DISTRIBUTED SYSTEMS

Principles and Paradigms

Second Edition

Andrew S. Tanenbaum
Maarten Van Steen

DISTRIBUTED SYSTEMS

Principles and Paradigms

Second Edition

About the Authors

Andrew S. Tanenbaum has an S.B. degree from M.I.T. and a Ph.D. from the University of California at Berkeley. He is currently a Professor of Computer Science at the Vrije Universiteit in Amsterdam, The Netherlands, where he heads the Computer Systems Group. Until stepping down in Jan. 2005, for 12 years he had been Dean of the Advanced School for Computing and Imaging, an interuniversity graduate school doing research on advanced parallel, distributed, and imaging systems.

In the past. he has done research on compilers, operating systems, networking, and local-area distributed systems. His current research focuses primarily on computer security, especially in operating systems, networks, and large wide-area distributed systems. Together, all these research projects have led to over 125 refereed papers in journals and conference proceedings and five books, which have been translated into 21 languages.

Prof. Tanenbaum has also produced a considerable volume of software. He was the principal architect of the Amsterdam Compiler Kit, a toolkit for writing portable compilers, as well as of MINIX, a small UNIX clone aimed at very high reliability. It is available for free at www.minix3.org.This system provided the inspiration and base on which Linux was developed. He was also one of the chief designers of *Amoeba* and *Globe.*

His Ph.D. students have gone on to greater glory after getting their degrees. He is very proud of them. In this respect he resembles a mother hen.

Prof. Tanenbaum is a Fellow of the ACM, a Fellow of the the IEEE, and a member of the Royal Netherlands Academy of Arts and Sciences. He is also winner of the 1994 ACM Karl V. Karlstrom Outstanding Educator Award, winner of the 1997 *ACM/SIGCSE* Award for Outstanding Contributions to Computer Science Education, and winner of the 2002 Texty award for excellence in textbooks. In 2004 he was named as one of the five new Academy Professors by the Royal Academy. His home page is at www.cs.vu.nl/r-ast.

Maarten van Steen is a professor at the Vrije Universiteit, Amsterdam, where he teaches operating systems, computer networks, and distributed systems. He has also given various highly successful courses on computer systems related subjects to ICT professionals from industry and governmental organizations.

Prof. van Steen studied Applied Mathematics at Twente University and received a Ph.D. from Leiden University in Computer Science. After his graduate studies he went to work for an industrial research laboratory where he eventually became head of the Computer Systems Group, concentrating on programming support for parallel applications.

After five years of struggling simultaneously do research and management, he decided to return to academia, first as an assistant professor in Computer Science at the Erasmus University Rotterdam, and later as an assistant professor in Andrew Tanenbaum's group at the Vrije Universiteit Amsterdam. Going back to university was the right decision; his wife thinks so, too.

His current research concentrates on large-scale distributed systems. Part of his research focuses on Web-based systems, in particular adaptive distribution and replication in Globule, a content delivery network of which his colleague Guillaume Pierre is the chief designer. Another subject of extensive research is fully decentralized (gossip-based) peer-to-peer systems of which results have been included in Tribler, a BitTorrent application developed in collaboration with colleagues from the Technical University of Delft.

DISTRIBUTED SYSTEMS

Principles and Paradigms

Second Edition

Andrew S. Tanenbaum
Maarten Van Steen

Vrije Universiteit
Amsterdam, The Netherlands

PEARSON
Prentice Hall

Upper Saddle River, NJ 07458

Library of Congress Cataloging-in-Publication Data

Tanenbaum, Andrew S.
Distributed systems: principles and paradigms / Andrew S. Tanenbaum, Maarten Van Steen.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-239227-5
1. Electronic data processing--Distributed processing. 2. Distributed operating systems (Computers) I. Steen, Maarten van. II. Title.
QA 76.9.D5T36 2006
005.4'476--dc22
2006024063

Vice President and Editorial Director, ECS: *Marcia J. Horton*
Executive Editor: *Tracy Dunkelberger*
Editorial Assistant: *Christianna Lee*
Associtate Editor: *Carole Stivder*
Executive Managing Editor: *Vince O'Brien*
Managing Editor: *Csmille Tremecoste*
Production Editor: *Craig Little*
Director of Creative Services: *Paul Belfanti*
Creative Director: *Juan Lopez*
Art Director: *Heather Scott*
Cover Designer: *Tamara Newnam*
Art Editor: *Xiaohong Zhu*
Manufacturing Manager, ESM: *Alexis Heydt-Long*
Manufacturing Buyer: *Lisa McDowell*
Executive Marketing Manager: *Robin O'Brien*
Marketing Assistant: *Mack Patterson*

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN: 0-13-239227-5

Pearson Education Ltd., *London*
Pearson Education Australia Pty. Ltd., *Sydney*
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd., *Hong Kong*
Pearson Education Canada, Inc., *Toronto*
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education-Japan, *Tokyo*
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., *Upper Saddle River, New Jersey*

To Suzanne, Barbara, Marvin, and the memory of Bram and Sweetie

-AST

To Marielle, Max, and Elke

-MvS

CONTENTS

PREFACE

Distributed systems form a rapidly changing field of computer science. Since the previous edition of this book, exciting new topics have emerged such as peer-to-peer computing and sensor networks, while others have become much more mature, like Web services and Web applications in general. Changes such as these required that we revised our original text to bring it up-to-date.

This second edition reflects a major revision in comparison to the previous one. We have added a separate chapter on architectures reflecting the progress that has been made on organizing distributed systems. Another major difference is that there is now much more material on decentralized systems, in particular peer-to-peer computing. Not only do we discuss the basic techniques, we also pay attention to their applications, such as file sharing, information dissemination, content-delivery networks, and publish/subscribe systems.

Next to these two major subjects, new subjects are discussed throughout the book. For example, we have added material on sensor networks, virtualization, server clusters, and Grid computing. Special attention is paid to self-management of distributed systems, an increasingly important topic as systems continue to scale.

Of course, we have also modernized the material where appropriate. For example, when discussing consistency and replication, we now focus on consistency models that are more appropriate for modem distributed systems rather than the original models, which were tailored to high-performance distributed computing. Likewise, we have added material on modem distributed algorithms, including GPS-based clock synchronization and localization algorithms.

Although unusual., we have nevertheless been able to *reduce* the total number of pages. This reduction is partly caused by discarding subjects such as distributed garbage collection and electronic payment protocols, and also reorganizing the last four chapters.

As in the previous edition, the book is divided into two parts. Principles of distributed systems are discussed in chapters 2-9, whereas overall approaches to how distributed applications should be developed (the paradigms) are discussed in chapters 10-13. Unlike the previous edition, however, we have decided not to discuss complete case studies in the paradigm chapters. Instead, each principle is now explained through a representative case. For example, object invocations are now discussed as a communication principle in Chap. 10 on object-based distributed systems. This approach allowed us to condense the material, but also to make it more enjoyable to read and study.

Of course. we continue to draw extensively from practice to explain what distributed systems are all about.. Various aspects of real-life systems such as WebSphere MQ, DNS, GPS, Apache, CORBA, Ice, NFS, Akamai, TIBlRendezvous. Jini, and many more are discussed throughout the book.. These examples illustrate the thin line between theory and practice, which makes distributed systems such an exciting field.

A number of people have contributed to this book in various ways. We would especially like to thank D. Robert Adams, Arno Bakker, Coskun Bayrak, Jacques Chassin de Kergommeaux, Randy Chow, Michel Chaudron, Puneet Singh Chawla, Fabio Costa, Cong Du, Dick Epema, Kevin Fenwick, Chandana Gamage. Ali Ghodsi, Giorgio Ingargiola, Mark Jelasity, Ahmed Kamel, Gregory Kapfhammer, Jeroen Ketema, Onno Kubbe, Patricia Lago, Steve MacDonald, Michael J. McCarthy, M. Tamer Ozsu, Guillaume Pierre, Avi Shahar, Swaminathan Sivasubramanian, Chintan Shah, Ruud Stegers, Paul Tymann, Craig E. Wills, Reuven Yagel, and Dakai Zhu for reading parts of the manuscript, helping identifying mistakes in the previous edition, and offering useful comments.

Finally, we would like to thank our families. Suzanne has been through this process seventeen times now. That's a lot of times for me but also for her. Not once has she said: "Enough is enough" although surely the thought has occurred to her. Thank you. Barbara and Marvin now have a much better idea of what professors do for a living and know the difference between a good textbook and a bad one. They are now an inspiration to me to try to produce more good ones than bad ones (AST).

Because I took a sabbatical leave to update the book, the whole business of writing was also much more enjoyable for Mariëlle, She is beginning to get used to it, but continues to remain supportive while alerting me when it is indeed time to redirect attention to more important issues. lowe her many thanks. Max and Elke by now have a much better idea of what writing a book means, but compared to what they are reading themselves, find it difficult to understand what is so exciting about these strange things called distributed systems. I can't blame them (MvS).

1

INTRODUCTION

Computer systems are undergoing a revolution. From 1945, when the modem computer era began, until about 1985, computers were large and expensive. Even minicomputers cost at least tens of thousands of dollars each. As a result, most organizations had only a handful of computers, and for lack of a way to connect them, these operated independently from one another.

Starting around the the mid-1980s, however, two advances in technology began to change that situation. The first was the development of powerful microprocessors. Initially, these were 8-bit machines, but soon 16-, 32-, and 64-bit CPUs became common. Many of these had the computing power of a mainframe (i.e., large) computer, but for a fraction of the price.

The amount of improvement that has occurred in computer technology in the past half century is truly staggering and totally unprecedented in other industries. From a machine that cost 10 million dollars and executed 1 instruction per second. we have come to machines that cost 1000 dollars and are able to execute 1 billion instructions per second, a price/performance gain of 1013.If cars had improved at this rate in the same time period, a Rolls Royce would now cost 1 dollar and get a billion miles per gallon. (Unfortunately, it would probably also have a 200-page manual telling how to open the door.)

The second development was the invention of high-speed computer networks. **Local-area networks, or LANs** allow hundreds of machines within a building to be connected in such a way that small amounts of information can be transferred between machines in a few microseconds or so. Larger amounts of data can be moved between machines at rates of 100 million to 10 billion bits/sec. Wide-area networks or WANs allow miJIions of machines all over the earth to be connected at speeds varying from 64 Kbps (kilobits per second) to gigabits per second.

The result of these technologies is that it is now not only feasible, but easy, to put together computing systems composed of large numbers of computers connected by a high-speed network. They are usually caned computer networks or distributed systems, in contrast to the previous centralized systems (or single-processor systems) consisting of a single computer, its peripherals, and perhaps some remote terminals.

1.1 DEFINITION OF A DISTRIBUTED SYSTEM

Various definitions of distributed systems have been given in the literature, none of them satisfactory, and none of them in agreement with any of the others. For our purposes it is sufficient to give a loose characterization:

> *A distributed system is a collection of independent computers that appears to its users as a single coherent system.*

This definition has several important aspects. The first one is that a distributed system consists of components (i.e., computers) that are autonomous. A second aspect is that users (be they people or programs) think they are dealing with a single system. This means that one way or the other the autonomous components need to collaborate. How to establish this collaboration lies at the heart of developing distributed systems. Note that no assumptions are made concerning the type of computers. In principle, even within a single system, they could range from high-performance mainframe computers to small nodes in sensor networks. Likewise, no assumptions are made on the way that computers are interconnected. We will return to these aspects later in this chapter.

Instead of going further with definitions, it is perhaps more useful to concentrate on important characteristics of distributed systems. One important characteristic is that differences between the various computers and the ways in which they communicate are mostly hidden from users. The same holds for the internal organization of the distributed system. Another important characteristic is that users and applications can interact with a distributed system in a consistent and uniform way, regardless of where and when interaction takes place.

In principle, distributed systems should also be relatively easy to expand or scale. This characteristic is a direct consequence of having independent computers, but at the same time, hiding how these computers actually take part in the system as a whole. A distributed system will normally be continuously available, although perhaps some parts may be temporarily out of order. Users and applications should not notice that parts are being replaced or fixed, or that new parts are added to serve more users or applications..

In order to support heterogeneous computers and networks while offering a single-system view, distributed systems are often organized by means of a layer of software-that is, logically placed between a higher-level layer consisting of users and applications, and a layer underneath consisting of operating systems and basic communication facilities, as shown in Fig. 1-1 Accordingly, such a distributed system is sometimes called middleware.

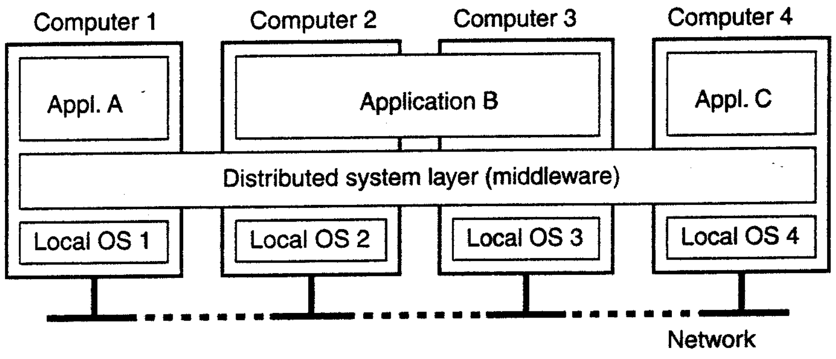


Figure I-I. A distributed system organized as middleware. The middleware layer extends over multiple machines, and offers each application the same interface.

Fig. 1-1 shows four networked computers and three applications, of which application *B* is distributed across computers 2 and 3. Each application is offered the same interface. The distributed system provides the means for components of a single distributed application to communicate with each other, but also to let different applications communicate. At the same time, it hides, as best and reasonable as possible, the differences in hardware and operating systems from each application.

1.2 GOALS

Just because it is possible to build distributed systems does not necessarily mean that it is a good idea. After all, with current technology it is also possible to put four floppy disk drives on a personal computer. It is just that doing so would be pointless. In this section we discuss four important goals that should be met to make building a distributed system worth the effort. A distributed system should make resources easily accessible; it should reasonably hide the fact that resources are distributed across a network; it should be open; and it should be scalable.

1.2.1 Making Resources Accessible

The main goal of a distributed system is to make it easy for the users (and applications) to access remote resources, and to share them in a controlled and efficient way. Resources can be just about anything, but typical examples include things like printers, computers, storage facilities, data, files, Web pages, and networks, to name just a few. There are many reasons for wanting to share resources. One obvious reason is that of economics. For example, it is cheaper to let a printer be shared by several users in a smaJl office than having to buy and maintain a separate printer for each user. Likewise, it makes economic sense to share costly resources such as supercomputers, high-performance storage systems, imagesetters, and other expensive peripherals.

Connecting users and resources also makes it easier to collaborate and exchange information, as is clearly illustrated by the success of the Internet with its simple protocols for exchanging files, mail. documents, audio, and video. The connectivity of the Internet is now leading to numerous virtual organizations in which geographicaJJy widely-dispersed groups of people work together by means of groupware, that is, software for coJJaborative editing, teleconferencing, and so on. Likewise, the Internet connectivity has enabled electronic commerce allowing us to buy and sell all kinds of goods without actually having to go to a store or even leave home.

However, as connectivity and sharing increase, security is becoming increasingly important. In current practice, systems provide little protection against eavesdropping or intrusion on communication. Passwords and other sensitive information are often sent as cJeartext (i.e., unencrypted) through the network, or stored at servers that we can only hope are trustworthy. In this sense, there is much room for improvement. For example, it is currently possible to order goods by merely supplying a credit card number. Rarely is proof required that the customer owns the card. In the future, placing orders this way may be possible only if you can actually prove that you physicaJJy possess the card by inserting it into a card reader.

Another security problem is that of tracking communication to build up a preference profile of a specific user (Wang et aL, 1998). Such tracking explicitly violates privacy, especially if it is done without notifying the user. A related problem is that increased connectivity can also lead to unwanted communication, such as electronic junk mail, often called spam. In such cases, what we may need is to protect ourselves using special information filters that select incoming messages based on their content.

1.2.2 **Distribution** Transparency

An important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers. A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be transparent. Let us first take a look at what kinds of transparency exist in distributed systems. After that we will address the more general question whether transparency is always required.

Types of Transparency

The concept of transparency can be applied to several aspects of a distributed system, the most important ones shown in Fig. 1-2.

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource

Figure 1-2. Different forms of transparency in a distributed system (ISO, 1995).

Access transparency deals with hiding differences in data representation and the way that resources can be accessed by users. At a basic level, we wish to hide differences in machine architectures, but more important is that we reach agreement on how data is to be represented by different machines and operating systems. For example, a distributed system may have computer systems that run different operating systems, each having their own file-naming conventions. Differences in naming conventions, as well as how files can be manipulated, should all be hidden from users and applications.

An important group of transparency types has to do with the location of a resource. Location transparency refers to the fact that users cannot tell where a resource is physically located in the system. Naming plays an important role in achieving location transparency. In particular, location transparency can be achieved by assigning only logical names to resources, that is, names in which the location of a resource is not secretly encoded. An example of a such a name is the URL http://www.prenhall.com/index.html which gives no clue about the location of Prentice Hall's main Web server. The URL also gives no clue as to whether *index.html* has always been at its current location or was recently moved there. Distributed systems in which resources can be moved without affecting how those resources can be accessed are said to provide migration transparency. Even stronger is the situation in which resources can be relocated *while* they are being accessed without the user or application noticing anything. In such cases, the system is said to support relocation transparency. An example of relocation transparency is when mobile users can continue to use their wireless laptops while moving from place to place without ever being (temporarily) disconnected.

As we shall see, replication plays a very important role in distributed systems. For example, resources may be replicated to increase availability or to improve performance by placing a copy close to the place where it is accessed. Replication transparency deals with hiding the fact that several copies of a resource exist. To hide replication from users, it is necessary that all replicas have the same name. Consequently, a system that supports replication transparency should generally support location transparency as well, because it would otherwise be impossible to refer to replicas at different locations.

We already mentioned that an important goal of distributed systems is to allow sharing of resources. In many cases, sharing resources is done in a cooperative way, as in the case of communication. However, there are also many examples of competitive sharing of resources. For example, two independent users may each have stored their files on the same file server or may be accessing the same tables in a shared database. In such cases, it is important that each user does not notice that the other is making use of the same resource. This phenomenon is called concurrency transparency. An important issue is that concurrent access to a shared resource leaves that resource in a consistent state. Consistency can be achieved through locking mechanisms, by which users are, in turn, given exclusive access to the desired resource. A more refined mechanism is to make use of transactions, but as we shall see in later chapters, transactions are quite difficult to implement in distributed systems.

A popular alternative definition of a distributed system, due to Leslie Lamport, is "You know you have one when the crash of a computer you've never heard of stops you from getting any work done." This description puts the finger on another important issue of distributed systems design: dealing with failures. Making a distributed system failure transparent means that a user does not notice that a resource (he has possibly never heard of) fails to work properly, and that the system subsequently recovers from that failure. Masking failures is one of the hardest issues in distributed systems and is even impossible when certain apparently realistic assumptions are made, as we will discuss in Chap. 8. The main difficulty in masking failures lies in the inability to distinguish between a dead resource and a painfully slow resource. For example, when contacting a busy Web server, a browser will eventually time out and report that the Web page is unavailable..At that point, the user cannot conclude that the server is really down.

Degree of Transparency

Although distribution transparency is generally considered preferable for any distributed system, there are situations in which attempting to completely hide all distribution aspects from users is not a good idea. An example is requesting your electronic newspaper to appear in your mailbox before 7 A.M. local time, as usual, while you are currently at the other end of the world living in a different time zone. Your morning paper will not be the morning paper you are used to.

Likewise, a wide-area distributed system that connects a process in San Francisco to a process in Amsterdam cannot be expected to hide the fact that Mother

Nature will not allow it to send a message from one process to the other in less than about 35 milliseconds. In practice it takes several hundreds of milliseconds using a computer network. Signal transmission is not only limited by the speed of light. but also by limited processing capacities of the intermediate switches.

There is also a trade-off between a high degree of transparency and the performance of a system. For example, many Internet applications repeatedly try to contact a server before finally giving up. Consequently, attempting to mask a transient server failure before trying another one may slow down the system as a whole. In such a case, it may have been better to give up earlier, or at least let the user cancel the attempts to make contact

Another example is where we need to guarantee that several replicas, located on different continents, need to be consistent all the time. In other words, if one copy is changed, that change should be propagated to all copies before allowing any other operation. It is clear that a single update operation may now even take seconds to complete, something that cannot be hidden from users.

Finally, there are situations in which it is not at all obvious that hiding distribution is a good idea. As distributed systems are expanding to devices that people carry around, and where the very notion of location and context awareness is becoming increasingly important, it may be best to actually *expose* distribution rather than trying to hide it. This distribution exposure will become more evident when we discuss embedded and ubiquitous distributed systems later in this chapter. As a simple example, consider an office worker who wants to print a file from her notebook computer. It is better to send the print job to a busy nearby printer, rather than to an idle one at corporate headquarters in a different country.

There are also other arguments against distribution transparency. Recognizing that full distribution transparency is simply impossible, we should ask ourselves whether it is even wise to *pretend* that we can achieve it. It may be much better to make distribution explicit so that the user and application developer are never tricked into believing that there is such a thing as transparency. The result will be that users will much better understand the (sometimes unexpected) behavior of a distributed system, and are thus much better prepared to deal with this behavior.

The conclusion is that aiming for distribution transparency may be a nice goal when designing and implementing distributed systems, but that it should be considered together with other issues such as performance and comprehensibility. The price for not being able to achieve full transparency may be surprisingly high.

1.2.3 Openness

Another important goal of distributed systems is openness. An **open distributed** system is a system that offers services according to standard rules that describe the syntax and semantics of those services. For example, in computer networks, standard rules govern the format, contents, and meaning of messages sent and received. Such rules are formalized in protocols. In distributed systems, services are generally specified through interfaces, which are often described in an Interface Definition Language (IDL). Interface definitions written in an IDL nearly always capture only the syntax of services. In other words, they specify precisely the names of the functions that are available together with types of the parameters, return values, possible exceptions that can be raised, and so on. The hard part is specifying precisely what those services do, that is, the semantics of interfaces. In practice, such specifications are always given in an informal way by means of natural language.

If properly specified, an interface definition allows an arbitrary process that needs a certain interface to talk to another process that provides that interface. It also allows two independent parties to build completely different implementations of those interfaces, leading to two separate distributed systems that operate in exactly the same way. Proper specifications are complete and neutral. Complete means that everything that is necessary to make an implementation has indeed been specified. However, many interface definitions are not at all complete, so that it is necessary for a developer to add implementation-specific details. Just as important is the fact that specifications do not prescribe what an implementation should look like: they should be neutral. Completeness and neutrality are important for interoperability and portability (Blair and Stefani, 1998). Interoperability characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard. Portability characterizes to what extent an application developed for a distributed system *A* can be executed, without modification, on a different distributed system *B* that implements the same interfaces as *A*.

Another important goal for an open distributed system is that it should be easy to configure the system out of different components (possibly from different developers). Also, it should be easy to add new components or replace existing ones without affecting those components that stay in place. In other words, an open distributed system should also be extensible. For example, in an extensible system, it should be relatively easy to add parts that run on a different operating system, or even to replace an entire file system. As many of us know from daily practice, attaining such flexibility is easier said than done.

Separating Policy from Mechanism

To achieve flexibility in open distributed systems, it is crucial that the system is organized as a collection of relatively small and easily replaceable or adaptable components. This implies that we should provide definitions not only for the highest-level interfaces, that is, those seen by users and applications, but also definitions for interfaces to internal parts of the system and describe how those parts interact. This approach is relatively new. Many older and even contemporary systems are constructed using a monolithic approach in which components are only logically separated but implemented as one huge program. This approach makes it hard to replace or adapt a component without affecting the entire system. Monolithic systems thus tend to be closed instead of open.

The need for changing a distributed system is often caused by a component that does not provide the optimal policy for a specific user or application. As an example, consider caching in the World Wide Web. Browsers generally allow users to adapt their caching policy by specifying the size of the cache, and whether a cached document should always be checked for consistency, or perhaps only once per session. However, the user cannot influence other caching parameters, such as how long a document may remain in the cache, or which document should be removed when the cache fills up. Also, it is impossible to make caching decisions based on the *content* of a document. For instance, a user may want to cache railroad timetables, knowing that these hardly change, but never information on current traffic conditions on the highways.

What we need is a separation between policy and mechanism. In the case of Web caching, for example, a browser should ideally provide facilities for only storing documents, and at the same time allow users to decide which documents are stored and for how long. In practice, this can be implemented by offering a rich set of parameters that the user can set (dynamically). Even better is that a user can implement his own policy in the form of a component that can be plugged into the browser. Of course, that component must have an interface that the browser can understand so that it can call procedures of that interface.

1.2.4 Scalability

Worldwide connectivity through the Internet is rapidly becoming as common as being able to send a postcard to anyone anywhere around the world. With this in mind, scalability is one of the most important design goals for developers of distributed systems.

Scalability of a system can be measured along at least three different dimensions (Neuman, 1994). First, a system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system. Second, a geographically scalable system is one in which the users and resources may lie far apart. Third, a system can be administratively scalable,/~~aning that it can still be easy to manage even if it spans many independent administrative organizations. Unfortunately, a system that is scalable in one or more of these dimensions often exhibits some loss of performance as the system scales up.

Scalability Problems

When a system needs to scale, very different types of problems need to be solved. Let us first consider scaling with respect to size. If more users or resources need to be supported, we are often confronted with the limitations of centralized services, data, and algorithms (see Fig. 1-3). For example, many services are centralized in the sense that they are implemented by means of only a single server running on a specific machine in the distributed system. The problem with this scheme is obvious: the server can become a bottleneck as the number of users and applications grows. Even if we have virtually unlimited processing and storage capacity, communication with that server will eventually prohibit further growth.

Unfortunately. using only a single server is sometimes unavoidable. Imagine that we have a service for managing highly confidential information such as medical records, bank accounts. and so on. In such cases, it may be best to implement that service by means of a single server in a highly secured separate room, and protected from other parts of the distributed system through special network components. Copying the server to several locations to enhance performance maybe out of the question as it would make the service less secure.

Concept	Example
Centralized services	A single server for all users
Centralized data	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

Figure 1-3. Examples of scalability limitations.

Just as bad as centralized services are centralized data. How should we keep track of the telephone numbers and addresses of 50 million people? Suppose that each data record could be fit into 50 characters. A single 2.5-gigabyte disk partition would provide enough storage. But here again, having a single database would undoubtedly saturate all the communication lines into and out of it. Likewise, imagine how the Internet would work if its Domain Name System (DNS) was still implemented as a single table. DNS maintains information on millions of computers worldwide and forms an essential service for locating Web servers. If each request to resolve a URL had to be forwarded to that one and only DNS server, it is dear that no one would be using the Web (which, by the way, would solve the problem).

Finally, centralized algorithms are also a bad idea. In a large distributed system, an enormous number of messages have tobe routed over many lines. From a theoretical point of view, the optimal way to do this is collect complete information about the load on all machines and lines, and then run an algorithm to compute all the optimal routes. This information can then be spread around the system to improve the routing.

. The trouble is that collecting and transporting all the input and output information would again be a bad idea because these messages would overload part of the network. In fact, any algorithm that operates by collecting information from all the sites, sends it to a single machine for processing, and then distributes the results should generally be avoided. Only decentralized algorithms should be used. These algorithms generally have the following characteristics, which distinguish them from centralized algorithms:

1. No machine has complete information about the system state.
2. Machines make decisions based only on local information,
3. Failure of one machine does not ruin the algorithm.
4. There is no implicit assumption that a global clock exists.

The first three follow from what we have said so far. The last is perhaps less obvious but also important. Any algorithm that starts out with: "At precisely 12:00:00 all machines shall note the size of their output queue" will fail because it is impossible to get all the clocks exactly synchronized. Algorithms should take into account the lack of exact clock synchronization. The larger the system, the larger the uncertainty. On a single LAN, with considerable effort it may be possible to get all clocks synchronized down to a few microseconds, but doing this nationally or internationally is tricky.

Geographical scalability has its own problems. One of the main reasons why it is currently hard to scale existing distributed systems that were designed for local-area networks is that they are based on synchronous communication. In this form of communication, a party requesting service, generally referred to as a client, blocks until a reply is sent back. This approach generally works fine in LANs where communication between two machines is generally at worst a few hundred microseconds. However, in a wide-area system, we need to take into account that interprocess communication may be hundreds of milliseconds, three orders of magnitude slower. Building interactive applications using synchronous communication in wide-area systems requires a great deal of care (and not a little patience).

Another problem that hinders geographical scalability is that communication in wide-area networks is inherently unreliable, and virtually always point-to-point. In contrast, local-area networks generally provide highly reliable communication facilities based on broadcasting, making it much easier to develop distributed systems. For example, consider the problem of locating a service. In a local-area system, a process can simply broadcast a message to eve\)' machine, asking if it is running the service it needs. Only those machines that Have that service respond, each providing its network address in the reply message. Such a location scheme is unthinkable in a wide-area system: just imagine what would happen if we tried to locate a service this way in the Internet. Instead, special location services need to be designed, which may need to scale worldwide and be capable of servicing a billion users. We return to such services in Chap. 5.

Geographical scalability is strongly related to the problems of centralized solutions that hinder size scalability. If we have a system with many centralized components, it is clear that geographical scalability will be limited due to the performance and reliability problems resulting from wide-area communication. In addition, centralized components now lead to a waste of network resources. Imagine that a single mail server is used for an entire country. This would mean that sending an e-mail to your neighbor would first have to go to the central mail server, which may be hundreds of miles away. Clearly, this is not the way to go.

Finally, a difficult, and in many cases open question is how to scale a distributed system across multiple, independent administrative domains. A major problem that needs to be solved is that of conflicting policies with respect to resource usage (and payment), management, and security.

For example, many components of a distributed system that reside within a single domain can often be trusted by users that operate within that same domain. In such cases, system administration may have tested and certified applications, and may have taken special measures to ensure that such components cannot be tampered with. In essence, the users trust their system administrators. However, this trust does not expand naturally across domain boundaries.

If a distributed system expands into another domain, two types of security measures need to be taken. First of all, the distributed system has to protect itself against malicious attacks from the new domain. For example, users from the new domain may have only read access to the file system in its original domain. Likewise, facilities such as expensive image setters or high-performance computers may not be made available to foreign users. Second, the new domain has to protect itself against malicious attacks from the distributed system. A typical example is that of downloading programs such as applets in Web browsers. Basically, the new domain does not know behavior what to expect from such foreign code, and may therefore decide to severely limit the access rights for such code. The problem, as we shall see in Chap. 9, is how to enforce those limitations.

Scaling Techniques

Having discussed some of the scalability problems brings us to the question of how those problems can generally be solved. In most cases, scalability problems in distributed systems appear as performance problems caused by limited capacity of servers and network. There are now basically only three techniques for scaling: hiding communication latencies, distribution, and replication [see also Neuman (1994)].

Hiding communication latencies is important to achieve geographical scalability. The basic idea is simple: try to avoid waiting for responses to remote (and potentially distant) service requests as much as possible. For example, when a service has been requested at a remote machine, an alternative to waiting for a reply from the server is to do other useful work at the requester's side. Essentially, what this means is constructing the requesting application in such a way that it uses only asynchronous communication. When a reply comes in, the application is interrupted and a special handler is called to complete the previously-issued request. Asynchronous communication can often be used in batch-processing systems and parallel applications, in which more or less independent tasks can be scheduled for execution while another task is waiting for communication to complete. Alternatively, a new thread of control can be started to performthe request. Although it blocks waiting for the reply, other threads in the process can continue.

However, there are many applications that cannot make effective use of asynchronous communication. For example, in interactive applications when a user sends a request he will generally have nothing better to do than to wait for the answer. In such cases, a much better solution is to reduce the overall communication, for example, by moving part of the computation that is normally done at the server to the client process requesting the service. A typical case where this approach works is accessing databases using forms. Filling in forms can be done by sending a separate message for each field, and waiting for an acknowledgment from the server, as shown in Fig. 1-4(a). For example, the server may check for syntactic errors before accepting an entry. A much better solution is to ship the code for filling in the form, and possibly checking the entries, to the client, and have the client return a completed form, as shown in Fig. 1-4(b). This approach of shipping code is now widely supported by the Web in the form of Java applets and Javascript.

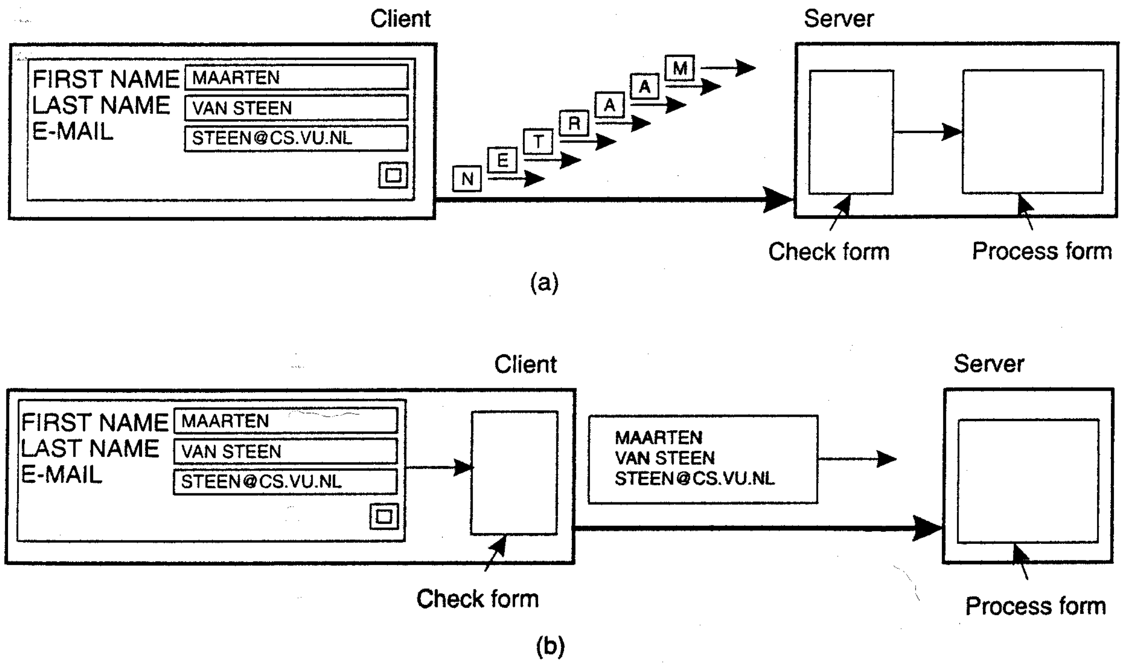


Figure 1-4. The difference between letting (a) a server or (b) a client check forms as they are being filled.

Another important scaling technique is distribution. Distribution involves taking a component, splitting it into smaller parts, and subsequently spreading those parts across the system. An excellent example of distribution is the Internet Domain Name System (DNS). The DNS name space is hierarchically organized into a tree of **domains,** which are divided into nonoverlapping **zones,** as shown in Fig. 1-5. The names in each zone are handled by a single name server. Without going into too many details, one can think of each path name,being the name of a host in the Internet, and thus associated with a network address of that host. Basically, resolving a name means returning the network address of the associated host. Consider, for example, the name *nl.vu.cs.flits.* To resolve this name, it is first passed to the server of zone 21 (see Fig. 1-5) which returns the address of the server for zone 22, to which the rest of name, *vu.cs.flits,* can be handed. The server for 22 will return the address of the server for zone 23, which is capable of handling the last part of the name and will return the address of the associated host.

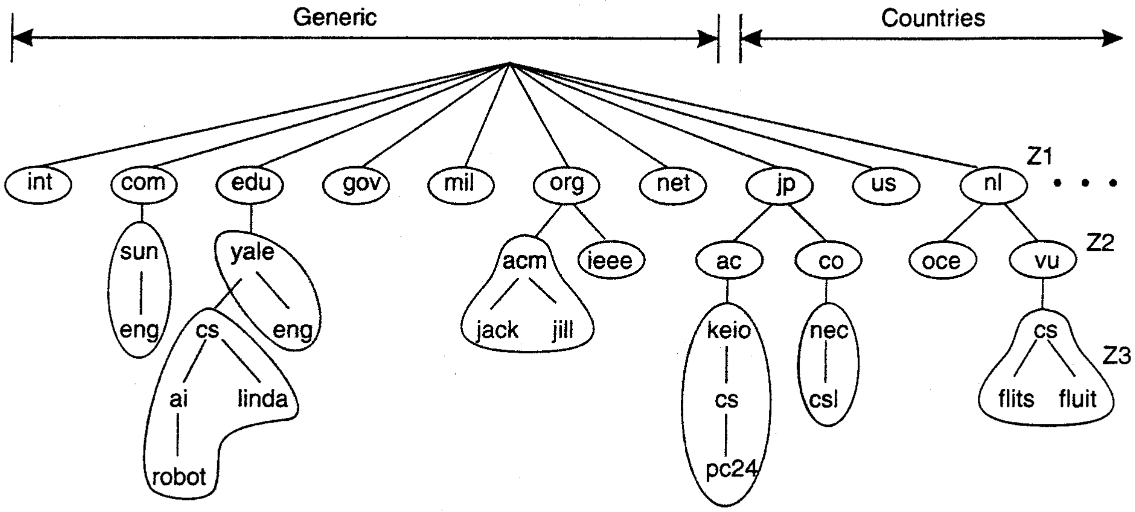


Figure 1-5. An example of dividing the DNS name space into zones.

This example illustrates how the *naming service,* as provided by DNS, is distributed across several machines, thus avoiding that a single server has to deal with all requests for name resolution.

As another example, consider the World Wide Web. To most users, the Web appears to be an enormous document-based information system in which each document has its own unique name in the form of a URL. Conceptually, it may even appear as if there is only a single server. However, the Web is physically distributed across a large number of servers, each handling a number of Web documents. The name of the server handling a document is encoded into that document's URL. It is only because of this distribution of documents that the Web has been capable of scaling to its current size.

Considering that scalability problems often appear in the form of performance degradation, it is generally a good idea to actually **replicate** components across a distributed system. Replication not only increases availability, but also helps to balance the load between components leading to better performance. Also, in geographically widely-dispersed systems, having a copy nearby can hide much of the communication latency problems mentioned before.

Caching is a special form of replication, although the distinction between the two is often hard to make or even artificial. As in the case of replication, caching results in making a copy of a resource, generally in the proximity of the client accessing that resource. However, in contrast to replication, caching is a decision made by the client of a resource, and not by the owner of a resource. Also, caching happens on demand whereas replication is often planned in advance.

There is one serious drawback to caching and replication that may adversely affect scalability. Because we now have multiple copies of a resource, modifying one copy makes that copy different from the others. Consequently, caching and replication leads to consistency problems.

To what extent inconsistencies can be tolerated depends highly on the usage of a resource. For example, many Web users fmd it acceptable that their browser returns a cached document of which the validity has not been checked for the last few minutes. However, there are also many cases in which strong consistency guarantees need to be met, such as in the case of electronic stock exchanges and auctions. The problem with strong consistency is that an update must be immediately propagated to all other copies. Moreover, if two updates happen concurrently, it is often also required that each copy is updated in the same order. Situations such as these generally require some global synchronization mechanism. Unfortunately, such mechanisms are extremely hard or even impossible to implement in a scalable way, as she insists that photons and electrical signals obey a speed limit of 187 *miles/msec* (the speed of light). Consequently, scaling by replication may introduce other, inherently nonscalable solutions. We return to replication and consistency in Chap. 7.

When considering these scaling techniques, one could argue that size scalability is the least problematic from a technical point of view. In many cases, simply increasing the capacity of a machine will the save the day (at least temporarily and perhaps at significant costs). Geographical scalability is a much tougher problem as Mother Nature is getting in our way. Nevertheless, practice shows that combining distribution, replication, and caching techniques with different forms of consistency will often prove sufficient in many cases. Finally, administrative scalability seems to be the most difficult one, partly also because we need to solve nontechnical problems (e.g., politics of organizations and human collaboration). Nevertheless, progress has been made in this area, by simply *ignoring* administrative domains. The introduction and now widespread use of peer-to-peer technology demonstrates what can be achieved if end users simply take over control (Aberer and Hauswirth, 2005; Lua et al., 2005; and Oram, 2001). However, let it be clear that peer-to-peer technology can at best be only a partial solution to solving administrative scalability. Eventually, it will have to be dealt with.

1.2.5 Pitfalls

It should be clear by now that developing distributed systems can be a formidable task. As we will see many times throughout this book, there are so many issues to consider at the same time that it seems that only complexity can be the result. Nevertheless, by following a number of design principles, distributed systems can be developed that strongly adhere to the goals we set out in this chapter. Many principles follow the basic rules of decent software engineering and wiJI not be repeated here.

However, distributed systems differ from traditional software because components are dispersed across a network. Not taking this dispersion into account during design time is what makes so many systems needlessly complex and results in mistakes that need to be patched later on. Peter Deutsch, then at Sun Microsystems, formulated these mistakes as the following false assumptions that everyone makes when developing a distributed application for the first time:

1. The network is reliable.
2. The network is secure.
3. The network is homogeneous.
4. The topology does not change.
5. Latency is zero.
6. Bandwidth is infinite.
7. Transport cost is zero.
8. There is one administrator.

Note how these assumptions relate to properties that are unique to distributed systems: reliability, security, heterogeneity, and topology of the network; latency and bandwidth; transport costs; and finally administrative domains. When developing nondistributed applications, many of these issues will most likely not show up.

Most of the principles we discuss in this book relate immediately to these assumptions. In all cases, we will be discussing solutions to problems, that are caused by the fact that one or more assumptions are false. For example, reliable networks simply do not exist, leading to the impossibility of achieving failure transparency. We devote an entire chapter to deal with the fact that networked communication is inherently insecure. We have already argued that distributed systems need to take heterogeneity into account. In a similar vein, when discussing replication for solving scalability problems, we are essentially tackling latency and bandwidth problems. We will also touch upon management issues at various points throughout this book, dealing with the false assumptions of zero-cost transportation and a single administrative domain.

1.3 TYPES OF DISTRIBUTED SYSTEMS

Before starting to discuss the principles of distributed systems, let us first take a closer look at the various types of distributed systems. In the following we make a distinction between distributed computing systems, distributed information systems, and distributed embedded systems.

1.3.1 Distributed Computing Systems

An important class of distributed systems is the one used for high-performance computing tasks. Roughly speaking, one can make a distinction between two subgroups. In cluster computing the underlying hardware consists of a collection of similar workstations or PCs, closely connected by means of a high-speed local-area network. In addition, each node runs the same operating system.

The situation becomes quite different in the case of grid computing. This subgroup consists of distributed systems that are often constructed as a federation of computer systems, where each system may fall under a different administrative domain, and may be very different when it comes to hardware, software, and deployed network technology.

Cluster Computing Systems

Cluster computing systems became popular when the price/performance ratio of personal computers and workstations improved. At a certain point, it became financially and technically attractive to build a supercomputer using off-the-shelf technology by simply hooking up a collection of relatively simple computers in a high-speed network. In virtually all cases, cluster computing is used for parallel programming in which a single (compute intensive) program is run in parallel on multiple machines.

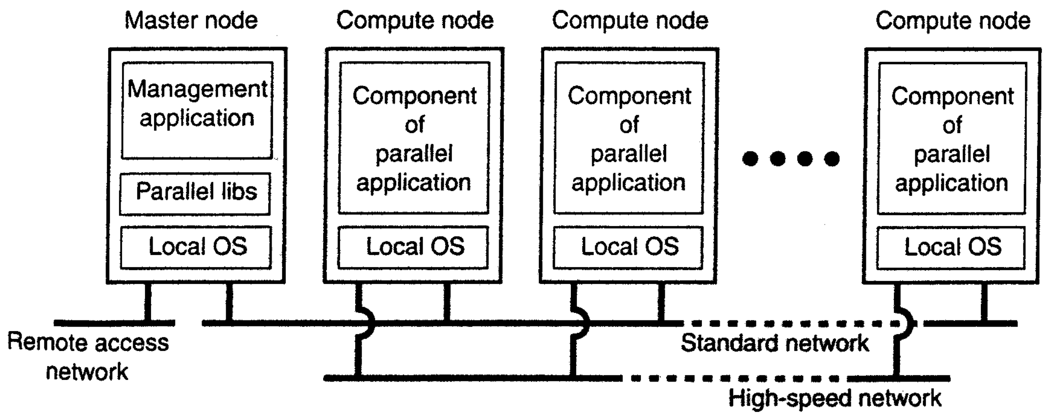


Figure 1-6. An example of a cluster computing system.

One well-known example of a cluster computer is formed by Linux-based Beowulf clusters, of which the general configuration is shown in Fig. 1-6. Each cluster consists of a collection of compute nodes that are controlled and accessed by means of a single master node. The master typically handles the allocation of nodes to a particular parallel program, maintains a batch queue of submitted jobs, and provides an interface for the users of the system. As such, the master actually runs the middleware needed for the execution of programs and management of the cluster, while the compute nodes often need nothing else but a standard operating system.

An important part of this middleware is formed by the libraries for executing parallel programs. As we will discuss in Chap. 4, many of these libraries effectively provide only advanced message-based communication facilities, but are not capable of handling faulty processes, security, etc.

As an alternative to this hierarchical organization, a symmetric approach is followed in the MOSIX system (Amar et at, 2004). MOSIX attempts to provide a single-system image of a cluster, meaning that to a process a cluster computer offers the ultimate distribution transparency by appearing to be a single computer. As we mentioned, providing such an image under all circumstances is impossible. In the case of MOSIX, the high degree of transparency is provided by allowing processes to dynamically and preemptively migrate between the nodes that make up the cluster. Process migration allows a user to start an application on any node (referred to as the home node), after which it can transparently move to other nodes, for example, to make efficient use of resources. We will return to process migration in Chap. 3.

Grid Computing Systems

A characteristic feature of cluster computing is its homogeneity. In most cases, the computers in a cluster are largely the same, they all have the same operating system, and are all connected through the same network. In contrast, grid computing systems have a high degree of heterogeneity: no assumptions are made concerning hardware, operating systems, networks, administrative domains, security policies, etc.

A key issue in a grid computing system is that resources from different organizations are brought together to allow the collaboration of a group of people or institutions. Such a collaboration is realized in the form of a virtual organization. The people belonging to the same virtual organization have access rights to the resources that are provided to that organization. Typically, resources consist of compute servers (including supercomputers, possibly implemented as cluster computers), storage facilities, and databases. In addition, special networked devices such as telescopes, sensors, etc., can be provided as well.

Given its nature, much of the software for realizing grid computing evolves around providing access to resources from different administrative domains, and to only those users and applications that belong to a specific virtual organization. For this reason, focus is often on architectural issues. An architecture proposed by Foster et al. (2001). is shown in Fig. 1-7

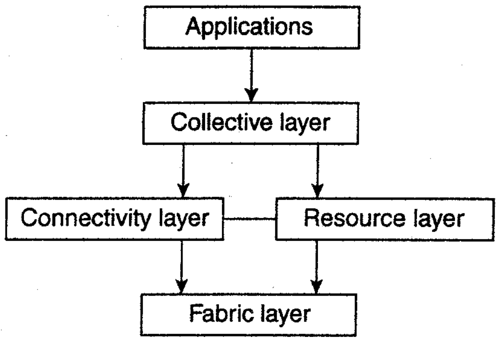


Figure 1-7. A layered architecture for grid computing systems.

The architecture consists of four layers. The lowest *fabric layer* provides interfaces to local resources at a specific site. Note that these interfaces are tailored to allow sharing of resources within a virtual organization. Typically, they will provide functions for querying the state and capabilities of a resource, along with functions for actual resource management (e.g., locking resources).

The *connectivity layer* consists of communication protocols for supporting grid transactions that span the usage of multiple resources. For example, protocols are needed to transfer data between resources, or to simply access a resource from a remote location. In addition, the connectivity layer will contain security protocols to authenticate users and resources. Note that in many cases human users are not authenticated; instead, programs acting on behalf of the users are authenticated. In this sense, delegating rights from a user to programs is an important function that needs to be supported in the connectivity layer. We return extensively to delegation when discussing security in distributed systems.

The *resource layer* is responsible for managing a single resource. It uses the functions provided by the connectivity layer and calls directly the interfaces made available by the fabric layer. For example, this layer will offer functions for obtaining configuration information on a specific resource, or, in general, to perform specific operations such as creating a process or reading data. The resource layer is thus seen to be responsible for access control, and hence will rely on the authentication performed as part of the connectivity layer.

The next layer in the hierarchy is the *collective layer.* It deals with handling access to multiple resources and typically consists of services for resource discovery, allocation and scheduling of tasks onto multiple resources, data replication, and so on. Unlike the connectivity and resource layer, which consist of a relatively small, standard collection of protocols, the collective layer may consist of many different protocols for many different purposes, reflecting the broad spectrum of services it may offer to a virtual organization.

Finally, the *application layer* consists of the applications that operate within a virtual organization and which make use of the grid computing environment..

Typically the collective, connectivity, and resource layer form the heart of what could be called a grid middleware layer. These layers jointly provide access to and management of resources that are potentially dispersed across multiple sites. An important observation from a middleware perspective is that with grid computing the notion of a site (or administrative unit) is common. This prevalence is emphasized by the gradual shift toward a service-oriented architecture in which sites offer access to the various layers through a collection of Web services (Joseph et al.. 2004). This, by now, has led to the definition of an alternative architecture known as the Open Grid Services Architecture (OGSA). This architecture consists of various layers and many components, making it rather complex. Complexity seems to be the fate of any standardization process. Details on OGSA can be found in Foster et al. (2005).

1.3.2 Distributed Information Systems

Another important class of distributed systems is found in organizations that were confronted with a wealth of networked applications, but for which interoperability turned out to be a painful experience. Many of the existing middleware solutions are the result of working with an infrastructure in which it was easier to integrate applications into an enterprise-wide information system (Bernstein, 1996; and Alonso et al., 2004).

We can distinguish several levels at which integration took place. In many cases, a networked application simply consisted of a server running that application (often including a database) and making it available to remote programs, called clients. Such clients could send a request to the server for executing a specific operation, after which a response would be sent back. Integration at the lowest level would allow clients to wrap a number of requests, possibly for different servers, into a single larger request and have it executed as a distributed transaction. The key idea was that all, or none of the requests would be executed.

As applications became more sophisticated and were gradually separated into independent components (notably distinguishing database components from processing components), it became clear that integration should also take place by letting applications communicate directly with each other. This has now led to a huge industry that concentrates on enterprise application integration (EAl). In the following, we concentrate on these two forms of distributed systems.

Transaction Processing Systems

To clarify our discussion, let us concentrate on database applications. In practice, operations on a database are usually carried out in the form of transactions. Programming using transactions requires special primitives that must either be supplied by the underlying distributed system or by the language runtime system. Typical examples of transaction primitives are shown in Fig. 1-8. The exact list of primitives depends on what kinds of objects are being used in the transaction (Gray and Reuter, 1993). In a mail system, there might be primitives to send, receive, and forward mail. In an accounting system, they might be quite different. READ and WRITE are typical examples, however. Ordinary statements, procedure calls, and so on, are also allowed inside a transaction. In particular, we mention that remote procedure calls (RPCs), that is, procedure calls to remote servers, are often also encapsulated in a transaction, leading to what is known as a transactional RPC. We discuss RPCs extensively in Chap. 4.

Primitive	**Description**
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Figure 1-8. Example primitives for transactions.

BEGIN_ TRANSACTION and END_TRANSACTION are used to delimit the scope of a transaction. The operations between them form the body of the transaction. The characteristic feature of a transaction is either all of these operations are executed or none are executed. These may be system calls, library procedures, or bracketing statements in a language, depending on the implementation.

This all-or-nothing property of transactions is one of the four characteristic properties that transactions have. More specifically, transactions are:

1. Atomic: To the outside world, the transaction happens indivisibly.
2. Consistent: The transaction does not violate system invariants.
3. Isolated: Concurrent transactions do not interfere with each other.
4. Durable: Once a transaction commits, the changes are permanent.

These properties are often referred to by their initial letters: ACID.

The first key property exhibited by all transactions is that they are atomic. This property ensures that each transaction either happens completely, or not at all, and if it happens, it happens in a single indivisible, instantaneous action. While a transaction is in progress, other processes (whether or not they are themselves involved in transactions) cannot see any of the intermediate states.

The second property says that they are consistent. What this means is that if the system has certain invariants that must always hold, if they held before the transaction, they will hold afterward too. For example. in a banking system, a key invariant is the law of conservation of money. After every internal transfer, the amount of money in the bank must be the same as it was before the transfer, but for a brief moment during the transaction, this invariant may be violated. The violation is not visible outside the transaction, however.

The third property says that transactions are isolated or serializable. What it means is that if two or more transactions are running at the same time, to each of them and to other processes, the final result looks as though all transactions ian sequentially in some (system dependent) order.

The fourth property says that transactions are durable. It refers to the fact that once a transaction commits, no matter what happens, the transaction goes forward and the results become permanent. No failure after the commit can undo the results or cause them to be lost. (Durability is discussed extensively in Chap. 8.)

So far, transactions have been defined on a single database. A nested transaction is constructed from a number of subtransactions, as shown in Fig. 1-9. The top-level transaction may fork off children that run in parallel with one another, on different machines, to gain performance or simplify programming. Each of these children may also execute one or more subtransactions, or fork off its own children.

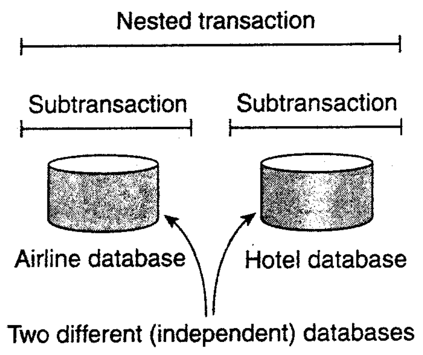


Figure 1-9. A nested transaction.

Subtransactions give rise to a subtle, but important, problem. Imagine that a transaction starts several subtransactions in parallel, and one of these commits. making its results visible to the parent transaction. After further computation, the parent aborts, restoring the entire system to the state it had before the top-level transaction started. Consequently, the results of the subtransaction that committed must nevertheless be undone. Thus the permanence referred to above applies only to top-level transactions.

Since transactions can be nested arbitrarily deeply, considerable administration is needed to get everything right. The semantics are clear, however. When any transaction or subtransaction starts, it is conceptually given a private copy of all data in the entire system for it to manipulate as it wishes. If it aborts, its private universe just vanishes, as if it had never existed. If it commits, its private universe replaces the parent's universe. Thus if a subtransaction commits and then later a new subtransaction is started, the second one sees the results produced by the first one. Likewise, if an enclosing (higher-level) transaction aborts, all its underlying subtransactions have to be aborted as well.

Nested transactions are important in distributed systems, for they provide a natural way of distributing a transaction across multiple machines. They follow a *logical* division of the work of the original transaction. For example, a transaction for planning a trip by which three different flights need to be reserved can be logically split up into three subtransactions. Each of these subtransactions can be managed separately and independent of the other two.

In the early days of enterprise middleware systems, the component that handled distributed (or nested) transactions formed the core for integrating applications at the server or database level. This component was called a transaction processing monitor or TP monitor for short. Its main task was to allow an application to access multiple server/databases by offering it a transactional programming model, as shown in Fig. 1-10.

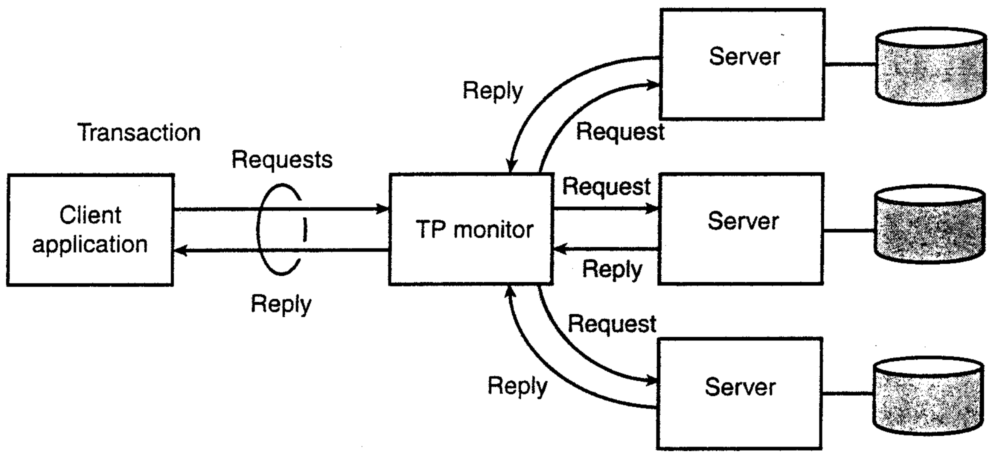


Figure 1-10. The role of a TP monitor in distributed systems.

Enterprise Application Integration

As mentioned, the more applications became decoupled from the databases they were built upon, the more evident it became that facilities were needed to integrate applications independent from their databases. In particular, application components should be able to communicate directly with each other and not merely by means of the request/reply behavior that was supported by transaction processing systems.

This need for interapplication communication led to many different communication models, which we will discuss in detail in this book (and for which reason we shall keep it brief for now). The main idea was that existing applications could directly exchange information, as shown in Fig. 1-11.

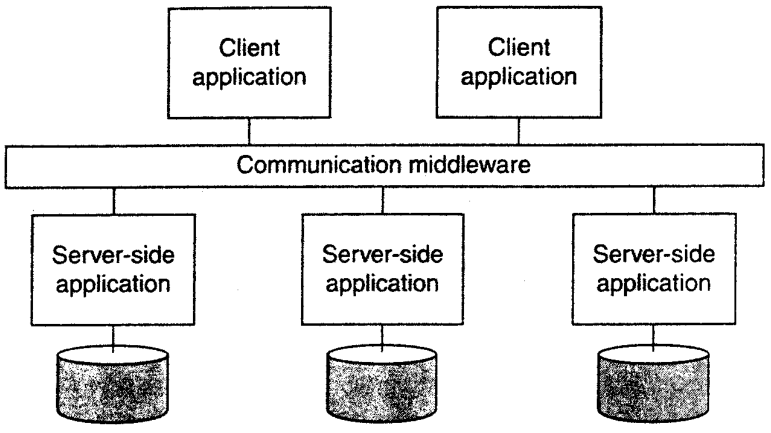


Figure 1-11. MiddJeware as a communication facilitator in enterprise application integration.

Several types of communication middleware exist. With remote procedure calls (RPC), an application component can effectively send a request to another application component by doing a local procedure call, which results in the request being packaged as a message and sent to the callee. Likewise, the result will be sent back and returned to the application as the result of the procedure call.

As the popularity of object technology increased, techniques were developed to allow calls to remote objects, leading to what is known as remote method invocations (RMI). An RMI is essentially the same as an RPC, except that it operates on objects instead of applications.

RPC and RMI have the disadvantage that the caller and callee both need to be up and running at the time of communication. In addition, they need to know exactly how to refer to each other. This tight coupling is often experienced as a serious drawback, and has led to what is known as message-oriented middleware, or simply MOM. In this case, applications simply send messages to logical contact points, often described by means of a subject. Likewise, applications can indicate their interest for a specific type of message, after which the communication middleware will take care that those messages are delivered to those applications. These so-called publish/subscribe systems form an important and expanding class of distributed systems. We will discuss them at length in Chap. 13.

1.3.3 Distributed Pervasive Systems

The distributed systems we have been discussing so far are largely characterized by their stability: nodes are fixed and have a more or less permanent and high-quality connection to a network. To a certain extent, this stability has been realized through the various techniques that are discussed in this book and which aim at achieving distribution transparency. For example, the wealth of techniques for masking failures and recovery will give the impression that only occasionally things may go wrong. Likewise, we have been able to hide aspects related to the actual network location of a node, effectively allowing users and applications to believe that nodes stay put.

However, matters have become very different with the introduction of mobile and embedded computing devices. We are now confronted with distributed systems in which instability is the default behavior. The devices in these, what we refer to as **distributed** pervasive systems, are often characterized by being small, battery-powered, mobile, and having only a wireless connection, although not all these characteristics apply to all devices. Moreover, these characteristics need not necessarily be interpreted as restrictive, as is illustrated by the possibilities of modem smart phones (Roussos et aI., 2005).

As its name suggests, a distributed pervasive system is part of our surroundings (and as such, is generally inherently distributed). An important feature is the general lack of human administrative control. At best, devices can be configured by their owners, but otherwise they need to automatically discover their environment and "nestle in" as best as possible. This nestling in has been made more precise by Grimm et al. (2004) by formulating the following three requirements for pervasive applications:

1. Embrace contextual changes.
2. Encourage ad hoc composition.
3. Recognize sharing as the default.

Embracing contextual changes means that a device must be continuously be aware of the fact that its environment may change all the time. One of the simplest changes is discovering that a network is no longer available, for example, because a user is moving between base stations. In such a case, the application should react, possibly by automatically connecting to another network, or taking other appropriate actions.

Encouraging ad hoc composition refers to the fact that many devices in pervasive systems will be used in very different ways by different users. As a result, it should be easy to configure the suite of applications running on a device, either by the user or through automated (but controlled) interposition.

One very important aspect of pervasive systems is that devices generally join the system in order to access (and possibly provide) information. This calls for means to easily read, store, manage, and share information. In light of the intermittent and changing connectivity of devices, the space where accessible information resides will most likely change all the time.

Mascolo et al. (2004) as well as Niemela and Latvakoski (2004) came to similar conclusions: in the presence of mobility, devices should support easy and application-dependent adaptation to their local environment. They should be able to efficiently discover services and react accordingly. It should be clear from these requirements that distribution transparency is not really in place in pervasive systems. In fact, distribution of data, processes, and control is *inherent* to these systems, for which reason it may be better just to simply expose it rather than trying to hide it. Let us now take a look at some concrete examples of pervasive systems.

Home Systems

An increasingly popular type of pervasive system, but which may perhaps be the least constrained, are systems built around home networks. These systems generally consist of one or more personal computers, but more importantly integrate typical consumer electronics such as TVs, audio and video equipment, gaming devices, (smart) phones, PDAs, and other personal wearables into a single system. In addition, we can expect that all kinds of devices such as kitchen appliances, surveillance cameras, clocks, controllers for lighting, and so on, will all be hooked up into a single distributed system.

From a system's perspective there are several challenges that need to be addressed before pervasive home systems become reality. An important one is that such a system should be completely self-configuring and self-managing. It cannot be expected that end users are willing and able to keep a distributed home system up and running if its components are prone to errors (as is the case with many of today's devices.) Much has already been accomplished through the Universal Plug and Play (UPnP) standards by which devices automatically obtain IP addresses, can discover each other, etc. (DPnP Forum, 2003). However, more is needed. For example, it is unclear how software and firmware in devices can be easily updated without manual intervention, or when updates do take place, that compatibility with other devices is not violated.

Another pressing issue is managing what is known as a *"personal space."* Recognizing that a home system consists of many shared as well as personal devices, and that the data in a home system is also subject to sharing restrictions, much attention is paid to realizing such personal spaces. For example, part of Alice's personal space may consist of her agenda, family photo's, a diary, music and videos that she bought, etc. These personal assets should be stored in such a way that Alice has access to them whenever appropriate. Moreover, parts of this personal space should be (temporarily) accessible to others, for example, when she needs to make a business appointment.

Fortunately, things may become simpler. It has long been thought that the personal spaces related to home systems were inherently distributed across the various devices. Obviously, such a dispersion can easily lead to significant synchronization problems. However, problems may be alleviated due to the rapid increase in the capacity of hard disks, along with a decrease in their size. Configuring a multi-terabyte storage unit for a personal computer is not really a problem. At the same time, portable hard disks having a capacity of hundreds of gigabytes are being placed inside relatively small portable media players. With these continuously increasing capacities, we may see pervasive home systems adopt an architecture in which a single machine acts as a master (and is hidden away somewhere in the basement next to the central heating), and all other fixed devices simply provide a convenient interface for humans. Personal devices will then be crammed with daily needed information, but will never run out of storage.

However, having enough storage does not solve the problem of managing personal spaces. Being able to store huge amounts of data shifts the problem to storing *relevant* data and being able to find it later. Increasingly we will see pervasive systems, like home networks, equipped with what are called recommenders, programs that consult what other users have stored in order to identify similar taste, and from that subsequently derive which content to place in one's personal space. An interesting observation is that the amount of information that recommender programs need to do their work is often small enough to allow them to be run on PDAs (Miller et al., 2004).

Electronic Health Care Systems

Another important and upcoming class of pervasive systems are those related to (personal) electronic health care. With the increasing cost of medical treatment, new devices are being developed to monitor the well-being of individuals and to automatically contact physicians when needed. In many of these systems, a major goal is to prevent people from being hospitalized.

Personal health care systems are often equipped with various sensors organized in a (preferably wireless) body-area network (BAN). An important issue is that such a network should at worst only minimally hinder a person. To this end, the network should be able to operate while a person is moving, with no strings (i.e., wires) attached to immobile devices.

This requirement leads to two obvious organizations, as shown in Fig. 1-12. In the first one, a central hub is part of the BAN and collects data as needed. From time to time, this data is then offloaded to a larger storage device. The advantage of this scheme is that the hub can also manage the BAN. In the second scenario, the BAN is continuously hooked up to an external network, again through a wireless connection, to which it sends monitored data. Separate techniques will need to be deployed for managing the BAN. Of course, further connections to a physician or other people may exist as well.

From a distributed system's perspective we are immediately confronted with questions such as:

1. Where and how should monitored data be stored?
2. How can we prevent loss of crucial data?
3. What infrastructure is needed to generate and propagate alerts?

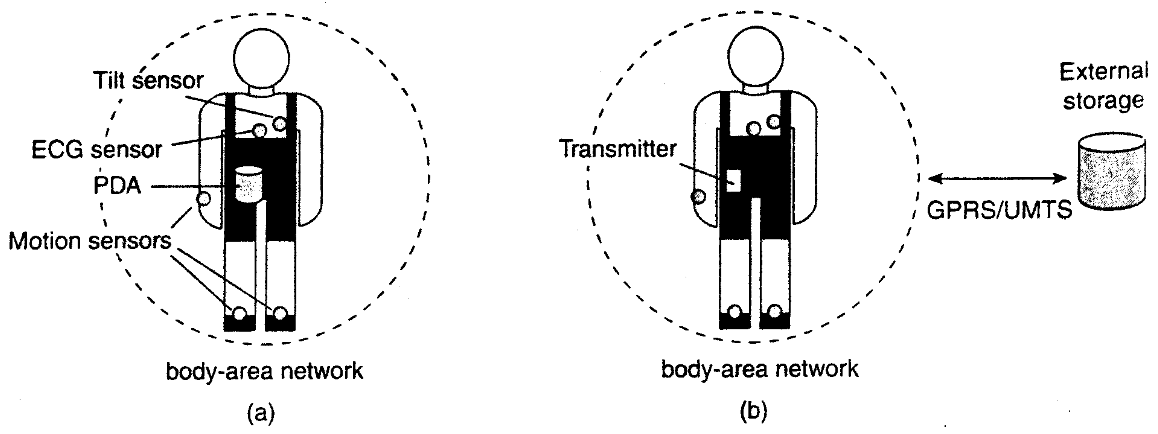


Figure 1-12. Monitoring a person in a pervasive electronic health care system, using (a) a local hub or (b) a continuous wireless connection.

4. How can physicians provide online feedback?
5. How can extreme robustness of the monitoring system be realized?
6. What are the security issues and how can the proper policies be enforced?

Unlike home systems, we cannot expect the architecture of pervasive health care systems to move toward single-server systems and have the monitoring devices operate with minimal functionality. On the contrary: for reasons of efficiency, devices and body-area networks will be required to support in-network data processing, meaning that monitoring data will, for example, have to be aggregated before permanently storing it or sending it to a physician. Unlike the case for distributed information systems, there is yet no clear answer to these questions.

Sensor Networks

Our last example of pervasive systems is sensor networks. These networks in many cases form part of the enabling technology for pervasiveness and we see that many solutions for sensor networks return in pervasive applications. What makes sensor networks interesting from a distributed system's perspective is that in virtually all cases they are used for processing information. In this sense, they do more than just provide communication services. which is what traditional computer networks are all about. Akyildiz et al. (2002) provide an overview from a networking perspective. A more systems-oriented introduction to sensor networks is given by Zhao and Guibas (2004). Strongly related are mesh networks which essentially form a collection of (fixed) nodes that communicate through wireless links. These networks may form the basis for many medium-scale distributed systems. An overview is provided in Akyildiz et al. (2005).

A sensor network typically consists of tens to hundreds or thousands of relatively small nodes, each equipped with a sensing device. Most sensor networks use wireless communication, and the nodes are often battery powered. Their limited resources, restricted communication capabilities, and constrained power consumption demand that efficiency be high on the list of design criteria.

The relation with distributed systems can be made clear by considering sensor networks as distributed databases. This view is quite common and easy to understand when realizing that many sensor networks are deployed for measurement and surveillance applications (Bonnet et aL., 2002). In these cases, an operator would like to extract information from (a part of) the network by simply issuing queries such as "What is the northbound traffic load on Highway I?" Such queries resemble those of traditional databases. In this case, the answer will probably need to be provided through collaboration of many sensors located around Highway 1, while leaving other sensors untouched.

To organize a sensor network as a distributed database, there are essentially two extremes, as shown in Fig. 1-13. First, sensors do not cooperate but simply send their data to a centralized database located at the operator's site. The other extreme is to forward queries to relevant sensors and to let each compute an answer, requiring the operator to sensibly aggregate the returned answers.

Neither of these solutions is very attractive. The first one requires that sensors send all their measured data through the network, which may waste network resources and energy. The second solution may also be wasteful as it discards the aggregation capabilities of sensors which would allow much less data to be returned to the operator. What is needed are facilities for in-network data processing, as we also encountered in pervasive health care systems.

In-network processing can be done in numerous ways. One obvious one is to forward a query to all sensor nodes along a tree encompassing all nodes and to subsequently aggregate the results as they are propagated back to the root, where the initiator is located. Aggregation will take place where two or more branches of the tree come to together. As simple as this scheme may sound, it introduces difficult questions:

1. How do we (dynamically) set up an efficient tree in a sensor network?

2. How does aggregation of results take place? Can it be controlled?

3. What happens when network links fail?

These questions have been partly addressed in TinyDB, which' implements a declarative(database) interface to wireless sensor networks. In essence, TinyDB can use any tree-based routing algorithm. An intermediate node will collect and aggregate the results from its children, along with its own findings, and send that toward the root. To make matters efficient, queries span a period of time allowing

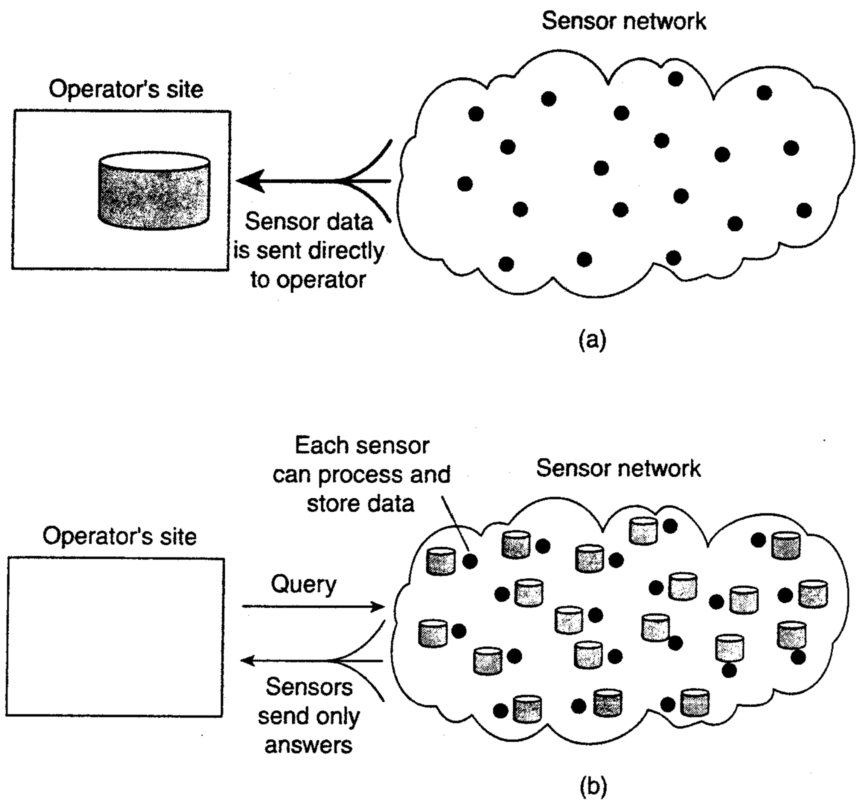

Sensor network
Operator's site
Sensor data is sent directly to operator
(a)
Each sensor can process and store data
Sensor network
Operator's site
Query
Sensors send only answers
(b)

Figure 1-13. Organizing a sensor network database, while storing and processing data (a) only at the operator's site or (b) only at the sensors.

for careful scheduling of operations so that network resources and energy are optimally consumed. Details can be found in Madden et al. (2005).

However, when queries can be initiated from different points in the network, using single-rooted trees such as in TinyDB may not be efficient enough. As an alternative, sensor networks may be equipped with special nodes where results are forwarded to, as well as the queries related to those results. To give a simple example, queries and results related temperature readings are collected at a different location than those related to humidity measurements. This approach corresponds directly to the notion of publish/subscribe systems, which we will discuss extensively in Chap. 13.

1.4 SUMMARY

Distributed systems consist of autonomous computers that work together to give the appearance of a single coherent system. One important advantage is that they make it easier to integrate different applications running on different computers into a single system. Another advantage is that when properly designed, distributed systems scale well with respect to the size of the underlying network. These advantages often come at the cost of more complex software, degradation of performance, and also often weaker security. Nevertheless, there is considerable interest worldwide in building and installing distributed systems.

Distributed systems often aim at hiding many of the intricacies related to the distribution of processes, data, and control. However, this distribution transparency not only comes at a performance price, but in practical situations it can never be fully achieved. The fact that trade-offs need to be made between achieving various forms of distribution transparency is inherent to the design of distributed systems, and can easily complicate their understanding. -

Matters are further complicated by the fact that many developers initially make assumptions about the underlying network that are fundamentally wrong. Later, when assumptions are dropped, it may turn out to be difficult to mask unwanted behavior. A typical example is assuming that network latency is not significant. Later, when porting an existing system to a wide-area network, hiding latencies may deeply affect the system's original design. Other pitfalls include assuming that the network is reliable, static, secure, and homogeneous.

Different types of distributed systems exist which can be classified as being oriented toward supporting computations, information processing, and pervasiveness. Distributed computing systems are typically deployed for high-performance applications often originating from the field of parallel computing. A huge class of distributed can be found in traditional office environments where we see databases playing an important role. Typically, transaction processing systems are deployed in these environments. Finally, an emerging class of distributed systems is where components are small and the system is composed in an ad hoc fashion, but most of all is no longer managed through a system administrator. This last class is typically represented by ubiquitous computing environments.

PROBLEMS

1. An alternative definition for a distributed system is that of a collection of independent computers providing the view of being a *single system,* that is, it is completely hidden from users that there even multiple computers. Give an example where this view would come in very handy.

2. What is the role of middleware in a distributed system?

3. Many networked systems are organized in terms of a back office and a front office. How does organizations match with the coherent view we demand for a distributed ~~m? –

4. Explain what is meant by (distribution) transparency, and give examples of different types of transparency.

5. Why is it sometimes so hard to hide the occurrence and recovery from failures in a distributed system?

6. Why is it not always a good idea to aim at implementing the highest degree of transparency possible?

7. What is an open distributed system and what benefits does openness provide?

8. Describe precisely what is meant by a scalable system.

9. Scalability can be achieved by applying different techniques. What are these techniques?

10. Explain what is meant by a virtual organization and give a hint on how such organizations could be implemented.

11. When a transaction is aborted. we have said that the world is restored to its previous state. as though the transaction had never happened. We lied. Give an example where resetting the world is impossible.

12. Executing nested transactions requires some form of coordination. Explain what a coordinator should actually do.

13. We argued that distribution transparency may not be in place for pervasive systems. This statement is not true for all types of transparencies. Give an example.

14. We already gave some examples of distributed pervasive systems: home systems. electronic health-care systems. and sensor networks. Extend this list with more examples.

15. (Lab assignment) Sketch a design for a home system consisting of a separate media server that will allow for the attachment of a wireless client. The latter is connected to (analog) audio/video equipment and transforms the digital media streams to analog output. The server runs on a separate machine. possibly connected to the Internet. but has no keyboard and/or monitor connected.

2

ARCHITECTURES

Distributed systems are often complex pieces of software of which the components are by definition dispersed across multiple machines. To master their complexity, it is crucial that these systems are properly organized. There are different ways on how to view the organization of a distributed system, but an obvious one is to make a distinction between the logical organization of the collection of software components and on the other hand the actual physical realization.

The organization of distributed systems is mostly about the software components that constitute the system. These **software architectures** tell us how the various software components are to be organized and how they should interact. In this chapter we will first pay attention to some commonly applied approaches toward organizing (distributed) computer systems.

The actual realization of a distributed system requires that we instantiate and place software components on real machines. There are many different choices that can be made in doing so. The final instantiation of a software architecture is also referred to as a **system architecture**. In this chapter we will look into traditional centralized architectures in which a single server implements most of the software components (and thus functionality), while remote clients can access that server using simple communication means. In addition, we consider decentralized architectures in which machines more or less play equal roles, as well as hybrid organizations.

As we explained in Chap. I, an important goal of distributed systems is to separate applications from underlying platforms by providing a middleware layer.

Adopting such a layer is an important architectural decision, and its main purpose is to provide distribution transparency. However, trade-offs need to be made to achieve transparency, which has led to various techniques to make middleware adaptive. We discuss some of the more commonly applied ones in this chapter, as they affect the organization of the middleware itself.

Adaptability in distributed systems can also be achieved by having the system monitor its own behavior and taking appropriate measures when needed. This 'insight has led to a class of what are now referred to as autonomic systems. These distributed systems are frequently organized in the form of feedback control loops. which form an important architectural element during a system's design. In this chapter, we devote a section to autonomic distributed systems.

2.1 ARCHITECTURAL STYLES

We start our discussion on architectures by first considering the logical organization of distributed systems into software components, also referred to as software architecture (Bass et al., 2003). Research on software architectures has matured considerably and it is now commonly accepted that designing or adopting an architecture is crucial for the successful development of large systems.

For our discussion, the notion of an architectural style is important.. Such a style is formulated in terms of components, the way that components are connected to each other, the data exchanged between components. and finally how these elements are jointly configured into a system. A component is a modular unit with well-defined required and provided interfaces that is replaceable within its environment (OMG, 2004b). As we shall discuss below, the important issue about a component for distributed systems is that it can be replaced, provided we respect its interfaces. A somewhat more difficult concept to grasp is that of a connector, which is generally described as a mechanism that mediates communication, coordination, or cooperation among components (Mehta et al., 2000; and Shaw and Clements, 1997). For example, a connector can be formed by the facilities for (remote) procedure calls, message passing, or streaming data.

Using components and connectors, we can come to various configurations, which, in tum have been classified into architectural styles. Several styles have by now been identified, of which the most important ones for distributed systems are:

1. Layered architectures
2. Object-based architectures
3. Data-centered architectures
4. Event-based architectures

The basic idea for the layered style is simple: components are organized in a layered fashion where a component at layer *L;* is allowed to call components at the underlying layer L_{i-1}, but not the other way around, as shown in Fig. 2-1(a). This model has been widely adopted by the networking community; we briefly review it in Chap. 4. An key observation is that control generally flows from layer to layer: requests go down the hierarchy whereas the results flow upward.

A far looser organization is followed in object-based architectures, which are illustrated in Fig. 2-1(b). In essence, each object corresponds to what we have defined as a component, and these components are connected through a (remote) procedure call mechanism. Not surprisingly, this software architecture matches the client-server system architecture we described above. The layered and object-based architectures still form the most important styles for large software systems (Bass et al., 2003).

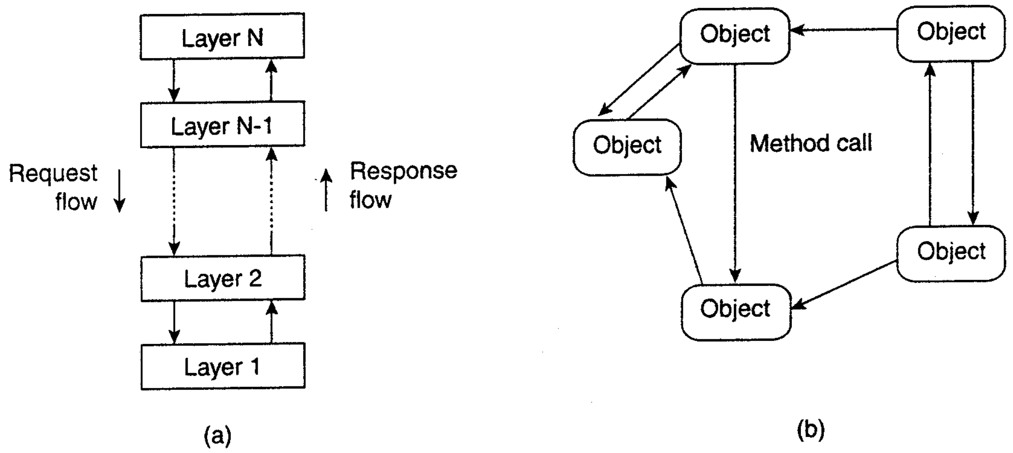


Figure 2-1. The (a) layered and (b) object-based architectural style.

Data-centered architectures evolve around the idea that processes communicate through a common (passive or active) repository. It can be argued that for distributed systems these architectures are as important as the layered and object-based architectures. For example, a wealth of networked applications have been developed that rely on a shared distributed file system in which virtually all communication takes place through files. Likewise, Web-based distributed systems, which we discuss extensively in Chap. 12, are largely data-centric: processes communicate through the use of shared Web-based data services.

In event-based architectures, processes essentially communicate through the propagation of events, which optionally also carry data, as shown in Fig. 2-2(a). For distributed systems, event propagation has generally been associated with what are known as publish/subscribe systems (Eugster et al., 2003). The basic idea is that processes publish events after which the middleware ensures that only those processes that subscribed to those events will receive them. The main advantage of event-based systems is that processes are loosely coupled. In principle, they need not explicitly refer to each other. This is also referred to as being decoupled in space, or referentially decoupled.

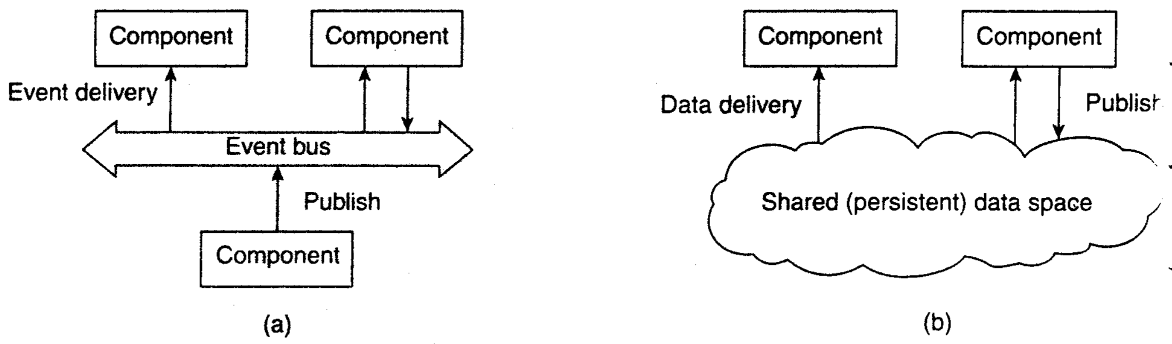


Figure 2-2. The (a) event-based and (b) shared data-space architectural style.

Event-based architectures can be combined with data-centered architectures, yielding what is also known as shared data. spaces. The essence of shared data spaces is that processes are now also decoupled in time: they need not both be active when communication takes place. Furthermore, many shared data spaces use a SQL-like interface to the shared repository in that sense that data can be accessed using a description rather than an explicit reference, as is the case with files. We devote Chap. 13 to this architectural style.

What makes these software architectures important for distributed systems is that they all aim at achieving (at a reasonable level) distribution transparency. However, as we have argued, distribution transparency requires making trade-offs between performance, fault tolerance, ease-of-programming, and so on. As there is no single solution that will meet the requirements for all possible distributed applications, researchers have abandoned the idea that a single distributed system can be used to cover 90% of all possible cases.

2.2 SYSTEM ARCHITECTURES

Now that we have briefly discussed some common architectural styles, let us take a look at how many distributed systems are actually organized by considering where software components are placed. Deciding on software components, their interaction, and their placement leads 10 an instance of a software architecture, also called a system architecture (Bass et aI., 2003). We will discuss centralized and decentralized organizations, as wen as various hybrid forms.

2.2.1 Centralized Architectures

Despite the lack of consensus on many distributed systems issues, there is one issue that many researchers and practitioners agree upon: thinking in terms of *clients* that request services from *servers* helps us understand and manage the complexity of distributed systems and that is a good thing.

In the basic client-server model, processes in a distributed system are divided into two (possibly overlapping) groups. A server is a process implementing a specific service, for example, a file system service or a database service. A client is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply. This client-server interaction, also known as request-reply behavior is shown in Fig. 2-3

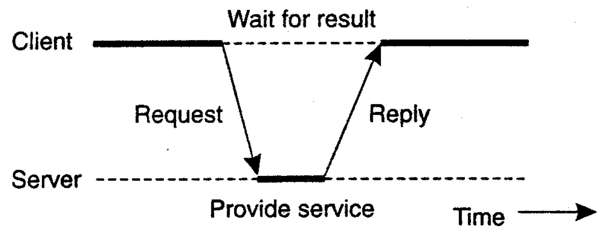


Figure 2-3. General interaction between a client and a server.

Communication between a client and a server can be implemented by means of a simple connectionless protocol when the underlying network is fairly reliable as in many local-area networks. In these cases, when a client requests a service, it simply packages a message for the server, identifying the service it wants, along with the necessary input data. The message is then sent to the server. The latter, in turn, will always wait for an incoming request, subsequently process it, and package the results in a reply message that is then sent to the client.

Using a connectionless protocol has the obvious advantage of being efficient. As long as messages do not get lost or corrupted, the request/reply protocol just sketched works fine. Unfortunately, making the protocol resistant to occasional transmission failures is not trivial. The only thing we can do is possibly let the client resend the request when no reply message comes in. The problem, however, is that the client cannot detect whether the original request message was lost, or that transmission of the reply failed. If the reply was lost, then resending a request may result in performing the operation twice. If the operation was something like "transfer $10,000 from my bank account," then clearly, it would have been better that we simply reported an error instead. On the other hand, if the operation was "tell me how much money I have left," it would be perfectly acceptable to resend the request. When an operation can be repeated multiple times without harm, it is said to be idempotent. Since some requests are idempotent and others are not it should be clear that there is no single solution for dealing with lost messages. We defer a detailed discussion on handling transmission failures to Chap. 8.

As an alternative, many client-server systems use a reliable connection-oriented protocol. Although this solution is not entirely appropriate in a local-area network due to relatively low performance, it works perfectly tine in wide-area systems in which communication is inherently unreliable. For example, virtually all Internet application protocols are based on reliable TCPIIPconnections. In this case, whenever a client requests a service, it first sets up a connection to the server before sending the request. The server generally uses that same connection to send the reply message, after which the connection is torn down. The trouble is that setting up and tearing down a connection is relatively costly, especially when the request and reply messages are small.

Application Layering

The client-server model has been subject to many debates and controversies over the years. One of the main issues was how to draw a clear distinction between a client and a server. Not surprisingly, there is often no clear distinction. For example, a server for a distributed database may continuously act as a client because it is forwarding requests to different file servers responsible for implementing the database tables. In such a case, the database server itself essentially does no more than process queries.

However, considering that many client-server applications are targeted toward supporting user access to databases, many people have advocated a distinction between the following three levels, essentially following the layered architectural style we discussed previously:

1. The user-interface level
2. The processing level
3. The data level

The user-interface level contains all that is necessary to directly interface with the user, such as display management. The processing level typically contains the applications. The data level manages the actual data that is being acted on.

Clients typically implement the user-interface level. This level consists of the programs that allow end users to interact with applications. There is a considerable difference in how sophisticated user-interface programs are.

The simplest user-interface program is nothing more than a character-based screen. Such an interface has been typically used in mainframe environments. In those cases where the mainframe controls all interaction, including the keyboard and monitor, one can hardly speak of a client-server environment. However, in many cases, the user's terminal does some local processing such as echoing typed keystrokes, or supporting form-like interfaces in which a complete entry is to be edited before sending it to the main computer.

Nowadays, even in mainframe environments, we see more advanced user interfaces. Typically, the client machine offers at least a graphical display in which pop-up or pull-down menus are used, and of which many of the screen controls are handled through a mouse instead of the keyboard. Typical examples of such interfaces include the X-Windows interfaces as used in many UNIX environments, and earlier interfaces developed for MS-DOS PCs and Apple Macintoshes.

Modern user interfaces offer considerably more functionality by allowing applications to share a single graphical window, and to use that window to exchange data through user actions. For example, to delete a file, it is usually possible to move the icon representing that file to an icon representing a trash can. Likewise, many word processors allow a user to move text in a document to another position by using only the mouse. We return to user interfaces in Chap. 3.

Many client-server applications can be constructed from roughly three different pieces: a part that handles interaction with a user, a part that operates on a database or file system, and a middle part that generally contains the core functionality of an application. This middle part is logically placed at the processing level. In contrast to user interfaces and databases, there are not many aspects common to the processing level. Therefore, we shall give several examples to make this level clearer.

As a first example, consider an Internet search engine. Ignoring all the animated banners, images, and other fancy window dressing, the user interface of a search engine is very simple: a user types in a string of keywords and is subsequently presented with a list of titles of Web pages. The back end is formed by a huge database of Web pages that have been prefetched and indexed. The core of the search engine is a program that transforms the user's string of keywords into one or more database queries. It subsequently ranks the results into a list, and transforms that list into a series of HTML pages. Within the client-server model, this information retrieval part is typically placed at the processing level. Fig. 2-4 shows this organization.

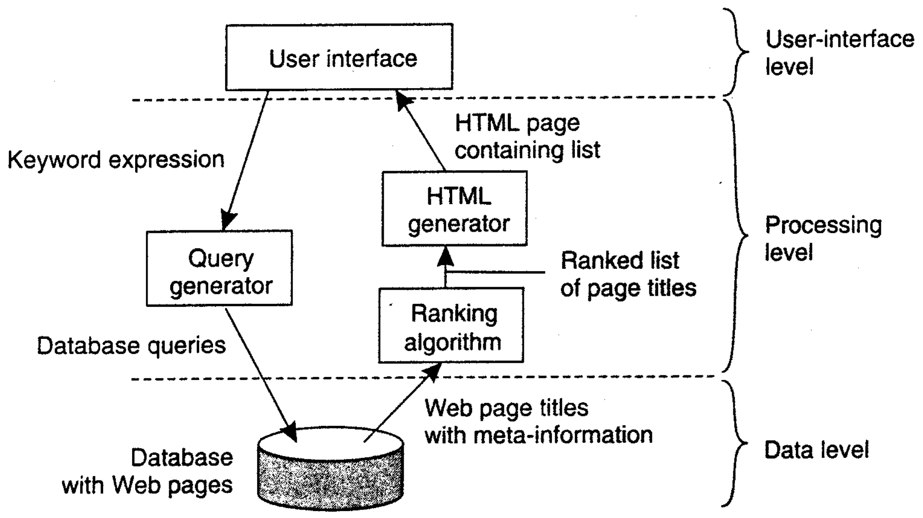


Figure 2-4. The simplified organization of an Internet search engine into three different layers.

As a second example, consider a decision support system for a stock brokerage. Analogous to a search engine, such a system can be divided into a front end implementing the user interface, a back end for accessing a database with the financial data, and the analysis programs between these two. Analysis of financial data may require sophisticated methods and techniques from statistics and artificial intelligence. In some cases, the core of a financial decision support system may even need to be executed on high-performance computers in order to achieve the throughput and responsiveness that is expected from its users.

As a last example, consider a typical desktop package, consisting of a word processor, a spreadsheet application, communication facilities, and so on. Such "office" suites are generally integrated through a common user interface that supports compound documents, and operates on files from the user's home directory. (In an office environment, this home directory is often placed on a remote file server.) In this example, the processing level consists of a relatively large collection of programs, each having rather simple processing capabilities.

The data level in the client-server model contains the programs that maintain the actual data on which the applications operate. An important property of this level is that data are often persistent, that is, even if no application is running, data will be stored somewhere for next use. In its simplest form, the data level consists of a file system, but it is more common to use a full-fledged database. In the client-server model, the data level is typically implemented at the server side.

Besides merely storing data, the data level is generally also responsible for keeping data consistent across different applications. When databases are being used, maintaining consistency means that metadata such as table descriptions, entry constraints and application-specific metadata are also stored at this level. For example, in the case of a bank, we may want to generate a notification when a customer's credit card debt reaches a certain value. This type of information can be maintained through a database trigger that activates a handler for that trigger at the appropriate moment.

In most business-oriented environments, the data level is organized as a relational database. Data independence is crucial here.. The data are organized independent of the applications in such a way that changes in that organization do not affect applications, and neither do the applications affect the data organization. Using relational databases in the client-server model helps separate the processing level from the data level, as processing and data are considered independent.

However, relational databases are not always the ideal choice. A characteristic feature of many applications is that they operate on complex data types that are more easily modeled in terms of objects than in terms of relations. Examples of such data types range from simple polygons and circles to representations of aircraft designs, as is the case with computer-aided design (CAD) systems.

In those cases where data operations are more easily expressed in terms of object manipulations, it makes sense to implement the data level by means of an object-oriented or object-relational database. Notably the latter type has gained popularity as these databases build upon the widely dispersed relational data model, while offering the advantages that object-orientation gives.

Multitiered Architectures

The distinction into three logical levels as discussed so far, suggests a number of possibilities for physically distributing a client-server application across several machines. The simplest organization is to have only two types of machines:

1. A client machine containing only the programs implementing (part of) the user-interface level
2. A server machine containing the rest, that is the programs implementing the processing and data level

In this organization everything is handled by the server while the client is essentially no more than a dumb terminal, possibly with a pretty graphical interface. There are many other possibilities, of which we explore some of the more common ones in this section.

One approach for organizing the clients and servers is to distribute the programs in the application layers of the previous section across different machines, as shown in Fig. 2-5 [see also Umar (1997); and Jing et al. (1999)]. As a first step, we make a distinction between only two kinds of machines: client machines and server machines, leading to what is also referred to as a (physically) two-tiered architecture.

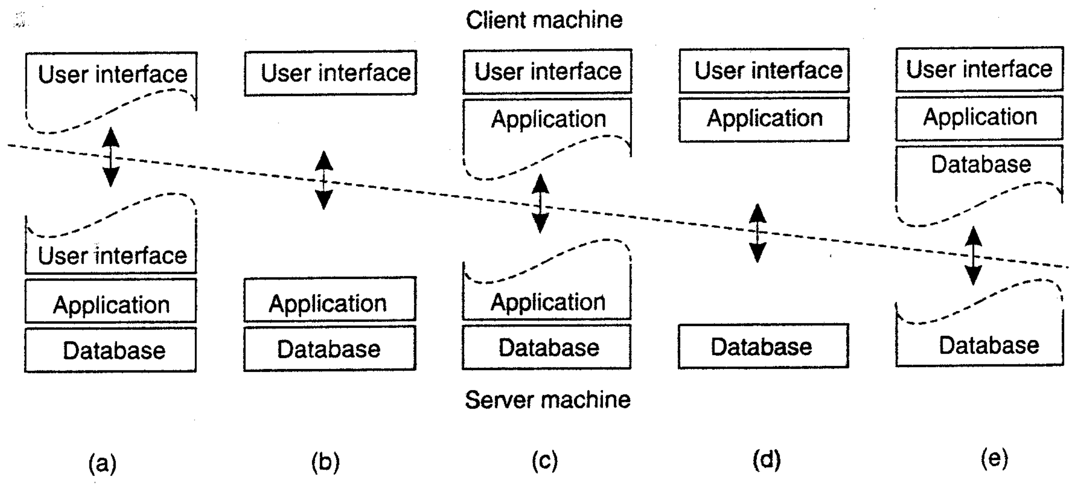


Figure 2-5. Alternative client-server organizations (a)-(e).

One possible organization is to have only the terminal-dependent part of the user interface on the client machine, as shown in Fig. 2-5(a), and give the applications remote control over the presentation of their data. An alternative is to place the entire user-interface software on the client side, as shown in Fig. 2-5(b). In such cases, we essentially divide the application into a graphical front end, which communicates with the rest of the application (residing at the server) through an application-specific protocol. In this model, the front end (the client software) does no processing other than necessary for presenting the application's interface.

Continuing along this line of reasoning, we may also move part of the application to the front end, as shown in Fig. 2-5(c). An example where this makes sense is where the application makes use of a form that needs to be filled in entirely before it can be processed. The front end can then check the correctness and consistency of the form, and where necessary interact with the user. Another example of the organization of Fig. 2:.5(c), is that of a word processor in which the basic editing functions execute on the client side where they operate on locally cached, or in-memory data. but where the advanced support tools such as checking the spelling and grammar execute on the server side.

In many client-server environments, the organizations shown in Fig. 2-5(d) and Fig. 2-5(e) are particularly popular. These organizations are used where the client machine is a PC or workstation, connected through a network to a distributed file system or database. Essentially, most of the application is running on the client machine, but all operations on files or database entries go to the server. For example, many banking applications run on an end-user's machine where the user prepares transactions and such. Once finished, the application contacts the database on the bank's server and uploads the transactions for further processing. Fig. 2-5(e) represents the situation where the client's local disk contains part of the data. For example, when browsing the Web, a client can gradually build a huge cache on local disk of most recent inspected Web pages.

We note that for a few years there has been a strong trend to move away from the configurations shown in Fig. 2-5(d) and Fig. 2-5(e) in those case that client software is placed at end-user machines. In these cases, most of the processing and data storage is handled at the server side. The reason for this is simple: although client machines do a lot, they are also more problematic to manage. Having more functionality on the client machine makes client-side software more prone to errors and more dependent on the client's underlying platform (i.e., operating system and resources). From a system's management perspective, having what are called fat clients is not optimal. Instead the thin clients as represented by the organizations shown in Fig. 2-5(a)-(c) are much easier, perhaps at the cost of less sophisticated user interfaces and client-perceived performance.

Note that this trend does not imply that we no longer need distributed systems. On the contrary, what we are seeing is that server-side solutions are becoming increasingly more distributed as a single server is being replaced by multiple servers running on different machines. In particular, when distinguishing only client and server machines as we have done so far, we miss the point that a server may sometimes need to act as a client, as shown in Fig. 2-6, leading to a (physically) three-tiered architecture.

In this architecture, programs that form part of the processing level reside on a separate server, but may additionally be partly distributed across the client and server machines. A typical example of where a three-tiered architecture is used is

Figure 2-6. An example of a server acting as client.

in transaction processing. As we discussed in Chap. 1, a separate process, called the transaction processing monitor, coordinates all transactions across possibly different data servers. /

Another, but very different example where we often see a three-tiered architecture is in the organization of Web sites. In this case, a Web server acts as an entry point to a site, passing requests to an application server where the actual processing takes place. This application server, in tum, interacts with a database server. For example, an application server may be responsible for running the code to inspect the available inventory of some goods as offered by an electronic bookstore. To do so, it may need to interact with a database containing the raw inventory data. We will come back to Web site organization in Chap. 12.

2.2.2 Decentralized Architectures

Multitiered client-server architectures are a direct consequence of dividing applications into a user-interface, processing components, and a data level. The different tiers correspond directly with the logical organization of applications. In many business environments, distributed processing is equivalent to organizing a client-server application as a multitiered architecture. We refer to this type of distribution as vertical distribution. The characteristic feature of vertical distribution is that it is achieved by placing *logically* different components on different machines. The term is related to the concept of vertical fragmentation as used in distributed relational databases, where it means that tables are split column-wise, and subsequently distributed across multiple machines (Oszu and Valduriez, 1999).

Again, from a system management perspective, having a vertical distribution can help: functions are logically and physically split across multiple machines, where each machine is tailored to a specific group of functions. However, vertical distribution is only one way of organizing client-server applications. In modem architectures, it is often the distribution of the clients and the servers that counts, which we refer to as horizontal distribution. In this type of distribution, a client or server may be physically split up into logically equivalent parts, but each part is operating on its own share of the complete data set, thus balancing the load. In this section we will take a look at a class of modern system architectures that support horizontal distribution, known as peer-to-peer systems.

From a high-level perspective, the processes that constitute a peer-to-peer system are all equal. This means that the functions that need to be carried out are represented by every process that constitutes the distributed system. As a consequence, much of the interaction between processes is symmetric: each process will act as a client and a server at the same time (which is also referred to as acting as a servent).

Given this symmetric behavior, peer-to-peer architectures evolve around the question how to organize the processes in an overlay network, that is, a network in which the nodes are formed by the processes and the links represent the possible communication channels (which are usually realized a§ TCP connections). In general, a process cannot communicate directly with an arbitrary other process, but is required to send messages through the available communication channels. Two types of overlay networks exist: those that are structured and those that are not. These two types are surveyed extensively in Lua et a1. (2005) along with numerous examples. Aberer et a1. (2005) provide a reference architecture that allows for a more formal comparison of the different types of peer-to-peer systems. A survey taken from the perspective of content distribution is provided by Androutsellis- Theotokis and Spinellis (2004).

Structured Peer-to-Peer Architectures

In a structured peer-to-peer architecture, the overlay network is constructed using a deterministic procedure. By far the most-used procedure is to organize the processes through a distributed hash table (DHT). In a DHT-based system, data items are assigned a random key from a large identifier space, such as a 128-bit or 160-bit identifier. Likewise, nodes in the system are also assigned a random number from the same identifier space. The crux of every DHT-based system is then to implement an efficient and deterministic scheme that uniquely maps the key of a data item to the identifier of a node based on some distance metric (Balakrishnan. 2003). Most importantly, when looking up a data item, the network address of the node responsible for that data item is returned. Effectively, this is accomplished by *routing* a request for a data item to the responsible node.

For example, in the Chord system (Stoica et al., 2003) the nodes are logically organized in a ring such that a data item with key k is mapped to the node with the smallest identifier $id \sim k$. This node is referred to as the *successor* of key k and denoted as *succ(k),* as shown in Fig. 2-7. To actually look up the data item, an application running on an arbitrary node would then call the function LOOKUP(k) which would subsequently return the network address of *succ(k).* At that point, the application can contact the node to obtain a copy of the data item.

Figure 2-7. The mapping of data items onto nodes in Chord.

We will not go into algorithms for looking up a key now, but defer that discussion until Chap. 5 where we describe details of various naming systems. Instead, let us concentrate on how nodes organize themselves into an overlay network, or, in other words, **membership management.** In the following, it is important to realize that looking up a key does not follow the logical organization of nodes in the ring from Fig. 2-7. Rather, each node will maintain shortcuts to other nodes in such a way that lookups can generally be done in *O(log (N)* number of steps, where *N* is the number of nodes participating in the overlay.

Now consider Chord again. When a node wants to join the system, it starts with generating a random identifier *id.* Note that if the identifier space is large enough, then provided the random number generator is of good quality, the probability .of generating an identifier that is already assigned to an actual node is close to zero. Then, the node can simply do a lookup on *id,* which will return the network address of *succiid).* At that point, the joining node can simply contact *succiid)* and its predecessor and insert itself in the ring. Of course, this scheme requires that each node also stores information on its predecessor. Insertion also yields that each data item whose key is now associated with node *id,* is transferred from *succiid).*

Leaving is just as simple: node *id* informs its departure to its predecessor and successor, and transfers its data items to *succ(id).*

Similar approaches are followed in other DHT-based systems. As an example, consider the **Content Addressable Network** (CAN), described in Ratnasamy et a1. (2001). CAN deploys a d-dimensional Cartesian coordinate space, which is completely partitioned among all all the nodes that participate in the system. For purpose of illustration. let us consider only the 2-dimensional case, of which an example is shown in Fig. 2-8.

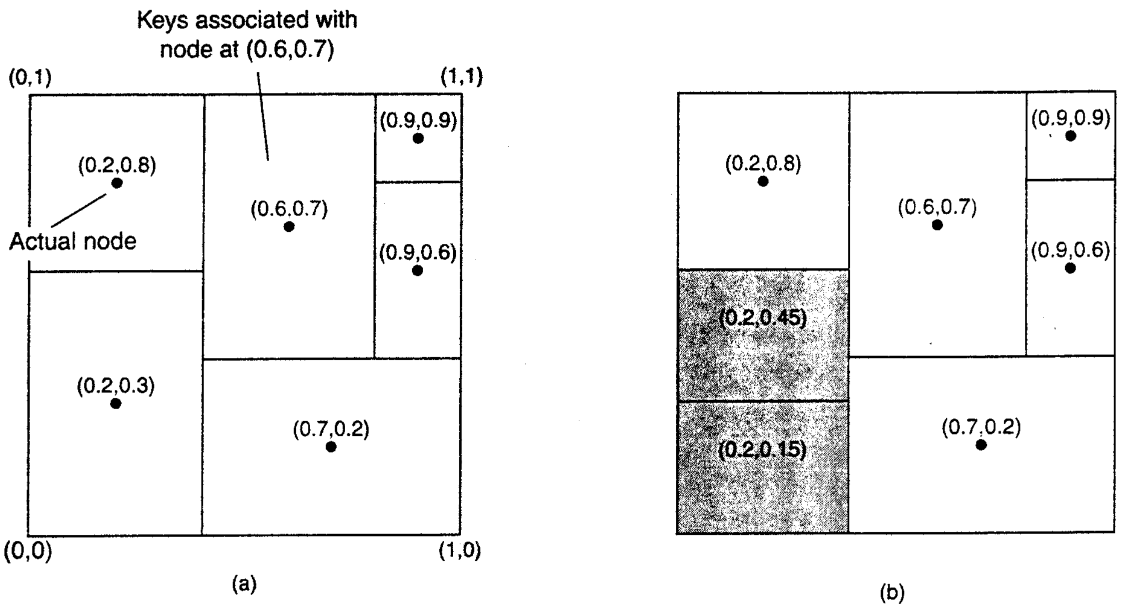


Figure 2-8. (a) The mapping of data items onto nodes in CAN. (b) Splitting a region when a node joins.

Fig.2-8(a) shows how the two-dimensional space [0, 1]x[O, 1] is divided among six nodes. Each node has an associated region. Every data item in CAN will be assigned a unique point in this space, after which it is also clear which node is responsible for that data. (ignoring data items that fall on the border of multiple regions, for which a deterministic assignment rule is used).

When a node *P* wants to join a CAN system, it picks an arbitrary point from the coordinate space and subsequently looks up the node Q in whose region that point falls. This lookup is accomplished through positioned-based routing. of which the details are deferred until later chapters. Node Q then splits its region into two halves, as shown in Fig. 2-8(b). and one half is assigned to the node *P*. Nodes keep track of their neighbors, that is, nodes responsible for adjacent region. When splitting a region, the joining node *P* can easily come to know who its new neighbors are by asking node *P*. As in Chord, the data items for which node *P* is now responsible are transferred from node Q.

Leaving is a bit more problematic in CAN. Assume that in Fig. 2-8. the node with coordinate (0.6, 0.7) leaves. Its region will be assigned to one of its neighbors, say the node at (0.9,0.9), but it is clear that simply merging it and obtaining a rectangle cannot be done. In this case, the node at (0.9,0.9) will simply take care of that region and inform the old neighbors of this fact. Obviously. this may lead to less symmetric partitioning of the coordinate space, for which reason a background process is periodically started to repartition the entire space.

Unstructured Peer-to- Peer Architectures

Unstructured peer-to-peer systems largely rely on randomized algorithms for constructing an overlay network. The main idea is that each node maintains a list of neighbors, but that this list is constructed in a more or less random way. Likewise, data items are assumed to be randomly placed on nodes. As a consequence, when a node needs to locate a specific data item, the only thing it can effectively do is flood the network with a search query (Risson and Moors, 2006). We will return to searching in unstructured overlay networks in Chap. 5, and for now concentrate on membership management.

One of the goals of many unstructured peer-to-peer systems is to construct an overlay network that resembles a random graph. The basic model is that each node maintains a list of c neighbors, where, ideally, each of these neighbors represents a randomly chosen *live* node from the current set of nodes. The list of neighbors is also referred to as a partial view. There are many ways to construct such a partial view. Jelasity et al. (2004, 2005a) have developed a framework that captures many different algorithms for overlay construction to allow for evaluations and comparison. In this framework, it is assumed that nodes regularly exchange entries from their partial view. Each entry identifies another node in the network, and has an associated age that indicates how old the reference to that node is. Two threads are used, as shown in Fig. 2-9.

The active thread takes the initiative to communicate with another node. It selects that node from its current partial view. Assuming that entries need to be *pushed* to the selected peer, it continues by constructing a buffer containing *c/2+* I entries, including an entry identifying itself. The other entries are taken from the current partial view.

W \ne noce l~ a\~Cin puU mode it ~l\\ ~a\.\1m Q 'ieSp~il~e"'i~ID fue ~e\e'\:..~\\ *peer. That peer,* in *the meantime, will* also have constructed a buffer *by* means the passive thread shown in Fig. 2-9(b), whose activities strongly resemble that of the active thread.

The crucial point is *the construction* of *a new partial view. This view, for initiating* as well as for the contacted peer, will contain exactly, c entries, part of which will come from received buffer. In essence, there are two ways to construct the new view. First, the two nodes may decide to discard the entries that they had sent to each other. Effectively, this means that they will *swap* part of their original views. The second approach is to discard as many *old* entries as possible. In general, it turns out that the two approaches are complementary [see Jelasity et al. (2005a) for the details]. It turns out that many membership management protocols for unstructured overlays fit this framework. There are a number of interesting observations to make.

First, let us assume that when a node wants to join it contacts an arbitrary other node, possibly from a list of well-known access points. This access point is just a regular member of the overlay, except that we can assume it to be highly

```
Actions by active thread (periodically repeated):

    select a peer P from the current partial view;
    if PUSH_MODE {
        mybuffer = [(MyAddress, 0)];
        permute partial view;
        move H oldest entries to the end;
        append first c/2 entries to mybuffer;
        send mybuffer to P;
    } else {
        send trigger to P;
    }
    if PULL_MODE {
        receive P's buffer;
    }
    construct a new partial view from the current one and P's buffer;
    increment the age of every entry in the new partial view;
                                        (a)
Actions by passive thread:

    receive buffer from any process Q;
    if PULL_MODE {
        mybuffer = [(MyAddress, 0)];
        permute partial view;
        move H oldest entries to the end;
        append first c/2 entries to mybuffer;
        send mybuffer to P;
    }
    construct a new partial view from the current one and P's buffer;
    increment the age of every entry in the new partial view;
                                        (b)
```

Figure 2-9. (a) The steps taken by the active thread. (b) The steps take by the passive thread.

available. In this case, it turns out that protocols that use only *push mode* or only *pull mode* can fairly easily lead to disconnected overlays. In other words, groups of nodes will become isolated and will never be able to reach every other node in the network. Clearly, this is an undesirable feature, for which reason it makes more sense to let nodes actually *exchange* entries.

Second, leaving the network turns out to be a very simple operation provided the nodes exchange partial views on a regular basis. In this case, a node can simply depart without informing any other node. What will happen is that when a node *P* selects one of its apparent neighbors, say node *Q*, and discovers that *Q* no longer responds, it simply removes the entry from its partial view to select another peer. It turns out that when constructing a new partial view, a node follows the policy to discard as many old entries as possible, departed nodes will rapidly be forgotten. In other words, entries referring to departed nodes will automatically be quickly removed from partial views.

However, there is a price to pay when this strategy is followed. To explain, consider for a node *P* the set of nodes that have an entry in their partial view that refers to *P*. Technically, this is known as the indegree of a node. The higher node *P's* indegree is, the higher the probability that some other node will decide to contact *P*. In other words, there is a danger that *P* will become a popular node, which could easily bring it into an imbalanced position regarding workload. Systematically discarding old entries turns out to promote nodes to ones having a high indegree. There are other trade-offs in addition, for which we refer to Jelasity et al. (2005a).

Topology Management of Overlay Networks

Although it would seem that structured and unstructured peer-to-peer systems form strict independent classes, this 'need actually not be case [see also Castro et al. (2005)]. One key observation is that by carefully exchanging and selecting entries from partial views, it is possible to construct and maintain specific topologies of overlay networks. This topology management is achieved by adopting a two-layered approach, as shown in Fig. 2-10.

Figure 2·10. A two-layered approach for constructing and maintaining specific overlay topologies using techniques from unstructured peer-to-peer systems.

The lowest layer constitutes an unstructured peer-to-peer system in which nodes periodically exchange entries of their partial views with the aim to maintain an accurate random graph. Accuracy in this case refers to the fact that the partial view should be filled with entries referring to randomly selected *live* nodes.

The lowest layer passes its partial view to the higher layer, where an additional selection of entries takes place. This then leads to a second list of neighbors corresponding to the desired topology. Jelasity and Babaoglu (2005) propose to use a *ranking function* by which nodes are ordered according to some criterion relative to a given node. A simple ranking function is to order a set of nodes by increasing distance from a given node *P*. In that case, node *P* will gradually build up a list of its nearest neighbors, provided the lowest layer continues to pass randomly selected nodes.

As an illustration, consider a logical grid of size $N \times N$ with a node placed on each point of the grid. Every node is required to maintain a list of c nearest neighbors, where the distance between a node at (a_1,a_2) and (b_1,b_2) is defined as d_1+d_2 with $d_i=min(N-|a_i-b_i|, |a_i-b_i|)$. If the lowest layer periodically executes the protocol as outlined in Fig. 2-9, the topology that will evolve is a torus, shown in Fig. 2-11.

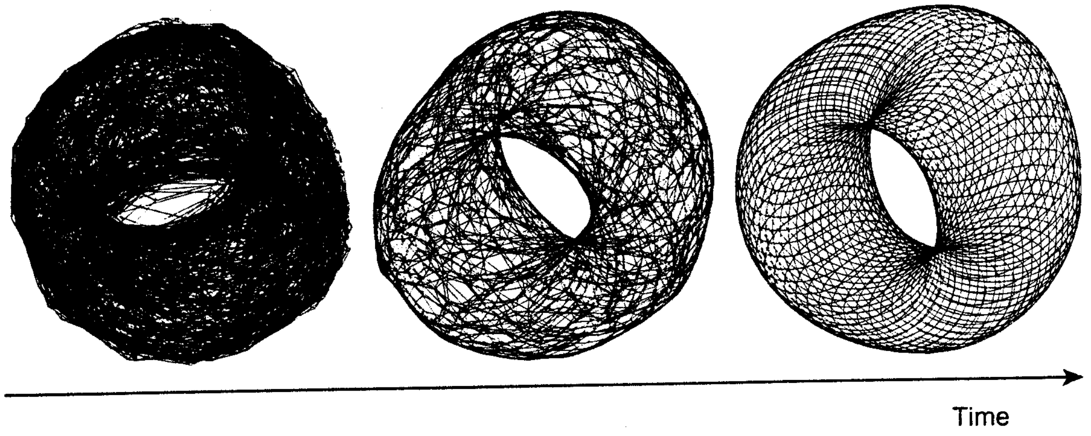


Figure 2-11. Generating a specific overlay network using a two-layered unstructured peer-to-peer system [adapted with permission from Jelasity and Babaoglu (2005)].

Of course, completely different ranking functions can be used. Notably those that are related to capturing the semantic proximity of the data items as stored at a peer node are interesting. This proximity allows for the construction of semantic overlay networks that allow for highly efficient search algorithms in unstructured peer-to-peer systems. We will return to these systems in Chap. 5 when we discuss attribute-based naming.

Superpeers

Notably in unstructured peer-to-peer systems, locating relevant data items can become problematic as the network grows. The reason for this scalability problem is simple: as there is no deterministic way of routing a lookup request to a specific data item, essentially the only technique a node can resort to is flooding the request. There are various ways in which flooding can be dammed, as we will discuss in Chap. 5, but as an alternative many peer-to-peer systems have proposed to make use of special nodes that maintain an index of data items.

There are other situations in which abandoning the symmetric nature of peer-to-peer systems is sensible. Consider a collaboration of nodes that offer resources to each other. For example, in a collaborative content delivery network (CDN), nodes may offer storage for hosting copies of Web pages allowing Web clients to access pages nearby, and thus to access them quickly. In this case a node *P* may need to seek for resources in a specific part of the network. In that case, making use of a broker that collects resource usage for a number of nodes that are in each other's proximity will allow to quickly select a node with sufficient resources.

Nodes such as those maintaining an index or acting as a broker are generally referred to as superpeers. As their name suggests, superpeers are often also organizedin a peer-to-peer network, leading to a hierarchical organization as explained in Yang and Garcia-Molina (2003). A simple example of such an organization is shown in Fig. 2-12. In this organization, every regular peeris connected as a client to a superpeer. All communication from and to a regular peer proceeds through that peer's associated superpeer.

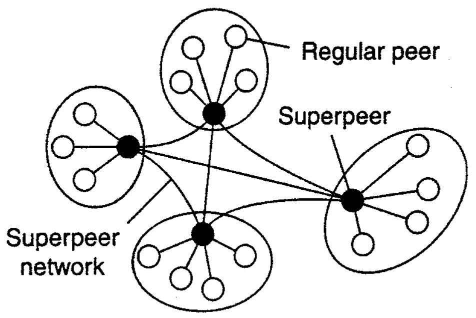


Figure 2-12. A hierarchical organization of nodes into a superpeer network.

In many cases, the client-superpeer relation is fixed: whenever a regular peer joins the network, it attaches to one of the superpeers and remains attached until it leaves the network. Obviously, it is expected that superpeers are long-lived processes with a high availability. To compensate for potential unstable behavior of a superpeer, backup schemes can be deployed, such as pairing every superpeer with another one and requiring clients to attach to both.

Having a fixed association with a superpeer may not always be the best solution. For example, in the case of file-sharing networks, it may be better for a client to attach to a superpeer that maintains an index of files that the client is generally interested in. In that case, chances are bigger that when a client is looking for a specific file, its superpeer will know where to find it. Garbacki et al. (2005) describe a relatively simple scheme in which the client-superpeer relation can change as clients discover better superpeers to associate with. In particular, a superpeer returning the result of a lookup operation is given preference over other superpeers.,

As we have seen, peer-to-peer networks offer a flexible means for nodes to join and leave the network. However, with superpeer networks a new problem is introduced, namely how to select the nodes that are eligible to become superpeer.

This problem is closely related to the leader-election problem, which we discuss in Chap. 6, when we return to electing superpeers in a peer-to-peer network.

2.2.3 Hybrid Architectures

So far, we have focused on client-server architectures and a number of peer-to-peer architectures. Many distributed systems combine architectural features, as we already came across in superpeer networks. In this section we take a look at some specific classes of distributed systems in which client-server solutions are combined with decentralized architectures.

Edge-Server Systems

An important class of distributed systems that is organized according to a hybrid architecture is formed by edge-server systems. These systems are deployed on the Internet where servers are placed "at the edge" of the network. This edge is formed by the boundary between enterprise networks and the actual Internet, for example, as provided by an Internet Service Provider (ISP). Likewise, where end users at home connect to the Internet through their ISP, the ISP can be considered as residing at the edge of the Internet. This leads to a general organization as shown in Fig. 2-13.

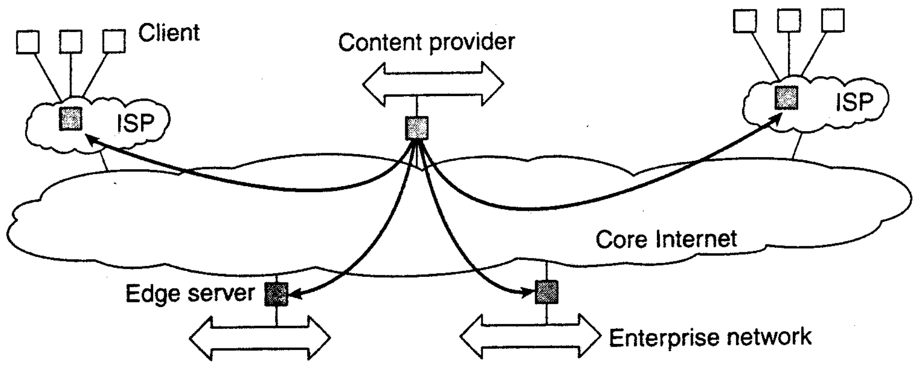


Figure 2-13. Viewing the Internet as consisting of a collection of edge servers.

End users, or clients in general, connect to the Internet by means of an edge server. The edge server's main purpose is to serve content, possibly after applying filtering and transcoding functions. More interesting is the fact that a collection of edge servers can be used to optimize content and application distribution. The basic model is that for a specific organization, one edge server acts as an origin server from which all content originates. That server can use other edge servers for replicating Web pages and such (Leff et al., 2004: Nayate et al., 2004; and Rabinovich and Spatscheck, 2002). We will return to edge-server systems in Chap. 12 when we discuss Web-based solutions.

Collaborative Distributed Systems

Hybrid structures are notably deployed in collaborative distributed systems. The main issue in many of these systems to first get started, for which often a traditional client-server scheme is deployed. Once a node has joined the system, it can use a fully decentralized scheme for collaboration.

To make matters concrete, let us first consider the BitTorrent file-sharing system (Cohen, 2003). BitTorrent is a peer-to-peer file downloading system. Its principal working is shown in Fig. 2-14 The basic idea is that when an end user is looking for a file, he downloads chunks of the file from other users until the downloaded chunks can be assembled together yielding the complete file. An important design goal was to ensure collaboration. In most file-sharing systems, a significant fraction of participants merely download files but otherwise contribute close to nothing (Adar and Huberman, 2000; Saroiu et al., 2003; and Yang et al., 2005). To this end, a file can be downloaded only when the downloading client is providing content to someone else. We will return to this "tit-for-tat" behavior shortly.

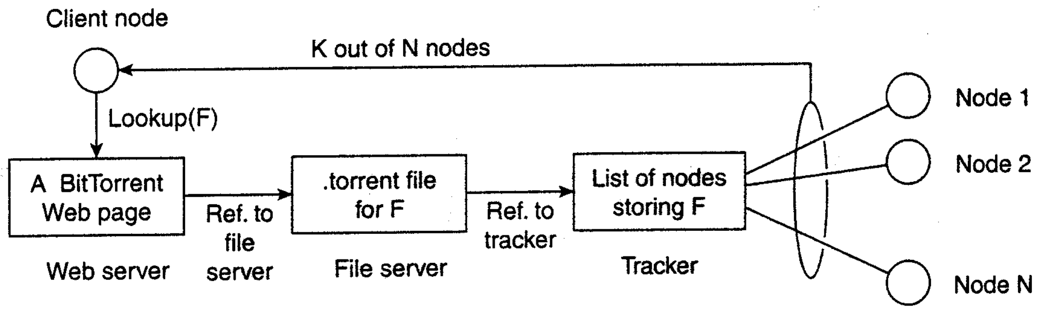


Figure 2-14. The principal working of BitTorrent [adapted with permission from Pouwelse et al. (2004)].

To download a me, a user needs to access a global directory, which is just one of a few well-known Web sites. Such a directory contains references to what are called *.torrent* files. A *.torrent* file contains the information that is needed to download a specific file. In particular, it refers to what is known as a tracker, which is a server that is keeping an accurate account of *active* nodes that have (chunks) of the requested file. An active node is one that is currently downloading another file. Obviously, there will be many different trackers, although there will generally be only a single tracker per file (or collection of files).

Once the nodes have been identified from where chunks can be downloaded, the downloading node effectively becomes active. At that point, it will be forced to help others, for example by providing chunks of the file it is downloading that others do not yet have. This enforcement comes from a very simple rule: if node *P* notices that node Q is downloading more than it is uploading, *P* can decide to decrease the rate at which it sends data *toQ*. This scheme works well provided *P* has something to download from Q. For this reason, nodes are often supplied with references to many other nodes putting them in a better position to trade data.

Clearly, BitTorrent combines centralized with decentralized solutions. As it turns out, the bottleneck of the system is, not surprisingly, formed by the trackers.

As another example, consider the Globule collaborative content distribution network (Pierre and van Steen, 2006). Globule strongly resembles the edge-server architecture mentioned above. In this case, instead of edge servers, end users (but also organizations) voluntarily provide enhanced Web servers that are capable of collaborating in the replication of Web pages. In its simplest form, each such server has the following components:

1. A component that can redirect client requests to other servers.
2. A component for analyzing access patterns.
3. A component for managing the replication of Web pages.

The server provided by Alice is the Web server that normally handles the traffic for Alice's Web site and is called the origin server for that site. It collaborates with other servers, for example, the one provided by Bob, to host the pages from Bob's site. In this sense, Globule is a decentralized distributed system. Requests for Alice's Web site are initially forwarded to her server, at which point they may be redirected to one of the other servers. Distributed redirection is also supported.

However, Globule also has a centralized component in the form of its broker. The broker is responsible for registering servers, and making these servers known to others. Servers communicate with the broker completely analogous to what one would expect in a client-server system. For reasons of availability, the broker can be replicated, but as we shall later in this book, this type of replication is widely applied in order to achieve reliable client-server computing.

2.3 ARCHITECTURES VERSUS MIDDLEW ARE

When considering the architectural issues we have discussed so far, a question that comes to mind is where middleware fits in. As we discussed in Chap. 1, middleware forms a layer between applications and distributed platforms. as shown in Fig. 1-1. An important purpose is to provide a degree of distribution transparency, that is, to a certain extent hiding the distribution of data, processing, and control from applications.

What is comonly seen in practice is that middleware systems actually follow a specific architectural sytle. For example, many middleware solutions have adopted an object-based architectural style, such as CORBA (OMG. 2004a). Others, like TIB/Rendezvous (TIBCO, 2005) provide middleware that follows the event-based architectural style. In later chapters, we will come across more examples of architectural styles.

Having middleware molded according to a specific architectural style has the benefit that designing applications may become simpler. However, an obvious drawback is that the middleware may no longer be optimal for what an application developer had in mind. For example, COREA initially offered only objects that could be invoked by remote clients. Later, it was felt that having only this form of interaction was too restrictive, so that other interaction patterns such as messaging were added. Obviously, adding new features can easily lead to bloated middleware solutions.

In addition, although middleware is meant to provide distribution transparency, it is generally felt that specific solutions should be adaptable to application requirements. One solution to this problem is to make several versions of a middleware system, where each version is tailored to a specific class of applications. An approach that is generally considered better is to make middleware systems such that they are easy to configure, adapt, and customize as needed by an application. As a result, systems are now being developed in which a stricter separation between policies and mechanisms is being made. This has led to several mechanisms by which the behavior of middleware can be modified (Sadjadi and McKinley, 2003). Let us take a look at some of the commonly followed approaches.

2.3.1 Interceptors

Conceptually, an **interceptor** is nothing but a software construct that will break the usual flow of control and allow other (application specific) code to be executed. To make interceptors generic may require a substantial implementation effort, as illustrated in Schmidt et al. (2000), and it is unclear whether in such cases generality should be preferred over restricted applicability and simplicity. Also, in many cases having only limited interception facilities will improve management of the software and the distributed system as a whole.

To make matters concrete, consider interception as supported in many object-based distributed systems. The basic idea is simple: an object *A* can call a method that belongs to an object *B*, while the latter resides on a different machine than *A*. As we explain in detail later in the book, such a remote-object invocation is carried as a three-step approach:

1. Object *A* is offered a local interface that is exactly the same as the interface offered by object *B*. *A* simply calls the method available in that interface.

2. The call by *A* is transformed into a generic object invocation, made possible through a general object-invocation interface offered by the middleware at the machine where *A* resides.

3. Finally, the generic object invocation is transformed into a message that is sent through the transport-level network interface as offered by *A's* local operating system.

This scheme is shown in Fig. 2-15.

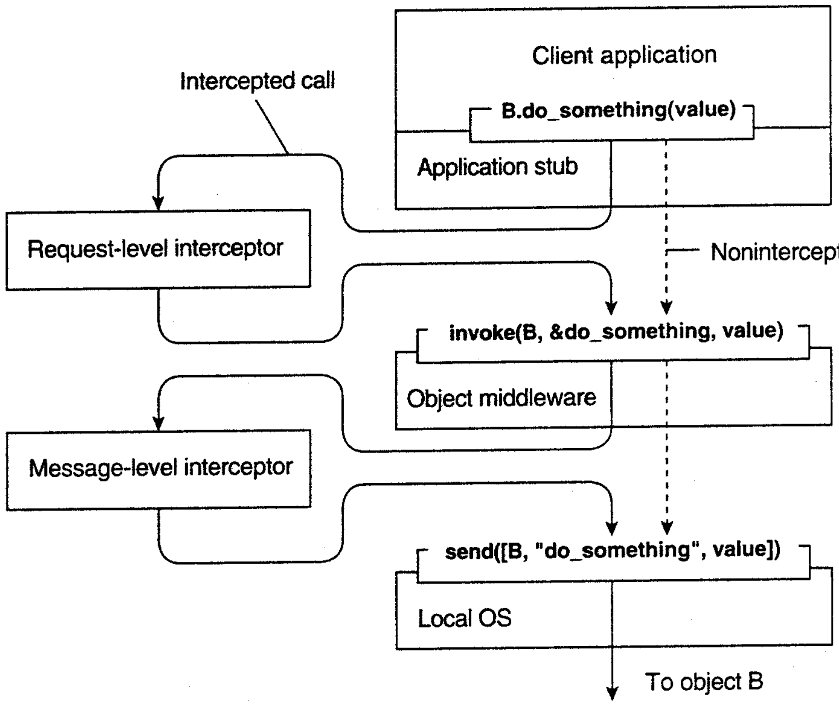


Figure 2-15. Using interceptors to handle remote-object invocations.

After the first step, the call B.do_something(value) is transformed into a generic call such as invoke(B, &do_something, value) with a reference to *B's* method and the parameters that go along with the call. Now imagine that object *B* is replicated. In that case, each replica should actually be invoked. This is a clear point where interception can help. What the request-level interceptor will do is simply call invoke(B, &do_something, value) for each of the replicas. The beauty of this an is that the object *A* need not be aware of the replication of *B,* but also the object middleware need not have special components that deal with this replicated call. Only the request-level interceptor, which may be *added* to the middleware needs to know about *B's* replication.

In the end, a call to a remote object will have to be sent over the network. In practice, this means that the messaging interface as offered by the local operating system will need to be invoked. At that level, a message-level interceptor may assist in transferring the invocation to the target object. For example, imagine that the parameter value actually corresponds to a huge array of data. In that case, it may be wise to fragment the data into smaller parts to have it assembled again at the destination. Such a fragmentation may improve performance or reliability. Again, the middleware need not be aware of this fragmentation; the lower-level interceptor will transparently handle the rest of the communication with the local operating system.

2.3.2 General Approaches to Adaptive Software

What interceptors actually offer is a means to adapt the middleware. The need for adaptation comes from the fact that the environment in which distributed applications are executed changes continuously. Changes include those resulting from mobility, a strong variance in the quality-of-service of networks, failing hardware, and battery drainage, amongst others. Rather than making applications responsible for reacting to changes, this task is placed in the middleware.

These strong influences from the environment have brought many designers of middleware to consider the construction of *adaptive software.* However, adaptive software has not been as successful as anticipated. As many researchers and developers consider it to be an important aspect of modern distributed systems, let us briefly pay some attention to it. McKinley et al. (2004) distinguish three basic techniques to come to software adaptation:

1. Separation of concerns
2. Computational reflection
3. Component-based design

Separating concerns relates to the traditional way of modularizing systems: separate the parts that implement functionality from those that take care of other things (known as *extra functionalities)* such as reliability, performance, security, etc. One can argue that developing middleware for distributed applications is largely about handling extra functionalities independent from applications. The main problem is that we cannot easily separate these extra functionalities by means of modularization. For example, simply putting security into a separate module is not going to work. Likewise, it is hard to imagine how fault tolerance can be isolated into a separate box and sold as an independent service. Separating and subsequently weaving these *cross-cutting* concerns into a (distributed) system is the major theme addressed by **aspect-oriented software development** (Filman et al., 2005). However, aspect orientation has not yet been successfully applied to developing large-scale distributed systems, and it can be expected that there is still a long way to go before it reaches that stage.

Computational reflection refers to the ability of a program to inspect itself and, if necessary, adapt its behavior (Kon et al., 2002). Reflection has been built into programming languages, including Java, and offers a powerful facility for runtime modifications. In addition, some middleware systems provide the means to apply reflective techniques. However, just as in the case of aspect orientation, reflective middleware has yet to prove itself as a powerful tool to manage the complexity of large-scale distributed systems. As mentioned by Blair et al. (2004), applying reflection to a broad domain of applications is yet to be done.

Finally, component-based design supports adaptation through composition. A system may either be configured statically at design time, or dynamically at runtime. The latter requires support for late binding, a technique that has been successfully applied in programming language environments, but also for operating systems where modules can be loaded and unloaded at will. Research is now well underway to allow automatically selection of the best implementation of a component during runtime (Yellin, 2003), but again, the process remains complex for distributed systems, especially when considering that replacement of one component requires knowning what the effect of that replacement on other components will be. In many cases, components are less independent as one may think.

2.3.3 Discussion

Software architectures for distributed systems, notably found as middleware, are bulky and complex. In large part, this bulkiness and complexity arises from the need to be general in the sense that distribution transparency needs to be provided. At the same time applications have specific extra-functional requirements that conflict with aiming at fully achieving this transparency. These conflicting requirements for generality and specialization have resulted in middleware solutions that are highly flexible. The price to pay, however, is complexity. For example, Zhang and Jacobsen (2004) report a 50% increase in the size of a particular software product in just four years since its introduction, whereas the total number of files for that product had tripled during the same period. Obviously, this is not an encouraging direction to pursue.

Considering that virtually all large software systems are nowadays required to execute in a networked environment, we can ask ourselves whether the complexity of distributed systems is simply an inherent feature of attempting to make distribution transparent. Of course, issues such as openness are equally important, but the need for flexibility has never been so prevalent as in the case of middleware.

Coyler et al. (2003) argue that what is needed is a stronger focus on (external) simplicity, a simpler way to construct middleware by components, and application independence. Whether any of the techniques mentioned above forms the solution is subject to debate. In particular, none of the proposed techniques so far have found massive adoption, nor have they been successfully applied tQ large-scale systems.

The underlying assumption is that we need *adaptive software* in the sense that the software should be allowed to change as the environment changes. However, one should question whether adapting to a changing environment is a good reason to adopt changing the software. Faulty hardware, security attacks, energy drainage, and so on, all seem to be environmental influences that can (and should) be anticipated by software.

The strongest, and certainly most valid, argument for supporting adaptive software is that many distributed systems cannot be shut down. This constraint calls for solutions to replace and upgrade components on the fly, but is not clear whether any of the solutions proposed above are the best ones to tackle this maintenance problem.

What then remains is that distributed systems should be able to react to changes in their environment by, for example, switching policies for allocating resources. All the software components to enable such an adaptation will already be in place. It is the algorithms contained in these components and which dictate the behavior that change their settings. The challenge is to let such reactive behavior take place without human intervention. This approach is seen to work better when discussing the physical organization of distributed systems when decisions are taken about where components are placed, for example. We discuss such system architectural issues next.

2.4 SELF-MANAGEMENT IN DISTRIBUTED SYSTEMS

Distributed systems-and notably their associated middleware-need to provide general solutions toward shielding undesirable features inherent to networking so that they can support as many applications as possible. On the other hand, full distribution transparency is not what most applications actually want, resulting in application-specific solutions that need to be supported as well. We have argued that, for this reason, distributed systems should be adaptive, but notably when it comes to adapting their execution behavior and not the software components they comprise.

When adaptation needs to be done automatically, we see a strong interplay between system architectures and software architectures. On the one hand, we need to organize the components of a distributed system such that monitoring and adjustments can be done, while on the other hand we need to decide where the processes are to be executed that handle the adaptation.

In this section we pay explicit attention to organizing distributed systems as high-level feedback-control systems allowing automatic adaptations to changes. This phenomenon is also known as **autonomic computing** (Kephart, 2003) or self.star systems (Babaoglu et al., 2005). The latter name indicates the variety by which automatic adaptations are being captured: self-managing, self-healing, self-configuring, self-optimizing, and so on. We resort simply to using the name self-managing systems as coverage of its many variants.

2.4.1 The Feedback Control Model

There are many different views on self-managing systems, but what most have in common (either explicitly or implicitly) is the assumption that adaptations take place by means of one or more feedback control loops. Accordingly, systems that are organized by means of such loops are referred to as feedback control systems. Feedback control has since long been applied in various engineering fields, and its mathematical foundations are gradually also finding their way in computing systems (Hellerstein et al., 2004; and Diao et al., 2005). For self-managing systems, the architectural issues are initially the most interesting. The basic idea behind this organization is quite simple, as shown in Fig. 2-16.

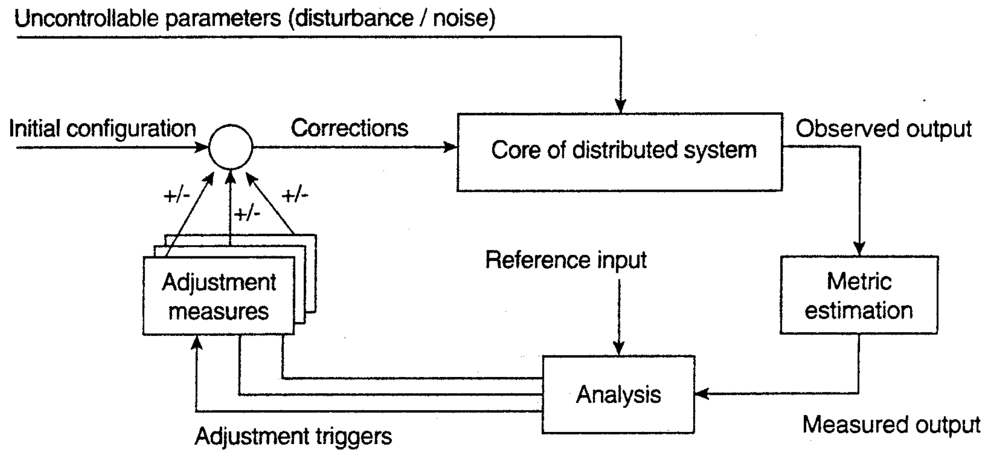


Figure 2-16. The logical organization of a feedback control system.

The core of a feedback control system is formed by the components that need to be managed. These components are assumed to be driven through controllable input parameters, but their behavior may be influenced by all kinds of *uncontrollable* input, also known as disturbance or noise input. Although disturbance will often come from the environment in which a distributed system is executing, it may well be the case that unanticipated component interaction causes unexpected behavior.

There are essentially three elements that form the feedback control loop. First, the system itself needs to be monitored, which requires that various aspects of the system need to be measured. In many cases, measuring behavior is easier said than done. For example, round-trip delays in the Internet may vary wildly, and also depend on what exactly is being measured. In such cases, accurately estimating a delay may be difficult indeed. Matters are further complicated when a node *A* needs to estimate the latency between two other completely different nodes *B* and C, without being able to intrude on either two nodes. For reasons as this, a feedback control loop generally contains a logical metric estimation component.

Another part of the feedback control loop analyzes the measurements and compares these to reference values. This feedback analysis component forms the heart of the control loop, as it will contain the algorithms that decide on possible adaptations.

The last group of components consist of various mechanisms to directly influence the behavior of the system. There can be many different mechanisms: placing replicas, changing scheduling priorities, switching services, moving data for reasons"of availability, redirecting requests to different servers, etc. The analysis component will need to be aware of these mechanisms and their (expected) effect on system behavior. Therefore, it will trigger one or several mechanisms, to subsequently later observe the effect.

An interesting observation is that the feedback control loop also fits the manual management of systems. The main difference is that the analysis component is replaced by human administrators. However, in order to properly manage any distributed system, these administrators will need decent monitoring equipment as well as decent mechanisms to control the behavior of the system. It should be clear that properly analyzing measured data and triggering the correct actions makes the development of self-managing systems so difficult.

It should be stressed that Fig. 2-16 shows the *logical* organization of a self-managing system, and as such corresponds to what we have seen when discussing software architectures. However, the *physical* organization may be very different. For example, the analysis component may be fully distributed across the system. Likewise, taking performance measurements are usually done at each machine that is part of the distributed system. Let us now take a look at a few concrete examples on how to monitor, analyze, and correct distributed systems in an automatic fashion. These examples will also illustrate this distinction between logical and physical organization.

2.4.2 Example: Systems Monitoring with Astrolabe

As our first example, we consider Astrolabe (Van Renesse et al., 2003), which is a system that can support general monitoring of very large distributed systems. In the context of self-managing systems, Astrolabe is to be positioned as a general tool for observing systems behavior. Its output can be used to feed into an analysis component for deciding on corrective actions.

Astrolabe organizes a large collection of hosts into a hierarchy of zones. The lowest-level zones consist of just a single host, which are subsequently grouped into zones of increasing size. The top-level zone covers all hosts. Every host runs an Astrolabe process, called an *agent,* that collects information on the zones in which that host is contained. The agent also communicates with other agents with the aim to spread zone information across the entire system.

Each host maintains a set of *attributes* for collecting local information. For example, a host may keep track of specific files it stores, its resource usage, and so on. Only the attributes as maintained directly by hosts, that is, at the lowest level of the hierarchy are writable. Each zone can also have a collection of attributes, but the values of these attributes are *computed* from the values of lower level zones.

Consider the following simple example shown in Fig. 2-17 with three hosts, *A, B,* and C grouped into a zone. Each machine keeps track of its IP address, CPU load, available free memory. and the number of active processes. Each of these attributes can be directly written using local information from each host. At the zone level, only aggregated information can be collected, such as the average CPU load, or the average number of active processes.

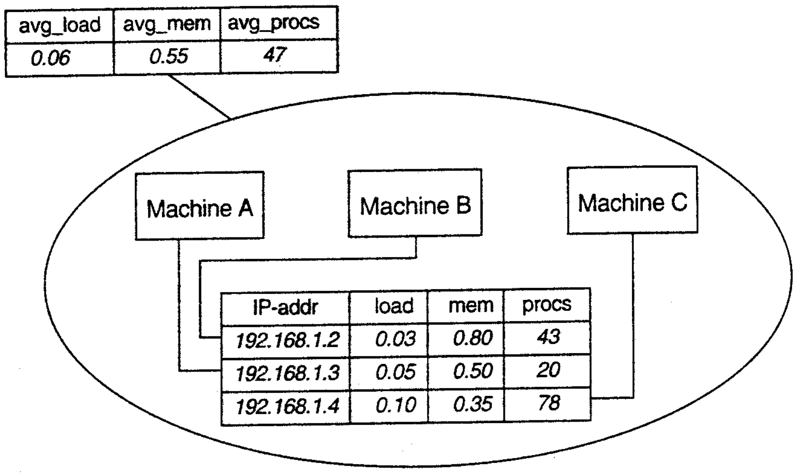


Figure 2-17. Data collection and information aggregation in Astrolabe.

Fig. 2-17 shows how the information as gathered by each machine can be viewed as a record in a database, and that these records jointly form a relation (table). This representation is done on purpose: it is the way that Astrolabe views all the collected data. However, per zone information can only be computed from the basic records as maintained by hosts.

Aggregated information is obtained by programmable aggregation functions, which are very similar to functions available in the relational database language SQL. For example, assuming that the host information from Fig. 2-17 is maintained in a local table called *hostinfo,* we could collect the average number of processes for the zone containing machines *A, B,* and C, through the simple SQL query

```
SELECT AVG(procs) AS aV9_procs FROM hostinfo
```

Combined with a few enhancements to SQL, it is not hard to imagine that more informative queries can be formulated.

Queries such as these are continuously evaluated *by* each agent running on each host. Obviously, this is possible only if zone information is propagated to all nodes that comprise Astrolabe. To this end, an agent running on a host is responsible for computing parts of the tables of its associated zones. Records for which it holds no computational responsibility are occasionally sent to it through a simple, yet effective exchange procedure known as gossiping. Gossiping protocols will be discussed in detail in Chap. 4. Likewise, an agent will pass computed results to other agents as well.

The result of this information exchange is that eventually, all agents that needed to assist in obtaining some aggregated information will see the same result (provided that no changes occur in the meantime).

2.4.3 Example: Differentiating Replication Strategies in Globule

Let us now take a look at Globule, a collaborative content distribution network (Pierre and van Steen, 2006). Globule relies on end-user servers being placed in the Internet, and that these servers collaborate to optimize performance through replication of Web pages. To this end, each origin server (i.e., the server responsible for handling updates of a specific Web site), keeps track of access patterns on a per-page basis. Access patterns are expressed as read and write operations for a page, each operation being timestamped and logged by the origin server for that page.

In its simplest form, Globule assumes that the Internet can be viewed as an edge-server system as we explained before. In particular, it assumes that requests can always be passed through an appropriate edge server, as shown in Fig. 2-18. This simple model allows an origin server to see what would have happened if it had placed a replica on a specific edge server. On the one hand, placing a replica closer to clients would improve client-perceived latency, but this will induce traffic between the origin server and that edge server in order to keep a replica consistent with the original page.

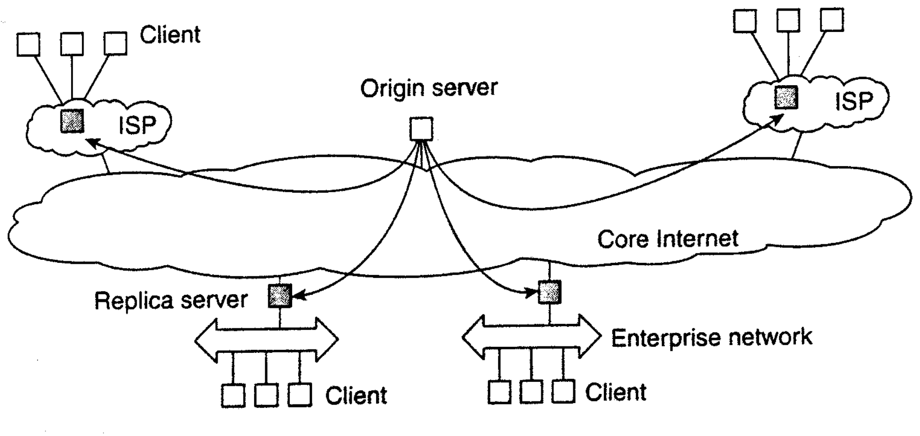


Figure 2-18. The edge-server model assumed by Globule.

When an origin server receives a request for a page, it records the IP address from where the request originated, and looks up the ISP or enterprise network associated with that request using the *WHOIS* Internet service (Deutsch et aI., 1995). The origin server then looks for the nearest existing replica server that could act as edge server for that client, and subsequently computes the latency to that server along with the maximal bandwidth. In its simplest configuration, Globule assumes that the latency between the replica server and the requesting user machine is negligible, and likewise that bandwidth between the two is plentiful.

Once enough requests for a page have been collected, the origin server performs a simple "what-if analysis." Such an analysis boils down to evaluating several replication policies, where a policy describes where a specific page is replicated to, and how that page is kept consistent. Each replication policy incurs a cost that can be expressed as a simple linear function:

$$cost=(W1\ \ xm1)+(w2xm2)+ \quad \ldots \quad +(wnxm_n)$$

where mk denotes a performance metric and Wk is the weight indicating how important that metric is. Typical performance metrics are the aggregated delays between a client and a replica server when returning copies of Web pages, the total consumed bandwidth between the origin server and a replica server for keeping a replica consistent, and the number of stale copies that are (allowed to be) returned to a client (Pierre et aI., 2002).

For example, assume that the typical delay between the time a client C issues a request and when that page is returned from the best replica server is *de* ms. Note that what the best replica server is, is determined by a replication policy. Let m_1 denote the aggregated delay over a given time period, that is, $m_1 = L\, de.$ If the origin server wants to optimize client-perceived latency, it will choose a relatively high value for w_1- As a consequence, only those policies that actually minimize m_1 will show to have relatively low costs.

In Globule, an origin server regularly evaluates a few tens of replication polices using a trace-driven simulation, for each Web page separately. From these simulations, a best policy is selected and subsequently enforced. This may imply that new replicas are installed at different edge servers, or that a different way of keeping replicas consistent is chosen. The collecting of traces, the evaluation of replication policies, and the enforcement of a selected policy is all done automatically.

There are a number of subtle issues that need to be dealt with. For one thing, it is unclear how many requests need to be collected before an evaluation of the current policy can take place. To explain, suppose that at time *T;* the origin server selects policy *p* for the next period until'Ii+I' This selection takes place based on a series of past requests that were issued between *'Ii*-1 and *'Ii.* Of course, in hindsight at time *'1i+I,* the server may come to the conclusion that it should have selected policy *p** given the actual requests that were issued between *'Ii* and *'Ii*+I. If *p** is different from *p,* then the selection of *p* at *'Ii* was wrong.

As it turns out, the percentage of wrong predictions is dependent on the length of the series of requests (called the trace length) that are used to predict and select

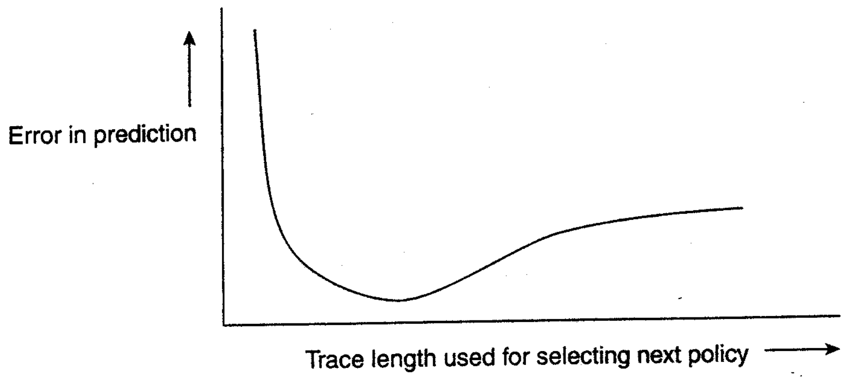


Figure 2-19. The dependency between prediction accuracy and trace length.

a next policy. This dependency is sketched in Fig. 2-19. What is seen is that the error in predicting the best policy goes up if the trace is not long enough. This is easily explained by the fact that we need enough requests to do a proper evaluation. However, the error also increases if we use too many requests. The reason for this is that a very long trace length captures so many *changes* in access patterns that predicting the best policy to follow becomes difficult, if not impossible. This phenomenon is well known and is analogous to trying to predict the weather for tomorrow by looking at what happened during the immediately preceding 100 years. A much better prediction can be made by just looking only at the recent past.

Finding the optimal trace length can be done automatically as well. We leave it as an exercise to sketch a solution to this problem.

2.4.4 Example: Automatic Component Repair Management in Jade

When maintaining clusters of computers, each running sophisticated servers, it becomes important to alleviate management problems. One approach that can be applied to servers that are built using a component-based approach, is to detect component failures and have them automatically replaced. The Jade system follows this approach (Bouchenak et al., 2005). We describe it briefly in this section.

Jade is built on the Fractal component model, a Java implementation of a framework that allows components to be added and removed at runtime (Bruneton et al., 2004). A component in Fractal can have two types of interfaces. A server interface is used to call methods that are implemented by that component. A client interface is used by a component to call other components. Components are connected to each other by binding interfaces. For example, a client interface of component C_1 can be bound to the server interface of component C_2. A primitive binding means that a call to a client interface directly leads to calling the bounded server interface. In the case of composite binding, the call may proceed through one or more other components, for example, because the client and server interface did not match and some kind of conversion is needed. Another reason may be that the connected components lie on different machines.

Jade uses the notion of a repair management domain. Such a domain consists of a number of nodes, where each node represents a server along with the components that are executed by that server. There is a separate node manager which is responsible for adding and removing nodes from the domain. The node manager may be replicated for assuring high availability.

Each node is equipped with failure detectors, which monitor the health of a node or one of its components and report any failures to the node manager. Typically, these detectors consider exceptional changes in the state of component, the usage of resources, and the actual failure of a component. Note that the latter may actually mean that a machine has crashed.

When a failure has been detected, a repair procedure is started. Such a procedure is driven by a repair policy, partly executed by the node manager. Policies are stated explicitly and are carried out depending on the detected failure. For example, suppose a node failure has been detected. In that case, the repair policy may prescribe that the following steps are to be carried out:

1. Terminate every binding between a component on a nonfaulty node, and a component on the node that just failed.
2. Request the node manager to start and add a new node to the domain.
3. Configure the new node with exactly the same components as those on the crashed node.
4. Re-establish all the bindings that were previously terminated.

In this example, the repair policy is simple and will only work when no crucial data has been lost (the crashed components are said to be stateless).

The approach followed by Jade is an example of self-management: upon the detection of a failure, a repair policy is automatically executed to bring the system as a whole into a state in which it was before the crash. Being a component-based system, this automatic repair requires specific support to allow components to be added and removed at runtime. In general, turning legacy applications into self-managing systems is not possible.

2.5 SUMMARY

Distributed systems can be organized in many different ways. We can make a distinction between software architecture and system architecture. The latter considers where the components that constitute a distributed system are placed across the various machines. The former is more concerned about the logical organization of the software: how do components interact, it what ways can they be structured, how can they be made independent, and so on.

A key idea when talking about architectures is architectural style. A style reflects the basic principle that is followed in organizing the interaction between the software components comprising a distributed system. Important styles include layering, object orientation, event orientation, and data-space orientation.

There are many different organizations of distributed systems. An important class is where machines are divided into clients and servers. A client sends a request to a server, who will then produce a result that is returned to the client. The client-server architecture reflects the traditional way of modularizing software in which a module calls the functions available in another module. By placing different components on different machines, we obtain a natural physical distribution of functions across a collection of machines.

Client-server architectures are often highly centralized. In decentralized architectures we often see an equal role played by the processes that constitute a distributed system, also known as peer-to-peer systems. In peer-to-peer systems, the processes are organized into an overlay network, which is a logical network in which every process has a local list of other peers that it can communicate with. The overlay network can be structured, in which case deterministic schemes can be deployed for routing messages between processes. In unstructured networks, the list of peers is more or less random, implying that search algorithms need to be deployed for locating data or other processes.

As an alternative, self-managing distributed systems have been developed. These systems, to an extent, merge ideas from system and software architectures. Self-managing systems can be generally organized as feedback-control loops. Such loops contain a monitoring component by the behavior of the distributed system is measured, an analysis component to see whether anything needs to be adjusted, and a collection of various instruments for changing the behavior. Feedback -control loops can be integrated into distributed systems at numerous places. Much research is still needed before a common understanding how such loops such be developed and deployedis reached.

PROBLEMS

1. If a client and a server are placed far apart, we may see network latency dominating overall performance. How can we tackle this problem?
2. What is a three-tiered client-server architecture?
3. What is the difference between a vertical distribution and a horizontal distribution?

4. Consider a chain of processes $P_1, P_2, \ldots, P_n$ implementing a multitiered client-server architecture. Process P_i is client of process P_{i+1}, and P_i will return a reply to P_{i-1} only after receiving a reply from P_{i+1}. What are the main problems with this organization when taking a look at the request-reply performance at process P_1?

5. In a structured overlay network, messages are routed according to the topology of the overlay. What is an important disadvantage of this approach?

6. Consider the CAN network from Fig. 2-8. How would you route a message from the node with coordinates (0.2,0.3) to the one with coordinates (0.9,0.6)?

7. Considering that a node in CAN knows the coordinates of its immediate neighbors, a reasonable routing policy would be to forward a message to the closest node toward the destination. How good is this policy?

8. Consider an unstructured overlay network in which each node randomly chooses c neighbors. If P and Q are both neighbors of R, what is the probability that they are also neighbors of each other?

9. Consider again an unstructured overlay network in which every node randomly chooses c neighbors. To search for a file, a node floods a request to its neighbors and requests those to flood the request once more. How many nodes will be reached?

10. Not every node in a peer-to-peer network should become superpeer. What are reasonable requirements that a superpeer should meet?

11. Consider a BitTorrent system in which each node has an outgoing link with a bandwidth capacity B_{out} and an incoming link with bandwidth capacity B_{in}. Some of these nodes (called seeds) voluntarily offer files to be downloaded by others. What is the maximum download capacity of a BitTorrent client if we assume that it can contact at most one seed at a time?

12. Give a compelling (technical) argument why the tit-for-tat policy as used in BitTorrent is far from optimal for file sharing in the Internet..

13. We gave two examples of using interceptors in adaptive middleware. What other examples come to mind?

14. To what extent are interceptors dependent on the middle ware where they are deployed?

15. Modem cars are stuffed with electronic devices. Give some examples of feedback control systems in cars.

16. Give an example of a self-managing system in which the analysis component is completely distributed or even hidden.

17. Sketch a solution to automatically determine the best trace length for predicting replication policies in Globule.

18. (Lab assignment) Using existing software, design and implement a BitTorrent-based system for distributing files to many clients from a single, powerful server. Matters are simplified by using a standard Web server that can operate as tracker.

3

PROCESSES

In this chapter, we take a closer look at how the different types of processes playa crucial role in distributed systems. The concept of a process originates from the field of operating systems where it is generally defined as a program in execution. From an operating-system perspective, the management and scheduling of processes are perhaps the most important issues to deal with. However, when it comes to distributed systems, other issues tum out to be equally or more important.

For example, to efficiently organize client-server systems, it is often convenient to make use of multithreading techniques. As we discuss in the first section, a main contribution of threads in distributed systems is that they allow clients and servers to be constructed such that communication and local processing can overlap, resulting in a high level of performance.

In recent years, the concept of virtualization has gained popularity. Virtualization allows an application, and possibly also its complete environment including the operating system, to run concurrently with other applications, but highly independent of the underlying hardware and platforms, leading to a high degree of portability. Moreover, virtualization helps in isolating failures caused by errors or security problems. It is an important concept for distributed systems, and we pay attention to it in a separate section.

As we argued in Chap. 2, client-server organizations are important in distributed systems. In this chapter, we take a closer look at typical organizations of both clients and servers. We also pay attention to general design issues for servers.

An important issue, especially in wide-area distributed systems, is moving processes between different machines. Process migration or more specifically, code migration, can help in achieving scalability, but can also help to dynamically configure clients and servers. What is actually meant by code migration and what its implications are is also discussed in this chapter.

3.1 THREADS

Although processes form a building block in distributed systems, practice indicates that the granularity of processes as provided by the operating systems on which distributed systems are built is not sufficient. Instead, it turns out that having a finer granularity in the form of multiple threads of control per process makes it much easier to build distributed applications and to attain better performance. In this section, we take a closer look at the role of threads in distributed systems and explain why they are so important. More on threads and how they can be used to build applications can be found in Lewis and Berg (998) and Stevens (1999).

3.1.1 Introduction to Threads

To understand the role of threads in distributed systems, it is important to understand what a process is, and how processes and threads relate. To execute a program, an operating system creates a number of virtual processors, each one for running a different program. To keep track of these virtual processors, the operating system has a process **table,** containing entries to store CPU register values, memory maps, open files, accounting information. privileges, etc. A process is often defined as a program in execution, that is, a program that is currently being executed on one of the operating system's virtual processors. An important issue is that the operating system takes great care to ensure that independent processes cannot maliciously or inadvertently affect the correctness of each other's behavior. In other words, the fact that multiple processes may be concurrently sharing the same CPU and other hardware resources is made transparent. Usually, the operating system requires hardware support to enforce this separation.

This concurrency transparency comes at a relatively high price. For example, each time a process is created, the operating system must create a complete independent address space. Allocation can mean initializing memory segments by, for example, zeroing a data segment, copying the associated program into a text segment, and setting up a stack for temporary data. Likewise, switching the CPU between two processes may be relatively expensive as well. Apart from saving the CPU context (which consists of register values, program counter, stack pointer, etc.), the operating system will also have to modify registers of the memory management unit (MMU) and invalidate address translation caches such as in the translation lookaside buffer (TLB). In addition, if the operating system supports more processes than it can simultaneously hold in main memory, it may have to swap processes between main memory and disk before the actual switch can take place.

Like a process, a thread executes its own piece of code, independently from other threads. However, in contrast to processes, no attempt is made to achieve a high degree of concurrency transparency if this would result in performance degradation. Therefore, a thread system generally maintains only the minimum information to allow a CPU to be shared by several threads. In particular, a thread context often consists of nothing more than the CPU context, along with some other information for thread management. For example, a thread system may keep track of the fact that a thread is currently blocked on a mutex variable, so as not to select it for execution. Information that is not strictly necessary to manage multiple threads is generally ignored. For this reason, protecting data against inappropriate access by threads within a single process is left entirely to application developers.

There are two important implications of this approach. First of all, the performance of a multithreaded application need hardly ever be worse than that of its single-threaded counterpart. In fact, in many cases, multithreading leads to a performance gain. Second, because threads are not automatically protected against each other the way processes are, development of multithreaded applications requires additional intellectual effort. Proper design and keeping things simple, as usual, help a lot. Unfortunately, current practice does not demonstrate that this principle is equally well understood.

Thread Usage in Nondistributed Systems

Before discussing the role of threads in distributed systems, let us first consider their usage in traditional, nondistributed systems. There are several benefits to multithreaded processes that have increased the popularity of using thread systems.

i'ne most 1:m-poron\\)e'i\'tl\\ \.~'m.'t~'i.\)\\\.~ \~\ \\\.~\ \.~ ~ ~\.~;¥,t~-t.N~d ~t()c-ess, ~1l.~~~'l:~~a. l;1lQ.c.kiu.~&'!&temcall is executed. tile Qrocess as a wriore *is* MocKea'. 10 Illustrate, corrsrirer Jff <1flfllic«ti<Jt7 s~k cZS cZ s~e.2dshc>e!*prOgE.wlJ,* , a,mj *asscattc tkat« «sercootioUOllS)Y.:md lZ;!cEacJ)ve)y w..avts JD!'h.ange* values, An important property of a spreadsheet program is that It maintains the runcnonai dependencies between different cells, often from different spreadsheets. Therefore, whenever a cell is modified, all dependent cells are automatically updated. When a user changes the value in a single cell, such a modification can trigger a large series of computations. If there is only a single thread of control, computation cannot proceed while the program is waiting for input. Likewise, it is not easy to provide input while dependencies are being calculated. The easy solution is to have at least two threads of control: one for handling interaction with the user and one for updating the spreadsheet. In the mean time, a third thread could be used for backing up the spreadsheet to disk while the other two are doing their work.

Another advantage of multithreading is that it becomes possible to exploit parallelism when executing the program on a multiprocessor system. In that case, each thread is assigned to a different CPU while shared data are stored in shared main memory. When properly designed, such parallelism can be transparent: the process will run equally well on a uniprocessor system, albeit slower. Multithreading for parallelism is becoming increasingly important with the availability of relatively cheap multiprocessor workstations. Such computer systems are typically used for running servers in client-server applications.

Multithreading is also useful in the context of large applications. Such applications are often developed as a collection of cooperating programs, each to be executed by a separate process. This approach is typical for a UNIX environment. Cooperation between programs is implemented by means of interprocess communication (IPC) mechanisms. For UNIX systems, these mechanisms typically include (named) pipes, message queues, and shared memory segments [see also Stevens and Rago (2005)]. The major drawback of all IPC mechanisms is that communication often requires extensive context switching, shown at three different points in Fig. 3-1.

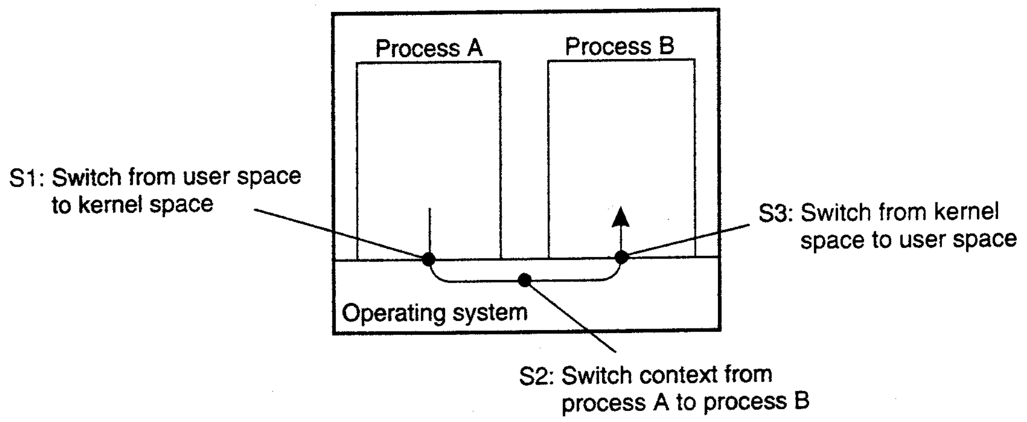

Process A
Process B
S1: Switch from user space to kernel space
S3: Switch from kernel space to user space
Operating system
S2: Switch context from process A to process B

Figure 3-1. Context switching as the result of IPC.

Because IPC requires kernel intervention, a process will generally first have to switch from user mode to kernel mode, shown as S 1 in Fig. 3-1. This requires changing the memory map in the MMU, as well as flushing the TLB. Within the kernel, a process context switch takes place (52 in the figure), after which the other party can be activated by switching from kernel mode to user mode again (53 in Fig. 3-1). The latter switch again requires changing the MMU map and flushing the TLB.

Instead of using processes, an application can also be constructed such that different parts are executed by separate threads. Communication between those parts is entirely dealt with by using shared data. Thread switching can sometimes be done entirely in user space, although in other implementations, the kernel is aware of threads and schedules them. The effect can be a dramatic improvement in performance.

Finally, there is also a pure software engineering reason to use threads: many applications are simply easier to structure as a collection of cooperating threads. Think of applications that need to perform several (more or less independent) tasks. For example, in the case of a word processor, separate threads can be used for handling user input, spelling and grammar checking, document layout, index generation, etc.

Thread Implementation

Threads are often provided in the form of a thread package. Such a package contains operations to create and destroy threads as well as operations on synchronization variables such as mutexes and condition variables. There are basically two approaches to implement a thread package. The first approach is to construct a thread library that is executed entirely in user mode. The second approach is to have the kernel be aware of threads and schedule them.

A user-level thread library has a number of advantages. First, it is cheap to create and destroy threads. Because all thread administration is kept in the user's address space, the price of creating a thread is primarily determined by the cost for allocating memory to set up a thread stack. Analogously, destroying a thread mainly involves freeing memory for the stack, which is no longer used. Both operations are cheap.

A second advantage of user-level threads is that switching thread context can often be done in just a few instructions. Basically, only the values of the CPU registers need to be stored and subsequently reloaded with the previously stored values of the thread to which it is being switched. There is no need to change memory maps, flush the TLB, do CPU accounting, and so on. Switching thread context is done when two threads need to synchronize, for example, when entering a section of shared data.

However, a major drawback of user-level threads is that invocation of a blocking system call will immediately block the entire process to which the thread belongs, and thus also all the other threads in that process. As we explained, threads are particularly useful to structure large applications into parts that could be logically executed at the same time. In that case, blocking on I/O should not prevent other parts to be executed in the meantime. For such applications, user-level threads are of no help.

These problems can be mostly circumvented by implementing threads in the operating system's kernel. Unfortunately, there is a high price to pay: every thread operation (creation, deletion, synchronization, etc.), will have to be carried out by the kernel. requiring a system call. Switching thread contexts may now become as expensive as switching process contexts. As a result, most of the performance benefits of using threads instead of processes then disappears.

A solution lies in a hybrid form of user-level and kernel-level threads, generally referred to as **lightweight** processes (LWP). An LWP runs in the context of a single (heavy-weight) process, and there can be several LWPs per process. In addition to having LWPs, a system also offers a user-level thread package. offering applications the usual operations for creating and destroying threads. In addition. the package provides facilities for thread synchronization. such as mutexes and condition variables. The important issue is that the thread package is implemented entirely in user space. In other words. all operations on threads are carried out without intervention of the kernel.

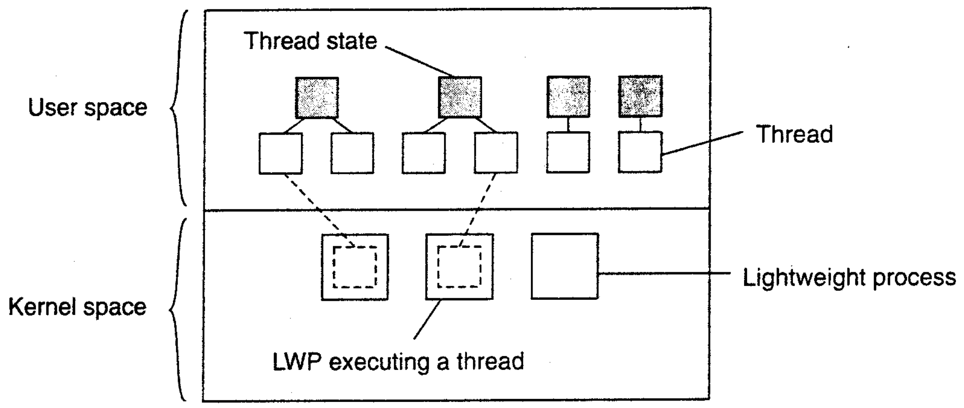


Figure 3-2. Combining kernel-level lightweight processes and user-level threads.

The thread package can be shared by multiple LWPs, as shown in Fig. 3-2. This means that each LWP can be running its own (user-level) thread. Multithreaded applications are constructed by creating threads, and subsequently assigning each thread to an LWP. Assigning a thread to an LWP is normally implicit and hidden from the programmer.

The combination of (user-level) threads and L\VPs works as follows. The thread package has a single routine to schedule the next thread. When creating an LWP (which is done by means of a system call), the LWP is given its own stack, and is instructed to execute the scheduling routine in search of a thread to execute. If there are several LWPs, then each of them executes the scheduler. The thread table, which is used to keep track of the current set of threads, is thus shared by the LWPs. Protecting this table to guarantee mutually exclusive access is done by means of mutexes that are implemented entirely in user space. In other words, synchronization between LWPs does not require any kernel support.

When an LWP finds a runnable thread, it switches context to that thread. Meanwhile, other LWPs may be looking for other runnable threads as well. If a thread needs to block on a mutex or condition variable, it does the necessary administration and eventually calls the scheduling routine. 'When another runnable thread has been found, a context switch is made to that thread. The beauty of all this is that the LWP executing the thread need not be informed: the context switch is implemented completely in user space and appears to the LWP as normal program code.

Now let us see what happens when a thread does a blocking system call. In that case, execution changes from user mode to kernel mode. but still continues in the context of the current LWP. At the point where the current LWP can no longer continue, the operating system may decide to switch context to another LWP, which also implies that a context switch is made back to user mode. The selected LWP will simply continue where it had previously left off.

There are several advantages to using LWPs in combination with a user-level thread package. First, creating, destroying, and synchronizing threads is relatively cheap and involves no kernel intervention at all. Second, provided that a process has enough LWPs, a blocking system call will not suspend the entire process. Third, there is no need for an application to know about the LWPs. All it sees are user-level threads. Fourth, LWPs can be easily used in multiprocessing environments, by executing different LWPs on different CPUs. This multiprocessing can be hidden entirely from the application. The only drawback of lightweight processes in combination with user-level threads is that we still need to create and destroy LWPs, which is just as expensive as with kernel-level threads. However, creating and destroying LWPs needs to be done only occasionally, and is often fully controlled by the operating system.

An alternative, but similar approach to lightweight processes, is to make use of scheduler activations (Anderson et al., 1991). The most essential difference between scheduler activations and LWPs is that when a thread blocks on a system call, the kernel does an *upcall* to the thread package, effectively calling the scheduler routine to select the next runnable thread. The same procedure is repeated when a thread is unblocked. The advantage of this approach is that it saves management of LWPs by the kernel. However, the use of upcalls is considered less elegant, as it violates the structure of layered systems, in which calls only to the next lower-level layer are permitted.

3.1.2 Threads in Distributed Systems

An important property of threads is that they can provide a convenient means of allowing blocking system calls without blocking the entire process in which the thread is running. This property makes threads particularly attractive to use in distributed systems as it makes it much easier to express communication in the form of maintaining multiple logical connections at the same time. We illustrate this point by taking a closer look at multithreaded clients and servers, respectively.

Multithreaded Clients

To establish a high degree of distribution transparency, distributed systems that operate in wide-area networks may need to conceal long interprocess message propagation times. The round-trip delay in a wide-area network can easily be in the order of hundreds of milliseconds. or sometimes even seconds.

The usual way to hide communication latencies is to initiate communication and immediately proceed with something else. A typical example where this happens is in Web browsers. In many cases, a Web document consists of an HTML file containing plain text along with a collection of images, icons, etc. To fetch each element of a Web document, the browser has to set up a TCPIIP connection, read the incoming data, and pass it to a display component. Setting up a connection as well as reading incoming data are inherently blocking operations. When dealing with long-haul communication, we also have the disadvantage that the time for each operation to complete may be relatively long.

A Web browser often starts with fetching the HTML page and subsequently displays it. To hide communication latencies as much as possible, some browsers start displaying data while it is still coming in. While the text is made available to the user, including the facilities for scrolling and such, the browser continues with fetching other files that make up the page, such as the images. The latter are displayed as they are brought in. The user need thus not wait until all the components of the entire page are fetched before the page is made available.

In effect, it is seen that the Web browser is doing a number of tasks simultaneously. As it turns out, developing the browser as a multithreaded client simplifies matters considerably. As soon as the main HTML file has been fetched, separate threads can be activated to take care of fetching the other parts. Each thread sets up a separate connection to the server and pulls in the data. Setting up a connection and reading data from the server can be programmed using the standard (blocking) system calls, assuming that a blocking call does not suspend the entire process. As is also illustrated in Stevens (1998), the code for each thread is the same and, above all, simple. Meanwhile, the user notices only delays in the display of images and such, but can otherwise browse through the document.

There is another important benefit to using multithreaded Web browsers in which several connections can be opened simultaneously. In the previous example, several connections were set up to the same server. If that server is heavily loaded, or just plain slow, no real performance improvements will be noticed compared to pulling in the files that make up the page strictly one after the other.

However, in many cases, Web servers have been replicated across multiple machines, where each server provides exactly the same set of Web documents. The replicated servers are located at the same site, and are known under the same name. When a request for a Web page comes in, the request is forwarded to one of the servers, often using a round-robin strategy or some other load-balancing technique (Katz et al., 1994). When using a multithreaded client, connections may be set up to different replicas, allowing data to be transferred in parallel, effectively establishing that the entire Web document is fully displayed in a much shorter time than with a nonreplicated server. This approach is possible only if the client can handle truly parallel streams of incoming data. Threads are ideal for this purpose.

~ultithreaded Servers

Although there are important benefits to multithreaded clients, as we have seen, the main use of multithreading in distributed systems is found at the server side. Practice shows that multithreading not only simplifies server code considerably, but also makes it much easier to develop servers that exploit parallelism to attain high performance, even on uniprocessor systems. However, now that multiprocessor computers are widely available as general-purpose workstations, multithreading for parallelism is even more useful.

To understand the benefits of threads for writing server code, consider the organization of a file server that occasionally has to block waiting for the disk. The file server normally waits for an incoming request for a file operation, subsequently carries out the request, and then sends back the reply. One possible, and particularly popular organization is shown in Fig. 3-3. Here one thread, the **dispatcher,** reads incoming requests for a file operation. The requests are sent by clients to a well-known end point for this server. After examining the request, the server chooses an idle (i.e., blocked) **worker thread** and hands it the request.

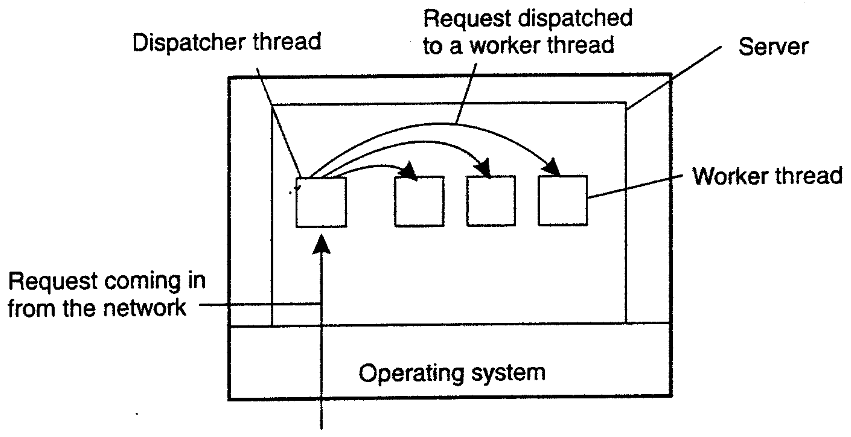

Dispatcher thread
Request dispatched to a worker thread
Server
Worker thread
Request coming in from the network
Operating system

Figure 3-3. A multithreaded server organized in a dispatcher/worker model.

The worker proceeds by performing a blocking read on the *local* file system, which may cause the thread to be suspended until the data are fetched from disk. If the thread is suspended, another thread is selected to be executed. For example, the dispatcher may be selected to acquire more work. Alternatively, another worker thread can be selected that is now ready to run.

Now consider how the file server might have been written in the absence of threads. One possibility is to have it operate as a single thread. The main loop of the file server gets a request, examines it, and carries it out to completion before getting the next one. While waiting for the disk, the server is idle and does not process any other requests. Consequently, requests from other clients cannot be handled. In addition, if the file server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the file server is waiting for the disk. The net result is that many fewer requests/sec can be processed. Thus threads gain considerable performance, but each thread is programmed sequentially, in the usual way.

So far we have seen two possible designs: a multithreaded file server and a single-threaded file server. Suppose that threads are not available but the system designers find the performance loss due to single threading unacceptable. A third possibility is to run the server as a big finite-state machine. When a request comes in, the one and only thread examines it. If it can be satisfied from the cache, fine, but if not, a message must be sent to the disk.

However, instead of blocking, it records the state of the current request in a table and then goes and gets the next message. The next message may either be a request for new work or a reply from the disk about a previous operation. If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table and the reply processed and subsequently sent to the client. In this scheme, the server will have to make use of nonblocking calls to send and receive.

In this design, the "sequential process" model that we had in the first two cases is lost. The state of the computation must be explicitly saved and restored in the table for every message sent and received. In effect, we are simulating threads and their stacks the hard way. The process is being operated as a finite-state machine that gets an event and then reacts to it, depending on what is in it.

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

Figure 3-4. Three ways to construct a server.

It should now be clear what threads have to offer. They make it possible to retain the idea of sequential processes that make blocking system calls (e.g., an RPC to talk to the disk) and still achieve parallelism. Blocking system calls make programming easier and parallelism improves performance. The single-threaded server retains the ease and simplicity of blocking system calls, but gives up some amount of performance. The finite-state machine approach achieves high performance through parallelism, but uses nonblocking calls, thus is hard to program. These models are summarized in Fig. 3-4.

3.2 VIRTUALIZATION

Threads and processes can be seen as a way to do more things at the same time. In effect, they allow us build (pieces of) programs that appear to be executed simultaneously. On a single-processor computer, this simultaneous execution is, of course, an illusion. As there is only a single CPU, only an instruction from a single thread or process will be executed at a time. By rapidly switching between threads and processes, the illusion of parallelism is created.

This separation between having a single CPU and being able to pretend there are more can be extended to other resources as well, leading to what is known as resource virtualization. This virtualization has been applied for many decades, but has received renewed interest as (distributed) computer systems have become more commonplace and complex, leading to the situation that application software is mostly always outliving its underlying systems software and hardware. In this section, we pay some attention to the role of virtualization and discuss how it can be realized.

3.2.1 The Role of Virtualization in Distributed Systems

In practice, every (distributed) computer system offers a programming interface to higher level software, as shown in Fig. 3-5(a). There are many different types of interfaces, ranging from the basic instruction set as offered by a CPU to the vast collection of application programming interfaces that are shipped with many current middleware systems. In its essence, virtualization deals with extending or replacing an existing interface so as to mimic the behavior of another system, as shown in Fig.3-5(b). We will come to discuss technical details on virtualization shortly, but let us first concentrate on why virtualization is important for distributed systems.

One of the most important reasons for introducing virtualization in the 1970s, was to allow legacy software to run on expensive mainframe hardware. The software not only included various applications, but in fact also the operating systems they were developed for. This approach toward supporting legacy software has been successfully applied on the IBM 370 mainframes (and their successors) that offered a virtual machine to which different operating systems had been ported.

As hardware became cheaper, computers became more powerful, and the number of different operating system flavors was reducing, virtualization became less of an issue. However, matters have changed again since the late 1990s for several reasons, which we will now discuss.

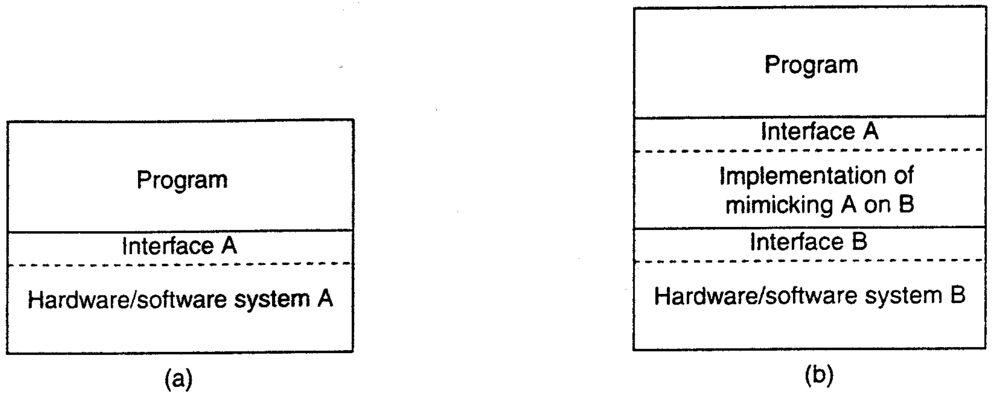

Program
Interface A
Hardware/software system A
(a)
Program
Interface A
Implementation of mimicking A on B
Interface B
Hardware/software system B
(b)

Figure 3-5. (a) General organization between a program, interface, and system. (b) General organization of virtualizing system A on top of system B.

First, while hardware and low-level systems software change reasonably fast, software at higher levels of abstraction (e.g., middleware and applications), are much more stable. In other words, we are facing the situation that legacy software cannot be maintained in the same pace as the platforms it relies on. Virtualization can help here by porting the legacy interfaces to the new platforms and thus immediately opening up the latter for large classes of existing programs.

Equally important is the fact that networking has become completely pervasive. It is hard to imagine that a modern computer is not connected to a network. In practice, this connectivity requires that system administrators maintain a large and heterogeneous collection of server computers, each one running very different applications, which can be accessed by clients. At the same time the various resources should be easily accessible to these applications. Virtualization can help a lot: the diversity of platforms and machines can be reduced by essentially letting each application run on its own virtual machine, possibly including the related libraries *and* operating system, which, in turn, run on a common platform.

This last type of virtualization provides a high degree of portability and flexibility. For example, in order to realize content delivery networks that can easily support replication of dynamic content, Awadallah and Rosenblum (2002) argue that management becomes much easier if edge servers would support virtualization, allowing a complete site, including its environment to be dynamically copied. As we will discuss later, it is primarily such portability arguments that make virtualization an important mechanism for distributed systems.

3.2.2 Architectures of Virtual Machines

There are many different ways in which virtualization can be realized in practice. An overview of these various approaches is described by Smith and Nair (2005). To understand the differences in virtualization, it is important to realize that computer systems generally offer four different types of interfaces, at four different levels:

1. An interface between the hardware and software, consisting of machine instructions that can be invoked by any program.
2. An interface between the hardware and software, consisting of machine instructions that can be invoked only by privileged programs, such as an operating system.
3. An interface consisting of system calls as offered by an operating system.
4. An interface consisting of library calls, generally forming what is known as an application programming interface (API). In many cases, the aforementioned system calls are hidden by an API.

These different types are shown in Fig. 3-6. The essence of virtualization is to mimic the behavior of these interfaces.

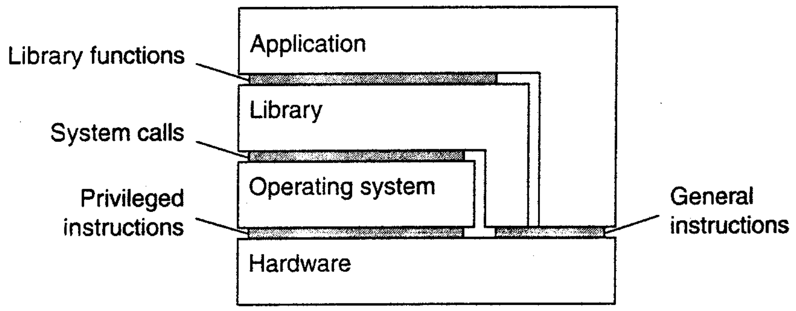


Figure 3-6. Various interfaces offered by computer systems.

Virtualization can take place in two different ways. First, we can build a runtime system that essentially provides an abstract instruction set that is to be used for executing applications. Instructions can be interpreted (as is the case for the Java runtime environment), but could also be emulated as is done for running Windows applications on UNIX platforms. Note that in the latter case, the emulator will also have to mimic the behavior of system calls, which has proven to be generally far from trivial. This type of virtualization leads to what Smith and Nair (2005) call a process virtual machine, stressing that virtualization is done essentially only for a single process.

An alternative approach toward virtualization is to provide a system that is essentially implemented as a layer completely shielding the original hardware, but offering the complete instruction set of that same (or other hardware) as an interface. Crucial is the fact that this interface can be offered *simultaneously* to different programs. As a result, it is now possible to have multiple, and different operating systems run independently and concurrently on the same platform. The layer is generally referred to as a **virtual machine monitor** (VMM). Typical examples of this approach are VMware (Sugerman et al., 2001) and Xen (Barham et at, 2003). These two different approaches are shown in Fig. 3-7.

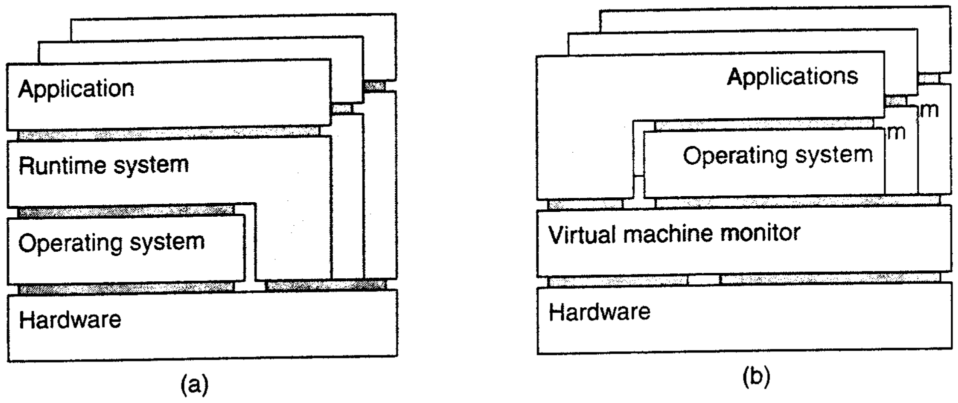


Figure 3-7. (a) A process virtual machine, with multiple instances of (application, runtime) combinations. (b) A virtual machine monitor. with multiple instances of (applications, operating system) combinations.

As argued by Rosenblum and Garfinkel (2005), VMMs will become increasingly important in the context of reliability and security for (distributed) systems. As they allow for the isolation of a complete application and its environment, a failure caused by an error or security attack need no longer affect a complete machine. In addition, as we also mentioned before, portability is greatly improved as VMMs provide a further decoupling between hardware and software, allowing a complete environment to be moved from one machine to another.

3.3 CLIENTS

In the previous chapters we discussed the client-server modeL the roles of clients and servers, and the ways they interact. Let us now take a closer look at the anatomy of clients and servers, respectively. We start in this section with a discussion of clients. Servers are discussed in the next section.

3.3.1 Networked User Interfaces

A major task of client machines is to provide the means for users to interact with remote servers. There are roughly two ways in which this interaction can be supported. First, for each remote service the client machine will have a separate counterpart that can contact the service over the network. A typical example is an agenda running on a user's PDA that needs to synchronize with a remote, possibly shared agenda. In this case, an application-level protocol will handle the synchronization, as shown in Fig. 3-8(a).

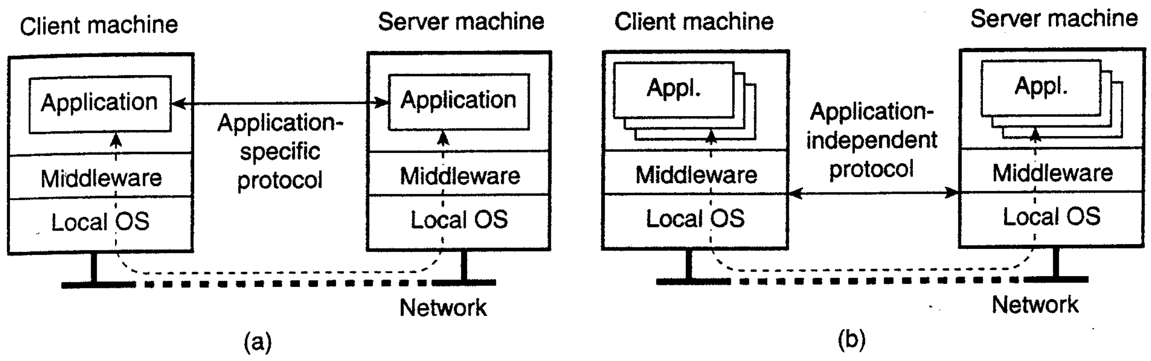


Figure 3-8. (a) A networked application with its own protocol.. (b) A general solution to allow access to remote applications.

A second solution is to provide direct access to remote services by only offering a convenient user interface. Effectively, this means that the client machine is used only as a terminal with no need for local storage, leading to an application-neutral solution as shown in Fig. 3-8(b). In the case of networked user interfaces, everything is processed and stored at the server. This thin-client approach is receiving more attention as Internet connectivity increases, and hand-held devices are becoming more sophisticated. As we argued in the previous chapter, thin-client solutions are also popular as they ease the task of system management. Let us take a look at how networked user interfaces can be supported.

Example: The X Window System

Perhaps one of the oldest and still widely-used networked user interfaces is the X Window system. The X Window System, generally referred to simply as X, is used to control bit-mapped terminals, which include a monitor, keyboard, and a pointing device such as a mouse. In a sense, X can be viewed as that part of an operating system that controls the terminal. The heart of the system is formed by what we shall call the X kernel.. It contains all the terminal-specific device drivers, and as such, is generally highly hardware dependent.

The X kernel offers a relatively low-level interface for controlling the screen, but also for capturing events from the keyboard and mouse. This interface is made available to applications as a library called *Xlib.* This general organization is shown in Fig. 3-9.

The interesting aspect of X is that the X kernel and the X applications need not necessarily reside on the same machine. In particular, X provides the X protocol, which is an application-level communication protocol by which an instance of *Xlib* can exchange data and events with the X kernel. For example, *Xlib* can send

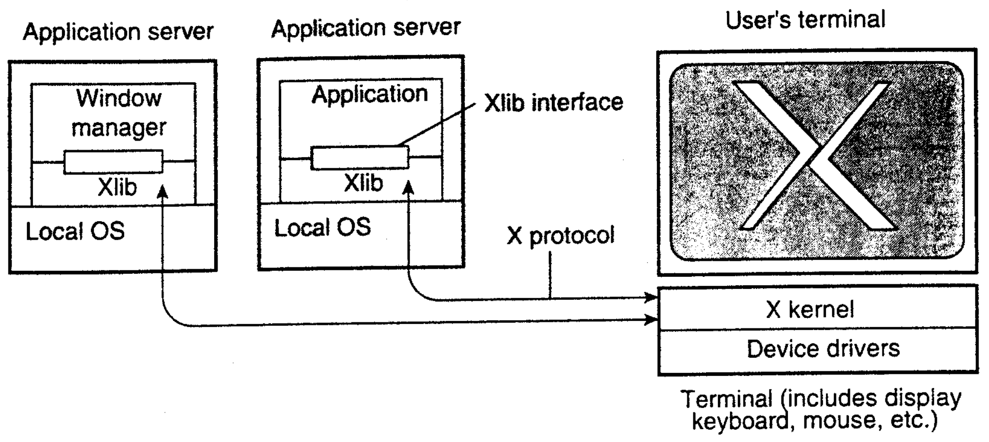


Figure 3-9. The basic organization of the X Window System.

requests to the X kernel for creating or killing a window, setting colors, and defining the type of cursor to display, among many other requests. In turn, the X kernel will react to local events such as keyboard and mouse input by sending event packets back to *Xlib*.

Several applications can communicate at the same time with the X kernel. There is one specific application that is given special rights, known as the **window manager.** This application can dictate the "look and feel" of the display as it appears to the user. For example, the window manager can prescribe how each window is decorated with extra buttons, how windows are to be placed on the display, and so. Other applications will have to adhere to these rules.

It is interesting to note how the X window system actually fits into client-server computing. From what we have described so far, it should be clear that the X kernel receives requests to manipulate the display. It gets these requests from (possibly remote) applications. In this sense, the X kernel acts as a server, while the applications play the role of clients. This terminology has been adopted by X, and although strictly speaking is correct, it can easily lead to confusion.

Thin-Client Network Computing

Obviously, applications manipulate a display using the specific display commands as offered by X. These commands are generally sent over the network where they are subsequently executed by the X kernel. By its nature, applications written for X should preferably separate application logic from user-interface commands. Unfortunately, this is often not the case. As reported by Lai and Nieh (2002), it turns out that much of the application logic and user interaction are tightly coupled, meaning that an application will send many requests to the X kernel for which it will expect a response before being able to make a next step. This synchronous behavior may adversely affect performance when operating over a wide-area network with long latencies.

There are several solutions to this problem. One is to re-engineer the implementation of the X protocol, as is done with NX (Pinzari, 2003). An important part of this work concentrates on bandwidth reduction by compressing X messages. First, messages are considered to consist of a fixed part, which is treated as an identifier, and a variable part. In many cases, multiple messages will have the same identifier in which case they will often contain similar data. This property can be used to send only the differences between messages having the same identifier.

Both the sending and receiving side maintain a local cache of which the entries can be looked up using the identifier of a message. When a message is sent, it is first looked up in the local cache. If found, this means that a previous message with the same identifier but possibly different data had been sent. In that case, differential encoding is used to send only the differences between the two. At the receiving side, the message is also looked up in the local cache, after which decoding through the differences can take place. In the cache miss, standard compression techniques are used, which generally already leads to factor four improvement in bandwidth. Overall, this technique has reported bandwidth reductions up to a factor 1000, which allows X to also run through low-bandwidth links of only 9600 kbps.

An important side effect of caching messages is that the sender and receiver have shared information on what the current status of the display is. For example, the application can request geometric information on various objects by simply requesting lookups in the local cache. Having this shared information alone already reduces the number of messages required to keep the application and the display synchronized.

Despite these improvements, X still requires having a display server running. This may be asking a lot, especially if the display is something as simple as a cell phone. One solution to keeping the software at the display very simple is to let all the processing take place at the application side. Effectively, this means that the entire display is controlled up to the pixel level at the application side. Changes in the bitmap are then sent over the network to the display, where they are immediately transferred to the local frame buffer.

This approach requires sophisticated compression techniques in order to prevent bandwidth availability to become a problem. For example, consider displaying a video stream at a rate of 30 frames per second on a 320 x 240 screen. Such a screen size is common for many PDAs. If each pixel is encoded by 24 bits, then without compression we would need a bandwidth of approximately 53 Mbps. Compression is clearly needed in such a case, and many techniques are currently being deployed. Note, however, that compression requires decompression at the receiver, which, in turn, may be computationally expensive without hardware support. Hardware support can be provided, but this raises the devices cost.

The drawback of sending raw pixel data in comparison to higher-level protocols such as X is that it is impossible to make any use of application semantics, as these are effectively lost at that level. Baratto et al. (2005) propose a different technique. In their solution, referred to as THINC, they provide a few high-level display commands that operate at the level of the video device drivers. These commands are thus device dependent, more powerful than raw pixel operations, but less powerful compared to what a protocol such as X offers. The result is that display servers can be much simpler, which is good for CPU usage, while at the same time application-dependent optimizations can be used to reduce bandwidth and synchronization.

In THINC, display requests from the application are intercepted and translated into the lower level commands. By intercepting application requests, THINC can make use of application semantics to decide what combination of lower level commands can be used best. Translated commands are not immediately sent out to the display, but are instead queued. By batching several commands it is possible to aggregate display commands into a single one, leading to fewer messages. For example, when a new command for drawing in a particular region of the screen effectively overwrites what a previous (and still queued) command would have established, the latter need not be sent out to the display. Finally, instead of letting the display ask for refreshments, THINC always pushes updates as they come available. This push approach saves latency as there is no need for an update request to be sent out by the display.

As it turns out, the approach followed by THINC provides better overall performance, although very much in line with that shown by NX. Details on performance comparison can be found in Baratto et al. (2005).

Compound Documents

Modem user interfaces do a lot more than systems such as X or its simple applications. In particular, many user interfaces allow applications to share a single graphical window, and to use that window to exchange data through user actions. Additional actions that can be performed by the user include what are generally called drag-and-drop operations, and in-place editing, respectively.

A typical example of drag-and-drop functionality is moving an icon representing a file *A* to an icon representing a trash can, resulting in the file being deleted. In this case, the user interface will need to do more than just arrange icons on the display: it will have to pass the name of the file *A* to the application associated with the trash can as soon as *A's* icon has been moved above that of the trash can application. Other examples easily come to mind.

In-place editing can best be illustrated by means of a document containing text and graphics. Imagine that the document is being displayed within a standard word processor. As soon as the user places the mouse above an image, the user interface passes that information to a drawing program to allow the user to modify the image. For example, the user may have rotated the image, which may effect the placement of the image in the document. The user interface therefore finds out what the new height and width of the image are, and passes this information to the word processor. The latter, in turn, can then automatically update the page layout of the document.

The key idea behind these user interfaces is the notion of a compound document, which can be defined as a collection of documents, possibly of very different kinds (like text, images, spreadsheets, etc.), which are seamlessly integrated at the user-interface level. A user interface that can handle compound documents hides the fact that different applications operate on different parts of the document. To the user, all parts are integrated in a seamless way. When changing one part affects other parts, the user interface can take appropriate measures, for example, by notifying the relevant applications.

Analogous to the situation described for the X Window System, the applications associated with a compound document do not have to execute on the client's machine. However, it should be clear that user interfaces that support compound documents may have to do a lot more processing than those that do not.

3.3.2 Client-Side Software for Distribution Transparency

Client software comprises more than just user interfaces. In many cases, parts of the processing and data level in a client-server application are executed on the client side as well. A special class is formed by embedded client software, such as for automatic teller machines (ATMs), cash registers, barcode readers, TV set-top boxes, etc. In these cases, the user interface is a relatively small part of the client software, in contrast to the local processing and communication facilities.

Besides the user interface and other application-related software, client software comprises components for achieving distribution transparency. Ideally, a client should not be aware that it is communicating with remote processes. In contrast, distribution is often less transparent to servers for reasons of performance and correctness. For example, in Chap. 6 we will show that replicated servers sometimes need to communicate in order to establish that operations are performed in a specific order at each replica.

Access transparency is generally handled through the generation of a client stub from an interface definition of what the server has to offer. The stub provides the same interface as available at the server, but hides the possible differences in machine architectures, as well as the actual communication.

There are different ways to handle location, migration, and relocation transparency. Using a convenient naming system is crucial, as we shall also see in the next chapter. In many cases, cooperation with client-side software is also important. For example, when a client is already bound to a server, the client can be directly informed when the server changes location. In this case, the client's middleware can hide the server's current geographical location from the user, and also transparently rebind to the server if necessary. At worst, the client's application may notice a temporary loss of performance.

In a similar way, many distributed systems implement replication transparency by means of client-side solutions. For example, imagine a distributed system with replicated servers, Such replication can be achieved by forwarding a request to each replica, as shown in Fig. 3-10. Client-side software can transparently collect all responses and pass a single response to the client application.

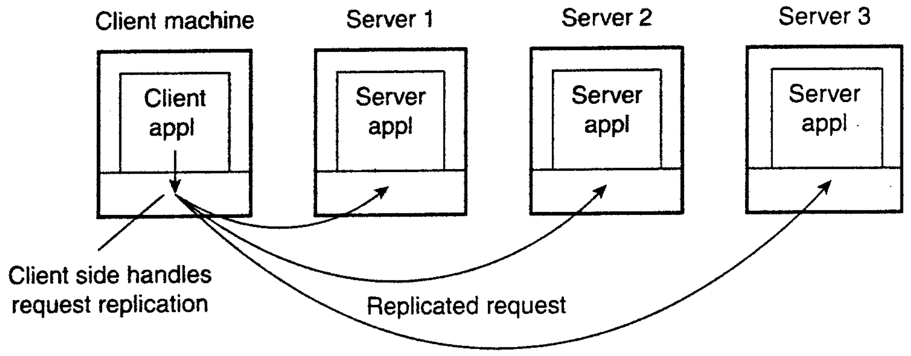


Figure 3-10. Transparent replication of a server using a client-side solution.

Finally, consider failure transparency. Masking communication failures with a server is typically done through client middleware. For example, client middleware can be configured to repeatedly attempt to connect to a server, or perhaps try another server after several attempts. There are even situations in which the client middleware returns data it had cached during a previous session, as is sometimes done by Web browsers that fail to connect to a server.

Concurrency transparency can be handled through special intermediate servers, notably transaction monitors, and requires less support from client software. Likewise, persistence transparency is often completely handled at the server.

3.4 SERVERS

Let us now take a closer look at the organization of servers. In the following pages, we first concentrate on a number of general design issues for servers, to be followed by a discussion of server clusters.

3.4.1 General Design Issues

A server is a process implementing a specific service on behalf of a collection of clients. In essence, each server is organized in the same way: it waits for an incoming request from a client and subsequently ensures that the request is taken care of, after which it waits for the next incoming request.

There are several ways to organize servers. In the case of an iterative server, the server itself handles the request and, if necessary, returns a response to the requesting client. A concurrent server does not handle the request itself, but passes it to a separate thread or another process, after which it immediately waits for the next incoming request. A multithreaded server is an example of a concurrent server. An alternative implementation of a concurrent server is to fork a new process for each new incoming request. This approach is followed in many UNIX systems. The thread or process that handles the request is responsible for returning a response to the requesting client.

Another issue is where clients contact a server. In all cases, clients send requests to an end point, also called a port, at the machine where the server is running. Each server listens to a specific end point. How do clients know the end point of a service? One approach is to globally assign end points for well-known services. For example, servers that handle Internet FTP requests always listen to TCP port 21. Likewise, an HTTP server for the World Wide Web will always listen to TCP port 80. These end points have been assigned by the Internet Assigned Numbers Authority (lANA), and are documented in Reynolds and Postel (1994). With assigned end points, the client only needs to find the network address of the machine where the server is running. As we explain in the next chapter, name services can be used for that purpose.

There are many services that do not require a preassigned end point. For example, a time-of-day server may use an end point that is dynamically assigned to it 9Y its local operating system. In that case, a client will first have to look up the end point. One solution is to have a special daemon running on each machine that runs servers. The daemon keeps track of the current end point of each service implemented by a co-located server. The daemon itself listens to a well-known end point. A client will first contact the daemon, request the end point, and then c~mtact the specific server, as shown in Fig. 3-11(a).

It is common to associate an end point with a specific service. However, actually implementing each service by means of a separate server may be a waste of resources. For example, in a typical UNIX system, it is common to have lots of servers running simultaneously, with most of them passively waiting until a client request comes in. Instead of having to keep track of so many passive processes, it is often more efficient to have a single superserver listening to each end point associated with a specific service, as shown in Fig. 3-1l(b). This is the approach taken, for example, with the *inetd* daemon in UNIX. *Inetd* listens to a number of well-known ports for Internet services. When a request comes in, the daemon forks a process to take further care of the request. That process will exit after it is finished.

Another issue that needs to be taken into account when designing a server is whether and how a server can be interrupted. For example, consider a user who has just decided to upload a huge file to an FTP server. Then, suddenly realizing that it is the wrong file, he wants to interrupt the server to cancel further data

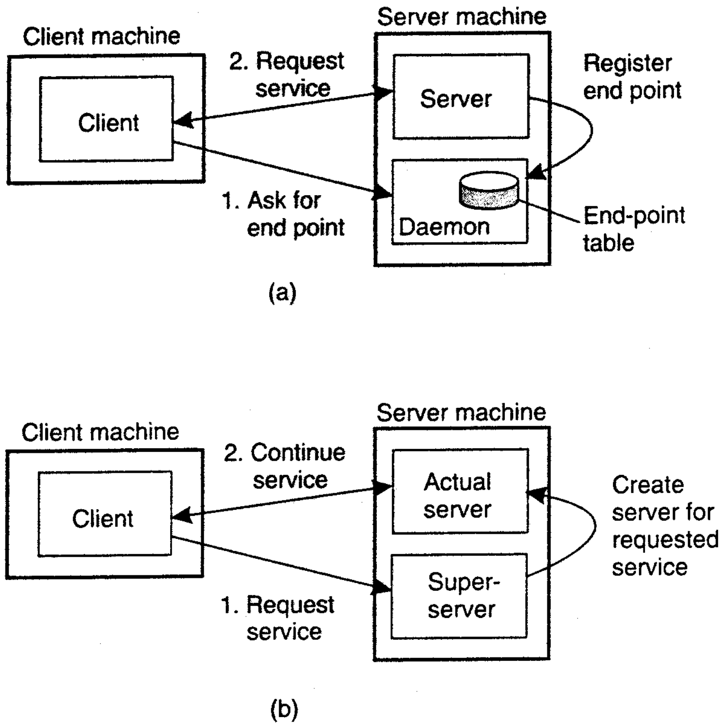


Figure 3-11. (a) Client-to-server binding using a daemon. (b) Client-to-server binding using a superserver..

transmission. There are several ways to do this. One approach that works only too well in the current Internet (and is sometimes the only alternative) is for the user to abruptly exit the client application (which will automatically break the connection to the server), immediately restart it, and pretend nothing happened. The server will eventually tear down the old connection, thinking the client has probably crashed.

A much better approach for handling communication interrupts is to develop the client and server such that it is possible to send **out-of-band** data, which is data that is to be processed by the server before any other data from that client. One solution is to let the server listen to a separate control end point to which the client sends out-of-band data, while at the same time listening (with a lower priority) to the end point through which the normal data passes. Another solution is to send out-of-band data across the same connection through which the client is sending the original request.. In TCP, for example, it is possible to transmit urgent data. When urgent data are received at the server, the latter is interrupted (e.g.• through a signal in UNIX systems), after which it can inspect the data and handle them accordingly.

A final, important design issue, is whether or not the server is stateless. A stateless server does not keep information on the state of its clients, and can change its own state without having to inform any client (Birman, 2005). A Web server, for example, is stateless. It merely responds to incoming HTTP requests, which can be either for uploading a file to the server or (most often) for fetching a file. When the request has been processed, the Web server forgets the client completely. Likewise, the collection of files that a Web server manages (possibly in cooperation with a file server), can be changed without clients having to be informed.

Note that in many stateless designs, the server actually does maintain information on its clients, but crucial is the fact that if this information is lost, it will not lead to a disruption of the service offered by the server. For example, a Web server generally logs all client requests. This information is useful, for example, to decide whether certain documents should be replicated, and where they should be replicated to. Clearly, there is no penalty other than perhaps in the form of suboptimal performance if the log is lost.

A particular form of a stateless design is where the server maintains what is known as soft state. In this case, the server promises to maintain state on behalf of the client, but only for a limited time. After that time has expired, the server falls back to default behavior, thereby discarding any information it kept on account of the associated client. An example of this type of state is a server promising to keep a client informed about updates, but only for a limited time. After that, the client is required to poll the server for updates. Soft-state approaches originate from protocol design in computer networks, but can be equally applied to server design (Clark, 1989; and Lui et al., 2004).

In contrast, a stateful server generally maintains persistent information on its clients. This means that the information needs to be explicitly deleted by the server. A typical example is a file server that allows a client to keep a local copy of a file, even for performing update operations. Such a server would maintain a table containing *(client, file)* entries. Such a table allows the server to keep track of which client currently has the update permissions on which file, and thus possibly also the most recent version of that file.

This approach can improve the performance of read and write operations as perceived by the client. Performance improvement over stateless servers is often an important benefit of stateful designs. However, the example also illustrates the major drawback of stateful servers. If the server crashes, it has to recover its table of *(client, file)* entries, or otherwise it cannot guarantee that it has processed the most recent updates on a file. In general, a stateful server needs to recover its entire state as it was just before the crash. As we discuss in Chap. 8, enabling recovery can introduce considerable complexity. In a stateless design, no special measures need to be taken at all for a crashed server to recover. It simply starts running again, and waits for client requests to come in.

Ling et al. (2004) argue that one should actually make a distinction between (temporary) session state and permanent state. The example above is typical for session state: it is associated with a series of operations by a single user and should be maintained for a some time, but not indefinitely. As it turns out, session state is often maintained in three-tiered client-server architectures, where the application server actually needs to access a database server through a series of queries before being able to respond to the requesting client. The issue here is that no real harm is done if session state is lost, provided that the client can simply reissue the original request. This observation allows for simpler and less reliable storage of state.

What remains for permanent state is typically information maintained in databases, such as customer information, keys associated with purchased software, etc. However, for most distributed systems, maintaining session state already implies a stateful design requiring special measures when failures do happen and making explicit assumptions about the durability of state stored at the server. We will return to these matters extensively when discussing fault tolerance.

When designing a server, the choice for a stateless or stateful design should not affect the services provided by the server. For example, if files have to be opened before they can be read from, or written to, then a stateless server should one way or the other mimic this behavior. A common solution, which we discuss in more detail in Chap. 11. is that the server responds to a read or write request by first opening the referred file, then does the actual read or write operation, and immediately closes the file again.

In other cases, a server may want to keep a record on a client's behavior so that it can more effectively respond to its requests. For example, Web servers sometimes offer the possibility to immediately direct a client to his favorite pages. This approach is possible only if the server has history information on that client. When the server cannot maintain state, a common solution is then to let the client send along additional information on its previous accesses. In the case of the Web, this information is often transparently stored by the client's browser in what is called a cookie, which is a small piece of data containing client-specific information that is of interest to the server. Cookies are never executed by a browser; they are merely stored.

The first time a client accesses a server, the latter sends a cookie along with the requested Web pages back to the browser, after which the browser safely tucks the cookie away. Each subsequent time the client accesses the server, its cookie for that server is sent along with the request. Although in principle, this approach works fine, the fact that cookies are sent back for safekeeping by the browser is often hidden entirely from users. So much for privacy. Unlike most of grandma's cookies, these cookies should stay where they are baked.

3.4.2 Server Clusters

In Chap. 1 we briefly discussed cluster computing as one of the many appearances of distributed systems. We now take a closer look at the organization of server clusters, along with the salient design issues.

General Organization

Simply put, a server cluster is nothing else but a collection of machines connected through a network, where each machine runs one or more servers. The server clusters that we consider here, are the ones in which the machines are connected through a local-area network, often offering high bandwidth and low latency.

In most cases, a server cluster is logically organized into three tiers, as shown in Fig. 3-12. The first tier consists of a (logical) switch through which client requests are routed. Such a switch can vary widely. For example, transport-layer switches accept incoming TCP connection requests and pass requests on to one of servers in the cluster, as we discuss below. A completely different example is a Web server that accepts incoming HTTP requests, but that partly passes requests to application servers for further processing only to later collect results and return an HTTP response.

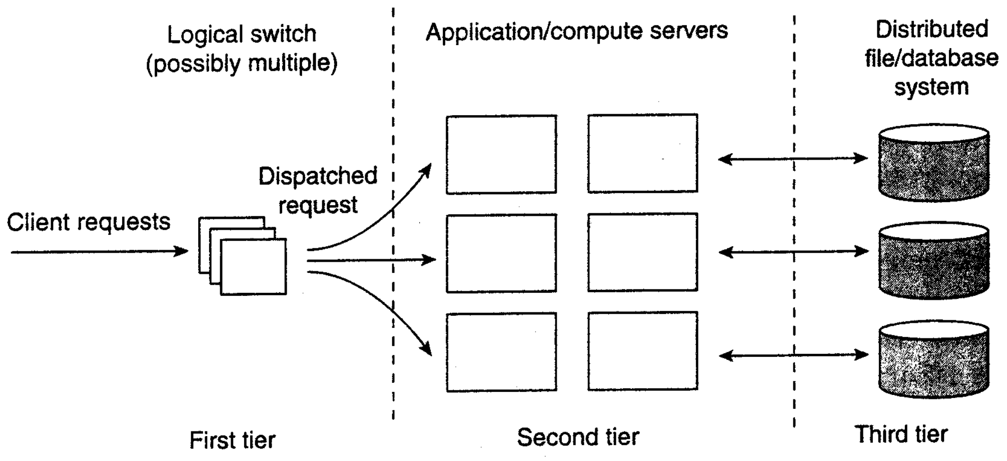

Logical switch (possibly multiple)
Application/compute servers
Distributed file/database system
Dispatched request
Client requests
First tier
Second tier
Third tier

Figure 3-12. The general organization of a three-tiered server cluster.

As in any multitiered client-server architecture, many server clusters also contain servers dedicated to application processing. In cluster computing, these are typically servers running on high-performance hardware dedicated to delivering compute power. However, in the case of enterprise server clusters, it may be the case that applications need only run on relatively low-end machines, as the required compute power is not the bottleneck, but access to storage is.

This brings us the third tier, which consists of data-processing servers, notably file and database servers. Again, depending on the usage of the server cluster, these servers may be running an specialized machines, configured for high-speed disk access and having large server-side data caches.

Of course, not all server clusters will follow this strict separation. It is frequently the case that each machine is equipped with its own local storage, often integrating application and data processing in a single server leading to a two-tiered architecture. For example, when dealing with streaming media by means of a server cluster, it is common to deploy a two-tiered system architecture, where each machine acts as a dedicated media server (Steinmetz and Nahrstedt, 2004).

When a server cluster offers multiple services, it may happen that different machines run different application servers. As a consequence, the switch will have to be able to distinguish services or otherwise it cannot forward requests to the proper machines. As it turns out, many second-tier machines run only a single application. This limitation comes from dependencies on available software and hardware, but also that different applications are often managed by different administrators. The latter do not like to interfere with each other's machines.

As a consequence, we may find that certain machines are temporarily idle, while others are receiving an overload of requests. What would be useful is to temporarily migrate services to idle machines. A solution proposed in Awadallah and Rosenblum (2004), is to use virtual machines allowing a relative easy migration of code to real machines. We will return to code migration later in this chapter.

Let us take a closer look at the first tier, consisting of the switch. An important design goal for server clusters is to hide the fact that there are multiple servers. In other words, client applications running on remote machines should have no need to know anything about the internal organization of the cluster. This access transparency is invariably offered by means of a single access point, in turn implemented through some kind of hardware switch such as a dedicated machine.

The switch forms the entry point for the server cluster, offering a single network address. For scalability and availability, a server cluster may have multiple access points, where each access point is then realized by a separate dedicated machine. We consider only the case of a single access point.

A standard way of accessing a server cluster is to set up a TCP connection over which application-level requests are then sent as part of a session. A session ends by tearing down the connection. In the case of transport-layer switches, the switch accepts incoming TCP connection requests, and hands off such connections to one of the servers (Hunt et al, 1997; and Pai et al., 1998). The principle working of what is commonly known as TCP handoff is shown in Fig. 3-13.

When the switch receives a TCP connection request, it subsequently identifies the best server for handling that request, and forwards the request packet to that server. The server, in turn, will send an acknowledgment back to the requesting client, but inserting the switch's IP address as the source field of the header of the IP packet carrying the TCP segment. Note that this spoofing is necessary for the client to continue executing the TCP protocol: it is expecting an answer back from the switch, not from some arbitrary server it is has never heard of before. Clearly, a TCP-handoff implementation requires operating-system level modifications.

It can already be seen that the switch can play an important role in distributing the load among the various servers. By deciding where to forward a request to, the

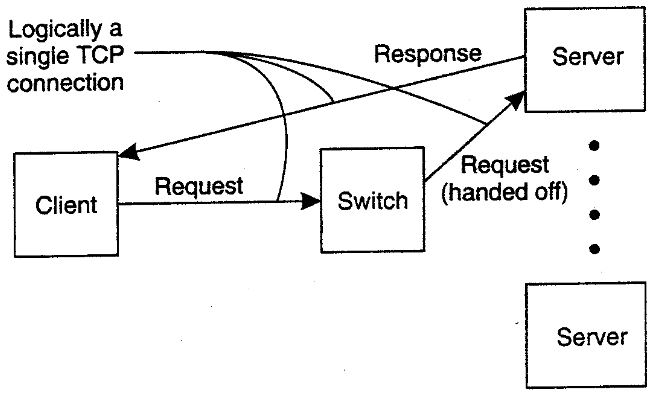


Figure 3-13. The principle ofTCP handoff.

switch also decides which server is to handle further processing of the request. The simplest load-balancing policy that the switch can follow is round robin: each time it picks the next server from its list to forward a request to.

More advanced server selection criteria can be deployed as well. For example, assume multiple services are offered by the server cluster. If the switch can distinguish those services when a request comes in, it can then take informed decisions on where to forward the request to. This server selection can still take place at the transport level, provided services are distinguished by means of a port number. One step further is to have the switch actually inspect the payload of the incoming request. This method can be applied only if it is known what that payload can look like. For example, in the case of Web servers, the switch can eventually expect an HTTP request, based on which it can then decide who is to process it. We will return to such content-aware request distribution when we discuss Web-based systems in Chap. 12.

Distributed Servers

The server clusters discussed so far are generally rather statically configured. In these clusters, there is often an separate administration machine that keeps track of available servers, and passes this information to other machines as appropriate, such as the switch.

As we mentioned, most server clusters offer a single access point. When that point fails, the cluster becomes unavailable. To eliminate this potential problem, several access points can be provided, of which the addresses are made publicly available. For example, the Domain Name System (DNS) can return several addresses, all belonging to the same host name. This approach still requires clients to make several attempts if one of the addresses fails. Moreover, this does not solve the problem of requiring static access points.

Having stability, like a long-living access point, is a desirable feature from a client's and a server's perspective. On the other hand, it also desirable to have a high degree of flexibility in configuring a server cluster, including the switch. This observation has lead to a design of a distributed server which effectively is nothing but a possibly dynamically changing set of machines, with also possibly varying access points, but which nevertheless appears to the outside world as a single. powerful machine. The design of such a distributed server is given in Szymaniak et al. (2005). We describe it briefly here.

The basic idea behind a distributed server is that clients benefit from a robust, high-performing, stable server. These properties can often be provided by high-end mainframes, of which some have an acclaimed mean time between failure of more than 40 years. However, by grouping simpler machines transparently into a cluster, and not relying on the availability of a single machine, it may be possible to achieve a better degree of stability than by each component individually. For example, such a cluster could be dynamically configured from end-user machines, as in the case of a collaborative distributed system.

Let us concentrate on how a stable access point can be achieved in such a system. The main idea is to make use of available networking services, notably mobility support for IP version 6 (MIPv6). In MIPv6, a mobile node is assumed to have a home network where it normally resides and for which it has an associated stable address, known as its home address (HoA). This home network has a special router attached, known as the home agent, which will take care of traffic to the mobile node when it is away. To this end, when a mobile node attaches to a foreign network, it will receive a temporary care-of address (CoA) where it can be reached. This care-of address is reported to the node's home agent who will then see to it that all traffic is forwarded to the mobile node. Note that applications communicating with the mobile node will only see the address associated with the node's home network. They will never see the care-of address.

This principle can be used to offer a stable address of a distributed server. In this case, a single unique contact address is initially assigned to the server cluster. The contact address will be the server's life-time address to be used in all communication with the outside world. At any time, one node in the distributed server will operate as an access point using that contact address, but this role can easily be taken over by another node. What happens is that the access point records its own address as the care-of address at the home agent associated with the distributed server. At that point, all traffic will be directed to the access point, who will then take care in distributing requests among the currently participating nodes. If the access point fails, a simple fail-over mechanism comes into place by which another access point reports a new care-of address.

This simple configuration would make the home agent as well as the access point a potential bottleneck as all traffic would flow through these two machines. This situation can be avoided by using an MIPv6 feature known as *route optimization.* Route optimization works as follows. Whenever a mobile node with home address *HA* reports its current care-of address, say *CA*, the horne agent can forward *CA* to a client. The latter will then locally store the pair *(HA, CA)*. From that moment on, communication will be directly forwarded to *CA*. Although the application at the client side can still use the horne address, the underlying support software for MIPv6 will translate that address to *CA* and use that instead.

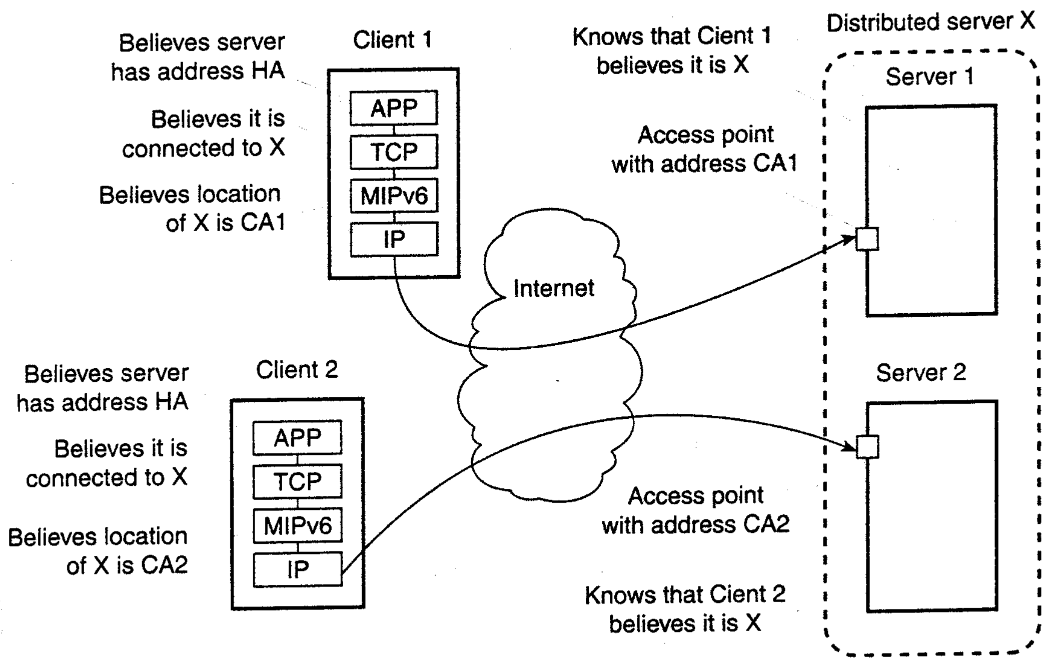

Believes server has address HA
Believes it is connected to X
Believes location of X is CA1
Client 1
APP
TCP
MIPv6
IP
Knows that Cient 1 believes it is X
Access point with address CA1
Distributed server X
Server 1
Internet
Believes server has address HA
Believes it is connected to X
Believes location of X is CA2
Client 2
APP
TCP
MIPv6
IP
Server 2
Access point with address CA2
Knows that Cient 2 believes it is X

Figure 3-14. Route optimization in a distributed server.

Route optimization can be used to make different clients believe they are communicating with a single server, where, in fact, each client is communicating with a different member node of the distributed server, as shown in Fig. 3-14. To this end, when an access point of a distributed server forwards a request from client C_1 to, say node S_1 (with address *CA* 1), it passes enough information to S_1 to let it initiate the route optimization procedure by which eventually the client is made to believe that the care-of address is CA_1. This will allow C_1 to store the pair *(HA, CA* 1). During this procedure, the access point (as well as the horne agent) tunnel most of the traffic between C_1 and S_1. This will prevent the horne agent from believing that the care-of address has changed, so that it will continue to communicate with the access point.

Of course, while this route optimization procedure is taking place, requests from other clients may still corne in. These remain in a pending state at the access point until they can be forwarded. The request from another client C_2 may then be forwarded to member node S_2 (with address *CA* 2), allowing the latter to let client

C_2 store the pair (HA, CA_2). As a result, different clients will be directly communicating with different members of the distributed server, where each client application still has the illusion that this server has address *HA*. The home agent continues to communicate with the access point talking to the contact address.

3.4.3 Managing Server Clusters

A server cluster should appear to the outside world as a single computer, as is indeed often the case. However, when it comes to managing a cluster, the situation changes dramatically. Several attempts have been made to ease the management of server clusters as we discuss next.

Common Approaches

By far the most common approach to managing a server cluster is to extend the traditional managing functions of a single computer to that of a cluster. In its most primitive form, this means that an administrator can log into a node from a remote client and execute local managing commands to monitor, install, and change components.

Somewhat more advanced is to hide the fact that you need to login into a node and instead provide an interface at an administration machine that allows to collect information from one or more servers, upgrade components, add and remove nodes, etc. The main advantage of the latter approach is that collective operations, which operate on a group of servers, can be more easily provided. This type of managing server clusters is widely applied in practice, exemplified by management software such as Cluster Systems Management from IBM (Hochstetler and Beringer, 2004).

However, as soon as clusters grow beyond several tens of nodes, this type of management is not the way to go. Many data centers need to manage thousands of servers, organized into many clusters but all operating collaboratively. Doing this by means of centralized administration servers is simply out of the question. Moreover, it can be easily seen that very large clusters need continuous repair management (including upgrades). To simplify matters, if p is the probability that a server is currently faulty, and we assume that faults are independent, then for a cluster of N servers to operate without a single server being faulty is $(1-p)^N$. With $p=0.001$ and $N=1000$, there is only a 36 percent chance that all the servers are correctly functioning.

As it turns out, support for very large server clusters is almost always ad hoc. There are various rules of thumb that should be considered (Brewer, 2001), but there is no systematic approach to dealing with massive systems management. Cluster management is still very much in its infancy, although it can be expected that the self-managing solutions as discussed in the previous chapter will eventually find their way in this area, after more experience with them has been gained.

Example: PlanetLab

Let us now take a closer look at a somewhat unusual cluster server. PlanetLab is a collaborative distributed system in which different organizations each donate one or more computers, adding up to a total of hundreds of nodes. Together, these computers form a 1-tier server cluster, where access, processing, and storage can all take place on each node individually. Management of PlanetLab is by necessity almost entirely distributed. Before we explain its basic principles, let us first describe the main architectural features (Peterson et al., 2005).

In PlanetLab, an organization donates one or more nodes, where each node is easiest thought of as just a single computer, although it could also be itself a cluster of machines. Each node is organized as shown in Fig. 3-15. There are two important components (Bavier et al., 2004). The first one is the virtual machine monitor (VMM), which is an enhanced Linux operating system. The enhancements mainly comprise adjustments for supporting the second component, namely vservers. A (Linux) vserver can best be thought of as a separate environment in which a group of processes run. Processes from different vservers are *completely* independent. They cannot directly share any resources such as files, main memory, and network connections as is normally the case with processes running on top of an operating systems. Instead, a vserver provides an environment consisting of its own collection of software packages, programs, and networking facilities. For example, a vserver may provide an environment in which a process will notice that it can make use of Python 1.5.2 in combination with an older Apache Web server, say *httpd* 1.3.1. In contrast, another vserver may support the latest versions of Python and *httpd*. In this sense, calling a vserver a "server" is a bit of a misnomer as it really only isolates groups of processes from each other. We return to vservers briefly below.

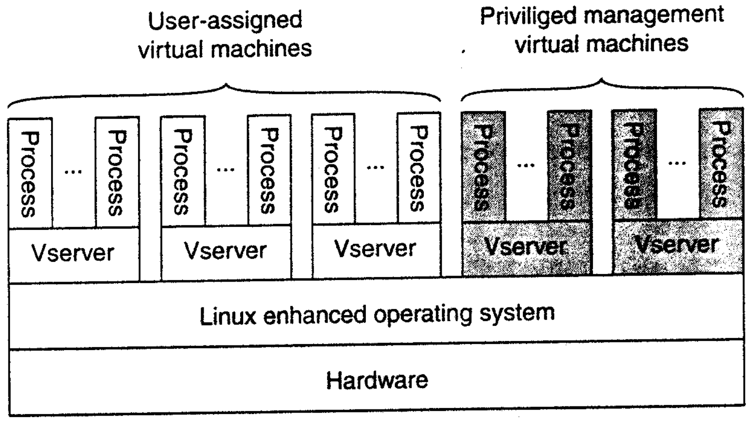

User-assigned virtual machines
Priviliged management virtual machines
Process
...
Process
Vserver
Linux enhanced operating system
Hardware

Figure 3-15. The basic organization of a PlanetLab node.

The Linux VMM ensures that vservers are separated: processes in different vservers are executed concurrently and independently, each making use only of the software packages and programs available in their own environment. The isolation between processes in different vservers is strict. For example, two processes in different vservers may have the same user ill, but this does not imply that they stem from the same user. This separation considerably eases supporting users from different organizations that want to use PlanetLab as, for example, a testbed to experiment with completely different distributed systems and applications.

To support such experimentations, PlanetLab introduces the notion of a slice, which is a set of vservers, each vserver running on a different node. A slice can thus be thought of as a virtual server cluster, implemented by means of a collection of virtual machines. The virtual machines in PlanetLab run on top of the Linux operating system, which has been extended with a number of kernel modules

There are several issues that make management of PlanetLab a special problem. Three salient ones are:

1. Nodes belong to different organizations. Each organization should be allowed to specify who is allowed to run applications on their nodes, and restrict resource usage appropriately.
2. There are various monitoring tools available, but they all assume a very specific combination of hardware and software. Moreover, they are all tailored to be used within a single organization.
3. Programs from different slices but running on the same node should not interfere with each other. This problem is similar to process independence in operating systems.

Let us take a look at each of these issues in more detail.

Central to managing PlanetLab resources is the node manager. Each node has such a manager, implemented by means of a separate vserver, whose only task is to create other vservers on the node it manages and to control resource allocation. The node manager does not make any policy decisions; it is merely a mechanism to provide the essential ingredients to get a program running on a given node.

Keeping track of resources is done by means of a resource specification, or *rspee* for short. An *rspee* specifies a time interval during which certain resources have been allocated. Resources include disk space, file descriptors, inbound and outbound network bandwidth, transport-level end points, main memory, and CPU usage. An *rspee* is identified through a globally unique 128-bit identifier known as a resource capability *(reap).* Given an *reap,* the node manager can look up the associated *rspee* in a local table.

Resources are bound to slices. In other words, in order to make use of resources, it is necessary to create a slice. Each slice is associated with a service provider, which can best be seen as an entity having an account on PlanetLab.

Every slice can then be identified by a *(principal.sid, slice.tag)* pair, where the *principal.iid* identifies the provider and *slice.stag* is an identifier chosen by the provider.

To create a new slice, each node will run a slice creation service (SCS), which, in tum, can contact the node manager requesting it to create a vserver and to allocate resources. The node manager itself cannot be contacted directly over a network, allowing it to concentrate only on local resource management. In tum, the SCS will not accept slice-creation requests from just anybody. Only specific slice authorities are eligible for requesting the creation of a slice. Each slice authority will have access rights to a collection of nodes. The simplest model is that there is only a single slice authority that is allowed to request .slice creation on all nodes.

To complete the picture, a service provider will contact a slice authority and request it to create a slice across a collection of nodes. The service provider will be known to the slice authority, for example, because it has been previously authenticated and subsequently registered as a PlanetLab user. In practice, PlanetLab users contact a slice authority by means of a Web-based service. Further details can be found in Chun and Spalink (2003).

What this procedure reveals is that managing PlanetLab is done through intermediaries. One important class of such intermediaries is formed by slice authorities. Such authorities have obtained credentials at nodes to create slides. Obtaining these credentials has been achieved out-of-band, essentially by contacting system administrators at various sites. Obviously, this is a time-consuming process which not be carried out by end users (or, in PlanetLab terminology; service providers).

Besides slice authorities, there are also management authorities. Where a slice authority concentrates only on managing slices, a management authority is responsible for keeping an eye on nodes. In particular, it ensures that the nodes under its regime run the basic PlanetLab software and abide to the rules set out by PlanetLab. Service providers trust that a management authority provides nodes that will behave properly.

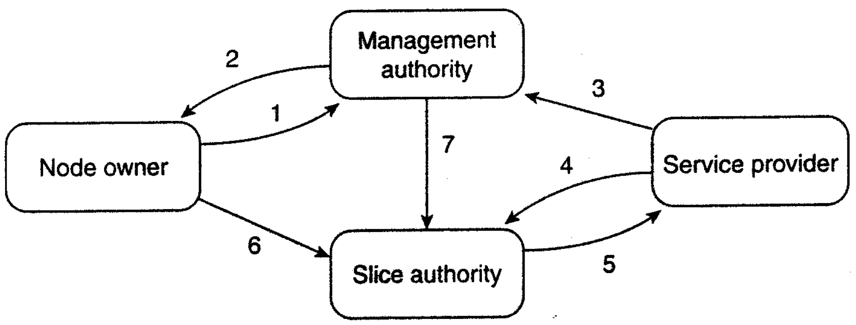


Figure 3-16. The management relationships between various PlanetLab entities.

This organization leads to the management structure shown in Fig. 3-16. described in terms of trust relationships in Peterson et at (2005). The relations are as follows:

1. A node owner puts its node under the regime of a management authority, possibly restricting usage where appropriate.
2. A management authority provides the necessary software to add a node to PlanetLab.
3. A service provider registers itself with a management authority. trusting it to provide well-behaving nodes.
4. A service provider contacts a slice authority to create a slice on a collection of nodes.
5. The slice authority needs to authenticate the service provider.
6. A node owner provides a slice creation service for a slice authority to create slices. It essentially delegates resource management to the slice authority.
7. A management authority delegates the creation of slices to a slice authority.

These relationships cover the problem of delegating nodes in a controlled way such that a node owner can rely on a decent and secure management. The second issue that needs to be handled is monitoring. What is needed is a unified approach to allow users to see how well their programs are behaving within a specific slice.

PlanetLab follows a simple approach. Every node is equipped with a collection of sensors, each sensor being capable of reporting information such as CPU usage, disk activity, and so on. Sensors can be arbitrarily complex, but the important issue is that,they always report information on a per-node basis. This information is made available by means of a Web server: every sensor is accessible through simple HTTP requests (Bavier et at, 2004).

Admittedly, this approach to monitoring is still rather primitive, but it should be seen as a basis for advanced 'monitoring schemes. For example, there is, in principle, no reason why Astrolabe, which we discussed in Chap. 2, cannot be used for aggregated sensor readings across multiple nodes.

Finally, to come to our third management issue, namely the protection of programs against each other, PlanetLab uses Linux virtual servers (called vservers) to isolate slices. As mentioned, the main idea of a vserver is to run applications in there own environment, which includes all files that are normally shared across a single machine. Such a separation can be achieved relatively easy by means of the UNIX chroot command, which effectively changes the root of the file system from where applications will look for files. Only the superuser can execute chroot.

Of course, more is needed. Linux virtual servers not only separate the file system, but also normally shared information on processes, network addresses, memory usage, and so on. As a consequence, a physical machine is actually partitioned into multiple units, each unit corresponding to a full-fledged Linux environment, isolated from the other parts. An overview of Linux virtual servers can be found in Potzl et al. (2005).

3.5 CODE MIGRATION

So far, we have been mainly concerned with distributed systems in which communication is limited to passing data. However, there are situations in which passing programs, sometimes even while they are being executed, simplifies the design of a distributed system. In this section, we take a detailed look at what code migration actually is. We start by considering different approaches to code migration, followed by a discussion on how to deal with the local resources that a migrating program uses. A particularly hard problem is migrating code in heterogeneous systems, which is also discussed.

3.5.1 Approaches to Code Migration

Before taking a look at the different forms of code migration, let us first consider why it may be useful to migrate code.

Reasons for Migrating Code

Traditionally, code migration in distributed systems took place in the form of process migration in which an entire process was moved from one machine to another (Milojicic et al., 2000). Moving a running process to a different machine is a costly and intricate task, and there had better be a good reason for doing so. That reason has always been performance. The basic idea is that overall system performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines. Load is often expressed in terms of the CPU queue length or CPU utilization, but other performance indicators are used as well.

Load distribution algorithms by which decisions are made concerning the allocation and redistribution of tasks with respect to a set of processors, play an important role in compute-intensive systems. However, in many modem distributed systems, optimizing computing capacity is less an issue than, for example, trying to minimize communication. Moreover, due to the heterogeneity of the underlying platforms and computer networks, performance improvement through code migration is often based on qualitative reasoning instead of mathematical models.

Consider, as an example, a client-server system in which the server manages a huge database. If a client application needs to perform many database operations involving large quantities of data, it may be better to ship part of the client application to the server and send only the results across the network. Otherwise, the network may be swamped with the transfer of data from the server to the client. In this case, code migration is based on the assumption that it generally makes sense to process data close to where those data reside.

This same reason can be used for migrating parts of the server to the client. For example, in many interactive database applications, clients need to fill in forms that are subsequently translated into a series of database operations. Processing the form at the client side, and sending only the completed form to the server, can sometimes avoid that a relatively large number of small messages need to cross the network. The result is that the client perceives better performance, while at the same time the server spends less time on form processing and communication.

Support for code migration can also help improve performance by exploiting parallelism, but without the usual intricacies related to parallel programming. A typical example is searching for information in the Web. It is relatively simple to implement a search query in the form of a small mobile program, called a mobile agent, that moves from site to site. By making several copies of such a program, and sending each off to different sites, we may be able to achieve a linear speedup compared to using just a single program instance.

Besides improving performance, there are other reasons for supporting code migration as well. The most important one is that of flexibility. The traditional approach to building distributed applications is to partition the application into different parts, and decide in advance where each part should be executed. This approach, for example, has led to the different multitiered client-server applications discussed in Chap. 2.

However, if code can move between different machines, it becomes possible to dynamically configure distributed systems. For example, suppose a server implements a standardized interface to a file system. To allow remote clients to access the file system, the server makes use of a proprietary protocol. Normally, the client-side implementation of the file system interface, which is based on that protocol, would need to be linked with the client application. This approach requires that the software be readily available to the client at the time the client application is being developed.

An alternative is to let the server provide the client's implementation no sooner than is strictly necessary, that is, when the client binds to the server. At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server. This principle is shown in Fig. 3-17. This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing code is standardized. Also, it is necessary that the downloaded code can be executed on the client's machine. Different solutions are discussed below and in later chapters.

The important advantage of this model of dynamically downloading client-side software is that clients need not have all the software preinstalled to talk to

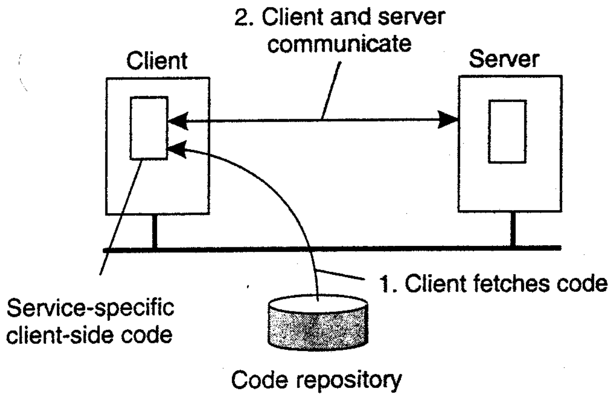

2. Client and server communicate
Client
Server
1. Client fetches code
Service-specific client-side code
Code repository

Figure 3-17. The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

servers. Instead, the software can be moved in as necessary, and likewise, discarded when no longer needed. Another advantage is that as long as interfaces are standardized, we can change the client-server protocol and its implementation as often as we like. Changes will not affect existing client applications that rely on the server. There are, of course, also disadvantages. The most serious one, which we discuss in Chap. 9, has to do with security. Blindly trusting that the downloaded code implements only the advertised interface while accessing your unprotected hard disk and does not send the juiciest parts to heaven-knows-who may not always be such a good idea.

Models for Code Migration

Although code migration suggests that we move only code between machines, the term actually covers a much richer area. Traditionally, communication in distributed systems is concerned with exchanging data between processes. Code migration in the broadest sense deals with moving programs between machines, with the intention to have those programs be executed at the target. In some cases, as in process migration, the execution status of a program, pending signals, and other parts of the environment must be moved as well.

To get a better understanding of the different models for code migration, we use a framework described in Fuggetta et al. (1998). In this framework, a process consists of three segments. The *code segment* is the part that contains the set of instructions that make up the program that is being executed. The *resource segment* contains references to external resources needed. by the process, such as files, printers, devices, other processes, and so on. Finally, an *execution segment* is used to store the current execution state of a process, consisting of private data, the stack, and, of course, the program counter.

The bare minimum for code migration is to provide only weak mobility. In this model, it is possible to transfer only the code segment, along with perhaps some initialization data. A characteristic feature of weak mobility is that a transferred program is always started from one of several predefined starting positions. This is what happens, for example, with Java applets, which always start execution from the beginning. The benefit of this approach is its simplicity. Weak mobility requires only that the target machine can execute that code, which essentially boils down to making the code portable. We return to these matters when discussing migration in heterogeneous systems.

In contrast to weak mobility, in systems that support strong mobility the execution segment can be transferred as well. The characteristic feature of strong mobility is that a running process can be stopped, subsequently moved to another machine, and then resume execution where it left off. Clearly, strong mobility is much more general than weak mobility, but also much harder to implement.

Irrespective of whether mobility is weak or strong, a further distinction can be made between sender-initiated and receiver-initiated migration. In sender-initiated migration, migration is initiated at the machine where the code currently resides or is being executed. Typically, sender-initiated migration is done when uploading programs to a compute server. Another example is sending a search program across the Internet to a Web database server to perform the queries at that server. In receiver-initiated migration, the initiative for code migration is taken by the target machine. Java applets are an example of this approach.

Receiver-initiated migration is simpler than sender-initiated migration. In many cases, code migration occurs between a client and a server, where the client takes the initiative for migration. Securely uploading code to a server, as is done in sender-initiated migration, often requires that the client has previously been registered and authenticated at that server. In other words, the server is required to know all its clients, the reason being is that the client will presumably want access to the server's resources such as its disk. Protecting such resources is essential. In contrast, downloading code as in the receiver-initiated case, can often be done anonymously. Moreover, the server is generally not interested in the client's resources. Instead, code migration to the client is done only for improving client-side performance. To that end, only a limited number of resources need to be protected, such as memory and network connections. We return to secure code migration extensively in Chap. 9.

In the case of weak mobility, it also makes a difference if the migrated code is executed by the target process, or whether a separate process is started. For example, Java applets are simply downloaded by a Web browser and are executed in the browser's address space. The benefit of this approach is that there is no need to start a separate process, thereby avoiding communication at the target machine. The main drawback is that the target process needs to be protected against malicious or inadvertent code executions. A simple solution is to let the operating system take care of that by creating a separate process to execute the migrated code.

Note that this solution does not solve the resource-access problems mentioned above. They still have to be dealt with.

Instead of moving a running process, also referred to as process migration, strong mobility can also be supported by remote cloning. In contrast to process migration, cloning yields an exact copy of the original process, but now running on a different machine. The cloned process is executed in parallel to the original process. In UNIX systems, remote cloning takes place by forking off a child process and letting that child continue on a remote machine. The benefit of cloning is that the model closely resembles the one that is already used in many applications. The only difference is that the cloned process is executed on a different machine. In this sense, migration by cloning is a simple way to improve distribution transparency.

The various alternatives for code migration are summarized in Fig. 3-18.

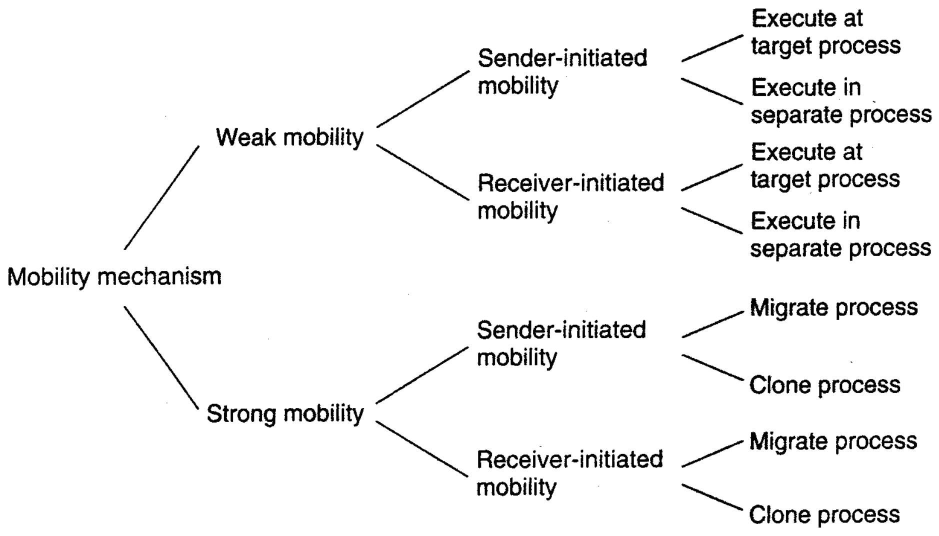

Mobility mechanism
Weak mobility
Strong mobility
Sender-initiated mobility
Receiver-initiated mobility
Sender-initiated mobility
Receiver-initiated mobility
Execute at target process
Execute in separate process
Execute at target process
Execute in separate process
Migrate process
Clone process
Migrate process
Clone process

Figure 3-18. Alternatives for code migration.

3.5.2 Migration and Local Resources

So far, only the migration of the code and execution segment has been taken into account. The resource segment requires some special attention. What often makes code migration so difficult is that the resource segment cannot always be simply transferred along with the other segments without being changed. For example, suppose a process holds a reference to a specific TCP port through which it was communicating with other (remote) processes. Such a reference is held in its resource segment. When the process moves to another location, it will have to give up the port and request a new one at the destination. In other cases, transferring a reference need not be a problem. For example, a reference to a file by means of an absolute URL will remain valid irrespective of the machine where the process that holds the URL resides.

To understand the implications that code migration has on the resource segment, Fuggetta et al. (1998) distinguish three types of process-to-resource bindings. The strongest binding is when a process refers to a resource by its identifier. In that case, the process requires precisely the referenced resource, and nothing else. An example of such a binding by identifier is when a process uses a VRL to refer to a specific Web site or when it refers to an FrP server by means of that server's Internet address. In the same line of reasoning, references to local communication end points also lead to a binding by identifier.

A weaker form of process-to-resource binding is when only the value of a resource is needed. In that case, the execution of the process would not be affected if another resource would provide that same value. A typical example of binding by value is when a program relies on standard libraries, such as those for programming in C or Java. Such libraries should always be locally available, but their exact location in the local file system may differ between sites. Not the specific files, but their content is important for the proper execution of the process.

Finally, the weakest form of binding is when a process indicates it needs only a resource of a specific type. This binding by type is exemplified by references to local devices, such as monitors, printers, and so on.

When migrating code, we often need to change the references to resources, but cannot affect the kind of process-to-resource binding. If, and exactly how a reference should be changed, depends on whether that resource can be moved along with the code to the target machine. More specifically, we need to consider the resource-to-machine bindings, and distinguish the following cases. Unattached resources can be easily moved between different machines, and are typically (data) files associated only with the program that is to be migrated. In contrast, moving or copying a fastened resource may be possible, but only at relatively high costs. Typical examples of fastened resources are local databases and complete Web sites. Although such resources are, in theory, not dependent on their current machine, it is often infeasible to move them to another environment. Finally, fixed resources are intimately bound to a specific machine or environment and cannot be moved. Fixed resources are often local devices. Another example of a fixed resource is a local communication end point.

Combining three types of process-to-resource bindings, and three types of resource-to-machine bindings, leads to nine combinations that we need to consider when migrating code. These nine combinations are shown in Fig. 3-19.

Let us first consider the possibilities when a process is bound to a resource by identifier. When the resource is unattached, it is generally best to move it along with the migrating code. However, when the resource is shared by other processes, an alternative is to establish a global reference, that is, a reference that can cross machine boundaries. An example of such a reference is a URL. When the resource is fastened or fixed, the best solution is also to create a global reference.

		Resource-to-machine binding		
		Unattached	Fastened	Fixed
Process-to-resource binding	By identifier	MV (or GR)	GR (or MV)	GR
	By value	CP (or MV,GR)	GR (or CP)	GR
	By type	RB (or MV,CP)	RB (or GR,CP)	RB (or GR)

GR	Establish a global systemwide reference
MV	Move the resource
CP	Copy the value of the resource
RB	Rebind process to locally-available resource

Figure 3-19. Actions to be taken with respect to the references to local resources when migrating code to another machine.

It is important to realize that establishing a global reference may be more than just making use of URLs, and that the use of such a reference is sometimes prohibitively expensive. Consider, for example, a program that generates high-quality images for a dedicated multimedia workstation. Fabricating high-quality images in real time is a compute-intensive task, for which reason the program may be moved to a high-performance compute server. Establishing a global reference to the multimedia workstation means setting up a communication path between the compute server and the workstation. In addition, there is significant processing involved at both the server and the workstation to meet the bandwidth requirements of transferring the images. The net result may be that moving the program to the compute server is not such a good idea, only because the cost of the global reference is too high.

Another example of where establishing a global reference is not always that easy is when migrating a process that is making use of a local communication end point. In that case, we are dealing with a fixed resource to which the process is bound by the identifier. There are basically two solutions. One solution is to let the process set up a connection to the source machine after it has migrated and install a separate process at the source machine that simply forwards all incoming messages. The main drawback of this approach is that whenever the source machine malfunctions, communication with the migrated process may fail. The alternative solution is to have all processes that communicated with the migrating process, change *their* global reference, and send messages to the new communication end point at the target machine.

The situation is different when dealing with bindings by value. Consider first a fixed resource. The combination of a fixed resource and binding by value occurs, for example, when a process assumes that memory can be shared between processes. Establishing a global reference in this case would mean that we need to implement a distributed form of shared memory. In many cases, this is not really a viable or efficient solution.

Fastened resources that are referred to by their value, are typically runtime libraries. Normally, copies of such resources are readily available on the target machine, or should otherwise be copied before code migration takes place. Establishing a global reference is a better alternative when huge amounts of data are to be copied, as may be the case with dictionaries and thesauruses in text processing systems.

The easiest case is when dealing with unattached resources. The best solution is to copy (or move) the resource to the new destination, unless it is shared by a number of processes. In the latter case, establishing a global reference is the only option.

The last case deals with bindings by type. Irrespective of the resource-to-machine binding, the obvious solution is to rebind the process to a locally available resource of the same type. Only when such a resource is not available, will we need to copy or move the original one to the new destination, or establish a global reference.

3.5.3 Migration in Heterogeneous Systems

So far, we have tacitly assumed that the migrated code can be easily executed at the target machine. This assumption is in order when dealing with homogeneous systems. In general, however, distributed systems are constructed on a heterogeneous collection of platforms, each having their own operating system and machine architecture. Migration in such systems requires that each platform is supported, that is, that the code segment can be executed on each platform. Also, we need to ensure that the execution segment can be properly represented at each platform.

The problems coming from heterogeneity are in many respects the same as those of portability. Not surprisingly, solutions are also very similar. For example, at the end of the 1970s, a simple solution to alleviate many of the problems of porting Pascal to different machines was to generate machine-independent intermediate code for an abstract virtual machine (Barron, 1981). That machine, of course, would need to be implemented on many platforms, but it would then allow Pascal programs to be run anywhere. Although this simple idea was widely used for some years, it never really caught on as the general solution to portability problems for other languages, notably C.

About 25 years later, code migration in heterogeneous systems is being attacked by scripting languages and highly portable languages such as Java. In essence, these solutions adopt the same approach as was done for porting Pascal. All such solutions have in common that they rely on a (process) virtual machine that either directly interprets source code (as in the case of scripting languages), or otherwise interprets intermediate code generated by a compiler (as in Java). Being in the right place at the right time is also important for language developers.

Recent developments have started to weaken the dependency on programming languages. In particular, solutions have been proposed not only to migrate processes, but to migrate entire computing environments. The basic idea is to compartmentalize the overall environment and to provide processes in the same part their own view on their computing environment.

If the compartmentalization is done properly, it becomes possible to decouple a part from the underlying system and actually migrate it to another machine. In this way, migration would actually provide a form of strong mobility for processes, as they can then be moved at any point during their execution, and continue where they left off when migration completes. Moreover, many of the intricacies related to migrating processes while they have bindings to local resources may be solved, as these bindings are in many cases simply preserved. The local resources, namely, are often part of the environment that is being migrated.

There are several reasons for wanting to migrate entire environments, but perhaps the most important one is that it allows continuation of operation while a machine needs to be shutdown. For example, in a server cluster, the systems administrator may decide to shut down or replace a machine, but will not have to stop all its running processes. Instead, it can temporarily freeze an environment, move it to another machine (where it sits next to other, existing environments), and simply unfreeze it again. Clearly, this is an extremely powerful way to manage long-running compute environments and their processes.

Let us consider one specific example of migrating virtual machines, as discussed in Clark et al. (2005). In this case, the authors concentrated on real-time migration of a virtualized operating system, typically something that would be convenient in a cluster of servers where a tight coupling is achieved through a single, shared local-area network. Under these circumstances, migration involves two major problems: migrating the entire memory image and migrating bindings to local resources.

As to the first problem, there are, in principle, three ways to handle migration (which can be combined):

1. Pushing memory pages to the new machine and resending the ones that are later modified during the migration process.
2. Stopping the current virtual machine; migrate memory, and start the new virtual machine.
3. Letting the new virtual machine pull in new pages as needed, that is, let processes start on the new virtual machine immediately and copy memory pages on demand.

The second option may lead to unacceptable downtime if the migrating virtual machine is running a live service, that is, one that offers continuous service. On the other hand, a pure on-demand approach as represented by the third option may extensively prolong the migration period, but may also lead to poor performance because it takes a long time before the working set of the migrated processes has been moved to the new machine.

As an alternative, Clark et al. (2005) propose to use a pre-copy approach which combines the first option, along with a brief stop-and-copy phase as represented by the second option. As it turns out, this combination can lead to service downtimes of 200 ms or less.

Concerning local resources, matters are simplified when dealing only with a cluster server. First, because there is a single network, the only thing that needs to be done is to announce the new network-to-MAC address binding, so that clients can contact the migrated processes at the correct network interface. Finally, if it can be assumed that storage is provided as a separate tier (like we showed in Fig. 3-12), then migrating binding to files is similarly simple.

The overall effect is that, instead of migrating processes, we now actually see that an entire operating system can be moved between machines.

3.6 SUMMARY

Processes play a fundamental role in distributed systems as they form a basis for communication between different machines. An important issue is how processes are internally organized and, in particular, whether or not they support multiple threads of control. Threads in distributed systems are particularly useful to continue using the CPU when a blocking I/O operation is performed. In this way, it becomes possible to build highly-efficient servers that run multiple threads in parallel, of which several may be blocking to wait until disk I/O or network communication completes.

Organizing a distributed application in terms of clients and servers has proven to be useful. Client processes generally implement user interfaces, which may range from very simple displays to advanced interfaces that can handle compound documents. Client software is furthermore aimed at achieving distribution transparency by hiding details concerning the communication with servers, where those servers are currently located, and whether or not servers are replicated. In addition, client software is partly responsible for hiding failures and recovery from failures.

Servers are often more intricate than clients, but are nevertheless subject to only a relatively few design issues. For example, servers can either be iterative or concurrent, implement one or more services, and can be stateless or stateful. Other design issues deal with addressing services and mechanisms to interrupt a server after a service request has been issued and is possibly already being processed.

Special attention needs to be paid when organizing servers into a cluster. A common objective is hide the internals of a cluster from the outside world. This means that the organization of the cluster should be shielded from applications. To this end, most clusters use a single entry point that can hand off messages to servers in the cluster. A challenging problem is to transparently replace this single entry point by a fully distributed solution.

An important topic for distributed systems is the migration of code between different machines. Two important reasons to support code migration are increasing performance and flexibility. When communication is expensive, we can sometimes reduce communication by shipping computations from the server to the client, and let the client do as much local processing as possible. Flexibility is increased if a client can dynamically download software needed to communicate with a specific server. The downloaded software can be specifically targeted to that server, without forcing the client to have it preinstalled.

Code migration brings along problems related to usage of local resources for which it is required that either resources are migrated as well, new bindings to local resources at the target machine are established, or for which systemwide network references are used. Another problem is that code migration requires that we take heterogeneity into account. Current practice indicates that the best solution to handle heterogeneity is to use virtual machines. These can take either the form of process virtual machines as in the case of, for example, Java, or through using virtual machine monitors that effectively allow the migration of a collection of processes along with their underlying operating system.

PROBLEMS

1. In this problem you are to compare reading a file using a single-threaded file server and a multithreaded server. It takes 15 msec to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in a cache in main memory. If a disk operation is needed, as is the case one-third of the time, an additional 75 msec is required, during which time the thread sleeps. How many requests/sec can the server handle if it is single threaded? If it is multithreaded?
2. Would it make sense to limit the number of threads in a server process?
3. In the text, we described a multithreaded tile server, showing why it is better than a single-threaded server and a finite-state machine server. Are there any circumstances in which a single-threaded server might be better? Give an example.
4. Statically associating only a single thread with a lightweight process is not such a good idea. Why not?
5. Having only a single lightweight process per process is also not such a good idea. Why not?
6. Describe a simple scheme in which there are as many lightweight processes as there are runnable threads.

7. X designates a user's terminal as hosting the server, while the application is referred to as the client. Does this make sense?

8. The X protocol suffers from scalability problems. How can these problems be tackled?

9. Proxies can support replication transparency by invoking each replica, as explained in the text. Can (the server side of) an application be subject to a replicated calls?

10. Constructing a concurrent server by spawning a process has some advantages and disadvantages compared to multithreaded servers. Mention a few.

11. Sketch the design of a multithreaded server that supports multiple protocols using sockets as its transport-level interface to the underlying operating system.

12. How can we prevent an application from circumventing a window manager, and thus being able to completely mess up a screen?

13. Is a server that maintains a TCP/IP connection to a client stateful or stateless?

14. Imagine a Web server that maintains a table in which client IP addresses are mapped to the most recently accessed Web pages. When a client connects to the server, the server looks up the client in its table, and if found, returns the registered page. Is this server stateful or stateless?

15. Strong mobility in UNIX systems could be supported by allowing a process to fork a child on a remote machine. Explain how this would work.

16. In Fig. 3-18 it is suggested that strong mobility cannot be combined with executing migrated code in a target process. Give a counterexample.

17. Consider a process *P* that requires access to file *F* which is locally available on the machine where *P* is currently running. When *P* moves to another machine, it still requires access to *F*. If the file-to-machine binding is fixed, how could the systemwide reference to *F* be implemented?

18. Describe in detail how TCl' packets flow in the case of TCP handoff, along with the information on source and destination addresses in the various headers.

4

COMMUNICATION

Interprocess communication is at the heart of all distributed systems. It makes no sense to study distributed systems without carefully examining the ways that processes on different machines can exchange information. Communication in distributed systems is always based on low-level message passing as offered by the underlying network. Expressing communication through message passing is harder than using primitives based on shared memory, as available for nondistributed platforms. Modem distributed systems often consist of thousands or even millions of processes scattered across a network with unreliable communication such as the Internet. Unless the primitive communication facilities of computer networks are replaced by something else, development of large-scale distributed applications is extremely difficult.

In this chapter, we start by discussing the rules that communicating processes must adhere to, known as protocols, and concentrate on structuring those protocols in the form of layers. We then look at three widely-used models for communication: Remote Procedure Call (RPC), Message-Oriented Middleware (MOM), and data streaming. We also discuss the general problem of sending data to multiple receivers, called multicasting.

Our first model for communication in distributed systems is the remote procedure call (RPC). An RPC aims at hiding most of the intricacies of message passing, and is ideal for client-server applications.

In many distributed applications, communication does not follow the rather strict pattern of client-server interaction. **In** those cases, it turns out that thinking in terms of messages is more appropriate. However, the low-level communication facilities of computer networks are in many ways not suitable due to their lack of distribution transparency. An alternative is to use a high-level message-queuing model, in which communication proceeds much the same as in electronic mail systems. Message-oriented middleware (MOM) is a subject important enough to warrant a section of its own.

With the advent of multimedia distributed systems, it became apparent that many systems were lacking support for communication of continuous media, such as audio and video. What is needed is the notion of a stream that can support the continuous flow of messages, subject to various timing constraints. Streams are discussed in a separate section.

Finally, since our understanding of setting up multicast facilities has improved, novel and elegant solutions for data dissemination have emerged. We pay separate attention to this subject in the last section of this chapter.

4.1 FUNDAMENTALS

Before we start our discussion on communication in distributed systems, we first recapitulate some of the fundamental issues related to communication. In the next section we briefly discuss network communication protocols, as these form the basis for any distributed system. After that, we take a different approach by classifying the different types of communication that occurs in distributed systems.

4.1.1 Layered Protocols

Due to the absence of shared memory, all communication in distributed systems is based on sending and receiving (low level) messages. When process *A* wants to communicate with process *B*, it first builds a message in its own address space. Then it executes a system call that causes the operating system to send the message over the network to *B*. Although this basic idea sounds simple enough, in order to prevent chaos, *A* and *B* have to agree on the meaning of the bits being sent. If *A* sends a brilliant new novel written in French and encoded in IBM's EBCDIC character code, and *B* expects the inventory of a supermarket written in English and encoded in ASCII, communication will be less than optimal.

Many different agreements are needed. How many volts should be used to signal a 0-bit, and how many volts for a 1-bit? How does the receiver know which is the last bit of the message? How can it detect if a message has been damaged or lost, and what should it do if it finds out? How long are numbers, strings, and other data items, and how are they represented? In short, agreements are needed at a variety of levels, varying from the low-level details of bit transmission to the high-level details of how information is to be expressed.

To make it easier to deal with the numerous levels and issues involved in communication, the International Standards Organization (ISO) developed a reference model that clearly identifies the various levels involved, gives them standard names, and points out which level should do which job. This model is called the Open Systems Interconnection Reference Model (Day and Zimmerman, 1983), usually abbreviated as ISO OSI or sometimes just the OSI model. It should be emphasized that the protocols that were developed as part of the OSI model were never widely used and are essentially dead now. However, the underlying model itself has proved to be quite useful for understanding computer networks. Although we do not intend to give a full description of this model and all of its implications here, a short introduction will be helpful. For more details, see Tanenbaum (2003).

The OSI model is designed to allow open systems to communicate. An open system is one that is prepared to communicate with any other open system by using standard rules that govern the format, contents, and meaning of the messages sent and received. These rules are formalized in what are called protocols. To allow a group of computers to communicate over a network, they must all agree on the protocols to be used. A distinction is made between two general types of protocols. With connection oriented protocols, before exchanging data the sender and receiver first explicitly establish a connection, and possibly negotiate the protocol they will use. When they are done, they must release (terminate) the connection. The telephone is a connection-oriented communication system. With connectionless protocols, no setup in advance is needed. The sender just transmits the first message when it is ready. Dropping a letter in a mailbox is an example of connectionless communication. With computers, both connection-oriented and connectionless communication are common.

In the OSI model, communication is divided up into seven levels or layers, as shown in Fig. 4-1. Each layer deals with one specific aspect of the communication. In this way, the problem can be divided up into manageable pieces, each of which can be solved independent of the others. Each layer provides an interface to the one above it. The interface consists of a set of operations that together define the service the layer is prepared to offer its users.

When process *A* on machine 1 wants to communicate with process *B* on machine 2, it builds a message and passes the message to the application layer on its machine. This layer might be a library procedure, for example, but it could also be implemented in some other way (e.g., inside the operating system, on an external network processor, etc.). The application layer software then adds a header to the front of the message and passes the resulting message across the layer 6/7 interface to the presentation layer. The presentation layer in tum adds its own header and passes the result down to the session layer, and so on. Some layers add not only a header to the front, but also a trailer to the end. When it hits the bottom, the physical layer actually transmits the message (which by now might look as shown in Fig. 4-2) by putting it onto the physical transmission medium.

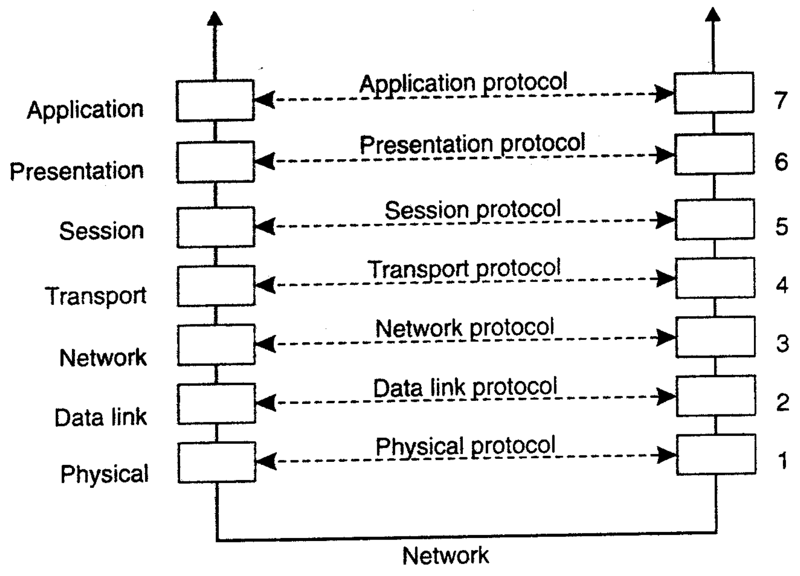


Figure 4-1. Layers, interfaces, and protocols in the OSI model.

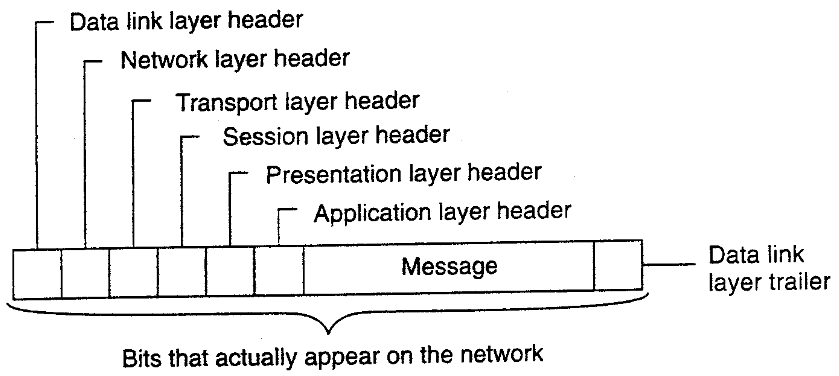


Figure 4-2. A typical message as it appears on the network.

When the message arrives at machine 2, it is passed upward, with each layer stripping off and examining its own header. Finally, the message arrives at the receiver, process *B*, which may reply to it using the reverse path. The information in the layer *n* header is used for the layer *n* protocol.

As an example of why layered protocols are important, consider communication between two companies, Zippy Airlines and its caterer, Mushy Meals, Inc. Every month, the head of passenger service at Zippy asks her secretary to contact the sales manager's secretary at Mushy to order 100,000 boxes of rubber chicken. Traditionally, the orders went via the post office. However, as the postal service deteriorated, at some point the two secretaries decided to abandon it and communicate bye-mail. They could do this without bothering their bosses, since their protocol deals with the physical transmission of the orders, not their contents.

Similarly, the head of passenger service can decide to drop the rubber chicken and go for Mushy's new special, prime rib of goat, without that decision affecting the secretaries. The thing to notice is that we have two layers here, the bosses and the secretaries. Each layer has its own protocol (subjects of discussion and technology) that can be changed independently of the other one. It is precisely this independence that makes layered protocols attractive. Each one can be changed as technology improves, without the other ones being affected.

In the OSI model, there are not two layers, but seven, as we saw in Fig. 4-1. The collection of protocols used in a particular system is called a protocol suite or protocol stack.. It is important to distinguish a *reference model* from its actual *protocols.* As we mentioned, the OSI protocols were never popular. In contrast, protocols developed for the Internet, such as TCP and IP, are mostly used. In the following sections, we will briefly examine each of the OSI layers in turn, starting at the bottom. However, instead of giving examples of OSI protocols, where appropriate, we will point out some of the Internet protocols used in each layer.

Lower-Level Protocols

We start with discussing the three lowest layers of the OSI protocol suite. Together, these layers implement the basic functions that encompass a computer network.

The physical layer is concerned with transmitting the Os and Is. How many volts to use for 0 and 1, how many bits per second can be sent, and whether transmission can take place in both directions simultaneously are key issues in the physical layer. In addition, the size and shape of the network connector (plug), as well as the number of pins and meaning of each are of concern here.

The physical layer protocol deals with standardizing the electrical, mechanical, and signaling interfaces so that when one machine sends a 0 bit it is actually received as a 0 bit and not a 1 bit. Many physical layer standards have been developed (for different media), for example, the RS-232-C standard for serial communication lines.

The physical layer just sends bits. As long as no errors occur, all is well. However, real communication networks are subject to errors, so some mechanism is needed to detect and correct them. This mechanism is the main task of the data link layer. What it does is to group the bits into units, sometimes called frames, and see that each frame is correctly received.

The data link layer does its work by putting a special bit pattern on the start and end of each frame to mark them, as well as computing a checksum by adding up all the bytes in the frame in a certain way. The data link layer appends the checksum to the frame. When the frame arrives, the receiver recomputes the checksum from the data and compares the result to the checksum following the frame. If the two agree, the frame is considered correct and is accepted. It they disagree. the receiver asks the sender to retransmit it. Frames are assigned sequence numbers (in the header), so everyone can tell which is which.

On a LAN, there is usually no need for the sender to locate the receiver. It just puts the message out on the network and the receiver takes *it* off. A wide-area network, however, consists of a large number of machines, each with some number of lines to other machines, rather like a large-scale map showing major cities and roads connecting them. For a message to get from the sender to the receiver it may have to make a number of hops, at each one choosing an outgoing line to use. The question of how to choose the best path is called routing, and is essentially the primary task of the network layer.

The problem is complicated by the fact that the shortest route is not always the best route. What really matters is the amount of delay on a given route, which, in tum, is related to the amount of traffic and the number of messages queued up for transmission over the various lines. The delay can thus change over the course of time. Some routing algorithms try to adapt to changing loads, whereas others are content to make decisions based on long-term averages.

At present, the most widely used network protocol is the connectionless **IP** (Internet Protocol), which is part of the Internet protocol suite. An IP packet (the technical term for a message in the network layer) can be sent without any setup. Each IP packet is routed to its destination independent of all others. No internal path is selected and remembered.

Transport Protocols

The transport layer forms the last part of what could be called a basic network protocol stack, in the sense that it implements all those services that are not provided at the interface of the network layer, but which are reasonably needed to build network applications. In other words, the transport layer turns the underlying network into something that an application developer can use.

Packets can be lost on the way from the sender to the receiver. Although some applications can handle their own error recovery, others prefer a reliable connection. The job of the transport layer is to provide this service. The idea is that the application layer should be able to deliver a message to the transport layer with the expectation that it will be delivered without loss.

Upon receiving a message from the application layer, the transport layer breaks it into pieces small enough for transmission, assigns each one a sequence number, and then sends them all. The discussion in the transport layer header concerns which packets have been sent, which have been received, how many more the receiver has room to accept, which should be retransmitted, and similar topics.

Reliable transport connections (which by definition are connection oriented) can be built on top of connection-oriented or connectionless network services. In the former case all the packets will arrive in the correct sequence (if they arrive at all), but in the latter case it is possible for one packet to take a different route and arrive earlier than the packet sent before it. It is up to the transport layer software to put everything back in order to maintain the illusion that a transport connection is like a big tube-you put messages into it and they come out undamaged and in the same order in which they went in. Providing this end-to-end communication behavior is an important aspect of the transport layer.

The Internet transport protocol is called TCP (Transmission Control Protocol) and is described in detail in Comer (2006). The combination TCPIIP is now used as a de facto standard for network communication. The Internet protocol suite also supports a connectionless transport protocol called UDP (Universal Datagram Protocol), which is essentially just IP with some minor additions. User programs that do not need a connection-oriented protocol normally use UDP.

Additional transport protocols are regularly proposed. For example, to support real-time data transfer, the Real-time Transport Protocol (RTP) has been defined. RTP is a framework protocol in the sense that it specifies packet formats for real-time data without providing the actual mechanisms for guaranteeing data delivery. In addition, it specifies a protocol for monitoring and controlling data transfer of RTP packets (Schulzrinne et al., 2003).

Higher- Level Protocols

Above the transport layer, OSI distinguished three additional layers. In practice, only the application layer is ever used. In fact, in the Internet protocol suite, everything above the transport layer is grouped together. In the face of middleware systems, we shall see in this section that neither the OSI nor the Internet approach is really appropriate.

The session layer is essentially an enhanced version of the transport layer. It provides dialog control, to keep track of which party is currently talking, and it provides synchronization facilities. The latter are useful to allow users to insert checkpoints into long transfers, so that in the event of a crash, it is necessary to go back only to the last checkpoint, rather than all the way back to the beginning. In practice, few applications are interested in the session layer and it is rarely supported. It is not even present in the Internet protocol suite. However, in the context of developing middleware solutions, the concept of a session and its related protocols has turned out to be quite relevant, notably when defining higher-level communication protocols.

Unlike the lower layers, which are concerned with getting the bits from the sender to the receiver reliably and efficiently, the presentation layer is concerned with the meaning of the bits. Most messages do not consist of random bit strings, but more structured information such as people's names, addresses, amounts of money, and so on. In the presentation layer it is possible to define records containing fields like these and then have the sender notify the receiver that a message contains a particular record in a certain format. This makes it easier for machines with different internal representations to communicate with each other.

The OSI application layer was originally intended to contain a collection of standard network applications such as those for electronic mail, file transfer, and terminal emulation. By now, it has become the container for all applications and protocols that in one way or the other do not fit into one of the underlying layers. From the perspective of the OSI reference model, virtually all distributed systems are just applications.

What is missing in this model is a clear distinction between applications, application-specific protocols, and general-purpose protocols. For example, the Internet File Transfer Protocol (FTP) (Postel and Reynolds, 1985; and Horowitz and Lunt, 1997) defines a protocol for transferring files between a client and server machine. The protocol should not be confused with the *ftp* program, which is an end-user application for transferring files and which also (not entirely by coincidence) happens to implement the Internet FrP.

Another example of a typical application-specific protocol is the HyperText Transfer Protocol (HTTP) (Fielding et aI., 1999), which is designed to remotely manage and handle the transfer of Web pages. The protocol is implemented by applications such as Web browsers and Web servers. However, HTTP is now also used by systems that are not intrinsically tied to the Web. For example, Java's object-invocation mechanism uses HTTP to request the invocation of remote objects that are protected by a firewall (Sun Microsystems, 2004b).

There are also many general-purpose protocols that are useful to many applications, but which cannot be qualified as transport protocols. In many cases, such protocols fall into the category of middleware protocols, which we discuss next.

Middleware Protocols

Middleware is an application that logically lives (mostly) in the application layer, but which contains many general-purpose protocols that warrant their own layers, independent of other, more specific applications. A distinction can be made between high-level communication protocols and protocols for establishing various middleware services.

There are numerous protocols to support a variety of middleware services. For example, as we discuss in Chap. 9, there are various ways to establish authentication, that is, provide proof of a claimed identity. Authentication protocols are not closely tied to any specific application, but instead, can be integrated into a middleware system as a general service. Likewise, authorization protocols by which authenticated users and processes are granted access only to those resources for which they have authorization. tend to have a general, application-independent nature.

As another example, we shall consider a number of distributed commit protocols in Chap. 8. Commit protocols establish that in a group of processes either all processes carry out a particular operation, or that the operation is not carried out at all. This phenomenon is also referred to as atomicity and is widely applied in transactions. As we shall see, besides transactions, other applications, like fault-tolerant ones, can also take advantage of distributed commit protocols.

As a last example, consider a distributed locking protocol by which a resource can be protected against simultaneous access by a collection of processes that are distributed across multiple machines. We shall come across a number of such protocols in Chap. 6. Again, this is an example of a protocol that can be used to implement a general middleware service, but which, at the same time, is highly independent of any specific application.

Middleware communication protocols support high-level communication services. For example, in the next two sections we shall discuss protocols that allow a process to call a procedure or invoke an object on a remote machine in a highly transparent way. Likewise, there are high-level communication services for setting and synchronizing streams for transferring real-time data, such as needed for multimedia applications. As a last example, some middleware systems offer reliable multicast services that scale to thousands of receivers spread across a wide-area network.

Some of the middleware communication protocols could equally well belong in the transport layer, but there may be specific reasons to keep them at a higher level. For example, reliable multicasting services that guarantee scalability can be implemented only if application requirements are taken into account. Consequently, a middleware system may offer different (tunable) protocols, each in tum iraplernented using different transport protocols, but offering a single interface.

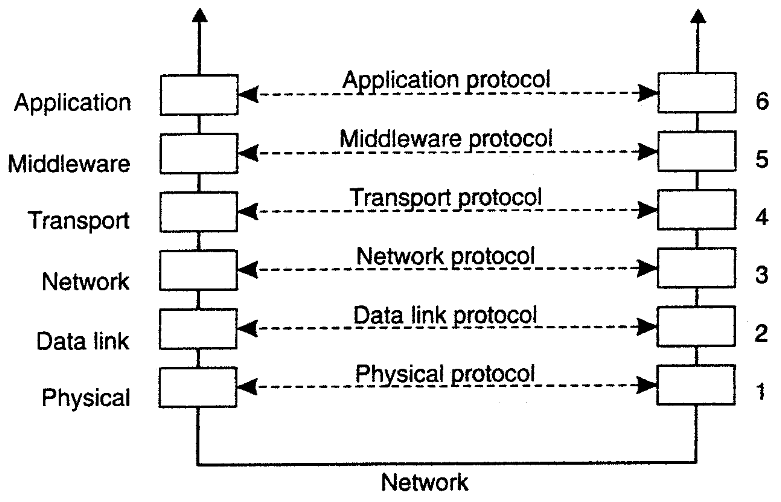


Figure 4-3. An adapted reference model for networked communication.

Taking this approach to layering leads to a slightly adapted reference model for communication, as shown in Fig. 4-3. Compared to the OSI model, the session and presentation layer have been replaced by a single middleware layer that contains application-independent protocols. These protocols do not belong in the lower layers we just discussed. The original transport services may also be offered as a middleware service, without being modified. This approach is somewhat analogous to offering UDP at the transport level. Likewise, middleware communication services may include message-passing services comparable to those offered by the transport layer.

In the remainder of this chapter, we concentrate on four high-level middleware communication services: remote procedure calls, message queuing services, support for communication of continuous media through streams, and multicasting. Before doing so, there are other general criteria for distinguishing (middleware) communication which we discuss next.

4.1.2 Types of Communication

To understand the various alternatives in communication that middleware can offer to applications, we view the middleware as an additional service in client-server computing, as shown in Fig. 4-4. Consider, for example an electronic mail system. In principle, the core of the mail delivery system can be seen as a middleware communication service. Each host runs a user agent allowing users to compose, send, and receive e-mail. A sending user agent passes such mail to the mail delivery system, expecting it, in tum, to eventually deliver the mail to the intended recipient. Likewise, the user agent at the receiver's side connects to the mail delivery system to see whether any mail has come in. If so, the messages are transferred to the user agent so that they can be displayed and read by the user.

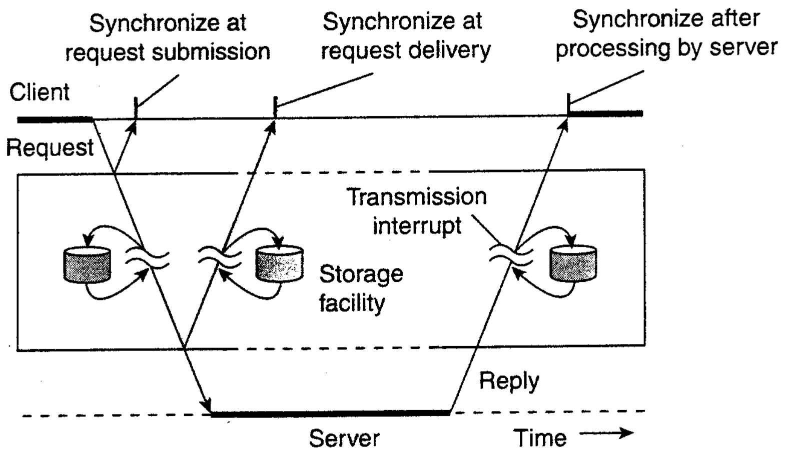

Synchronize at request submission
Synchronize at request delivery
Synchronize after processing by server
Client
Request
Transmission interrupt
Storage facility
Reply
Server
Time

Figure 4-4. Viewing middleware as an intermediate (distributed) service in application-level communication.

An electronic mail system is a typical example in which communication is persistent. With persistent communication, a message that has been submitted for transmission is stored by the communication middleware as long as it takes to deliver it to the receiver. In this case, the middleware will store the message at one or several of the storage facilities shown in Fig. 4-4. As a consequence, it is not necessary for the sending application to continue execution after submitting the message. Likewise, the receiving application need not be executing when the message is submitted.

In contrast, with transient communication, a message is stored by the communication system only as long as the sending and receiving application are executing. More precisely, in terms of Fig. 4-4, the middleware cannot deliver a message due to a transmission interrupt, or because the recipient is currently not active, it will simply be discarded. Typically, all transport-level communication services offer only transient communication. In this case, the communication system consists traditional store-and-forward routers. If a router cannot deliver a message to the next one or the destination host, it will simply drop the message.

Besides being persistent or transient, communication can also be asynchronous or synchronous. The characteristic feature of asynchronous communication is that a sender continues immediately after it has submitted its message for transmission. This means that the message is (temporarily) stored immediately by the middleware upon submission. With synchronous communication, the sender is blocked until its request is known to be accepted. There are essentially three points where synchronization can take place. First, the sender may be blocked until the middleware notifies that it will take over transmission of the request. Second, the sender may synchronize until its request has been delivered to the intended recipient. Third, synchronization may take place by letting the sender wait until its request has been fully processed, that is, up the time that the recipient returns a response.

Various combinations of persistence and synchronization occur in practice. Popular ones are persistence in combination with synchronization at request submission, which is a common scheme for many message-queuing systems, which we discuss later in this chapter. Likewise, transient communication with synchronization after the request has been fully processed is also widely used. This scheme corresponds with remote procedure calls, which we also discuss below.

Besides persistence and synchronization, we should also make a distinction between discrete and streaming communication. The examples so far all fall in the category of discrete communication: the parties communicate by messages, each message forming a complete unit of information. In contrast, streaming involves sending multiple messages, one after the other, where the messages are related to each other by the order they are sent, or because there is a temporal relationship. We return to streaming communication extensively below.

4.2 REMOTE PROCEDURE CALL

Many distributed systems have been based on explicit message exchange between processes. However, the procedures send and receive do not conceal communication at all, which is important to achieve access transparency in distributed systems. This problem has long been known, but little was done about it until a paper by Birrell and Nelson (1984) introduced a completely different way of handling communication. Although the idea is refreshingly simple (once someone has thought of it). the implications are often subtle. In this section we will examine the concept, its implementation, its strengths, and its weaknesses.

In a nutshell, what Birrell and Nelson suggested was allowing programs to call procedures located on other machines. When a process on machine *A* calls' a procedure on machine *B,* the calling process on *A* is suspended, and execution of the called procedure takes place on *B.* Information can be transported from the caller to the callee in the parameters and can come back in the procedure result.. No message passing at all is visible to the programmer. This method is known as **Remote Procedure Call,** or often just RPC.

While the basic idea sounds simple and elegant, subtle problems exist.. To start with, because the calling and called procedures run on different machines, they execute in different address spaces, which causes complications. Parameters and results also have to be passed, which can be complicated, especially if the machines are not identical. Finally, either or both machines can crash and each of the possible failures causes different problems. Still, most of these can be dealt with, and RPC is a widely-used technique that underlies many distributed systems.

4.2.1 Basic RPC Operation

We first start with discussing conventional procedure calls, and then explain how the call itself can be split into a client and server part that are each executed on different machines.

Conventional Procedure Call

To understand how RPC works, it is important first to fully understand how a conventional (i.e., single machine) procedure call works. Consider a call in C like

```
count =tead(td, but, nbytes);
```

where *fd* is an .integer indicating a file, *buf* is an array of characters into which data are read, and *nbytes* is another integer telling how many bytes to read. If the call is made from the main program, the stack will be as shown in Fig. 4-5(a) before the call. To make the call, the caller pushes the parameters onto the stack in order, last one first, as shown in Fig. 4-5(b). (The reason that C compilers push the parameters in reverse order has to do with *printj--by* doing so, *print!* can always locate its first parameter, the format string.) After the read procedure has finished running, it puts the return value in a register, removes the return address, and transfers control back to the caller. The caller then removes the parameters from the stack, returning the stack to the original state it had before the call.

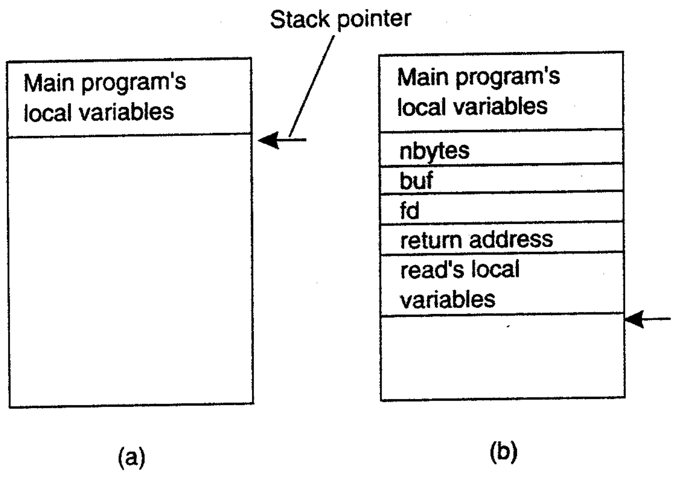


Figure 4-5. (a) Parameter passing in a local procedure call: the stack before the call to read. (b) The stack while the called procedure is active.

Several things are worth noting. For one, in C, parameters can be call-by-value or call-by-reference. A value parameter, such as *fd* or *nbytes,* is simply copied to the stack as shown in Fig. 4-5(b). To the called procedure, a value parameter is just an initialized local variable. The called procedure may modify it, but such changes do not affect the original value at the calling side.

A reference parameter in C is a pointer to a variable (i.e., the address of the variable), rather than the value of the variable. In the call to read. the second parameter is a reference parameter because arrays are always passed by reference in C. What is actually pushed onto the stack is the address of the character array. If the called procedure uses this parameter to store something into the character array, it *does* modify the array in the calling procedure. The difference between call-by-value and call-by-reference is quite important for RPC, as we shall see.

One other parameter passing mechanism also exists, although it is not used in C. It is called call-by-copy/restore. It consists of having the variable copied to the stack by the caller, as in call-by-value, and then copied back after the call, overwriting the caller's original value. Under most conditions, this achieves exactly the same effect as call-by-reference, but in some situations. such as the same parameter being present multiple times in the parameter list. the semantics are different. The call-by-copy/restore mechanism is not used in many languages.

The decision of which parameter passing mechanism to use is normally made by the language designers and is a fixed property of the language. Sometimes it depends on the data type being passed. In C, for example, integers and other scalar types are always passed by value, whereas arrays are always passed by reference, as we have seen. Some Ada compilers use copy/restore for in out parameters, but others use call-by-reference. The language definition permits either choice, which makes the semantics a bit fuzzy.

Client and Server Stubs

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In other words, we want RPC to be transparent-the calling procedure should not be aware that the called procedure is executing on a different machine or vice versa. Suppose that a program needs to read some data from a file. The programmer puts a call to read in the code to get the data. In a traditional (single-processor) system, the read routine is extracted from the library by the linker and inserted into the object program. It is a short procedure, which is generally implemented by calling an equivalent read system call. In other words, the read procedure is a kind of interface between the user code and the local operating system.

Even though read does a system call, it is called in the usual way, by pushing the parameters onto the stack, as shown in Fig. 4-5(b). Thus the programmer does not know that read is actually doing something fishy.

RPC achieves its transparency in an analogous way. When read is actually a remote procedure (e.g., one that will run on the file server's machine), a different version of read, called a client stub, is put into the library. Like the original one, it, too, is called using the calling sequence of Fig. 4-5(b). Also like the original one, it too, does a call to the local operating system. Only unlike the original one, it does not ask the operating system to give it data. Instead, it packs the parameters into a message and requests that message to be sent to the server as illustrated in Fig. 4-6. Following the call to send, the client stub calls receive, blocking itself until the reply comes back.

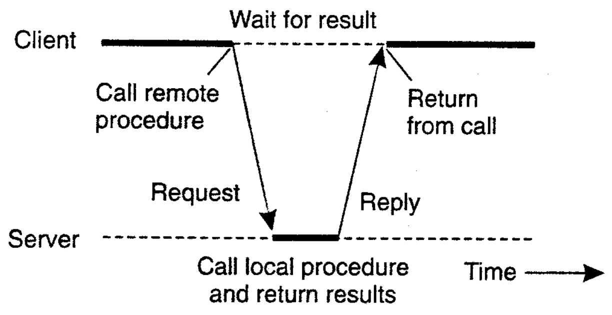


Figure 4-6. Principle of RPC between a client and server program.

When the message arrives at the server, the server's operating system passes it up to a server stub. A server stub is the server-side equivalent of a client stub: it is a piece of code that transforms requests coming in over the network into local procedure calls. Typically the server stub will have called receive and be blocked waiting for incoming messages. The server stub unpacks the parameters from the message and then calls the server procedure in the usual way (i.e., as in Fig. 4-5). From the server's point of view, it is as though it is being called directly by the client-the parameters and return address are all on the stack where they belong and nothing seems unusual. The server performs its work and then returns the result to the caller in the usual way. For example, in the case of read, the server will fill the buffer, pointed to by the second parameter, with the data. This buffer will be internal to the server stub.

When the server stub gets control back after the call has completed, it packs the result (the buffer) in a message and calls send to return it to the client. After that, the server stub usually does a call to receive again, to wait for the next incoming request.

When the message gets back to the client machine, the client's operating system sees that it is addressed to the client process (or actually the client stub, but the operating system cannot see the difference). The message is copied to the waiting buffer and the client process unblocked. The client stub inspects the message, unpacks the result, copies it to its caller, and returns in the usual way. When the caller gets control following the call to read, all it knows is that its data are available. It has no idea that the work was done remotely instead of by the local operating system.

This blissful ignorance on the part of the client is the beauty of the whole scheme. As far as it is concerned, remote services are accessed by making ordinary (i.e., local) procedure calls, not by calling send and receive. All the details of the message passing are hidden away in the two library procedures, just as the details of actually making system calls are hidden away in traditional libraries.

To summarize, a remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's as sends the message to the remote as.
4. The remote as gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local as.
8. The server's as sends the message to the client's as.
9. The client's as gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

The net effect of all these steps is to convert the local call by the client procedure to the client stub, to a local call to the server procedure without either client or server being aware of the intermediate steps or the existence of the network.

4.2.2 Parameter Passing

The function of the client stub is to take its parameters, pack them into a message, and send them to the server stub. While this sounds straightforward, it is not quite as simple as it at first appears. In this section we will look at some of the issues concerned with parameter passing in RPC systems.

Passing Value Parameters

Packing parameters into a message is called parameter marshaling. As a very simple example, consider a remote procedure, add(i, j), that takes two integer parameters *i* and *j* and returns their arithmetic sum as a result.. (As a practical matter, one would not normally make such a simple procedure remote due to the overhead, but as an example it will do.) The call to add, is shown in the left-hand portion (in the client process) in Fig. 4-7. The client stub takes its two parameters and puts them in a message as indicated, It also puts the name or number of the procedure to be called in the message because the server might support several different calls, and it has to be told which one is required.

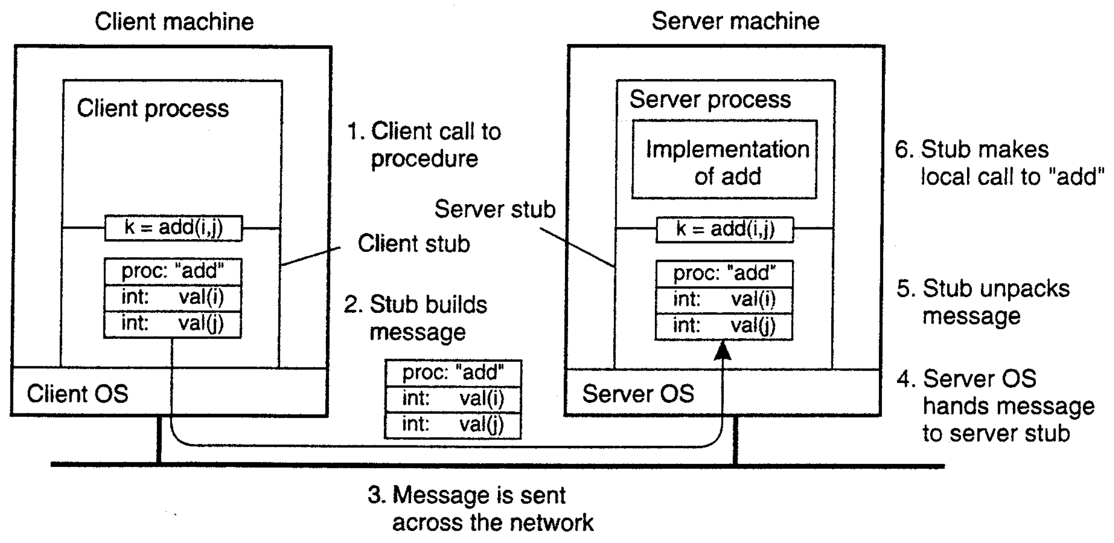


Figure 4-7. The steps involved in a doing a remote computation through RPC.

When the message arrives at the server, the stub examines the message to see which procedure is needed and then makes the appropriate call. If the server also supports other remote procedures, the server stub might have a switch statement in it to select the procedure to be called, depending on the first field of the message. The actual call from the stub to the server looks like the original client call, except that the parameters are variables initialized from the incoming message.

When the server has finished, the server stub gains control again. It takes the result sent back by the server and packs it into a message. This message is sent back back to the client stub. which unpacks it to extract the result and returns the value to the waiting client procedure.

As long as the client and server machines are identical and all the parameters and results are scalar types. such as integers, characters, and Booleans, this model works fine. However, in a large distributed system, it is common that multiple machine types are present. Each machine often has its own representation for numbers, characters, and other data items. For example, IRM mainframes use the EBCDIC character code, whereas IBM personal computers use ASCII. As a consequence, it is not possible to pass a character parameter from an IBM PC client to an IBM mainframe server using the simple scheme of Fig. 4-7: the server will interpret the character incorrectly.

Similar problems can occur with the representation of integers (one's complement versus two's complement) and floating-point numbers. In addition, an even more annoying problem exists because some machines, such as the Intel Pentium, number their bytes from right to left, whereas others, such as the Sun SPARC, number them the other way. The Intel format is called little endian and the SPARC format is called big endian, after the politicians in *Gulliver's Travels* who went to war over which end of an egg to break (Cohen, 1981). As an example, consider a procedure with two parameters, an integer and a four-character string. Each parameter requires one 32-bit word. Fig.4-8(a) shows what the parameter portion of a message built by a client stub on an Intel Pentium might look like, The first word contains the integer parameter, 5 in this case, and the second contains the string "JILL."

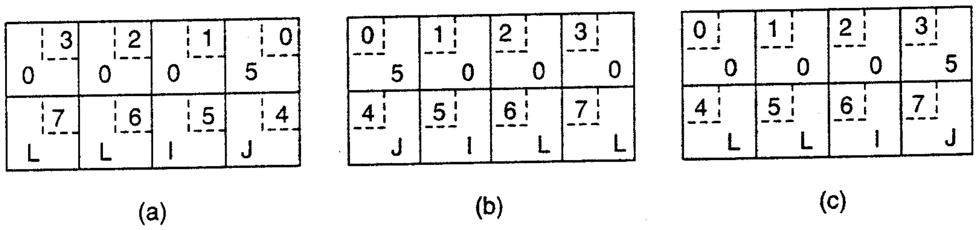


Figure 4-8. (a) The original message on the Pentium. (b) The message after receipt on the SPARe. (c) The message after being inverted. The little numbers in boxes indicate the address of each byte.

Since messages are transferred byte for byte (actually, bit for bit) over the network, the first byte sent is the first byte to arrive. In Fig. 4-8(b) we show what the message of Fig. 4-8(a) would look like if received by a SPARC, which numbers its bytes with byte 0 at the left (high-order byte) instead of at the right (low-order byte) as do all the Intel chips. When the server stub reads the parameters at addresses 0 and 4, respectively, it will find an integer equal to 83,886,080 (5×2^{24}) and a string "JILL".

One obvious, but unfortunately incorrect, approach is to simply invert the bytes of each word after they are received, leading to Fig. 4-8(c). Now the integer is 5 and the string is "LLIJ". The problem here is that integers are reversed by the different byte ordering, but strings are not. Without additional information about what is a string and what is an integer, there is no way to repair the damage.

Passing Reference Parameters

We now come to a difficult problem: How are pointers, or in general, references passed? The answer is: only with the greatest of difficulty, if at all. Remember that a pointer is meaningful only within the address space of the process in which it is being used. Getting back to our read example discussed earlier, if the second parameter (the address of the buffer) happens to be 1000 on the client, one cannot just pass the number 1000 to the server and expect it to work. Address 1000 on the server might be in the middle of the program text.

One solution is just to forbid pointers and reference parameters in general. However, these are so important that this solution is highly undesirable. In fact, it is not necessary either. In the read example, the client stub knows that the second parameter points to an array of characters. Suppose, for the moment, that it also knows how big the array is. One strategy then becomes apparent: copy the array into the message and send it to the server. The server stub can then call the server with a pointer to this array, even though this pointer has a different numerical value than the second parameter of read has. Changes the server makes using the pointer (e.g., storing data into it) directly affect the message buffer inside the server stub. When the server finishes, the original message can be sent back to the client stub, which then copies it back to the client. In effect, call-by-reference has been replaced by copy/restore. Although this is not always identical, it frequently is good enough.

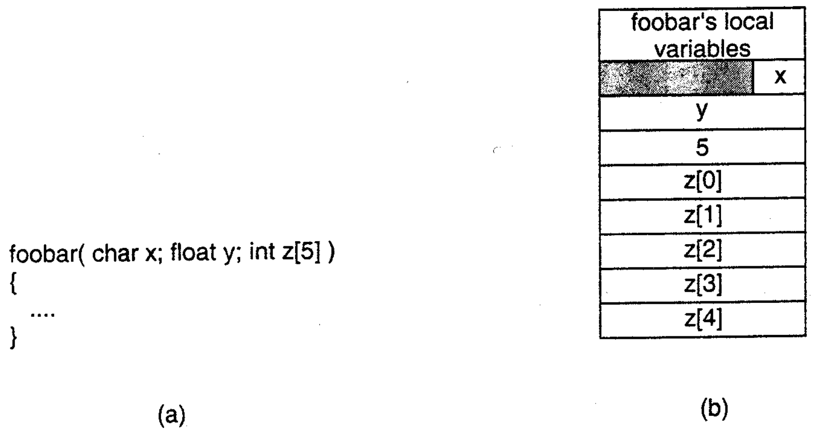
One optimization makes this mechanism twice as efficient. If the stubs know whether the buffer is an input parameter or an output parameter to the server, one of the copies can be eliminated. If the array is input to the server (e.g., in a call to write) it need not be copied back. If it is output, it need not be sent over in the first place.

As a final comment, it is worth noting that although we can now handle pointers to simple arrays and structures, we still cannot handle the most general case of a pointer to an arbitrary data structure such as a complex graph. Some systems attempt to deal with this case by actually passing the pointer to the server stub and generating special code in the server procedure for using pointers. For example, a request may be sent back to the client to provide the referenced data.

Parameter Specification and Stub Generation

From what we have explained so far, it is clear that hiding a remote procedure call requires that the caller and the callee agree on the format of the messages they exchange, and that they follow the same steps when it comes to, for example, passing complex data structures. In other words, both sides in an RPC should follow the same protocol or the RPC will not work correctly.

As a simple example, consider the procedure of Fig. 4-9(a). It has three parameters, a character, a floating-point number, and an array of five integers. Assuming a word is four bytes, the RPC protocol might prescribe that we should transmit a character in the rightmost byte of a word (leaving the next 3 bytes empty), a float as a whole word, and an array asa group of words equal to the array length, preceded by a word giving the length, as shown in Fig. 4-9(b). Thus given these rules, the client stub for foobar knows that it must use the format of Fig. 4-9(b), and the server stub knows that incoming messages for foobar will have the format of Fig. 4-9(b).



```
foobar( char x; float y; int z[5] )
{
  ....
}
```

Figure 4-9. (a) A procedure. (b) The corresponding message.

Defining the message format is one aspect of an RPC protocol, but it is not sufficient. What we also need is the client and the server to agree on the representation of simple data structures, such as integers, characters, Booleans, etc. For example, the protocol could prescribe that integers are represented in two's complement, characters in 16-bit Unicode, and floats in the IEEE standard #754 format, with everything stored in little endian. With this additional information, messages can be unambiguously interpreted.

With the encoding rules now pinned down to the last bit, the only thing that remains to be done is that the caller and callee agree on the actual exchange of messages. For example, it may be decided to use a connection-oriented transport service such as TCPIIP. An alternative is to use an unreliable datagram service and let the client and server implement an error control scheme as part of the RPC protocol. In practice, several variants exist.

Once the RPC protocol has been fully defined, the client and server stubs need to be implemented. Fortunately, stubs for the same protocol but different procedures normally differ only in their interface to the applications. An interface consists of a collection of procedures that can be called by a client, and which are implemented by a server. An interface is usually available in the same programing language as the one in which the client or server is written (although this is strictly speaking, not necessary). To simplify matters, interfaces are often specified by means of an Interface Definition Language (IDL). An interface specified in such an IDL is then subsequently compiled into a client stub and a server stub, along with the appropriate compile-time or run-time interfaces.

Practice shows that using an interface definition language considerably simplifies client-server applications based on RPCs. Because it is easy to fully generate client and server stubs, all RPC-based middleware systems offer an IDL to support application development. In some cases, using the IDL is even mandatory, as we shall see in later chapters.

4.2.3 Asynchronous RPC

As in conventional procedure calls, when a client calls a remote procedure, the client will block until a reply is returned. This strict request-reply behavior is unnecessary when there is no result to return, and only leads to blocking the client while it could have proceeded and have done useful work just after requesting the remote procedure to be called. Examples of where there is often no need to wait for a reply include: transferring money from one account to another, adding entries into a database, starting remote services, batch processing, and so on.

To support such situations, RPC systems may provide facilities for what are called asynchronous RPCs, by which a client immediately continues after issuing the RPC request. With asynchronous RPCs, the server immediately sends a reply back to the client the moment the RPC request is received, after which it calls the requested procedure. The reply acts as an acknowledgment to the client that the server is going to process the RPC. The client will continue without further blocking as soon as it has received the server's acknowledgment. Fig. 4-lO(b) shows how client and server interact in the case of asynchronous RPCs. For comparison, Fig. 4-10(a) shows the normal request-reply behavior.

Asynchronous RPCs can also be useful when a reply will be returned but the client is not prepared to wait for it and do nothing in the meantime. For example, a client may want to prefetch the network addresses of a set of hosts that it expects to contact soon. While a naming service is collecting those addresses, the client may want to do other things. In such cases, it makes sense to organize the communication between the client and server through two asynchronous RPCs, as shown in Fig. 4-11. The client first calls the server to hand over a list of host names that should be looked up, and continues when the server has acknowledged the receipt of that list. The second call is done by the server, who calls the client to hand over the addresses it found. Combining two asynchronous RPCs is sometimes also referred to as a deferred synchronous RPC.

It should be noted that variants of asynchronous RPCs exist in which the client continues executing immediately after sending the request to the server. In

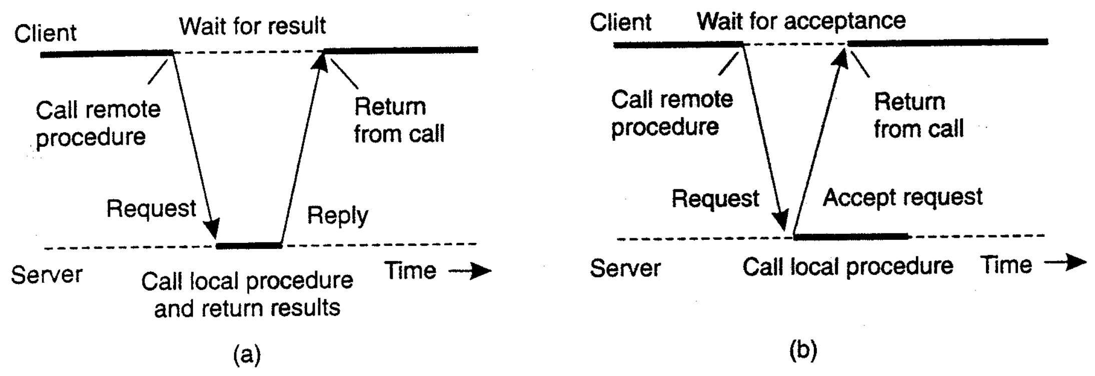


Figure 4-10. (a) The interaction between client and server in a traditional RPc. (b) The interaction using asynchronous RPc.

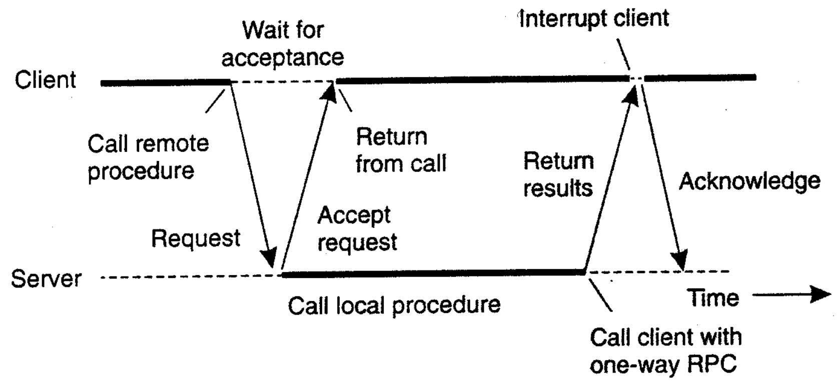


Figure 4-11. A client and server interacting through two asynchronous RPCs.

other words, the client does not wait for an acknowledgment of the server's acceptance of the request. We refer to such RPCs as one-way RPCs. The problem with this approach is that when reliability is not guaranteed, the client cannot know for sure whether or not its request will be processed. We return to these matters in Chap. 8. Likewise, in the case of deferred synchronous RPC, the client may poll the server to see whether the results are available yet instead of letting the server calling back the client.

4.2.4 Example: DCE RPC

Remote procedure calls have been widely adopted as the basis of middleware and distributed systems in general. In this section, we take a closer look at one specific RPC system: the Distributed Computing Environment (DeE), which was developed by the Open Software Foundation (OSF), now called The Open Group. DCE RPC is not as popular as some other RPC systems, notably Sun RPC. However, DCE RPC is nevertheless representative of other RPC systems, and its specifications have been adopted in Microsoft's base system for distributed computing, DCOM (Eddon and Eddon,]998). We start with a brief introduction to DCE, after which we consider the principal workings of DCE RPC. Detailed technical information on how to develop RPC-based applications can be found in Stevens (1999).

Introduction to DCE

DCE is a true middleware system in that it is designed to execute as a layer of abstraction between existing (network) operating systems and distributed applications. Initially designed for UNIX, it has now been ported to all major operating systems including VMS and Windows variants, as well as desktop operating systems. The idea is that the customer can take a collection of existing machines, add the DCE software, and then be able to run distributed applications, all without disturbing existing (nondistributed) applications. Although most of the DCE package runs in user space, in some configurations a piece (part of the distributed file system) must be added to the kernel. The Open Group itself only sells source code, which vendors integrate into their systems.

The programming model underlying all of DCE is the client-server model, which was extensively discussed in the previous chapter. User processes act as clients to access remote services provided by server processes. Some of these services are part of DCE itself, but others belong to the applications and are written by the applications programmers. All communication between clients and servers takes place by means of RPCs.

There are a number of services that form part of DCE itself. The distributed file service is a worldwide file system that provides a transparent. way of accessing any file in the system in the same way. It can either be built on top of the hosts' native file systems or used instead of them. The directory service is used to keep track of the location of all resources in the system. These resources include machines, printers, servers, data, and much more, and they may be distributed geographically over the entire world. The directory service allows a process to ask for a resource and not have to be concerned about where it is, unless the process cares. The security service allows resources of all kinds to be protected, so access can be restricted to authorized persons. Finally, the distributed time service is a service that attempts to keep clocks on the different machines globally synchronized. As we shall see in later chapters, having some notion of global time makes it much easier to ensure consistency in a distributed system.

Goals of DCE RPC

The goals of the DCE RPC system are relatively traditional. First and foremost, the RPC system makes it possible for a client to access a remote service by simply calling a local procedure. This interface makes it possible for client (i.e., application) programs to be written in a simple way, familiar to most programmers. It also makes it easy to have large volumes of existing code run in a distributed environment with few, if any, changes.

It is up to the RPC system to hide all the details from the clients, and, to some extent, from the servers as well. To start with, the RPC system can automatically locate the correct server, and subsequently set up the communication between client and server software (generally called binding). It can also handle the message transport in both directions, fragmenting and reassembling them as needed (e.g., if one of the parameters is a large array). Finally, the RPC system can automatically handle data type conversions between the client and the server, even if they run on different architectures and have a different byte ordering;

As a consequence of the RPC system's ability to hide the details, clients and servers are highly independent of one another. A client can be written in Java and a server in C, or vice versa. A client and server can run on different hardware platforms and use different operating systems. A variety of network protocols and data representations are also supported, all without any intervention from the client or server.

Writing a Client and a Server

The DCE RPC system consists of a number of components, including languages, libraries, daemons, and utility programs, among others. Together these make it possible to write clients and servers. In this section we will describe the pieces and how they fit together. The entire process of writing and using an RPC client and server is summarized in Fig. 4-12.

In a client-server system, the glue that holds everything together is the interface definition, as specified in the Interface Definition Language, or IDL. It permits procedure declarations in a form closely resembling function prototypes in ANSI C. IDL files can also contain type definitions, constant declarations, and other information needed to correctly marshal parameters and unmarshal results. Ideally, the interface definition should also contain a formal definition of what the procedures do, but such a definition is beyond the current state of the art, so the interface definition just defines the syntax of the calls, not their semantics. At best the writer can add a few comments describing what the procedures do.

A crucial element in every IDL file is a globally unique identifier for the specified interface. The client sends this identifier in the first RPC message and the server verifies that it is correct. In this way, if a client inadvertently tries to bind to the wrong server, or even to an older version of the right server, the server will detect the error and the binding will not take place.

Interface definitions and unique identifiers are closely related in DCE. As illustrated in Fig. 4-12, the first step in writing a client/server application is usually calling the *uuidgen* program, asking it to generate a prototype IDL file containing an interface identifier guaranteed never to be used again in any interface

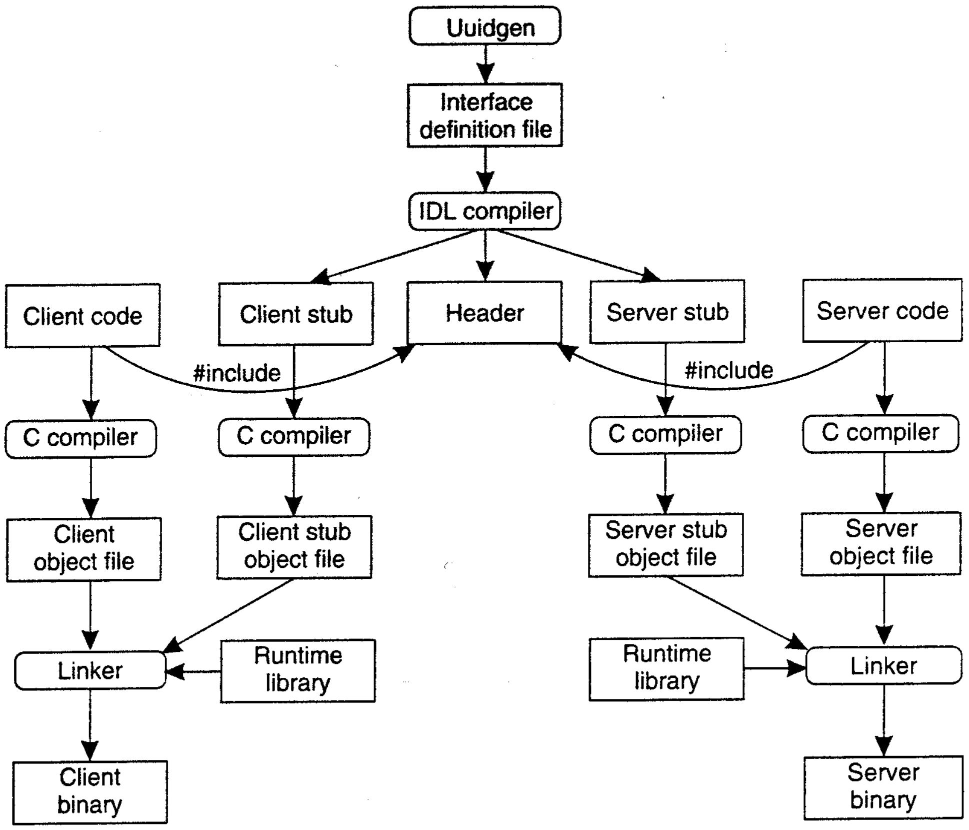


Figure 4-12. The steps in writing a client and a server in DCE RPC.

generated anywhere by *uuidgen.* Uniqueness is ensured by encoding in it the location and time of creation. It consists of a 128-bit binary number represented in the IDL file as an ASCII string in hexadecimal.

The next step is editing the IDL file, filling in the names of the remote procedures and their parameters. It is worth noting that RPC is not totally transparent-for example, the client and server cannot share global variables-but the IDL rules make it impossible to express constructs that are not supported.

When the IDL file is complete, the IDL compiler is called to process it. The output of the IDL compiler consists of three files:

1. A header file (e.g., *interface.h,* in C terms).
2. The client stub.
3. The server stub.

The header file contains the unique identifier, type definitions, constant definitions, and function prototypes. It should be included (using *#include)* in both the client and server code. The client stub contains the actual procedures that the client program will call. These procedures are the ones responsible for collecting and packing the parameters into the outgoing message and then calling the runtime system to send it. The client stub also handles unpacking the reply and returning values to the client. The server stub contains the procedures called by the runtime system on the server machine when an incoming message arrives. These, in turn, call the actual server procedures that do the work.

The next step is for the application writer to write the client and server code. Both of these are then compiled, as are the two stub procedures. The resulting client code and client stub object files are then linked with the runtime library to produce the executable binary for the client. Similarly, the server code and server stub are compiled and linked to produce the server's binary. At runtime, the client and server are started so that the application is actually executed as well.

Binding a Client to a Server

To allow a client to call a server, it is necessary that the server be registered and prepared to accept incoming calls. Registration of a server makes it possible for a client to locate the server and bind to it. Server location is done in two steps:

1. Locate the server's machine.
2. Locate the server (i.e., the correct process) on that machine.

The second step is somewhat subtle. Basically, what it comes down to is that to communicate with a server, the client needs to know an end point, on the server's machine to which it can send messages. An end point (also commonly known as a port) is used by the server's operating system to distinguish incoming messages for different processes. In DCE, a table of *(server, end point)pairs* is maintained on each server machine by a process called the DCE daemon. Before it becomes available for incoming requests, the server must ask the operating system for an end point. It then registers this end point with the DCE daemon. The DCE daemon records this information (including which protocols the server speaks) in the end point table for future use.

The server also registers with the directory service by providing it the network address of the server's machine and a name under which the server can be looked up. Binding a client to a server then proceeds as shown in Fig. 4-13.

Let us assume that the client wants to bind to a video server that is locally known under the name */local/multimedia/video/movies* .. It passes this name to the directory server, which returns the network address of the machine running the video server. The client then goes to the DCE daemon on that machine (which has a well-known end point), and asks it to look up the end point of the video server in its end point table. Armed with this information, the RPC can now take place. On subsequent RPCs this lookup is not needed. DCE also gives clients the ability to do more sophisticated searches for a suitable server when that is needed. Secure RPC is also an option where confidentiality or data integrity is crucial.

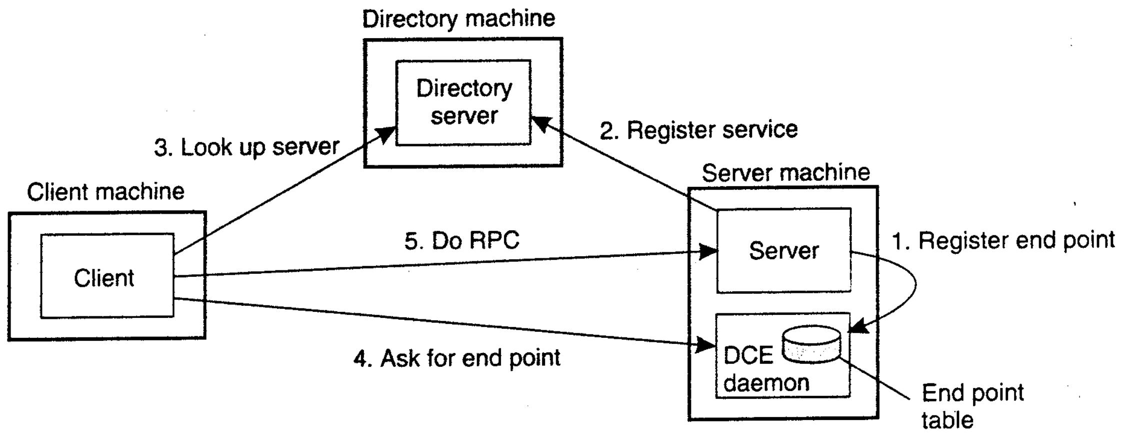


Figure 4-13. Client-to-server binding in DCE.

Performing an RPC

The actual RPC is carried out transparently and in the usual way. The client stub marshals the parameters to the runtime library for transmission using the protocol chosen at binding time. When a message arrives at the server side, it is routed to the correct server based on the end point contained in the incoming message. The runtime library passes the message to the server stub, which unmarshals the parameters and calls the server. The reply goes back by the reverse route.

DCE provides several semantic options. The default is at-most-once operation, in which case no call is ever carried out more than once, even in the face of system crashes. In practice, what this means is that if a server crashes during, an RPC and then recovers quickly, the client does not repeat the operation, for fear that it might already have been carried out once.

Alternatively, it is possible to mark a remote procedure as idempotent (in the IDL file), in which case it can be repeated multiple times without harm. For example, reading a specified block from a file can be tried over and over until it succeeds. When an idempotent RPC fails due to a server crash. the client can wait until the server reboots and then try again. Other semantics are also available (but rarely used), including broadcasting the RPC to all the machines on the local network, We return to RPC semantics in Chap. 8, when discussing RPC in the presence of failures.

4.3 MESSAGE-ORIENTED COMMUNICATION

Remote procedure calls and remote object invocations contribute to hiding communication in distributed systems, that is, they enhance access transparency. Unfortunately, neither mechanism is always appropriate. In particular, when it cannot be assumed that the receiving side is executing at the time a request is issued, alternative communication services are needed. Likewise, the inherent synchronous nature of RPCs, by which a client is blocked until its request has been processed, sometimes needs to be replaced by something else.

That something else is messaging. In this section we concentrate on message-oriented communication in distributed systems by first taking a closer look at what exactly synchronous behavior is and what its implications are. Then, we discuss messaging systems that assume that parties are executing at the time of communication. Finally, we will examine message-queuing systems that allow processes to exchange information, even if the other party is not executing at the time communication is initiated.

4.3.1 Message-Oriented Transient Communication

Many distributed systems and applications are built directly on top of the simple message-oriented model offered by the transport layer. To better understand and appreciate the message-oriented systems as part of middleware solutions, we first discuss messaging through transport-level sockets.

Berkeley Sockets

Special attention has been paid to standardizing the interface of the transport layer to allow programmers to make use of its entire suite of (messaging) protocols through a simple set of primitives. Also, standard interfaces make it easier to port an application to a different machine.

As an example, we briefly discuss the sockets interface as introduced in the 1970s in Berkeley UNIX. Another important interface is XTI, which stands for the X10pen Transport Interface, formerly called the Transport Layer Interface (TLI), and developed by AT&T. Sockets and XTI are very similar in their model of network programming, but differ in their set of primitives.

Conceptually, a socket is a communication end point to which an application can write data that are to be sent out over the underlying network, and from which incoming data can be read. A socket forms an abstraction over the actual communication end point that is used by the local operating system for a specific transport protocol. In the following text, we concentrate on the socket primitives for TCP, which are shown in Fig. 4-14.

Servers generally execute the first four primitives, normally in the order given. When calling the socket primitive, the caller creates a new communication end point for a specific transport protocol. Internally, creating a communication end point means that the local operating system reserves resources to accommodate sending and receiving messages for the specified protocol.

The bind primitive associates a local address with the newly-created socket. For example, a server should bind the IP address of its machine together with a (possibly well-known) port number to a socket. Binding tells the operating system that the server wants to receive messages only on the specified address and port.

Primitive	Meaning
Socket	Create a new communication end point
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Figure 4-14. The socket primitives for TCPIIP.

The listen primitive is called only in the case of connection-oriented communication. It is a nonblocking call that allows the local operating system to reserve enough buffers for a specified maximum number of connections that the caller is willing to accept.

A call to accept blocks the caller until a connection request arrives. When a request arrives, the local operating system creates a new socket with the same properties as the original one, and returns it to the caller. This approach will allow the server to, for example, fork off a process that will subsequently handle the actual communication through the new connection. The server, in the meantime, can go back and wait for another connection request on the original socket.

Let us now take a look at the client side. Here, too, a socket must first be created using the socket primitive, but explicitly binding the socket to a local address is not necessary, since the operating system can dynamically allocate a port when the connection is set up. The connect primitive requires that the caller specifies the transport-level address to which a connection request is to be sent. The client is blocked until a connection has been set up successfully, after which both sides can start exchanging information through the send and receive primitives. Finally, closing a connection is symmetric when using sockets, and is established by having both the client and server call the close primitive. The general pattern followed by a client and server for connection-oriented communication using sockets is shown in Fig. 4-15. Details about network programming using sockets and other interfaces in a UNIX environment can be found in Stevens (1998).

The Message-Passing Interface (MPI)

With the advent of high-performance multicomputers, developers have been looking for message-oriented primitives that would allow them to easily write highly efficient applications. This means that the primitives should be at a convenient level of abstraction (to ease application development), and that their

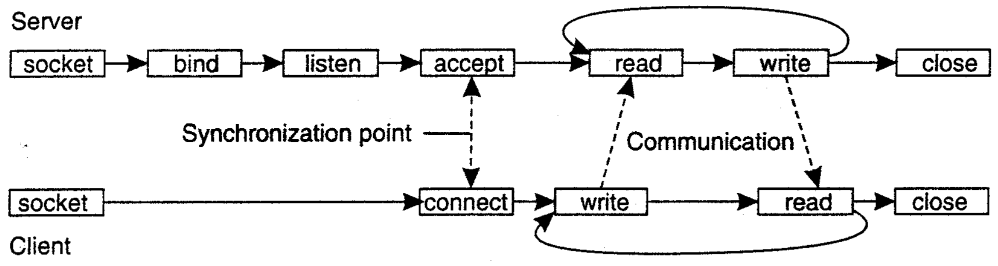


Figure 4-15. Connection-oriented communication pattern using sockets.

implementation incurs only minimal overhead. Sockets were deemed insufficient for two reasons. First, they were at the wrong level of abstraction by supporting only simple send and receive primitives. Second, sockets had been designed to communicate across networks using general-purpose protocol stacks such as TCPIIP. They were not considered suitable for the proprietary protocols developed for high-speed interconnection networks, such as those used in high-performance server clusters. Those protocols required an 'interface that could handle more advanced features, such as different forms of buffering and synchronization.

The result was that most interconnection networks and high-performance multicomputers were shipped with proprietary communication libraries. These libraries offered a wealth of high-level and generally efficient communication primitives. Of course, all libraries were mutually incompatible, so that application developers now had a portability problem.

The need to be hardware and platform independent eventually led to the definition of a standard for message passing, simply called the Message-Passing Interface or MPI. MPI is designed for parallel applications and as such is tailored to transient communication. It makes direct use of the underlying network.. Also, it assumes that serious failures such as process crashes or network partitions are fatal and do not require automatic recovery.

MPI assumes communication takes place within a knowngroup of processes. Each group is assigned an identifier. Each process within a group is also assigned a (local) identifier. A *(group/D, process/D)* pair therefore uniquely identifies the source or destination of a message, and is used instead of a transport-level address. There may be several, possibly overlapping groups of processes involved in a computation and that are all executing at the same time.

At the core of MPI are messaging primitives to support transient communication, of which the most intuitive ones are summarized in Fig. 4-16.

Transient asynchronous communication is supported by means of the MPI_bsend primitive. The sender submits a message for transmission, which is generally first copied to a local buffer in the MPI runtime system. When the message has been copied. the sender continues. The local MPI runtime system will remove the message from its local buffer and take care of transmission as soon as a receiver has called a receive primitive.

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

Figure 4-16. Some of the most intuitive message-passing primitives of MPI.

There is also a blocking send operation, called MPLsend, of which the semantics are implementation dependent. The primitive MPLsend may either block the caller until the specified message has been copied to the MPI runtime system at the sender's side, or until the receiver has initiated a receive operation. Synchronous communication by which the sender blocks until its request is accepted for further processing is available through the MPI~ssend primitive. Finally, the strongest form of synchronous communication is also supported: when a sender calls MPLsendrecv, it sends a request to the receiver and blocks until the latter returns a reply. Basically, this primitive corresponds to a normal RPC.

Both MPLsend and MPLssend have variants that avoid copying messages from user buffers to buffers internal to the local MPI runtime system. These variants correspond to a form of asynchronous communication. With MPI_isend, a sender passes a pointer to the message after which the MPI runtime system takes care of communication. The sender immediately continues. To prevent overwriting the message before communication completes, MPI offers primitives to check for completion, or even to block if required. As with MPLsend, whether the message has actually been transferred to the receiver or that it has merely been copied by the local MPI runtime system to an internal buffer is left unspecified.

Likewise, with MPLissend, a sender also passes only a pointer to the :MPI runtime system. When the runtime system indicates it has processed the message, the sender is then guaranteed that the receiver has accepted the message and is now working on it.

The operation MPLrecv is called to receive a message; it blocks the caller until a message arrives. There is also an asynchronous variant, called MPLirecv, by which a receiver indicates that is prepared to accept a message. The receiver can check whether or not a message has indeed arrived, or block until one does.

The semantics of MPI communication primitives are not always straightforward, and different primitives can sometimes be interchanged without affecting the correctness of a program. The official reason why so many different forms of communication are supported is that it gives implementers of MPI systems enough possibilities for optimizing performance. Cynics might say the committee could not make up its collective mind, so it threw in everything. MPI has been designed for high-performance parallel applications, which makes it easier to understand its diversity in different communication primitives.

More on MPI can be found in Gropp et al. (1998b) The complete reference in which the over 100 functions in MPI are explained in detail, can be found in Snir et al. (1998) and Gropp et al. (1998a)

4.3.2 Message-Oriented Persistent Communication

We now come to an important class of message-oriented middle ware services, generally known as message-queuing systems, or just Message-Oriented Middleware (MOM). Message-queuing systems provide extensive support for persistent asynchronous communication. The essence of these systems is that they offer intermediate-term storage capacity for messages, without requiring either the sender or receiver to be active during message transmission. An important difference with Berkeley sockets and MPI is that message-queuing systems are typically targeted to support message transfers that are allowed to take minutes instead of seconds or milliseconds. We first explain a general approach to message-queuing systems, and conclude this section by comparing them to more traditional systems, notably the Internet e-mail systems.

Message-Queuing Model

The basic idea behind a message-queuing system is that applications communicate by inserting messages in specific queues. These messages are forwarded over a series of communication servers and are eventually delivered to the destination, even if it was down when the message was sent. In practice, most communication servers are directly connected to each other. In other words, a message is generally transferred directly to a destination server. In principle, each application has its own private queue to which other applications can send messages. A queue can be read only by its associated application, but it is also possible for multiple applications to share a single queue.

An important aspect of message-queuing systems is that a sender is generally given only the guarantees that its message will eventually be inserted in the recipient's queue. No guarantees are given about when, or even if the message will actually be read, which is completely determined by the behavior of the recipient.

These semantics permit communication loosely-coupled in time. There is thus no need for the receiver to be executing when a message is being sent to its queue. Likewise, there is no need for the sender to be executing at the moment its message is picked up by the receiver. The sender and receiver can execute completely independently of each other. In fact, once a message has been deposited in a queue, it will remain there until it is removed, irrespective of whether its sender or receiver is executing. This gives us four combinations with respect to the execution mode of the sender and receiver, as shown in Fig. 4-17.

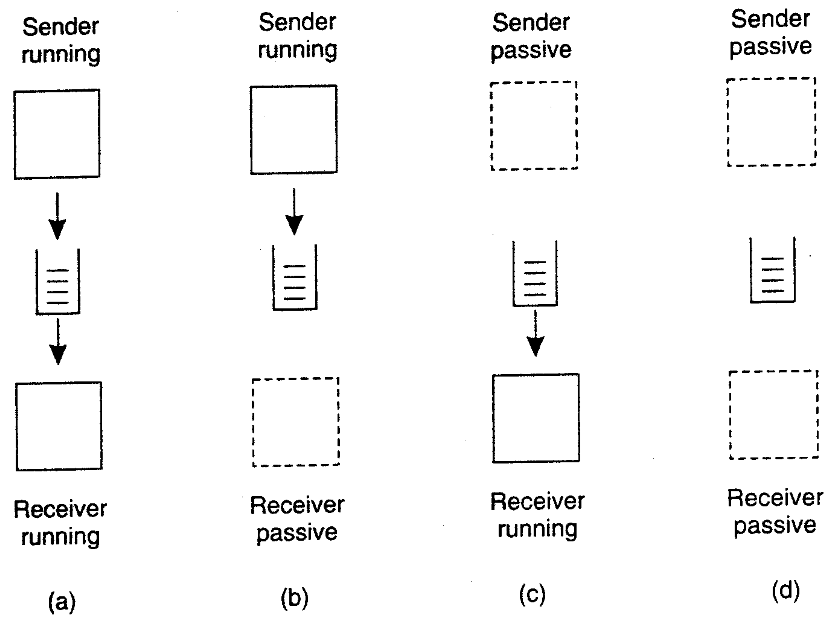


Figure 4-17. Four combinations for loosely-coupled communications using queues.

In Fig.4-17(a), both the sender and receiver execute during the entire transmission of a message. In.Fig. 4-17(b), only the sender is executing, while the receiver is passive, that is, in a state in which message delivery is not possible. Nevertheless, the sender can still send messages. The combination of a passive sender and an executing receiver is shown in Fig. 4-17(c). In this case, the receiver can read messages that were sent to it, but it is not necessary 'that their respective senders are executing as well. Finally, in Fig. 4-17(d), we see the situation that the system is storing (and possibly transmitting) messages even while sender and receiver are passive.

Messages can, in principle, contain any data. The only important aspect from the perspective of middleware is that messages are properly addressed. In practice, addressing is done by providing a systemwide unique name of the destination queue. In some cases, message size may be limited, although it is also possible that the underlying system takes care of fragmenting and assembling large messages in a way that is completely transparent to applications. An effect of this approach is that the basic interface offered to applications can be extremely simple, as shown in Fig. 4-18.

The put primitive is called by a sender to pass a message to the underlying system that is to be appended to the specified queue. As we explained. this is a

Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

Figure 4-18. Basic interface to a queue in a message-queuing system.

nonblocking call. The get primitive is a blocking call by which an authorized process can remove the longest pending message in the specified queue. The process is blocked only if the queue is empty. Variations on this call allow searching for a specific message in the queue, for example, using a priority, or a matching pattern. The nonblocking variant is given by the poll primitive. If the queue is empty, or if a specific message could not be found, the calling process simply continues.

Finally, most queuing systems also allow a process to install a handler as a *callback function,* which is automatically invoked whenever a message is put into the queue. Callbacks can also be used to automatically start a process that will fetch messages from the queue if no process is currently executing. This approach is often implemented by means of a daemon on the receiver's side that continuously monitors the queue for incoming messages and handles accordingly.

General Architecture of a Message-Queuing System

Let us now take a closer look at what a general message-queuing system looks like. One of the first restrictions that we make is that messages can be put only' into queues that are *local* to the sender, that is, queues on the same machine, or no worse than on a machine nearby such as on the same LAN that can be efficiently reached through an RPC. Such a queue is called the source queue. Likewise, messages can be read only from local queues. However, a message put into a queue will contain the specification of a destination queue to which it should be transferred. It is the responsibility of a message-queuing system to provide queues to senders and receivers and take care that messages are transferred from their source to their destination queue.

It is important to realize that the collection of queues is distributed across multiple machines. Consequently, for a message-queuing system to transfer messages, it should maintain a mapping of queues to network locations. In practice, this means that it should maintain a (possibly distributed) database of queue names to network locations, as shown in Fig. 4-19. Note that such a mapping is completely analogous to the use of the Domain Name System (DNS) for e-mail in the Internet. For example, when sending a message to the logical *mail* address steen@cs.vu.nl, the mailing system will query DNS to find the *network* (i.e., IP) address of the recipient's mail server to use for the actual message transfer.

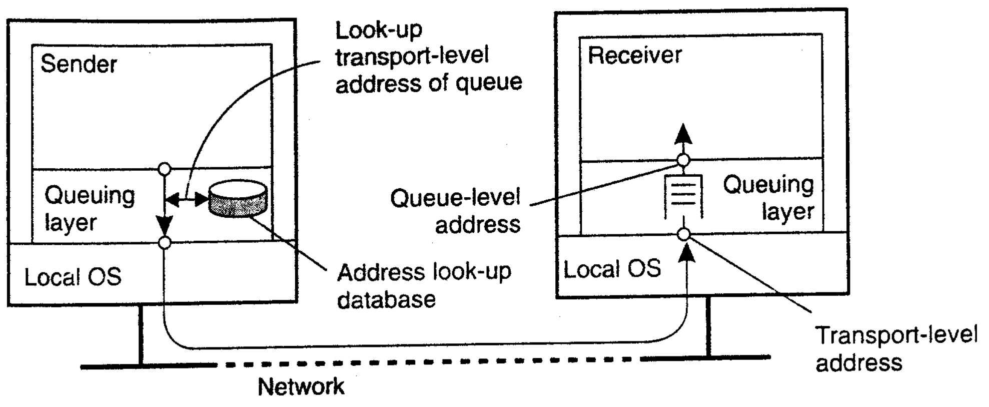


Figure 4-19. The relationship between queue-level addressing and network-level addressing.

Queues are managed by queue managers. Normally, a queue manager interacts directly with the application that is sending or receiving a message. However, there are also special queue managers that operate as routers, or relays: they forward incoming messages to other queue managers. In this way, a message-queuing system may gradually grow into a complete, application-level, overlay network, on top of an existing computer network. This approach is similar to the construction of the early MBone over the Internet, in which ordinary user processes were configured as multicast routers. As it turns out, multicasting through overlay networks is still important as we will discuss later in this chapter.

Relays can be convenient for a number of reasons. For example, in many message-queuing systems, there is no general naming service available that can dynamically maintain queue-to-location mappings. Instead, the topology of the queuing network is static, and each queue manager needs a copy of the queue-to-location mapping. It is needless to say that in large-scale queuing systems. this approach can easily lead to network-management problems.

One solution is to use a few routers that know about the network topology. When a sender *A* puts a message for destination *B* in its local queue, that message is first transferred to the nearest router, say *Rl,* as shown in Fig. 4-20. At that point, the router knows what to do with the message and forwards it in the direction of *B*. For example, *Rl* may derive from *B's* name that the message should be forwarded to router *R2*. In this way, only the routers need to be updated when queues are added or removed. while every other queue manager has to know only where the nearest router is.

Relays can thus generally help build scalable message-queuing systems. However, as queuing networks grow, it is clear that the manual configuration of networks will rapidly become completely unmanageable. The only solution is to adopt dynamic routing schemes as is done for computer networks. In that respect, it is somewhat surprising that such solutions are not yet integrated into some of the popular message-queuing systems.

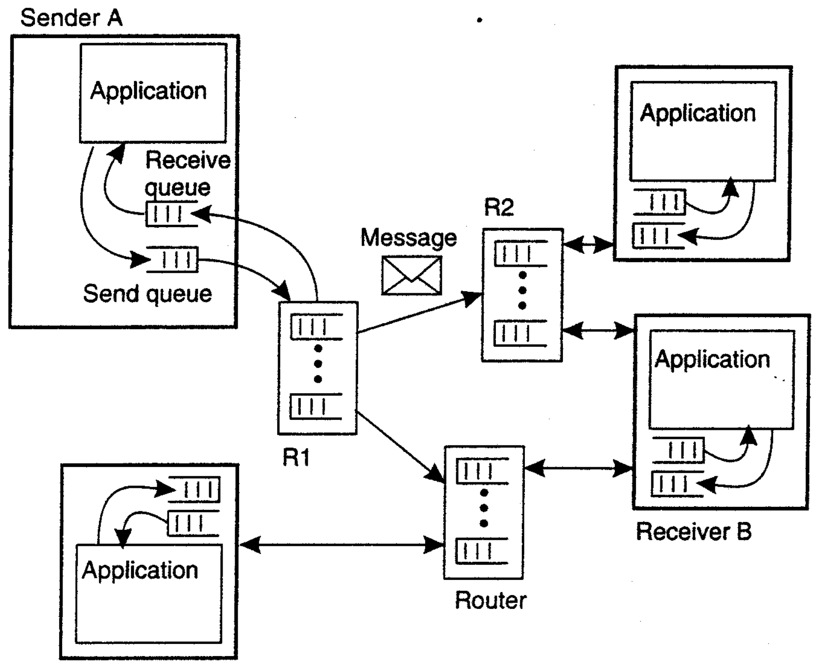


Figure 4-20. The general organization of a message-queuing system with routers.

Another reason why relays are used is that they allow for secondary processing of messages. For example, messages may need to be logged for reasons of security or fault tolerance. A special form of relay that we discuss in the next section is one that acts as a gateway, transforming messages into a format that can be understood by the receiver.

Finally, relays can be used for multicasting purposes. In that case, an incoming message is simply put into each send queue.

Message Brokers

An important application area of message-queuing systems is integrating existing and new applications into a single, coherent distributed information system. Integration requires that applications can understand the messages they receive. In practice, this requires the sender to have its outgoing messages in the same format as that of the receiver.

The problem with this approach is that each time an application is added to the system that requires a separate message format, each potential receiver will have to be adjusted in order to produce that format.

An alternative is to agree on a common message format, as is done with traditional network protocols. Unfortunately, this approach will generally not work for message-queuing systems. The problem is the level of abstraction at which these systems operate. A common message format makes sense only if the collection of processes that make use of that format indeed have enough in common. If the collection of applications that make up a distributed information system is highly diverse (which it often is), then the best common format may well be no more than a sequence of bytes.

Although a few common message formats for specific application domains have been defined, the general approach is to learn to live with different formats, and try to provide the means to make conversions as simple as possible. In message-queuing systems, conversions are handled by special nodes in a queuing network, known as message brokers. A message broker acts as an application-level gateway in a message-queuing system. Its main purpose is to convert incoming messages so that they can be understood by the destination application. Note that to a message-queuing system, a message broker is just another application. as shown in Fig. 4-21. In other words, a message broker is generally not considered to be an integral part of the queuing system.

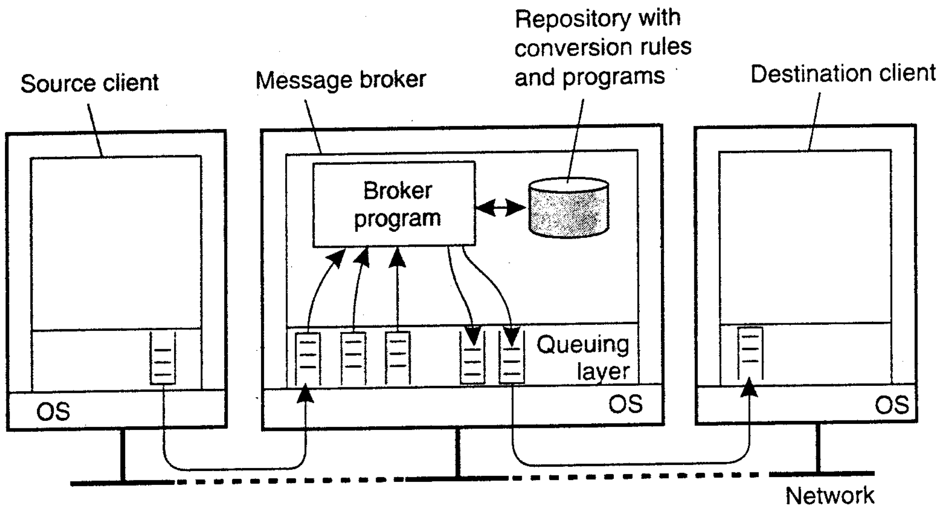


Figure 4-21. The general organization of a message broker in a message-queuing system.

A message broker can be as simple as a reformatter for messages. For example, assume an incoming message contains a table from a database, in which records are separated by a special *end-oj-record* delimiter and fields within a record have a known, fixed length. If the destination application expects a different delimiter between records, and also expects that fields have variable lengths, a message broker can be used to convert messages to the format expected by the destination.

In a more advanced setting, a message broker may act as an application-level gateway, such as one that handles the conversion between two different database applications. In such cases, frequently it cannot be guaranteed that all information contained in the incoming message can actually be transformed into something appropriate for the outgoing message.

However, more common is the use of a message broker for advanced enterprise application integration (EAI) as we discussed in Chap. 1. In this case, rather than (only) converting messages, a broker is responsible for matching applications based on the messages that are being exchanged. In such a model, called publish/subscribe, applications send messages in the form of *publishing.* In particular, they may publish a message on topic X, which is then sent to the broker. Applications that have stated their interest in messages on topic X, that is, who have *subscribed* to those messages, will then receive these messages from the broker. More advanced forms of mediation are also possible, but we will defer further discussion until Chap. 13.

At the heart of a message broker lies a repository of rules and programs that can transform a message of type *TI* to one of type *T2.* The problem is defining the rules and developing the programs. Most message broker products come with sophisticated development tools, but the bottom line is still that the repository needs to be filled by experts. Here we see a perfect example where commercial products are often misleadingly said to provide "intelligence," where, in fact, the only intelligence is to be found in the heads of those experts.

A Note on Message-Queuing Systems

Considering what we have said about message-queuing systems, it would appear that they have long existed in the form of implementations for e-mail services. E-mail systems are generally implemented through a collection of mail servers that store and forward messages on behalf of the users on hosts directly connected to the server. Routing is generally left out, as e-mail systems can make direct use of the underlying transport services. For example, in the mail protocol for the Internet, SMTP (Postel, 1982), a message is transferred by setting up a direct TCP connection to the destination mail server.

What makes e-mail systems special compared to message-queuing systems is that they are primarily aimed at providing direct support for end users. This explains, for example, why a number of groupware applications are based directly on an e-mail system (Khoshafian and Buckiewicz 1995). In addition, e-mail systems may have very specific requirements such as automatic message filtering, support for advanced messaging databases (e.g., to easily retrieve previously stored messages), and so on.

General message-queuing systems are not aimed at supporting only end users. An important issue is that they are set up to enable persistent communication between processes, regardless of whether a process is running a user application. handling access to a database, performing computations, and so on. This approach leads to a different set of requirements for message-queuing systems than pure e-mail systems. For example, e-mail systems generally need not provide guaranteed message delivery, message priorities, logging facilities, efficient multicasting, load balancing, fault tolerance, and so on for general usage.

General-purpose message-queuing systems, therefore, have a wide range of applications, including e-mail, workflow, groupware, and batch processing. However, as we have stated before, the most important application area is the integration of a (possibly widely-dispersed) collection of databases and applications into a federated information system (Hohpe and Woolf, 2004). For example, a query expanding several databases may need to be split into subqueries that are forwarded to individual databases. Message-queuing systems assist by providing the basic means to package each subquery into a message and routing it to the appropriate database. Other communication facilities we have discussed in this chapter are far less appropriate.

4.3.3 Example: IBM's WebSphere Message-Queuing System

To help understand how message-queuing systems work in practice, let us take a look at one specific system, namely the message-queuing system that is part of IBM's WebSphere product. Formerly known as MQSeries, it is now referred to as WebSphere MQ. There is a wealth of documentation on WebSphere MQ, and in the following we can only resort to the basic principles. Many architectural details concerning message-queuing networks can be found in IBM (2005b, 2005d). Programming message-queuing networks is not something that can be learned on a Sunday afternoon, and MQ's programming guide (IBM, 2005a) is a good example showing that going from principles to practice may require substantial effort.

Overview

The basic architecture of an MQ queuing network is quite straightforward, and is shown in Fig. 4-22. All queues are managed by queue managers. A queue manager is responsible for removing messages from its send queues, and forwarding those to other queue managers. Likewise, a queue manager is responsible for handling incoming messages by picking them up from the underlying network and subsequently storing each message in the appropriate input queue. To give an impression of what messaging can mean: a message has a maximum default size of 4 MB, but this can be increased up to 100 MB. A queue is normally restricted to 2 GB of data, but depending on the underlying operating system, this maximum can be easily set higher.

Queue managers are pairwise connected through message channels, which are an abstraction of transport-level connections. A message channel is a unidirectional, reliable connection between a sending and a receiving queue manager, through which queued messages are transported. For example, an Internet-based message channel is implemented as a TCP connection. Each of the two ends of a message channel is managed by a message channel agent (MCA). A sending MCA is basically doing nothing else than checking send queues for a message, wrapping it into a transport-level packet, and sending it along the connection to its associated receiving MCA. Likewise, the basic task of a receiving MCA is listening for an incoming packet, unwrapping it, and subsequently storing the unwrapped message into the appropriate queue.

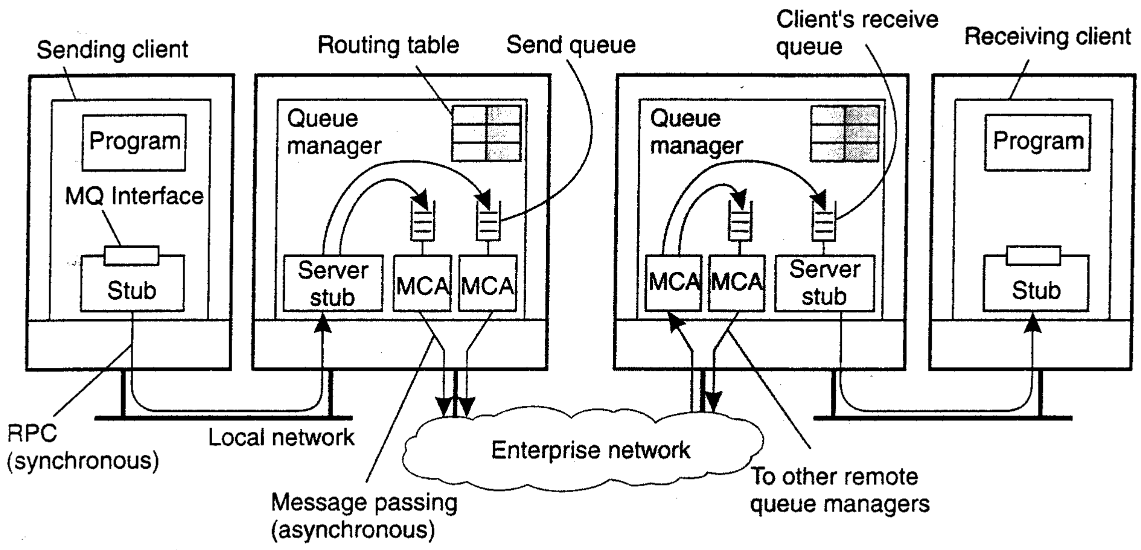


Figure 4-22. General organization of IBM's message-queuing system.

Queue managers can be linked into the same process as the application for which it manages the queues. In that case, the queues are hidden from the application behind a standard interface, but effectively can be directly manipulated by the application. An alternative organization is one in which queue managers and applications run on separate machines. In that case, the application is offered the same interface as when the queue manager is colocated on the same machine. However, the interface is implemented as a proxy that communicates with the queue manager using traditional RPC-based synchronous communication. In this way, MQ basically retains the model that only queues local to an application can be accessed.

Channels

An important component of MQ is formed by the message channels. Each message channel has exactly one associated send queue from which it fetches the messages it should transfer to the other end. Transfer along the channel can take place only if both its sending and receiving MCA are up and running. Apart from starting both MCAs manually, there are several alternative ways to start a channel, some of which we discuss next.

One alternative is to have an application directly start its end of a channel by activating the sending or receiving MCA. However, from a transparency point of view, this is not a very attractive alternative. A better approach to start a *sending* MeA is to configure the channel's send queue to set off a trigger when a message is first put into the queue. That trigger is associated with a handler to start the sending MCA so that it can remove messages from the send queue.

Another alternative is to start an MCA over the network. In particular, if one side of a channel is already active, it can send a control message requesting that the other MCA to be started. Such a control message is sent to a daemon listening to a well-known address on the same machine as where the other MCA is to be started.

Channels are stopped automatically after a specified time has expired during which no more messages were dropped into the send queue.

Each MCA has a set of associated attributes that determine the overall behavior of a channel. Some of the attributes are listed in Fig. 4-23. Attribute values of the sending and receiving MCA should be compatible and perhaps negotiated first before a channel can be set up. For example, both MCAs should obviously support the same transport protocol. An example of a nonnegotiable attribute is whether or not messages are to be delivered in the same order as they are put into the send queue. If one MCA wants FIFO delivery, the other must comply. An example of a negotiable attribute value is the maximum message length, which will simply be chosen as the minimum value specified by either MCA.

Attribute	Description
Transport type	Determines the transport protocol to be used
FIFO delivery	Indicates that messages are to be delivered in the order they are sent
Message length	Maximum length of a single message
Setup retry count	Specifies maximum number of retries to start up the remote MCA
Delivery retries	Maximum times MCA will try to put received message into queue

Figure 4-23. Some attributes associated with message channel agents.

Message Transfer

To transfer a message from one queue manager to another (possibly remote) queue manager, it is necessary that each message carries its destination address, for which a transmission header is used. An address in MQ consists of two parts. The first part consists of the name of the queue manager to which the message is to be delivered. The second part is the name of the destination queue resorting under that manager to which the message is to be appended.

Besides the destination address, it is also necessary to specify the route that a message should follow. Route specification is done by providing the name of the local send queue to which a message is to be appended. Thus it is not necessary to provide the full route in a message. Recall that each message channel has exactly one send queue. By telling to which send queue a message is to be appended, we efectively specify to which queue manager a message is to be forwarded.

In most cases, routes are explicitly stored inside a queue manager in a routing table. An entry in a routing table is a pair *(destQM, sendQ),* where *destQM* is the name of the destination queue manager, and *sendQ* is the name of the local send queue to which a message for that queue manager should be appended. (A routing table entry is called an alias in MQ.)

It is possible that a message needs to be transferred across multiple queue managers before reaching its destination. Whenever such an intermediate queue manager receives the message, it simply extracts the name of the destination queue manager from the message header, and does a routing-table look-up to find the local send queue to which the message should be appended.

It is important to realize that each queue manager has a systemwide unique name that is effectively used as an identifier for that queue manager. The problem with using these names is that replacing a queue manager, or changing its name, will affect all applications that send messages to it. Problems can be alleviated by using a local alias for queue manager names. An alias defined within a queue manager *Ml* is another name for a queue manager *M2,* but which is available only to applications interfacing to *Ml.* An alias allows the use of the same (logical) name for a queue, even if the queue manager of that queue changes. Changing the name of a queue manager requires that we change its alias in all queue managers. However, applications can be left unaffected.

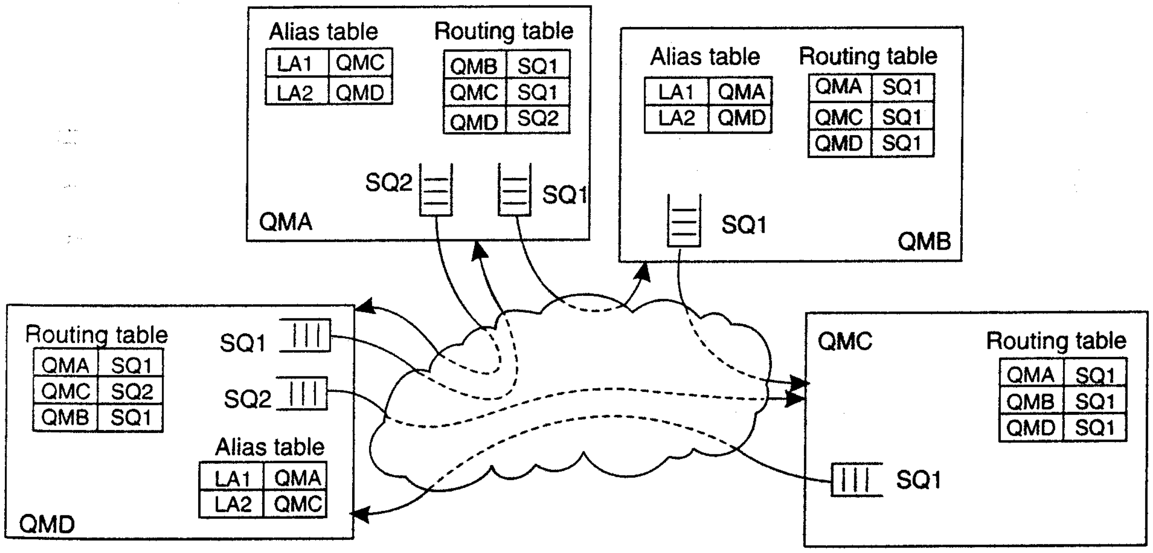


Figure 4-24. The general organization of an MQ queuing network using routing tables and aliases.

The principle of using routing tables and aliases is shown in Fig. 4-24. For example, an application linked to queue manager *QMA* can refer to a remote queue manager using the local alias *LAJ*. The queue manager will first look up the actual destination in the alias table to find it is queue manager *QMC*. The route to *QMC* is found in the routing table, which states that messages for *QMC* should be appended to the outgoing queue *SQl*, which is used to transfer messages to queue manager *QMB*. The latter will use its routing table to forward the message to *QMC*.

Following this approach of routing and aliasing leads to a programming interface that, fundamentally, is relatively simple, called the Message Queue Interface (MQI). The most important primitives of MQI are summarized in Fig. 4-25.

Primitive	**Description**
MQopen	Open a (possibly remote) queue
MQclose	Close a queue
MQput	Put a message into an opened queue
MQget	Get a message from a (local) queue

Figure 4-25. Primitives available in the message-queuing interface.

To put messages into a queue, an application calls the MQopen primitive, specifying a destination queue in a specific queue manager. The queue manager can be named using the locally-available alias. Whether the destination queue is actually remote or not is completely transparent to the application. MQopen should also be called if the application wants to get messages from its local queue. Only local queues can be opened for reading incoming messages. When an application is finished with accessing a queue, it should close it by calling MQclose.

Messages can be written to, or read from, a queue using MQput and MQget, respectively. In principle, messages are removed from a queue on a priority basis. Messages with the same priority are removed on a first-in, first-out basis, that is, the longest pending message is removed first. It is also possible to request for specific messages. Finally, MQ provides facilities to signal applications when messages have arrived, thus avoiding that an application will continuously have to poll a message queue for incoming messages.

Managing Overlay Networks

From the description so far, it should be clear that an important part of managing MQ systems is connecting the various queue managers into a consistent overlay network. Moreover, this network needs to be maintained over time. For small networks, this maintenance will not require much more than average administrative work, but matters become complicated when message queuing is used to integrate and disintegrate large existing systems.

A major issue with MQ is that overlay networks need to be manually administrated. This administration not only involves creating channels between queue managers, but also filling in the routing tables. Obviously, this can grow into a nightmare. Unfortunately, management support for MQ systems is advanced only in the sense that an administrator can set virtually every possible attribute, and tweak any thinkable configuration. However, the bottom line is that channels and routing tables need to be manually maintained.

At the heart of overlay management is the **channel control function** component, which logically sits between message channel agents. This component allows an operator to monitor exactly what is going on at two end points of a channel. In addition, it is used to create channels and routing tables, but also to manage the queue managers that host the message channel agents. In a way, this approach to overlay management strongly resembles the management of cluster servers where a single administration server is used. In the latter case, the server essentially offers only a remote shell to each machine in the cluster, along with a few collective operations to handle groups of machines. The good news about distributed-systems management is that it offers lots of opportunities if you are looking for an area to explore new solutions to serious problems.

4.4 STREAM-ORIENTED COMMUNICATION

Communication as discussed so far has concentrated on exchanging more-or-less independent and complete units of information. Examples include a request for invoking a procedure, the reply to such a request, and messages exchanged between applications as in message-queuing systems. The characteristic feature of this type of communication is that it does not matter at what particular point in time communication takes place. Although a system may perform too slow or too fast, timing has no effect on correctness.

There are also forms of communication in which timing plays a crucial role. Consider, for example, an audio stream built up as a sequence of 16-bit samples, each representing the amplitude of the sound wave as is done through Pulse Code Modulation (PCM). Also assume that the audio stream represents CD quality, meaning that the original sound wave has been sampled at a frequency of 44,100 Hz. To reproduce the original sound, it is essential that the samples in the audio stream are played out in the order they appear in the stream, but also at intervals of exactly 1/44,100 sec. Playing out at a different rate will produce an incorrect version of the original sound.

The question that we address in this section is which facilities a distributed system should offer to exchange time-dependent information such as audio and video streams. Various network protocols that deal with stream-oriented communication are discussed in Halsall (2001). Steinmetz and Nahrstedt (2004) provide an overall introduction to multimedia issues, part of which forms stream-oriented communication. Query processing on data streams is discussed in Babcock et al. (2002).

4.4.1 Support for Continuous Media

Support for the exchange of time-dependent information is often formulated as support for continuous media. A medium refers to the means by which information is conveyed. These means include storage and transmission media, presentation media such as a monitor, and so on. An important type of medium is the way that information is *represented.* In other words, how is information encoded in a computer system? Different representations are used for different types of information. For example, text is generally encoded as ASCII or Unicode. Images can be represented in different formats such as GIF or lPEG. Audio streams can be encoded in a computer system by, for example, taking 16-bit samples using PCM.

In continuous (representation) media, the temporal relationships between different data items are fundamental to correctly interpreting what the data actually means. We already gave an example of reproducing a sound wave by playing out an audio stream. As another example, consider motion. Motion can be represented by a series of images in which successive images must be displayed at a uniform spacing *T* in time, typically 30-40 msec per image. Correct reproduction requires not only showing the stills in the correct order, but also at a constant frequency of *l/T* images per second.

In contrast to continuous media, discrete (representation) media, is characterized by the fact that temporal relationships between data items are *not* fundamental to correctly interpreting the data. Typical examples of discrete media include representations of text and still images, but also object code or executable files.

Data Stream

To capture the exchange of time-dependent information, distributed systems generally provide support for data streams. A data stream is nothing but a sequence of data units. Data streams can be applied to discrete as well as continuous media. For example, UNIX pipes or TCPIIP connections are typical examples of (byte-oriented) discrete data streams. Playing an audio file typically requires setting up a continuous data stream between the file and the audio device.

Timing is crucial to continuous data streams. To capture timing aspects, a distinction is often made between different transmission modes. In asynchronous transmission mode the data items in a stream are transmitted one after the other, but there are no further timing constraints on when transmission of items should take place. This is typically the case for discrete data streams. For example, a file can be transferred as a data stream, but it is mostly irrelevant exactly when the transfer of each item completes.

In synchronous transmission mode, there is a maximum end-to-end delay defined for each unit in a data stream. Whether a data unit is transferred much faster than the maximum tolerated delay is not important. For example, a sensor may sample temperature at a certain rate and pass it through a network to an operator. In that case, it may be important that the end-to-end propagation time through the network is guaranteed to be lower than the time interval between taking samples, but it cannot do any harm if samples are propagated much faster than necessary.

Finally, in **isochronous transmission mode,** it is necessary that data units are transferred on time. This means that data transfer is subject to a maximum *and* minimum end-to-enddelay, also referred to as bounded (delay) jitter. Isochronous transmission mode is particularly interesting for distributed multimedia systems, as it plays a crucial role in representing audio and video. In this chapter, we consider only continuous data streams using isochronous transmission, which we will refer to simply as streams.

Streams can be simple or complex. A **simple stream** consists of only a single sequence of data, whereas a **complex stream** consists of several related simple streams, called **substreams.** The relation between the substreams in a complex stream is often also time dependent. For example, stereo audio can be transmitted by means of a complex stream consisting of two substreams, each used for a single audio channel. It is important, however, that those two substreams are continuously synchronized. In other words, data units from each stream are to be communicated pairwise to ensure the effect of stereo. Another example of a complex stream is one for transmitting a movie. Such a stream could consist of a single video stream, along with two streams for transmitting the sound of the movie in stereo. A fourth stream might contain subtitles for the deaf, or a translation into a different language than the audio. Again, synchronization of the substreams is important. If synchronization fails, reproduction of the movie fails. We return to stream synchronization below.

From a distributed systems perspective, we can distinguish several elements that are needed for supporting streams. For simplicity, we concentrate on streaming stored data, as opposed to streaming live data. In the latter case. data is captured in real time and sent over the network to recipients. The main difference between the two is that streaming live data leaves less opportunities for tuning a stream. Following Wu et al. (2001), we can then sketch a general client-server architecture for supporting continuous multimedia streams as shown in Fig. 4-26.

This general architecture reveals a number of important issues that need to be dealt with. In the first place, the multimedia data, notably video and to a lesser extent audio, will need to be compressed substantially in order to reduce the required storage and especially the network capacity. More important from the perspective of communication are controlling the quality of the transmission and synchronization issues. We discuss these issues next.

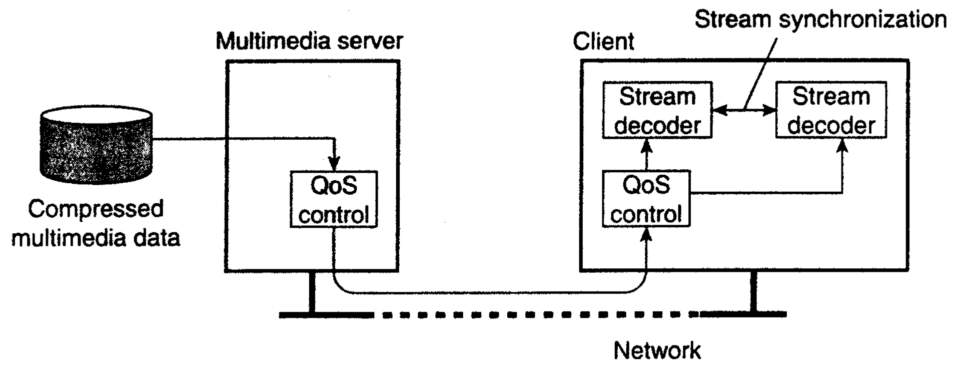


Figure 4-26. A general architecture for streaming stored multimedia data over a network.

4.4.2 Streams and Quality of Service

Timing (and other nonfunctional) requirements are generally expressed as Quality of Service (QoS) requirements. These requirements describe what is needed from the underlying distributed system and network to ensure that, for example, the temporal relationships in a stream can be preserved. QoS for continuous data streams mainly concerns timeliness, volume, and reliability. In this section we take a closer look at QoS and its relation to setting up a stream.

Much has been said about how to specify required QoS (see, e.g., Jin and Nahrstedt, 2004). From an application's perspective, in many cases it boils down to specifying a few important properties (Halsall, 2001):

1. The required bit rate at which data should be transported.
2. The maximum delay until a session has been set up (i.e., when an application can start sending data).
3. The maximum end-to-end delay (i.e., how long it will take until a data unit makes it to a recipient).
4. The maximum delay variance, or jitter.
5. The maximum round-trip delay.

It should be noted that many refinements can be made to these specifications, as explained, for example, by Steinmetz and Nahrstadt (2004). However, when dealing with stream-oriented communication that is based on the Internet protocol stack, we simply have to live with the fact that the basis of communication is formed by an extremely simple, best-effort datagram service: IP. When the going gets tough, as may easily be the case in the Internet, the specification of IP allows a protocol implementation to drop packets whenever it sees fit. Many, if not all distributed systems that support stream-oriented communication, are currently built on top of the Internet protocol stack. So much for QoS specifications. (Actually, IP does provide some QoS support, but it is rarely implemented.)

Enforcing QoS

Given that the underlying system offers only a best-effort delivery service, a distributed system can try to conceal as much as possible of the *lack* of quality of service. Fortunately, there are several mechanisms that it can deploy.

First, the situation is not really so bad as sketched so far. For example, the Internet provides a means for differentiating classes of data by means of its differentiated services. A sending host can essentially mark outgoing packets as belonging to one of several classes, including an expedited forwarding class that essentially specifies that a packet should be forwarded by the current router with absolute priority (Davie et al., 2002). In addition, there is also an assured forwarding class, by which traffic is divided into four subclasses, along with three ways to drop packets if the network gets congested. Assured forwarding therefore effectively defines a range of priorities that can be assigned to packets, and as such allows applications to differentiate time-sensitive packets from noncritical ones.

Besides these network-level solutions, a distributed system can also help in getting data across to receivers. Although there are generally not many tools available, one that is particularly useful is to use buffers to reduce jitter. The principle is simple, as shown in Fig. 4-27. Assuming that packets are delayed with a certain variance when transmitted over the network, the receiver simply stores them in a buffer for a maximum amount of time. This will allow the receiver to pass packets to the application at a regular rate, knowing that there will always be enough packets entering the buffer to be played back at that rate.

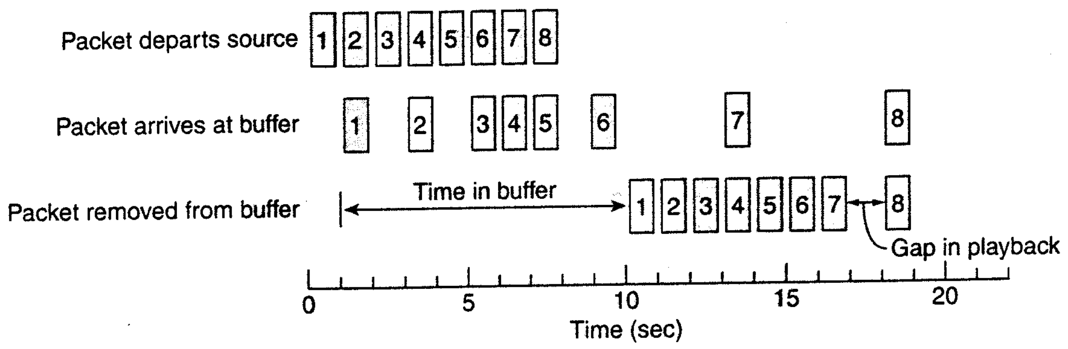


Figure 4-27. Using a buffer to reduce jitter.

Of course, things may go wrong, as is illustrated by packet #8 in Fig. 4-27. The size of the receiver's buffer corresponds to 9 seconds of packets to pass to the application. Unfortunately, packet #8 took 11 seconds to reach the receiver, at which time the buffer will have been completely emptied. The result is a gap in the playback at the application. The only solution is to increase the buffer size. The obvious drawback is that the delay at which the receiving application can start playing back the data contained in the packets increases as well.

Other techniques can be used as well. Realizing that we are dealing with an underlying best-effort service also means that packets may be lost. To compensate for this loss in quality of service. we need to apply error correction techniques (Perkins et al., 1998; and Wah et al., 2000). Requesting the sender to retransmit a missing packet is generally out of the question, so that forward error correction (FEe) needs to be applied. A well-known technique is to encode the outgoing packets in such a way that any *k* out of *n* received packets is enough to reconstruct *k* correct packets.

One problem that may occur is that a single packet contains multiple audio and video frames. As a consequence, when a packet is lost, the receiver may actually perceive a large gap when playing out frames. This effect can be somewhat circumvented by interleaving frames, as shown in Fig. 4-28. In this way, when a packet is lost, the resulting gap in successive frames is distributed over time. Note, however, that this approach does require a larger receive buffer in comparison to noninterleaving, and thus imposes a higher start delay for the receiving application. For example, when considering Fig.4-28(b), to play the first four frames, the receiver will need to have four packets delivered, instead of only one packet in comparison to noninterleaved transmission.

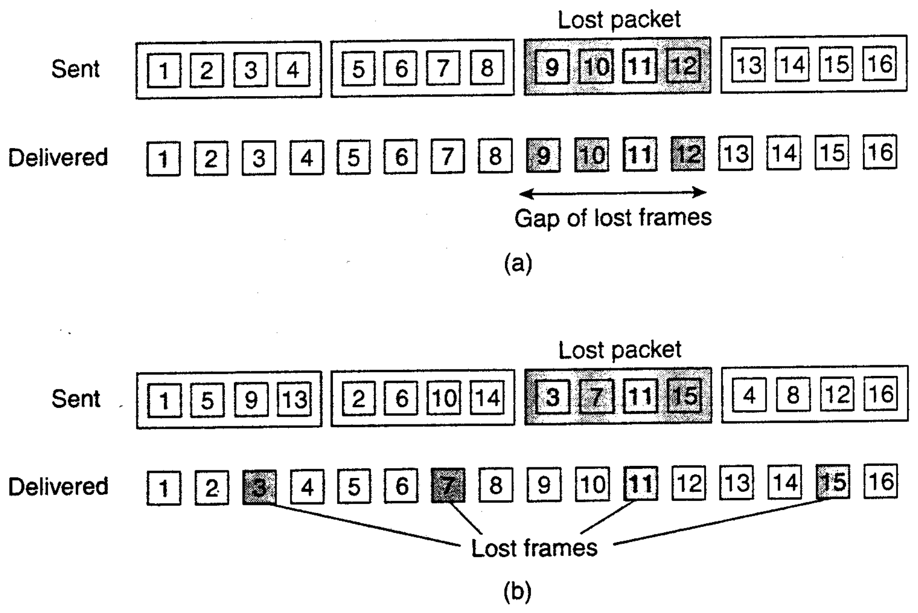


Figure 4-28. The effect of packet loss in (a) noninterleaved transmission and (b) interleaved transmission.

4.4.3 Stream Synchronization

An important issue in multimedia systems is that different streams, possibly in the form of a complex stream, are mutually synchronized. Synchronization of streams deals with maintaining temporal relations between streams. Two types of synchronization occur.

The simplest form of synchronization is that between a discrete data stream and a continuous data stream. Consider, for example, a slide show on the Web that has been enhanced with audio. Each slide is transferred from the server to the client in the form of a discrete data stream. At the same time, the client should play out a specific (part of an) audio stream that matches the current slide that is also fetched from the server. In this case, the audio stream is to be 'synchronized with the presentation of slides.

A more demanding type of synchronization is that between continuous data streams. A daily example is playing a movie in which the video stream needs to be synchronized with the audio, commonly referred to as lip synchronization. Another example of synchronization is playing a stereo audio stream consisting of two substreams, one for each channel.. Proper play out requires that the two substreams are tightly synchronized: a difference of more than 20 μsec can distort the stereo effect.

Synchronization takes place at the level of the data units of which a stream is made up. In other words, we can synchronize two streams only between data units. The choice of what exactly a data unit is depends very much on the level of abstraction at which a data stream is viewed. To make things concrete, consider again a CD-quality (single-channel) audio stream. At the finest granularity, such a stream appears as a sequence of 16-bit samples. With a sampling frequency of 44,100 Hz, synchronization with other audio streams could, in theory, take place approximately every 23 μsec. For high-quality stereo effects, it turns out that synchronization at this level is indeed necessary.

However, when we consider synchronization between an audio stream and a video stream for lip synchronization, a much coarser granularity can be taken. As we explained, video frames need to be displayed at a rate of 25 Hz or more. Taking the widely-used NTSC standard of 29.97 Hz, we could group audio samples into logical units that last as long as a video frame is displayed (33 msec). With an audio sampling frequency of 44,100 Hz, an audio data unit can thus be as large as 1470 samples, or 11,760 bytes (assuming each sample is 16 bits). In practice, larger units lasting 40 or even 80 msec can be tolerated (Steinmetz, 1996).

Synchronization Mechanisms

Let us now see how synchronization is actually done. Two issues need to be distinguished: (1) the basic mechanisms for synchronizing two streams, and (2) the distribution of those mechanisms in a networked environment.

Synchronization mechanisms can be viewed at several different levels of abstraction. At the lowest level, synchronization is done explicitly by operating on the data units of simple streams. This principle is shown in Fig. 4-29. In essence, there is a process that simply executes read and write operations on several simple streams, ensuring that those operations adhere to specific timing and synchronization constraints.

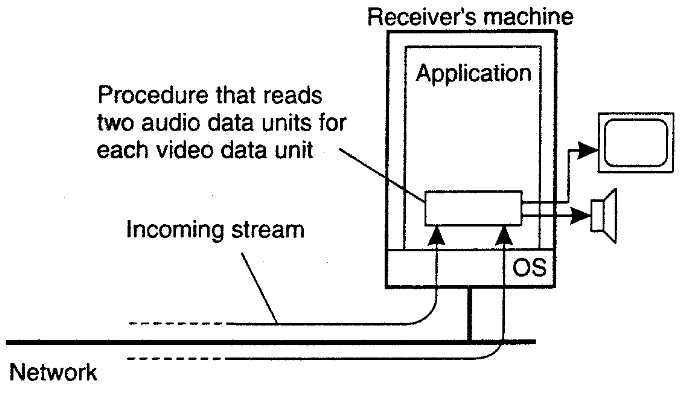

Receiver's machine
Application
Procedure that reads two audio data units for each video data unit
Incoming stream
OS
Network

Figure 4-29. The principle of explicit synchronization on the level data units.

For example, consider a movie that is presented as two input streams. The video stream contains uncompressed low-quality images of 320x240 pixels, each encoded by a single byte, leading to video data units of 76,800 bytes each. Assume that images are to be displayed at 30 Hz, or one image every 33 msec. The audio stream is assumed to contain audio samples grouped into units of 11760 bytes, each corresponding to 33 ms of audio, as explained above. If the input process can handle 2.5 MB/sec, we can achieve lip synchronization by simply alternating between reading an image and reading a block of audio samples every 33 ms.

The drawback of this approach is that the application is made completely responsible for implementing synchronization while it has only low-level facilities available. A better approach is to offer an application an interface that allows it to more easily control streams and devices. Returning to our example, assume that the video display has a control interface that allows it to specify the rate at which images should be displayed. In addition, the interface offers the facility to register a user-defined handler that is called each time *k* new images have arrived. An analogous interface is offered by the audio device. With these control interfaces, an application developer can write a simple monitor program consisting of two handlers, one for each stream, that jointly check if the video and audio stream are sufficiently synchronized, and if necessary, adjust the rate at which video or audio units are presented.

This last example is illustrated in Fig. 4-30, and is typical for many multimedia middleware systems. In effect, multimedia middleware offers a collection of interfaces for controlling audio and video streams, including interfaces for controlling devices such as monitors, cameras, microphones, etc. Each device and stream has its own high-level interfaces, including interfaces for notifying an application when some event occurred. The latter are subsequently used to write handlers for synchronizing streams. Examples of such interfaces are given in Blair and Stefani (1998).

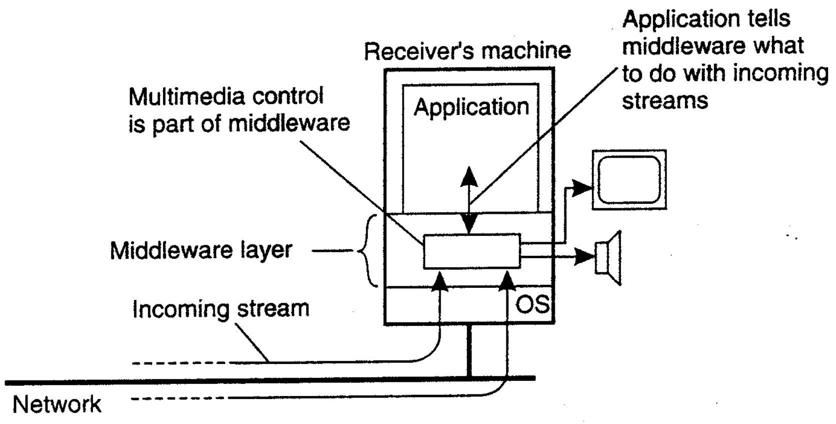

Receiver's machine
Application tells middleware what to do with incoming streams
Multimedia control is part of middleware
Application
Middleware layer
Incoming stream
OS
Network

Figure 4-30. The principle of synchronization as supported by high-level interfaces.

The distribution of synchronization mechanisms is another issue that needs to be looked at. First, the receiving side of a complex stream consisting of substreams that require synchronization, needs to know exactly what to do. In other words, it must have a complete *synchronization specification* locally available. Common practice is to provide this information implicitly by multiplexing the different streams into a single stream containing all data units, including those for synchronization.

This latter approach to synchronization is followed for MPEG streams. The MPEG (Motion Picture Experts Group) standards form a collection of algorithms for compressing video and audio. Several MPEG standards exist. MPEG-2, for example, was originally designed for compressing broadcast quality video into 4 to 6 Mbps. In MPEG-2, an unlimited number of continuous and discrete streams can be merged into a single stream. Each input stream is first turned into a stream of packets that carry a timestamp based on a 90-kHz system clock. These streams are subsequently multiplexed into a program stream then consisting of variable-length packets, but which have in common that they all have the same time base. The receiving side demultiplexes the stream, again using the timestamps of each packet as the basic mechanism for interstream synchronization.

Another important issue is whether synchronization should take place at the sending or the receiving side. If the sender handles synchronization, it may be possible to merge streams into a single stream with a different type of data unit. Consider again a stereo audio stream consisting of two substreams, one for each channel. One possibility is to transfer each stream independently to the receiver and let the latter synchronize the samples pairwise. Obviously, as each substream may be subject to different delays, synchronization can be extremely difficult. A better approach is to merge the two substreams at the sender. The resulting stream consists of data units consisting of pairs of samples, one for each channel. The receiver now merely has to read in a data unit, and split it into a left and right sample. Delays for both channels are now identical.

4.5 MULTICAST COMMUNICATION

An important topic in communication in distributed systems is the support for sending data to multiple receivers, also known as multicast communication. For many years, this topic has belonged to the domain of network protocols, where numerous proposals for network-level and transport-level solutions have been implemented and evaluated (Janie, 2005; and Obraczka, 1998). A major issue in all solutions was setting up the communication paths for information dissemination. In practice, this involved a huge management effort, in many cases requiring human intervention. In addition, as long as there is no convergence of proposals, ISPs have shown to be reluctant to support multicasting (Diot et al., 2000).

With the advent of peer-to-peer technology, and notably structured overlay management, it became easier to set up communication paths. As peer-to-peer solutions are typically deployed at the application layer, various application-level multicasting techniques have been introduced. In this section, we will take a brief look at these techniques.

Multicast communication can also be accomplished in other ways than setting up explicit communication paths. As we also explore in this section. gossip-based information dissemination provides simple (yet often less efficient) ways for multicasting.

4.5.1 Application-Level Multicasting

The basic idea in application-level multicasting is that nodes organize into an overlay network, which is then used to disseminate information to its members. An important observation is that network routers are not involved in group membership. As a consequence, the connections between nodes in the overlay network may cross several physical links, and as such, routing messages within the overlay may not be optimal in comparison to what could have been achieved by network-level routing.

A crucial design issue is the construction of the overlay network. In essence, there are two approaches (El-Sayed, 2003). First, nodes may organize themselves directly into a tree, meaning that there is a unique (overlay) path between every pair of nodes. An alternative approach is that nodes organize into a mesh network in which every node will have multiple neighbors and, in general, there exist multiple paths between every pair of nodes. The main difference between the two is that the latter generally provides higher robustness: if a connection breaks (e.g., because a node fails), there will still be an opportunity to disseminate information without having to immediately reorganize the entire overlay network.

To make matters concrete, let us consider a relatively simple scheme for constructing a multicast tree in Chord, which we described in Chap. 2. This scheme was originally proposed for Scribe (Castro et al., 2002) which is an application-level multicasting scheme built on top of Pastry (Rowstron and Druschel, 2001). The latter is also a DHT -based peer-to-peer system.

Assume a node wants to start a multicast session. To this end, it simply generates a multicast identifier, say *mid* which is just a randomly-chosen 160-bit key. It then looks up *succ(mid)*, which is the node responsible for that key, and promotes it to become the root of the multicast tree that will be used to sending data to interested nodes. In order to join the tree, a node *P* simply executes the operation LOOKUP(mid) having the effect that a lookup message with the request to join the multicast group *mid* will be routed from *P* to *succ(mid)*. As we mentioned before, the routing algorithm itself will be explained in detail in Chap. 5.

On its way toward the root, the join request will pass several nodes. Assume it first reaches node Q. If Q had never seen a join request for *mid* before, it will become a forwarder for that group. At that point, *P* will become a child of Q whereas the latter will continue to forward the join request to the root. If the next node on the root, say *R* is also not yet a forwarder, it will become one and record Q as its child and continue to send the join request.

On the other hand, if Q (or *R*) is already a forwarder for *mid*, it will also record the previous sender as its child (i.e., *P* or Q, respectively), but there will not be a need to send the join request to the root anymore, as Q (or *R*) will already be a member of the multicast tree.

Nodes such as *P* that have explicitly requested to join the multicast tree are, by definition, also forwarders. The result of this scheme is that we construct a multicast tree across the overlay network with two types of nodes: pure forwarders that act as helpers, and nodes that are also forwarders, but have explicitly requested to join the tree. Multicasting is now simple: a node merely sends a multicast message toward the root of the tree by again executing the LOOKUP(mid) operation, after which that message can be sent along the tree.

We note that this high-level description of multicasting in Scribe does not do justice to its original design. The interested reader is therefore encouraged to take a look at the details, which can be found in Castro et al. (2002).

Overlay Construction

From the high-level description given above, it should be clear that although building a tree by itself is not that difficult once we have organized the nodes into an overlay, building an efficient tree may be a different story. Note that in our description so far, the selection of nodes that participate in the tree does not take into account any performance metrics: it is purely based on the (logical) routing of messages through the overlay.

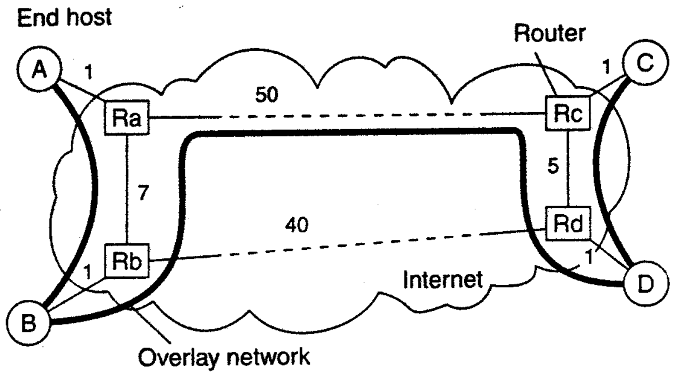


Figure 4-31. The relation between links in an overlay and actual network-level routes.

To understand the problem at hand, take a look at Fig. 4-31 which shows a small set of four nodes that are organized in a simple overlay network, with node *A* forming the root of a multicast tree. The costs for traversing a physical link are also shown. Now, whenever *A* multicasts a message to the other nodes, it is seen that this message will traverse each of the links *<B, Rb>*, *<Ra, Rb>*, *<Rc, Rd>*, and *<D, Rd>* twice. The overlay network would have been more efficient if we had not constructed an overlay link from *B* to *D*, but instead from *A* to *C*. Such a configuration would have saved the double traversal across links *<Ra, Rb>* and *<Rc, Rd>*.

The quality of an application-level multicast tree is generally measured by three different metrics: link stress, stretch, and tree cost. **Link stress** is defined per link and counts how often a packet crosses the same link (Chu et al., 2002). A link stress greater than 1 comes from the fact that although at a logical level a packet may be forwarded along two different connections, part of those connections may actually correspond to the same physical link, as we showed in Fig. 4-31.

The **stretch** or **Relative Delay Penalty** (RDP) measures the ratio in the delay between two nodes in the overlay, and the delay that those two nodes would experience in the underlying network. For example, in the overlay network, messages from *B* to *C* follow the route *B* ~ *Rb* ~ *Ra* ~ *Rc* ~ *C*, having a total cost of 59 units. However, messages would have been routed in the underlying network along the path *B* ~ *Rb* ~ *Rd* ~ *Rc* ~ *C*, with a total cost of 47 units, leading to a stretch of 1.255. Obviously, when constructing an overlay network, the goal is to minimize the aggregated stretch, or similarly, the average RDP measured over all node pairs.

Finally, the **tree cost** is a global metric, generally related to minimizing the aggregated link costs. For example, if the cost of a link is taken to be the delay between its two end nodes, then optimizing the tree cost boils down to finding a minimal spanning tree in which the total time for disseminating information to all nodes is minimal.

To simplify matters somewhat, assume that a multicast group has an associated and well-known node that keeps track of the nodes that have joined the tree. When a new node issues a join request, it contacts this rendezvous node to obtain a (potentially partial) list of members. The goal is to select the best member that can operate as the new node's parent in the tree. Who should it select? There are many alternatives and different proposals often follow very different solutions.

Consider, for example, a multicast group with only a single source. In this case, the selection of the best node is obvious: it should be the source (because in that case we can be assured that the stretch will be equal to 1). However, in doing so, we would introduce a star topology with the source in the middle. Although simple, it is not difficult to imagine the source may easily become overloaded. In other words, selection of a node will generally be constrained in such a way that only those nodes may be chosen who have *k* or less neighbors, with *k* being a design parameter. This constraint severely complicates the tree-establishment algorithm, as a good solution may require that part of the existing tree is reconfigured.

Tan et al. (2003) provide an extensive overview and evaluation of various solutions to this problem. As an illustration, let us take a closer look at one specific family, known as switch-trees (Helder and Jamin, 2002). The basic idea is simple. Assume we already have a multicast tree with a single source as root. In this tree, a node *P* can switch parents by dropping the link to its current parent in favor of a link to another node. The only constraints imposed on switching links is that the new parent can never be a member of the subtree rooted at *P* (as this would partition the tree and create a loop), and that the new parent will not have too many immediate children. The latter is needed to limit the load of forwarding messages by any single node.

There are different criteria for deciding to switch parents. A simple one is to optimize the route to the source, effectively minimizing the delay when a message is to be multicast. To this end, each node regularly receives information on other nodes (we will explain one specific way of doing this below). At that point, the node can evaluate whether another node would be a better parent in terms of delay along the route to the source, and if so, initiates a switch.

Another criteria could be whether the delay to the potential other parent is lower than to the current parent. If every node takes this as a criterion, then the aggregated delays of the resulting tree should ideally be minimal. In other words, this is an example of optimizing the cost of the tree as we explained above. However, more information would be needed to construct such a tree, but as it turns out, this simple scheme is a reasonable heuristic leading to a good approximation of a minimal spanning tree.

As an example, consider the case where a node *P* receives information on the neighbors of its parent. Note that the neighbors consist of *P*'s grandparent, along *with* the other siblings of *P's* parent. Node *P* can then evaluate the delays to each of these nodes and subsequently choose the one with the lowest delay, say *Q,* as *its* new parent. To that end, it sends a switch request to *Q*. To prevent loops from being formed due to concurrent switching requests. a node that has an outstanding switch request will simply refuse to process any incoming requests. In effect, this leads to a situation where only completely independent switches can be carried out simultaneously. Furthermore, *P* will provide *Q* with enough information to allow the latter to conclude that both nodes have the same parent, or that *Q* is the grandparent.

An important problem that we have not yet addressed is node failure. In the case of switch-trees, a simple solution is proposed: whenever a node notices that its parent has failed, it simply attaches itself to the root. At that point, the optimization protocol can proceed as usual and will eventually place the node at a good point in the multicast tree. Experiments described in Helder and Jamin (2002) show that the resulting tree is indeed close to a minimal spanning one.

4.5.2 Gossip-Based Data Dissemination

An increasingly important technique for disseminating information is to rely on *epidemic behavior.* Observing how diseases spread among people, researchers have since long investigated whether simple techniques could be developed for spreading information in very large-scale distributed systems. The main goal of these epidemic protocols is to rapidly propagate information among a large collection of nodes using only local information. In other words, there is no central component by which information dissemination is coordinated.

To explain the general principles of these algorithms, we assume that all ·updates for a specific data item are initiated at a single node. In this way, we simply avoid write-write conflicts. The following presentation is based on the classical paper by Demers et al. (1987) on epidemic algorithms. A recent overview of epidemic information dissemination can be found in Eugster at el. (2004).

Information Dissemination Models

As the name suggests, epidemic algorithms are based on the theory of epidemics, which studies the spreading of infectious diseases. In the case of large-scale distributed systems, instead of spreading diseases, they spread information. Research on epidemics for distributed systems also aims at a completely different goal: whereas health organizations will do their utmost best to prevent infectious diseases from spreading across large groups of people, designers of epidemic algorithms for distributed systems will try to "infect" all nodes with new information as fast as possible.

Using the terminology from epidemics, a node that is part of a distributed system is called infected if it holds data that it is willing to spread to other nodes. A node that has not yet seen this data is called susceptible. Finally, an updated node that is not willing or able to spread its data is said to be removed. Note that we assume we can distinguish old from new data, for example, because it has been timestamped or versioned. In this light, nodes are also said to spread updates.

A popular propagation model is that of anti-entropy. In this model, a node P picks another node Q at random, and subsequently exchanges updates with Q. There are three approaches to exchanging updates:

1. P only pushes its own updates to Q
2. P only pulls in new updates from Q
3. P and Q send updates to each other (i.e., a push-pull approach)

When it comes to rapidly spreading updates, only pushing updates turns out to be a bad choice. Intuitively, this can be understood as follows. First, note that in a pure push-based approach, updates can be propagated only by infected nodes. However, if many nodes are infected, the probability of each one selecting a susceptible node is relatively small. Consequently, chances are that a particular node remains susceptible for a long period simply because it is not selected by an infected node.

In contrast, the pull-based approach works much better when many nodes are infected. In that case, spreading updates is essentially triggered by susceptible nodes. Chances are large that such a node will contact an infected one to subsequently pull in the updates and become infected as well.

It can be shown that if only a single node is infected, updates will rapidly spread across all nodes using either form of anti-entropy, although push-pull remains the best strategy (Jelasity et al., 2005a). Define a round as spanning a period in which every node will at least once have taken the initiative to exchange updates with a randomly chosen other node. It can then be shown that the number of rounds to propagate a single update to all nodes takes $O(\log(N))$ rounds, where N is the number of nodes in the system. This indicates indeed that propagating updates is fast, but above all scalable.

One specific variant of this approach is called rumor spreading, or simply gossiping. It works as follows. If node P has just been updated for data item x, it contacts an arbitrary other node Q and tries to push the update to Q. However, it is possible that Q was already updated by another node. In that case, P may lose interest in spreading the update any further, say with probability $1/k$. In other words, it then becomes removed.

Gossiping is completely analogous to real life. When Bob has some hot news to spread around, he may phone his friend Alice telling her all about it. Alice, like Bob, will be really excited to spread the gossip to her friends as well. However, she will become disappointed when phoning a friend, say Chuck, only to hear that the news has already reached him. Chances are that she will stop phoning other friends, for what good is it if they already know?

Gossiping turns out to be an excellent way of rapidly spreading news. However, it cannot guarantee that all nodes will actually be updated (Demers et al., 1987). It can be shown that when there is a large number of nodes that participate in the epidemics, the fraction *s* of nodes that will remain ignorant of an update, that is, remain susceptible, satisfies the equation:

s = *e*-(*k* + \)(1-.1')

Fig. 4-32 shows *in* (s) as a function of *k*. For example, if *k* = 4, *In (51)=-4.97,* so that *s* is less than 0.007, meaning that less than 0.7% of the nodes remain susceptible. Nevertheless, special measures are needed to guarantee that those nodes will also be updated. Combining anti-entropy with gossiping will do the trick.

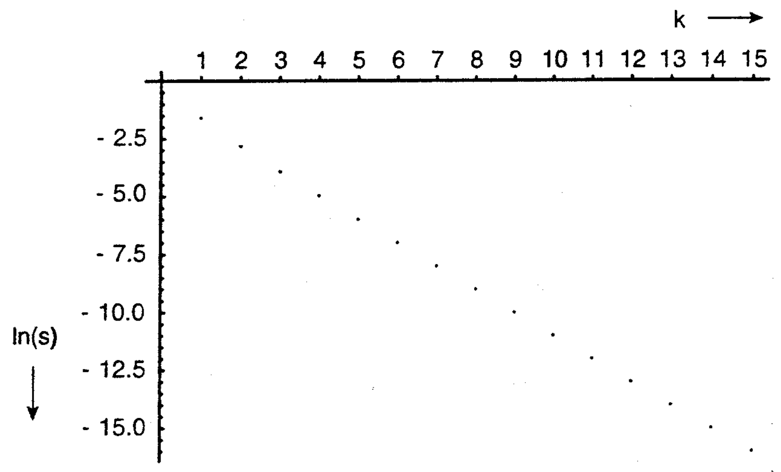


Figure 4-32. The relation between the fraction *s* of update-ignorant nodes and the parameter *k* in pure gossiping. The graph displays *lnis)* as a function of *k*.

One of the main advantages of epidemic algorithms is their scalability, due to the fact that the number of synchronizations between processes is relatively small compared to other propagation methods. For wide-area systems, Lin and Marzullo (1999) show that it makes sense to take the actual network topology into account to achieve better results. In their approach, nodes that are connected to only a few other nodes are contacted with a relatively high probability. The underlying assumption is that such nodes form a bridge to other remote parts of the network; therefore, they should be contacted as soon as possible. This approach is referred to as **directional** gossiping and comes in different variants.

This problem touches upon an important assumption that most epidemic solutions make, namely that a node can randomly select any other node to gossip with. This implies that, in principle, the complete set of nodes should be known to each member. In a large system, this assumption can never hold.

Fortunately, there is no need to have such a list. As we explained in Chap. 2, maintaining a partial view that is more or less continuously updated will organize the collection of nodes into a random graph. By regularly updating the partial view of each node, random selection is no longer a problem.

Removing Data

Epidemic algorithms are extremely good for spreading updates. However, they have a rather strange side-effect: spreading the *deletion* of a data item is hard. The essence of the problem lies in the fact that deletion of a data item destroys all information on that item. Consequently, when a data item is simply removed from a node, that node will eventually receive old copies of the data item and interpret those as updates on something it did not have before.

The trick is to record the deletion of a data item as just another update, and keep a record of that deletion. In this way, old copies will not be interpreted as something new, but merely treated as versions that have been updated by a delete operation. The recording of a deletion is done by spreading death certificates.

Of course, the problem with death certificates is that they should eventually be cleaned up, or otherwise each node will gradually build a huge local database of historical information on deleted data items that is otherwise not used. Demers et al. (1987) propose to use what they call dormant death certificates. Each death certificate is timestamped when it is created. If it can be assumed that updates propagate to all nodes within a known finite time, then death certificates can be removed after this maximum propagation time has elapsed.

However, to provide hard guarantees that deletions are indeed spread to all nodes, only a very few nodes maintain dormant death certificates that are never thrown away. Assume node P has such a certificate for data item x. If by any chance an obsolete update for x reaches P, P will react by simply spreading the death certificate for x again.

Applications

To finalize this presentation, let us take a look at some interesting applications of epidemic protocols. We already mentioned spreading updates, which is perhaps the most widely-deployed application. Also, in Chap. 2 we discussed how providing positioning information about nodes can assist in constructing specific topologies. In the same light, gossiping can be used to discover nodes that have a few outgoing wide-area links, to subsequently apply directional gossiping as we mentioned above.

Another interesting application area is simply collecting, or actually aggregating information (Jelasity et al., 2005b). Consider the following information exchange. Every node *i* initially chooses an arbitrary number, say x_i. When node *i* contacts node *j*, they each update their value as:

$$x_i, x_j \leftarrow (x_i + x_j) / 2$$

Obviously. after this exchange, both *i* and *j* will have the same value. In fact. it is not difficult to see that eventually all nodes will have the same value, namely, the average of all initial values. Propagation speed is again exponential..

What use does computing the average have? Consider the situation that all nodes *i* have set x_i to zero, except for x_1, which has set it to 1:

$$x_i \leftarrow \begin{cases} 1 & \text{if } i = 1 \\ 0 & \text{if } i > 1 \end{cases}$$

If there N nodes, then eventually each node will compute the average, which is $1/N$. As a consequence, every node *i* can estimate the size of the system as being $1/x_i$. This information alone can be used to dynamically adjust various system parameters. For example, the size of the partial view (i.e., the number of neighbors that each nodes keeps track of) should be dependent on the total number of participating nodes. Knowing this number will allow a node to dynamically adjust the size of its partial view. Indeed, this can be viewed as a property of self-management.

Computing the average may prove to be difficult when nodes regularly join and leave the system. One practical solution to this problem is to introduce epochs. Assuming that node 1 is stable, it simply starts a new epoch now and then. When node *i* sees a new epoch for the first time, it resets its own variable x_i to zero and starts computing the average again.

Of course, other results can also be computed. For example, instead of having a fixed node (x_1) start the computation of the average, we can easily pick a random node as follows. Every node *i* initially sets x_i to a random number from the same interval, say [0,1], and also stores it permanently as m_i. Upon an exchange between nodes *i* and *j*, each change their value to:

$$x_i, x_j \leftarrow max(x_i, x_j)$$

Each node *i* for which $m_i < x_i$ will lose the competition for being the initiator in starting the computation of the average. In the end, there will be a single winner. Of course, although it is easy to conclude that a node has lost, it is much more difficult to decide that it has won, as it remains uncertain whether all results have come in. The solution to this problem is to be optimistic: a node always assumes it is the winner until proven otherwise. At that point, it simply resets the variable it is using for computing the average to zero. Note that by now, several different computations (in our example computing a maximum and computing an average) may be executing concurrently.

4.6 SU~IMARY

Having powerful and flexible facilities for communication between processes is essential for any distributed system. In traditional network applications, communication is often based on the low-level message-passing primitives offered by the transport layer. An important issue in middleware systems is to offer a higher level of abstraction that will make it easier to express communication between processes than the support offered by the interface to the transport layer.

One of the most widely used abstractions is the Remote Procedure Call (RPC). The essence of an RPC is that a service is implemented by means of a procedure, of which the body is executed at a server. The client is offered only the signature of the procedure, that is, the procedure's name along with its parameters. When the client calls the procedure, the client-side implementation, called a stub, takes care of wrapping the parameter values into a message and sending that to the server. The latter calls the actual procedure and returns the results, again in a message. The client's stub extracts the result values from the return message and passes it back to the calling client application.

RPCs offer synchronous communication facilities, by which a client is blocked until the server has sent a reply. Although variations of either mechanism exist by which this strict synchronous model is relaxed, it turns out that general-purpose, high-level message-oriented models are often more convenient.

In message-oriented models, the issues are whether or not communication is persistent, and whether or not communication is synchronous. The essence of persistent communication is that a message that is submitted for transmission, is stored by the communication system as long as it takes to deliver it. In other words, neither the sender nor the receiver needs to be up and running for message transmission to take place. In transient communication, no storage facilities are offered, so that the receiver must be prepared to accept the message when it is sent.

In asynchronous communication, the sender is allowed to continue immediately after the message has been submitted for transmission, possibly before it has even been sent. In synchronous communication, the sender is blocked at least until a message has been received. Alternatively, the sender may be blocked until message delivery has taken place or even until the receiver has responded as with RPCs.

Message-oriented middleware models generally offer persistent asynchronous communication, and are used where RPCs are not appropriate. They are often used to assist the integration of (widely-dispersed) collections of databases into large-scale information systems. Other applications include e-mail and workflow.

A very different form of communication is that of streaming, in which the issue is whether or not two successive messages have a temporal relationship. In continuous data streams, a maximum end-to-end delay is specified for each message. In addition, it is also required that messages are sent subject to a minimum end-to-end delay. Typical examples of such continuous data streams are video and audio streams. Exactly what the temporal relations are, or what is expected from the underlying communication subsystem in terms of quality of service is often difficult to specify and to implement. A complicating factor is the role of jitter. Even if the average performance is acceptable, substantial variations in delivery time may lead to unacceptable performance.

Finally, an important class of communication protocols in distributed systems is multicasting. The basic idea is to disseminate information from one sender to multiple receivers. We have discussed two different approaches. First, multicasting can be achieved by setting up a tree from the sender to the receivers. Considering that it is now well understood how nodes can self-organize into peer-to-peer system, solutions have also appeared to dynamically set up trees in a decentralized fashion.

Another important class of dissemination solutions deploys epidemic protocols. These protocols have proven to be very simple, yet extremely robust. Apart from merely spreading messages, epidemic protocols can also be efficiently deployed for aggregating information across a large distributed system.

PROBLEMS

1. In many layered protocols, each layer has its own header. Surely it would be more efficient to have a single header at the front of each message with all the control in it than all these separate headers. Why is this not done?
2. Why are transport-level communication services often inappropriate for building distributed applications?
3. A reliable multicast service allows a sender to reliably pass messages to a collection of receivers. Does such a service belong to a middleware layer, or should it be part of a lower-level layer?
4. Consider a procedure *incr* with two integer parameters. The procedure adds one to each parameter. Now suppose that it is called with the same variable twice, for example. as *incr(i, i)*. If *i* is initially 0. what value will it have afterward if call-by-reference is used? How about if copy/restore is used?
5. C has a construction called a union, in which a field of a record (called a struct in C) can hold anyone of several alternatives. At run time, there is no sure-fire way to tell which one is in there. Does this feature of C have any implications for remote procedure call? Explain your answer.
6. One way to handle parameter conversion in RPC systems is to have each machine send parameters in its native representation, with the other one doing the translation, if need be. The native system could be indicated by a code in the first byte. However, since locating the first byte in the first word is precisely the problem, can this work?

7. Assume a client calls an asynchronous RPC to a server, and subsequently waits until the server returns a result using another asynchronous RPC. Is this approach the same as letting the client execute a normal RPC? What if we replace the asynchronous RPCs with asynchronous RPCs?

8. Instead of letting a server register itself with a daemon as in DCE, we could also choose to always assign it the same end point. That end point can then be used in references to objects in the server's address space. What is the main drawback of this scheme?

9. Would it be useful also to make a distinction between static and dynamic RPCs?

10. Describe how connectionless communication between a client and a server proceeds when using sockets.

11. Explain the difference between the primitives MPLbsend and MPLisend in MPI.

12. Suppose that you could make use of only transient asynchronous communication primitives, including only an asynchronous receive primitive. How would you implement primitives for transient *synchronous* communication?

13. Suppose that you could make use of only transient synchronous communication primitives. How would you implement primitives for transient *asynchronous* communication?

14. Does it make sense to implement persistent asynchronous communication by means of RPCs?

15. In the text we stated that in order to automatically start a process to fetch messages from an input queue, a daemon is often used that monitors the input queue. Give an alternative implementation that does not make use of a daemon.

16. Routing tables in IBM WebSphere, and in many other message-queuing systems, are configured manually. Describe a simple way to do this automatically.

17. With persistent communication, a receiver generally has its own local buffer where messages can be stored when the receiver is not executing. To create such a buffer, we may need to specify its size. Give an argument why this is preferable, as well as one against specification of the size.

18. Explain why transient synchronous communication has inherent scalability problems, and how these could be solved.

19. Give an example where multicasting is also useful for discrete data streams.

20. Suppose that in a sensor network measured temperatures are not timestarnped by the sensor, but are immediately sent to the operator. Would it be enough to guarantee only a maximum end-to-end delay?

21. How could you guarantee a maximum end-to-end delay when a collection of computers is organized in a (logical or physical) ring?

22. How could you guarantee a minimum end-to-end delay when a collection of computers is organized in a (logical or physical) ring?

23. Despite that multicasting is technically feasible, there is very little support to deploy it in the Internet. The answer to this problem is to be sought in down-to-earth business models: no one really knows how to make money out of multicasting. What scheme can you invent?

24. Normally, application-level multicast trees are optimized with respect stretch, which is measured in terms of delay or hop counts. Give an example where this metric could lead to very poor trees.

25. When searching for files in an unstructured peer-to-peer system, it may help to restrict the search to nodes that have files similar to yours. Explain how gossiping can help to find those nodes.