DISTRIBUTED SYSTEMS

Concepts and Design

Fifth Edition

International Edition

George Coulouris
Jean Dollimore
Tim Kindberg
Gordon Blair

PEARSON

DISTRIBUTED SYSTEMS

Concepts and Design

Fifth Edition

This page intentionally left blank

DISTRIBUTED SYSTEMS

Concepts and Design

Fifth Edition

George Coulouris
Cambridge University

Jean Dollimore
formerly of Queen Mary, University of London

Tim Kindberg
matter 2 media

Gordon Blair
Lancaster University

International Edition contributions by:
Arup Kumar Bhattacharjee
RCC Institute of Information Technology, Kolkata

Soumen Mukherjee
RCC Institute of Information Technology, Kolkata

Addison-Wesley

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montréal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editorial Director: Marcia Horton
Editor-in-Chief: Michael Hirsch
Executive Editor: Matt Goldstein
Editorial Assistant: Chelsea Bell
Vice President, Marketing: Patrice Jones
Marketing Manager: Yezan Alayan
Marketing Coordinator: Kathryn Ferranti
Vice President, Production: Vince O'Brien
Managing Editor: Jeff Holcomb
Senior Production Project Manager: Marilyn Lloyd
Publisher, International Edition: Angshuman Chakraborty
Acquisitions Editor, International Edition: Arunabha Deb
Publishing Assistant, International Edition: Shokhi Shah
Senior Operations Supervisor: Alan Fischer
Manufacturing Buyer: Lisa McDowell
Art Director: Jayne Conte
Cover Designer: Suzanne Duda
Cover Image: Sky: © amygdala_imagery; Kite: © Alamy; Mobile phone: © yasinguneysu/iStock
Media Editor: Daniel Sandin
Media Project Manager: Wanda Rockwell
Cover Printer: Lehigh-Phoenix Color

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsoninternationaleditions.com

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

10 9 8 7 6 5 4 3 2 1
14 13 12 11 10

Typeset in Times New Roman by authors using FrameMaker
Printed and bound by Edwards Brothers in The United States of America

The publisher's policy is to use paper manufactured from sustainable forests.

ISBN 10: 0-273-76059-9
ISBN 13: 978-0-273-76059-7

Addison-Wesley
is an imprint of

PEARSON

www.pearsonhighered.com

CONTENTS

PREFACE

This fifth edition of our textbook appears at a time when the Internet and the Web continue to grow and have an impact on every aspect of our society. For example, the introductory chapter of the book notes their impact on application areas as diverse as finance and commerce, arts and entertainment and the emergence of the information society more generally. It also highlights the very demanding requirements of application domains such as web search and multiplayer online games. From a distributed systems perspective, these developments are placing substantial new demands on the underlying system infrastructure in terms of the range of applications and the workloads and system sizes supported by many modern systems. Important trends include the increasing diversity and ubiquity of networking technologies (including the increasing importance of wireless networks), the inherent integration of mobile and ubiquitous computing elements into the distributed systems infrastructure

New to the fifth edition

New chapters:

Indirect Communication: Covering group communication, publish-subscribe and case studies on JavaSpaces, JMS, WebSphere and Message Queues.

Distributed Objects and Components: Covering component-based middleware and case studies on Enterprise JavaBeans, Fractal and CORBA.

Designing Distributed Systems: Devoted to a major new case study on the Google infrastructure.

Topics added to other chapters: Cloud computing, network virtualization, operating system virtualization, message passing interface, unstructured peer-to-peer, tuple spaces, loose coupling in relation to web services.

Other new case studies: Skype, Gnutella, TOTA, L^{2}imbo, BitTorrent, End System Multicast.

See the table on page 15 for further details of the changes.

(leading to radically different physical architectures), the need to support multimedia services and the emergence of the cloud computing paradigm, which challenges our perspective of distributed systems services.

The book aims to provide an understanding of the principles on which the Internet and other distributed systems are based; their architecture, algorithms and design; and how they meet the demands of contemporary distributed applications. We begin with a set of seven chapters that together cover the building blocks for a study of distributed systems. The first two chapters provide a conceptual overview of the subject, outlining the characteristics of distributed systems and the challenges that must be addressed in their design: scalability, heterogeneity, security and failure handling being the most significant. These chapters also develop abstract models for understanding process interaction, failure and security. They are followed by other foundational chapters devoted to the study of networking, interprocess communication, remote invocation, indirect communication and operating system support.

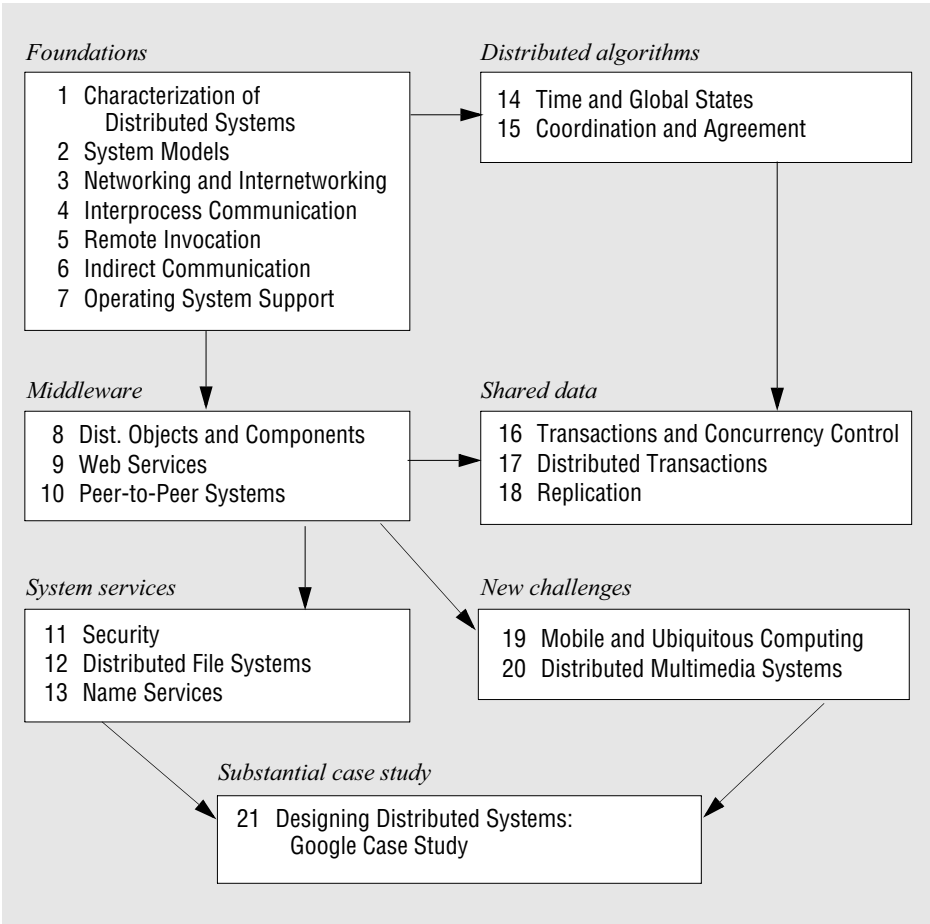
The next set of chapters covers the important topic of middleware, examining different approaches to supporting distributed applications including distributed objects and components, web services and alternative peer-to-peer solutions. We then cover the well-established topics of security, distributed file systems and distributed naming before moving on to important data-related aspects including distributed transactions and data replication. Algorithms associated with all these topics are covered as they arise and also in separate chapters devoted to timing, coordination and agreement.

The book culminates in chapters that address the emerging areas of mobile and ubiquitous computing and distributed multimedia systems before presenting a substantial case study focusing on the design and implementation of the distributed systems infrastructure that supports Google both in terms of core search functionality and the increasing range of additional services offered by Google (for example, Gmail and Google Earth). This last chapter has an important role in illustrating how all the architectural concepts, algorithms and technologies introduced in the book can come together in a coherent overall design for a given application domain.

Purposes and readership

The book is intended for use in undergraduate and introductory postgraduate courses. It can equally be used for self-study. We take a top-down approach, addressing the issues to be resolved in the design of distributed systems and describing successful approaches in the form of abstract models, algorithms and detailed case studies of widely used systems. We cover the field in sufficient depth and breadth to enable readers to go on to study most research papers in the literature on distributed systems.

We aim to make the subject accessible to students who have a basic knowledge of object-oriented programming, operating systems and elementary computer architecture. The book includes coverage of those aspects of computer networks relevant to distributed systems, including the underlying technologies for the Internet and for wide area, local area and wireless networks. Algorithms and interfaces are presented throughout the book in Java or, in a few cases, ANSI C. For brevity and clarity of presentation, a form of pseudo-code derived from Java/C is also used.



Organization of the book

The diagram shows the book's chapters under seven main topic areas. It is intended to provide a guide to the book's structure and to indicate recommended navigation routes for instructors wishing to provide, or readers wishing to achieve, understanding of the various subfields of distributed system design.

References

The existence of the World Wide Web has changed the way in which a book such as this can be linked to source material, including research papers, technical specifications and standards. Many of the source documents are now available on the Web; some are available only there. For reasons of brevity and readability, we employ a special form of reference to web material that loosely resembles a URL: references such as [www.omg.org] and [www.rsasecurity.com I] refer to documentation that is available

only on the Web. They can be looked up in the reference list at the end of the book, but the full URLs are given only in an online version of the reference list at the book's web site, www.cdk5.net/refs where they take the form of clickable links. Both versions of the reference list include a more detailed explanation of this scheme.

Changes relative to the fourth edition

Before embarking on the writing of this new edition, we carried out a survey of teachers who used the fourth edition. From the results, we identified the new material required and a number of changes to be made. In addition, we recognized the increasing diversity of distributed systems, particularly in terms of the range of architectural approaches available to distributed systems developers today. This required significant changes to the book, especially in the earlier (foundational) chapters.

Overall, this led to our writing three entirely new chapters, making substantial changes to a number of other chapters and making numerous insertions throughout the book to fold in new material. Many of the chapters have been changed to reflect new information that has become available about the systems described. These changes are summarized in the table below. To help teachers who have used the fourth edition, wherever possible we have preserved the structure adopted from the previous edition. Where material has been removed, we have placed this on our companion web site together with material removed from previous editions. This includes the case studies on ATM, interprocess communication in UNIX, CORBA (a shortened version of which remains in Chapter 8), the Jini distributed events specification and Grid middleware (featuring OGSA and the Globus toolkit), as well as the chapter on distributed shared memory (a brief summary of which is now included in Chapter 6).

Some of the chapters in the book, such as the new chapter on indirect communication (Chapter 6), cover a lot of material. Teachers may elect to cover the broad spectrum before choosing two or three techniques to examine in more detail (for example, group communication, given its foundational role, and publish-subscribe or message queues, given their prevalence in commercial distributed systems).

The chapter ordering has been changed to accommodate the new material and to reflect changes in the relative importance of certain topics. For a full understanding of some topics readers may find it necessary to follow a forward reference. For example, there is material in Chapter 9 on XML security techniques that will make better sense once the sections that it references in Chapter 11 Security have been absorbed.

Acknowledgements

We are very grateful to the following teachers who participated in our survey: Guohong Cao, Jose Fortes, Bahram Khalili, George Blank, Jinsong Ouyang, JoAnne Holliday, George K. Thiruvathukal, Joel Wein, Tao Xie and Xiaobo Zhou.

We would like to thank the following people who reviewed the new chapters or provided other substantial help: Rob Allen, Roberto Baldoni, John Bates, Tom Berson, Lynne Blair, Geoff Coulson, Paul Grace, Andrew Herbert, David Hutchison, Laurent Mathy, Rajiv Ramdhany, Richard Sharp, Jean-Bernard Stefani, Rip Sohan, Francois

New chapters:	
6 Indirect Communication	Includes events and notification from 4th edition.
8 Distributed Objects and Components	Includes a precised version of the CORBA case study from the 4th edition.
21 Designing Distributed Systems	Includes a major new case study on Google
Chapters which have undergone substantial changes:	
1 Characterization of DS	*Significant restructuring of material* New Section 1.2: Examples of distributed systems Section 1.3.4: Cloud computing introduced
2 System Models	*Significant restructuring of material* New Section 2.2: Physical models Section 2.3: Major rewrite to reflect new book content and associated architectural perspectives
4 Interprocess Communication	*Several updates* Client-server communication moved to Chapter 5 New Section 4.5: Network virtualization (includes case study on Skype) New Section 4.6: Case study on MPI Case study on IPC in UNIX removed
5 Remote Invocation	*Significant restructuring of material* Client-server communication moved to here Progression introduced from client-server communication through RPC to RMI Events and notification moved to Chapter 6
Chapters to which new material has been added/removed, but without structural changes:	
3 Networking and Internetworking	*Several updates* Section 3.5: material on ATM removed
7 Operating System Support	New Section 7.7: OS virtualization
9 Web Services	Section 9.2: Discussion added on loose coupling
10 Peer-to-Peer Systems	New Section 10.5.3: Unstructured peer-to-peer (including a new case study on Gnutella)
15 Coordination and Agreement	Material on group communication moved to Ch. 6
18 Replication	Material on group communication moved to Ch. 6
19 Mobile and Ubiquitous Computing	Section 19.3.1: New material on tuple spaces (TOTA and L^2imbo)
20 Distributed Multimedia Systems	Section 20.6: New case studies added on BitTorrent and End System Multicast
The remaining chapters have received only minor modifications.	

Taiani, Peter Triantafillou, Gareth Tyson and the late Sir Maurice Wilkes. We would also like to thank the staff at Google who provided insights into the design rationale for Google Infrastructure, namely: Mike Burrows, Tushar Chandra, Walfredo Cirne, Jeff Dean, Sanjay Ghemawat, Andrea Kirmse and John Reumann.

Our copy editor, Rachel Head also provided outstanding support.

Web site

As before, we continue to maintain a web site with a wide range of material designed to assist teachers and readers. This web site can be accessed via the URL:

www.cdk5.net

The web site includes:

Instructor's Guide: We provide supporting material for teachers comprising:

- complete artwork of the book available as PowerPoint files;
- chapter-by-chapter teaching hints;
- solutions to the exercises, protected by a password available only to teachers.

Instructor resources for the International Edition are available at www.pearsoninternationaleditions.com/coulouris

Reference list: The list of references that can be found at the end of the book is replicated at the web site. The web version of the reference list includes active links for material that is available online.

Errata list: A list of known errors in the book is maintained, with corrections. The errors will be corrected when new impressions are printed and a separate errata list will be provided for each impression. (Readers are encouraged to report any apparent errors they encounter to the email address below.)

Supplementary material: We maintain a set of supplementary material for each chapter. This consists of source code for the programs in the book and relevant reading material that was present in previous editions of the book but was removed for reasons of space. References to this supplementary material appear in the book with links such as www.cdk5.net/ipc (the URL for supplementary material relating to the Interprocess Communication chapter). Two entire chapters from the 4th edition are not present in this one; they can be accessed at the URLs:

CORBA Case Study	www.cdk5.net/corba
Distributed Shared Memory	www.cdk5.net/dsm

George Coulouris
Jean Dollimore
Tim Kindberg
Gordon Blair
London, Bristol and Lancaster, 2011
authors@cdk5.net

1

CHARACTERIZATION OF DISTRIBUTED SYSTEMS

A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages. This definition leads to the following especially significant characteristics of distributed systems: concurrency of components, lack of a global clock and independent failures of components.

We look at several examples of modern distributed applications, including web search, multiplayer online games and financial trading systems, and also examine the key underlying trends driving distributed systems today: the pervasive nature of modern networking, the emergence of mobile and ubiquitous computing, the increasing importance of distributed multimedia systems, and the trend towards viewing distributed systems as a utility. The chapter then highlights resource sharing as a main motivation for constructing distributed systems. Resources may be managed by servers and accessed by clients or they may be encapsulated as objects and accessed by other client objects.

The challenges arising from the construction of distributed systems are the heterogeneity of their components, openness (which allows components to be added or replaced), security, scalability – the ability to work well when the load or the number of users increases – failure handling, concurrency of components, transparency and providing quality of service. Finally, the Web is discussed as an example of a large-scale distributed system and its main features are introduced.

1.1 Introduction

Networks of computers are everywhere. The Internet is one, as are the many networks of which it is composed. Mobile phone networks, corporate networks, factory networks, campus networks, home networks, in-car networks – all of these, both separately and in combination, share the essential characteristics that make them relevant subjects for study under the heading *distributed systems*. In this book we aim to explain the characteristics of networked computers that impact system designers and implementors and to present the main concepts and techniques that have been developed to help in the tasks of designing and implementing systems that are based on them.

We define a distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages. This simple definition covers the entire range of systems in which networked computers can usefully be deployed.

Computers that are connected by a network may be spatially separated by any distance. They may be on separate continents, in the same building or in the same room. Our definition of distributed systems has the following significant consequences:

Concurrency: In a network of computers, concurrent program execution is the norm. I can do my work on my computer while you do your work on yours, sharing resources such as web pages or files when necessary. The capacity of the system to handle shared resources can be increased by adding more resources (for example. computers) to the network. We will describe ways in which this extra capacity can be usefully deployed at many points in this book. The coordination of concurrently executing programs that share resources is also an important and recurring topic.

No global clock: When programs need to cooperate they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clocks – there is no single global notion of the correct time. This is a direct consequence of the fact that the *only* communication is by sending messages through a network. Examples of these timing problems and solutions to them will be described in Chapter 14.

Independent failures: All computer systems can fail, and it is the responsibility of system designers to plan for the consequences of possible failures. Distributed systems can fail in new ways. Faults in the network result in the isolation of the computers that are connected to it, but that doesn't mean that they stop running. In fact, the programs on them may not be able to detect whether the network has failed or has become unusually slow. Similarly, the failure of a computer, or the unexpected termination of a program somewhere in the system (a *crash*), is not immediately made known to the other components with which it communicates. Each component of the system can fail independently, leaving the others still running. The consequences of this characteristic of distributed systems will be a recurring theme throughout the book.

The prime motivation for constructing and using distributed systems stems from a desire to share resources. The term 'resource' is a rather abstract one, but it best characterizes the range of things that can usefully be shared in a networked computer system. It

extends from hardware components such as disks and printers to software-defined entities such as files, databases and data objects of all kinds. It includes the stream of video frames that emerges from a digital video camera and the audio connection that a mobile phone call represents.

The purpose of this chapter is to convey a clear view of the nature of distributed systems and the challenges that must be addressed in order to ensure that they are successful. Section 1.2 gives some illustrative examples of distributed systems, with Section 1.3 covering the key underlying trends driving recent developments. Section 1.4 focuses on the design of resource-sharing systems, while Section 1.5 describes the key challenges faced by the designers of distributed systems: heterogeneity, openness, security, scalability, failure handling, concurrency, transparency and quality of service. Section 1.6 presents a detailed case study of one very well known distributed system, the World Wide Web, illustrating how its design supports resource sharing.

1.2 Examples of distributed systems

The goal of this section is to provide motivational examples of contemporary distributed systems illustrating both the pervasive role of distributed systems and the great diversity of the associated applications.

As mentioned in the introduction, networks are everywhere and underpin many everyday services that we now take for granted: the Internet and the associated World Wide Web, web search, online gaming, email, social networks, eCommerce, etc. To illustrate this point further, consider Figure 1.1, which describes a selected range of key commercial or social application sectors highlighting some of the associated established or emerging uses of distributed systems technology.

As can be seen, distributed systems encompass many of the most significant technological developments of recent years and hence an understanding of the underlying technology is absolutely central to a knowledge of modern computing. The figure also provides an initial insight into the wide range of applications in use today, from relatively localized systems (as found, for example, in a car or aircraft) to global-scale systems involving millions of nodes, from data-centric services to processor-intensive tasks, from systems built from very small and relatively primitive sensors to those incorporating powerful computational elements, from embedded systems to ones that support a sophisticated interactive user experience, and so on.

We now look at more specific examples of distributed systems to further illustrate the diversity and indeed complexity of distributed systems provision today.

1.2.1 Web search

Web search has emerged as a major growth industry in the last decade, with recent figures indicating that the global number of searches has risen to over 10 billion per calendar month. The task of a web search engine is to index the entire contents of the World Wide Web, encompassing a wide range of information styles including web pages, multimedia sources and (scanned) books. This is a very complex task, as current estimates state that the Web consists of over 63 billion pages and one trillion unique web

Figure 1.1 Selected application domains and associated networked applications

Finance and commerce	The growth of eCommerce as exemplified by companies such as Amazon and eBay, and underlying payments technologies such as PayPal; the associated emergence of online banking and trading and also complex information dissemination systems for financial markets.
The information society	The growth of the World Wide Web as a repository of information and knowledge; the development of web search engines such as Google and Yahoo to search this vast repository; the emergence of digital libraries and the large-scale digitization of legacy information sources such as books (for example, Google Books); the increasing significance of user-generated content through sites such as YouTube, Wikipedia and Flickr; the emergence of social networking through services such as Facebook and MySpace.
Creative industries and entertainment	The emergence of online gaming as a novel and highly interactive form of entertainment; the availability of music and film in the home through networked media centres and more widely in the Internet via downloadable or streaming content; the role of user-generated content (as mentioned above) as a new form of creativity, for example via services such as YouTube; the creation of new forms of art and entertainment enabled by emergent (including networked) technologies.
Healthcare	The growth of health informatics as a discipline with its emphasis on online electronic patient records and related issues of privacy; the increasing role of telemedicine in supporting remote diagnosis or more advanced services such as remote surgery (including collaborative working between healthcare teams); the increasing application of networking and embedded systems technology in assisted living, for example for monitoring the elderly in their own homes.
Education	The emergence of e-learning through for example web-based tools such as virtual learning environments; associated support for distance learning; support for collaborative or community-based learning.
Transport and logistics	The use of location technologies such as GPS in route finding systems and more general traffic management systems; the modern car itself as an example of a complex distributed system (also applies to other forms of transport such as aircraft); the development of web-based map services such as MapQuest, Google Maps and Google Earth.
Science	The emergence of the Grid as a fundamental technology for eScience, including the use of complex networks of computers to support the storage, analysis and processing of (often very large quantities of) scientific data; the associated use of the Grid as an enabling technology for worldwide collaboration between groups of scientists.
Environmental management	The use of (networked) sensor technology to both monitor and manage the natural environment, for example to provide early warning of natural disasters such as earthquakes, floods or tsunamis and to co-ordinate emergency response; the collation and analysis of global environmental parameters to better understand complex natural phenomena such as climate change.

addresses. Given that most search engines analyze the entire web content and then carry out sophisticated processing on this enormous database, this task itself represents a major challenge for distributed systems design.

Google, the market leader in web search technology, has put significant effort into the design of a sophisticated distributed system infrastructure to support search (and indeed other Google applications and services such as Google Earth). This represents one of the largest and most complex distributed systems installations in the history of computing and hence demands close examination. Highlights of this infrastructure include:

- an underlying physical infrastructure consisting of very large numbers of networked computers located at data centres all around the world;
- a distributed file system designed to support very large files and heavily optimized for the style of usage required by search and other Google applications (especially reading from files at high and sustained rates);
- an associated structured distributed storage system that offers fast access to very large datasets;
- a lock service that offers distributed system functions such as distributed locking and agreement;
- a programming model that supports the management of very large parallel and distributed computations across the underlying physical infrastructure.

Further details on Google's distributed systems services and underlying communications support can be found in Chapter 21, a compelling case study of a modern distributed system in action.

1.2.2 Massively multiplayer online games (MMOGs)

Massively multiplayer online games offer an immersive experience whereby very large numbers of users interact through the Internet with a persistent virtual world. Leading examples of such games include Sony's EverQuest II and EVE Online from the Finnish company CCP Games. Such worlds have increased significantly in sophistication and now include, complex playing arenas (for example EVE, Online consists of a universe with over 5,000 star systems) and multifarious social and economic systems. The number of players is also rising, with systems able to support over 50,000 simultaneous online players (and the total number of players perhaps ten times this figure).

The engineering of MMOGs represents a major challenge for distributed systems technologies, particularly because of the need for fast response times to preserve the user experience of the game. Other challenges include the real-time propagation of events to the many players and maintaining a consistent view of the shared world. This therefore provides an excellent example of the challenges facing modern distributed systems designers.

A number of solutions have been proposed for the design of massively multiplayer online games:

- Perhaps surprisingly, the largest online game, EVE Online, utilises a *client-server* architecture where a single copy of the state of the world is maintained on a

centralized server and accessed by client programs running on players' consoles or other devices. To support large numbers of clients, the server is a complex entity in its own right consisting of a cluster architecture featuring hundreds of computer nodes (this client-server approach is discussed in more detail in Section 1.4 and cluster approaches are discussed in Section 1.3.4). The centralized architecture helps significantly in terms of the management of the virtual world and the single copy also eases consistency concerns. The goal is then to ensure fast response through optimizing network protocols and ensuring a rapid response to incoming events. To support this, the load is partitioned by allocating individual 'star systems' to particular computers within the cluster, with highly loaded star systems having their own dedicated computer and others sharing a computer. Incoming events are directed to the right computers within the cluster by keeping track of movement of players between star systems.

- Other MMOGs adopt more distributed architectures where the universe is partitioned across a (potentially very large) number of servers that may also be geographically distributed. Users are then dynamically allocated a particular server based on current usage patterns and also the network delays to the server (based on geographical proximity for example). This style of architecture, which is adopted by EverQuest, is naturally extensible by adding new servers.
- Most commercial systems adopt one of the two models presented above, but researchers are also now looking at more radical architectures that are not based on client-server principles but rather adopt completely decentralized approaches based on peer-to-peer technology where every participant contributes resources (storage and processing) to accommodate the game. Further consideration of peer-to-peer solutions is deferred until Chapters 2 and 10).

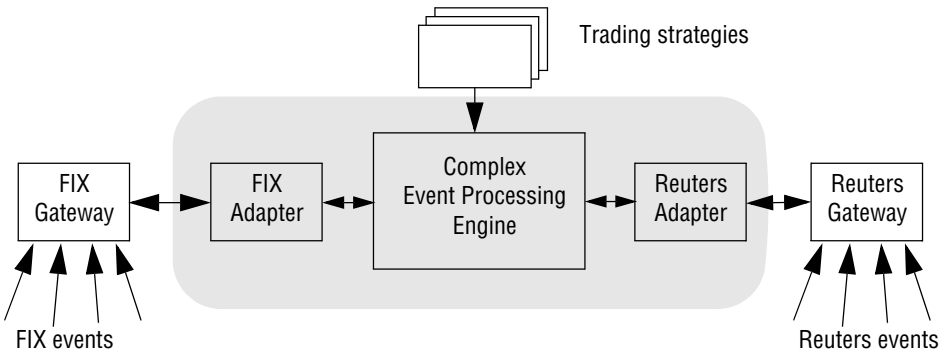
1.2.3 Financial trading

As a final example, we look at distributed systems support for financial trading markets. The financial industry has long been at the cutting edge of distributed systems technology with its need, in particular, for real-time access to a wide range of information sources (for example, current share prices and trends, economic and political developments). The industry employs automated monitoring and trading applications (see below).

Note that the emphasis in such systems is on the communication and processing of items of interest, known as *events* in distributed systems, with the need also to deliver events reliably and in a timely manner to potentially very large numbers of clients who have a stated interest in such information items. Examples of such events include a drop in a share price, the release of the latest unemployment figures, and so on. This requires a very different style of underlying architecture from the styles mentioned above (for example client-server), and such systems typically employ what are known as *distributed event-based systems*. We present an illustration of a typical use of such systems below and return to this important topic in more depth in Chapter 6.

Figure 1.2 illustrates a typical financial trading system. This shows a series of event feeds coming into a given financial institution. Such event feeds share the

Figure 1.2 An example financial trading system



following characteristics. Firstly, the sources are typically in a variety of formats, such as Reuters market data events and FIX events (events following the specific format of the Financial Information eXchange protocol), and indeed from different event technologies, thus illustrating the problem of heterogeneity as encountered in most distributed systems (see also Section 1.5.1). The figure shows the use of adapters which translate heterogeneous formats into a common internal format. Secondly, the trading system must deal with a variety of event streams, all arriving at rapid rates, and often requiring real-time processing to detect patterns that indicate trading opportunities. This used to be a manual process but competitive pressures have led to increasing automation in terms of what is known as Complex Event Processing (CEP), which offers a way of composing event occurrences together into logical, temporal or spatial patterns.

This approach is primarily used to develop customized algorithmic trading strategies covering both buying and selling of stocks and shares, in particular looking for patterns that indicate a trading opportunity and then automatically responding by placing and managing orders. As an example, consider the following script:

```
WHEN
    MSFT price moves outside 2% of MSFT Moving Average
FOLLOWED-BY (
    MyBasket moves up by 0.5%
    AND
        HPQ's price moves up by 5%
        OR
        MSFT's price moves down by 2%
    )
)
ALL WITHIN
    any 2 minute time period
THEN
    BUY MSFT
    SELL HPQ
```

This script is based on the functionality provided by Apama [www.progress.com], a commercial product in the financial world originally developed out of research carried out at the University of Cambridge. The script detects a complex temporal sequence based on the share prices of Microsoft, HP and a basket of other share prices, resulting in decisions to buy or sell particular shares.

This style of technology is increasingly being used in other areas of financial systems including the monitoring of trading activity to manage risk (in particular, tracking exposure), to ensure compliance with regulations and to monitor for patterns of activity that might indicate fraudulent transactions. In such systems, events are typically intercepted and passed through what is equivalent to a compliance and risk firewall before being processed (see also the discussion of firewalls in Section 1.3.1 below).

1.3 Trends in distributed systems

Distributed systems are undergoing a period of significant change and this can be traced back to a number of influential trends:

- the emergence of pervasive networking technology;
- the emergence of ubiquitous computing coupled with the desire to support user mobility in distributed systems;
- the increasing demand for multimedia services;
- the view of distributed systems as a utility.

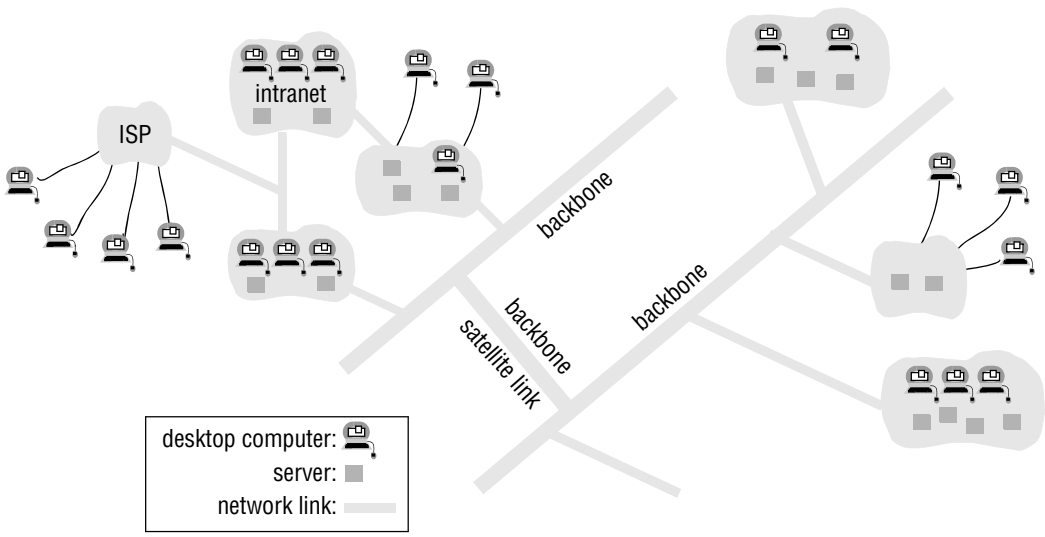
1.3.1 Pervasive networking and the modern Internet

The modern Internet is a vast interconnected collection of computer networks of many different types, with the range of types increasing all the time and now including, for example, a wide range of wireless communication technologies such as WiFi, WiMAX, Bluetooth (see Chapter 3) and third-generation mobile phone networks. The net result is that networking has become a pervasive resource and devices can be connected (if desired) at any time and in any place.

Figure 1.3 illustrates a typical portion of the Internet. Programs running on the computers connected to it interact by passing messages, employing a common means of communication. The design and construction of the Internet communication mechanisms (the Internet protocols) is a major technical achievement, enabling a program running anywhere to address messages to programs anywhere else and abstracting over the myriad of technologies mentioned above.

The Internet is also a very large distributed system. It enables users, wherever they are, to make use of services such as the World Wide Web, email and file transfer. (Indeed, the Web is sometimes incorrectly equated with the Internet.) The set of services is open-ended – it can be extended by the addition of server computers and new types of service. The figure shows a collection of intranets – subnetworks operated by companies and other organizations and typically protected by firewalls. The role of a *firewall* is to protect an intranet by preventing unauthorized messages from leaving or entering. A

Figure 1.3 A typical portion of the Internet



firewall is implemented by filtering incoming and outgoing messages. Filtering might be done by source or destination, or a firewall might allow only those messages related to email and web access to pass into or out of the intranet that it protects. Internet Service Providers (ISPs) are companies that provide broadband links and other types of connection to individual users and small organizations, enabling them to access services anywhere in the Internet as well as providing local services such as email and web hosting. The intranets are linked together by backbones. A *backbone* is a network link with a high transmission capacity, employing satellite connections, fibre optic cables and other high-bandwidth circuits.

Note that some organizations may not wish to connect their internal networks to the Internet at all. For example, police and other security and law enforcement agencies are likely to have at least some internal intranets that are isolated from the outside world (the most effective firewall possible – the absence of any physical connections to the Internet). Firewalls can also be problematic in distributed systems by impeding legitimate access to services when resource sharing between internal and external users is required. Hence, firewalls must often be complemented by more fine-grained mechanisms and policies, as discussed in Chapter 11.

The implementation of the Internet and the services that it supports has entailed the development of practical solutions to many distributed system issues (including most of those defined in Section 1.5). We shall highlight those solutions throughout the book, pointing out their scope and their limitations where appropriate.

1.3.2 Mobile and ubiquitous computing

Technological advances in device miniaturization and wireless networking have led increasingly to the integration of small and portable computing devices into distributed systems. These devices include:

- Laptop computers.
- Handheld devices, including mobile phones, smart phones, GPS-enabled devices, pagers, personal digital assistants (PDAs), video cameras and digital cameras.
- Wearable devices, such as smart watches with functionality similar to a PDA.
- Devices embedded in appliances such as washing machines, hi-fi systems, cars and refrigerators.

The portability of many of these devices, together with their ability to connect conveniently to networks in different places, makes *mobile computing* possible. Mobile computing is the performance of computing tasks while the user is on the move, or visiting places other than their usual environment. In mobile computing, users who are away from their 'home' intranet (the intranet at work, or their residence) are still provided with access to resources via the devices they carry with them. They can continue to access the Internet; they can continue to access resources in their home intranet; and there is increasing provision for users to utilize resources such as printers or even sales points that are conveniently nearby as they move around. The latter is also known as *location-aware* or *context-aware computing*. Mobility introduces a number of challenges for distributed systems, including the need to deal with variable connectivity and indeed disconnection, and the need to maintain operation in the face of device mobility (see the discussion on mobility transparency in Section 1.5.7).

Ubiquitous computing is the harnessing of many small, cheap computational devices that are present in users' physical environments, including the home, office and even natural settings. The term 'ubiquitous' is intended to suggest that small computing devices will eventually become so pervasive in everyday objects that they are scarcely noticed. That is, their computational behaviour will be transparently and intimately tied up with their physical function.

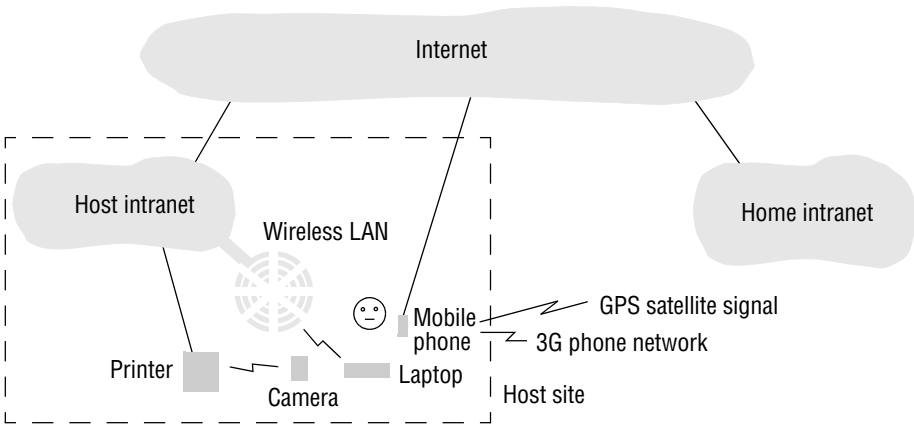
The presence of computers everywhere only becomes useful when they can communicate with one another. For example, it may be convenient for users to control their washing machine or their entertainment system from their phone or a 'universal remote control' device in the home. Equally, the washing machine could notify the user via a smart badge or phone when the washing is done.

Ubiquitous and mobile computing overlap, since the mobile user can in principle benefit from computers that are everywhere. But they are distinct, in general. Ubiquitous computing could benefit users while they remain in a single environment such as the home or a hospital. Similarly, mobile computing has advantages even if it involves only conventional, discrete computers and devices such as laptops and printers.

Figure 1.4 shows a user who is visiting a host organization. The figure shows the user's home intranet and the host intranet at the site that the user is visiting. Both intranets are connected to the rest of the Internet.

The user has access to three forms of wireless connection. Their laptop has a means of connecting to the host's wireless LAN. This network provides coverage of a

Figure 1.4 Portable and handheld devices in a distributed system



few hundred metres (a floor of a building, say). It connects to the rest of the host intranet via a gateway or access point. The user also has a mobile (cellular) telephone, which is connected to the Internet. The phone gives access to the Web and other Internet services, constrained only by what can be presented on its small display, and may also provide location information via built-in GPS functionality. Finally, the user carries a digital camera, which can communicate over a personal area wireless network (with range up to about 10m) with a device such as a printer.

With a suitable system infrastructure, the user can perform some simple tasks in the host site using the devices they carry. While journeying to the host site, the user can fetch the latest stock prices from a web server using the mobile phone and can also use the built-in GPS and route finding software to get directions to the site location. During the meeting with their hosts, the user can show them a recent photograph by sending it from the digital camera directly to a suitably enabled (local) printer or projector in the meeting room (discovered using a location service). This requires only the wireless link between the camera and printer or projector. And they can in principle send a document from their laptop to the same printer, utilizing the wireless LAN and wired Ethernet links to the printer.

This scenario demonstrates the need to support *spontaneous interoperation,* whereby associations between devices are routinely created and destroyed – for example by locating and using the host's devices, such as printers. The main challenge applying to such situations is to make interoperation fast and convenient (that is, spontaneous) even though the user is in an environment they may never have visited before. That means enabling the visitor's device to communicate on the host network, and associating the device with suitable local services – a process called *service discovery*.

Mobile and ubiquitous computing represent lively areas of research, and the various dimensions mentioned above are discussed in depth in Chapter 19.

1.3.3 Distributed multimedia systems

Another important trend is the requirement to support multimedia services in distributed systems. Multimedia support can usefully be defined as the ability to support a range of media types in an integrated manner. One can expect a distributed system to support the storage, transmission and presentation of what are often referred to as discrete media types, such as pictures or text messages. A distributed multimedia system should be able to perform the same functions for continuous media types such as audio and video; that is, it should be able to store and locate audio or video files, to transmit them across the network (possibly in real time as the streams emerge from a video camera), to support the presentation of the media types to the user and optionally also to share the media types across a group of users.

The crucial characteristic of continuous media types is that they include a temporal dimension, and indeed, the integrity of the media type is fundamentally dependent on preserving real-time relationships between elements of a media type. For example, in a video presentation it is necessary to preserve a given throughput in terms of frames per second and, for real-time streams, a given maximum delay or latency for the delivery of frames (this is one example of quality of service, discussed in more detail in Section 1.5.8).

The benefits of distributed multimedia computing are considerable in that a wide range of new (multimedia) services and applications can be provided on the desktop, including access to live or pre-recorded television broadcasts, access to film libraries offering video-on-demand services, access to music libraries, the provision of audio and video conferencing facilities and integrated telephony features including IP telephony or related technologies such as Skype, a peer-to-peer alternative to IP telephony (the distributed system infrastructure underpinning Skype is discussed in Section 4.5.2). Note that this technology is revolutionary in challenging manufacturers to rethink many consumer devices. For example, what is the core home entertainment device of the future – the computer, the television, or the games console?

Webcasting is an application of distributed multimedia technology. Webcasting is the ability to broadcast continuous media, typically audio or video, over the Internet. It is now commonplace for major sporting or music events to be broadcast in this way, often attracting large numbers of viewers (for example, the Live8 concert in 2005 attracted around 170,000 simultaneous users at its peak).

Distributed multimedia applications such as webcasting place considerable demands on the underlying distributed infrastructure in terms of:

- providing support for an (extensible) range of encoding and encryption formats, such as the MPEG series of standards (including for example the popular MP3 standard otherwise known as MPEG-1, Audio Layer 3) and HDTV;
- providing a range of mechanisms to ensure that the desired quality of service can be met;
- providing associated resource management strategies, including appropriate scheduling policies to support the desired quality of service;
- providing adaptation strategies to deal with the inevitable situation in open systems where quality of service cannot be met or sustained.

Further discussion of such mechanisms can be found in Chapter 20.

1.3.4 Distributed computing as a utility

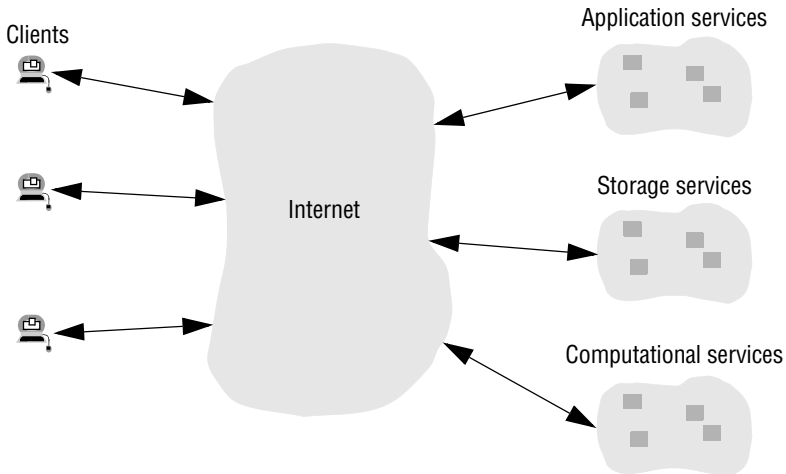
With the increasing maturity of distributed systems infrastructure, a number of companies are promoting the view of distributed resources as a commodity or utility, drawing the analogy between distributed resources and other utilities such as water or electricity. With this model, resources are provided by appropriate service suppliers and effectively rented rather than owned by the end user. This model applies to both physical resources and more logical services:

- Physical resources such as storage and processing can be made available to networked computers, removing the need to own such resources on their own. At one end of the spectrum, a user may opt for a remote storage facility for file storage requirements (for example, for multimedia data such as photographs, music or video) and/or for backups. Similarly, this approach would enable a user to rent one or more computational nodes, either to meet their basic computing needs or indeed to perform distributed computation. At the other end of the spectrum, users can access sophisticated *data centres* (networked facilities offering access to repositories of often large volumes of data to users or organizations) or indeed computational infrastructure using the sort of services now provided by companies such as Amazon and Google. Operating system virtualization is a key enabling technology for this approach, implying that users may actually be provided with services by a virtual rather than a physical node. This offers greater flexibility to the service supplier in terms of resource management (operating system virtualization is discussed in more detail in Chapter 7).
- Software services (as defined in Section 1.4) can also be made available across the global Internet using this approach. Indeed, many companies now offer a comprehensive range of services for effective rental, including services such as email and distributed calendars. Google, for example, bundles a range of business services under the banner Google Apps [www.google.com I]. This development is enabled by agreed standards for software services, for example as provided by web services (see Chapter 9).

The term *cloud computing* is used to capture this vision of computing as a utility. A cloud is defined as a set of Internet-based application, storage and computing services sufficient to support most users' needs, thus enabling them to largely or totally dispense with local data storage and application software (see Figure 1.5). The term also promotes a view of everything as a service, from physical or virtual infrastructure through to software, often paid for on a per-usage basis rather than purchased. Note that cloud computing reduces requirements on users' devices, allowing very simple desktop or portable devices to access a potentially wide range of resources and services.

Clouds are generally implemented on cluster computers to provide the necessary scale and performance required by such services. A *cluster computer* is a set of interconnected computers that cooperate closely to provide a single, integrated high-performance computing capability. Building on projects such as the NOW (Network of Workstations) Project at Berkeley [Anderson *et al*. 1995, now.cs.berkeley.edu] and Beowulf at NASA [www.beowulf.org], the trend is towards utilizing commodity hardware both for the computers and for the interconnecting networks. Most clusters

Figure 1.5 Cloud computing



consist of commodity PCs running a standard (sometimes cut-down) version of an operating system such as Linux, interconnected by a local area network. Companies such as HP, Sun and IBM offer blade solutions. *Blade servers* are minimal computational elements containing for example processing and (main memory) storage capabilities. A blade system consists of a potentially large number of blade servers contained within a blade enclosure. Other elements such as power, cooling, persistent storage (disks), networking and displays, are provided either by the enclosure or through virtualized solutions (discussed in Chapter 7). Through this solution, individual blade servers can be much smaller and also cheaper to produce than commodity PCs.

The overall goal of cluster computers is to provide a range of cloud services, including high-performance computing capabilities, mass storage (for example through data centres), and richer application services such as web search (Google, for example relies on a massive cluster computer architecture to implement its search engine and other services, as discussed in Chapter 21).

Grid computing (as discussed in Chapter 9, Section 9.7.2) can also be viewed as a form of cloud computing. The terms are largely synonymous and at times ill-defined, but Grid computing can generally be viewed as a precursor to the more general paradigm of cloud computing with a bias towards support for scientific applications.

1.4 Focus on resource sharing

Users are so accustomed to the benefits of resource sharing that they may easily overlook their significance. We routinely share hardware resources such as printers, data resources such as files, and resources with more specific functionality such as search engines.

Looked at from the point of view of hardware provision, we share equipment such as printers and disks to reduce costs. But of far greater significance to users is the sharing of the higher-level resources that play a part in their applications and in their everyday work and social activities. For example, users are concerned with sharing data in the form of a shared database or a set of web pages – not the disks and processors on which they are implemented. Similarly, users think in terms of shared resources such as a search engine or a currency converter, without regard for the server or servers that provide these.

In practice, patterns of resource sharing vary widely in their scope and in how closely users work together. At one extreme, a search engine on the Web provides a facility to users throughout the world, users who need never come into contact with one another directly. At the other extreme, in *computer-supported cooperative working* (CSCW), a group of users who cooperate directly share resources such as documents in a small, closed group. The pattern of sharing and the geographic distribution of particular users determines what mechanisms the system must supply to coordinate users' actions.

We use the term *service* for a distinct part of a computer system that manages a collection of related resources and presents their functionality to users and applications. For example, we access shared files through a file service; we send documents to printers through a printing service; we buy goods through an electronic payment service. The only access we have to the service is via the set of operations that it exports. For example, a file service provides *read*, *write* and *delete* operations on files.

The fact that services restrict resource access to a well-defined set of operations is in part standard software engineering practice. But it also reflects the physical organization of distributed systems. Resources in a distributed system are physically encapsulated within computers and can only be accessed from other computers by means of communication. For effective sharing, each resource must be managed by a program that offers a communication interface enabling the resource to be accessed and updated reliably and consistently.

The term *server* is probably familiar to most readers. It refers to a running program (a *process*) on a networked computer that accepts requests from programs running on other computers to perform a service and responds appropriately. The requesting processes are referred to as *clients,* and the overall approach is known as *client-server computing*. In this approach, requests are sent in messages from clients to a server and replies are sent in messages from the server to the clients. When the client sends a request for an operation to be carried out, we say that the client *invokes an operation* upon the server. A complete interaction between a client and a server, from the point when the client sends its request to when it receives the server's response, is called a *remote invocation*.

The same process may be both a client and a server, since servers sometimes invoke operations on other servers. The terms 'client' and 'server' apply only to the roles played in a single request. Clients are active (making requests) and servers are passive (only waking up when they receive requests); servers run continuously, whereas clients last only as long as the applications of which they form a part.

Note that while by default the terms 'client' and 'server' refer to *processes* rather than the computers that they execute upon, in everyday parlance those terms also refer to the computers themselves. Another distinction, which we shall discuss in Chapter 5,

is that in a distributed system written in an object-oriented language, resources may be encapsulated as objects and accessed by client objects, in which case we speak of a *client object* invoking a method upon a *server object*.

Many, but certainly not all, distributed systems can be constructed entirely in the form of interacting clients and servers. The World Wide Web, email and networked printers all fit this model. We discuss alternatives to client-server systems in Chapter 2.

An executing web browser is an example of a client. The web browser communicates with a web server, to request web pages from it. We consider the Web and its associated client-server architecture in more detail in Section 1.6.

1.5 Challenges

The examples in Section 1.2 are intended to illustrate the scope of distributed systems and to suggest the issues that arise in their design. In many of them, significant challenges were encountered and overcome. As the scope and scale of distributed systems and applications is extended the same and other challenges are likely to be encountered. In this section we describe the main challenges.

1.5.1 Heterogeneity

The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks. Heterogeneity (that is, variety and difference) applies to all of the following:

- networks;
- computer hardware;
- operating systems;
- programming languages;
- implementations by different developers.

Although the Internet consists of many different sorts of network (illustrated in Figure 1.3), their differences are masked by the fact that all of the computers attached to them use the Internet protocols to communicate with one another. For example, a computer attached to an Ethernet has an implementation of the Internet protocols over the Ethernet, whereas a computer on a different sort of network will need an implementation of the Internet protocols for that network. Chapter 3 explains how the Internet protocols are implemented over a variety of different networks.

Data types such as integers may be represented in different ways on different sorts of hardware – for example, there are two alternatives for the byte ordering of integers. These differences in representation must be dealt with if messages are to be exchanged between programs running on different hardware.

Although the operating systems of all computers on the Internet need to include an implementation of the Internet protocols, they do not necessarily all provide the same application programming interface to these protocols. For example, the calls for exchanging messages in UNIX are different from the calls in Windows.

Different programming languages use different representations for characters and data structures such as arrays and records. These differences must be addressed if programs written in different languages are to be able to communicate with one another.

Programs written by different developers cannot communicate with one another unless they use common standards, for example, for network communication and the representation of primitive data items and data structures in messages. For this to happen, standards need to be agreed and adopted – as have the Internet protocols.

Middleware • The term *middleware* applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. The Common Object Request Broker (CORBA), which is described in Chapters 4, 5 and 8, is an example. Some middleware, such as Java Remote Method Invocation (RMI) (see Chapter 5), supports only a single programming language. Most middleware is implemented over the Internet protocols, which themselves mask the differences of the underlying networks, but all middleware deals with the differences in operating systems and hardware – how this is done is the main topic of Chapter 4.

In addition to solving the problems of heterogeneity, middleware provides a uniform computational model for use by the programmers of servers and distributed applications. Possible models include remote object invocation, remote event notification, remote SQL access and distributed transaction processing. For example, CORBA provides remote object invocation, which allows an object in a program running on one computer to invoke a method of an object in a program running on another computer. Its implementation hides the fact that messages are passed over a network in order to send the invocation request and its reply.

Heterogeneity and mobile code • The term *mobile code* is used to refer to program code that can be transferred from one computer to another and run at the destination – Java applets are an example. Code suitable for running on one computer is not necessarily suitable for running on another because executable programs are normally specific both to the instruction set and to the host operating system.

The *virtual machine* approach provides a way of making code executable on a variety of host computers: the compiler for a particular language generates code for a virtual machine instead of a particular hardware order code. For example, the Java compiler produces code for a Java virtual machine, which executes it by interpretation. The Java virtual machine needs to be implemented once for each type of computer to enable Java programs to run.

Today, the most commonly used form of mobile code is the inclusion Javascript programs in some web pages loaded into client browsers. This extension of Web technology is discussed further in Section 1.6.

1.5.2 Openness

The openness of a computer system is the characteristic that determines whether the system can be extended and reimplemented in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.

Openness cannot be achieved unless the specification and documentation of the key software interfaces of the components of a system are made available to software developers. In a word, the key interfaces are *published*. This process is akin to the standardization of interfaces, but it often bypasses official standardization procedures, which are usually cumbersome and slow-moving.

However, the publication of interfaces is only the starting point for adding and extending services in a distributed system. The challenge to designers is to tackle the complexity of distributed systems consisting of many components engineered by different people.

The designers of the Internet protocols introduced a series of documents called 'Requests For Comments', or RFCs, each of which is known by a number. The specifications of the Internet communication protocols were published in this series in the early 1980s, followed by specifications for applications that run over them, such as file transfer, email and telnet by the mid-1980s. This practice has continued and forms the basis of the technical documentation of the Internet. This series includes discussions as well as the specifications of protocols. Copies can be obtained from [www.ietf.org]. Thus the publication of the original Internet communication protocols has enabled a variety of Internet systems and applications including the Web to be built. RFCs are not the only means of publication. For example, the World Wide Web Consortium (W3C) develops and publishes standards related to the working of the Web [www.w3.org].

Systems that are designed to support resource sharing in this way are termed *open distributed systems* to emphasize the fact that they are extensible. They may be extended at the hardware level by the addition of computers to the network and at the software level by the introduction of new services and the reimplementation of old ones, enabling application programs to share resources. A further benefit that is often cited for open systems is their independence from individual vendors.

To summarize:

- Open systems are characterized by the fact that their key interfaces are published.
- Open distributed systems are based on the provision of a uniform communication mechanism and published interfaces for access to shared resources.
- Open distributed systems can be constructed from heterogeneous hardware and software, possibly from different vendors. But the conformance of each component to the published standard must be carefully tested and verified if the system is to work correctly.

1.5.3 Security

Many of the information resources that are made available and maintained in distributed systems have a high intrinsic value to their users. Their security is therefore of considerable importance. Security for information resources has three components: confidentiality (protection against disclosure to unauthorized individuals), integrity (protection against alteration or corruption), and availability (protection against interference with the means to access the resources).

Section 1.1 pointed out that although the Internet allows a program in one computer to communicate with a program in another computer irrespective of its

location, security risks are associated with allowing free access to all of the resources in an intranet. Although a firewall can be used to form a barrier around an intranet, restricting the traffic that can enter and leave, this does not deal with ensuring the appropriate use of resources by users within an intranet, or with the appropriate use of resources in the Internet, that are not protected by firewalls.

In a distributed system, clients send requests to access data managed by servers, which involves sending information in messages over a network. For example:

1. A doctor might request access to hospital patient data or send additions to that data.
2. In electronic commerce and banking, users send their credit card numbers across the Internet.

In both examples, the challenge is to send sensitive information in a message over a network in a secure manner. But security is not just a matter of concealing the contents of messages – it also involves knowing for sure the identity of the user or other agent on whose behalf a message was sent. In the first example, the server needs to know that the user is really a doctor, and in the second example, the user needs to be sure of the identity of the shop or bank with which they are dealing. The second challenge here is to identify a remote user or other agent correctly. Both of these challenges can be met by the use of encryption techniques developed for this purpose. They are used widely in the Internet and are discussed in Chapter 11.

However, the following two security challenges have not yet been fully met:

Denial of service attacks: Another security problem is that a user may wish to disrupt a service for some reason. This can be achieved by bombarding the service with such a large number of pointless requests that the serious users are unable to use it. This is called a *denial of service* attack. There have been several denial of service attacks on well-known web services. Currently such attacks are countered by attempting to catch and punish the perpetrators after the event, but that is not a general solution to the problem. Countermeasures based on improvements in the management of networks are under development, and these will be touched on in Chapter 3.

Security of mobile code: Mobile code needs to be handled with care. Consider someone who receives an executable program as an electronic mail attachment: the possible effects of running the program are unpredictable; for example, it may seem to display an interesting picture but in reality it may access local resources, or perhaps be part of a denial of service attack. Some measures for securing mobile code are outlined in Chapter 11.

1.5.4 Scalability

Distributed systems operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet. A system is described as *scalable* if it will remain effective when there is a significant increase in the number of resources and the number of users. The number of computers and servers in the Internet has increased dramatically. Figure 1.6 shows the increasing number of computers and web servers during the 12-year history of the Web up to 2005 [zakon.org]. It is interesting to note the significant growth in both computers and web servers in this period, but also that the

relative percentage is flattening out – a trend that is explained by the growth of fixed and mobile personal computing. One web server may also increasingly be hosted on multiple computers.

The design of scalable distributed systems presents the following challenges:

Controlling the cost of physical resources: As the demand for a resource grows, it should be possible to extend the system, at reasonable cost, to meet it. For example, the frequency with which files are accessed in an intranet is likely to grow as the number of users and computers increases. It must be possible to add server computers to avoid the performance bottleneck that would arise if a single file server had to handle all file access requests. In general, for a system with n users to be scalable, the quantity of physical resources required to support them should be at most $O(n)$ – that is, proportional to n. For example, if a single file server can support 20 users, then two such servers should be able to support 40 users. Although that sounds an obvious goal, it is not necessarily easy to achieve in practice, as we show in Chapter 12.

Controlling the performance loss: Consider the management of a set of data whose size is proportional to the number of users or resources in the system – for example, the table with the correspondence between the domain names of computers and their Internet addresses held by the Domain Name System, which is used mainly to look up DNS names such as www.amazon.com. Algorithms that use hierarchic structures scale better than those that use linear structures. But even with hierarchic structures an increase in size will result in some loss in performance: the time taken to access hierarchically structured data is $O(\log n)$, where n is the size of the set of data. For a system to be scalable, the maximum performance loss should be no worse than this.

Preventing software resources running out: An example of lack of scalability is shown by the numbers used as Internet (IP) addresses (computer addresses in the Internet). In the late 1970s, it was decided to use 32 bits for this purpose, but as will be explained in Chapter 3, the supply of available Internet addresses is running out. For this reason, a new version of the protocol with 128-bit Internet addresses is being adopted, and this will require modifications to many software components. To be fair

Figure 1.6 Growth of the Internet (computers and web servers)

Date	*Computers*	*Web servers*	*Percentage*
1993, July	1,776,000	130	0.008
1995, July	6,642,000	23,500	0.4
1997, July	19,540,000	1,203,096	6
1999, July	56,218,000	6,598,697	12
2001, July	125,888,197	31,299,592	25
2003, July	~200,000,000	42,298,371	21
2005, July	353,284,187	67,571,581	19

to the early designers of the Internet, there is no correct solution to this problem. It is difficult to predict the demand that will be put on a system years ahead. Moreover, overcompensating for future growth may be worse than adapting to a change when we are forced to – larger Internet addresses will occupy extra space in messages and in computer storage.

Avoiding performance bottlenecks: In general, algorithms should be decentralized to avoid having performance bottlenecks. We illustrate this point with reference to the predecessor of the Domain Name System, in which the name table was kept in a single master file that could be downloaded to any computers that needed it. That was fine when there were only a few hundred computers in the Internet, but it soon became a serious performance and administrative bottleneck. The Domain Name System removed this bottleneck by partitioning the name table between servers located throughout the Internet and administered locally – see Chapters 3 and 13.

Some shared resources are accessed very frequently; for example, many users may access the same web page, causing a decline in performance. We shall see in Chapter 2 that caching and replication may be used to improve the performance of resources that are very heavily used.

Ideally, the system and application software should not need to change when the scale of the system increases, but this is difficult to achieve. The issue of scale is a dominant theme in the development of distributed systems. The techniques that have been successful are discussed extensively in this book. They include the use of replicated data (Chapter 18), the associated technique of caching (Chapters 2 and 12) and the deployment of multiple servers to handle commonly performed tasks, enabling several similar tasks to be performed concurrently.

1.5.5 Failure handling

Computer systems sometimes fail. When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation. We shall discuss and classify a range of possible failure types that can occur in the processes and networks that comprise a distributed system in Chapter 2.

Failures in a distributed system are partial – that is, some components fail while others continue to function. Therefore the handling of failures is particularly difficult. The following techniques for dealing with failures are discussed throughout the book:

Detecting failures: Some failures can be detected. For example, checksums can be used to detect corrupted data in a message or a file. Chapter 2 explains that it is difficult or even impossible to detect some other failures, such as a remote crashed server in the Internet. The challenge is to manage in the presence of failures that cannot be detected but may be suspected.

Masking failures: Some failures that have been detected can be hidden or made less severe. Two examples of hiding failures:

1. Messages can be retransmitted when they fail to arrive.
2. File data can be written to a pair of disks so that if one is corrupted, the other may still be correct.

Just dropping a message that is corrupted is an example of making a fault less severe – it could be retransmitted. The reader will probably realize that the techniques described for hiding failures are not guaranteed to work in the worst cases; for example, the data on the second disk may be corrupted too, or the message may not get through in a reasonable time however often it is retransmitted.

Tolerating failures: Most of the services in the Internet do exhibit failures – it would not be practical for them to attempt to detect and hide all of the failures that might occur in such a large network with so many components. Their clients can be designed to tolerate failures, which generally involves the users tolerating them as well. For example, when a web browser cannot contact a web server, it does not make the user wait for ever while it keeps on trying – it informs the user about the problem, leaving them free to try again later. Services that tolerate failures are discussed in the paragraph on redundancy below.

Recovery from failures: Recovery involves the design of software so that the state of permanent data can be recovered or 'rolled back' after a server has crashed. In general, the computations performed by some programs will be incomplete when a fault occurs, and the permanent data that they update (files and other material stored in permanent storage) may not be in a consistent state. Recovery is described in Chapter 17.

Redundancy: Services can be made to tolerate failures by the use of redundant components. Consider the following examples:

1. There should always be at least two different routes between any two routers in the Internet.

2. In the Domain Name System, every name table is replicated in at least two different servers.

3. A database may be replicated in several servers to ensure that the data remains accessible after the failure of any single server; the servers can be designed to detect faults in their peers; when a fault is detected in one server, clients are redirected to the remaining servers.

The design of effective techniques for keeping replicas of rapidly changing data up-to-date without excessive loss of performance is a challenge. Approaches are discussed in Chapter 18.

Distributed systems provide a high degree of availability in the face of hardware faults. The *availability* of a system is a measure of the proportion of time that it is available for use. When one of the components in a distributed system fails, only the work that was using the failed component is affected. A user may move to another computer if the one that they were using fails; a server process can be started on another computer.

1.5.6 Concurrency

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to

access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time.

The process that manages a shared resource could take one client request at a time. But that approach limits throughput. Therefore services and applications generally allow multiple client requests to be processed concurrently. To make this more concrete, suppose that each resource is encapsulated as an object and that invocations are executed in concurrent threads. In this case it is possible that several threads may be executing concurrently within an object, in which case their operations on the object may conflict with one another and produce inconsistent results. For example, if two concurrent bids at an auction are 'Smith: $122' and 'Jones: $111', and the corresponding operations are interleaved without any control, then they might get stored as 'Smith: $111' and 'Jones: $122'.

The moral of this story is that any object that represents a shared resource in a distributed system must be responsible for ensuring that it operates correctly in a concurrent environment. This applies not only to servers but also to objects in applications. Therefore any programmer who takes an implementation of an object that was not intended for use in a distributed system must do whatever is necessary to make it safe in a concurrent environment.

For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent. This can be achieved by standard techniques such as semaphores, which are used in most operating systems. This topic and its extension to collections of distributed shared objects are discussed in Chapters 7 and 17.

1.5.7 Transparency

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components. The implications of transparency are a major influence on the design of the system software.

The ANSA Reference Manual [ANSA 1989] and the International Organization for Standardization's Reference Model for Open Distributed Processing (RM-ODP) [ISO 1992] identify eight forms of transparency. We have paraphrased the original ANSA definitions, replacing their migration transparency with our own mobility transparency, whose scope is broader:

Access transparency enables local and remote resources to be accessed using identical operations.

Location transparency enables resources to be accessed without knowledge of their physical or network location (for example, which building or IP address).

Concurrency transparency enables several processes to operate concurrently using shared resources without interference between them.

Replication transparency enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.

Failure transparency enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.

Mobility transparency allows the movement of resources and clients within a system without affecting the operation of users or programs.

Performance transparency allows the system to be reconfigured to improve performance as loads vary.

Scaling transparency allows the system and applications to expand in scale without change to the system structure or the application algorithms.

The two most important transparencies are access and location transparency; their presence or absence most strongly affects the utilization of distributed resources. They are sometimes referred to together as *network transparency*.

As an illustration of access transparency, consider a graphical user interface with folders, which is the same whether the files inside the folder are local or remote. Another example is an API for files that uses the same operations to access both local and remote files (see Chapter 12). As an example of a lack of access transparency, consider a distributed system that does not allow you to access files on a remote computer unless you make use of the ftp program to do so.

Web resource names or URLs are location-transparent because the part of the URL that identifies a web server domain name refers to a computer name in a domain, rather than to an Internet address. However, URLs are not mobility-transparent, because someone's personal web page cannot move to their new place of work in a different domain – all of the links in other pages will still point to the original page.

In general, identifiers such as URLs that include the domain names of computers prevent replication transparency. Although the DNS allows a domain name to refer to several computers, it picks just one of them when it looks up a name. Since a replication scheme generally needs to be able to access all of the participating computers, it would need to access each of the DNS entries by name.

As an illustration of the presence of network transparency, consider the use of an electronic mail address such as *Fred.Flintstone@stoneit.com*. The address consists of a user's name and a domain name. Sending mail to such a user does not involve knowing their physical or network location. Nor does the procedure to send an email message depend upon the location of the recipient. Thus electronic mail within the Internet provides both location and access transparency (that is, network transparency).

Failure transparency can also be illustrated in the context of electronic mail, which is eventually delivered, even when servers or communication links fail. The faults are masked by attempting to retransmit messages until they are successfully delivered, even if it takes several days. Middleware generally converts the failures of networks and processes into programming-level exceptions (see Chapter 5 for an explanation).

To illustrate mobility transparency, consider the case of mobile phones. Suppose that both caller and callee are travelling by train in different parts of a country, moving

from one environment (cell) to another. We regard the caller's phone as the client and the callee's phone as a resource. The two phone users making the call are unaware of the mobility of the phones (the client and the resource) between cells.

Transparency hides and renders anonymous the resources that are not of direct relevance to the task in hand for users and application programmers. For example, it is generally desirable for similar hardware resources to be allocated interchangeably to perform a task – the identity of a processor used to execute a process is generally hidden from the user and remains anonymous. As pointed out in Section 1.3.2, this may not always be what is required: for example, a traveller who attaches a laptop computer to the local network in each office visited should make use of local services such as the send mail service, using different servers at each location. Even within a building, it is normal to arrange for a document to be printed at a particular, named printer: usually one that is near to the user.

1.5.8 Quality of service

Once users are provided with the functionality that they require of a service, such as the file service in a distributed system, we can go on to ask about the quality of the service provided. The main nonfunctional properties of systems that affect the quality of the service experienced by clients and users are *reliability*, *security* and *performance*. *Adaptability* to meet changing system configurations and resource availability has been recognized as a further important aspect of service quality.

Reliability and security issues are critical in the design of most computer systems. The performance aspect of quality of service was originally defined in terms of responsiveness and computational throughput, but it has been redefined in terms of ability to meet timeliness guarantees, as discussed in the following paragraphs.

Some applications, including multimedia applications, handle *time-critical data* – streams of data that are required to be processed or transferred from one process to another at a fixed rate. For example, a movie service might consist of a client program that is retrieving a film from a video server and presenting it on the user's screen. For a satisfactory result the successive frames of video need to be displayed to the user within some specified time limits.

In fact, the abbreviation QoS has effectively been commandeered to refer to the ability of systems to meet such deadlines. Its achievement depends upon the availability of the necessary computing and network resources at the appropriate times. This implies a requirement for the system to provide guaranteed computing and communication resources that are sufficient to enable applications to complete each task on time (for example, the task of displaying a frame of video).

The networks commonly used today have high performance − for example, BBC iPlayer generally performs acceptably − but when networks are heavily loaded their performance can deteriorate, and no guarantees are provided. QoS applies to operating systems as well as networks. Each critical resource must be reserved by the applications that require QoS, and there must be resource managers that provide guarantees. Reservation requests that cannot be met are rejected. These issues will be addressed further in Chapter 20.

1.6 Case study: The World Wide Web

The World Wide Web [www.w3.org I, Berners-Lee 1991] is an evolving system for publishing and accessing resources and services across the Internet. Through commonly available web browsers, users retrieve and view documents of many types, listen to audio streams and view video streams, and interact with an unlimited set of services.

The Web began life at the European centre for nuclear research (CERN), Switzerland, in 1989 as a vehicle for exchanging documents between a community of physicists connected by the Internet [Berners-Lee 1999]. A key feature of the Web is that it provides a *hypertext* structure among the documents that it stores, reflecting the users' requirement to organize their knowledge. This means that documents contain *links* (or *hyperlinks*) – references to other documents and resources that are also stored in the Web.

It is fundamental to the user's experience of the Web that when they encounter a given image or piece of text within a document, this will frequently be accompanied by links to related documents and other resources. The structure of links can be arbitrarily complex and the set of resources that can be added is unlimited – the 'web' of links is indeed world-wide. Bush [1945] conceived of hypertextual structures over 50 years ago; it was with the development of the Internet that this idea could be manifested on a world-wide scale.

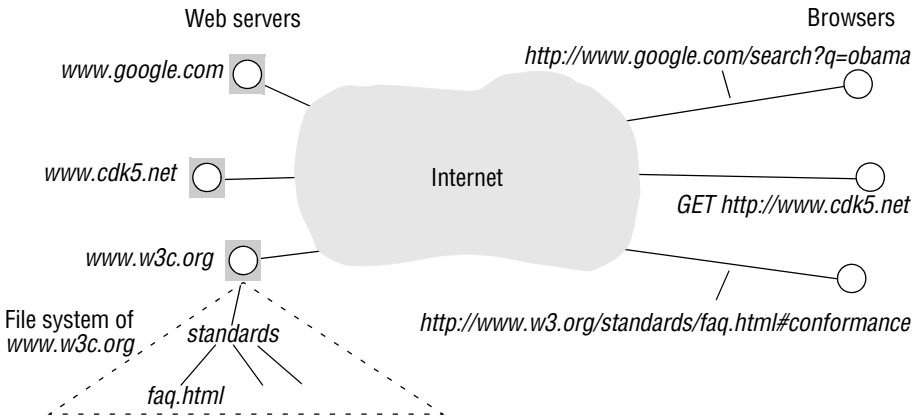
The Web is an *open* system: it can be extended and implemented in new ways without disturbing its existing functionality (see Section 1.5.2). First, its operation is based on communication standards and document or content standards that are freely published and widely implemented. For example, there are many types of browser, each in many cases implemented on several platforms; and there are many implementations of web servers. Any conformant browser can retrieve resources from any conformant server. So users have access to browsers on the majority of the devices that they use, from mobile phones to desktop computers.

Second, the Web is open with respect to the types of resource that can be published and shared on it. At its simplest, a resource on the Web is a web page or some other type of *content* that can be presented to the user, such as media files and documents in Portable Document Format. If somebody invents, say, a new image-storage format, then images in this format can immediately be published on the Web. Users require a means of viewing images in this new format, but browsers are designed to accommodate new content-presentation functionality in the form of 'helper' applications and 'plug-ins'.

The Web has moved beyond these simple data resources to encompass services, such as electronic purchasing of goods. It has evolved without changing its basic architecture. The Web is based on three main standard technological components:

- the HyperText Markup Language (HTML), a language for specifying the contents and layout of pages as they are displayed by web browsers;
- Uniform Resource Locators (URLs), also known as Uniform Resource Identifiers (URIs), which identify documents and other resources stored as part of the Web;
- a client-server system architecture, with standard rules for interaction (the HyperText Transfer Protocol – HTTP) by which browsers and other clients fetch documents and other resources from web servers. Figure 1.7 shows some web servers, and browsers making requests to them. It is an important feature that users may locate and manage their own web servers anywhere on the Internet.

Figure 1.7 Web servers and web browsers



We now discuss these components in turn, and in so doing explain the operation of browsers and web servers when a user fetches web pages and clicks on the links within them.

HTML • The HyperText Markup Language [www.w3.org II] is used to specify the text and images that make up the contents of a web page, and to specify how they are laid out and formatted for presentation to the user. A web page contains such structured items as headings, paragraphs, tables and images. HTML is also used to specify links and which resources are associated with them.

Users may produce HTML by hand, using a standard text editor, but they more commonly use an HTML-aware 'wysiwyg' editor that generates HTML from a layout that they create graphically. A typical piece of HTML text follows:

```
<IMG SRC = "http://www.cdk5.net/WebExample/Images/earth.jpg">          1
<P>                                                                    2
Welcome to Earth! Visitors may also be interested in taking a look at the   3
<A HREF = "http://www.cdk5.net/WebExample/moon.html">Moon</A>.         4
</P>                                                                   5
```

This HTML text is stored in a file that a web server can access – let us say the file *earth.html*. A browser retrieves the contents of this file from a web server – in this case a server on a computer called *www.cdk5.net*. The browser reads the content returned by the server and renders it into formatted text and images laid out on a web page in the familiar fashion. Only the browser – not the server – interprets the HTML text. But the server does inform the browser of the type of content it is returning, to distinguish it from, say, a document in Portable Document Format. The server can infer the content type from the filename extension '.html'.

Note that the HTML directives, known as *tags*, are enclosed by angle brackets, such as <*P*>. Line 1 of the example identifies a file containing an image for presentation. Its URL is *http://www.cdk5.net/WebExample/Images/earth.jpg*. Lines 2 and 5 are directives to begin and end a paragraph, respectively. Lines 3 and 4 contain text to be displayed on the web page in the standard paragraph format.

Line 4 specifies a link in the web page. It contains the word 'Moon' surrounded by two related HTML tags, *<A HREF...>* and *</A>*. The text between these tags is what appears in the link as it is presented on the web page. Most browsers are configured to show the text of links underlined by default, so what the user will see in that paragraph is:

Welcome to Earth! Visitors may also be interested in taking a look at the Moon.

The browser records the association between the link's displayed text and the URL contained in the *<A HREF...>* tag – in this case:

http://www.cdk5.net/WebExample/moon.html

When the user clicks on the text, the browser retrieves the resource identified by the corresponding URL and presents it to the user. In the example, the resource is an HTML file specifying a web page about the Moon.

URLs • The purpose of a Uniform Resource Locator [www.w3.org III] is to identify a resource. Indeed, the term used in web architecture documents is Uniform Resource *Identifier* (URI), but in this book the better-known term URL will be used when no confusion can arise. Browsers examine URLs in order to access the corresponding resources. Sometimes the user types a URL into the browser. More commonly, the browser looks up the corresponding URL when the user clicks on a link or selects one of their 'bookmarks'; or when the browser fetches a resource embedded in a web page, such as an image.

Every URL, in its full, absolute form, has two top-level components:

scheme : scheme-specific-identifier

The first component, the 'scheme', declares which type of URL this is. URLs are required to identify a variety of resources. For example, *mailto:joe@anISP.net* identifies a user's email address; *ftp://ftp.downloadIt.com/software/aProg.exe* identifies a file that is to be retrieved using the File Transfer Protocol (FTP) rather than the more commonly used protocol HTTP. Other examples of schemes are 'tel' (used to specify a telephone number to dial, which is particularly useful when browsing on a mobile phone) and 'tag' (used to identify an arbitrary entity).

The Web is open with respect to the types of resources it can be used to access, by virtue of the scheme designators in URLs. If somebody invents a useful new type of 'widget' resource – perhaps with its own addressing scheme for locating widgets and its own protocol for accessing them – then the world can start using URLs of the form *widget:...*. Of course, browsers must be given the capability to use the new 'widget' protocol, but this can be done by adding a plug-in.

HTTP URLs are the most widely used, for accessing resources using the standard HTTP protocol. An HTTP URL has two main jobs: to identify which web server maintains the resource, and to identify which of the resources at that server is required. Figure 1.7 shows three browsers issuing requests for resources managed by three web servers. The topmost browser is issuing a query to a search engine. The middle browser requires the default page of another web site. The bottommost browser requires a web page that is specified in full, including a path name relative to the server. The files for a given web server are maintained in one or more subtrees (directories) of the server's file system, and each resource is identified by a path name relative to the server.

In general, HTTP URLs are of the following form:

http:// servername [:port] [/pathName] [?query] [#fragment]

where items in square brackets are optional. A full HTTP URL always begins with the string 'http://' followed by a server name, expressed as a Domain Name System (DNS) name (see Section 13.2). The server's DNS name is optionally followed by the number of the 'port' on which the server listens for requests (see Chapter 4), which is 80 by default. Then comes an optional path name of the server's resource. If this is absent then the server's default web page is required. Finally, the URL optionally ends in a query component – for example, when a user submits the entries in a form such as a search engine's query page – and/or a fragment identifier, which identifies a component of the resource.

Consider the URLs:

http://www.cdk5.net
http://www.w3.org/standards/faq.html#conformance
http://www.google.com/search?q=obama

These can be broken down as follows:

Server DNS name	*Path name*	*Query*	*Fragment*
www.cdk5.net	(default)	(none)	(none)
www.w3.org	standards/faq.html	(none)	intro
www.google.com	search	q=obama	(none)

The first URL designates the default page supplied by *www.cdk5.net.* The next identifies a fragment of an HTML file whose path name is *standards/faq.html* relative to the server *www.w3.org*. The fragment's identifier (specified after the '#' character in the URL) is *intro*, and a browser will search for that fragment identifier within the HTML text after it has downloaded the whole file. The third URL specifies a query to a search engine. The path identifies a program called 'search', and the string after the '?' character encodes a query string supplied as arguments to this program. We discuss URLs that identify programmatic resources in more detail when we consider more advanced features below.

Publishing a resource: While the Web has a clearly defined model for accessing a resource from its URL, the exact methods for publishing resources on the Web are dependent upon the web server implementation. In terms of low-level mechanisms, the simplest method of publishing a resource on the Web is to place the corresponding file in a directory that the web server can access. Knowing the name of the server *S* and a path name for the file *P* that the server can recognize, the user then constructs the URL as *http://S/P*. The user puts this URL in a link from an existing document or distributes the URL to other users, for example by email.

It is common for such concerns to be hidden from users when they generate content. For example, 'bloggers' typically use software tools, themselves implemented as web pages, to create organized collections of journal pages. Product pages for a company's web site are typically created using a *content management system*, again by

directly interacting with the web site through administrative web pages. The database or file system on which the product pages are based is transparent.

Finally, Huang *et al.* [2000] provide a model for inserting content into the Web with minimal human intervention. This is particularly relevant where users need to extract content from a variety of devices, such as cameras, for publication in web pages.

HTTP • The HyperText Transfer Protocol [www.w3.org IV] defines the ways in which browsers and other types of client interact with web servers. Chapter 5 will consider HTTP in more detail, but here we outline its main features (restricting our discussion to the retrieval of resources in files):

Request-reply interactions: HTTP is a 'request-reply' protocol. The client sends a request message to the server containing the URL of the required resource. The server looks up the path name and, if it exists, sends back the resource's content in a reply message to the client. Otherwise, it sends back an error response such as the familiar '404 Not Found'. HTTP defines a small set of operations or *methods* that can be performed on a resource. The most common are GET, to retrieve data from the resource, and POST, to provide data to the resource.

Content types: Browsers are not necessarily capable of handling every type of content. When a browser makes a request, it includes a list of the types of content it prefers – for example, in principle it may be able to display images in 'GIF' format but not 'JPEG' format. The server may be able to take this into account when it returns content to the browser. The server includes the content type in the reply message so that the browser will know how to process it. The strings that denote the type of content are called MIME types, and they are standardized in RFC 1521 [Freed and Borenstein 1996]. For example, if the content is of type 'text/html' then a browser will interpret the text as HTML and display it; if the content is of type 'image/GIF' then the browser will render it as an image in 'GIF' format; if the content type is 'application/zip' then it is data compressed in 'zip' format, and the browser will launch an external helper application to decompress it. The set of actions that a browser will take for a given type of content is configurable, and readers may care to check these settings for their own browsers.

One resource per request: Clients specify one resource per HTTP request. If a web page contains nine images, say, then the browser will issue a total of ten separate requests to obtain the entire contents of the page. Browsers typically make several requests concurrently, to reduce the overall delay to the user.

Simple access control: By default, any user with network connectivity to a web server can access any of its published resources. If users wish to restrict access to a resource, then they can configure the server to issue a 'challenge' to any client that requests it. The corresponding user then has to prove that they have the right to access the resource, for example, by typing in a password.

Dynamic pages • So far we have described how users can publish web pages and other content stored in files on the Web. However, much of the users' experience of the Web is that of interacting with services rather than retrieving data. For example, when purchasing an item at an online store, the user often fills out a *web form* to provide personal details or to specify exactly what they wish to purchase. A web form is a web

page containing instructions for the user and input widgets such as text fields and check boxes. When the user submits the form (usually by pressing a button or the 'return' key), the browser sends an HTTP request to a web server, containing the values that the user has entered.

Since the result of the request depends upon the user's input, the server has to *process* the user's input. Therefore the URL or its initial component designates a *program* on the server, not a file. If the user's input is a reasonably small set of parameters it is often sent as the *query* component of the URL, using the GET method; alternatively, it is sent as additional data in the request using the POST method. For example, a request containing the following URL invokes a program called 'search' at www.google.com and specifies a query string of 'obama': *http://www.google.com/search?q=obama.*

That 'search' program produces HTML text as its output, and the user will see a listing of pages that contain the word 'obama'. (The reader may care to enter a query into their favourite search engine and notice the URL that the browser displays when the result is returned.) The server returns the HTML text that the program generates just as though it had retrieved it from a file. In other words, the difference between static content fetched from a file and content that is dynamically generated is transparent to the browser.

A program that web servers run to generate content for their clients is referred to as a Common Gateway Interface (CGI) program. A CGI program may have any application-specific functionality, as long as it can parse the arguments that the client provides to it and produce content of the required type (usually HTML text). The program will often consult or update a database in processing the request.

Downloaded code: A CGI program runs at the server. Sometimes the designers of web services require some service-related code to run inside the browser, at the user's computer. In particular, code written in Javascript [www.netscape.com] is often downloaded with a web page containing a form, in order to provide better-quality interaction with the user than that supported by HTML's standard widgets. A Javascript-enhanced page can give the user immediate feedback on invalid entries, instead of forcing the user to check the values at the server, which would take much longer.

Javascript can also be used to update parts of a web page's contents without fetching an entirely new version of the page and re-rendering it. These dynamic updates occur either due to a user action (such as clicking on a link or a radio button), or when the browser acquires new data from the server that supplied the web page. In the latter case, since the timing of the data's arrival is unconnected with any user action at the browser itself, it is termed *asynchronous*. A technique known as *AJAX* (Asynchronous Javascript And XML) is used in such cases. AJAX is described more fully in Section 2.3.2.

An alternative to a Javascript program is an *applet*: an application written in the Java language [Flanagan 2002], which the browser automatically downloads and runs when it fetches a corresponding web page. Applets may access the network and provide customized user interfaces. For example, 'chat' applications are sometimes implemented as applets that run on the users' browsers, together with a server program. The applets send the users' text to the server, which in turn distributes it to all the applets for presentation to the user. We discuss applets in more detail in Section 2.3.1.

Web services • So far we have discussed the Web largely from the point of view of a user operating a browser. But programs other than browsers can be clients of the Web, too; indeed, programmatic access to web resources is commonplace.

However, HTML is inadequate for programmatic interoperation. There is an increasing need to exchange many types of structured data on the Web, but HTML is limited in that it is not extensible to applications beyond information browsing. HTML has a static set of structures such as paragraphs, and they are bound up with the way that the data is to be presented to users. The Extensible Markup Language (XML) (see Section 4.3.3) has been designed as a way of representing data in standard, structured, application-specific forms. In principle, data expressed in XML is portable between applications since it is *self-describing*: it contains the names, types and structure of the data elements within it. For example, XML may be used to describe products or information about users, for many different services or applications. In the HTTP protocol, XML data can be transmitted by the POST and GET operations. In AJAX it can be used to provide data to Javascript programs in browsers.

Web resources provide service-specific operations. For example, in the store at amazon.com, web service operations include one to order a book and another to check the current status of an order. As we have mentioned, HTTP provides a small set of operations that are applicable to any resource. These include principally the GET and POST methods on existing resources, and the PUT and DELETE operations, respectively. for creating and deleting web resources. Any operation on a resource can be invoked using one of the GET or POST methods, with structured content used to specify the operation's parameters, results and error responses. The so-called REST (REpresentational State Transfer) architecture for web services [Fielding 2000] adopts this approach on the basis of its extensibility: every resource on the Web has a URL and responds to the same set of operations, although the processing of the operations can vary widely from resource to resource. The flip-side of that extensibility can be a lack of robustness in how software operates. Chapter 9 further describes REST and takes an in-depth look at the web services framework, which enables the designers of web services to describe to programmers more specifically what service-specific operations are available and how clients must access them.

Discussion of the Web • The Web's phenomenal success rests upon the relative ease with which many individual and organizational sources can publish resources, the suitability of its hypertext structure for organizing many types of information, and the openness of its system architecture. The standards upon which its architecture is based are simple and they were widely published at an early stage. They have enabled many new types of resources and services to be integrated.

The Web's success belies some design problems. First, its hypertext model is lacking in some respects. If a resource is deleted or moved, so-called 'dangling' links to that resource may still remain, causing frustration for users. And there is the familiar problem of users getting 'lost in hyperspace'. Users often find themselves confused, following many disparate links, referencing pages from a disparate collection of sources, and of dubious reliability in some cases.

Search engines are a highly popular alternative to following links as a means of finding information on the Web, but these are imperfect at producing what the user specifically intends. One approach to this problem, exemplified in the Resource

Description Framework [www.w3.org V], is to produce standard vocabularies, syntax and semantics for expressing metadata about the things in our world, and to encapsulate that metadata in corresponding web resources for programmatic access. Rather than searching for words that occur in web pages, programs can then, in principle, perform searches against the metadata to compile lists of related links based on semantic matching. Collectively, the web of linked metadata resources is what is meant by the *semantic web.*

As a system architecture the Web faces problems of scale. Popular web servers may experience many 'hits' per second, and as a result the response to users can be slow. Chapter 2 describes the use of caching in browsers and proxy servers to increase responsiveness, and the division of the server's load across clusters of computers.

1.7 Summary

Distributed systems are everywhere. The Internet enables users throughout the world to access its services wherever they may be located. Each organization manages an intranet, which provides local services and Internet services for local users and generally provides services to other users in the Internet. Small distributed systems can be constructed from mobile computers and other small computational devices that are attached to a wireless network.

Resource sharing is the main motivating factor for constructing distributed systems. Resources such as printers, files, web pages or database records are managed by servers of the appropriate type. For example, web servers manage web pages and other web resources. Resources are accessed by clients – for example, the clients of web servers are generally called browsers.

The construction of distributed systems produces many challenges:

Heterogeneity: They must be constructed from a variety of different networks, operating systems, computer hardware and programming languages. The Internet communication protocols mask the difference in networks, and middleware can deal with the other differences.

Openness: Distributed systems should be extensible – the first step is to publish the interfaces of the components, but the integration of components written by different programmers is a real challenge.

Security: Encryption can be used to provide adequate protection of shared resources and to keep sensitive information secret when it is transmitted in messages over a network. Denial of service attacks are still a problem.

Scalability: A distributed system is scalable if the cost of adding a user is a constant amount in terms of the resources that must be added. The algorithms used to access shared data should avoid performance bottlenecks and data should be structured hierarchically to get the best access times. Frequently accessed data can be replicated.

Failure handling: Any process, computer or network may fail independently of the others. Therefore each component needs to be aware of the possible ways in which

the components it depends on may fail and be designed to deal with each of those failures appropriately.

Concurrency: The presence of multiple users in a distributed system is a source of concurrent requests to its resources. Each resource must be designed to be safe in a concurrent environment.

Transparency: The aim is to make certain aspects of distribution invisible to the application programmer so that they need only be concerned with the design of their particular application. For example, they need not be concerned with its location or the details of how its operations are accessed by other components, or whether it will be replicated or migrated. Even failures of networks and processes can be presented to application programmers in the form of exceptions – but they must be handled.

Quality of service. It is not sufficient to provide access to services in distributed systems. In particular, it is also important to provide guarantees regarding the qualities associated with such service access. Examples of such qualities include parameters related to performance, security and reliability.

EXERCISES

1.1 We define a distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages. What are the consequences of defining a distributed system in this manner? *page 18*

1.2 How might the clocks in two computers that are linked by a local network be synchronized without reference to an external time source? What factors limit the accuracy of the procedure you have described? How could the clocks in a large number of computers connected by the Internet be synchronized? Discuss the accuracy of that procedure. *page 18*

1.3 Consider the implementation strategies for massively multiplayer online games as discussed in Section 1.2.2. In particular, what advantages do you see in adopting a single server approach for representing the state of the multiplayer game? What problems can you identify and how might they be resolved? *page 21*

1.4 A user arrives at a railway station that they has never visited before, carrying a PDA that is capable of wireless networking. Suggest how the user could be provided with information about the local services and amenities at that station, without entering the station's name or attributes. What technical challenges must be overcome? *page 29*

1.5 Distributed systems are going through a period of significant change, which can be traced back to a number of influential trends. Can you identify what these might be? *page 24*

1.6 Due to the increasing maturity of distributed systems infrastructure, organizations are moving towards viewing distributed systems as a utility. In this model, resources are provided by appropriate service suppliers and effectively rented rather than owned by the end user. Explain this model with respect to physical resources and software services. Can you give examples of some companies that support such software services? *page 29*

1.7 A server program written in one language (for example, C++) provides the implementation of a BLOB object that is intended to be accessed by clients that may be written in a different language (for example, Java). The client and server computers may have different hardware, but all of them are attached to an internet. Describe the problems due to each of the five aspects of heterogeneity that need to be solved to make it possible for a client object to invoke a method on the server object. *page 32*

1.8 What is client-server computing? Which of these roles is an active role, and which is a passive one? Explain remote invocation in this context. *page 31*

1.9 Suppose that the operations of the BLOB object are separated into two categories – public operations that are available to all users and protected operations that are available only to certain named users. State all of the problems involved in ensuring that only the named users can use a protected operation. Supposing that access to a protected operation provides information that should not be revealed to all users, what further problems arise? *page 34*

1.10 The INFO service manages a potentially very large set of resources, each of which can be accessed by users throughout the Internet by means of a key (a string name). Discuss an approach to the design of the names of the resources that achieves the minimum loss of performance as the number of resources in the service increases. Suggest how the INFO service can be implemented so as to avoid performance bottlenecks when the number of users becomes very large. *page 35*

1.11 A distributed system is described as scalable if it remains effective when there is a significant increase in the number of resources and the number of users. However, these systems sometimes face performance bottlenecks. How can these be avoided? *page 37*

1.12 A server process maintains a shared information object such as the BLOB object of Exercise 1.7. Give arguments for and against allowing the client requests to be executed concurrently by the server. In the case that they are executed concurrently, give an example of possible 'interference' that can occur between the operations of different clients. Suggest how such interference may be prevented. *page 38*

1.13 The ANSA Reference Manual [ANSA 1989] identified eight forms of transparency in a distributed system. Which are the two most important transparencies among these? *page 40*

1.14 One of the main standard technological components of the Web is the HyperText Transfer Protocol (HTTP), which defines the ways in which browsers and other types of clients interact with web servers. What are the different features of the HTTP? *page 46*

1.15 What are the two main functions of an HTTP URL? Explain its general form. Identify the server DNS name, path name, query, and fragment for the URL http://www.google.com/search?q=sabretooth. *page 44*

This page intentionally left blank

2

SYSTEM MODELS

This chapter provides an explanation of three important and complementary ways in which the design of distributed systems can usefully be described and discussed:

Physical models consider the types of computers and devices that constitute a system and their interconnectivity, without details of specific technologies.

Architectural models describe a system in terms of the computational and communication tasks performed by its computational elements; the computational elements being individual computers or aggregates of them supported by appropriate network interconnections. *Client-server* and *peer-to-peer* are two of the most commonly used forms of architectural model for distributed systems.

Fundamental models take an abstract perspective in order to describe solutions to individual issues faced by most distributed systems.

There is no global time in a distributed system, so the clocks on different computers do not necessarily give the same time as one another. All communication between processes is achieved by means of messages. Message communication over a computer network can be affected by delays, can suffer from a variety of failures and is vulnerable to security attacks. These issues are addressed by three models:

- The interaction model deals with performance and with the difficulty of setting time limits in a distributed system, for example for message delivery.
- The failure model attempts to give a precise specification of the faults that can be exhibited by processes and communication channels. It defines reliable communication and correct processes.
- The security model discusses the possible threats to processes and communication channels. It introduces the concept of a secure channel, which is secure against those threats.

2.1 Introduction

Systems that are intended for use in real-world environments should be designed to function correctly in the widest possible range of circumstances and in the face of many possible difficulties and threats (for some examples, see the box at the bottom of this page). The discussion and examples of Chapter 1 suggest that distributed systems of different types share important underlying properties and give rise to common design problems. In this chapter we show how the properties and design issues of distributed systems can be captured and discussed through the use of descriptive models. Each type of model is intended to provide an abstract, simplified but consistent description of a relevant aspect of distributed system design:

Physical models are the most explicit way in which to describe a system; they capture the hardware composition of a system in terms of the computers (and other devices, such as mobile phones) and their interconnecting networks.

Architectural models describe a system in terms of the computational and communication tasks performed by its computational elements; the computational elements being individual computers or aggregates of them supported by appropriate network interconnections.

Fundamental models take an abstract perspective in order to examine individual aspects of a distributed system. In this chapter we introduce fundamental models that examine three important aspects of distributed systems: *interaction models*, which consider the structure and sequencing of the communication between the elements of the system; *failure models*, which consider the ways in which a system may fail to operate correctly and; *security models*, which consider how the system is protected against attempts to interfere with its correct operation or to steal its data.

Difficulties and threats for distributed systems • Here are some of the problems that the designers of distributed systems face.

Widely varying modes of use: The component parts of systems are subject to wide variations in workload – for example, some web pages are accessed several million times a day. Some parts of a system may be disconnected, or poorly connected some of the time – for example, when mobile computers are included in a system. Some applications have special requirements for high communication bandwidth and low latency – for example, multimedia applications.

Wide range of system environments: A distributed system must accommodate heterogeneous hardware, operating systems and networks. The networks may differ widely in performance – wireless networks operate at a fraction of the speed of local networks. Systems of widely differing scales, ranging from tens of computers to millions of computers, must be supported.

Internal problems: Non-synchronized clocks, conflicting data updates and many modes of hardware and software failure involving the individual system components.

External threats: Attacks on data integrity and secrecy, denial of service attacks.

2.2 Physical models

A physical model is a representation of the underlying hardware elements of a distributed system that abstracts away from specific details of the computer and networking technologies employed.

Baseline physical model: A distributed system was defined in Chapter 1 as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages. This leads to a minimal physical model of a distributed system as an extensible set of computer nodes interconnected by a computer network for the required passing of messages.

Beyond this baseline model, we can usefully identify three generations of distributed systems.

Early distributed systems: Such systems emerged in the late 1970s and early 1980s in response to the emergence of local area networking technology, usually Ethernet (see Section 3.5). These systems typically consisted of between 10 and 100 nodes interconnected by a local area network, with limited Internet connectivity and supported a small range of services such as shared local printers and file servers as well as email and file transfer across the Internet. Individual systems were largely homogeneous and openness was not a primary concern. Providing quality of service was still very much in its infancy and was a focal point for much of the research around such early systems.

Internet-scale distributed systems: Building on this foundation, larger-scale distributed systems started to emerge in the 1990s in response to the dramatic growth of the Internet during this time (for example, the Google search engine was first launched in 1996). In such systems, the underlying physical infrastructure consists of a physical model as illustrated in Chapter 1, Figure 1.3; that is, an extensible set of nodes interconnected by a *network of networks* (the Internet). Such systems exploit the infrastructure offered by the Internet to become truly global. They incorporate large numbers of nodes and provide distributed system services for global organizations and across organizational boundaries. The level of heterogeneity in such systems is significant in terms of networks, computer architecture, operating systems, languages employed and the development teams involved. This has led to an increasing emphasis on open standards and associated middleware technologies such as CORBA and more recently, web services. Additional services were employed to provide end-to-end quality of service properties in such global systems.

Contemporary distributed systems: In the above systems, nodes were typically desktop computers and therefore relatively static (that is, remaining in one physical location for extended periods), discrete (not embedded within other physical entities) and autonomous (to a large extent independent of other computers in terms of their physical infrastructure). The key trends identified in Section 1.3 have resulted in significant further developments in physical models:

- The emergence of mobile computing has led to physical models where nodes such as laptops or smart phones may move from location to location in a distributed system, leading to the need for added capabilities such as service discovery and support for spontaneous interoperation.

- The emergence of ubiquitous computing has led to a move from discrete nodes to architectures where computers are embedded in everyday objects and in the surrounding environment (for example, in washing machines or in smart homes more generally).
- The emergence of cloud computing and, in particular, cluster architectures has led to a move from autonomous nodes performing a given role to pools of nodes that together provide a given service (for example, a search service as offered by Google).

The end result is a physical architecture with a significant increase in the level of heterogeneity embracing, for example, the tiniest embedded devices utilized in ubiquitous computing through to complex computational elements found in Grid computing. These systems deploy an increasingly varied set of networking technologies and offer a wide variety of applications and services. Such systems potentially involve up to hundreds of thousands of nodes.

Distributed systems of systems • A recent report discusses the emergence of ultra-large-scale (ULS) distributed systems [www.sei.cmu.edu]. The report captures the complexity of modern distributed systems by referring to such (physical) architectures as *systems of systems* (mirroring the view of the Internet as a network of networks). A system of systems can be defined as a complex system consisting of a series of subsystems that are systems in their own right and that come together to perform a particular task or tasks.

As an example of a system of systems, consider an environmental management system for flood prediction. In such a scenario, there will be sensor networks deployed to monitor the state of various environmental parameters relating to rivers, flood plains, tidal effects and so on. This can then be coupled with systems that are responsible for predicting the likelihood of floods, by running (often complex) simulations on, for example, cluster computers (as discussed in Chapter 1). Other systems may be established to maintain and analyze historical data or to provide early warning systems to key stakeholders via mobile phones.

Summary • The overall historical development captured in this section is summarized in Figure 2.1, with the table highlighting the significant challenges associated with contemporary distributed systems in terms of managing the levels of heterogeneity and providing key properties such as openness and quality of service.

2.3 Architectural models

The architecture of a system is its structure in terms of separately specified components and their interrelationships. The overall goal is to ensure that the structure will meet present and likely future demands on it. Major concerns are to make the system reliable, manageable, adaptable and cost-effective. The architectural design of a building has similar aspects – it determines not only its appearance but also its general structure and architectural style (gothic, neo-classical, modern) and provides a consistent frame of reference for the design.

Figure 2.1 Generations of distributed systems

Distributed systems:	*Early*	*Internet-scale*	*Contemporary*
Scale	Small	Large	Ultra-large
Heterogeneity	Limited (typically relatively homogenous configurations)	Significant in terms of platforms, languages and middleware	Added dimensions introduced including radically different styles of architecture
Openness	Not a priority	Significant priority with range of standards introduced	Major research challenge with existing standards not yet able to embrace complex systems
Quality of service	In its infancy	Significant priority with range of services introduced	Major research challenge with existing services not yet able to embrace complex systems

In this section we describe the main architectural models employed in distributed systems – the architectural styles of distributed systems. In particular, we lay the groundwork for a thorough understanding of approaches such as client-server models, peer-to-peer approaches, distributed objects, distributed components, distributed event-based systems and the key differences between these styles.

The section adopts a three-stage approach:

- looking at the core underlying architectural elements that underpin modern distributed systems, highlighting the diversity of approaches that now exist;
- examining composite architectural patterns that can be used in isolation or, more commonly, in combination, in developing more sophisticated distributed systems solutions;
- and finally, considering middleware platforms that are available to support the various styles of programming that emerge from the above architectural styles.

Note that there are many trade-offs associated with the choices identified in this chapter in terms of the architectural elements employed, the patterns adopted and (where appropriate) the middleware used, for example affecting the performance and effectiveness of the resulting system. Understanding such trade-offs is arguably the key skill in distributed systems design.

2.3.1 Architectural elements

To understand the fundamental building blocks of a distributed system, it is necessary to consider four key questions:

- What are the entities that are communicating in the distributed system?

- How do they communicate, or, more specifically, what *communication paradigm* is used?
- What (potentially changing) roles and responsibilities do they have in the overall architecture?
- How are they mapped on to the physical distributed infrastructure (what is their *placement*)?

Communicating entities • The first two questions above are absolutely central to an understanding of distributed systems; what is communicating and how those entities communicate together define a rich design space for the distributed systems developer to consider. It is helpful to address the first question from a system-oriented and a problem-oriented perspective.

From a system perspective, the answer is normally very clear in that the entities that communicate in a distributed system are typically *processes*, leading to the prevailing view of a distributed system as processes coupled with appropriate interprocess communication paradigms (as discussed, for example, in Chapter 4), with two caveats:

- In some primitive environments, such as sensor networks, the underlying operating systems may not support process abstractions (or indeed any form of isolation), and hence the entities that communicate in such systems are *nodes*.
- In most distributed system environments, processes are supplemented by *threads*, so, strictly speaking, it is threads that are the endpoints of communication.

At one level, this is sufficient to model a distributed system and indeed the fundamental models considered in Section 2.4 adopt this view. From a programming perspective, however, this is not enough, and more problem-oriented abstractions have been proposed:

Objects: Objects have been introduced to enable and encourage the use of object-oriented approaches in distributed systems (including both object-oriented design and object-oriented programming languages). In distributed object-based approaches, a computation consists of a number of interacting objects representing natural units of decomposition for the given problem domain. Objects are accessed via interfaces, with an associated interface definition language (or IDL) providing a specification of the methods defined on an object. Distributed objects have become a major area of study in distributed systems, and further consideration is given to this topic in Chapters 5 and 8.

Components: Since their introduction a number of significant problems have been identified with distributed objects, and the use of component technology has emerged as a direct response to such weaknesses. Components resemble objects in that they offer problem-oriented abstractions for building distributed systems and are also accessed through interfaces. The key difference is that components specify not only their (provided) interfaces but also the assumptions they make in terms of other components/interfaces that must be present for a component to fulfil its function – in other words, making all dependencies explicit and providing a more complete contract for system construction. This more contractual approach encourages and

enables third-party development of components and also promotes a purer compositional approach to constructing distributed systems by removing hidden dependencies. Component-based middleware often provides additional support for key areas such as deployment and support for server-side programming [Heineman and Councill 2001]. Further details of component-based approaches can be found in Chapter 8.

Web services: Web services represent the third important paradigm for the development of distributed systems [Alonso *et al*. 2004]. Web services are closely related to objects and components, again taking an approach based on encapsulation of behaviour and access through interfaces. In contrast, however, web services are intrinsically integrated into the World Wide Web, using web standards to represent and discover services. The World Wide Web consortium (W3C) defines a web service as:

> ... a software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artefacts. A Web service supports direct interactions with other software agents using XML-based message exchanges via Internet-based protocols.

In other words, web services are partially defined by the web-based technologies they adopt. A further important distinction stems from the style of use of the technology. Whereas objects and components are often used within an organization to develop tightly coupled applications, web services are generally viewed as complete services in their own right that can be combined to achieve value-added services, often crossing organizational boundaries and hence achieving business to business integration. Web services may be implemented by different providers and using different underlying technologies. Web services are considered further in Chapter 9.

Communication paradigms • We now turn our attention to how entities communicate in a distributed system, and consider three types of communication paradigm:

- interprocess communication;
- remote invocation;
- indirect communication.

Interprocess communication refers to the relatively low-level support for communication between processes in distributed systems, including message-passing primitives, direct access to the API offered by Internet protocols (socket programming) and support for multicast communication. Such services are discussed in detail in Chapter 4.

Remote invocation represents the most common communication paradigm in distributed systems, covering a range of techniques based on a two-way exchange between communicating entities in a distributed system and resulting in the calling of a remote operation, procedure or method, as defined further below (and considered fully in Chapter 5):

Request-reply protocols: Request-reply protocols are effectively a pattern imposed on an underlying message-passing service to support client-server computing. In

particular, such protocols typically involve a pairwise exchange of messages from client to server and then from server back to client, with the first message containing an encoding of the operation to be executed at the server and also an array of bytes holding associated arguments and the second message containing any results of the operation, again encoded as an array of bytes. This paradigm is rather primitive and only really used in embedded systems where performance is paramount. The approach is also used in the HTTP protocol described in Section 5.2. Most distributed systems will elect to use remote procedure calls or remote method invocation, as discussed below, but note that both approaches are supported by underlying request-reply exchanges.

Remote procedure calls: The concept of a remote procedure call (RPC), initially attributed to Birrell and Nelson [1984], represents a major intellectual breakthrough in distributed computing. In RPC, procedures in processes on remote computers can be called as if they are procedures in the local address space. The underlying RPC system then hides important aspects of distribution, including the encoding and decoding of parameters and results, the passing of messages and the preserving of the required semantics for the procedure call. This approach directly and elegantly supports client-server computing with servers offering a set of operations through a service interface and clients calling these operations directly as if they were available locally. RPC systems therefore offer (at a minimum) access and location transparency.

Remote method invocation: Remote method invocation (RMI) strongly resembles remote procedure calls but in a world of distributed objects. With this approach, a calling object can invoke a method in a remote object. As with RPC, the underlying details are generally hidden from the user. RMI implementations may, though, go further by supporting object identity and the associated ability to pass object identifiers as parameters in remote calls. They also benefit more generally from tighter integration into object-oriented languages as discussed in Chapter 5.

The above set of techniques all have one thing in common: communication represents a two-way relationship between a sender and a receiver with senders explicitly directing messages/invocations to the associated receivers. Receivers are also generally aware of the identity of senders, and in most cases both parties must exist at the same time. In contrast, a number of techniques have emerged whereby communication is indirect, through a third entity, allowing a strong degree of decoupling between senders and receivers. In particular:

- Senders do not need to know who they are sending to (*space uncoupling*).
- Senders and receivers do not need to exist at the same time (*time uncoupling*).

Indirect communication is discussed in more detail in Chapter 6.

Key techniques for indirect communication include:

Group communication: Group communication is concerned with the delivery of messages to a set of recipients and hence is a multiparty communication paradigm supporting one-to-many communication. Group communication relies on the abstraction of a group which is represented in the system by a group identifier.

Recipients elect to receive messages sent to a group by joining the group. Senders then send messages to the group via the group identifier, and hence do not need to know the recipients of the message. Groups typically also maintain group membership and include mechanisms to deal with failure of group members.

Publish-subscribe systems: Many systems, such as the financial trading example in Chapter 1, can be classified as information-dissemination systems wherein a large number of producers (or publishers) distribute information items of interest (events) to a similarly large number of consumers (or subscribers). It would be complicated and inefficient to employ any of the core communication paradigms discussed above for this purpose and hence publish-subscribe systems (sometimes also called distributed event-based systems) have emerged to meet this important need [Muhl *et al*. 2006]. Publish-subscribe systems all share the crucial feature of providing an intermediary service that efficiently ensures information generated by producers is routed to consumers who desire this information.

Message queues: Whereas publish-subscribe systems offer a one-to-many style of communication, message queues offer a point-to-point service whereby producer processes can send messages to a specified queue and consumer processes can receive messages from the queue or be notified of the arrival of new messages in the queue. Queues therefore offer an indirection between the producer and consumer processes.

Tuple spaces: Tuple spaces offer a further indirect communication service by supporting a model whereby processes can place arbitrary items of structured data, called tuples, in a persistent tuple space and other processes can either read or remove such tuples from the tuple space by specifying patterns of interest. Since the tuple space is persistent, readers and writers do not need to exist at the same time. This style of programming, otherwise known as generative communication, was introduced by Gelernter [1985] as a paradigm for parallel programming. A number of distributed implementations have also been developed, adopting either a client-server-style implementation or a more decentralized peer-to-peer approach.

Distributed shared memory: Distributed shared memory (DSM) systems provide an abstraction for sharing data between processes that do not share physical memory. Programmers are nevertheless presented with a familiar abstraction of reading or writing (shared) data structures as if they were in their own local address spaces, thus presenting a high level of distribution transparency. The underlying infrastructure must ensure a copy is provided in a timely manner and also deal with issues relating to synchronization and consistency of data. An overview of distributed shared memory can be found in Chapter 6.

The architectural choices discussed so far are summarized in Figure 2.2.

Roles and responsibilities • In a distributed system processes – or indeed objects, components or services, including web services (but for the sake of simplicity we use the term process throughout this section) – interact with each other to perform a useful activity, for example, to support a chat session. In doing so, the processes take on given roles, and these roles are fundamental in establishing the overall architecture to be

Figure 2.2 Communicating entities and communication paradigms

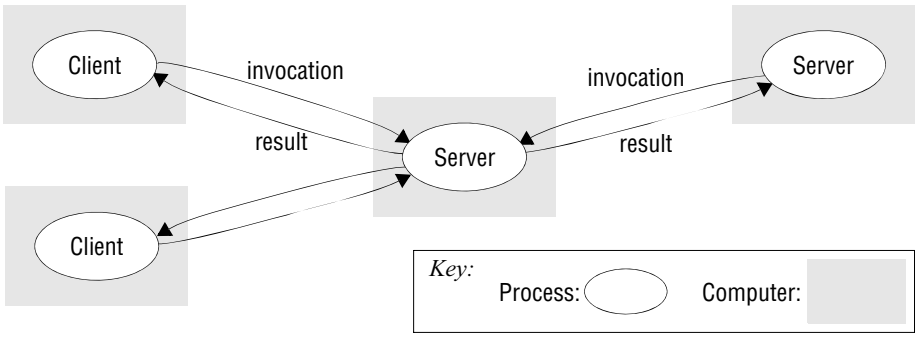
Communicating entities (what is communicating)		*Communication paradigms (how they communicate)*		
System-oriented entities	*Problem-oriented entities*	*Interprocess communication*	*Remote invocation*	*Indirect communication*
Nodes	Objects	Message passing	Request-reply	Group communication
Processes	Components	Sockets	RPC	Publish-subscribe
	Web services	Multicast	RMI	Message queues
				Tuple spaces
				DSM

adopted. In this section, we examine two architectural styles stemming from the role of individual processes: client-server and peer-to-peer.

Client-server: This is the architecture that is most often cited when distributed systems are discussed. It is historically the most important and remains the most widely employed. Figure 2.3 illustrates the simple structure in which processes take on the roles of being clients or servers. In particular, client processes interact with individual server processes in potentially separate host computers in order to access the shared resources that they manage.

Servers may in turn be clients of other servers, as the figure indicates. For example, a web server is often a client of a local file server that manages the files in which the web pages are stored. Web servers and most other Internet services are clients of the DNS service, which translates Internet domain names to network addresses. Another web-related example concerns *search engines*, which enable users to look up summaries of information available on web pages at sites throughout the Internet. These summaries are made by programs called *web crawlers*, which run in the background at a search engine site using HTTP requests to access web servers throughout the Internet. Thus a search engine is both a server and a client: it responds to queries from browser clients and it runs web crawlers that act as clients of other web servers. In this example, the server tasks (responding to user queries) and the crawler tasks (making requests to other web servers) are entirely independent; there is little need to synchronize them and they may run concurrently. In fact, a typical search engine would normally include many concurrent threads of execution, some serving its clients and others running web crawlers. In Exercise 2.5, the reader is invited to consider the only synchronization issue that does arise for a concurrent search engine of the type outlined here.

Figure 2.3 Clients invoke individual servers



Peer-to-peer: In this architecture all of the processes involved in a task or activity play similar roles, interacting cooperatively as *peers* without any distinction between client and server processes or the computers on which they run. In practical terms, all participating processes run the same program and offer the same set of interfaces to each other. While the client-server model offers a direct and relatively simple approach to the sharing of data and other resources, it scales poorly. The centralization of service provision and management implied by placing a service at a single address does not scale well beyond the capacity of the computer that hosts the service and the bandwidth of its network connections.

A number of placement strategies have evolved in response to this problem (see the discussion of placement below), but none of them addresses the fundamental issue – the need to distribute shared resources much more widely in order to share the computing and communication loads incurred in accessing them amongst a much larger number of computers and network links. The key insight that led to the development of peer-to-peer systems is that the network and computing resources owned by the users of a service could also be put to use to support that service. This has the useful consequence that the resources available to run the service grow with the number of users.

The hardware capacity and operating system functionality of today's desktop computers exceeds that of yesterday's servers, and the majority are equipped with always-on broadband network connections. The aim of the peer-to-peer architecture is to exploit the resources (both data and hardware) in a large number of participating computers for the fulfilment of a given task or activity. Peer-to-peer applications and systems have been successfully constructed that enable tens or hundreds of thousands of computers to provide access to data and other resources that they collectively store and manage. One of the earliest instances was the Napster application for sharing digital music files. Although Napster was not a pure peer-to-peer architecture (and also gained notoriety for reasons beyond its architecture), its demonstration of feasibility has resulted in the development of the architectural model in many valuable directions. A more recent and widely used instance is the BitTorrent file-sharing system (discussed in more depth in Section 20.6.2).

Figure 2.4 A service provided by multiple servers

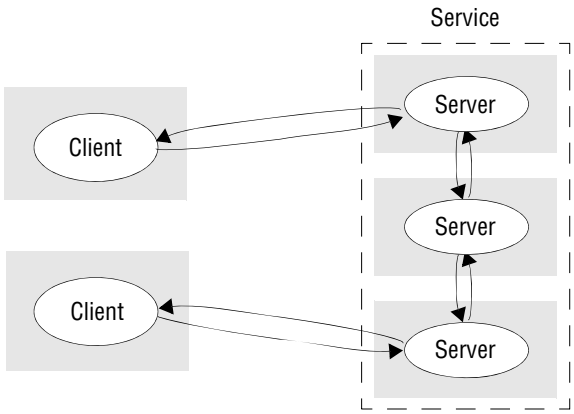


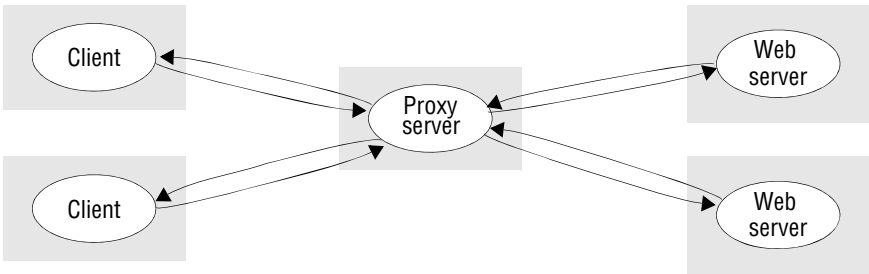
Figure 2.4 illustrates the form of a peer-to-peer application. Applications are composed of large numbers of peer processes running on separate computers and the pattern of communication between them depends entirely on application requirements. A large number of data objects are shared, an individual computer holds only a small part of the application database, and the storage, processing and communication loads for access to objects are distributed across many computers and network links. Each object is replicated in several computers to further distribute the load and to provide resilience in the event of disconnection of individual computers (as is inevitable in the large, heterogeneous networks at which peer-to-peer systems are aimed). The need to place individual objects and retrieve them and to maintain replicas amongst many computers renders this architecture substantially more complex than the client-server architecture.

The development of peer-to-peer applications and middleware to support them is described in depth in Chapter 10.

Placement • The final issue to be considered is how entities such as objects or services map on to the underlying physical distributed infrastructure which will consist of a potentially large number of machines interconnected by a network of arbitrary complexity. Placement is crucial in terms of determining the properties of the distributed system, most obviously with regard to performance but also to other aspects, such as reliability and security.

The question of where to place a given client or server in terms of machines and processes within machines is a matter of careful design. Placement needs to take into account the patterns of communication between entities, the reliability of given machines and their current loading, the quality of communication between different machines and so on. Placement must be determined with strong application knowledge, and there are few universal guidelines to obtaining an optimal solution. We therefore focus mainly on the following placement strategies, which can significantly alter the characteristics of a given design (although we return to the key issue of mapping to physical infrastructure in Section 2.3.2 , where we look at tiered architecture):

Figure 2.5 Web proxy server



- mapping of services to multiple servers;
- caching;
- mobile code;
- mobile agents.

Mapping of services to multiple servers: Services may be implemented as several server processes in separate host computers interacting as necessary to provide a service to client processes (Figure 2.4). The servers may partition the set of objects on which the service is based and distribute those objects between themselves, or they may maintain replicated copies of them on several hosts. These two options are illustrated by the following examples.

The Web provides a common example of partitioned data in which each web server manages its own set of resources. A user can employ a browser to access a resource at any one of the servers.

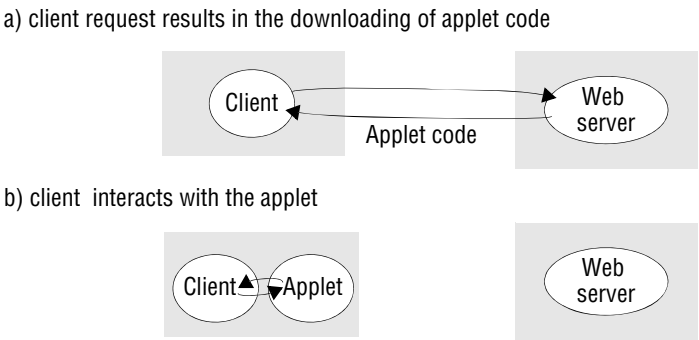
An example of a service based on replicated data is the Sun Network Information Service (NIS), which is used to enable all the computers on a LAN to access the same user authentication data when users log in. Each NIS server has its own replica of a common password file containing a list of users' login names and encrypted passwords. Chapter 18 discusses techniques for replication in detail.

A more closely coupled type of multiple-server architecture is the cluster, as introduced in Chapter 1. A cluster is constructed from up to thousands of commodity processing boards, and service processing can be partitioned or replicated between them.

Caching: A *cache* is a store of recently used data objects that is closer to one client or a particular set of clients than the objects themselves. When a new object is received from a server it is added to the local cache store, replacing some existing objects if necessary. When an object is needed by a client process, the caching service first checks the cache and supplies the object from there if an up-to-date copy is available. If not, an up-to-date copy is fetched. Caches may be co-located with each client or they may be located in a proxy server that can be shared by several clients.

Caches are used extensively in practice. Web browsers maintain a cache of recently visited web pages and other web resources in the client's local file system, using a special HTTP request to check with the original server that cached pages are up-to-date before displaying them. Web proxy servers (Figure 2.5) provide a shared cache of

Figure 2.6 Web applets



web resources for the client machines at a site or across several sites. The purpose of proxy servers is to increase the availability and performance of the service by reducing the load on the wide area network and web servers. Proxy servers can take on other roles; for example, they may be used to access remote web servers through a firewall.

Mobile code: Chapter 1 introduced mobile code. Applets are a well-known and widely used example of mobile code – the user running a browser selects a link to an applet whose code is stored on a web server; the code is downloaded to the browser and runs there, as shown in Figure 2.6. An advantage of running the downloaded code locally is that it can give good interactive response since it does not suffer from the delays or variability of bandwidth associated with network communication.

Accessing services means running code that can invoke their operations. Some services are likely to be so standardized that we can access them with an existing and well-known application – the Web is the most common example of this, but even there, some web sites use functionality not found in standard browsers and require the downloading of additional code. The additional code may, for example, communicate with the server. Consider an application that requires that users be kept up-to-date with changes as they occur at an information source in the server. This cannot be achieved by normal interactions with the web server, which are always initiated by the client. The solution is to use additional software that operates in a manner often referred to as a *push* model – one in which the server instead of the client initiates interactions. For example, a stockbroker might provide a customized service to notify customers of changes in the prices of shares; to use the service, each customer would have to download a special applet that receives updates from the broker's server, displays them to the user and perhaps performs automatic buy and sell operations triggered by conditions set up by the customer and stored locally in the customer's computer.

Mobile code is a potential security threat to the local resources in the destination computer. Therefore browsers give applets limited access to local resources, using a scheme discussed in Section 11.1.1.

Mobile agents: A mobile agent is a running program (including both code and data) that travels from one computer to another in a network carrying out a task on someone's behalf, such as collecting information, and eventually returning with the results. A mobile agent may make many invocations to local resources at each site it visits – for

example, accessing individual database entries. If we compare this architecture with a static client making remote invocations to some resources, possibly transferring large amounts of data, there is a reduction in communication cost and time through the replacement of remote invocations with local ones.

Mobile agents might be used to install and maintain software on the computers within an organization or to compare the prices of products from a number of vendors by visiting each vendor's site and performing a series of database operations. An early example of a similar idea is the so-called worm program developed at Xerox PARC [Shoch and Hupp 1982], which was designed to make use of idle computers in order to carry out intensive computations.

Mobile agents (like mobile code) are a potential security threat to the resources in computers that they visit. The environment receiving a mobile agent should decide which of the local resources it should be allowed to use, based on the identity of the user on whose behalf the agent is acting – their identity must be included in a secure way with the code and data of the mobile agent. In addition, mobile agents can themselves be vulnerable – they may not be able to complete their task if they are refused access to the information they need. The tasks performed by mobile agents can be performed by other means. For example, web crawlers that need to access resources at web servers throughout the Internet work quite successfully by making remote invocations to server processes. For these reasons, the applicability of mobile agents may be limited.

2.3.2 Architectural patterns

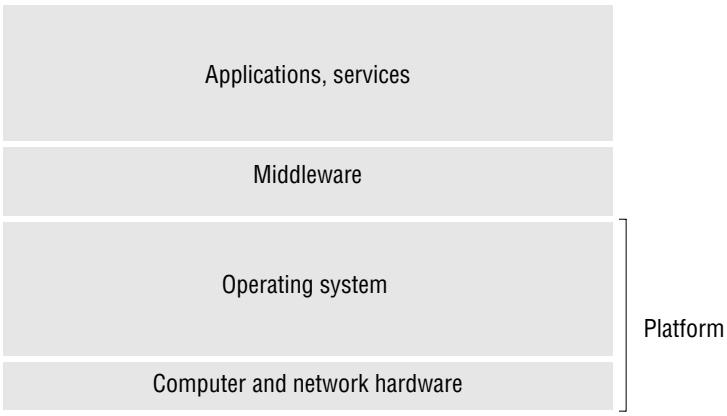
Architectural patterns build on the more primitive architectural elements discussed above and provide composite recurring structures that have been shown to work well in given circumstances. They are not themselves necessarily complete solutions but rather offer partial insights that, when combined with other patterns, lead the designer to a solution for a given problem domain.

This is a large topic, and many architectural patterns have been identified for distributed systems. In this section, we present several key architectural patterns in distributed systems, including layering and tiered architectures and the related concept of thin clients (including the specific mechanism of virtual network computing). We also examine web services as an architectural pattern and give pointers to others that may be applicable in distributed systems.

Layering • The concept of layering is a familiar one and is closely related to abstraction. In a layered approach, a complex system is partitioned into a number of layers, with a given layer making use of the services offered by the layer below. A given layer therefore offers a software abstraction, with higher layers being unaware of implementation details, or indeed of any other layers beneath them.

In terms of distributed systems, this equates to a vertical organization of services into service layers. A distributed service can be provided by one or more server processes, interacting with each other and with client processes in order to maintain a consistent system-wide view of the service's resources. For example, a network time service is implemented on the Internet based on the Network Time Protocol (NTP) by server processes running on hosts throughout the Internet that supply the current time to any client that requests it and adjust their version of the current time as a result of

Figure 2.7 Software and hardware service layers in distributed systems



interactions with each other. Given the complexity of distributed systems, it is often helpful to organize such services into layers. We present a common view of a layered architecture in Figure 2.7 and develop this view in increasing detail in Chapters 3 to 6.

Figure 2.7 introduces the important terms *platform* and *middleware*, which we define as follows:

- A platform for distributed systems and applications consists of the lowest-level hardware and software layers. These low-level layers provide services to the layers above them, which are implemented independently in each computer, bringing the system's programming interface up to a level that facilitates communication and coordination between processes. Intel x86/Windows, Intel x86/Solaris, Intel x86/Mac OS X, Intel x86/Linux and ARM/Symbian are major examples.
- Middleware was defined in Section 1.5.1 as a layer of software whose purpose is to mask heterogeneity and to provide a convenient programming model to application programmers. Middleware is represented by processes or objects in a set of computers that interact with each other to implement communication and resource-sharing support for distributed applications. It is concerned with providing useful building blocks for the construction of software components that can work with one another in a distributed system. In particular, it raises the level of the communication activities of application programs through the support of abstractions such as remote method invocation; communication between a group of processes; notification of events; the partitioning, placement and retrieval of shared data objects amongst cooperating computers; the replication of shared data objects; and the transmission of multimedia data in real time. We return to this important topic in Section 2.3.3 below.

Tiered architecture • Tiered architectures are complementary to layering. Whereas layering deals with the vertical organization of services into layers of abstraction, tiering is a technique to organize functionality of a given layer and place this functionality into

appropriate servers and, as a secondary consideration, on to physical nodes. This technique is most commonly associated with the organization of applications and services as in Figure 2.7 above, but it also applies to all layers of a distributed systems architecture.

Let us first examine the concepts of two- and three-tiered architecture. To illustrate this, consider the functional decomposition of a given application, as follows:

- the presentation logic, which is concerned with handling user interaction and updating the view of the application as presented to the user;
- the application logic, which is concerned with the detailed application-specific processing associated with the application (also referred to as the business logic, although the concept is not limited only to business applications);
- the data logic, which is concerned with the persistent storage of the application, typically in a database management system.

Now, let us consider the implementation of such an application using client-server technology. The associated two-tier and three-tier solutions are presented together for comparison in Figure 2.8 (a) and (b), respectively.

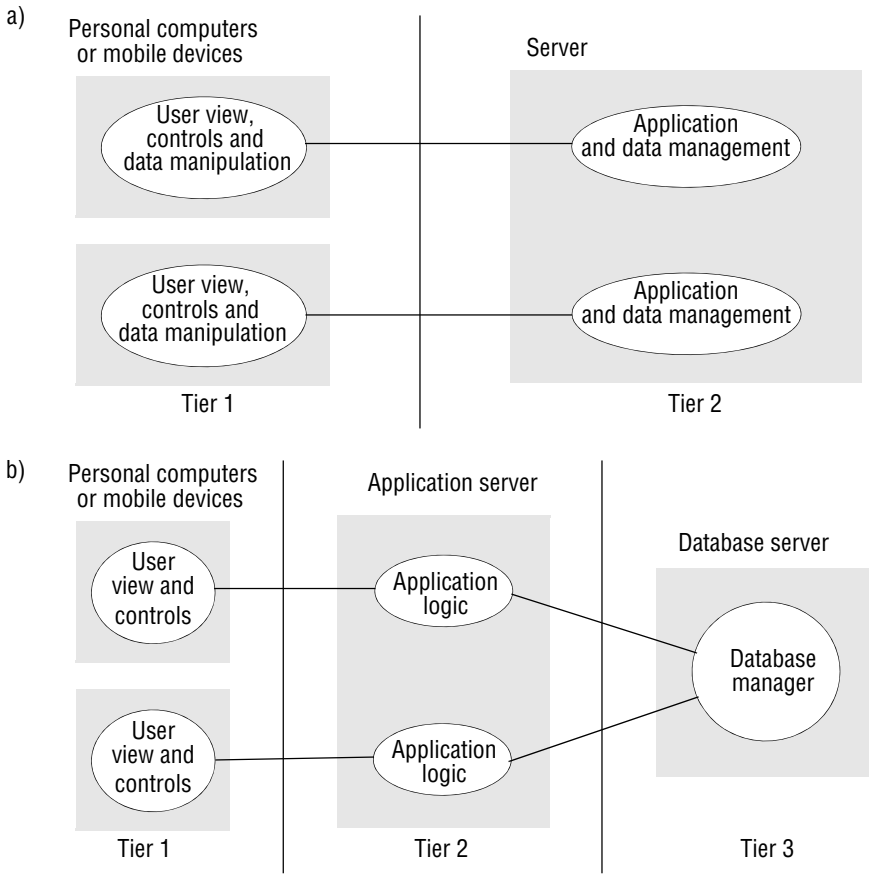
In the two-tier solution, the three aspects mentioned above must be partitioned into two processes, the client and the server. This is most commonly done by splitting the application logic, with some residing in the client and the remainder in the server (although other solutions are also possible). The advantage of this scheme is low latency in terms of interaction, with only one exchange of messages to invoke an operation. The disadvantage is the splitting of application logic across a process boundary, with the consequent restriction on which parts of the logic can be directly invoked from which other part.

In the three-tier solution, there is a one-to-one mapping from logical elements to physical servers and hence, for example, the application logic is held in one place, which in turn can enhance maintainability of the software. Each tier also has a well-defined role; for example, the third tier is simply a database offering a (potentially standardized) relational service interface. The first tier can also be a simple user interface allowing intrinsic support for thin clients (as discussed below). The drawbacks are the added complexity of managing three servers and also the added network traffic and latency associated with each operation.

Note that this approach generalizes to n-tiered (or multi-tier) solutions where a given application domain is partitioned into n logical elements, each mapped to a given server element. As an example, Wikipedia, the web-based publicly editable encyclopedia, adopts a multi-tier architecture to deal with the high volume of web requests (up to 60,000 page requests per second).

The role of AJAX: In Section 1.6 we introduced AJAX (Asynchronous Javascript And XML) as an extension to the standard client-server style of interaction used in the World Wide Web. AJAX meets the need for fine-grained communication between a Javascript front-end program running in a web browser and a server-based back-end program holding data describing the state of the application. To recapitulate, in the standard web style of interaction a browser sends an HTTP request to a server for a page, image or other resource with a given URL. The server replies by sending an entire page that is either read from a file on the server or generated by a program, depending on which type

Figure 2.8 Two-tier and three-tier architectures



of resource is identified in the URL. When the resultant content is received at the client, the browser presents it according to the relevant display method for its MIME type (*text*/*html*, *image*/*jpg*, etc.). Although a web page may be composed of several items of content of different types, the entire page is composed and presented by the browser in the manner specified in its HTML page definition.

This standard style of interaction constrains the development of web applications in several significant ways:

- Once the browser has issued an HTTP request for a new web page, the user is unable to interact with the page until the new HTML content is received and presented by the browser. This time interval is indeterminate, because it is subject to network and server delays.
- In order to update even a small part of the current page with additional data from the server, an entire new page must be requested and displayed. This results in a delayed response to the user, additional processing at both the client and the server and redundant network traffic.

Figure 2.9 AJAX example: soccer score updates

```
new Ajax.Request('scores.php?game=Arsenal:Liverpool',
      {onSuccess: updateScore});

function updateScore(request) {
.....
      ( request contains the state of the Ajax request including the returned result.
            The result is parsed to obtain some text giving the score, which is used
            to update the relevant portion of the current page.)
.....
 }
```

- The contents of a page displayed at a client cannot be updated in response to changes in the application data held at the server.

The introduction of Javascript, a cross-platform and cross-browser programming language that is downloaded and executed in the browser, constituted a first step towards the removal of those constraints. Javascript is a general-purpose language enabling both user interface and application logic to be programmed and executed in the context of a browser window.

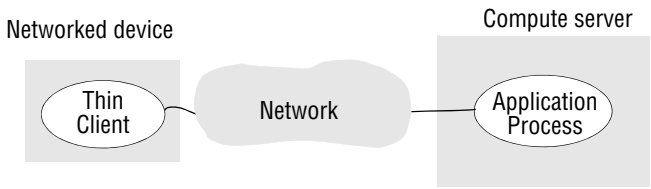
AJAX is the second innovative step that was needed to enable major interactive web applications to be developed and deployed. It enables Javascript front-end programs to request new data directly from server programs. Any data items can be requested and the current page updated selectively to show the new values. Indeed, the front end can react to the new data in any way that is useful for the application.

Many web applications allow users to access and update substantial shared datasets that may be subject to change in response to input from other clients or data feeds received by a server. They require a responsive front-end component running in each client browser to perform user interface actions such as menu selection, but they also require access to a dataset that must be held at server to enable sharing. Such datasets are generally too large and too dynamic to allow the use of any architecture based on the downloading of a copy of the entire application state to the client at the start of a user's session for manipulation by the client.

AJAX is the 'glue' that supports the construction of such applications; it provides a communication mechanism enabling front-end components running in a browser to issue requests and receive results from back-end components running on a server. Clients issue requests through the Javascript *XmlHttpRequest* object, which manages an HTTP exchange (see Section 1.6) with a server process. Because *XmlHttpRequest* has a complex API that is also somewhat browser-dependent, it is usually accessed through one of the many Javascript libraries that are available to support the development of web applications. In Figure 2.9 we illustrate its use in the *Prototype.js* Javascript library [www.prototypejs.org].

The example is an excerpt from a web application that displays a page listing up-to-date scores for soccer matches. Users may request updates of scores for individual games by clicking on the relevant line of the page, which executes the first line of the

Figure 2.10 Thin clients and computer servers



example. The *Ajax.Request* object sends an HTTP request to a *scores.php* program located at the same server as the web page. The *Ajax.Request* object then returns control, allowing the browser to continue to respond to other user actions in the same window or other windows. When the *scores.php* program has obtained the latest score it returns it in an HTTP response. The *Ajax.Request* object is then reactivated; it invokes the *updateScore* function (because it is the *onSuccess* action), which parses the result and inserts the score at the relevant position in the current page. The remainder of the page remains unaffected and is not reloaded.

This illustrates the type of communication used between Tier 1 and Tier 2 components. Although *Ajax.Request* (and the underlying *XmlHttpRequest* object) offers both synchronous and asynchronous communication, the asynchronous version is almost always used because the effect on the user interface of delayed server responses is unacceptable.

Our simple example illustrates the use of AJAX in a two-tier application. In a three-tier application the server component (*scores.php* in our example) would send a request to a data manager component (typically an SQL query to a database server) for the required data. That request would be synchronous, since there is no reason to return control to the server component until the request is satisfied.

The AJAX mechanism constitutes an effective technique for the construction of responsive web applications in the context of the indeterminate latency of the Internet, and it has been very widely deployed. The Google Maps application [www.google.com II] is an outstanding example. Maps are displayed as an array of contiguous 256 x 256 pixel images (called *tiles*). When the map is moved the visible tiles are repositioned by Javascript code in the browser and additional tiles needed to fill the visible area are requested with an AJAX call to a Google server. They are displayed as soon as they are received, but the browser continues to respond to user interaction while they are awaited.

Thin clients • The trend in distributed computing is towards moving complexity away from the end-user device towards services in the Internet. This is most apparent in the move towards cloud computing (discussed in Chapter 1) but can also be seen in tiered architectures, as discussed above. This trend has given rise to interest in the concept of a *thin client*, enabling access to sophisticated networked services, provided for example by a cloud solution, with few assumptions or demands on the client device. More specifically, the term thin client refers to a software layer that supports a window-based user interface that is local to the user while executing application programs or, more generally, accessing services on a remote computer. For example, Figure 2.10 illustrates a thin client accessing a compute server over the Internet. The advantage of this approach is that potentially simple local devices (including, for example, smart phones

and other resource-constrained devices) can be significantly enhanced with a plethora of networked services and capabilities. The main drawback of the thin client architecture is in highly interactive graphical activities such as CAD and image processing, where the delays experienced by users are increased to unacceptable levels by the need to transfer image and vector information between the thin client and the application process, due to both network and operating system latencies.

This concept has led to the emergence of *virtual network computing* (VNC). This technology was first introduced by researchers at the Olivetti and Oracle Research Laboratory [Richardson *et al.* 1998]; the initial concept has now evolved into implementations such as RealVNC [www.realvnc.com], which is a software solution, and Adventiq [www.adventiq.com], which is a hardware-based solution supporting the transmission of keyboard, video and mouse events over IP (KVM-over-IP). Other VNC implementationss include Apple Remote Desktop, TightVNC and Aqua Connect.

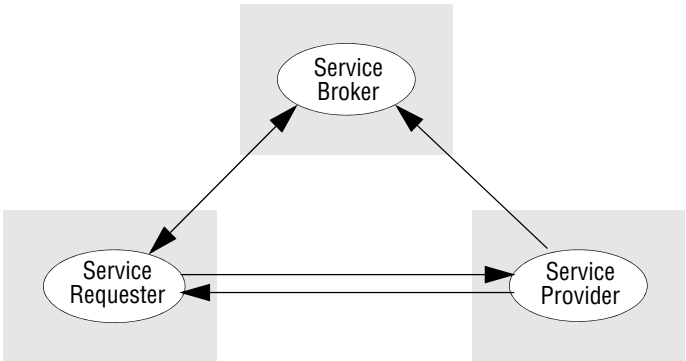
The concept is straightforward, providing remote access to graphical user interfaces. In this solution, a VNC client (or viewer) interacts with a VNC server through a VNC protocol. The protocol operates at a primitive level in terms of graphics support, based on framebuffers and featuring one operation: the placement of a rectangle of pixel data at a given position on the screen (some solutions, such as XenApp from Citrix operate at a higher level in terms of window operations [www.citrix.com]). This low-level approach ensures the protocol will work with any operating system or application. Although it is straightforward, the implication is that users are able to access their computer facilities from anywhere on a wide range of devices, representing a significant step forward in mobile computing.

Virtual network computing has superseded network computers, a previous attempt to realise thin client solutions through simple and inexpensive hardware devices that are completely reliant on networked services, downloading their operating system and any application software needed by the user from a remote file server. Since all the application data and code is stored by a file server, the users may migrate from one network computer to another. In practice, virtual network computing has proved to be a more flexible solution and now dominates the marketplace.

Other commonly occurring patterns • As mentioned above, a large number of architectural patterns have now been identified and documented. Here are a few key examples:

- The *proxy* pattern is a commonly recurring pattern in distributed systems designed particularly to support location transparency in remote procedure calls or remote method invocation. With this approach, a proxy is created in the local address space to represent the remote object. This proxy offers exactly the same interface as the remote object, and the programmer makes calls on this proxy object and hence does not need to be aware of the distributed nature of the interaction. The role of proxies in supporting such location transparency in RPC and RMI is discussed further in Chapter 5. Note that proxies can also be used to encapsulate other functionality, such as the placement policies of replication or caching.
- The use of *brokerage* in web services can usefully be viewed as an architectural pattern supporting interoperability in potentially complex distributed infrastructures. In particular, this pattern consists of the trio of service provider,

Figure 2.11 The web service architectural pattern



service requester and service broker (a service that matches services provided to those requested), as shown in Figure 2.11. This brokerage pattern is replicated in many areas of distributed systems, for example with the registry in Java RMI and the naming service in CORBA (as discussed in Chapters 5 and 8, respectively).

- *Reflection* is a pattern that is increasingly being used in distributed systems as a means of supporting both introspection (the dynamic discovery of properties of the system) and intercession (the ability to dynamically modify structure or behaviour). For example, the introspection capabilities of Java are used effectively in the implementation of RMI to provide generic dispatching (as discussed in Section 5.4.2). In a reflective system, standard service interfaces are available at the base level, but a meta-level interface is also available providing access to the components and their parameters involved in the realization of the services. A variety of techniques are generally available at the meta-level, including the ability to intercept incoming messages or invocations, to dynamically discover the interface offered by a given object and to discover and adapt the underlying architecture of the system. Reflection has been applied in a variety of areas in distributed systems, particularly within the field of reflective middleware, for example to support more configurable and reconfigurable middleware architectures [Kon *et al*. 2002].

Further examples of architectural patterns related to distributed systems can be found in Bushmann *et al*. [2007].

2.3.3 Associated middleware solutions

Middleware has already been introduced in Chapter 1 and revisited in the discussion of layering in Section 2.3.2 above. The task of middleware is to provide a higher-level programming abstraction for the development of distributed systems and, through layering, to abstract over heterogeneity in the underlying infrastructure to promote interoperability and portability. Middleware solutions are based on the architectural models introduced in Section 2.3.1 and also support more complex architectural

Figure 2.12 Categories of middleware

Major categories:	*Subcategory*	*Example systems*
Distributed objects (Chapters 5, 8)	Standard	RM-ODP
	Platform	CORBA
	Platform	Java RMI
Distributed components (Chapter 8)	Lightweight components	Fractal
	Lightweight components	OpenCOM
	Application servers	SUN EJB
	Application servers	CORBA Component Model
	Application servers	JBoss
Publish-subscribe systems (Chapter 6)	-	CORBA Event Service
	-	Scribe
	-	JMS
Message queues (Chapter 6)	-	Websphere MQ
	-	JMS
Web services (Chapter 9)	Web services	Apache Axis
	Grid services	The Globus Toolkit
Peer-to-peer (Chapter 10)	Routing overlays	Pastry
	Routing overlays	Tapestry
	Application-specific	Squirrel
	Application-specific	OceanStore
	Application-specific	Ivy
	Application-specific	Gnutella

patterns. In this section, we briefly review the major classes of middleware that exist today and prepare the ground for further study of these solutions in the rest of the book.

Categories of middleware • Remote procedure calling packages such as Sun RPC (Chapter 5) and group communication systems such as ISIS (Chapters 6 and 18) were amongst the earliest instances of middleware. Since then a wide range of styles of middleware have emerged, based largely on the architectural models introduced above. We present a taxonomy of such middleware platforms in Figure 2.12, including cross-references to other chapters that cover the various categories in more detail. It must be stressed that the categorizations are not exact and that modern middleware platforms tend to offer hybrid solutions. For example, many distributed object platforms offer distributed event services to complement the more traditional support for remote method invocation. Similarly, many component-based platforms (and indeed other categories of platform) also support web service interfaces and standards, for reasons of interoperability. It should also be stressed that this taxonomy is not intended to be complete in terms of the set of middleware standards and technologies available today,

but rather is intended to be indicative of the major classes of middleware. Other solutions (not shown) tend to be more specific, for example offering particular communication paradigms such as message passing, remote procedure calls, distributed shared memory, tuple spaces or group communication.

The top-level categorization of middleware in Figure 2.12 is driven by the choice of communicating entities and associated communication paradigms, and follows five of the main architectural models: distributed objects, distributed components, publish-subscribe systems, message queues and web services. These are supplemented by peer-to-peer systems, a rather separate branch of middleware based on the cooperative approach discussed in Section 2.3.1. The subcategory of distributed components shown as application servers also provides direct support for three-tier architectures. In particular, application servers provide structure to support a separation between application logic and data storage, along with support for other properties such as security and reliability. Further detail is deferred until Chapter 8.

In addition to programming abstractions, middleware can also provide infrastructural distributed system services for use by application programs or other services. These infrastructural services are tightly bound to the distributed programming model that the middleware provides. For example, CORBA (Chapter 8) provides applications with a range of CORBA services, including support for making applications secure and reliable. As mentioned above and discussed further in Chapter 8, application servers also provide intrinsic support for such services.

Limitations of middleware • Many distributed applications rely entirely on the services provided by middleware to support their needs for communication and data sharing. For example, an application that is suited to the client-server model such as a database of names and addresses, can rely on middleware that provides only remote method invocation.

Much has been achieved in simplifying the programming of distributed systems through the development of middleware support, but some aspects of the dependability of systems require support at the application level.

Consider the transfer of large electronic mail messages from the mail host of the sender to that of the recipient. At first sight this a simple application of the TCP data transmission protocol (discussed in Chapter 3). But consider the problem of a user who attempts to transfer a very large file over a potentially unreliable network. TCP provides some error detection and correction, but it cannot recover from major network interruptions. Therefore the mail transfer service adds another level of fault tolerance, maintaining a record of progress and resuming transmission using a new TCP connection if the original one breaks.

A classic paper by Saltzer, Reed and Clarke [Saltzer *et al.* 1984] makes a similar and valuable point about the design of distributed systems, which they call the 'the end-to-end argument'. To paraphrase their statement:

> Some communication-related functions can be completely and reliably implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that function as a feature of the communication system itself is not always sensible. (Although an incomplete version of the function provided by the communication system may sometimes be useful as a performance enhancement).

It can be seen that their argument runs counter to the view that all communication activities can be abstracted away from the programming of applications by the introduction of appropriate middleware layers.

The nub of their argument is that correct behaviour in distributed programs depends upon checks, error-correction mechanisms and security measures at many levels, some of which require access to data within the application's address space. Any attempt to perform the checks within the communication system alone will guarantee only part of the required correctness. The same work is therefore likely to be duplicated in application programs, wasting programming effort and, more importantly, adding unnecessary complexity and redundant computations.

There is not space to detail their arguments further here, but reading the cited paper is strongly recommended – it is replete with illuminating examples. One of the original authors has recently pointed out that the substantial benefits that the use of the argument brought to the design of the Internet are placed at risk by recent moves towards the specialization of network services to meet current application requirements [www.reed.com].

This argument poses a real dilemma for middleware designers, and indeed the difficulties are increasing given the wide range of applications (and associated environmental conditions) in contemporary distributed systems (see Chapter 1). In essence, the right underlying middleware behaviour is a function of the requirements of a given application or set of applications and the associated environmental context, such as the state and style of the underlying network. This perception is driving interest in context-aware and adaptive solutions to middleware, as discussed in Kon *et al* [2002].

2.4 Fundamental models

All the above, quite different, models of systems share some fundamental properties. In particular, all of them are composed of processes that communicate with one another by sending messages over a computer network. All of the models share the design requirements of achieving the performance and reliability characteristics of processes and networks and ensuring the security of the resources in the system. In this section, we present models based on the fundamental properties that allow us to be more specific about their characteristics and the failures and security risks they might exhibit.

In general, such a fundamental model should contain only the essential ingredients that we need to consider in order to understand and reason about some aspects of a system's behaviour. The purpose of such a model is:

- To make explicit all the relevant assumptions about the systems we are modelling.
- To make generalizations concerning what is possible or impossible, given those assumptions. The generalizations may take the form of general-purpose algorithms or desirable properties that are guaranteed. The guarantees are dependent on logical analysis and, where appropriate, mathematical proof.

There is much to be gained by knowing what our designs do, and do not, depend upon. It allows us to decide whether a design will work if we try to implement it in a particular system: we need only ask whether our assumptions hold in that system. Also, by making

our assumptions clear and explicit, we can hope to prove system properties using mathematical techniques. These properties will then hold for any system meeting our assumptions. Finally, by abstracting only the essential system entities and characteristics away from details such as hardware, we can clarify our understanding of our systems.

The aspects of distributed systems that we wish to capture in our fundamental models are intended to help us to discuss and reason about:

Interaction: Computation occurs within processes; the processes interact by passing messages, resulting in communication (information flow) and coordination (synchronization and ordering of activities) between processes. In the analysis and design of distributed systems we are concerned especially with these interactions. The interaction model must reflect the facts that communication takes place with delays that are often of considerable duration, and that the accuracy with which independent processes can be coordinated is limited by these delays and by the difficulty of maintaining the same notion of time across all the computers in a distributed system.

Failure: The correct operation of a distributed system is threatened whenever a fault occurs in any of the computers on which it runs (including software faults) or in the network that connects them. Our model defines and classifies the faults. This provides a basis for the analysis of their potential effects and for the design of systems that are able to tolerate faults of each type while continuing to run correctly.

Security: The modular nature of distributed systems and their openness exposes them to attack by both external and internal agents. Our security model defines and classifies the forms that such attacks may take, providing a basis for the analysis of threats to a system and for the design of systems that are able to resist them.

As aids to discussion and reasoning, the models introduced in this chapter are necessarily simplified, omitting much of the detail of real-world systems. Their relationship to real-world systems, and the solution in that context of the problems that the models help to bring out, is the main subject of this book.

2.4.1 Interaction model

The discussion of system architectures in Section 2.3 indicates that fundamentally distributed systems are composed of many processes, interacting in complex ways. For example:

- Multiple server processes may cooperate with one another to provide a service; the examples mentioned above were the Domain Name System, which partitions and replicates its data at servers throughout the Internet, and Sun's Network Information Service, which keeps replicated copies of password files at several servers in a local area network.
- A set of peer processes may cooperate with one another to achieve a common goal: for example, a voice conferencing system that distributes streams of audio data in a similar manner, but with strict real-time constraints.

Most programmers will be familiar with the concept of an *algorithm* – a sequence of steps to be taken in order to perform a desired computation. Simple programs are

controlled by algorithms in which the steps are strictly sequential. The behaviour of the program and the state of the program's variables is determined by them. Such a program is executed as a single process. Distributed systems composed of multiple processes such as those outlined above are more complex. Their behaviour and state can be described by a *distributed algorithm* – a definition of the steps to be taken by each of the processes of which the system is composed, *including the transmission of messages between them*. Messages are transmitted between processes to transfer information between them and to coordinate their activity.

The rate at which each process proceeds and the timing of the transmission of messages between them cannot in general be predicted. It is also difficult to describe all the states of a distributed algorithm, because it must deal with the failures of one or more of the processes involved or the failure of message transmissions.

Interacting processes perform all of the activity in a distributed system. Each process has its own state, consisting of the set of data that it can access and update, including the variables in its program. The state belonging to each process is completely private – that is, it cannot be accessed or updated by any other process.

In this section, we discuss two significant factors affecting interacting processes in a distributed system:

- Communication performance is often a limiting characteristic.
- It is impossible to maintain a single global notion of time.

Performance of communication channels • The communication channels in our model are realized in a variety of ways in distributed systems − for example, by an implementation of streams or by simple message passing over a computer network. Communication over a computer network has the following performance characteristics relating to latency, bandwidth and jitter:

- The delay between the start of a message's transmission from one process and the beginning of its receipt by another is referred to as *latency*. The latency includes:
  - The time taken for the first of a string of bits transmitted through a network to reach its destination. For example, the latency for the transmission of a message through a satellite link is the time for a radio signal to travel to the satellite and back.
  - The delay in accessing the network, which increases significantly when the network is heavily loaded. For example, for Ethernet transmission the sending station waits for the network to be free of traffic.
  - The time taken by the operating system communication services at both the sending and the receiving processes, which varies according to the current load on the operating systems.
- The *bandwidth* of a computer network is the total amount of information that can be transmitted over it in a given time. When a large number of communication channels are using the same network, they have to share the available bandwidth.
- *Jitter* is the variation in the time taken to deliver a series of messages. Jitter is relevant to multimedia data. For example, if consecutive samples of audio data are played with differing time intervals, the sound will be badly distorted.

Computer clocks and timing events • Each computer in a distributed system has its own internal clock, which can be used by local processes to obtain the value of the current time. Therefore two processes running on different computers can each associate timestamps with their events. However, even if the two processes read their clocks at the same time, their local clocks may supply different time values. This is because computer clocks drift from perfect time and, more importantly, their drift rates differ from one another. The term *clock drift rate* refers to the rate at which a computer clock deviates from a perfect reference clock. Even if the clocks on all the computers in a distributed system are set to the same time initially, their clocks will eventually vary quite significantly unless corrections are applied.

There are several approaches to correcting the times on computer clocks. For example, computers may use radio receivers to get time readings from the Global Positioning System with an accuracy of about 1 microsecond. But GPS receivers do not operate inside buildings, nor can the cost be justified for every computer. Instead, a computer that has an accurate time source such as GPS can send timing messages to other computers in its network. The resulting agreement between the times on the local clocks is, of course, affected by variable message delays. For a more detailed discussion of clock drift and clock synchronization, see Chapter 14.

Two variants of the interaction model • In a distributed system it is hard to set limits on the time that can be taken for process execution, message delivery or clock drift. Two opposing extreme positions provide a pair of simple models − the first has a strong assumption of time and the second makes no assumptions about time:

Synchronous distributed systems: Hadzilacos and Toueg [1994] define a synchronous distributed system to be one in which the following bounds are defined:

- The time to execute each step of a process has known lower and upper bounds.
- Each message transmitted over a channel is received within a known bounded time.
- Each process has a local clock whose drift rate from real time has a known bound.

It is possible to suggest likely upper and lower bounds for process execution time, message delay and clock drift rates in a distributed system, but it is difficult to arrive at realistic values and to provide guarantees of the chosen values. Unless the values of the bounds can be guaranteed, any design based on the chosen values will not be reliable. However, modelling an algorithm as a synchronous system may be useful for giving some idea of how it will behave in a real distributed system. In a synchronous system it is possible to use timeouts, for example, to detect the failure of a process, as shown in Section 2.4.2 below.

Synchronous distributed systems can be built. What is required is for the processes to perform tasks with known resource requirements for which they can be guaranteed sufficient processor cycles and network capacity, and for processes to be supplied with clocks with bounded drift rates.

Asynchronous distributed systems: Many distributed systems, such as the Internet, are very useful without being able to qualify as synchronous systems. Therefore we need an alternative model. An asynchronous distributed system is one in which there are no bounds on:

- Process execution speeds – for example, one process step may take only a picosecond and another a century; all that can be said is that each step may take an arbitrarily long time.
- Message transmission delays – for example, one message from process A to process B may be delivered in negligible time and another may take several years. In other words, a message may be received after an arbitrarily long time.
- Clock drift rates – again, the drift rate of a clock is arbitrary.

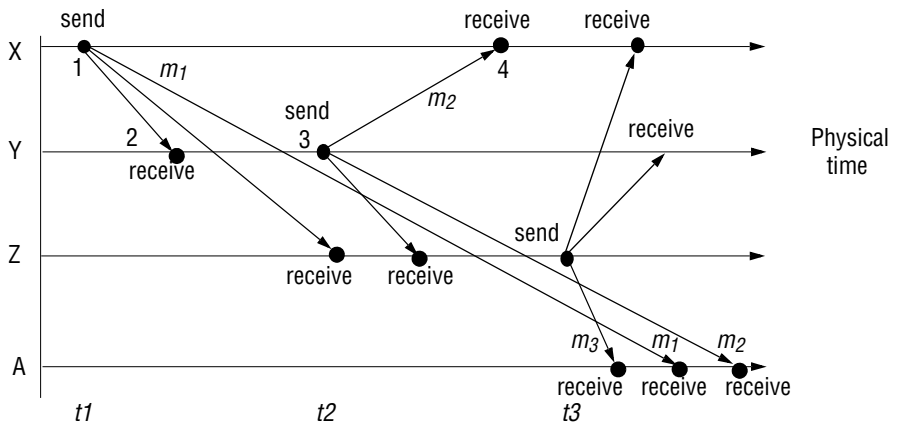
The asynchronous model allows no assumptions about the time intervals involved in any execution. This exactly models the Internet, in which there is no intrinsic bound on server or network load and therefore on how long it takes, for example, to transfer a file using FTP. Sometimes an email message can take days to arrive. The box on this page illustrates the difficulty of reaching an agreement in an asynchronous distributed system.

But some design problems can be solved even with these assumptions. For example, although the Web cannot always provide a particular response within a reasonable time limit, browsers have been designed to allow users to do other things while they are waiting. Any solution that is valid for an asynchronous distributed system is also valid for a synchronous one.

Actual distributed systems are very often asynchronous because of the need for processes to share the processors and for communication channels to share the

Agreement in Pepperland • Two divisions of the Pepperland army, 'Apple' and 'Orange', are encamped at the top of two nearby hills. Further along the valley below are the invading Blue Meanies. The Pepperland divisions are safe as long as they remain in their encampments, and they can send out messengers reliably through the valley to communicate. The Pepperland divisions need to agree on which of them will lead the charge against the Blue Meanies and when the charge will take place. Even in an asynchronous Pepperland, it is possible to agree on who will lead the charge. For example, each division can send the number of its remaining members, and the one with most will lead (if a tie, division Apple wins over Orange). But when should they charge? Unfortunately, in asynchronous Pepperland, the messengers are very variable in their speed. If, say, Apple sends a messenger with the message 'Charge!', Orange might not receive the message for, say, three hours; or it may take, say, five minutes to arrive. In a synchronous Pepperland, there is still a coordination problem, but the divisions know some useful constraints: every message takes at least *min* minutes and at most *max* minutes to arrive. If the division that will lead the charge sends a message 'Charge!', it waits for *min* minutes; then it charges. The other division waits for 1 minute after receipt of the message, then charges. Its charge is guaranteed to be after the leading division's, but no more than ($max - min + 1$) minutes after it.

Figure 2.13 Real-time ordering of events



network. For example, if too many processes of unknown character are sharing a processor, then the resulting performance of any one of them cannot be guaranteed. But there are many design problems that cannot be solved for an asynchronous system that can be solved when some aspects of time are used. The need for each element of a multimedia data stream to be delivered before a deadline is such a problem. For problems such as these, a synchronous model is required.

Event ordering • In many cases, we are interested in knowing whether an event (sending or receiving a message) at one process occurred before, after or concurrently with another event at another process. The execution of a system can be described in terms of events and their ordering despite the lack of accurate clocks.

For example, consider the following set of exchanges between a group of email users, X, Y, Z and A, on a mailing list:

1. User X sends a message with the subject *Meeting*.
2. Users Y and Z reply by sending a message with the subject *Re: Meeting*.

In real time, X's message is sent first, and Y reads it and replies; Z then reads both X's message and Y's reply and sends another reply, which references both X's and Y's messages. But due to the independent delays in message delivery, the messages may be delivered as shown in Figure 2.13, and some users may view these two messages in the wrong order. For example, user A might see:

Inbox:		
Item	*From*	*Subject*
23	Z	Re: Meeting
24	X	Meeting
25	Y	Re: Meeting

If the clocks on X's, Y's and Z's computers could be synchronized, then each message could carry the time on the local computer's clock when it was sent. For example, messages m_1, m_2 and m_3 would carry times t_1, t_2 and t_3 where $t_1<t_2<t_3$. The messages received will be displayed to users according to their time ordering. If the clocks are roughly synchronized, then these timestamps will often be in the correct order.

Since clocks cannot be synchronized perfectly across a distributed system, Lamport [1978] proposed a model of *logical time* that can be used to provide an ordering among the events at processes running in different computers in a distributed system. Logical time allows the order in which the messages are presented to be inferred without recourse to clocks. It is presented in detail in Chapter 14, but we suggest here how some aspects of logical ordering can be applied to our email ordering problem.

Logically, we know that a message is received after it was sent. Therefore we can state a logical ordering for pairs of events shown in Figure 2.13, for example, considering only the events concerning X and Y:

X sends m_1 before Y receives m_1; Y sends m_2 before X receives m_2.

We also know that replies are sent after receiving messages, so we have the following logical ordering for Y:

Y receives m_1 before sending m_2.

Logical time takes this idea further by assigning a number to each event corresponding to its logical ordering, so that later events have higher numbers than earlier ones. For example, Figure 2.13 shows the numbers 1 to 4 on the events at X and Y.

2.4.2 Failure model

In a distributed system both processes and communication channels may fail – that is, they may depart from what is considered to be correct or desirable behaviour. The failure model defines the ways in which failure may occur in order to provide an understanding of the effects of failures. Hadzilacos and Toueg [1994] provide a taxonomy that distinguishes between the failures of processes and communication channels. These are presented under the headings omission failures, arbitrary failures and timing failures.

The failure model will be used throughout the book. For example:

- In Chapter 4, we present the Java interfaces to datagram and stream communication, which provide different degrees of reliability.
- Chapter 5 presents the request-reply protocol, which supports RMI. Its failure characteristics depend on the failure characteristics of both processes and communication channels. The protocol can be built from either datagram or stream communication. The choice may be decided according to a consideration of simplicity of implementation, performance and reliability.
- Chapter 17 presents the two-phase commit protocol for transactions. It is designed to complete in the face of well-defined failures of processes and communication channels.

Omission failures • The faults classified as *omission failures* refer to cases when a process or communication channel fails to perform actions that it is supposed to do.

Process omission failures: The chief omission failure of a process is to crash. When we say that a process has crashed we mean that it has halted and will not execute any further steps of its program ever. The design of services that can survive in the presence of faults can be simplified if it can be assumed that the services on which they depend crash cleanly – that is, their processes either function correctly or else stop. Other processes may be able to detect such a crash by the fact that the process repeatedly fails to respond to invocation messages. However, this method of crash detection relies on the use of *timeouts* – that is, a method in which one process allows a fixed period of time for something to occur. In an asynchronous system a timeout can indicate only that a process is not responding – it may have crashed or may be slow, or the messages may not have arrived.

A process crash is called *fail-stop* if other processes can detect certainly that the process has crashed. Fail-stop behaviour can be produced in a synchronous system if the processes use timeouts to detect when other processes fail to respond and messages are guaranteed to be delivered. For example, if processes p and q are programmed for q to reply to a message from p, and if process p has received no reply from process q in a maximum time measured on p's local clock, then process p may conclude that process q has failed. The box opposite illustrates the difficulty of detecting failures in an asynchronous system or of reaching agreement in the presence of failures.

Communication omission failures: Consider the communication primitives *send* and *receive*. A process p performs a *send* by inserting the message m in its outgoing message buffer. The communication channel transports m to q's incoming message buffer. Process q performs a *receive* by taking m from its incoming message buffer and delivering it (see Figure 2.14). The outgoing and incoming message buffers are typically provided by the operating system.

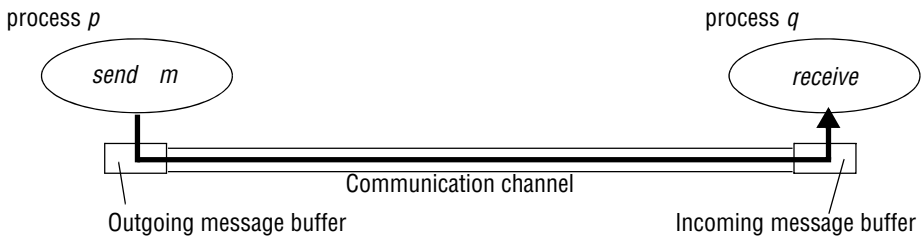
The communication channel produces an omission failure if it does not transport a message from p's outgoing message buffer to q's incoming message buffer. This is known as 'dropping messages' and is generally caused by lack of buffer space at the receiver or at an intervening gateway, or by a network transmission error, detected by a checksum carried with the message data. Hadzilacos and Toueg [1994] refer to the loss of messages between the sending process and the outgoing message buffer as *send-omission failures*, to loss of messages between the incoming message buffer and the receiving process as *receive-omission failures*, and to loss of messages in between as *channel omission failures*. The omission failures are classified together with arbitrary failures in Figure 2.15.

Failures can be categorized according to their severity. All of the failures we have described so far are *benign* failures. Most failures in distributed systems are benign. Benign failures include failures of omission as well as timing failures and performance failures.

Arbitrary failures • The term *arbitrary* or *Byzantine* failure is used to describe the worst possible failure semantics, in which any type of error may occur. For example, a process may set wrong values in its data items, or it may return a wrong value in response to an invocation.

An arbitrary failure of a process is one in which it arbitrarily omits intended processing steps or takes unintended processing steps. Arbitrary failures in processes

Figure 2.14 Processes and channels



cannot be detected by seeing whether the process responds to invocations, because it might arbitrarily omit to reply.

Communication channels can suffer from arbitrary failures; for example, message contents may be corrupted, nonexistent messages may be delivered or real messages may be delivered more than once. Arbitrary failures of communication channels are rare

Failure detection • In the case of the Pepperland divisions encamped at the tops of hills (see page 81), suppose that the Blue Meanies are after all sufficient in strength to attack and defeat either division while encamped – that is, that either can fail. Suppose further that, while undefeated, the divisions regularly send messengers to report their status. In an asynchronous system, neither division can distinguish whether the other has been defeated or the time it is taking for the messengers to cross the intervening valley is just very long. In a synchronous Pepperland, a division can tell for sure if the other has been defeated by the absence of a regular messenger. However, the other division may have been defeated just after it sent the latest messenger.

Impossibility of reaching timely agreement in the presence of communication failures • We have been assuming that the Pepperland messengers always manage to cross the valley eventually; but now suppose that the Blue Meanies can capture any messenger and prevent them from arriving. (We shall assume it is impossible for the Blue Meanies to brainwash the messengers to give the wrong message – the Meanies are not aware of their treacherous Byzantine precursors.) Can the Apple and Orange divisions send messages so that they both consistently decide to charge at the Meanies or both decide to surrender? Unfortunately, as the Pepperland theoretician Ringo the Great proved, in these circumstances the divisions cannot guarantee to decide consistently what to do. To see this, assume to the contrary that the divisions run a Pepperland protocol that achieves agreement. Each proposes 'Charge!' or 'Surrender!', and the protocol results in them both agreeing on one or the other course of action. Now consider the last message sent in any run of the protocol. The messenger that carries it could be captured by the Blue Meanies, so the end result must be the same whether the message arrives or not. We can dispense with it. Now we can apply the same argument to the final message that remains. But this argument applies again to that message and will continue to apply, so we shall end up with no messages sent at all! This shows that no protocol that guarantees agreement between the Pepperland divisions can exist if messengers can be captured.

Figure 2.15 Omission and arbitrary failures

Class of failure	*Affects*	*Description*
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a *send* operation but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times or commit omissions; a process may stop or take an incorrect step.

because the communication software is able to recognize them and reject the faulty messages. For example, checksums are used to detect corrupted messages, and message sequence numbers can be used to detect nonexistent and duplicated messages.

Timing failures • Timing failures are applicable in synchronous distributed systems where time limits are set on process execution time, message delivery time and clock drift rate. Timing failures are listed in Figure 2.16. Any one of these failures may result in responses being unavailable to clients within a specified time interval.

In an asynchronous distributed system, an overloaded server may respond too slowly, but we cannot say that it has a timing failure since no guarantee has been offered.

Real-time operating systems are designed with a view to providing timing guarantees, but they are more complex to design and may require redundant hardware. Most general-purpose operating systems such as UNIX do not have to meet real-time constraints.

Timing is particularly relevant to multimedia computers with audio and video channels. Video information can require a very large amount of data to be transferred. Delivering such information without timing failures can make very special demands on both the operating system and the communication system.

Masking failures • Each component in a distributed system is generally constructed from a collection of other components. It is possible to construct reliable services from components that exhibit failures. For example, multiple servers that hold replicas of data can continue to provide a service when one of them crashes. A knowledge of the failure characteristics of a component can enable a new service to be designed to mask the

Figure 2.16 Timing failures

Class of failure	*Affects*	*Description*
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

failure of the components on which it depends. A service *masks* a failure either by hiding it altogether or by converting it into a more acceptable type of failure. For an example of the latter, checksums are used to mask corrupted messages, effectively converting an arbitrary failure into an omission failure. We shall see in Chapters 3 and 4 that omission failures can be hidden by using a protocol that retransmits messages that do not arrive at their destination. Chapter 18 presents masking by means of replication. Even process crashes may be masked, by replacing the process and restoring its memory from information stored on disk by its predecessor.

Reliability of one-to-one communication • Although a basic communication channel can exhibit the omission failures described above, it is possible to use it to build a communication service that masks some of those failures.

The term *reliable communication* is defined in terms of validity and integrity as follows:

Validity: Any message in the outgoing message buffer is eventually delivered to the incoming message buffer.

Integrity: The message received is identical to one sent, and no messages are delivered twice.

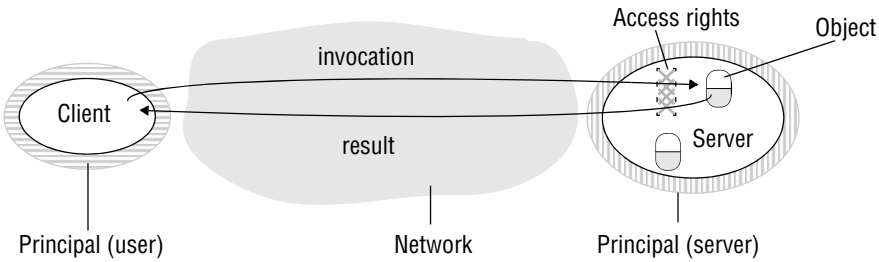
The threats to integrity come from two independent sources:

- Any protocol that retransmits messages but does not reject a message that arrives twice. Protocols can attach sequence numbers to messages so as to detect those that are delivered twice.
- Malicious users that may inject spurious messages, replay old messages or tamper with messages. Security measures can be taken to maintain the integrity property in the face of such attacks.

2.4.3 Security model

In Chapter 1 we identified the sharing of resources as a motivating factor for distributed systems, and in Section 2.3 we described their architecture in terms of processes, potentially encapsulating higher-level abstractions such as objects, components or

Figure 2.17 Objects and principals



services, and providing access to them through interactions with other processes. That architectural model provides the basis for our security model:

> the security of a distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access.

Protection is described in terms of objects, although the concepts apply equally well to resources of all types.

Protecting objects • Figure 2.17 shows a server that manages a collection of objects on behalf of some users. The users can run client programs that send invocations to the server to perform operations on the objects. The server carries out the operation specified in each invocation and sends the result to the client.

Objects are intended to be used in different ways by different users. For example, some objects may hold a user's private data, such as their mailbox, and other objects may hold shared data such as web pages. To support this, *access rights* specify who is allowed to perform the operations of an object – for example, who is allowed to read or to write its state.

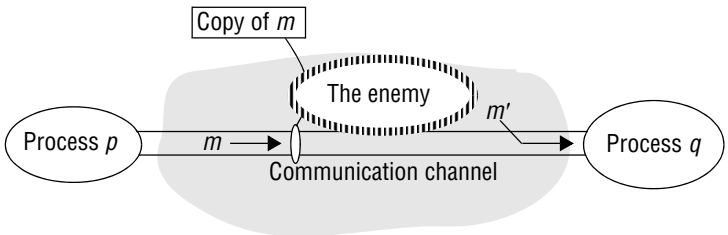
Thus we must include users in our model as the beneficiaries of access rights. We do so by associating with each invocation and each result the authority on which it is issued. Such an authority is called a *principal*. A principal may be a user or a process. In our illustration, the invocation comes from a user and the result from a server.

The server is responsible for verifying the identity of the principal behind each invocation and checking that they have sufficient access rights to perform the requested operation on the particular object invoked, rejecting those that do not. The client may check the identity of the principal behind the server to ensure that the result comes from the required server.

Securing processes and their interactions • Processes interact by sending messages. The messages are exposed to attack because the network and the communication service that they use are open, to enable any pair of processes to interact. Servers and peer processes expose their interfaces, enabling invocations to be sent to them by any other process.

Distributed systems are often deployed and used in tasks that are likely to be subject to external attacks by hostile users. This is especially true for applications that

Figure 2.18 The enemy



handle financial transactions, confidential or classified information or any other information whose secrecy or integrity is crucial. Integrity is threatened by security violations as well as communication failures. So we know that there are likely to be threats to the processes of which such applications are composed and to the messages travelling between the processes. But how can we analyze these threats in order to identify and defeat them? The following discussion introduces a model for the analysis of security threats.

The enemy • To model security threats, we postulate an enemy (sometimes also known as the adversary) that is capable of sending any message to any process and reading or copying any message sent between a pair of processes, as shown in Figure 2.18. Such attacks can be made simply by using a computer connected to a network to run a program that reads network messages addressed to other computers on the network, or a program that generates messages that make false requests to services, purporting to come from authorized users. The attack may come from a computer that is legitimately connected to the network or from one that is connected in an unauthorized manner.

The threats from a potential enemy include *threats to processes* and *threats to communication channels*.

Threats to processes: A process that is designed to handle incoming requests may receive a message from any other process in the distributed system, and it cannot necessarily determine the identity of the sender. Communication protocols such as IP do include the address of the source computer in each message, but it is not difficult for an enemy to generate a message with a forged source address. This lack of reliable knowledge of the source of a message is a threat to the correct functioning of both servers and clients, as explained below:

Servers: Since a server can receive invocations from many different clients, it cannot necessarily determine the identity of the principal behind any particular invocation. Even if a server requires the inclusion of the principal's identity in each invocation, an enemy might generate an invocation with a false identity. Without reliable knowledge of the sender's identity, a server cannot tell whether to perform the operation or to reject it. For example, a mail server would not know whether the user behind an invocation that requests a mail item from a particular mailbox is allowed to do so or whether it was a request from an enemy.

Clients: When a client receives the result of an invocation from a server, it cannot necessarily tell whether the source of the result message is from the intended server

or from an enemy, perhaps 'spoofing' the mail server. Thus the client could receive a result that was unrelated to the original invocation, such as a false mail item (one that is not in the user's mailbox).

Threats to communication channels: An enemy can copy, alter or inject messages as they travel across the network and its intervening gateways. Such attacks present a threat to the privacy and integrity of information as it travels over the network and to the integrity of the system. For example, a result message containing a user's mail item might be revealed to another user or it might be altered to say something quite different.

Another form of attack is the attempt to save copies of messages and to replay them at a later time, making it possible to reuse the same message over and over again. For example, someone could benefit by resending an invocation message requesting a transfer of a sum of money from one bank account to another.

All these threats can be defeated by the use of *secure channels*, which are described below and are based on cryptography and authentication.

Defeating security threats • Here we introduce the main techniques on which secure systems are based. Chapter 11 discusses the design and implementation of secure distributed systems in much more detail.

Cryptography and shared secrets: Suppose that a pair of processes (for example, a particular client and a particular server) share a secret; that is, they both know the secret but no other process in the distributed system knows it. Then if a message exchanged by that pair of processes includes information that proves the sender's knowledge of the shared secret, the recipient knows for sure that the sender was the other process in the pair. Of course, care must be taken to ensure that the shared secret is not revealed to an enemy.

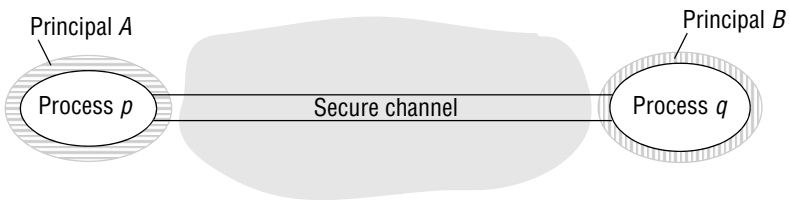
Cryptography is the science of keeping messages secure, and *encryption* is the process of scrambling a message in such a way as to hide its contents. Modern cryptography is based on encryption algorithms that use secret keys – large numbers that are difficult to guess – to transform data in a manner that can only be reversed with knowledge of the corresponding decryption key.

Authentication: The use of shared secrets and encryption provides the basis for the *authentication* of messages – proving the identities supplied by their senders. The basic authentication technique is to include in a message an encrypted portion that contains enough of the contents of the message to guarantee its authenticity. The authentication portion of a request to a file server to read part of a file, for example, might include a representation of the requesting principal's identity, the identity of the file and the date and time of the request, all encrypted with a secret key shared between the file server and the requesting process. The server would decrypt this and check that it corresponds to the unencrypted details specified in the request.

Secure channels: Encryption and authentication are used to build secure channels as a service layer on top of existing communication services. A secure channel is a communication channel connecting a pair of processes, each of which acts on behalf of a principal, as shown in Figure 2.19. A secure channel has the following properties:

- Each of the processes knows reliably the identity of the principal on whose behalf the other process is executing. Therefore if a client and server communicate via a secure channel, the server knows the identity of the principal behind the

Figure 2.19 Secure channels



invocations and can check their access rights before performing an operation. This enables the server to protect its objects correctly and allows the client to be sure that it is receiving results from a *bona fide* server.

- A secure channel ensures the privacy and integrity (protection against tampering) of the data transmitted across it.
- Each message includes a physical or logical timestamp to prevent messages from being replayed or reordered.

The construction of secure channels is discussed in detail in Chapter 11. Secure channels have become an important practical tool for securing electronic commerce and the protection of communication. Virtual private networks (VPNs, discussed in Chapter 3) and the Secure Sockets Layer (SSL) protocol (discussed in Chapter 11) are instances.

Other possible threats from an enemy • Section 1.5.3 introduced very briefly two further security threats – denial of service attacks and the deployment of mobile code. We reiterate these as possible opportunities for the enemy to disrupt the activities of processes:

Denial of service: This is a form of attack in which the enemy interferes with the activities of authorized users by making excessive and pointless invocations on services or message transmissions in a network, resulting in overloading of physical resources (network bandwidth, server processing capacity). Such attacks are usually made with the intention of delaying or preventing actions by other users. For example, the operation of electronic door locks in a building might be disabled by an attack that saturates the computer controlling the electronic locks with invalid requests.

Mobile code: Mobile code raises new and interesting security problems for any process that receives and executes program code from elsewhere, such as the email attachment mentioned in Section 1.5.3. Such code may easily play a Trojan horse role, purporting to fulfil an innocent purpose but in fact including code that accesses or modifies resources that are legitimately available to the host process but not to the originator of the code. The methods by which such attacks might be carried out are many and varied, and the host environment must be very carefully constructed in order to avoid them. Many of these issues have been addressed in Java and other mobile code systems, but the recent history of this topic has included the exposure of

some embarrassing weaknesses. This illustrates well the need for rigorous analysis in the design of all secure systems.

The uses of security models • It might be thought that the achievement of security in distributed systems would be a straightforward matter involving the control of access to objects according to predefined access rights and the use of secure channels for communication. Unfortunately, this is not generally the case. The use of security techniques such as encryption and access control incurs substantial processing and management costs. The security model outlined above provides the basis for the analysis and design of secure systems in which these costs are kept to a minimum, but threats to a distributed system arise at many points, and a careful analysis of the threats that might arise from all possible sources in the system's network environment, physical environment and human environment is needed. This analysis involves the construction of a *threat model* listing all the forms of attack to which the system is exposed and an evaluation of the risks and consequences of each. The effectiveness and the cost of the security techniques needed can then be balanced against the threats.

2.5 Summary

As illustrated in Section 2.2, distributed systems are increasingly complex in terms of their underlying physical characteristics; for example, in terms of the scale of systems, the level of heterogeneity inherent in such systems and the real demands to provide end-to-end solutions in terms of properties such as security. This places increasing importance on being able to understand and reason about distributed systems in terms of models. This chapter followed up consideration of the underlying physical models with an in-depth examination of the architectural and fundamental models that underpin distributed systems.

This chapter has presented an approach to describing distributed systems in terms of an encompassing architectural model that makes sense of this design space examining the core issues of what is communicating and how these entities communicate, supplemented by consideration of the roles each element may play together with the appropriate placement strategies given the physical distributed infrastructure. The chapter also introduced the key role of architectural patterns in enabling more complex designs to be constructed from the underlying core elements, such as the client-server model highlighted above, and highlighted major styles of supportive middleware solutions, including solutions based on distributed objects, components, web services and distributed events.

In terms of architectural models, the client-server approach is prevalent – the Web and other Internet services such as FTP, news and mail as well as web services and the DNS are based on this model, as are filing and other local services. Services such as the DNS that have large numbers of users and manage a great deal of information are based on multiple servers and use data partition and replication to enhance availability and fault tolerance. Caching by clients and proxy servers is widely used to enhance the performance of a service. However, there is now a wide variety of approaches to modelling distributed systems including alternative philosophies such as peer-to-peer

computing and support for more problem-oriented abstractions such as objects, components or services.

The architectural model is complemented by fundamental models, which aid in reasoning about properties of the distributed system in terms of, for example, performance, reliability and security. In particular, we presented models of interaction, failure and security. They identify the common characteristics of the basic components from which distributed systems are constructed. The interaction model is concerned with the performance of processes and communication channels and the absence of a global clock. It identifies a synchronous system as one in which known bounds may be placed on process execution time, message delivery time and clock drift. It identifies an asynchronous system as one in which no bounds may be placed on process execution time, message delivery time and clock drift – which is a description of the behaviour of the Internet.

The failure model classifies the failures of processes and basic communication channels in a distributed system. Masking is a technique by which a more reliable service is built from a less reliable one by masking some of the failures it exhibits. In particular, a reliable communication service can be built from a basic communication channel by masking its failures. For example, its omission failures may be masked by retransmitting lost messages. Integrity is a property of reliable communication – it requires that a message received be identical to one that was sent and that no message be sent twice. Validity is another property – it requires that any message put in the outgoing buffer be delivered eventually to the incoming message buffer.

The security model identifies the possible threats to processes and communication channels in an open distributed system. Some of those threats relate to integrity: malicious users may tamper with messages or replay them. Others threaten their privacy. Another security issue is the authentication of the principal (user or server) on whose behalf a message was sent. Secure channels use cryptographic techniques to ensure the integrity and privacy of messages and to authenticate pairs of communicating principals.

EXERCISES

2.1 What is the main disadvantage of distributed systems which exploit the infrastructure offered by the Internet? How can this be overcome? *page 55*

2.2 What problems do you foresee in the direct coupling between communicating entities that is implicit in remote invocation approaches? Consequently, what advantages do you anticipate from a level of decoupling as offered by space and time uncoupling? Note: you might want to revisit this answer after reading Chapters 5 and 6. *page 59*

2.3 What is the range of techniques covered by remote invocation? Briefly explain each technique. *page 60*

2.4 How are entities, such as objects or services, mapped on to the underlying physical distributed infrastructure? *page 64*

2.5 A search engine is a web server that responds to client requests to search in its stored indexes and (concurrently) runs several web crawler tasks to build and update the indexes. What are the requirements for synchronization between these concurrent activities? *page 62*

2.6 The host computers used in peer-to-peer systems are often simply desktop computers in users' offices or homes. What are the implications of this for the availability and security of any shared data objects that they hold and to what extent can any weaknesses be overcome through the use of replication? *pages 63, 64*

2.7 How is caching useful in placement strategies? What are its disadvantages? *page 65*

2.8 What is a mobile agent? How can it be a potential security threat? *page 67*

2.9 Consider a hypothetical car hire company and sketch out a three-tier solution to the provision of their underlying distributed car hire service. Use this to illustrate the benefits and drawbacks of a three-tier solution considering issues such as performance, scalability, dealing with failure and also maintaining the software over time. *page 68*

2.10 Provide a concrete example of the dilemma offered by Saltzer's end-to-end argument in the context of the provision of middleware support for distributed applications (you may want to focus on one aspect of providing dependable distributed systems, for example related to fault tolerance or security). *page 76*

2.11 Consider a simple server that carries out client requests without accessing other servers. Explain why it is generally not possible to set a limit on the time taken by such a server to respond to a client request. What would need to be done to make the server able to execute requests within a bounded time? Is this a practical option? *page 78*

2.12 For each of the factors that contribute to the time taken to transmit a message between two processes over a communication channel, state what measures would be needed to set a bound on its contribution to the total time. Why are these measures not provided in current general-purpose distributed systems? *page 79*

2.13 What are the two variants of the interaction model in distributed systems? On what points do they differ? *page 80*

2.14 Consider two communication services for use in asynchronous distributed systems. In service A, messages may be lost, duplicated or delayed and checksums apply only to headers. In service B, messages may be lost, delayed or delivered too fast for the recipient to handle them, but those that are delivered arrive with the correct contents.

Describe the classes of failure exhibited by each service. Classify their failures according to their effects on the properties of validity and integrity. Can service B be described as a reliable communication service? *page 83*, *page 87*

2.15 Consider a pair of processes X and Y that use the communication service B from Exercise 2.14 to communicate with one another. Suppose that X is a client and Y a server and that an *invocation* consists of a request message from X to Y, followed by Y carrying out the request, followed by a reply message from Y to X. Describe the classes of failure that may be exhibited by an invocation. *page 83*

2.16 Suppose that a basic disk read can sometimes read values that are different from those written. State the type of failure exhibited by a basic disk read. Suggest how this failure may be masked in order to produce a different benign form of failure. Now suggest how to mask the benign failure. *page 86*

2.17 How can the security of a distributed system be achieved? How can processes and their interactions be secured? *pages 88, 89*

2.18 Cryptography is the science of keeping messages secure. Explain, with an example, how it can be used in authentication to maintain confidentiality. *pages 90, 91*

This page intentionally left blank

3

NETWORKING AND INTERNETWORKING

Distributed systems use local area networks, wide area networks and internetworks for communication. The performance, reliability, scalability, mobility and quality of service characteristics of the underlying networks impact the behaviour of distributed systems and hence affect their design. Changes in user requirements have resulted in the emergence of wireless networks and of high-performance networks with quality of service guarantees.

The principles on which computer networks are based include protocol layering, packet switching, routing and data streaming. Internetworking techniques enable heterogeneous networks to be integrated. The Internet is the major example; its protocols are almost universally used in distributed systems. The addressing and routing schemes used in the Internet have withstood the impact of its enormous growth. They are now undergoing revision to accommodate future growth and to meet new application requirements for mobility, security and quality of service.

The design of specific network technologies is illustrated in three case studies: Ethernet, IEEE 802.11 (WiFi) and Bluetooth wireless networking.

3.1 Introduction

The networks used in distributed systems are built from a variety of *transmission media*, including wire, cable, fibre and wireless channels; hardware devices, including routers, switches, bridges, hubs, repeaters and network interfaces; and software components, including protocol stacks, communication handlers and drivers. The resulting functionality and performance available to distributed system and application programs is affected by all of these. We shall refer to the collection of hardware and software components that provide the communication facilities for a distributed system as a *communication subsystem*. The computers and other devices that use the network for communication purposes are referred to as *hosts*. The term *node* is used to refer to any computer or switching device attached to a network.

The Internet is a single communication subsystem providing communication between all of the hosts that are connected to it. The Internet is constructed from many *subnets*. A subnet is a unit of routing (delivering data from one part of the Internet to another); it is a collection of nodes that can all be reached on the same physical network. The Internet's infrastructure includes an architecture and hardware and software components that effectively integrate diverse subnets into a single data communication service.

The design of a communication subsystem is strongly influenced by the characteristics of the operating systems used in the computers of which the distributed system is composed as well as the networks that interconnect them. In this chapter, we consider the impact of network technologies on the communication subsystem; operating system issues are discussed in Chapter 7.

This chapter is intended to provide an introductory overview of computer networking with reference to the communication requirements of distributed systems. Readers who are not familiar with computer networking should regard it as an underpinning for the remainder of the book, while those who are will find that this chapter offers an extended summary of those aspects of computer networking that are particularly relevant for distributed systems.

Computer networking was conceived soon after the invention of computers. The theoretical basis for packet switching was introduced in a paper by Leonard Kleinrock [1961]. In 1962, J.C.R. Licklider and W. Clark, who participated in the development of the first timesharing system at MIT in the early 1960s, published a paper discussing the potential for interactive computing and wide area networking that presaged the Internet in several respects [DEC 1990]. In 1964, Paul Baran produced an outline of a practical design for reliable and effective wide area networks [Baran 1964]. Further material and links on the history of computer networking and the Internet can be found in the following sources: [www.isoc.org, Comer 2007, Kurose and Ross 2007].

In the remainder of this section we discuss the communication requirements of distributed systems. We give an overview of network types in Section 3.2 and an introduction to networking principles in Section 3.3. Section 3.4 deals specifically with the Internet. The chapter concludes with detailed case studies on the Ethernet, IEEE 802.11 (WiFi) and Bluetooth networking technologies in Section 3.5.

3.1.1 Networking issues for distributed systems

Early computer networks were designed to meet a few, relatively simple application requirements. Network applications such as file transfer, remote login, electronic mail and newsgroups were supported. The subsequent development of distributed systems with support for distributed application programs accessing shared files and other resources set a higher standard of performance to meet the needs of interactive applications.

More recently, following the growth and commercialization of the Internet and the emergence of many new modes of use, more stringent requirements for reliability, scalability, mobility, security and quality of service have emerged. In this section, we define and describe the nature of each of these requirements.

Performance • The network performance parameters that are of primary interest for our purposes are those affecting the speed with which individual messages can be transferred between two interconnected computers. These are the latency and the point-to-point data transfer rate:

Latency is the delay that occurs after a send operation is executed and before data starts to arrive at the destination computer. It can be measured as the time required to transfer an empty message. Here we are considering only network latency, which forms a part of the process-to-process latency defined in Section 2.4.1.

Data transfer rate is the speed at which data can be transferred between two computers in the network once transmission has begun, usually quoted in bits per second.

Following from these definitions, the time required for a network to transfer a message containing *length* bits between two computers is:

$$\textit{Message transmission time} = \textit{latency} + \textit{length} / \textit{data transfer rate}$$

The above equation is valid for messages whose length does not exceed a maximum that is determined by the underlying network technology. Longer messages have to be segmented and the transmission time is the sum of the times for the segments.

The transfer rate of a network is determined primarily by its physical characteristics, whereas the latency is determined primarily by software overheads, routing delays and a load-dependent statistical element arising from conflicting demands for access to transmission channels. Many of the messages transferred between processes in distributed systems are small in size; latency is therefore often of equal or greater significance than transfer rate in determining performance.

The *total system bandwidth* of a network is a measure of throughput – the total volume of traffic that can be transferred across the network in a given time. In many local area network technologies, such as Ethernet, the full transmission capacity of the network is used for every transmission and the system bandwidth is the same as the data transfer rate. But in most wide area networks messages can be transferred on several different channels simultaneously, and the total system bandwidth bears no direct relationship to the transfer rate. The performance of networks deteriorates in conditions of overload – when there are too many messages in the network at the same time. The precise effect of overload on the latency, data transfer rate and total system bandwidth of a network depends strongly on the network technology.

Now consider the performance of client-server communication. The time required to transmit a short request message and receive a short reply between nodes on a lightly loaded local network (including system overheads) is about half a millisecond. This should be compared with the sub-microsecond time required to invoke an operation on an application-level object in the local memory. Thus, despite advances in network performance, the time required to access shared resources on a local network remains about a thousand times greater than that required to access resources that are resident in local memory. But networks often outperform hard disks; networked access to a local web server or file server with a large in-memory cache of frequently used files can match or outstrip access to files stored on a local hard disk.

On the Internet, round-trip latencies are in the 5–500 ms range, with means of 20–200 ms depending on distance [www.globalcrossing.net], so requests transmitted across the Internet are 10–100 times slower than those sent on fast local networks. The bulk of this time difference derives from switching delays at routers and contention for network circuits.

Section 7.5.1 discusses and compares the performance of local and remote operations in greater detail.

Scalability • Computer networks are an indispensable part of the infrastructure of modern societies. In Figure 1.6 we showed the growth in the number of host computers and web servers connected to the Internet over a 12-year period ending in 2005. The growth since then has been so rapid and diverse that it is difficult to find recent reliable statistics. The potential future size of the Internet is commensurate with the population of the planet. It is realistic to expect it to include several billion nodes and hundreds of millions of active hosts.

These numbers indicate the future changes in size and load that the Internet must handle. The network technologies on which it is based were not designed to cope with even the Internet's current scale, but they have performed remarkably well. Some substantial changes to the addressing and routing mechanisms are in progress in order to handle the next phase of the Internet's growth; these will be described in Section 3.4. For simple client-server applications such as the Web, we would expect future traffic to grow at least in proportion to the number of active users. The ability of the Internet's infrastructure to cope with this growth will depend upon the economics of use, in particular charges to users and the patterns of communication that actually occur – for example, their degree of locality.

Reliability • Our discussion of failure models in Section 2.4.2 describes the impact of communication errors. Many applications are able to recover from communication failures and hence do not require guaranteed error-free communication. The end-to-end argument (Section 2.3.3) further supports the view that the communication subsystem need not provide totally error-free communication; the detection of communication errors and their correction is often best performed by application-level software. The reliability of most physical transmission media is very high. When errors occur they are usually due to failures in the software at the sender or receiver (for example, failure by the receiving computer to accept a packet) or buffer overflow rather than errors in the network.

Security • Chapter 11 sets out the requirements and techniques for achieving security in distributed systems. The first level of defence adopted by most organizations is to protect its networks and the computers attached to them with a *firewall*. A firewall creates a protection boundary between the organization's intranet and the rest of the Internet. The purpose of the firewall is to protect the resources in all of the computers inside the organization from access by external users or processes and to control the use of resources outside the firewall by users inside the organization.

A firewall runs on a gateway – a computer that stands at the network entry point to an organization's intranet. The firewall receives and filters all of the messages travelling into and out of an organization. It is configured according to the organization's security policy to allow certain incoming and outgoing messages to pass through it and to reject all others. We shall return to this topic in Section 3.4.8.

To enable distributed applications to move beyond the restrictions imposed by firewalls there is a need to produce a secure network environment in which a wide range of distributed applications can be deployed, with end-to-end authentication, privacy and security. This finer-grained and more flexible form of security can be achieved through the use of cryptographic techniques. It is usually applied at a level above the communication subsystem and hence is not dealt with here but in Chapter 11. Exceptions include the need to protect network components such as routers against unauthorized interference with their operation and the need for secure links to mobile devices and other external nodes to enable them to participate in a secure intranet – the *virtual private network* (VPN) concept, discussed in Section 3.4.8.

Mobility • Mobile devices such as laptop computers and Internet-capable mobile phones are moved frequently between locations and reconnected at convenient network connection points or even used while on the move. Wireless networks provide connectivity to such devices, but the addressing and routing schemes of the Internet were developed before the advent of these mobile devices and are not well adapted to their need for intermittent connection to many different subnets. The Internet's mechanisms have been adapted and extended to support mobility, but the expected future growth in the use of mobile devices will demand further development.

Quality of service • In Chapter 1, we defined quality of service as including the ability to meet deadlines when transmitting and processing streams of real-time multimedia data. This imposes major new requirements on computer networks. Applications that transmit multimedia data require guaranteed bandwidth and bounded latencies for the communication channels that they use. Some applications vary their demands dynamically and specify both a minimum acceptable quality of service and a desired optimum. The provision of such guarantees and their maintenance is the subject of Chapter 20.

Multicasting • Most communication in distributed systems is between pairs of processes, but there often is also a need for one-to-many communication. While this can be simulated by *sends* to several destinations, that is more costly than necessary and may not exhibit the fault-tolerance characteristics required by applications. For these reasons, many network technologies support the simultaneous transmission of messages to several recipients.

3.2 Types of network

Here we introduce the main types of network that are used to support distributed systems: *personal area networks*, *local area networks*, *wide area networks*, *metropolitan area networks* and the wireless variants of them. *Internetworks* such as the Internet are constructed from networks of all these types. Figure 3.1 shows the performance characteristics of the various types of network discussed below.

Some of the names used to refer to types of networks are confusing because they seem to refer to the physical extent (local area, wide area), but they also identify physical transmission technologies and low-level protocols. These are different for local and wide area networks, although some network technologies, such as ATM (Asynchronous Transfer Mode), are suitable for both local and wide area applications and some wireless networks also support local and metropolitan area transmission.

We refer to networks that are composed of many interconnected networks, integrated to provide a single data communication medium, as internetworks. The Internet is the prototypical internetwork; it is composed of millions of local, metropolitan and wide area networks. We describe its implementation in some detail in Section 3.4.

Personal area networks (PANs) • PANs are a subcategory of local networks in which the various digital devices carried by a user are connected by a low-cost, low-energy network. Wired PANs are not of much significance because few users wish to be encumbered by a network of wires on their person, but wireless personal area networks (WPANs) are of increasing importance due to the number of personal devices such as mobile phones, tablets, digital cameras, music players and so on that are now carried by many people. We describe the Bluetooth WPAN in Section 3.5.3.

Local area networks (LANs) • LANs carry messages at relatively high speeds between computers connected by a single communication medium, such as twisted copper wire,

Figure 3.1 Network performance

	Example	*Range*	*Bandwidth (Mbps)*	*Latency (ms)*
Wired:				
LAN	Ethernet	1–2 kms	10–10,000	1–10
WAN	IP routing	worldwide	0.010–600	100–500
MAN	ATM	2–50 kms	1–600	10
Internetwork	Internet	worldwide	0.5–600	100–500
Wireless:				
WPAN	Bluetooth (IEEE 802.15.1)	10–30m	0.5–2	5–20
WLAN	WiFi (IEEE 802.11)	0.15–1.5 km	11–108	5–20
WMAN	WiMAX (IEEE 802.16)	5–50 km	1.5–20	5–20
WWAN	3G phone	cell: 1–-5 km	348–14.4	100–500

coaxial cable or optical fibre. A *segment* is a section of cable that serves a department or a floor of a building and may have many computers attached. No routing of messages is required within a segment, since the medium provides direct connections between all of the computers connected to it. The total system bandwidth is shared between the computers connected to a segment. Larger local networks, such as those that serve a campus or an office building, are composed of many segments interconnected by switches or hubs (see Section 3.3.7). In local area networks, the total system bandwidth is high and latency is low, except when message traffic is very high.

Several local area technologies were developed in the 1970s including Ethernet, token rings and slotted rings. Each provides an effective and high-performance solution, but Ethernet emerged as the dominant technology for wired local area networks. It was originally produced in the early 1970s with a bandwidth of 10 Mbps (million bits per second) and extended to 100 Mbps, 1000 Mbps (1 gigabit per second) and 10 Gbps versions more recently. We describe the principles of operation of Ethernet networks in Section 3.5.1.

There is a very large installed base of local area networks, serving virtually all working environments that contain more than one or two personal computers or workstations. Their performance is generally adequate for the implementation of distributed systems and applications. Ethernet technology lacks the latency and bandwidth guarantees needed by many multimedia applications. ATM networks were developed to fill this gap, but their cost has inhibited their adoption in local area applications. Instead, high-speed Ethernets have been deployed in a switched mode that overcomes these drawbacks to a significant degree, though not as effectively as ATM.

Wide area networks (WANs) • WANs carry messages at lower speeds between nodes that are often in different organizations and may be separated by large distances. They may be located in different cities, countries or continents. The communication medium is a set of communication circuits linking a set of dedicated computers called *routers.* They manage the communication network and route messages or packets to their destinations. In most networks, the routing operations introduce a delay at each point in the route, so the total latency for the transmission of a message depends on the route that it follows and the traffic loads in the various network segments that it traverses. In current networks these latencies can be as high as 0.1 to 0.5 seconds. The speed of electronic signals in most media is close to the speed of light, and this sets a lower bound on the transmission latency for long-distance networks. For example, the propagation delay for a signal to travel from Europe to Australia via a terrestrial link is approximately 0.13 seconds and signals via a geostationary satellite between any two points on the Earth's surface are subject to a delay of approximately 0.20 seconds.

Bandwidths available across the Internet also vary widely. Speeds of up to 600 Mbps are commonly available, but speeds of 1–10 Mbps are more typically experienced for bulk transfers of data.

Metropolitan area networks (MANs) • This type of network is based on the high-bandwidth copper and fibre optic cabling recently installed in some towns and cities for the transmission of video, voice and other data over distances of up to 50 kilometres. A variety of technologies have been used to implement the routing of data in MANs, ranging from Ethernet to ATM.

The DSL (Digital Subscriber Line) and cable modem connections now available in many countries are an example. DSL typically uses ATM switches located in telephone exchanges to route digital data onto twisted pairs of copper wire (using high-frequency signalling on the existing wiring used for telephone connections) to the subscriber's home or office at speeds in the range 1–10 Mbps. The use of twisted copper wire for DSL subscriber connections limits the range to about 5.5 km from the switch. Cable modem connections use analogue signalling on cable television networks to achieve speeds of up to 15 Mbps over coaxial cable with greater range than DSL.

The term DSL actually represents a family of technologies, sometimes referred to as xDSL and including for example ADSL (or Asymmetric Digital Subscriber Line). Latest developments include VDSL and VDSL2 (Very High Bit Rate DSL), which are capable of speeds of up to 100 Mbps and designed to support a range of multimedia traffic including High Definition TV (HDTV).

Wireless local area networks (WLANs) • WLANs are designed for use in place of wired LANs to provide connectivity for mobile devices, or simply to remove the need for a wired infrastructure to connect computers within homes and office buildings to each other and the Internet. They are in widespread use in several variants of the IEEE 802.11 standard (WiFi), offering bandwidths of 10–100 Mbps over ranges up to 1.5 kilometres. Section 3.5.2 gives further information on their method of operation.

Wireless metropolitan area networks (WMANs) • The IEEE 802.16 WiMAX standard is targeted at this class of network. It aims to provide an alternative to wired connections to home and office buildings and to supersede 802.11 WiFi networks in some applications.

Wireless wide area networks (WWANs) • Most mobile phone networks are based on digital wireless network technologies such as the GSM (Global System for Mobile communication) standard, which is used in most countries of the world. Mobile phone networks are designed to operate over wide areas (typically entire countries or continents) through the use of cellular radio connections; their data transmission facilities therefore offer wide area mobile connections to the Internet for portable devices. The cellular networks mentioned above offer relatively low data rates – 9.6 to 33 kbps – but the 'third generation' (3G) of mobile phone networks is now available, with data transmission rates in the range of 2–14.4 Mbps while stationary and 348 kbps while moving (for example in a car). The underlying technology is referred to as UMTS (Universal Mobile Telecommunications System). A path has also been defined to evolve UMTS towards 4G data rates of up to 100 Mbps. Readers interested in digging more deeply than we are able to here into the rapidly evolving technologies of mobile and wireless networks of all types are referred to Stojmenovic's excellent handbook [2002].

Internetworks • An internetwork is a communication subsystem in which several networks are linked together to provide common data communication facilities that overlay the technologies and protocols of the individual component networks and the methods used for their interconnection.

Internetworks are needed for the development of extensible, open distributed systems. The openness characteristic of distributed systems implies that the networks used in distributed systems should be extensible to very large numbers of computers, whereas individual networks have restricted address spaces and some have performance

limitations that are incompatible with their large-scale use. In internetworks, a variety of local and wide area network technologies can be integrated to provide the networking capacity needed by each group of users. Thus internetworks bring many of the benefits of open systems to the provision of communication in distributed systems.

Internetworks are constructed from a variety of component networks. They are interconnected by dedicated switching computers called *routers* and general-purpose computers called *gateways*, and an integrated communication subsystem is produced by a software layer that supports the addressing and transmission of data to computers throughout the internetwork. The result can be thought of as a 'virtual network' constructed by overlaying an internetwork layer on a communication medium that consists of the underlying networks, routers and gateways. The Internet is the major instance of internetworking, and its TCP/IP protocols are an example of this integration layer.

Network errors • An additional point of comparison not mentioned in Figure 3.1 is the frequency and types of failure that can be expected in the different types of network. The reliability of the underlying data transmission media is very high in all types except wireless networks, where packets are frequently lost due to external interference. But packets may be lost in all types of network due to processing delays and buffer overflow at switches and at the destination node. This is by far the most common cause of packet loss.

Packets may also be delivered in an order different from that in which they were transmitted. This arises only in networks where separate packets are individually routed – principally wide area networks. Finally, duplicate copies of packets can be delivered. This is usually a consequence of an assumption by the sender that a packet has been lost; the packet is retransmitted, and both the original and the retransmitted copy then turn up at the destination.

3.3 Network principles

The basis for all computer networks is the packet-switching technique first developed in the 1960s. This enables data packets addressed to different destinations to share a single communications link, unlike the circuit-switching technology that underlies conventional telephony. Packets are queued in a buffer and transmitted when the link is available. Communication is asynchronous – messages arrive at their destination after a delay that varies depending upon the time that packets take to travel through the network.

3.3.1 Packet transmission

In most applications of computer networks the requirement is for the transmission of logical units of information, or *messages* – sequences of data items of arbitrary length. But before a message is transmitted it is subdivided into *packets*. The simplest form of packet is a sequence of binary data (an array of bits or bytes) of restricted length,

together with addressing information sufficient to identify the source and destination computers. Packets of restricted length are used:

- so that each computer in the network can allocate sufficient buffer storage to hold the largest possible incoming packet;
- to avoid the undue delays that would occur in waiting for communication channels to become free if long messages were transmitted without subdivision.

3.3.2 Data streaming

The transmission and display of audio and video in real time is referred to as *streaming*. It requires much higher bandwidths than most other forms of communication in distributed systems. We have already noted in Chapter 2 that multimedia applications rely upon the transmission of streams of audio and video data elements at guaranteed high rates and with bounded latencies.

A video stream requires a bandwidth of about 1.5 Mbps if the data is compressed, or 120 Mbps if uncompressed. UDP internet packets are generally used to hold the video frames, but because the flow is continuous as opposed to the intermittent traffic generated by typical client-server interactions, the packets are handled somewhat differently. The *play time* of a multimedia element such as a video frame is the time at which it must be displayed (for a video element) or converted to sound (for a sound sample). For example, in a stream of video frames with a frame rate of 24 frames per second, frame N has a play time that is $N/24$ seconds after the stream's start time. Elements that arrive at their destination later than their play time are no longer useful and will be dropped by the receiving process.

The timely delivery of audio and video streams depends upon the availability of connections with adequate quality of service – bandwidth, latency and reliability must all be considered. Ideally, adequate quality of service should be guaranteed. In general the Internet does not offer that capability, and the quality of real-time video streams sometimes reflects that, but in proprietary intranets such as those operated by media companies, guarantees are sometimes achieved. What is required is the ability to establish a channel from the source to the destination of a multimedia stream, with a predefined route through the network, a reserved set of resources at each node through which it will travel and buffering where appropriate to smooth any irregularities in the flow of data through the channel. Data can then be passed through the channel from sender to receiver at the required rate.

ATM networks are specifically designed to provide high bandwidth and low latencies and to support quality of service by the reservation of network resources. IPv6, the new network protocol for the Internet outlined in Section 3.4.4, includes features that enable each of the IP packets in a real-time stream to be identified and treated separately from other data at the network level.

Communication subsystems that provide quality of service guarantees require facilities for the preallocation of network resources and the enforcement of the allocations. The Resource Reservation Protocol (RSVP) [Zhang *et al.* 1993] enables applications to negotiate the preallocation of bandwidth for real-time data streams. The Real Time Transport Protocol (RTP) [Schulzrinne *et al*. 1996] is an application-level data transfer protocol that includes details of the play time and other timing