

# CHARACTERIZATION OF DISTRIBUTED SYSTEMS

- 1.1 Introduction
- 1.2 Examples of distributed systems
- 1.3 Trends in distributed systems
- 1.4 Focus on resource sharing
- 1.5 Challenges
- 1.6 Case study: The World Wide Web
- 1.7 Summary

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

<i>Finance and commerce</i>	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.
<i>The information society</i>	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.
<i>Creative industries and entertainment</i>	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.
<i>Healthcare</i>	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.
<i>Education</i>	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.
<i>Transport and logistics</i>	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.
<i>Science</i>	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.
<i>Environmental management</i>	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.

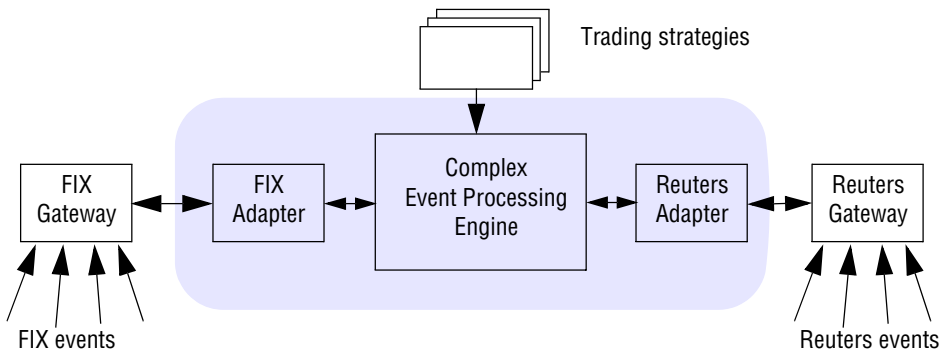
- Other MMOGs adopt more distributed architectures where the universe is partitioned across a (potentially very large) number of servers that may also be geographically distributed. Users are then dynamically allocated a particular server based on current usage patterns and also the network delays to 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](http://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.

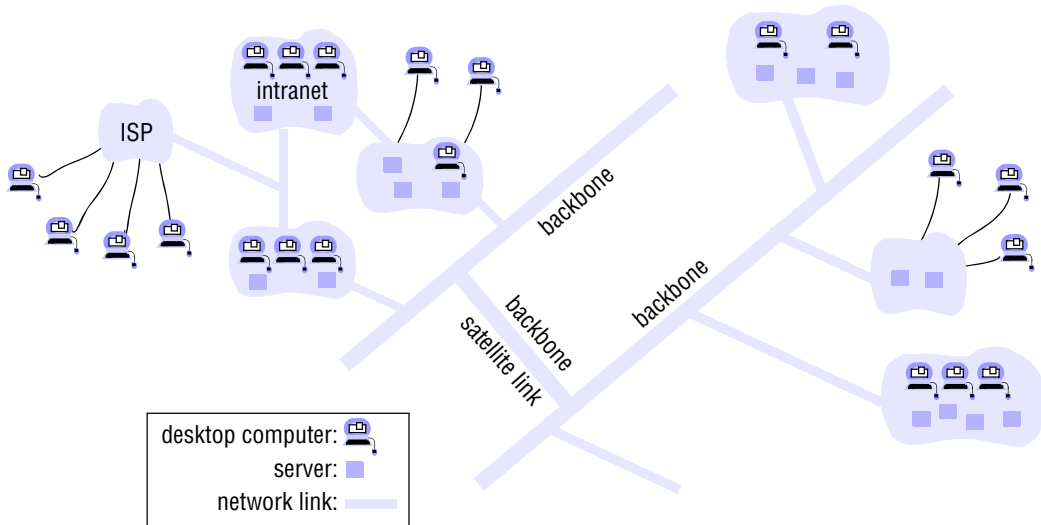
### 1.3.1 Pervasive networking and the modern Internet

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).

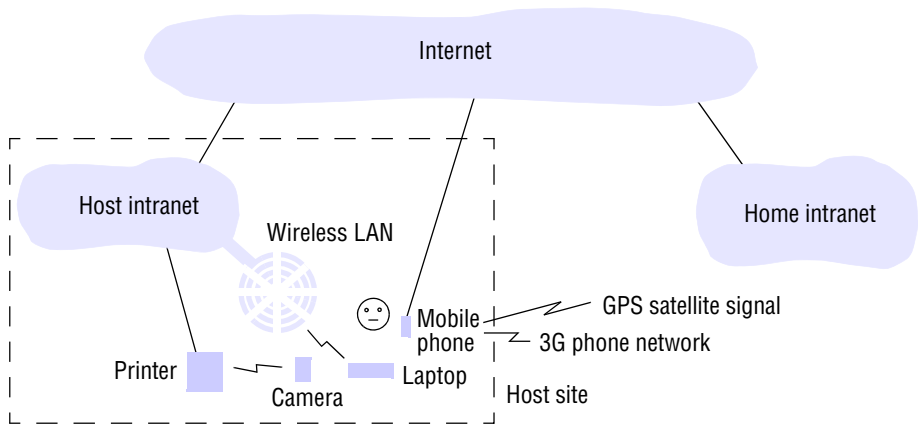
*Ubiquitous computing* is the harnessing of many small, cheap computational devices that are present in users’ physical environments, including the home, office and even natural settings. The term ‘ubiquitous’ is intended to suggest that small computing devices will eventually become so pervasive in 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.

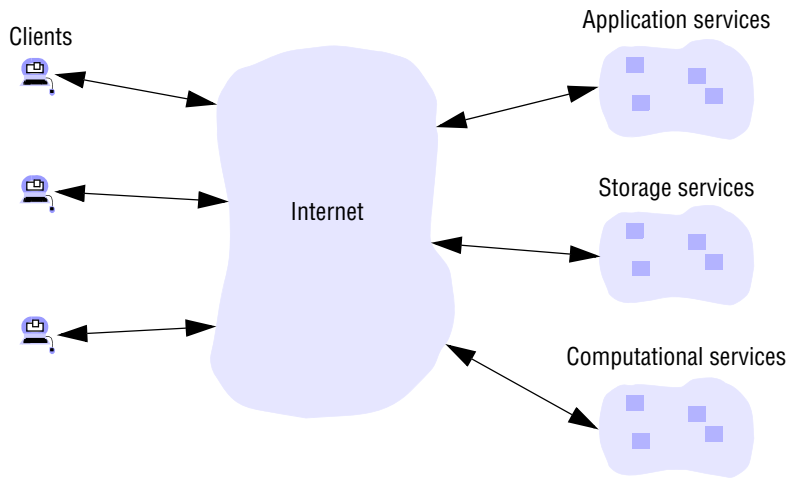
### 1.3.4 Distributed computing as a utility

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](http://www.google.com) 1]. 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](http://now.cs.berkeley.edu)] and Beowulf at NASA [[www.beowulf.org](http://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](http://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](http://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 reimplementing 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](http://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](http://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.

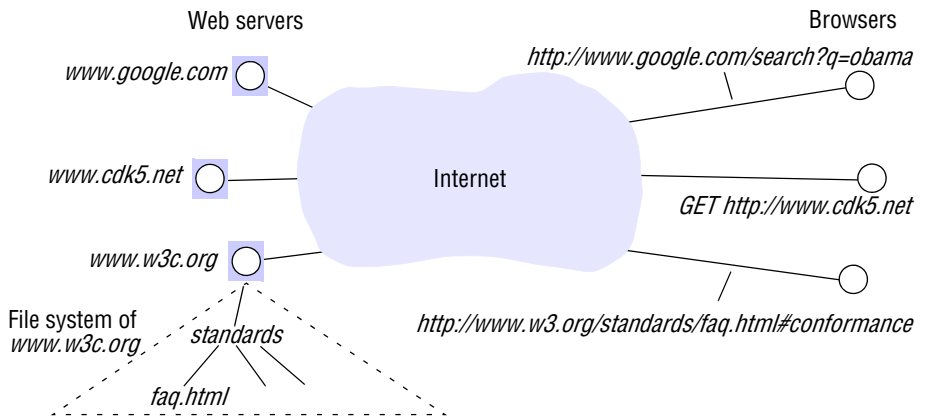
It is fundamental to the user's experience of the Web that when they encounter a given image or piece of text within a document, this will frequently be accompanied by links to related documents and other resources. The structure of links can be arbitrarily complex and the set of resources that can be 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](http://www.w3.org)] 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:

<code>&lt;IMG SRC = "http://www.cdk5.net/WebExample/Images/earth.jpg"&gt;</code>	1
<code>&lt;P&gt;</code>	2
<code>Welcome to Earth! Visitors may also be interested in taking a look at the</code>	3
<code>&lt;A HREF = "http://www.cdk5.net/WebExample/moon.html"&gt;Moon&lt;/A&gt;.</code>	4
<code>&lt;/P&gt;</code>	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](http://www.w3.org)] 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:

<i>Server DNS name</i>	<i>Path name</i>	<i>Query</i>	<i>Fragment</i>
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](http://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](http://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](http://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](http://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*

## SYSTEM MODELS

- 2.1 Introduction
- 2.2 Physical models
- 2.3 Architectural models
- 2.4 Fundamental models
- 2.5 Summary

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

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

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

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

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

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

## 2.1 Introduction

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

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

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

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

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

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

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

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

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

## 2.2 Physical models

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

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

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

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

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

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

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

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

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

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

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

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

## 2.3 Architectural models

---

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



**Figure 2.1** Generations of distributed systems

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

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

The section adopts a three-stage approach:

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

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

2.3.1 Architectural elements

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Indirect communication is discussed in more detail in Chapter 6.

Key techniques for indirect communication include:

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

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

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

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

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

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

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

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

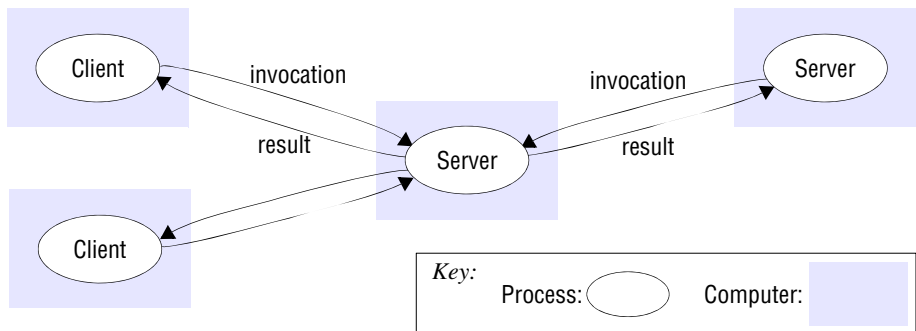
**Figure 2.2** Communicating entities and communication paradigms

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

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

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

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

**Figure 2.3** Clients invoke individual servers

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

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

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

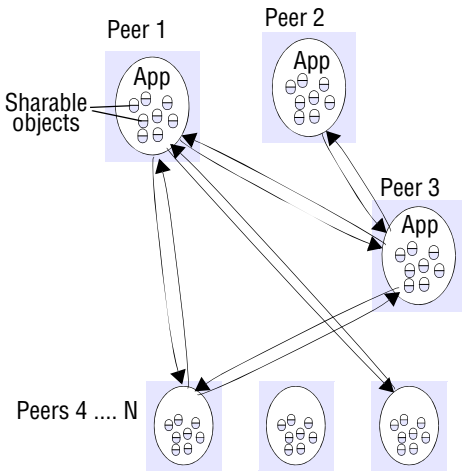
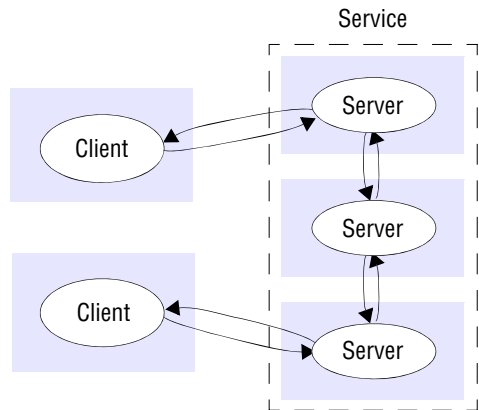
**Figure 2.4a** Peer-to-peer architecture**Figure 2.4b** A service provided by multiple servers

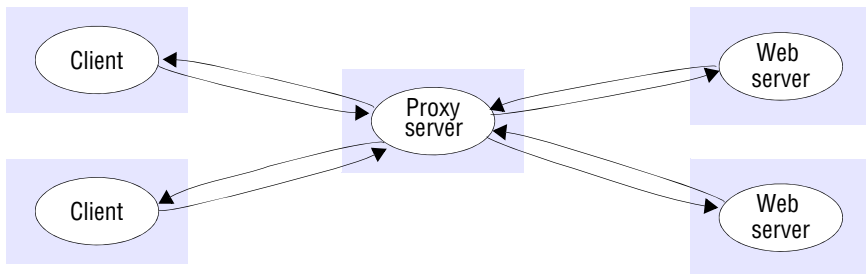
Figure 2.4a illustrates the form of a peer-to-peer application. Applications are composed of large numbers of peer processes running on separate computers and the pattern of communication between them depends entirely on application requirements. A large number of data objects are shared, an individual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

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

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

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



**Figure 2.5** Web proxy server

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

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

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

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

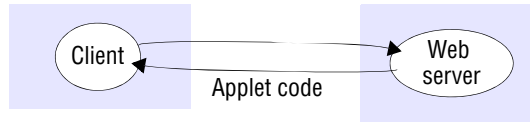
A more closely coupled type of multiple-server architecture is the cluster, as introduced in Chapter 1. A cluster is constructed from up to thousands of commodity processing boards, and service processing can be partitioned or replicated between them.

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

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

**Figure 2.6** Web applets

a) client request results in the downloading of applet code



b) client interacts with the applet



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

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

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

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

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

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

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

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

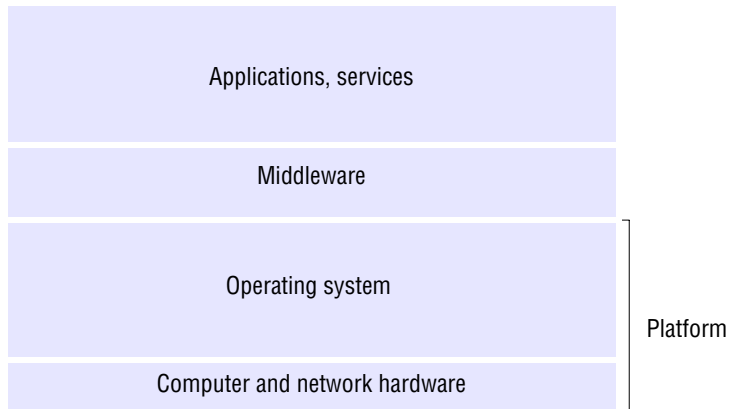
## 2.3.2 Architectural patterns

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

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

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

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

**Figure 2.7** Software and hardware service layers in distributed systems

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

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

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

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

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

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

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

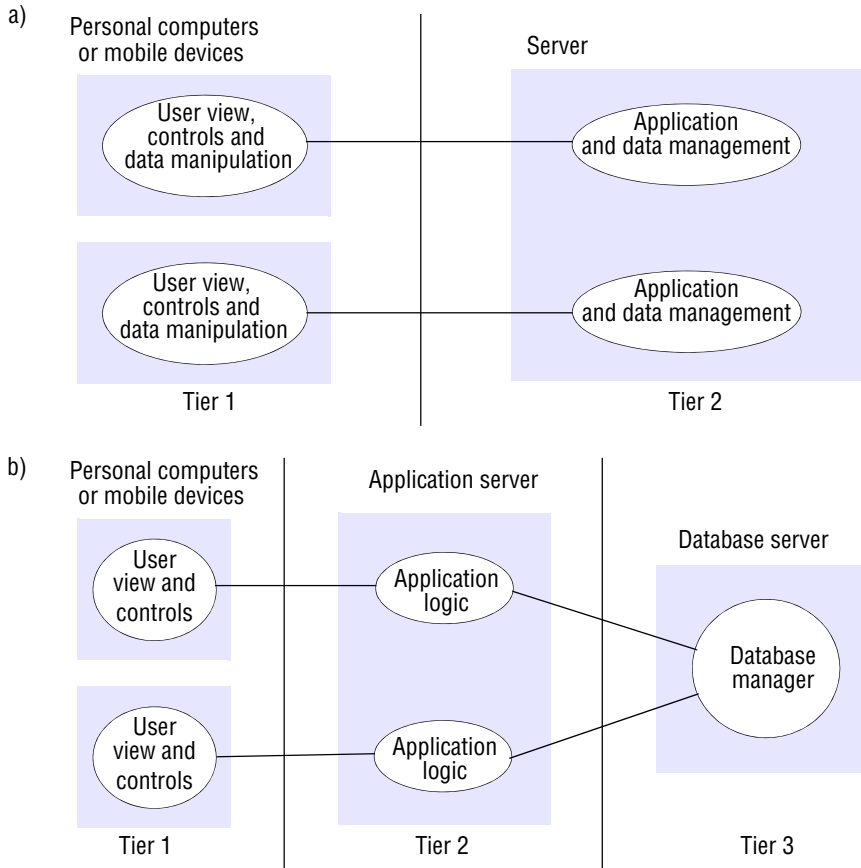
Now, let us consider the implementation of such an application using client-server technology. The associated two-tier and three-tier solutions are presented together for comparison in Figure 2.8 (a) and (b), respectively.

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

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

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

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

**Figure 2.8** Two-tier and three-tier architectures

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

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

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

**Figure 2.9** AJAX example: soccer score updates

---

```

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

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

```

---

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

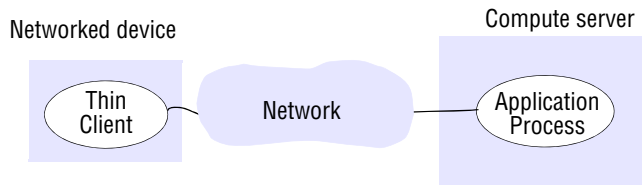
The introduction of Javascript, a cross-platform and cross-browser programming language that is downloaded and executed in the browser, constituted a first step towards the removal of those constraints. Javascript is a general-purpose language enabling both user interface and application logic to be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

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

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

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

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

**Figure 2.10** Thin clients and computer servers

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

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

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

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

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



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

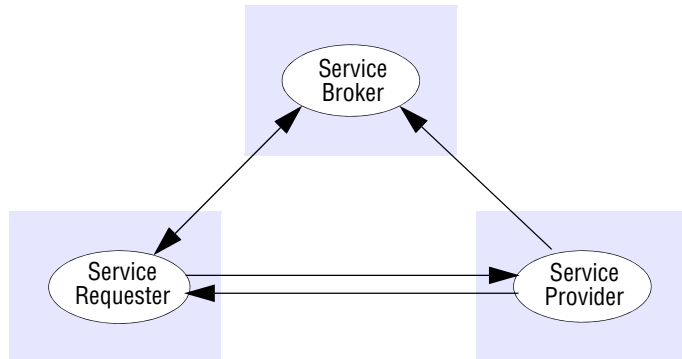
This concept has led to the emergence of *virtual network computing* (VNC). This technology was first introduced by researchers at the Olivetti and Oracle Research Laboratory [Richardson *et al.* 1998]; the initial concept has now evolved into implementations such as RealVNC [[www.realvnc.com](http://www.realvnc.com)],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

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

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

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

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

**Figure 2.11** The web service architectural pattern

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

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

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

### 2.3.3 Associated middleware solutions

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

**Figure 2.12** Categories of middleware

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 2.4 Fundamental models

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

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

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

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

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

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

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

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

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

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

### 2.4.1 Interaction model

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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



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

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

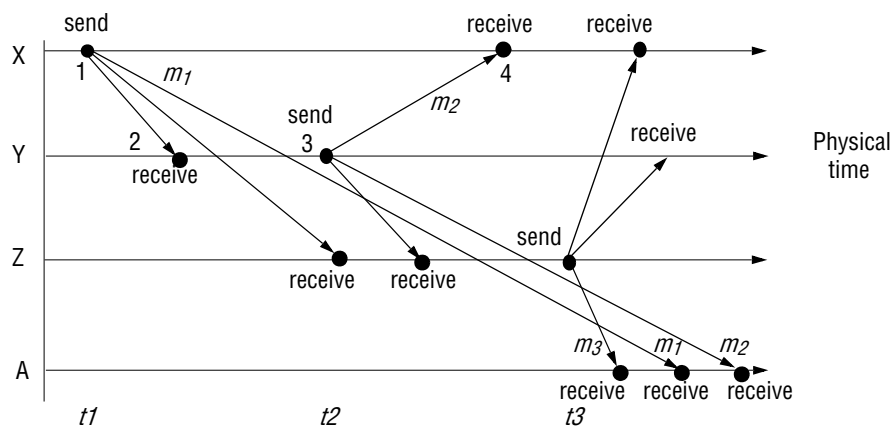
The asynchronous model allows no assumptions about the time intervals involved in any execution. This exactly models the Internet, in which there is no intrinsic bound on server or network load and therefore on how long it takes, for example, to transfer a file using FTP. Sometimes an email message can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

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

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

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

Figure 2.13 Real-time ordering of events



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

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

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

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

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

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

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

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

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

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

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

Y receives  $m_1$  before sending  $m_2$ .

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

## 2.4.2 Failure model

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

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

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

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

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

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

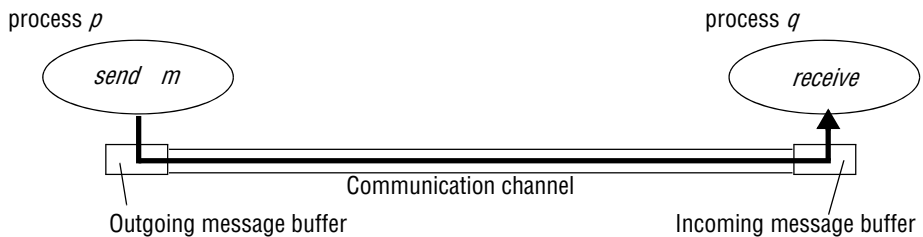
**Communication omission failures:** Consider the communication primitives *send* and *receive*. A process  $p$  performs a *send* by inserting the message  $m$  in its outgoing message buffer. The communication channel transports  $m$  to  $q$ 's incoming message buffer. Process  $q$  performs a *receive*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

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

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

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

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

**Figure 2.14** Processes and channels

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

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

**Failure detection** • In the case of the Pepperland divisions encamped at the tops of hills (see page 65), suppose that the Blue Meanies are after all sufficient in strength to attack and defeat either division while encamped – that is, that either can fail. Suppose further that, while undefeated, the divisions regularly send messengers to report their status. In an asynchronous system, neither division can distinguish whether the other has been defeated or the time it is taking for the messengers to cross the intervening valley is just very long. In a synchronous Pepperland, a division can tell for sure if the other has been defeated by the absence of a regular messenger. However, the other division may have been defeated just after it sent the latest messenger.

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

Figure 2.15 Omission and arbitrary failures

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

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

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

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

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

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

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

**Figure 2.16** Timing failures

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

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

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

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

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

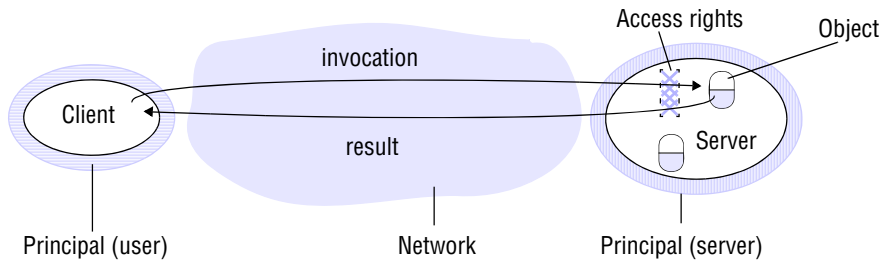
*Integrity:* The message received is identical to one sent, and no messages are delivered twice.

The threats to integrity come from two independent sources:

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

### 2.4.3 Security model

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

**Figure 2.17** Objects and principals

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

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

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

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

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

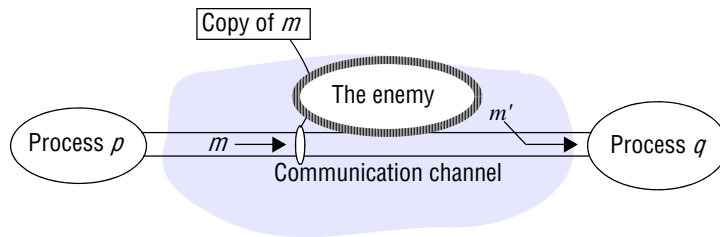
Thus we must include users in our model as the beneficiaries of access rights. We do so by associating with each invocation and each result the authority on which it is issued. Such an authority is called a *principal*. A principal may be a user or a process. In our illustration, the invocation comes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

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

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

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



**Figure 2.18** The enemy

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

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

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

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

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

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

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

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

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

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

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

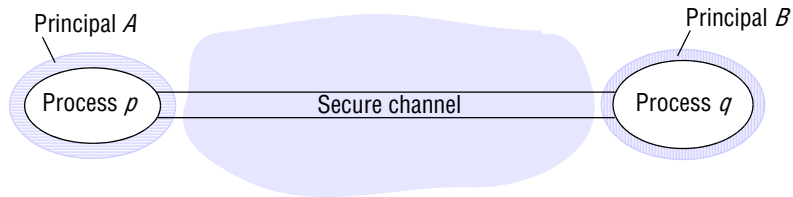
**Cryptography and shared secrets:** Suppose that a pair of processes (for example, a particular client and a particular server) share a secret; that is, they both know the secret but no other process in the distributed system knows it. Then if a message exchanged by that pair of processes includes information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

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

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

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

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

**Figure 2.19** Secure channels

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

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

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

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

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

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

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

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

## 2.5 Summary

---

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

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

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

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

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

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

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

## EXERCISES

- 2.1 Provide three specific and contrasting examples of the increasing levels of heterogeneity experienced in contemporary distributed systems as defined in Section 2.2. *page 39*
- 2.2 What problems do you foresee in the direct coupling between communicating entities that is implicit in remote invocation approaches? Consequently, what advantages do you anticipate from a level of decoupling as offered by space and time uncoupling? Note: you might want to revisit this answer after reading Chapters 5 and 6. *page 43*
- 2.3 Describe and illustrate the client-server architecture of one or more major Internet applications (for example, the Web, email or netnews). *page 46*
- 2.4 For the applications discussed in Exercise 2.1, what placement strategies are employed in implementing the associated services? *page 48*

- 2.5 A search engine is a web server that responds to client requests to search in its stored indexes and (concurrently) runs several web crawler tasks to build and update the indexes. What are the requirements for synchronization between these concurrent activities? *page 46*
- 2.6 The host computers used in peer-to-peer systems are often simply desktop computers in users' offices or homes. What are the implications of this for the availability and security of any shared data objects that they hold and to what extent can any weaknesses be overcome through the use of replication? *pages 47, 48*
- 2.7 List the types of local resource that are vulnerable to an attack by an untrusted program that is downloaded from a remote site and run in a local computer. *page 50*
- 2.8 Give examples of applications where the use of mobile code is beneficial. *page 50*
- 2.9 Consider a hypothetical car hire company and sketch out a three-tier solution to the provision of their underlying distributed car hire service. Use this to illustrate the benefits and drawbacks of a three-tier solution considering issues such as performance, scalability, dealing with failure and also maintaining the software over time. *page 52*
- 2.10 Provide a concrete example of the dilemma offered by Saltzer's end-to-end argument in the context of the provision of middleware support for distributed applications (you may want to focus on one aspect of providing dependable distributed systems, for example related to fault tolerance or security). *page 60*
- 2.11 Consider a simple server that carries out client requests without accessing other servers. Explain why it is generally not possible to set a limit on the time taken by such a server to respond to a client request. What would need to be done to make the server able to execute requests within a bounded time? Is this a practical option? *page 62*
- 2.12 For each of the factors that contribute to the time taken to transmit a message between two processes over a communication channel, state what measures would be needed to set a bound on its contribution to the total time. Why are these measures not provided in current general-purpose distributed systems? *page 63*
- 2.13 The Network Time Protocol service can be used to synchronize computer clocks. Explain why, even with this service, no guaranteed bound is given for the difference between two clocks. *page 64*
- 2.14 Consider two communication services for use in asynchronous distributed systems. In service A, messages may be lost, duplicated or delayed and checksums apply only to headers. In service B, messages may be lost, delayed or delivered too fast for the recipient to handle them, but those that are delivered arrive with the correct contents.  
Describe the classes of failure exhibited by each service. Classify their failures according to their effects on the properties of validity and integrity. Can service B be described as a reliable communication service? *page 67, page 71*
- 2.15 Consider a pair of processes X and Y that use the communication service B from Exercise 2.14 to communicate with one another. Suppose that X is a client and Y a server and that an *invocation* consists of a request message from X to Y, followed by Y carrying out the request, followed by a reply message from Y to X. Describe the classes of failure that may be exhibited by an invocation. *page 67*

- 2.16 Suppose that a basic disk read can sometimes read values that are different from those written. State the type of failure exhibited by a basic disk read. Suggest how this failure may be masked in order to produce a different benign form of failure. Now suggest how to mask the benign failure. *page 70*
- 2.17 Define the integrity property of reliable communication and list all the possible threats to integrity from users and from system components. What measures can be taken to ensure the integrity property in the face of each of these sources of threats. *pages 71, 74*
- 2.18 Describe possible occurrences of each of the main types of security threat (threats to processes, threats to communication channels, denial of service) that might occur in the Internet. *pages 74, 75*

## NETWORKING AND INTERNETWORKING

- 3.1 Introduction
- 3.2 Types of network
- 3.3 Network principles
- 3.4 Internet protocols
- 3.5 Case studies: Ethernet, WiFi and Bluetooth
- 3.6 Summary

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

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

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



## 3.1 Introduction

---

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

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

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

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

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

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

### 3.1.1 Networking issues for distributed systems

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 3.2 Types of network

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

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

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

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

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

**Figure 3.1** Network performance

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

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

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

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

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

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

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

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

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

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

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

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

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

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



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

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

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

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

## 3.3 Network principles

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

### 3.3.1 Packet transmission

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



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

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

### 3.3.2 Data streaming

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

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

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

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

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

requirements in each packet. The availability of effective implementations of these protocols in the general Internet will depend upon substantial changes to the transport and network layers. Chapter 20 discusses the needs of distributed multimedia applications in more detail.

### 3.3.3 Switching schemes

A network consists of a set of nodes connected together by circuits. To transmit information between two arbitrary nodes, a switching system is required. Here we define the four types of switching that are used in computer networking.

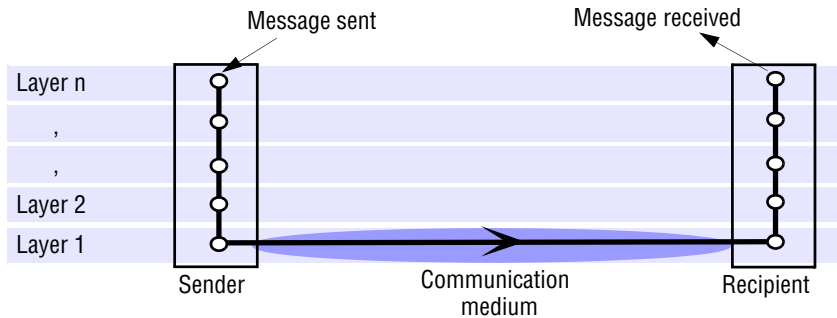
**Broadcast** • Broadcasting is a transmission technique that involves no switching. Everything is transmitted to every node, and it is up to potential receivers to notice transmissions addressed to them. Some LAN technologies, including Ethernet, are based on broadcasting. Wireless networking is necessarily based on broadcasting, but in the absence of fixed circuits the broadcasts are arranged to reach nodes grouped in *cells*.

**Circuit switching** • At one time telephone networks were the only telecommunication networks. Their operation was simple to understand: when a caller dialed a number, the pair of wires from her phone to the local exchange was connected by an automatic switch at the exchange to the pair of wires connected to the other party's phone. For a long-distance call the process was similar but the connection would be switched through a number of intervening exchanges to its destination. This system is sometimes referred to as the *plain old telephone system*, or POTS. It is a typical *circuit-switching network*.

**Packet switching** • The advent of computers and digital technology brought many new possibilities for telecommunication. At the most basic level, it brought processing and storage. These made it possible to construct a different kind of communication network called a *store-and-forward network*. Instead of making and breaking connections to build circuits, a store-and-forward network just forwards packets from their source to their destination. There is a computer at each switching node (that is, wherever several circuits need to be interconnected). Each packet arriving at a node is first stored in memory at the node and then processed by a program that transmits it on an outgoing circuit, which transfers the packet to another node that is closer to its ultimate destination.

There is nothing really new in this idea: the postal system is a store-and-forward network for letters, with the processing done by humans or machinery at sorting offices. But in a computer network packets can be stored and processed fast enough to give an illusion of instantaneous transmission, even though the packet has to be routed through many nodes.

**Frame relay** • In reality, it takes anything from a few tens of microseconds to a few milliseconds to switch a packet through each network node in a store-and-forward network. This switching delay depends on the packet size, hardware speed and quantity of other traffic, but its lower bound is determined by the network bandwidth, since the entire packet must be received before it can be forwarded to another node. Much of the Internet is based on store-and-forward switching, and as we have already seen, even short Internet packets typically take up to 200 milliseconds to reach their destinations. Delays of this magnitude are too long for real-time applications such as telephony and

**Figure 3.2** Conceptual layering of protocol software

video conferencing, where delays of less than 50 milliseconds are needed to sustain high-quality conversation.

The *frame relay* switching method brings some of the advantages of circuit switching to packet-switching networks. They overcome the delay problems by switching small packets (called *frames*) on the fly. The switching nodes (which are usually special-purpose parallel digital processors) route frames based on the examination of their first few bits; frames as a whole are not stored at nodes but pass through them as short streams of bits. ATM networks are a prime example; high-speed ATM networks can transmit packets across networks consisting of many nodes in a few tens of microseconds.

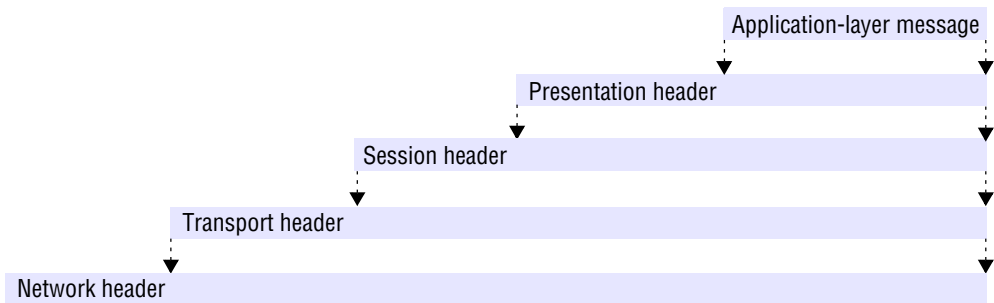
### 3.3.4 Protocols

The term *protocol* is used to refer to a well-known set of rules and formats to be used for communication between processes in order to perform a given task. The definition of a protocol has two important parts to it:

- a specification of the sequence of messages that must be exchanged;
- a specification of the format of the data in the messages.

The existence of well-known protocols enables the separate software components of distributed systems to be developed independently and implemented in different programming languages on computers that may have different order codes and data representations.

A protocol is implemented by a pair of software modules located in the sending and receiving computers. For example, a *transport protocol* transmits messages of any length from a sending process to a receiving process. A process wishing to transmit a message to another process issues a call to a transport protocol module, passing it a message in the specified format. The transport software then concerns itself with the transmission of the message to its destination, subdividing it into packets of some specified size and format that can be transmitted to the destination via the *network protocol* – another, lower-level protocol. The corresponding transport protocol module in the receiving computer receives the packet via the network-level protocol module and

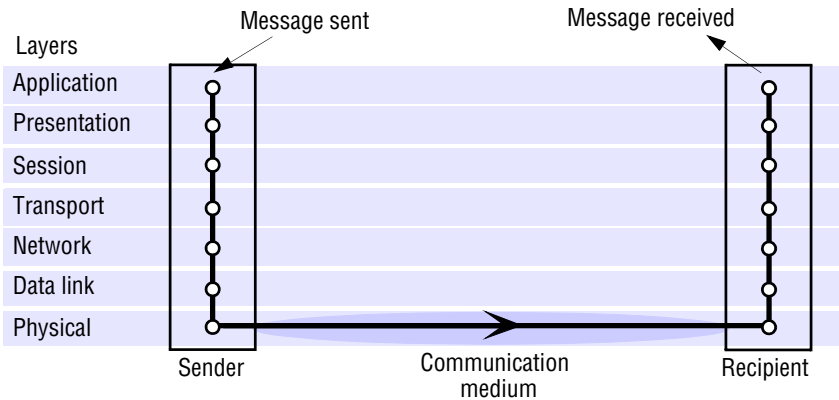
**Figure 3.3** Encapsulation as it is applied in layered protocols

performs inverse transformations to regenerate the message before passing it to a receiving process.

**Protocol layers** • Network software is arranged in a hierarchy of layers. Each layer presents an interface to the layers above it that extends the properties of the underlying communication system. A layer is represented by a module in every computer connected to the network. Figure 3.2 illustrates the structure and the flow of data when a message is transmitted using a layered protocol. Each module appears to communicate directly with a module at the same level in another computer in the network, but in reality data is not transmitted directly between the protocol modules at each level. Instead, each layer of network software communicates by local procedure calls with the layers above and below it.

On the sending side, each layer (except the topmost, or *application layer*) accepts items of data in a specified format from the layer above it and applies transformations to encapsulate the data in the format specified for that layer before passing it to the layer below for further processing. Figure 3.3 illustrates this process as it applies to the top four layers of the OSI protocol suite (discussed in the next subsection). The figure shows the packet headers that hold most network-related data items, but for clarity it omits the trailers that are present in some types of packet; it also assumes that the application-layer message to be transmitted is shorter than the underlying network's maximum packet size. If not, it would have to be encapsulated in several network-layer packets. On the receiving side, the converse transformations are applied to data items received from the layer below before they are passed to the layer above. The protocol type of the layer above is included in the header of each layer, to enable the protocol stack at the receiver to select the correct software components to unpack the packets.

Thus each layer provides a service to the layer above it and extends the service provided by the layer below it. At the bottom is a *physical layer*. This is implemented by a communication medium (copper or fibre optic cables, satellite communication channels or radio transmission) and by analogue signalling circuits that place signals on the communication medium at the sending node and sense them at the receiving node. At receiving nodes data items are received and passed upwards through the hierarchy of software modules, being transformed at each stage until they are in a form that can be passed to the intended recipient process.

**Figure 3.4** Protocol layers in the ISO Open Systems Interconnection (OSI) protocol model

**Protocol suites** • A complete set of protocol layers is referred to as a *protocol suite* or a *protocol stack*, reflecting the layered structure. Figure 3.4 shows a protocol stack that conforms to the seven-layer Reference Model for *Open Systems Interconnection* (OSI) adopted by the International Organization for Standardization (ISO) [ISO 1992]. The OSI Reference Model was adopted in order to encourage the development of protocol standards that would meet the requirements of open systems.

The purpose of each level in the OSI Reference Model is summarized in Figure 3.5. As its name implies, it is a framework for the definition of protocols and not a definition for a specific suite of protocols. Protocol suites that conform to the OSI model must include at least one specific protocol at each of the seven levels that the model defines.

Protocol layering brings substantial benefits in simplifying and generalizing the software interfaces for access to the communication services of networks, but it also carries significant performance costs. The transmission of an application-level message via a protocol stack with  $N$  layers typically involves  $N$  transfers of control to the relevant layer of software in the protocol suite, at least one of which is an operating system entry, and taking  $N$  copies of the data as a part of the encapsulation mechanism. All of these overheads result in data transfer rates between application processes that are much lower than the available network bandwidth.

Figure 3.5 includes examples from protocols used in the Internet, but the implementation of the Internet does not follow the OSI model in two respects. First, the application, presentation and session layers are not clearly distinguished in the Internet protocol stack. Instead, the application and presentation layers are implemented either as a single middleware layer or separately within each application. Thus CORBA implements inter-object invocations and data representations in a middleware library that is included in each application process (see Chapter 8 for further details on CORBA). Web browsers and other applications that require secure channels employ the Secure Sockets Layer (Chapter 11) as a procedure library in a similar manner.

Second, the session layer is integrated with the transport layer. Internetwork protocol suites include an application layer, a transport layer and an *internetwork layer*.

**Figure 3.5** OSI protocol summary

<i>Layer</i>	<i>Description</i>	<i>Examples</i>
Application	Protocols at this level are designed to meet the communication requirements of specific applications, often defining the interface to a service.	HTTP, FTP, SMTP, CORBA IIOP
Presentation	Protocols at this level transmit data in a network representation that is independent of the representations used in individual computers, which may differ. Encryption is also performed in this layer, if required.	TLS security, CORBA data representation
Session	At this level reliability and adaptation measures are performed, such as detection of failures and automatic recovery.	SIP
Transport	This is the lowest level at which messages (rather than packets) are handled. Messages are addressed to communication ports attached to processes. Protocols in this layer may be connection-oriented or connectionless.	TCP, UDP
Network	Transfers data packets between computers in a specific network. In a WAN or an internetwork this involves the generation of a route passing through routers. In a single LAN no routing is required.	IP, ATM virtual circuits
Data link	Responsible for transmission of packets between nodes that are directly connected by a physical link. In a WAN transmission is between pairs of routers or between routers and hosts. In a LAN it is between any pair of hosts.	Ethernet MAC, ATM cell transfer, PPP
Physical	The circuits and hardware that drive the network. It transmits sequences of binary data by analogue signalling, using amplitude or frequency modulation of electrical signals (on cable circuits), light signals (on fibre optic circuits) or other electromagnetic signals (on radio and microwave circuits).	Ethernet base-band signalling, ISDN

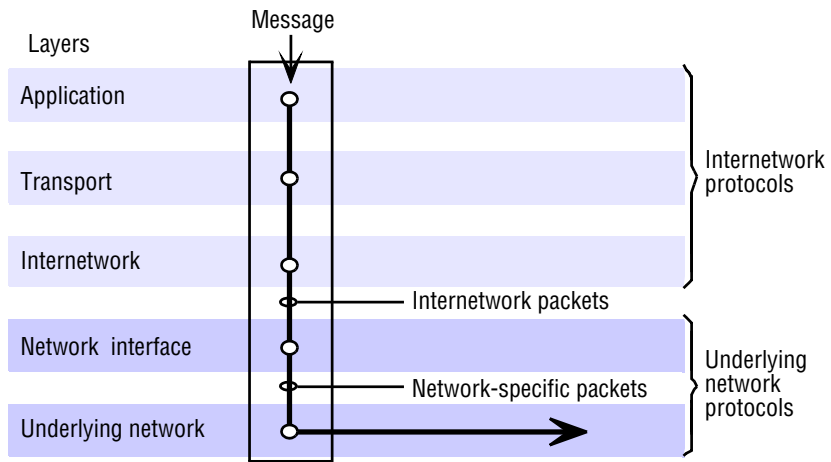
The internetwork layer is a ‘virtual’ network layer that is responsible for transmitting internetwork packets to a destination computer. An *internetwork packet* is the unit of data transmitted over an internetwork.

Internetwork protocols are overlaid on underlying networks as illustrated in Figure 3.6. The *network interface* layer accepts internetwork packets and converts them into packets suitable for transmission by the network layers of each underlying network.

**Packet assembly** • The task of dividing messages into packets before transmission and reassembling them at the receiving computer is usually performed in the transport layer.

The network-layer protocol packets consist of a *header* and a *data field*. In most network technologies, the data field is variable in length, with the maximum length called the *maximum transfer unit* (MTU). If the length of a message exceeds the MTU of the underlying network layer, it must be fragmented into chunks of the appropriate size, with sequence numbers for use on reassembly, and transmitted in multiple packets. For example, the MTU for Ethernets is 1500 bytes – no more than that quantity of data can be transmitted in a single Ethernet packet.

Although the IP protocol stands in the position of a network-layer protocol in the Internet suite of protocols, its MTU is unusually large at 64 kbytes (8 kbytes is often used in practice because some nodes are unable to handle such large packets).

**Figure 3.6** Internetwork layers

Whichever MTU value is adopted for IP packets, packets larger than the Ethernet MTU can arise and they must be fragmented for transmission over Ethernets.

**Ports** • The transport layer’s task is to provide a network-independent message transport service between pairs of network *ports*. Ports are software-defined destination points at a host computer. They are attached to processes, enabling data transmission to be addressed to a specific process at a destination node. Next, we discuss the addressing of ports as they are implemented in the Internet and most other networks. Chapter 4 describes their programming.

**Addressing** • The transport layer is responsible for delivering messages to destinations with *transport addresses* that are composed of the *network address* of a host computer and a *port number*. A network address is a numeric identifier that uniquely identifies a host computer and enables it to be located by nodes that are responsible for routing data to it. In the Internet every host computer is assigned an IP number, which identifies it and the subnet to which it is connected, enabling data to be routed to it from any other node (as described in the following sections). In Ethernets there are no routing nodes; each host is responsible for recognizing and picking up packets addressed to it.

Well-known Internet services such as HTTP and FTP have been allocated *contact port numbers* and these are registered with a central authority (the Internet Assigned Numbers Authority (IANA) [[www.iana.org](http://www.iana.org) I]). To access a service at a given host, a request is sent to the relevant port at the host. Some services, such as FTP (contact port: 21), then allocate a new port (with a private number) and send the number of the new port to the client. The client uses the new port for the remainder of a transaction or a session. Other services, such as HTTP (contact port: 80), transact all of their business through the contact port.

Port numbers below 1023 are defined as *well-known ports* whose use is restricted to privileged processes in most operating systems. The ports between 1024 and 49151 are *registered ports* for which IANA holds service descriptions, and the remaining ports up to 65535 are available for private purposes. In practice, all of the ports above 1023

can be used for private purposes, but computers using them for private purposes cannot simultaneously access the corresponding registered services.

A fixed port number allocation does not provide an adequate basis for the development of distributed systems which often include a multiplicity of servers including dynamically allocated ones. Solutions to this problem involve the dynamic allocation of ports to services and the provision of binding mechanisms to enable clients to locate services and their ports using symbolic names. Some of these are discussed further in Chapter 5.

**Packet delivery** • There are two approaches to the delivery of packets by the network layer:

*Datagram packet delivery:* The term ‘datagram’ refers to the similarity of this delivery mode to the way in which letters and telegrams are delivered. The essential feature of datagram networks is that the delivery of each packet is a ‘one-shot’ process; no setup is required, and once the packet is delivered the network retains no information about it. In a datagram network a sequence of packets transmitted by a single host to a single destination may follow different routes (if, for example, the network is capable of adaptation to handle failures or to mitigate the effects of localized congestion), and when this occurs they may arrive out of sequence.

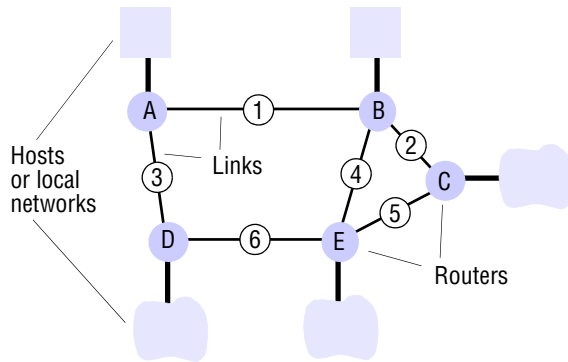
Every datagram packet contains the full network address of the source and destination hosts; the latter is an essential parameter for the routing process, which we describe in the next section. Datagram delivery is the concept on which packet networks were originally based, and it can be found in most of the computer networks in use today. The Internet’s network layer (IP), Ethernet and most wired and wireless local network technologies are based on datagram delivery.

*Virtual circuit packet delivery:* Some network-level services implement packet transmission in a manner that is analogous to a telephone network. A virtual circuit must be set up before packets can pass from a source host A to destination host B. The establishment of a virtual circuit involves the identification of a route from the source to the destination, possibly passing through several intermediate nodes. At each node along the route a table entry is made, indicating which link should be used for the next stage of the route.

Once a virtual circuit has been set up, it can be used to transmit any number of packets. Each network-layer packet contains only a virtual circuit number in place of the source and destination addresses. The addresses are not needed, because packets are routed at intermediate nodes by reference to the virtual circuit number. When a packet reaches its destination the source can be determined from the virtual circuit number.

The analogy with telephone networks should not be taken too literally. In the POTS a telephone call results in the establishment of a physical circuit from the caller to the callee, and the voice links from which it is constructed are reserved for their exclusive use. In virtual circuit packet delivery the circuits are represented only by table entries in routing nodes, and the links along which the packets are routed are used only for the time taken to transmit a packet; they are free for other uses for the rest of the time. A single link may therefore be employed in many separate virtual circuits. The most important virtual circuit network technology in current use is



**Figure 3.7** Routing in a wide area network

ATM; we have already mentioned (in Section 3.3.3) that it benefits from lower latencies for the transmission of individual packets; this is a direct result of its use of virtual circuits. The requirement for a setup phase does, however, result in a short delay before any packets can be sent to a new destination.

The distinction between datagram and virtual circuit packet delivery in the network layer should not be confused with a similarly named pair of mechanisms in the transport layer: connectionless and connection-oriented transmission. We describe these in Section 3.4.6 in the context of the Internet transport protocols, UDP (connectionless) and TCP (connection-oriented). Here we simply note that each of these modes of transmission can be implemented over either type of network layer.

### 3.3.5 Routing

Routing is a function that is required in all networks except those LANs, such as Ethernet, that provide direct connections between all pairs of attached hosts. In large networks, *adaptive routing* is employed: the best route for communication between two points in the network is re-evaluated periodically, taking into account the current traffic in the network and any faults such as broken connections or routers.

The delivery of packets to their destinations in a network such as the one shown in Figure 3.7 is the collective responsibility of the routers located at connection points. Unless the source and destination hosts are on the same LAN, the packet has to be transmitted in a series of hops, passing through router nodes. The determination of routes for the transmission of packets to their destinations is the responsibility of a *routing algorithm* implemented by a program in the network layer at each node.

A routing algorithm has two parts:

1. It must make decisions that determine the route taken by each packet as it travels through the network. In circuit-switched network layers such as X.25 and frame-relay networks such as ATM, the route is determined whenever a virtual circuit or connection is established. In packet-switched network layers such as IP it is

**Figure 3.8** Routing tables for the network in Figure 3.7

Routings from A			Routings from B			Routings from C		
To	Link	Cost	To	Link	Cost	To	Link	Cost
A	local	0	A	1	1	A	2	2
B	1	1	B	local	0	B	2	1
C	1	2	C	2	1	C	local	0
D	3	1	D	1	2	D	5	2
E	1	2	E	4	1	E	5	1

Routings from D			Routings from E		
To	Link	Cost	To	Link	Cost
A	3	1	A	4	2
B	3	2	B	4	1
C	6	2	C	5	1
D	local	0	D	6	1
E	6	1	E	local	0

determined separately for each packet, and the algorithm must be particularly simple and efficient if it is not to degrade network performance.

2. It must dynamically update its knowledge of the network based on traffic monitoring and the detection of configuration changes or failures. This activity is less time-critical; slower and more computation-intensive techniques can be used.

Both of these activities are distributed throughout the network. The routing decisions are made on a hop-by-hop basis, using locally held information to determine the next hop to be taken by each incoming packet. The locally held routing information is updated periodically by an algorithm that distributes information about the states of the links (their loads and failure status).

**A simple routing algorithm** • The algorithm that we describe here is a ‘distance vector’ algorithm. This will provide a basis for the discussion in Section 3.4.3 of the *link-state* algorithm that has been used since 1979 as the main routing algorithm in the Internet. Routing in networks is an instance of the problem of path finding in graphs. Bellman’s shortest path algorithm, published well before computer networks were developed [Bellman 1957], provides the basis for the distance vector method. Bellman’s method was converted into a distributed algorithm suitable for implementation in large networks by Ford and Fulkerson [1962], and protocols based on their work are often referred to as ‘Bellman–Ford’ protocols.

Figure 3.8 shows the routing tables that would be held at each of the routers for the network of Figure 3.7, assuming that the network has no failed links or routers. Each row provides the routing information for packets addressed to a given destination. The *link* field specifies the outgoing link for packets addressed to the destination. The *cost*

field is simply a calculation of the vector distance, or the number of hops to the given destination. For store-and-forward networks with links of similar bandwidth, this gives a reasonable estimate of the time for a packet to travel to the destination. The cost information stored in the routing tables is not used during packet-routing actions taken by part 1 of the routing algorithm, but it is required for the routing table construction and maintenance actions in part 2.

The routing tables contain a single entry for each possible destination, showing the next *hop* that a packet must take towards its destination. When a packet arrives at a router the destination address is extracted and looked up in the local routing table. The resulting entry in the routing table identifies the outgoing link that should be used to route the packet onwards towards its destination.

For example, when a packet addressed to C is submitted to the router at A, the router examines the entry for C in its routing table. It shows that the packet should be routed outwards from A on the link labelled 1. The packet arrives at B and the same procedure is followed using the routing table at B, which shows that the onward route to C is via the link labelled 2. When the packet arrives at C the routing table entry shows 'local' instead of a link number. This indicates that the packet should be delivered to a local host.

Now let us consider how the routing tables are built up and how they are maintained when faults occur in the network – that is, how part 2 of the routing algorithm described above is performed. Because each routing table specifies only a single hop for each route, the construction or repair of the routing information can proceed in a distributed fashion. A router exchanges information about the network with its neighbouring nodes by sending a summary of its routing table using a *router information protocol* (RIP). The RIP actions performed at a router are described informally as follows:

1. *Periodically, and whenever the local routing table changes*, send the table (in a summary form) to all accessible neighbours. That is, send an RIP packet containing a copy of the table on each non-faulty outgoing link.
2. *When a table is received from a neighbouring router*, if the received table shows a route to a new destination, or a better (lower-cost) route to an existing destination, update the local table with the new route. If the table was received on link  $n$  and it gives a different cost than the local table for a route that begins with link  $n$ , replace the cost in the local table with the new cost. This is done because the new table was received from a router that is closer to the relevant destination and is therefore always more authoritative for routes that pass through it.

This algorithm is more precisely described by the pseudo-code program shown in Figure 3.9, where  $Tr$  is a table received from another router and  $Tl$  is the local table. Ford and Fulkerson [1962] have shown that the steps described above are sufficient to ensure that the routing tables will converge on the best routes to each destination whenever there is a change in the network. The frequency  $t$  with which routing tables are propagated, even when no changes have occurred, is designed to ensure that stability is maintained, for example, in the case that some RIP packets are lost. The value for  $t$  adopted throughout the Internet is 30 seconds.

**Figure 3.9** Pseudo-code for RIP routing algorithm

*Send:* Each  $t$  seconds or when  $Tl$  changes, send  $Tl$  on each non-faulty outgoing link.

*Receive:* Whenever a routing table  $Tr$  is received on link  $n$ :

```

for all rows  $Rr$  in  $Tr$  {
  if ( $Rr.link \neq n$ ) {
     $Rr.cost = Rr.cost + 1$ ;
     $Rr.link = n$ ;
    if ( $Rr.destination$  is not in  $Tl$ ) add  $Rr$  to  $Tl$ ; // add new destination to  $Tl$ 
    else for all rows  $Rl$  in  $Tl$  {
      if ( $Rr.destination = Rl.destination$  and
          ( $Rr.cost < Rl.cost$  or  $Rl.link = n$ ))  $Rl = Rr$ ;
      //  $Rr.cost < Rl.cost$  : remote node has better route
      //  $Rl.link = n$  : remote node is more authoritative
    }
  }
}

```

To deal with faults, each router monitors its links and acts as follows:

When a faulty link  $n$  is detected, set  $cost$  to  $\infty$  for all entries in the local table that refer to the faulty link and perform the *Send* action.

Thus the information that the link is broken is represented by an infinite value for the cost to the relevant destinations. When this information is propagated to neighbouring nodes it will be processed according to the *Receive* action (note  $\infty+1 = \infty$ ) and then propagated further until a node is reached that has a working route to the relevant destinations, if one exists. The node that still has a working route will eventually propagate its table, and the working route will replace the faulty one at all nodes.

The vector-distance algorithm can be improved in various ways: costs (also known as the *metric*) can be based on the actual bandwidths of the links and the algorithm can be modified to increase its speed of convergence and to avoid some undesirable intermediate states, such as loops, that may occur before convergence is achieved. A routing information protocol with these enhancements was the first routing protocol used in the Internet, now known as RIP-1 and described in RFC 1058 [Hedrick 1988]. But the solutions for the problems caused by slow convergence are not totally effective, and this leads to inefficient routing and packet loss while the network is in intermediate states.

Subsequent developments in routing algorithms have been in the direction of increasing the amount of knowledge of the network that is held at each node. The most important family of algorithms of this type are known as *link-state algorithms*. They are based on the distribution and updating of a database at each node that represents all, or a substantial portion, of the network. Each node is then responsible for computing the optimum routes to the destinations shown in its database. This computation can be performed by a variety of algorithms, some of which avoid known problems in the Bellman–Ford algorithm such as slow convergence and undesirable intermediate states. The design of routing algorithms is a substantial topic, and our discussion of it here is

necessarily limited. We return to it in Section 3.4.3 with a description of the operation of the RIP-1 algorithm, one of the first used for IP routing and still in use in many parts of the Internet. For extensive coverage of routing in the Internet, see Huitema [2000], and for further material on routing algorithms in general see Tanenbaum [2003].

### 3.3.6 Congestion control

The capacity of a network is limited by the performance of its communication links and switching nodes. When the load at any particular link or node approaches its capacity, queues will build up at hosts trying to send packets and at intermediate nodes holding packets whose onward transmission is blocked by other traffic. If the load continues at the same high level, the queues will continue to grow until they reach the limit of available buffer space.

Once this state is reached at a node, the node has no option but to drop further incoming packets. As we have already noted, the occasional loss of packets at the network level is acceptable and can be remedied by retransmission initiated at higher levels. But if the rate of packet loss and retransmission reaches a substantial level, the effect on the throughput of the network can be devastating. It is easy to see why this is the case: if packets are dropped at intermediate nodes, the network resources that they have already consumed are wasted and the resulting retransmissions will require a similar quantity of resources to reach the same point in the network. As a rule of thumb, when the load on a network exceeds 80% of its capacity, the total throughput tends to drop as a result of packet losses unless usage of heavily loaded links is controlled.

Instead of allowing packets to travel through the network until they reach over-congested nodes, where they will have to be dropped, it would be better to hold them at earlier nodes until the congestion is reduced. This will result in increased delays for packets but will not significantly degrade the total throughput of the network. *Congestion control* is the name given to techniques that are designed to achieve this.

In general, congestion control is achieved by informing nodes along a route that congestion has occurred and that their rate of packet transmission should therefore be reduced. For intermediate nodes, this will result in the buffering of incoming packets for a longer period. For hosts that are sources of the packets, the result may be to queue packets before transmission or to block the application process that is generating them until the network can handle them.

All datagram-based network layers, including IP and Ethernets, rely on the end-to-end control of traffic. That is, the sending node must reduce the rate at which it transmits packets based only on information that it receives from the receiver. Congestion information may be supplied to the sending node by explicit transmission of special messages (called *choke packets*) requesting a reduction in transmission rate, by the implementation of a specific transmission control protocol (from which TCP derives its name – Section 3.4.6 explains the mechanism used in TCP) or by observing the occurrence of dropped packets (if the protocol is one in which each packet is acknowledged).

In some virtual circuit based networks, congestion information can be received and acted on at each node. Although ATM uses virtual circuit delivery, it relies on quality of service management (see Chapter 20) to ensure that each circuit can carry the required traffic.

### 3.3.7 Internetworking

There are many network technologies with different network-, link- and physical-layer protocols. Local networks are built with Ethernet technologies, while wide area networks are built over analogue and digital telephone networks of various types, satellite links and wide area ATM networks. Individual computers and local networks are linked to the Internet or intranets by modems and by wireless and DSL connections.

To build an integrated network (an *internetwork*) we must integrate many subnets, each of which is based on one of these network technologies. To make this possible, the following are needed:

1. a unified internetwork addressing scheme that enables packets to be addressed to any host connected to any subnet;
2. a protocol defining the format of internetwork packets and giving rules according to which they are handled;
3. interconnecting components that route packets to their destinations in terms of internetwork addresses, transmitting the packets using subnets with a variety of network technologies.

For the Internet, (1) is provided by IP addresses, (2) is the IP protocol and (3) is performed by components called *Internet routers*. The IP protocol and IP addressing are described in some detail in Section 3.4. Here we describe the functions of Internet routers and some other components that are used to link networks together.

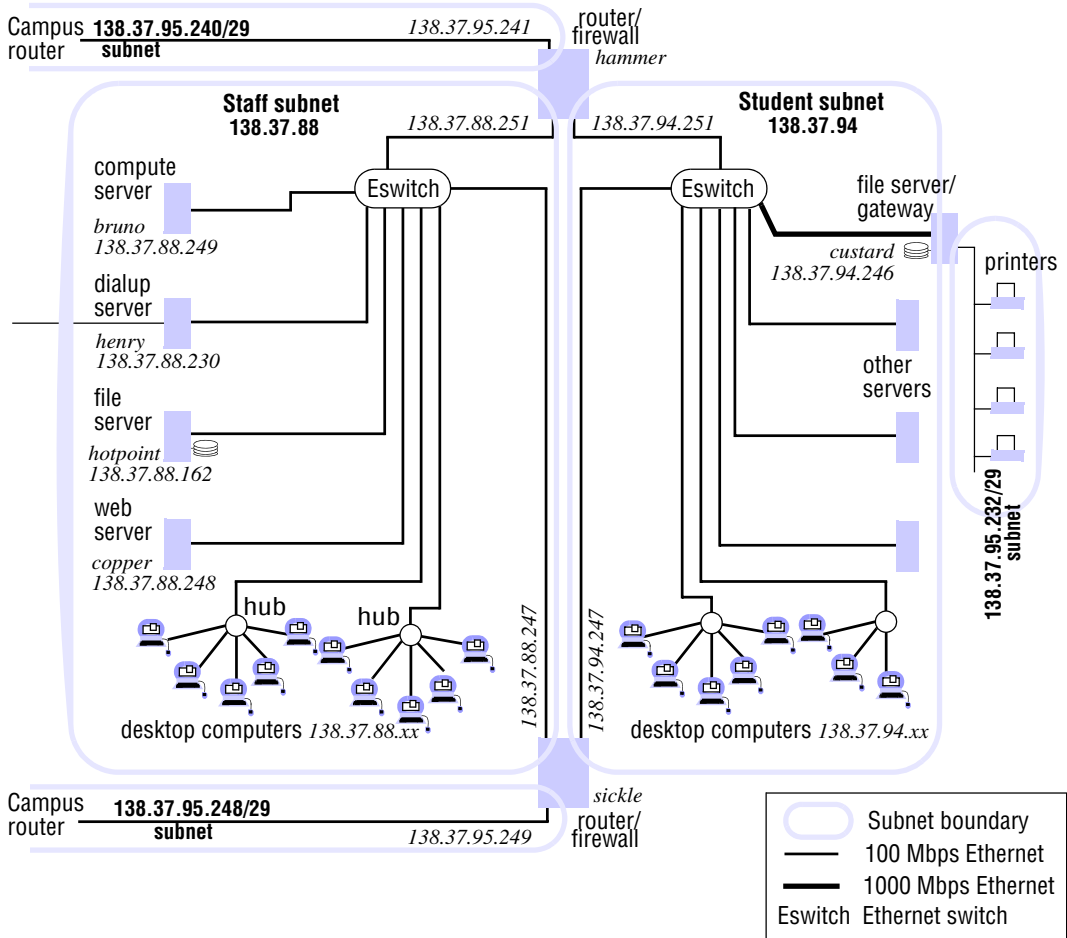
Figure 3.10 shows a small part of the campus intranet at a British university. Many of the details shown will be explained in later sections. Here we note that the portion shown in the figure comprises several subnets interconnected by routers. There are five subnets, three of which share the IP network 138.37.95 (using the classless interdomain routing scheme described in Section 3.4.3). The numbers in the diagram are IP addresses; their structure will be explained in Section 3.4.1. The routers in the diagram are members of multiple subnets and have an IP address for each subnet, shown against the connecting links.

The routers (hostnames: *hammer* and *sickle*) are, in fact, general-purpose computers that also fulfil other purposes. One of those purposes is to serve as firewalls; the role of a firewall is closely linked with the routing function, as we describe in Section 3.4. The 138.37.95.232/29 subnet is not connected to the rest of the network at the IP level. Only the file server *custard* can access it to provide a printing service on the attached printers via a server process that monitors and controls the use of the printers.

All of the links in Figure 3.10 are Ethernets. The bandwidth of most of them is 100 Mbps, but one is 1000 Mbps because it carries a large volume of traffic between a large number of computers used by students and *custard*, the file server that holds all of their files.

There are two Ethernet switches and several Ethernet hubs in the portion of the network illustrated. Both types of component are transparent to IP packets. An Ethernet hub is simply a means of connecting together several segments of Ethernet cable, all of which form a single Ethernet at the network protocol level. All of the Ethernet packets received by the host are relayed to all of the segments. An Ethernet switch connects

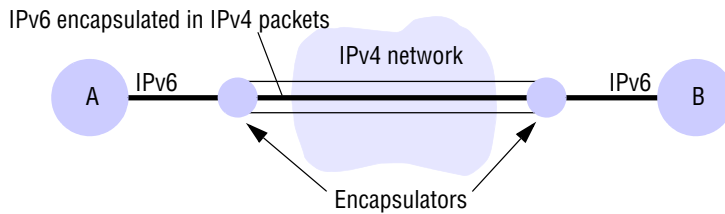
**Figure 3.10** Simplified view of part of a university campus network



several Ethernets, routing the incoming packets only to the Ethernet to which the destination host is connected.

**Routers** • We have noted that routing is required in all networks except those such as Ethernets and wireless networks, in which all of the hosts are connected by a single transmission medium. Figure 3.7 shows such a network with five routers connected by six links. In an internetwork, the routers may be linked by direct connections, as is shown in Figure 3.7, or they may be interconnected through subnets, as shown for *custard* in Figure 3.10. In both cases, the routers are responsible for forwarding the internetwork packets that arrive on any connection to the correct outgoing connection, as explained above. They maintain routing tables for that purpose.

**Bridges** • Bridges link networks of different types. Some bridges link several networks, and these are referred to as bridge/routers because they also perform routing functions. For example, the wider campus network includes a Fibre Distributed Data Interface

**Figure 3.11** Tunnelling for IPv6 migration

(FDDI) backbone (not shown on Figure 3.10), and this is linked to the Ethernet subnets in the figure by bridge/routers.

**Hubs** • Hubs are simply a convenient means of connecting hosts and extending segments of Ethernet and other broadcast local network technologies. They have a number of sockets (typically 4–64), to each of which a host computer can be connected. They can also be used to overcome the distance limitations on single segments and provide a means of adding additional hosts.

**Switches** • Switches perform a similar function to routers, but for local networks (normally Ethernets) only. That is, they interconnect several separate Ethernets, routing the incoming packets to the appropriate outgoing network. They perform their task at the level of the Ethernet network protocol. When they start up they have no knowledge of the wider internetwork and build up routing tables by the observation of traffic, supplemented by broadcast requests when they lack information.

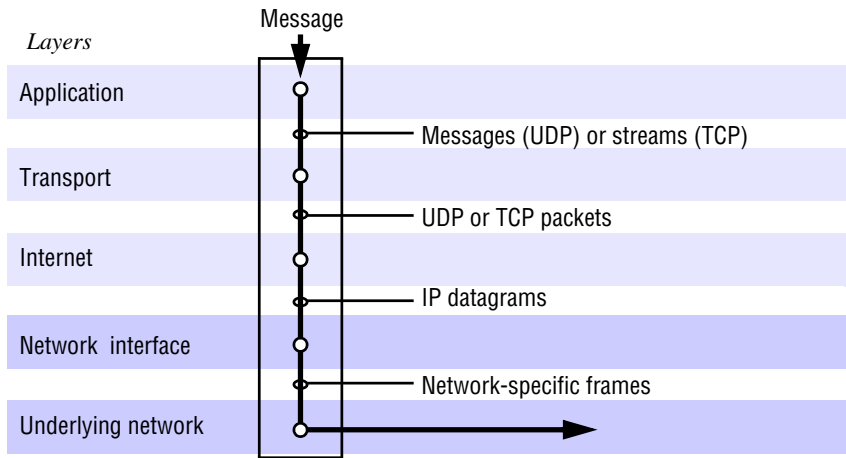
The advantage of switches over hubs is that they separate the incoming traffic and transmit it only on the relevant outgoing network, reducing congestion on the other networks to which they are connected.

**Tunnelling** • Bridges and routers transmit internetwork packets over a variety of underlying networks by translating between their network-layer protocols and an internetwork protocol, but there is one situation in which the underlying network protocol can be hidden from the layers above it without the use of an internetwork protocol. A pair of nodes connected to separate networks of the same type can communicate through another type of network by constructing a protocol ‘tunnel’. A protocol tunnel is a software layer that transmits packets through an alien network environment.

The following analogy explains the reason for the choice of terminology and provides another way to think about tunnelling. A tunnel through a mountain enables a road to transport cars where it would otherwise be impossible. The road is continuous – the tunnel is transparent to the application (cars). The road is the transport mechanism, and the tunnel enables it to work in an alien environment.

Figure 3.11 illustrates the use of tunnelling to support the migration of the Internet to the IPv6 protocol. IPv6 is intended to replace the version of IP still widely in use, IPv4, and is incompatible with it. (Both IPv4 and IPv6 are described in Section 3.4.) During the period of transition to IPv6 there will be ‘islands’ of IPv6 networking in the sea of IPv4. In our illustration A and B are such islands. At the boundaries of islands



**Figure 3.12** TCP/IP layers

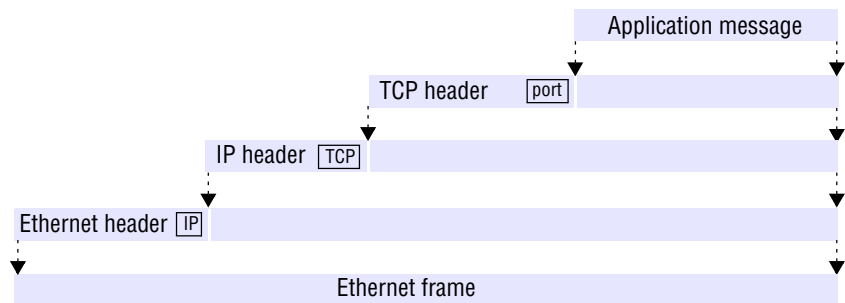
IPv6 packets are encapsulated in IPv4 and transported over the intervening IPv4 networks in that manner.

For another example, MobileIP (described in Section 3.4.5) transmits IP packets to mobile hosts anywhere in the Internet by constructing a tunnel to them from their home base. The intervening network nodes do not need to be modified to handle the MobileIP protocol. The IP multicast protocol is handled in a similar way, relying on a few routers that support IP multicast routing to determine the routes, but transmitting IP packets through other routers using standard IP addresses. The PPP protocol for the transmission of IP packets over serial links provides yet another example.

## 3.4 Internet protocols

We describe here the main features of the TCP/IP suite of protocols and discuss their advantages and limitations when used in distributed systems.

The Internet emerged from two decades of research and development work on wide area networking in the USA, commencing in the early 1970s with the ARPANET – the first large-scale computer network development [Leiner *et al.* 1997]. An important part of that research was the development of the TCP/IP protocol suite. TCP stands for Transmission Control Protocol, IP for Internet Protocol. The widespread adoption of the TCP/IP and Internet application protocols in national research networks, and more recently in commercial networks in many countries, has enabled the national networks to be integrated into a single internetwork that has grown extremely rapidly to its present size, with more than 60 million hosts. Many application services and application-level protocols (shown in parentheses in the following list) now exist based on TCP/IP, including the Web (HTTP), email (SMTP, POP), netnews (NNTP), file transfer (FTP) and Telnet (telnet). TCP is a transport protocol; it can be used to support applications directly, or additional protocols can be layered on it to provide additional features. For

**Figure 3.13** Encapsulation as it occurs when a message is transmitted via TCP over an Ethernet

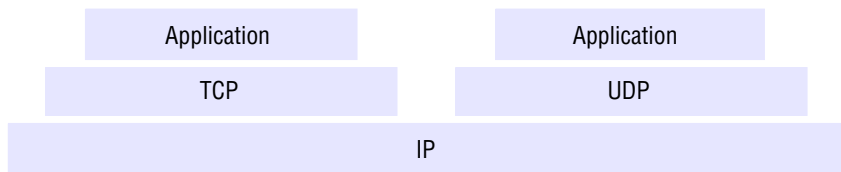
example, HTTP is usually transported by the direct use of TCP, but when end-to-end security is required, the Transport Layer Security (TLS) protocol (described in Section 11.6.3) is layered on top of TCP to produce secure channels and HTTP messages are transmitted via the secure channels.

The Internet protocols were originally developed primarily to support simple wide area applications such as file transfer and electronic mail, involving communication with relatively high latencies between geographically dispersed computers, but they turned out to be efficient enough to support the requirements of many distributed applications on both wide area and local networks and they are now almost universally used in distributed systems. The resulting standardization of communication protocols has brought immense benefits.

The general illustration of internetwork protocol layers of Figure 3.6 is translated into the specific Internet case in Figure 3.12. There are two transport protocols – TCP (Transport Control Protocol) and UDP (User Datagram Protocol). TCP is a reliable connection-oriented protocol, and UDP is a datagram protocol that does not guarantee reliable transmission. The Internet Protocol is the underlying ‘network’ protocol of the Internet virtual network – that is, IP datagrams provide the basic transmission mechanism for the Internet and other TCP/IP networks. We placed the word ‘network’ in quotation marks in the preceding sentence because it is not the only network layer involved in the implementation of Internet communication. This is because the Internet protocols are usually layered over another network technology, such as Ethernet, which already provides a network layer that enables the computers attached to the same network to exchange datagrams. Figure 3.13 illustrates the encapsulation of packets that would occur for the transmission of a message via TCP over an underlying Ethernet. The tags in the headers are the protocol types for the layers above, needed for the receiving protocol stack to correctly unpack the packets. In the TCP layer, the receiver’s port number serves a similar purpose, enabling the TCP software component at the receiving host to pass the message to a specific application-level process.

The TCP/IP specifications [Postel 1981a; 1981b] do not specify the layers below the Internet datagram layer – IP packets in the Internet layer are transformed into packets for transmission over almost any combination of underlying networks or data links.

For example, IP ran initially over the ARPANET, which consisted of hosts and an early version of routers (called PSEs) connected by long-distance data links. Today it is

**Figure 3.14** The programmer's conceptual view of a TCP/IP Internet

used over virtually every known network technology, including ATM, local area networks such as Ethernets, and token ring networks. IP is implemented over serial lines and telephone circuits via the PPP protocol [Parker 1992], enabling it to be used for communication with modem connections and other serial links.

The success of TCP/IP is based on the protocols' independence from the underlying transmission technology, enabling internetworks to be built up from many heterogeneous networks and data links. Users and application programs perceive a single virtual network supporting TCP and UDP and implementors of TCP and UDP see a single virtual IP network, hiding the diversity of the underlying transmission media. Figure 3.14 illustrates this view.

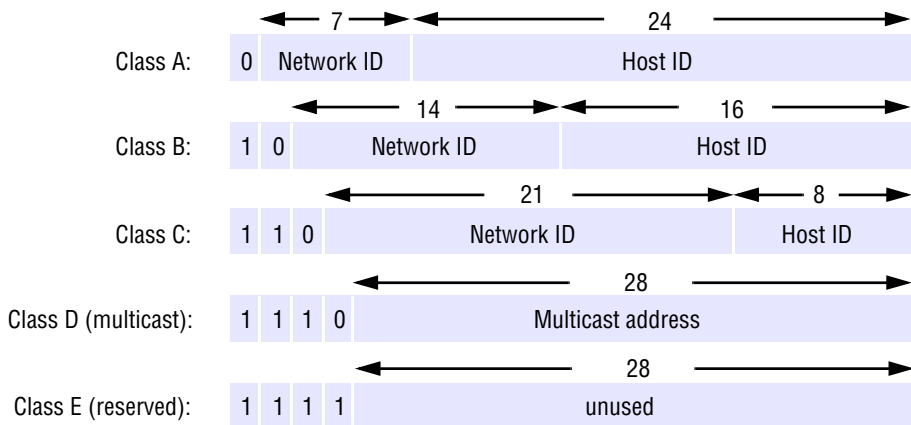
In the next two sections we describe the IP addressing scheme and the IP protocol. The Domain Name System – which converts domain names such as *www.amazon.com*, *hpl.hp.com*, *stanford.edu* and *qmw.ac.uk*, with which Internet users are so familiar, into IP addresses – is introduced in Section 3.4.7 and described more fully in Chapter 13.

The version of IP in predominant use throughout the Internet is IPv4 (since January 1984), and that is the version that we shall describe in the next two sections. But the rapid growth in the use of the Internet led to the publication of a specification of a new version (IPv6) to overcome the addressing limitations of IPv4 and add features to support some new requirements. We describe IPv6 in Section 3.4.4. Because of the vast amount of software that will be affected, a gradual migration to IPv6 is planned over a period of 10 years or more.

### 3.4.1 IP addressing

Perhaps the most challenging aspect of the design of the Internet protocols was the construction of schemes for naming and addressing hosts and for routing IP packets to their destinations. The scheme used for assigning host addresses to networks and the computers connected to them had to satisfy the following requirements:

- It must be *universal* – any host must be able to send packets to any other host in the Internet.
- It must be efficient in its use of the address space – it is impossible to predict the ultimate size of the Internet and the number of network and host addresses likely to be required. The address space must be carefully partitioned to ensure that addresses will not run out. In 1978–82, when the specifications for the TCP/IP protocols were being developed, provision for  $2^{32}$  or approximately 4 billion addressable hosts (about the same as the population of the world at that time) was

**Figure 3.15** Internet address structure, showing field sizes in bits

considered adequate. This judgement has proved to be short-sighted, for two reasons:

- The rate of growth of the Internet has far outstripped all predictions.
- The address space has been allocated and used much less efficiently than expected.
- The addressing scheme must lend itself to the development of a flexible and efficient routing scheme, but the addresses themselves cannot contain very much of the information needed to route a packet to its destination.

Today the overwhelming majority of Internet traffic continues to use the IP version 4 address and packet format defined three decades ago. The scheme assigns an IP address to each host in the Internet – a 32-bit numeric identifier containing a network identifier, which uniquely identifies one of the subnetworks in the Internet, and a host identifier, which uniquely identifies the host's connection to that network. It is these addresses that are placed in IP packets and used to route them to their destinations.

The design adopted for the Internet address space is shown in Figure 3.15. There are four allocated classes of Internet address – A, B, C and D. Class D is reserved for Internet multicast communication, which is implemented in only some Internet routers and is discussed further in Section 4.4.1. Class E contains a range of unallocated addresses, which are reserved for future requirements.

These 32-bit Internet addresses, containing a network identifier and host identifier, are usually written as a sequence of four decimal numbers separated by dots. Each decimal number represents one of the four bytes, or *octets*, of the IP address. The permissible values for each class of network address are shown in Figure 3.16.

Three classes of address were designed to meet the requirements of different types of organization. The Class A addresses, with a capacity for  $2^{24}$  hosts on each subnet, are reserved for very large networks such as the US NSFNet and other national wide area networks. Class B addresses are allocated to organizations that operate networks likely

**Figure 3.16** Decimal representation of Internet addresses

	<i>octet 1</i>	<i>octet 2</i>	<i>octet 3</i>	<i>Range of addresses</i>
	<i>Network ID</i>		<i>Host ID</i>	
Class A:	1 to 127	0 to 255	0 to 255	1.0.0.0 to 127.255.255.255
	<i>Network ID</i>		<i>Host ID</i>	
Class B:	128 to 191	0 to 255	0 to 255	128.0.0.0 to 191.255.255.255
	<i>Network ID</i>		<i>Host ID</i>	
Class C:	192 to 223	0 to 255	0 to 255	192.0.0.0 to 223.255.255.255
	<i>Network ID</i>		<i>Host ID</i>	
Class D (multicast):	224 to 239	0 to 255	0 to 255	224.0.0.0 to 239.255.255.255
	<i>Network ID</i>		<i>Host ID</i>	
Class E (reserved):	240 to 255	0 to 255	0 to 255	240.0.0.0 to 255.255.255.255
	<i>Network ID</i>		<i>Host ID</i>	

to contain more than 255 computers, and Class C addresses are allocated to all other network operators.

Internet addresses with host identifiers 0 and all 1s (binary) are used for special purposes. Addresses with the host identifier set to 0 are used to refer to ‘this host’, and a host identifier that is all 1s is used to address a broadcast message to all of the hosts connected to the network specified in the network identifier part of the address.

Network identifiers are allocated by the Internet Assigned Numbers Authority (IANA) to organizations with networks connected to the Internet. Host identifiers for the computers on each network connected to the Internet are assigned by the managers of the relevant networks.

Since host addresses include a network identifier, any computer that is connected to more than one network must have separate addresses on each, and whenever a computer is moved to a different network, its Internet address must change. These requirements can lead to substantial administrative overheads, for example in the case of portable computers.

In practice, the IP address allocation scheme has not turned out to be very effective. The main difficulty is that network administrators in user organizations cannot easily predict future growth in their need for host addresses, and they tend to overestimate, requesting Class B addresses when in doubt. Around 1990 it became evident that based on the rate of allocation at the time, IP addresses were likely to run out around 1996. Three steps were taken. The first was to initiate the development of a new IP protocol and addressing scheme, the result of which was the specification of IPv6.

The second step was to radically modify the way in which IP addresses were allocated. A new address allocation and routing scheme designed to make more effective use of the IP address space was introduced, called classless interdomain routing (CIDR). We describe CIDR in Section 3.4.3. The local network illustrated in

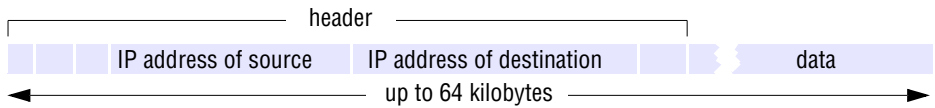
**Figure 3.17** IP packet layout

Figure 3.10 includes several Class C sized subnets in the range 138.37.88–138.37.95, linked by routers. The routers manage the delivery of IP packets to all of the subnets. They also handle traffic between the subnets and from the subnets to the rest of the world. The figure also illustrates the use of CIDR to subdivide a Class B address space to produce several Class C sized subnets.

The third step was to enable unregistered computers to access the Internet indirectly through routers that implement a Network Address Translation (NAT) scheme. We describe this scheme in Section 3.4.3.

### 3.4.2 The IP protocol

The IP protocol transmits datagrams from one host to another, if necessary via intermediate routers. The full IP packet format is rather complex, but Figure 3.17 shows the main components. There are several header fields, not shown in the diagram, that are used by the transmission and routing algorithms.

IP provides a delivery service that is described as offering *unreliable* or *best-effort* delivery semantics, because there is no guarantee of delivery. Packets can be lost, duplicated, delayed or delivered out of order, but these errors arise only when the underlying networks fail or buffers at the destination are full. The only checksum in IP is a header checksum, which is inexpensive to calculate and ensures that any corruptions in the addressing and packet management data will be detected. There is no data checksum, which avoids overheads when crossing routers, leaving the higher-level protocols (TCP and UDP) to provide their own checksums – a practical instance of the end-to-end argument (Section 2.3.3).

The IP layer puts IP datagrams into network packets suitable for transmission in the underlying network (which might, for example, be an Ethernet). When an IP datagram is longer than the MTU of the underlying network, it is broken into smaller packets at the source and reassembled at its final destination. Packets can be further broken up to suit the underlying networks encountered during the journey from source to destination. (Each packet has a fragment identifier to enable out-of-order fragments to be collected.)

The IP layer must also insert a ‘physical’ network address of the message destination to the underlying network. It obtains this from the address resolution module in the Internet network interface layer, which is described in the next subsection.

**Address resolution** • The address resolution module is responsible for converting Internet addresses to network addresses for a specific underlying network (sometimes called physical addresses). For example, if the underlying network is an Ethernet, the address resolution module converts 32-bit Internet addresses to 48-bit Ethernet addresses.

This translation is network technology dependent:

- Some hosts are connected directly to Internet packet switches; IP packets can be routed to them without address translation.
- Some local area networks allow network addresses to be assigned to hosts dynamically, and the addresses can be conveniently chosen to match the host identifier portion of the Internet address – translation is simply a matter of extracting the host identifier from the IP address.
- For Ethernet and some other local networks, the network address of each computer is hard-wired into its network interface hardware and bears no direct relation to its Internet address – translation depends upon knowledge of the correspondence between IP addresses and addresses for the hosts on the local network and is done using an address resolution protocol (ARP).

We now outline the implementation of an ARP for Ethernet. It uses dynamic enquiries in order to operate correctly when computers are added to a local network but exploits caching to minimize enquiry messages. Consider first the case in which a host computer connected to an Ethernet uses IP to transmit a message to another computer on the same Ethernet. The IP software module on the sending computer must translate the recipient's Internet address that it finds in the IP packet to an Ethernet address before the packet can be delivered. It invokes the ARP module on the sending computer to do so.

The ARP module on each host maintains a cache of (*IP address, Ethernet address*) pairs that it has previously obtained. If the required IP address is in the cache, then the query is answered immediately. If not, then ARP transmits an Ethernet broadcast packet (an ARP request packet) on the local Ethernet containing the desired IP address. Each of the computers on the local Ethernet receives the ARP request packet and checks the IP address in it to see whether it matches its own IP address. If it does, an ARP reply packet is sent to the originator of the ARP request containing the sender's Ethernet address; otherwise the ARP request packet is ignored. The originating ARP module adds the new *IP address* to *Ethernet address* mapping to its local cache of (*IP address, Ethernet address*) pairs so that it can respond to similar requests in the future without broadcasting an ARP request. Eventually, the ARP cache at each computer will contain an (*IP address, Ethernet address*) pair for all of the computers that IP packets are sent to. Thus ARP broadcasts will be needed only when a computer is newly connected to the local Ethernet.

**IP spoofing** • We have seen that IP packets include a source address – the IP address of the sending computer. This, together with a port address encapsulated in the data field (for UDP and TCP packets), is often used by servers to generate a return address. Unfortunately, it is not possible to guarantee that the source address given is in fact the address of the sender. A malicious sender can easily substitute an address that is different from its own. This loophole has been the source of several well-known attacks, including the distributed denial of service attacks of February 2000 [Farrow 2000] mentioned in Chapter 1, Section 1.5.3. The method used was to issue many *ping* service requests to a large number of computers at several sites (ping is a simple service designed to check the availability of a host). These malicious ping requests all contained the IP address of a target computer in their sender address field. The ping responses were

therefore all directed to the target, whose input buffers were overwhelmed, preventing any legitimate IP packets getting through. This attack is discussed further in Chapter 11.

### 3.4.3 IP routing

The IP layer routes packets from their source to their destination. Each router in the Internet implements IP-layer software to provide a routing algorithm.

**Backbones** • The topological map of the Internet is partitioned conceptually into *autonomous systems* (ASs), which are subdivided into *areas*. The intranets of most large organizations such as universities and large companies are regarded as ASs, and they will usually include several areas. In Figure 3.10, the campus intranet is an AS and the portion shown is an area. Every AS in the topological map has a *backbone* area. The collection of routers that connect non-backbone areas to the backbone and the links that interconnect those routers are called the backbone of the network. The links in the backbone are usually of high bandwidth and are replicated for reliability. This hierarchic structure is a conceptual one that is exploited primarily for the management of resources and the maintenance of the components. It does not affect the routing of IP packets.

**Routing protocols** • RIP-1, the first routing algorithm used in the Internet, is a version of the distance-vector algorithm described in Section 3.3.5. RIP-2 (described in RFC 1388 [Malkin 1993]) was developed from it to accommodate several additional requirements, including classless interdomain routing, better multicast routing and the need for authentication of RIP packets to prevent attacks on the routers.

As the scale of the Internet has expanded and the processing capacity of routers has increased, there has been a move towards the adoption of algorithms that do not suffer from the slow convergence and potential instability of distance-vector algorithms. The direction of the move is towards the link-state class of algorithms mentioned in Section 3.3.5 and the algorithm called *open shortest path first* (OSPF). This protocol is based on a path-finding algorithm that is due to Dijkstra [1959] and has been shown to converge more rapidly than the RIP algorithm.

We should note that the adoption of new routing algorithms in IP routers can proceed incrementally. A change in routing algorithm results in a new version of the RIP protocol, and a version number is carried by each RIP packet. The IP protocol does not change when a new RIP protocol is introduced. Any IP router will correctly forward incoming IP packets on a reasonable, if not optimum, route, whatever version of RIP they use. But for routers to cooperate in the updating of their routing tables, they must share a similar algorithm. For this purpose the topological areas defined above are used. Within each area a single routing algorithm applies, and the routers within an area cooperate in the maintenance of their routing tables. Routers that support only RIP-1 are still commonplace and they coexist with routers that support RIP-2 and OSPF, using backwards-compatibility features incorporated in the newer protocols.

In 1993, empirical observations [Floyd and Jacobson 1993] showed that the 30-second frequency with which RIP routers exchange information was producing a periodicity in the performance of IP transmissions. The average latency for IP packet transmissions showed a peak at 30-second intervals. This was traced to the behaviour of routers performing the RIP protocol – on receipt of an RIP packet, routers would delay the onward transmission of any IP packets that they held until the routing table update



process was complete for all RIP packets received to date. This tended to cause the routers to perform the RIP actions in lock-step. The correction recommended was for routers to adopt a random value in the range of 15–45 seconds for the RIP update period.

**Default routes** • Up to now, our discussion of routing algorithms has suggested that every router maintains a full routing table showing the route to every destination (subnet or directly connected host) in the Internet. At the current scale of the Internet this is clearly infeasible (the number of destinations is probably already in excess of 1 million and still growing very rapidly).

Two possible solutions to this problem come to mind, and both have been adopted in an effort to alleviate the effects of the Internet’s growth. The first solution is to adopt some form of topological grouping of IP addresses. Prior to 1993, nothing could be inferred from an IP address about its location. In 1993, as part of the move to simplify and economize on the allocation of IP addresses that is discussed below under CIDR, the decision was taken that for future allocations, the following regional locations would be applied:

- Addresses 194.0.0.0 to 195.255.255.255 are in Europe
- Addresses 198.0.0.0 to 199.255.255.255 are in North America
- Addresses 200.0.0.0 to 201.255.255.255 are in Central and South America
- Addresses 202.0.0.0 to 203.255.255.255 are in Asia and the Pacific

Because these geographical regions also correspond to well-defined topological regions in the Internet and just a few gateway routers provide access to each region, this enables a substantial simplification of routing tables for those address ranges. For example, a router outside Europe can have a single table entry for the range of addresses 194.0.0.0 to 195.255.255.255 that sends all IP packets with destinations in that range on the same route to the nearest European gateway router. But note that before the date of that decision, IP addresses were allocated largely without regard to topology or geography. Many of those addresses are still in use, and the 1993 decision does nothing to reduce the scale of routing table entries for those addresses.

The second solution to the routing table size explosion problem is simpler and very effective. It is based on the observation that the accuracy of routing information can be relaxed for most routers as long as some key routers (those closest to the backbone links) have relatively complete routing tables. The relaxation takes the form of a *default* destination entry in routing tables. The default entry specifies a route to be used for all IP packets whose destinations are not included in the routing table. To illustrate this, consider Figures 3.7 and 3.8 and suppose that the routing table for node C is altered to show:

<i>Routings from C</i>		
<i>To</i>	<i>Link</i>	<i>Cost</i>
B	2	1
C	local	0
E	5	1
Default	5	-

Thus node C is ignorant of nodes A and D. It will route all packets addressed to them via link 5 to E. What is the consequence? Packets addressed to D will reach their destination without loss of efficiency in routing, but packets addressed to A will make an extra hop, passing through E and B on the way. In general, the use of default routings trades routing efficiency for table size. But in some cases, especially where a router is on a spur, so that all outward messages must pass through a single point, there is no loss of efficiency. The default routing scheme is heavily used in Internet routing; no single router holds routes to all destinations in the Internet.

**Routing on a local subnet** • Packets addressed to hosts on the same network as the sender are transmitted to the destination host in a single hop, using the host identifier part of the address to obtain the address of the destination host on the underlying network. The IP layer simply uses ARP to get the network address of the destination and then uses the underlying network to transmit the packets.

If the IP layer in the sending computer discovers that the destination is on a different network, it must send the message to a local router. It uses ARP to get the network address of the gateway or router and then uses the underlying network to transmit the packet to it. Gateways and routers are connected to two or more networks and they have several Internet addresses, one for each network to which they are attached.

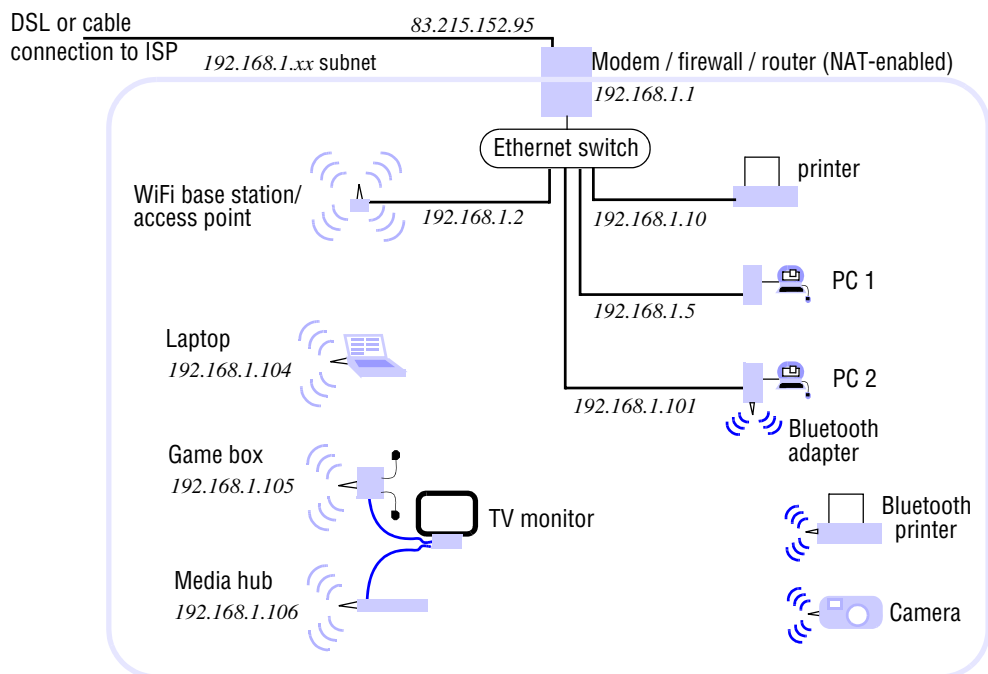
**Classless interdomain routing (CIDR)** • The shortage of IP addresses referred to in Section 3.4.1 led to the introduction in 1996 of this scheme for allocating addresses and managing the entries in routing tables. The main problem was a scarcity of Class B addresses – those for subnets with more than 255 hosts connected. Plenty of Class C addresses were available. The CIDR solution for this problem is to allocate a batch of contiguous Class C addresses to a subnet requiring more than 255 addresses. The CIDR scheme also makes it possible to subdivide a Class B address space for allocation to multiple subnets.

Batching Class C addresses sounds like a straightforward step, but unless it is accompanied by a change in routing table format, it has a substantial impact on the size of routing tables and hence the efficiency of the algorithms that manage them. The change adopted was to add a *mask* field to the routing tables. The mask is a bit pattern that is used to select the portion of an IP address that is compared with the routing table entry. This effectively enables the host/subnet address to be any portion of the IP address, providing more flexibility than the classes A, B and C – hence the name *classless* interdomain routing. Once again, these changes to routers are made on an incremental basis, so some routers perform CIDR and others use the old class-based algorithms.

This works because the newly allocated ranges of Class C addresses are assigned modulo 256, so each range represents an integral number of Class C sized subnet addresses. On the other hand, some subnets also make use of CIDR to subdivide the range of addresses in a single network, of Class A, B or C. If a collection of subnets is connected to the rest of the world entirely by CIDR routers, then the ranges of IP addresses used within the collection can be allocated to individual subnets in chunks determined by a binary mask of any size.

For example, a Class C address space can be subdivided into 32 groups of 8. Figure 3.10 contains an example of the use of the CIDR mechanism to split the

**Figure 3.18** A typical NAT-based home network



138.37.95 Class C sized subnet into several groups of eight host addresses that are routed differently. The separate groups are denoted by notations 138.37.95.232/29, 138.37.95.248/29 and so on. The /29 portion of these addresses denotes an attached 32-bit binary mask with 29 leading 1s and three trailing 0s.

**Unregistered addresses and Network Address Translation (NAT)** • Not all of the computers and devices that access the Internet need to be assigned globally unique IP addresses. Computers that are attached to a local network and access to the Internet through a NAT-enabled router can rely upon the router to redirect incoming UDP and TCP packets for them. Figure 3.18 illustrates a typical home network with computers and other network devices linked to the Internet through a NAT-enabled router. The network includes Internet-enabled computers that are connected to the router by a wired Ethernet connection as well as others that are connected through a WiFi access point. For completeness some Bluetooth-enabled devices are shown, but these are not connected to the router and hence cannot access the Internet directly. The home network has been allocated a single registered IP address (83.215.152.95) by its Internet service provider. The approach described here is suitable for any organization wishing to connect computers without registered IP addresses to the Internet.

All of the Internet-enabled devices on the home network have been assigned unregistered IP addresses on the 192.168.1.x Class C subnet. Most of the internal computers and devices are allocated individual IP addresses dynamically by a Dynamic

Host Configuration Protocol (DHCP) service running on the router. In our illustration the numbers above 192.168.1.100 are used by the DHCP service and the nodes with lower numbers (such as PC 1) have been allocated numbers manually, for a reason explained later in this subsection. Although all of these addresses are completely hidden from the rest of the Internet by the NAT router, it is conventional to use a range of addresses from one of three blocks of addresses (10.z.y.x, 172.16.y.x or 192.168.y.x) that IANA has reserved for private internets.

NAT is described in RFC 1631 [Egevang and Francis 1994] and extended in RFC 2663 [Srisuresh and Holdrege 1999]. NAT-enabled routers maintain an address translation table and exploit the source and destination port number fields in the UDP and TCP packets to assign each incoming reply message to the internal computer that sent the corresponding request message. Note that the source port given in a request message is always used as the destination port in the corresponding reply message.

The most commonly used variant of NAT addressing works as follows:

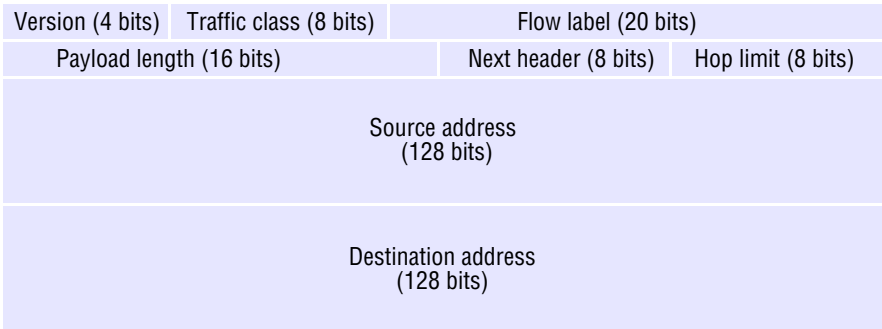
- When a computer on the internal network sends a UDP or TCP packet to a computer outside it, the router receives the packet and saves the source IP address and port number to an available slot in its address translation table.
- The router replaces the source address in the packet with the router's IP address and the source port with a *virtual port number* that indexes the table slot containing the sending computer's address information.
- The packet with the modified source address and port number is then forwarded towards its destination by the router. The address translation table now holds a mapping from virtual port numbers to real internal IP addresses and port numbers for all packets sent recently by computers on the internal network.
- When the router receives a UDP or TCP packet from an external computer it uses the destination port number in the packet to access a slot in the address translation table. It replaces the destination address and destination port in the received packet with those stored in the slot and forwards the modified packet to the internal computer identified by the destination address.

The router will retain a port mapping and reuse it as long as it appears to be in use. A timer is reset each time the router accesses an entry in the table. If the entry is not accessed again before the timer expires, the entry is removed from the table.

The scheme described above deals satisfactorily with the commonest modes of communication for nonregistered computers, in which they act as clients to external services such as web servers. But it does not enable them to act as servers to handle incoming requests. To deal with that case, NAT routers can be configured manually to forward all of the incoming requests on a given port to one particular internal computer. Computers that act as servers must retain the same internal IP address and this is achieved by allocating their addresses manually (as was done for PC 1). This solution to the problem of providing external access to services is satisfactory as long as there is no requirement for more than one internal computer to offer a service on any given port.

NAT was introduced as a short-term solution to the problem of IP address allocation for personal and home computers. It has enabled the expansion of Internet use to proceed far further than was originally anticipated, but it does impose some

Figure 3.19 IPv6 header layout



limitations, of which the last point is an example. IPv6 must be seen as the next step, enabling full Internet participation for all computers and portable devices.

3.4.4 IP version 6

A more permanent solution to the addressing limitations of IPv4 was also pursued, and this led to the development and adoption of a new version of the IP protocol with substantially larger addresses. The IETF noticed the potential problems arising from the 32-bit addresses of IPv4 as early as 1990 and initiated a project to develop a new version of the IP protocol. IPv6 was adopted by the IETF in 1994 and a strategy for migration to it was recommended.

Figure 3.19 shows the layout of IPv6 headers. We do not propose to cover their construction in detail here. Readers are referred to Tanenbaum [2003] or Stallings [2002] for tutorial material on IPv6 and to Huitema [1998] for a blow-by-blow account of the IPv6 design process and implementation plans. Here we will outline the main advances that IPv6 embodies:

*Address space:* IPv6 addresses are 128 bits (16 bytes) long. This provides for a truly astronomical number of addressable entities:  $2^{128}$ , or approximately  $3 \times 10^{38}$ . Tanenbaum calculates that this is sufficient to provide  $7 \times 10^{23}$  IP addresses per square metre across the entire surface of the Earth. More conservatively, Huitema made a calculation assuming that IP addresses are allocated as inefficiently as telephone numbers and came up with a figure of 1000 IP addresses per square metre of the Earth’s surface (land and water).

The IPv6 address space is partitioned. We cannot detail the partitioning here, but even the minor partitions (one of which will hold the entire range of IPv4 addresses, mapped one-to-one) are far larger than the total IPv4 space. Many partitions (representing 72% of the total) are reserved for purposes as yet undefined. Two large partitions (each comprising 1/8 of the address space) are allocated for general purposes and will be assigned to normal network nodes. One of them is intended to be organized according to the geographic locations of the addressed nodes and the other according to their organizational locations. This allows two

alternative strategies for aggregating addresses for routing purposes – it remains to be seen which will prove more effective or popular.

*Routing speed:* The complexity of the basic IPv6 header and the processing required at each node are reduced. No checksum is applied to the packet content (payload), and no fragmentation can occur once a packet has begun its journey. The former is considered acceptable because errors can be detected at higher levels (TCP does include a content checksum), and the latter is achieved by supporting a mechanism for determining the smallest MTU before a packet is transmitted.

*Real-time and other special services:* The *traffic class* and *flow label* fields are concerned with this. Multimedia streams and other sequences of real-time data elements can be transmitted as part of an identified flow. The first 6 bits of the *traffic class* field can be used with the *flow label* or independently to enable specific packets to be handled more rapidly or with higher reliability than others. Traffic class values 0 through 8 are for transmissions that can be slowed without disastrous effects on the application. Other values are reserved for packets whose delivery is time-dependent. Such packets must either be delivered promptly or dropped – late delivery is of no value.

Flow labels enable resources to be reserved in order to meet the timing requirements of specific real-time data streams, such as live audio and video transmissions. Chapter 20 discusses these requirements and methods for the allocation of resources for them. Of course, the routers and transmission links in the Internet have limited resources, and the concept of reserving them for specific users and applications has not previously been considered. The use of these facilities of IPv6 will depend upon major enhancements to the infrastructure and the development of suitable methods for charging and arbitrating the allocation of resources.

*Future evolution:* The key to the provision for future evolution is the *next header* field. If non-zero, it defines the type of an extension header that is included in the packet. There are currently extension header types that provide additional data for special services of the following types: information for routers, route definition, fragment handling, authentication, encryption and destination handling. Each extension header type has a specific size and a defined format. Further extension header types will be defined as new service requirements arise. An extension header, if present, follows the basic header and precedes the payload and includes a *next header* field, enabling multiple extension headers to be employed.

*Multicast and anycast:* Both IPv4 and IPv6 include support for the transmission of IP packets to multiple hosts using a single address (one that is in the range reserved for the purpose). The IP routers are then responsible for routing the packet to all of the hosts that have subscribed to the group identified by the relevant address. Further details on IP multicast communication can be found in Section 4.4.1. In addition, IPv6 supports a new mode of transmission called *anycast*. This service delivers a packet to at least one of the hosts that subscribes to the relevant address.

*Security:* Up to now, Internet applications that require authenticated or private data transmission have relied on the use of cryptographic techniques in the application layer. The end-to-end argument supports the view that this is the right place for it. If security is implemented at the IP level, then users and application developers depend

upon the correctness of the code that implements it in each router along the way, and they must trust the routers and other intermediate nodes to handle cryptographic keys.

The advantage of implementing security at the IP level is that it can be applied without the need for security-aware implementations of application programs. For example, system managers can implement it in a firewall and apply it uniformly to all external communication without incurring the cost of encryption for internal communication. Routers may also exploit an IP-level security mechanism to secure the routing table update messages that they exchange between themselves.

Security in IPv6 is implemented through the *authentication* and *encrypted security payload* extension header types. These implement features equivalent to the secure channel concept introduced in Section 2.4.3. The payload is encrypted and/or digitally signed as required. Similar security features are also available in IPv4 using IP tunnelling between routers or hosts that implement the IPSec specification (see RFC 2411 [Thayer 1998]).

**Migration from IPv4** • The consequences for the existing Internet infrastructure of a change in its basic protocol are profound. IP is processed in the TCP/IP protocol stack at every host and in the software of every router. IP addresses are handled in many application and utility programs. All of these require upgrading to support the new version of IP, but the change is made inevitable by the forthcoming exhaustion of the address space provided by IPv4. The IETF working group responsible for IPv6 has defined a migration strategy – essentially it involves the implementation of ‘islands’ of IPv6 routers and hosts communicating with other IPv6 islands via tunnels and gradually merging into larger islands.

As we have noted, IPv6 routers and hosts should have no difficulty in handling mixed traffic, since the IPv4 address space is embedded in the IPv6 space. All of the major operating systems (Windows XP, Mac OS X, Linux and other Unix variants) already include implementations of UDP and TCP sockets (as described in Chapter 4) over IPv6, enabling applications to be migrated with a simple upgrade.

The theory of this strategy is technically sound, but implementation progress has been very slow, perhaps because CIDR and NAT have relieved the pressure to a greater extent than anticipated. This has begun to change in the mobile phone and portable device markets, though. All of these devices are likely to be Internet-enabled in the near future and they cannot easily be hidden behind NAT routers. For example, it is projected that more than a billion IP devices will be deployed in India and China by 2014. Only IPv6 can address needs such as that.

### 3.4.5 MobileIP

Mobile computers such as laptops and tablets are connected to the Internet at different locations as they migrate. In its owner’s office a laptop may be connected to a local Ethernet connected to the Internet through a router, it may be connected via a mobile phone while it is in transit by car or train, then it may be attached to an Ethernet at another site. The user will wish to access services such as email and the Web at any of these locations.

Simple access to services does not require a mobile computer to retain a single address, and it may acquire a new IP address at each site; that is the purpose of the



Dynamic Host Configuration Protocol (DHCP), which enables a newly connected computer to dynamically acquire an IP address in the address range of the local subnet and discover the addresses of local resources such as a DNS server from a local DHCP server. It will also need to discover what local services (such as printing, mail delivery and so on) are available at each site that it visits. Discovery services are a type of naming service that assist with this; they are described in Chapter 19 (Section 19.2).

There may be files or other resources on the laptop to which others require access, or the laptop may be running a distributed application such as a share-monitoring service that receives notifications of specified events, such as stocks that the user holds passing a preset threshold. If a mobile computer is to remain accessible to clients and resource-sharing applications when it moves between local networks and wireless networks, it must retain a single IP number, but IP routing is subnet-based. Subnets are at fixed locations, and the correct routing of packets to them depends upon their position on the network.

MobileIP is a solution for the latter problem. The solution is implemented transparently, so IP communication continues normally when a mobile host computer moves between subnets at different locations. It is based upon the permanent allocation of a normal IP address to each mobile host on a subnet in its ‘home’ domain.

When the mobile host is connected at its home base, packets are routed to it in the normal way. When it is connected to the Internet elsewhere, two agent processes take responsibility for rerouting. The agents are a *home agent* (HA) and a *foreign agent* (FA). These processes run on convenient fixed computers at the home site and at the current location of the mobile host.

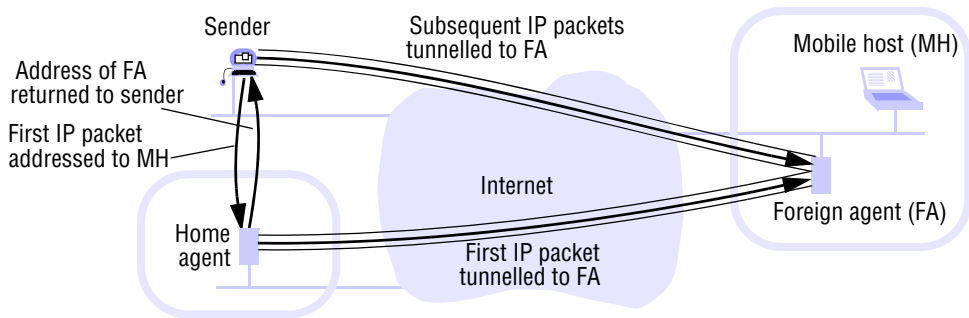
The HA is responsible for holding up-to-date knowledge of the mobile host’s current location (the IP address by which it can be reached). It does this with the assistance of the mobile host itself. When a mobile host leaves its home site, it should inform the HA, and the HA notes the mobile host’s absence. During the absence it will behave as a proxy; in order to do so, it tells the local routers to cancel any cached records relating to the mobile host’s IP address. While it is acting as a proxy, the HA responds to ARP requests concerning the mobile host’s IP address, giving its own local network address as the network address of the mobile host.

When the mobile host arrives at a new site, it informs the FA at that site. The FA allocates a ‘care-of address’ to it – a new, temporary IP address on the local subnet. The FA then contacts the HA, giving it the mobile host’s home IP address and the care-of address that has been allocated to it.

Figure 3.20 illustrates the MobileIP routing mechanism. When an IP packet addressed to the mobile host’s home address is received at the home network, it is routed to the HA. The HA then encapsulates the IP packet in a MobileIP packet and sends it to the FA. The FA unpacks the original IP packet and delivers it to the mobile host via the local network to which it is currently attached. Note that the method by which the HA and the FA reroute the original packet to its intended recipient is an instance of the tunnelling technique described in Section 3.3.7.

The HA also sends the care-of address of the mobile host to the original sender. If the sender is MobileIP-enabled, it will note the new address and use it for subsequent communication with the mobile host, avoiding the overheads of rerouting via the HA. If it is not, then it will ignore the change of address and subsequent communication will continue to be rerouted via the HA.



**Figure 3.20** The MobileIP routing mechanism

The MobileIP solution is effective, but hardly efficient. A solution that treats mobile hosts as first-class citizens would be preferable, allowing them to wander without giving prior notice and routing packets to them without any tunnelling or re-routing. We should note that this apparently difficult feat is exactly what is achieved by the cellular phone network – mobile phones do not change their number as they move between cells, or even between countries. Instead, they simply notify the local cellular phone base station of their presence from time to time.

### 3.4.6 TCP and UDP

TCP and UDP provide the communication capabilities of the Internet in a form that is useful for application programs. Application developers might wish for other types of transport service, for example to provide real-time guarantees or security, but such services would generally require more support in the network layer than IPv4 provides. TCP and UDP can be viewed as a faithful reflection at the application programming level of the communication facilities that IPv4 has to offer. IPv6 is another story; it will certainly continue to support TCP and UDP, but it includes capabilities that cannot be conveniently accessed through TCP and UDP. It may be useful to introduce additional types of transport service to exploit them, once the deployment of IPv6 is sufficiently wide to justify their development.

Chapter 4 describes the characteristics of both TCP and UDP from the point of view of distributed program developers. Here we shall be quite brief, describing only the functionality that they add to IP.

**Use of ports** • The first characteristic to note is that, whereas IP supports communication between pairs of computers (identified by their IP addresses), TCP and UDP, as transport protocols, must provide process-to-process communication. This is accomplished by the use of ports. *Port numbers* are used for addressing messages to processes within a particular computer and are valid only within that computer. A port number is a 16-bit integer. Once an IP packet has been delivered to the destination host, the TCP- or UDP-layer software dispatches it to a process via a specific port at that host.

**UDP features** • UDP is almost a transport-level replica of IP. A UDP datagram is encapsulated inside an IP packet. It has a short header that includes the source and

destination port numbers (the corresponding host addresses are present in the IP header), a length field and a checksum. UDP offers no guarantee of delivery. We have already noted that IP packets may be dropped because of congestion or network error. UDP adds no additional reliability mechanisms except the checksum, which is optional. If the checksum field is non-zero, the receiving host computes a check value from the packet contents and compares it with the received checksum; packets for which they do not match are dropped.

Thus UDP provides a means of transmitting messages of up to 64 kbytes in size (the maximum packet size permitted by IP) between pairs of processes (or from one process to several in the case of datagrams addressed to IP multicast addresses), with minimal additional costs or transmission delays above those due to IP transmission. It incurs no setup costs and it requires no administrative acknowledgement messages. But its use is restricted to those applications and services that do not require reliable delivery of single or multiple messages.

**TCP features** • TCP provides a much more sophisticated transport service. It provides reliable delivery of arbitrarily long sequences of bytes via stream-based programming abstraction. The reliability guarantee entails the delivery to the receiving process of all of the data presented to the TCP software by the sending process, in the same order. TCP is connection-oriented. Before any data is transferred, the sending and receiving processes must cooperate in the establishment of a bidirectional communication channel. The connection is simply an end-to-end agreement to perform reliable data transmission; intermediate nodes such as routers have no knowledge of TCP connections, and the IP packets that transfer the data in a TCP transmission do not necessarily all follow the same route.

The TCP layer includes additional mechanisms (implemented over IP) to meet the reliability guarantees. These are:

*Sequencing:* A TCP sending process divides the stream into a sequence of data segments and transmits them as IP packets. A sequence number is attached to each TCP segment. It gives the byte number within the stream for the first byte of the segment. The receiver uses the sequence numbers to order the received segments before placing them in the input stream at the receiving process. No segment can be placed in the input stream until all lower-numbered segments have been received and placed in the stream, so segments that arrive out of order must be held in a buffer until their predecessors arrive.

*Flow control:* The sender takes care not to overwhelm the receiver or the intervening nodes. This is achieved by a system of segment acknowledgements. Whenever a receiver successfully receives a segment, it records its sequence number. From time to time the receiver sends an acknowledgement to the sender, giving the sequence number of the highest-numbered segment in its input stream together with a *window size*. If there is a reverse flow of data, acknowledgements are carried in the normal data segments; otherwise they travel in acknowledgement segments. The window size field in the acknowledgement segment specifies the quantity of data that the sender is permitted to send before the next acknowledgement.

When a TCP connection is used for communication with a remote interactive program, data may be produced in small quantities but in a very bursty manner. For

example, keyboard input may result in only a few characters per second, but the characters should be sent sufficiently quickly for the user to see the results of their typing. This is dealt with by setting a timeout  $T$  on local buffering – typically 0.5 seconds. With this simple scheme, a segment is sent to the receiver whenever data has been waiting in the output buffer for  $T$  seconds, or the contents of the buffer reach the MTU limit. This buffering scheme cannot add more than  $T$  seconds to the interactive delay. Nagle has described another algorithm that produces less traffic and is more effective for some interactive applications [Nagle 1984]. Nagle's algorithm is used in many TCP implementations. Most TCP implementations are configurable, allowing applications to change the value of  $T$  or to select one of several buffering algorithms.

Because of the unreliability of wireless networks and the resulting frequent loss of packets, these flow-control mechanisms are not particularly relevant for wireless communication. This is one of the reasons for the adoption of a different transport mechanism in the WAP family of protocols for wide area mobile communication. But the implementation of TCP for wireless networks is also important, and modifications to the TCP mechanism have been proposed for this purpose [Balakrishnan *et al.* 1995, 1996]. The idea is to implement a TCP support component at the wireless base station (the gateway between wired and wireless networks). The support component snoops on TCP segments to and from the wireless network, retransmitting any outbound segments that are not acknowledged rapidly by the mobile receiver and requesting retransmissions of inbound segments when gaps in the sequence numbers are noticed.

*Retransmission:* The sender records the sequence numbers of the segments that it sends. When it receives an acknowledgement it notes that the segments were successfully received, and it may then delete them from its outgoing buffers. If any segment is not acknowledged within a specified timeout, the sender retransmits it.

*Buffering:* The incoming buffer at the receiver is used to balance the flow between the sender and the receiver. If the receiving process issues *receive* operations more slowly than the sender issues *send* operations, the quantity of data in the buffer will grow. Usually it is extracted from the buffer before it becomes full, but ultimately the buffer may overflow, and when that happens incoming segments are simply dropped without recording their arrival. Their arrival is therefore not acknowledged and the sender is obliged to retransmit them.

*Checksum:* Each segment carries a checksum covering the header and the data in the segment. If a received segment does not match its checksum, the segment is dropped.

### 3.4.7 Domain names

The design and implementation of the Domain Name System (DNS) is described in detail in Chapter 13; we give a brief overview here to complete our discussion of the Internet protocols. The Internet supports a scheme for the use of symbolic names for hosts and networks, such as *binkley.cs.mcgill.ca* or *essex.ac.uk*. The named entities are organized into a naming hierarchy. The named entities are called *domains* and the symbolic names are called *domain names*. Domains are organized in a hierarchy that is

intended to reflect their organizational structure. The naming hierarchy is entirely independent of the physical layout of the networks that constitute the Internet. Domain names are convenient for human users, but they must be translated to Internet (IP) addresses before they can be used as communication identifiers. This is the responsibility of a specific service, the DNS. Application programs pass requests to the DNS to convert the domain names that users specify into Internet addresses.

The DNS is implemented as a server process that can be run on host computers anywhere in the Internet. There are at least two DNS servers in each domain, and often more. The servers in each domain hold a partial map of the domain name tree below their domain. They must hold at least the portion consisting of all of the domain and host names within their domain, but they often contain a larger portion of the tree. DNS servers handle requests for the translation of domain names outside their portion of the tree by issuing requests to DNS servers in the relevant domains, proceeding recursively from right to left, resolving the name in segments. The resulting translation is then cached at the server handling the original request so that future requests for the resolution of names referring to the same domain will be resolved without reference to other servers. The DNS would not be workable without the extensive use of caching, since the 'root' name servers would be consulted in almost every case, creating a service access bottleneck.

### 3.4.8 Firewalls

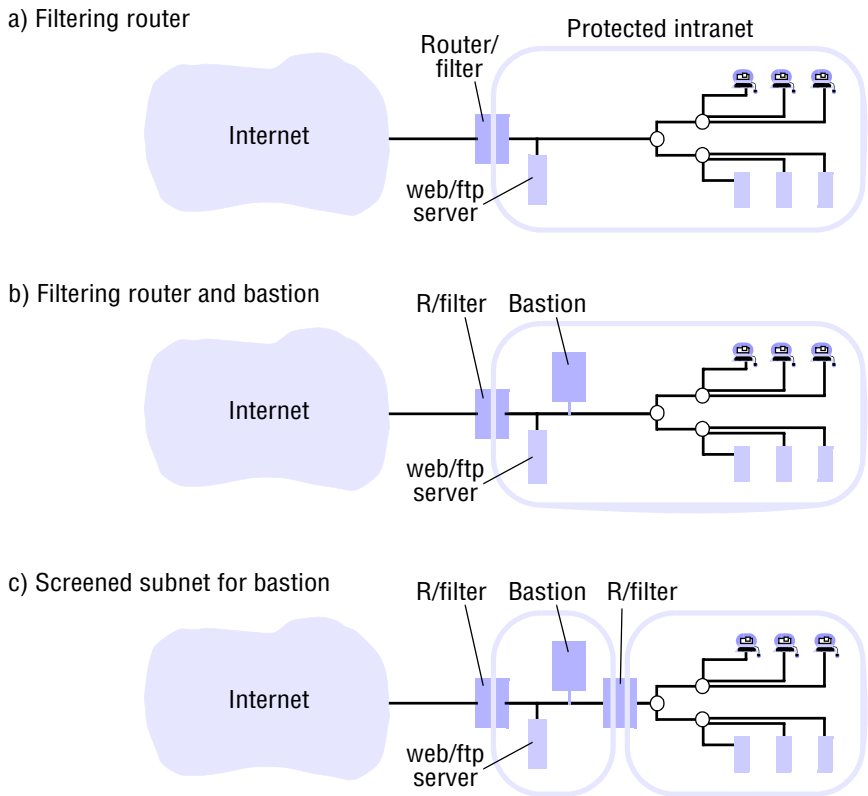
Almost all organizations need Internet connectivity in order to provide services to their customers and other external users and to enable their internal users to access information and services. The computers in most organizations are quite diverse, running a variety of operating systems and application software. The security of their software is even more varied; some of it may include state-of-the-art security, but much of it will have little or no capability to ensure that incoming communications can be trusted and outgoing communications are private when required. In summary, in an intranet with many computers and a wide range of software it is inevitable that some parts of the system will have weaknesses that expose it to security attacks. Forms of attack are detailed further in Chapter 11.

The purpose of a firewall is to monitor and control all communication into and out of an intranet. A firewall is implemented by a set of processes that act as a gateway to an intranet (Figure 3.21a), applying a security policy determined by the organization.

The aims of a firewall security policy may include any or all of the following:

*Service control:* To determine which services on internal hosts are accessible for external access and to reject all other incoming service requests. Outgoing service requests and the responses to them may also be controlled. These filtering actions can be based on the contents of IP packets and the TCP and UDP requests that they contain. For example, incoming HTTP requests may be rejected unless they are directed to an official web server host.

*Behaviour control:* To prevent behaviour that infringes the organization's policies, is antisocial or has no discernible legitimate purpose and is hence suspected of forming part of an attack. Some of these filtering actions may be applicable at the IP or TCP level, but others may require interpretation of messages at a higher level. For

**Figure 3.21** Firewall configurations

example, filtering of email ‘spam’ attacks may require examination of the sender’s email address in message headers or even the message contents.

*User control:* The organization may wish to discriminate between its users, allowing some to access external services but inhibiting others from doing so. An example of user control that is perhaps more socially acceptable than some is to prevent the acknowledging of software except to users who are members of the system administration team, in order to prevent virus infection or to maintain software standards. This particular example would in fact be difficult to implement without inhibiting the use of the Web by ordinary users.

Another instance of user control is the management of dialup and other connections provided for offsite users. If the firewall is also the host for modem connections, it can authenticate the user at connection time and can require the use of a secure channel for all communication (to prevent eavesdropping, masquerading and other attacks on the external connection). That is the purpose of the virtual private network technology described in the next subsection.

The policy has to be expressed in terms of filtering operations that are performed by filtering processes operating at several different levels:

*IP packet filtering:* This is a filter process examining individual IP packets. It may make decisions based on the destination and source addresses. It may also examine the *service type* field of IP packets and interpret the contents of the packets based on the type. For example, it may filter TCP packets based on the port number to which they are addressed, and since services are generally located at well-known ports, this enables packets to be filtered based on the service requested. For example, many sites prohibit the use of NFS servers by external clients.

For performance reasons, IP filtering is usually performed by a process within the operating system kernel of a router. If multiple firewalls are used, the first may mark certain packets for more exhaustive examination by a later firewall, allowing ‘clean’ packets to proceed. It is possible to filter based on sequences of IP packets, for example, to prevent access to an FTP server before a login has been performed.

*TCP gateway:* A TCP gateway process checks all TCP connection requests and segment transmissions. When a TCP gateway process is installed, the setting up of TCP connections can be controlled and TCP segments can be checked for correctness (some denial of service attacks use malformed TCP segments to disrupt client operating systems). When desired, they can be routed through an application-level gateway for content checking.

*Application-level gateway:* An application-level gateway process acts as a *proxy* for an application process. For example, a policy may be desired that allows certain internal users to make Telnet connections to certain external hosts. When a user runs a Telnet program on their local computer, it attempts to establish a TCP connection with a remote host. The request is intercepted by the TCP gateway. The TCP gateway starts a *Telnet proxy* process and the original TCP connection is routed to it. If the proxy approves the Telnet operation (i.e., if the user is authorized to use the requested host) it establishes another connection to the requested host and relays all of the TCP packets in both directions. A similar proxy process would run on behalf of each Telnet client, and similar proxies might be employed for FTP and other services.

A firewall is usually composed of several processes working at different protocol levels. It is common for firewall duties to be shared by more than one computer for performance and fault-tolerance reasons. In all of the configurations described below and illustrated in Figure 3.21, we show a public web and FTP server without protection. It holds only published information that requires no protection against public access, and its server software ensures that only authorized internal users can update it.

IP packet filtering is normally done by a router – a computer with at least two network addresses on separate IP networks – that runs an RIP process, an IP packet-filtering process and as few other processes as possible. The router/filter must run only trusted software in a manner that enables its enforcement of filtering policies to be guaranteed. This involves ensuring that no Trojan horse processes can run on it and that the filtering and routing software have not been modified or tampered with. Figure 3.21(a) shows a simple firewall configuration that relies only on IP filtering and employs a single router for that purpose. The network configuration in Figure 3.10 includes two router/filters acting as firewalls of this type for performance and reliability reasons. They both obey the same filtering policy and the second does not increase the security of the system.

When TCP and application-level gateway processes are required, these usually run on a separate computer, which is known as a *bastion*. (The term originates from the construction of fortified castles; it is a protruding watchtower from which the castle may be defended or defenders may negotiate with those desiring entry.) A bastion computer is a host that is located inside the intranet protected by an IP router/filter and runs the TCP and application-level gateways (Figure 3.21b). Like the router/filter, the bastion must run only trusted software. In a well-secured intranet, proxies must be used for access to all outside services. Readers may be familiar with the use of proxies for web access. These are an instance of the use of firewall proxies; they are often constructed in a manner that integrates a web cache server (described in Chapter 2). This and other proxies are likely to require substantial processing and storage resources.

Security can be enhanced by employing two router/filters in series, with the bastion and any public servers located on a separate subnet linking the router/filters (Figure 3.21c). This configuration has several security advantages:

- If the bastion policy is strict, the IP addresses of hosts in the intranet need not even be published to the outside world, and the addresses in the outside world need not be known to internal computers, since all external communication passes through proxy processes in the bastion, has access to both.
- If the first router/filter is penetrated or compromised, the second, which is invisible from outside the intranet and hence less vulnerable, remains to pick up and reject unacceptable IP packets.

**Virtual private networks** • Virtual private networks (VPNs) extend the firewall protection boundary beyond the local intranet by the use of cryptographically protected secure channels at the IP level. In Section 3.4.4, we outlined the IP security extensions available in IPv6 and IPv4 with IPSec tunnelling [Thayer 1998]. These are the basis for the implementation of VPNs. They may be used for individual external users or to implement secure connections between intranets located at different sites using public Internet links.

For example, a member of staff may need to connect to the organization's intranet via an Internet service provider. Once connected, they should have the same capabilities as a user inside the firewall. This can be achieved if their local host implements IP security. The local host holds one or more cryptographic keys that it shares with the firewall, and these are used to establish a secure channel at connection time. Secure channel mechanisms are described in detail in Chapter 11.

## 3.5 Case studies: Ethernet, WiFi and Bluetooth

---

Up to this point we have discussed the principles involved in the construction of computer networks and we have described IP, the 'virtual network layer' of the Internet. To complete the chapter, we describe the principles and implementations of three actual networks.

In the early 1980s, the US Institute of Electrical and Electronic Engineers (IEEE) established a committee to specify a series of standards for local area networks (the 802 Committee [IEEE 1990]), and its subcommittees have produced a series of



specifications that have become the key standards for LANs. In most cases, the standards are based on pre-existing industry standards that emerged from research done in the 1970s. The relevant subcommittees and the standards that they have been published to date are shown in Figure 3.22.

They differ in performance, efficiency, reliability and cost, but they all provide relatively high-bandwidth networking capabilities over short and medium distances. The IEEE 802.3 Ethernet standard has largely won the battle for the wired LAN marketplace, and we describe it in Section 3.5.1 as our representative wired LAN technology. Although Ethernet implementations are available for several bandwidths, the principles of operation are identical in all of them.

The IEEE 802.5 Token Ring standard was a significant competitor for much of the 1990s, offering advantages over Ethernet in terms of efficiency and its support for bandwidth guarantees, but it has now disappeared from the marketplace. Readers interested in a brief description of this interesting LAN technology can find one at [www.cdk5.net/networking](http://www.cdk5.net/networking). The widespread use of Ethernet switches (as opposed to hubs) has enabled Ethernets to be configured in a manner that offers bandwidth and latency guarantees (as discussed further in Section 3.5.1, subsection *Ethernet for real-time and quality of service critical applications*), and this is one reason for its displacement of token ring technology.

The IEEE 802.4 Token Bus standard was developed for industrial applications with real-time requirements and is employed in that domain. The IEEE 802.6 Metropolitan Area standard covers distances up to 50 km and is intended for use in networks that span towns and cities.

The IEEE 802.11 Wireless LAN standard emerged somewhat later but holds a major position in the marketplace with products from many vendors under the commercial name WiFi, and is installed in a large proportion of mobile and handheld computing devices. The IEEE 802.11 standard is designed to support communication at speeds up to 54 Mbps over distances of up to 150 m between devices equipped with simple wireless transmitter/receivers. We describe its principles of operation in Section 3.5.2. Further details on IEEE 802.11 networks can be found in Crow *et al.* [1997] and Kurose and Ross [2007].

The IEEE 802.15.1 Wireless Personal Area Network standard (Bluetooth) was based on a technology first developed in 1999 by the Ericsson company to transport low-

**Figure 3.22** IEEE 802 network standards

IEEE no.	Name	Title	Reference
802.3	Ethernet	CSMA/CD Networks (Ethernet)	[IEEE 1985a]
802.4		Token Bus Networks	[IEEE 1985b]
802.5		Token Ring Networks	[IEEE 1985c]
802.6		Metropolitan Area Networks	[IEEE 1994]
802.11	WiFi	Wireless Local Area Networks	[IEEE 1999]
802.15.1	Bluetooth	Wireless Personal Area Networks	[IEEE 2002]
802.15.4	ZigBee	Wireless Sensor Networks	[IEEE 2003]
802.16	WiMAX	Wireless Metropolitan Area Networks	[IEEE 2004a]



bandwidth digital voice and data between devices such as tablets, mobile phones and headsets and was subsequently standardized in 2002 as IEEE 802.15.1. Section 3.5.3 contains a description of Bluetooth.

IEEE 802.15.4 (ZigBee) is another WPAN standard aimed at providing data communication for very low-bandwidth low-energy devices in the home such as remote controls, burglar alarm and heating system sensors, and ubiquitous devices such as active badges and tag readers. Such networks are termed *wireless sensor networks* and their applications and communication characteristics are discussed in Chapter 19.

The IEEE 802.16 Wireless MAN standard (commercial name: WiMAX) was ratified in 2004 and 2005. The IEEE 802.16 standard is designed as an alternative to cable and DSL links for the ‘last mile’ connection to homes and offices. A variant of the standard is intended to supersede 802.11 WiFi networks as the main connection technology for laptop computers and mobile devices in outdoor and indoor public areas.

The ATM technology emerged from major research and standardization efforts in the telecommunications and computer industries in the late 1980s and early 1990s [CCITT 1990]. Its purpose is to provide a high-bandwidth wide area digital networking technology suitable for telephone, data and multimedia (high-quality audio and video) applications. Although the uptake has been slower than expected, ATM is now the dominant technology for very high speed wide area networking. It was also seen in some quarters as a replacement for Ethernet in LAN applications, but it has been less successful in that marketplace due to competition from 100 Mbps and 1000 Mbps Ethernets, which are available at much lower cost. Further details on ATM and on other high-speed network technologies can be found in Tanenbaum [2003] and Stallings [2002].

### 3.5.1 Ethernet

The Ethernet was developed at the Xerox Palo Alto Research Center in 1973 [Metcalfe and Boggs 1976; Shoch *et al.* 1982, 1985] as part of the programme of research carried out there on personal workstations and distributed systems. The pilot Ethernet was the first high-speed local network, demonstrating the feasibility and usefulness of high-speed local networks linking computers on a single site, allowing them to communicate at high transmission speeds with low error rates and without switching delays. The original prototype Ethernet ran at 3 Mbps. Ethernet systems are now available with bandwidths ranging from 10 Mbps to 1000 Mbps.

We shall describe the principles of operation of the 10 Mbps Ethernet specified in IEEE Standard 802.3 [IEEE 1985a]. This was the first widely deployed local area network technology. The 100 Mbps variant is now more commonly used; its principles of operation are identical. We conclude this section with a list of the more important variants of Ethernet transmission technology and bandwidth that are available. For comprehensive descriptions of the Ethernet in all its variations, see Spurgeon [2000].

A single Ethernet is a simple or branching bus-like connection line using a transmission medium consisting of one or more continuous segments of cable linked by hubs or repeaters. Hubs and repeaters are simple devices that link pieces of wire, enabling the same signals to pass through all of them. Several Ethernets can be linked at the Ethernet network protocol level by Ethernet switches or bridges. Switches and

bridges operate at the level of Ethernet frames, forwarding them to adjacent Ethernets when their destination is there. Linked Ethernets appear as a single network to higher protocol layers, such as IP (see Figure 3.10, where the IP subnets 138.37.88 and 138.37.94 are each composed of several Ethernets linked by components marked *Eswitch*). In particular, the ARP protocol (Section 3.4.2) is able to resolve IP addresses to Ethernet addresses across linked sets of Ethernets; each ARP request is broadcast on all of the linked networks in a subnet.

The method of operation of Ethernets is defined by the phrase ‘carrier sensing, multiple access with collision detection’ (abbreviated: CSMA/CD) and they belong to the class of *contention bus* networks. Contention buses use a single transmission medium to link all of the hosts. The protocol that manages access to the medium is called a *medium access control* (MAC) protocol. Because a single link connects all hosts, the MAC protocol combines the functions of a data link layer protocol (responsible for the transmission of packets on communication links) and a network protocol (responsible for delivery of packets to hosts) in a single protocol layer.

**Packet broadcasting** • The method of communication in CSMA/CD networks is by broadcasting packets of data on the transmission medium. All stations are continuously ‘listening’ to the medium for packets that are addressed to them. Any station wishing to transmit a message broadcasts one or more packets (called *frames* in the Ethernet specification) on the medium. Each packet contains the address of the destination station, the address of the sending station and a variable-length sequence of bits representing the message to be transmitted. Data transmission proceeds at 10 Mbps (or at the higher speeds specified for 100 and 1000 Mbps Ethernets) and packets vary in length between 64 and 1518 bytes, so the time required to transmit a packet on a 10 Mbps Ethernet is 50–1200 microseconds, depending on its length. The MTU is specified as 1518 bytes in the IEEE standard, although there is no technical reason for any particular fixed limit except the need to limit delays caused by contention.

The address of the destination station normally refers to a single network interface. Controller hardware at each station receives a copy of every packet. It compares the destination address in each packet with a wired-in local address, ignoring packets addressed to other stations and passing those with a matching address to the local host. The destination address may also specify a broadcast or a multicast address. Ordinary addresses are distinguished from broadcast and multicast addresses by their higher-order bit (0 and 1, respectively). An address consisting of all 1s is reserved for use as a broadcast address and is used when a message is to be received by all of the stations on the network. This is used, for example, to implement the ARP IP address resolution protocol. Any station that receives a packet with a broadcast address will pass it on to its local host. A multicast address specifies a limited form of broadcast that is received by a group of stations whose network interfaces have been configured to receive packets with that multicast address. Not all implementations of Ethernet network interfaces can recognize multicast addresses.

The Ethernet network protocol (providing for the transmission of Ethernet packets between pairs of hosts) is implemented in the Ethernet hardware interface; protocol software is required for the transport layer and those above it.

**Ethernet packet layout** • The packets (or more correctly, frames) transmitted by stations on the Ethernet have the following layout:

bytes: 7	1	6	6	2	46 < length < 1500	4
Preamble	S	Destination address	Source address	Length of data	Data for transmission	Checksum

Apart from the destination and source addresses already mentioned, frames include a fixed 8-byte prefix, a length field, a data field and a checksum. The prefix is used for hardware timing purposes and consists of a preamble of 7 bytes, each containing the bit pattern 10101010 followed by a single-byte start frame delimiter (S in the diagram) with the pattern 10101011.

Despite the fact that the specification does not allow more than 1024 stations on a single Ethernet, addresses occupy 6 bytes, providing  $2^{48}$  different addresses. This enables every Ethernet hardware interface to be given a unique address by its manufacturer, ensuring that all of the stations in any interconnected set of Ethernets will have unique addresses. The US Institute of Electrical and Electronic Engineers (IEEE) acts as an allocation authority for Ethernet addresses, allocating separate ranges of 48-bit addresses to the manufacturers of Ethernet hardware interfaces. These are referred to as MAC addresses, since they are used by the medium access control layer. In fact, MAC addresses allocated in this fashion have also been adopted as unique addresses for use in other network types in the IEEE 802 family, including 802.11 (WiFi) and 802.15.1 (Bluetooth).

The data field contains all or part (if the message length exceeds 1500 bytes) of the message that is being transmitted. The lower bound of 46 bytes on the data field ensures a minimum packet length of 64 bytes, which is necessary in order to guarantee that collisions will be detected by all stations on the network, as explained below.

The frame check sequence is a checksum generated and inserted by the sender and used to validate packets by the receiver. Packets with incorrect checksums are simply dropped by the data link layer in the receiving station. This is another example of the application of the end-to-end argument: to guarantee the transmission of a message, a transport-layer protocol such as TCP, which acknowledges receipt of each packet and retransmits any unacknowledged packets, must be used. The incidence of data corruption in local networks is so small that the use of this method of recovery when guaranteed delivery is required is entirely satisfactory and it enables a less costly transport protocol such as UDP to be employed when there is no need for delivery guarantees.

**Packet collisions** • Even in the relatively short time that it takes to transmit packets there is a finite probability that two stations on the network will attempt to transmit messages simultaneously. If a station attempts to transmit a packet without checking whether the medium is in use by other stations, a collision may occur.

The Ethernet has three mechanisms to deal with this possibility. The first is called *carrier sensing*: the interface hardware in each station listens for the presence of a signal (known as the *carrier* by analogy with radio broadcasting) in the medium. When a station wishes to transmit a packet, it waits until no signal is present in the medium and then begins to transmit.

Unfortunately, carrier sensing does not prevent all collisions. The possibility of collision remains due to the finite time  $\tau$  for a signal inserted at a point in the medium (travelling at electronic speed: approximately  $2 \times 10^8$  metres per second) to reach all other points. Consider two stations A and B that are ready to transmit packets at almost the same time. If A begins to transmit first, B can check and find no signal in the medium at any time  $t < \tau$  after A has begun to transmit. B then begins to transmit, *interfering* with A's transmission. Both A's packet and B's packet will be damaged by the interference.

The technique used to recover from such interference is called *collision detection*. Whenever a station is transmitting a packet through its hardware output port, it also listens on its input port and the two signals are compared. If they differ, then a collision has occurred. When this happens the station stops transmitting and produces a *jamming signal* to ensure that all stations recognize the collision. As we have already noted, a minimum packet length is necessary to ensure that collisions are always detected. If two stations transmit approximately simultaneously from opposite ends of the network, they will not become aware of the collision for  $2\tau$  seconds (because the first sender must be still transmitting when it receives the second signal). If the packets that they transmit take less than  $\tau$  to be broadcast, the collision will not be noticed, since each sending station would not see the other packet until after it has finished transmitting its own, whereas stations at intermediate points would receive both packets simultaneously, resulting in data corruption.

After the jamming signal, all transmitting and listening stations cancel the current packet. The transmitting stations then have to try to transmit their packets again. A further difficulty now arises. If the stations involved in the collision all attempt to retransmit their packets immediately after the jamming signal, another collision will probably occur. To avoid this, a technique known as *back-off* is used. Each of the stations involved in a collision chooses to wait a time  $n\tau$  before retransmitting. The value of  $n$  is a random integer chosen separately at each station and bounded by a constant  $L$  defined in the network software. If a further collision occurs, the value of  $L$  is doubled and the process is repeated if necessary for up to 10 attempts.

Finally, the interface hardware at the receiving station computes the check sequence and compares it with the checksum transmitted in the packet. Using all of these techniques, the stations connected to the Ethernet are able to manage the use of the medium without any centralized control or synchronization.

**Ethernet efficiency** • The efficiency of an Ethernet is the ratio of the number of packets transmitted successfully as a proportion of the theoretical maximum number that could be transmitted without collisions. It is affected by the value of  $\tau$ , since the interval of  $2\tau$  seconds after a packet transmission starts is the 'window of opportunity' for collisions – no collision can occur later than  $2\tau$  seconds after a packet starts to be transmitted. It is also affected by the number of stations on the network and their level of activity.

For a 1 km cable, the value of  $\tau$  is less than 5 microseconds and the probability of collisions is small enough to ensure high efficiency. The Ethernet can achieve a channel utilization of between 80 and 95%, although the delays due to contention become noticeable when 50% utilization is exceeded. Because the loading is variable, it is impossible to *guarantee* the delivery of a given message within any fixed time, since the network might be fully loaded when the message is ready for transmission. But the

Figure 3.23 Ethernet ranges and speeds

	10Base5	10BaseT	100BaseT	1000BaseT
Data rate	10 Mbps	10 Mbps	100 Mbps	1000 Mbps
<i>Max. segment lengths:</i>				
Twisted wire (UTP)	100 m	100 m	100 m	25 m
Coaxial cable (STP)	500 m	500 m	500 m	25 m
Multi-mode fibre	2000 m	2000 m	500 m	500 m
Mono-mode fibre	25000 m	25000 m	20000 m	2000 m

probability of transferring the message with a given delay is as good as, or better than, that of other network technologies.

Empirical measurements of the performance of an Ethernet at Xerox PARC, reported by Shoch and Hupp [1980], confirm this analysis. In practice, the load on Ethernets used in distributed systems varies quite widely. Many networks are used primarily for asynchronous client-server interactions, and these operate for most of the time with no stations waiting to transmit. Their low level of contention results in a channel utilization close to 1. Networks that support bulk data access for large numbers of users experience more load, and those that carry multimedia streams are liable to be overwhelmed if more than a few streams are transmitted concurrently.

**Physical implementations** • The description above defines the MAC-layer protocol for all Ethernets. Widespread adoption across a large marketplace has resulted in the availability of very low-cost controller hardware to perform the algorithms required for its implementation, and this is included as a standard part of many desktop and consumer computers.

A wide range of physical Ethernet implementations have been based on it to offer a variety of performance and cost trade-offs and to exploit increased hardware performance. The variations result from the use of different transmission media – coaxial cable, twisted copper wire (similar to telephone wiring) and optical fibre – with differing limits on transmission range, and from the use of higher signalling speeds, resulting in greater system bandwidth and generally shorter transmission ranges. The IEEE has adopted a number of standards for physical-layer implementations, and a naming scheme is used to distinguish them. Names such as 10Base5 and 100BaseT are used. They have the following form:

<R><B><L>

Where:  $R$  = data rate in Mbps

$B$  = medium signalling type (baseband or broadband)

$L$  = maximum segment length in metres/100 or T  
(twisted pair cable hierarchy)

We tabulate the bandwidth and maximum range of various currently available standard configurations and cable types in Figure 3.23. Configurations ending with the T designation are implemented with UTP cabling – unshielded twisted wires (telephone wiring) – and this is organized as a hierarchy of hubs with computers as the leaves of the tree. In that case, the segment lengths given in our table are twice the maximum permissible distance from a computer to a hub.

**Ethernet for real-time and quality of service critical applications** • It is often argued that the Ethernet MAC protocol is inherently unsuitable for real-time or quality of service critical applications because of its lack of a guaranteed delivery delay. But it should be noted that most Ethernet installations are now based on the use of MAC-level switches, as illustrated in Figure 3.10 and described in Section 3.3.7 (rather than hubs or cables with a tap for each connection, as was formerly the case). The use of switches throughout results in a separate segment for each host with no packets transmitted on it other than those addressed to that host. Hence if traffic to the host is from a single source, there is no contention for the medium – efficiency is 100% and latency is constant. The possibility of contention arises only at the switches, and these can be, and often are, designed to handle several packets concurrently. Hence a lightly loaded switched Ethernet installation approximates to 100% efficiency with a constant low latency, and they are therefore often successfully used in these critical application areas.

A further step towards real-time support for Ethernet-style MAC protocols is described in [Rether; Pradhan and Chiueh 1998] and a similar scheme is implemented in an open-source Linux extension [RTnet]. These software approaches address the contention problem by implementing an application-level cooperative protocol to reserve timeslots for the use of the medium. This protocol depends upon the cooperation of all the hosts connected to a segment.

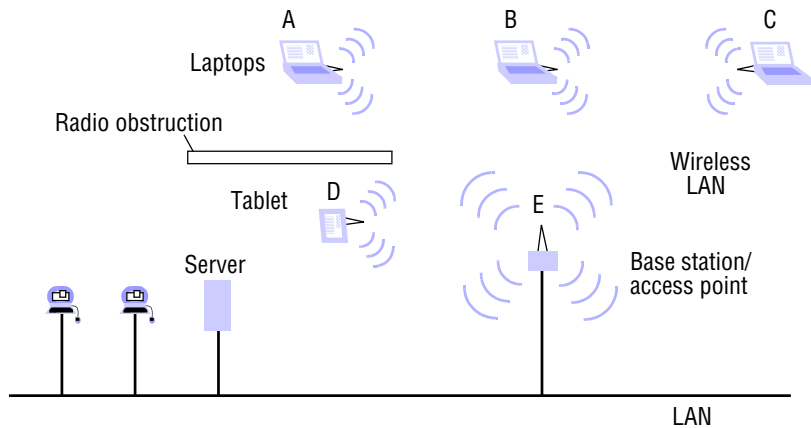
### 3.5.2 IEEE 802.11 (WiFi) wireless LAN

In this section, we summarize the special characteristics of wireless networking that must be addressed by a wireless LAN technology and explain how IEEE 802.11 addresses them. The IEEE 802.11 standard extends the carrier-sensing multiple access (CSMA) principle employed by Ethernet (IEEE 802.3) technology to suit the characteristics of wireless communication. The 802.11 standard is intended to support communication between computers located within about 150 metres of each other at speeds up to 54 Mbps.

Figure 3.24 illustrates a portion of an intranet including a wireless LAN. Several mobile wireless devices communicate with the rest of the intranet through a base station that is an *access point* to the wired LAN. A wireless network that connects to the world through an access point to a conventional LAN is known as an *infrastructure network*.

An alternative configuration for wireless networking is known as an *ad hoc network*. Ad hoc networks do not include an access point or base station. They are built ‘on the fly’ as a result of the mutual detection of two or more mobile devices with wireless interfaces in the same vicinity. An ad hoc network might occur, for example, when two or more laptop users in a room initiate a connection to any available station. They might then share files by launching a file server process on one of the machines.

At the physical level, IEEE 802.11 networks use radio frequency signals (in the licence-free 2.4 GHz and 5 GHz bands) or infrared signalling as the transmission medium. The radio version of the standard has received the most commercial attention, and we shall describe that. The IEEE 802.11b standard was the first variant to see widespread use. It operates in the 2.4 GHz band and supports data communication at up to 11 Mbps. It has been installed from 1999 onwards with base stations in many offices, homes and public places, enabling laptop computers and handheld devices to access local networked devices or the Internet. IEEE 802.11g is a more recent enhancement of

**Figure 3.24** Wireless LAN configuration

802.11b that uses the same 2.4 GHz band but a different signalling technique to achieve speeds up to 54 Mbps. Finally, the 802.11a variant works in the 5 GHz band and delivers a more certain 54 Mbps of bandwidth over a somewhat shorter range. All variants use various frequency-selection and frequency-hopping techniques to avoid external interference and mutual interference between independent wireless LANs, which we shall not detail here. We focus instead on the changes to the CSMA/CD mechanism that are needed in the MAC layer for all versions of 802.11 to enable broadcast data transmission to be used with radio transmission.

Like Ethernet, the 802.11 MAC protocol offers equal opportunities to all stations to use the transmission channel, and any station may transmit directly to any other. A MAC protocol controls the use of the channel by the various stations. As for the Ethernet, the MAC layer also performs the functions of both a data link layer and a network layer, delivering data packets to the hosts on a network.

Several problems arise from the use of radio waves rather than wires as the transmission medium. These problems stem from the fact that the carrier-sensing and collision-detection mechanisms employed in Ethernets are effective only when the strength of signals is approximately the same throughout a network.

We recall that the purpose of carrier sensing is to determine whether the medium is free at all points between the sending and receiving stations, and that of collision detection is to determine whether the medium in the vicinity of the receiver is free from interference during the transmission. Because signal strength is not uniform throughout the space in which wireless LANs operate, carrier detection and collision detection may fail in the following ways:

*Hidden stations:* Carrier sensing may fail to detect that another station on the network is transmitting. This is illustrated in Figure 3.24. If tablet D is transmitting to the base station E, laptop A may not be able to sense D's signal because of the radio obstruction shown. A might then start transmitting, causing a collision at E unless steps are taken to prevent this.



*Fading:* Due to the inverse square law of electromagnetic wave propagation, the strength of radio signals diminishes rapidly with the distance from the transmitter. Stations within a wireless LAN may be out of range of other stations in the same LAN. Thus in Figure 3.24, laptop A may not be able to detect a transmission by C, although each of them can transmit successfully to B or E. Fading defeats both carrier sensing and collision detection.

*Collision masking:* Unfortunately, the ‘listening’ technique used in the Ethernet to detect collisions is not very effective in radio networks. Because of the inverse square law referred to above, the locally generated signal will always be much stronger than any signal originating elsewhere, effectively drowning out the remote transmission. So, laptops A and C might both transmit simultaneously to E and neither would detect that collision, but E would receive only a garbled transmission.

Despite its fallibility, carrier sensing is not dispensed with in IEEE 802.11 networks; rather, it is augmented by the addition of a *slot reservation* mechanism to the MAC protocol. The resulting scheme is called *carrier sensing, multiple access with collision avoidance* (CSMA/CA).

When a station is ready to transmit, it senses the medium. If it detects no carrier signal it may assume that one of the following conditions is true:

1. The medium is available.
2. An out-of-range station is in the process of requesting a slot.
3. An out-of-range station is using a slot that it had previously reserved.

The slot-reservation protocol involves the exchange of a pair of short messages (frames) between the intending sender and the receiver. The first is a *request to send* (RTS) frame from the sender to the receiver. The RTS message specifies a duration for the slot requested. The receiver replies with a *clear to send* (CTS) frame, repeating the duration of the slot. The effect of this exchange is as follows:

- Stations within range of the sender will pick up the RTS frame and take note of the duration.
- Stations within range of the receiver will pick up the CTS frame and take note of the duration.

As a result, all of the stations within range of both the sender and the receiver will refrain from transmitting for the duration of the requested slot, leaving the channel free for the sender to transmit a data frame of the appropriate length. Finally, successful receipt of the data frame is acknowledged by the receiver to help deal with the problem of external interference with the channel. The slot-reservation feature of the MAC protocol helps to avoid collisions in these ways:

- The CTS frames help to avoid the hidden station and fading problems.
- The RTS and CTS frames are short, so the risk of collisions with them are low. If one is detected, or an RTS does not result in a CTS, a random back-off period is used, as in Ethernet.



- When the RTS and CTS frames have been correctly exchanged, there should be no collisions involving the subsequent data and acknowledgement frames unless intermittent fading prevented a third party from receiving either of them.

**Security** • The privacy and integrity of communication is an obvious concern for wireless networks. Any station that is within range and equipped with a receiver/transmitter might seek to join a network, or, failing that, it might eavesdrop on transmissions between other stations. The first attempt to address the security issues for 802.11 is entitled Wired Equivalent Privacy (WEP). Unfortunately, WEP is anything but what its name implies. Its security design was flawed in several ways that enabled it to be broken fairly easily. We describe its weaknesses and summarize the Wi-Fi Protected Access (WPA) system that succeeded it in Section 11.6.4.

### 3.5.3 IEEE 802.15.1 Bluetooth wireless PAN

Bluetooth is a wireless personal area network technology that emerged from the need to link mobile phones, laptop computers and other personal devices without wires. A special interest group (SIG) of mobile phone and computer manufacturers led by L.M. Ericsson developed a specification for a wireless personal area network (WPAN) for the transmission of digital voice streams as well as data [Haartsen *et al.* 1998]. Version 1.0 of the Bluetooth standard was published in 1999, borrowing its name from a Viking king. We describe Version 1.1 here. It was published in 2002 resolving some problems. The IEEE 802.15 Working Group then adopted it as standard 802.15.1 and published a specification for the physical and data link layers [IEEE 2002].

Bluetooth networks differ substantially from IEEE 802.11 (WiFi), the only other widely adopted wireless networking standard, in ways that reflect the different application requirements of WPANs and the different cost and energy consumption targets for which they are designed. Bluetooth aims to support very small, low-cost devices such as ear-mounted wireless headsets receiving digital audio streams from a mobile phone as well as interconnections between computers, phones, tablets and other mobile devices. The cost target was to add only five dollars to the cost of a handheld device and the energy target to utilize only a small fraction of the total battery power used by a phone or tablet, enabling operation for several hours even with lightweight batteries used in wearable devices such as headsets.

The intended applications require less bandwidth and a shorter transmission range than typical wireless LAN applications. This is fortunate because Bluetooth operates in the same crowded 2.4 GHz licence-free frequency band as WiFi networks, cordless phones and many emergency service communication systems. Transmission is at low energy, hopping at a rate of 1600 times per second between 79 1 MHz sub-bands of the permitted frequency band to minimize the effects of interference. The output power of normal Bluetooth devices is 1 milliwatt, giving a coverage of only 10 metres; 100 milliwatt devices with a range of up to 100 metres are permitted for applications such as home networks. Energy efficiency is further improved by the inclusion of an *adaptive range* facility, which adjusts the transmitted power to a lower level when partner devices are nearby (as determined by the strength of the signals initially received).

Bluetooth nodes associate dynamically in pairs with no prior knowledge required. The protocol for association is described below. After a successful association the

initiating node has the role of *master* and the other *slave*. A *Piconet* is a dynamically associated network composed of one master and up to seven active slaves. The master controls the use of the communication channel, allocating timeslots to each slave. A node that is in more than one Piconet can act as a bridge, enabling the masters to communicate – multiple Piconets linked in this fashion are termed a *scatternet*. Most types of device have the capacity to act as either master or slave.

All Bluetooth nodes are also equipped with a globally unique 48-bit MAC address (see Section 3.5.1), although it is only the master's MAC address that is used in the protocol. When a slave becomes active in a Piconet, it is assigned a temporary local address in the range 1 to 7 to reduce the length of packet headers. In addition to the seven active slaves, a Piconet may contain up to 255 *parked* nodes in low-power mode awaiting an activation signal from the master.

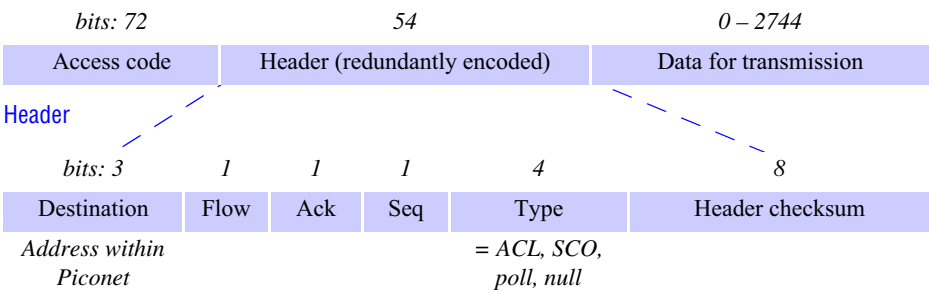
**Association protocol** • To conserve energy, devices remain in sleep or *standby* mode before any associations are made or when no recent communication has occurred. In standby mode they wake to listen for activation messages at intervals ranging from 0.64 to 2.56 seconds. To associate with a known nearby node (*parked*), the initiating node transmits a train of 16 *page* packets, on 16 frequency sub-bands, which may have to be repeated several times. To contact any unknown node within range, the initiator must first broadcast a train of *inquiry* messages. These transmission trains can occupy up to about 5 seconds in the worst case, leading to a maximum association time of 7–10 seconds.

Association is followed by an optional authentication exchange based on user-supplied or previously received authentication tokens, to ensure that the association is with the intended node and not an imposter. A slave then remains synchronized to the master by observing regularly transmitted packets from the master, even when they are not addressed to the slave. A slave that is inactive can be placed in parked mode by the master, freeing its slot in the Piconet for use by another node.

The requirement to support synchronous communication channels with adequate quality of service for the transmission of two-way real-time audio (for example, between a phone and its owner's wireless headset) as well as asynchronous communication for data exchange dictated a network architecture very different from the best-effort multiple-access design of Ethernet and WiFi networks. Synchronous communication is achieved by the use of a simple two-way communication protocol between a master and one of its slaves, termed a *synchronous connection oriented* (SCO) link on which master and slave must send alternating synchronized packets. Asynchronous communication is achieved by an *asynchronous connection-less* (ACL) link on which the master sends asynchronous poll packets to its slaves periodically and the slaves transmit only after receiving a poll.

All variants of the Bluetooth protocol use frames that fit within the structure shown in Figure 3.25. Once a Piconet has been established, the *access code* consists of a fixed preamble to synchronize the sender and receiver and identify the start of a slot, followed by a code derived from the master's MAC address that uniquely identifies the Piconet. The latter ensures that frames are correctly routed in situations where there are multiple overlapping Piconets. Because the medium is likely to be noisy and real-time communication cannot rely on retransmission, each bit in the header is transmitted in triplicate to provide redundancy for both the information and the checksum parts.

Figure 3.25 Bluetooth frame structure



SCO packets (e.g., for voice data) have a 240-bit payload containing 80 bits of data triplicated, filling exactly one timeslot.

The address field is just 3 bits to allow addressing to any of the seven currently active slaves. A zero address from the master indicates a broadcast. There are single-bit fields for flow control, acknowledgement and sequence numbering. The flow-control bit is used by a slave to indicate to the master that its buffers are full; the master should await a frame with a non-zero acknowledgement bit from the slave. The sequence number bit is inverted on each new frame sent to the same node; this enables duplicate (i.e., retransmitted) frames to be detected.

SCO links are used in time-critical applications such as the transmission of a two-way voice conversation. Packets must be short to keep the latency low, and there is little purpose in reporting or retransmitting corrupted packets in such applications since the retransmitted data would arrive too late to be useful. So the SCO protocol uses a simple, highly redundant protocol in which 80 bits of voice data are normally transmitted in triplicate to produce a 240-bit payload. Any two matching 80-bit replicas are taken as valid.

On the other hand, ACL links are used for data-transfer applications such as address book synchronization between a computer and a phone with a larger payload. The payload is not replicated but may contain an internal checksum that is checked at the application level, and in the case of failure retransmission can be requested.

Data is transmitted in packets occupying timeslots of 625 microseconds clocked and allocated by the master node. Each packet is transmitted on a different frequency in a hopping sequence defined by the master node. Because these slots are not large enough to allow a substantial payload, frames may be extended to occupy one, three or five slots. These characteristics and the underlying physical transmission method result in a maximum total throughput of 1 Mbps for a Piconet, accommodating up to three synchronous duplex channels of 64 Kpbs between a master and its slaves or a channel for asynchronous data transfer at rates up to 723 Kbps. These throughputs are calculated for the most redundant version of the SCO protocol, as described above. Other protocol variants are defined that trade the robustness and simplicity (and therefore low computational cost) of triplicated data for higher throughput.

Unlike most network standards, Bluetooth includes specifications (called *profiles*) for several application-level protocols, some of which are very specific to particular applications. The purpose of these profiles is to increase the likelihood that devices

manufactured by different vendors will interwork. Thirteen application profiles are covered: generic access, service discovery, serial port, generic object exchange, LAN access, dialup networking, fax, cordless telephony, intercom, headset, object push, file transfer and synchronization. Others are in preparation, including ambitious attempts to transmit high-quality music and even video over Bluetooth.

Bluetooth occupies a special niche in the range of wireless local networks. It achieves its ambitious design goal of supporting synchronous real-time audio communication with satisfactory quality of service (see Chapter 20 for further discussion of quality of service issues) as well as asynchronous data transfer using very low cost, compact and portable hardware, low power and very limited bandwidth.

Its principal limitation is the time taken (up to 10 seconds) for association of new devices. This impedes its use for certain applications, especially where devices are moving relative to each other, preventing its use, for example, to pay road tolls or to transmit promotional information to mobile phone users as they pass a store. A useful further reference on Bluetooth networking is the book by Bray and Sturman [2002].

Version 2.0 of the Bluetooth standard, with data throughputs up to 3 Mbps – sufficient to carry CD-quality audio – was released in 2004. Other improvements included a faster association mechanism and larger Piconet addresses. Versions 3 and 4 of the standard were under development at the time of writing. Version 3 integrates a Bluetooth control protocol with a WiFi data transfer layer to achieve throughputs up to 24 Mbps. Version 4 is under development as an ultra-low power Bluetooth technology for devices requiring a very long battery life.

## 3.6 Summary

We have focused here on the networking concepts and techniques that are needed as a basis for distributed systems, approaching them from the point of view of a distributed system designer. Packet networks and layered protocols provide the basis for communication in distributed systems. Local area networks are based on packet broadcasting on a shared medium; Ethernet is the dominant technology. Wide area networks are based on packet switching to route packets to their destinations through a connected network. Routing is a key mechanism and a variety of routing algorithms are used, of which the distance-vector method is the most basic but effective. Congestion control is needed to prevent overflow of buffers at the receiver and at intermediate nodes.

Internetworks are constructed by layering a ‘virtual’ internetwork protocol over collections of networks linked together by routers. The Internet TCP/IP protocols enable computers in the Internet to communicate with one another in a uniform manner, irrespective of whether they are on the same local area network or in different countries. The Internet standards include many application-level protocols that are suitable for use in wide area distributed applications. IPv6 has the much larger address space needed for the future evolution of the Internet and provision for new application requirements such as quality of service and security.

Mobile users are supported by MobileIP for wide area roaming and by wireless LANs based on IEEE 802 standards for local connectivity.

## EXERCISES

- 3.1 A client sends a 200 byte request message to a service, which produces a response containing 5000 bytes. Estimate the total time required to complete the request in each of the following cases, with the performance assumptions listed below:
- i) using connectionless (datagram) communication (for example, UDP);
  - ii) using connection-oriented communication (for example, TCP);
  - iii) when the server process is in the same machine as the client.
- [Latency per packet (local or remote,  
incurred on both send and receive): 5 ms  
Connection setup time (TCP only): 5 ms  
Data transfer rate: 10 Mbps  
MTU: 1000 bytes  
Server request processing time: 2 ms  
Assume that the network is lightly loaded.]
- pages 82, 122*
- 3.2 The Internet is far too large for any router to hold routing information for all destinations. How does the Internet routing scheme deal with this issue? *pages 98, 114*
- 3.3 What is the task of an Ethernet switch? What tables does it maintain? *pages 105, 130*
- 3.4 Make a table similar to Figure 3.5 describing the work done by the software in each protocol layer when Internet applications and the TCP/IP suite are implemented over an Ethernet. *pages 94, 122, 130*
- 3.5 How has the end-to-end argument [Saltzer *et al.* 1984] been applied to the design of the Internet? Consider how the use of a virtual circuit network protocol in place of IP would impact the feasibility of the World Wide Web. *pages 61, 96, 106, [www.reed.com](http://www.reed.com)*
- 3.6 Can we be sure that no two computers in the Internet have the same IP address? *page 108*
- 3.7 Compare connectionless (UDP) and connection-oriented (TCP) communication for the implementation of each of the following application-level or presentation-level protocols:
- i) virtual terminal access (for example, Telnet);
  - ii) file transfer (for example, FTP);
  - iii) user location (for example, rwho, finger);
  - iv) information browsing (for example, HTTP);
  - v) remote procedure call.
- page 122*
- 3.8 Explain how it is possible for a sequence of packets transmitted through a wide area network to arrive at their destination in an order that differs from that in which they were sent. Why can't this happen in a local network? *pages 97, 131*

- 3.9 A specific problem that must be solved in remote terminal access protocols such as Telnet is the need to transmit exceptional events such as ‘kill signals’ from the ‘terminal’ to the host in advance of previously transmitted data. Kill signals should reach their destination ahead of any other ongoing transmissions. Discuss the solution of this problem with connection-oriented and connectionless protocols. *page 122*
- 3.10 What are the disadvantages of using network-level broadcasting to locate resources:
- i) in a single Ethernet?
  - ii) in an intranet?
- To what extent is Ethernet multicast an improvement on broadcasting? *page 130*
- 3.11 Suggest a scheme that improves on MobileIP for providing access to a web server on a mobile device that is sometimes connected to the Internet by the mobile phone network and at other times has a wired connection to the Internet at one of several locations. *page 120*
- 3.12 Show the sequence of changes to the routing tables in Figure 3.8 that will occur (according to the RIP algorithm given in Figure 3.9) after the link labelled 3 in Figure 3.7 is broken. *pages 98–101*
- 3.13 Use the diagram in Figure 3.13 as a basis for an illustration showing the segmentation and encapsulation of an HTTP request to a server and the resulting reply. Assume that the request is a short HTTP message, but the reply includes at least 2000 bytes of HTML. *page 93, 107*
- 3.14 Consider the use of TCP in a Telnet remote terminal client. How should the keyboard input be buffered at the client? Investigate Nagle’s and Clark’s algorithms [Nagle 1984, Clark 1982] for flow control and compare them with the simple algorithm described on page 103 when TCP is used by:
- a) a web server,
  - b) a Telnet application,
  - c) a remote graphical application with continuous mouse input.
- pages 102, 123*
- 3.15 Construct a network diagram similar to Figure 3.10 for the local network at your institution or company. *page 104*
- 3.16 Describe how you would configure a firewall to protect the local network at your institution or company. What incoming and outgoing requests should it intercept? *page 125*
- 3.17 How does a newly installed personal computer connected to an Ethernet discover the IP addresses of local servers? How does it translate them to Ethernet addresses? *page 111*
- 3.18 Can firewalls prevent denial of service attacks such as the one described on page 112? What other methods are available to deal with such attacks? *page 112, 125*

*This page intentionally left blank*

# 4

## INTERPROCESS COMMUNICATION

- 4.1 Introduction
- 4.2 The API for the Internet protocols
- 4.3 External data representation and marshalling
- 4.4 Multicast communication
- 4.5 Network virtualization: Overlay networks
- 4.6 Case study: MPI
- 4.7 Summary

This chapter is concerned with the characteristics of protocols for communication between processes in a distributed system – that is, interprocess communication.

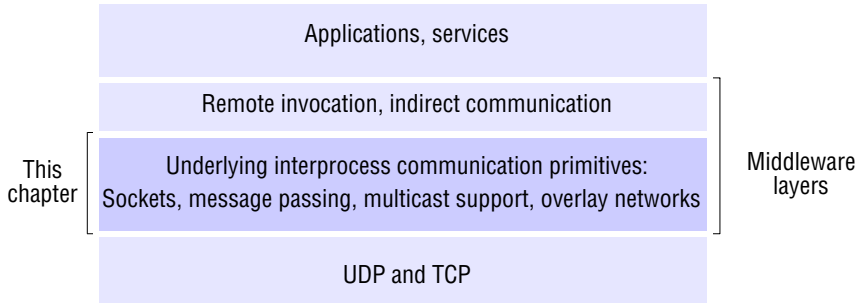
Interprocess communication in the Internet provides both datagram and stream communication. The Java APIs for these are presented, together with a discussion of their failure models. They provide alternative building blocks for communication protocols. This is complemented by a study of protocols for the representation of collections of data objects in messages and of references to remote objects. Together, these services offer support for the construction of higher-level communication services, as discussed in the following two chapters.

The interprocess communication primitives discussed above all support point-to-point communication, yet it is equally useful to be able to send a message from one sender to a group of receivers. The chapter also considers multicast communication, including IP multicast and the key concepts of reliability and ordering of messages in multicast communication.

Multicast is an important requirement for distributed applications and must be provided even if underlying support for IP multicast is not available. This is typically provided by an overlay network constructed on top of the underlying TCP/IP network. Overlay networks can also provide support for file sharing, enhanced reliability and content distribution.

The Message Passing Interface (MPI) is a standard developed to provide an API for a set of message-passing operations with synchronous and asynchronous variants.



**Figure 4.1** Middleware layers

## 4.1 Introduction

This and the next two chapters are concerned with the communication aspects of middleware, although the principles discussed are more widely applicable. This one is concerned with the design of the components shown in the darker layer in Figure 4.1. The layer above it is discussed in Chapter 5, which examines remote invocation, and Chapter 6, which is concerned with indirect communications paradigms.

Chapter 3 discussed the Internet transport-level protocols UDP and TCP without saying how middleware and application programs could use these protocols. The next section of this chapter introduces the characteristics of interprocess communication and then discusses UDP and TCP from a programmer's point of view, presenting the Java interface to each of these two protocols, together with a discussion of their failure models.

The application program interface to UDP provides a *message passing* abstraction – the simplest form of interprocess communication. This enables a sending process to transmit a single message to a receiving process. The independent packets containing these messages are called *datagrams*. In the Java and UNIX APIs, the sender specifies the destination using a socket – an indirect reference to a particular port used by the destination process at a destination computer.

The application program interface to TCP provides the abstraction of a two-way *stream* between pairs of processes. The information communicated consists of a stream of data items with no message boundaries. Streams provide a building block for producer-consumer communication. A producer and a consumer form a pair of processes in which the role of the first is to produce data items and the role of the second is to consume them. The data items sent by the producer to the consumer are queued on arrival at the receiving host until the consumer is ready to receive them. The consumer must wait when no data items are available. The producer must wait if the storage used to hold the queued data items is exhausted.

Section 4.3 is concerned with how the objects and data structures used in application programs can be translated into a form suitable for sending messages over the network, taking into account the fact that different computers may use different representations for simple data items. It also discusses a suitable representation for object references in a distributed system.

Section 4.4 discusses multicast communication: a form of interprocess communication in which one process in a group of processes transmits the same message to all members of the group. After explaining IP multicast, the section discusses the need for more reliable forms of multicast.

Section 4.5 examines the increasingly important topic of overlay networks. An overlay network is a network that is built over another network to permit applications to route messages to destinations not specified by an IP address. Overlay networks can enhance TCP/IP networks by providing alternative, more specialized network services. They are important in supporting multicast communication and peer-to-peer communication.

Finally, Section 4.6 presents a case study of a significant message-passing service, MPI, developed by the high-performance computing community.

## 4.2 The API for the Internet protocols

---

In this section, we discuss the general characteristics of interprocess communication and then discuss the Internet protocols as an example, explaining how programmers can use them, either by means of UDP messages or through TCP streams.

Section 4.2.1 revisits the message communication operations *send* and *receive* introduced in Section 2.3.2, with a discussion of how they synchronize with one another and how message destinations are specified in a distributed system. Section 4.2.2 introduces *sockets*, which are used in the application programming interface to UDP and TCP. Section 4.2.3 discusses UDP and its API in Java. Section 4.2.4 discusses TCP and its API in Java. The APIs for Java are object oriented but are similar to the ones designed originally in the Berkeley BSD 4.x UNIX operating system; a case study on the latter is available on the web site for the book [[www.cdk5.net/ipc](http://www.cdk5.net/ipc)]. Readers studying the programming examples in this section should consult the online Java documentation or Flanagan [2002] for the full specification of the classes discussed, which are in the package *java.net*.

### 4.2.1 The characteristics of interprocess communication

Message passing between a pair of processes can be supported by two message communication operations, *send* and *receive*, defined in terms of destinations and messages. To communicate, one process sends a message (a sequence of bytes) to a destination and another process at the destination receives the message. This activity involves the communication of data from the sending process to the receiving process and may involve the synchronization of the two processes. Section 4.2.3 gives definitions for the *send* and *receive* operations in the Java API for the Internet protocols, with a further case study of message passing (MPI) offered in Section 4.6.

**Synchronous and asynchronous communication** • A queue is associated with each message destination. Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues. Communication between the

sending and receiving processes may be either synchronous or asynchronous. In the *synchronous* form of communication, the sending and receiving processes synchronize at every message. In this case, both *send* and *receive* are *blocking* operations. Whenever a *send* is issued the sending process (or thread) is blocked until the corresponding *receive* is issued. Whenever a *receive* is issued by a process (or thread), it blocks until a message arrives.

In the *asynchronous* form of communication, the use of the *send* operation is *non-blocking* in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process. The *receive* operation can have blocking and non-blocking variants. In the non-blocking variant, the receiving process proceeds with its program after issuing a *receive* operation, which provides a buffer to be filled in the background, but it must separately receive notification that its buffer has been filled, by polling or interrupt.

In a system environment such as Java, which supports multiple threads in a single process, the blocking *receive* has no disadvantages, for it can be issued by one thread while other threads in the process remain active, and the simplicity of synchronizing the receiving threads with the incoming message is a substantial advantage. Non-blocking communication appears to be more efficient, but it involves extra complexity in the receiving process associated with the need to acquire the incoming message out of its flow of control. For these reasons, today's systems do not generally provide the non-blocking form of *receive*.

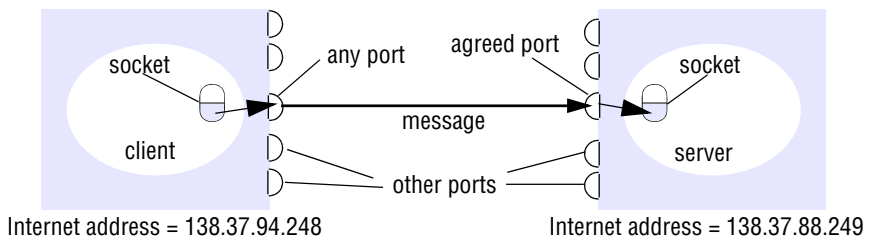
**Message destinations** • Chapter 3 explains that in the Internet protocols, messages are sent to (*Internet address, local port*) pairs. A local port is a message destination within a computer, specified as an integer. A port has exactly one receiver (multicast ports are an exception, see Section 4.5.1) but can have many senders. Processes may use multiple ports to receive messages. Any process that knows the number of a port can send a message to it. Servers generally publicize their port numbers for use by clients.

If the client uses a fixed Internet address to refer to a service, then that service must always run on the same computer for its address to remain valid. This can be avoided by using the following approach to providing location transparency:

- Client programs refer to services by name and use a name server or binder (see Section 5.4.2) to translate their names into server locations at runtime. This allows services to be relocated but not to migrate – that is, to be moved while the system is running.

**Reliability** • Chapter 2 defines reliable communication in terms of validity and integrity. As far as the validity property is concerned, a point-to-point message service can be described as reliable if messages are guaranteed to be delivered despite a 'reasonable' number of packets being dropped or lost. In contrast, a point-to-point message service can be described as unreliable if messages are not guaranteed to be delivered in the face of even a single packet dropped or lost. For integrity, messages must arrive uncorrupted and without duplication.

**Ordering** • Some applications require that messages be delivered in *sender order* – that is, the order in which they were transmitted by the sender. The delivery of messages out of sender order is regarded as a failure by such applications.

**Figure 4.2** Sockets and ports

## 4.2.2 Sockets

Both forms of communication (UDP and TCP) use the *socket* abstraction, which provides an endpoint for communication between processes. Sockets originate from BSD UNIX but are also present in most other versions of UNIX, including Linux as well as Windows and the Macintosh OS. Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process, as illustrated in Figure 4.2. For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs. Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number. Processes may use the same socket for sending and receiving messages. Each computer has a large number ( $2^{16}$ ) of possible port numbers for use by local processes for receiving messages. Any process may make use of multiple ports to receive messages, but a process cannot share ports with other processes on the same computer. (Processes using IP multicast are an exception in that they do share ports – see Section 4.4.1.) However, any number of processes may send messages to the same port. Each socket is associated with a particular protocol – either UDP or TCP.

**Java API for Internet addresses** • As the IP packets underlying UDP and TCP are sent to Internet addresses, Java provides a class, *InetAddress*, that represents Internet addresses. Users of this class refer to computers by Domain Name System (DNS) hostnames (see Section 3.4.7). For example, instances of *InetAddress* that contain Internet addresses can be created by calling a static method of *InetAddress*, giving a DNS hostname as the argument. The method uses the DNS to get the corresponding Internet address. For example, to get an object representing the Internet address of the host whose DNS name is *bruno.dcs.qmul.ac.uk*, use:

```
InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmul.ac.uk");
```

This method can throw an *UnknownHostException*. Note that the user of the class does not need to state the explicit value of an Internet address. In fact, the class encapsulates the details of the representation of Internet addresses. Thus the interface for this class is not dependent on the number of bytes needed to represent Internet addresses – 4 bytes in IPv4 and 16 bytes in IPv6.

### 4.2.3 UDP datagram communication

A datagram sent by UDP is transmitted from a sending process to a receiving process without acknowledgement or retries. If a failure occurs, the message may not arrive. A datagram is transmitted between processes when one process *sends* it and another *receives* it. To send or receive messages a process must first create a socket bound to an Internet address of the local host and a local port. A server will bind its socket to a *server port* – one that it makes known to clients so that they can send messages to it. A client binds its socket to any free local port. The *receive* method returns the Internet address and port of the sender, in addition to the message, allowing the recipient to send a reply.

The following are some issues relating to datagram communication:

*Message size:* The receiving process needs to specify an array of bytes of a particular size in which to receive a message. If the message is too big for the array, it is truncated on arrival. The underlying IP protocol allows packet lengths of up to  $2^{16}$  bytes, which includes the headers as well as the message. However, most environments impose a size restriction of 8 kilobytes. Any application requiring messages larger than the maximum must fragment them into chunks of that size. Generally, an application, for example DNS, will decide on a size that is not excessively large but is adequate for its intended use.

*Blocking:* Sockets normally provide non-blocking *sends* and blocking *receives* for datagram communication (a non-blocking *receive* is an option in some implementations). The *send* operation returns when it has handed the message to the underlying UDP and IP protocols, which are responsible for transmitting it to its destination. On arrival, the message is placed in a queue for the socket that is bound to the destination port. The message can be collected from the queue by an outstanding or future invocation of *receive* on that socket. Messages are discarded at the destination if no process already has a socket bound to the destination port.

The method *receive* blocks until a datagram is received, unless a timeout has been set on the socket. If the process that invokes the *receive* method has other work to do while waiting for the message, it should arrange to use a separate thread. Threads are discussed in Chapter 7. For example, when a server receives a message from a client, the message may specify work to do, in which case the server will use separate threads to do the work and to wait for messages from other clients.

*Timeouts:* The *receive* that blocks forever is suitable for use by a server that is waiting to receive requests from its clients. But in some programs, it is not appropriate that a process that has invoked a *receive* operation should wait indefinitely in situations where the sending process may have crashed or the expected message may have been lost. To allow for such requirements, timeouts can be set on sockets. Choosing an appropriate timeout interval is difficult, but it should be fairly large in comparison with the time required to transmit a message.

*Receive from any:* The *receive* method does not specify an origin for messages. Instead, an invocation of *receive* gets a message addressed to its socket from any origin. The *receive* method returns the Internet address and local port of the sender, allowing the recipient to check where the message came from. It is possible to connect a datagram socket to a particular remote port and Internet address, in which case the socket is only able to send messages to and receive messages from that address.

**Failure model for UDP datagrams** • Chapter 2 presents a failure model for communication channels and defines reliable communication in terms of two properties: integrity and validity. The integrity property requires that messages should not be corrupted or duplicated. The use of a checksum ensures that there is a negligible probability that any message received is corrupted. UDP datagrams suffer from the following failures:

*Omission failures:* Messages may be dropped occasionally, either because of a checksum error or because no buffer space is available at the source or destination. To simplify the discussion, we regard send-omission and receive-omission failures (see Figure 2.15) as omission failures in the communication channel.

*Ordering:* Messages can sometimes be delivered out of sender order.

Applications using UDP datagrams are left to provide their own checks to achieve the quality of reliable communication they require. A reliable delivery service may be constructed from one that suffers from omission failures by the use of acknowledgements. Section 5.2 discusses how reliable request-reply protocols for client-server communication may be built over UDP.

**Use of UDP** • For some applications, it is acceptable to use a service that is liable to occasional omission failures. For example, the Domain Name System, which looks up DNS names in the Internet, is implemented over UDP. Voice over IP (VOIP) also runs over UDP. UDP datagrams are sometimes an attractive choice because they do not suffer from the overheads associated with guaranteed message delivery. There are three main sources of overhead:

- the need to store state information at the source and destination;
- the transmission of extra messages;
- latency for the sender.

The reasons for these overheads are discussed in Section 4.2.4.

**Java API for UDP datagrams** • The Java API provides datagram communication by means of two classes: *DatagramPacket* and *DatagramSocket*.

*DatagramPacket:* This class provides a constructor that makes an instance out of an array of bytes comprising a message, the length of the message and the Internet address and local port number of the destination socket, as follows:

*Datagram packet*

array of bytes containing message	length of message	Internet address	port number
-----------------------------------	-------------------	------------------	-------------

An instance of *DatagramPacket* may be transmitted between processes when one process *sends* it and another *receives* it.

This class provides another constructor for use when receiving a message. Its arguments specify an array of bytes in which to receive the message and the length of the array. A received message is put in the *DatagramPacket* together with its length and the Internet address and port of the sending socket. The message can be retrieved from the *DatagramPacket* by means of the method *getData*. The methods *getPort* and *getAddress* access the port and Internet address.

**Figure 4.3** UDP client sends a message to the server and gets a reply

```

import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
        // args give message contents and server hostname
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request =
                new DatagramPacket(m, m.length(), aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
        } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        } catch (IOException e){System.out.println("IO: " + e.getMessage());}
        } finally { if(aSocket != null) aSocket.close();}
    }
}

```

---

*DatagramSocket*: This class supports sockets for sending and receiving UDP datagrams. It provides a constructor that takes a port number as its argument, for use by processes that need to use a particular port. It also provides a no-argument constructor that allows the system to choose a free local port. These constructors can throw a *SocketException* if the chosen port is already in use or if a reserved port (a number below 1024) is specified when running over UNIX.

The class *DatagramSocket* provides methods that include the following:

*send* and *receive*: These methods are for transmitting datagrams between a pair of sockets. The argument of *send* is an instance of *DatagramPacket* containing a message and its destination. The argument of *receive* is an empty *DatagramPacket* in which to put the message, its length and its origin. The methods *send* and *receive* can throw *IOExceptions*.

*setSoTimeout*: This method allows a timeout to be set. With a timeout set, the *receive* method will block for the time specified and then throw an *InterruptedIOException*.

*connect*: This method is used for connecting to a particular remote port and Internet address, in which case the socket is only able to send messages to and receive messages from that address.

**Figure 4.4** UDP server repeatedly receives a request and sends it back to the client

```

import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        } catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        } finally {if (aSocket != null) aSocket.close();}
    }
}

```

Figure 4.3 shows the program for a client that creates a socket, sends a message to a server at port 6789 and then waits to receive a reply. The arguments of the *main* method supply a message and the DNS hostname of the server. The message is converted to an array of bytes, and the DNS hostname is converted to an Internet address. Figure 4.4 shows the program for the corresponding server, which creates a socket bound to its server port (6789) and then repeatedly waits to receive a request message from a client, to which it replies by sending back the same message.

#### 4.2.4 TCP stream communication

The API to the TCP protocol, which originates from BSD 4.x UNIX, provides the abstraction of a stream of bytes to which data may be written and from which data may be read. The following characteristics of the network are hidden by the stream abstraction:

*Message sizes:* The application can choose how much data it writes to a stream or reads from it. It may deal in very small or very large sets of data. The underlying implementation of a TCP stream decides how much data to collect before transmitting it as one or more IP packets. On arrival, the data is handed to the application as requested. Applications can, if necessary, force data to be sent immediately.

*Lost messages:* The TCP protocol uses an acknowledgement scheme. As an example of a simple scheme (which is not used in TCP), the sending end keeps a record of each



IP packet sent and the receiving end acknowledges all the arrivals. If the sender does not receive an acknowledgement within a timeout, it retransmits the message. The more sophisticated sliding window scheme [Comer 2006] cuts down on the number of acknowledgement messages required.

*Flow control:* The TCP protocol attempts to match the speeds of the processes that read from and write to a stream. If the writer is too fast for the reader, then it is blocked until the reader has consumed sufficient data.

*Message duplication and ordering:* Message identifiers are associated with each IP packet, which enables the recipient to detect and reject duplicates, or to reorder messages that do not arrive in sender order.

*Message destinations:* A pair of communicating processes establish a connection before they can communicate over a stream. Once a connection is established, the processes simply read from and write to the stream without needing to use Internet addresses and ports. Establishing a connection involves a *connect* request from client to server followed by an *accept* request from server to client before any communication can take place. This could be a considerable overhead for a single client-server request and reply.

The API for stream communication assumes that when a pair of processes are establishing a connection, one of them plays the client role and the other plays the server role, but thereafter they could be peers. The client role involves creating a stream socket bound to any port and then making a *connect* request asking for a connection to a server at its server port. The server role involves creating a listening socket bound to a server port and waiting for clients to request connections. The listening socket maintains a queue of incoming connection requests. In the socket model, when the server *accepts* a connection, a new stream socket is created for the server to communicate with a client, meanwhile retaining its socket at the server port for listening for *connect* requests from other clients.

The pair of sockets in the client and server are connected by a pair of streams, one in each direction. Thus each socket has an input stream and an output stream. One of the pair of processes can send information to the other by writing to its output stream, and the other process obtains the information by reading from its input stream.

When an application *closes* a socket, this indicates that it will not write any more data to its output stream. Any data in the output buffer is sent to the other end of the stream and put in the queue at the destination socket, with an indication that the stream is broken. The process at the destination can read the data in the queue, but any further reads after the queue is empty will result in an indication of end of stream. When a process exits or fails, all of its sockets are eventually closed and any process attempting to communicate with it will discover that its connection has been broken.

The following are some outstanding issues related to stream communication:

*Matching of data items:* Two communicating processes need to agree as to the contents of the data transmitted over a stream. For example, if one process writes an *int* followed by a *double* to a stream, then the reader at the other end must read an *int* followed by a *double*. When a pair of processes do not cooperate correctly in their use of a stream, the reading process may experience errors when interpreting the data or may block due to insufficient data in the stream.

*Blocking:* The data written to a stream is kept in a queue at the destination socket. When a process attempts to read data from an input channel, it will get data from the queue or it will block until data becomes available. The process that writes data to a stream may be blocked by the TCP flow-control mechanism if the socket at the other end is queuing as much data as the protocol allows.

*Threads:* When a server accepts a connection, it generally creates a new thread in which to communicate with the new client. The advantage of using a separate thread for each client is that the server can block when waiting for input without delaying other clients. In an environment in which threads are not provided, an alternative is to test whether input is available from a stream before attempting to read it; for example, in a UNIX environment the *select* system call may be used for this purpose.

**Failure model** • To satisfy the integrity property of reliable communication, TCP streams use checksums to detect and reject corrupt packets and sequence numbers to detect and reject duplicate packets. For the sake of the validity property, TCP streams use timeouts and retransmissions to deal with lost packets. Therefore, messages are guaranteed to be delivered even when some of the underlying packets are lost.

But if the packet loss over a connection passes some limit or the network connecting a pair of communicating processes is severed or becomes severely congested, the TCP software responsible for sending messages will receive no acknowledgements and after a time will declare the connection to be broken. Thus TCP does not provide reliable communication, because it does not guarantee to deliver messages in the face of all possible difficulties.

When a connection is broken, a process using it will be notified if it attempts to read or write. This has the following effects:

- The processes using the connection cannot distinguish between network failure and failure of the process at the other end of the connection.
- The communicating processes cannot tell whether the messages they have sent recently have been received or not.

**Use of TCP** • Many frequently used services run over TCP connections, with reserved port numbers. These include the following:

*HTTP:* The Hypertext Transfer Protocol is used for communication between web browsers and web servers; it is discussed in Section 5.2.

*FTP:* The File Transfer Protocol allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection.

*Telnet:* Telnet provides access by means of a terminal session to a remote computer.

*SMTP:* The Simple Mail Transfer Protocol is used to send mail between computers.

**Java API for TCP streams** • The Java interface to TCP streams is provided in the classes *ServerSocket* and *Socket*:

*ServerSocket:* This class is intended for use by a server to create a socket at a server port for listening for *connect* requests from clients. Its *accept* method gets a *connect* request from the queue or, if the queue is empty, blocks until one arrives. The result of executing *accept* is an instance of *Socket* – a socket to use for communicating with the client.

**Figure 4.5** TCP client makes connection to server, sends request and receives reply

```

import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out =
                new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]);    // UTF is a string encoding; see Sec 4.3
            String data = in.readUTF();
            System.out.println("Received: " + data );
        } catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage());
        } catch (EOFException e){System.out.println("EOF:"+e.getMessage());
        } catch (IOException e){System.out.println("IO:"+e.getMessage());
        } finally {if(s!=null) try {s.close();}catch (IOException e){/*close failed*/}}
    }
}

```

*Socket*: This class is for use by a pair of processes with a connection. The client uses a constructor to create a socket, specifying the DNS hostname and port of a server. This constructor not only creates a socket associated with a local port but also *connects* it to the specified remote computer and port number. It can throw an *UnknownHostException* if the hostname is wrong or an *IOException* if an IO error occurs.

The *Socket* class provides the methods *getInputStream* and *getOutputStream* for accessing the two streams associated with a socket. The return types of these methods are *InputStream* and *OutputStream*, respectively – abstract classes that define methods for reading and writing bytes. The return values can be used as the arguments of constructors for suitable input and output streams. Our example uses *DataInputStream* and *DataOutputStream*, which allow binary representations of primitive data types to be read and written in a machine-independent manner.

Figure 4.5 shows a client program in which the arguments of the *main* method supply a message and the DNS hostname of the server. The client creates a socket bound to the hostname and server port 7896. It makes a *DataInputStream* and a *DataOutputStream* from the socket's input and output streams, then writes the message to its output stream and waits to read a reply from its input stream. The server program in Figure 4.6 opens a server socket on its server port (7896) and listens for *connect* requests. When one arrives, it makes a new thread in which to communicate with the client. The new thread

**Figure 4.6** TCP server makes a connection for each client and then echoes the client's request

---

```

import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[]) {
        try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }
        } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
    }
}
class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream( clientSocket.getInputStream());
            out =new DataOutputStream( clientSocket.getOutputStream());
            this.start();
        } catch(IOException e) {System.out.println("Connection:"+e.getMessage());}
    }
    public void run(){
        try {
            // an echo server
            String data = in.readUTF();
            out.writeUTF(data);
        } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());}
        } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
        } finally { try {clientSocket.close();}catch (IOException e){/*close failed*/}}
    }
}

```

---

creates a *DataInputStream* and a *DataOutputStream* from its socket's input and output streams and then waits to read a message and write the same one back.

As our message consists of a string, the client and server processes use the method *writeUTF* of *DataOutputStream* to write it to the output stream and the method *readUTF* of *DataInputStream* to read it from the input stream. UTF-8 is an encoding that represents strings in a particular format, which is described in Section 4.3.

When a process has closed its socket, it will no longer be able to use its input and output streams. The process to which it has sent data can read the data in its queue, but any further reads after the queue is empty will result in an *EOFException*. Attempts to use a closed socket or to write to a broken stream result in an *IOException*.

## 4.3 External data representation and marshalling

---

The information stored in running programs is represented as data structures – for example, by sets of interconnected objects – whereas the information in messages consists of sequences of bytes. Irrespective of the form of communication used, the data structures must be flattened (converted to a sequence of bytes) before transmission and rebuilt on arrival. The individual primitive data items transmitted in messages can be data values of many different types, and not all computers store primitive values such as integers in the same order. The representation of floating-point numbers also differs between architectures. There are two variants for the ordering of integers: the so-called *big-endian* order, in which the most significant byte comes first; and *little-endian* order, in which it comes last. Another issue is the set of codes used to represent characters: for example, the majority of applications on systems such as UNIX use ASCII character coding, taking one byte per character, whereas the Unicode standard allows for the representation of texts in many different languages and takes two bytes per character.

One of the following methods can be used to enable any two computers to exchange binary data values:

- The values are converted to an agreed external format before transmission and converted to the local form on receipt; if the two computers are known to be the same type, the conversion to external format can be omitted.
- The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary.

Note, however, that bytes themselves are never altered during transmission. To support RMI or RPC, any data type that can be passed as an argument or returned as a result must be able to be flattened and the individual primitive data values represented in an agreed format. An agreed standard for the representation of data structures and primitive values is called an *external data representation*.

*Marshalling* is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message. *Unmarshalling* is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination. Thus marshalling consists of the translation of structured data items and primitive values into an external data representation. Similarly, unmarshalling consists of the generation of primitive values from their external data representation and the rebuilding of the data structures.

Three alternative approaches to external data representation and marshalling are discussed (with a fourth considered in Chapter 21, when we examine Google's approach to representing structured data):

- CORBA's common data representation, which is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA. It can be used by a variety of programming languages (see Chapter 8).
- Java's object serialization, which is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored on a disk. It is for use only by Java.
- XML (Extensible Markup Language), which defines a textual format for representing structured data. It was originally intended for documents containing textual self-describing structured data – for example documents accessible on the Web – but it is now also used to represent the data sent in messages exchanged by clients and servers in web services (see Chapter 9).

In the first two cases, the marshalling and unmarshalling activities are intended to be carried out by a middleware layer without any involvement on the part of the application programmer. Even in the case of XML, which is textual and therefore more accessible to hand-encoding, software for marshalling and unmarshalling is available for all commonly used platforms and programming environments. Because marshalling requires the consideration of all the finest details of the representation of the primitive components of composite objects, the process is likely to be error-prone if carried out by hand. Compactness is another issue that can be addressed in the design of automatically generated marshalling procedures.

In the first two approaches, the primitive data types are marshalled into a binary form. In the third approach (XML), the primitive data types are represented textually. The textual representation of a data value will generally be longer than the equivalent binary representation. The HTTP protocol, which is described in Chapter 5, is another example of the textual approach.

Another issue with regard to the design of marshalling methods is whether the marshalled data should include information concerning the type of its contents. For example, CORBA's representation includes just the values of the objects transmitted, and nothing about their types. On the other hand, both Java serialization and XML do include type information, but in different ways. Java puts all of the required type information into the serialized form, but XML documents may refer to externally defined sets of names (with types) called *namespaces*.

Although we are interested in the use of an external data representation for the arguments and results of RMIs and RPCs, it does have a more general use for representing data structures, objects or structured documents in a form suitable for transmission in messages or storing in files.

Two other techniques for external data representation are worthy of mention. Google uses an approach called *protocol buffers* to capture representations of both stored and transmitted data, which we examine in Section 20.4.1. There is also considerable interest in JSON (JavaScript Object Notation) as an approach to external data representation [[www.json.org](http://www.json.org)]. Protocol buffers and JSON represent a step towards more lightweight approaches to data representation (when compared, for example, to XML).

**Figure 4.7** CORBA CDR for constructed types

Type	Representation
<i>sequence</i>	length (unsigned long) followed by elements in order
<i>string</i>	length (unsigned long) followed by characters in order (can also have wide characters)
<i>array</i>	array elements in order (no length specified because it is fixed)
<i>struct</i>	in the order of declaration of the components
<i>enumerated</i>	unsigned long (the values are specified by the order declared)
<i>union</i>	type tag followed by the selected member

### 4.3.1 CORBA's Common Data Representation (CDR)

CORBA CDR is the external data representation defined with CORBA 2.0 [OMG 2004a]. CDR can represent all of the data types that can be used as arguments and return values in remote invocations in CORBA. These consist of 15 primitive types, which include *short* (16-bit), *long* (32-bit), *unsigned short*, *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean* (TRUE, FALSE), *octet* (8-bit), and *any* (which can represent any basic or constructed type); together with a range of composite types, which are described in Figure 4.7. Each argument or result in a remote invocation is represented by a sequence of bytes in the invocation or result message.

*Primitive types:* CDR defines a representation for both big-endian and little-endian orderings. Values are transmitted in the sender's ordering, which is specified in each message. The recipient translates if it requires a different ordering. For example, a 16-bit *short* occupies two bytes in the message, and for big-endian ordering, the most significant bits occupy the first byte and the least significant bits occupy the second byte. Each primitive value is placed at an index in the sequence of bytes according to its size. Suppose that the sequence of bytes is indexed from zero upwards. Then a primitive value of size  $n$  bytes (where  $n = 1, 2, 4$  or  $8$ ) is appended to the sequence at an index that is a multiple of  $n$  in the stream of bytes. Floating-point values follow the IEEE standard, in which the sign, exponent and fractional part are in bytes  $0$ – $n$  for big-endian ordering and the other way round for little-endian. Characters are represented by a code set agreed between client and server.

*Constructed types:* The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order, as shown in Figure 4.7.

Figure 4.8 shows a message in CORBA CDR that contains the three fields of a *struct* whose respective types are *string*, *string* and *unsigned long*. The figure shows the sequence of bytes with four bytes in each row. The representation of each string consists of an *unsigned long* representing its length followed by the characters in the string. For simplicity, we assume that each character occupies just one byte. Variable-length data is padded with zeros so that it has a standard form, enabling marshalled data or its checksum to be compared. Note that each *unsigned long*, which occupies four bytes,

Figure 4.8 CORBA CDR message

<i>index in sequence of bytes</i>	<i>← 4 bytes →</i>	<i>notes on representation</i>
0–3	5	<i>length of string</i>
4–7	"Smit"	<i>'Smith'</i>
8–11	"h__"	
12–15	6	<i>length of string</i>
16–19	"Lond"	<i>'London'</i>
20–23	"on__"	
24–27	1984	<i>unsigned long</i>

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1984}

starts at an index that is a multiple of four. The figure does not distinguish between the big- and little-endian orderings. Although the example in Figure 4.8 is simple, CORBA CDR can represent any data structure that can be composed from the primitive and constructed types, but without using pointers.

Another example of an external data representation is the Sun XDR standard, which is specified in RFC 1832 [Srinivasan 1995b] and described in [www.cdk5.net/ipc](http://www.cdk5.net/ipc). It was developed by Sun for use in the messages exchanged between clients and servers in Sun NFS (see Chapter 13).

The type of a data item is not given with the data representation in the message in either the CORBA CDR or the Sun XDR standard. This is because it is assumed that the sender and recipient have common knowledge of the order and types of the data items in a message. In particular, for RMI or RPC, each method invocation passes arguments of particular types, and the result is a value of a particular type.

**Marshalling in CORBA** • Marshalling operations can be generated automatically from the specification of the types of data items to be transmitted in a message. The types of the data structures and the types of the basic data items are described in CORBA IDL (see Section 8.3.1), which provides a notation for describing the types of the arguments and results of RMI methods. For example, we might use CORBA IDL to describe the data structure in the message in Figure 4.8 as follows:

```
struct Person{
    string name;
    string place;
    unsigned long year;
};
```

The CORBA interface compiler (see Chapter 5) generates appropriate marshalling and unmarshalling operations for the arguments and results of remote methods from the definitions of the types of their parameters and results.



### 4.3.2 Java object serialization

In Java RMI, both objects and primitive data values may be passed as arguments and results of method invocations. An object is an instance of a Java class. For example, the Java class equivalent to the *Person struct* defined in CORBA IDL might be:

```
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;
    public Person(String aName, String aPlace, int aYear) {
        name = aName;
        place = aPlace;
        year = aYear;
    }
    // followed by methods for accessing the instance variables
}
```

The above class states that it implements the *Serializable* interface, which has no methods. Stating that a class implements the *Serializable* interface (which is provided in the *java.io* package) has the effect of allowing its instances to be serialized.

In Java, the term *serialization* refers to the activity of flattening an object or a connected set of objects into a serial form that is suitable for storing on disk or transmitting in a message, for example, as an argument or the result of an RMI. Deserialization consists of restoring the state of an object or a set of objects from their serialized form. It is assumed that the process that does the deserialization has no prior knowledge of the types of the objects in the serialized form. Therefore some information about the class of each object is included in the serialized form. This information enables the recipient to load the appropriate class when an object is deserialized.

The information about a class consists of the name of the class and a version number. The version number is intended to change when major changes are made to the class. It can be set by the programmer or calculated automatically as a hash of the name of the class and its instance variables, methods and interfaces. The process that deserializes an object can check that it has the correct version of the class.

Java objects can contain references to other objects. When an object is serialized, all the objects that it references are serialized together with it to ensure that when the object is reconstructed, all of its references can be fulfilled at the destination. References are serialized as *handles*. In this case, the handle is a reference to an object within the serialized form – for example, the next number in a sequence of positive integers. The serialization procedure must ensure that there is a 1–1 correspondence between object references and handles. It must also ensure that each object is written once only – on the second or subsequent occurrence of an object, the handle is written instead of the object.

To serialize an object, its class information is written out, followed by the types and names of its instance variables. If the instance variables belong to new classes, then their class information must also be written out, followed by the types and names of their instance variables. This recursive procedure continues until the class information and types and names of the instance variables of all of the necessary classes have been

Figure 4.9 Indication of Java serialized form

Serialized values				Explanation
Person	8-byte version number		h0	class name, version number
3	int year	java.lang.String name	java.lang.String place	number, type and name of instance variables
1984	5 Smith	6 London	h1	values of instance variables

The true serialized form contains additional type markers; h0 and h1 are handles.

written out. Each class is given a handle, and no class is written more than once to the stream of bytes (the handles being written instead where necessary).

The contents of the instance variables that are primitive types, such as integers, chars, booleans, bytes and longs, are written in a portable binary format using methods of the *ObjectOutputStream* class. Strings and characters are written by its *writeUTF* method using the Universal Transfer Format (UTF-8), which enables ASCII characters to be represented unchanged (in one byte), whereas Unicode characters are represented by multiple bytes. Strings are preceded by the number of bytes they occupy in the stream.

As an example, consider the serialization of the following object:

*Person p = new Person("Smith", "London", 1984);*

The serialized form is illustrated in Figure 4.9, which omits the values of the handles and of the type markers that indicate the objects, classes, strings and other objects in the full serialized form. The first instance variable (1984) is an integer that has a fixed length; the second and third instance variables are strings and are preceded by their lengths.

To make use of Java serialization, for example to serialize the *Person* object, create an instance of the class *ObjectOutputStream* and invoke its *writeObject* method, passing the *Person* object as its argument. To deserialize an object from a stream of data, open an *ObjectInputStream* on the stream and use its *readObject* method to reconstruct the original object. The use of this pair of classes is similar to the use of *DataOutputStream* and *DataInputStream*, illustrated in Figures 4.5 and 4.6.

Serialization and deserialization of the arguments and results of remote invocations are generally carried out automatically by the middleware, without any participation by the application programmer. If necessary, programmers with special requirements may write their own version of the methods that read and write objects. To find out how to do this and to get further information about serialization in Java, read the tutorial on object serialization [[java.sun.com](http://java.sun.com) II]. Another way in which a programmer may modify the effects of serialization is by declaring variables that should not be serialized as *transient*. Examples of things that should not be serialized are references to local resources such as files and sockets.

**The use of reflection** • The Java language supports *reflection* – the ability to enquire about the properties of a class, such as the names and types of its instance variables and methods. It also enables classes to be created from their names, and a constructor with

given argument types to be created for a given class. Reflection makes it possible to do serialization and deserialization in a completely generic manner. This means that there is no need to generate special marshalling functions for each type of object, as described above for CORBA. To find out more about reflection, see Flanagan [2002].

Java object serialization uses reflection to find out the class name of the object to be serialized and the names, types and values of its instance variables. That is all that is needed for the serialized form.

For deserialization, the class name in the serialized form is used to create a class. This is then used to create a new constructor with argument types corresponding to those specified in the serialized form. Finally, the new constructor is used to create a new object with instance variables whose values are read from the serialized form.

### 4.3.3 Extensible Markup Language (XML)

XML is a markup language that was defined by the World Wide Web Consortium (W3C) for general use on the Web. In general, the term *markup language* refers to a textual encoding that represents both a text and details as to its structure or its appearance. Both XML and HTML were derived from SGML (Standardized Generalized Markup Language) [ISO 8879], a very complex markup language. HTML (see Section 1.6) was designed for defining the appearance of web pages. XML was designed for writing structured documents for the Web.

XML data items are tagged with ‘markup’ strings. The tags are used to describe the logical structure of the data and to associate attribute-value pairs with logical structures. That is, in XML, the tags relate to the structure of the text that they enclose, in contrast to HTML, in which the tags specify how a browser could display the text. For a specification of XML, see the pages on XML provided by W3C [[www.w3.org](http://www.w3.org) VI].

XML is used to enable clients to communicate with web services and for defining the interfaces and other properties of web services. However, XML is also used in many other ways, including in archiving and retrieval systems – although an XML archive may be larger than a binary one, it has the advantage of being readable on any computer. Other examples of uses of XML include for the specification of user interfaces and the encoding of configuration files in operating systems.

XML is *extensible* in the sense that users can define their own tags, in contrast to HTML, which uses a fixed set of tags. However, if an XML document is intended to be used by more than one application, then the names of the tags must be agreed between them. For example, clients usually use SOAP messages to communicate with web services. SOAP (see Section 9.2.1) is an XML format whose tags are published for use by web services and their clients.

Some external data representations (such as CORBA CDR) do not need to be self-describing, because it is assumed that the client and server exchanging a message have prior knowledge of the order and the types of the information it contains. However, XML was intended to be used by multiple applications for different purposes. The provision of tags, together with the use of namespaces to define the meaning of the tags, has made this possible. In addition, the use of tags enables applications to select just those parts of a document it needs to process: it will not be affected by the addition of information relevant to other applications.

**Figure 4.10** XML definition of the *Person* structure

```

<person id="123456789">
    <name>Smith</name>
    <place>London</place>
    <year>1984</year>
    <!-- a comment -->
</person >

```

---

XML documents, being textual, can be read by humans. In practice, most XML documents are generated and read by XML processing software, but the ability to read XML can be useful when things go wrong. In addition, the use of text makes XML independent of any particular platform. The use of a textual rather than a binary representation, together with the use of tags, makes the messages large, so they require longer processing and transmission times, as well as more space to store. A comparison of the efficiency of messages using the SOAP XML format and CORBA CDR is given in Section 9.2.4. However, files and messages can be compressed – HTTP version 1.1 allows data to be compressed, which saves bandwidth during transmission.

**XML elements and attributes** • Figure 4.10 shows the XML definition of the *Person* structure that was used to illustrate marshalling in CORBA CDR and Java. It shows that XML consists of tags and character data. The character data, for example *Smith* or *1984*, is the actual data. As in HTML, the structure of an XML document is defined by pairs of tags enclosed in angle brackets. In Figure 4.10, *<name>* and *<place>* are both tags. As in HTML, layout can generally be used to improve readability. Comments in XML are denoted in the same way as those in HTML.

**Elements:** An element in XML consists of a portion of character data surrounded by matching start and end tags. For example, one of the elements in Figure 4.10 consists of the data *Smith* contained within the *<name> ... </name>* tag pair. Note that the element with the *<name>* tag is enclosed in the element with the *<person id="123456789"> ... </person >* tag pair. The ability of an element to enclose another element allows hierarchic data to be represented – a very important aspect of XML. An empty tag has no content and is terminated with */>* instead of *>*. For example, the empty tag *<european/>* could be included within the *<person> ...</person>* tag.

**Attributes:** A start tag may optionally include pairs of associated attribute names and values such as *id="123456789"*, as shown above. The syntax is the same as for HTML, in which an attribute name is followed by an equal sign and an attribute value in quotes. Multiple attribute values are separated by spaces.

It is a matter of choice as to which items are represented as elements and which ones as attributes. An element is generally a container for data, whereas an attribute is used for labelling that data. In our example, *123456789* might be an identifier used by the application, whereas *name*, *place* and *year* might be displayed. Also, if data contains substructures or several lines, it must be defined as an element. Attributes are for simple values.

**Names:** The names of tags and attributes in XML generally start with a letter, but can also start with an underline or a colon. The names continue with letters, digits, hyphens, underscores, colons or full stops. Letters are case-sensitive. Names that start with *xml* are reserved.

**Binary data:** All of the information in XML elements must be expressed as character data. But the question is: how do we represent encrypted elements or secure hashes – both of which, as we shall see in Section 9.5 are used in XML security? The answer is that they can be represented in *base64* notation [Freed and Borenstein 1996], which uses only the alphanumeric characters together with +, / and =, which has a special meaning.

**Parsing and well-formed documents** • An XML document must be well formed – that is, it must conform to rules about its structure. A basic rule is that every start tag has a matching end tag. Another basic rule is that all tags are correctly nested – for example, `<x>..<y>..</y>..</x>` is correct, whereas `<x>..<y>..</x>..</y>` is not. Finally, every XML document must have a single root element that encloses all the other elements. These rules make it very simple to implement parsers for XML documents. When a parser reads an XML document that is not well formed, it will report a fatal error.

**CDATA:** XML parsers normally parse the contents of elements because they may contain further nested structures. If text needs to contain an angle bracket or a quote, it may be represented in a special way: for example, `&lt;` represents the opening angle bracket. However, if a section should not be parsed – for example, because it contains special characters – it can be denoted as *CDATA*. For example, if a place name is to include an apostrophe, then it could be specified in either of the two following ways:

```
<place> King&apos; Cross </place >
<place> <![CDATA [King's Cross]]></place >
```

**XML prolog:** Every XML document must have a *prolog* as its first line. The prolog must at least specify the version of XML in use (which is currently 1.0). For example:

```
<?XML version = "1.0" encoding = "UTF-8" standalone = "yes"?>
```

The prolog may specify the encoding (UTF-8 is the default and was explained in Section 4.3.2). The term *encoding* refers to the set of codes used to represent characters – ASCII being the best-known example. Note that in the XML prolog, ASCII is specified as *us-ascii*. Other possible encodings include ISO-8859-1 (or Latin-1) – an 8-bit encoding whose first 128 values are ASCII, with the rest being used to represent the characters in Western European languages – and various other 8-bit encodings for representing other alphabets, for example, Greek or Cyrillic.

An additional attribute may be used to state whether the document stands alone or is dependent on external definitions.

**XML namespaces** • Traditionally, namespaces provide a means for scoping names. An XML namespace is a set of names for a collection of element types and attributes that is referenced by a URL. Any other XML document can use an XML namespace by referring to its URL.

Any element that makes use of an XML namespace can specify that namespace as an attribute called *xmlns*, whose value is a URL referring to the file containing the namespace definitions. For example:

```
xmlns:pers = "http://www.cdk5.net/person"
```

**Figure 4.11** Illustration of the use of a namespace in the *Person* structure

```

<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">
  <pers:name> Smith </pers:name>
  <pers:place> London </pers:place >
  <pers:year> 1984 </pers:year>
</person>

```

---

The name after *xmlns*, in this case *pers* can be used as a prefix to refer to the elements in a particular namespace, as shown in Figure 4.11. The *pers* prefix is bound to *http://www.cdk4.net/person* for the *person* element. A namespace applies within the context of the enclosing pair of start and end tags unless overridden by an enclosed namespace declaration. An XML document may be defined in terms of several different namespaces, each of which is referenced by a unique prefix.

The namespace convention allows an application to make use of multiple sets of external definitions in different namespaces without the risk of name clashes.

**XML schemas** • An XML schema [[www.w3.org](http://www.w3.org) VIII] defines the elements and attributes that can appear in a document, how the elements are nested and the order and number of elements, and whether an element is empty or can include text. For each element, it defines the type and default value. Figure 4.12 gives an example of a schema that defines the data types and structure of the XML definition of the *person* structure in Figure 4.10.

The intention is that a single schema definition may be shared by many different documents. An XML document that is defined to conform to a particular schema may also be validated by means of that schema. For example, the sender of a SOAP message may use an XML schema to encode it, and the recipient will use the same XML schema to validate and decode it.

**Figure 4.12** An XML schema for the *Person* structure

```

<xsd:schema xmlns:xsd = URL of XML schema definitions >
  <xsd:element name= "person" type = "personType" />
  <xsd:complexType name= "personType">
    <xsd:sequence>
      <xsd:element name = "name" type= "xs:string"/>
      <xsd:element name = "place" type= "xs:string"/>
      <xsd:element name = "year" type= "xs:positiveInteger"/>
    </xsd:sequence>
    <xsd:attribute name= "id" type = "xs:positiveInteger"/>
  </xsd:complexType>
</xsd:schema>

```

---

**Document type definitions:** Document type definitions (DTDs) [[www.w3.org](http://www.w3.org) VI] were provided as a part of the XML 1.0 specification for defining the structure of XML documents and are still widely used for that purpose. The syntax of DTDs is different from the rest of XML and it is quite limited in what it can specify; for example, it cannot describe data types and its definitions are global, preventing element names from being duplicated. DTDs are not used for defining web services, although they may still be used to define documents that are transmitted by web services.

**APIs for accessing XML** • XML parsers and generators are available for most commonly used programming languages. For example, there is Java software for writing out Java objects as XML (marshalling) and for creating Java objects from such structures (unmarshalling). Similar software is available in Python for Python data types and objects.

#### 4.3.4 Remote object references

This section applies only to languages such as Java and CORBA that support the distributed object model. It is not relevant to XML.

When a client invokes a method in a remote object, an invocation message is sent to the server process that hosts the remote object. This message needs to specify which particular object is to have its method invoked. A *remote object reference* is an identifier for a remote object that is valid throughout a distributed system. A remote object reference is passed in the invocation message to specify which object is to be invoked. Chapter 5 explains that remote object references are also passed as arguments and returned as results of remote method invocations, that each remote object has a single remote object reference and that remote object references can be compared to see whether they refer to the same remote object. Here, we discuss the external representation of remote object references.

Remote object references must be generated in a manner that ensures uniqueness over space and time. In general, there may be many processes hosting remote objects, so remote object references must be unique among all of the processes in the various computers in a distributed system. Even after the remote object associated with a given remote object reference is deleted, it is important that the remote object reference is not reused, because its potential invokers may retain obsolete remote object references. Any attempt to invoke a deleted object should produce an error rather than allow access to a different object.

There are several ways to ensure that a remote object reference is unique. One way is to construct a remote object reference by concatenating the Internet address of its host computer and the port number of the process that created it with the time of its creation and a local object number. The local object number is incremented each time an object is created in that process.

The port number and time together produce a unique process identifier on that computer. With this approach, remote object references might be represented with a format such as that shown in Figure 4.13. In the simplest implementations of RMI, remote objects live only in the process that created them and survive only as long as that process continues to run. In such cases, the remote object reference can be used as the address of the remote object. In other words, invocation messages are sent to the Internet



**Figure 4.13** Representation of a remote object reference

32 bits	32 bits	32 bits	32 bits	
Internet address	port number	time	object number	interface of remote object

address in the remote reference and to the process on that computer using the given port number.

To allow remote objects to be relocated into a different process on a different computer, the remote object reference should not be used as the address of the remote object. Section 8.3.3 discusses a form of remote object reference that allows objects to be activated in different servers throughout its lifetime.

The peer-to-peer overlay systems described in Chapter 10 use a form of remote object reference that is completely independent of location. Messages are routed to resources by means of a distributed routing algorithm.

The last field of the remote object reference shown in Figure 4.13 contains some information about the interface of the remote object, for example, the interface name. This information is relevant to any process that receives a remote object reference as an argument or as the result of a remote invocation, because it needs to know about the methods offered by the remote object. This point is explained again in Section 5.4.2.

## 4.4 Multicast communication

The pairwise exchange of messages is not the best model for communication from one process to a group of other processes, which may be necessary, for example, when a service is implemented as a number of different processes in different computers, perhaps to provide fault tolerance or to enhance availability. A *multicast operation* is more appropriate – this is an operation that sends a single message from one process to each of the members of a group of processes, usually in such a way that the membership of the group is transparent to the sender. There is a range of possibilities in the desired behaviour of a multicast. The simplest multicast protocol provides no guarantees about message delivery or ordering.

Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:

1. *Fault tolerance based on replicated services:* A replicated service consists of a group of servers. Client requests are multicast to all the members of the group, each of which performs an identical operation. Even when some of the members fail, clients can still be served.
2. *Discovering services in spontaneous networking:* Section 1.3.2 defines service discovery in the context of spontaneous networking. Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.



3. *Better performance through replicated data:* Data are replicated to increase the performance of a service – in some cases replicas of the data are placed in users' computers. Each time the data changes, the new value is multicast to the processes managing the replicas.
4. *Propagation of event notifications:* Multicast to a group may be used to notify processes when something happens. For example, in Facebook, when someone changes their status, all their friends receive notifications. Similarly, publish-subscribe protocols may make use of group multicast to disseminate events to subscribers (see Chapter 6).

In this section introduce IP multicast and then review the needs of the above uses of group communication to see which of them can be satisfied by IP multicast. For those that cannot, we propose some further properties for group communication protocols in addition to those provided by IP multicast.

#### 4.4.1 IP multicast – An implementation of multicast communication

This section discusses IP multicast and presents Java's API to it via the *MulticastSocket* class.

**IP multicast** • *IP multicast* is built on top of the Internet Protocol (IP). Note that IP packets are addressed to computers – ports belong to the TCP and UDP levels. IP multicast allows the sender to transmit a single IP packet to a set of computers that form a multicast group. The sender is unaware of the identities of the individual recipients and of the size of the group. A *multicast group* is specified by a Class D Internet address (see Figure 3.15) – that is, an address whose first 4 bits are 1110 in IPv4.

Being a member of a multicast group allows a computer to receive IP packets sent to the group. The membership of multicast groups is dynamic, allowing computers to join or leave at any time and to join an arbitrary number of groups. It is possible to send datagrams to a multicast group without being a member.

At the application programming level, IP multicast is available only via UDP. An application program performs multicasts by sending UDP datagrams with multicast addresses and ordinary port numbers. It can join a multicast group by making its socket join the group, enabling it to receive messages to the group. At the IP level, a computer belongs to a multicast group when one or more of its processes has sockets that belong to that group. When a multicast message arrives at a computer, copies are forwarded to all of the local sockets that have joined the specified multicast address and are bound to the specified port number. The following details are specific to IPv4:

*Multicast routers:* IP packets can be multicast both on a local network and on the wider Internet. Local multicasts use the multicast capability of the local network, for example, of an Ethernet. Internet multicasts make use of multicast routers, which forward single datagrams to routers on other networks, where they are again multicast to local members. To limit the distance of propagation of a multicast datagram, the sender can specify the number of routers it is allowed to pass – called the *time to live*, or TTL for short. To understand how routers know which other routers have members of a multicast group, see Comer [2007].

*Multicast address allocation:* As discussed in Chapter 3, Class D addresses (that is, addresses in the range 224.0.0.0 to 239.255.255.255) are reserved for multicast traffic and managed globally by the Internet Assigned Numbers Authority (IANA). The management of this address space is reviewed annually, with current practice documented in RFC 3171 [Albanna *et al.* 2001]. This document defines a partitioning of this address space into a number of blocks, including:

- Local Network Control Block (224.0.0.0 to 224.0.0.225), for multicast traffic within a given local network.
- Internet Control Block (224.0.1.0 to 224.0.1.225).
- Ad Hoc Control Block (224.0.2.0 to 224.0.255.0), for traffic that does not fit any other block.
- Administratively Scoped Block (239.0.0.0 to 239.255.255.255), which is used to implement a scoping mechanism for multicast traffic (to constrain propagation).

Multicast addresses may be permanent or temporary. Permanent groups exist even when there are no members – their addresses are assigned by IANA and span the various blocks mentioned above. For example, 224.0.1.1 in the Internet block is reserved for the Network Time Protocol (NTP), as discussed in Chapter 14, and the range 224.0.6.000 to 224.0.6.127 in the ad hoc block is reserved for the ISIS project (see Chapters 6 and 18). Addresses are reserved for a variety of purposes, from specific Internet protocols to given organizations that make heavy use of multicast traffic, including multimedia broadcasters and financial institutions. A full list of reserved addresses can be seen on the IANA web site [[www.iana.org](http://www.iana.org) II].

The remainder of the multicast addresses are available for use by temporary groups, which must be created before use and cease to exist when all the members have left. When a temporary group is created, it requires a free multicast address to avoid accidental participation in an existing group. The IP multicast protocol does not directly address this issue. If used locally, relatively simple solutions are possible – for example setting the TTL to a small value, making collisions with other groups unlikely. However, programs using IP multicast throughout the Internet require a more sophisticated solution to this problem. RFC 2908 [Thaler *et al.* 2000] describes a multicast address allocation architecture (MALLOC) for Internet-wide applications, that allocates unique addresses for a given period of time and in a given scope. As such, the proposal is intrinsically bound with the scoping mechanisms mentioned above. A client-server solution is adopted whereby clients request a multicast address from a multicast address allocation server (MAAS), which must then communicate across domains to ensure allocations are unique for the given lifetime and scope.

**Failure model for multicast datagrams** • Datagrams multicast over IP multicast have the same failure characteristics as UDP datagrams – that is, they suffer from omission failures. The effect on a multicast is that messages are not guaranteed to be delivered to any particular group member in the face of even a single omission failure. That is, some but not all of the members of the group may receive it. This can be called *unreliable* multicast, because it does not guarantee that a message will be delivered to any member of a group. Reliable multicast is discussed in Chapter 15.

**Figure 4.14** Multicast peer joins a group and sends and receives datagrams

```

import java.net.*;
import java.io.*;
public class MulticastPeer{
    public static void main(String args[]){
        // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s = null;
        try {
            InetAddress group = InetAddress.getByName(args[1]);
            s = new MulticastSocket(6789);
            s.joinGroup(group);
            byte [] m = args[0].getBytes();
            DatagramPacket messageOut =
                new DatagramPacket(m, m.length, group, 6789);
            s.send(messageOut);
            byte[] buffer = new byte[1000];
            for(int i=0; i< 3; i++) { // get messages from others in group
                DatagramPacket messageIn =
                    new DatagramPacket(buffer, buffer.length);
                s.receive(messageIn);
                System.out.println("Received:" + new String(messageIn.getData()));
            }
            s.leaveGroup(group);
        } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        } catch (IOException e){System.out.println("IO: " + e.getMessage());}
        } finally { if(s != null) s.close();}
    }
}

```

**Java API to IP multicast** • The Java API provides a datagram interface to IP multicast through the class *MulticastSocket*, which is a subclass of *DatagramSocket* with the additional capability of being able to join multicast groups. The class *MulticastSocket* provides two alternative constructors, allowing sockets to be created to use either a specified local port (6789, in Figure 4.14) or any free local port. A process can join a multicast group with a given multicast address by invoking the *joinGroup* method of its multicast socket. Effectively, the socket joins a multicast group at a given port and it will receive datagrams sent by processes on other computers to that group at that port. A process can leave a specified group by invoking the *leaveGroup* method of its multicast socket.

In the example in Figure 4.14, the arguments to the *main* method specify a message to be multicast and the multicast address of a group (for example, "228.5.6.7"). After joining that multicast group, the process makes an instance of *DatagramPacket* containing the message and sends it through its multicast socket to the multicast group address at port 6789. After that, it attempts to receive three multicast messages from its

peers via its socket, which also belongs to the group on the same port. When several instances of this program are run simultaneously on different computers, all of them join the same group, and each of them should receive its own message and the messages from those that joined after it.

The Java API allows the TTL to be set for a multicast socket by means of the *setTimeToLive* method. The default is 1, allowing the multicast to propagate only on the local network.

An application implemented over IP multicast may use more than one port. For example, the MultiTalk [\[mbone\]](#) application, which allows groups of users to hold text-based conversations, has one port for sending and receiving data and another for exchanging control data.

#### 4.4.2 Reliability and ordering of multicast

The previous section stated the failure model for IP multicast, which suffers from omission failures. A datagram sent from one multicast router to another may be lost, thus preventing all recipients beyond that router from receiving the message. Also, when a multicast on a local area network uses the multicasting capabilities of the network to allow a single datagram to arrive at multiple recipients, any one of those recipients may drop the message because its buffer is full.

Another factor is that any process may fail. If a multicast router fails, the group members beyond that router will not receive the multicast message, although local members may do so.

Ordering is another issue. IP packets sent over an internetwork do not necessarily arrive in the order in which they were sent, with the possible effect that some group members receive datagrams from a single sender in a different order from other group members. In addition, messages sent by two different processes will not necessarily arrive in the same order at all the members of the group.

**Some examples of the effects of reliability and ordering** • We now consider the effect of the failure semantics of IP multicast on the four examples of the use of replication in the introduction to Section 4.4.

1. *Fault tolerance based on replicated services*: Consider a replicated service that consists of the members of a group of servers that start in the same initial state and always perform the same operations in the same order, so as to remain consistent with one another. This application of multicast requires that either all of the replicas or none of them should receive each request to perform an operation – if one of them misses a request, it will become inconsistent with the others. In most cases, this service would require that all members receive request messages in the same order as one another.
3. *Discovering services in spontaneous networking*: One way for a process to discover services in spontaneous networking is to multicast requests at periodic intervals, and for the available services to listen for those multicasts and respond. An occasional lost request is not an issue when discovering services. In fact, Jini uses IP multicast in its protocol for discovering services. This is described in Section 19.2.1.

3. *Better performance through replicated data*: Consider the case where the replicated data itself, rather than operations on the data, are distributed by means of multicast messages. The effect of lost messages and inconsistent ordering would depend on the method of replication and the importance of all replicas being totally up-to-date.
4. *Propagation of event notifications*: The particular application determines the qualities required of multicast. For example, the Jini lookup services use IP multicast to announce their existence (see Section 19.2.1).

These examples suggest that some applications require a multicast protocol that is more reliable than IP multicast. In particular, there is a need for *reliable multicast*, in which any message transmitted is either received by all members of a group or by none of them. The examples also suggest that some applications have strong requirements for ordering, the strictest of which is called *totally ordered multicast*, in which all of the messages transmitted to a group reach all of the members in the same order.

Chapter 15 will define and show how to implement reliable multicast and various useful ordering guarantees, including totally ordered multicast.

## 4.5 Network virtualization: Overlay networks

---

The strength of the Internet communication protocols is that they provide, through their API (Section 4.2), a very effective set of building blocks for the construction of distributed software. However, a growing range of different classes of application (including, for example, peer-to-peer file sharing and Skype) coexist in the Internet. It would be impractical to attempt to alter the Internet protocols to suit each of the many applications running over them – what might enhance one of them could be detrimental to another. In addition, the IP transport service is implemented over a large and ever-increasing number of network technologies. These two factors have led to the interest in network virtualization.

Network virtualization [Petersen *et al.* 2005] is concerned with the construction of many different virtual networks over an existing network such as the Internet. Each virtual network can be designed to support a particular distributed application. For example, one virtual network might support multimedia streaming, as in BBC iPlayer, BoxeeTV [[boxee.tv](http://boxee.tv)] or Hulu [[hulu.com](http://hulu.com)], and coexist with another that supports a multiplayer online game, both running over the same underlying network. This suggests an answer to the dilemma raised by Salzer’s end-to-end argument (see Section 2.3.3): an application-specific virtual network can be built above an existing network and optimized for that particular application, without changing the characteristics of the underlying network.

Chapter 3 showed that computer networks have addressing schemes, protocols and routing algorithms; similarly, each virtual network has its own particular addressing scheme, protocols and routing algorithms, but redefined to meet the needs of particular application classes.

### 4.5.1 Overlay networks

An *overlay network* is a virtual network consisting of nodes and virtual links, which sits on top of an underlying network (such as an IP network) and offers something that is not otherwise provided:

- a service that is tailored towards the needs of a class of application or a particular higher-level service – for example, multimedia content distribution;
- more efficient operation in a given networked environment – for example routing in an ad hoc network;
- an additional feature – for example, multicast or secure communication.

This leads to a wide variety of types of overlay as captured by Figure 4.15. Overlay networks have the following advantages:

- They enable new network services to be defined without requiring changes to the underlying network, a crucial point given the level of standardization in this area and the difficulties of amending underlying router functionality.
- They encourage experimentation with network services and the customization of services to particular classes of application.
- Multiple overlays can be defined and can coexist, with the end result being a more open and extensible network architecture.

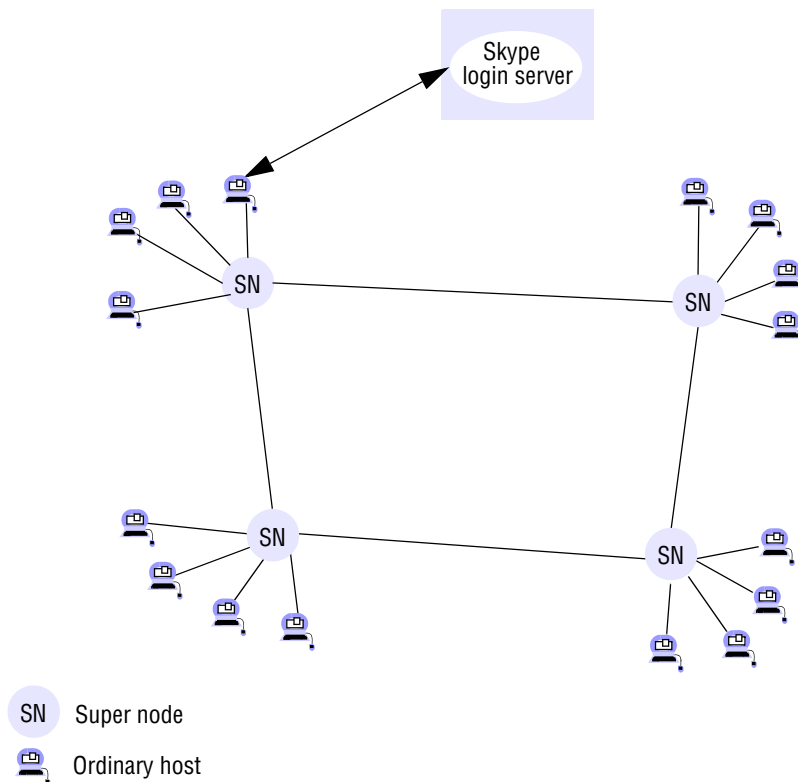
The disadvantages are that overlays introduce an extra level of indirection (and hence may incur a performance penalty) and they add to the complexity of network services when compared, for example, to the relatively simple architecture of TCP/IP networks.

Overlays can be related to the familiar concept of layers (as introduced in Chapters 2 and 3). Overlays are layers, but layers that exist outside the standard architecture (such as the TCP/IP stack) and exploit the resultant degrees of freedom. In particular, overlay developers are free to redefine the core elements of a network as mentioned above, including the mode of addressing, the protocols employed and the approach to routing, often introducing radically different approaches more tailored towards particular application classes of operating environments. For example, distributed hash tables introduce a style of addressing based on a key space and also build a topology in such a way that a node in the topology either owns the key or has a link to a node that is closer to the owner (a style of routing known as *key-based routing*). The topology is most commonly in the form of a ring.

We exemplify the successful use of an overlay network by discussing Skype. Further examples of overlays will be given throughout the book. For example, Chapter 10 presents details of the protocols and structures adopted by peer-to-peer file sharing, along with further information on distributed hash tables. Chapter 19 considers both wireless ad hoc networks and disruption-tolerant networks in the context of mobile and ubiquitous computing and Chapter 20 examines overlay support for multimedia streaming.

**Figure 4.15** Types of overlay

<i>Motivation</i>	<i>Type</i>	<i>Description</i>
<i>Tailored for application needs</i>	Distributed hash tables	One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard hash table but in a networked environment).
	Peer-to-peer file sharing	Overlay structures that focus on constructing tailored addressing and routing mechanisms to support the cooperative discovery and use (for example, download) of files.
	Content distribution networks	Overlays that subsume a range of replication, caching and placement strategies to provide improved performance in terms of content delivery to web users; used for web acceleration and to offer the required real-time performance for video streaming [ <a href="http://www.kontiki.com">www.kontiki.com</a> ].
<i>Tailored for network style</i>	Wireless ad hoc networks	Network overlays that provide customized routing protocols for wireless ad hoc networks, including proactive schemes that effectively construct a routing topology on top of the underlying nodes and reactive schemes that establish routes on demand typically supported by flooding.
	Disruption-tolerant networks	Overlays designed to operate in hostile environments that suffer significant node or link failure and potentially high delays.
<i>Offering additional features</i>	Multicast	One of the earliest uses of overlay networks in the Internet, providing access to multicast services where multicast routers are not available; builds on the work by Van Jacobsen, Deering and Casner with their implementation of the MBone (or Multicast Backbone) [ <a href="http://mbone">mbone</a> ].
	Resilience	Overlay networks that seek an order of magnitude improvement in robustness and availability of Internet paths [ <a href="http://nms.csail.mit.edu">nms.csail.mit.edu</a> ].
	Security	Overlay networks that offer enhanced security over the underling IP network, including virtual private networks, for example, as discussed in Section 3.4.8.

**Figure 4.16** Skype overlay architecture

### 4.5.2 Skype: An example of an overlay network

Skype is a peer-to-peer application offering Voice over IP (VoIP). It also includes instant messaging, video conferencing and interfaces to the standard telephony service through SkypeIn and SkypeOut. The software was developed by Kazaa in 2003 and hence shares many of the characteristics of the Kazaa peer-to-peer file-sharing application [Leibowitz *et al.* 2003]. It is widely deployed, with an estimated 370 million users as of the start of 2009.

Skype is an excellent case study of the use of overlay networks in real-world (and large-scale) systems, indicating how advanced functionality can be provided in an application-specific manner and without modification of the core architecture of the Internet. Skype is a virtual network in that it establishes connections between people (Skype subscribers who are currently active). No IP address or port is required to establish a call. The architecture of the virtual network supporting Skype is not widely publicized but researchers have studied Skype through a variety of methods, including traffic analysis, and its principles are now in the public domain. Much of the detail of the description that follows is taken from the paper by Baset and Schulzrinne [2006], which contains a detailed study of the behaviour of Skype.



**Skype architecture** • Skype is based on a peer-to-peer infrastructure consisting of ordinary users' machines (referred to as hosts) and super nodes – super nodes are ordinary Skype hosts that happen to have sufficient capabilities to carry out their enhanced role. Super nodes are selected on demand based a range of criteria including bandwidth available, reachability (the machine must have a global IP address and not be hidden behind a NAT-enabled router, for example) and availability (based on the length of time that Skype has been running continuously on that node). This overall structure is captured in Figure 4.16.

**User connection** • Skype users are authenticated via a well-known login server. They then make contact with a selected super node. To achieve this, each client maintains a cache of super node identities (that is, IP address and port number pairs). At first login this cache is filled with the addresses of around seven super nodes, and over time the client builds and maintains a much larger set (perhaps several hundred).

**Search for users** • The main goal of super nodes is to perform the efficient search of the global index of users, which is distributed across the super nodes. The search is orchestrated by the client's chosen super node and involves an expanding search of other super nodes until the specified user is found. On average, eight super nodes are contacted. A user search typically takes between three and four seconds to complete for hosts that have a global IP address (and slightly longer, five to six seconds, if behind a NAT-enabled router). From experiments, it appears that intermediary nodes involved in the search cache the results to improve performance.

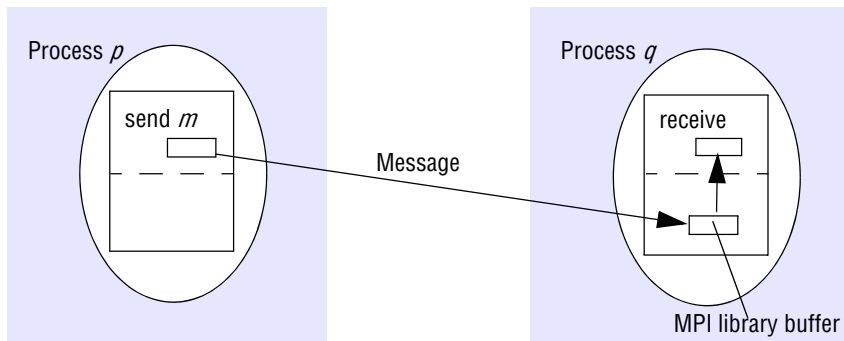
**Voice connection** • Once the required user is discovered, Skype establishes a voice connection between the two parties using TCP for signalling call requests and terminations and either UDP or TCP for the streaming audio. UDP is preferred but TCP, along with the use of an intermediary node, is used in certain circumstances to circumvent firewalls (see Baset and Schulzrinne [2006] for details). The software used for encoding and decoding audio plays a key part in providing the excellent call quality normally attained using Skype, and the associated algorithms are carefully tailored to operate in Internet environments at 32 kbps and above.

## 4.6 Case study: MPI

---

Message passing was introduced in Section 4.2.1, which outlines the basic principles of exchanging messages between two processes using *send* and *receive* operations. The synchronous variant of message passing is realised by blocking *send* and *receive* calls, whereas the asynchronous variant requires a non-blocking form of *send*. The end result is a paradigm for distributed programming that is lightweight, efficient and in many ways minimal.

This style of distributed programming is attractive in classes of system where performance is paramount, most notably in high-performance computing. In this section, we present a case study of the Message Passing Interface standard, developed by the high performance computing community. MPI was first introduced in 1994 by the MPI Forum [[www.mpi-forum.org](http://www.mpi-forum.org)] as a reaction against the wide variety of proprietary approaches that were in use for message passing in this field. The standard

**Figure 4.17** An overview of point-to-point communication in MPI

has also been strongly influential in Grid computing (discussed in Chapter 9), for example through the development of GridMPI [[www.gridmpi.org](http://www.gridmpi.org)]. The goal of the MPI Forum was to retain the inherent simplicity, practicality and efficiency of the message-passing approach but enhance this with portability through presenting a standardized interface independent of the operating system or programming language-specific socket interface. MPI was also designed to be flexible, and the result is a comprehensive specification of message passing in all its variants (with over 115 operations). Applications use the MPI interface via a message-passing library available for a variety of operating systems and programming languages, including C++ and Fortran.

The underlying architectural model for MPI is relatively simple and captured in Figure 4.17. This is similar to the model introduced in Section 4.2.1, but with the added dimension of explicitly having MPI library buffers in both the sender and the receiver, managed by the MPI library and used to hold data in transit. Note that this figure shows one pathway from the sender to the receiver via the receiver's MPI library buffer (other options, for example using the sender's MPI library buffer, will become apparent below).

To provide a flavour of this complexity, let us examine a number of the variants of *send* summarized in Figure 4.18. This is a refinement of the view of message passing as presented in Section 4.2.1, offering more choice and control and effectively separating the semantics of synchronous/asynchronous and blocking/non-blocking message passing.

We start by examining the four blocking operations presented in the associated column of Figure 4.18. The key to understanding this set of operations is to appreciate that blocking is interpreted as 'blocked until it is safe to return', in the sense that application data has been copied into the MPI system and hence is in transit or delivered and therefore the application buffer can be reused (for example, for the next *send* operation). This then enables various interpretations of 'being safe to return'. The *MPI\_Send* operation is a generic operation that simply requires that this level of safety is provided (in practice, this is often implemented using *MPI\_Ssend*). *MPI\_Ssend* is exactly the same as synchronous (and blocking) message passing as introduced in Section 4.2.1, with safety interpreted as delivered, whereas *MPI\_Bsend* has weaker

**Figure 4.18** Selected send operations in MPI

<i>Send operations</i>	<i>Blocking</i>	<i>Non-blocking</i>
<i>Generic</i>	<i>MPI_Send</i> : the sender blocks until it is safe to return – that is, until the message is in transit or delivered and the sender’s application buffer can therefore be reused.	<i>MPI_Isend</i> : the call returns immediately and the programmer is given a communication request handle, which can then be used to check the progress of the call via <i>MPI_Wait</i> or <i>MPI_Test</i> .
<i>Synchronous</i>	<i>MPI_Ssend</i> : the sender and receiver synchronize and the call only returns when the message has been delivered at the receiving end.	<i>MPI_Issend</i> : as with <i>MPI_Isend</i> , but with <i>MPI_Wait</i> and <i>MPI_Test</i> indicating whether the message has been delivered at the receive end.
<i>Buffered</i>	<i>MPI_Bsend</i> : the sender explicitly allocates an MPI buffer library (using a separate <i>MPI_Buffer_attach</i> call) and the call returns when the data is successfully copied into this buffer.	<i>MPI_Ibsend</i> : as with <i>MPI_Isend</i> but with <i>MPI_Wait</i> and <i>MPI_Test</i> indicating whether the message has been copied into the sender’s MPI buffer and hence is in transit.
<i>Ready</i>	<i>MPI_Rsend</i> : the call returns when the sender’s application buffer can be reused (as with <i>MPI_Send</i> ), but the programmer is also indicating to the library that the receiver is ready to receive the message, resulting in potential optimization of the underlying implementation.	<i>MPI_Irsend</i> : the effect is as with <i>MPI_Isend</i> , but as with <i>MPI_Rsend</i> , the programmer is indicating to the underlying implementation that the receiver is guaranteed to be ready to receive (resulting in the same optimizations),

semantics in that the message is considered safe when it has been copied into the preallocated MPI library buffer and is still in transit. *MPI\_Rsend* is a rather curious operation in which the programmer specifies that they know that the receiver is ready to receive the message. If this is known, the underlying implementation can be optimized in that there is no need to check if there is a buffer available to receive the message, avoiding a handshake. This is clearly a rather dangerous operation that will fail if the assumption about being ready is invalid. From the figure, it is possible to observe the elegant symmetry for non-blocking send operations, this time defined over the semantics of the associated *MPI\_Wait* and *MPI\_Test* operations (note also the consistent naming convention across all the operations).

The standard also supports both blocking and non-blocking receive (*MPI\_recv* and *MPI\_irecv*, respectively), and the variants of send and receive can be paired in any combination, offering the programmer rich control over the semantics of message passing. In addition, the standard defines rich patterns of multiway communication (referred to as collective communication) including, for example, scatter (one to many) and gather (many to one) operations.

## 4.7 Summary

The first section of this chapter showed that the Internet transmission protocols provide two alternative building blocks from which application protocols may be constructed. There is an interesting trade-off between the two protocols: UDP provides a simple message-passing facility that suffers from omission failures but carries no built-in performance penalties, on the other hand, in good conditions TCP guarantees message delivery, but at the expense of additional messages and higher latency and storage costs.

The second section showed three alternative styles of marshalling. CORBA and its predecessors choose to marshal data for use by recipients that have prior knowledge of the types of its components. In contrast, when Java serializes data, it includes full information about the types of its contents, allowing the recipient to reconstruct it purely from the content. XML, like Java, includes full type information. Another big difference is that CORBA requires a specification of the types of data items to be marshalled (in IDL) in order to generate the marshalling and unmarshalling methods, whereas Java uses reflection in order to serialize objects and deserialize their serial form. But a variety of different means are used for generating XML, depending on the context. For example, many programming languages, including Java, provide processors for translating between XML and language-level objects.

Multicast messages are used in communication between the members of a group of processes. IP multicast provides a multicast service for both local area networks and the Internet. This form of multicast has the same failure semantics as UDP datagrams, but in spite of suffering from omission failures it is a useful tool for many applications of multicast. Some other applications have stronger requirements – in particular, that multicast delivery should be atomic; that is, it should have all-or-nothing delivery. Further requirements on multicast are related to the ordering of messages, the strongest of which requires that all members of a group receive all of the messages in the same order.

Multicast can also be supported by overlay networks in cases where, for example, IP multicast is not supported. More generally, overlay networks offer a service of virtualization of the network architecture, allowing specialist network services to be created on top of underlying networking infrastructure, (for example, UDP or TCP). Overlay networks partially address the problems associated with Saltzer's end-to-end argument by allowing the generation of more application-specific network abstractions.

The chapter concluded with a case study of the MPI specification, developed by the high-performance computing community and featuring flexible support for message passing together with additional support for multiway message passing.

## EXERCISES

- 4.1 Is it conceivably useful for a port to have several receivers? *page 148*
- 4.2 A server creates a port that it uses to receive requests from clients. Discuss the design issues concerning the relationship between the name of this port and the names used by clients. *page 148*
- 4.3 The programs in Figure 4.3 and Figure 4.4 are available at [www.cdk5.net/ipc](http://www.cdk5.net/ipc). Use them to make a test kit to determine the conditions in which datagrams are sometimes dropped. Hint: the client program should be able to vary the number of messages sent and their size; the server should detect when a message from a particular client is missed. *page 150*
- 4.4 Use the program in Figure 4.3 to make a client program that repeatedly reads a line of input from the user, sends it to the server in a UDP datagram message, then receives a message from the server. The client sets a timeout on its socket so that it can inform the user when the server does not reply. Test this client program with the server program in Figure 4.4. *page 150*
- 4.5 The programs in Figure 4.5 and Figure 4.6 are available at [www.cdk5.net/ipc](http://www.cdk5.net/ipc). Modify them so that the client repeatedly takes a line of user's input and writes it to the stream and the server reads repeatedly from the stream, printing out the result of each read. Make a comparison between sending data in UDP datagram messages and over a stream. *page 153*
- 4.6 Use the programs developed in Exercise 4.5 to test the effect on the sender when the receiver crashes, and vice-versa. *page 153*
- 4.7 Sun XDR marshals data by converting it into a standard big-endian form before transmission. Discuss the advantages and disadvantages of this method when compared with CORBA CDR. *page 160*
- 4.8 Sun XDR aligns each primitive value on a 4-byte boundary, whereas CORBA CDR aligns a primitive value of size  $n$  on an  $n$ -byte boundary. Discuss the trade-offs in choosing the sizes occupied by primitive values. *page 160*
- 4.9 Why is there no explicit data typing in CORBA CDR? *page 160*
- 4.10 Write an algorithm in pseudo-code to describe the serialization procedure described in Section 4.3.2. The algorithm should show when handles are defined or substituted for classes and instances. Describe the serialized form that your algorithm would produce when serializing an instance of the following class, *Couple*:

```

class Couple implements Serializable{
    private Person one;
    private Person two;
    public Couple(Person a, Person b) {
        one = a;
        two = b;
    }
}

```

*page 162*

- 4.11 Write an algorithm in pseudo-code to describe deserialization of the serialized form produced by the algorithm defined in Exercise 4.10. Hint: use reflection to create a class from its name, to create a constructor from its parameter types and to create a new instance of an object from the constructor and the argument values. *page 162*
- 4.12 Why can't binary data be represented directly in XML, for example, by representing it as Unicode byte values? XML elements can carry strings represented as *base64*. Discuss the advantages or disadvantages of using this method to represent binary data. *page 164*
- 4.13 Define a class whose instances represent remote object references. It should contain information similar to that shown in Figure 4.13 and should provide access methods needed by higher-level protocols (see request-reply in Chapter 5, for example). Explain how each of the access methods will be used by that protocol. Give a justification for the type chosen for the instance variable containing information about the interface of the remote object. *page 168*
- 4.14 IP multicast provides a service that suffers from omission failures. Make a test kit, possibly based on the program in Figure 4.14, to discover the conditions under which a multicast message is sometimes dropped by one of the members of the multicast group. The test kit should be designed to allow for multiple sending processes. *page 170*
- 4.15 Outline the design of a scheme that uses message retransmissions with IP multicast to overcome the problem of dropped messages. Your scheme should take the following points into account:
- i) There may be multiple senders.
  - ii) Generally only a small proportion of messages are dropped.
  - iii) Recipients may not necessarily send a message within any particular time limit.
- Assume that messages that are not dropped arrive in sender order. *page 173*
- 4.16 Your solution to Exercise 4.15 should have overcome the problem of dropped messages in IP multicast. In what sense does your solution differ from the definition of reliable multicast? *page 173*
- 4.17 Devise a scenario in which multicasts sent by different clients are delivered to two group members in different orders. Assume that some form of message retransmission is in use, but that messages that are not dropped arrive in sender order. Suggest how recipients might remedy this situation. *page 173*
- 4.18 Revisit the Internet architecture as introduced in Chapter 3 (see Figures 3.12 and 3.14). What impact does the introduction of overlay networks have on this architecture, and in particular on the programmer's conceptual view of the Internet? *page 175*
- 4.19 What are the main arguments for adopting a super node approach in Skype? *page 177*
- 4.20 As discussed in Section 4.6, MPI offers a number of variants of *send* including the *MPI\_Rsend* operation, which assumes the receiver is ready to receive at the time of sending. What optimizations in implementation are possible if this assumption is correct and what are the repercussions of this assumption being false? *page 180*

## 5.5 Case study: Java RMI

Java RMI extends the Java object model to provide support for distributed objects in the Java language. In particular, it allows objects to invoke methods on remote objects using the same syntax as for local invocations. In addition, type checking applies equally to remote invocations as to local ones. However, an object making a remote invocation is aware that its target is remote because it must handle *RemoteExceptions*; and the implementor of a remote object is aware that it is remote because it must implement the *Remote* interface. Although the distributed object model is integrated into Java in a natural way, the semantics of parameter passing differ because the invoker and target are remote from one another.

The programming of distributed applications in Java RMI should be relatively simple because it is a single-language system – remote interfaces are defined in the Java language. If a multiple-language system such as CORBA is used, the programmer needs to learn an IDL and to understand how it maps onto the implementation language. However, even in a single-language system, the programmer of a remote object must consider its behaviour in a concurrent environment.

In the remainder of this introduction, we give an example of a remote interface, then discuss the parameter-passing semantics with reference to the example. Finally, we discuss the downloading of classes and the binder. The second section of this case study discusses how to build client and server programs for the example interface. The third section is concerned with the design and implementation of Java RMI. For full details of Java RMI, see the tutorial on remote invocation [[java.sun.com](http://java.sun.com) 1].

In this case study, the CORBA case study in Chapter 8 and the discussion of web services in Chapter 9, we use a *shared whiteboard* as an example. This is a distributed program that allows a group of users to share a common view of a drawing surface containing graphical objects, such as rectangles, lines and circles, each of which has been drawn by one of the users. The server maintains the current state of a drawing by providing an operation for clients to inform it about the latest shape one of their users has drawn and keeping a record of all the shapes it has received. The server also provides operations allowing clients to retrieve the latest shapes drawn by other users by polling the server. The server has a version number (an integer) that it increments each time a new shape arrives and attaches to the new shape. The server provides operations allowing clients to enquire about its version number and the version number of each shape, so that they may avoid fetching shapes that they already have.

**Remote interfaces in Java RMI** • Remote interfaces are defined by extending an interface called *Remote* provided in the *java.rmi* package. The methods must throw *RemoteException*, but application-specific exceptions may also be thrown. Figure 5.16 shows an example of two remote interfaces called *Shape* and *ShapeList*. In this example, *GraphicalObject* is a class that holds the state of a graphical object – for example, its type, its position, enclosing rectangle, line colour and fill colour – and provides operations for accessing and updating its state. *GraphicalObject* must implement the *Serializable* interface. Consider the interface *Shape* first: the *getVersion* method returns an integer, whereas the *getAllState* method returns an instance of the class *GraphicalObject*. Now consider the interface *ShapeList*: its *newShape* method passes an instance of *GraphicalObject* as its argument but returns an object with a remote



**Figure 5.16** Java Remote interfaces *Shape* and *ShapeList*

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException;
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

---

interface (that is, a remote object) as its result. An important point to note is that both ordinary objects and remote objects can appear as arguments and results in a remote interface. The latter are always denoted by the name of their remote interface. In the next subsection, we discuss how ordinary objects and remote objects are passed as arguments and results.

**Parameter and result passing** • In Java RMI, the parameters of a method are assumed to be *input* parameters and the result of a method is a single *output* parameter. Section 4.3.2 describes Java serialization, which is used for marshalling arguments and results in Java RMI. Any object that is serializable – that is, that implements the *Serializable* interface – can be passed as an argument or result in Java RMI. All primitive types and remote objects are serializable. Classes for arguments and result values are downloaded to the recipient by the RMI system where necessary.

*Passing remote objects:* When the type of a parameter or result value is defined as a remote interface, the corresponding argument or result is always passed as a remote object reference. For example, in Figure 5.16, line 2, the return value of the method *newShape* is defined as *Shape* – a remote interface. When a remote object reference is received, it can be used to make RMI calls on the remote object to which it refers.

*Passing non-remote objects:* All serializable non-remote objects are copied and passed by value. For example, in Figure 5.16 (lines 2 and 1) the argument of *newShape* and the return value of *getAllState* are both of type *GraphicalObject*, which is serializable and is passed by value. When an object is passed by value, a new object is created in the receiver's process. The methods of this new object can be invoked locally, possibly causing its state to differ from the state of the original object in the sender's process.

Thus, in our example, the client uses the method *newShape* to pass an instance of *GraphicalObject* to the server; the server makes a remote object of type *Shape* containing the state of the *GraphicalObject* and returns a remote object reference to it. The arguments and return values in a remote invocation are serialized to a stream using the method described in Section 4.3.2, with the following modifications:



**Figure 5.17** The *Naming* class of Java RMI registry

*void rebind (String name, Remote obj)*

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 5.18, line 3.

*void bind (String name, Remote obj)*

This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

*void unbind (String name, Remote obj)*

This method removes a binding.

*Remote lookup(String name)*

This method is used by clients to look up a remote object by name, as shown in Figure 5.20, line 1. A remote object reference is returned.

*String [] list()*

This method returns an array of *Strings* containing the names bound in the registry.

1. Whenever an object that implements the *Remote* interface is serialized, it is replaced by its remote object reference, which contains the name of its (the remote object's) class.
2. When any object is serialized, its class information is annotated with the location of the class (as a URL), enabling the class to be downloaded by the receiver.

**Downloading of classes** • Java is designed to allow classes to be downloaded from one virtual machine to another. This is particularly relevant to distributed objects that communicate by means of remote invocation. We have seen that non-remote objects are passed by value and remote objects are passed by reference as arguments and results of RMIs. If the recipient does not already possess the class of an object passed by value, its code is downloaded automatically. Similarly, if the recipient of a remote object reference does not already possess the class for a proxy, its code is downloaded automatically. This has two advantages:

1. There is no need for every user to keep the same set of classes in their working environment.
2. Both client and server programs can make transparent use of instances of new classes whenever they are added.

As an example, consider the whiteboard program and suppose that its initial implementation of *GraphicalObject* does not allow for text. A client with a textual object can implement a subclass of *GraphicalObject* that deals with text and pass an instance to the server as an argument of the *newShape* method. After that, other clients may retrieve the instance using the *getAllState* method. The code of the new class will be downloaded automatically from the first client to the server and then to other clients as needed.

**Figure 5.18** Java class *ShapeListServer* with *main* method

```

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();           1
            ShapeList stub =                                           2
                (ShapeList) UnicastRemoteObject.exportObject(aShapeList,0);3
            Naming.rebind("/bruno.ShapeList", stub );                 4
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());
        }
    }
}

```

**RMRegistry** • The RMRegistry is the binder for Java RMI. An instance of RMRegistry should normally run on every server computer that hosts remote objects. It maintains a table mapping textual, URL-style names to references to remote objects hosted on that computer. It is accessed by methods of the *Naming* class, whose methods take as an argument a URL-formatted string of the form:

*//computerName:port/objectName*

where *computerName* and *port* refer to the location of the RMRegistry. If they are omitted, the local computer and default port are assumed. Its interface offers the methods shown in Figure 5.17, in which the exceptions are not listed – all of the methods can throw a *RemoteException*.

Used in this way, clients must direct their *lookup* enquiries to particular hosts. Alternatively, it is possible to set up a system-wide binding service. To achieve this, it is necessary to run an instance of the RMRegistry in the networked environment and then use the class *LocateRegistry*, which is in *java.rmi.registry*, to discover this registry. More specifically, this class contains a *getRegistry* method that returns an object of type *Registry* representing the remote binding service:

```
public static Registry getRegistry() throws RemoteException
```

Following this, it is then necessary to issue a call of *rebind* on this returned *Registry* object to establish a connection with the remote RMRegistry.

## 5.5.1 Building client and server programs

This section outlines the steps necessary to produce client and server programs that use the *Remote* interfaces *Shape* and *ShapeList* shown in Figure 5.16. The server program is a simplified version of a whiteboard server that implements the two interfaces *Shape* and *ShapeList*. We describe a simple polling client program and then introduce the callback

**Figure 5.19** Java class *ShapeListServant* implements interface *ShapeList*

```

import java.util.Vector;

public class ShapeListServant implements ShapeList {
    private Vector theList;           // contains the list of Shapes
    private int version;
    public ShapeListServant(){...}

    public Shape newShape(GraphicalObject g) {                1
        version++;
        Shape s = new ShapeServant( g, version);              2
        theList.addElement(s);
        return s;
    }
    public Vector allShapes(){...}
    public int getVersion() { ... }
}

```

technique that can be used to avoid the need to poll the server. Complete versions of the classes illustrated in this section are available at [www.cdk5.net/rmi](http://www.cdk5.net/rmi).

**Server program** • The server is a whiteboard server: it represents each shape as a remote object instantiated by a servant that implements the *Shape* interface and holds the state of a graphical object as well as its version number; it represents its collection of shapes by using another servant that implements the *ShapeList* interface and holds a collection of shapes in a *Vector*.

The server program consists of a *main* method and a servant class to implement each of its remote interfaces. The *main* method of the server class is shown in Figure 5.18, with the key steps contained in the lines marked 1 to 4:

- In line 1, the server creates an instance of *ShapeListServant*.
- Lines 2 and 3 use the method *exportObject* (defined on *UnicastRemoteObject*) to make this object available to the RMI runtime, thereby making it available to receive incoming invocations. The second parameter of *exportObject* specifies the TCP port to be used for incoming invocations. It is normal practice to set this to zero, implying that an anonymous port will be used (one that is generated by the RMI runtime). Using *UnicastRemoteObject* ensures that the resultant object lives only as long as the process in which it is created (an alternative is to make this an *Activatable* object that is, one that lives beyond the server instance).
- Finally, line 4 binds the remote object to a name in the RMI registry. Note that the value bound to the name is a remote object reference, and its type is the type of its remote interface – *ShapeList*.

The two servant classes are *ShapeListServant*, which implements the *ShapeList* interface, and *ShapeServant*, which implements the *Shape* interface. Figure 5.19 gives an outline of the class *ShapeListServant*.

**Figure 5.20** Java client of *ShapeList*

```

import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;

public class ShapeListClient{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        ShapeList aShapeList = null;
        try{
            aShapeList = (ShapeList) Naming.lookup("//bruno.ShapeList"); 1
            Vector sList = aShapeList.allShapes(); 2
        } catch(RemoteException e) {System.out.println(e.getMessage());}
        } catch(Exception e) {System.out.println("Client: " + e.getMessage());}
    }
}

```

The implementation of the methods of the remote interface in a servant class is completely straightforward because it can be done without any concern for the details of communication. Consider the method *newShape* in Figure 5.19 (line 1), which could be called a factory method because it allows the client to request the creation of a servant. It uses the constructor of *ShapeServant*, which creates a new servant containing the *GraphicalObject* and version number passed as arguments. The type of the return value of *newShape* is *Shape* – the interface implemented by the new servant. Before returning, the method *newShape* adds the new shape to its vector that contains the list of shapes (line 2).

The *main* method of a server needs to create a security manager to enable Java security to apply the protection appropriate for an RMI server. A default security manager called *RMISecurityManager* is provided. It protects the local resources to ensure that the classes that are loaded from remote sites cannot have any effect on resources such as files, but it differs from the standard Java security manager in allowing the program to provide its own class loader and to use reflection. If an RMI server sets no security manager, proxies and classes can only be loaded from the local classpath, in order to protect the program from code that is downloaded as a result of remote method invocations.

**Client program** • A simplified client for the *ShapeList* server is illustrated in Figure 5.20. Any client program needs to get started by using a binder to look up a remote object reference. Our client sets a security manager and then looks up a remote object reference for the remote object using the *lookup* operation of the RMI registry (line 1). Having obtained an initial remote object reference, the client continues by sending RMIs to that remote object or to others discovered during its execution according to the needs of its application. In our example, the client invokes the method *allShapes* in the remote object (line 2) and receives a vector of remote object references to all of the shapes currently stored in the server. If the client was implementing a whiteboard display, it would use the server's *getAllState* method in the *Shape* interface to retrieve each of the graphical objects in the vector and display them in a window. Each time the user finishes

drawing a graphical object, it will invoke the method *newShape* in the server, passing the new graphical object as its argument. The client will keep a record of the latest version number at the server, and from time to time it will invoke *getVersion* at the server to find out whether any new shapes have been added by other users. If so, it will retrieve and display them.

**Callbacks** • The general idea behind callbacks is that instead of clients polling the server to find out whether some event has occurred, the server should inform its clients whenever that event occurs. The term *callback* is used to refer to a server's action of notifying clients about an event. Callbacks can be implemented in RMI as follows:

- The client creates a remote object that implements an interface that contains a method for the server to call. We refer to this as a *callback object*.
- The server provides an operation allowing interested clients to inform it of the remote object references of their callback objects. It records these in a list.
- Whenever an event of interest occurs, the server calls the interested clients. For example, the whiteboard server would call its clients whenever a graphical object is added.

The use of callbacks avoids the need for a client to poll the objects of interest in the server and its attendant disadvantages:

- The performance of the server may be degraded by the constant polling.
- Clients cannot notify users of updates in a timely manner.

However, callbacks have problems of their own. First, the server needs to have up-to-date lists of the clients' callback objects, but clients may not always inform the server before they exit, leaving the server with incorrect lists. The *leasing* technique discussed in Section 5.4.3 can be used to overcome this problem. The second problem associated with callbacks is that the server needs to make a series of synchronous RMIs to the callback objects in the list. See Chapter 6 for some ideas about solving the second problem.

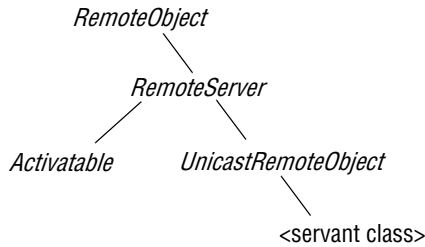
We illustrate the use of callbacks in the context of the whiteboard application. The *WhiteboardCallback* interface could be defined as follows:

```
public interface WhiteboardCallback implements Remote {
    void callback(int version) throws RemoteException;
};
```

This interface is implemented as a remote object by the client, enabling the server to send the client a version number whenever a new object is added. But before the server can do this, the client needs to inform the server about its callback object. To make this possible, the *ShapeList* interface requires additional methods such as *register* and *deregister*, defined as follows:

```
int register(WhiteboardCallback callback) throws RemoteException;
void deregister(int callbackId) throws RemoteException;
```

After the client has obtained a reference to the remote object with the *ShapeList* interface (for example, in Figure 5.20, line 1) and created an instance of its callback object, it uses the *register* method of *ShapeList* to inform the server that it is interested in receiving

**Figure 5.21** Classes supporting Java RMI

callbacks. The *register* method returns an integer (the *callbackId*) referring to the registration. When the client is finished it should call *deregister* to inform the server it no longer requires callbacks. The server is responsible for keeping a list of interested clients and notifying all of them each time its version number increases.

## 5.5.2 Design and implementation of Java RMI

The original Java RMI system used all of the components shown in Figure 5.15. But in Java 1.2, the reflection facilities were used to make a generic dispatcher and to avoid the need for skeletons. Prior to J2SE 5.0, the client proxies were generated by a compiler called *rmic* from the compiled server classes (not from the definitions of the remote interfaces). However, this step is no longer necessary with recent versions of J2SE, which contain support for the dynamic generation of stub classes at runtime.

**Use of reflection** • Reflection is used to pass information in request messages about the method to be invoked. This is achieved with the help of the class *Method* in the reflection package. Each instance of *Method* represents the characteristics of a particular method, including its class and the types of its arguments, return value and exceptions. The most interesting feature of this class is that an instance of *Method* can be invoked on an object of a suitable class by means of its *invoke* method. The *invoke* method requires two arguments: the first specifies the object to receive the invocation and the second is an array of *Object* containing the arguments. The result is returned as type *Object*.

To return to the use of the *Method* class in RMI: the proxy has to marshal information about a method and its arguments into the *request* message. For the method it marshals an object of class *Method*. It puts the arguments into an array of *Objects* and then marshals that array. The dispatcher unmarshals the *Method* object and its arguments in the array of *Objects* from the *request* message. As usual, the remote object reference of the target will have been unmarshalled and the corresponding local object reference obtained from the remote reference module. The dispatcher then calls the *Method* object's *invoke* method, supplying the target and the array of argument values. When the method has been executed, the dispatcher marshals the result or any exceptions into the *reply* message. Thus the dispatcher is generic – that is, the same dispatcher can be used for all classes of remote object, and no skeletons are required.

**Java classes supporting RMI** • Figure 5.21 shows the inheritance structure of the classes supporting Java RMI servers. The only class that the programmer need be aware of is *UnicastRemoteObject*, which every simple servant class needs to extend. The class *UnicastRemoteObject* extends an abstract class called *RemoteServer*, which provides

abstract versions of the methods required by remote servers. *UnicastRemoteObject* was the first example of *RemoteServer* to be provided. Another called *Activatable* is available for providing activatable objects. Further alternatives might provide for replicated objects. The class *RemoteServer* is a subclass of *RemoteObject* that has an instance variable holding the remote object reference and provides the following methods:

*equals*: This method compares remote object references.

*toString*: This method gives the contents of the remote object reference as a *String*.

*readObject*, *writeObject*: These methods deserialize/serialize remote objects.

In addition, the *instanceOf* operator can be used to test remote objects.

## 5.6 Summary

This chapter has discussed three paradigms for distributed programming – request-reply protocols, remote procedure calls and remote method invocation. All of these paradigms provide mechanisms for distributed independent entities (processes, objects, components or services) to communicate directly with one another.

Request-reply protocols provide lightweight and minimal support for client-server computing. Such protocols are often used in environments where overheads of communication must be minimized – for example, in embedded systems. Their more common role is to support either RPC or RMI, as discussed below.

The remote procedure call approach was a significant breakthrough in distributed systems, providing higher-level support for programmers by extending the concept of a procedure call to operate in a networked environment. This provides important levels of transparency in distributed systems. However, due to their different failure and performance characteristics and to the possibility of concurrent access to servers, it is not necessarily a good idea to make remote procedure calls appear to be exactly the same as local calls. Remote procedure calls provide a range of invocation semantics, from *maybe* invocations through to *at-most-once* semantics.

The distributed object model is an extension of the local object model used in object-based programming languages. Encapsulated objects form useful components in a distributed system, since encapsulation makes them entirely responsible for managing their own state, and local invocation of methods can be extended to remote invocation. Each object in a distributed system has a remote object reference (a globally unique identifier) and a remote interface that specifies which of its operations can be invoked remotely.

Middleware implementations of RMI provide components (including proxies, skeletons and dispatchers) that hide the details of marshalling, message passing and locating remote objects from client and server programmers. These components can be generated by an interface compiler. Java RMI extends local invocation to remote invocation using the same syntax, but remote interfaces must be specified by extending an interface called *Remote* and making each method throw a *RemoteException*. This ensures that programmers know when they make remote invocations or implement remote objects, enabling them to handle errors or to design objects suitable for concurrent access.