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

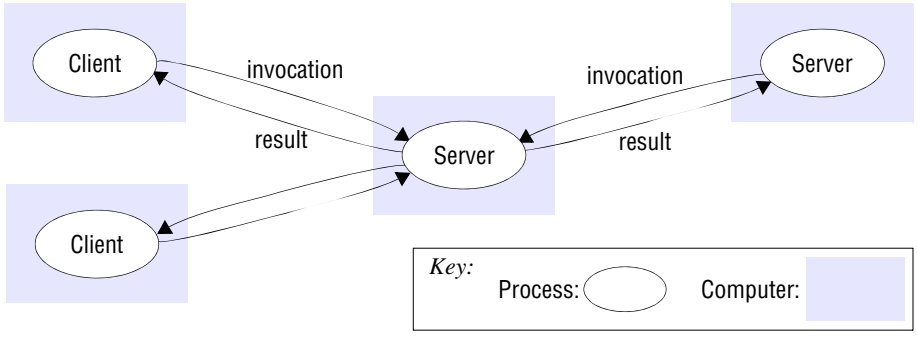
Communicating entities (what is communicating)		*Communication paradigms (how they communicate)*		
System-oriented entities	*Problem-oriented entities*	*Interprocess communication*	*Remote invocation*	*Indirect communication*
Nodes	Objects	Message passing	Request-reply	Group communication
Processes	Components	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 2.4a Peer-to-peer architecture

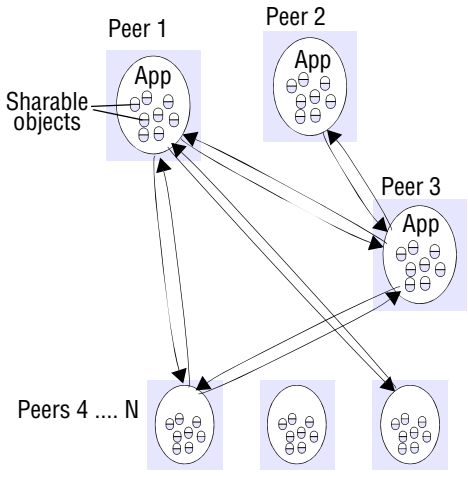


Figure 2.4b A service provided by multiple servers

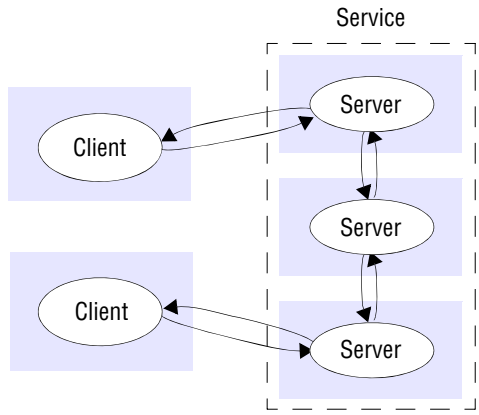


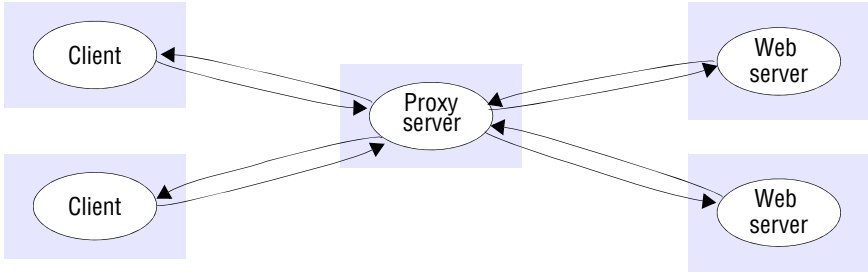
Figure 2.4a illustrates the form of a peer-to-peer application. Applications are composed of large numbers of peer processes running on separate computers and the pattern of communication between them depends entirely on application requirements. A large number of data objects are shared, an individual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

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

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

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

Figure 2.5 Web proxy server



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

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

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

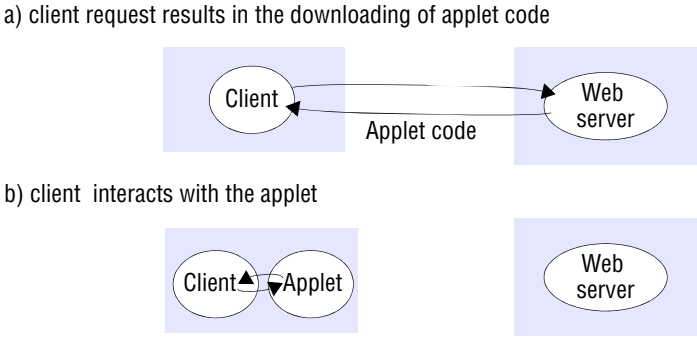
An example of a service based on replicated data is the Sun Network Information Service (NIS), which is used to enable all the computers on a LAN to access the same user authentication data when users log in. Each NIS server has its own replica of a common password file containing a list of users' login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

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

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

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

Figure 2.6 Web applets



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

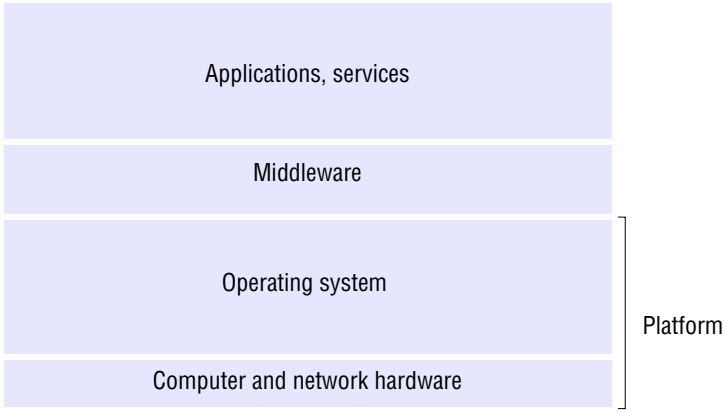
Architectural patterns build on the more primitive architectural elements discussed above and provide composite recurring structures that have been shown to work well in given circumstances. They are not themselves necessarily complete solutions but rather offer partial insights that, when combined with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