DISTRIBUTED SYSTEMS

Concepts and Design

Fifth Edition

George Coulouris
Jean Dollimore
Tim Kindberg
Gordon Blair

This page intentionally left blank

DISTRIBUTED SYSTEMS

Concepts and Design

Fifth Edition

1

CHARACTERIZATION OF DISTRIBUTED SYSTEMS

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

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

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

1.1 Introduction

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

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

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

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

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

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

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

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

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

1.2 Examples of distributed systems

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

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

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

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

1.2.1 Web search

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

Figure 1.1 Selected application domains and associated networked applications

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

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

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

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

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

1.2.2 Massively multiplayer online games (MMOGs)

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

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

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

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

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

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

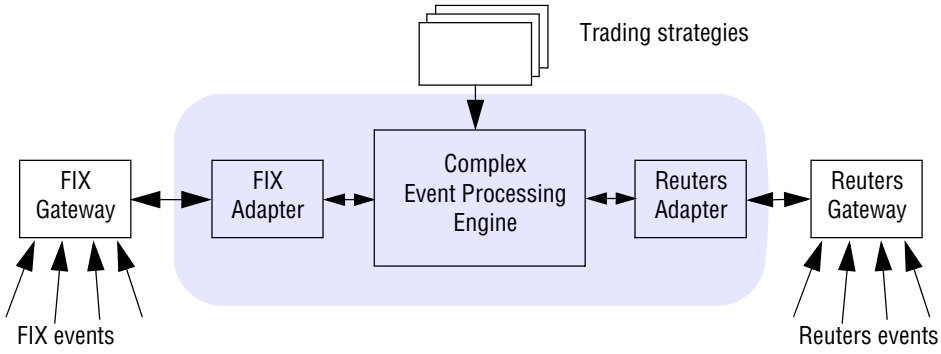
1.2.3 Financial trading

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

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

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

Figure 1.2 An example financial trading system



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

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

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

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

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

1.3 Trends in distributed systems

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

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

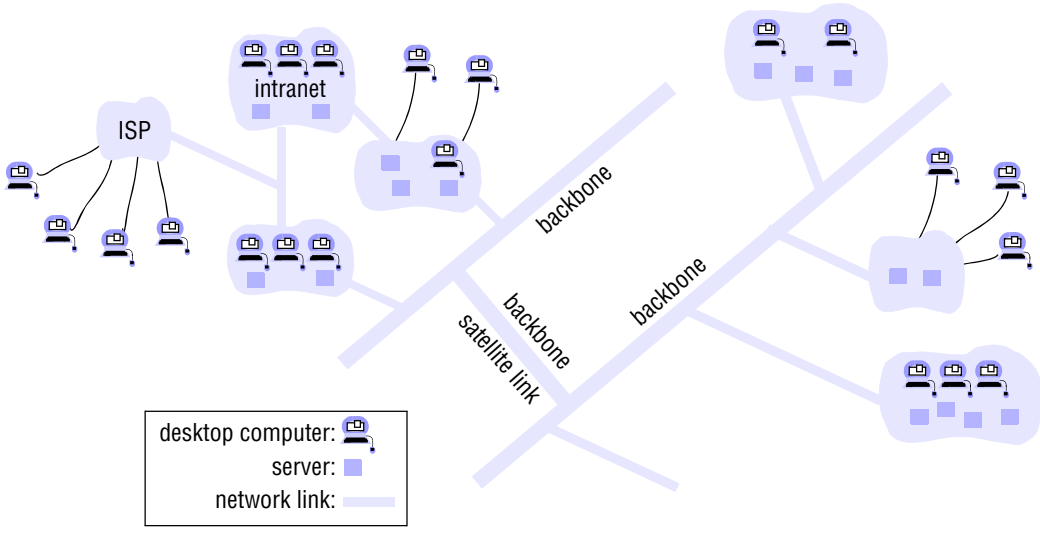
1.3.1 Pervasive networking and the modern Internet

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

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

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

Figure 1.3 A typical portion of the Internet



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

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

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

1.3.2 Mobile and ubiquitous computing

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

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

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

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

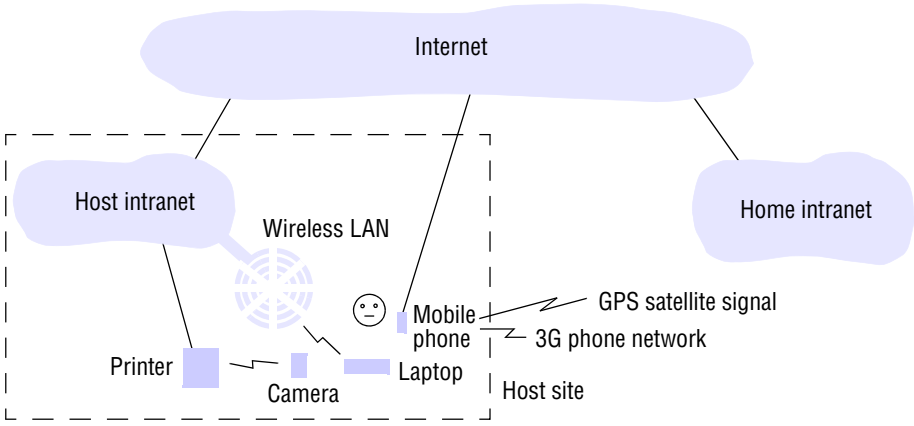
The presence of computers everywhere only becomes useful when they can communicate with one another. For example, it may be convenient for users to control their washing machine or their entertainment system from their phone or a 'universal remote control' device in the home. Equally, the washing machine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

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

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

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

Figure 1.4 Portable and handheld devices in a distributed system



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

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

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

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

1.3.3 Distributed multimedia systems

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

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

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

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

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

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

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

1.3.4 Distributed computing as a utility

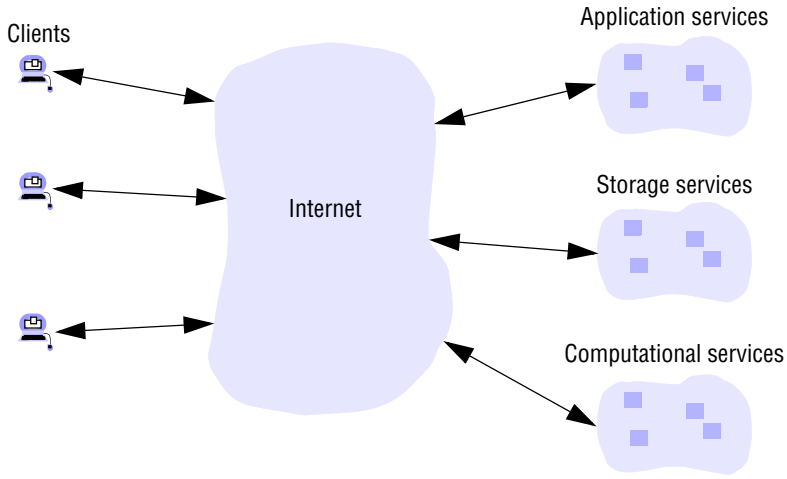
With the increasing maturity of distributed systems infrastructure, a number of companies are promoting the view of distributed resources as a commodity or utility, drawing the analogy between distributed resources and other utilities such as water or electricity. With this model, resources are provided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

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

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

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

Figure 1.5 Cloud computing



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

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

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

1.4 Focus on resource sharing

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

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

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

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

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

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

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

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

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

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

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

1.5 Challenges

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

1.5.1 Heterogeneity

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

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

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

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

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

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

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

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

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

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

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

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

1.5.2 Openness

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

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

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

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

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

To summarize:

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

1.5.3 Security

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

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

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

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

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

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

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

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

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

1.5.4 Scalability

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

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

The design of scalable distributed systems presents the following challenges:

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

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

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

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

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

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

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

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

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

1.5.5 Failure handling

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.5.6 Concurrency

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

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

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

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

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

1.5.7 Transparency

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.5.8 Quality of service

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

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

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

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

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

1.6 Case study: The World Wide Web

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

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

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

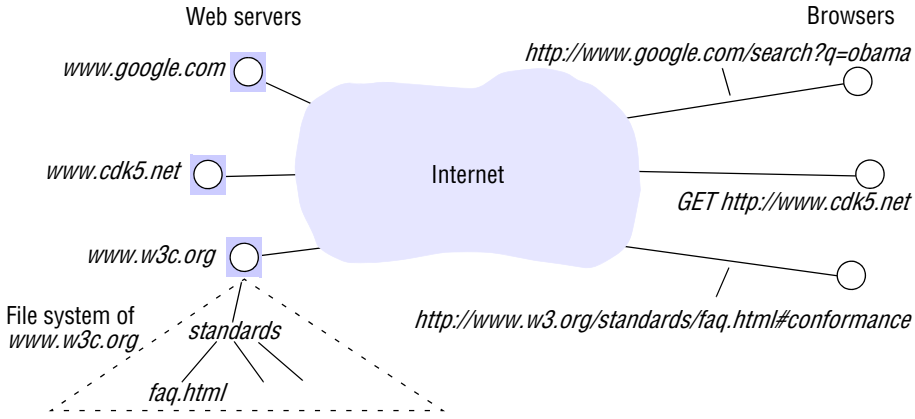
The Web is an *open* system: it can be extended and implemented in new ways without disturbing its existing functionality (see Section 1.5.2). First, its operation is based on communication standards and document or content standards that are freely published and widely implemented. For example, there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

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

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

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

Figure 1.7 Web servers and web browsers



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

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

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

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

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

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

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

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

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

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

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

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

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

scheme : scheme-specific-identifier

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

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

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

In general, HTTP URLs are of the following form:

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

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

Consider the URLs:

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

These can be broken down as follows:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.7 Summary

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

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

The construction of distributed systems produces many challenges:

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

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

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

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

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

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

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

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

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

EXERCISES

1.1 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, 14*

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 Consider the implementation strategies for massively multiplayer online games as discussed in Section 1.2.2. In particular, what advantages do you see in adopting a single server approach for representing the state of the multiplayer game? What problems can you identify and how might they be resolved? *page 5*

1.4 A user arrives at a railway station that they has never visited before, carrying a PDA that is capable of wireless networking. Suggest how the user could be provided with information about the local services and amenities at that station, without entering the station's name or attributes. What technical challenges must be overcome? *page 13*

1.5 Compare and contrast cloud computing with more traditional client-server computing? What is novel about cloud computing as a concept? *pages 13, 14*

1.6 Use the World Wide Web as an example to illustrate the concept of resource sharing, client and server. 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? *pages 14, 26*

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*

1.14 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 26*

1.15 Give an example of an HTTP URL. List the main components of an HTTP URL, stating how their boundaries are denoted and illustrating each one from your example. To what extent is an HTTP URL location-transparent? *page 26*

This page intentionally left blank