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

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

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 978-1-4613-5666-0 ISBN 978-1-4615-1663-7 (eBook)
DOI 10.1007/978-1-4615-1663-7

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.

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

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

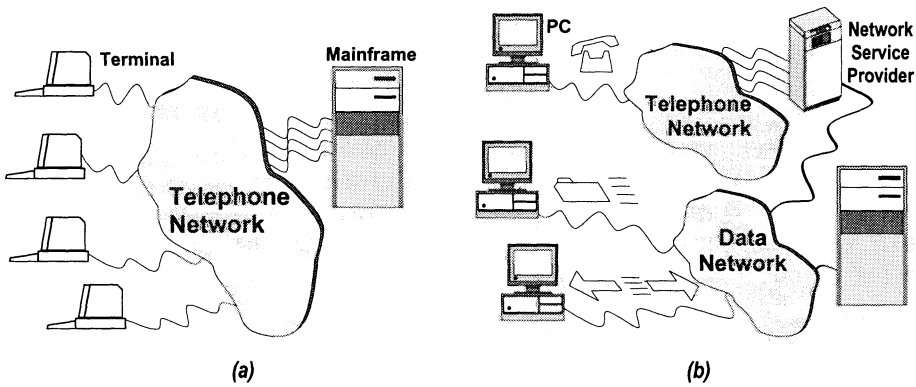

Figure 1.1. Remote Access Architectures: (a) Plain Telephone Line; (b) Data Network

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.

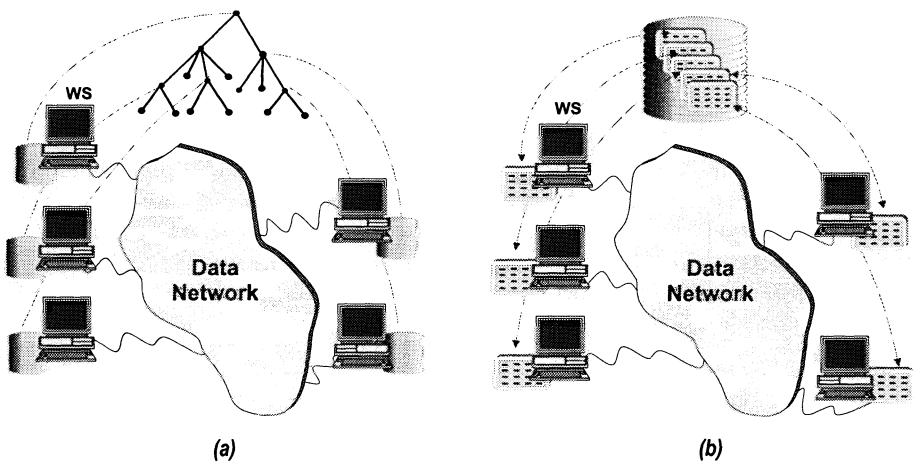

Figure 1.2. Distributed File (a) and Memory (b) Architectures

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.

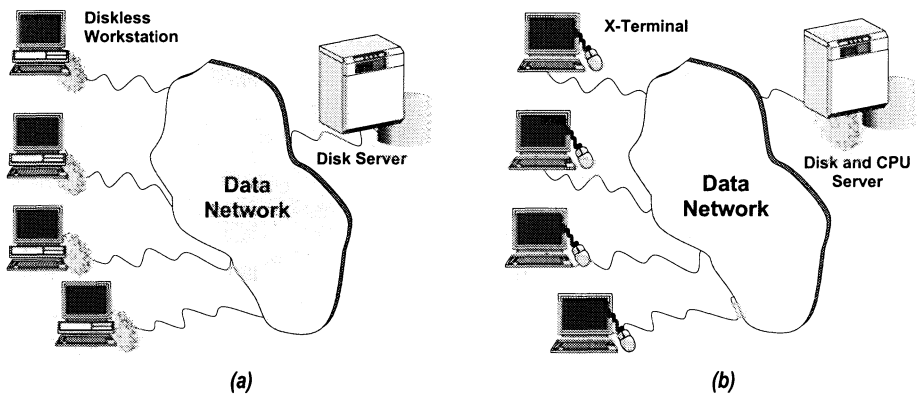

Figure 1.3. Diskless and X-Terminal Architectures

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).

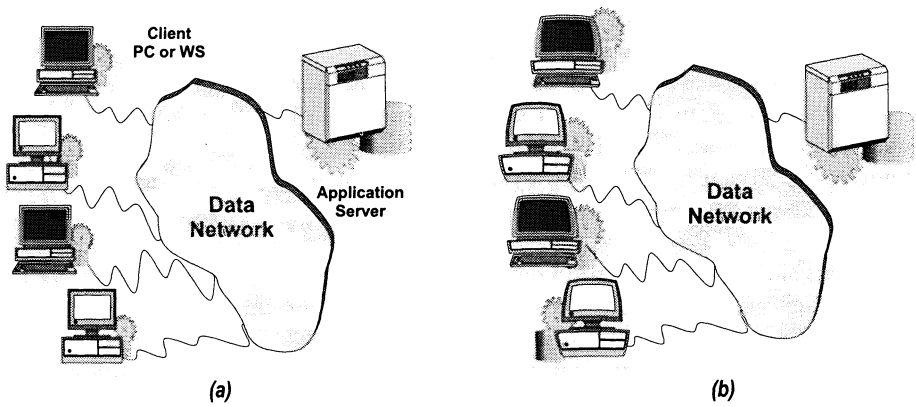

Figure 1.4. Client-Server Architectures

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.

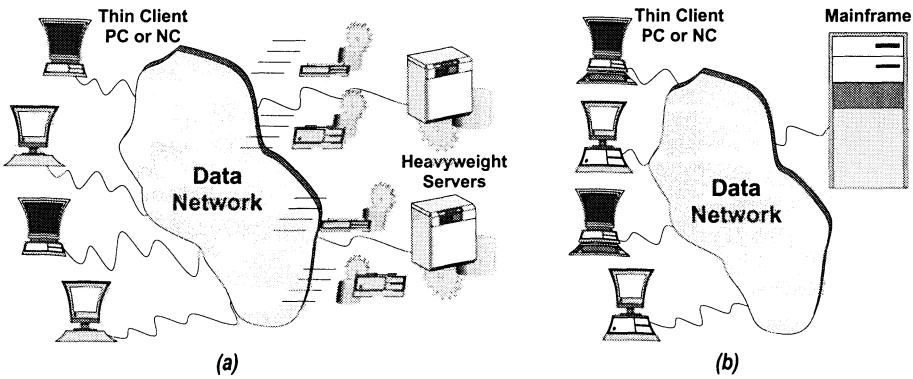

Figure 1.5. 3-tier Client-Server or Thin-Client Architectures

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

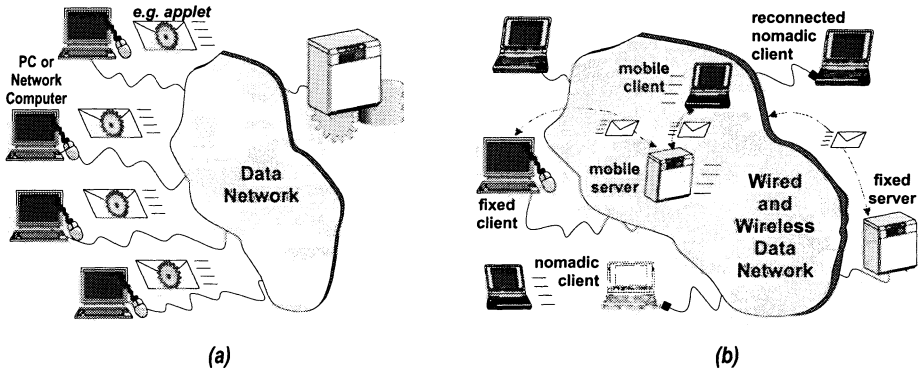

Figure 1.6. Mobile Code Architectures: (a) Portable and Mobile Code; (b) Mobile Nodes

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.

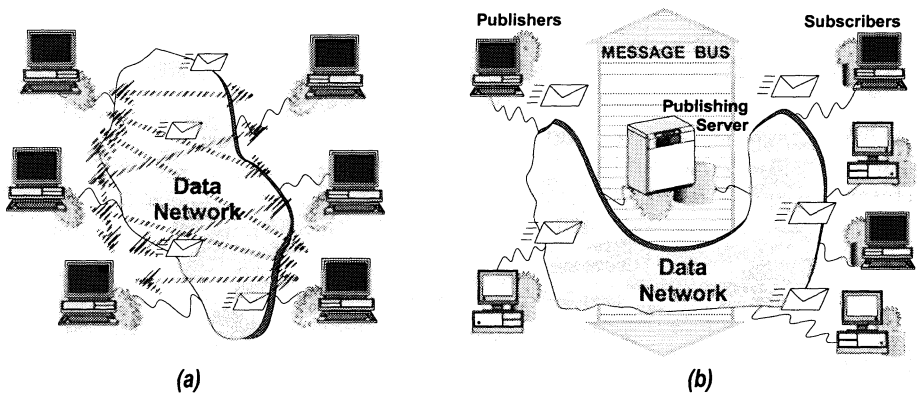

Figure 1.7. Event-based Architectures: (a) Multipeer; (b) Publisher-subscriber

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.

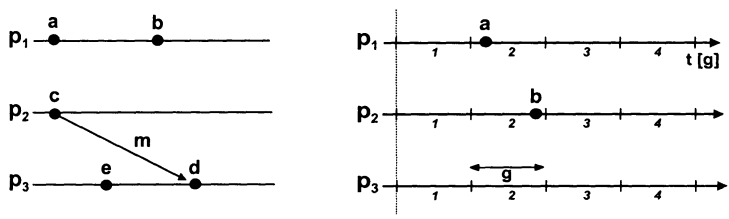

Figure 1.8. Space-Time and Lattice Diagrams

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

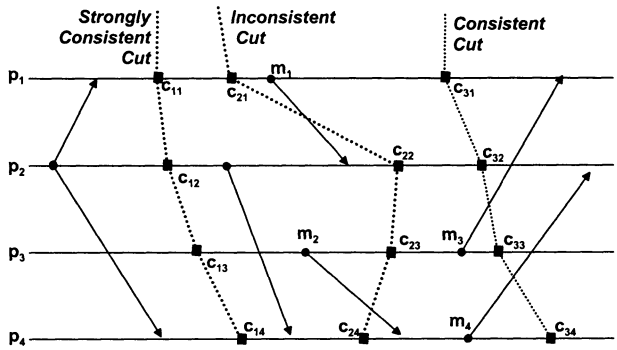

Figure 1.9. Cuts and Global States

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).

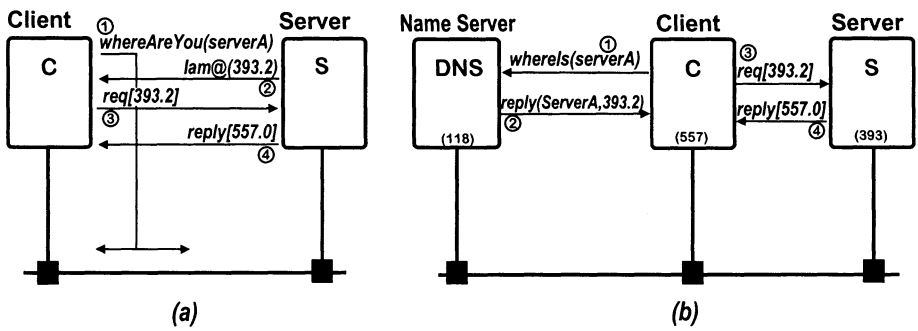

Figure 2.1. Name to Address Translation: (a) Broadcast; (b) Name Server

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.

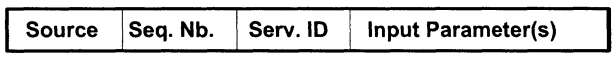

Figure 2.2. Possible Message Format

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

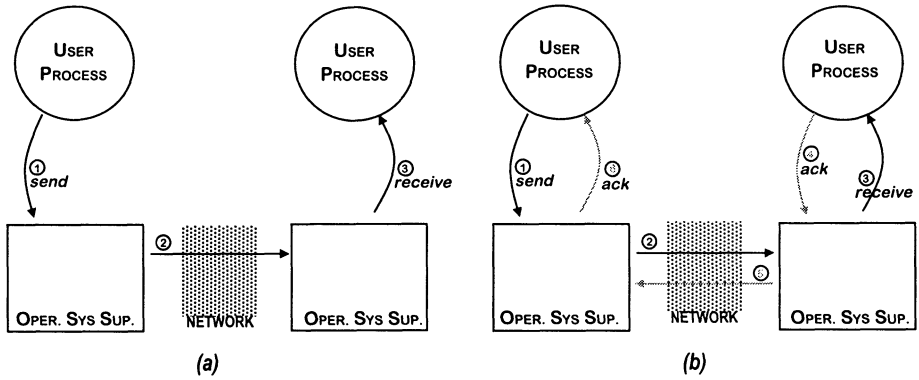

Figure 2.3. Message Passing Protocols: (a) Send-Receive;(b) Acknowledged-Send

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

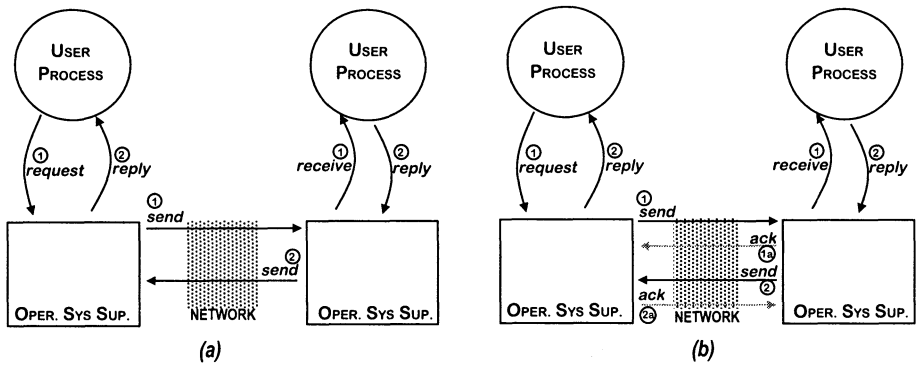

Figure 2.4. Remote Operation Protocols: (a) Plain Request-Reply; (b) Acknowledged

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

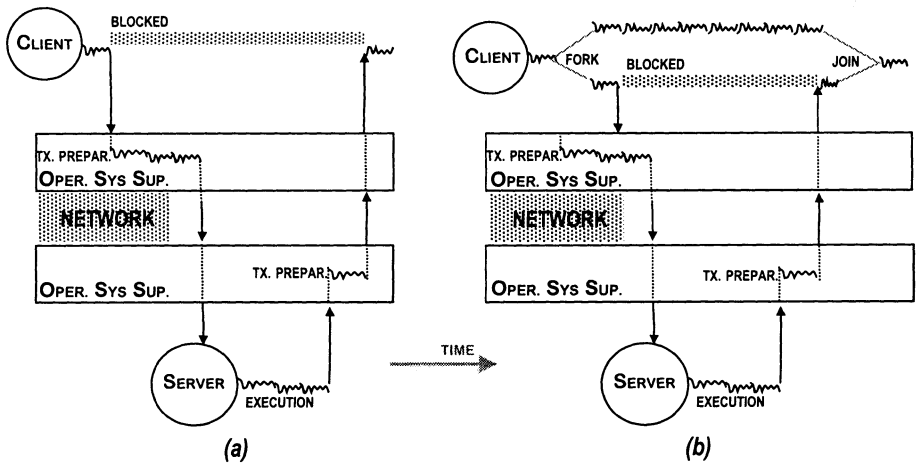

Figure 2.5. Remote Operation Interfaces: (a) Blocking; (b) Non-Blocking

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).

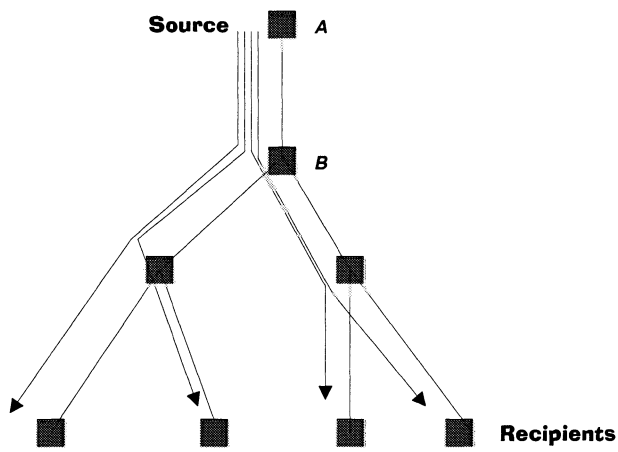

Figure 2.6. Multicast Tree

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.

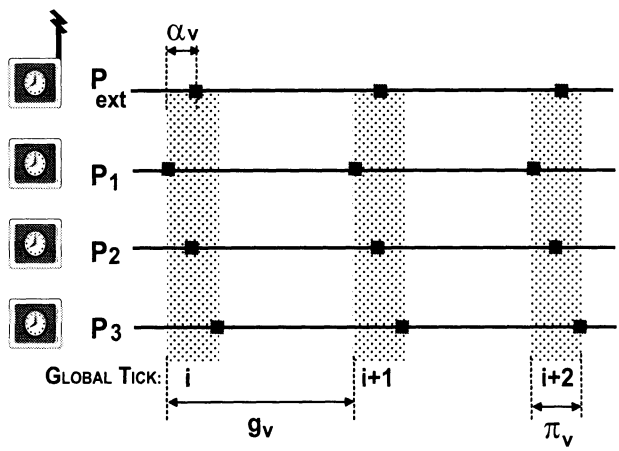

Figure 2.7. Properties of a Global Clock

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:

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

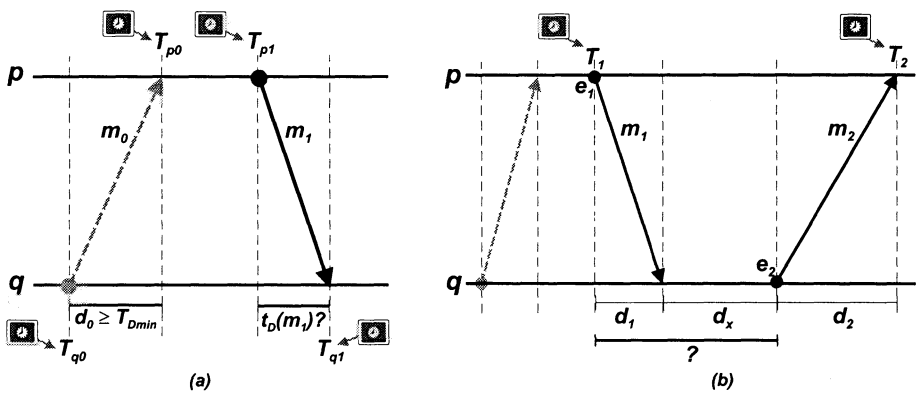

Figure 2.8. Round-trip Duration Measurement: (a) Message delay; (b) Distributed Duration

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.

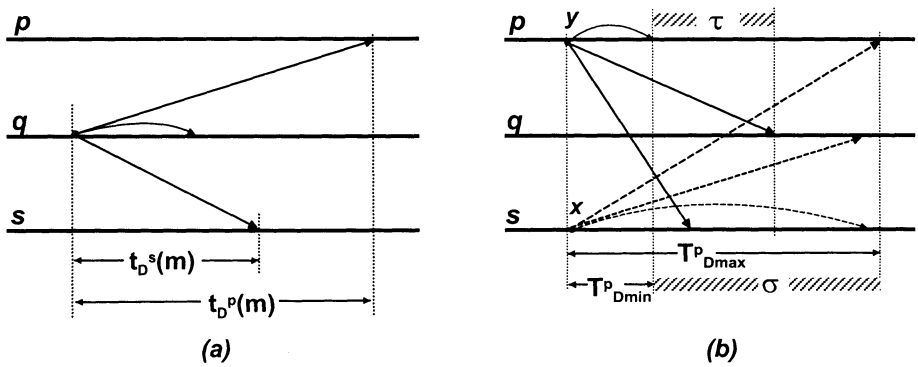

Figure 2.9. Synchronism Metrics: (a) Delivery Time (t_D); (b) Steadiness (σ), Tightness (τ)

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 ($c(m)$);

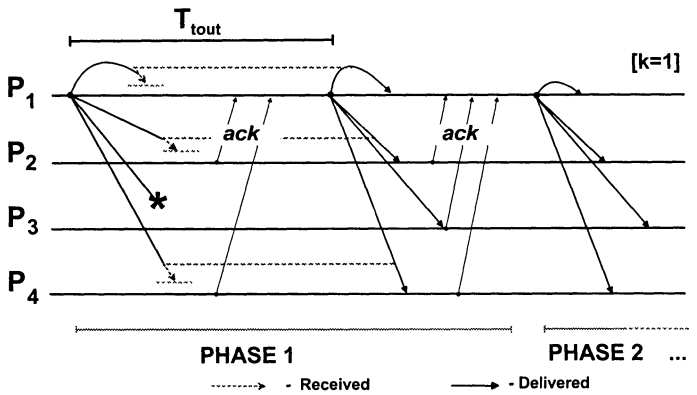

Figure 2.10. Unsteady and Untight Synchronous Protocol

- 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.

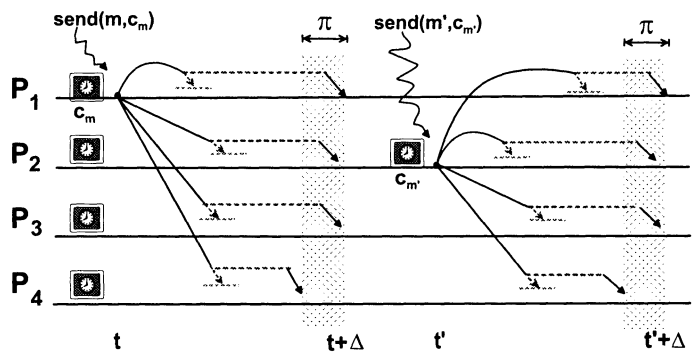

Figure 2.11. Steady and Tight Δ-protocol

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.

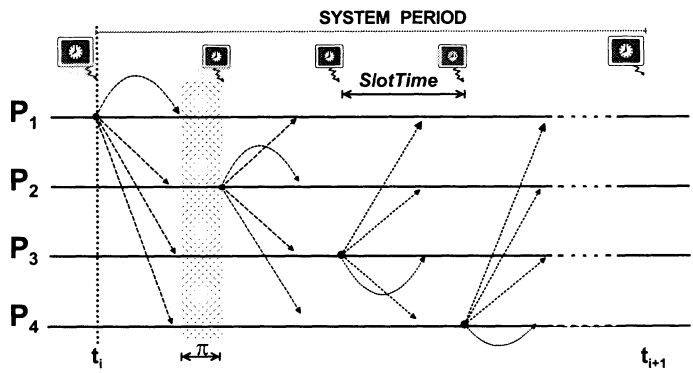

Figure 2.12. Steady and Tight TDMA Protocol

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.

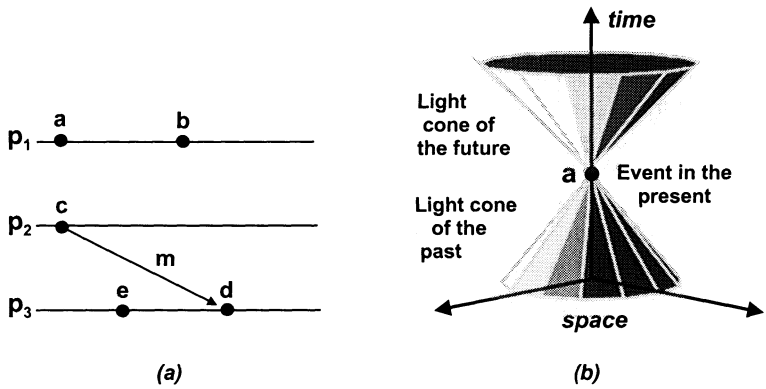

Figure 2.13. Precedence: (a) Space-time View; (b) Relativistic Light Cone

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.

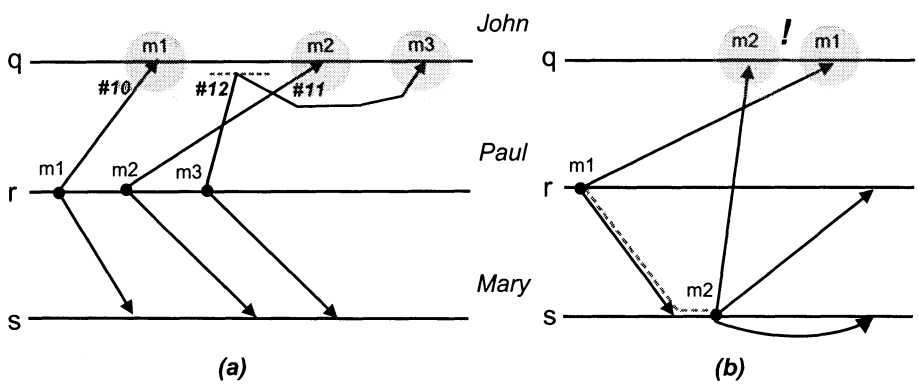

Figure 2.14. FIFO Order: (a) An Example; (b) FIFO Insufficient

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 $m_3 \stackrel{l}{\rightarrow} m_2$

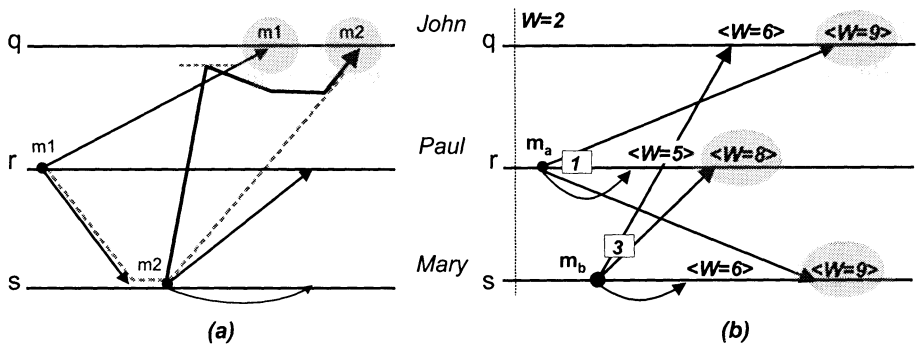

Figure 2.15. Causal Order: (a) An Example; (b) Causal Insufficient

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

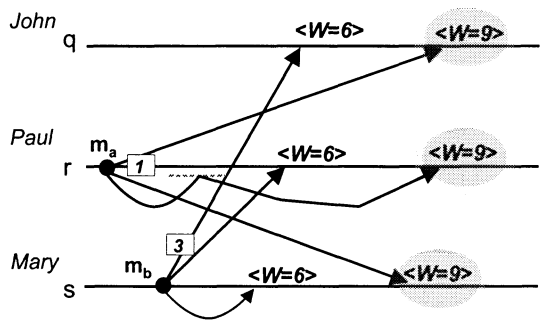

Figure 2.16. Total Order

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.

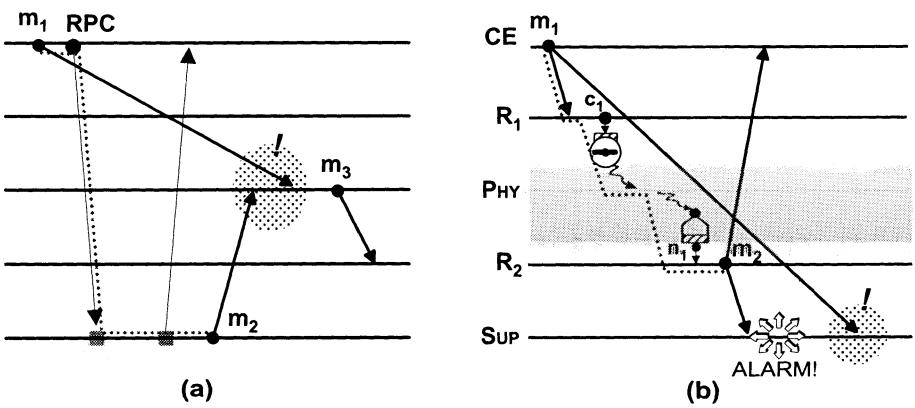

Figure 2.17. Hidden Channel Examples: (a) Other Protocols; (b) Physical Feedback

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):

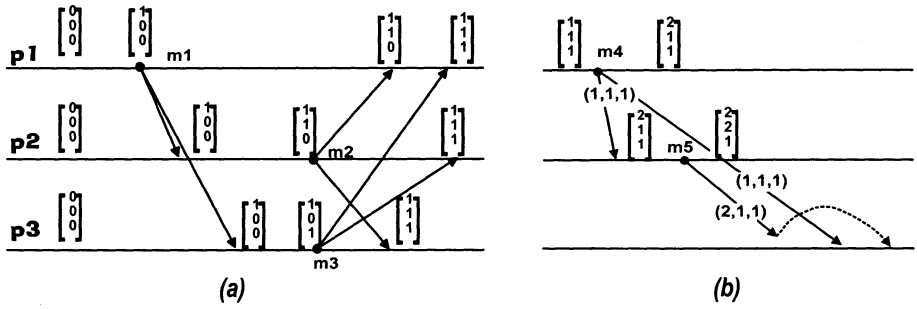

Figure 2.18. Vector Clocks: (a) Principle; (b) Practical Use

- 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

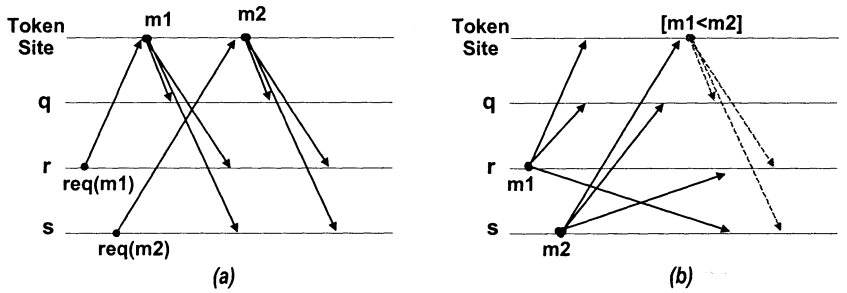

Figure 2.19. Total Ordering with Sequencer: (a) Point-to-Point Send; (b) Broadcast

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).

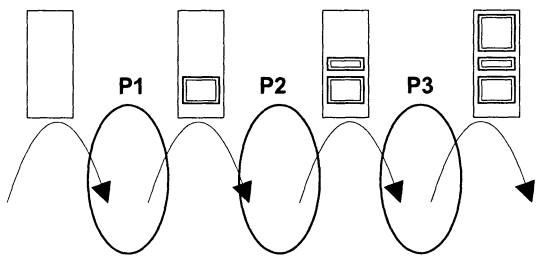

Figure 2.20. Assembly Line of Producer-Consumer

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

```
00  while (1) {
01      item = produce();
02      wait (sem-free, 1);
03      wait (mutex);
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 2.24. 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.

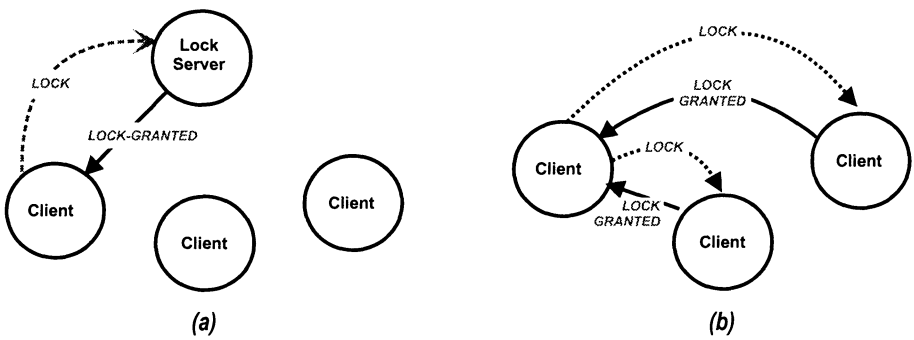

Figure 2.25. Distributed Mutual Exclusion Control: (a) Centralized; (b) Distributed

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.

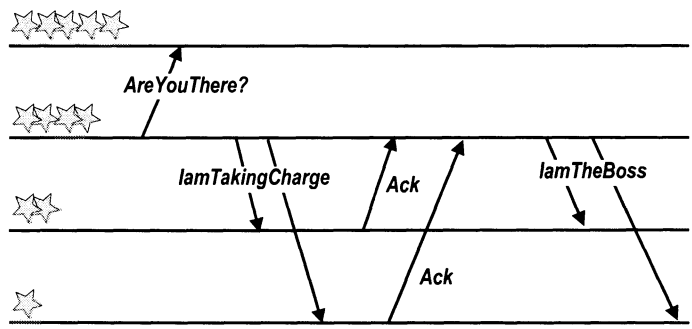

Figure 2.26. Leader Election

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.

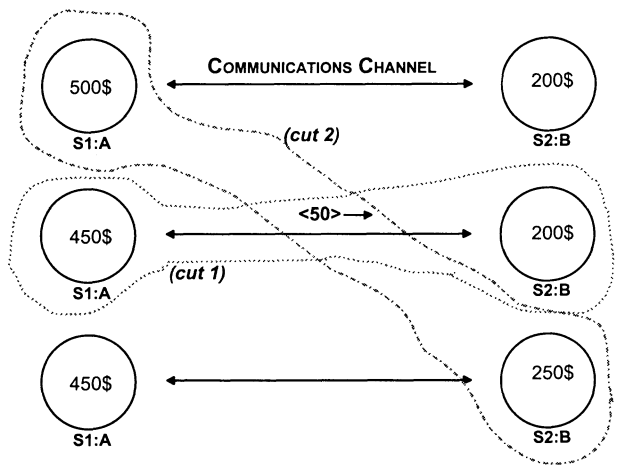

Figure 2.28. Ad-hoc State Snapshots

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.

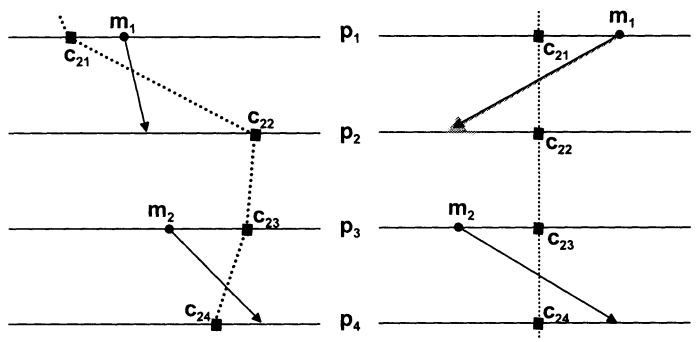

Figure 2.29. Inconsistent Cut

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.

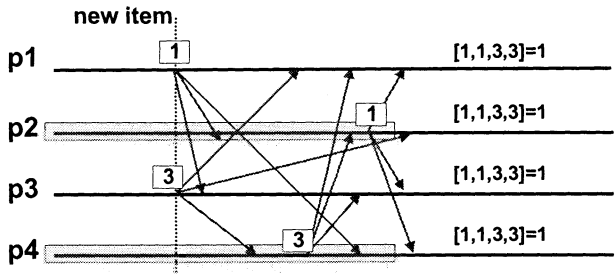

Figure 2.30. Simple Consensus Protocol

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.

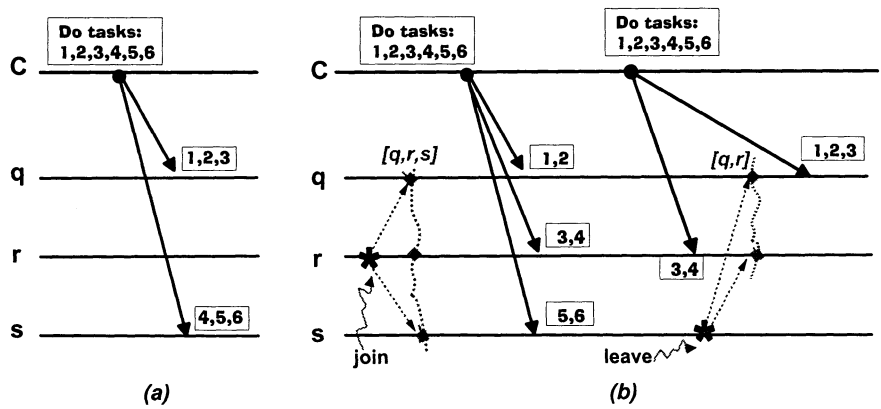

Figure 2.31. Ad-Hoc View Change

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).

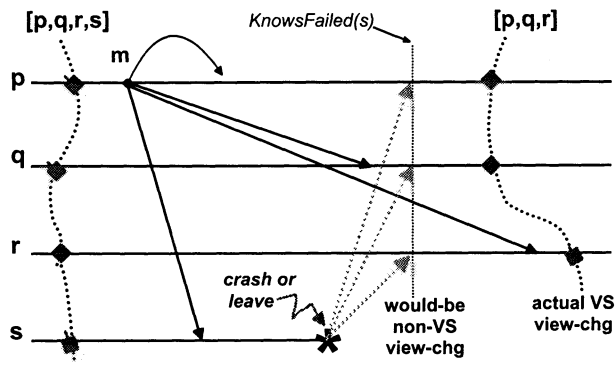

Figure 2.32. View-Synchronous View Change

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.

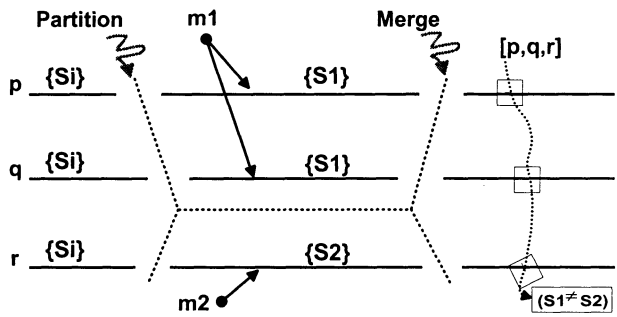

Figure 2.33. State Divergence with Partitioning

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.

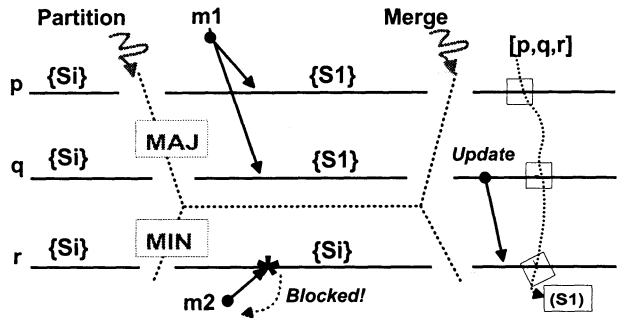

Figure 2.34. Primary Partition

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;

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

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

```
semaphore db-lock;

transaction sumAll begin
    x := 0;
    wait (db-lock);
    x := x + A;
    x := x + B;
    x := x + C;
    x := x + D;
    signal (db-lock);
    print x;
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).

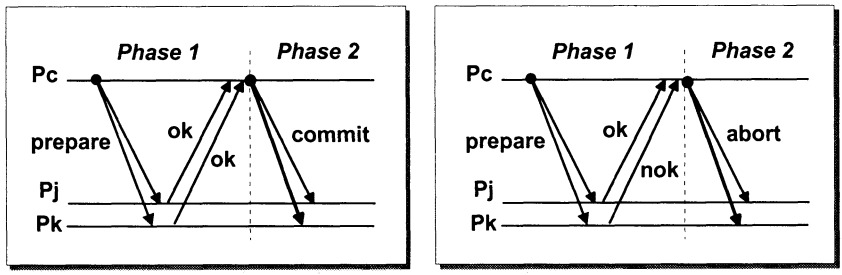

Figure 2.38. Two-phase Commit Protocol: (a) Commit; (b) Abort

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.

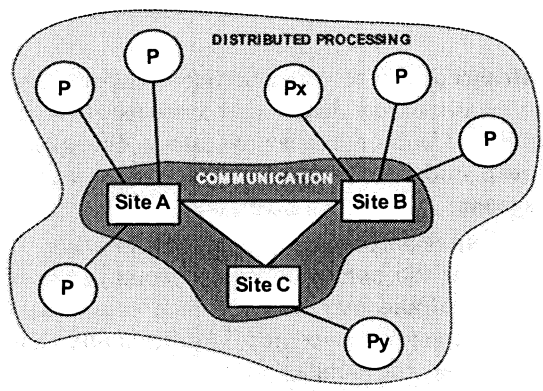

Figure 3.1. Decoupling Communication from Processing with Site-Participant Clustering

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).

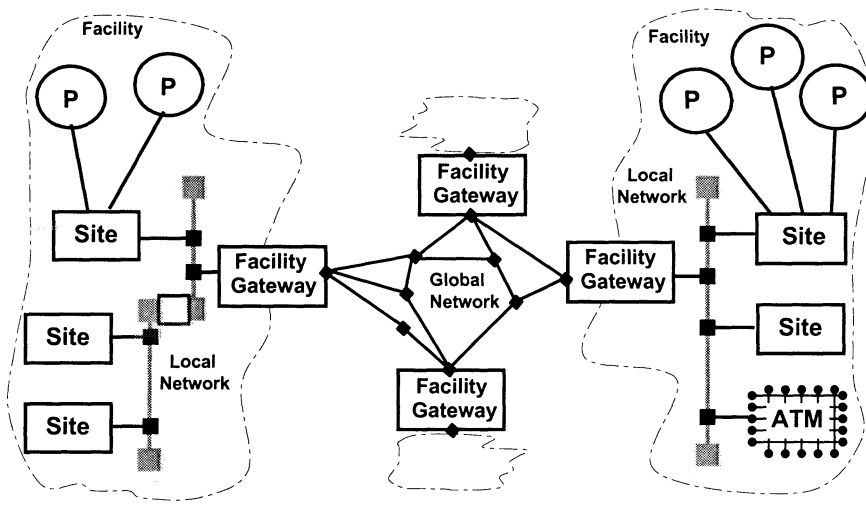

Figure 3.2. 2-tier WAN-of-LANs

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.

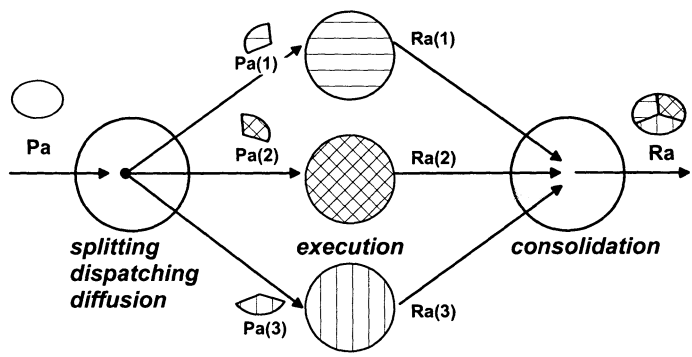

Figure 3.3. Flow Diagram of Coordination Activities

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.

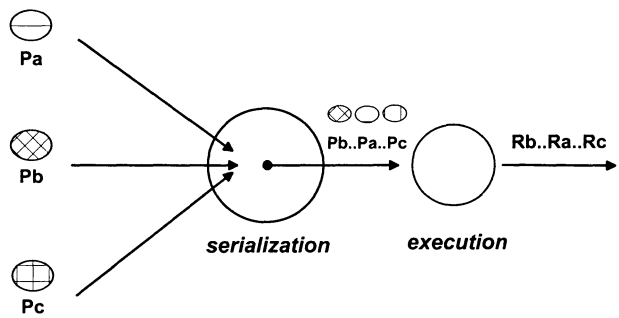

Figure 3.4. Flow Diagram of Sharing Activities

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.

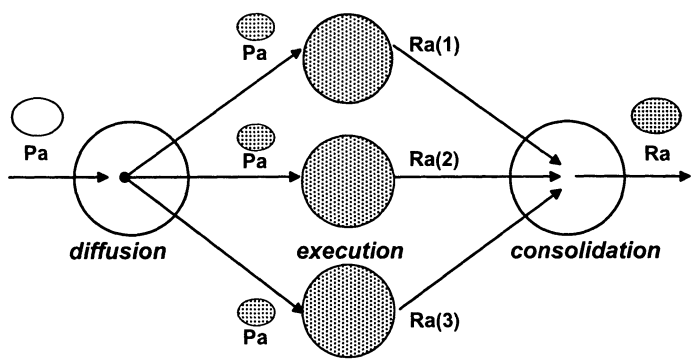

Figure 3.5. Flow Diagram of Replication Activities

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.

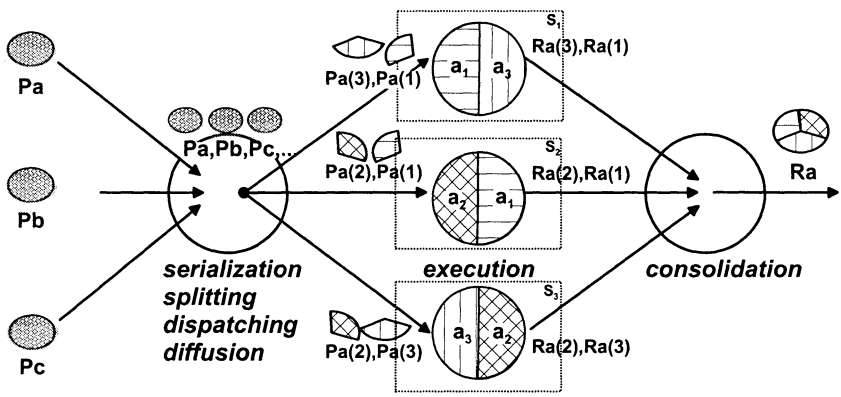

Figure 3.6. Combining Activities

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.

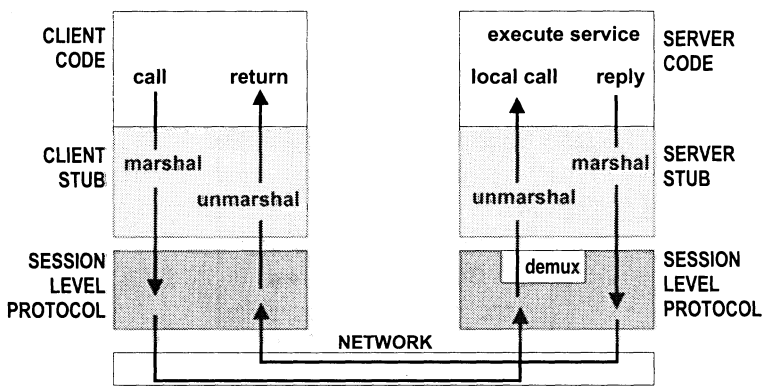

Figure 3.7. RPC in Action

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.

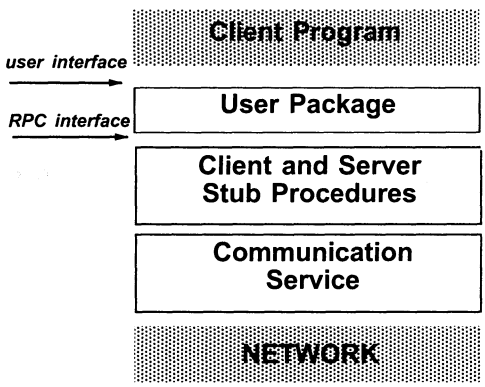

Figure 3.8. RPC Architecture

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.

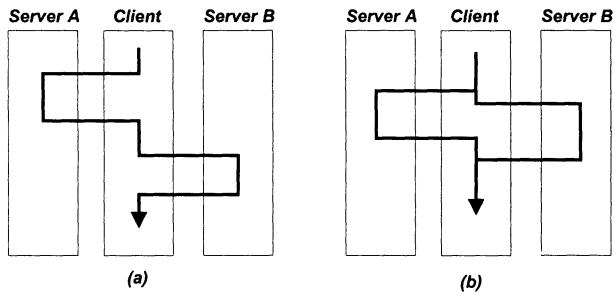

Figure 3.9. Client Threads: (a) Single-threaded; (b) Multi-threaded

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.

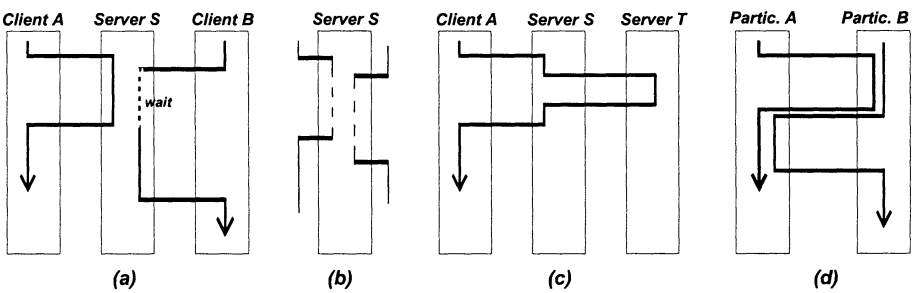

Figure 3.10. Server Threads: (a) Single-threaded; (b) Multi-threaded; (c) Multi-tiered; (d) Conversational

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

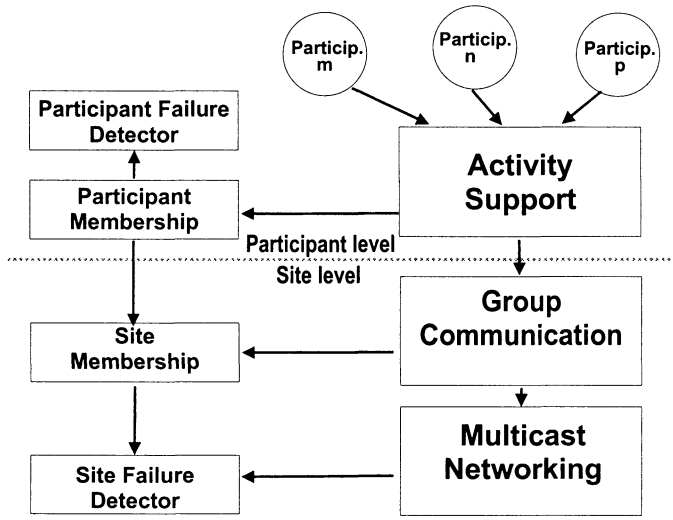

Figure 3.11. Group-Oriented Architecture

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.

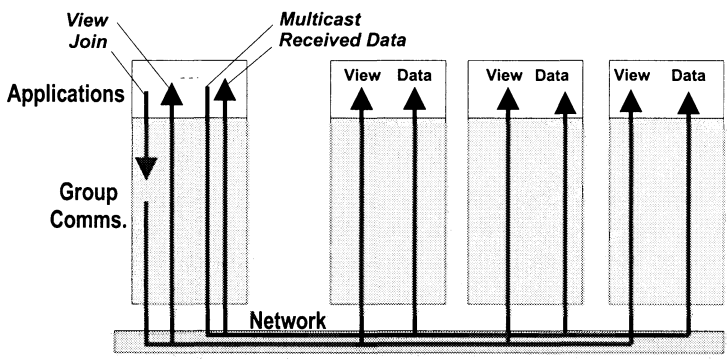

Figure 3.12. Groups in Action

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.

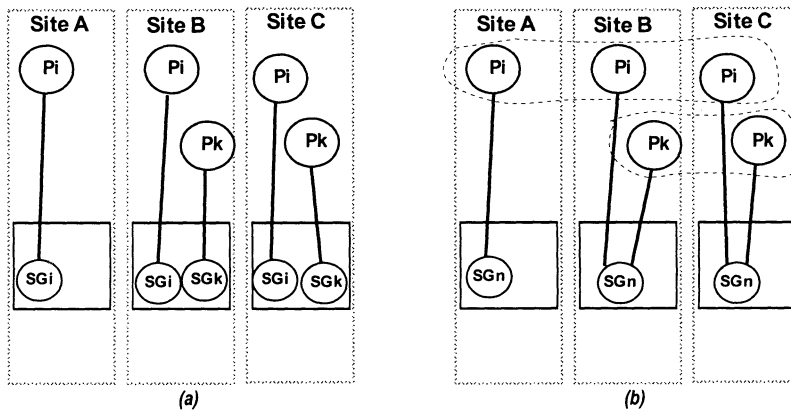

Figure 3.13. Group Access Methods: (a) Normal; (b) Lightweight

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

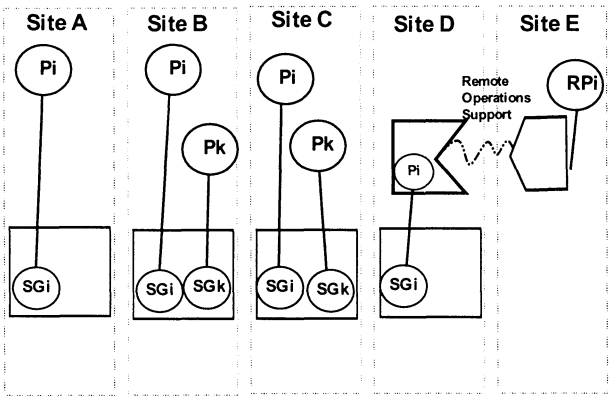

Figure 3.13 *(continued).* Group Access Methods: (c) Remote

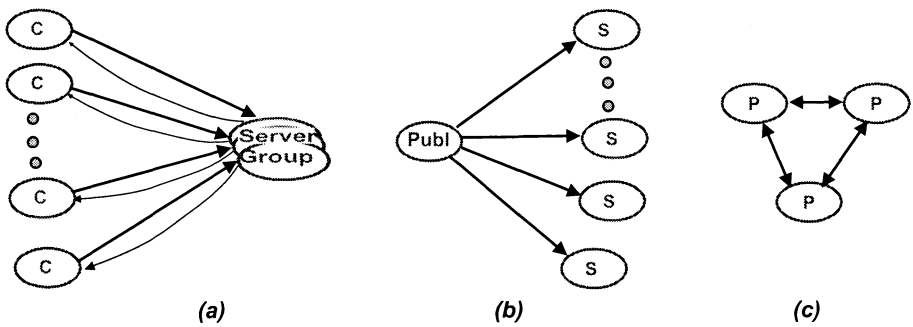

Figure 3.14. Basic Group Programming Models: (a) Client-Server; (b) Dissemination; (c) Multipeer

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).

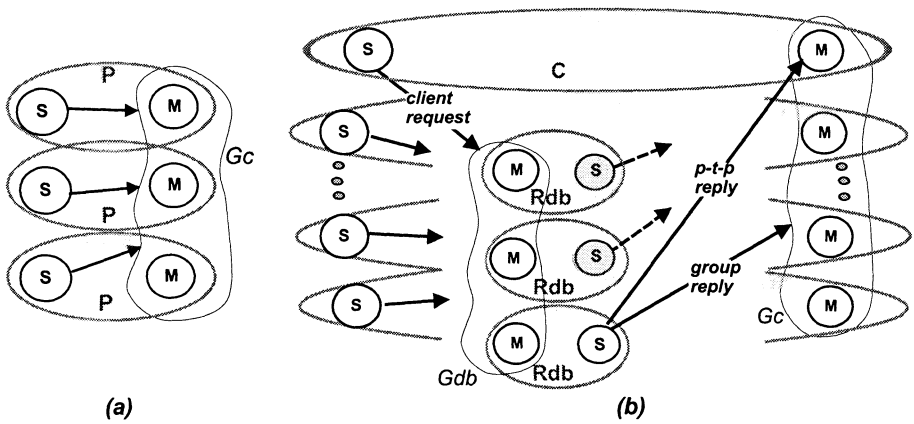

Figure 3.15. Implementing Group Models: (a) Multipeer; (b) Reliable Client-Server

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-*

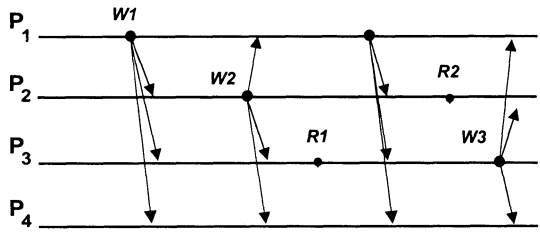

Figure 3.16. Strong Consistency

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.

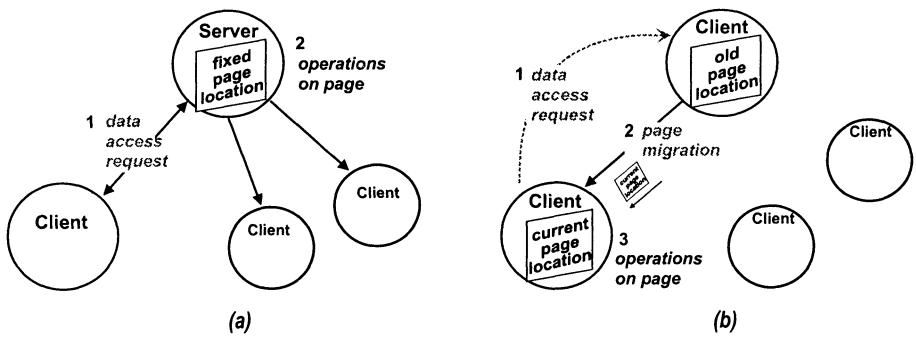

Figure 3.17. DSM architectures: (a) Centralized; (b) Migration

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.

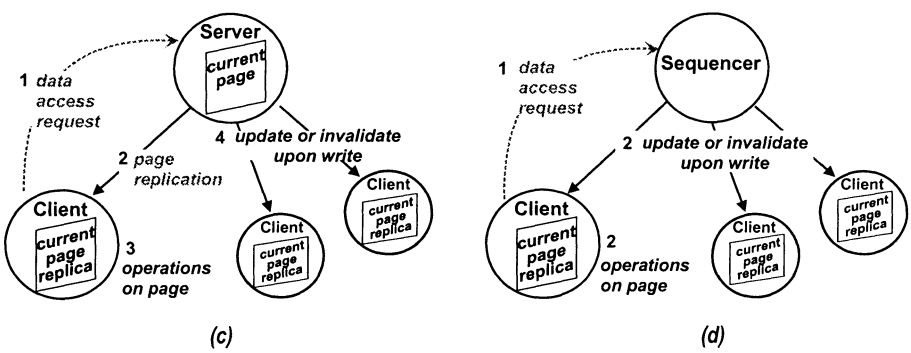

Figure 3.17 *(continued)*
DSM arch.: (c) Read-only Replication; (d) Read-Write Replication

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).

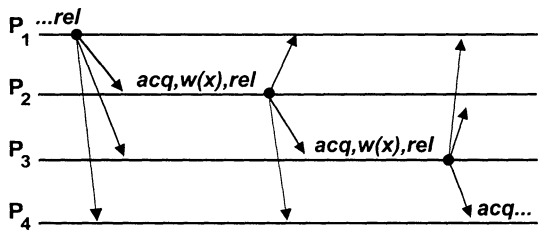

Figure 3.18. Release Consistency

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.

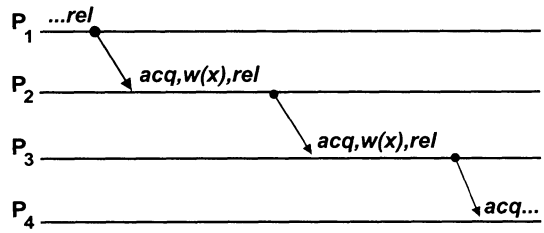

Figure 3.19. Lazy Release Consistency

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

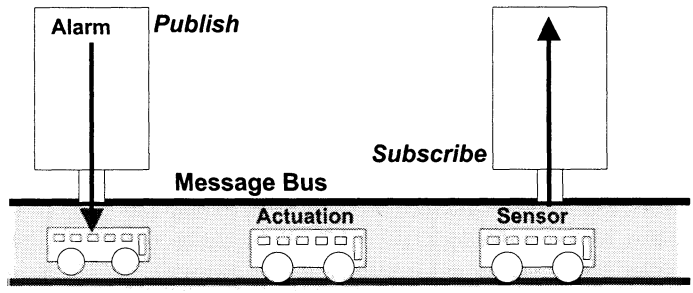

Figure 3.20. Message-bus Architecture

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).

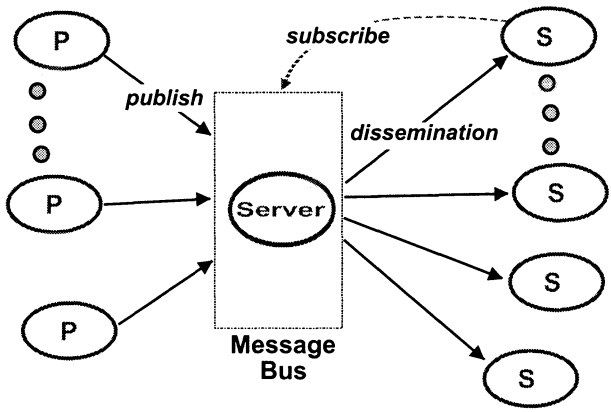

Figure 3.21. Implementing a Publish-Subscriber System

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.

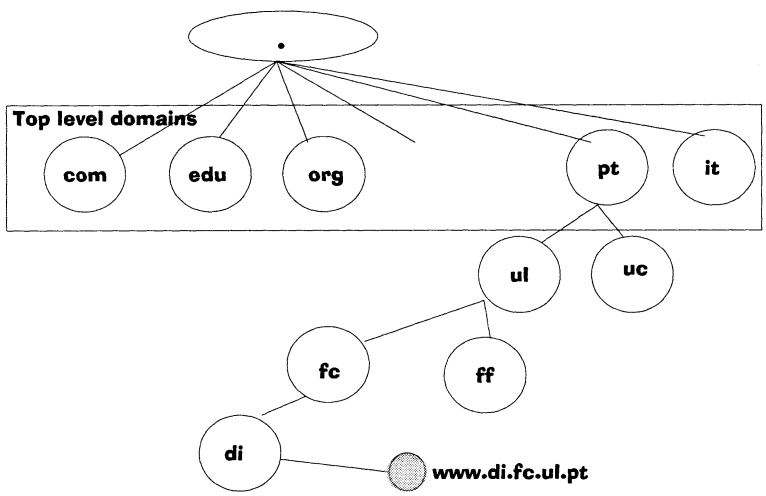

Figure 4.1. Hierarchical DNS Name Space

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.

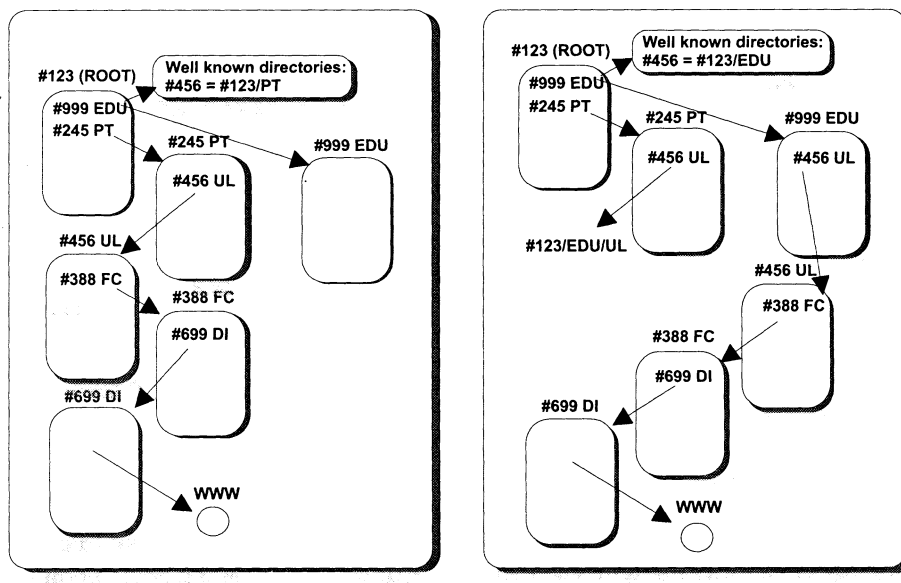

Figure 4.3. Hierarchical GNS Name Space

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.:
$\langle C{=}PT/O{=}ULisboa/U{=}FacSciences/L{=}DptInformatics/CN{=}Luís\rangle$

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: $\langle PT, ULisboa, FacSciences, DptInformatics, Luís\rangle$.

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

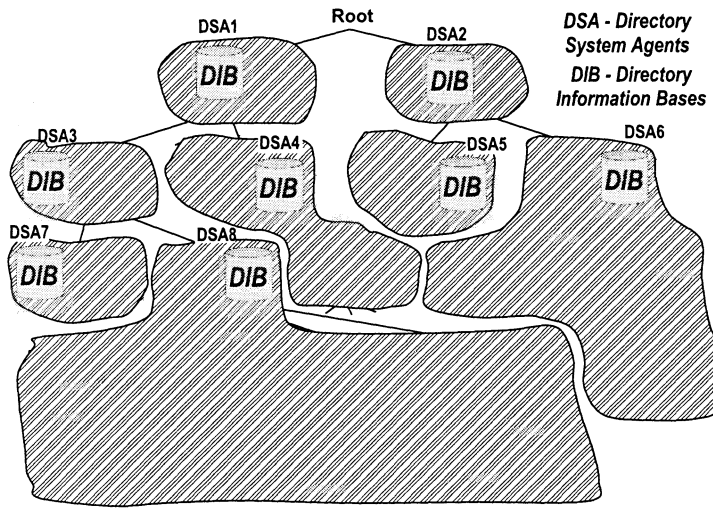

Figure 4.4. An X.500 Directory Structure

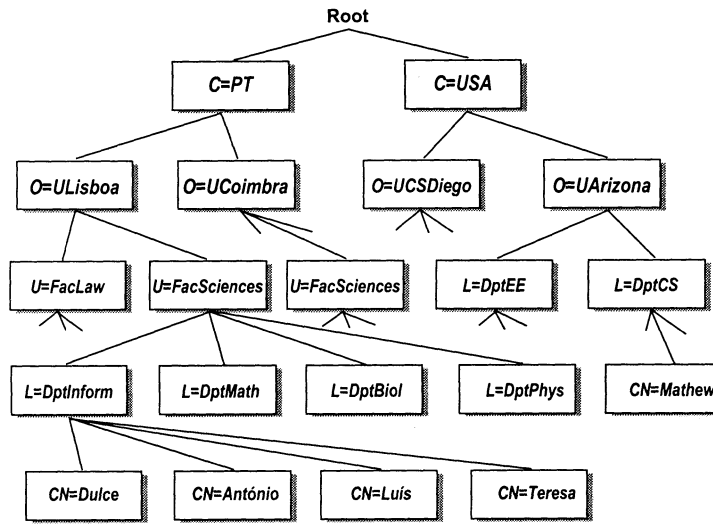

Figure 4.5. An X.500 Distinguished Name Tree

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.

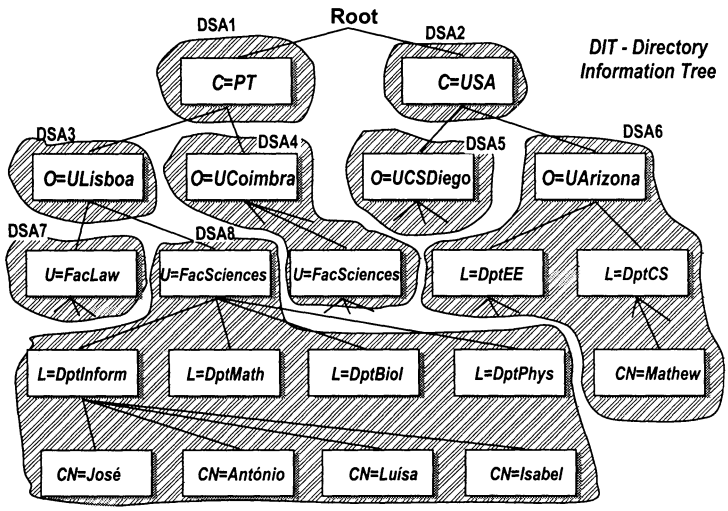

Figure 4.6. Name Tree under the X.500 Directory Service

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

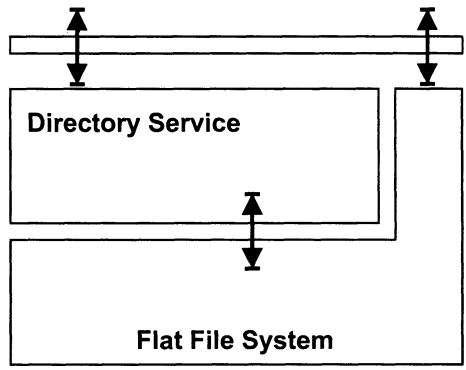

Figure 4.7. File System Architecture

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