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 characteristics of distributed systems: concurrency of components, lack of a global clock and independent failures of components.

We give three examples of distributed systems:

- the Internet;
- an intranet, which is a portion of the Internet managed by an organization;
- mobile and ubiquitous computing.

The sharing of resources is 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 Web is discussed as an example of resource sharing and its main features are introduced.

The challenges arising from the construction of distributed systems are the heterogeneity of its components, openness, which allows components to be added or replaced, security, scalability – the ability to work well when the number of users increases – failure handling, concurrency of components and transparency.

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 the same room. Our definition of distributed systems has the following significant consequences:

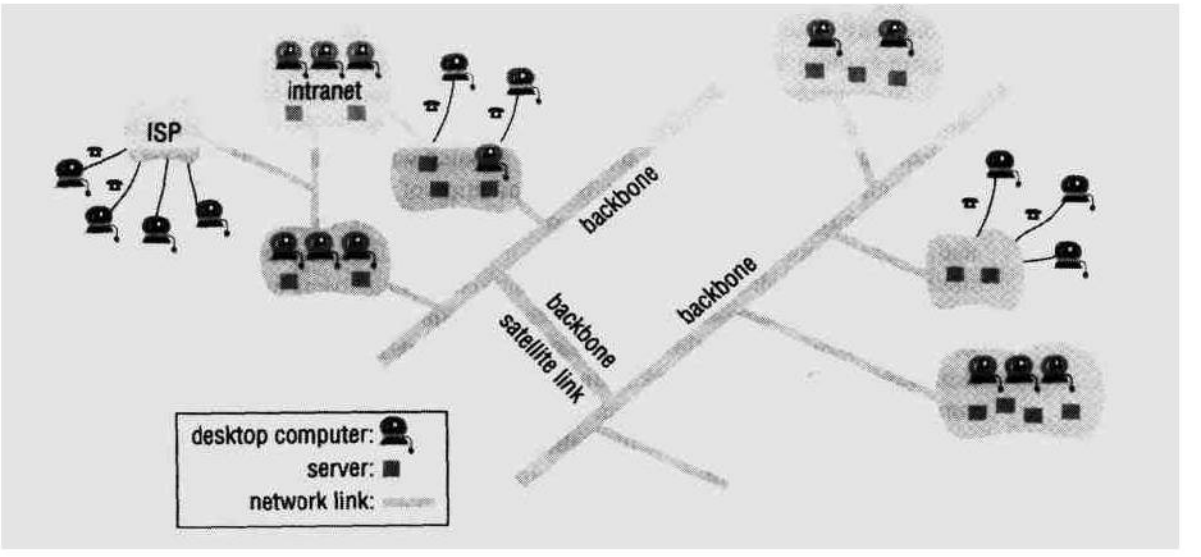
Concurrency: In a network of computers, concurrent program execution is the norm. I can do my work on my computer while you do your work on yours, sharing resources such as web pages or files when necessary. The capacity of the system to handle shared resources can be increased by adding more resources (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 10.

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 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

Figure 1.1 A typical portion of the Internet



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 key examples of distributed systems, the components from which they are constructed and their purposes. Section 1.3 explores the design of resource-sharing systems in the context of the World Wide Web. Section 1.4 describes the key challenges faced by the designers of distributed systems: heterogeneity, openness, security, scalability, failure handling, concurrency and the need for transparency.

1.2 Examples of distributed systems

Our examples are based on familiar and widely used computer networks: the Internet, intranets and the emerging technology of networks based on mobile devices. They are designed to exemplify the wide range of services and applications that are supported by computer networks and to begin the discussion of the technical issues that underlie their implementation.

1.2.1 The Internet

The Internet is a vast interconnected collection of computer networks of many different types. Figure 1.1 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.

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. Internet Service Providers (ISPs) are companies that provide modem 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.

Multimedia services are available in the Internet, enabling users to access audio and video data including music, radio and TV channels and to hold phone and video conferences. The capacity of the Internet to handle the special communication requirements of multimedia data is currently quite limited because it does not provide the necessary facilities to reserve network capacity for individual streams of data. Chapter 15 discusses the needs of distributed multimedia systems.

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 1.4). We shall highlight those solutions throughout the book, pointing out their scope and their limitations where appropriate.

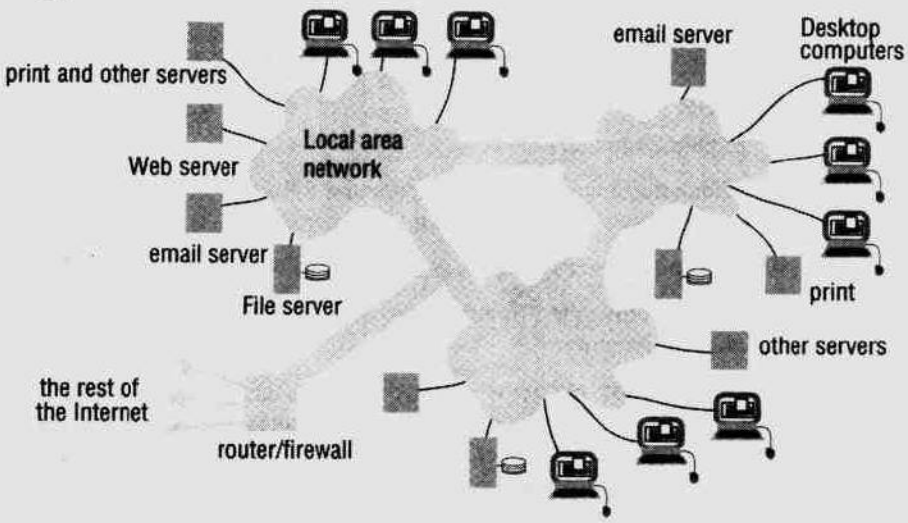
1.2.2 Intranets

An intranet is a portion of the Internet that is separately administered and has a boundary that can be configured to enforce local security policies. Figure 1.2 shows a typical intranet. It is composed of several local area networks (LANs) linked by backbone connections. The network configuration of a particular intranet is the responsibility of the organization that administers it and may vary widely – ranging from a LAN on a single site to a connected set of LANs belonging to branches of a company or other organization in different countries.

An intranet is connected to the Internet via a router, which allows the users inside the intranet to make use of services elsewhere such as the Web or email. It also allows the users in other intranets to access the services it provides. Many organizations need to protect their own services from unauthorized use by possibly malicious users elsewhere. For example, a company will not want secure information to be accessible to users in competing organizations, and a hospital will not want sensitive patient data to be revealed. Companies also want to protect themselves from harmful programs such as viruses entering and attacking the computers in the intranet and possibly destroying valuable data.

The role of a *firewall* is to protect an intranet by preventing unauthorized messages leaving or entering. A firewall is implemented by filtering incoming and outgoing messages, for example according to their source or destination. A firewall

Figure 1.2 A typical intranet



might for example allow only those messages related to email and web access to pass into or out of the intranet that it protects.

Some organizations do 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 networks that are isolated from the outside world, and the UK National Health Service has chosen to take the view that sensitive patient-related medical data can only be adequately protected by maintaining a physically separate internal network. Some military organizations disconnect their internal networks from the Internet at times of war. But even those organizations will wish to benefit from the huge range of application and system software that employs Internet communication protocols. The solution that is usually adopted by such organizations is to operate an intranet as described above, but without the connections to the Internet. Such an intranet can dispense with the firewall; or, to put it another way, it has the most effective firewall possible – the absence of any physical connections to the Internet.

The main issues arising in the design of components for use in intranets are:

- File services are needed to enable users to share data; the design of these is discussed in Chapter 8.
- Firewalls tend to impede legitimate access to services – when resource sharing between internal and external users is required, firewalls must be complemented by the use of fine-grained security mechanisms; these are discussed in Chapter 7.
- The cost of software installation and support is an important issue. These costs can be reduced by the use of system architectures such as network computers and thin clients, described in Chapter 2.

1.2.3 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 personal digital assistants (PDAs), mobile phones, pagers, 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 (also called *nomadic computing* [Kleinrock 1997, www.cooltown.hp.com]) 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 that are conveniently nearby as they move around. The latter is also known as *location-aware computing*.

Ubiquitous computing [Weiser 1993] is the harnessing of many small, cheap computational devices that are present in users' physical environments, including the home, office and elsewhere. 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.

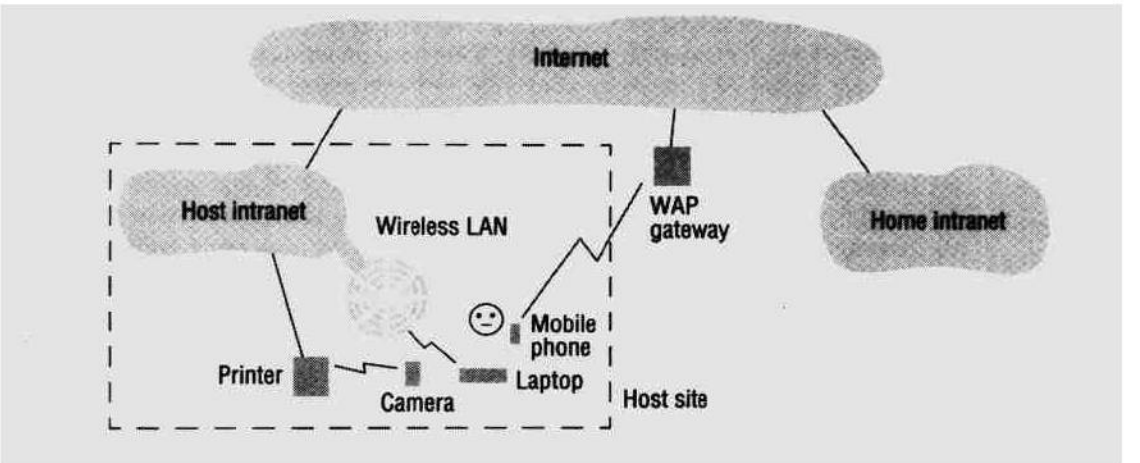
The presence of computers everywhere only becomes useful when they can communicate with one another. For example, it would be convenient for users to control their washing machine and their hi-fi system from a 'universal remote control' device in the home. Equally, the washing machine could page the user via a smart badge or watch 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 1.3 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 few hundreds of metres (a floor of a building, say). It connects to the rest of the host intranet via a gateway. The user also has a mobile (cellular) telephone, which is connected to the Internet using the Wireless Application Protocol (WAP) via a gateway

Figure 1.3 Portable and handheld devices in a distributed system



(see Chapter 3). The phone gives access to pages of simple, textual information, which it presents on its small display. Finally, the user carries a digital camera, which can communicate over an infra-red link when pointed at a corresponding device such as a printer.

With a suitable system infrastructure, the user can perform some simple tasks in the host site using the devices that they carry. While journeying to the host site, the user can fetch the latest stock prices from a web server using the mobile phone. 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 printer in the meeting room. This requires only the infra-red link between the camera and printer. 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.

Mobile and ubiquitous computing raise significant system issues [Milojicic *et al.* 1999, p. 266, Weiser 1993]. Section 2.2.3 presents an architecture for mobile computing and outlines the issues that arise from it, including how to support the discovery of resources in a host environment; eliminating the need for users to reconfigure their mobile devices as they move around; helping users to cope with limited connectivity as they travel; and providing privacy and other security guarantees to users and the environments that they visit.

1.3 Resource sharing and the Web

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 their

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 that those are implemented on. 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 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 a *clients*. 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. In so far as they are distinct, clients are active and servers are passive; servers run continuously, whereas clients last only as long as the applications of which they form a part.

Note that by default the terms 'client' and 'server' refer to *processes* rather than the computers that they execute upon, although 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 now examine the Web in more detail.

1.3.1 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 browser software such as Netscape and Internet Explorer, users use the Web to retrieve and view documents of many types, to listen to audio streams and view video streams, and to 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* – references to other documents and resources that are also stored in the Web.

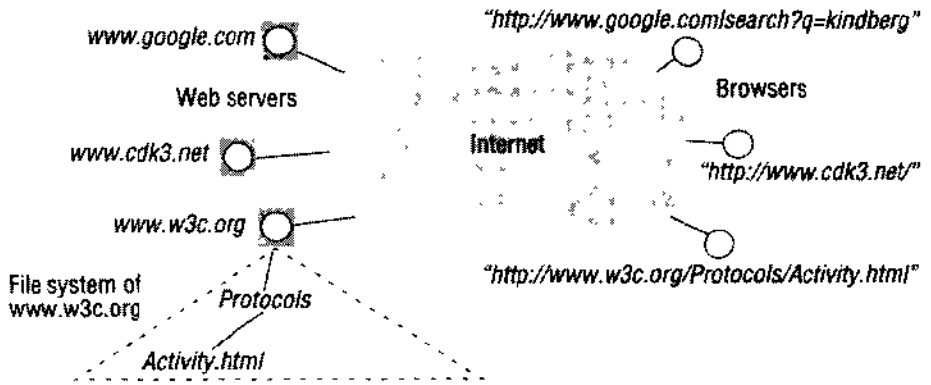
It is fundamental to the user's experience of the Web that when he or she encounters a given image or piece of text within a document, this will frequently be accompanied by links to related documents and other resources. The structure of links can be arbitrarily complex and the set of resources that can be added is unlimited – the 'web' of links is indeed world-wide. Bush [1945] conceived of hypertextual structures over fifty 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 1.4.2). First, its operation is based on communication standards and document 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 PDAs 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 stored in a file and presented to the user, such as program files, media files, and documents in PostScript or 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:

Figure 1.4 Web servers and web browsers



- The HyperText Markup Language (HTML) is a language for specifying the contents and layout of pages as they are displayed by web browsers.
- Uniform Resource Locators (URLs), which identify documents and other resources stored as part of the Web. Chapter 9 discusses other web identifiers.
- 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 1.4 shows some web servers, and browsers making requests of them. It is an important feature that users may locate and manage their own web servers anywhere on the Internet.

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 either produce HTML by hand, using a standard text editor, or they can 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.cdk3.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.cdk3.net/WebExample/moon.html">Moon</A>.         4
<P>                                                                    5
(etcetera)                                                             6
```

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.cdk3.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 PostScript. 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.cdk3.net/WebExample/Images/earth.jpg*. Lines 2 and 5 are each a directive to begin a new paragraph. Lines 3 and 6 each contain some text that is 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, 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.cdk3.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 in such a way as to enable the browser to locate that resource. Browsers examine URLs in order to fetch the corresponding resources from web servers. 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 an an image.

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

scheme : scheme-specific-location

The first component, the 'scheme', declares which type of URL this is. URLs are required to specify the locations of a variety of resources and also to specify a variety of communication protocols to retrieve them. 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 'nntp' (used to specify a Usenet news group), and 'telnet' (used to log in to a computer).

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 helper application or plug-in.

HTTP URLs are the most widely used, for fetching resources using the standard HTTP protocol. An HTTP URL has two main jobs to do: to identify which web server maintains the resource, and to identify which of the resources at that server is required. Figure 1.4 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 sub-trees (directories) of the server's file system, and each resource is identified by the file's path name relative to the server.

In general, HTTP URLs are of the following form:

http:// servername [:port] [/pathNameOnServer] [?arguments]

– 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 Service (DNS) name (see Section 9.2). The server's DNS name is optionally followed by the number of the 'port' on which the server listens for requests (see Chapter 4). 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 set of arguments – for example, when a user submits the entries in a form such as a search engine's query page.

Consider the URLs:

http://www.cdk3.net/
http://www.w3.org/Protocols/Activity.html
http://www.google.com/search?q=kindberg.

These can be broken down as follows:

Server DNS name	*Pathname on server*	*Arguments*
www.cdk3.net	(default)	(none)
www.w3.org	Protocols/Activity.html	(none)
www.google.com	search	q=kindberg

The first URL designates the default page supplied by *www.cdk3.net*. The next identifies a file on the server *www.w3.org* whose path name is *Protocols/Activity.html*. The third URL specifies a query to a search engine. The path identifies a program called 'search', and the string after the '?' character encodes the arguments to this program – in this case it specifies the query string. We discuss URLs that denote programs in more detail when we consider more advanced features below.

Readers may also have observed the presence of an *anchor* at the end of a URL – a name preceded by a '#' such as '#references', which denotes a point inside a document. Anchors are not part of URLs themselves but are defined as part of the HTML specification. Only browsers interpret anchors, to display web pages from the specified point. Browsers always retrieve entire web pages from servers, not parts of them denoted by anchors.

Publishing a resource: While the Web has a clear model for retrieving a resource from its URL, the method for publishing a resource on the Web remains unwieldy and normally requires human intervention. To publish a resource on the Web, a user must first 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 can then construct the URL as *http://S/P*. The user places this URL in a link from an existing document or distributes the URL to other users, for example by email.

There are certain pathname conventions that servers recognize. For example, a pathname beginning *~joe* is by convention in a subdirectory *public_html* of user joe's home directory. Similarly, a path name that ends in a directory name rather than a simple file conventionally refers to a file in that directory called *index.html*.

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 any other types of client interact with web servers. Chapter 4 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 only needs the part of the URL that follows the server's own DNS name.) The server looks up the pathname and, if it exists, sends back the file's contents in a reply message to the client. Otherwise, it sends back an error response.

Content types: Browsers are not necessarily capable of handling or making good use of 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 [Borenstein and Freed 1993]. 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: In HTTP version 1.0 (which is the most widely used version at the time of writing), the client requests 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.

More advanced features – services and dynamic pages ◊ So far we have described how users can publish web pages and other content stored in files on the Web. The content may change over time, but it is the same for everyone. However, much of users' experience of the Web is that of services with which the user can interact. For example, when purchasing an item at an on-line store, the user often fills out a *web form* to give their 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 reasonably short then it is usually sent as the final component of the URL, following a '?' character (otherwise it is sent as additional data in the request). For example, a request containing the following URL invokes a program called 'search' at www.google.com and specifies a query string of 'kindberg': *http://www.google.com/search?q=kindberg*.

The 'search' program produces HTML text as its output, and the user will see a listing of pages that contain the word 'kindberg'. (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 often 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. For example, code written in Javascript [www.netscape.com] is often downloaded with a web 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 entire new version of the page and re-rendering it.

Javascript has quite limited functionality. By contrast, an *applet* is an application that the browser automatically downloads and runs when it fetches a corresponding web page. Applets may access the network and provide customized user interfaces, using the facilities of the Java language [java.sun.com, Flanagan 1997]. 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 2.2.3.

Discussion of the Web ◊ The Web's phenomenal success rests upon the ease with which resources can be published, 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 resource and service 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, then 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 following confusingly many disparate links, referencing pages from a disparate collection of sources, of dubious reliability in some cases. Search engines are a useful complement to link-following as a means of finding information on the Web, but these are notoriously 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 standardize the format of metadata about web resources. Metadata records the attributes of web resources, and this is read by tools to assist users who process web resources *en masse*, such as when searching or compiling lists of related links.

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. More recently, the Extensible Markup Language (XML) [www.w3.org VI] has been designed as a way of representing data in standard, structured, application-specific forms. For example, XML is being used to describe the capabilities of devices and to describe personal information held about users. XML is a meta-language for describing data, which makes data portable between applications. The Extensible Stylesheet Language [www.w3.org VII] is used to declare how data in XML format should be presented to users. By using two different stylesheets, for example, the same information about an individual user could be represented on a web page graphically in one case and as a simple list in another.

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 is slow. Chapter 2 describes the use of caching in browsers and proxy servers to alleviate these effects. But the Web's client-server architecture means that it has no efficient means of keeping users up to date with the latest versions of pages. Users have to press their browser's 'reload' button to ensure that they have the latest information, and browsers are forced to communicate with servers to check whether a local copy of a resource is still valid.

Finally, a web page is not always a satisfactory user interface. The interface widgets defined for HTML are limited, and designers often include applets or many images in web pages to make them look and function more acceptably. There is a consequent increase in the download time.

1.4 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. Although distributed systems are to be found everywhere, their design is quite simple and there is still a lot of scope to develop more ambitious services and applications. Many of the challenges discussed in this section have already been met, but future designers need to be aware of them and to be careful to take them into account.

1.4.1 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 1.1), 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 NT.

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. CORBA, which is described in Chapters 4, 5 and 17, is an example. Some middleware, such as Java RMI (see Chapter 5) supports only a single programming language. Most middleware is implemented over the Internet protocols, which themselves mask the difference 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 code that can be sent from one computer to another and run at the destination – Java applets are an example. Since the instruction set of a computer depends on its hardware, machine code suitable for running on one type of computer hardware is not suitable for running on another. For example, PC users sometimes send executable files as email attachments to be run by the recipient, but a recipient will not be able to run it, for example, on a Macintosh or Linux computer.

The *virtual machine* approach provides a way of making code executable on any hardware: 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 the Java virtual machine, which needs to be implemented once for each type of hardware to enable Java programs to run. However, the Java solution is not generally applicable to programs written in other languages.

1.4.2 Openness

The openness of a computer system is the characteristic that determines whether the system can be extended and re-implemented 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 huge variety of Internet systems and applications to be built, for example, the Web, which is quite a recent addition to the services in the Internet. RFCs are not the only means of publication. For example, CORBA is published through a series of technical documents, including a complete specification of the interfaces of its services. See [www.omg.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 re-implementation 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.

1.4.3 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 7.

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. At the time of writing (early 2000), there have been several recent 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 7.

1.4.4 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 will remain effective when there is a significant increase in the number of resources and the number of users. The Internet provides an illustration of a distributed system in which the number of computers and services has increased dramatically. Figure 1.5 shows the increase in the number of computers in the Internet during the 20 years up to 1999, and

Figure 1.5 Computers in the Internet

Date	*Computers*	*Web servers*
1979, Dec.	188	0
1989, July	130,000	0
1999, July	56,218,000	5,560,866

Figure 1.6 shows the increasing number of computers and web servers during the six-year history of the Web up to 1999, see [info.isoc.org].

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 8.

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 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 will probably run out in the early 2000s. For this reason, the new version of the protocol will use 128-bit Internet addresses. However, to be fair 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, over-compensating for future growth may be worse than adapting to a change when we are forced to – large Internet addresses 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

Figure 1.6 Computers vs. Web servers in the Internet

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

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 9.

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 (Chapters 8 and 14), the associated technique of caching (Chapters 2 and 8) and the deployment of multiple servers to handle commonly performed tasks, enabling several similar tasks to be performed concurrently.

1.4.5 Failure handling

Computer systems sometimes fail. When faults occur in hardware or software, programs may produce incorrect results or they 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 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 13.

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 14.

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.

1.4.6 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 6 and 12.

1.4.7 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 Standards Organization'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 location.

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 8). 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. Note that although mail programs accept user names for local users, they do append the local domain name. Sending mail to such a user does not involve knowing their physical or network location. Nor does the procedure to send a mail 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 from 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. 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. Also, to a programmer developing parallel programs, not all processors are anonymous. He or she may be interested in the processors used to execute a parallel program, at least to the extent of their number and interconnection topology.

1.5 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 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 its 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.

EXERCISES

1.1 Give five types of hardware resource and five types of data or software resource that can usefully be shared. Give examples of their sharing as it occurs in practice in distributed systems. *pages 2, 7–9*

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 2*

1.3 A user arrives at a railway station that she 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 6*

1.4 What are the advantages and disadvantages of HTML, URLs and HTTP as core technologies for information browsing? Are any of these technologies suitable as a basis for client-server computing in general? *page 9*

1.5 Use the World Wide Web as an example to illustrate the concept of resource sharing, client and server.

Resources in the World Wide Web and other services are named by URLs. What do the initials URL denote? Give examples of three different sorts of web resources that can be named by URLs. *page 7*

1.6 Give an example of a URL.

List the three main components of a URL, stating how their boundaries are denoted and illustrating each one from your example.

To what extent is a URL location transparent? *page 7*

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 16*

1.8 An open distributed system allows new resource sharing services such as the BLOB object in Exercise 1.7 to be added and accessed by a variety of client programs. Discuss in the context of this example, to what extent the needs of openness differ from those of heterogeneity. *page 17*

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 18*

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 19*

1.11 List the three main software components that may fail when a client process invokes a method in a server object, giving an example of a failure in each case. Suggest how the components can be made to tolerate one another's failures. *page 21*

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 22*

1.13 A service is implemented by several servers. Explain why resources might be transferred between them. Would it be satisfactory for clients to multicast all requests to the group of servers as a way of achieving mobility transparency for clients? *page 23*

2

SYSTEM MODELS

An architectural model of a distributed system is concerned with the placement of its parts and the relationships between them. Examples include the client-server model and the peer process model. The client-server model can be modified by:

- the partition of data or replication at cooperating servers;
- the caching of data by proxy servers and clients;
- the use of mobile code and mobile agents;
- the requirement to add and remove mobile devices in a convenient manner.

Fundamental models are concerned with a more formal description of the properties that are common in all of the architectural models.

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 bring out the common properties and design issues for distributed systems in the form of descriptive models. Each model is intended to provide an abstract, simplified but consistent description of a relevant aspect of distributed system design.

An architectural model defines the way in which the components of systems interact with one another and the way in which they are mapped onto an underlying network of computers. In Section 2.2, we describe the layered structure of distributed system software and the main architectural models that determine the locations and interactions of the components. We discuss variants of the client-server model, including those due to the use of mobile code. We consider the features of a distributed system to which mobile devices can be added or removed conveniently. Finally, we look at the general design requirements for distributed systems.

In Section 2.3, we introduce three fundamental models that help to reveal key problems for the designers of distributed systems. Their purpose is to specify the design issues, difficulties and threats that must be resolved in order to develop distributed systems that fulfil their tasks correctly, reliably and securely. The fundamental models provide abstract views of just those characteristics of distributed systems that affect their *dependability* characteristics – correctness, reliability and security.

Difficulties for and threats to 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, many modes of hardware and software failure involving the individual components of a system.

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

2.2 Architectural models

The architecture of a system is its structure in terms of separately specified components. 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) provides a consistent frame of reference for the design.

In this section, we shall describe the main architectural models employed in distributed systems – the architectural styles of distributed systems. We build our architectural models around the concepts of process and object introduced in Chapter 1. An architectural model of a distributed system first simplifies and abstracts the functions of the individual components of a distributed system and then it considers:

- the placement of the components across a network of computers – seeking to define useful patterns for the distribution of data and workload;
- the interrelationships between the components – that is, their functional roles and the patterns of communication between them.

An initial simplification is achieved by classifying processes as *server processes*, *client processes* and *peer processes* – the latter being processes that cooperate and communicate in a symmetrical manner to perform a task. This classification of processes identifies the responsibilities of each and hence helps us to assess their workloads and to determine the impact of failures in each of them. The results of this analysis can then be used to specify the placement of the processes in a manner that meets performance and reliability goals for the resulting system.

Some more dynamic systems can be built as variations on the client-server model:

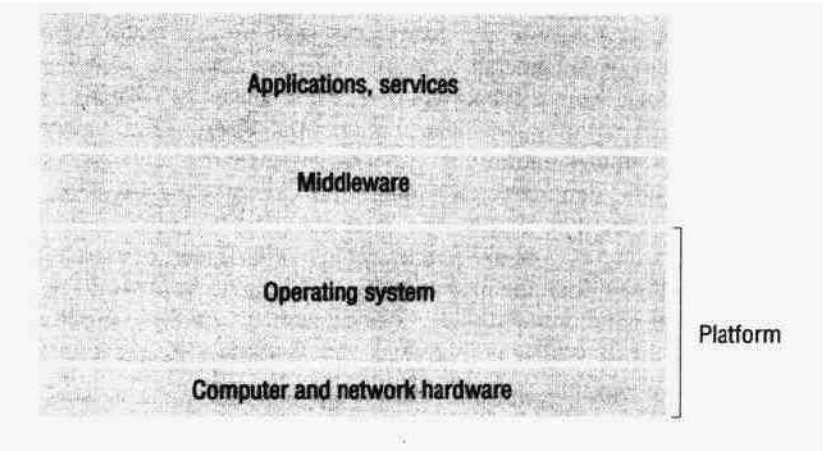
- The possibility of moving code from one process to another allows a process to delegate tasks to another process: for example, clients can download code from servers and run it locally. Objects and the code that accesses them can be moved to reduce access delays and minimize communication traffic.
- Some distributed systems are designed to enable computers and other mobile devices to be added or removed seamlessly, allowing them to discover the available services and to offer their services to others.

There are several widely used patterns for the allocation of work in a distributed system that have an important impact on the performance and effectiveness of the resulting system. The actual placement of the processes that make up a distributed system in a network of computers is also influenced by many detailed issues of performance, reliability, security and cost. The architectural models described here can provide only a somewhat simplified view of the more important patterns of distribution.

2.2.1 Software layers

The term *software architecture* referred originally to the structuring of software as layers or modules in a single computer and more recently in terms of services offered and requested between processes located in the same or different computers. This

Figure 2.1 Software and hardware service layers in distributed systems



process- and service-oriented view can be expressed in terms of *service layers*. We introduce this view in Figure 2.1 and develop it in increasing detail in Chapters 3 to 6. A *server* is a process that accepts requests from other processes. 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 interactions with each other.

Figure 2.1 introduces the important terms *platform* and *middleware*, which we define as follows:

Platform ◊ The lowest-level hardware and software layers are often referred to as a *platform* for distributed systems and applications. 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, Sun SPARC/SunOS, Intel x86/Solaris, PowerPC/MacOS, Intel x86/Linux are major examples.

Middleware ◊ Middleware was defined in Section 1.4.1 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. Middleware 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 replication of shared data and the transmission of multimedia data in real time. Group communication is introduced in Chapter 4 and dealt with in detail in

Chapters 11 and 14. Event notification is described in Chapter 5. Data replication is discussed in Chapter 14 and multimedia systems in Chapter 15.

Remote procedure calling packages such as Sun RPC (described in Chapter 5) and and group communication systems such as Isis (Chapter 14) were amongst the earliest, and currently the most widely-exploited instances of middleware. Object-oriented middleware products and standards include the Object Management Group's Common Object Request Broker Architecture (CORBA), Java Remote Object Invocation (RMI), Microsoft's Distributed Common Object Model (DCOM) and the ISO/ITU-T's Reference Model for Open Distributed Processing (RM-ODP). CORBA and Java RMI are described in Chapters 5 and 17; details on DCOM and RM-ODP can be found in Redmond [1997] and Blair and Stefani [1997].

Middleware can also provide services for use by application programs. They are infrastructural services, tightly bound to the distributed programming model that the middleware provides. For example, CORBA offers a variety of services that provide applications with facilities, which include naming, security, transactions, persistent storage and event notification. Some of the CORBA services are discussed in Chapter 17. The services in the top layer of Figure 2.1 are the domain-specific services that utilize the middleware – its communication operations and its own services.

Limitations of middleware: Many distributed applications rely entirely on the services provided by the available 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.

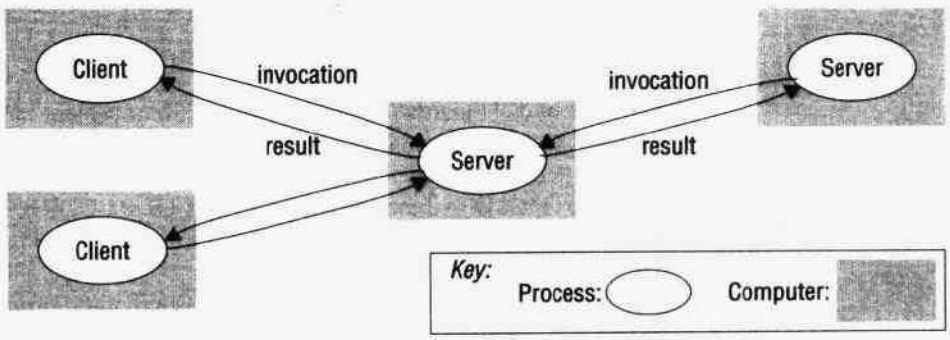
Consider the transfer of large electronic mail messages from the mail host of the sender to that of the recipient. At first sight this a simple application of the TCP data transmission protocol (discussed in Chapter 3). But consider the problem of a user who attempts to transfer a very large file over a potentially unreliable network. TCP provides some error detection and correction, but it cannot recover from major network interruptions. 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. (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.

Figure 2.2 Clients invoke individual servers



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 performing redundant computations.

There is not space to detail their arguments further here; 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].

2.2.2 System architectures

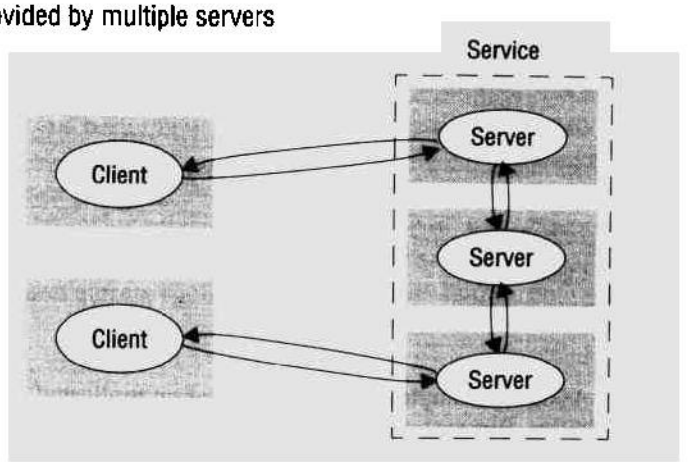
The division of responsibilities between system components (applications, servers and other processes) and the placement of the components on computers in the network is perhaps the most evident aspect of distributed system design. It has major implications for the performance, reliability and security of the resulting system. In this section, we outline the principal architectural models on which this distribution of responsibilities is based.

In a distributed system, processes with well-defined responsibilities interact with each other to perform a useful activity. In this section, our attention will focus on the placement of processes, illustrated in the manner of Figure 2.2, showing the disposition of processes (ellipses) in computers (grey boxes). We use the terms 'invocation' and 'result' to label messages - they could equally be labelled as 'request' and 'reply'.

The main types of architectural model are illustrated in Figure 2.2 to Figure 2.5 and described below.

Client-server model ◊ 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 2.2 illustrates the simple structure in which client processes interact

Figure 2.3 A service provided by multiple servers



with individual server processes in separate host computers in order to access the shared resources that they manage.

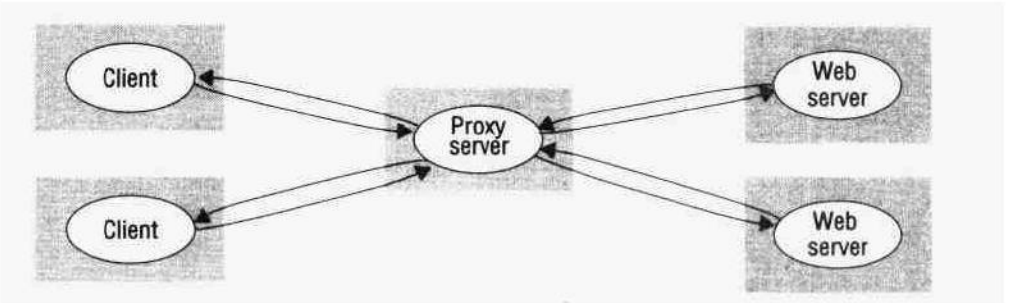
Servers may in turn be clients of other servers, as the figure indicates. For example, a web server is often a client of a local file server that manages the files in which the web pages are stored. Web servers and most other Internet services are clients of the DNS service, which translates Internet 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 2.4, the reader is invited to consider the only synchronization issue that does arise for a concurrent search engine of the type outlined here.

Services provided by 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 2.3). The servers may partition the set of objects on which the service is based and distribute them 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.

Replication is used to increase performance and availability and to improve fault tolerance. It provides multiple consistent copies of data in processes running in different computers. For example, the web service provided at *altavista.digital.com* is mapped onto several servers that have the database replicated in memory.

Figure 2.4 Web proxy server



Another example of a service based on replicated data is the Sun NIS (Network Information Service), which is used by computers on a LAN when users log in. Each NIS server has its own replica of the password file containing a list of users' login names and encrypted passwords. Chapter 14 discusses techniques for replication in detail.

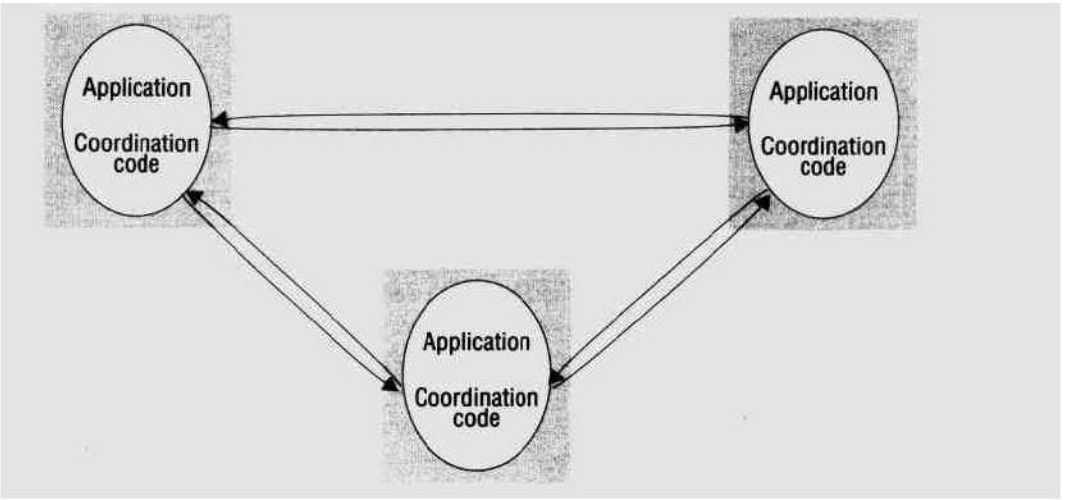
Proxy servers and caches ◊ A *cache* is a store of recently used data objects that is closer than the objects themselves. When a new object is received at a computer it is added to the cache store, replacing some existing objects if necessary. When an object is needed by a client process the caching 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 collocated 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 2.4) provide a shared cache of web resources for the client machines at a site or across several sites. The purpose of proxy servers is to increase 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.

Peer processes ◊ In this architecture all of the processes play similar roles, interacting cooperatively as *peers* to perform a distributed activity or computation without any distinction between clients and servers. In this model, code in the peer processes maintains consistency of application-level resources and synchronizes application-level actions when necessary. Figure 2.5 shows a three-way instance of this; in general, n peer processes may interact with each other, and the pattern of communication will depend on application requirements.

The elimination of server processes reduces inter-process communication delays for access to local objects. Consider a distributed 'whiteboard' application allowing users on several computers to view and interactively modify a picture that is shared between them (Floyd *et al.* [1997] is an example). This can be implemented as an application process at each site that relies on middleware layers to perform event notification and group communication to notify all the application processes of changes to the picture. This provides better interactive response for users of distributed shared objects than could generally be obtained with a server-based architecture.

Figure 2.5 A distributed application based on peer processes



2.2.3 Variations on the client-server model

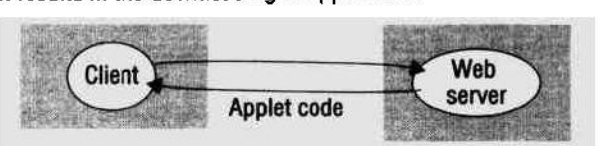
Several variations on the client-server model can be derived from the consideration of the following factors:

- the use of mobile code and mobile agents;
- users' need for low-cost computers with limited hardware resources that are simple to manage;
- the requirement to add and remove mobile devices in a convenient manner.

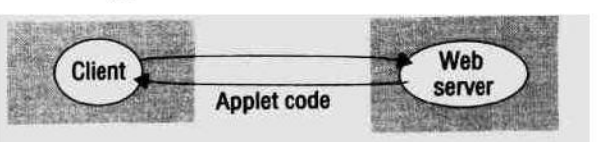
Mobile code ◊ Chapter 1 introduced mobile code. Applets are a well-known and widely used example of mobile code – the user running a browser selects a link to an applet whose code is stored on a web server; the code is downloaded to the browser and runs there, as shown in Figure 2.6. An advantage of running the downloaded code locally is

Figure 2.6 Web applets

a) client request results in the downloading of applet code



b) client interacts with the applet



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 should 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.

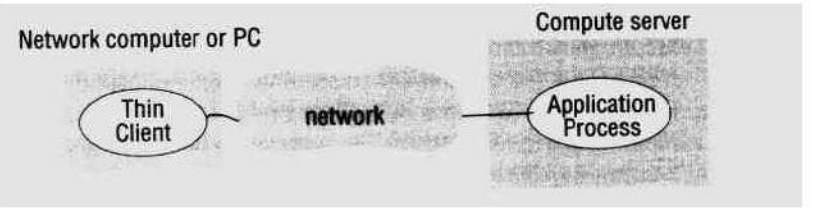
Chapter 1 mentioned that mobile code is a potential security threat to the local resources in the destination computer. Therefore browsers give applets limited access to local resources using a scheme discussed in Section 7.1.1.

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, 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 the site of each vendor 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 on 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.

Figure 2.7 Thin clients and compute servers



Network computers ◊ In the architecture illustrated in Figure 1.2 applications run on a desktop computer local to the user. The operating systems and application software for desktop computers typically require much of the active code and data to be located on a local disk. But the management of application files and the maintenance of a local software base requires considerable technical effort of a nature that most users are not qualified to provide.

The network computer is a response to this problem. It downloads its operating system and any application software needed by the user from a remote file server. Applications are run locally but the files are managed by a remote file server. Network applications such as a web browser can also be run. Since all the application data and code is stored by a file server, the users may migrate from one network computer to another. The processor and memory capacities of a network computer can be constrained in order to reduced its cost.

If a disk is included, it holds only a minimum of software. The remainder of the disk is used as *cache storage* holding copies of software and data files that have recently been loaded from servers. The maintenance of the cache requires no manual effort: cached objects are invalidated whenever a new version of the file is written at the relevant server.

Thin clients ◊ The term *thin client* refers to a software layer that supports a window-based user interface on a computer that is local to the user while executing application programs on a remote computer (Figure 2.7). This architecture has the same low management and hardware costs as the network computer scheme, but instead of downloading the code of applications into the user's computer, it runs them on a *compute server* - a powerful computer that has the capacity to run large numbers of applications simultaneously. The compute server will typically be a multiprocessor or cluster computer (see Chapter 6) running a multiprocessor version of an operating system such as UNIX or Windows NT.

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 by the need to transfer image and vector information between the thin client and the application process, incurring both network and operating system latencies.

Thin client implementations: Thin client systems are simple in concept and their implementation is straightforward in some environments. For example, most variants of UNIX include the X-11 window system which is discussed in the box on the following page.

There are several points in the pipeline of graphical operations required to support a graphical user interface at which the boundary between client and server can be drawn. X-11 draws the boundary at the level of graphical primitives for line and shape drawing and window handling. The Citrix *WinFrame* product [www.citrix.com] is a widely used commercial implementation of the thin client concept that operates in a similar manner. This product provides a thin client process running on a wide variety of platforms and supporting a desktop providing interactive access to applications running on Windows NT hosts. Other implementations include the Teleporting and Virtual Network Computer (VNC) systems developed at AT&T Laboratories in Cambridge, England [Richardson *et al.* 1998]. VNC draws the boundary at the level of operations on screen pixels and maintains the graphical context for users when they move between computers.

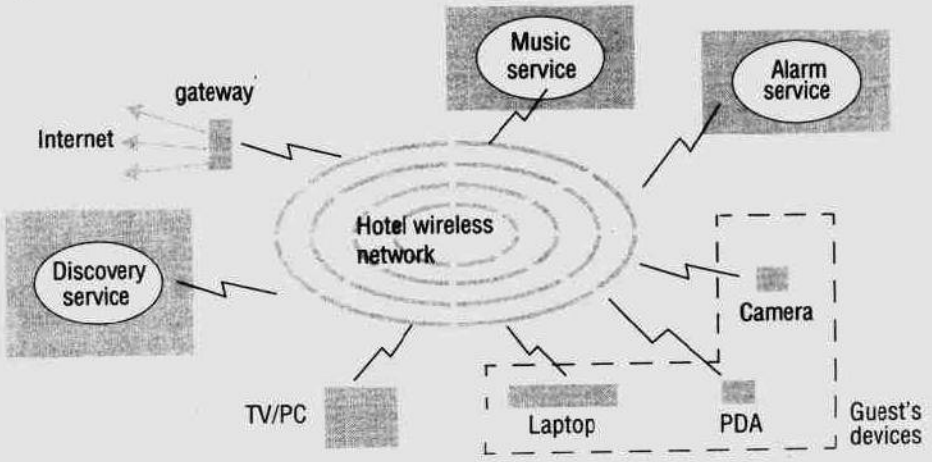
Mobile devices and spontaneous networking ◊ As Chapter 1 explained, the world is increasingly populated by small and portable computing devices, including laptops, handheld devices such as personal digital assistants (PDAs), mobile phones and digital cameras, wearable computers such as smart watches, and devices embedded in everyday appliances such as washing machines. Many of these devices are capable of wireless networking, with metropolitan or greater ranges (GSM, CDPD), ranges of hundreds of metres (WaveLAN), or a few metres (BlueTooth, infra-red and HomeRF). The shorter-range networks have bandwidths up to the order of 10 megabits/second; GSM promises in the order of hundreds of kilobits/second.

With appropriate integration into our distributed systems, these devices provide support for mobile computing (see Section 1.2.3), whereby users carry their mobile devices between network environments and take advantage of local and remote services as they do so. The form of distribution that integrates mobile devices and other devices into a given network is perhaps best described by the term *spontaneous networking*. This term is used to encompass applications that involve the connection of both mobile and non-mobile devices to networks in a more informal manner than has hitherto been possible. Devices such as those embedded in appliances provide services to users and to

The X-11 window system ◊ The X-11 window system [Scheifler and Gettys 1986] is a process that manages the display and interactive input devices (keyboard, mouse) of the computer on which it runs. X-11 provides an extensive library of procedures (the *X-11 protocol*) for displaying and modifying graphical objects in windows as well as the creation and manipulation of the windows themselves.

The X-11 system is referred to as a *window server* process. The clients of the X-11 server are the application programs that the user is currently interacting with. The client programs communicate with the server by invoking operations in the X-11 protocol; these include operations to draw text and graphical objects in windows. The clients need not be located in the same computer as the server, since the server procedures are always invoked using an RPC mechanism. The X-11 window server therefore has the key properties of a thin client. (X-11 inverts the client-server terminology. The X-11 server derives its name from the graphical display services that it provides to application programs, whereas thin client software is named with reference to its use of application programs running on a compute server.)

Figure 2.8 Spontaneous networking in a hotel



other nearby devices; portable devices such as PDAs give the user access to the services in their current location, as well as global services such as the Web.

Chapter 1 gave an example of a user visiting a host organization. As another illustration of spontaneous networking, Figure 2.8 shows a wireless network that covers a hotel suite. The suite's hi-fi system provides a music service, by which users can pipe any combination of music selections to speakers in the bedroom or bathroom, or to wirelessly connected headsets. The alarm system provides a wake-up service via the music service. The TV/PC (a television with a 'set-top box') acts as both a television and a computer. It provides access to the Web (via the hotel's Internet gateway), including a web-based interface to all hotel facilities. It provides a viewing service for images that the user has stored on their camera or camcorder. It can also provide the user's favourite applications, on a rental basis.

The figure shows the devices that the user brings to the hotel suite: a laptop, a digital camera and a PDA, whose infra-red capability enables it to function as a 'universal controller' – a device similar to a television's remote control but which can be used to control a variety of devices around the hotel room, including the lighting and music service.

While some of the devices in the hotel such as printers and automated vending machines are not routinely portable, there is a requirement to introduce them with minimal human intervention. Their connection to the network and access to the services of the network should be made possible without prior notice or prior administrative actions.

The key features of spontaneous networking are:

Easy connection to a local network: Wireless links avoid the need for pre-installed cabling and avoid the inconvenience and reliability issues surrounding plugs and sockets. They are the only communication links that can be maintained while on the move (on trains, planes, cars, bicycles or boats). A device brought into a new network

environment is transparently reconfigured to obtain connectivity there: the user should not have to type in the names or addresses of local services to achieve this.

Easy integration with local services: Devices that find themselves inserted into (and moving between) existing networks of devices discover automatically what services are provided there, with no special configuration actions by the user. In the hotel example, the guest's digital camera discovers viewing services such as that of the TV/PC automatically, so that the guest can send the camera's stored images for display.

Spontaneous networking raises some significant design issues. First, there is the challenge of supporting convenient connection and integration. For example, Internet addressing and routing algorithms assume that computers are located on a particular subnetwork. If a computer is moved to another subnet it can no longer be accessed with the same Internet address. An approach to the problem is discussed in Chapter 3.

Other issues arise for mobile users. These users may experience limited connectivity as they travel; and the spontaneous nature of their connectivity gives rise to security problems.

Limited connectivity: Users are not always connected as they move around. For example, they may be intermittently disconnected from a wireless network as they travel on a train through tunnels. They may also be totally disconnected for longer periods of time in regions where wireless connectivity ceases altogether or when it is too expensive to remain connected. The issue here is how the system can support the user so that they can continue to work while disconnected. Chapter 14 discusses this topic.

Security and privacy: Many security and personal privacy issues arise in scenarios such as that of the hotel guests. The hotel and its guests are vulnerable to security attacks from guests or hotel employees who attempt wireless connections in unsupervised ways. Some systems track the physical locations of users as they move around (such as systems based on 'active badges' [Want *et al.* 1992]), and this may threaten the users' privacy. Finally, a facility that enables users to access their home intranet while on the move may expose data that is supposed to remain behind the intranet firewall, or it may open up the intranet to attacks from outside.

Discovery services: Spontaneous networking requires client processes running on portable devices and other appliances to access services on the networks to which they are connected. But in a world of devices that perform diverse functions, how can clients be connected to the services that they need in order to complete useful tasks? Let us explore this in the context of a specific example: on returning to the hotel, our guest wishes to view some photographs that they have taken with a digital camera, print some of them for local use and send a selection of them back home electronically. Her room has a TV/PC. The hotel might have a digital colour printer service, or if not, it probably has a fax machine service.

We cannot expect every printer manufacturer to implement exactly the same protocol for clients to access its printing services. Even if that *was* the case, clients can often be made adaptable, as our example is intended to indicate: a digital camera client might display the photographs on the local TV set or send them to the hotel's fax machine.

What is required is a means for clients to discover what services are available in the network to which they are connected and to investigate their properties. The purpose of a *discovery service* is to accept and store details of services that are available on the network and to respond to queries from clients about them. More precisely, a discovery service offers two interfaces:

- A *registration service* accepts registration requests from servers and records the details that they contain in the discovery service's database of currently available services.
- A *lookup service* accepts queries concerning available services and searches its database for registered services that match the queries. The result returned includes sufficient details to enable clients to select between several similar services based on their attributes and to make a connection to one or more of them.

Returning to our example, a *discovery service* in the hotel includes details of the printing and viewing services available in the hotel, with details such as the room number in which the display or printer is located, the quality (resolution) of the device, imaging models that the device accepts, colour capabilities and so on. The guest's digital camera queries the discovery service for printing and viewing services, processes the resulting list for proximity to the room in which it is located and for compatibility with its imaging model and offers the user a menu of suitable devices. The user selects the TV/PC in the room or a printer and dispatches some images to it. Other images might be sent to friends and family as email attachments.

Chapter 9 includes a closer look at discovery services.

2.2.4 Interfaces and objects

The set of functions available for invocation in a process (whether it is a server or a peer process) is specified by one or more *interface definitions*. Interface definitions are described fully in Chapter 5, but the concept will be familiar to those already conversant with languages such as Modula, C++ or Java. In the basic form of client-server architecture, each server process is seen as a single entity with a fixed interface defining the functions that can be invoked in it.

In object-oriented languages such as C++ and Java, with appropriate additional support, distributed processes can be constructed in a more object-oriented manner. Many objects can be encapsulated in server or peer processes, and references to them are passed to other processes so that their methods can be accessed by remote invocation. This is the approach adopted by CORBA and by Java with its remote method invocation (RMI) mechanism. In this model, the number and the types (in languages that support mobile code such as Java) of objects hosted by each process may vary as system activities require, and in some implementations the locations of objects may also change.

Whether we adopt a static client–server architecture or the more dynamic object-oriented model outlined in the preceding paragraph, the distribution of responsibilities between processes and between computers remains an important aspect of the design. In the traditional architectural model, the responsibilities are statically allocated (a file server, for example, is responsible only for files, not for web pages, their proxies or other

types of object). But in the object-oriented model new services, and in some cases new types of object, can be instantiated and immediately be made available for invocation.

2.2.5 Design requirements for distributed architectures

The factors motivating the distribution of objects and processes in a distributed system are numerous and their significance is considerable. Sharing was first achieved in the timesharing systems of the 1960s with the use of shared files. The benefits of sharing were quickly recognized and exploited in multi-user operating systems such as UNIX and multi-user database systems such as Oracle to enable processes to share system resources and devices (file storage capacity, printers, audio and video streams) and application processes to share application objects

The arrival of cheap computing power in the form of microprocessor chips removed the need for the sharing of central processors, and the availability of medium-performance computer networks and the continuing need for the sharing of relatively costly hardware resources such as printers and disk storage led to the development of the distributed systems of the 1970s and 1980s. Today, resource sharing is taken for granted. But effective data sharing on a large scale remains a substantial challenge – with changing data (and most data changes with time), the possibility of concurrent and conflicting updates arises. The control of concurrent updates in shared data is the subject of Chapters 12 and 13.

Performance issues ◊ The challenges arising from the distribution of resources extend well beyond the need for the management of concurrent updates. Performance issues arising from the limited processing and communication capacities of computers and networks are considered under the following subheadings:

Responsiveness: Users of interactive applications require a fast and consistent response to interaction; but client programs often need to access shared resources. When a remote service is involved, the speed at which the response is generated is determined not just by the load and performance of the server and the network but also by delays in all the software components involved – the client and server operating systems' communication and middleware services (remote invocation support, for example) as well as the code of the process that implements the service.

In addition, the transfer of data between processes and the associated switching of control is relatively slow, even when the processes reside in the same computer. To achieve good interactive response times, systems must be composed of relatively few software layers, and the quantities of data transferred between the client and server must be small.

These issues are demonstrated by the performance of web-browsing clients, where the fastest response is achieved when accessing locally cached pages and images because these are held by the client application. Remote text pages are also accessed reasonably quickly because they are small, but graphical images incur longer delays because of the volume of data involved.

Throughput: A traditional measure of performance for computer systems is the throughput – the rate at which computational work is done. We are interested in the ability of a distributed system to perform work for all its users. This is affected by processing speeds at clients and servers and by data transfer rates. Data that is located

on a remote server must be transferred from the server process to the client process, passing through several software layers in both computers. The throughput of the intervening software layers is important, as well as that of the network.

Balancing computational loads: One of the purposes of distributed systems is to enable applications and service processes to proceed concurrently without competing for the same resources and to exploit the available computational resources (processor, memory and network capacities). For example, the ability to run applets on client computers removes load from the web server, enabling it to provide a better service. A more significant example is in the use of several computers to host a single service. This is needed, for example, by some heavily loaded web servers (search engines, large commercial sites). This exploits the facility of the DNS domain name lookup service to return one of several host addresses for a single domain name (see Section 9.2.3).

In some cases, load balancing may involve moving partially completed work as the loads on hosts change. This calls for a system that can support the movement of running processes between computers.

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 non-functional 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 recently been recognized as a further important aspect of service quality. Birman has long argued for the importance of these quality aspects and his book [Birman 1996] provides some interesting perspectives of their impact on system design.

Reliability and security issues are critical in the design of most computer systems. They are strongly related to two of our fundamental models: the failure model and the security model, introduced in Sections 2.3.2 and 2.3.3.

The performance aspect of quality of service was until recently defined in terms of responsiveness and computational throughput, but it has recently been redefined in terms of ability to meet timeliness guarantees as discussed in the following paragaphs. With either definition, the performance aspect of distributed systems is strongly related to the interaction model defined in Section 2.3.1. All three of these fundamental models are important reference points throughout the book.

Some 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, for example to browse the Web, may have quite good performance characteristics, but when they are heavily loaded their performance deteriorates significantly – in no way can they be said to provide quality of

service. 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 15.

Use of caching and replication ◊ The performance issues outlined above often appear to be major obstacles to the successful deployment of distributed systems, but much progress has been made in the design of systems that overcome them by the use of data replication and caching. Section 2.2.2 introduced caches and web proxy servers, without discussing how cached copies of resources can be kept up to date when the resource at a server is updated. A variety of different cache consistency protocols are used to suit different applications for example, Chapter 8 gives the protocols used by two different file server designs. We now outline the web-caching protocol that is part of the HTTP protocol, which is described in Section 4.4.

Web-caching protocol: Both web browsers and proxy servers cache responses to client requests to web servers. Therefore a client request may be satisfied by a response cached by the browser or by a proxy server between it and the web server. The cache consistency protocol can be configured to provide browsers with fresh (reasonably up-to-date) copies of the resources held by the web server. But, to allow for performance, availability and disconnected operation the freshness condition may be relaxed.

A browser or proxy can *validate* a cached response by checking with the original web server to see whether it is still up to date. If it fails the test, the web server returns a fresh response, which is cached instead of the stale response. Browsers and proxies validate cached responses when clients request the corresponding resources. But they need not perform a validation if the cached response is sufficiently fresh. Even though a web server knows when a resource is updated, it does not notify the browsers and proxies with caches – to do that the web server would need to keep a record of interested browsers and proxies for each of its resources. To enable browsers and proxies to determine whether their stored responses are stale, web servers assign approximate expiry times to their resources, for example estimated from the last time they were updated. An expiry time can be misleading, because a web resource may be updated at any time. Whenever a web server replies to a request, the expiry time of the resource and the current time at the server are attached to the response.

Browsers and proxies store the expiry time and server time together with the cached response. This enables a browser or proxy receiving future requests to calculate whether a cached response is likely to be stale. It calculates whether a cached response is stale by comparing its age with the expiry time. The *age* of a response is the sum of the time the response has been cached and the server time. Note that this calculation does not depend on the computer clocks at the web server and the browser or proxy agreeing with one another.

Dependability issues ◊ Dependability is a requirement in most application domains. It is crucial not only in command and control activities, where life and limb may well be at stake, but also in many commercial applications, including the rapidly developing domain of Internet commerce, where the financial safety and soundness of the participants depends upon the dependability of the systems that they operate. In Section 2.1, we defined the dependability of computer systems as *correctness*, *security* and *fault tolerance*. Here we discuss the architectural impact of the needs for security and fault-

tolerance, leaving the description of relevant techniques for achieving them to later in the book. The development of techniques for checking or ensuring the correctness of distributed and concurrent programs is the subject of much current and recent research. Although some promising results have been achieved, few if any of them are yet mature enough for deployment in practical applications.

Fault tolerance: Dependable applications should continue to function correctly in the presence of faults in hardware, software and networks. Reliability is achieved through redundancy — the provision of multiple resources so that the system and application software can reconfigure and continue to perform its tasks in the presence of faults. Redundancy is expensive, and there are limits to the extent to which it can be employed; hence there are also limits to the degree of fault tolerance that can be achieved.

At the architectural level, redundancy requires the use of multiple computers at which each component process of the system can run and multiple communication paths through which messages can be transmitted. Data and processes can then be replicated wherever needed to provide the required level of fault tolerance. A common form of redundancy is the provision of several replicas of a data item at different computers – so long as one of the computers is still running, the data may be accessed. Some critical applications such as air traffic control systems require a very high guarantee of fault tolerance of data, which involves a high cost in keeping the multiple replicas up to date. Chapter 14 discusses this matter further.

Other forms of redundancy are used to make communication protocols reliable. For example, messages are retransmitted until an acknowledgement message has been received. The reliability of the protocols underlying RMI is covered in Chapters 4 and 5.

Security: The architectural impact of the requirement for security concerns the need to locate sensitive data and other resources only in computers that can be secured effectively against attack. For example, a hospital database contains patient records with components that are sensitive and should be seen only by certain clinicians, while other components are more widely available. It would not be appropriate to construct a system that loads an entire patient record into a user's desktop computer when it is accessed, because the typical desktop computer does not constitute a secure environment – users can run programs to access or update any part of the data stored in their personal computer. A security model that addresses wider requirements for security is introduced in Section 2.3.3, and Chapter 7 describes the techniques that are available for its achievement.

2.3 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 given in the previous section, which are concerned primarily with the performance and reliability characteristics of processes and networks and 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, a model contains only the essential ingredients that we need to consider in order to understand and reason about some aspects of a system's behaviour. A system model has to address the questions:

- What are the main entities in the system?
- How do they interact?
- What are the characteristics that affect their individual and collective behaviour?

The purpose of 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 (i.e. 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.

2.3.1 Interaction model

The discussion of system architectures in Section 2.2 indicates that 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 Service, 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 sending of a message by one process and 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 then 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 associate timestamps with their events. However, even if 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 relative amount that a computer clock differs 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 would 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 GPS 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 the 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 11.

Two variants of the interaction model ◊ In a distributed system it is hard to set time limits on the time 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 the section on the failure model.

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, for example 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 zero 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 the next 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 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. The box below illustrates the impossibility of synchronizing clocks in an asynchronous system.

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:

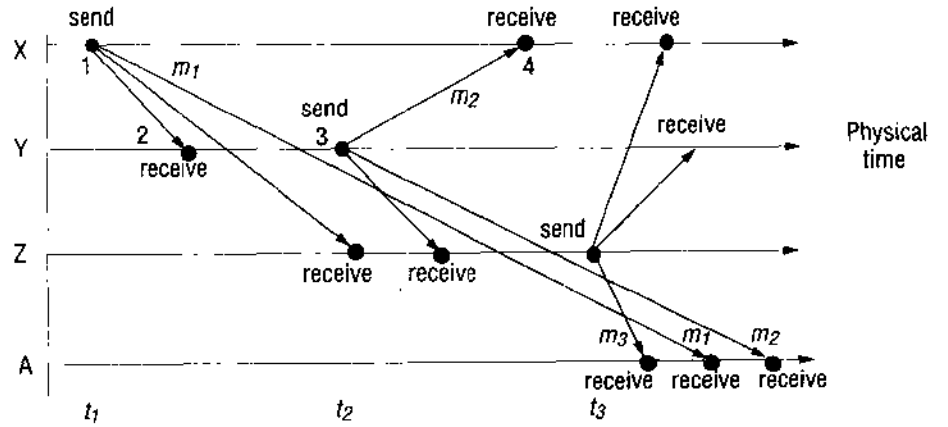
1. User X sends a message with the subject *Meeting*;
2. Users Y and Z reply by sending a message with the subject *Re: Meeting*;

In real time, X's message was sent first, Y reads it and replies; Z reads both X's message and Y's reply and then 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 2.9, 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

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 sends 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 have taken, say, five minutes. 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 2.9 Real-time ordering of event.



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 10, 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 2.9, 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, therefore 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 2.9 shows the numbers 1 to 4 on the events at X and Y.

2.3.2 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 4 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 13 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.

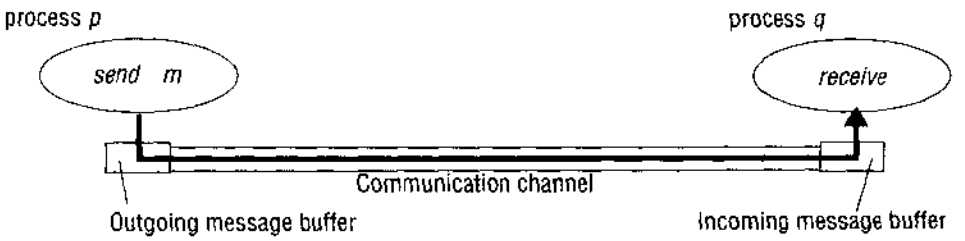
Process omission failures: The chief omission failure of a process is to crash. When we say that a process has crashed we mean that it has halted and will not execute any further steps of its program ever. The design of services that can survive in the presence of faults can be simplified if it can be assumed that the services on which it depends crash cleanly – that is, the 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 2.10). 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

Figure 2.10 Processes and channels



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*; loss of messages between the incoming message buffer and the receiving process as *receive-omission failures*; and loss of messages in between as

Failure detection ◊ In the case of the Pepperland divisions encamped at the tops of hills (see page 52), 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 if the time 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 agreement in the presence of 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 her 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 2.11 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*, 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, commit omissions; a process may stop or take an incorrect step.

channel omission failures. The omission failures are classified together with arbitrary failures in Figure 2.11.

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. Therefore arbitrary failures in processes 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 or non-existent messages may be delivered or real messages may be delivered more than once. Arbitrary failures of communication channels are rare 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 non-existent 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 2.12. Any one of these failures may result in responses being unavailable to clients within a specified time interval.

Figure 2.12 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.

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. To deliver 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 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 14 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.

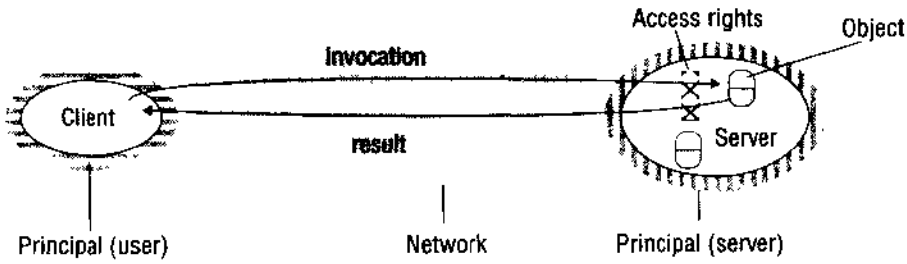
Reliability of one-to-one communication ◊ Although a basic communication channel can exhibit the omission failures described above, it is possible to use it to build a communication service that masks some of those failures.

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.

Figure 2.13 Objects and principals



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 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.

2.3.3 Security model

In Section 2.2 we identified the sharing of resources as a motivating factor for distributed systems and we described their system architecture in terms of processes encapsulating objects 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.

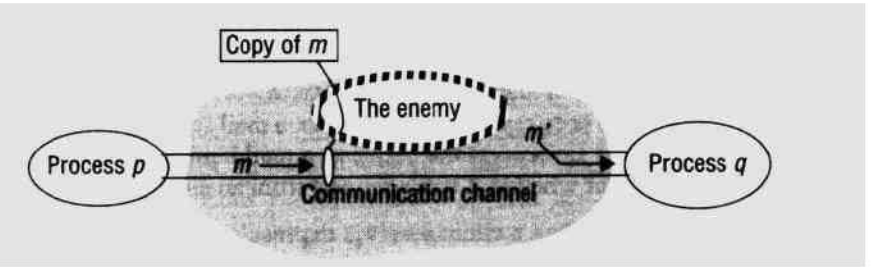
Protection is described in terms of objects, although the concepts apply equally well to resources of all types.

Protecting objects ◊ Figure 2.13 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.

Figure 2.14 The enemy



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 is 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 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 analyse 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 between a pair of processes, as shown in Figure 2.14. 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 and 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 are discussed under the headings *threats to processes*, *threats to communication channels* and *denial of service*.

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.

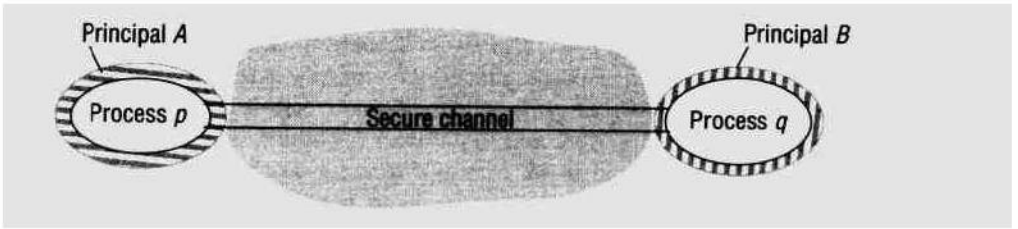
Defeating security threats ◊ Here we introduce the main techniques on which secure systems are based. Chapter 7 discusses the design and implementation of secure distributed systems in some detail.

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

Figure 2.15 Secure channels



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 2.15. 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 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 time stamp to prevent messages from being replayed or reordered.

The construction of secure channels is discussed in detail in Chapter 7. 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 7) are instances.

Other possible threats from an enemy ◊ Section 1.4.3 introduced very briefly two 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 1.4.3. 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.

2.4 Summary

Most distributed systems are arranged according to one of a variety of architectural models. The client-server model is prevalent – the Web and other Internet services such as ftp, news and mail as well as 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. In the peer process model, distributed application processes communicate directly with one another without using a server as an intermediary.

The ability to move code from one process to another has resulted in some variants of the client-server model. The most common example of this is the applet whose code is supplied by a web server to be run by a client, providing functionalities not available in the client and improved performance due to being close to the user.

The existence of portable computers, PDAs and other digital devices and their integration into distributed systems allows users to access local and Internet services when they are away from their desktop computer. The presence of computing devices with wireless communication capabilities in everyday objects such as washing machines and burglar alarms allows them to provide services accessible on a wireless network in

the home. A characteristic of mobile devices in a distributed system is that they connect and disconnect unpredictably, leading to a requirement for a discovery service by which mobile servers can offer their services and mobile clients can look them up.

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 Describe and illustrate the client-server architecture of one or more major Internet applications (for example the Web, email or netnews). *page 34*

2.2 For the applications discussed in Exercise 2.1 state how the servers cooperate in providing a service. *page 35*

2.3 How do the applications discussed in Exercise 2.1 involve the partitioning and/or replication (or caching) of data amongst servers? *page 35*

2.4 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 34*

2.5 Suggest some applications for the peer process model, distinguishing between cases when the state of all peers needs to be identical and cases that demand less consistency. *page 36*

2.6 List the types of local resource that are vulnerable to an attack by an untrusted program that is downloaded from a remote site and run in a local computer. *page 37*

2.7 Give examples of applications where the use of mobile code is beneficial. *page 37*

2.8 What factors affect the responsiveness of an application that accesses shared data managed by a server? Describe remedies that are available and discuss their usefulness. *page 44*

2.9 Distinguish between buffering and caching. *page 46*

2.10 Give some examples of faults in hardware and software that can/cannot be tolerated by the use of redundancy in a distributed system. To what extent does the use of redundancy in the appropriate cases make a system fault-tolerant? *page 47*

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 49*

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 49*

2.13 The Network Time Protocol service can be used to synchronize computer clocks. Explain why, even with this service, no guaranteed bound is given for the difference between two clocks. *page 50*

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 effect on the properties of validity and integrity. Can service B be described as a reliable communication service? *page 53 and page 57*

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 53*

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 57*

2.17 Define the integrity property of reliable communication and list all the possible threats to integrity from users and from system components. What measures can be taken to ensure the integrity property in the face of each of these sources of threats. *pages 57, 60*

2.18 Describe possible occurrences of each of the main types of security threat (threats to processes, threats to communication channels, denial of service) that might occur in the Internet. *page 59*