrical or mechanical world will completely fall apart in the high-speed world of modern computers and the pervasive Web of network links. A classical example of the modern era is the DES cryptographic algorithm: the U.S. authorities, when it was introduced in 1977, imposed a key length which they believed could be cracked by brute-force (testing approximately 10^{16} combinations of the 56 bits of the key) only by their own powerful machinery, and no one else's, for many years to come. Today, that task is within the reach of amateurs with good computational resources. So, beware of technological changes.

Level of threat and degree of vulnerability are not precise quantitative measures, but awareness of these two facets is a fundamental strategic issue: the architect must balance her design options in order to achieve the desired risk of operation. We can give examples of what may go wrong when one of the terms is not taken into consideration, leading to a false impression of security:

Example 1: Many mainframe-based OLTP[1] systems rely on private networks of point-to-point leased lines. We have found many often, among architects of these systems, a false feeling of security of the leased line (wrongly assumed low level of threat), which can be tapped and intercepted as any other. However, this feeling relaxes the requirements on security of the user-to-database interaction (high degree of vulnerability), leading to a high risk of operation.

Example 2: The Secure Sockets Layer (SSL) protocol reportedly ensures secure client-server interactions between browsers and WWW servers. Users have tended to accept that the interactions are secure (assumed low degree of vulnerability), without quantifying *how secure.* Netscape's implementation of the SSL protocol was broken because of a bug that allowed to replicate session keys and thus decrypt any communication. The corrected version was then broken at least twice through brute-force attacks on the export version (at that time, the only one available to companies outside the U.S.A.), which used short (40-bit) keys. Several companies have built their commerce servers around SSL, some of them to perform financially significant transactions such as home banking. Some of these servers, because of the assumption of low vulnerability, were put on the Internet for spontaneous transactions with few or no restrictions to probing and access anonymity. Financial value and openess foreshadow a situation of high level of threat, leading once more to a high risk of operation.

[1] On-Line Transaction Processing.

No feasible system is 100% risk-free or secure. The level of threat and degree of vulnerability can be made to decrease in an asymptotic-like manner, where the limit is, respectively, a useless or an extraordinarily expensive system. Here is a good question an architect would like to see answered— *"What is the right balance?"*:

- There is no universal answer, but the level of threat is mostly dictated by the type and function of the system in question: Does it have to stand on the Internet or can it be behind a firewall? Must accesses be completely anonymous or can have clients identify themselves?
- The degree of vulnerability is dictated by the hardware and software used, and their configuration: Will a normal operating system be used, or a security-hardened one? Will strong or weak cryptography be used? Will strong authentication be made, or just O.S. access control?

A useful figure to equate the risk related with the operation of a given system is an estimate of the **cost to intrude** it under given conditions. For instance, when Netscape released the above-mentioned second version of the SSL implementation, they reported that it would cost at least USD10,000 to break an Internet session, in computing time. The cost of intruding a system versus the value of the service being provided allows the architect to make a risk assessment. Someone who spends 10 000 EURO[2] to break into a system and get 100 EURO worth of bounty, is doing a bad business. Unfortunately, these estimates may fail: shortly after Netscape's announcement, a student using a single but powerful desktop graphics workstation, broke the export version for just USD600. However, what went wrong here was not the principle and the attitude of Netscape, just the risk assessment they made, which was too optimistic. These estimates may also fail when values other than economic are at stake, such as political, for a potential attacker.

18.3.4 Open System Security Policies

The main vulnerabilities of open distributed systems lie in:

- Networks:
  - bugs of communication protocols
  - broadcast or wireless nature of networks
  - openess to anonymous access
  - the human element (administrator and user)
- Hosts:
  - bugs of the O.S. and widespread application software
  - wrong configurations (backdoors)
  - human element (administrator and user)

[2]The EURO is the currency of the European Union. It is worth approximately one USD.

There is a major threat to current operation of open systems, which is the general lack of strong authentication in current network and O.S. technology, leading to impersonation threats that can undermine the best cryptosystem, or the most conservative access control policy. In detail, the main threats to a networked system are the possibility of: scanning for resource and vulnerability discovery; eavesdropping with sniffers; active attacks with spoofers. On the other hand, the main threats on hosts are: the wide availability of exploits for almost any O.S.; automated penetration attacks such as viruses and trojans; dictionary attacks on passwords. A security policy, informal as it may be, is the key to securing any system. We suggest and discuss here a simple strategy for implementing a single-level security policy for an open system:

- selecting the system components;
- evaluating their vulnerabilities;
- adjusting threats to the desired risk according to a security policy;
- checking whether the right balance was achieved;
- iterate once more if not;
- implement the policy.

The **4P policies** are simple enough to be understood and implemented without much effort. They divide the strategies for protection into four possible choices, one of which is selected as the policy for the system in question. The 4P policies are[3]:

Paranoid	all is forbidden
Prudent	all that is not explicitly allowed is forbidden
Permissive	all that is not explicitly forbidden is allowed
Promiscuous	all is allowed

A *paranoid* policy is very simple to implement, because it means to completely isolate the system. It would be the implicit policy of enterprises in the old days. However, short of specific critical subsystems in an organization, it is not applicable to a whole organization in these times of connectivity and openess. The *promiscuous* policy is also simple to implement, because it lies in not making any access restrictions. It used to be the implicit policy of academic organizations in the old days of an "academic" and friendly Internet. It is doubtful that an organization's system following the promiscuous policy would survive long enough to do anything useful in these times of hackers and threat. The prudent and permissive policies will be the main workhorses in configuring the majority of systems. The *prudent policy* assumes that everything is forbidden unless explicitly allowed. It is the most difficult to implement, but the most effective. The difficulty lies in the nuisance it presents to users until it is tuned, if that state is ever reached. Its implementation consists of denying

[3]The credit for these mnemonic designations is attributed to BBN Cambridge.

all accesses by default, and explicitly opening all the desired accesses in whatever desired modes, one by one. Until the system stabilizes, users experience difficulty in accessing services not specified, which would normally be allowed in a more permissive configuration. The *permissive policy* assumes that everything is allowed unless explicitly forbidden. It is simple to implement, based on allowing all accesses by default, and selectively deny a number of accesses listed in a first specification of 'don'ts'. It is normally tuned at the cost of suffering attacks and closing the respective backdoors or suspicious accesses. A real system will normally be divided in subsystems, to which different policies may be allocated. Additionally, the starting point of the implementation of either the prudent or permissive policies may be a semi-allowed or semi-denied state, depending on the viewpoint.

18.3.5 Is a TCB Implementable?

The most effective protection or cryptographic mechanism can be defeated by a vulnerability, not in itself, but in the O.S. of the host if it is supposed to run securely that is, in a trusted computing base (TCB). The approaches to implement a TCB fall into two classes:

security kernel- designing an operating system kernel intended to be secure from the start and building the rest of the O.S. around it;

security enhancement- starting with an existing operating system and designing security in, by retrofitting or simply configuring the O.S.

The central question is *coverage*, exactly in the same sense that it was studied in Section 6.2 of the Fault Tolerance part: the probability with which the TCB properties hold. The *security kernel* approach is the only road to achieving extremely high coverage of the TCB properties: interposition, shielding, and validation. Alternatively, an operating system designer may select an existing operating system and improve it by *retrofitting* security into it, getting what is called a *security-enhanced kernel.* A milder form of security enhancement that does not involve source code modification is achieved by *configuring* an existing commercial operating system.

When attacks just take the form of probing, direct or by software, or the risk of sporadically successful penetration attacks (for unlikely) is accepted, a security-enhanced system will be adequate. Then, the choice between retrofitted or configured kernels depends on the balance between: price, performance, and accepted risk. Configured kernels may represent an adequate solution when: assets are of moderate value and removal of vulnerable functions does not compromise effectiveness of the service; or for protecting a first rampart, like a firewall, when there are complementary measures, such as intrusion detection. For systems which, despite the high value of the assets at stake, have to endure a high level of threat, such as open financial transactional servers, retrofitted kernels may be a sensible decision, in order to keep the risk of operation at acceptable levels. The security kernel approach for a TCB is definitely the solution to consider when the system holds very sensitive data and may be subjected to attacks at the level of penetration or subversion.

18.3.6 Preventing Attacks

The TCB concept attempts at eliminating vulnerabilities. It is part of a prevention strategy: if vulnerabilities disappear, attacks cannot be successful. This is feasible for very focused components, implementing critical functions, but is hardly a good strategy for the whole of the system. However, we can specialize the strategy to preventing attacks, the next step down the ladder, i.e., the malicious faults that take advantage from vulnerabilities. This has two facets: not letting attacks be produced; not letting them cause intrusions.

The first approach concerns all techniques for placing the system or the desired components out of the reach of attacks, that is, the direct reduction of the level of threat. For example, placing a sensitive database behind a firewall, without access from the outside. The second approach has to do with letting the attack be done, detecting it and acting after it is performed and before it succeeds in causing an intrusion. For example: by defusing a time bomb or deleting a virus before they are activated; by detecting scanning from a suspect host and blocking that host.

18.3.7 Detecting and Reacting to Intrusion

The prevention strategy is not always adequate. In this case, we must redefine the strategy to act one degree further down the ladder: attacks will be there, and cause intrusions. We can: detect, recover from, or mask intrusions. In fault tolerance terms, we might call this strategy *attack tolerance.* Tolerance and prevention are often used in combination.

Intrusion detection is a well known field. Intrusion detection systems (IDS) aim at detecting attacks and intrusions. They are based on native logging mechanisms of O.S.s and networks, and on specific instrumentation (sensors and actuators) placed wherever appropriate in the system, such as network sniffers, and system-call interceptors. Depending on the type of IDS, its detectors may be sensitive to different stimuli. Some are based on learning the *patterns* of attacks, and using a *knowledge* base of known attacks to detect them. Others learn the normal *behavior* modes of the system, and base intrusion detection on the occurrence of *anomalies* in that behavior. The latter are more adaptive, both to the evolving behavior of the system, and to new attacks, since they do not have to know attacks at all. Subsequent to detection, there is reaction, which may go from event alarms to automated or semi-automated countermeasures. Countermeasures take several forms, and are specific of the kind of attack and the make of the IDS. They may involve sanitizing attacked components (neutralizing viruses and Trojan horses), or neutralizing outside attackers, by barring their way in a firewall, for example.

Detection is not always accurate, or fast enough. The risk is for intrusions to take place that may cause failure before being detected. In this case, there is room for more sophisticated techniques involving *intrusion masking.* The principle is based on providing the system with redundancy such that in the presence of intrusions it continues to work correctly. For example, if a file is

encrypted, fragmented and scattered through several hosts, the intrusion of one such host will not reveal the file, regardless of whether the intrusion is detected or not (Deswarte et al., 1991). Threshold cryptography is also resilient to a number of malicious participants.

Past the real-time reaction to an attack, there is often the need to repair the system: removing the remnants of the attack (e.g., deleting Trojan horses, which may involve formatting disks); hole-plugging or removing the vulnerability that caused the attack (e.g., an account with a broken password). These actions are also called fault treatment in a fault tolerance sense.

18.3.8 Avoiding Disruption

Disruption attacks attempt at damaging data or causing denial of service. They affect the properties of integrity and availability. It is generally recognized that availability and integrity preservation are hard to achieve. Integrity preservation may start with attack prevention by limited physical separation. Next, it requires access control. However, replication should be used in order to tolerate successful attacks, that may damage some but not all the replicas.

On the other hand, the classical techniques of access control and cryptography provide little help against denial of service attacks. Some of them are external, and may come from spoofed hosts, making countermeasures extremely difficult. Authentication plays a role here, because it allows filtering out and even fight back the malicious external users. In consequence, this also suggests that allowing completely anonymous accesses to public servers favors denial-of-service attacks. Replication, inasmuch as it is used for availability with respect to accidental faults, may also yield positive results, if set up in a way that common-mode attacks to all replicas are prevented.

18.4 USING CRYPTOGRAPHIC PROTOCOLS

We studied four main cryptographic building blocks: symmetric encryption; asymmetric encryption; secure hash; and signature. Their functionality serves different purposes. Their relative speed also varies quite a lot: if we wanted to give a gross estimate of their relative performance, we would say that if asymmetric algorithms run at speed 1, then signature ones also run at speed 1, while symmetric run at up to speed 1000, and hashes run at speed 3000. Combined in several ways, they yield protocols that perform a number of interesting functions.

18.4.1 Protocol Types

Cryptographic protocols use and combine cryptographic algorithms in several forms. Generally speaking, there are two basic *protocol types*, depicted in Figure 18.3, with regard to the approach taken to enforce correct behavior in the presence of malicious faults or attacks:

Self-enforcing	in which correct behavior is guaranteed by the protocol, which ensures trust among the participants
Trusted-third-party	in which correct behavior is guaranteed by a third party, which builds trust among the participants; trusted-third-party protocols assume three facets, depending on *when* trust is built:
adjudicated:	in which correct behavior is guaranteed a posteriori, by an adjudicator who in case of a fault will determine the responsible participant(s), by analyzing information (evidence) collected during the execution of the protocol; the participants trust that whatever may go wrong will be corrected by the adjudicator
arbitrated:	in which correct behavior is guaranteed during the execution, by an arbiter who ensures that the protocol executes correctly by participating of every execution; the participants trust that whatever might go wrong is prevented from happening by the arbiter
certified:	correct behavior is guaranteed a priori, by a certification authority who will issue and provide participants with certificates appropriate to build trust between them; the participants trust that nothing may go wrong that is guaranteed by the certification authority

In essence, a self-enforcing protocol, depicted in Figure 18.3a, guarantees that the properties of the service are achieved solely by the mutual interactions between the principals. Whatever faults occur, they are tolerated from within the protocol. That is, the protocol is a error-containment domain, hence its name. Since nothing comes free, the complexity normally migrates to the underlying algorithms.

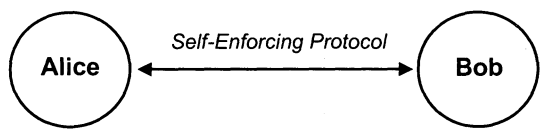

Figure 18.3. Protocol Types: (a) Self-Enforcing

The other two types both have in common the fact that they resort to the assistance of what is called a **trusted third party** (TTP). A TTP is an entity which is trusted by all principals to perform its functions correctly and impartially. As such, typical functions of a TTP are: adjudication, arbitration, and certification. Depending on the type of TTP, we call TTP-protocols adjudicated, arbitrated, or certified, as depicted in Figure 18.3b.

The decision about which one to use lies in the answer to the following questions: can we afford an incorrect behavior at all? Can we build trust beforehand (off-line), or do we need to watch every execution (on-line)? If a country's court system is efficient, we may go ahead in a business, knowing that if something goes wrong, we will be compensated. However, if it is slow

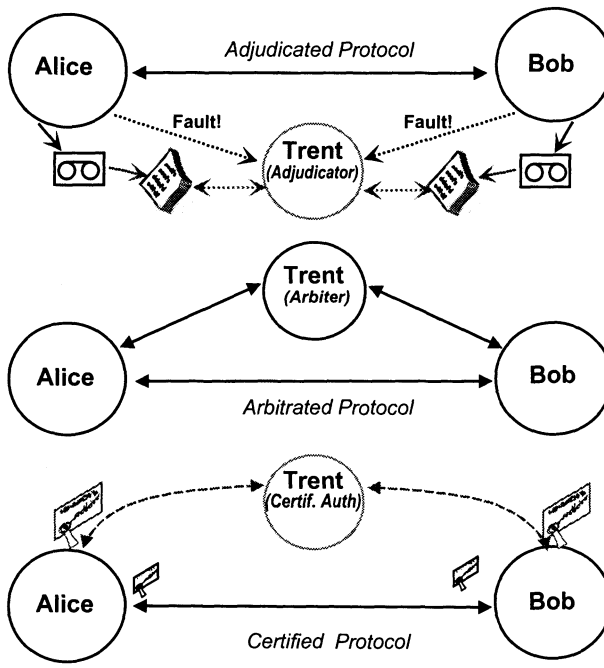

Figure 18.3 *(continued)*
. Protocol Types: (b) Trusted-Third-Party— adjudicated; arbitrated; certified

and unreliable, perhaps we had better not be cheated and take all necessary precautions: if we believe in the credentials our partners present, that is enough; otherwise, we should have someone trusted by both follow the negotiation.

Technically speaking, adjudicated protocols take the first approach, performing fault tolerance with long error latency in error recovery, whereas arbitrated and certified protocols take the second, performing fault prevention. There is a substantial difference in overhead between them: arbiters must always be on-line and thus they are a bottleneck and a single point of failure; certification authorities act prior to the execution of the protocol, and thence they may even be off-line, if certificates are obtained beforehand; adjudicators act in background and only when one of the parties suspects anything. The only constant overhead for the latter is the collection of evidence during the execution of the protocol.

18.4.2 Block Cipher Modes

There are several modes to use block ciphers. We will present them using DES as an example, but they can be applied to any block cipher. At first sight, the obvious way to use DES would be: to break the cleartext in 64-bit (or 8-byte) chunks; to do *padding* with some pattern to fill in the last block if the text is

not a multiple of 64 bits; to encrypt the blocks one at a time and concatenate the result. This is called the *Electronic Code Book (ECB)* mode. ECB mode is susceptible to analysis and replay attacks, since an 8-byte cleartext chunk always encrypts to the same cyphertext.

The systematic solution is to make blocks depend on one another. *Cipher Block Chaining (CBC)* applies feedback to a block cipher: the plaintext of block i is XORed with ciphertext of block $i-1$ and then encrypted. This way, Mallory cannot cut and paste a message arbitrarily. However, this is not perfect yet: he can still analyze/replay whole messages— such as encrypted password messages— because with CBC, two identical messages get the same ciphertext. The solution is to encrypt a block of random data as the first block, called *Initialization Vector (IV)*. The initial difference propagates to the whole message because of the feedback and so the same message never yields the same ciphertext twice.

Interestingly enough, block ciphers can be implemented as stream ciphers, in a mode called *Cipher Feedback (CFB)*, which allows encryption of an arbitrary stream of bits. *Output Feedback (OFB)* is yet another streaming block cipher, with interesting properties. It uses DES to generate a pseudo one-time pad, by feeding it with the ciphertext of the previous block (starting with IV).

MACs can be generated with Block Chipers. There is a standard MAC protocol, ANSI X9.9 (ANSI X9.9, 1986), which uses the *CBC residue* approach: the message is encrypted by DES in CBC mode, but only the last block is used as a 64-bit hash (*see* Section 18.4). The recipient verifies in the same way. Internet protocols, such as SNMP, use the HMAC protocol (RFC2104).

18.4.3 Double and Triple Encryption

It is generally believed that encrypting more than once with different keys improves the security of a block cipher. This is not always so, and in fact for DES double encryption ($C = E_{K_b}(E_{K_a}(M))$) would be much like using a 57-bit key, because of an attack known as **meet-in-the-middle** (Merkle and Hellman, 1981). However, triple encryption technique works, and can be very robust with a subtlety: the middle operation is decrypt, using two keys in the following manner: $C = E_{K_a}(D_{K_b}(E_{K_a}(M)))$. It is called *Encrypt-Decrypt-Encrypt (EDE)*. If a single key is n bits, the equivalent key length is $2n$. DES is an obvious candidate: a 112-bit 3DES sidesteps worries discussed earlier about the fragility of DES.

18.4.4 Signing and Encrypting

When transmitting a document it may be desirable to enforce both authenticity and integrity on one hand, and confidentially on the other hand. This is achieved by signing and encrypting. Now, how should it be done? Sign then encrypt? Encrypt then sign? If we encrypt first, we will be signing something unintelligible. This is not very wise. Besides, it is cryptographically vulnerable (Anderson and Needham, 1995).

The protocol can be derived from the message-digest public-key signature protocol (*see* Figure 17.10). Alice wants to send a signed and confidential message M to Bob. The protocol is explained in Figure 18.4.

		Action	Description
1	A	$h_m = H(M)$ $s_m = S_a(h_m)$	Alice computes the message digest and signs the 128-bit digest with her key
2	A → B	$\langle E_b(M, s_m) \rangle$	Alice encrypts both the signature and the message with Bob's public key, and sends it to Bob
3	B	$D_b(M, s_m)$	Bob decrypts first with his private key
4	B	$h_m = H(M)$ $v_m = V_a(s_m)$ $v_m = h_m$?	Bob verifies the signature using the following procedure: he hashes the message, then runs V_a on the signature; if the result is equal, then M is ok, and was signed by Alice

Figure 18.4. Public-key Signature and Encryption

A public key protocol such as RSA can be used for both functions, encryption and signature. It is even tempting to use the same keys. However, there are attacks that can take advantage of these vulnerabilities, so care must be taken on how to design and use the protocol. Good sense advises: not to use the same keys for signing and encrypting; if possible, not to use the same algorithm for signing and encrypting; sign before encrypting, and in any case, never put the user in a position of signing unknown things.

18.4.5 Hybrid Cryptography

Nothing could be more wrong than considering that symmetric and asymmetric cryptography are alternative or competitive encryption approaches. In fact, they are complementary, and there is a current trend to associate symmetric cryptography with payload encryption, and asymmetric cryptography with key and signature encryption, in what is called *hybrid cryptography.* We are going to study two examples of approaches used in most current systems: hybrid cryptographic channels and envelopes. Alice and Bob can execute symmetric encryption/decryption, although they don't currently share any key. They also rely on public cryptography: Alice has private key Kr_a, Bob has Alice's public key Kp_a, and vice-versa for Kr_b and Kp_b.

Let us look at how the first protocol could work. Alice and Bob wish to communicate and will establish a *hybrid cryptographic channel* for the purpose: a secure channel where the session key for the channel is exchanged by public-key cryptography, and communication is done with symmetric cryptography using the session key. The protocol is explained in Figure 18.5. Signing may appear to be an overkill, but in fact it foils attacks by impersonation: anyone

		Action	Description
1	B	$s_m = S_b(K_{ss})$	Bob generates a random key K_{ss} and signs it with his private key
2	B $\rightarrow$ A	$E_a(s_m)$	Bob further encrypts the signed session key with Alice's public key, and sends it to Alice
3	A	$s_m = D_a(E_a(s_m))$	Alice retrieves the signed key s_m by decrypting with her private key
4	A	$v_m = V_b(s_m)$ v_m $OK?$ $K_{ss} = v_m$	Alice verifies Bob's signature with his public key, retrieving K_{ss}
5	A,B	use K_{ss}	Alice and Bob communicate with symmetric encryption using K_{ss} as the key

Figure 18.5. Hybrid Cryptographic Channel

could encrypt something for Alice. This very simple protocol explains the principle of hybrid cryptographic channels. More sophisticated short-term key-exchange protocols with public-key and hybrid cryptography will be studied in Section 18.6.

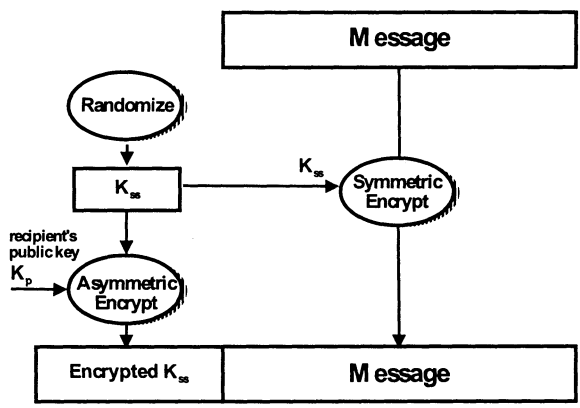

Figure 18.6. Hybrid Cryptographic Envelope

The second approach, that we call hybrid cryptographic envelope, uses the key-encrypting key principle: the message is encrypted with symmetric cryptography, and the encryption key goes along with the envelope, encrypted by public-key cryptography. This second protocol is depicted in Figure 18.6. Alice wishes to send sporadic messages to Bob, and for that purpose, she wraps them in a *hybrid cryptographic envelope.* Alice generates a random symmetric key,

encrypts the message with it, and then encrypts the session key with Bob's public key and concatenates it with the message.

18.5 AUTHENTICATION MODELS

In Section 17.9 we addressed the authentication paradigm. We understood that there are essentially three types of authentication: unilateral- principal A authenticates itself to principal B; mutual- principals A and B mutually authenticate themselves; mediated- principal A is authenticated to B by principal T, whom they both trust. These types are implemented through essentially three *mechanisms for authentication* of a principal A with a principal B:

Password	authentication is based on A submitting to B a unique pair ⟨ `username,password` ⟩ that B recognizes
Shared-secret	authentication is based on A proving to B that A knows a secret K that they and no one else share, without showing the secret
Signature	authentication is based on A proving to B to have signed something using its digital signature S, that no one else can produce

In what follows, we discuss several fully-fledged authentication models, which are essentially instantiations of the several authentication types materialized with combinations of the mechanisms just described. We analyzed them under the light of two groups of threats: reading of the authentication service state; eavesdropping on remote interactions.

18.5.1 Password-based Authentication

The simplest way to authenticate an authorized user in a system is password-based authentication. The problem is the following: a principal A submits a pair ⟨`username,password`⟩ to an authentication service S, and the pair should uniquely identify it. However, this method is not very secure. The intruder may read the state of S, so the password file should be protected: by obscuring its contents, for example by replacing the password with its hash or with a cryptographic checksum depending on the password, so that it cannot be reversed (*one-way encryption*); by making it difficult for the intruder to read the password file (*shadowing*). Even so, the intruder can decipher the password of a user by guessing, what is also called *cracking* in hacker lingo. He can try them directly, exhaustively or by selecting probable passwords, called *on-line password guessing*, which is infeasible if passwords are long enough. Else, he can do *off-line password guessing*, if he gets access to the hashed password file. He runs each candidate password through the hash algorithm and checks if the result is the same as in the password file. The attack can be made quicker by using a dictionary to base the search on (usual words in the language, words related with the user, her working field and her organization, etc.). This is called a **dictionary,attack**, and it is surprising how many passwords can be

guessed in a normal organization. This is why the password creation, change and deletion procedures discussed in Section 18.3.2 are important, in order to harden the passwords against guessing.

The second problem with password authentication is that it is very insecure when login is remote. The password is submitted in cleartext, and so an eavesdropper may read it and reuse later. Compared to the trouble of cracking a password file, this is a lot easier, and only requires that the hacker controls one machine in a network, so beware of what has become a very common threat. Since eavesdropping or sniffing on a network can go unnoticed, there is no remedy for this but changing the approach: encrypting whatever is exchanged or exchanging only public or trivial things. Later in this section we will learn how to do more resilient authentication exchanges.

Password File Protection An interesting technique for password file protection is employed by UNIX and other systems. The generic mechanism works as follows:

1. *the 8-character (7-bit ASCII)* password *is converted into a 56-bit key to be used in a DES-derived encryption function called* crypt*; crypt is also parameterized with a 12-bit* salt *value, a randomized quantity that makes two entries of the same password always look different;*
2. `crypt` *is a cryptographic checksum algorithm: it starts using an all-zero block as input, uses the result as the input of the next round, and performs 25 rounds, using the key and the salt;*
3. *the result is translated to an 11-character printable ASCII* string*, and the password entry of a user in the password file becomes the triplet* ⟨userId, salt, string⟩*;*
4. *when a user logs in, she supplies* ⟨userId, password⟩*; the operating system indexes the password file with* userId *and grabs* ⟨salt, string⟩*;*
5. *the cryptographic checksum is performed on* ⟨salt, password⟩ *and the result compared with* string*. If they match, the user is authenticated.*

One-Time Passwords Systems offering one-time passwords (OTP) change the password each time it is used. In consequence, this approach is of the "exchanging trivial things" type: the plaintext passwords the eavesdropper sees passing are useless by the time he collects them. Existing one-time password systems fall in the class of the *challenge-response* systems: the authenticator sends a challenge in the form of *"give me password number x"*; the user should reply with the appropriate password. Passwords may be computed interactively by the user upon receiving the challenge, which requires that her machine can run the computation or that she is assisted by a device such as a pocket-calculator password generator. Else, a ⟨*challenge,password*⟩ list, with all the passwords corresponding to every challenge, is given in advance to the user.

The essential property of the mathematical function used to compute the challenge is that the password should be deduced neither from the challenge nor from past passwords. It would seem that one-way hashes would be extremely appropriate for this. Lamport invented a function for a one-time pass-

word scheme (Lamport, 1981) based on generating a sequence of n passwords by hashing a password n times. Of course, the system must start from password p_n and work backwards, to avoid that password p_{n+1} is deduced from password p_n by simply hashing one more time. In conclusion, note that one-time passwords are resilient against both reading penetration and eavesdropping. One inconvenience of the scheme is that passwords eventually get exhausted, requiring the process to be re-initiated. This however is not a problem for short-term use (e.g., during trips). The S/Key system, discussed in Section 19.1, is an example of OTP system.

18.5.2 Shared-Secret Authentication

The approach of "encrypting whatever is exchanged" is appropriate for long-term use. Shared-secret authentication is one such example, where principal A (Alice) and authenticator S (Stuart) share a secret known only to both of them, let us call it K_{as}. They perform cryptographic interactions so that S is persuaded that its peer *knows* K_{as}. At this point, S *believes* that it is A on the other end, since only A could know the secret. We sketch the two basic mechanisms for doing it. Actually, mechanism (a) can use either true encryption or a cryptographic checksum (one-way encryption):

- *a) Alice sends* userId*; Stuart sends challenge "if you are Alice, encrypt X for me"; Alice sends response* $E_{as}(X)$*, which S checks;*

- *b) Alice sends* userId*; Stuart sends challenge "if you are Alice, decrypt* $E_{as}(X)$ *for me"; Alice sends response X*

18.5.3 Signature-based Authentication

Yet another "encrypting what is exchanged" approach, signature-based authentication uses public-key cryptography. Principal A (Alice) has private key Kr_a, and authenticator S (Stuart) has A's public key Kp_a. They perform cryptographic interactions so that S is persuaded that its peer knows the pair of Kp_a. At this point, S believes that it is A on the other end, since only A could know the private pair of Kp_a. There are essentially two mechanisms for doing it:

- *a) Alice sends* userId*; Stuart sends challenge "if you are Alice, sign X for me"; Alice sends response* $S_a(X)$*, which S verifies with Alice's public key;*

- *b) Alice sends* userId*; Stuart sends challenge "if you are Alice, decrypt* $E_a(X)$ *for me"; Alice decrypts with her private key and sends response X.*

18.5.4 Mutual Authentication

We have been discussing how to authenticate principal A to principal S. However, what if someone impersonates S and cheats on A? The solution to that problem is mutual authentication. None of the mechanisms discussed so far allows mutual authentication, but that can be achieved by enhancing the same mechanisms.

Shared-secret Mutual Authentication Shared-secret mutual authentication can be derived from the unilateral mechanism, by having S authenticate to A in the same way. The principle is explained in Figure 18.7.

		Action	Description
1	A → S	$\langle A \rangle$	Alice sends *userId*
2	S → A	$\langle X_s \rangle$	Stuart sends his challenge X_s for Alice to encrypt
3	A → S	$\langle E_{as}(X_s), X_a \rangle$	Alice sends the encrypted challenge in response together with her challenge X_a
4	S → A	$\langle E_{as}(X_a) \rangle$	Stuart in turn sends the encrypted response to Alice's challenge
5	A,S		Both believe they're talking to each other

Figure 18.7. Shared-secret Mutual Authentication

This protocol is only vulnerable to Mallory impersonating Stuart to collect material to do a key guessing attack, which is difficult per se. Moreover, the feasibility of this attack depends on the strength of the cipher used.

Mutual Authentication by Signature Signature-based mutual authentication can also be derived from the unilateral protocol, by having S authenticate to A in the same way. The modified protocol is explained in Figure 18.8.

		Action	Description
1	A → S	$\langle A, X_a \rangle$	Alice sends *userId* and challenge for Stuart to sign
2	S → A	$\langle X_s, S_s(X_a) \rangle$	Stuart sends his challenge X_b for Alice to sign, and signs Alice's challenge
3	A → S	$\langle S_a(X_s) \rangle$	Alice sends the signed challenge in response
4	A,S		Both believe they're talking to each other

Figure 18.8. Mutual Authentication by Signature

Alice believes it is Stuart on the other end, because she could verify his signature on her challenge. Only Stuart could have signed the challenge. Stuart believes it is Alice on the other end, exactly for the same reasoning applied to Alice's signature.

18.5.5 Mediated Authentication

Alice shares a secret K_{at} with arbiter Trent (so does Bob with K_{bt}), and can authenticate herself to Trent (so can Bob) in the sense of our discussion on shared-secret authentication. The problem now is to extend this to authentication of Alice to Bob and vice-versa, *mediated* by Trent. The principle of mediated authentication protocols is the following:

1. *Alice sends her* Id *and Bob's to Trent, requesting an authenticated session;*
2. *Trent arranges for a shared secret key K_{ab} to be distributed to both: to Alice, encrypted with her key K_{at}, and to Bob, encrypted with his key K_{bt};*
3. *Trent also ensures that it is distributed in a way that at the end of this process, they are mutually authenticated in the sense of the shared-secret principle (see Figure 18.7), where K_{ab} is the shared secret.*

		Action	Description
1	A $\rightarrow$ T	$\langle A, B, X_a \rangle$	Alice sends her and Bob's *Id* and nonce X_a to Trent
2	T	K_{ab}	Trent generates a key K_{ab} for Alice to share with Bob
3	T $\rightarrow$ A	$\langle E_a(X_a, B, K_{ab}, ticket= E_b(K_{ab}, A)) \rangle$	Trent sends Alice, encrypted with Alice's key: her nonce back, Bob's *Id*; the shared key. Also under the encryption goes a credential or ticket to Bob, encrypted with Bob's key, containing Alice's *Id* and the shared key
4	A $\rightarrow$ B	$\langle E_b(K_{ab}, A) \rangle$	Alice sends the ticket to Bob
5	B $\rightarrow$ A	$\langle E_{ab}(X_b) \rangle$	Bob retrieves the shared key, encrypts his nonce with it, and sends it to Alice
6	A $\rightarrow$ B	$\langle E_{ab}(X_b - 1) \rangle$	Alice retrieves and decrements the nonce, and sends it back encrypted with the shared key
7	A,B		Both believe they're talking to each other

Figure 18.9. Original Needham-Schroeder Authentication Protocol

Nonce-based Authentication Let us have a look at the basic Needham-Schroeder protocol (Needham and Schroeder, 1978), depicted in Figure 18.9. Recall that **nonce** is a quantity that is used only once. Sequence or random numbers or timestamps, are nonces. Nonce X_a in (1) tries to reassure Alice that she's addressing Trent, since he uses it in his reply (3) back to Alice. B in (3) reassures Alice that the shared key is meant for her connection to Bob, and no one else. The ticket cannot be seen by anyone but Bob, and reassures him that the shared key is meant for his connection to Alice. The nonce X_b sent in (5) and sent back decremented by Alice (6) proves to Bob that Alice knows K_{ab}.

Authentication based on Synchronized Clocks The Kerberos authentication protocol (Neuman and Ts'o, 1994) is depicted in Figure 18.10. It is very simple, and relies on roughly the same reasoning as the former protocol to achieve authentication. However, it further relies on principals having synchronized clocks to test each other's timestamps and detect replay attacks, within a window whose size depends on the precision of the clocks (*see Time and Clocks* in Chapter 2). Alice tests Trent's timestamp in (3) to see that it is current. Bob tests Alice's timestamp T_a in (4). Alice tests her incremented timestamp $T_a + 1$, which could only have come from the current interaction with Bob, since Alice sent message (4) at time T_a, protected with the shared key. The Kerberos Security Service (*see* Section 19.4) uses this protocol.

		Action	Description
1	A $\rightarrow$ T	$\langle A, B \rangle$	Alice sends her and Bob's *Id* to Trent
2	T	K_{ab}	Trent generates a shared key K_{ab} for Alice to share with Bob, and assembles a *keyId*, containing the key, its expiry time T_x, and the current timestamp T_t
3	T $\rightarrow$ A	$\langle E_a(keyId, B),$ *ticket*= $E_b(keyId, A) \rangle$	Trent sends Alice: (i) the *keyId* and Bob's *Id*, encrypted with Alice's key; (ii) a ticket to Bob, encrypted with Bob's key, containing the *keyId* and Alice's *Id*
4	A $\rightarrow$ B	$\langle E_{ab}(A, T_a),$ $E_b(keyId, A) \rangle$	Alice sends Bob the ticket together with her *Id* and current timestamp, encrypted with K_{ab}
5	B $\rightarrow$ A	$\langle E_{ab}(T_a + 1) \rangle$	Bob retrieves the shared key and the timestamp, increments the latter, encrypts it with the shared key, and sends it to Alice
6	A,B		Both believe they're talking to each other

Figure 18.10. Kerberos Authentication Protocol

18.5.6 Distributed Authentication

The several levels of indirection in a distributed system render the problem of authentication a complex one. That is, the authentication models we have just studied must incorporate delegation, and that is hard to do in face of: autonomy, scale, heterogeneity.

Distributed authentication is about authenticating a channel along which there is a user who sits at a host that sends messages on behalf of her through a network that carries messages on behalf of the host, and so forth. It is difficult to ensure that the basic principles of delegation in such an environment are followed correctly (*see Delegation* in Section 17.9) such as *specific delegation*, *authentication forwarding*, or *end-to-end authentication*.

Lampson et al. (Lampson, 1993) introduced a logic of authentication and delegation of *channels* in distributed systems:

- A **says** s – e.g., it is true that A produces statement (requests) $\langle read\ m \rangle$
- A **speaks for** B – for example, a terminal represents whoever sits at it, i.e., $\langle K_a$ speaks for $A \rangle$
- **handoff**: A **says** B **speaks for** A – what defines the delegation, i.e., A delegates on secure channel B
- **credential**: proof that $\langle A$ speaks for $B \rangle$ – what proves the delegation, i.e., that user A delegates on host B the access to critical data owned by A; to that purpose, A gives a credential to B, maybe a password typed at the terminal, or a card swiped through the terminal card reader

A more complex delegation would be: if C **speaks for** A, and C **says** (K_{ab} speaks for A), then K_{ab} **speaks for** A. In other words, if a secure (and authentic) channel from A sends a message with a session key K_{ab} for a third principal B to speak with A, then any message to B encrypted with K_{ab} speaks for A. That is, A delegated in C, and C announced that A delegated in K_{ab}.

18.6 KEY DISTRIBUTION APPROACHES

In Section 17.9, we have explained that key distribution has two facets: *long-term key distribution*, that puts in place all the keys necessary for system bootstrap and long-term use; and *short-term key exchange*, that exchanges the keys necessary for temporary use, such as message or email deliveries, or interactive sessions. Remember that we called these keys *session keys*.

The long-term key distribution problem is mostly concerned with distribution of public key certificates. Short-term key exchange mainly addresses shared secret keys for symmetric cryptography. Ad hoc distribution of public keys is not practical for large systems, composed of unknown users, so we will study a dedicated, "official" service, specializing in supplying public key certificates, called **Certification Authority (CA)**. Likewise, managing pair-wise keys in a large system is an insurmountable challenge. In consequence, we will study a specialized service, called a **Key Distribution Center (KDC)**, which significantly eases the task of further exchanging keys in the system.

We start by addressing the general long-term key distribution problem, and then we discuss short-term key exchange, where we will consider, when necessary, that some form of long-term shared secret or asymmetric key pair is already in place.

18.6.1 Certification Authorities (CA)

In open distributed systems, for example those working over the Internet, there may be a large number of participants, who do not trust and may even not know each other. Since public key cryptography is crucial for digital signatures and these are a powerful form of authentication, an obvious solution to this problem is to provide **certificates** containing a participant's name and its signature. It is only natural for this name-to-key translation to follow a principle similar

to the name or directory service in distributed systems (*see Naming and Addressing* in Chapter 2). The server is a trusted third party called Certification Authority (CA)), and besides a well-known address as the name service, it has a *well-known public key*. The CA signs every certificate it issues, and its signature can be verified by any participant. Other functions of the CA are the revocation of certificates, and the distribution of certificate revocation lists (CRLs).

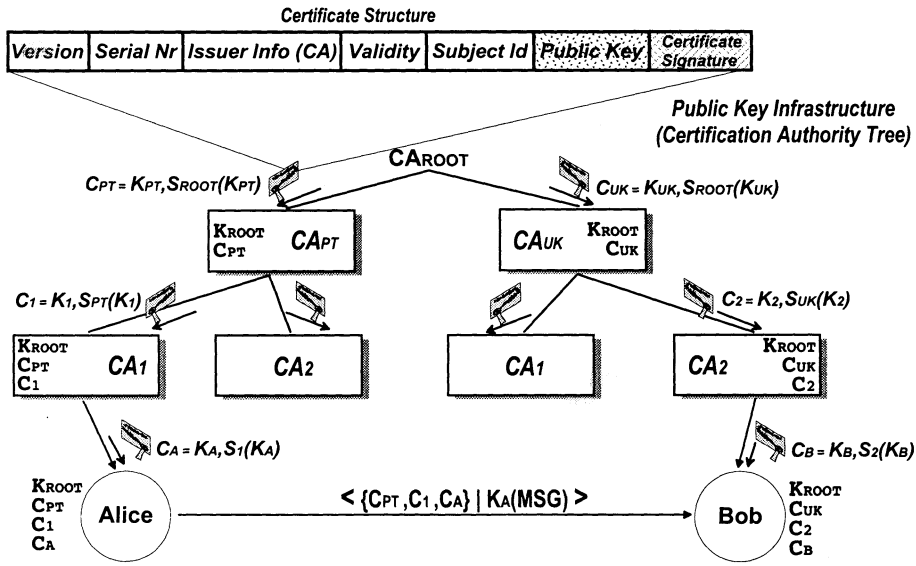

Figure 18.11. Public Key Infrastructure

Certificates are inherently secure to travel over the network, since they can neither be modified nor forged. The CA can work off-line, or have a very restricted interface (e.g., email, Web) to ease a tamperproof design. However, a CA may also be used to provide on-line proofs of identity. In a large-scale system such as the Internet, the CA's can form a hierarchy. Infrastructures for CA hierarchies are being standardized, the collective initiative is designated as the Public Key Infrastructure (PKI) and one such standard is the X.509 from the ITU (X.509, 1997). They work such that the CAs of the next level down have their public key certified (signed) by the root CA, and so forth down the hierarchy, such as represented in Figure 18.11, where part of an imaginary European X.509-like PKI is shown, with each CA keeping the certificates of all CAs in the chain up to the root.

The PKI is the primary support for wide use of public key cryptography: as defined by the X.509 standard, a certificate is a digital document that binds a public key to the identity or another attribute of its principal. As shown in Figure 18.11, a key certificate is normally composed of the identity of the principal, its public key, the timestamp of creation and validity, everything signed by the

issuing authority, and verifiable with the public key of the latter. Imagine that Alice in Portugal wishes to perform some operation with Bob in the UK, such that her public key is required— e.g., to have Bob encrypt things for her, or to check her signature. Alice has previously requested her certificate to CA_1 on the left, whose X.500 Distinguished Name is in fact $\{CA_{ROOT}, CA_{PT}, CA_1\}$, and in consequence, she keeps a chain of certificates $\{K_{ROOT}, C_{PT}, C_1, C_A\}$, which she sends Bob, alone or as part of one of the several public-key protocols available[4]. PKIs have been receiving great attention given their importance for electronic commerce. The way they are being deployed today is however not exempt from risks, as pointed out by Ellison & Schneier (Ellison and Schneier, 2000).

18.6.2 Key Distribution Centers (KDC)

A key-exchange mechanism for symmetric cryptography requires the distribution of one key per pair of participants. This is not practical, since it requires a very large amount of keys $(n\,(n-1)/2)$ and establishing trust among an unnecessarily large number of pairs of people, namely in a large-scale system.

If key distribution is centralized in a special service, then only n keys, for n principals, will be required. This service is performed by a trusted third party called a Key Distribution Center (KDC). The initial bootstrap process of exchanging the first key may be part of the off-line process of registering the new user. After that, everything goes through the KDC.

The KDC presents several disadvantages. The first is that it is a serious single point of failure, for two reasons: once compromised, it can impersonate anyone to anyone; and if it crashes, everything stops. Secondly, it is a bottleneck, because unlike CAs, it will be used on-line and in the critical path of the execution of protocols. Like CAs, KDCs can be interconnected. The Kerberos Security Service (*see* Section 19.4) exemplifies a KDC.

18.6.3 Short-term Key Exchange

Key-exchange with KDC Although key exchange may be performed with pair-wise symmetric cryptography, such a mechanism is not practical, since it requires a very large amount of keys. A more practical solution is to consider that in real-life systems, Alice and Bob do not trust each other, but each of them trusts a third party, which can be a KDC as we have discussed earlier. This is a good characterization of an open distributed system. Alice and Bob use a mediated authentication mechanism such as the ones described in Section 18.5.5, to have their key distributed by the KDC.

[4]The Distinguished Name is a unique name in the hierarchy that identifies a principal. It is composed by concatenating the names of the hierarchy above the principal, such as done with DNS names. We wanted to emphasize this aspect by showing that the lower CAs in PT and UK can have equal names.

Key-exchange with Diffie-Hellman Assume that Alice and Bob wish to exchange a session key K_{ss}, and rely on the Diffie-Hellman algorithm to create it without having to exchange secret values. According to the basic algorithm (*see* Figure 17.5), Alice and Bob exchange their public numbers y_a and y_b, and compute the same $K_s s$. Now suppose that just before they exchange numbers, Mallory gets in the middle and instead gives Alice y_{b_m}, and Bob y_{a_m}. This situation is depicted in Figure 18.12, and the effect is that instead of Alice and Bob sharing some key K_{ss}, Alice and Mallory end up sharing a key K_w, different from that shared by Mallory and Bob, K_z. However, Bob and Alice do not know that. If Mallory is fast enough that he can decrypt a message M coming from Alice and re-encrypt it to Bob on-the-fly, and vice-versa, the attack will go unnoticed. Needless to say that Mallory can leisurely study Alice's and Bob's interactions and prepare something worthwhile (suppose Bob was Alice's home banker).

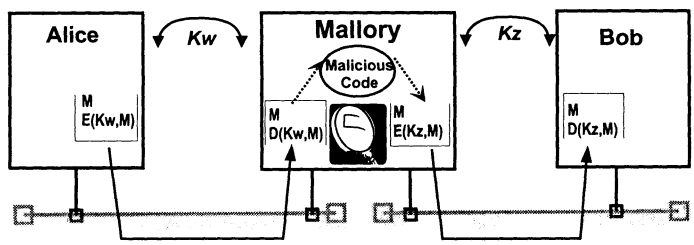

Figure 18.12. Man-in-the-Middle Attack

Mallory's attack is called a *man-in-the-middle attack*, or *bucket-brigade attack*. It belongs to the class of *spoofing attacks*, by impersonation in this case. The attack succeeds because D-H is a key exchange protocol without authentication. Next, we will see how to protect a D-H exchange with signatures or encryption.

Key-exchange with Public-Key Cryptography Alice and Bob wish to exchange a session key K_{ss}, but now they rely on public cryptography: Alice has private key Kr_a, Bob has Alice's public key Kp_a, and vice-versa for Kr_b and Kp_b. They can mutually authenticate themselves, as in the protocol of Figure 18.8.

One possibility could be to use one of the simple key-exchange mechanisms proposed for hybrid cryptography in Section 18.4.5: Bob generates a random key K_{ss} and sends it to Alice encrypted with her public key, $E_a(K_{ss})$; or Bob further signs the encrypted session key with his private key, $S_b(E_a(K_{ss}))$. The first approach would fail under a spoofing attack by impersonation. The second is robust, but a moderate problem is that it still is sensitive to read penetration attacks. If Mallory reads Alice's or Bob's state (i.e. $Kr_{a|b}$), he can decode past conversations.

Let us look at a more resilient protocol, depicted in Figure 18.13. This protocol can be seen as an enhanced variant of the encryption version (b) of the signature-based authentication mechanisms discussed in Section 18.5, adapted to mutual authentication and with the challenges encrypted since they are used in key generation. At the end, they can use both (now secret) challenges to generate a secret session key. More sophisticated operations than XOR can be envisaged. In fact, this is a modified version of the original public-key Needham-Schroeder authentication protocol (Needham and Schroeder, 1978). That protocol was recently shown to fall under a combined impersonation attack through Id spoofing and man-in-the-middle (Lowe, 1995). Step 2 has the fix w.r.t. the original: B's *Id* goes in the message.

		Action	Description
1	A $\rightarrow$ B	$\langle E_b(A, X_a)\rangle$	Alice sends *Id* and challenge to Bob, encrypted with his public key
2	B $\rightarrow$ A	$\langle E_a(B, X_a, X_b)\rangle$	Bob decrypts X_a, sends *Id* and challenge X_b together with X_a to Alice, encrypted with her public key
3	A $\rightarrow$ B	$\langle E_b(X_b)\rangle$	Alice sends Bob's challenge back, encrypted
4	A,B		Both believe they're talking to each other
5	A,B	$K_{ss} = X_a \; XOR \; X_b$	Both have two secret numbers with which to generate a shared secret key

Figure 18.13. Key-exchange with Public-Key Cryptography

Another protocol, this one based on enhancing the 'key-exchange with Diffie-Hellman' mechanism just discussed, is explained in Figure 18.14. The protocol takes the best of two worlds: it relies on the robustness of the Diffie-Hellman mechanism to derive a secret key without exposing *any* communication with it; it relies on signatures to authenticate the principals involved. Each principal starts the D-H mechanism (*see* Figure 17.5), and before exchanging their public numbers (Y_a, Y_b), they sign and identify the relevant messages. They can then generate K_{ss} according to the D-H algorithm, with the guarantee that they are really talking to one another. Compared to the simple signed scheme that we gave in the beginning of this section, where the key is generated by one principal who signs and encrypts it, and then sends it to the other principal, this is more robust, since it is no longer sensitive to read penetration attacks. This is an excellent key exchange scheme.

Key-exchange with Hybrid Cryptography Now Alice wishes to exchange a session key K_{as} with Stuart, using another protocol class, combining symmetric and asymmetric cryptography. The EKE (Encrypted Key Exchange) protocol of Bellovin and Merritt (Bellovin and Merritt, 1992) is an example.

		Action	Description
1	A → B	$\langle S_a(A, Y_a) \rangle$	Alice sends *Id* and D-H number to Bob, signed with her private key
2	B → A	$\langle S_b(B, Y_b) \rangle$	Bob sends *Id* and D-H number to Alice, signed with his private key
3	A,B		Both believe they're talking to each other
4	A,B	$K_{ss} = DH(Y_a, Y_b)$	Both have the two D-H numbers with which to generate a shared secret key

Figure 18.14. Key-exchange with Signed Diffie-Hellman

It has several variants and is specially suited for when authentication and key exchange depend on a user password and are thus vulnerable to guessing attacks. The basic protocol is explained in Figure 18.15. Alice has password P, and there is password-derived secret key K_p, shared by Alice and Stuart; she also has an asymmetric key pair $\langle Ku_a, Kr_a \rangle$.

		Action	Description
1	A → S	$\langle E_p(A, Ku_a) \rangle$	Alice sends *Id* and Ku_a to Stuart, encrypted with K_p
2	S → A	$\langle E_p(E_a(K_{as})) \rangle$	Stuart generates session key K_{as}, encrypts it with Alice's public key, further encrypts with K_p, and sends the result to Alice
3	A,S	K_{as}	Alice retrieves K_{as}
4	A,S		They may proceed with mutual authentication (e.g., steps 1-3 of Fig. 18.13)

Figure 18.15. Encrypted Key-exchange

The resilience against guessing attacks comes from the fact that the information collected by the eavesdropper is practically useless: to test the guesses, he would have to break a double (symmetric-asymmetric) encryption. The protocol used in DASS (*see* Section 19.4) is another example of key-exchange with hybrid cryptography.

18.7 PROTECTION MODELS

Ensuring that an authorized principal, and only it, can access data or a service: that is what *protection* is generically about. Subjects may be explicitly identified (authentication before authorization) or implicitly assumed (e.g., address-

based authorization). There may be one or more levels of protection. The system may provide *single-level security*, where a *security perimeter* is defined, and all the entities inside the perimeter are considered benign. Alternatively, when protection against *insider attacks* is desired, the need-to-know rule is enforced and several classes of access control are defined, in what is called *multi-level security*. Protection may impact all resources or just a few of them, that is, all system operations may or not go through a *reference monitor*. Access control rules may be implemented in an ad hoc manner or according to some formal rule set, that is, the system may follow a discretionary or a mandatory access control policy, in which case the policy is normally dictated by a *formal security policy model*. Underlying architectural measures further strengthen the protection mechanisms. These are the several facets of the protection problem, that we study in the next few sections.

18.7.1 Authorization

Authorization of access for a principal may or not assume previous authentication, since in some forms of interaction, the user is authorized access simply because she is in a particular situation (such as sitting in front of a given terminal, knowing the secret phone number of a dial-up connection, or having an given IP source address or port). In others, authentication has been performed previously, and the user gets an object (e.g., a cryptographic credential) that proves authorization without revealing identity (*see* Section 18.5).

In order to be able to be automated, and also verifiable, authorization must be dictated by a *security policy*. A security policy consists of specifying: the security classification of users (subjects), that is, their ranking in terms of the sensitivity of information they can access; the access classification of data, system services, resources in general (objects), that is, the sensitivity of these resources, in terms of the organization activity; and the access control policy.

The *access control policy* is the specification of the way subjects can access objects, according to each other's classification, the sensitivity of the objects in terms of confidentiality or integrity, and the use of the least privilege and need-to-know principles. Common *security classifications* are, in increasing order of sensitiveness: *unclassified, restricted, confidential, secret, top secret*. These originated in the military and although usable in other settings, commercial applications may follow a more suitable hierarchy: *public, proprietary, internal*. In the end, we may see the security policy translated into the triples $\langle s, o, r \rangle$ that form the access control mechanism (*see* Section 17.10), by following the rules dictated by the classifications and the access control policy.

18.7.2 Reference Monitor Model

We have already discussed access control. However, for it to be effective under a system perspective, a few additional questions arise: Does access control apply to all subjects and objects or only to some? Are the rules deterministic or can

users modify them? Is the access control mechanism itself protected or done by the general operating system functions?

The *reference monitor* addresses these questions, with the aim of formally ensuring protection. Originally proposed in (Lampson, 1974), many works have been based on it, broadening the scope of the original definition. In the rest of this part we will use the notion of reference monitor. To help us with our explanations, let us identify a few useful properties of a generic reference monitor (RM), exemplified in Figure 18.16:

Completeness - the RM is invoked for any access to any object;

Obligation - the RM refers to an access control rule set;

Self-protection - the RM is immune to intrusion.

The completeness property is secured if the RM stands between all subjects and all objects. The obligation property is compatible with mandatory access control policies. One may of course implement an RM that does not exhibit the obligation property, following a discretionary policy[5]. The self-protection property is secured if the RM resides on a trusted computer base (TCB). As we discussed in Section 18.3, in certain applications one may implement an RM over an imperfect TCB and in that sense partially fulfill the self-protection property.

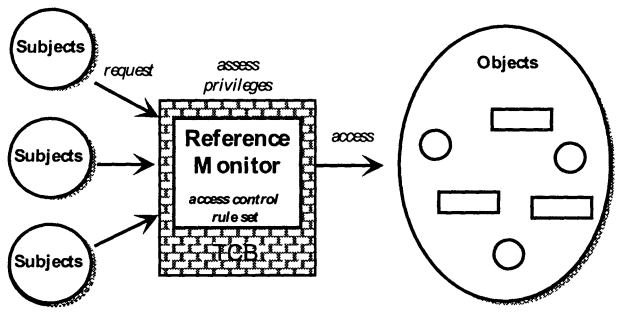

Figure 18.16. The Reference Monitor Model

18.8 ARCHITECTURAL PROTECTION: TOPOLOGY AND FIREWALLS

The way the architecture of the network is laid down helps implementing protection, and forms part of what we might call passive security measures. Devices like hubs, bridges and routers provide basic yet effective protection. Firewalls implement more sophisticated forms of architectural protection, both at physical and logical levels.

[5]The original work on the RM model did not follow a mandatory policy.

18.8.1 Topology

A naive form of internal network architecture is exemplified in Figure 18.17a. This flat approach has the consequence of exposing the whole infrastructure to an intruder that penetrates past the organization's entry router.

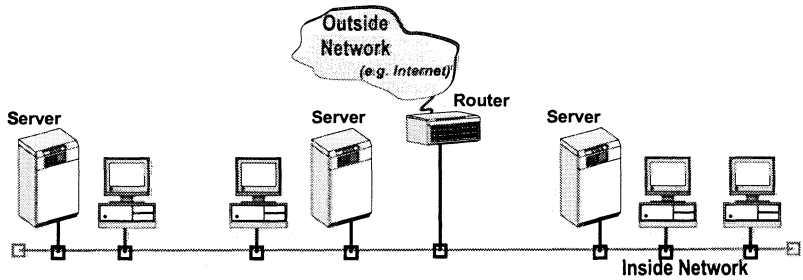

Figure 18.17. Network Architecture: (a) Flat

Figure 18.17b exemplifies the subnetting approach, that is, division of a network in two or more subnets, each with its own addressing mask, such that traffic is diverted right at the entry router to the appropriate subnet. This is valid both for traffic coming into the organization, and for traffic between different subnets of the organization. This division should be made according to an appropriate risk analysis: in the figure, the notion of a more exposed laboratory environment (Internal Network 2) versus the rest of the system (Internal Network 1) is patent. This is a primary form of error containment: intrusion in one subnet does not imply the immediate intrusion in the whole system. Subnetting also provides a convenient way for primary countermeasures: traffic may be easily blocked to/from specific networks, without affecting the operation of the whole organization.

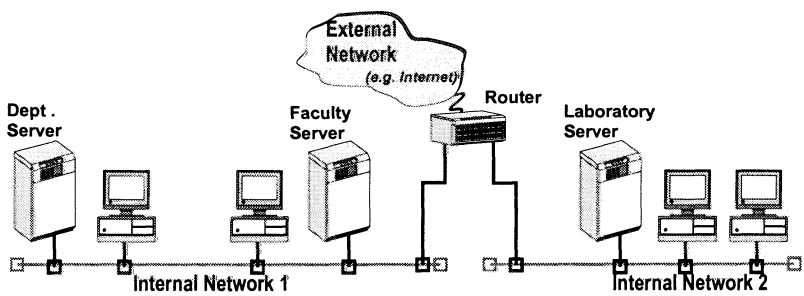

Figure 18.17 *(continued).* Network Architecture: (b) Subnets

The scenario depicted in Figure 18.17c exemplifies the utility of another device: the bridge. By placing the department services and the system administrator workstations in the Critical Network past the bridge, we neutralized

sniffing attacks coming from Internal Network 1. Last but not least, structured cabling, such as hub-based Ethernet, eases reconfiguration and primary countermeasures. Switched Ethernet further renders sniffing ineffective, since it restricts the broadcasting ability of the medium.

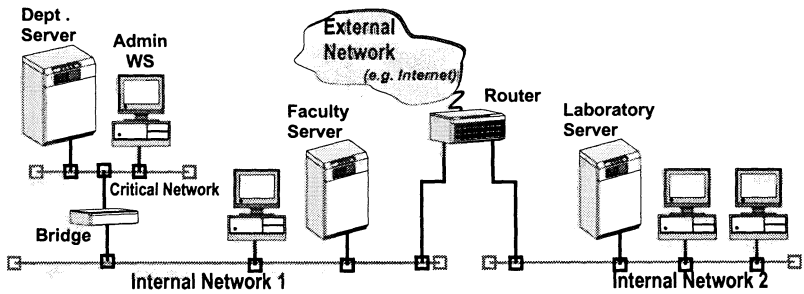

Figure 18.17 *(continued).* Network Architecture: (c) Bridges

18.8.2 Firewall Architecture

A basic firewall is like having a "doorman" at the (single) entrance of a facility: it allows or forbids information in and out, according to some criteria. More specifically, a firewall is a set of components placed between an external network and an internal network, with the following properties:

- all incoming and outgoing traffic must go through the firewall
- only authorized traffic must be able to get through
- the firewall hosts are trusted computing bases (TCBs)

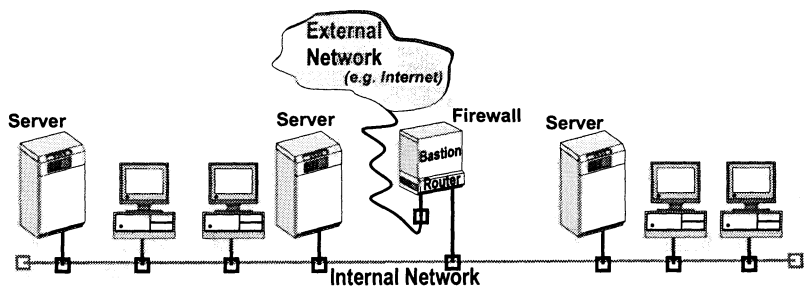

Figure 18.18. Single-level Firewall Architecture

The business of a firewall system designer is to approximate these properties, by putting the adequate firewall *functions* and *architecture* in place. Firewall architectures are built around routers, subnets and *bastions*, the trusted hosts that run firewall functions. These architectures take essentially two forms. The simplest, and most common, is a *single-level firewall* architecture, also known

as *screened-host firewall*, as depicted in Figure 18.18. The firewall comprises a router and a bastion, a combination also known as a *dual-homed* host, that stands between the external network (e.g., Internet) and the organization's (internal) network, such that all traffic is inspected by the bastion. In a variation of this architecture, the bastion stands single-homed in the internal network, but all outgoing and incoming traffic goes through the bastion (e.g., Internet → router → bastion → internal host, and vice-versa). This architecture places all hosts in the internal network at the same level of threat. The problem is that any current organization is bound to have services that should operate under different exposure scenarios.

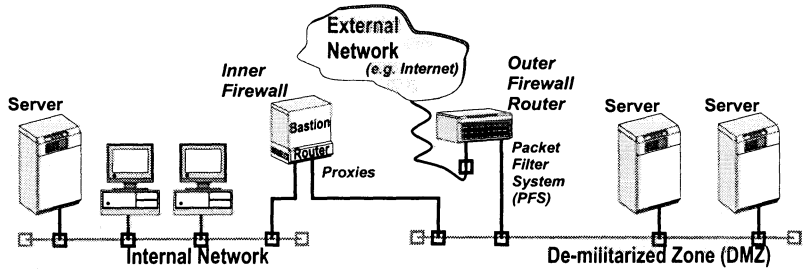

Figure 18.19. Two-level Firewall Architecture

Figure 18.19 presents the most used partial fix to that problem, a *two-level firewall* architecture, also known as *screened-subnet firewall.* The outermost firewall is normally also the outside router of the organization's network, and normally performs simple filtering functions. The inner firewall, a dual-homed host, performs more elaborate functions, such as representing internal protocols or applications. Between the inner and outer firewalls lies a subnet called the *de-militarized zone (DMZ).* The DMZ is the place to locate hosts necessarily subjected to high levels of threat, such as anonymous public front-ends, e.g., to Web page, directory or commerce services. The reason for placing these servers in the DMZ instead of just letting them be freely accessible from the Internet is that they can still enjoy some protection from the firewall system. Besides, the outer firewall is also a useful device for countermeasures: it can be instructed in real-time by the inner firewall to disable certain flows considered suspicious, that attack either the inner network hosts, or the DMZ hosts. In a variation of this architecture, the bastion stands in the DMZ, with all traffic between external and internal networks going through it. Several of these bastions may exist in a DMZ, processing different flows and services.

Firewall functions are the logical complement to the physical separation achieved by the architecture. They are divided into two main groups, that we study next:

filters - the traffic flow *passes through* the firewall to end services in the internal network, its content being inspected by filters on a go-no-go basis

proxies - the traffic flow *ends or starts in* the firewall, being intercepted and processed by representatives of end services in the internal network

Table 18.2 illustrates the main differences between the two. Filters work by inspection of packet data that goes through the firewall, and as such they are essentially stateless. Working at communications packet level, the filtering process is necessarily semantically limited (addresses, ports, interfaces). Rules are content oriented, such as *"deny packets containing source address X"*. Proxies intercept calls to the genuine servers, blocking direct communication through the firewall, and act as representatives of those servers. As such, they are stateful, dealing as much with data as with state, since they must handle the progression of service requests such as *"connect to FTP server"*. More than packets, they reason in terms of protocols, users, and services, which provides room for richer and thus more accurate protection semantics. Rules are both content and action oriented, such as *"allow user X to access service Y"*. Since nothing comes for free, proxy systems are normally less efficient than packet filter systems.

Certain firewalls implement a variant of PFS, called *stateful packet filter (SPF)*. Filters, although acting right above layer 3, probe further into each packet looking for known high-level protocol headers. SPFs are a form of *dynamic packet filters*, in that they adapt the rules to a flow of packets. Firewall-1 (*see* Section 19.2) is one example. On the other hand, a variant of proxy exists called *adaptive proxy*, where the proxy is capable of interpreting varying degrees of threat for different instances of a same service. Gauntlet (*see* Section 19.2) is one example.

Table 18.2. Comparison between Firewall Functions

Filters	Proxies
inspection	interception
stateless	stateful
data based	data+state based
poor semantics	rich semantics
packet level	protocol/user/service level
content oriented rules	action oriented rules
faster	slower

18.8.3 Packet Filter Systems

The principle of packet filtering systems (PFS) is shown in Figure 18.20a. The filtering mechanism is defined by the following:

- there is a list of rules to allow or deny packets through the filter in either direction;

- each list element is a tuple ⟨*action,origin,destination,type,direction,subnet*⟩, where: *action* is either *allow* or *deny*; *origin* is a complete source *Id*, for example TCP/IP address/port, or subnet address/mask; *destination* is the same, for a destination *Id*; *type* when available is the type of packet, which often corresponds to a given protocol; *direction* is one of *inbound* or *outbound*; *interface* represents the subnet from/to which the packet comes/goes, through one of the interfaces to which the firewall is directly connected;

- the headers of all incoming and outgoing packets are scrutinized against the contents of the list.

- the rules are applied in order; a packet is accepted immediately an "allow" rule becomes true, or rejected if a "deny" rule becomes true;

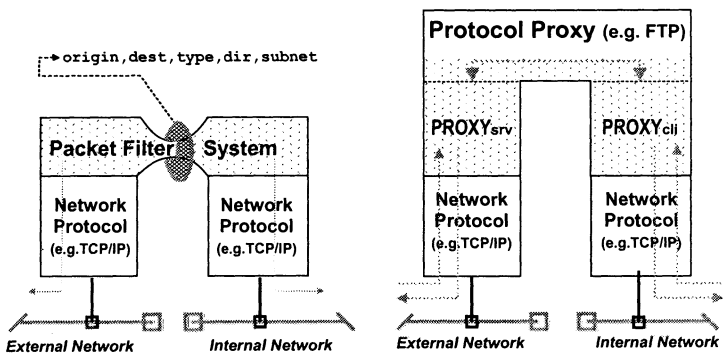

Figure 18.20. Types of Firewall functions: (a) Packet Filter, (b) Proxy

The PFS can be used for actions such as: blocking all packets coming from a suspicious host; disallowing telnet connections from the outside; enumerating the hosts allowed to access a given service. An example rule could be ⟨*deny from any to 194.117.21.00 typeTelnet inbound ie0*⟩, meaning that telnet packets coming from any machine in the subnet behind interface ie0, and addressed to any machine of protected subnet 194.117.21.00, are blocked. A prudent security policy can be implemented by using *allow* statements to specifically permit the desired traffic flows and denying everything by default as the last rule. Conversely, a permissive security policy will be implemented by using *deny* to block undesired flows, and allowing everything by default in the end. Firewall-1 (*see* Section 19.2) is a widely known example of PFS.

Network address translation (NAT) complements filtering to ensure logical separation. It consists of assigning invalid addresses to the internal network hosts, such that any traffic going in or out has to undergo a translation of the destination or source address, respectively, at the firewall router. This controls access of legitimate users to the external network, and hides the composition of the internal network to intruders.

18.8.4 Proxies

PFSs allow packet flows to internal hosts and are stateless: this means they cannot prevent direct probing and penetration attacks to internal hosts, as long as each individual packet looks innocent. If instead of inspecting passing traffic, it were intercepted and re-originated from the firewall, the chances of an intruder would be reduced. Figure 18.20b introduces such a firewall function, the proxy. Proxies reside in the firewall, and are representatives of application or protocol services (or daemons in UNIX language). A proxy stands for a genuine protocol server— such as HTTP, FTP, Telnet, or SMTP— which is normally in the internal network. The user wishing to access such a service is connected to the proxy server instead (left of the figure). The proxy client side performs the dialogue with the real server (right of the figure). Proxies relay all traffic back and forth between the user in the external network and the protocol server in the internal network. For that reason, proxies are also called *circuit gateways*. The proxy mechanism is defined by the following generic rules:

- there is a list of rules to allow, restrict or deny access to services;
- each element of the list is a service-specific tuple, that may comprise: *user Id* and *origin*; *service* and *server*; type of *access control* (e.g., ACM, ACL); type of *authentication* (e.g., password, signature); type of *restrictions* (e.g., from IDS); event *monitoring* and audit trail required; etc.;
- requests arriving at the proxy are tested against the rules, and serviced if "allowed" or blocked if "denied";
- state is maintained during service execution, to ensure it is carried on correctly.

Proxies significantly limit the freedom of action of an intruder in the internal network, and they are more precise than packet filters, since they act at a higher level of abstraction. A proxy rule may be something like ⟨*allow anyuser from domain cs.cornell.edu to ftp to FTP.di.fc.ul.pt requiring authentication X509_certificate requiring authorization local*⟩. It means that any user coming from cs.cornell.edu may attempt to log into the FTP server of di.fc.ul.pt, then pass an authentication process based on presenting a valid X.509 certificate, and then following the local ftp server ACL control.

18.8.5 Application Gateways

Application gateways are proxies working at a higher level of abstraction than circuit gateways. An application proxy server (a front-end of the application) is installed in the firewall, which for all purposes becomes the interface to the clients. The gateway, besides performing logging, validation and filtering functions, forwards the client's request to the real application server in the internal network, and receives the replies back. The overall picture is represented in Figure 18.20c.

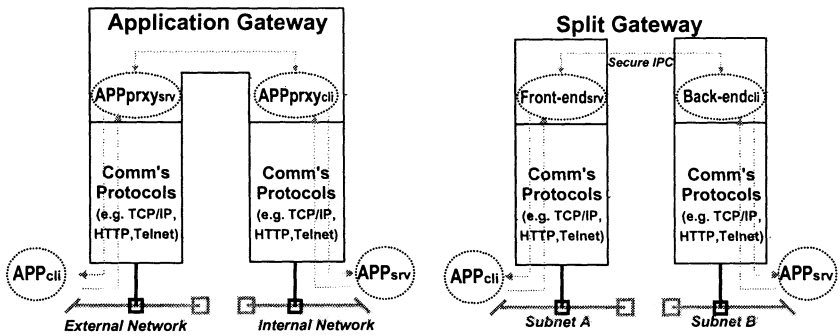

Figure 18.20 *(continued)*
. Types of Firewall functions: (c) Application Gateway; (d) Split Gateway

If all access from the outside is done through protocol and application gateways, an internal network may be completely closed to the outside at the protocol level. An intruder's freedom of action is then reduced to trying to tamper with the gateway or with the firewall's O.S. In consequence, application gateways are the most powerful and precise protection devices in firewall architectures, because they exert control at the highest possible semantic level: that of the application itself. However, they are also the less versatile and most difficult to build. Whereas COTS packet filters and protocol proxies abound for the best known network environments and protocols, application gateways must normally be custom made, with very few exceptions. Gauntlet (*see* Section 19.2) is a widely known example of proxy and application gateway system.

Note that there is nothing that prevents combining filters and proxies in the same bastion, or distributing them by the components of a firewall. In two-level firewalls, a current configuration is as shown in Figure 18.19: the outer firewall performs packet filtering and routing; and the inner firewall hosts protocol and application proxies. As a concluding remark, keep in mind that the effectiveness of the firewall functions we have described is in the inverse proportion of their generality. It is up to the architect to select the configuration that best addresses the tradeoff between security and functionality, according to the security policy previously defined.

18.8.6 Split Gateway Architectures

Sophistication of Internet-based distributed computing models calls for modularity and splitting of functions, in support of multi-tiered client-server operation. Today's applications are becoming significantly more complex and performance-demanding than is achievable by CGI-based connection of Web servers to application servers. These applications are using more powerful, stateful middleware based on the connection of front-end Web servers to back-end application servers, through enabling technologies such as Object Request Brokers, Internet Inter-ORB protocols (IIOP), JDBC and ODBC database con-

nectors, Web Request Brokers and Event Brokers. These constructs must be secure, and it makes sense that they operate across firewall architectures, for example, by running the back-end server in the internal network and then, both the front-end and the middleware in a firewall, or alternatively, the front-end server in a DMZ and the middleware in a firewall bastion.

To support the necessary modularity, address space separation, and location independence, we resort to what we may call a split gateway architecture, depicted in Figure 18.20d. It is an application gateway where the server-side and the client-side modules are stateful machines that operate independently, and communicate through a generic IPC. Both intra-host and inter-host IPC operation are supported transparently, such that the split gateway may either live on two separate hosts, as depicted in the figure, or in two disjoint realms of a protected O.S. in a single host. These hosts must be considered as bastions in what concerns their configuration and operation. IPC must be secure, in either configuration.

18.9 FORMAL SECURITY MODELS

In this section, we discuss formal ways of specifying and assessing security of computer systems. Namely, we discuss how, in a multi-level security mode, we specify which subject security classes can access which object access classes, and with what rights (e.g., *can a top-secret subject write to an unclassified object?*). Secondly, we describe standard security classification and evaluation criteria for computer systems (e.g., *how secure is computer system X?*).

18.9.1 Security Policy Models

It is obvious that for a verifiable goal to be attained, security policy rules must follow some formal specification. There exist several attempts to enforce mandatory access control policies in a formal way. The first such specification was the Bell-LaPadula model (Bell and LaPadula, 1973), which aimed at securing confidentiality. Other models followed, such as the Biba model, inspired by the Bell-LaPadula but addressing integrity instead (Biba, 1977), and the Clark-Wilson model, more adequate for enterprise systems (Clark and Wilson, 1987). Formal specification methods verifiable by model checking are discussed in (Ryan et al., 2000). There is not a generally accepted model addressing all needs, and this is a current research topic.

In generic terms, the relations between subjects and objects are established on the basis of their security classes and the need to know. We say s **dominates** o (or o is dominated by s) when the security class of s is at least as high as o's, and s needs to know o. We denote it as $s \geq o$.

Bell-LaPadula Model The Bell-LaPadula model, BeLa for short, describes the information flow in a system in terms of very primitive read/write operations. It aims at ensuring that the confidentiality property is respected in systems where data and computations of different security levels exist, and which can be accessed by subjects of different security classes. Consider sub-

ject s with security class $C(s)$, and object o, with security class $C(o)$. The access rights granted to s on o can be *read* or *write*. The properties of the BeLa model are then:

Simple Security Property - A subject s has *read* access to object o only if $C(s) \geq C(o)$

$*$-Property - A subject s has *write* access to object o only if $C(s) \leq C(o)$

Observe that a subject cannot read data from a level higher than its security class, that is, *read-up* is not possible. More counter-intuitively, the answer to the question in the beginning of the section— *can a top-secret subject write to an unclassified object?*— is "no!": a subject cannot write data to a level lower than its security class, that is, *write-down* is prevented. Also, if it writes data, it may not be able read it back. This causes a certain difficulty in retrieving, using and updating information for a repository. In fact, practical programming with this model entails significant complexity. However, note that by preventing read-up, confidentiality is protected in normal conditions. By making write-down impossible, information leakage attacks, directly or by means of Trojan horses, are blocked.

In the Biba model, which is the converse of the BeLa model, the security class interpretation concerns the sensitivity with regard to integrity. The model properties, conversely to the BeLa model, forbid *read-down* and *write-up*, as a way of preserving integrity of information. While no write-up is intuitive, note that the reason why read-down is prevented is to avoid corruption of the system with untrusted (low-integrity level) information.

18.9.2 Trusted Computer System Evaluation Criteria

Evaluating and grading the security of computer systems through objective evaluation criteria is important, for it allows us to compare systems of different models and makes. The Trusted Computer System Evaluation Criteria (**TCSEC**), or Orange Book (TCSEC, 1985) originating from the U.S.A., and the Information Technology Security Evaluation Criteria (**ITSEC**) (ITSEC, 1991), from Europe, were initial efforts in that direction. More recently, the bodies involved in both converged in a global standard, called Common Criteria for Information Technology Security Evaluation (**CC**) (CC-ITSE, 1998), bound to become an International Standard (ISO 15408) at the time of this writing.

The CC standard structures both the functionality requirements and the assurance requirements of a system i.e., what the product should do, and what trust can be placed in what it does. These requirements are expressed in terms of classes.

The standard is highly modular, making possible a huge number of combinations of classes of functionality and assurance requirements. However, it is expected that "typical" combinations emerge from the industry. The CC, like the preceding standards, provide several levels of trust. There are 7 Evaluation Assurance Levels (EAL) that measure the user trust on a system (EAL7 is

Table 18.3. Common Criteria (CC)- Security Levels

Levels	*Description*	*TCSEC equiv. Classes*
—		D
EAL1	functionally tested	–
EAL2	structurally tested	C1- Discretionary Security
EAL3	methodically tested and checked	C2- Controlled Access
EAL4	methodically designed, tested and reviewed	B1- Labeled Security
EAL5	semi-formally designed and tested	B2- Structured
EAL6	semi-formally designed, verified and tested	B3- Security Domains
EAL7	formally designed, verified and tested	A1- Verified Design

highest). Evaluation Assurance Levels are represented in Table 18.3, which also provides a mapping to the well-known TCSEC security classes. EAL1 applies when minimal protection, namely of personal information, is desired, but security is not a main concern. EAL2 and EAL3 are expected to have been tested against the functional criteria. In EAL4, it is expected that specific crucial subsets of the design have been methodically designed having security in mind, and that the whole has been thoroughly tested, and reviewed independently. Level EAL1-EAL4 assurance can generally be retrofitted into existing products and sub-systems (such as O.S.s). Levels above EAL4 require adequate design from the start. They provide maximum assurance, by application of specialized security engineering techniques. They also become more complex and expensive to implement. EAL5 would represent the top assurance still within the commercial systems area. EAL6 and EAL7 would apply to classified systems and military.

18.10 SECURE COMMUNICATION AND DISTRIBUTED PROCESSING

Secure channels and secure envelopes are basic paradigms of secure communication and the support for distributed processing models with security, such as remote sessions, RPC, and electronic mail. We study the above-mentioned models and mechanisms in this section.

18.10.1 Establishing Secure Channels

Secure channels, that we studied in Section 17.11, are one of the basic support primitives for distributed processing. They underlie file transfers, remote sessions, remote procedure calls, HTTP interactions, and so forth. They may be implemented in several ways. In what follows, we will make a general analysis of how to achieve each of the secure communication properties: authenticity, confidentiality, integrity. For simplicity, and without loss of generality, we consider the case of point-to-point communication.

Authenticity In order to achieve authenticity in a secure channel, the principals should authenticate themselves, in one of the styles shown in Figure 17.17. Mutual authentication desirably guarantees that both principals know whom they are talking to, mandatory if the channel is bi-directional. Several protocols are discussed in Section 18.5. If only symmetric cryptography is being used, shared-secret authentication protocols are the ones to be used. If asymmetric cryptography is available, then one can take advantage from the power of signature-based authentication. Since the channel is being established for exchanging a possibly large number of messages, with desirably low latency, it is convenient to avoid having to do authentication on every future message exchanged. Some of the short-term key exchange protocols studied in Section 18.6 are embedded with authentication, finishing by leaving the principals with a *session key* K_{ss}, known both of them and no one else. By majority of reason, any future message exchanged that has a suitable cryptographic function of that key enjoys the authenticity property if: messages are always encrypted with K_{ss}; or messages are cryptographically checksummed with a function depending on K_{ss} (*see* MACs in Section 17.5, and Figure 17.8). Note that the validity of these two assumptions relies on subtle aspects: (a) that the recipient knows something about the partial content (e.g., headers) or about the structure of messages; (b) that forging or modifying the encrypted product, may not possibly yield something intelligible in terms of (a), after decryption; (c) as (b), for the cryptographic checksum and its verification.

Confidentiality When confidentiality is desired, the communication must be encrypted. A channel using exclusively asymmetric cryptography for both authentication and encryption would be extremely simple and secure. However, it is not recommended for immediate communication, since it is very slow. Practical secure channels resort to two alternatives: purely symmetric cryptography; hybrid cryptography. In a symmetric system, after shared secret authentication and key exchange have been performed, as we just saw, the channel has a session key. With hybrid cryptography, principals start with using their public/private key pairs to exchange, encrypted and/or signed, a session key as well (*see* the protocol described in Section 18.4.5). After that, in both approaches, they use that key for symmetric encryption/decryption.

Integrity In order to achieve integrity, messages must be cryptographically protected in a way that any modification whatsoever (accidental or intentional) is detected. This can be achieved in one of two ways: by a cryptographic checksum, such as suggested for authenticity (in this role, it is often called a message integrity check, or MIC); or by a digital signature. Note that the proviso that we made under *Authenticity* above, on the recipient having to know what it expects after decryption or verification, still applies here. Integrity can also be protected by encryption, but the method is not general, since it requires a modified message to decrypt to garbage, and this is not always achievable.

18.10.2 Secure Tunnels

The simplest form of building a secure channel through a network such as the Internet is shown in Figure 18.21a. Encrypted and/or integrity protected payload data blocks are encapsulated in protocol packets, i.e., IP datagrams. This does not always work. Take networks A and C, interconnected by network B, and a problem: we want a packet to go from A to C, but for some reason it cannot circulate through B (e.g., B does not understand the protocol). The first approach will not work, so we have to use another form of secure channel.

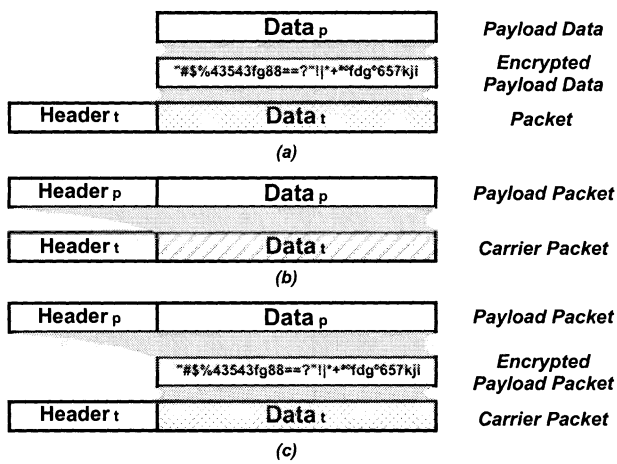

Figure 18.21. Tunneling: (a) Secure packet; (b) IP-over-IP; (c) Secure Tunnel

Let us start by understanding what a **tunnel** is: the encapsulation of a whole *payload packet* that circulates in network A, in a *carrier packet*, that circulates through network B, until network C, where the carrier packet is de-capsulated and the payload packet circulates again until the final destination. A classical tunnel is IP-over-IP, whose packet structure is depicted in Figure 18.21b. It consists of encapsulating a full IP datagram as if it were an upper layer service data unit, in another IP datagram. A **secure tunnel** is then a tunnel that guarantees the properties of a secure channel to the data carried inside it. Figure 18.21c suggests how it can work: the whole payload packet is treated as

a block of data, and is encrypted and/or integrity protected. The result is then treated as data and encapsulated in the carrier packet. In the final destination, the operations are repeated in reverse order: the outer packet is de-capsulated, the data decrypted/verified, and the inner packet launched on the network for its final destination. IPsec (*see* Section 19.3) is the forthcoming standard for secure channels and tunnels on the Internet.

18.10.3 Distributed Authentication and Authorization

Among the several functions that assist secure computing, two are fundamental: authentication and authorization. We studied them independently, and we saw that one can be done without the other: secure communication requires authentication but not authorization; access to information requires authorization, but does often without authentication.

For example, message authentication can be achieved through MACs, message authentication codes, which are a *cryptographic checksum* technique, or through a *digital signature*. The disadvantage of MAC w.r.t. signatures is that it is a shared secret technique, and thus a participant cannot hand the message over, that is, persuade third parties who do not share the secret, of its authenticity.

Access to services and information often resorts to the assistance of a Security Server (SS), which designates a trusted third party that performs authentication and authorization. Let us see how these two functions work together in supporting client-server operation. In what follows, we use AT for the authentication service, and AC for the authorization or access control service, although they are often co-located in the same server:

1. *Client C and the AT run a protocol in order to authenticate C to the AT, in the course of which C receives an authentication certificate, Acert, for further use in the system*
2. *C wishing to use resource or service S, runs a protocol with AC, if necessary using Acert, requesting authorization to access S, in the course of which C receives a privilege certificate, Pcert, for S*
3. *Client C, using Acert and Pcert, presents itself to the server hosting S, and they run a protocol aimed at:*
   - *authenticating C to the server, based on Acert;*
   - *validating C's privileges to access S, based on Pcert and the access control list maintained by the server*
4. *If C is cleared, access is performed to S*

The content of *Acert* and the protocol steps run between C and SS, and C and S, depend on the type of protocol being used, either an arbitrated (KDC based) or a certified (CA-based) trusted-third-party protocol (*see* Section 18.4). The content of *Pcert* depends on the particular way access control is performed, i.e., whether it is capacity or ACL based, or both (ACM). Authentication may be mutual, generalizing the use of authentication certificates, that is, S may have its own *Acert* that authenticates it to C. Kerberos (*see* Section 19.4) is an example of a KDC-based security service.

18.10.4 Secure Remote Session

Remote sessions (*see* Chapter 1.3) were among the first paradigms of distribution ever used. Primitives `rlogin`, `telnet`, `rsh`, or `ftp`, are distinguished examples of long-lived distribution paradigms. Let us follow the steps of establishing a remote session, seen from a connecting site:

1. *bind to remote host socket*
2. *establish low level network connection between hosts*
3. *communication starts*
4. *perform cleartext remote login (e.g. telnet) authentication— authentication follows traditional mechanisms (password, address-based)*
5. *start session, also in cleartext*

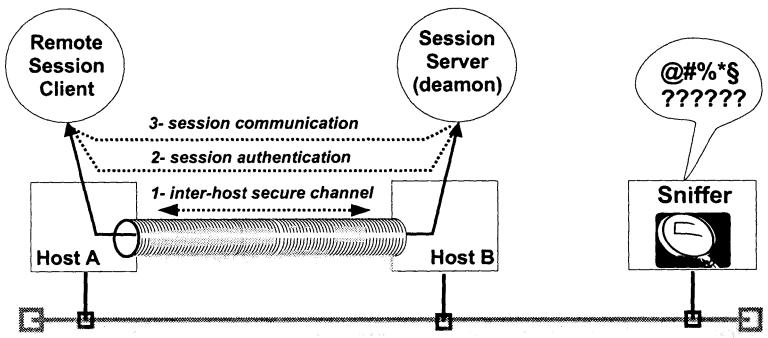

Figure 18.22. Secure Remote Session

However, remote session protocols were not designed to have security in mind, and became one of the most exploited vulnerabilities in distributed systems. Eavesdropping attacks easily yield not only conversation contents, but also, and more importantly, login/password pairs. A secure session should not be vulnerable to those attacks. To understand the principle of *secure* remote session, remember the desirable properties of the underlying secure channel: authenticity, integrity, confidentiality. The steps of establishing the secure session are the following (compare them with those of the cleartext session):

1. *bind and authenticate to remote socket*
2. *establish low level secure channel between hosts*
3. *encrypted communication starts*
4. *perform remote login (e.g. telnet) authentication*
5. *authentication can either follow traditional mechanisms (password, address-based) protected by secure channel, or be cryptographic (e.g., public key)*
6. *start session, either in cleartext or encrypted*

Observe Figure 18.22: notice that the first step is to establish a tamperproof channel between the hosts wishing to communicate, so that external attacks are not possible. However, at this point, the connecting host might be an attacker,

or the connected host might be a spoofer, and either would act inside the secure channel. Then, the second step is to authenticate whoever is using the secure channel in the other extremity. For a moderate level of security, the authentication mechanism may be a classical login/password pair, which now goes encrypted on the network. However, for demanding levels of security, it may alternatively be any of the strong, cryptographic authentication mechanisms that we studied in Section 18.5, such as public key signature. Also depending on the security versus performance tradeoff, the third and final step, session communication on the channel, may subsequently be encrypted or not. SSH and SSL (*see* Section 19.1) are examples of secure remote session protocols.

18.10.5 Secure Client-Server with RPC

The security concerns address not only remote sessions, but client-server operations in general, such as RPCs. Many current services and applications are based on RPC: some of which carrying sensitive information, such as distributed file systems like NFS; others perform sensitive operations, such as transactional managers. This concern raises the need for secure RPC: a remote procedure call facility with strong authentication, encryption and protection. In order to prepare for a secure RPC, the client must execute the following steps:

1. *client binds to the desired service/server as usual*
2. *in that process, the client makes a call to the RPC runtime instructing it that this is a secure RPC and specifying the desired security options:*
    - *type of authentication;*
    - *level of security;*

On the server side:

1. *server exports name and registers security capabilities with RPC runtime*
2. *server initiates activity, normally with a login to the security service*
3. *server checks the security attributes of each call, and:*
    - *authenticates according to the chosen type;*
    - *performs access control based on client's authorization for the invoked service (e.g. ACL based)*
4. *if all is OK, the service is executed with the required level of security*

As an example, Table 18.4 lists the typical options of a secure RPC. Type of authentication specifies what kind of authentication model is followed. Level of security involves combinations of integrity and confidentiality assurance. The server acts as a reference monitor for the secure RPCs performed by clients, authenticating them and then checking their authorization for the invoked operation. SUN Open Network Computing (ONC) RPC and DCE RPC are examples of RPC packages with security facilities. In conclusion, note that using secure RPC instead of normal RPC does not involve much complexity, besides one or two additional calls to the runtime environment.

Table 18.4. RPC Security and Authentication Options

Parameter	*Options*
Authentication	none name-based (UNIX-like) shared-secret signature
Security	none integrity only integrity and confidentiality

18.10.6 Secure Envelopes and E-Mail

Secure envelopes complement secure channels as the basic communication support primitives for distributed processing. They are relevant in electronic e-mail, messaging systems in general, and transactional systems. Secure envelopes should resort to per-message security. There is no point in establishing a connection since message sending is sporadic. Again, let us see how each of the secure communication properties that we studied in Section 17.11 is achieved. We continue to consider the case of point-to-point communication, and consider the situation where there is no shared secret between principals, but public/private keys are in place, and known to the principals as appropriate.

Authenticity In order to achieve authenticity of a message in a secure envelope, the message should be signed with the sender's private key. The recipient can authenticate the sender by verifying it's signature. It is obviously wise to sign a digest of the message, as depicted in (with E_K as an asymmetric cipher, and K as the private key of the sender). See also Figure 17.10 in Section 17.6 for a description of such a protocol.

Confidentiality When confidentiality is desired, the message must be encrypted. Although the envelope is used for deferred communication, there is no point in being inefficient. In consequence, we will only use asymmetric encryption of the message for special cases of very small (control?) messages. Otherwise, hybrid cryptography with symmetric encryption of the message content seems more appropriate. The mechanism for generating a hybrid cryptographic envelope depicted in Figure 18.6 is perfectly appropriate.

Integrity After the steps to achieve either authenticity or confidentiality are performed, we have in place mechanisms to secure integrity. A digital signature as performed for authenticity guarantees message integrity. Encryption as performed for confidentiality also guarantees message integrity. Keep in mind the general remarks concerning integrity made throughout this section.

Secure E-Mail makes extensive use of the secure envelope concept, and in consequence secures the desirable properties of: *authenticity* of the sender of a message; *confidentiality* or privacy as used in this context, ensuring that only the recipient will read the message; *integrity* ensuring that the message is received as sent. Besides these obvious properties, electronic mail must have other security properties in emulation or improving those of paper mail:

- **non-repudiation of sending** - the recipient can prove that a message came from a given sender, who cannot deny
- **non-repudiation of delivery** - the sender can prove that a message was delivered to a given recipient, who cannot deny
- **anonymity** - the ability to deliver a message without revealing the identity of the sender
- **timestamping** - delivered messages can be totally ordered, even if a posteriori

PGP and PEM (*see* Section 19.1) implement the secure envelope concept and are used for secure e-mail.

18.11 ELECTRONIC TRANSACTION MODELS

What is an electronic transaction? It is a transaction involving assets, financial and other, made over computer and network systems. It assumes virtual payment instruments, which emulate conventional transaction protocols by informatic means. Electronic transactions (ET) are distributed in their nature, and assume several facets, which derive from the type of interaction of the players, the values involved, and the timing of payment that is, whether:

- ET values involved are average to low, typical of personal retail transactions, or are high, typical of wholesale inter-bank transactions;
- ETs take place on proprietary, or on open networks (e.g. Internet)
- buyer and merchant are introduced by a mediator, or just contact spontaneously;
- ETs need to contact the supporting infrastructure (e.g. PKI) on-line, or can perform off-line;

We are interested in personal retail transactions that take place on open networks, and in their several facets.

18.11.1 A Generic Model of Electronic Transaction

The generic model of ET is depicted in Figure 18.23. The main players are described in Table 18.5. The trusted third party materialized by the hierarchy of e-comm-specific certification authorities of the ET PKI is important to build trust between principals. When it is absent, such as in systems using only symmetric cryptography, the versatility of the system is limited, and the mediating role of the acquirer is bound to be more active in each transaction. When it exists, and whether it is concerned with credit card business, or digital cash or cheques, there is bound to be a Root CA which should be as independent as

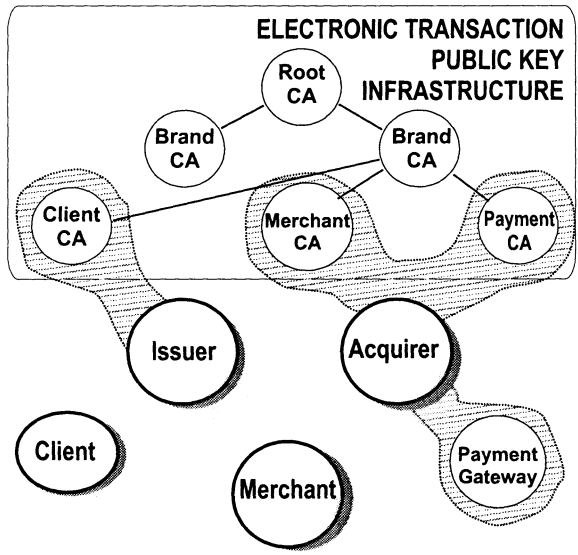

Figure 18.23. Generic Electronic Transaction Model

possible, so as to be above the Brand CAs (e.g. Mastercard). Brand operators break deals with Acquirers and Issuers, who set up their CAs under the Brand hierarchy. Real Acquirer and Issuer institutions may have deals with different Brands (e.g. Visa and Mastercard), but must maintain separate virtual trust chains. Acquirers also set up a payment gateway, the technical interface to the independent banking network, through which payments flow.

Table 18.5. Electronic Transaction Participants

Issuer	Normally a banking institution that issues the payment instruments: credit, debit or purse cards, digital cash, etc.
Client	The buyer in the transaction, he is normally a card holder and has an Id certificate, and is the transaction initiator
Merchant	The supplier of the good, he has an Id certificate
Acquirer	The mediator in the process, normally a financial institution that serves as a broker between the other players
ET PKI	The public key infrastructure, or a subset of it, concerned with facilitating electronic transactions, it issues all certificates

18.11.2 Classes of Electronic Transactions

According to our initial remarks, we may informally divide transactions in three classes, described in Table 18.6.

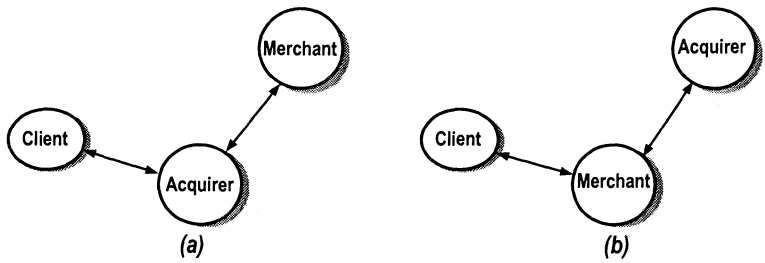

Figure 18.24. Electronic Transaction Classes: Mediated vs. Spontaneous

The mediated versus spontaneous classes are depicted in Figure 18.24. Observe that in the mediated class (Figure 18.24a), the mediator is in the way of the transaction, which must always be performed on-line. This is typical of earlier generation systems, where the rudimentary cryptography made it necessary that security and authenticity were ensured by the physical architecture, such as using proprietary networks and dedicated terminal devices. ATM networks are an example, where the transaction is "buying" money or paying for goods with the debit card. In these systems, the mediator soon becomes a bottleneck. The spontaneous class (Figure 18.24b) makes it possible for the client to produce credentials that authenticate it to the merchant, allowing the transaction to proceed as far as possible.

Table 18.6. Classes of Electronic Transactions

Mediated	The client must each time be introduced by a mediator that builds trust between the client and the merchant
Spontaneous	The client contacts the merchant spontaneously and presents stand-alone authentication credentials (e.g. certificates); the ET terminates either on-line or off-line:
Off-line	The credentials presented are enough to complete the transaction by the sole communication between client and merchant (e.g. digital cash)
On-line	The credentials presented are not enough to complete the transaction, requiring communication with the support infrastructure (e.g. checking credit card validity and ceiling)

The spontaneous off-line versus on-line classes are depicted in Figure 18.25. Continuing our discussion, observe that in the off-line case (Figure 18.25a), the client, prior to the transaction, acquires payment instruments to the issuer (1)

that have stand-alone validity (e.g. she loads her smart card electronic wallet with digital cash at the card issuing bank). Then, she contacts the merchant, performs some electronic commerce protocol and finally gets to the stage of payment. The transaction can proceed off-line because the merchant not only believes in the client's Id certificate, but also on the payment credentials she produced (2), and gives her the goods after keeping her payment. The merchant can later contact the acquirer (3), possibly with a bunch of payments, and consolidate them. The acquirer collects the issuing bank (4) through the banking network.

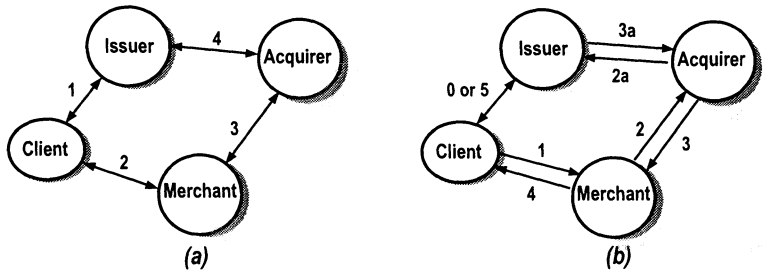

Figure 18.25. Spontaneous Electronic Transaction Classes: off-line vs. on-line

Sometimes, although the merchant accepts the client's Id certificate, the validity of the payment credentials must be checked, e.g. to prevent fraud (e.g. to check double spending of digital cheques, or the validity of credit cards). This requires the transaction to go on-line (Figure 18.25b): the client addresses the merchant (1) and when they get to the payment phase, the merchant goes on-line and contacts the acquirer (2). Depending on the specific type of transaction (e.g. credit or debit), the acquirer may return the payment authorization code (3) after performing local checks (e.g. credit card ceiling and validity), or may instead contact the issuing bank (2a-3a) for further checks (e.g. cheque double spending, account balance). When authorization comes, the merchant handles the goods to the client (4). It may capture payment later, because it has the irrevocable authorization code. Whether the client acquired payment instruments beforehand (0) or the issuer later collects from the client (5), depends on the specific business. An example of the former would be digital prepaid cheques, and an example of the later would be credit card operations.

18.11.3 An Analysis of Electronic Transaction Security

The cryptography used in ETs is relevant for the class to implement, and for the security properties observed by the client and by the operators, namely privacy and fraud protection. *Symmetric* cryptography does not allow mutual authentication in spontaneous transactions, and as such in absence of a arbiter or adjudicator, fraud cannot be effectively handled. Only mediated on-line transactions apply. Privacy is thus not great. The use of *asymmetric* cryptography is a pre-requisite, in order for certification authorities to be in place. In this

case, spontaneous on-line transactions are immediately possible that achieve fraud prevention, since clients can present stand-alone identity certificates.

Off-line ETs are also possible, with protocols where fraud, although not prevented, can be detected, and the culprit identified. But this also depends on the business risk analysis and on how efficient is the court system. With *asymmetric* cryptography and *blind signatures*, one can build protocols that support non-traceable spontaneous ETs, with digital cheques or cash. That was shown in Section 17.7: on-line transactions prevent fraud, whereas with off-line transactions, fraud can at least be detected. The tradeoff to be made depends again on the risk analysis. Another aspect of the problem is that most current ET systems provide *one-sided security.* It is highly desirable to go towards *multi-party security* architectures, where the security of each participant does not depend on his a priori trust on the other participants, and where privacy and non-traceability can be ensured as much as possible. We are discussing these issues further in Section 19.5.

18.12 SUMMARY AND FURTHER READING

In this chapter, we discussed the main models of secure distributed computing. The first objective of the chapter was to provide insight to the system architect, about the main architectural options, strategies and frameworks that she has available. The second was to discuss the main models in a problem-oriented manner, establishing links, whenever possible, to the paradigms learned in the previous chapter.

As further reading, we advise the following works. Slade does a fairly complete practical study on computer viruses (Slade, 1995). Neumann gives an interesting account of several security related risks and hazards, some of them caused by a wrong evaluation of the severity of faults, or by the layout of inadequate strategies (Neumann, 1995). Further study on Lampson's model of distributed authentication can be found in (Lampson et al., 1992). In (Abrams et al., 1995) there are excellent studies on access control mechanisms and policies, and on security policies.

On attacks and countermeasures, there is an anti-eavesdropping mechanism described in (Rivest and Shamir, 1984), using an *interlock protocol.* Sophisticated spoofing attacks against Web pages or network downloadable software are reported in (Brewer et al., 1995; Felten et al., 1996). Attacks on using the same public key protocol for signing and encrypting are detailed in (Dolev and Yao, 1981; Kaufman et al., 1995), or (Schneier, 1996). Abadi and Needham do a study on attack-resilient design of protocols in (Abadi and Needham, 1994). Needham discusses attacks to a secure channel in (Needham, 1993).

A distributed TCB implementation is discussed in (Nicomette and Deswarte, 1997). There is an advanced discussion on authentication pitfalls in (Kaufman et al., 1995). Authentication and key distribution protocols can be further studied in the following publications. An attack on Needham and Schroeder (Needham and Schroeder, 1978) original protocol was reported by Denning in (Denning and Sacco, 1981), and corrected in (Needham and Schroeder, 1987).

Otway and Rees improved the latter protocol in (Otway and Rees, 1987). Denning and later the Kerberos protocol (Neuman and Ts'o, 1994) proposed to use timestamps as a form to foil replay attacks. However, Gong (Gong, 1992) showed that if an attacker succeeds in de-synchronizing the clocks, he can replay old messages that seem current to the slow clocks. This is called a *suppress-replay attack.* Neuman and Stubblebine corrected the problem in (Neuman and Stubblebine, 1993). See also (Gollmann, 2000) for a discussion on the pitfalls of verification of authentication protocols.

On protection, Cheswick and Bellovin wrote one of the most complete essays on firewalls (Cheswick and Bellovin, 1997). Formal access control models other than BeLa and BiBa exist, such as Denning's (Denning, 1976). Recent work on classification criteria taking fault tolerance and security both into account is the Squale Criteria (Corneillie et al., 1999). Security kernels are discussed with detail in (Ames et al., 1983; Schell, 1984). On the programming side, the Generic Security Service API (GSS-API) is an attempt to standardize an API for secure operations, independently from platform (Linn, 1996). The advantages are obvious, and for example, Kerberos V.5, among other products, is GSS-API compliant.

19 SECURE SYSTEMS AND PLATFORMS

This chapter gives examples of systems and platforms for secure computing. We are going to talk about remote operations and messaging, firewall systems, virtual private networks, authentication and authorization services, smart cards and payment systems, and secure electronic commerce. In each section, we will mention several examples in a summarized form, and then will describe one or two the most relevant in detail. Table 19.2 at the end of the chapter gives a few URL pointers to where information about most of these systems can be found. The table also points to the IETF Request for Comments site, where the RFCs cited can also be found.

19.1 REMOTE OPERATIONS AND MESSAGING

There exist a few remote secure session packages. The Secure Sockets Layer (SSL) is a basic secure channel plus a few ancillary protocols, which allows high-level remote session protocols to work securely, in a transparent way. Developed and used initially by Netscape, it ended-up as a de facto standard in its Version 3.0, and a variation of it is currently endorsed as a standard of the IETF, the *Transport Layer Security, (TLS V1.0 - RFC2246).* There is a freeware version of SSL 3.0, SSLeay independently developed by Eric Young, that is currently incorporated in the Apache HTTP server. based on SSLeay, the OpenSSL is a collective initiative for developing and making available free SSL software. Also with relation to HTTP, there is an alternative protocol, Secure HTTP

P. Verissimo et al. *Distributed Systems for System Architects*

(SHTTP) for achieving secure HTTP interactions, that has been around for years but has been overtaken in importance by SSL. When all that is needed is user authentication, secure, MAC based authentication of plain HTTP is specified in Basic and Digest Access Authentication (RFC2617). Secure Shell (SSH) is a suite of remote session protocols originally developed by Tatu Ylonen, commercialized by Secure Data Fellows, and currently endorsed as an IETF draft standard called SECSH. There are free versions of SSH for some systems, e.g. Linux. The SECUDE package is a freeware set of modules, libraries and APIs for developing remote session protocols, from GMD-Darmstadt in Germany. STEL is a freeware secure telnet developed at the University of Milano. S/Key is a one-time password system based on Lamport's hash, developed at Bellcore and now endorsed as an IETF RFC under the name of OTP, One-Time Password System (RFC2289). Privacy Enhancement for Internet Electronic Mail (PEM) and Pretty Good Privacy (PGP) are the two best known secure messaging packages, and can be used for secure e-mail amongst other things. PEM is a set of IETF RFCs (1421-1424) based on a somewhat complex structure, involving the PKI certification authority hierarchy. PGP is more lightweight in key management, and more versatile in functionality. PGP is currently undergoing a standardization effort, OpenPGP Message Format (RFC2440), to ensure interoperability of different implementations. Sun RPC and DCE RPC are examples of secure RPC packages. RSADSI supplies a few building modules for use in this kind of packages, such as RSAref, the main library of RSA cryptographic functions, and PKCS, the standard for formatting and encoding of cryptographic structures. Next, we analyze SSL, SSH, PGP and S/Key in detail.

19.1.1 Secure Sockets Layer (SSL)

SSL V3.0 has a basic secure channel layer, implemented by the *Record Protocol*, which uses a socket abstraction and runs on top of any transport protocol, such as, but not limited to, TCP/IP. This layer only knows about establishing a low-level secure channel and sending blocks of data back and forth, in a secure manner. SSL secure channels can securely encapsulate high-level session protocols, such as: HTTP, FTP, SMTP, or POP3. For example, to use HTTP with SSL, you just have to type URLs in the form `https://...`. SSL provides remote sessions with:

- anonymous, unilateral or mutual client/server authentication, with digital signature certificates whenever supported
- data compression
- communication encryption via symmetric cryptography
- message integrity via authentication codes (MAC)

A few ancillary SSL protocols recursively use the record layer to extend the capability of SSL to support secure remote session protocols. These are: the *Handshake Protocol*, the *Alert Protocol*, and the *ChangeCipherSpec Protocol*. The Record Protocol provides confidentiality and/or integrity of user message

flows, encapsulating user data in `record` messages, which are either protected with a MAC or MAC-protected and encrypted. The Handshake protocol performs client and server authentication. The Alert protocol signals errors and exceptions through `alert` messages. The ChangeCipherSpec protocol is used whenever the cipher specifications change. This can be done in the middle of a session. SSL Version 3.0 supports RSA, X.509 certificates or Fortezza[1] for authentication. Encryption may be done by DES or RC4, with a 128-bit key limited to 40 bits for export versions. Integrity is secured by means of SHA and MD5 MACs.

Handshake Protocol The Handshake protocol (*see* Figure 19.1) initiates a session, performing negotiation, authentication and session key exchange. The protocol starts with the client and server exchanging nonces (*C-Random, S-Random*) negotiating the SSL version, session Id, and type of cryptography and compression (10,11). Next, authentication and key exchange takes place. The server normally sends its public key certificate (20a). Alternatively, if it does not have one, it sends a key exchange message (20b) with additional data to make an ad hoc key exchange. The server may also request a client certificate, if mutual authentication is desired (21-22), otherwise, only the server authenticates to the client. At this point, Hello is terminated (23). The client now sends a client key exchange message (30) to set up the initial cryptography. Both exchange now messages specifying the type of cryptography that will be used, and finish by sending one another *Finished* messages (32-33).

Authentication and Key Generation The goal of the key exchange process within the Handshake protocol is to create a *pre-master-secret*, which in turn will lead to a *master-secret*, from which all other keys will finally be derived. The initial key exchange depends on the authentication mode (anonymous, unilateral or mutual) and the cryptography suite, which may be RSA or Diffie-Helman, either plain or signed.

We will study *authenticated RSA*, the most relevant for secure Web transactions. Authentication is combined with key exchange. The client creates the *pre-master-secret*, a 46-byte random plus 2-byte version ID. The client then encrypts the *pre-master-secret* with the public key in the server's certificate, sent in the respective *S-Cert* message to the client, and sends it to the server, in an *EncryptedPremasterSecret* record of the *ClientKeyExchange* message. When the client receives the *Finished* message, it may deduce that the server has successfully decoded the *EncryptedPremasterSecret*, and is thus authentic.

At this point, both the client and the server have the *pre-master-secret*. The *master-secret* is computed with a few hashing operations having the *pre-master-secret*, *C-Random*, and *S-Random* as parameters. After a few more hashing operations, the cryptographic checksumming (MAC) and encryption keys are extracted. The *ChangeCipherSpec* message synchronizes the end

[1]Fortezza is a key escrow hardware assisted protocol that we will not address here.

	Action	Description
10	C → S	Client C opens connection with *ClientHello* message: ⟨*C-Random, SessionId, CipherSuites, CompressionMethods*⟩
11	S → C	Server S sends *ServerHello* message: ⟨*S-Random, SessionId, CipherSuite, CompressionMethod*⟩
20a	S → C	*S sends its certificate:* ⟨S-Cert⟩...
20b	S → C	*or S sends a* ServerKeyExchange *message*
21	S → C	*S sends the client a* CertificateRequest *message*
22	C → S	*C sends its certificate:* ⟨C-Cert⟩
23	S → C	S sends *HelloDone* message
30	C → S	C sends a *ClientKeyExchange* message
31	C → S	*C sends S* CertificateVerify *message*
32	C ↔ S	C and S both send *ChangeCipherSpec* messages
33	C ↔ S	C and S both send *Finished* messages
-	C,S	They are authenticated and have a secure channel set up

Figure 19.1. SSL Handshake Protocol (italicized steps are either optional or alternative)

of this process. The subsequent *Finished* messages go MAC-protected and encrypted with the recently negotiated keys, and are used to test if the process was successful.

This concludes our study on how to make secure sessions on the Web with SSL. Given that most extranet (and also intranet) access to applications is currently via Web protocols, the security of the architecture per se deserves some brief comments. Despite the cryptographic material available for secure web-based applications, these may fail on account of hidden vulnerabilities of browsers, servers, and languages themselves, from HTTP to Java. So, much attention should be given to configuration and operation of web-based systems in a way that their security is not jeopardized by those vulnerabilities.

19.1.2 S/Key

S/Key one-time password (OTP) system is a simple package aiming at protecting remote sessions from passive eavesdropping attacks. It does not store sensitive information, and works with personal terminals, from workstations and PCs, to CRT terminals. S/Key can also authenticate FTP, besides Telnet.

Principle of Operation The OTP mechanism is inspired by Lamport's hash (*see* Section 18.5, *Password-based Authentication*). A primordial secret password p_s is generated from a random number *seed* and a secret user passphrase

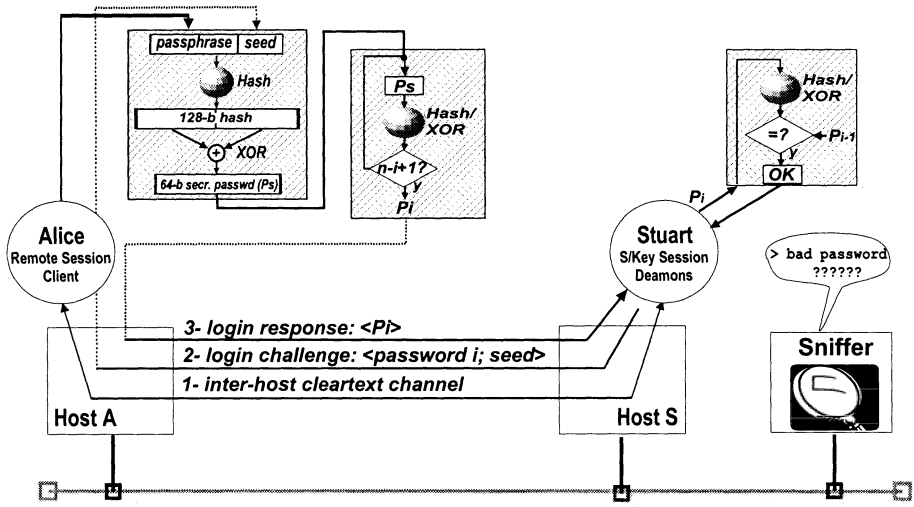

Figure 19.2. S/Key One-time Password System

P. A passphrase is an arbitrary length legible string. See Section 18.3.2 to recall why passphrases are good. P is concatenated to *seed*, and passes through a secure hash function. Several hash functions are currently supported, such as MD4, MD5 and SHA, so let us run our example with a generic H. The 128-bit output is halved, and both halves XOR'ed, yielding an 64-bit *secret password* p_s. Now, suppose we want to "buy" say a batch of $n = 16$ passwords. To generate the first 64-bit one-time password, $i = 1$, p_s goes through the hash/XOR function recursively n times. To get the second, it will go $n - 1$ times, until $i = n$, when it goes just once. The general expression to get password $p_i, 1 \leq i \leq n$, is: $p_i = H^{n-i+1}(p_s)$

The Real Thing S/Key operation is depicted in Figure 19.2. Alice gives her username. The server Stuart replies with the expected password *sequence number*, i, and the *seed*. The seed can be different from system to system, and thus allows Alice to use the same passphrase in all of them. Besides, it allows her to recycle the passphrase when passwords are exhausted. Now Alice has to have a local program installed on her computer to calculate the following: the program asks Alice to type in her passphrase P, the seed and the sequence number sent by the server; the output is p_i (*see* the expression in the last section), a highly random password; alternatively, she can request the system administrator to generate and print a list of passwords for her to take.

Alice sends the password down the line, in cleartext. How is it authenticated? The server database stores the last used password, p_{i-1}, so that it easily checks if p_i is a good password by hashing it once and confirming that $H(p_i) = p_{i-1}$. If the sniffer copies p_i, when he tries to use it, it will no longer work. Besides, since the algorithm works backwards, he cannot derive p_{i+1} from p_i either.

19.1.3 Secure Shell (SSH)

SSH is a secure session protocol suite that plugs classical holes in Internet/UNIX based protocols. It supports several encryption and authentication mechanisms. Besides protecting the login and authentication process, it also compresses and encrypts the session (DES,3DES,IDEA). MD5 is used for hashing. Authentication is modular: there exists the notion of server (host), service (application) and client authentication. SSH supports three authentication styles: traditional address-based (.rhosts, /etc/hosts.equiv) or UNIX password protected by the secure channel; traditional enhanced with RSA; pure RSA. Key distribution is also versatile: it can be manual, automatic or administrator based. There is a user authentication agent, in charge of keeping the RSA keys, in case RSA authentication is used. Several typical services are protected by this package:

- secure remote session (e.g., rlogin or telnet)
- secure remote execution (e.g., rsh)
- secure remote copy (e.g., rcp)

		Action	Description
1	C $\rightarrow$ S	$\langle C, service \rangle$	Client C requests service connection to SSH server S
2	S $\leftrightarrow$ C	$\langle versId \rangle$	C and S exchange version info
3	S $\rightarrow$ C	$\langle K_h, K_a, ciphTyp, X_s \rangle$	S sends RSA keys of host server, K_h (TYP 1024-bit), and application service, K_a (TYP 768-bit), cipher suites, and a challenge (64-bit random), all in cleartext
4	C,S	$SID = H(K_h + K_a + X_s)$	S and C compute a 128-bit session Id SID (+ means concatenate)
5	C $\rightarrow$ S	$\langle ciphTyp, X_s, E_h(E_a(K_{cs})) \rangle$	C generates a random 256-bit session key K_{cs}, and sends it to S, along with the chosen cipher, and the server challenge. The session key is XORed with SID, encrypted with K_a and then with K_h
6	S $\rightarrow$ C	$\langle E_{cs}(cfm) \rangle$	S extracts key K_{cs} and sends a confirmation encrypted with it
-	S,C	-	*// low-level secure channel established*
7	C $\leftrightarrow$ S	-	C now authenticates to the service in one of the methods available
8	C $\leftrightarrow$ S	$\langle E_{cs}(msg) \rangle$	Session proceeds, encrypted with K_{cs}

Figure 19.3. SSH Secure Shell

The basic secure session operation can be understood by looking at Figure 18.22 back in Section 18.10. The session establishment protocol, depicted

in Figure 19.3, underlies all the operation of SSH. Note that the first phase (1-6) is concerned with set-up of the low-level channel. Then, it is necessary to authenticate the remote session that will work on top of the channel (*see* the principles of *Secure Remote Session* in Section 18.10). This is done using one of the methods available: address-based with or without RSA, password, or RSA-only. Password authentication, the most used, is robust because it trusts nothing but the holder of the password, since it does not dialog with a login program, but with the SSH daemon.

SSH offers two useful additional functions: bi-directional TCP/IP port forwarding over the secure channel, implementing tunneling; X11 connection tunneling, to secure remote X terminal sessions, that usually go in the clear and are thus a security headache. The principle of tunneling is discussed in Section 19.3 (*see* Figure 19.7).

19.1.4 Pretty Good Privacy (PGP)

PGP is a freeware messaging and file encryption software, based on hybrid cryptography. Key management is based on public key cryptography (RSA), and payload encryption resorts to IDEA. It achieves the properties of secure envelopes.

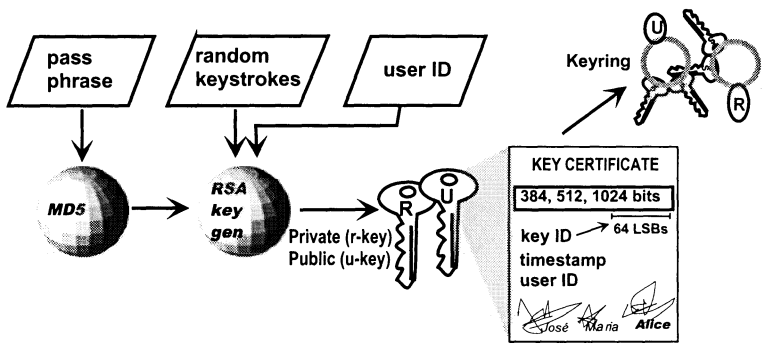

Figure 19.4. PGP - Key Generation

In PGP, all starts with key generation, shown in Figure 19.4. The user is prompted to supply some random data for the process (e.g., key strokes), and a passphrase. The passphrase is hashed and together with the random information and the user Id, they form the raw material to generate the RSA keys. Each key is then put in a certificate together with timestamp of generation and owner Id. Key certificates are kept in *keyrings* (public and private key rings). The public key certificate is then inserted on the *pubring*, whereas the private is inserted in the *secring*. Private keys are protected with the passphrase. Authentication is mutual, based on a ad-hoc chain of trust, instead of using a PKI: principals sign key certificates of other principals and so forth, creating a mutual chain of trust among clusters of people that are related.

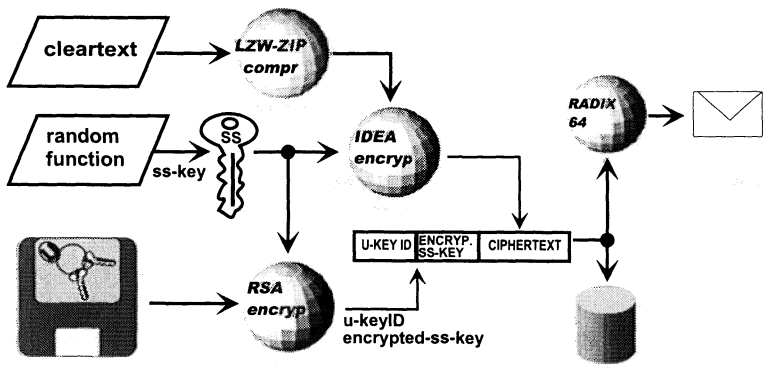

Figure 19.5. PGP Encryption

PGP can encrypt, sign, sign and encrypt. These operations follow the hybrid cryptographic envelope principle (*see* Figure 18.6) and are thus perfect for secure e-mail. Besides, PGP can encrypt local files with plain symmetric (IDEA) encryption, using a passphrase-derived key. The envelope encryption operation is shown in Figure 19.5. The cleartext is compressed first with the ZIP compression algorithm. As we pointed out earlier, this is a good idea, remember why? A symmetric encryption key (K_{ss}) is generated out of a random function. The cleartext is encrypted with IDEA using K_{ss}, and K_{ss} itself is RSA encrypted using public key K_u of the recipient as a key-encryption-key. Both the encrypted encryption key and the recipient Id go along with the ciphertext. Since PGP allows RSA key lengths in excess of 1024 bits, and IDEA itself does pretty well with 128-bit keys, this is bound to be very robust. Note that the result of encryption is a binary file. If the file goes to disk, this is OK. However, if it is an e-mail message, then RADIX-64 encoding converts it to an ASCII stream. Decryption is performed by reversing these operations at the other end: the recipient PGP extracts the key-encryption-key— the private recipient key K_r— from *secring*, decrypting K_{ss}, and then decrypting the payload with the latter.

Finally, signing is depicted in Figure 19.6. Signing follows the principle of digital signature with digests that we studied (*see* Figure 17.8). PGP makes sure that the cleartext has adequate format or control information to ensure it is verifiable at the other end. The text is hashed by MD5, and the result, concatenated with a random quantity to avoid replay attacks, is signed with the user's private key. A signature certificate is produced, by appending the key Id and the timestamp of generation. The cleartext, the certificate and the public key Id of the signer form the message, that is RADIX-64 coded if it should go by e-mail. The recipient PGP extracts the relevant public key from *pubring* and verifies the signature. Signed/encrypted messages combine both procedures.

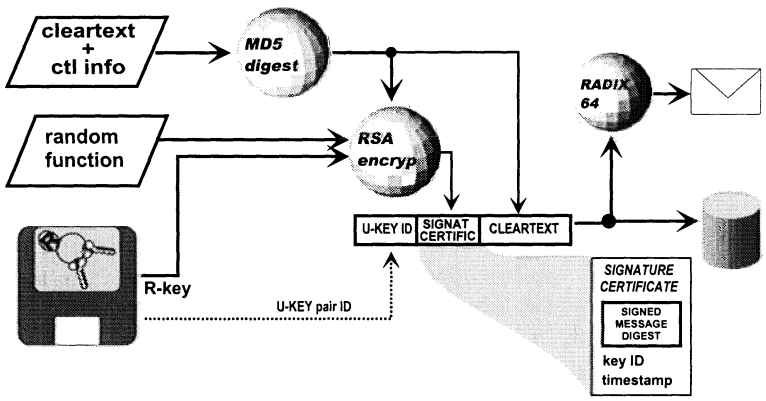

Figure 19.6. PGP Signatures

19.2 INTRANETS AND FIREWALL SYSTEMS

Intranets are the nickname for protected environments, normally organization networks closed to the outside or connected via protection devices, despite using Internet protocols. Since it hardly makes sense for an intranet to be physically disconnected, the most relevant devices for building intranet architectures are firewalls. There a number of commercial firewall systems, and perhaps the two best known, representing two competing classes, are the Checkpoint Firewall-1 and the Trusted Information Systems Gauntlet. Of the many free firewall packages around, a few are known to be effective and reasonably secure and bug free: the TIS firewall toolkit; the SOCKS proxy package (RFC1928-29,1961), and the LINUX packet filter. The TIS toolkit is a proxy package, supporting the most usual UNIX Internet daemons, such as telnet, rlogin, FTP, HTTP and mail. It has its own authentication server, supporting regular and one-time passwords. It also supports logging, and has a well-structured configuration and management interface, following a prudent policy. SOCKS is also a proxy package, but unlike TIS it bundles all servers in a single daemon, making it harder to fine-tune policies. It mainly supports telnet and ftp, plus a few ancillary services. Its security policy is less conservative than TIS. It supports authentication as well. The LINUX firewall package, IPchains, is a packet filter system that comes bundled with the distribution. Next, we study some of these systems with more detail.

19.2.1 Firewall-1

The Firewall-1 is filter oriented. However, it uses a form of stateful packet filtering (SPF), called Stateful Multi-Layer Inspection, which tries to understand the following high-level protocols: Telnet, FTP, SMTP, rlogin and rsh, NIS, NFS, HTTP, Gopher, Archie, WAIS, ICMP, RIP, SNMP. FW-1 supports several O.S.s, but it only supports two families of routers. FW-1 provides

for NAT (Network Address Translation) and for secure cryptographic channels between modules, using symmetric and asymmetric encryption: DH key exchange; RSA for public key certificates; and DES or FWZ1 session keys. It is divided into two modules that may or may not be co-located: the control and the filtering modules. The control module hosts the GUI interface and the Management module. Configuration is done through a very powerful GUI. The filtering module hosts the inspection module and the daemons. Besides the firewall daemon (fwd), it provides authentication daemons for use with several known services: atelnetd, aftpd, ahttpd, aclientd. Users can be individually authenticated for the above-mentioned services, and the authentication methods supported are: UNIX regular and one-time passwords, MD5 MACs, Kerberos, Smart Card support, SSL and SHTTP. FW-1 supports event and audit trail, firewall status monitoring and alarm generation (Status Monitor and Log Viewer). Access control rules are defined through a special purpose script language, INSPECT (Rule Base Manager). Following the SPF philosophy, rules have higher-level semantics than normal packet filters. The Network Object Manager defines security labels for networks, servers, routers, etc. The User Manager defines access rights for users on objects. The Service Manager manages services. Extension of services is simply done by the addition of an additional set of expressions and macros. Performance is good, as usual with packet filter systems. Firewall-1's main assets are: excellent GUI interface; overall performance; enhanced application-aware packet filtering; modular structure supporting a number of firewall architectures.

The Inspection module lives between the network interface of the bastion and layer 3 (e.g., IP), and inspects every incoming or outgoing packet. The baseline policy is prudent (*see Packet Filter Systems*, Section 18.8). The dynamic filter only opens the ports involved in cleared transfers, and closes them when the transfer ends. Connectionless protocols are difficult to follow by PFSs. FW-1 simulates a connection for UDP and similar protocols, so that it can follow a flow and reject alien packets. Similarly, RPC does dynamic port allocation. FW-1 monitors the portmapper and checks further RPC traffic against its cache of mapped ports.

19.2.2 TIS Gauntlet

The Gauntlet firewall is proxy oriented. It supports proxies for the best known services, such as: Telnet, rlogin and rsh, FTP, SMTP, POP, HTTP, Gopher, X-Windows, lpr. It also supports some packet filtering activity. Authentication includes: UNIX passwords, MD5 MACs, Smart Card support, SSL and SHTTP. Configuration is menu-driven, and addresses: the firewall architecture (network interfaces, dual or single-homed, addresses and services); configuration of access rules and user authentication; system integrity check against write penetration attacks; log and event report manager. Extension of services is done by the addition of new proxies. Performance is fair, as with any proxy-based system. Gauntlet's main assets are: proxies are in essence *transparent*, not requiring any adaptation or change in users and client applications; it has a framework

for developing custom application gateways, called *plug gateway proxy*, with which designers can support specific services and non-standard applications.

When a connection request comes in, the firewall analyzes the configuration rules (*see Proxies*, Section 18.8) and determines whether or not it should proceed. If so, the proxy contacts the end service, and from then on, the steps of this connection are performed by the proxy between the client and the end server, the proxy acting as server to the former, and client, to the latter (*see* Figure 18.20b in Section 18.8). However, everything happens at the proxy level, with the obvious performance implications. Later, TIS introduced the concept of *adaptive proxy*. An adaptive proxy uses a dynamic packet filter system (DPFS) at the internetwork layer. When a connection comes in, the DPFS notifies the proxy, providing information about the former. The proxy analyzes the connection parameters against the access control rules, as usual. However, when a connection is allowed through, it further decides if it proceeds at application level, or instead, because the connection is considered to be very low risk, it is forwarded directly at the internetwork layer. In that case, the dynamic packet filter manager inserts one or more rules for this connection. Subsequent packets of the connection are then automatically forwarded without consulting the proxy. Once a connection terminates, the connection rule is removed and the proxy is notified.

19.3 EXTRANETS AND VIRTUAL PRIVATE NETWORKS

In the measure that system architects became aware that leased lines are not secure links per se, and that using the Internet infrastructure provides significant financial gains, extranets emerged. Extranet technology aims at ensuring secure communication through the Internet, from the outside to an intranet, or between intranets, in essentially three situations: between distant facilities of the same organization; between facilities of different organizations; or between the facility and remote users belonging to the organization. This kind of architecture is thus relevant for: geographically distributed enterprises, in extension of their intranet; for virtual enterprises or enterprise networks, gathering suppliers, producers and clients, such as manufacturing clusters or business-to-business electronic commerce; or for mobile organization workers. The main attribute of extranet architectures is that security of external communication should approximate that achieved inside the intranet. Thus, important building blocks for these architectures are secure communication protocols (*see* Section 18.10), such as secure packets, tunnels and sessions, secure Internet and wireless communication protocols, and secure Web protocols.

19.3.1 Virtual Private Networks

The main architectural device for building an extranet is the *Virtual Private Network (VPN)*, an example of which is shown in Figure 19.7. Networks A and C, and the tunnel interconnecting them, constitute a very simple VPN over Network B. Networks A and C are intranets of the same organization, and it

is desired that traffic goes from one to the other as if they were in the same facility. For example, all addresses in both networks might be of the same domain, but different subnets. Either intranet is isolated from direct access to/from the Internet by a *security gateway*, such that the tunnel is laid between the two security gateways. These can be implemented by firewalls. The source and destination addresses of the payload packet on the left of the figure are the actual source in network A (Alice) and the final destination in network C (Bob), whereas the source and destination addresses of the carrier packet are those of the security gateways in each extremity. Tunnels may be set up with the desired granularity. They may carry the whole data between two networks, or there may be separate encrypted tunnels for critical connections, even inside untrusted intranets. Whatever the selection condition, routing tables inside the source network must route packets scheduled to go through a tunnel, to the tunnel mouth, that is, the security gateway, and not through the usual outgoing router. The VPN concept also addresses host-to-gateway tunnels, to support extranet client-server access to the intranet by remote users (e.g. travelling employees). You can now generalize the examples given and imagine a real installation with say half a dozen intranets, each of them interconnected to every other by a tunnel, and several remote access tunnels or secure channels, either from remote client-only offices, or mobile salesmen or executives.

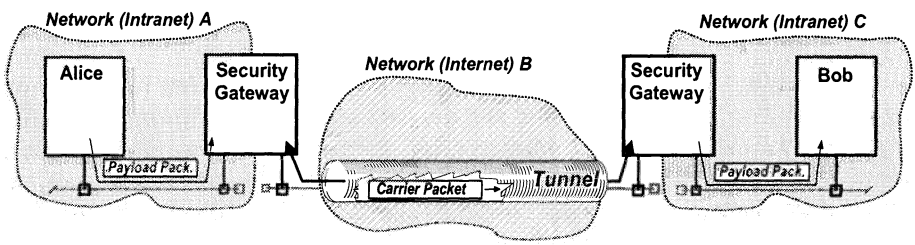

Figure 19.7. Virtual Private Network Architecture

The main technology behind extranets are tunnels. Most firewall manufacturers have extensions or separate packages implementing tunnels (e.g., Checkpoint, Gauntlet, Secure Data Fellows). Most of these implementations are not interoperable. In order to overcome this problem, the IETF is standardizing *IPsec* (Internet Protocol Security Architecture), a security architecture framework for IP.

19.3.2 Secure Internet Communication: IPsec

IPsec (Internet Protocol Security Architecture) is the current initiative of the IETF (RFC2401) to provide cryptographically-strong security for the IP protocol (Kent and Atkinson, 1998). It addresses: access control, connectionless integrity, data origin authentication, protection against replays, confidentiality, and limited traffic flow confidentiality. IPsec is a protocol-independent framework, that guarantees negotiable security properties to IP flows between two

nodes. Nodes are either hosts, or routers running IPsec, called *security gateways*. The properties are secured for armored data blocks, defined by a security header that encapsulates the attached data. The cryptographic operations on each module are specific to the several protocols that may be used. These security functions are implemented around two extension headers and respective processing protocols:

Authentication Header (AH) - provides connectionless integrity, data origin authentication, and protection against replays

Encapsulating Security Payload (ESP) header - provides confidentiality by encryption, and limited traffic flow confidentiality. It may also provide the functions of AH

IPsec headers can be combined with one another and with regular IP headers. The AH protects the integrity of a block of data, except for the fields that must be changed en-route. The AH includes security information for the receiver, the Security Parameter Index (SPI) field, and the authentication field, which has arbitrary length and depends on the algorithm being used. The ESP header includes again security information for the receiver (SPI), and the transformed data, according to the algorithm used. IPsec (either AH or ESP) can be used in two modes: transport-mode, which corresponds to the generic concept of secure channel depicted in Figure 17.18 back in Section 17.11; and tunnel-mode, which corresponds to the tunnel concept (a form of secure channel) illustrated in Figure 19.7 in this Section.

- **Transport-mode**– the protected data are upper layer service data units. This option encapsulates data from the layer above (TCP) with one of the IPsec headers, and then encapsulates it again in a normal IP datagram, achieving end-to-end security
- **Tunnel-mode**– protects full IP datagrams. This option builds a complete IPsec datagram, and then encapsulates it in a normal IP datagram. This IPsec-over-IP mode is useful for building tunnels, and for bypassing network areas that do not implement IPsec, achieving link security

Cryptographic checksums or signatures in AH, besides generally ensuring integrity, may provide reliable source address and sequencing information, to avoid spoofing and replay attacks. Data may be encrypted with ESP. Although IPsec is algorithm-independent, the default protocol is DES-CBC. Data flow confidentiality can be enforced to a certain extent with tunneling, since content of traffic, such as addresses, is hidden. Before payload transmission can start, IPsec must bootstrap through two crucial functions:

- **security association -** negotiation of protocols, ciphers and keys to be used
- **key distribution -** exchange of the keys needed for communication

The security association negotiation produces the above-mentioned SPI structure, which specifies things like: authentication and encryption algorithms; authentication and encryption keys; key and association lifetime. A security association is uniquely identified by a triple consisting of a Security Parameter Index (SPI), an IP Destination Address, and a security option (AH or ESP)

identifier. Key distribution may be manual or automated, in which case it uses a protocol. Several of the key exchange models that we studied in Section 18.6 are foreseen, both in the public-key and symmetric shared-secret areas. Security association and key management are at the time of this writing very active topics in the IETF, with the Internet Security Association and Key Management Protocol, ISAKMP/Oakley, being a strong candidate (RFC2408,RFC2412).

Headers may be combined to achieve further protection. For example, in transport mode, by applying ESP encryption for confidentiality of upper layer data, and then encapsulating again with AH for MAC-based integrity and authentication of the final IP packet. This is called *transport adjacency. Iterated tunneling* concerns building tunnels inside tunnels. Several combinations are possible, but perhaps an obvious and useful one is when specific tunnels, say ESP protected, are built from host to host in different intranets of an organization, to serve different applications, and then all these tunnels go, AH protected, through an outer, main tunnel, carrying all the traffic from one intranet to the other intranet across the Internet.

19.4 AUTHENTICATION AND AUTHORIZATION SERVICES

Authentication services exist for a number of applications and systems. Modem dial-up access is many often authenticated with front-ends such as Radius (RFC2138), from Livingston Enterprise, or the Cisco TACACS (RFC1492). RADIUS (Remote Authentication Dial In User Service) is a package for remote network access authentication in an open systems environment. RADIUS is independent from the communications protocol, and has two modules: the authentication server and the client protocols. RADIUS servers authenticate users against a UNIX password file, the Network Information Service (NIS), and an internal database. Password information is sent encrypted over the line, by a shared secret key. TACACS is similar to Radius, but some differences exist. TACACS uses TCP instead of UDP used by Radius. TCP is more resilient to errors, and provides immediate indication of communication or server failure. Radius sends a lot of relevant information in cleartext. Sensitive parameters such as username, authorized services, and accounting, can be captured by an intruder. TACACS encrypts all user information. TACACS has modular authentication, authorization and access control. TACACS can bind to Kerberos for authentication. Kerberos (Neuman and Ts'o, 1994), is the most widely used general purpose authentication and authorization server, included in services such as the Andrew File System and DCE. Several firewalls include hooks to Kerberos. Unlike Kerberos, which is KDC-based, the Distributed Authentication Security Service or DASS (Kaufman et al., 1995) is a distributed, CA-based authentication service developed at Digital and endorsed by the IETF (RFC1507).

19.4.1 Kerberos

Kerberos is conceptually divided in two modules, the Key Distribution Center (KDC) and the Ticket Granting Service (TGS). However, they reside in the same host and share the same database. The KDC handles the primary login of a principal. The TGS is invoked each time a principal needs a credential, or ticket, to access a regular system service. The TGS also checks the privileges of the principal to access the request service. The unit of modularity of Kerberos is a *realm*, the set of resources under the control of a KDC.

From now on, it is important that you have in mind the Kerberos authentication protocol, presented in Figure 18.10 back in Section 18.5. Our description will be based on Kerberos version 5. Each principal shares a *master key* with Kerberos, which it stores in a database. User's keys are generated from the user password through a cryptographic hash. Currently, Kerberos only supports DES. The database is encrypted with Kerberos own master key, K_{kdc}. Kerberos requires clocks to be synchronized, since it uses timestamps as nonces in defense against replay attacks. The allowed de-synchronization is five minutes. The credentials produced by Kerberos, called *tickets*, have a specifiable validity, limited to 21 hours. In what follows, we denote K_a as A's master key, K_{ab} as a session key shared between A and B, and $tick(A,B) = E_B(A, K_{ab}, T_t, T_l)$, as the ticket given to A in order to access B. The ticket contains A's Id, the shared key, the timestamp of creation T_t, and its time-to-live T_l.

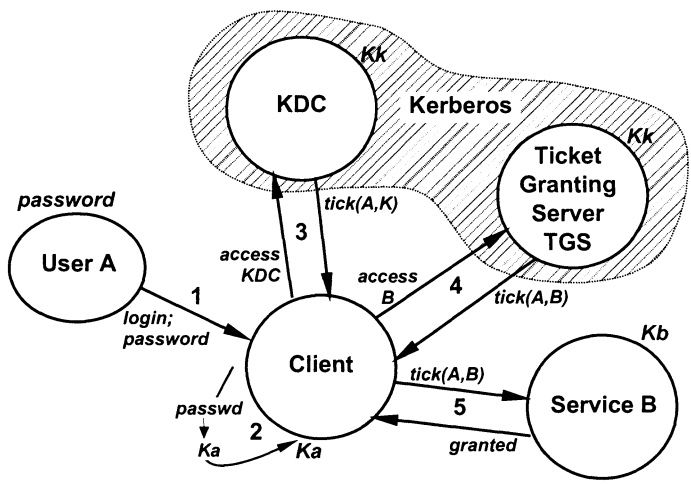

Figure 19.8. The Kerberos Security Service

The steps for a principal A to get access to a service B are depicted in Figure 19.8. The first step (1) is primary login. Principal A types in her $\langle login; password \rangle$ pair at the client host. The host hashes the password into A's master key K_a (2), and sends a login request to the KDC (3). Along with it, the client makes a proof of knowledge of K_a, by encrypting the current time with it. The KDC checks the time to see it is current (less than 5 minutes skew), which also proves that A knows K_a, and concludes the login process

by handing A a *conversation* key K_{ak}, and a bf Ticket-Granting Ticket, TGT, all encrypted with K_a. The TGT is a credential to be used in any subsequent addresses to Kerberos *in this login session*, and the key protects these interactions when necessary. Client A is now ready to access actual services. When A wants to access service B, she requests so to the TGS (4), presenting her TGT. The protocol develops as described back in Figure 18.10, with A getting *tick(A,B)* and K_{ab}, and presenting *tick(A,B)* to B (5). The ticket also contains authorization data, produced by the TGS after checking A's privileges to access B. After authentication and authorization is cleared by B, A and B share session key K_{ab} and A can access B.

19.4.2 DASS

The Distributed Authentication Security Service (DASS) is a distributed, hybrid cryptography authentication service. Whereas Kerberos is based on the KDC model and uses symmetric cryptography, DASS is a good sample of a CA-based system. It relies on long-term asymmetric (RSA) keys served by Public Key Infrastructures (hierarchies of Certification Authorities), primary login asymmetric (RSA) keys, and symmetric (DES) session keys. Alice has a long-term asymmetric key pair $\langle Ku_a, Kr_a \rangle$, and so does Bob. Furthermore, client Alice has a login password P, and she can generate a password-derived encryption key K_P. As well as for any other user, certification authority CA with public key Ku_{CA}, stores a record for Alice comprising $\langle A, Ku_a, E_P(Kr_a), H(P) \rangle$, that is, her public key, her private key encrypted with the password-derived key, and a hash of the password itself. Any public key certificate can be obtained from a CA (*see* Figure 18.11 in Section 18.6).

In what follows, we show how Alice initiates a session with Bob, in order to illustrate the functionality of DASS. The protocol is presented in Figure 19.9, and is self-explanatory. The process starts with a *login* phase, during which Alice *pre-authenticates* to the CA, by proving that she knows the password relevant to her record in the CA. Next, Alice performs the *authentication* phase with Bob, at the end of which they both have a session key. Note that after login, authentication of specific accesses is performed in just one message (unilateral) or two messages (mutual). Other relevant hybrid distributed authentication mechanisms are EKE (Encrypted Key Exchange), described back in Figure 18.15, and the NetWare authentication service.

19.5 SECURE ELECTRONIC COMMERCE AND PAYMENT SYSTEMS

Electronic Commerce *(e-comm)* comprises business made through informatic means. The payment is electronic, which prefigures an *electronic transaction* (*see* Section 18.11). The procurement may also be electronic, i.e. made through the Web. The goods themselves may also be electronic (on-line books, MPEG-3 contents, software packages). An e-comm purchase has five phases: *procurement; negotiation and order; payment system selection; payment authorization and capture; delivery of goods*. Generically, a customer navigates through a

		Action	Description
-			*// pre-authentication*
1	A → CA	$\langle E_{CA}(K_{ss}, h_P, T_A)\rangle$	Alice invents symmetric key K_{ss}, sends it together with password hash and local timestamp, encrypted with the CA key
2	CA → A	$\langle E_{ss}(E_P(Kr_a))\rangle$	CA verifies that Alice knows the password hash, and checks the clock skew. If OK, CA sends Alice her doubly encrypted long-term private key
3	A	$LC_{al} = S_a(Ku_{al}, T_x)$	Alice decrypts with K_{ss} and the password-derived key, recovering Kr_a. Then, she generates a login key pair $\langle Ku_{al}, Kr_{al}\rangle$, and produces a public *login key certificate* LC_{al}, signed with Kr_a, containing the login key and the expiry time
-			*// authentication*
5	A	$SC_{ab} = S_{al}(E_b(K_{ab}))$	Alice creates a session key K_{ab}, and wraps it a *session key credential*, SC_{ab}, encrypted for Bob and signed with the login key
6	A → B	$\langle LC_{al}, SC_{ab}, X_a\rangle$	Alice logs into Bob by sending the login key certificate, the session key credential, and an authenticator based on local time and the session key
7	B	–	Bob checks the skew of the timestamp and Alice's signature on LC_{al}, and extracts the login key, using it to verify SC_{ab} and extract K_{ab}
-			*// Alice is authenticated to Bob*
8	B → A	$\langle X_b\rangle$	Bob sends back an authenticator based on local time and the session key
-			*// Alice and Bob are mutually authenticated*

Figure 19.9. DASS Authentication Protocol

portal or virtual shopping, browses a catalogue, this is the procurement phase. Next, she identifies the good and negotiates with the merchant, for price, conditions, and so forth, hopefully getting to order the goods, by means of a Web transaction, e-mail, phone, fax, mail, etc. This is the negotiation and order phase. The payment system is then selected, from a wealth of possibilities: electronic cash, electronic cheques, credit card, electronic bank transfer, even paper cheques by mail. The merchant acts in order to be sure that he will be paid by the client (this is the payment authorization and capture phase), and then delivers the goods, the final phase. *Electronic shopping* protocols are emerging, covering most of the phases enumerated. However, current protocols focus on the most delicate ones, payment system selection and payment

authorization and capture, which pre-figure the so-called electronic transaction. In this section we are concerned with the security of payment devices, such as smart cards and digital wallets, and payment systems. The two most relevant frameworks for supporting electronic commerce are the SSL-related infrastructure and the Secure Electronic Transactions (SET).

19.5.1 Smart Cards

Smart cards are important pieces of e-comm gear. They provide a small, portable and secure means to carry value, and perform operations, some cryptographically secure. Smart cards occur in several types, described in Table 19.1.

Smart card technology is still recent (mid 70's) and is bound to evolve. Besides more powerful processors and larger memory for enhanced functionality, co-processors can be inserted for enhanced security e.g., to implement a guardian scheme (*see* Figure 17.16 in Section 17.7). A "distributed systems" philosophy has progressively emerged for the smart card area. The components of such an architecture are the smart card, the card terminal and the remote server. Standards have emerged, ISO-7816 is the basic one, specifying the mechanical and electronic structure, the I/O communication protocol and command definition for the card-terminal interface. The ETSI GSM standard specifies command messages for mobile phone SIM cards (*see* Section 19.3). The *EMV'96* standard (EMV'96: ICC Specifications for Payment Systems) has more recently been introduced by Europay, MasterCard and Visa, and endorsed by the industry in general.

Companies like Bull, GemPlus, Hitachi, Schlumberger, Motorola, IBM, currently support the ICC (Integrated Circuit Card or smart card) specification, which intends to be a standard for interoperability and secure operation of smart cards in electronic transactions. The specification consists of several parts, and addresses from the mechanical and electronic specification of cards and terminals and the minimum requirements of card and terminal functionalities, to business and applicational issues related with debit and credit on card. The *Java Card* standard, first appeared in 1996, is based on providing cards with a Java Card Runtime Environment, supporting a Java Card API on card, compatible with a subset of the Java language, in order to load and execute *Java Card Applets.* This allows cards to be programmed and loaded with programs, as normal distributed systems hosts. The Java Card API is bound to give a push to smart card based systems, namely in electronic commerce applications, because it is an *open system* specification: companies can develop their own products on the Java card.

19.5.2 Payment Systems

Electronic payment systems take several forms, as a matter of fact essentially emulating real payment systems:

- electronic cash
- electronic cheque

Table 19.1. Smart Card Types

MEMORY CARDS	Memory cards just have a block of non-volatile memory positions. They are very limited in functionality, and also in security, but are useful for small payments, such as phone cards. In this case, each position corresponds to a payment unit. Positions are cleared in the measure that they are "spent". Once cleared, the memory cannot be rewritten, so the card is discarded when finished.
LOGIC CARDS	Logic cards have limited processing capability, implemented by hardwired state machines, but represent an evolution with regard to memory cards.
PROCESSOR CARDS	Processor cards have increased processing capability, since they use microprocessors. They are capable of executing programs and protocols with outside devices. Some of these operations may be cryptographic. Processor (and logic) cards can either be *contact* or *contactless*:
Contact:	More usual, they have a range of contacts on the surface that connect to corresponding contacts on the card reader where they are inserted (instead of being swiped, as magnetic cards), to receive power and to dialogue.
Contactless:	They communicate with external devices by some wireless means. The s implest use short range electromagnetic fields, but the most elaborate resort to radio transponding, which is also used to energize the card. These cards have been used with success in highway toll systems in Europe.

- credit card
- electronic bank transfer

The last has been around for quite some time and is implemented through proprietary protocols in banking networks (e.g. SWIFT). The others are with the reach of the common user.

Ecash was developed by DigiCash, a company founded by David Chaum, the inventor of some of the protocols that we studied in Section 17.7. Ecash is a form of **digital cash** that allows fully anonymous spending. Clients and merchants must have accounts on an Ecash bank. The Ecash wallet, the cyberwallet, is loaded at the bank. Ecash enforces spontaneous on-line transactions: at the time of purchase, the merchant must be on-line with the bank, to ensure that the coins used for payment have not already been spent. Ecash has multi-party security. Ecash is easily integrated with the Web: the client runs the cyberwallet and the browser; the merchant runs a server and a CGI to run Ecash software. When payment of a purchase needs to be done, the merchant's Ecash server contacts the Ecash client, and then payment is performed.

Mondex was developed in the UK and has been in operation since mid'90. It is a prepayment smart-card based electronic cash system. The scheme uses a proprietary chip design from Hitachi, that creates end-to-end secure channels between chips. All devices with which a Mondex smart card should communicate must have such a chip. An optional PIN enhances personal security of the wallet, but if lost or damaged, the money inside cannot be recovered. The system supports multiple currencies and wallet-to-wallet transfer. There are no specific mechanisms for non-traceability, which is said to be ensured by a physical mechanism: the wallets, only way to reference a card holder, are distributed anonymously. Since the channel is end-to-end, Mondex cards can even be loaded by phone. More typically, they are loaded at a Mondex-enabled ATM, that talks to the bank, which has a sort of virtual vault, the Mondex Value Box, with a battery of Mondex chips to dialogue with remote client cards. Similarly, the merchant has a Value Transfer Terminal to receive payments.

CAFE Conditional Access for Europe, was a European project ended in 1996 that developed a secure electronic payment system using the blind signature principle. Unlike Ecash, CAFE used smartcards with guardians, which allowed completely non-traceable, fraud-free and off-line operation. CAFE has a few interesting characteristics. It is entirely public-key based, and achieves multi-party security. It can pay with cash but also sign cheques. It supports multiple currencies and multiple issuers of electronic money. CAFE uses high-quality infrared communication wallets, and provides recovery of lost, stolen, and damaged cards. An optional PIN enhances personal security of the wallet.

Millicent is a *micropayment* system developed at Digital, that allows payments down to USD0.001 to be made. It is still early to see whether this kind of systems will go anywhere, but perhaps they will find a use in Web navigation charging. Millicent uses a currency called *Scrip*, and believes in the principle that the cost of doing a fraud should be more than the value of the transaction. Users of the system aggregate payments until they have enough money to make a macropayment.

The two most relevant frameworks for supporting electronic commerce in general and *credit-card* based in particular, are the SSL-related infrastructure and the Secure Electronic Transactions (SET). We have already talked quite a lot about SSL. Electronic commerce around SSL involves a basic framework consisting of: SSL as a secure communication channel; a Public Key Infrastructure (PKI) in place which is recognized by the players; and SSL server and client authentication mechanisms. The players (client, merchant, acquirer, issuer) contact securely with each other using SSL-enabled servers and browsers. To perform mutual authentication of the principals and to authenticate the several instruments of the electronic transaction (e.g., a credit card credential), they resort to PKI certificate chains. To prevent fraud, they check certificate revocation lists routinely. In the next section, we will focus on an alternative framework: *SET*. The Secure Electronic Transactions protocol resulted from the convergence of early works by Visa and MasterCard on protocols for the secure presentation of credit cards, and is endorsed by several major players,

including American Express, IBM, Microsoft, and Netscape. SET has evolved in the recent years as a fully-fledged electronic transaction framework and architecture, together with the relevant protocols. Expectations about its success must obviously be confronted with those about the SSL-based infrastructure.

19.5.3 Secure Electronic Transactions (SET)

SET is oriented to credit card payment, and the overall model obeys that of the on-line spontaneous transaction, depicted in Figure 18.25b, back in Section 18.11. In terms of the figure, the SET protocol is concerned with steps 1,2,3 and 4, between the client (card holder), the merchant and the acquirer, who implements the *payment gateway* between the card-holder issuer and the merchant. The interaction between the acquirer and the issuer is currently secured via a proprietary *banking network*. The SET trust model relies on the existence of a trusted third party, a PKI, that builds trust among the players, by certifying all the signatures involved in the transaction. SET uses X.509 certificates, produced by a chain whose main elements are a supra-national root CA, which in turn will certify each brand CA (e.g., Visa), and on a third level: card holder CAs; acquirers serving as payment and merchant CAs, and running payment gateways. Unlike all the others', the card holder certificate has its credit card number (primary account number, PAN) blinded by concatenation with a nonce and a fixed sequence, and then hashed. SET uses *selective end-to-end encryption*, such that content may be selectively revealed to parties. SET messages are fairly dense, so we first present the overall picture of a SET payment transaction, in Figure 19.10, comprised of request/response pairs: the buyer initializes the protocol (PInit); and then emits the purchase order (P); the merchant requests authorization (Auth) for the payment; once cleared, the payment is captured (Cap); the card holder may do an optional inquiry (Inq) at any time during the transaction, to find out about its state.

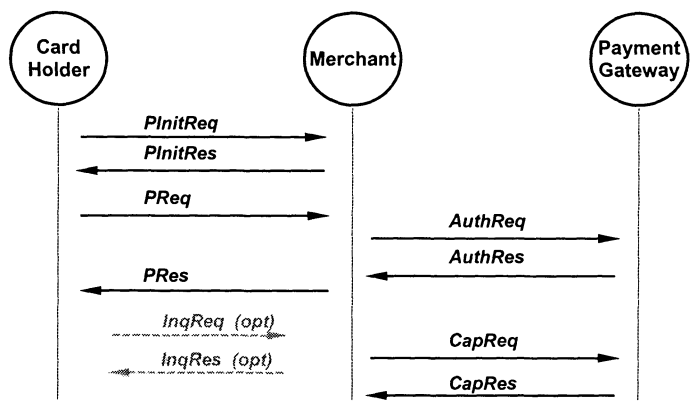

Figure 19.10. SET Payment Transaction

We now analyze the message structure, omitting unnecessary detail. The **PInitReq** message contains: the brand of card, an optional thumbprint or fingerprint (hash) of the certificates held by the card holder, a local transaction Id, and a nonce challenge. In response, the merchant generates a global transaction Id, timestamps it, includes the card holder's challenge and its own, signs everything with the merchant's key, and sends it to the card holder, together with any certificates that the latter might not have yet cached from former transactions, such as the merchant's and the acquirer's.

The card holder believes the merchant is OK when she receives a correct and fresh response (her challenge comes back), and in consequence she issues the Purchase Order (**PReq**). The purchase order comprises two parts: the Order Information (OI) and the Payment Instructions (PI). The OI contains the order description data (OIData) for the merchant, essentially data from the Init phase: global transaction Id, brand Id, the two challenges (to show freshness), and a nonce, to prevent dictionary attacks on the alphabetic contents of the OI, once hashed. The OI is validated with a *dual signature* field, which carries the hashes of both the OIData and PIData. This kind of signature relates the OI with the PI, and it has the property of being verifiable by only revealing one of OIData or PIData (to either the merchant or the acquirer). The card holder certificate goes along. The PI is a credential encrypted with the acquirer's key, containing payment data (PIData) for the acquirer. PI is forwarded by the merchant, who cannot read it. The PI contains the PIData: global transaction Id, amount, actual credit card data (CardData) encrypted with *extra-strong* plain 1024-bit RSA, and a hash of the order description OD. This is validated with the same kind of dual signature as the OI, and the whole (PIData plus signature) is finally encrypted. Sending PI is equivalent to signing the credit card ticket in a conventional transaction.

The merchant now verifies the signature on OI by following the certificate chain in the PKI. The merchant will request authorization and initiate capture. The **PRes** response may be issued at any time after this check. The authorization request (**AuthReq**) carries elements of the transaction (namely a hash of OD and of OIData) and the PI credential, all signed with the merchant's private key and encrypted with the acquirer's public key. If the transaction elements and those inside PI match, the acquirer knows that both the card holder and the merchant agree on the transaction (goods and amount): the dual signature in PI proves the connection of that order to the card holder. The acquirer obtains authorization from the banking network and if all is OK, sends the relevant code in an **AuthRes** message, together with a **Capture Token**, a credential for the merchant to get paid. The merchant will capture payment, eventually merging Capture Tokens of several transactions.

Remember that SET is the payment part of the whole purchase. SET can and should be integrated in broader Web-based electronic commerce applications. The SET consortium provides a SET reference API and implementation to guide implementors. It is free for non-commercial use.

19.6 MANAGING SECURITY ON THE INTERNET

Internet protocols have their *design vulnerabilities*. They improve with time, but will hardly disappear. On the other hand, configuration of a large facility is hard to do without any mistake, leaving *configuration vulnerabilities*. Attacks and intrusions may go un-noticed, if there are many new events arriving. This introduces *detection latency*, that may amplify the effects of an intrusion. These reasons are more than enough to justify an investment in *security management* of any facility. System Management strategies and tactics are discussed with more detail in the Management Part (Part V) of this book, namely how to insert these technologies in a coherent management framework. Amongst the relevant functions, we are concerned with: security enhancement tools; fault diagnosis tools; intrusion detection tools; auditing tools. These functions, as well as tools to perform them, will be detailed in Section 24.7 of that part.

19.7 SUMMARY AND FURTHER READING

This chapter gave examples of systems and platforms for secure computing. The objective of the chapter was to provide the reader with some knowledge about existing products and systems, but above all to relate these systems with the notions learned throughout this Part. We reviewed remote operations and messaging packages, firewall and virtual private network systems, authentication and authorization services, devices and frameworks for secure electronic commerce.

Further reading on Java and Web Security can be found in (McGraw and Felten, 1997; Garfinkel and Spafford, 1997). A thorough discussion on electronic payment systems is done in (Mahony et al., 1997). Cheswick gives a detailed account of firewall and Internet-related security tools (Cheswick and Bellovin, 1997). In (Quinn, 1996), an up-to-date survey is given of tools for UNIX host and networking security. Table 19.2 gives a few pointers to information about some of the systems described in this chapter. Some of the sites are extremely complete repositories of security-related software.

Table 19.2. Pointers to Information about Secure Systems and Platforms

Class of System	*System*	*Pointers*
CERT		www.cert.org
ITU	(ex-CCITT)	www.itu.int
IETF RFCs		www.rfc-editor.org
	SSL	home.netscape.com/eng/ssl3
Remote	**SSLeay**	ftp.psy.uq.oz.au/pub/Crypto/SSL
Operations		www.ssleay.org
and		www.openssl.org
Messaging	**SHTTP**	www.homeport.org/~adam/shttp.html
	TLS	www.ietf.org/html.charters/tls-charter.html
	SSH	www.ssh.org
		www.uni-karlsruhe.de/~ig25/ssh-faq
	PGP	www.pgpi.com/
	S/Key	ftp.bellcore.com/pub/nmh/skey
		ftp.cerias.purdue.edu/pub/tools/unix/netutils/skey
		www.ietf.org/html.charters/otp-charter.html
	OPIE	ftp.inner.net/pub/opie/opie-2.32.tar.gz
	RSADSI	www.rsa.com
	SECUDE	www.darmstadt.gmd.de/secude
	PEM	ripem.msu.edu
	DCE-RPC	www.opengroup.org/dce
	SUN-ONC	www.sun.com
	FW-1	www.checkpoint.com/products/firewall-1/index.html
Firewall	**Gauntlet**	www.nai.com
Systems	**TIS toolkit**	ftp.tis.com/pub/firewalls/toolkit
	SOCKS	ftp.cerias.purdue.edu/pub/tools/unix/firewalls/socks
	SSLproxy	www.obdev.at/Products/sslproxy.html
	squid	squid.nlanr.net/Squid
	VPN-1	www.checkpoint.com/products/vpn1/index.html
Virtual	**VPN+**	www.datafellows.com
Private	**IPsec**	www.ietf.org/html.charters/ipsec-charter.html
Networks		www.antd.nist.gov/antd/html/security.html
	ISAKMP	www.antd.nist.gov/antd/html/security.html
	Stunnel	www.stunnel.org
	Web sec.	www.cs.princeton.edu/sip
Authentication	**Kerberos**	athena-dist.mit.edu/pub/kerberos
and	**Radius**	www.livingston.com/tech/technotes/500
Authorization		ftp.cerias.purdue.edu/pub/tools/unix/netutils/radius
Services	**TACACS**	www.cisco.com/warp/public/707/index.shtml
	Java Card	www.gemplus.com
Secure	**EMV'96**	www.mastercard.com/emv
Electronic	**DigiCash**	www.digicash.com
Commerce	**Mondex**	www.mondex.com
and	**Millicent**	www.millicent.digital.com
Payment	**SIBS**	www.sibs.pt/en/multibanco.html
Systems	**SET**	www.setco.org/set_specifications.html

V Management

The direct forces fight the enemy on the ground, but the indirect forces ensure victory. Their combinations are infinite.

— Sun Tzu, The Art of War, circa 500 B.C.

Contents

Overview

Part V, Management, addresses the management of distributed systems, that is, the issue of ensuring that distributed systems are configured correctly in order to provide adequate service, and that they remain correctly configured and providing adequate service throughout their life. In the measure that distributed systems technologies achieve maturity and widespread use, management will become one of the most important disciplines in the area. This part introduces the Fundamental Concepts of Management in Chapter 21, and continues in Chapter 22 with the main Paradigms for Distributed Systems Management, such as: managers, managed objects and MIBs, domains, main management functions (e.g., configuration, fault, accounting), and monitoring. Chapter 23 addresses Models of Network and Distributed Systems Management, and Chapter 24 discusses Management Systems and Platforms. In these two chapters the notions of previous chapters are consolidated. Frameworks and strategies for management are discussed, and the relevant models presented: centralized, decentralized and integrated management, domains. Chapter 25 finalizes the case study, this time: managing VP'63.

20 CASE STUDY: MAKING VP'63 SECURE

This chapter brings our case study one step further: making the VP'63 (VintagePort'63) Large-Scale Information System secure. Increased distribution of the infrastructure through the Internet, combined with remote access of company salespersons dictated this step in the project, in order to address concerns with privacy and integrity of the company's information system. As selling on the Internet becomes attractive, plans are also made for setting-up an electronic commerce server, a major step for a company that did not even have a passive Web presence.

20.1 FIRST STEPS TOWARDS SECURITY

The reader should recall that this is the next step of a project implementing a strategic plan for the modernization of VP'63, started in Chapter 5, and continued in the Case-Study *chapters of each part of this book. The reader may wish to review the previous parts, in order to get in context with the project.*

The team identified the following problem areas with regard to security, deriving from the corporate strategic plan:

- point-to-multipoint payload interconnection flows between the enterprise units, now made through open networks (e.g. Internet);
- point-to-point remote session interconnections between employees and enterprise units, now made through open networks (e.g. POTS, GSM, Internet);

P. Verissimo et al. *Distributed Systems for System Architects*

- multipoint-to-point anonymous connections from anywhere on the Internet to the commercial Web site, not only to acquire information, but also to perform electronic transactions.

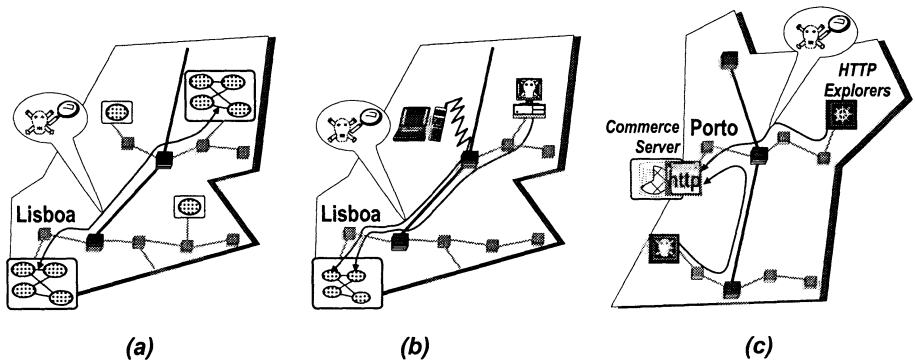

Figure 20.1. Security Problem Areas: (a) Site Interconnection; (b) External Remote Access; (c) Anonymous Transactions

The risk of operation was evaluated for these problem areas. One of the premises of the project is the use of COTS components, with their known vulnerabilities, which can reduced by configuration and/or function elimination. A preliminary abstract analysis of the degree of vulnerability suggested that this presents disadvantages (vulnerabilities do exist) and advantages (they are well-known and fixes exist), but yields a high cost-effectiveness ratio. On the other hand, a preliminary abstract analysis of the level of threat revealed the following:

- Site interconnection (Figure 20.1a)– the payload flow may be subjected to attacks on confidentiality and integrity.
- External remote access (Figure 20.1b)– individual access sessions may fall to intrusion campaigns that compromise the authenticity property, and from then on, the confidentiality and integrity of the internal state of the system.
- Anonymous transactions (Figure 20.1c)– attacks on the commerce server protocols with the attempt of fraud may assume several facets; general and perhaps distributed denial-of-service attacks are also to be feared.

The architectural approach for security will be laid out around: the extranet and the virtual private network of the company over the Internet; the intranet and its firewall gateways to the outside. The team decided that the set-up for business-to-business (B2B) transactions will be deferred to a later phase when the technologies to be installed now are mature. This is because B2B depends both on commerce server and on VPN technologies, which will be developed with separate purposes in this phase.

20.2 GLOBAL SECURITY: EXTRANET AND VPN

Recalling the infrastructure laid out in the first phase of the project, the infrastructure should now evolve to a strict WAN-of-LANs organization, where every facility has a single logical connection point to the Internet, the Facility Gateway. All internet addresses behind the Gateway are invalid, which brings a certain degree of protection to probing (e.g. port scanning) attacks, and on the other hand allows creating a seamless virtual domain that spans all facilities, so that all nodes anywhere in the company's installations are seen as being in a single network. For this to be possible, IP-over-IP tunnels are created between every Gateway and all the others.

Figure 20.2 depicts the big picture of the VP'63 Virtual Private Network (VPN) design, interconnecting all VP'63 facilities over the Internet following this model. In order for the payload traffic to be protected against attacks, the tunnels are secured using link encryption between Gateways.

This set-up can be generalized in several ways under an extranet perspective. To begin with, it solves the problem of remote fixed client-only offices, i.e., the small installations that once used to have a single remote terminal hooked by leased line or dial-up. These offices will establish secure payload tunnels to main facilities in the same way. The other problem are nomadic salesmen or executive notebooks, bound to access the VP'63 network through the Internet or modem dial-up. Functionally, they should desirably have the same kind of direct access *into* the VP'63 intranet as provided by the site interconnection tunnels. However, given the mobility and sporadic character of access, building trust on these connections is more difficult. In consequence, they had better be provided through an external remote access service that establishes a more powerful filtering point at the Gateway, to be addressed upon the detailed Intranet and Gateway design.

20.3 LOCAL SECURITY: INTRANET AND FACILITY GATEWAY

The architecture of the intranet of a facility is shown in Figure 20.3, focusing on the Facility Gateway architecture. Under a security viewpoint, the Gateway must protect the intranet and provide services hosted by internal servers in a secure way. The team has studied the design of the following services: portal passive services (rendering, etc.); portal active services (messaging, search, transactions, etc.); remote access (from the outside); internet navigation and messaging (from the inside).

The Gateway is normally laid out around a two-level or screened-subnet firewall architecture. The minimal functionality a Facility Gateway should provide is the insertion in the VPN infrastructure, and for simple installations (small client offices) that can be ensured by a single host acting as a bastion router. Figure 20.3 depicts the maximal Gateway architecture, in fact a set of hosts, providing all the services foreseen. The outer firewall is the router that provides access to the Internet and implements the secure IP-over-IP protocols (e.g. IPSec). It also implements the NAT (network address translation) that

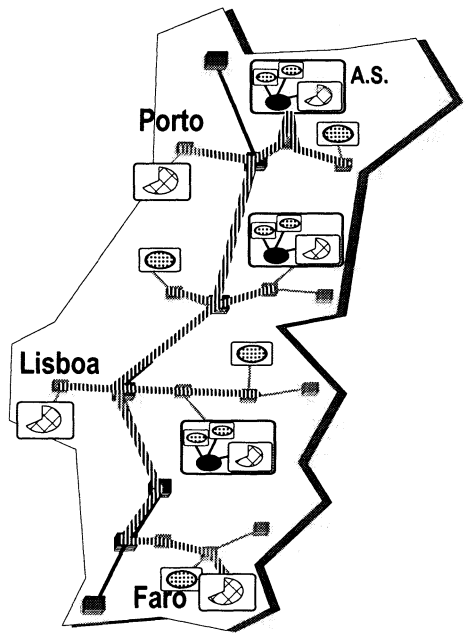

Figure 20.2. Extranet with view of the VPN

hides VP'63 internal addresses. As a firewall, it performs some form of packet filtering, necessarily limited since some of the services on the DMZ are for anonymous access. Still, on the intrusion detection side it is capable of some counter-reaction. The inner firewall is a bastion router, acting as a multi-port firewall to the intranet subnets. Between the outer and inner firewalls lies the DMZ (de-militarized zone), where extranet services are installed.

The passive services of the portal are ensured by a Web (HTTP) server with local storage of static pages for immediate rendering. The Web server is placed on the DMZ, since it serves anonymous accesses. It also acts as the overall portal for all other public access services from the outside. As such, it also provides hooks for active services of the portal: email to the enterprise, but more importantly, it connects to a lightweight transactional server on the intranet. The lightweight server is so called due to the underlying philosophy: hosting fragments or replicas of the global database so that they may have a better performance serving the basic on-line commerce (e-comm) applications, search queries and electronic transactions. Once the architecture in place, new contents and new commerce offers can be readily offered in a scalable way. The performance of this lightweight solution may be fairly high with an adequate configuration and a correct balance of the fragment semantics, between read-only, weak consistency (caching) and strong consistency (active replication). It also provides an indirect way of reconciling operations with the business information system, through the global database, without burdening

its own transactional front-end with e-commerce transactions, which have a highly unpredictable behavior and evolution.

The remote access service from the outside is given a low level of trust, as discussed before. As such, a dial-up RAS service is hosted in a DMZ node, with dial-up line/caller authentication. From then on, remote requests, both via dial-up and via the Internet through the outer firewall, are treated equally and directed to a proxy remote session (telnet) RAS server on the inner firewall. Plaintext telnet connections are not allowed under any circumstance, employees will be instructed to have a secure telnet package installed on their portable machines. Roaming access from alien machines will not be allowed. Requests for all the above-mentioned services are authenticated on a need basis on a strong authentication server, placed on the intranet but accessible from any firewall and DMZ services. This offers incremental levels of authentication, e.g., operation-dependent authentication for e-comm transactions. The authentication server hosts private (employee) and semi-private (regular client) credentials, but may also be hooked to existing PKI-CA systems (anonymous users).

The tunnels merely serve to reach another facility, and nowhere else on the Internet, so direct Internet connection from the inside has to be specifically addressed. Internet navigation and email sending are the only outgoing services to be supported at this stage, provided through central outgoing HTTP and email servers located on the main facility. This may have some impact on performance, namely on the Web side, but is otherwise transparent from applications and provides a necessary control point.

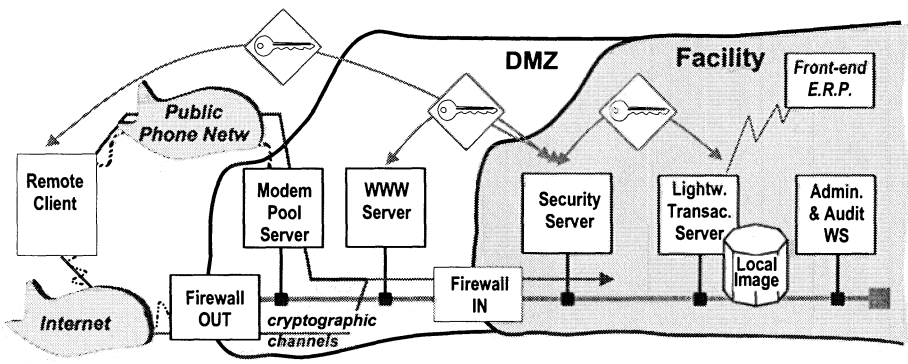

Figure 20.3. Intranet with view of the Portal

Further Issues

These issues need some refinement now, and the reader was assigned the study of a few questions that were still left to be solved:

Q.4. 1 Sketch the routing hops for an IP packet going between nodes in two facilities separated by a secure tunnel.

Q.4. 2 Propose the detail of the Gateway cryptographic set-up underlined in Figure 20.3 (key types, distribution, and integration in protocols).

Q.4. 3 Is there a secure way for employees to access email and news via alien Web browsers while roaming without access to a company machine?

Q.4. 4 How should outgoing direct Internet access be decentralized on a per-facility basis, if increased use starts creating a bottleneck on the central HTTP and email servers?

Q.4. 5 How can the Web server designed for the portal be improved w.r.t. fault tolerance and load balancing?

Q.4. 6 How can the transactional and search engine front-end designed for the portal be improved w.r.t. fault tolerance?

Q.4. 7 Discuss other alternatives for remote access and e-commerce based on different balances between threat and vulnerability than those assumed in the present design.

Q.4. 8 Denial-of-service attacks may indeed become a concern. Discuss the possible design of some form of availability measures facing such attacks.

Q.4. 9 The architecture proposed is essentially an attack prevention one. Discuss the possible design of some form of attack tolerance.

Q.4. 10 Delineate a strategy for networking and data security assuming potentially harmful insider users, which have been precluded from the current model.

21 FUNDAMENTAL CONCEPTS OF MANAGEMENT

This chapter discusses the problem of management. The fundamental concepts are presented, and the rationale for configuring and managing systems is discussed. The main architectures for systems management are introduced, in order to be further developed in the following chapters.

21.1 A DEFINITION OF MANAGEMENT

What is management? Systems management is the set of *planning, supervision* and *control* functions of a system, such that it provides the adequate service, as expected by its users. The service is defined upon the *configuration* of the system.

Systems management includes strategic as well as tactical factors. **Strategic** factors are concerned with establishing *management policies*, and planning the system architecture and functionality so that these policies are fulfilled. **Tactical** factors address the measures and mechanisms put in practice to actually fulfill the strategic objectives, and the timely reaction to the varying operating conditions such that the system maintains its functionality.

Why is management necessary? Whereas all previous parts of this book were concerned with conferring architectural, functional or non-functional properties to a system, management is concerned with the global measures that assist in maintaining the whole of these properties through the system's useful life. Systems evolve, and need *planning* and *configuration* in order to adapt to change.

P. Verissimo et al. *Distributed Systems for System Architects*

Systems today are interconnected, and in consequence isolated and uncoordinated efforts may have undesirable or even harmful effects. This requires *integrated* approaches, and adequate *tools*. Finally, systems became so complex that manual approaches are obviously insufficient, requiring as much *automation* of functions as possible.

As the quote with which we opened this part metaphorically implies, the combinations between strategy and tactics, organization and technology, planning and reacting, are infinite. But the successful combinations are only a few.

21.1.1 The Management Life Cycle

A management support system works pretty much as a process control system. The "controller" is the *manager* and the "process" is the *managed system*. The "control cycle" is depicted in Figure 21.1. The manager monitors the state of the system by receiving *events* from it and interpreting and processing them under the light of the management policy. The manager controls the system by issuing *control operations* on it. These operations are either dictated by strategic management or issued as a result of the processing of events from the system. That is, the manager either initiates some action, such as installing new routing tables or configuring a new printer, or reacts to an environment change, such as repairing a partitioned network, or performing load balancing on a pool of servers upon detection of overload.

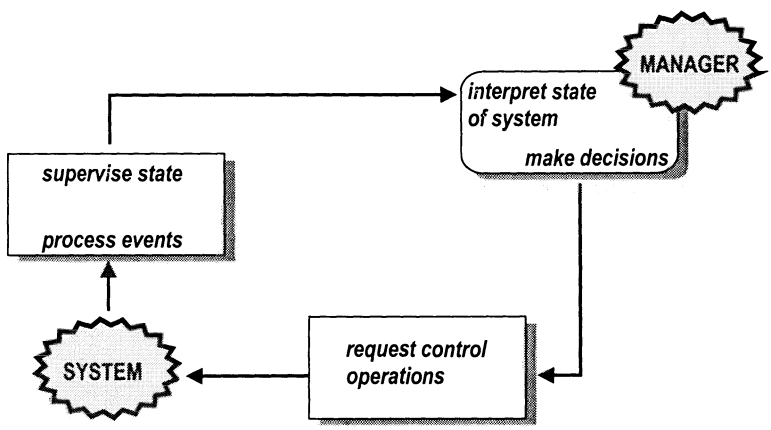

Figure 21.1. Management Life Cycle

A common representation of the flow of information concerned with managing a system (Sloman, 1994) is given in Figure 21.2. The flow starts at strategic level, with long term directives (strategic management policies) and immediate action directives (strategic management decisions) issued to the tactical managers, who *interpret* them. The mission of the managers is to implement strategic decisions in the best possible manner. For that, they issue *control*

commands to the system being managed, that act on the *resources*. Some of these commands may consist of polling or sampling the state of the resources. Resources respond to these solicited actions, and may also trigger unsolicited events, or *notifications*, back up to the managers, informing them of changes of state. Managers *monitor* the system through these solicited and unsolicited actions. Some of them require feedback in the form of new commands that close the control loop.

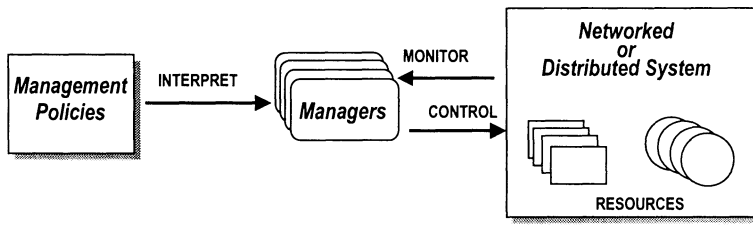

Figure 21.2. Management Information Flow

21.1.2 Organizational vs. Technical Management

In face of what was just said, we should understand that management exists at two levels of abstraction in an organization:

- **organization-level** – dictated by the strategic executives of the organization, not necessarily the field executives
- **technical-level** – performed by the technical executives, or *systems administrators*

In most organizations, the CTO (Chief Technology Officer) is the liaison between the two, since she discusses the strategic issues with her fellow executive managers, and coordinates the systems administration teams. The CTO should enforce the above-mentioned separation of duties. Failure to do so may lead to abnormal and undesired situations, whose extreme examples would be: letting the technical staff acquire knowledge and decision power that belong to the managerial area; putting the technical staff under the direct orders of executives who do not master the technologies.

A key factor of success in management is the adequacy of the system, and of its information and management models, to the models of human thinking and of the organization it serves. Inappropriate models may have been at the root of many a failure of the introduction of informatics[1] in businesses. Two orders of reasons arise when debating this problem. The first is concerned with the mapping between computer-level information and human-level perception.

[1] "Informatics" is a word of european origin getting increased acceptance in the community of computer users and developers. It is used to denote in general terms all that is related with use of computers and networks in information processing, access and manipulation.

Quoting Mintzberg: "Many management information systems (MIS) seem not to be for management at all. They are computer information systems and proceed on the assumption that managers care that the information has been processed by a machine" (Mintzberg, 1989). Computer-generated information is often too limited, too general, too late and too imprecise for human managers to handle adequately. The second is concerned with the mapping of roles in the organization onto the representations allowed by the actual computing model. Think of the following example: why should a senior system administrator—who is a technician and not an executive, let alone the CEO (Chief Executive Officer)— have read and write (or delete...) access to all the information of the company where she works? She normally has it indeed, because of the way most commercial systems work (she has `root access`), but is this a faithful metaphor of that company's business model? Most probably not, that is, she as a middle officer should have neither the power (e.g., to block or destroy the system), nor the knowledge (e.g., of the whole salary policy) indirectly given to her by the way the system is set up.

Can we do something about it? We are persuaded that the answers lie in *systems architecture*, and we hope this book may give a few contributions. From the earlier parts: the mapping of the functional characteristics of businesses onto the functional attributes of technologies; the provision of non-functional attributes to overcome the shortcomings of technologies (dependability, timeliness, security). From this part: the notion that the dichotomy between organizational and technical management levels must be cast into the system architecture; and that this can only be made through the adequate models and tools to handle the information flow between both levels.

21.1.3 Management Support Services and Functions

So what is management in practical terms? Imagine a large enterprise, with facilities in several locations, and thousands of interconnected machines. It is necessary to control how the system is laid-down and configured, draw a map of the existing links, keep directories of registered users and service names. It is also necessary to diagnose, circumvent or repair faults and recover from errors. These faults must include malicious faults that affect security. Performance and quality-of-service guarantees must be preserved for the various services. The utilization of resources by the several users must be accounted for, and so forth. The main management functions arise from these needs:

- configuration management
- fault management
- accounting management
- performance management
- quality-of-service (QoS) management
- security management
- name and directory management

These functions are supported by a few classes of services, such as:

- remote operation execution
- management information storage
- event reporting
- log control

Operations related with the functions above are triggered on remote devices. These operations read and modify the state of management information, whose storage is performed both at the devices and at the managing hosts. The paradigm that supports this concept of management data repository is called **Management Information Base (MIB)**. The MIB holds the data structures concerning the managed resources, and their format is standardized for most architectures. It is through the MIB that most management operations are performed: reading status of resources, writing state variables. The *structure of management information (SMI)* is the collection of specifications of these structures and variables, normally organized in standards.

Event reporting is concerned with the unsolicited notifications (also called up-calls) of resources to the management entities, by which they report unprogrammed events, such as: changes in the environment configuration (new hosts discovered), errors (printer out of paper), failures (a link that is down), and so forth. Events need some processing, sometimes at the source, in order that the destination is not showered with irrelevant or redundant event notifications. *Event discriminators* are special programs that filter and select events according to pre-defined rules (e.g., thresholds), and pre-process them in order that the information that arrives at the upper layers has more elaborate semantic contents than the raw event (counters, rates). For example, a number of error events may be produced as a consequence of a failing network link: several garbled frame transmissions; frames that are completely lost; excess conflict or collisions on the network access. Instead of producing n event reports, a discriminator may send a `failed()` report up as a consequence of integrating those several low-level events in the same window of time (*see Arrival Distributions* in Chapter 12).

It is a normal procedure that events are logged for ulterior analysis. The *log control* is concerned with the conditions in which this log is performed: if the log record is issued after a threshold on a level or on the number of consecutive occurrences is exceeded; if the log itself has a water mark and generates an alert after it is reached; if the log is done at the resource where events were produced, or at a central point (a manager), etc.

21.1.4 Distributed Systems Management

Until now, we have discussed management in general terms. It should be clear however, both from these introductory remarks and from previous parts of the book, that the interesting management is distributed systems management, since current systems have that nature. Distributed systems management (DSM) is concerned with ensuring that distributed applications execute

correctly, over a distributed infrastructure that also remains correct. This is important, since it indirectly explains the difference between DSM and NM (network management). In fact, Network Management has been a well-known and established discipline before distributed systems have gained their current momentum. It is concerned with the infrastructure— the computer and telecommunication networks— and with how the information goes back and forth, whereas DSM is concerned with how applications use that information in order to provide services to the users.

Table 21.1. Comparison between NM and DSM Functions

Network Mgt.	Distr. Systems Mgt.
node connectivity	information flow
reaction to partitions	information storage
load/congestion control	system SW integrity
performance tuning	service availability
routing	load balancing

Of course, they are complementary, but it is important to establish a difference, since given the complexity of today's systems, there is room for specialization in either field. For the sake of example, Table 21.1 presents a listing of typical functions of network and distributed systems management.

21.2 SYSTEMS MANAGEMENT ARCHITECTURES

Systems management architectures have evolved during the recent years. There have been several factors behind that evolution, amongst which we stress: the expansion of internetworking brought the need to cope with heterogeneity and domain independence; the expansion of distributed systems created the opportunity to use distributed systems techniques for decentralized but coordinated management. Evolution took place in several stages, expressed by classes of management architectures, which we describe next.

21.2.1 Homogeneous with Centralized Management

The homogeneous with centralized management architectures, depicted in Figure 21.3a, prefigure the situation of the early networks, mostly proprietary, and where little interconnection existed. These systems were small-scale and had by nature a centralized management, implemented by a single management system, and a single console, the station from which management is performed. Management was fairly straightforward, since systems were homogeneous and small.

21.2.2 Heterogeneous with Uncoordinated Management

Figure 21.3b shows how these architectures evolved, when networked systems started to proliferate. Whether homogeneous or heterogeneous, they consisted of gluing together several of the above-mentioned smaller networks, managed locally. In this new scenario, individual networks continued to be managed independently, in an uncoordinated way, despite the fact that they already had some interconnection. The uncoordinated management architectures represent the state of affairs where several network architectures exist, but where loose management is adequate, since they do not require close interaction. Namely, there exist several management systems, and several uncoordinated or very loosely coordinated consoles. This was the status quo just before the advent of large-scale distributed systems.

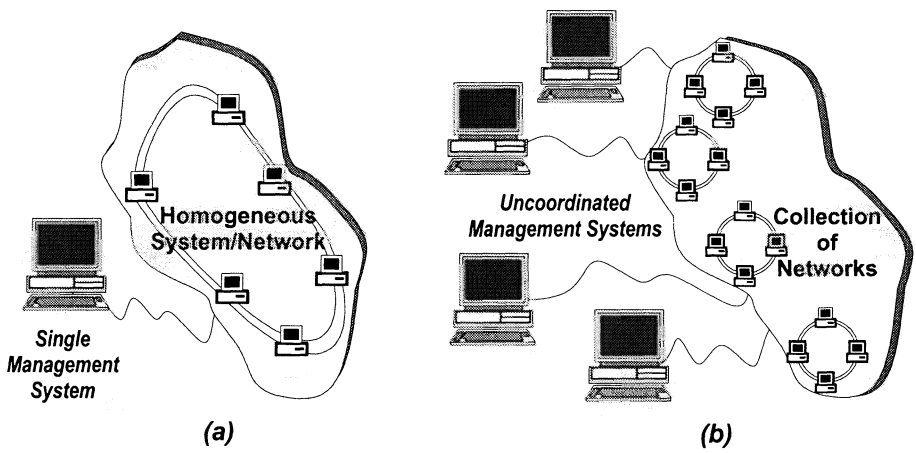

Figure 21.3. Management Architectures: (a) Homogeneous with Centralized Management; (b) Heterogeneous with Uncoordinated Management

21.2.3 Heterogeneous with Coordinated Management

The heterogeneous with coordinated management architectures, illustrated in Figure 21.4a, apply in situations with the same degree of technical evolution as above. However, the several network domains must act together, because the system has a moderate or large scale, and its components have significant interaction— e.g., they belong to or are controlled by a same organization and share significant amounts of data per time unit. In consequence, the management is physically centralized in a control room where all the several management systems and all the consoles are located. Although management is technically heterogeneous (each console runs a proprietary subsystem) it is coordinated at organizational level.

21.2.4 Heterogeneous with Centralized Management

In the meantime large-scale distributed systems made their appearance and the Internet technologies contributed to this evolution. The scale of systems under the control of an organization becomes such that several heterogeneous systems have to coexist under the same managerial umbrella. Besides, those systems have to be managed in a closely coupled way, since they support distributed applications that run across them.

This situation paved the way for the appearance of heterogeneous architectures with centralized management: whilst multiple management systems exist, they are ran from a single console. That is, the organizational-level coordination is mapped onto the technical-level coordination of all the consoles, from a centralized station. We also observe the utilization of distributed systems technologies to help manage the distributed system itself. In this kind of architectures the console centralization, as suggested in Figure 21.4b, is achieved through remote session protocols, such as Telnet, RPC and X-Windows.

The centralized console corresponds to the high-level notion of a physical control or `admin` center. Nevertheless, the console is location independent: the center can change location and be instantiated elsewhere. Technically, this works by having the physical management console, wherever it is, open windows over each of the management subsystems depicted in the figure. Since each system has its own interface, the heterogeneity of applications is still visible, at least for the more subtle semantic details. However, this is a major step towards integration of management functions, which we discuss next.

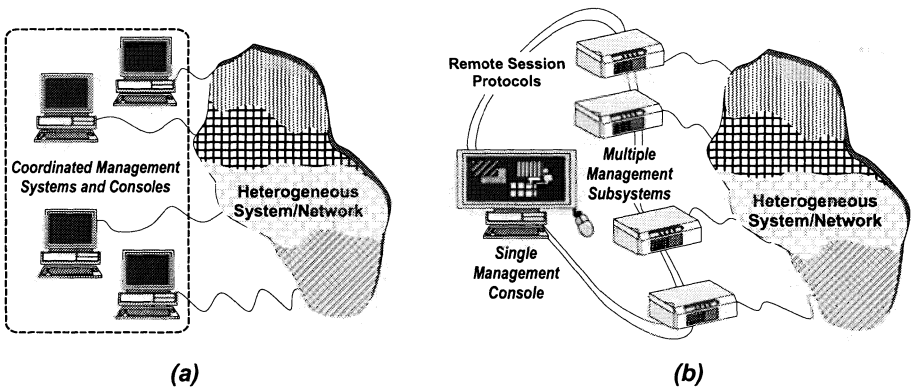

Figure 21.4. Management Architectures: (a) Heterogeneous with Coordinated Management; (b) Heterogeneous with Centralized Management

21.2.5 Heterogeneous with Integrated Management

The heterogeneous architecture with integrated management is an advanced distributed systems management architecture, and the state-of-the-art approach, in what concerns available commercial systems. As depicted in Figure 21.5a,

the advance with regard to the centralized management architecture is that the multiple management systems have a local scope (they could have local consoles), they can run local management programs by delegation of high-level management.

This architecture is more sophisticated than the previous centralized one, in that it assumes a degree of delegation and in consequence of hierarchy. Organizational-level management coordination is mapped onto a single *integrated management system and console.* The above-mentioned hierarchy also hides the heterogeneity of the local management systems: the applications running on the console address the local subsystems in a homogeneous manner, even if they have different makes. This is done through the utilization of distributed systems technologies, including but going beyond the level of abstraction of remote sessions: remote management protocols and APIs, common GUI interfaces, and sometimes a common database (MIB) representation.

The several systems have thus an apparent homogeneity, since at least their graphical representation and user interface at the central management system are uniform. Standardized protocols establish the dialogue between the integrated management system, and the local— and to a certain extent autonomous— management subsystems. Such as with centralized management architectures, there is location independence: the integrated management system and its console can change location and be instantiated anywhere.

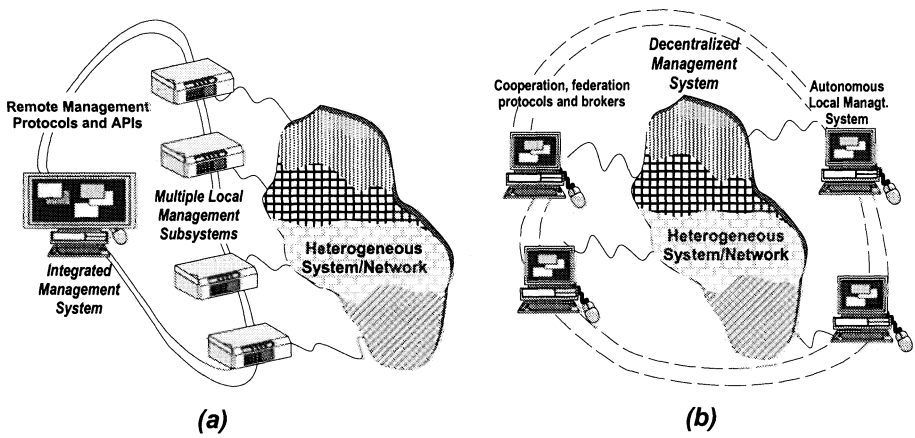

Figure 21.5. Management Architectures: (a) Heterogeneous with Integrated Management; (b) Heterogeneous with Decentralized Management

21.2.6 Heterogeneous with Decentralized Management

Distributed applications are attaining a scope, both in complexity and scale, that renders centralized or integrated management ineffective or even impossible. Observe enterprise networks, business-to-business *e-comm*, or other interactions that are done between realms of distributed systems that belong to

different organizations, or even countries. Within the scope of network management, the Internet has been managed under such a perspective: sophisticated routing protocols can make the difference between routing inside what are called *autonomous systems (AS)*, and between different AS's, which are essentially independent management realms.

However, large-scale distributed systems present more complex problems than just communication. This evolution requires a sort of federated management, in essence a heterogeneous architecture with decentralized management, as exemplified in Figure 21.5b. Management is fully decentralized. The organizational-level management is supposedly cooperative, and technical management is performed within the scope of each autonomous local management system, which may itself follow an integrated management approach, with its console and local applications. The effective construction of these architectures relies on distributed system paradigms such as: request brokers, message buses and enabling protocols and algorithms for federation and cooperation.

21.3 CONFIGURATION OF DISTRIBUTED SYSTEMS

A great deal of the success in operating and managing a distributed system depends on whether it is configured adequately. The use of structured configuration methods, besides allowing to build better systems, makes them more manageable. Configuring distributed systems architectures has two major steps: the hardware architecture; and the software architecture.

Configuring the *hardware architecture* begins with selecting the components: routers, hubs, links, and so forth, for the network infrastructure; servers, to install the services; and workstations, for the users to access the system. Then, placing and interconnecting these components, through physical links. That forms the infrastructure, and is concerned with defining the network layout and placing the hosts or nodes. These processes are iterative, and it is normal to come back to hardware configuration after having a first pass at software configuration.

Configuring the *software architecture* consists of selecting the software components: drivers and protocols; service modules; client modules when applicable. Then, placing and interconnecting these modules. This latter part is concerned with activities such as: placing services with servers, following rationale such as load balancing, proximity, or criticality; interconnecting protocols with higher-layer services, such as binding application modules to communication protocols; interconnecting multi-tier services, such as binding a web server to a web request broker and finally to a database engine. Part of the binding is dynamic, during runtime, making use of previously configured services such as name or directory services, and brokers or traders (ANSA, 1990).

Components should be modular, encapsulated, and have a well-defined interface, where both the services required by the component, and the services supplied by it are represented (*see* Figure 21.6a, for a representation based on the model of (Magee et al., 1993)). Note that this is concerned with a generic definition of systems architecture, and not necessarily related with a *client-*

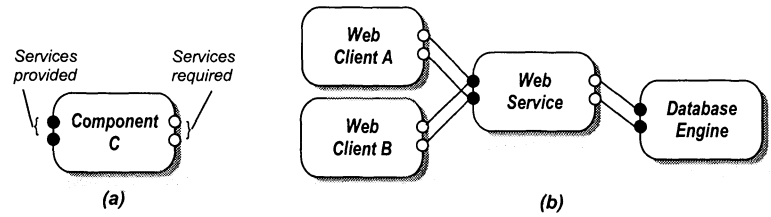

Figure 21.6. Configuration of Distributed Systems

server relation. The interface definition should be precise and non-ambiguous. This suggests the use of an object oriented approach, and of interface specification languages **(ISL)** or interface definition languages **(IDL)**. These languages may be further augmented by graphical configuration methods, where software architectures can be defined by interconnecting software components graphically, sometimes in more than one level. Figure 21.6b shows one such example of configuration, where two web clients (browsers) "require", and are thus connected to (by instantiating the adequate protocols), a web service (HTTP server), which in turn "requires" the provision of the database engine service (possibly through a CGI) to supply the material with which to build the dynamic pages needed for the web server to provide the service requested by the web clients.

System configuration may be static or dynamic. *Static configuration* assumes that whatever the initial configuration is, it will remain stable during the system lifetime, such as in Conic or Durra (Magee et al., 1989; Barbacci et al., 1993). It is through *dynamic configuration* that the desirable relationship with distributed systems management is established. In fact, the idea is that systems change, in the course of operational events such as faults or of mere evolution, and it should be possible to accommodate that change without pain, that is, without stopping the system and going back to the design desk and testing laboratory. Incorporating the ability to support operational changes in the system configuration requires foreseeing those changes, for example by defining several operational modes, and instantiating the relevant components when needed. Incorporating evolution can be achieved by expressing the system configuration in the form of a configuration database, and defining the initial system configuration in a non-declarative language, such that the configuration may change during the system lifetime in a non-programmed way, such as done in Darwin (Magee et al., 1993) or Olan (Bellissard et al., 1996). Clipper (Agnew et al., 1994) and Evolution (Radestock and Eisenbach, 1996) are other examples of dynamically reconfigurable configuration languages.

21.4 SUMMARY AND FURTHER READING

In this chapter we debated the introductory notions concerning configuration and management of distributed systems. After defining management and discussing its lifecycle, we presented management in practical terms, by introduc-

ing the main functions and support services. We presented the several architectures for distributed systems management, ranging between homogeneity and heterogeneity, and between centralization, decentralization, integration and autonomy. We finalized by making an introductory discussion of the problem of distributed systems configuration. In (Sloman, 1994), a good introduction to systems management, and distributed systems management in particular, can be found, where some of the discussed issues are further detailed. A distributed programming environment based on 'configuration programming' is presented in (Magee et al., 1994).

22 PARADIGMS FOR DISTRIBUTED SYSTEMS MANAGEMENT

This chapter addresses the main paradigms for distributed systems management. Management models have been developed in the past few years, mainly in the course of standardization activities, such as OSI Systems Management, the Internet Engineering Task Force, or the Open Distributed Processing initiative, but also under significant research effort. As these models have matured, a number of significant paradigms have been retained, and made it possible to define the generic body of research and technology of today's systems management. We will make a non-exhaustive effort to study the main paradigms, and in consequence, we will address: managers and managed objects, domains, management information bases, and the several management functions— configuration, faults, performance and QoS, accounting, security, names and directories.

22.1 MANAGERS AND MANAGED OBJECTS

Management is about *managers*, the performers of the act of managing, and the targets of the act, the *managed objects*. The manager executes management functions, by requesting operations on the objects managed by it, and by receiving notifications from those objects.

Managed objects are a form of uniformly modeling whatever is manageable or needs to be managed. We talk of objects in a broad sense, which obviously intends to capture some of the interesting properties of genuine objects: the

P. Verissimo et al. *Distributed Systems for System Architects*

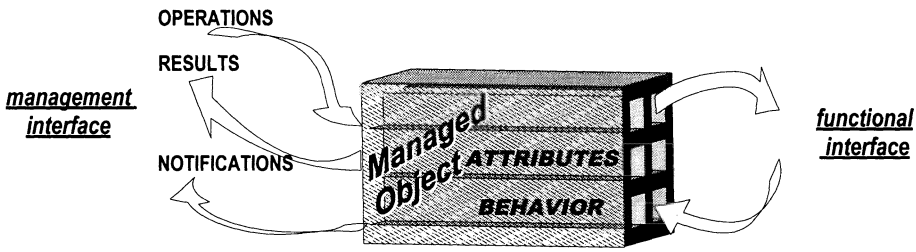

Figure 22.1. Modeling a Managed Object

encapsulation, the well-defined interface. Managed objects have a certain functional encapsulation. Sometimes, a natural one, around a hardware module such as an ethernet adapter board, sometimes one that has to be extracted from a bundled subsystem, such as a disk inside a computer. Besides their functional interface, they also have a **management interface**. Likewise, this interface is not always a natural one, since some objects are hardware units, and the interface has to be implemented by software structures outside them, in some controlling station that represents their state in the best manner possible.

This model of a managed object is represented in Figure 22.1, where we can see the separation between interfaces. The object itself is characterized by its behavior, represented by actions invoked at the interfaces, and its attributes. The management interface allows three kinds of **actions**:

operations:	requests invoked on the object, either to read its state, or to modify it
action req.	to perform an action on an object, such as writing a variable (attribute), open a port, reset a connection
information req.	to read the state of an attribute, e.g., the state of error counters, or of a network printer
results	issued back to the caller by the managed object, in response to invoked operations
notifications	unsolicited information sent by the managed object to the manager, about events observed by the object

The **attributes** are the object properties visible from the outside, and have a value that depends on their nature. Some attributes are writable, others are read-only (e.g. counters externally available, but updated internally by the object), others are constants (e.g., identifications, network board addresses). Part of the operations on objects are primitive operations on attributes: `get()` or `read()`, to obtain attribute values; `replace()` or `write()` to change the attribute value. Complex actions, such as combined actions on object attributes, are represented by a template of the form `application_specific()`. Attributes are *never* accessed directly, but through the management interface, so that the appropriate validations can be made (e.g., bounds checking, ac-

```
LaserPrinter MANAGED OBJECT CLASS
DERIVED FROM ...  (existing top class)
CHARACTERIZED BY:
  BEHAVIOR
  ATTRIBUTES
      PrinterID   GET,
      MACaddress  GET,
      IPaddress   GET,
      OnLine      GET,
      TonerLevel  GET,
      CurrentTray GET,
      DoubleSide  GET,
      TimeSincePrinterReset GET,
      .........
    ACTIONS
      OnOffLinePrinterToggleAction,
      ResetPrinterAction,
      .........
     NOTIFICATIONS
      MaintenanceRequired,
      OutOfPaper,
      OutOfToner,
      .........
NAME BINDING
SUBORDINATE OBJECT CLASS  LaserPrinter;
NAMED BY ...    (existing superior class)
WITH ATTRIBUTE            PrinterID;
.........
```

Figure 22.2. Example of Managed Object: a Network Laser Printer

cess control, etc.). Other general actions on objects are: `create()`, which creates an instance of the object; `delete()`, which destroys an instance; and `action()`, which encapsulates one of several management functions (e.g., configuration, fault, performance/QoS, accounting, or security). The object can in turn invoke a `notification()`, or up-call, to the manager, to inform it of some relevant event (e.g., buffer overflow, printer out of paper). An example description of a networked laser printer as a managed object is given in Figure 22.2. The attributes, actions and notifications are defined in the description. For example, if `get(OnLine)` returns `true`, and it is followed by `action(OnOffLinePrinterToggleAction)`, the printer will go off-line.

22.2 DOMAINS

Management domains are groupings of objects for the purpose of establishing management policies. They are an important paradigm, firstly as a vehicle for mapping management policies to mechanisms, secondly as a means for easily executing management operations on groups of objects. The following are examples of domains: workstations on a same LAN, managed locally; the set of servers that form a distributed file system; the group of internal routers of an organization facility.

Domains allow a set of operations, such as: `include()` and `remove()`, which add or remove an object ID to a domain; `list()`, which lists all objects in a

domain; `move()` is built by doing `include()` in new domain and `remove()` from old domain; `create()` and `delete()` operations, which we saw earlier, are in fact done under the scope of domains, if the system supports them. Why? If an object existed outside any domain, there would be no management policy for it, and in consequence it could not be managed.

Domains have another advantage: by policy, restrictions can be established to the operations on a domain, and certain guarantees on the correctness of management operations can be automated. For example: a domain of Intel machines can be set-up so that only iAPX-compatible binaries may be installed on the objects of that domain; a domain whose objects are compatible with a given management protocol can disallow the insertion of objects that are not. Inside a domain, sub-sets of objects may be addressed by regular expressions, for example ⟨*all workstations of domain LAB1 with Linux O.S.*⟩.

22.3 MANAGEMENT INFORMATION BASE

The management information base *(MIB)* paradigm is the conceptual repository of the management information. It should contain the descriptions of all the resources relevant to the system management, such as:

- network components: hosts, gateways, routers,...
- protocol entities: TCP, XNS, IP, X.25,...
- dynamic objects: TCP connections, secure tunnels,...
- administrative objects: trouble tickets, user records, installation procedures,...
- auxiliary objects: schedules, event records, filters,...
- application objects: services, algorithms,...
- object attributes: error counters, flags, water marks,...

After having introduced the managed object paradigm, it is obvious that the MIB must be constructed in terms of the former, as depicted in Figure 22.3. The description of the objects should be done in a common, standardized, and structured way, for example as shown back in Figure 22.2. Note that it is desirable that access to the MIB is automated, as much as possible. MIBs may have a very large number of objects, and management code elements may be reused, or applied to groups of objects, belonging to devices that are possibly of different brands and makes. This is only possible if the description of the objects is precise and has a well-determined content. Structured languages exist to achieve these objectives, such as ASN.1 (Abstract Syntax Notation) (ASN.1, 1990).

Technically, the MIB may assume several forms. It is normally distributed, scattered through several devices. Management platforms may have more complex MIBs, possibly under the control of a database manager, that hold caches or copies of managed objects residing elsewhere. In terms of behavior, a MIB captures much of the notions we studied in the Real-Time Part of this book, of active and real-time databases (*see* Chapter 13): the information stored has temporal validity; changes in the database may cause triggers to be produced.

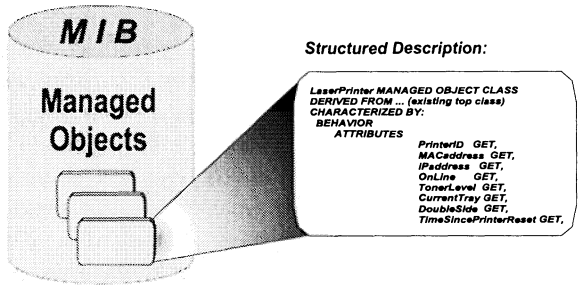

Figure 22.3. The Management Information Base (MIB)

22.4 MANAGEMENT FUNCTIONS

In this section we will introduce the main management functions, some of which will be detailed in the subsequent sections: configuration management, fault management, performance and QoS management, accounting management, security management, name and directory management. Some of these will deserve a detailed discussion in the sections to follow.

Configuration management is concerned with analyzing and maintaining the configuration of a network or system. It is normal that this information is graphically displayed. Typical functions of configuration management are: automated discovery of network topology; automated update of configuration; remote configuration and reconfiguration.

Fault management is concerned with detecting and solving error situations. We have studied the fundamental concepts and mechanisms relevant to fault tolerance in Part II of this book. Let us put the subject in context with management. Faults give origin to errors, which are reported in alarms. Alarms may be divided in a few broad classes: communication, software, hardware, environment, and non-functional attributes such as QoS or security. The alarm may further carry a few useful parameters, such as: probable cause (e.g., the physical fault causing the error); severity (the potential for causing damage); origin (location of the object causing or suffering the error); and optional information about a possible exceed threshold (e.g., omission degree, that is, the number of allowed omission errors during an interval). After the alarm, comes the solution: reconfiguring the system under operator intervention, repairing the fault that caused the error, or tolerating the fault with redundancy. Fault diagnosis, either by routine or after service is re-established, is desirable: connectivity testing, data and protocol engine integrity, link integrity (loop-back and echo), self-test. Typical functions of fault management are: monitoring the system to detect alarms and perform preventive fault diagnosis; alarm processing; error confinement and recovery; interfacing with operator and user assistant tools, such as trouble ticket and help desk facilities; audit trail, i.e. logging alarms for ulterior analysis and fault diagnosis.

Two related functions, **Performance** and **QoS** management, address the problem of maintaining the performance of the system within specified limits. QoS management is a more precise measure of the non-functional attributes of a distributed system. Performance management is more concerned with network metrics: speed, throughput and latency. It may include performance as well as fault parameters, such as the allowed omission degree or error rates. Typical functions of performance/QoS management are: general QoS parameter measurement; monitoring of the system for detection of QoS violations, such as network congestion, host overload, unusual delays; execution of specific measurements, calculation of statistics and production of reports; medium-term capacity planning, such as latency and throughput.

Accounting management has some analogies to performance management, but its main functions focus on user metrics, such as: collection of utilization data; construction of statistics; resource usage accounting; allocation of resource usage quotas and costing. Such as with performance management, there are special objects devoted to accounting management. The *meter control* and *meter data* objects respectively control the acquisition and storage of accounting data, such as kilobytes of bandwidth, megabytes of disk storage, CPU seconds, etc. One meter control object may control several meter data objects. The *usage record* objects hold the relevant records of utilization per user, that will be used to produce for example per user accounting reports.

We have already dedicated quite a few pages to security in this book. Talking about **Security** management is talking about applying those notions in the context of management. Part of the strategic management policy concerns security, and tactical management with this regard concerns the necessary measures for the implementation of the security policy. Typical functions of security management are thus: intrusion detection; authentication; protection measures; contingency plans for security hazards, such as intrusion countermeasures. Security management is mostly related with how to do the mapping or establish the correspondence between *security policy*, and *security measures*.

Name and **Directory management** are two sides of a same coin. We have already discussed the need for name and directory services in Part I of this book (*see Name and Directory Services* in Chapter 4). Name and directory services bear an obvious relationship with management, since in order to manage objects, we need information about them, such as how to name them and their whereabouts.

22.5 CONFIGURATION MANAGEMENT

The basis of configuration management is that objects make available configuration information, in the form of attributes. The description of managed objects relevant to configuration management is made in terms of three classes of attributes:

- **state** - referring to the global state of the object, such as if it is up or down;

- **status** - referring to detailed state variables internal to the object, such as whether a certain functionality is available, or a certain alarm was issued;
- **relation** - referring to the relations among objects, such as if the object is a replica or back-up of another.

The object state variables may for example be (Langsford, 1994): *operational*– whether the object is `disabled` or `enabled`; *utilization*– whether the object is normally loaded (`active`), heavily loaded (`busy`), or free (`idle`); *administrative*– under operator intervention (`locked`), or in normal operation (`unlocked`). All these variables but the *administrative* are read-only. The status attributes denote the object state variables visible from the outside, for example the `OnLine` variable, or the `OutOfPaper` alarm. The relation attributes may denote that the object belongs to a `Group`, or that it is the spare copy `StandByOf` another object.

Planning is related with configuration management, in the sense that it precedes configuration. As a matter of fact, they alternate, as the system evolves and needs to respond to new challenges. Planning is a strategic action, and it is very important for network or system management. In fact, we have already mentioned that the system must have the capacity of adapting to changing situations, of having a certain dynamics, of complying with QoS specifications. This takes place on-line, while the system is running, it is part of the tactical abilities of the system to respond to new situations. However, none of this can be achieved if the system has not been configured adequately. Strategic planning concerns the phase *before* the system is configured, or intermediate phases where it is necessary to reconfigure the system in order that it can respond to significant modifications of the operational conditions. It consists of taking into account all the requirements that the system will be faced with and study configuration alternatives, playing with hardware and software architecture, after which a configuration plan is prepared and executed, sometimes step-by-step, to ensure painless commissioning.

Sometimes, a network or distributed system is already in place when a management system or platform is initialized. In this particular situation, the management platform must *learn* the configuration of the system being managed. It can be loaded manually, or the platform can find it by itself, through **automated discovery**, an interesting paradigm of configuration management. Automated discovery consists of capturing information from the environment, in order to discover, locate, and collect information about devices, and their managed objects. Automated discovery is straightforward when all system devices respond to the protocols used by the platform to perform its discovery campaign. When that does not happen, an effort must be made to complement it (Norton, 1994; Siamwalla et al., 1999). Protocols at different layers may contribute to discovery. For example, a router may discover what is at the end of each link it is connected to, and share this information with the other routers, creating a topology map of the network. Any host (e.g., a bridge) can learn about the hosts located in the LAN segment(s) it is connected to, by logging the MAC source addresses it sees passing. These are examples of *passive dis-*

covery. In complement, there may be *active discovery* actions, where devices purposely scan for new resources. In practical systems, what happens is that the platform discovers a very large percentage of devices passively, complements that through active discovery, and the information about configuration ends-up being completed and tuned manually.

22.6 PERFORMANCE AND QOS MANAGEMENT

There are special objects devoted to performance/QoS management. These *metric objects* serve to collect statistical information about the system evolution, and have methods that compute statistical functions about the utilization of resources, quality of the infrastructure, and so forth. These methods can be simple averaging functions such as computing the mean of the last n samples of an attribute, or more sophisticated ones that avoid storing $n - 1$ samples, such as an exponentially weighted mean (Sloman, 1994).

QoS management has become increasingly important as QoS architectures proliferate, for example in areas such as distributed multimedia. As a matter of fact, it should be seen as a generalization of performance management, that is, we could talk only about QoS management. Its goal is to ensure that a *QoS specification* for a service remains within the specified parameters for the duration of service provision. Recall that a **QoS specification** consists of a list of dimensions, and values and intervals for them (*see Quality-of-Service Models* in Chapter 13). For example:

$$\langle \texttt{throughput} > 1Mb/s;\ 100\mu s < \texttt{latency} < 50ms;\ \texttt{BER} < 10^{-5} \rangle$$

Once contracted, a QoS specification may be violated both by the provider or by the contractor. In fact, ensuring a given QoS at the start of an application is not a guarantee that the QoS will hold for the application lifetime. The problem is to *maintain* the QoS of *individual* activities, supplied by resources *shared* by other activities, in the presence of changes in the *operating conditions* of the infrastructure. One part of the solution to the problem is *monitoring* the infrastructure, to ensure that the QoS remains within the parameters, and detect deviations early enough. This management function is assisted by QoS failure detectors. Monitoring may indicate the systematic (statistically meaningful) failure of a QoS parameter. This is reported up to the support middleware or to the application, with an event notification. There are several possible responses to this event: termination; renegotiation; adaptation. Termination is the outcome when the application is not capable of operating with a lesser QoS. Renegotiation takes place as an attempt of the application to contract a similar QoS, sometimes with a different combination of parameters, or a different network support. Adaptation occurs when the application is elastic enough to withstand the reduction in QoS, an adapt to the new operating characteristics (*see* again *Quality-of-Service Models* in Chapter 13). The other facet of maintaining QoS is *policing* the user activity to ensure that the QoS requested at every moment is within the contracted QoS. For example, the user might be grabbing more bandwidth than it had contracted, possibly jeopardizing the QoS of other applications.

22.7 NAME AND DIRECTORY MANAGEMENT

Recall that a name service allows the identification and location of users and services without requiring previous knowledge of their whereabouts. Directory services have a broader scope than their name service counterparts. They hold more information about subjects than just the location.

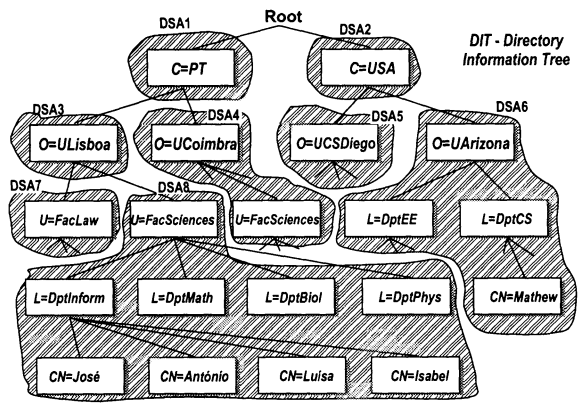

Figure 22.4. Managing X.500 Information

From the management viewpoint, a directory service, more than a name service, offers crucial support. We refer to *Name and Directory Services* in Chapter 4) to show how object names and other information can be managed in a large-scale infrastructure under X.500, as depicted in Figure 22.4. In order to manage an object, we need to have some information about it. This information is stored in the *Directory Information Tree (DIT)*, managed by several *Directory System Agents (DSA)* hierarchically. DSAs are the managers of the entities of the subsystem they subtend (shaded areas). A name can be looked up by traversing the DIT following the distinguished name fields down to the final leaf, the common name of the object or subject. Once the name obtained, all the relevant information about the corresponding object can also be retrieved.

22.8 MONITORING

Monitoring is a paradigm representing the set of activities aiming at knowing the state and evolution of the system during its operation. It is the basis of most tactical management functions (the reader will perhaps remember that we mentioned the word 'monitoring' several times during our discussion of other paradigms in this chapter). As a matter of fact, tactical management relies on the reactive system principle: the loop of monitoring (state acquisition) and control (state modification).

Acquisition is performed through *sensors*, which may be implemented either in hardware (e.g., power supply voltage drop sensor), or in software (e.g.,

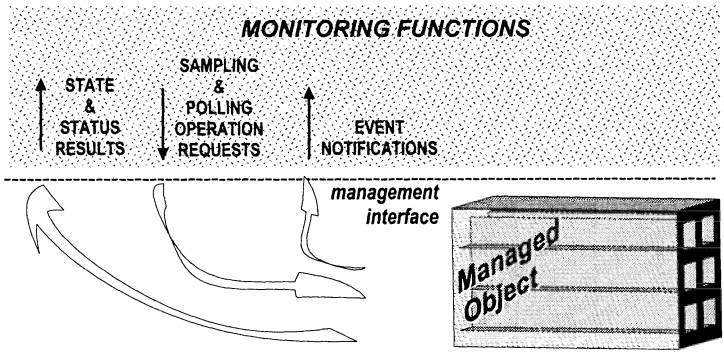

Figure 22.5. Monitoring

boolean state transition detector). Sensors are incorporated in the managed object concept, and monitoring is characterized by interactions that extract state snapshots and collect events from managed objects, as presented in Figure 22.5. The monitoring subsystem must acquire information both by polling or sampling the managed system state (solicited operations to the managed object) and by capturing all the events produced by the managed system (unsolicited notifications from the managed object). In this matter, monitoring incorporates notions on input-output sensing and actuating, studied in the Real-Time Part of this book (*see* Chapter 12).

22.9 SUMMARY AND FURTHER READING

This chapter addressed the main paradigms available to the systems architect in order to implement the management models that we discuss next. Namely, we discussed managers and managed objects as the fundamental actors, the role of domains in structuring management operations, the MIB (management information base) as the conceptual central repository and representative of the structure and state of the resources, and the main management functions, of which we followed with detail: configuration, performance and QoS, name and directories, and monitoring. For a deeper study on the paradigms debated in this chapter, see (Hayes, 1993; Sloman, 1994). Namely, Hutchison et al. do a treatment on quality of service management, Zatti addresses name and directory management. See also a CORBA-based QoS management framework in (Hong et al., 1999). Further material on fault detection and alarm correlation can be found in (Ricciulli and Shacham, 1997; Lewis and Dreo, 1993; Hood and Ji, 1996). An application of the domains paradigm to role-based enterprise management can be found in (Lupu and Sloman, 1997). A study on information push and pull paradigms applied to web-based management is done in (Martin-Flatin, 1999). In (Crane et al., 1995; Fosså, 1997), object-oriented and graphical configuration management environments based on the Darwin (Magee et al., 1993) language are presented.

23 MODELS OF NETWORK AND DISTRIBUTED SYSTEMS MANAGEMENT

This chapter aims at giving the reader a global view of the problem of management. After studying several paradigms, we see how they fit in several models for management of networks and distributed systems. We start by presenting management frameworks: functional, such as configuration, management, monitoring; and structural, such as the tool and platform levels. Then we discuss the strategic alternatives the architect is faced with, clarifying the difference between strategy and tactics, and the subtleties between distribution, centralization and decentralization. Finally, specific models for distributed systems management are presented.

23.1 MANAGEMENT FRAMEWORKS

Network or Distributed Systems Management? It is important to emphasize the difference between network and distributed systems management again in this context. The framework of network management is focused on communication, and is the more mature branch of management. Network management (NM), or the telecommunications management network (TMN) are of course adequate frameworks for telecom or computer network operators. Distributed systems management (DSM) has the broader scope of system support and application processing, in addition to networking support and communication. In our opinion, it is very advisable for an enterprise to talk about DSM whenever possible, despite eventually having a considerable networking infras-

P. Verissimo et al. *Distributed Systems for System Architects*

tructure. The end purpose of most networking facilities we see today is the support of distributed applications for organizations and for public use. DSM is the adequate framework for that level of abstraction, whereas NM is the framework to be considered by whoever is providing the infrastructure, be it network operators, or the organization itself, or both.

Configuration Together with planning and development, configuration, that we addressed in Section 21.3, materializes the framework of preparation of the hardware and software architecture of the system. This involves selecting the hardware and software components, and then placing and interconnecting them. We also discussed structured configuration methods in order to improve manageability of the system. In essence, successful configuration depends on how well a few desirable characteristics are mastered:

- graphical programming— easy placement and interconnection
- well-defined components— modularity and encapsulation
- modeling of behavior and interactions— precise interface specification

Management Understood as the set of activities concerned with ensuring the correct operation of the system while at work, management is concerned with tactical issues, also called short-term management, as opposed to strategic or long-term management. In Section 22.4 we saw that the main management functions are: configuration management, fault management, performance and QoS management, accounting management, security management, name and directory management. These management functions are performed by managers, as we saw in Section 22.1. They operate on managed objects, either directly, or through agents that establish a hierarchy, or through proxies that implement gateway functions, as we will see in Section 23.3. The management information model is materialized by the Management Information Base, or MIB. The MIB, introduced in Section 22.3, is the conceptual repository of all the information relevant to management. We will see that there are several standard MIB formats in Sections 24.1 and 24.2.

Monitoring Introduced in Section 22.8, monitoring is the framework of real-time supervision of the state and of the evolution of the system. It assists most of the management functions. Note that management functions are structured around the MIB, and a good part of the content of the latter refers to time-sensitive information that resides on the managed objects (e.g., error counters, number of open connections, used memory). Besides, a great deal of on-line or tactical management operation is event-triggered, that is, managed objects send information up (notifications) that need attention, processing and timely reaction. In consequence, we need a monitoring subsystem that follows these events and extracts state information in real-time: periodically for state samples, and upon their occurrence for events. We are going to present a complete monitoring model in Section 23.9 that foresees data processing capabilities, dissemination to peer managers, and graphical presentation facilities. The reader

will note a few keywords relevant to monitoring that have been introduced in the Real-Time part of this book: timeliness; sensing and actuating; event-triggered.

The Tool Level Structurally, management and monitoring functions are performed by special applications that we call management tools. Tools specialize in one or a few functions and are normally installed at the manager location. For example, we can have a configuration management tool, or a security management tool. The console from where the tool is run is the operator's console for those functions. We will discuss tools in Section 24.4.

The Platform Level Obviously, a fully-fledged management facility requires a comprehensive set of management and monitoring functions. A platform is a set of tools integrated in a single package, such that the different tools share common information acquired from the managed objects, and have similar, if not common, interfaces. Platforms can be distributed, but they can be controlled from a single location, where the management console is instantiated. We will discuss platforms in Section 24.5.

23.2 STRATEGIES FOR DISTRIBUTED SYSTEMS MANAGEMENT

Strategic Management Strategic management has a long-term view of the system. It starts with system planning and configuration. Strategic planning aims at replying to requirements and casting them into the configuration of the system. Once the system configured, strategic management is about establishing management policies, that is, preparing for the tactical management of the system. Systems should not be managed ad hoc. Worse than not having a management platform is installing such a platform without knowing precisely what to do with it. Strategic management decisions are things such as deciding about restrictions of access and space quota to the public FTP directory per class of outside and inside user, or defining the backup policy with regard to periodicity and completeness, per class of service and user.

Tactical Management Tactical management is the real-time, reactive, and short-term part of systems management. Supposedly educated by a management policy, tactical management concerns the execution of several management functions, processing information obtained by monitoring. Tactical management is a good part of the management activity, and most of the actions are concerned with responding to unforeseen situations, raised by events and changes in state variables. Tactical management decisions are for example establishing disk quotas for the FTP directory given an actual disk capacity, increasing the disk quota upon the organization of an event by the institution, or implementing an automated hierarchical and periodic backup procedure.

Distributed Management or Distributed Systems Management? Distributed management is about managing systems in a distributed way. Dis-

tributed systems management is about managing distributed systems. Obviously, distributed managing of distributed systems seems a good idea, but we are really talking about different things. Note that management is classically centralized, in the person of the *manager*, who runs a *management console*, from where she controls the system. This characteristic *does not need* to change because a system is distributed. In fact, networks have a great geographical dispersion, and they have been managed in a non-distributed way for years. On the other hand, a great deal of the existing distributed platforms follow paradigms that aim at achieving distribution transparency, that is, hiding distribution from the users, and making the system look as a huge single virtual machine. It may even be a good strategy for small-scale homogeneous systems.

But that characteristic *should* change when the system is not only distributed but is also heterogeneous, has geographical dispersion and is possibly large-scale. As we noted in Section 21.2, the fact is that the proliferation of heterogeneous and large-scale distributed systems and networks made it impossible to do centralized management of these complex infrastructures in a non-distributed way. Distributed protocols and algorithms are necessary to: handle the correct execution of remote operations as if they were executed from the central console, perform manager-initiated group operations and information dissemination reliably, manage replicated information, and so forth.

Centralizing or Decentralizing? The last paragraph prefigures a strategy that favors the use of distributed *techniques* in support of centralized management, as an artifact of the solution to problems such as heterogeneity, scale, unreliability, inconsistency, etc. What is called integrated management takes the use of distributed techniques even further, in order to run the management protocols and applications themselves as distributed protocols and applications, albeit from a central locus of control. It is a centralized management strategy where transparency is achieved as much as possible through the use of sometimes sophisticated distribution techniques. However, note that this attitude maintains the classical *manager-centric* view of management. Alternatively, we may follow a strategy in favor of *decentralized* management of a distributed system. In this case, distribution appears as a natural consequence of the autonomy of the local systems. The same techniques that have recently been used to support autonomous and cooperating distributed applications may be used for these management systems, such as request brokers, message buses, and so forth (*see* Chapter 4). The several models for centralized, decentralized and integrated management will be addressed in Sections 23.4, 23.5, and 23.6.

23.3 A GENERIC MANAGEMENT MODEL

A generic management model is presented in Figure 23.1, in terms of which all specific models that we encounter can be described. The manager is the entity that executes management functions on, and collects information from, the managed objects, either directly or through assisting components. In fact, managers may access managed objects in one of three ways:

- directly;
- through agents in which they delegate direct access to objects;
- through proxies, to access alien resources.

Managers typically reside in the management console host, whereas agents and proxies normally reside in managed hosts, and control a set of devices, for example those of the host itself, or those of a LAN segment where the host resides. The interactions shown in dashed lines between manager and agent are in fact performed by **management communication protocols**. The same happens with agents and managed objects when residing in different sites, or with managers and managed objects when accessed directly. These protocols are either proprietary or standardized, though the trend is for a standardization of all management access, for example through the ISO CMIP or the Internet Simple Network Management Protocol (SNMP). Managed objects are shown as possessing two interfaces, the *functional interface*, from where it is possible to make the object perform its functions, and the *management interface*, from where it is possible to manage the object.

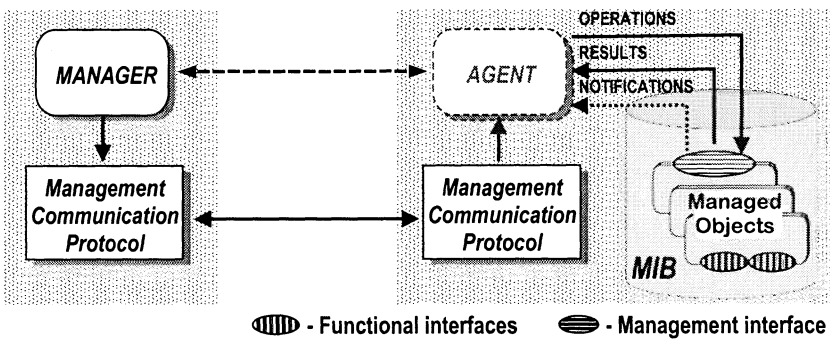

Figure 23.1. Generic Management Model

Agents As sub-managers, so to speak, agents perform operations on behalf of the high-level managers. As show in Figure 23.1, the agent accesses the objects through operation requests, in result of high-level commands issued to it by the manager. It gets results and notifications from the objects it subtends, which it forwards to the manager. The manager may access managed objects directly, a situation appropriate to small systems, corresponding to the configuration where the dotted 'agent' box disappears in Figure 23.1, and the manager uses the management communication protocols to issue requests, get results, and receive notifications.

The agent concept is inspired by a "hardware" view of the system, the view partaken by the ISO OSI model. In an abstract view of the system, for example that conveyed by the ODP model, instead of being special components, the manager and the agent are both objects. The manager object accesses managed objects directly or delegates in an agent, which becomes in fact a *composite*

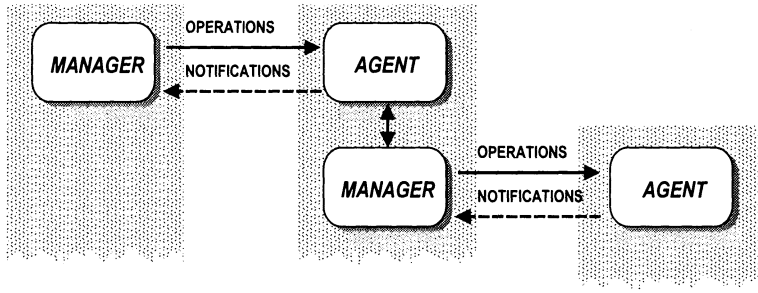

Figure 23.2. Management Hierarchy

managed object. That is, the agent presents the manager with a managed object interface, so that all the manager sees are objects. However, the agent behaves as a manager to the objects below on behalf of the actual manager. This renders the model homogeneous, since the manager accesses composite objects as normal managed objects, through their management operation-result-notification interface, and these objects access in turn a cluster of actual managed objects, through the same kind of management interface.

Hierarchy By recursing the manager-agent relation, one may introduce hierarchy in the model. This is extremely useful when managing large-scale architectures, and/or when the system architecture is itself hierarchical. As shown in Figure 23.2, the agent offers a "slave" interface to the manager, while acting to the layer immediately below— another agent— as a *mid-level manager*, and so forth. Note that the 'composite object' model also fits perfectly in a hierarchical structure: an object is a manager for a collection of managed objects below, but is a managed object to the upper layer manager object, and so forth. *Delegation* is the mechanism for introducing decentralization down the hierarchy (Goldszmidt and Yemini, 1995). Managers may delegate an increasingly higher level of functionality on the agents, conferring them the sort of properties (e.g., autonomy, reactivity, pro-activeness) that ultimately lead to highly decentralized structures based on intelligent and/or mobile agents (Sahai and Morin, 1998; Zhang and Covaci, 1997).

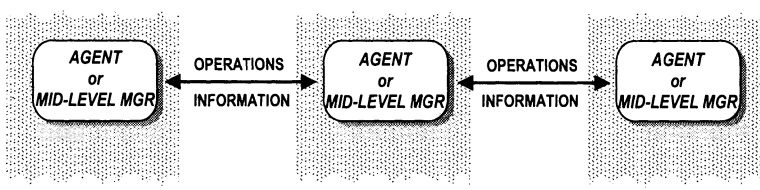

Figure 23.3. Management Cooperation

Cooperation Agents may cooperate to fulfill a management activity. Rather than hierarchical, this establishes a peer-to-peer relation between agents or mid-level managers. As suggested in Figure 23.3, agents cooperate in order to perform a complex management task, exchanging information and operation requests. Hierarchy and cooperation may be combined: cooperation relations may be established between the agents of a given level of the hierarchy. These agents may have a one-to-one relationship with a manager, or a one-to-many relationship, understood as a collective delegation from the manager to a group of agents, in order to perform some task. Inside the group, agents establish a many-to-many, or cooperative relationship. For example, the manager-agency paradigm (Post et al., 1996) defines a specialization of the manager-agent hierarchy, where a cluster of agents can be grouped in an *agency*, which has a common management policy, and consistency requirements among its members.

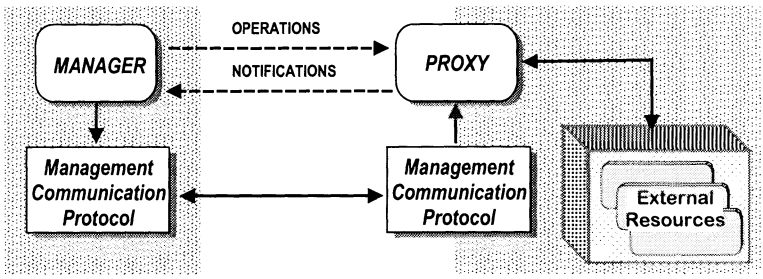

Figure 23.4. Proxy Management

Proxies There must be a way of accessing alien devices, that is, devices that implement a different management communication protocol, proprietary or not, or do not implement any protocol at all (for example because they are too simple, or just passive). In this case, a proxy agent or object acts for the manager as a normal agent or a managed object, and they both communicate through the management communication protocol of the architecture, as depicted in Figure 23.4. On the other end it dialogues with the alien resources in whatever manner necessary, sometimes through direct wired connections. In essence, it plays the role of a gateway in a communication stack or of a transformer in an object model.

23.4 CENTRALIZED MANAGEMENT MODEL

Centralized local management is the classical model, inspired by the traditional mainframe-oriented "computing center" concept. Homogeneity and geographical concentration made integration happen naturally. The structure was also compact in terms of personnel, who resided at a single central facility. With the advent of networking, a simple management tool would extend the existing functionality to manage the network infrastructure. This model was challenged when the use of networks expanded, and it became necessary to coordinate sev-

eral clusters of resources, either because they lived in different locations or had heterogeneous functionality.

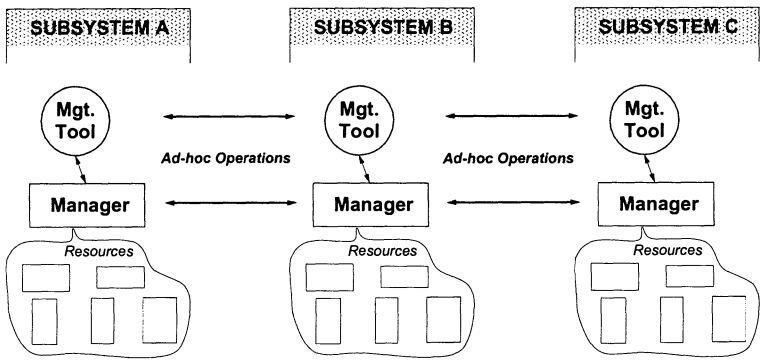

Figure 23.5. Islands of Management in the Centralized Model

This first attempt at the transposition from local to distributed management originated what we may call *islands of management.* This primitive model relied on simple tool and protocol level scripts and file transfers in order to loosely coordinate operations, but it retained all the autonomy of each island, as depicted in Figure 23.5. The islands preserved the computing-center model of management, originating a syndrome of competing power by the local managers. This situation conflicted with the increasing integration of information in each enterprise, requiring tighter coordination.

23.5 INTEGRATED MANAGEMENT MODEL

Management had inevitably to attain the desired level of integration, which was achieved in this evolution interim through mobile management teams, to execute the same actions at all sites, or through a greater cooperation between the autonomous management teams, to a large extent manual.

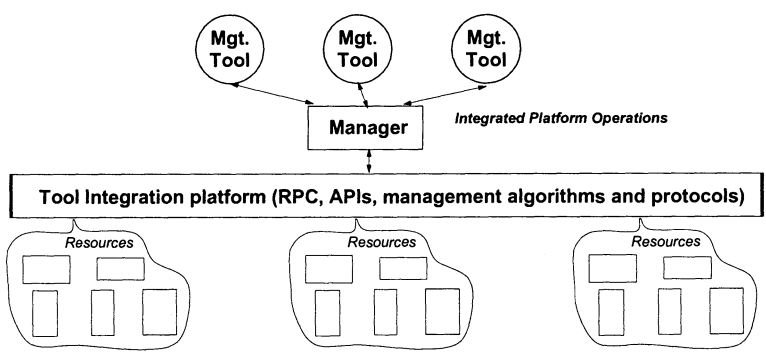

Figure 23.6. Integrated Management Model

The integrated management model solves the conflict, by allowing management to be organizationally centralized, both strategically and tactically, without incurring the shortcomings of physically centralized management. Major strategic decisions impact all subsystems in the same manner, and tactical management actions are performed in a centrally coordinated way. However, by resorting to the adequate distributed systems techniques, this model achieves a separation of concerns between the organizational and technical levels. Technically, management actions can be performed remotely, in a distributed manner, from a management platform. This platform is location independent. The model is shown in Figure 23.6. Integration has two facets: running several tools in a coordinated way; and managing the whole of the system resources in the several subsystems. Amongst the necessary techniques are: remote operation support (remote session, RPC); APIs common to several tools; distributed algorithms/protocols, such as management communication protocols; and common information formats (e.g. MIB). More recently, a form of integrated management, *web-based management*, emerged and became quite fashionable. It takes advantage of web technologies to implement lightweight client-server models (*see 3-tier Client-server Architectures* in Chapter 1). The form-based remote access mechanisms improve tool interface integration around the HTTP-HTML panoply, and allow lightweight, highly location-independent access.

23.6 DECENTRALIZED MANAGEMENT MODEL

The integrated management model is technically distributed, but still organizationally centralized— in philosophy and operation. This situation conflicts with the increasing integration of distributed applications in geographically distributed enterprises, and across enterprises, in enterprise networks. In these scenarios, chances are that organizations wish to retain a certain autonomy in managing their facilities, while still contributing to the operation of the whole large scale distributed system, in sort of a federated way.

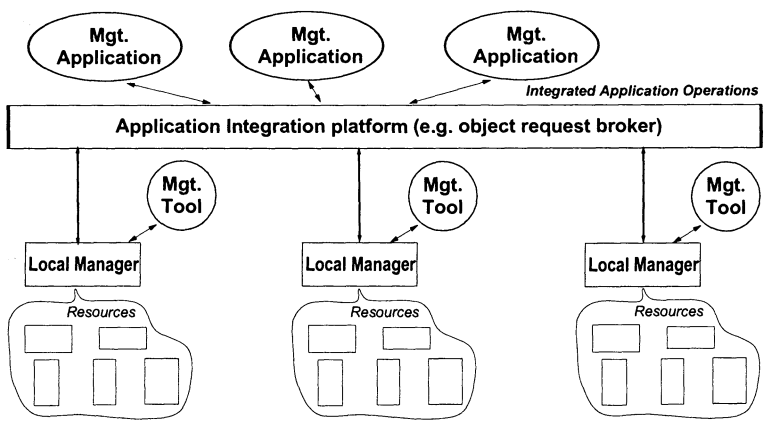

Figure 23.7. Decentralized Management Model

The decentralized management model is based on two premises: distributed and decentralized tactical management; more or less decentralized strategical management. The model is depicted in Figure 23.7. Local subsystems have local tactical management, with the necessary tools and/or platforms. Coordination is achieved through an integration platform. Given the structure of this model, the level of abstraction of the integration platform is necessarily higher than that of the integrated management model. For example, it could be at the level of an object request broker through which high-level applications dialogue with the local platforms, gathering information and providing high-level directives. Such as with political models, strategic management decentralization assumes several degrees, from confederated models to federated models. That is, the degree of freedom of local management, or in opposite terms, the degree of control exerted by the applications on local management platforms, may vary.

The **agent** paradigm has assumed great importance in the development of decentralized management models. Mobile agents have been used for building decentralized management architectures, exemplified in several works: the Java-based mobile intelligent agent framework discussed in (Zhang and Covaci, 1997), or by the 'mobile network manager' concept implemented by the MAGENTA environment (Sahai and Morin, 1998). Using the principle of delegation already discussed in Section 23.3, managers commit specific functions to mobile agents that execute them in remote parts of the system. Agents may even itinerate through the system to perform their function. Mobile agents partake the same security concerns that have already been pointed out to Java (McGraw and Felten, 1997; Garfinkel and Spafford, 1997). However, if these problems are adequately addressed, mobile agents may become the main paradigm for decentralized systems management.

Note that we have described the several generic models in an evolving way. Arriving at decentralized management through integrated management, even if only conceptually, considerably helps the understanding of the problems of distribution and decentralization, and the important difference between the two that we have always emphasized throughout this part of the book. Next we describe some models in particular, such as OSI, ODP, monitoring, and domains.

23.7 OSI MANAGEMENT MODEL

In relation to our generic model, the OSI Systems Management model is largely based on the manager/agent duality. This duality is exclusive, that is, a manager cannot interact directly with the objects subtended by an agent. Managed objects are external representations of the devices (hardware or software) they manage, and as such the conceptual functional and management interfaces do not always co-reside.

The management information representation (MIB formats, etc.) is specified in the standard Structure of Management Information (SMI) (ISO10165, 1992). OSI foresees several of the management functions we have discussed in the last

chapter, namely, configuration, faults and alarms, monitoring and event management, log control, performance, accounting, and security. These are specified under standard Systems Management Functions (SMF) (ISO10164, 1992). The architecture is depicted in Figure 23.8, and relies on the full OSI communication stack up to Layer 7, including accessory Layer 7 services such as ACSE (Association Control Service Element - ISO 8649/8650) and ROSE (Remote Operations Service Element - ISO 9072-1/2). There are essentially three categories of management entities: layer management, common management information; system management applications.

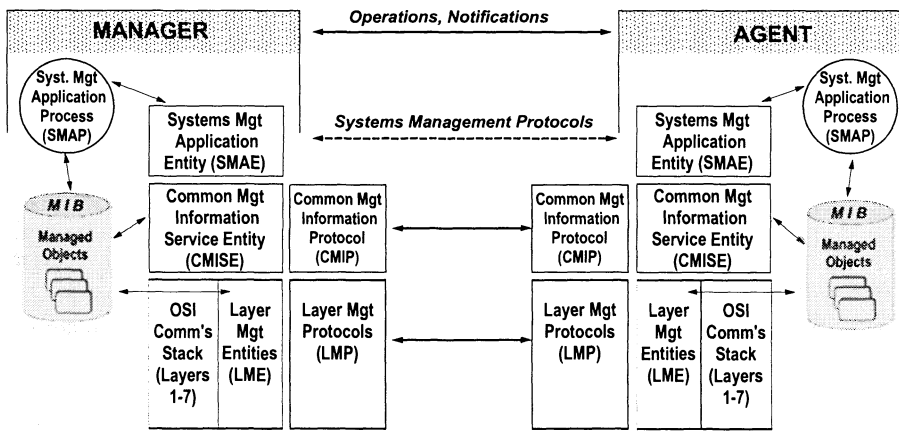

Figure 23.8. OSI Management Model

The OSI model intends the communication layers to have tactical management capability. These low-level functions are simple and autonomous, and normally do not require high-level intervention. They are typical of real-time management functions that can be automated and need be performed responsively. For example, reconfiguring the medium after a ring fault in a token ring network, recovering the token after a loss in a token bus, or rerouting in a wide-area network on account of a link failure. These functions are performed by the Layer Management Entities *(LME)*, through appropriate Layer Management Protocols at each of the lower OSI layers.

Next, the Common Management Information Service Entities *(CMISE)* define the interaction types between managers and agents. They are based on the connection-oriented remote operation paradigm (request-response), although certain responses are optional. These interactions are performed by the Common Management Information Protocol *(CMIP)*, which interprets for OSI the management communication protocol foreseen in the generic model. CMIP resorts to ACSE to establish an association (an application-level connection in OSI), and then to ROSE for the request/reply interactions. The main primitive classes supported by CMISE are: *association*, *operation*, *notification*. All begins with establishing an association with a remote managed host. Then

several operations can be issued, such as `create` and `delete`, respectively to create a new managed object, or delete it, or to access attributes, such as `get` to request the value of an attribute, or `set`, to set its value. Managed hosts may issue unsolicied notifications with `event-reports`.

The CMISE primitives are essentially a template for primitive operations on the MIB objects, which may be combined into more complex sequences. These complex operations are requested by Systems Management Applications, supported on Systems Management Application Entities (SMAE). The management applications run cooperatively in Systems Management Application Processes *(SMAP)*, that is, between a *Manager SMAP* and *Agent SMAPs*, as depicted in Figure 23.8.

23.8 ODP MANAGEMENT MODEL

We saw that under the OSI philosophy, the managed objects are entities external to the components they represent. If the component is hardware, this is normal. However, if it is for example a protocol process, it would not be necessary, and introduces coherence problems that might be avoided if the managed object state resided with the component itself. This is the approach of the ODP management model, which relies on the fact that all components are objects.

In terms of our generic management model, the manager and the agents— if they exist— are fully-fledged objects. Managed objects comprise both the functional and the management interfaces, such that the management functionality is part of the object, unlike the OSI model. Objects can be arranged in a manager/managed-object hierarchy according to the structure and scale of the system.

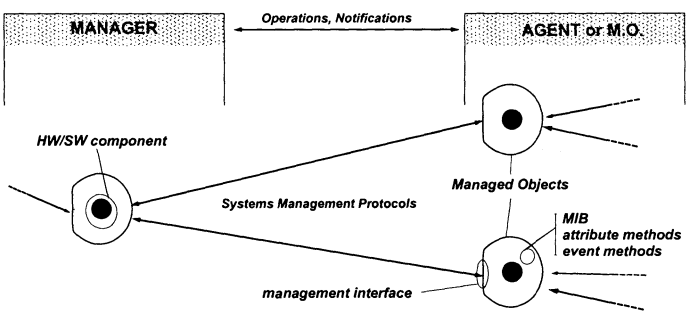

Figure 23.9. ODP Management Model

A simplified view of the ODP management model is presented in Figure 23.9. Note that components are wrapped with all the necessary functionality to manage and be managed, including the management interface, the methods pertaining to the management functions that we studied, and the state in the form of MIB elements. This configuration accounts for the use of the term *self-managed object* in ODP.

23.9 MONITORING MODEL

Monitoring is horizontal to many management functions, and it may be complex enough to be structured as a subsystem in a management architecture. A fairly complete model of monitoring (Mansouri-Samani and Sloman, 1994) is presented in Figure 23.10:

- **acquisition**– the monitor acquires information from the system being monitored: state reports; event reports
- **processing**– the raw information is treated
- **dissemination**– treated information is sent to other units
- **presentation**– information is presented at the management consoles

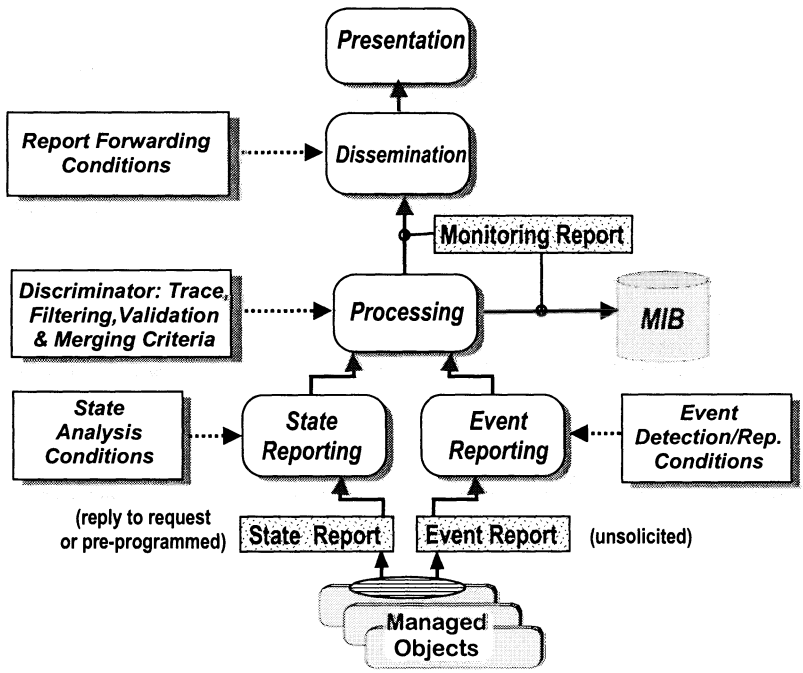

Figure 23.10. A Model of Monitoring

Sampling and polling are parameterized by pre-defined conditions. Event detection and reporting conditions are also parameterized. Several conditions influence event detection, such as: intrusiveness, the measure in which the sensor disturbs the managed object; location, whether detection is internal to the managed object, or external, e.g., by sampling; promptness, whether the event is detected immediately it takes place, or in a deferred manner, by record analysis. This raw information is often post-processed, since it is sometimes necessary to obtain the information in other forms. This involves several kinds of functions, such as: generating composite variables, such as means, rates or

counters; building histograms; filtering glitches out; validating, such as checking bounds; merging data from different sources. Processing takes place under several classes of procedures, such as: construction of attribute traces during pre-defined intervals, merging of different traces, validation and filtering of results, updating of the MIB.

One of the functions of monitoring is to generate *reports*:

- **event reports** – such as the depassing of thresholds, or occurrence of errors
- **state reports** – amount of memory allocated, current network connections
- **solicited (or on-request) reports** – observation and report generation of state variable samples triggered by the manager at pre-defined instants
- **unsolicited (or on-demand) reports** – event report generated by the managed object, triggers the observation at the adequate moment, under several event detection conditions

In a distributed system, the treated information is disseminated between managers, and presented, normally in a graphical way (histogram bars, graphics, charts, meters and counters, and so forth), to the operator.

23.10 DOMAINS MODEL

The concept of domain has been introduced earlier. Let us look at domains as a model for structuring managed objects (Sloman and Twidle, 1994). Domains help coping not only with scale, but also with heterogeneity, both of resources and of their management policies. Domains can establish *indirection*, *grouping*, and *hierarchy*. Domains *do not* establish encapsulation. Indirection derives from the way objects are referenced. Each member object has a unique identifier, which provides an indirection to it. Objects are grouped in a domain. The *object set* (or policy set) of a domain is the identification of the group of objects belonging to that domain. The object set can be referred to collectively, through a group identifier. An object can belong to more than one domain. Finally, domains accept hierarchy: a domain may be member of another domain.

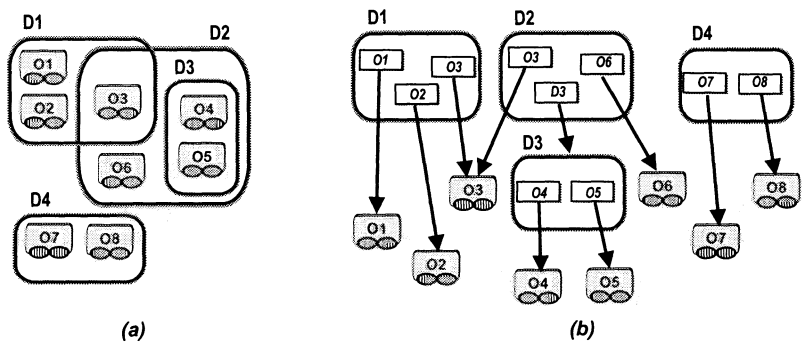

Figure 23.11. Domains Model: (a) user view; (b) implementation view

What was said above suggests the three possible relations between domains: A *disjoint* from B (distinct domains); A *overlapping* with B (objects in more than one domain); A *contained* in B (hierarchical domains). An example of a collection of objects defined in terms of domains is given in Figure 23.11. Figure 23.11a provides the user view. It shows that there exist four domains $D1$ through $D4$. The structure was chosen to emphasize the relations we have just mentioned. Note that $D4$ is disjoint from any other. $O3$ belongs simultaneously to $D1$ and $D2$, so the latter overlap. $D3$ is contained in $D2$. Figure 23.11b provides the implementation view. Observe that objects preserve whatever status they have in the system: there is no encapsulation, the domains data structure "points" to objets instead.

23.10.1 Policy and Role Based Management

Domains offer an adequate basis for establishing management policies. Policy-based management has been gaining importance in the measure that systems grow bigger and more complex. Policies establish relations between subjects and objects, or between managers and managed objects, with a view of specifying implementation-independent behaviors, such as authorization, and obligation. Policies have been used recently to specify security, and QoS, to name a couple.

Policies enable the application of role theory to management. Role-based management consists of the definition of roles in an organization, and of obligation and authorization policies to specify role relationships and interactions (e.g., client-server, producer-consumer, peer-to-peer). An application of the domains paradigm to role-based enterprise management can be found in (Lupu and Sloman, 1997).

23.11 SUMMARY AND FURTHER READING

In this chapter, we discussed the main models of distributed systems management. The first objective of the chapter was to provide insight to the systems architect on the main strategies and frameworks available. The second objective was to discuss the main management models in a problem-oriented manner, establishing links, whenever possible, to the paradigms learned in the previous chapter. The models were purposely described in an evolving way, trying to clarify in the mind of the reader the sometimes subtle issues at stake in the checkboard of management: distribution and decentralization, policy and politics, scale and heterogeneity, interconnectivity and interdependency, etc. For further reading, please see (Tschichholz et al., 1996). A recent survey on distributed systems management can be found in (Martin-Flatin et al., 1999). Management policies are discussed in (Wies, 1994; Koch and Kramer, 1995; Lupu and Sloman, 1997). More recently, mobile and intelligent agent technologies have been proposed for distributed systems management (Magedanz and Eckardt, 1996; Zhang and Covaci, 1997). An excellent survey on, and evaluation of, mobile code approaches in management can be found

in (Baldi et al., 1997). Fault-tolerant systems management in the scope of ODP is discussed in (Powell, 1991).

In (Hegering and Abeck, 1994) a detailed discussion is found on integrated management, as well as interesting notions on strategical management issues. A detailed study of the monitoring model is given by Mansouri-Samani and Sloman in (Mansouri-Samani and Sloman, 1994). The OSI Management Model is addressed with detail in both of the above-mentioned works. For the ODP model in general, see (ODP, 1987). Fusion between the ISO stack model and the ODP object model has been tried by several research groups (Deri and Ban, 1997; Znaty et al., 1995; Banker and Mellquist, 1995; Mazumdar, 1996). ISO has endorsed the 'domains' concept in management, and it is used in several documents, e.g., in (ISO10040, 1992). A detailed specification of a domains model was produced in the scope of project DOMINO, a revised version of which can be found in (Sloman and Twidle, 1994).

The Telecommunications Management Network (TMN) model addresses the management of telecommunication networks (Aidarous and Plevyak, 1998). These networks have evolved from essentially dumb copper infrastructures to fully-fledged distributed systems (Znaty and Hubaux, 1997). This evolution is related with the Telecommunications Intelligent Network Architecture (TINA) framework. Material on this subject can be found in (Sloman, 1994; Dupuy et al., 1995).

Enterprise management has a connection with systems management, which involves managing systems and networks from the perspective of the company organization and of human cooperation. It is a relevant subject for a deeper approach into managing large-scale corporate systems, and some notes on the subject can be found in (Daneshgar and Ray, 1997).

24 MANAGEMENT SYSTEMS AND PLATFORMS

This chapter gives examples of management systems and platforms. Namely, we discuss ISO (CMISE/CMIP) and Internet (SNMP) management services and protocols, standard MIBs, management tools and platforms, and the Distributed Management Environment (DME). We finish the chapter with a presentation of several tools specifically addressing security management. In each section of the chapter, we will mention several examples in a summarized form, and then will describe one or two of the most relevant in detail. Table 24.6 at the end of the chapter gives a few URL pointers to where information about most of these systems can be found. The table also points to the IETF Request for Comments, ISO and ITU sites, where any cited standards can also be found.

24.1 CMISE/CMIP: ISO MANAGEMENT

Recall, as introduced in Section 23.7, that ISO management is based on a set of services, CMISE, Common Management Information Services (CMISE, 1988), and that those services are implemented by a management communication protocol, CMIP, Common Management Information Protocol (CMIP, 1988). CMISE is organized around three classes of services: association, operation, and notification. The main primitive services of CMISE are:

P. Verissimo et al. *Distributed Systems for System Architects*

common management association	o `m-initialize`, `m-terminate`, and `m-abort`, respectively to start, terminate or abort a management connection
management operation	o `m-get` and `m-cancel-get`, respectively to request the value of an attribute, and cancel that request while pending; o `set`, to set the value of an attribute; o `action`, to invoke a specific action on a managed object, for example, request head cleaning of a printer; o `create` and `delete`, respectively to create a new managed object, or delete it
management notification	o `m-event-report`, to issue agent-to-manager notifications

According to the specific management needs, managed hosts may implement several types of management *associations*: event; event/monitor; monitor/control; full manager/agent. The *event* association is used for example when we want a simple device to send nothing but notifications to a manager host, using only management notification services. The *event/monitor* association allows the manager to read (monitor) the status of devices, in addition to receiving events. The *monitor/control* association is typically used for configuration of a system. The *full manager/agent* association is obviously used when we wish to impose no restrictions on the operations between two hosts.

event association	hosts only send `m-event-report` messages
event/monitor association	additionally uses operation service `m-get`
monitor/control association	uses all management operation services, but does not use management notification services
full manager/agent association	implements all CMISE services enumerated above

As we explained in Section 23.7, CMIP interprets requests from CMISE, and uses the ACSE and ROSE protocols to reach remote hosts. There is a fairly straighforward mapping between the management *operation* and *notification* CMISE services and CMIP primitives or *protocol data unit* types (PDUs). This relationship is shown in Table 24.1. For a notification, CMIP entities exchange `m-Event-Report`, and then `m-Event-Report-Cfm` as a confirmation. Getting data is done by `m-Get` primitives. In response, one or more `m-Linked-Reply` PDUs are returned with the requested data. The interaction may be canceled while pending by an `m-Cancel-Get-Cfm` PDU. Setting attributes is done with `m-Set`, and confirmed with `m-Set-Cfm`. Actions are triggered by `m-Action` PDUs. They may be confirmed with `m-Action-cfm`, or with `m-Linked-Reply`, when a result must be returned.

Table 24.1. CMISE services and CMIP PDUs

Interaction	Service	Protocol (CMIP)
notification	M-EVENT-REPORT	m-Event-Report, m-Event-Report-Cfm
get data	M-GET	m-Get, m-Linked-Reply
cancel get	M-CANCEL-GET	m-Cancel-Get-Cfm
set data	M-SET	m-Set, m-Set-Cfm, m-Linked-Reply
action	M-ACTION	m-Action, m-Action-cfm, m-Linked-Reply
create	M-CREATE	m-Create
delete	M-DELETE	m-Delete

24.2 SNMP: INTERNET MANAGEMENT

Simpler, but less powerful and versatile than CMISE/CMIP, the implementation of the Simple Network Management Protocol or SNMP (RFC1157), is on the other hand more straightforward, and has spawned its success in the Internet world and not only (*see* also RFC1155, the Internet Structure of Management Information – SMI). SNMP has evolved since its inception, through SNMPv2 (*see* mainly RFC1901, RFC1902) and is currently SNMPv3 (*see* mainly RFC2271, RFC2274). SNMPv2 introduces a more elaborate structure for the SMI. Whereas SNMPv1 only supported TCP/IP, SNMPv2 is multiprotocol: IP, Appletalk, Novell IPX, ISO Connectionless Network Protocol (CLNP). Whereas in the CMISE framework managed hosts or agents may be asked to perform complex actions, in SNMP agent-side interactions are mostly reduced to reading and writing attributes. The SNMPv2 workplan addressed security aspects, but only with SNMPv3 were these finally addressed, in 1998 (Stallings, 1998). The main evolution brought by SNMPv3 was to establish a common framework for incorporating security in all SNMP versions. More recently, agent technology was introduced in SNMP, through the Agent Extensibility (AgentX) Protocol (RFC2257), which allows communication between master agents and subagents.

With relation to the generic model presented in Section 23.3, the SNMP management model is based on the manager/agent relation, the *SNMP station*, and the *SNMP agent*. It also supports proxies. SNMPv2 supports hierarchy, in the form of intermediate manager/agents, or *mid-level managers*. Typically, agents reside in managed nodes (PCs, WSs, network printers, routers, and so forth), from where they control the devices they subtend. SNMP services and PDUs are shown in Table 24.2. Note that there are no actions defined, as we already had mentioned. SNMP only allows simple reading and writing of attributes (`Get-req` and `Set-req`), and notifications from agents (`Trap`). The response to any request comes in the form of a single response PDU, `Resp`. When getting a structure, such as a table, `Get-next-req` PDUs are used after the first `Get-req` PDU, returning the next element of the structure in order. `Inform-req` and `Get-bulk-req` were introduced in SNMPv2. `Inform-req` supports unsolicited

Table 24.2. SNMP services and PDUs

Interaction	Protocol (SNMP)
notification	Trap
get data	Get-req, Get-next-req, Resp
set data	Set-req, Resp
information	Inform-req, Resp
get bulk data	Get-bulk-req, Resp

manager-to-manager communication, to exchange and/or disseminate information about management operations. `Get-bulk-req` optimizes the reading of large amounts of data, which was awkwardly done in SNMPv1 with a stream of `Get-next-req`. `Get-bulk-req` asks for the next n values, instead of just one.

While SNMPv1 lack of security made it possible for anyone to act as a manager and change managed objects at will in any system to which he had physical access, SNMPv3 provides an authentication and encryption framework. Authentication uses the HMAC protocol (RFC2104), with a choice of MD5 or SHA as the hash function (RFC2202). Encryption is performed by the DES protocol in CBC mode. SNMPv3 establishes a security policy by defining an attack model (akin to a fault model in fault tolerance). SNMPv3 should secure against: modification of information– changing in-transit message parameters; masquerading– impersonating an authorized manager in requesting operations; message stream modification– reordering or replaying messages; disclosure– unauthorized read of exchanged management information. It *does not* secure against: denial of service; traffic analysis. For notes on the mentioned protocols, the reader is advised to *see Using Cryptographic Protocols* in Chapter 18.

24.3 STANDARD MIBS

We have seen that standardization of the information representation is crucial for the interoperability of applications and protocols in different hosts and devices in a system. For example, for the Internet we have the Structure of Management Information, SMI (RFC1155 or RFC1442 for SNMP version 2), and detailed standards for the several relevant MIBs. As examples, we have the Internet MIB-II (RFC1213, RFC1450 for SNMP v.2), the Bridge MIB (RFC1493), or the Remote Network Monitoring RMON MIB, versions 1 and 2 (RFC1757, RFC2021).

Table 24.3 presents a few of the data types defined in the Internet SMI standard. MIB standards further define the specific contents of the data structures associated with the entities relevant to management. For example, there will be MIB definitions for TCP, IP, routers, bridges, network adapters, etc. Manufacturers will have "space" asigned in the MIB conceptual structure to insert data

Table 24.3. Example Abstract Data Types defined in SMI

Type	Description
IpAddress	an IP address
Counter	nonnegative increasing integer $O(2^{32})$
Gauge	nonnegative floating integer $O(2^{32})$
TimeTicks	counter in $10ms$ increments
Opaque	unformatted text

about their line of products. A MIB is organized hierarchically, so that MIB searches can be systematized easily as MIB graph traversals, and optimized using graph theory. At the time of this writing, the current effort is towards standardization of a global, framework independent MIB, where Internet, ISO, ITU, all fit. MIB objects are generically defined in ASN.1.

MIBs are divided in *groups*. For example, MIB-II features the following groups: `system`; `interfaces`; `ip`; `icmp`; `tcp`; `udp`; `egp`; `transmission`; `snmp`. The names are self-explanatory for the most part. The `system` group represents the system where the entity resides. The `interface` group represents each specific interface of a network device. ICMP is the control message protocol for TCP/IP, EGP (Exterior Gateway Protocol) is the IP routing protocol between autonomous systems, and the respective groups concern the relevant variables. The `transmission` group represents the physical media entities.

In each group, the necessary data objects are defined, obeying to the data types specified in the SMI. These data types correspond to attributes of one or more entities, and are specified according to the several management function needs. For example, Table 24.4 presents a few of the MIB-II `tcp` group objects, for configuration and for performance management. Other objects exist, in these and in other groups, for accounting management, fault management, and so forth.

One problem haunting management platforms is the network load caused by polling remote devices, specially in large-scale systems. An interesting standard addressing this problem is the RMON MIB, Remote Network Monitoring. The underlying idea is the concept of *remote monitoring device*. Such a device will supposedly assist network management, by gathering data in remote places, and making it accessible to the managing nodes. The RMON 1 standard is Ethernet based. Later, it was extended to the higher layers, 3 through 7, in RMON 2. This greatly increased the efficiency and accuracy of the remote monitoring device, also called *probe*. For example, it can track the traffic of a web-based application between two specific hosts. RMON devices can be either dedicated, or live as software modules in functional devices, such as hubs, bridges, routers, or PCs. However, the performance of the normal functions of host devices may be affected by RMON monitoring. For example, PC-based

Table 24.4. Example tcp group MIB Objects

Objects for Configuration Management	
Object	Description
tcpRtoMin	minimum retransmission timeout
tcpRtoMax	maximum retransmission timeout
tcpMaxConn	maximum number of open connections
tcpCurrEstab	current number of open connections
Objects for Performance Management	
Object	Description
tcpAttemptFails	number of failed connection attempts
tcpEstabResets	number of connection resets
tcpInErrs	number of reception errors
tcpOutSegs	rate of transmitted segments

monitoring of a network requires enabling promiscuous mode reception, i.e., receiving all passing frames.

An RMON device only has to implement part of the MIB it is concerned with. It can be normally off-line, and can be programmed to only contact the manager (notification) when certain conditions are met: a failure; certain thresholds passed; other conditions dictated by the manager (e.g., new host found). RMON2 allows a probe to only return the values that have changed since the last poll. The probe can also post-process data (e.g., create composite variables), and execute complex actions (e.g., host discovery), offloading work from the manager host. Finally, the probe supports multiple management platforms, that is, it can serve several managers. Incidentally, note the analogy of remote monitoring with distributed sensing, and of probes with representatives (*see Entities and Representatives* and *Input/Output* in Chapter 12).

24.4 MANAGEMENT AND CONFIGURATION TOOLS

What is required of management tools? In essence, whatever we have discussed in the previous chapters that can be performed in an automated fashion. For example:

- console management
- event monitoring and management
- remote control
- current management functions (configuration, accounting, etc.)
- software distribution
- backup management
- storage management
- server pool management

- security management (protection, intrusion detection, etc.)

There are a number of tools for these several different purposes, and in general one can talk about the following classes:

- testers
- network analyzers
- management software packages
- distributed management protocol stacks
- integrated management systems
- help desk systems
- trouble ticket systems

24.4.1 Testers

The simplest tools, testers are, as the name implies, devoted to testing basic hardware functions. Hardware probes test continuity of electrical circuits and cables, or digital levels (logic probe). Signal generators serve to inject signals in circuits or cables and test their reaction. Time domain reflectometers (TDR) are sophisticated devices that test the state of transmission lines and cables carrying high frequency signals. When these lines are twisted, smashed or simply flickering, they do not propagate high frequency signals adequately (e.g. 100Mb/s Ethernet), while still passing a simple continuity test. This is mainly because spurious reflected signals develop on the cable and disturb the original signal, hence the name of the detector. Impedance meters measure impedance, another parameter that only makes sense at moderate to high frequency, and must exhibit a stable value throughout the whole link between two nodes. Electrical field meters measure radiated power (e.g., strength, direction), for wireless communication.

24.4.2 Network Analyzers and Monitoring Tools

One level of abstraction up we have protocol signalling. Network (or protocol) analyzers can follow *all* the communication between two (or more) parties, and dissect the execution of a given protocol. Industrial network analyzers are normally dedicated machines with real-time operating systems and fast communication adapters that work in promiscuous mode, listening to everything on a medium. These machines normally have powerful filtering functions that allow pinpointing: packet types; origin and/or destination; contents; dialogues, etc. They are also capable of detecting errors in the protocol execution. Modern network analyzers are multi-protocol, i.e., they understand several of the major protocols. It is also possible to configure a software-based network analyzer, by using an adequately powerful machine (e.g., a high-end PC) and a software package, normally comprised of a modified network driver (e.g., Ethernet) that works in promiscuous mode, and a filtering, processing and rendering module.

Several of these analyzers for local area networks, also known as sniffers, are discussed later in this chapter (*see* Section 24.7).

Two of the main attributes of analyzers and monitoring tools are non-intrusiveness, the degree in which the tool does not disturb the system under observation, and precision, the degree in which the measurements correspond to the magnitudes being measured. Monitoring tools with hardware sensors exhibit better precision and smaller intrusiveness. On the other hand, they provide significant amounts of raw information that must be processed. Software sensors are implemented through the instrumentation of code. They exhibit moderate precision and are significantly intrusive. However, they address high level information (e.g., a certain step of the code), and can act at very specific points to gather information. An example passive, software-based performance measurement tool is described in (Malan and Jahanian, 1998). Hybrid systems take the best of both worlds, by using low-level instrumentation in hardware, hooked to software sensors. Consider the following example problem, complicated enough to deserve some hybrid tooling:

> *"Accurately measuring the interval between the reception instants of the first frame after a hardware trigger, at two network adapters in the same LAN"*

That is, there is a trigger signal, after which we want to catch the first frame on the LAN, and measure the interval between the reception of that frame at two different network adapters. Note that this interval is of the order of the microseconds, and depends on the difference in network propagation delay and in reception processing speed at each adapter. As a hybrid solution, we make a hardware sensor by hooking the hardware trigger signal to a hardware interrupt line. Then, we build a software sensor activated by the trigger sensor. The software sensor is composed of monitoring code that waits for the next received frame, and produces an interrupt when that happens. The interrupt handling code activates a hardware actuator, for example by flipping one of the control pins of an unused RS-232C serial line connector. All these steps are done at both hosts. By measuring the interval between the pin flips at the two hosts with a simple external device such as an oscilloscope or an interval meter, we have an extremely accurate monitoring tool with relatively little hardware apparatus.

Amongst the existing monitoring and analyzing tools, we emphasize a few easily available ones. The *SNMP MIB Browser*, based on the WWW, is developed in the Technical University of Braunschweig (Germany). It is a simple CGI script written in the Tool Command Language (Tcl), which uses the Tnm Tcl extension for network management applications, and allows browsing SNMP MIB contents in a structured away. *Beholder*, or BTNG, is an RMON compliant Ethernet network monitor developed at the Technical University of Delft (Netherlands), which can be remotely queried by means of SNMP. Beholder is accompanied by the Tricklet package: a set of SNMP utilities for OS/2 and UNIX. *Ethereal* is a network protocol analyzer for Unix, in the context of the GNU-GPL effort. It examines both real-time network data and data from a capture file on disk. It allows a user to browse the data, viewing packets with

programmable levels of detail. Ethereal features a display filter language and the ability to view the ASCII contents of a TCP connection.

Multi Router Traffic Grapher, or MRTG, is a tool developed at the Swiss Federal Institute of Technology (Switzerland), to monitor the traffic load on network links, as seen for example through routers. MRTG generates HTML pages containing GIF images of the traffic. *RRDtool* from the same institution builds on MRTG to provide fast round-robin database storage and display of time-dependent data (i.e. network bandwidth, machine-room temperature, server load average).

META is an interesting example of distributed monitoring tool is META (Wood, 1991), developed in Cornell University (USA). META runs on UNIX. It is distributed, and has a neat layered structure. Its functional part is a rule-based system written in a special language, Lomita. META not only monitors but can also control the managed system. For that, it features a sensor/actuator layer that deals with the infrastructure itself. This layer conceals the specific characteristics of the devices, and on top of it an abstract data model of the system is constructed, on whose state the rules of the policy layer are applied. Its data model is implemented on an active real-time database, capable of: triggering actions based on modifications of the state of the database (e.g., **when** condition **do** action); and expressing conditions depending on time durations (e.g., X **until** Y).

24.4.3 Management Software Packages and Protocols

The core of modern management systems are specialized software packages that implement specific management functions. Remote operation is possible if those packages can talk with devices through a common protocol. This is where management communication protocol stacks fit, such as CMIP or SNMP, which should be supported by both managing and managed hosts. On top of these stacks, management processes including the above-mentioned software packages dialogue. Software packages can be used to build tools that perform one or several related functions, such as configuration, or performance management (Zeltserman and Puoplo, 1998). These tools normally reside in the managing host(s), or manager(s), and talk remotely to the other (managed) hosts or devices. For example, *Zebra* is a free routing software (GNU Generic Public License or GPL) package that manages TCP/IP based routing protocols. It supports Border Gateway Protocol, BGP-4 (RFC1771) as well as RIPv1, RIPv2 and OSPFv2. Zebra software, unlike traditional, Gated based, monolithic architectures.

Tcl Extensions for Network Management Applications, or *Scotty*, is a software package developed at the Technical University of Braunschweig (Germany), which simplifies the implementation of portable, script-based, specific network management functions. Scotty is based on the scripting language Tcl, and has two main components. The first one is the Tnm Tcl Extension which provides access to network management information sources. The second com-

ponent is the Tkined network editor which provides a framework for constructing an extensible network management system.

24.4.4 Application Configuration and Construction

Tools and environments for system and application configuration programming have emerged in the past few years. *Regis* is a programming environment for constructing distributed programs and systems, developed in the University of London, Imperial College (UK). Structural aspects of distributed programs—which are orthogonal to its algorithmics— are expressed in a configuration language. Regis is based on the architectural configuration language Darwin. Other aspects, such as inter-process communication, are orthogonal to the application structure. Programmers can create new communication classes (not necessarily RPC) independently from the program and the interaction style. Regis is assisted by companion software, such as the Software Architect's Assistant, and a Tcl/Tk Graph Widget. *AAA*, "Agents Anytime Anywhere", is an agent development environment, created at INRIA (France). It is materialized as a Java agent-based platform, using the message-bus paradigm. It supports the configuration and development of modular and configurable applications. The *TACOMA* project focuses on operating system support for agents. TACOMA is a collaboration between the University of Tromsø (Norway), Cornell University (USA) and the University of California San Diego (USA). An agent in TACOMA can be installed and executed on a remote computer, and may explicitly migrate to other hosts in the network during execution. Tacoma currently features a web agent that can be used to construct management tools, for example, for remote monitoring. A striking example is StormCast, a wide-area network weather and environmental monitoring application accessible over the internet.

24.4.5 Integrated Management Systems and Applications

When several subsystems, even heterogeneous, fall under the realm of a single management system, we say we have an integrated management system (IMS). Several freeware and commercial products fall into this designation. Current, significant IMS are built according to standardized frameworks, such as those originating from ISO or IETF (CMISE/CMIP or SNMP/RMON), which are manufacturer-independent and support open systems. They are often complemented with powerful database management, and versatile graphical interfaces.

Besides the many commercial IMSs, there are a few freely available ones. *LANdb* provides network managers a means of cataloging all connections, closets, and network hardware on a network. It uses scripting and database query languages in order to provide an efficient web-based frontend to a complete network management package. LANdb is distributed under the GNU-GPL. *MibMaster* is intended to be used on any SNMP-compliant network environment, MIB Master allows performing web-based management. SNMP agents can be viewed and/or modified using any Web browser. *UTopia* is an ATM

management tool developed at the University of Twente that helps managers to configure their ATM switches. UTopia is web-based. The web client part is implemented as a JAVA applet.

GxSNMP is a GNU-GPL network management application developed in the context of the GNOME project. GNOME is the GNU Network Object Model Environment. The GNOME project intends to build a complete, easy-to-use desktop environment and application development framework.

SMB-SNMP is a management application based on MSFT Windows, developed at the University of Pisa (Italy). It maps SNMP MIBs as files of a Windows or WNT directory. MIB variables are represented as text ot HTML files. It uses the Samba file system to transparently map file operations on SNMP primitives. Whenever a variable is read/modified an SNMP get/set is issued to the remote SNMP agent. SNMP resources can thus be managed without any specialized software: the files can be edited by any normal editor.

24.4.6 Help Desks and Trouble Ticket Systems

Open systems, beyond a certain scale, require help desk functionality, normally provided through a center that ensures information providing and first-line assistance to users. A management help desk is little different from other kinds of help desks. It may be either public or private, e.g., fully open or confined to employees of an enterprise. Several media can be used, sometimes in alternative or complement, to serve the user. Telephone is the most obvious and the one with greater promptness. Fax or email provide a means for deferred attention, which can reduce the costs of a telephone interaction. More recently, web-based interfaces started proliferating. This kind of interface offers a potential for multimedia not available in the other means.

The help desk can have have a flat structure for simple systems (and simple problems). Otherwise, in a large/complex system, it may be structured by hierarchy and specialization. That is, personnel is stratified by expertise, and that expertise may be divided between groups. Calls arrive at the front-line, less expert and more generalist personnel, and may go up the ladder should that be required. Routing of requests may be semi- or fully-automated.

The main requirements to be satisfied by a help desk system are:

- interactive access to management information (if possible, concerning the function or device being asked about)
- interactive diagnosis guide (set-by-step methodology for arriving at the root of the problem)
- frequently asked questions (FAQ) manual (both for user and manager)
- knowledge base (a "FAQ" with sophisticated search methods)
- trouble ticket system (a complement to the help desk function, to track problems that remain unsolved for a while)

Some requests addressed to the help desk derive from malfunctions or other kinds of problems that require further attention. There are several reasons

for organizing the response to these anomalies: to log requests, for legal and administrative reasons; to sequence requests, eventually prioritizing them; to classify problems, creating a typology for them; to create technical memory in the system, for frequent or recurrent events. This is done through a Trouble Ticket System.

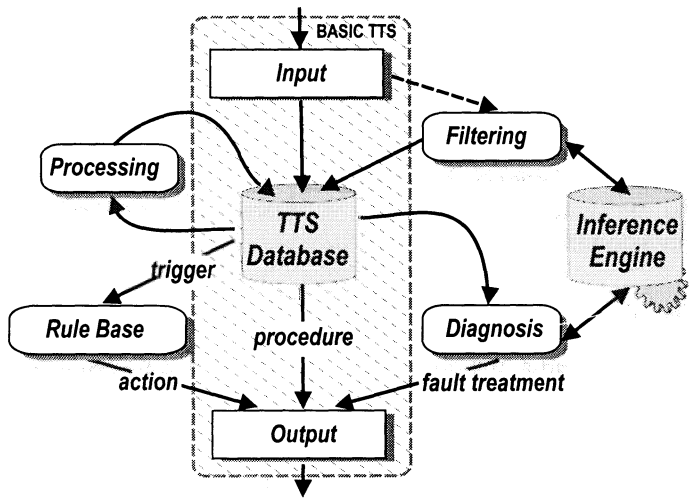

Figure 24.1. Structure of a Trouble Ticket System

A TTS is mainly an anomaly record and response system. It receives events, stores and processes them, and prepares the adequate response. Given the amount and kind of information it gathers, a TTS can be integrated with fault management functions to enhance the latter (Lewis and Dreo, 1993). TTS events can be generated by users, but they can also be generated automatically, by devices. After being processed, they originate a *trouble ticket (TT)*. The notification primitive that we studied in the managed object model can be used to send events to a TTS. The structure of a TTS is depicted in Figure 24.1, and has the following main blocks: input and output subsystems for interface with the outside (users, infrastructure, and managers); database for storing all necessary information (e.g., TTs, historical records); filtering and processing modules for the main TTS functions (e.g., housekeeping, filtering irrelevant requests); rule base and diagnosis modules for automating some of the TTS functions (e.g., automatic generation of actions upon certain triggers, fault diagnosis based on previous knowledge processed by an inference engine).

Not all TTS are as sophisticated as the model given in Figure 24.1, however, a basic TTS has at least the functionality included in the dashed part. A TTS in action is briefly as follows. The notification event is received, and filtered. The problem is logged with all the necessary information, in what makes a trouble ticket (TT): origin and location of the fault, nature of the fault, possible cause, timestamp, contact person. The TT is classified, against ordering, type and

priority, and then dispatched to a manager on duty. If the TTS allows data mining, the historical records are searched for a match with a similar problem. A *current problem log* is also opened, and will be maintained until the problem is solved, with all relevant information, such as further findings or steps taken to remedy. Automated instrumentation (management tools) may be used to diagnose the problem better and attempt to correct it. If the problem persists or is too complex, the TT is routed to a more senior manager. In background, both the managed system and the TTS operation should be quality controlled: Are there many problems? Are we solving most of them? Are we solving them well?

24.5 MANAGEMENT PLATFORMS

What is a management platform? We have seen that management requires the assistance of several functions that are normally performed by isolated tools. A platform is a working base where the manager has access to a set of tools that act in a coordinated way, interact among themselves, and share the same raw data. Platforms are a pre-condition for integrated management, inasmuch as distributed management frameworks are a pre-condition for open platforms to work.

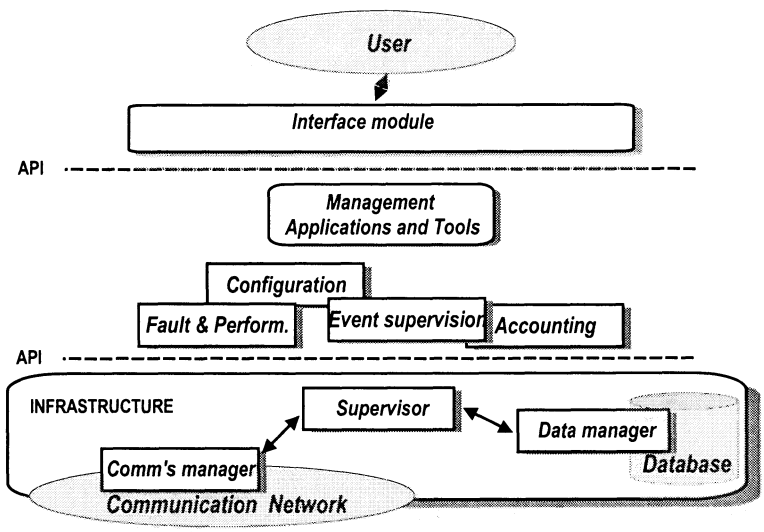

Figure 24.2. Generic Structure of a Management Platform

A generic block diagram of a platform is given in Figure 24.2. The platform is supposed to be distributed, and has essentially three levels: the user interface level, where all graphical rendering facilities lie; the managing level, where management functions are concentrated; the infrastructure level, where the device specific parts are located (managed objects, agent functionality, etc.).

Let us consider the manager host. The *management console* runs in this host, with the *interface level* software. The several software packages running the necessary management functions run at the *managing level* (most modern platforms are modular with this respect, one can install only the tools needed), provide a substrate for building management applications and tools. They communicate with the user interface through a API common to all application modules, so that display is homogeneous. They communicate with the *infrastructure level* through another API, also common to all application modules. At this level, the *supervisor* is the kernel-level part that handles and dispatches all management related requests. It interacts with the *data manager*, which controls the database (MIB) operations, and with the *communications manager*, which runs the management communication protocols, that dialogue with the other managed hosts in the system. Through this protocol, the platform can invoke operations on remote managed hosts and objects.

A device may have more or less powerful machinery for management. Let us consider a managed host, possibly subtending several simpler devices. It will only have an implementation of the infrastructural level, that will dialogue through the management communication protocols with the managing host, where the rest of the platform functionality resides. The infrastructure can have a simplified supervisor, a MIB database specific of the subtended devices, and a fully-fledged communication stack (e.g., SNMP).

The integrated management model (*see* Section 23.5) is normally materialized around management platforms. There are several commercial management platforms, amongst which Sun Solstice, IBM Tivoli, HP OpenView. The latter will be discussed with some detail.

HP OpenView The reference example of management platform is OpenView from HP. The architecture of OpenView is presented in Figure 24.3. The architecture is modular, is pretty much in line with the generic platform structure that we have just presented.

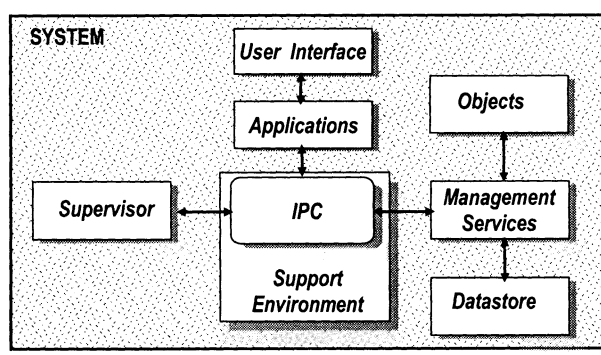

Figure 24.3. HP OpenView Management Platform

Currently available on several operating systems, OpenView was one of the first open platforms based on standards, namely ISO protocols and Internet MIBs. Although it is a proprietary system, it soon became a market reference, and was one of the main contributors to the Distributed Management Environment initiative of the OSF. OpenView is distributed, and its components are spread throughout all the managed hosts. The managing host runs the manager console, and the platform server, the OpenView Network Management Server. In each host one may install only the necessary modules. In Figure 24.3 we can see the several modules of OpenView:

system	the host where the modules are installed
supervisor	the kernel level manager of local resources
IPC	interprocess communication, also called *postmaster*, handles message transactions between platform modules, local or remote, through the adequate protocols (local IPC, remote operations, CMIP, SNMP, etc.)
support environment	the platform modules rely on a set of local support services, such as directories, file transfer, etc.
objects	the representation of the managed objects themselves
management services	management function modules necessary in this host
datastore	management of the data repositories
user interface	interface with a user console when necessary
applications	management applications

High-Level Platforms The management models have evolved, as we have studied in Chapter 23, and so have management platforms. Support began to emerge for the global planning and configuration aspects, and for the strategic management of large and/or loosely coupled (e.g., federated) facilities, in the form of high-level platforms. High-level platforms perform strategic support to management in a decentralized fashion, such as helping with the definition and refinement of policies. For that reason, they must be capable of integrating several platforms under their realm, such that it makes sense to call them **platforms of platforms**. One of the enabling factors for this integration to be possible is what has been denominated *platform middleware*, the set of support technologies for high level integration of applications. Object request broker, message bus and agent technologies are promising options for platform middleware. This technology maps onto the decentralized management model (*see* Section 23.6).

Examples of high-level platforms are OperationCenter from HP and Spectrum from Cabletron. Essentially, they provide an environment-independent platform, based on a virtual network machine, to which all underlying platforms are translated. The machine can thus dialogue with the tools and applications resident in the underlying platforms, and it can also interface directly the sev-

eral standard management protocols, or support new ones. Clients interface this virtual platform and run high-level applications, mostly of the strategic nature we have suggested above. Clients of high-level platforms supposedly belong to strategic management teams.

24.6 DME: DISTRIBUTED MANAGEMENT ENVIRONMENT

The Distributed Management Environment or DME is a multi-vendor distributed management platform originally developed by the Open Software Foundation (Chappell, 1992). It was based on several contributing technologies, amongst which Bull, IBM, HP and Tivoli. DME was important in that it launched the generic architectural foundations of distributed systems management platforms. DME lost momentum because of the hesitations of vendors to endorse it, since a common platform would shave any competitive advantages of their products. Namely, it ceased being supported by the now-called Open Group, who has discarded plans to port CORBA to DME as its object broker.

The success of the Web also brought lightweight, desktop-oriented, *web-based management*, a dressing of the integrated management models which changes the focus of platform development towards the desktop. Research on web-based management models is very active. Standardization is lead by the Open Group (OG) Management Program, and by the Distributed Management Task Force (DMTF) (*see* Table 24.6 for URLs). Both are consortia of companies operating in the field. Together, they are defining the object-oriented Common Information Model. The DMTF is investing on a model for vendor-independent remote management operations, has defined a Desktop Management Interface (DMI) for the purpose, and has clearly endorsed the Web-Based Enterprise Management (WBEM) model.

We can say DME fulfilled its role, in influencing a whole generation of platforms. Vendors have in most cases adopted DME concepts, and provide some of its proposed functionality in their products. For these reasons, it is worthwhile analyzing DME. Its functional structure consists of the following modules: object management framework, network management option, distributed services. The *object management framework* is the central piece, based on cooperating peer-to-peer objects, oriented to distributed systems management, on top of which are based the *distributed services.* These are infrastructure independent, and extensible.

DME is object based. Both managed objects and all functions are specified and implemented as objects, in the sense of CORBA and ODP (*see* Section 23.8). However, with regard to our generic management model, where there was a distinct hierarchy between manager and managed object or agent, these objects in DME are *cooperating*, that is, they have a peer relationship. Of course, a manager/agent relationship may be superimposed (and in fact often is) on DME objects. The architecture of DME is depicted in Figure 24.4, and consists of the following building blocks: object services; management services; management applications; management user interface (MUI); support services.

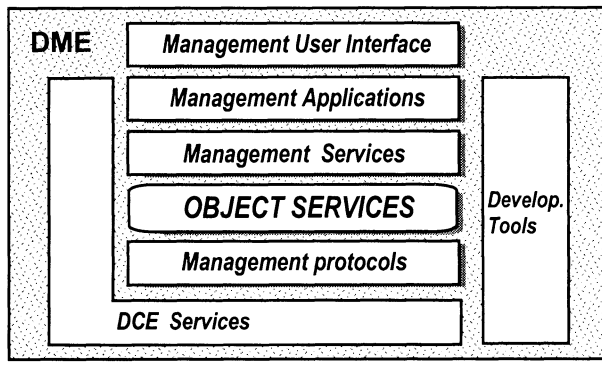

Figure 24.4. Architecture of DME

The *Object Services* are the core of DME. They are implemented by *object servers*, and also feature a notification service, the *event service*, which is crucial to implement interactions of the above-mentioned peer-to-peer nature, awkward to build with mere RPC. The object services supply the basic modules for the construction of the rest of the management services and applications.

The *Management Services* are built on top of the object services, and consist of building blocks for applications in the several system management areas.

The *Management Applications* rely both on the object services and on the management services. They interact with the user through a *Management User Interface* module. The *Support Services* consist of: DCE (*see DCE* in Chapter 4); management protocols; and development tools.

24.7 MANAGING SECURITY ON THE INTERNET

Internet protocols have their *design vulnerabilities*. In order to trace them and neutralize their effect, the *security management* of a system is of extreme importance. The functions relevant to security management are listed in Table 24.5.

First of all, in order to look for vulnerabilities we have to know about them. Then, we also might appreciate advice on how to remove them. Several institutional entities cooperatively centralize incident notices (e.g., new viruses, attacks), and cures for them (e.g., patches, releases, scripts). One relevant example is the CERT, Computer Emergency Response Team, whose URL is given in Table 24.6, as well as the URLs of most of the systems described in this chapter. The technologies cited above are freeware tools available to assist the administrator or to enhance security of installations. The insertion of this kind of technologies in a coherent management framework has been addressed in Chapter 23, where we discussed System Management strategies and tactics. In this section, we are going to briefly review the technologies themselves.

Table 24.5.

security enhancement tools	in this class we have tools that render machines and software more robust, for example cryptographic communication software, filtering and wrapping software, or packages that encrypt, sign or checksum critical software, to detect modifications; examples of such software are Tripwire, Xinetd, Tcpwrapper, Portmapper, and Cracklib
fault diagnosis tools	in this class we have for example packages that scan the facility looking for design or configuration vulnerabilities; examples of such software are Crack, COPS, Tiger, ISS, Satan, Merlin, Trojan
intrusion detection tools	in this class we have for example packages that perform real-time supervision, looking for anomalous behavior or state of the system, or abnormal patterns of usage, in order to detect intrusions; examples of such software are Scandetector, CPM, AID, AAID, NID, ASAX, Hummer
auditing tools	in this class we have for example packages that perform logging and build audit trails of the system, in order for the administrator to analyze events a posteriori e.g., correlate attacks to detect intrusion campaigns; examples of such software are Tcpdump, Analyzer, Swatch, Logdaemon, Netlog, Netman

24.7.1 Security Enhancement Tools

Tripwire is an integrity monitor tool for Unix systems. It uses message digest algorithms to checksum files and guarantee their integrity, by detecting tampering with file contents, in result of intrusions. *Xinetd* is a replacement for inetd, the internet daemon in UNIX systems. It supports access control based on the address of the remote host and the time of access. It also provides extensive logging capabilities, including server start time, remote host address, remote username, server run time, and actions requested. *Tcpwrapper* allows monitoring and controlling connections to the main inetd communication ports: `tftp`, `exec`, `ftp`, `rsh`, `telnet`, `rlogin`, `finger`, and `systat` ports. Also includes a library so that other programs can be controlled and monitored in the same fashion. *Portmapper3* is a replacement of the original `portmapper` program, known to have security flaws such as allowing anyone to read or modify its tables and forwarding any request so that it appears to come from the local system. Portmapper3 does essentially the same as Tcpwrapper, but for RPC based programs invoked by the standard `portmapper`. The Securelib shared library (eecs.nwu.edu:/pub/securelib.tar) implements access control for all kinds of (RPC) services, not just the `portmapper`. *Cracklib* is inspired by the Crack program (*see* next section). It is a library containing C functions that may be

used in a `passwd`-like program. CrackLib prevents users from choosing passwords that Crack could guess, by filtering them out at the source.

24.7.2 Fault Diagnosis Tools

Crack guesses eight-character standard Unix passwords, by standard guessing techniques. It is written to be flexible, configurable and fast. *COPS*, the Computer Oracle and Password System (COPS) package, examines a system for a number of known weaknesses and alerts the system administrator to them; in some cases it can automatically correct these problems. *Tiger* is similar to COPS, but more up to date, and easier to configure and use. It consist of system monitoring scripts that scan a Unix system looking for security problems. *ISS* is a multi-level security scanner that checks a UNIX system for a number of known security holes such as problems with sendmail, improperly configured NFS file sharing, etc. ISS can be used to probe entire network facilities. *Satan*, the System Administrator Tool for Analyzing Networks, is a network security analyzer. SATAN scans systems connected to the network, notifying about the existence of well-known, often exploited vulnerabilities. SATAN can scan the system from the outside, as hackers do, and thus provide a realistic analysis. *Courtney* monitors the network and identifies the source machines of SATAN probes/attacks, because SATAN is also used by hackers. Courtney receives input from tcpdump. If one machine connects to numerous services within a short time window, Courtney identifies that machine as a potential SATAN host. *Merlin* is a tool for managing and enhancing existing security tools. It can provide a graphical front-end to many popular tools, such as Tiger, COPS, Crack, and Tripwire. Merlin makes these tools easier to use, while at the same time extending their capabilities. *Trojan* is a trojan horse checking program. It examines a given search path and looks at all of the executables in the search path for people who can create a trojan horse that root can execute.

24.7.3 Intrusion Detection Tools

CPM Sniffer Detector, or Check Promiscuous Mode, checks a system for any network interfaces in promiscuous mode; this may indicate that an attacker has broken in and installed a sniffer program. *Scan-detector* is a tool to monitor for port scans of a Unix system. This is a frequent attack, and normally the first to be performed against a facility. Detecting port scans can give a security administrator precious early warning. *Adaptive Intrusion Detection System (AID)* is designed for network audit based monitoring of local area networks and used for investigating network and privacy oriented auditing. The system has a client-server architecture consisting of a central monitoring station and several agents (servers) on the monitored hosts. The central station hosts a manager (client) and an expert system. Heterogeneous UNIX environments are supported by having the agents produce OS independent data formats. Audit data are analyzed at the central station by a real-time expert system. Secure RPC is used for the communication between the manager and the agents. *ASAX*, Advanced

Security audit trail Analysis on uniX, is a distributed audit trail analysis system that also incorporates configuration analysis. The audit trail analysis system consists of a central master host and one or more monitored machines. The latter analyze their local audit data and send relevant events to the central host. Heterogeneity is achieved by using an O.S. independent data format. The system is rule-based, detecting known penetration patterns. *Hummer* is a distributed component for any intrusion detection system, that allows any two Internet hosts to share security information. It enables cooperative intrusion detection using data sharing between distinct sites, to counter the present threat of distributed intrusion campaigns— systemic attacks involving multiple hosts.

24.7.4 Auditing Tools

Tcpdump and *Analyzer* (formerly Windump) are packages for network monitoring and logging. Tcpdump is the best known and the most used such package. It programs the driver to be in promiscuous mode and grabs the network traffic. Analyzer is the port to Windows95 and WNT. These packages are normally assisted by analysis software, since they grab huges quantities of bulk data. *Swatch* aims at monitoring events on a large number of systems. It modifies certain programs to enhance their logging capabilities, and monitors the system logs for specific messages. *Logdaemon* is a package that provides modified versions of rshd, rlogind, ftpd, rexecd, login, and telnetd. These versions log significantly more information than the standard vendor versions, enabling better auditing of problems via the logfiles. It also includes support for the S/Key one-time password package. *Netlog* is a package that contains a TCP and UDP traffic logging system. It can be used for locating suspicious network traffic. *Netman* is a fairly complete toolbox for network monitoring and visualisation. Two of the tools provide a real-time picture of network communications, while the other provides retrospective packet analysis. These tools are designed to allow network managers to passively monitor a network and diagnose common network problems as quickly and efficiently as possible. Etherman is an X11 based tool which displays a representation of real-time Ethernet communications. Interman focusses on IP connectivity within a single segment. As with Etherman, this tool allows a real-time representation of network communications to be displayed. Packetman is a retrospective Ethernet packet analyser. This tool allows the capture and analysis of an Ethernet packet trace.

24.8 SUMMARY AND FURTHER READING

This chapter gave examples of systems and platforms for distributed systems management. We started by addressing tools and platforms and explaining their differences. We talked about testers, network analyzers, management software packages, distributed management protocol stacks, integrated management systems, help desk systems, trouble ticket systems and monitoring systems, integrated platforms. Then we discussed the two main management

frameworks, ISO CMISE/CMIP and IETF SNMP, presenting the relevant protocols, followed by a study of the main standard MIBs. Next, we addressed DME as the reference environment for distributed management platforms, and finalized by presenting a number of tools for managing Internet performance and security.

Further study is suggested on several areas. Management of, and based on, the World-Wide Web has deserved great attention, as exemplified by (Pras et al., 1997; Schonwalder and Toet, 1997; Hong et al., 1997; Thompson, 1998). High-level notations for the specification of network management functions help bridging the semantic gap between the sometimes simplistic management function standards and the often sophisticated requirements of application and tool builders (Brites et al., 1994; Pavlou et al., 1998; Hughes, 1993). Agent technology can be used to develop new generation management platforms, as discussed in (Muller, 1997).

Table 24.6 gives a few pointers to information about some of the systems described in this chapter. However, for further practical study on CMIP and SNMP, we suggest (Leinwand and Conroy, 1996) or (Stallings, 1999), besides the standards and RFC documents themselves. The Distributed Management Environment (DME) is further treated by Autrata & Strutt in (Sloman, 1994). Omnipoints (OMNIPoint, 1993) is a Network Management Forum initiative for achieving interoperability of management systems.

Table 24.6. Pointers to Information about Management Systems and Platforms

Sys. Class	*System*	*Pointers*
ISO		www.iso.ch
ITU	(ex-CCITT)	www.itu.int
RFCs	(IETF)	www.rfc-editor.org
ICANN	(Names & Nrs)	www.icann.org
OpenGroup	(ex-OSF)	www.opengroup.org/management
DMTF		www.dmtf.org
	(DMI spec.)	www.dmtf.org/spec/dmis.html
NMF		www.nmf.org
		www.tmforum.org
OMG	(Objects)	www.omg.org
GNOME	(GNU GPL)	www.gnome.org
ASN.1	(Syntax Not.)	www-sop.inria.fr/rodeo/personnel/hoschka/asn1.html
Management Related Sites	**SmurfWeb**	netman.cit.buffalo.edu
	SimpleWeb	www.simpleweb.org
	SimpleTimes	www.simple-times.org
	Agentlink	www.agentlink.org
Managem. Protocols	**CMISE/CMIP**	www.cs.ucl.ac.uk/research/osimis/share.htm
	SNMP	ftp.net.cmu.edu/pub/snmp
		www.gaertner.de/snmp
		ucd-snmp.ucdavis.edu
		www.snmpworld.com
	SNMPv3	www.ibr.cs.tu-bs.de/ietf/snmpv3
	Agentx	www.scguild.com/agentx
		ftp.net.cmu.edu/pub/agentx
	MIB Browser	www.ibr.cs.tu-bs.de/cgi-bin/sbrowser.cgi
	Beholder	dnpap.et.tudelft.nl/pub/btng/README
	Vendor MIBs	www.simpleweb.org/ietf/enterprise.html
	Ethereal	ethereal.zing.org
	MRTG	ee-staff.ethz.ch/~oetiker/webtools/mrtg
		ee-staff.ethz.ch/~oetiker/webtools/rrdtool
	Zebra	www.zebra.org
	Scotty	wwwhome.cs.utwente.nl/~schoenw/scotty
		www.ibr.cs.tu-bs.de/projects/scotty
Managem. and Configur. Packages Tools and Systems	**AAA**	www.dyade.fr/en/actions/aaa
	GxSNMP	www.gxsnmp.org
	LANdb	avenir.dhs.org/landb
	MibMaster	www.equival.com.au/mibmaster
	Regis	www-dse.doc.ic.ac.uk/~regis
	SMB-SNMP	jake.unipi.it/~deri/SMB_SNMP
	SAMBA	samba.anu.edu.au/samba
	StormCast	www.cs.uit.no/forskning/DOS/StormCast
	TACOMA	www.tacoma.cs.uit.no/
	Tcl	www.scriptics.com
	UTopia	www.simpleweb.org/nm/research/projects/utopia
Management Platforms	**HP OpenView**	www.openview.hp.com
	Tivoli	www.tivoli.com
	Spectrum	www.aprisma.com
	Solstice	www.sun.com/solstice
	DCE	www.opengroup.org/dce

Table 24.6 *(continued)*
. Pointers to Information about Management Systems and Platforms

Sys. Class	*System*	*Pointers*
Security Manag. Tools	**CERIAS**	www.cerias.purdue.edu
	tripwire	ftp.cerias.purdue.edu/pub/tools/unix/ids/tripwire
	Xinetd	qiclab.scn.rain.com/pub/security www.synack.net/pub/xinetd
	Tcpwrapper	ftp.ox.ac.uk/pub/comp/security/software/monitors/ ftp.cerias.purdue.edu/pub/tools/unix/netutils/tcp_wrappers
	Portmapper	ftp.cerias.purdue.edu/pub/tools/unix/netutils/portmap
	Cracklib	ftp.cerias.purdue.edu/pub/tools/unix/libs/cracklib
	Crack	ftp.cerias.purdue.edu/pub/tools/unix/pwdutils/crack
	COPS	ftp.cerias.purdue.edu/pub/tools/unix/scanners/cops
	Tiger	ftp.cerias.purdue.edu/pub/tools/unix/scanners/tiger
	ISS	ftp.cerias.purdue.edu/pub/tools/unix/scanners/iss
	Satan	www.cs.purdue.edu/coast/satan.html www.fish.com/~zen/satan/satan.html ftp.cerias.purdue.edu/pub/tools/unix/scanners/satan
	Courtney	ciac.llnl.gov/pub/ciac/sectools/unix/courtney ftp.cerias.purdue.edu/pub/tools/unix/logutils/courtney
	Merlin	ciac.llnl.gov/pub/ciac/sectools/unix/merlin/merlin.tar.gz
	Trojan	ftp.cerias.purdue.edu/pub/tools/unix/sysutils/trojan
	Sniff Det	ftp.cerias.purdue.edu/pub/tools/unix/sysutils/cpm ciac.llnl.gov/pub/ciac/sectools/unix/sniffdetect
	Scan Det	ftp.cerias.purdue.edu/pub/tools/unix/logutils/scan-detector
	AID	www-rnks.informatik.tu-cottbus.de/~sobirey/aid.e.html
	ASAX	www.info.fundp.ac.be/~amo/publications.html
	Hummer	www.csds.uidaho.edu/~hummer
	Tcpdump	ftp.cerias.purdue.edu/pub/tools/unix/netutils/tcpdump ftp.ee.lbl.gov
	Analyzer	netgroup-serv.polito.it/analyzer
	Swatch	ftp.cerias.purdue.edu/pub/tools/unix/logutils/swatch
	Logdaemon	ftp.cerias.purdue.edu/pub/tools/unix/logutils/logdaemon
	Netlog	ftp.cerias.purdue.edu/pub/tools/unix/logutils/netlog
	Netman	ftp.cerias.purdue.edu/pub/tools/unix/netutils/netman

25 CASE STUDY: MANAGING VP'63

This chapter finalizes our case study: managing the (VintagePort'63) Large-Scale Information System. VP'63 became significantly complex, and the company depends heavily on it. Its operation must remain stable, and its reconfiguration made as easy as possible. Tactical management mechanisms implementing strategic management policies will be studied, and developed around an integrated management platform.

25.1 ESTABLISHING MANAGEMENT STRATEGIES AND POLICIES

The reader should recall that this is the next step of a project implementing a strategic plan for the modernization of VP'63, started in Chapter 5, and continued in the Case-Study *chapters of each part of this book. The reader may wish to review the previous parts, in order to get in context with the project.*

The current infrastructure is managed on an ad hoc, uncoordinated way, since there was not until now a real distributed systems approach to the problem. The networking infrastructure evolved with the introduction of new segments and modules, and the corresponding network management points. Systems and applications are managed by staff local to the facilities. This situation is depicted in Figure 25.1.

Before attempting to do any change, a management strategy should be defined. The objectives of this investment on VP'63 are: to have a global and seamless information flow that serves decision making in the company; to al-

low corporate management decisions to be impressed as fast as possible on the information system. Corporate management is centralized, and as such, centralized strategic management is the option to make. The Chief Information Officer (CIO) helps define this strategy, in the form of management policies, and is responsible for its implementation by the tactical management team. Management policies should be defined in terms of resources (information and services) and users. They concern, amongst other things, the generic management policy for each service, and the characteristics of operations of users on resources.

Current management personnel expertise should be preserved, but perhaps reallocated under the viewpoint of the new organization. The core management team should be allocated to one, at most two, physical sites, from where they should be able to run the infrastructure. Then, more important facilities and specially those hosting factory automation subsystems will have some dedicated management personnel.

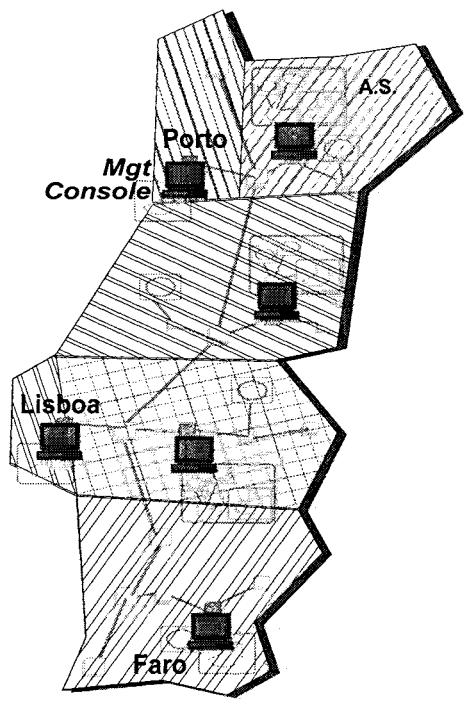

Figure 25.1. Uncoordinated Management

25.2 TOWARDS INTEGRATED MANAGEMENT

Integrated management is the best suited model to pursue the strategy underlined in the previous section. Given the geographical dispersion of the company, an integrated management platform should be selected that allows to perform

remote management on all managed resources in the company domain, composed of all facilities interconnected by the secure tunnels.

If the platform supports it, a composite management structure would prove quite effective in this system: hierarchical management, with mid-level agents located in the Gateway Facilities, each acting on the managed resources of their facility, and responding to the platform manager console above; and cooperative management among those mid-level agents.

The desired setting is shown in Figure 25.1a. The platform and its services are installed in the main facility at Porto, where the main management console is also installed. Given that important services also exist in Lisboa, a secondary management console is also installed there. The detail of the hierarchy to the inside of each facility is omitted in the drawing. All equipment should comply with the standard management communication protocol selected (e.g., SNMPv2, migrating to SNMPv3 a.s.a.p.), and with the standard MIB formats, such as Internet MIB-II and RMON2 MIB.

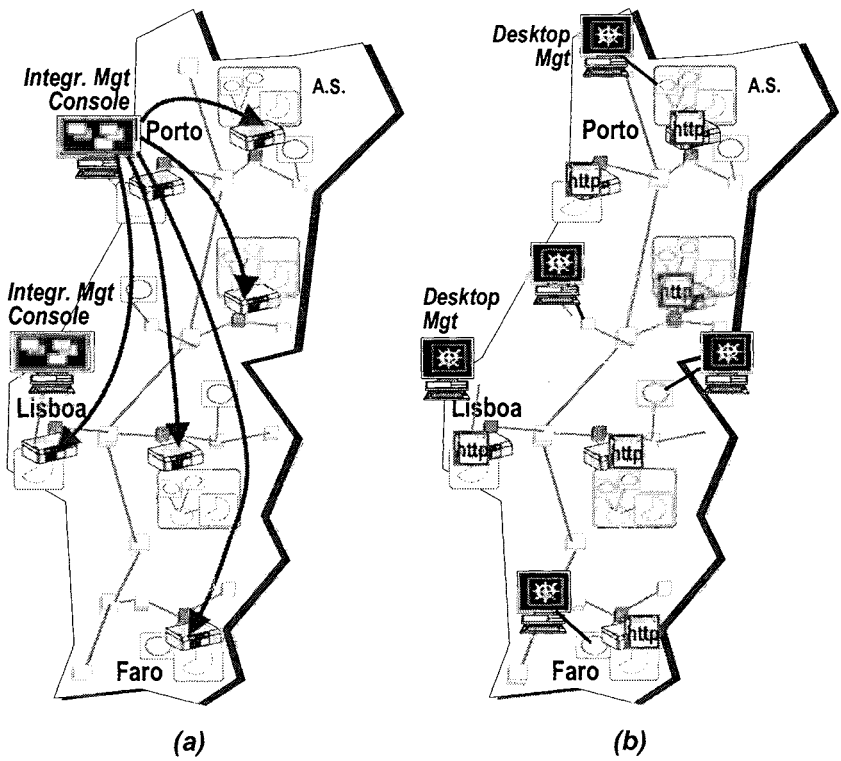

Figure 25.2. (a) Integrated Management; (b) Desktop Management

In a later phase, after the integrated management concept has stabilized, and the informatics culture of the company is more mature, an evolution towards desktop management may be envisaged, as depicted in Figure 25.1b. Most

of the current structure may remain. The integrated management agents and middle managers will be provided with HTTP servers. Many emerging equipments are already provided with individual web servers allowing web-based management. This evolution should be made as compatible as possible with the emerging DMI standard. This will allow a moderate but desirable decentralization of management, specially low-level local functions, since a desktop with a browser can virtually manage any equipment, depending on the access control capabilities of the user.

Further Issues

These issues need some refinement now, and the reader was assigned the study of a few questions that were still left unsolved:

Q.5. 11 *Select the actual tools that should equip a platform managing a system like VP'63.*

Q.5. 12 *Define a minimal structure for an enterprise-wide Help Desk and Trouble Ticket System.*

Q.5. 13 *Monitoring is addressed both as an industrial system (SCADA) function, and as a management function. Can they be aggregated or do they have different characteristics?*

References

Abadi, M. and Needham, R. (1994). Prudent engineering practice for cryptographic protocols. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy.*

Abdelzaher, T. and Shin, K. (1998). End-host architecture for qos-adaptive communication. In *Procs. of the 4th IEEE Real-Time Technology and Applications Symposium*, Denver, USA.

Abrams, M. (1998). *World Wide Web, Beyond the Basics.* Prentice Hall.

Abrams, M., Jajodia, S., and Podell, H., editors (1995). *Information Security.* IEEE CS Press.

ADA 83 (1983). Ans reference manual for the ada programming language. Technical Report Mil/Std 1815A-1983, American National Standards Institute.

Agnew, B., Hofmeister, C., and Purtilo, J. (1994). Planning for change. *IEEE/ IOP/BCS Distributed Systems Engineering Journal*, 1(5):313–322.

Agrawal, D. and El-Abbadi, A. (1990). Efficient techniques for replicated data management. In *Proceedings of the IEEE Workshop on the Management of Replicated Data*, pages 48–52, Houston, USA.

Agrawal, D. and El-Abbadi, A. (1991). An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transaction on Computer Systems.*

Aguilera, M., Chen, W., and Toueg, W. (1998). Failure detection and consensus in the crash-recovery model. In *Proc. 12th Int. Symposium on DIStributed Computing (formerly WDAG)*, pages 231–245, Andros, Greece. Sringer-Verlar LNCS 1499.

Ahamad, M. and Ammar, M. (1991). Multidemensional voting. *ACM Transactions on Computer Systems*, 9(4):339–431.

Ahamad, M., Hutto, P., and John, R. (1991). Implementing and programming causal distributed shared memory. In *Proceedings of the 11th IEEE International Conference on Distributed Computing Systems*, pages 274–281, Arlington, Texas, USA.

Aidarous, S. and Plevyak, T. (1998). *Telecommunications Network Management - Technologies and Implementations.* IEEE Press.

Almeida, C. and Veríssimo, P. (1996). Timing failure detection and real-time group communication in *quasi-synchronous* systems. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy.

Alpern, B. and Schneider, F. (1987). Recognizing safety and liveness. *Distributed Computing*, (2):117–126.

Alvisi, L., Elnozahy, E., Rao, S., Husain, S., and DeMel, A. (1999). An analysis of communication induced checkpointing. In *Digest of Papers, The 29th IEEE International Symposium on Fault-Tolerant Computing*, Madison - USA.

Alvisi, L. and Marzullo, K. (1993). Non-blocking and orphan-free message logging protocols. In *Digest of Papers, The 23rd IEEE International Symposium on Fault-Tolerant Computing*, pages 145–154, Toulouse, France.

Ames, S., Gasser Jr., M., and Schell, R. (1983). Security kernel design and implementation: An introduction. *Computer*, 16(7):14–22.

Amir, Y., Dolev, D., Kramer, S., and Malki, D. (1992). Membership algorithms for multicast communication groups. In *Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 292–312, Haifa, Israel.

Amir, Y., Dolev, D., Kramer, S., and Malki, D. (1993a). Transis: A communication sub-system for high-availability. In *Digest of Papers, The 22nd IEEE Int. Symp. on Fault-Tolerant Computing Systems*, pages 76–84.

Amir, Y., Moser, L., Melliar-Smith, P., Agarwal, D., and Ciarfella, P. (1993b). Fast message ordering and membership using a logical token-passing ring. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 551–560, Pittsburgh, Pennsylvania, USA.

Ananda, A., Tay, B., and Koh, E. (1992). A survey of asynchronous remote procedure calls. *Operating Systems Review*, 266(2):92.

Anderson, J. (1972). Computer security technology planning study. Technical Report ESD-TR-73-51, Hanscom AFB.

Anderson, R. and Needham, R. (1995). Robustness principles for public key protocols. In *Proceedings of the Advances in Cryptology– CRYPTO'95*. Springer-Verlag.

ANSA (1987). *ANSA Reference Manual, Release 00.03*. Advanced Networked Systems Architecture, ESPRIT technical week edition.

ANSA (1990). *ANSAware Release 3.0 Reference Manual*. APM Ltd, Cambridge.

ANSI X9.9 (1986). *American National Standard for Financial Institution Message Authentication (Wholesale)*.

Arlat, J., Aguera, M., Crouzet, Y., Fabre, J., Martins, E., and Powell, D. (1990). Fault injection for dependability validation: a methodology and some applications. *IEEE Trans. on SW Engineering*, Special Issue of Experimental Computer Science.

ASN.1 (1990). *Information Technology – Open Systems Interconnection – Specification of Abstract Notation One (ASN.1)*. ISO/IEC.

Audsley, N. (1993). *Flexible Scheduling of Hard Real-Time Systems*. PhD thesis, University of York, UK.

Aurrecoechea, C., Campbell, A., and Hauw, L. (1998). A survey of QoS architectures. *Multimedia Sys. Journal, Special Issue on QoS Arch.*, 6(3):138–151.

Babaoğlu, m. (1987). On the reliability of consensus-based fault-tolerant distributed computing systems. *ACM Transactions on Computer Systems*, 5(3):394–416.

Babaoğlu, m., Baker, M., Davoli, R., and Giachini, L.-A. (1994). RELACS: A communications infrastructure for constructing reliable applications in large-scale distributed systems. In *Procs. of the 28th Hawaii Int'l Confer. on System Sciences*.

Babaoğlu, m. and Marzullo, K. (1993). Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Mullender, S., editor, *Distributed Systems (2nd edition)*, chapter 4. Addison-Wesley.

Babaoğlu, O., Bartoli, A., and Dini, G. (2000). Programming partition-aware network applications. In Krakowiak, S. and Shrivastava, S., editors, *Recent Advances in Distributed Systems*, volume 1752 of *LNCS*, chapter 8. Springer-Verlag.

Babaoğlu, O., Drummond, R., and Stephenson, P. (1986). The impact of communication network properties on reliable broadcast protocols. *IEEE Transactions on Software Engineering*, (6):212–217.

Baker, S. (1997). *Corba Distributed Objects : Using Orbix*. Number ISBN: 0201924757. Addison-Wesley.

Bal, H. and Tanenbaum, A. (1988). Distributed programming with shared data. In *Procs. of the IEEE Conf. on Computer Languages*, pages 82–91.

Baldi, M., Gai, S., and Picco, G.-P. (1997). Exploiting code mobility in decentralized and flexible network management (ma'97). In *Proceedings of the 1st International Workshop on Mobile Agents 97*, pages 13–26, Berlin, Germany. Springer, Lecture Notes on Computer Science vol. 1219.

Banker, K. and Mellquist, P. (1995). Snmp++: A portable object-oriented approach to network management programming. *ConneXions*, 9(3):35–41.

Barabanov, M. and Yodaiken, V. (1997). Real-time linux. *Linux Journal*.

Barbacci, M., Weinstock, C., Doubleday, D., Gardner, M., and Lichota, R. (1993). Durra: A structure description language for developing distributed applications. *Sofware Engineering Journal*, 8(2):83–94.

Barborak, M., Malek, M., and Dahbura, A. (1993). The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220.

Barrett, P., Bond, P., Hilborne, A., Rodrigues, L., Seaton, D., Speirs, N., and Veríssimo, P. (1990). The Delta-4 Extra performance architecture (XPA). In *Digest of Papers, The 20th IEEE International Symposium on Fault-Tolerant Computing*, Newcastle-UK. also as INESC AR/21-90.

Bartlett, J., Gray, J., and Horst, B. (1987). Fault tolerance in Tandem computer systems. In Avizienis, A., Kopetz, H., and Laprie, J., editors, *Dependable Computing and Fault-Tolerant Systems*, volume 1, pages 55–76. Springer-Verlag.

Bauer, A., Bowden, R., Browne, J., Duggan, J., and Lyons, G. (1991). *Shop Floor Control Systems*. Chapman Hall, London.

Becker, L., Pereira, C., Dias, O., Teixeira, I., and Teixeira, J. (2000). Mosys a methodology for automatic object identification from system specification. In *Procs. of ISORC 2000, the 3rd IEEE Int'l Symp. on Object-Oriented Real-Time Distributed Computing*, pages 198–201, Newport Beach, USA.

Beekmann, D. (1989). CIM-OSA: Computer integrated manufacturing - open system architecture. *Int'l Journal Computed Integrated Manufacturing*.

Beertema, P. (1993). Common dns data file configuration errors. Technical Report RFC 1537, USc Inf. S. Inst.

Bell, D. and LaPadula, L. (1973). Secure computer systems: Mathematical foundations and model. Technical report, MITRE Corp.

Bellissard, L., Atallah, S., Boyer, F., and Riveill, M. (1996). Distributed application configuration. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems*, pages 579–585, Hong-Kong.

Bellovin, S. and Merritt, M. (1992). Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the 1992 IEEE Computer Society Conference on Research in Security and Privacy*, pages 72–84.

Ben-Ari, M. (1990). *Principles of Concurrent and Distributed Programming*. Prentice Hall.

Berndtsson, M. and Hansson, J. (1995). Issues in active real-time databases. In *Proceedings of the First ACM International Workshop on Active and Real-Time Database Systems*, pages 142–157, Skovde, Sweden.

Berners-Lee, T. and Cailliau, R. (1990). Worldwideweb: Proposal for a hypertext project. Technical report.

Bernstein, P., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

Bershad, B. and Zekauskas, M. (1991). Midway: Shared memory paralel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University.

Bhide, A., Elnozahy, E., and Morgan, S. (1991). A highly available network file server. In *Proceedings of the USENIX Winter Conference*, pages 199–205.

Biba, K. (1977). Integrity considerations for secure computer systems. Technical Report 76-372, U.S. Air Force Electronic Systems Division.

Birman, K., editor (1996). *Building Secure and Reliable Network Applications.* Number ISBN 1-884777-29-5. Manning Publications Co.

Birman, K. and Joseph, T. (1987). Reliable Communication in the Presence of Failures. *ACM, Transactions on Computer Systems*, 5(1).

Birman, K., Schiper, A., and Stephenson, P. (1991a). Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3).

Birman, K., Schiper, A., and Stephenson, P. (1991b). Lightweight Causal and Atomic Group Multicast. *ACM Transacs. on Computer Systems*, 9(3):272–314.

Birman, K. and van Renesse, R., editors (1994). *Reliable Distributed Computing With the ISIS Toolkit.* Number ISBN 0-8186-5342-6. IEEE CS Press.

Birrell, A. and Nelson, B. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1).

Bloom, J. and Dunlap, K. (1986). Experiences implementing bind, a distributed name server for the darpa internet. In *USENIX Summer*, pages 172–181.

Bodilsen, S. (1994). Scheduling theory and ada 9x. *Embedded Systems Programming*, pages 32–52.

Boehm, B. (1988). A spiral model of software development and enhancement. *IEEE Computer*, pages 61–72.

Boly, J., Bosselaers, A., Cramer, R., Michelsen, R., Mjolsnes, S., Muller, F., Pedersen, T., Pfitzmannn, B., de Rooji, P., Schoenmakers, B., Schunter, M., Vallée, L., and Waidner, M. (1994). The esprit project cafe – high security digital payment system. In *Proceedings of the Third ESORICS, European Symposium on Research in Computer Security*, pages 217–230. Springer-Verlag, Vol.875.

Boorstin, D. (1983). *The Discoverers.* Gradiva/Random House.

Bowen, N., Antognini, J., Regan, R., and Matsakis, N. (1997a). Availability in parallel systems: automatic process restart. *IBM Systems Journal*, 36(2):284–300.

Bowen, N., Elko, D., Isenberg, J., and Wang, G. (1997b). A locking facility for parallel systems. *IBM Systems Journal*, 36(2):202–220.

Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., and Yovine, S. (1998). Kronos: a model-checking tool for real-time systems. In *Proceedings of CAV'98, the 10th IEEE Conference Computer-Aided Verification.*

Bradley, D., Dawson, D., Burd, N., and Loader, A., editors (1991). *Mechatronics, Electronics in Products and Processes.* Chapman and Hall.

Brands, S. (1995). Electronic cash on the internet. In *Proceedings of the Internet Society 1995 Symposium on Network and Distributed Systems Security*, pages 64–84. IEEE Computer Society Press.

Brewer, E., Gauthier, P., Goldberg, I., and Wagner, D. (1995). Basic flaws in internet security and commerce. Technical report, Dpt. of CS, Berkeley University.

Brites, A., Simões, P., Leitão, P., Monteiro, E., and Fernandes, F. (1994). A high-level notation for the specification of network management applications. In *Proceedings of the INET'94/JENC5*, pages 5611–8.

Budhiraja, N., Marzullo, K., Schneider, F., and Toueg, S. (1993). The primary-backup approach. In Mullender, S., editor, *Distributed Systems, 2nd Edition*, ACM-Press, chapter 8. Addison-Wesley.

Burns, A. and Welling, A. (1996). Advanced fixed priority scheduling. In Joseph, M., editor, *Real-Time Systems*. Prentice-Hall.

Burns, A. and Wellings, A. (1995). *Hard Real-Time HOOD: A Structured Design Method for Hard Real-Time Ada Systems*. Elsevier.

Burns, A. and Wellings, A. (1996). *Real-Time Systems and Programming Languages*. International Computer Science Series. Addison-Wesley.

Buttazzo, G., editor (1997). *Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers.

Callahan, J. and Montgomery, T. (1996). Approaches to verification and validation of a reliable multicast protocol. *ACM Software Engeneering Notes*, 21(3).

CAN (1993). *Int'l Std.11898- Road vehicles - Interchange of digital information - Controller Area Network (CAN) for high-speed communication*. ISO.

Carreira, J., Madeira, H., and Silva, J. (1998). Xception: A technique for the experimental evaluation of dependability in modern computers. *Transactions on SW Engineering*, 24(2):125–136.

Carriero, N. and Gelertner, D. (1986). The S/Net's Linda Kernel. *ACM Transactions on Computer Systems*, 4(2).

Cart, M., Ferrie, J., and Mardyanto, S. (IFIP, 1987). Atomic broadcast protocol, preserving concurrency for an unreliable broadcast network. In Cabanel, J., Pujole, G., and Danthine, A., editors, *Local communication systems: LAN and PBX*. North-Holland.

Carter, J., Bennettt, J., and Zwanepoel, W. (1991). Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 152–164.

Carter, W. and Schneider, P. (1968). Design of dynamically checked computers. In *Proc. IFIP'68 World Computer Congress*, pages 878–883.

CC-ITSE (1998). *Common Criteria for Information Technology Security Evaluation*. ISO/IEC JTC 1.

Chandra, T., Hadzilacos, V., and Toueg, S. (1996). On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 322–330.

Chandra, T. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 34(1):225–267.

Chandy, K. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM*, 3(1):63–75.

Chang, J. and Maxemchuck, N. (1984). Reliable broadcast protocols. *ACM, Transactions on Computer Systems*, 2(3):251–273.

Chappell, D. (1992). The osf distributed management environment. *ConneXions*, 6(10):10–15.

Chaum, D. (1983). Blind signatures for untraceable payments. In *Proceedings of the Advances in Cryptology– Crypto'82*, pages 199–203. Plenum Press.

Chaum, D. (1992). Achieving electronic privacy. *Scientific American*, 267(2):96–101.

Chen, L. and Avizienis, A. (1978). N-version programming: A fault-tolerance approach to reliability of software operation. *8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, pages 3–9.

Chen, W., Toueg, S., and Aguilera, M. (2000). On the quality of service of failure detectors. In *Procs. of DSN 2000, the IEEE/IFIP Int'l Conf. on Dependable Systems and Networks.*

Chereque, M., Powell, D., Reynier, P., Richier, J.-L., and Voiron, J. (1992). Active replication in Delta-4. In *Digest of Papers, The 22nd IEEE Int'l Symp. on Fault-Tolerant Computing Systems*, page 28.

Cheriton, D. and Mann, T. (1989). Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183.

Cheriton, D. and Skeen, D. (1993). Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 14th Symposium on Operating Systems Principles*, Asheville, NC, USA.

Cheriton, D. and Zwaenepoel, W. (1985). Distributed process groups in the V-kernel. *ACM Tran. on Computer Systems*, 3(2).

Cheswick, W. and Bellovin, S. (1997). *Internet Security: Firewalls and Gateways, 2nd edition.* Addison-Wesley.

Cheung, S., Ammar, M., and Ahamad, M. (1990). The grid protocol: A high performance scheme for maintaining replicated data. In *Proceedings of the 6th Internation Conference on Data Engineering*, pages 438–445.

Chung, S., Lazowska, E., Notkin, D., and Zahorjan, J. (1989). Performance implications of design alternatives for remote procedure call stubs. In *Proceedings of the 9th Int'l IEEE Conference on Distributed Computing Systems*, pages 36–41, Newport Beach - USA.

Clark, D. and Wilson, D. (1987). A comparison of commercial and military computer security policies. In *Proc. of the IEEE Symp. on Security and Privacy*, pages 184–194.

Clegg, M. and Marzullo, K. (1996). Clock synchronization in hard real-time distributed systems. Technical Report CS96-478, University of California, San Diego, Department of Computer Science and Engineering.

CMIP (1988). *Open Systems Interconnection – Management Information Protocol Definition, Part 2: Common Management Information Protocol.* ISO.

CMISE (1988). *Open Systems Interconnection – Management Information Service Definition, Part 2: Common Management Information Service.* ISO.

CNMA (1993). CNMA Implementation Guide, Revision 6.01. Technical report, ESPRIT Project 7096.

Comer, D. (1991). *Internetworking With TCP/IP: Principles, Protocols, Architecture.* Prentice Hall.

Comer, D. (1997). *The Internet Book.* Prentice Hall.

Cooper, E. (1985). Replicated distributed programs. In *Procs. of the 10th ACM Symposium on Operating Systems Principles*, Berkeley– USA.

Corneillie, P., Deswarte, Y., Goodson, J., Hawes, A., Kaâniche, M., Kurth, H., Liebisch, G., Manning, T., Moreau, S., Steinacker, A., and Valentin, C. (1999). Squale– dependability assessment criteria (4th draft). Technical Report 98456, ACTS proj. AC097, LAAS, Squale Consortium.

Cornhill, D., Sha, L., Lehoczky, J., Rajkumar, R., and Tokuda, H. (1987). Limitations of ada for real-time scheduling. In *Proceedings of the ACM International Workshop on Real Time Ada Issues*, Ada Letters, pages 33–39.

Cosquer, F., Antunes, P., and Veríssimo, P. (1996). Enhancing dependability of cooperative applications in partitionable environments. In *Dependable Computing - EDCC-2*, volume 1150 of *LNCS*, pages 335–352. Springer-Verlag.

Crane, S., Dulay, N., Fossa, H., Kramer, J., Magee, J., Sloman, M., and Twidle, K. (1995). Configuration management for distributed software services. In *Proceedings of the 4th IFIP/IEEE Int'l Symposium on Integrated Network Management (ISINM'95)*, Santa Barbara, USA.

Cristian, F. (1988). Reaching agreement on processor group membership in synchronous distributed system. Technical Report RJ 5964 (59426), IBM Almaden Research Center.

Cristian, F. (1989). Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–148.

Cristian, F. (1990). Synchronous atomic broadcast for redundant broadcast channels. *The Journal of Real-Time Systems*, 2(1):195–212.

Cristian, F. (1994). Abstractions for fault-tolerance. In *Proceedings of the 13th IFIP World Computer Congress*, Hamburg.

Cristian, F., Dancey, B., and Dehn, J. (1996). Fault-tolerance in air traffic control systems. *ACM Transaction on Computer Systems*.

Cristian, F. and Fetzer, C. (1998). The timed asynchronous system model. In *Proceedings of the 28th IEEE Annual International Symposium on Fault-Tolerant Computing*, pages 140–149, Munich, Germany.

Cristian, F., H., A., Strong, R., and Dolev, D. (1985). Atomic broadcast: From simple message diffusion to byzantine agreement. In *Digest of Papers, The 15th IEEE International Symposium on Fault-Tolerant Computing*, Ann Arbor-USA.

Dana, P. (1996). Global positioning system (gps) time dissemination for real-time applications. *Journal of Real-Time Systems, this issue.*

Daneshgar, F. and Ray, P. (1997). Cooperative management based on awareness modelling. In *Proceedings of the 8th IFIP/IEEE Int'l Workshop on Distributed Systems Operations and Management (DSOM'97)*, Sydney, Australia.

Danzig, P., Obraczka, K., and Kumar, A. (1992). An analiysis of wide-area name server traffic. In *Proceedings of ACM SIGCOM 1992*, pages 281–292.

Davcev, D. (1989). A dynamic voting scheme in distributed systems. *IEEE Transactions on Software Engineering*, 15(1):93–97.

Debar, H., Dacier, M., and Wespi, A. (1999). A revised taxonomy for intrusion-detection systems. Technical Report 53, IBM Research, Zurich Research Laboratory.

Deering, S. (1989). Host extensions for ip multicasting. Technical Report RFC 1112, Stanford University, Stanford, CA, USA.

Deering, S., Estrin, D., Farinacci, D., Jacobson, V., and Liu, C-G. andWei, L. (1996). The PIM architecture for wide-area multicast routing. *IEEE/ACM Transactions on Networking*, 4(2):153–162.

Deitel, H. and Deitel, P. (2000). *Internet and World Wide Web: How to Program.* Number ISBN: 0130161438. Prentice Hall.

Denning, D. (1976). A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243.

Denning, D. and Sacco, G. (1981). Time-stamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536.

Deri, L. and Ban, B. (1997). Static vs. dynamic cmip/snmp network management using corba. In *Proceedings of the 4th Int'l Conference on Intelligence in Services and Networks (IS&N'97).*

DES (1977). *Data Encryption Standard.*

Deswarte, Y., Blain, L., and Fabre, J.-C. (1991). Intrusion tolerance in distributed systems. In *Proc. of the IEEE Symp. on Security and Privacy*, pages 110–121, Oakland - USA.

Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654.

DigiCash (1994). World's first electronic cash payment over computer networks. Technical report, DigiCash Press Release.

Dolev, D., Dwork, C., and Stockmeyer, L. (1983). On the minimal synchronism needed for distributed consensus. *24th Annual IEEE Symp. on Foundations of Computer Science.*

Dolev, D., Kramer, S., and Malki, D. (1993). Early delivery totally ordered multicast in asynchronous environments. In *Digest of Papers, The 23th IEEE Int'l Symp. on Fault-Tolerant Computing*, pages 544–553, Toulouse, France.

Dolev, D. and Yao, A. (1981). On the security of public key protocols. In *Proc. of the 22nd Annual Symp. on the Foundations of Computer Science*, pages 350–357.

Drummond, R. and Babaoğlu, O. (1993). Low Cost Clock Synchronization. *Distributed Computing*, 6:193–203.

DSS (1994). *Digital Signature Standard.*

Dupuy, F., Nilsson, G., and Inoue, Y. (1995). The tina consortium: Toward networking telecommunications information services. *IEEE Communications Magazine*, pages 78–83.

Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323.

EES (1994). *Escrowed Encryption Standard.*

ElGamal, T. (1985). A public-key cryptosystem and a signature scheme based on discrete logarithms. In *Proc. of Advances in Cryptology– CRYPTO'84*, pages 10–18. Springer-Verlag.

Ellison, C. and Schneier, B. (2000). Ten risks of pki: What you're not being told about public key infrastructure. *Computer Security Journal*, XVI(1).

Elnozahy, E., Alvizi, L., Wang, Y., and Johnson, D. (1999). A survey of rollback-recover protocols in message-passing systems. Technical Report CMU-CS-99-148, Carnegie Mellon University.

Elnozahy, E. and Zwaenepoel, W. (1992a). Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit. *IEEE Transactions on Computers*, 41(5):526–531.

Elnozahy, E. and Zwaenepoel, W. (1992b). Replicated distributed process in Manetho. In *Digest of Papers, The 22nd IEEE International Symposium on Fault-Tolerant Computing Systems*, page 18.

Eppinger, J.and Mummert, L. and Spector, A. (1991). *Camelot and Avalon.* Morgan Kaufmann Publishers, Inc.

Eswaran, K., Gray, J., Lorie, R., and Traiger, I. (1976). The notions of consistency and predicate locks in a database system. *Communics. of the ACM*, 19(11):624–633.

Ezhilchelvan, P., Macedo, R., and Shrivastava, S. (1995). Newtop: A fault-tolerant group communication protocol. In *Proc. of the 15th IEEE Int'l Conference on Distributed Computing Systems*, pages 296–306, Vancouver, Canada.

Fabre, J., Nicomette, V., Pérennou, T., Stroud, R., and Wu, Z. (1995). Implementing fault tolerant applications using reflective object-oriented programming. In *Digest of Papers of the 25th IEEE International Symposium on Fault-Tolerant Computing Systems*, pages 489–498.

FDDI, X. (1986). *FDDI documents: Media Access Layer, Physical and Medium Dependent Layer, Station Mgt.*

Felber, P., Grabinato, B., and Guerraoui, R. (1996). The design of a CORBA group communication service. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, pages 150–159, Niagara-on-the-Lake, Canada.

Felber, P., Guerraoui, R., and Schiper, A. (1997). Replicating objects using the corba event service. In *Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 14–19, Tunis, Tunisia.

Felten, E., Balfanz, D., Dean, D., and Wallach, D. (1996). Web spoofing: an internet con game. Technical Report 540-96, Princeton University, Department of CS.

Fetzer, C. and Cristian, F. (1996). Fail-awareness in timed asynchronous systems. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 314–321, New York, USA.

Fetzer, C. and Cristian, F. (1997a). Fail-awareness: An approach to construct fail-safe applications. In *Proc. of the 27th IEEE Annual Int'l Fault-Tolerant Computing Symposium*, pages 282–291, Seattle, USA.

Fetzer, C. and Cristian, F. (1997b). Integrating external and internal clock synchronization. *Journal of Real-Time Systems*, 12(2).

FIP (1990). *General Purpose Field Communication System – Part 3, WorldFIP.*

Fischer, M., Lynch, N., and Paterson, M. (1985). Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32:374–382.

Fisher, T., editor (1990). *Batch Control Systems: Design, Application and Implementation.* ISA.

Fohler, G. (1995). Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings of RTSS'95, the IEEE Real-Time Systems Symposium*, pages 152–161, Pisa, Italy.

Forestier, J., Forarino, C., and Franci-Zannettacci, P. (1989). Ada++: A class and inheritance extension for ada. In *Procs. of the Ada-Europe Int'l Conf.*, Ada Companion Series, Madrid-Spain. Cambridge Univ. Press.

Fosså, H. (1997). *Interactive Configuration Management for Distributed Systems.* PhD thesis, University of London, Imperial College.

Friday, A., Davies, N., Blair, G., and Cheverst, K. (1999). Developing adaptive applications: The MOST experience. *Journal of Integrated Computer-Aided Engineering*, 6(2).

Fritzke Jr., U., Ingels, P., Moustefaoui, A., and Raynal, M. (1998). Fault-tolerant total order multicast to asynchronous groups. In *Proc. 17th IEEE Symp. on Reliable Distributed Systems*, pages 228–234, West Lafayette, USA.

Furht, B., Grostick, D., Gluch, D., Rabbat, G., P., J., and McRoberts, M. (1991). *Real-time Unix Systems Design and Application Guide.* Kluwer.

Garcia-Molina, H. (1982). Elections in distributed computer systems. *IEEE Transactions on Computers*, C-31(1):48–59.

Garcia-Molina, H. and Barbara, D. (1985). How to assign votes in distributed system. *Journal of the ACM*, 32(4):841–860.

Garcia-Molina, H. and Spauster, A. (1991). Ordered and reliable multicast communication. *ACM Transactions on Computers Systems*, 9(3):242–271.

Garfinkel, S. and Spafford, G. (1997). *Web Security and Commerce.* O'Reilly.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibsons, P., Gupta, A., and Henessy, J. (1990). Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, USA.

Gifford, D. (1979). Weighted Voting For Replicated Data. In *Proceedings of the Seventh ACM Symposium on Operating System Principles*, pages 150–162.

Golding, R. (1992). Weak consistent group communication for wide-area systems. In *Proceedings of the Second IEEE Workshop on the Management of Replicated Data*, pages 13–16, Monterey, California.

Goldszmidt, G. and Yemini, Y. (1995). Distributed management by delegation. In *Proc. of the 15th IEEE Int'l Conference on Distributed Computing Systems.*

Gollmann, D. (2000). On the verification of cryptographic protocols – a tale of two committees. *Electronic Notes in Theoretical Computer Science*, 32.

Gong, L. (1992). A security risk of depending on synchronized clocks. *Operating Systems Review*, 26(1):49–53.

Goodman, J. (1989). Cache consistency and sequencial consistency. Technical Report 61, SCI Commitee.

Gorur, R. and Weaver, A. (1988). Setting target rotation times in an IEEE Token Bus network. *IEEE Transactions on Industrial Electronics*, 35(3).

Graw, G., Herrmann, P., and Krumm, H. (2000). Verification of uml-based real-time system designs by means of ctla. In *Proceedings of ISORC 2000, the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 86–95, Newport Beach, USA.

Gray, C. and Cheriton, D. (1989). Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210.

Gray, J. (1978). *Notes on Database Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag.

Gray, J. (1986). Why do computers stop and what can be done about it? In *Proceedings of the 5th IEEE Sycmp. on Reability in Distributed Software and Database Systems*, pages 3–12, Los Angeles, USA.

Gray, J. and Reuter, A. (1993). *Transaction processing: concepts and techniques.* Series in Data Management Systems. Morgan Kaufmann.

Guedes, P. and Castro, M. (1993). Distributed shared object memory. In *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems*, pages 142–149, Napa, California, USA.

Guerraoui, R., Hurfin, M., Mostefaoui, A., Oliveira, R., Raynal, M., and Schiper, A. (2000). Consensus in asynchronous distributed systems: a concise guided tour. In Krakowiak, S. and Shrivastava, S., editors, *Advances in Distributed Systems*, LNCS 1752, chapter 2, pages 33–47. Springer Verlag.

Guerraoui, R. and Schiper, A. (1997). Total order multicast to multiple groups. In *IEEE 17th Intl. Conf. Distributed Computing Systems*, pages 578–585.

Guo, K. and Rodrigues (1997). Dynamic light-weight groups. In *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems (ICDCS'17)*, Baltimore, Maryland, USA.

Gusella, R. and Zatti, S. (1989). The accuracy of the clock synchronization achieved by tempo in berkeley unix 4.3bsd. *IEEE Transactions on Software Engineering*, 15(7):847–853.

Guy, R., Page, T., Heidemann, J., and Popek, G. (1990). Name transparency in very large scale distributed file systems. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, pages 20–25, Huntsville, Alabama.

Haberman, S., Falciani, A., and Riggsby, M. (2000). *Mastering Lotus Notes and Domino R5 Premium Edition.* Number ISBN: 0782126359.

Hachiga, J. (1992). The Concepts and Technologies of Dependable and Real-time Computer Systems for Shinkansen Train Control. In *Proc. of the 2nd Int'l Workshop on Responsive Computer Systems*, Tokyo, Japan. Springer-Verlag.

Hadzilacos, V. and Toueg, S. (1994). A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca– USA.

Halpern, J. and Moses, Y. (1987). Knowledge and common knowledge in a distributed environment. Technical Report RJ4421, IBM Research Laboratory.

Halpern, J., Simons, B., Strong, R., and Dolev, D. (1984). Fault-Tolerant Clock Synchronization. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 89–102, Vancouver, Canada.

Halpern, J. and Suzuki, I. (1991). Clock synchronization and the power of broadcasting. *Distributed Computing*, 5(2):73–82.

Halsall, F. (1994). *Data Communications, Computer Networks and Open Systems, 3rd Ed.* Addison-Wesley.

Harper, R., Lala, J., and Deyst, J. (1988). Fault-tolerant parallel processor architecture overview. In *Digest of Papers, the 18th FTCS, IEEE Int'l Symp. on Fault-Tolerant Computing*, pages 252–257, Tokyo - Japan.

Hawking, S. (1988). *A Brief History of Time - from the Big Bang to Black Holes.* Gradiva.

Hayden, M. (1998). *The Ensemble System.* PhD thesis, Cornell University, Computer Science Department.

Hayes, S. (1993). Analyzing network performance management. *IEEE Communications Magazine*, pages 52–58.

Hedenetz, B. (1998). A development framework for ultra-dependable automotive systems based on a time-triggered architecture. In *Proceedings of RTSS'98, the IEEE Real-Time Systems Symposium*, pages 358–367, Madrid, Spain.

Hegering, H.-G. and Abeck, S. (1994). *Integrated Network and System Management.* Addison-Wesley.

Heiner, G. and Thurner, T. (1998). Time-triggered architecture for safety-related distributed real-time systems in transportation systems. In *Digest of Papers, The 28th IEEE Int'l Symp. on Fault-Tolerant Computing Systems*, Munich, Germany.

Henning, M. and Vinoski, S. (1999). *Advanced CORBA Programming with C++.* Number ISBN: 0201379279. Addison-Wesley.

Herlihy, M. and Wing, J. (1990). Linearizability: a correctness condition for concurrent objects. *ACM Transac. on Programming Languages and Systems*, 12(3):463–492.

Hiltunen, M. and Schlichting, R. (1993). An approach to constructing modular fault-tolerant protocols. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pages 105–114, Princeton, New Jersey.

Hiltunen, M. and Schlichting, R. (1994). Properties of membership services. Technical report, University of Arizona, Department of Computer Science, Tucson, AZ.

Hong, J., Kim, J.-S., and Park, J.-K. (1999). A corba-based quality of service management framework for distributed multimedia services and applications. *IEEE Network*, 13(2):70–79.

Hong, J., Kong, J.-Y., Yun, T.-H., and Kim, J.-S. (1997). Web-based intranet services and network management. *IEEE Communic's Magazine*, 35(10):100–110.

Hood, C. and Ji, C. (1996). Probabilistic network fault detection. In *Proceedings of the IEEE Globecom'96*, pages 1872–1876, London, UK.

Hopkins, A., Smith, T., and Lala, J. (1978). FTMP - A highly reliable fault-tolerant multiprocessor for aircraft. *Proceedings of IEEE*, 66(10):1221–1239.

Hughes, D. (1993). Esl-a script language for snmp (and then some!). Technical report, Bond University.

Hutchison, D., Coulson, G., Campbell, A., and Blair, G. (1994). Quality of service management in distributed systems. In Sloman, M., editor, *Network and Distributed Systems Management*, chapter 11. Addison-Wesley.

IEEE-RT (1994). Special issue on real-time systems. In *Procs. of the IEEE.*

ISO10040 (1992). *Information Technology – Open Systems Interconnection – Systems Management Overview.* ISO/IEC.

ISO10164 (1992). *Information Technology – Open Systems Interconnection – Systems Management Functions.* ISO/IEC.

ISO10165 (1992). *Information Technology – Open Systems Interconnection – Structure of Management Information.* ISO/IEC.

ISODE (1993). *ISODE Volume 1: Overview of ISODE.* England.

Issarny, V. (1993). An exception handling mechanism for parallel object-oriented programming: Towards reusable, robust distributed software. *Journal of Object-Oriented Programming*, 6(6):29–40.

ITSEC (1991). *Information Technology Security Evaluation Criteria, Ver. 1.2.*

Iyer, V. and Joshi, S. (1985). FDDI's 100 mb/s protocol improves on 802.5 specs 4mb/s limit. *EDN.*

Jahanian, F. and MoranJr, W. (1992). Strong, weak, and hybrid group membership. In *Proceedings of the Second IEEE Workshop on the Management of Replicated Data*, pages 34–38, Monterey, California.

Jalote, P. (1994). *Fault Tolerance in Distributed Systems.* Prentice-Hall.

Janetzky, D. and Watson, K. (1986). Token bus performance in MAP and PROWAY. In *Proceedings of the IFAC Workshop on Distributed Computer Protocol Systems.*

Jeffay, K. (1993). The real-time producer/consumer paradigm: A paradigm for construction of efficient, predictable real-time systems. In *Procs. of the ACM/SIGAPP Symp. on Applied Computing*, Indianapolis, USA.

Jeffay, K., Stanat, D., and Martel, C. (1991). On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of RTSS'91, the 12th IEEE Real-Time Systems Symposium*, pages 129–139.

Jensen, E. (2000). A proposed initial approach to distributed real-time java. In *Procs. of ISORC 2000, the Third IEEE Int'l Symp. on Object-Oriented Real-Time Distributed Computing*, pages 2–6, Newport Beach, USA.

Jensen, E. and Northcutt, J. (1990). Alpha: A non-proprietary os for large, complex, distributed real-time systems. In *Procs. of the IEEE Workshop on Experimental Distributed Systems*, pages 35–41, Alabama, USA.

Kaashoek, M. and Tanenbaum, A. (1991). Group communication in the Amoeba distributed operating system. In *Procs. of the 11th IEEE Int'l Conference on Distributed Computing Systems*, pages 222–230, Arlington, USA.

Kahn, D. (1967). *The Codebreakers: The Story of Secret Writing.* Macmillan Publishing Co.

Kaiser, J. and Livani, M. (1998). Invocation of real-time objects in a can-bus system. In *Proceedings of the ISORC 1998, IEEE International Symposium on Object-Oriented Real-Time Distributed Computing.*

Kaiser, J. and Mock, M. (1999). Implementing the real-time publisher/subscriber model on the controller area network. In *Procs. of the ISORC 1999, IEEE Int'l Symp. on Object-Oriented Real-Time Distributed Computing.*

Kaliski, B. (1993). A survey of encryption standards. *IEEE Micro*, 13(6):74–81.

Kalogeraki, V., Melliar-Smith, P., and Moser, L. (2000). Dynamic scheduling for soft real-time distributed object systems. In *Proceedings of ISORC 2000, the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 114–121, Newport Beach, USA.

Kaufman, C., Perlman, R., and Speciner, M. (1995). *Network Security, Private Communication in a Public World.* Prentice-Hall.

Kemme, B., Pedone, F., Alonso, G., and Schiper, A. (1999). Processing transactions over optimistic atomic broadcast protocols. In *Procs. of the 19th IEEE Int' l Conference on Distributed Computing Systems*, pages 424–431, Austin, USA.

Kent, S. and Atkinson, R. (1998). Security architecture for the internet protocol. Technical Report Request for Comments 2401, IETF.

Kieckhafer, R., Walter, C., Finn, A., and Thambidurai, P. (1988). The MAFT architecture for distributed fault tolerance. *IEEE Trans. on Computers*, 37(4).

Kim, K. and You, J. (1990). A highly decentralized implementation model for the programmer-transparent coordination (ptc) scheme for cooperative recovery. In *Digest of Papers, 20th IEEE Intl. Symposium on Fault-Tolerant Computing Systems*, pages 282–289, Newcastle - England.

Koch, T. and Kramer, B. (1995). Toward a comprehensive distributed systems management. *Open Distributed Processing*, pages 259–270.

Koops, B.-J. (1999). Crypto law survey. Technical Report Version 14.3.

Kopetz, H. (1992). Sparse Time versus Dense Time in Distributed Systems. In *Proc. of the 12th IEEE Int'l Conf. on Distributed Computing Systems*, Yokohama, Tokyo.

Kopetz, H. (1997). *Real-Time Systems.* Kluwer.

Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., and Zainlinger, R. (1989a). Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, pages 25–41.

Kopetz, H. and Grunsteidl, G. (1993). TTP - a Time-Triggered Protocol for Fault-Tolerant Real-Time Systems. In *Digest of Papers, The 23rd IEEE Int'l Symp. on F/T Computing*, Toulouse, France.

Kopetz, H., Grunsteidl, G., and Reisinger, J. (1989b). Fault-tolerant Membership Service in a Synchronous Distributed Real-time System. In *Proceedings of the IFIP WG10.4 Int'l Working Conference on Dependable Computing for Critical Applications*, Sta Barbara - USA.

Kopetz, H. and Ochsenreiter, W. (1987). Clock Synchronization in Distributed Real-Time Systems. *IEEE Transactions on Computers*, C-36(8):933–940.

Kopetz, H. and Veríssimo, P. (1993). Real-time and Dependability Concepts. In Mullender, S., editor, *Distributed Systems, 2nd Edition*, ACM-Press, chapter 16, pages 411–446. Addison-Wesley.

Korth, H., Soparkar, N., and Silberschatz, A., editors (1996). *Time-Constrained Transaction Management: Real-time Constraints in Database Transaction Systems.* Kluwer Academic Publishers.

Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Journal of Real-Time-Systems*, 2(4):255–299.

Kronenberg, N., Levy, H., and Strecker, W. (1987). Vaxclusters: A closely-coupled distributed system. *ACM Trans. on Computer Systems*, 4(3):130–146.

Kumar, A. and Cheung, S. (1991). A high availability $\sqrt{N}$ hierarquical grid algorithm for replicated data. *Information Processing Letters*, (40):311–316.

Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. (1992). Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391.

Lai, X. (1992). *On the Design and Security of Block Ciphers.* ETH Series in Information Processing, Vol.1. Konstanz Hartung-Gorre Verlag.

Lamb, J. and Lew, P. (1996). *Lotus Notes Network Design : For Notes Release 3 and 4.* Computer Communications. McGraw-Hill.

Lamport, L. (1978a). The implementation of reliable distributed multiprocess systems. *Computer Networks 2,* (1978):95–115.

Lamport, L. (1978b). Time, Clocks and the Ordering of Events in a Distributed System. *CACM,* 21(7):558–565.

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transacs. on Computers,* 28(9):690–691.

Lamport, L. (1981). Password identification with insecure communications. *Communications of the ACM,* 24(11):770–772.

Lamport, L. (1984). Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Transactions on Prog. Lang. and Systems,* 6(2).

Lamport, L. (1994). The temporal logic of actions. *ACM Transactions on Programming Language and Systems,* 16(3).

Lamport, L. and Melliar-Smith, P. (1985). Synchronizing Clocks in the Presence of Faults. *Journal of the ACM,* 32(1):52–78.

Lamport, L., Shostak, R., and Pease, M. (1982). The byzantine generals problem. *ACM Transactions on Prog. Lang. and Systems,* 4(3).

Lampson, B. (1974). Protection. *Operating Systems Review,* 8(1):18–24.

Lampson, B. (1981). Atomic transactions. In *Distributed Systems - Architecture and Implementation: An Advanced Course,* volume 105 of *Lecture Notes in Computer Science,* chapter 11, pages 246–265. Springer-Verlag.

Lampson, B. (1993). Authentication in distributed systems. In Mullender, S., editor, *Distributed Systems, 2nd Edition,* ACM-Press, chapter 21. Addison-Wesley.

Lampson, B., Abadi, M., and Wobber, E. (1992). Authentication in distributed systems: Theory and practice. *ACM Trans. on Computer Systems,* 10(4):265–310.

Langsford, A. (1994). Osi management model and standards. In Sloman, M., editor, *Network and Distributed Syst. Management,* chapter 4. Addison-Wesley.

Laplante, P. (1997). *Real-time systems design and analysis: an engineer's handbook, 2nd Edition.* IEEE Press.

Laprie, J. (1987). Dependability: A Unifying Concept for Reliable Computing and Fault-Tolerance. In *Resilient Computing Systems,* volume 2. Collins and Wiley.

Laprie, J.-C. (1992). Dependability: A unifying concept for reliable, safe, secure computing. In *IFIP Congress,* volume 1, pages 585–593.

Laprie, J.-C., editor (1998). *Dependability Handbook,* volume Report Nr.98-346 of *Laboratory for Dependability Engineering.* LAAS.

Lauer, H. and Satterwaite, E. (1979). The impact of mesa on systems design. In *Procs. of the 4th IEEE Int'l Conf. on Software Engineering,* pages 174–182.

Le Lann, G. and Rivière, N. (1993). Real-time communications over broadcast networks: the CSMA-DCR and the DOD-CSMA-CD protocols. Technical Report 1863, INRIA.

Lea, D. (1997). *Concurrent Programming in Java. Design Principles and Patterns.* Addison-Wesley.

Lee, P. and Anderson, T. (1990). *Fault-Tolerant: Principles and Practice, Second Edition.* Springer-Verlag.

Lehoczky, J. (1998). Scheduling communication networks carrying real-time traffic. In *Procs. of RTSS'98, the 19th IEEE Real-Time Systems Symp.*

Leiner, B., Cerf, V., Clark, D., Kahn, R., Kleinrock, L., Lynch, D., Postel, J., Roberts, L., and Wolff, S. (1997). The past and future history of the internet. *Communications of the ACM*, 40(2):102–108.

Leinwand, A. and Conroy, K. (1996). *Network Management: a Practical Perspective.* UNIX and Open Systems. Addison-Wesley.

Leslie, I., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R., and Hyden, E. (1996). The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communication*, 14(7).

Leung, J. and J., W. (1982). On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250.

Levine, P. (1987). The apollo domain distributed file system. In *Theory and Practice of Distributed Operating Systems.* Springer Verlag, NATO ASI Series.

Lewis, L. and Dreo, G. (1993). Extending trouble ticket systems to fault diagnostics. *IEEE Network*, 7(6):44–51.

Li, K. and Hudak, P. (1989). Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359.

Lin, J. and Paul, S. (1996). RMTP: a reliable multicast transport protocol. In *Proceedings of the IEEE INFOCOM'96*, pages 1414–1424.

Linn, J. (1993). Privacy enhancement for internet electronic mail: Part I-message encipherment and authentication procedures. Technical Report RFC1421, IETF.

Linn, J. (1996). Generic security service application programming interface (GSS-API), version 2. Technical Report Internet Draft, IETF.

Liskov, B. (1985). The ARGUS language and system. In *Distributed Systems, Methods and Tools for Specification*, volume 190 of *LNCS*. Springer-Verlag.

Liskov, B., Castro, M., Shrira, L., and Adya, A. (1999). Providing persistent objects in distributed systems. In *Proceedings of the ECOOP'99 - Object Oriented Programming*, number 1628 in Lecture Notes in Computer Science, pages 230–257, Lisbon, Portugal. Springer-Verlag.

Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., and Shrira, L. (1992). Efficient recovery in harp. In *Proceedings of the Second IEEE Workshop on the Management of Replicated Data*, pages 104–106, Monterey, California.

Liskov, B., Scheifler, R., Walker, E., and Weihl, W. (1987). Orphan detection. In *Digest of Papers, The 17th IEEE International Symposium on Fault-Tolerant Computing*, Pittsburgh-USA.

Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61.

Lockhart Jr., H. (1994). *OSF DCE.* McGraw-Hill.

Lowe, G. (1995). An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133.

Lundelius, J. and Lynch, N. (1984a). A New Fault-Tolerant Algorithm for Clock Syncr onization. In *Proceedings of the 3rd ACM SIGACT-SIGOPS Symp. on Prin ciples of Distrib. Computing*, pages 75–88, Vancouver-Canada.

Lundelius, J. and Lynch, N. (1984b). An upper and lower bound for clock synchronization. *Information and Control*, 62:190–204.

Lupu, E. and Sloman, M. (1997). Towards a role based framework for distributed systems management. *Journal of Network and Syst. Manag't*, 5(1).

Lynch, N. (1996). Data link protocols. In *Distributed Algorithms*, pages 691–732. Morgan-Kaufmann.

Lyu, M., editor (1995). *Software Fault Tolerance.* Wiley.

M. Shapiro, M. (1986). Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *Proceedings 6th IEEE Intl. Conf. on Distributed Computing Systems*, pages 198–204, Cambridge, USA.

Macedo, R., Ezhilchelvan, P., and Shrivastava, S. (1995). Flow control schemes for a fault-tolerant multicast protocol. Technical report, Univ. Newcastle upon Tyne.

Madeira, H. and Silva, J. (1994). Experimental evaluation of the fail-silent behaviour in computers without error masking. In *Digest of Papers, Fault-Tolerant Computers Symposium*, pages 350–359.

Magedanz, T. and Eckardt, T. (1996). Mobile software agent: A new paradigm for telecommunications management. In *Proceedings of the IEEE/IFIP Network and Management Operations Symposium (NOMS)*, Kyoto, Japan.

Magee, J., Dulay, N., and Kramer, J. (1993). Structuring parallel and distributed programs. *IEEE Software Engineering Journal*, 2(8):73–82.

Magee, J., Dulay, N., and Kramer, J. (1994). Regis: A constructive development environment for distributed programs. *IEEE/IOP/BCS Distributed Systems Engineering Journal*, 1(5):304–312.

Magee, J., Kramer, J., and Sloman, M. (1989). Constructing distributed systems in conic. *IEEE Transactions on Software Engineering*, SE-15(6):663–675.

Mahony, D., Peirce, M., and Tewari, H. (1997). *Electronic Payment Systems*. Artech House.

Malan, G. and Jahanian, F. (1998). An extensible probe architecture for network protocol performance measurement. In *Procs. of ACM SIGCOMM'98*.

Malkhi, D. and Reiter, M. (1998). Byzantine quorum systems. *Distributed Computing*, 11(4):203–213.

Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York.

Mansouri-Samani, M. and Sloman, M. (1994). Monitoring distributed systems. In Sloman, M., editor, *Network and Distributed Systems Management*, chapter 12, pages 303–347. Addison-Wesley.

MAP (1985). *Manufacturing Automation Protocol Specification V2.1*.

Martin-Flatin, J. (1999). Push vs. pull in web-based network management. In *Proceedings of the 6th IFIP/IEEE International Symposium on Integrated Network Management (IM'99)*, pages 3–18, USA.

Martin-Flatin, J.-P., Znaty, S., and Hubaux, J.-P. (1999). A survey of distributed network and systems management paradigms. *Journal of Network and Systems Management*, 7(1):9–26.

Marzullo, K. (1983). Maintaining the time in a distributed system. *ACM*, pages 295–305.

Marzullo, K. (1990). Tolerating failures of continuous valued sensors. *ACM Transactions on Computer Systems*, 8(4):284–304.

Maxion, R. and Olszewski, R. (1998). Improving software robustness with dependability cases. In *Digest of Papers, The 28th Int'l Symp. on Fault-Tolerant Computing Systems*, Munich, Germany. IEEE.

Mazumdar, S. (1996). Inter-domain management between corba and snmp: Web-based management - corba/snmp gateway approach. In *Proceedings of the 7th IEEE Int'l Workshop on Distributed Systems Operations and Management (DSOM'96)*, pages 28–30, L'Aquila, Italy.

McGraw, G. and Felten, E. (1997). *Java Security, Hostile Applets, Holes and Antidotes*. John Wiley.

Melliar-Smith, P., Moser, L., and Agrawala, V. (1990). Broadcast protocols for distributed systems. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):17–25.

Menezes, A., Van Oorschot, P., and Vanstone, S. (1999). *Handbook of Applied Cryptography, 4th ed.* CRC.

Merkle, R. (1978). Secure communication over insecure channels. *Communications of the ACM*, 21(4):294–299.

Merkle, R. and Hellman, M. (1981). On the security of multiple encryption. *Communications of the ACM*, 24(7):465–467.

Metcalfe, R. and Boggs, D. (1976). Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7).

Meyer, J. (1992). Performability: A retrospective and some pointers to the future. *Performance Evaluation North Holland*, 14, 3-4:139–155.

Meyer, J., Muralidhar, K., and Sanders, W. (1989). Performability of a token-bus network under transient faults. In *The 19th Annual IEEE International Symposium on Fault-Tolerant Computing*, Chicago-USA.

Micali, S. (1993). Fair public-key cryptosystems. In *Proceedings of the Advances in Cryptology-CRYPTO'92*, pages 113–138. Springer-Verlag.

MIL-STD-1553B (1988). *Field Bus Based on MIL-STD-1553B.*

Miller, C. (1999). *Multicast Networking and Applications.* Addison Wesley.

Mills, D. (1991). Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493.

Minar, N. (1999). A survey of the ntp network. Technical report, MIT.

Mintzberg, H. (1989). *Mintzberg on Management, Inside Our Strange World of Organizations.* Free Press, MacMillan.

Mishra, S., Peterson, L., and Schlichting, R. (1993). Consul: A communication substracte for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1(2):87–103.

Mishra, S. and Schlichting, R. (1992). Abstractions for constructing dependable distributed systems. Technical Report TR 92-19, The University of Arizona, Departement of Computer Science, Tucson, Arizona, USA.

MMS (1990). *MMS Specification - Part 1: Service definition, Part 2: Protocol specification.* International Organization for Standardization.

Mok, A. (1983). *Fundamental Design Problems of Distributed Systems for the Hard Real-time Environment.* PhD thesis, MIT, Cambridge-Mass., USA.

Morris/Satyanarayanan, Conner, M., Howard, J., Rosenthal, D., and Smith, F. (1986). Andrew: a Distributed Personal Computing Environment. *Communications of the ACM*, 29(3).

Moser, L., Amir, Y., Melliar-Smith, P., and Agarwal, D. (1994). Extended virtual synchrony. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, pages 56–65, Poland.

Moser, L., Melliar-Smith, P., Agarwal, A., Budhia, R., Lingley-Ppadopoulos, C., and Archambault, T. (1995). The Totem system. In *Digest of Papers of the 25th IEEE Int. Symp. on Fault-Tolerant Computing Systems*, pages 61–66.

Mossé, D., Melhem, R., and Ghosh, S. (1994). Analysis of a fault-tolerant multiprocessor scheduling algorithm. In *Digest of papers, the 24th FTCS, IEEE International Symposium on Fault-Tolerant Computing.*

Moy, J. (1994). Multicast routing extension for ospf. *Communications of the ACM*, 37(8):61–66.

Mullender, S., editor (1993). *Distributed Systems, 2nd Edition.* ACM-Press. Addison-Wesley.

Muller, N. (1997). Improving network operations with intelligent agents. *Journal of Network and Systems Management*, 7(3):116–126.

Needham, R. (1993). Cryptography and secure channels. In Mullender, S., editor, *Distributed Systems, 2nd Edition*, ACM-Press, chapter 20. Addison-Wesley.

Needham, R. and Schroeder, M. (1978). Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999.

Needham, R. and Schroeder, M. (1987). Authentication revisited. *Operating Systems Review*, 21(1):7.

Neuman, B. and Stubblebine, S. (1993). A note on the use of timestamps as nonces. *Operating Systems Review*, 27(2):10–14.

Neuman, B. and Ts'o, T. (1994). Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38.

Neumann, P. (1995). *Computer Related Risks*. Addison-Wesley.

Neves, N. and Fuchs, W. (1998). RENEW: A tool for fast and efficient implementation of checkpointing protocols. In *Digest of Papers, The 28th IEEE Int'l Symp. on Fault-Tolerant Computing Systems*, Munich, Germany.

Nicomette, V. and Deswarte, Y. (1997). An authorization scheme for distributed object systems. In *Proc. of the IEEE Symp. on Security and Privacy*, pages 21–30.

Norton, B. (1994). Integrating network discovery with network monitoring: The nsfnet method. In *Proceedings of the INET'94/JENC5*, page 5631 to 5636.

ODP (1987). *Proposed Revised Text for the New Work Item on the Basic Reference Model for Open Distributed Processing*. ISO/TC97/SC21 N1889.

Okamoto, T. and Ohta, K. (1992). Universal electronic cash. In *Proceedings of the Advances in Cryptology– CRYPTO'91*, pages 324–337. Springer-Verlag.

Oki, B., Pfuelgl, M., Siegel, A., and Skeen, D. (1993). The information bus- an architecture for extensible distributed systems. *ACM SIGOPS*, pages 58–68.

OMG (1997a). The Common Object Request Broker: Architecture and Specification.

OMG (1997b). CORBAservices: Common Object Services Specification.

OMNIPoint (1993). Understanding omnipoint, white paper. Technical report.

Otway, D. and Rees, O. (1987). Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10.

Ozden, B., Rastogi, R., and Silberschatz, A. (1996). Research issues in multimedia storage managers. *ACM Computing Surveys*.

Panzieri, F. and Roccetti, M. (2000). Responsive protocols for distributed multimedia applications. In Krakowiak, S. and Shrivastava, S., editors, *Advances in Distributed Systems*, volume 1752 of *LNCS*, chapter 7. Springer-Verlag.

Panzieri, F. and Shrivastava, S. (1988). Rajdoot: a remote procedure call mechanism supporting orphan detection and killing. *IEEE*, 14(1).

Paris, J.-F. and Sloope, P. (1992). Dynamic management of highly replicated data. In *Procs. of the 11th IEEE Symposium on Reliable Distributed Systems*, pages 20–28.

Parkinson, B. and Gilbert, S. (1983). Navstar: Global positioning system— ten years later. *Proceedings of the IEEE*, 71(10):1177–1186.

Patterson, D., Gibson, G., and Katz, R. (1988). A case for redundant arrays of inexpensive disks (RAID). In *Proc. of the ACM SIGMOD Conference*, volume 17, pages 109–116.

Paul, S. (1998). *Multicasting on the Internet and its Applications*. Kluwer Academic Publishers.

Pavlou, G., Liotta, A., Abbi, P., and Ceri, S. (1998). Cmis/p++: Extensions to cmis/p for increased expressiveness and efficiency in the manipulation of management information. *IEEE Network*, pages 10–20.

Pease, M., Shostak, R., and Lamport, L. (1980). Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2).

Peden, J. and Weaver, A. (1988). The utilization of priorities on token ring networks. In *Proc. of the 13th Conf. on Local Computer Networks*, Minneapolis, USA.

Pedone, F., Guerraoui, R., , and Schiper, A. (1998). Exploiting atomic broadcast in replicated databases. In *Proceedings of Europar Conference*, number 1470 in Lecture Notes in Computer Science, pages 513–520. Springer-Verlag.

Peterson, L., Buchholz, N., and Schlichting, R. (1989). Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–146.

Pfister, G. (1998). *In search of clusters. Second Edition.* Prentice Hall.

Pfleeger, C. (1996). *Security in Computing, 2nd Edition.* Prentice-Hall.

Pimentel, J. (1990). *Communication Networks for Manufacturing.* Prentice-Hall.

Pleinevaux, P. and Decotignie, J. (1988). Time critical communication networks: Field buses. *IEEE Network*, 2(3).

Poledna, S., editor (1995). *Fault-Tolerant Real-Time Systems: the Problem of Replica Determinism.* Kluwer Academic Publishers.

POSIX (1995). *Portable Operating System Interface (POSIX) - Part 1: API C Language - Real-Time Extensions.* ISBN 1-55937-375-X.

Post, M., Shen, C.-C., and Wei, J. (1996). The manager/agency paradigm for distributed network management. In *Proceedings of the IEEE Network Operations and Management Symposium (NOMS'96)*, pages 44–53.

Postel, J. (1978). Internetwork protocol specification - version 4. Technical Report IEN-41.

Powell, D., editor (1991). *Delta-4 - A Generic Architecture for Dependable Distributed Computing.* ESPRIT Research Reports. Springer Verlag.

Powell, D. (1992). Failure mode assumptions and assumption coverage. In *Proc. of The 22nd IEEE Int. Symp. on Fault-Tolerant Computing Systems*, page 386.

Powell, D. (1994). Distributed fault tolerance: Lessons from delta-4. *IEEE Micro*, pages 36–47.

Powell, D., Arlat, J., Beus-Dukic, L., Bondavalli, A., Coppola, P., Fantechi, A., Jenn, E., Rabejac, C., and A., W. (1999). GUARDS: A generic upgradable architecture for real-time dependable systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):580–599.

Powell, D., Seaton, D., Bonn, G., Veríssimo, P., and Waeselynk, F. (1995). The Delta-4 approach to dependability in open distributed computing systems. In Suri, N., Walter, C., and Hugue, M., editors, *Advances in Ultra-Dependable Distributed Systems.* IEEE CS Press. Reprinted from Digest of Papers, The 18th IEEE International Symp. on Fault-Tolerant Computing, Tokyo - Japan, June 1988.

Pras, A., Hazewinkel, H., and van Hengstum, E. (1997). Management of the world-wide web. In *Procs. of the SBRC'97*, pages 340–345, São Carlos, Brazil.

Preparata, F., Metze, G., and Chien, R. (1967). On the connection assignment problem of diagnosable systems. *IEEE Trans. on Electronic Computers*, 16(6):848–854.

Prodromides, K. and Sanders, W. (1993). Performability evaluation of csma/cd and csma/dcr protocols under transient faults. *IEEE Transactions on Reliability*, 42(1):166–127.

Profibus (1991). *General Purpose Field Communication System – Part 2, Profibus.*

Purimetla, B., Sivasankaran, R., Ramamritham, K., and Stankovic, J. (1995). Real-time databases: Issues and applications. In Son, S., editor, *Advances in Real-Time Systems.* Prentice-Hall.

Quinn, S. (1996). Unix host and network security tools. Technical report, National Institute of Standards and Technology.

Radestock, M. and Eisenbach, S. (1996). Agent-based configuration management. In *Proceedings of the Seventh IFIP/IEEE International Workshop on Distributed Systems: Operation and Management.*

Rajkumar, R., Gagliardi, M., and Sha, L. (1995). The real-time publisher/ subscriber interprocess communication model for distributed real-time systems: Design and implementation. In *Proceedings of RTAS'95, the IEEE Real-Time Technology and Applications Symposium.*

Ramamritham, K. (1995). The origin of tcs. In *Proceedings of the First ACM International Workshop on Active and Real-Time Database Systems*, pages 50–62, Skovde,Sweden. Springer-Verlag.

Ramamritham, K. (1996a). Dynamic priority scheduling. In Joseph, M., editor, *Real-Time Systems.* Prentice Hall.

Ramamritham, K. (1996b). Real-time databases. *International Journal of Distributed and Parallel Databases*, 1(2):199–226.

Ramamritham, K., Stankovic, J., and Zhao, W. (1989). Distributed sheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, 38(8):1110–1123.

Ramanathan, P., Kandlur, D., and Shin, K. (1990). Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems. *IEEE Trans. Computers*, C-39(4):514–524.

RAND (1955). *A Million Random Digits with 100,000 Normal Deviates.* Free Press Publishers.

Randell, B. (1975). System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2).

Raynal, M., Schiper, A., and Toueg, S. (1991). The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350.

Redmond, K. and Smith, T. (1980). *Project Whirlwind - The History of a Pioneer Computer.* Digital Press.

Reiter, M. (1996). Distributing trust with the rampart toolkit. *Communications of the ACM*, 39(4).

Rennels, D. (1984). Fault-tolerant computing - concepts and examples. *IEEE Transactions on Computers*, 33(12):1116–1129.

Ricart, G. and Agrawala, A. (1981). An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17.

Ricciulli, L. and Shacham, N. (1997). Modelling correlated alarms in network management systems. In *Procs. of the Conference on Communication Networks and Distributed Syst. Modeling and Simulation, CNDS'97*, pages 9–16.

Rivest, R. (1992). The MD5 message digest algorithm. Technical Report RFC 1321, IETF.

Rivest, R. and Shamir, A. (1984). How to expose an eavesdropper. *Communications of the ACM*, 27(4):393–395.

Rivest, R., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2).

Rodrigues, L., Fonseca, H., and Veríssimo, P. (1996). Totally ordered multicast in large-scale systems. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems*, pages 503–510, Hong Kong.

Rodrigues, L., Guerraoui, R., and Schiper, A. (1998a). Scalable atomic multicast. In *Proc. of the Seventh IEEE International Conf. on Computer Communications and Networks (IC3N'98)*, pages 840–847, Lafayette, USA.

Rodrigues, L., Guimarães, M., and Rufino, J. (1998b). Fault-tolerant clocks synchronization in can. In *Proc. of the 19th IEEE Real-Time Systems Symp.*

Rodrigues, L. and Raynal, M. (2000). Atomic broadcast in asynchronous crash-recovery distributed systems. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS'20)*, pages 288–295, Taipe, Taiwan.

Rodrigues, L., Siegel, E., and Veríssimo, P. (1994). A Replication-Transparent Remote Invocation Protocol. In *Proceedings of the 13th IEEE Symposium on Reliable Distributed Systems*, Dana Point, California.

Rodrigues, L. and Veríssimo, P. (1995). Causal separators for large-scale multicast communication. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 83–91, Vancouver, British Columbia, Canada.

Rodrigues, L. and Veríssimo, P. (2000). Topology-aware algorithms for large-scale communication. In Krakowiak, S. and Shrivastava, S., editors, *Advances in Distributed Systems*, LNCS 1752, chapter 6, pages 127–156. Springer-Verlag.

Rodrigues, L., Veríssimo, P., and Rufino, J. (1993). A low-level processor group membership protocol for LANs. In *Proc. of the 13th IEEE International Conference on Distributed Computing Systems*, pages 541–550, Pittsburgh, USA.

Rodrigues, L. and Veríssimo, P. (1992). xAMp: a Multi-primitive Group Communications Service. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, Houston, Texas.

Rom, R. (1988). A reconfiguration algorithm for a double-loop token-ring local area network. *IEEE Transactions on Computers*, 37(2).

Romão, A. (1994). Tools for DNS debugging. Technical Report RFC 1713, USc Inf. S. Inst.

RTS Journal (1997). The challenge of global time in large-scale distributed real-time systems, Schmid,U., ed. *Special Issue of the Journal of Real-Time Systems*, 12(1-3).

Rufino, J., Veríssimo, P., and Arroz, G. (1999). A Columbus' egg idea for CAN media redundancy. In *Digest of Papers, The 29th IEEE International Symposium on Fault-Tolerant Computing Systems*, Madison, Wisconsin - USA.

Rufino, J., Veríssimo, P., Arroz, G., Almeida, C., and Rodrigues, L. (1998). Fault-tolerant broadcasts in CAN. In *Digest of Papers, The 28th IEEE Int'l Symp. on Fault-Tolerant Computing Systems*, Munich, Germany.

Rufino, J. and Veríssimo, P. (1992). A study on the inaccessibility characteristics of ISO 8802/4 Token-Bus LANs. In *Proceedings of the IEEE INFOCOM'92 Conference on Computer Communications*, Florence, Italy. also INESC AR 16-92.

Ryan, P., Schneider, S., Roscoe, B., Goldsmith, M., and Lowe, G. (2000). *The Modelling and Analysis of Security Protocols.* Addison-Wesley.

Sahai, A. and Morin, C. (1998). Towards distributed and dynamic network management. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, New Orleans, USA.

Saksena, M., da Silva, J., and Agrawala, A. (1995). Design and implementation of Maruti-II. In Son, S., editor, *Advances in R-T Systems.* Prentice-Hall.

Saltzer, J. and Schroeder, M. (1975). The protection of information in computing systems. *Proceedings of the IEEE*, 63(9):1278–1308.

Sandberg, R. (1985). The sun network filesystem: Design, implementation and experience. In *Proc. of the Summer 1985 USENIX Confer.*, pages 119–130.

Schell, R. (1984). Security kernel design principles. Technical Report 84-2-7, Auerbach.

Schmid, U. and Schossmaier, K. (1997). Interval-based clock synchronization. *Journal of Real-Time Systems*, 12(2):173–228.

Schneider, F. (1987). Understanding protocols for byzantine clock synchronization. Technical report, Cornell University, Ithaca, New York.

Schneider, F. (1993). Replication management using the state-machine approach. In Mullender, S., editor, *Distributed Systems, 2nd Edition*, ACM-Press, chapter 7. Addison-Wesley.

Schneider, F., Gries, D., and Schlichting, R. (1984). Fault-tolerant broadcasts. *Science of Computer Programming*, (4):1–15.

Schneier, B. (1996). *Applied Cryptography, 2nd edition*. John Wiley.

Schonwalder, J. and Toet, M. (1997). Management of the world-wide web. In *Proceedings of the 8th IFIP/IEEE Int'l Workshop on Distributed Systems Operations and Management (DSOM'97)*, Sydney, Australia.

Schossmaier, K., Schmid, U., Horauer, M., and Loy, D. (1997). Specification and implementation of the universal time coordinated synchronization unit (UTCSU). *Journal of Real-Time Systems*, 12(3).

Sha, L., Rajkumar, R., and Lehoczky, J. (1990). Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Computers*, 39:1175–1185.

Shin, K. (1991). Harts: Distributed real-time architecture. *IEEE Computer*, 24(5):25–35.

Shirley, J., Hu, W., Magid, D., and Oram, A. (1994). *Guide to Writing Dce Applications*. Number ISBN: 1565920457. O'Reilly & Associates.

Shrivastava, S., Dixon, G., and Parrington, G. (1991). An Overview of the Arjuna Distributed Programming System. *IEEE Software*.

Siamwalla, R., Sharma, R., and Keshav, S. (1999). Discovering internet topology. In *Proceedings of the IEEE Infocom'99*, pages 21–25.

Silberschatz, A., Galvin, P., and Gagne, G. (2000). *Applied Operating System Concepts*. Number ISBN: 0471365084. Wiley.

Silva, L. and Silva, J. (1992). Global checkpointing for distributed programs. In *Procs. of the 11th IEEE Symp. on Reliable Distributed Systems*, pages 155–164.

Sinha, P. and Suri, N. (1999). On the use of formal techniques for analyzing dependable real-time protocols. In *Proc. of the 20th IEEE Real-Time Systems Symp.*

Siqueira, F. and Cahill, V. (2000). An open qos architecture for corba applications. In *Proc. of ISORC 2000, the Third IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing*, pages 328–335, Newport Beach, USA.

Skeen, D. (1985). Determining the last process to fail. *ACM Trans. on Computer Systems*, 3(1).

Slade, R. (1995). *Computer Viruses, 2nd edition*. Springer-Verlag.

Sloman, M., editor (1994). *Network and Distributed Systems Management*. Addison-Wesley.

Sloman, M. and Twidle, K. (1994). Domains: a framework for structuring management policy. In Sloman, M., editor, *Network and Distributed Systems Management*, chapter 16. Addison-Wesley.

Solomon, M., Landweber, L., and Neuhengen, D. (1982). The csnet name server. *Computer Networks*, 6(3):161–172.

Son, S. (1987). Using replication for high performance database support in distributed real-time systems. In *Proc. of RTSS'87, the 8th IEEE Real-Time Systems Symp.*

Song, X. and Liu, J. (1992). How well can data temporal consistency be maintained? In *Proceedings of the IEEE Symposium on Computer-Aided Control Systems Design.*

Spainhour, S. and Quercia, V. (1996). *Webmaster in a Nutshell.* O'Reilly.

Spector, A. (1987). Camelot: a distributed transaction facility for Mach and the Internet — an interim report. Research paper. CMU-CS-87-129, Carnegie Mellon University, CS Dept., Pittsburgh, PA, USA.

Speirs, N. and Barrett, P. (1989). Using passive replicates in Delta-4 to provide dependable distributed computing. In *Digest of Papers, The 19th IEEE International Symposium on Fault-Tolerant Computing*, Chicago-USA.

Sprunt, B., Sha, L., and Lehoczky, J. (1989). Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60.

Srikanth, T. Kand Toueg, S. (1987). Optimal Clock Synchronization. *Journal of the Association for Computing Machinery*, 34(3):627–645.

Stallings, W. (1998). Security comes to snmp: The new snmpv3 proposed internet standard. *The Internet Protocol Journal*, 1(3):2–12.

Stallings, W. (1999). *Cryptography and Network Security: Principles and Practice, 2nd Ed.* Prentice-Hall.

Stankovic, J. (1988). Misconceptions about real-time computing. *IEEE Computer.*

Stankovic, J. and Ramamritham, K. (1991). The Spring Kernel: A New Paradigm for Real-time Systems. *IEEE Software.*

Steiner, J., Neumann, C., and Schiller, J. (1988). Kerberos: An authentication service for open network systems. In *Proc. of USENIX Winter Conference*, pages 191–202.

Steiner, M., Tsudik, G., and Waidner, M. (1998). CLIQUES: A new approach to group key agreement. In *Proc. 18th IEEE International Conference on Distributed Computing Systems (ICDCS'98)*, pages 380–387, Amsterdam.

Stephenson, P. (1991). *Fast Causal Multicast.* PhD thesis, Cornell Univ.

Steusloff, H. (1981). The impact of distributed computer control systems on software. *Pergamon Press.*

Stewart, J. (1999). *BGP4.* Addison-Wesley.

Stiffler, J. (1978). Fault coverage and the point of diminishing returns. *Journal of Design Automation & Fault Tolerance Computing*, 2(4).

Strom, R. and Yemini, S. (1985). Optimistic recovery in distributed systems. *ACM Trans. on Computer Systems*, 3(3):204–226.

Suri, N., Hughe, M., and Walter, C. (1994). Synchronization issues in real-time systems. In *Proceedings of IEEE, Special Issue on Real Time Systems.*

Tanenbaum, A. (1992). *Modern Operating Systems.* Prentice-Hall.

Tanenbaum, A. (1995). *Distributed Operating Systems.* Prentice-Hall.

Tanenbaum, A. (1996). *Computer Networks.* Prentice-Hall, 3rd edition.

TCSEC (1985). *Trusted Computer System Evaluation Criteria.*

Thomas, R. (1979). A Majority Concensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2):180–209.

Thompson, J. (1998). Web-based enterprise management architecture. *IEEE Communications Magazine*, 36(3):80–86.

Tindell, K. (1994). *Fixed Priority Scheduling of Hard Real-Time Systems.* PhD thesis, University of York, UK.

Tindell, K. and Burns, A. (1994). Guaranteeing message latencies on Controler Area Network. In *Proc. of the 1st Int'l CAN Conference*, Mainz, Germany. CiA.

Tindell, K., Burns, A., and Wellings, A. (1994). Calculating Controller Area Network (CAN) message response times. In *Proceedings of the IFAC Workshop on Distributed Computer Control Systems*, Toledo, Spain.

Tindell, K., Burns, A., and Wellings, A. (1995). Analysis of hard real-time communications. *Real-Time Systems*, 9(2):147–171.

Token Bus (1985). *Token Passing Bus Access Method.*

Tokuda, H., Nakajima, T., and Rao, P. (1990). Real-time mach: Towards a predictable real-time system. In *Proceedings of the USENIX Mach Workshop.*

Tovar, E., Vasques, F., and Burns, A. (1999). Adding local priority-based dispatching mechanisms to p-net networks: A fixed priority approach. In *Proceedings of the 11th IEEE Euromicro Conference on Real-Time Systems*, pages 175 – 184, York, England.

Tschichholz, M., Tschammer, V., and Dittrich, A. (1996). Integrated approach to open distributed management. *Computer Communications*, (19):76–87.

Turek, J. and Shasha, D. (1992). The many faces of consensus in distributed systems. *IEEE Computer*, 25(6):8.

van Renesse, R., Birman, K., and Maffeis, S. (1996). Horus: A flexible group communications system. *Communications of the ACM*, 39(4):76–83.

Veríssimo, P. (1996). Causal delivery protocols in real-time systems: A generic model. *Journal of Real-Time Systems*, 10(1):45–73.

Veríssimo, P. and Almeida, C. (1995). Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models. *Bulletin of the Tech. Commit. on Operating Systems and Application Environments (TCOS)*, 7(4):35–39.

Veríssimo, P., Rodrigues, L., and Baptista, M. (1989). AMp: A highly parallel atomic multicast protocol. In *Proceedings of the ACM SIGCOM'89 Symposium*, pages 83–93, Austin, USA.

Veríssimo, P., Rodrigues, L., and Casimiro, A. (1997). Cesiumspray: a precise and accurate global clock service for large-scale systems. *Journal of Real-Time Systems*, 12(3):243–294.

Veríssimo, P. (1988). Redundant media mechanisms for dependable communication in token-bus LANs. In *Proceedings of the 13th IEEE Local Computer Network Conference*, Minneapolis-USA.

Veríssimo, P. (1993). Real-time Communication. In Mullender, S., editor, *Distributed Systems, 2nd Ed.*, ACM-Press, chapter 17, pages 447–490. Addison-Wesley.

Veríssimo, P. (1994). Ordering and Timeliness Requirements of Dependable Real-Time Programs. *Journal of Real-Time Systems, Kluwer*, 7(2):105–128.

Veríssimo, P., Barrett, P., Bond, P., Hilborne, A., Rodrigues, L., and Seaton, D. (1991). The Extra Performance Architecture (XPA). In Powell, D., editor, *Delta-4 - A Generic Architecture for Dependable Distributed Computing*, ESPRIT Research Reports, pages 211–266. Springer Verlag.

Veríssimo, P., Casimiro, A., and Fetzer, C. (2000). The timely computing base: Timely actions in the presence of uncertain timeliness. In *Procs. of DSN 2000, the IEEE/IFIP Int'l Conf. on Dependable Systems and Networks.*

Veríssimo, P. and Marques, J. (1990). Reliable broadcast for fault-tolerance on local computer networks. In *Procs. of the 9th IEEE Symp. on Reliable Distributed Systems*, Huntsville, Alabama-USA. Also INESC AR/24-90.

Veríssimo, P., Melro, S., Casimiro, A., and Silva, L. (1996). Distributed industrial information systems: Design and experience. In *Proceedings of BASYS'96, the 2nd IEEE/IFI International Conference on Information Technology for Balanced Automation SYStems in Manufacturing.*

Veríssimo, P. and Raynal, M. (2000). Time, clocks and temporal order. In Krakowiak, S. and Shrivastava, S., editors, *Recent Advances in Distributed Systems*, volume 1752 of *LNCS*, chapter 1. Springer-Verlag.

Veríssimo, P. and Rodrigues, L. (1992). A posteriori Agreement for Fault-tolerant Clock Synchronization on Broadcast Networks. In *Digest of Papers, The 22nd IEEE Int'l Symp. on F/T Computing*, Boston - USA.

Veríssimo, P., Rufino, J., and Ming, L. (1997). How hard is hard real-time communication on field-buses? In *Digest of Papers, The 27th IEEE International Symposium on Fault-Tolerant Computing*, Seattle - USA.

Vogels, W., Rodrigues, L., and Veríssimo, P. (1992). Fast group communication for standard workstations. In *Proceedings of the OpenForum'92 Technical Conference*, Utrecht, the Netherlands. EurOpen, UniForum.

Volg, C., Wolf, L., Herrtwich, R., and Wittig, H. (1996). Heirat - quality of service management for distributed multimedia systems. *Multimedia Systems Journal.*

Wakerly, J. (1978). *Error detecting codes, self-checking circuits and applications.* North Holland.

Waldo, J. (1999). Jini technology architectural overview. Technical report, Sun Microsystems.

Wang, Y.-M. and Fuchs, W. (1992). Optimistic message logging for independent checkpointing in message-passing systems. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, pages 147–154.

Watt, A. and Watt, M. (1992). *Advanced animation and rendering techniques - Theory and practice.* Addison-Wesley, New York.

Wayner, P. (1993). Should encryption be regulated? *Byte.*

Wellings, A., Beus-Dukic, L., and Powell, D. (1998). Real-time scheduling in a generic fault-tolerant architecture. In *Proceedings of RTSS'98, the IEEE Real-Time Systems Symposium*, pages 390–398, Madrid, Spain.

Wensley, J. H., Lamport, L., Goldberg, J., Green, M. W., Levitt, K. N., Melliar-Smith, P. M., Shostak, R. E., and Weinstock, C. B. (1978). SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255.

Wies, R. (1994). Policies in network and systems management - formal definition and architecture. *Journal of Network and Sys. Management*, 2(1):63–83.

Wikander, J.and Svensson, B., editor (1998). *Real-Time Systems in Mechatronic Applications.* Kluwer Academic Publishers.

Wilson, D. (1985). The stratus computer system. In *Resilient Computing Systems*, pages 208–231.

Wood, M. (1991). *Fault-Tolerant Management of Distributed Applications Using a Reactive System Architecture.* PhD thesis, Cornell University, USA.

Wood, M. (1993). Replicated RPC using Amoeba closed group communication. In *Proceedings of the 13th IEEE International Conference on Distributed Computing Systems*, pages 499–507, Pittsburgh, Pennsylvania, USA.

X.509 (1997). *Information technology - Open Systems Interconnection– The Directory: Authentication Framework.*

XTP (1998). *The Xpress Transport Protocol Specification.* XTP Forum Inc., 1394 Greenworth Place, Santa Barbara, USA.

Xu, J., Randell, B., Romanovsky, A., Stroud, R. Zorzo, A., Canver, E., and von Henke, F. (1999). Rigorous development of a safety-critical system based on coordinated atomic actions. In *Digest of Papers, The 29th IEEE International Symposium on Fault-Tolerant Computing Systems*, pages 68–75, Madison- USA.

Xu, J., Randell, B., Rubira-Calsavara, C., and Stroud, R. (1995). Toward an object-oriented approach to software fault tolerance. In *Recent Advances in Fault-Tolerant Parallel and Distributed Systems*, Computer Society Press, pages 226–233. IEEE.

Yau, S. and Karim, F. (2000). Component customization for object-oriented distributed real-time software development. In *Proceedings of ISORC 2000, the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 156–163, Newport Beach, USA.

Zeltserman, D. and Puoplo, G. (1998). *Building Network Management Tools with Tcl/Tk*. Prentice-Hall.

Zhang, T. and Covaci, S. (1997). Java-based mobile intelligent agents as network management solutions. In *Proceedings of the 8th Joint European Conference (JENC8)*, pages 12–15, Edinburgh, Scotland.

Zheng, Q. and Shin, K. G. (1992). Fault-tolerant real-time communication in distributed computing systems. In *Digest of Papers, The 22nd IEEE International Symposium on Fault-Tolerant Computing Systems*, pages 86–93.

Zimmermann, H. (1980). OSI Reference model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, COM-28(4):425–432.

Zimmermann, P. (1995). *The Official PGP User's Guide*. MIT Press.

Znaty, S., Genilloud, G., Gaspoz, J.-P., and Hubaux, J.-P. (1995). Networked systems: Introducing network management models into odp. In *Proceedings of the IEEE Globecom'95*, pages 121–126.

Znaty, S. and Hubaux, J.-P. (1997). Telecommunications services engineering: Principles, architectures and tools. In *Procs. of the ECOOP'97*, Finland.

Zuberi, K. and Shin, K. (1995). Non-preemptive scheduling of messages on controller area networks for real-time control applications. In *Procs. of RTAS'95, the IEEE Real-Time Technology and Applications Symp.*

Index

About the Authors

Paulo Veríssimo Paulo Veríssimo is a professor of the Department of Informatics, University of Lisboa Faculty of Sciences. He leads the Navigators research group, at the University of Lisboa. He has been in the coordinating team of several major national projects in informatics, and headed the participation of the group in several CEC ESPRIT projects. He belonged to the Executive Board of the CABERNET- ESPRIT Network of Excellence. Paulo Veríssimo is a member of Ordem dos Engenheiros, IEEE and ACM. He has served as program co-chair of the (ex-FTCS/DCCA) IEEE DSN 2001 conference and on the programme committees of several other conferences (IEEE, AFCET, ACM), and is an associate editor for the Telecommunications Systems Journal (Baltzer). He is author of more than 80 refereed publications in international scientific conferences and journals, and over a 100 technical reports. He is co-author of four books in distributed systems and dependability. He organised and lectured in the LISBOA'92 Advanced Course on Distributed Systems, and was director of the 3rd European Seminar on Advances on Distributed Systems, ERSADS 99.

Luís Rodrigues Luís Rodrigues graduated (1986), has a Master (1991) and a PhD (1996) in Electrotechnic and Computers Engineering, by the Instituto Superior Técnico de Lisboa (IST). He is Assistant Professor at Department of Informatics, University of Lisboa Faculty of Sciences. Previously he was at the Electrotechnic and Computers Engineering Department of Instituto Superior Técnico de Lisboa (he joined IST in 1989). From 1986 to 1996 he was a member of the Distributed Systems and Industrial Automation Group at INESC. Since 1996, he is a senior researcher of the Navigators group and a founding member of the LASIGE laboratory at University of Lisboa. He participated and contributes to several national and international projects such as, Delta-4, Estímulo, Broadcast, GODC, etc. His current interests include fault-tolerant and real-time distributed systems, group membership and communication, and replicated data management. He has more than 40 publications in these areas. He is a member of the Ordem dos Engenheiros, IEEE and ACM.