2

SYSTEM MODELS

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

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

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

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

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

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

2.1 Introduction

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

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

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

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

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

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

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

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

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

2.2 Physical models

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

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

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

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

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

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

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

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

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

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

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

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

2.3 Architectural models

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

Figure 2.1 Generations of distributed systems

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

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

The section adopts a three-stage approach:

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

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

2.3.1 Architectural elements

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Indirect communication is discussed in more detail in Chapter 6.

Key techniques for indirect communication include:

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

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

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

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

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

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

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

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

Figure 2.2 Communicating entities and communication paradigms

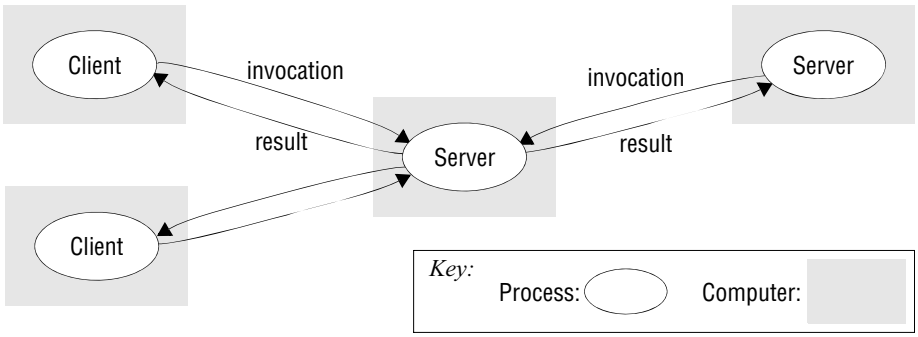
Communicating entities (what is communicating)		*Communication paradigms (how they communicate)*		
System-oriented entities	*Problem-oriented entities*	*Interprocess communication*	*Remote invocation*	*Indirect communication*
Nodes	Objects	Message passing	Request-reply	Group communication
Processes	Components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

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

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

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

Figure 2.3 Clients invoke individual servers



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

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

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

Figure 2.4 A service provided by multiple servers

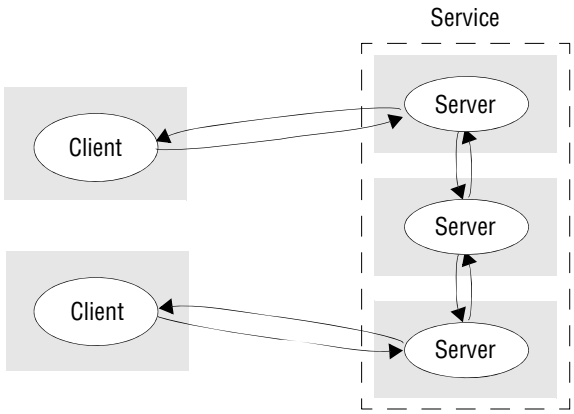


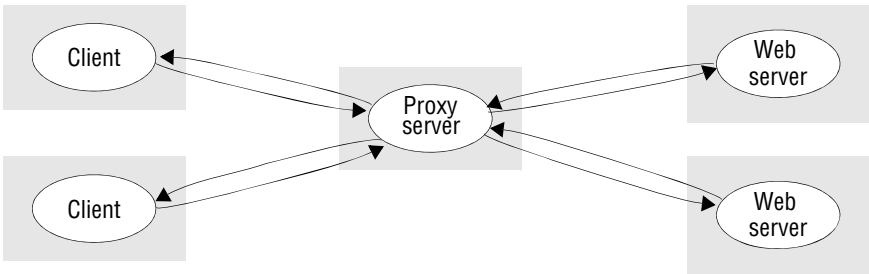
Figure 2.4 illustrates the form of a peer-to-peer application. Applications are composed of large numbers of peer processes running on separate computers and the pattern of communication between them depends entirely on application requirements. A large number of data objects are shared, an individual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

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

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

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

Figure 2.5 Web proxy server



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

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

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

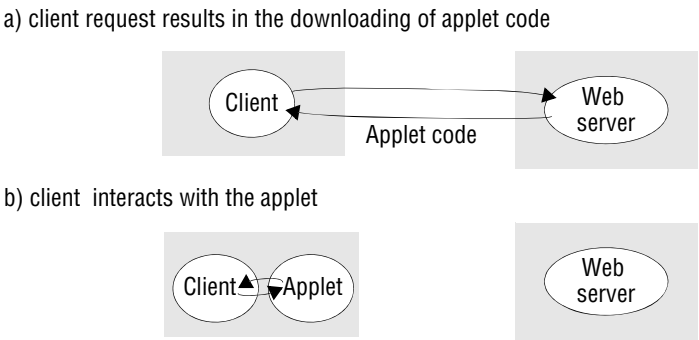
An example of a service based on replicated data is the Sun Network Information Service (NIS), which is used to enable all the computers on a LAN to access the same user authentication data when users log in. Each NIS server has its own replica of a common password file containing a list of users' login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

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

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

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

Figure 2.6 Web applets



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

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

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

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

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

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

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

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

2.3.2 Architectural patterns

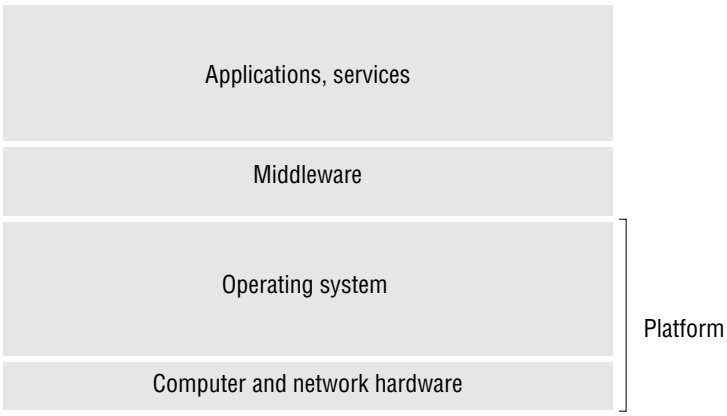
Architectural patterns build on the more primitive architectural elements discussed above and provide composite recurring structures that have been shown to work well in given circumstances. They are not themselves necessarily complete solutions but rather offer partial insights that, when combined with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

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

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

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

Figure 2.7 Software and hardware service layers in distributed systems



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

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

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

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

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

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

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

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

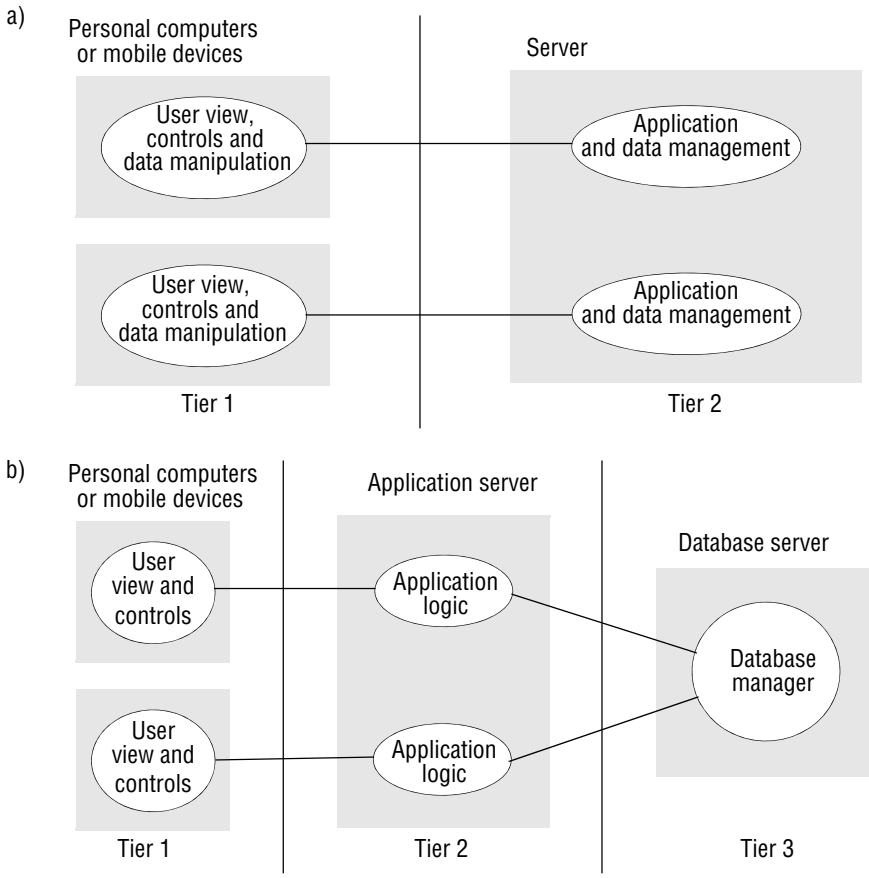
In the two-tier solution, the three aspects mentioned above must be partitioned into two processes, the client and the server. This is most commonly done by splitting the application logic, with some residing in the client and the remainder in the server (although other solutions are also possible).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

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

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

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

Figure 2.8 Two-tier and three-tier architectures



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

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

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

Figure 2.9 AJAX example: soccer score updates

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

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

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

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

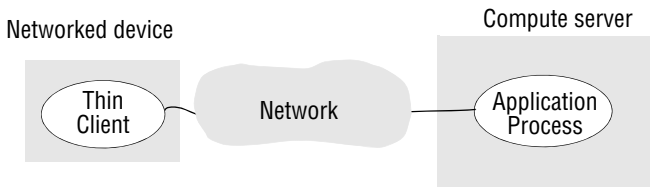
AJAX is the second innovative step that was needed to enable major interactive web applications to be developed and deployed. It enables Javascript front-end programs to request new data directly from server programs. Any data items can be requested and the current page updated selectively to show the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

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

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

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

Figure 2.10 Thin clients and computer servers



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

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

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

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

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

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

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

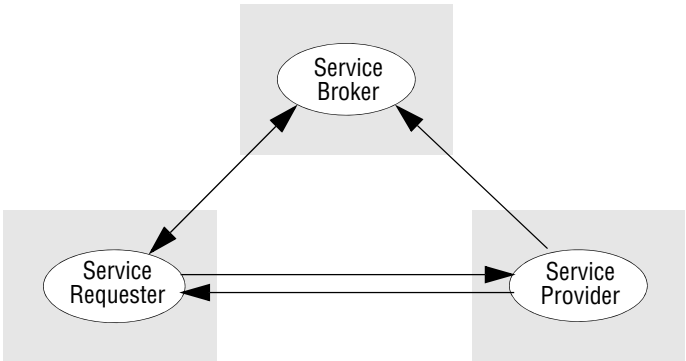
The concept is straightforward, providing remote access to graphical user interfaces. In this solution, a VNC client (or viewer) interacts with a VNC server through a VNC protocol. The protocol operates at a primitive level in terms of graphics support, based on framebuffers and featuring one operation: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

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

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

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

Figure 2.11 The web service architectural pattern



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

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

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

2.3.3 Associated middleware solutions

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

Figure 2.12 Categories of middleware

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4 Fundamental models

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

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

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

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

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

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

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

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

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

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

2.4.1 Interaction model

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

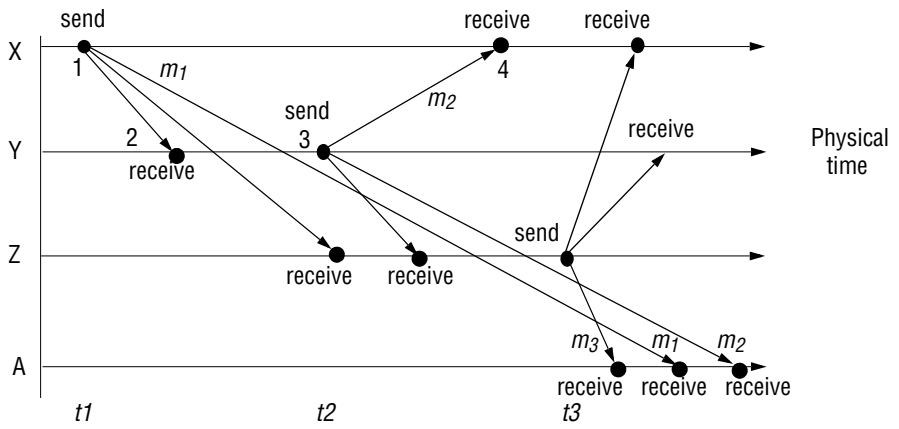
The asynchronous model allows no assumptions about the time intervals involved in any execution. This exactly models the Internet, in which there is no intrinsic bound on server or network load and therefore on how long it takes, for example, to transfer a file using FTP. Sometimes an email message can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

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

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

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

Figure 2.13 Real-time ordering of events



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

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

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

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

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

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

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

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

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

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

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

Y receives m_1 before sending m_2.

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

2.4.2 Failure model

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

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

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

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

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

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

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

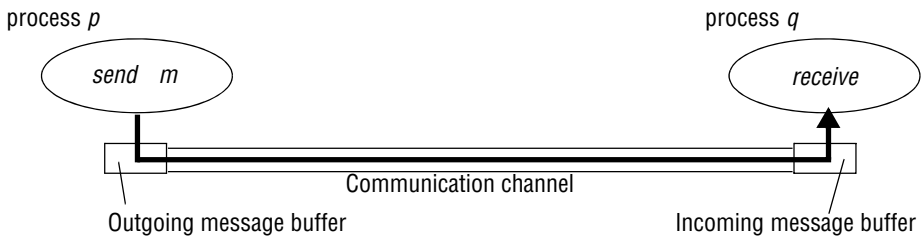
The communication channel produces an omission failure if it does not transport a message from p's outgoing message buffer to q's incoming message buffer. This is known as 'dropping messages' and is generally caused by lack of buffer space at the receiver or at an intervening gateway, or by a network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

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

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

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

Figure 2.14 Processes and channels



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

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

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

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

Figure 2.15 Omission and arbitrary failures

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

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

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

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

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

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

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

Figure 2.16 Timing failures

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

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

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

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

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

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

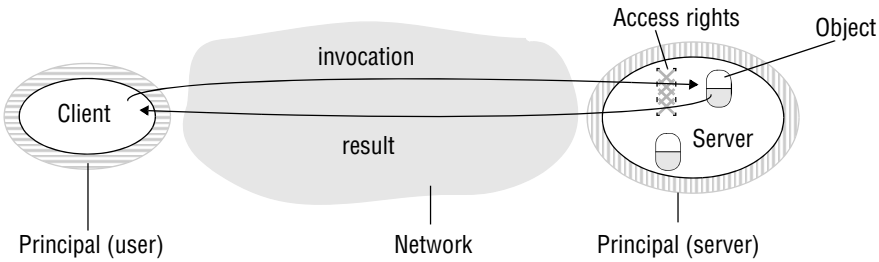
The threats to integrity come from two independent sources:

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

2.4.3 Security model

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

Figure 2.17 Objects and principals



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

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

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

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

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

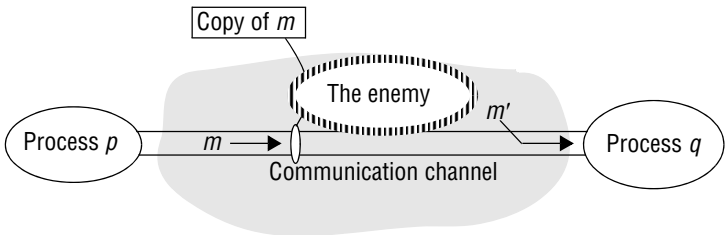
Thus we must include users in our model as the beneficiaries of access rights. We do so by associating with each invocation and each result the authority on which it is issued. Such an authority is called a *principal*. A principal may be a user or a process. In our illustration, the invocation comes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

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

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

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

Figure 2.18 The enemy



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

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

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

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

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

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

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

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

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

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

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

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

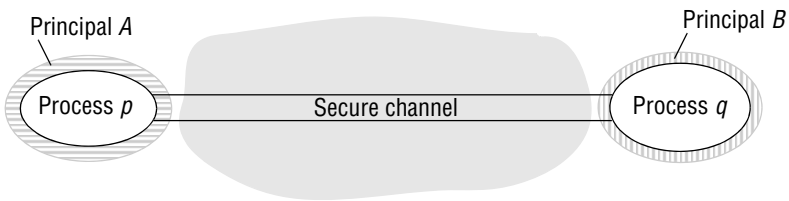
Cryptography is the science of keeping messages secure, and *encryption* is the process of scrambling a message in such a way as to hide its contents. Modern cryptography is based on encryption algorithms that use secret keys – large numbers that are difficult to guess – to transform data in a manner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

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

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

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

Figure 2.19 Secure channels



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

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

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

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

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

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

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

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

2.5 Summary

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

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

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

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

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

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

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

EXERCISES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This page intentionally left blank