



18

Distributed software engineering

Objectives

The objective of this chapter is to introduce distributed systems engineering and distributed systems architectures. When you have read this chapter, you will:

- know the key issues that have to be considered when designing and implementing distributed software systems;
- understand the client–server computing model and the layered architecture of client–server systems;
- have been introduced to commonly used patterns for distributed systems architectures and know the types of system for which each architecture is most applicable;
- understand the notion of software as a service, providing web-based access to remotely deployed application systems.

Contents

- 18.1** Distributed systems issues
- 18.2** Client–server computing
- 18.3** Architectural patterns for distributed systems
- 18.4** Software as a service

Virtually all large computer-based systems are now distributed systems. A distributed system is one involving several computers, in contrast with centralized systems where all of the system components execute on a single computer. Tanenbaum and Van Steen (2007) define a distributed system to be:

“...a collection of independent computers that appears to the user as a single coherent system.”

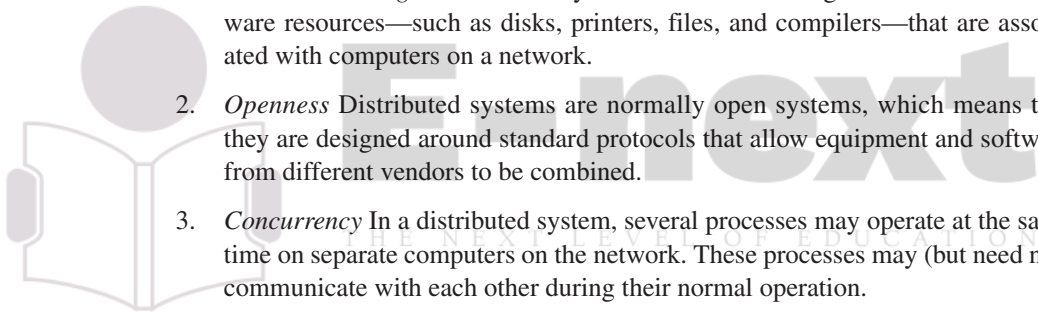
Obviously, the engineering of distributed systems has a great deal in common with the engineering of any other software. However, there are specific issues that have to be taken into account when designing this type of system. These arise because the system components may be running on independently managed computers and they communicate across a network.

Coulouris et al. (2005) identify the following advantages of using a distributed approach to systems development:

1. *Resource sharing* A distributed system allows the sharing of hardware and software resources—such as disks, printers, files, and compilers—that are associated with computers on a network.
2. *Openness* Distributed systems are normally open systems, which means that they are designed around standard protocols that allow equipment and software from different vendors to be combined.
3. *Concurrency* In a distributed system, several processes may operate at the same time on separate computers on the network. These processes may (but need not) communicate with each other during their normal operation.
4. *Scalability* In principle at least, distributed systems are scalable in that the capabilities of the system can be increased by adding new resources to cope with new demands on the system. In practice, the network linking the individual computers in the system may limit the system scalability.
5. *Fault tolerance* The availability of several computers and the potential for replicating information means that distributed systems can be tolerant of some hardware and software failures (see Chapter 13). In most distributed systems, a degraded service can be provided when failures occur; complete loss of service only occurs when there is a network failure.

For large-scale organizational systems, these advantages mean that distributed systems have largely replaced mainframe legacy systems that were developed in the 1990s. However, there are many personal computer application systems (e.g., photo editing systems) that are not distributed and which run on a single computer system. Most embedded systems are also single processor systems.

Distributed systems are inherently more complex than centralized systems. This makes them more difficult to design, implement, and test. It is harder to understand the emergent properties of distributed systems because of the complexity of the



interactions between system components and the system infrastructure. For example, rather than the performance of the system being dependent on the execution speed of one processor, it depends on the network bandwidth, the network load, and the speed of all of the computers that are part of the system. Moving resources from one part of the system to another can significantly affect the system's performance.

Furthermore, as all users of the WWW know, distributed systems are unpredictable in their response. The response time depends on the overall load on the system, its architecture and the network load. As all of these may change over a short time, the time taken to respond to a user request may vary dramatically from one request to another.

The most important development that has affected distributed software systems in the past few years is the service-oriented approach. Much of this chapter focuses on general issues of distributed systems, but I cover the notion of applications deployed as services in Section 18.4. This complements the material in Chapter 19, which focuses on services as components in a service-oriented architecture, and more general issues of service-oriented software engineering.

18.1 Distributed systems issues



As I discussed in the introduction to this chapter, distributed systems are more complex than systems that run on a single processor. This complexity arises because it is practically impossible to have a top-down model of control of these systems. The nodes in the system that deliver functionality are often independent systems with no single authority in charge of them. The network connecting these nodes is a separately managed system. It is a complex system in its own right and cannot be controlled by the owners of systems using the network. There is therefore an inherent unpredictability in the operation of distributed systems that has to be taken into account by the system designer.

Some of the most important design issues that have to be considered in distributed systems engineering are:

1. *Transparency* To what extent should the distributed system appear to the user as a single system? When it is useful for users to understand that the system is distributed?
2. *Openness* Should a system be designed using standard protocols that support interoperability or should more specialized protocols be used that restrict the freedom of the designer?
3. *Scalability* How can the system be constructed so that it is scaleable? That is, how can the overall system be designed so that its capacity can be increased in response to increasing demands made on the system?
4. *Security* How can usable security policies be defined and implemented that apply across a set of independently managed systems?



CORBA – Common Object Request Broker Architecture

CORBA is a well-known specification for a middleware system that was developed in the 1990s by the Object Management Group. It was intended as an open standard that would allow the development of middleware to support distributed component communications and execution, plus provide a set of standard services that could be used by these components.

Several implementations of CORBA were produced but the system never achieved critical mass. Users preferred proprietary systems or moved to service-oriented architectures.

<http://www.SoftwareEngineering-9.com/Web/DistribSys/Corba.html>

5. *Quality of service* How should the quality of service that is delivered to system users be specified and how should the system be implemented to deliver an acceptable quality of service to all users?
6. *Failure management* How can system failures be detected, contained (so that they have minimal effects on other components in the system), and repaired?



In an ideal world, the fact that a system is distributed would be transparent to users. This means that users would see the system as a single system whose behavior is not affected by the way that the system is distributed. In practice, this is impossible to achieve. Central control of a distributed system is impossible and, as a result, individual computers in a system may behave differently at different times. Furthermore, because it always takes a finite length of time for signals to travel across a network, network delays are unavoidable. The length of these delays depends on the location of resources in the system, the quality of the user's network connection, and the network load.

The design approach to achieving transparency depends on creating abstractions of the resources in a distributed system so that the physical realization of these resources can be changed without having to make changes in the application system. Middleware (discussed in Section 18.1.2) is used to map the logical resources referenced by a program onto the actual physical resources, and to manage the interactions between these resources.

In practice, it is impossible to make a system completely transparent and users, generally, are aware that they are dealing with a distributed system. You may therefore decide that it is best to expose the distribution to users. They can then be prepared for some of the consequences of distribution such as network delays, remote node failures, etc.

Open distributed systems are systems that are built according to generally accepted standards. This means that components from any supplier can be integrated into the system and can interoperate with the other system components. At the networking level, openness is now taken for granted with systems conforming to Internet protocols but at the component level, openness is still not universal. Openness implies that system components can be independently developed in any programming language and, if these conform to standards, they will work with other components.

The CORBA standard (Pope, 1997) developed in the 1990s, was intended to achieve this but this never achieved a critical mass of adopters. Rather, many companies chose to develop systems using proprietary standards for components from companies such as Sun and Microsoft. These provided better implementations and support software and better long-term support for industrial protocols.

Web service standards (discussed in Chapter 19) for service-oriented architectures were developed to be open standards. However, there is significant resistance to these standards because of their perceived inefficiency. Some developers of service-based systems have opted instead for so-called RESTful protocols as these have an inherently lower overhead than web service protocols.

The scalability of a system reflects its ability to deliver a high quality of service as demands on the system increase. Neuman (1994) identifies three dimensions of scalability:

1. *Size* It should be possible to add more resources to a system to cope with increasing numbers of users.
2. *Distribution* It should be possible to geographically disperse the components of a system without degrading its performance.
3. *Manageability* It should be possible to manage a system as it increases in size, even if parts of the system are located in independent organizations.

In terms of size, there is a distinction between scaling up and scaling out. Scaling up means replacing resources in the system with more powerful resources. For example, you may increase the memory in a server from 16 GB to 64 GB. Scaling out means adding additional resources to the system (e.g., an extra web server to work alongside an existing server). Scaling out is often more cost effective than scaling up but usually means that the system has to be designed so that concurrent processing is possible.

I have discussed general security issues and issues of security engineering in Part 2 of this book. However, when a system is distributed, the number of ways that the system may be attacked is significantly increased, compared to centralized systems. If a part of the system is successfully attacked then the attacker may be able to use this as a ‘back door’ into other parts of the system.

The types of attacks that a distributed system must defend itself against are the following:

1. *Interception*, where communications between parts of the system are intercepted by an attacker so that there is a loss of confidentiality.
2. *Interruption*, where system services are attacked and cannot be delivered as expected. Denial of service attacks involve bombarding a node with illegitimate service requests so that it cannot deal with valid requests.
3. *Modification*, where data or services in the system are changed by an attacker.
4. *Fabrication*, where an attacker generates information that should not exist and then uses this to gain some privileges. For example, an attacker may generate a false password entry and use this to gain access to a system.

The major difficulty in distributed systems is establishing a security policy that can be reliably applied to all of the components in a system. As I discussed in Chapter 11, a security policy sets out the level of security to be achieved by a system. Security mechanisms, such as encryption and authentication, are used to enforce the security policy. The difficulties in a distributed system arise because different organizations may own parts of the system. These organizations may have mutually incompatible security policies and security mechanisms. Security compromises may have to be made in order to allow the systems to work together.

The quality of service (QoS) offered by a distributed system reflects the system's ability to deliver its services dependably and with a response time and throughput that is acceptable to its users. Ideally, the QoS requirements should be specified in advance and the system designed and configured to deliver that QoS. Unfortunately, this is not always practicable, for two reasons:

1. It may not be cost effective to design and configure the system to deliver a high QoS under peak load. This could involve making resources available that are unused for much of the time. One of the main arguments for 'cloud computing' is that it partially addresses this problem. Using a cloud, it is easy to add resources as demand increases.
2. The QoS parameters may be mutually contradictory. For example, increased reliability may mean reduced throughput, as checking procedures are introduced to ensure that all system inputs are valid.

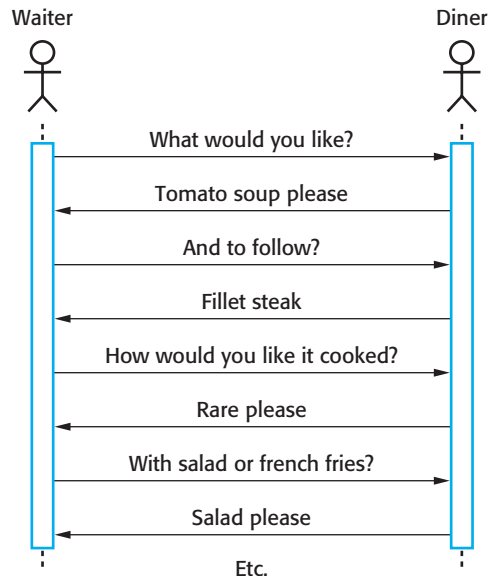
QoS is particularly critical when the system is dealing with time-critical data such as sound or video streams. In these circumstances, if the QoS falls below a threshold value then the sound or video may become so degraded that it is impossible to understand. Systems dealing with sound and video should include QoS negotiation and management components. These should evaluate the QoS requirements against the available resources and, if these are insufficient, negotiate for more resources or for a reduced QoS target.

In a distributed system, it is inevitable that failures will occur, so the system has to be designed to be resilient to these failures. Failure is so ubiquitous that one flippant definition of a distributed system suggested by Leslie Lamport, a prominent distributed systems researcher, is:

"You know that you have a distributed system when the crash of a system that you've never heard of stops you getting any work done."

Failure management involves applying the fault tolerance techniques discussed in Chapter 13. Distributed systems should therefore include mechanisms for discovering if a component of the system has failed, should continue to deliver as many services as possible in spite of that failure and, as far as possible, should automatically recover from the failure.

Figure 18.1 Procedural interaction between a diner and a waiter



18.1.1 Models of interaction

There are two fundamental types of interaction that may take place between the computers in a distributed computing system: procedural interaction and message-based interaction. Procedural interaction involves one computer calling on a known service offered by some other computer and (usually) waiting for that service to be delivered. Message-based interaction involves the ‘sending’ computer defining information about what is required in a message, which is then sent to another computer. Messages usually transmit more information in a single interaction than a procedure call to another machine.

To illustrate the difference between procedural and message-based interaction, consider a situation where you are ordering a meal in a restaurant. When you have a conversation with the waiter, you are involved in a series of synchronous, procedural interactions that define your order. You make a request; the waiter acknowledges that request; you make another request, which is acknowledged; and so on. This is comparable to components interacting in a software system where one component calls methods from other components. The waiter writes down your order along with the order of other people with you. He or she then passes this order, which includes details of everything that has been ordered, to the kitchen to prepare the food. Essentially, the waiter is passing a message to the kitchen staff defining the food to be prepared. This is message-based interaction.

I have illustrated this in Figure 18.1, which shows the synchronous ordering process as a series of calls and in Figure 18.2, which shows a hypothetical XML message that defines an order made by the table of three people. The difference between these forms of information exchange is clear. The waiter takes the order as a series of interactions, with each interaction defining part of the order. However,

```

<starter>
  <dish name = "soup" type = "tomato" />
  <dish name = "soup" type = "fish" />
  <dish name = "pigeon salad" />
</starter>
<main course>
  <dish name = "steak" type = "sirloin" cooking = "medium" />
  <dish name = "steak" type = "fillet" cooking = "rare" />
  <dish name = "sea bass">
</main>
<accompaniment>
  <dish name = "french fries" portions = "2" />
  <dish name = "salad" portions = "1" />
</accompaniment>

```

Figure 18.2
Message-based
interaction
between a
waiter and the
kitchen staff

the waiter has a single interaction with the kitchen where the message defines the complete order.

Procedural communication in a distributed system is usually implemented using remote procedure calls (RPCs). In RPC one component calls another component as if it was a local procedure or method. The middleware in the system intercepts this call and passes it to a remote component. This carries out the required computation and, via the middleware, returns the result to the calling component. In Java, remote method invocations (RMI) are comparable with, though not identical to, RPCs. The RMI framework handles the invocation of remote methods in a Java program.

RPCs require a ‘stub’ for the called procedure to be accessible on the computer that is initiating the call. The stub is called and it translates the procedure parameters into a standard representation for transmission to the remote procedure. Through the middleware, it then sends the request for execution to the remote procedure. The remote procedure uses library functions to convert the parameters into the required format, carries out the computation, and then communicates the results via the ‘stub’ that is representing the caller.

Message-based interaction normally involves one component creating a message that details the services required from another component. Through the system middleware, this is sent to the receiving component. The receiver parses the message, carries out the computations, and creates a message for the sending component with the required results. This is then passed to the middleware for transmission to the sending component.

A problem with the RPC approach to interaction is that both the caller and the callee need to be available at the time of the communication, and they must know how to refer to each other. In essence, an RPC has the same requirements as a local procedure or method call. By contrast, in a message-based approach, unavailability can be tolerated as the message simply stays in a queue until the receiver becomes available. Furthermore, it is not necessary for the sender and receiver of the message to be aware of each other. They simply communicate with the middleware, which is responsible for ensuring that messages are passed to the appropriate system.

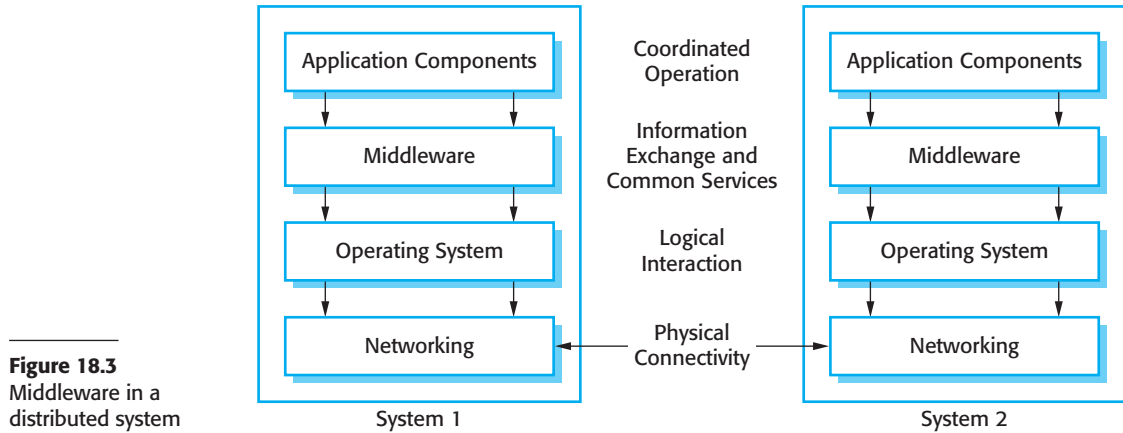


Figure 18.3
Middleware in a
distributed system

18.1.2 Middleware

The components in a distributed system may be implemented in different programming languages and may execute on completely different types of processor. Models of data, information representation, and protocols for communication may all be different. A distributed system therefore requires software that can manage these diverse parts, and ensure that they can communicate and exchange data.

The term ‘middleware’ is used to refer to this software—it sits in the middle between the distributed components of the system. This is illustrated in Figure 18.3, which shows that middleware is a layer between the operating system and application programs. Middleware is normally implemented as a set of libraries, which are installed on each distributed computer, plus a run-time system to manage communications.

Bernstein (1996) describes types of middleware that are available to support distributed computing. Middleware is general-purpose software that is usually bought off the shelf rather than written specially by application developers. Examples of middleware include software for managing communications with databases, transaction managers, data converters, and communication controllers.

In a distributed system, middleware normally provides two distinct types of support:

1. Interaction support, where the middleware coordinates interactions between different components in the system. The middleware provides location transparency in that it isn’t necessary for components to know the physical locations of other components. It may also support parameter conversion if different programming languages are used to implement components, event detection, and communication, etc.
2. The provision of common services, where the middleware provides reusable implementations of services that may be required by several components in the distributed system. By using these common services, components can easily interoperate and provide user services in a consistent way.

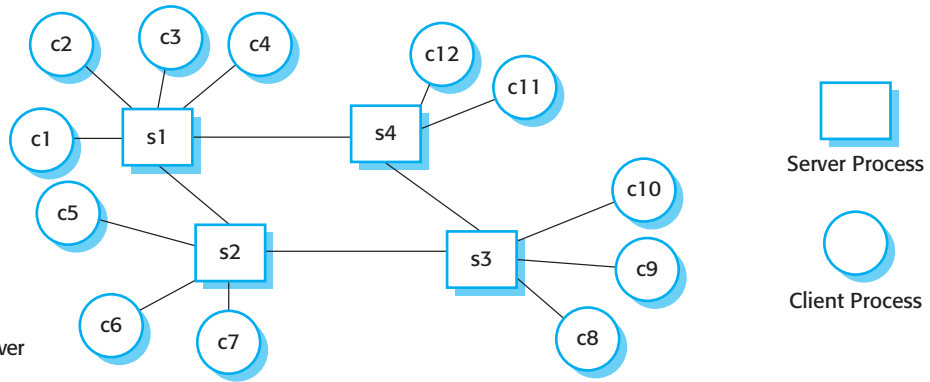


Figure 18.4 Client-server interaction

I have already given examples of the interaction support that middleware can provide in Section 18.1.1. You use middleware to support remote procedure and remote method calls, message exchange, etc.

Common services are those services that may be required by different components irrespective of the functionality of these components. As I discussed in Chapter 17, these may include security services (authentication and authorization), notification and naming services, and transaction management services, etc. You can think of these common services as being provided by a middleware container. You then deploy your component in that container and it can access and use these common services.

THE NEXT LEVEL OF EDUCATION

18.2 Client-server computing

Distributed systems that are accessed over the Internet are normally organized as client-server systems. In a client-server system, the user interacts with a program running on their local computer (e.g., a web browser or phone-based application). This interacts with another program running on a remote computer (e.g., a web server). The remote computer provides services, such as access to web pages, which are available to external clients. This client-server model, as I discussed in Chapter 6, is a very general architectural model of an application. It is not restricted to applications distributed across several machines. You can also use it as a logical interaction model where the client and the server run on the same computer.

In a client-server architecture, an application is modeled as a set of services that are provided by servers. Clients may access these services and present results to end users (Orfali and Harkey, 1998). Clients need to be aware of the servers that are available but do not know of the existence of other clients. Clients and servers are separate processes, as shown in Figure 18.4. This illustrates a situation in which there are four servers (s1–s4), that deliver different services. Each service has a set of associated clients that access these services.

Figure 18.5 Mapping of clients and servers to networked computers

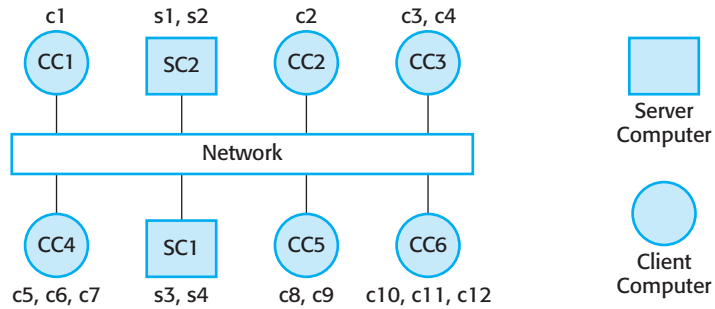


Figure 18.4 shows client and server processes rather than processors. It is normal for several client processes to run on a single processor. For example, on your PC, you may run a mail client that downloads mail from a remote mail server. You may also run a web browser that interacts with a remote web server and a print client that sends documents to a remote printer. Figure 18.5 illustrates the situation where the 12 logical clients shown in Figure 18.4 are running on six computers. The four server processes are mapped onto two physical server computers.

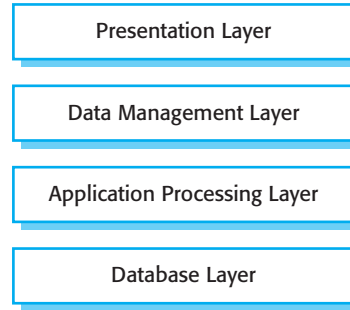
Several different server processes may run on the same processor but, often, servers are implemented as multiprocessor systems in which a separate instance of the server process runs on each machine. Load-balancing software distributes requests for service from clients to different servers so that each server does the same amount of work. This allows a higher volume of transactions with clients to be handled, without degrading the response to individual clients.

Client–server systems depend on there being a clear separation between the presentation of information and the computations that create and process that information. Consequently, you should design the architecture of distributed client–server systems so that they are structured into several logical layers, with clear interfaces between these layers. This allows each layer to be distributed to a different computer. Figure 18.6 illustrates this model, showing an application structured into four layers:

- A presentation layer that is concerned with presenting information to the user and managing all user interaction;
- A data management layer that manages the data that is passed to and from the client. This layer may implement checks on the data, generate web pages, etc.;
- An application processing layer that is concerned with implementing the logic of the application and so providing the required functionality to end users;
- A database layer that stores the data and provides transaction management services, etc.

The following section explains how different client–server architectures distribute these logical layers in different ways. The client–server model also underlies the notion of software as a service (SaaS), an increasingly important way of deploying software and accessing it over the Internet. I discuss this in Section 18.4.

Figure 18.6 Layered architectural model for client-server application



18.3 Architectural patterns for distributed systems

As I explained in the introduction to this chapter, designers of distributed systems have to organize their system designs to find a balance between performance, dependability, security, and manageability of the system. There is no universal model of system organization that is appropriate for all circumstances so various distributed architectural styles have emerged. When designing a distributed application, you should choose an architectural style that supports the critical non-functional requirements of your system.

In this section, I discuss five architectural styles:

1. Master-slave architecture, which is used in real-time systems in which guaranteed interaction response times are required.
2. Two-tier client-server architecture, which is used for simple client-server systems, and in situations where it is important to centralize the system for security reasons. In such cases, communication between the client and server is normally encrypted.
3. Multitier client-server architecture, which is used when there is a high volume of transactions to be processed by the server.
4. Distributed component architecture, which is used when resources from different systems and databases need to be combined, or as an implementation model for multi-tier client-server systems.
5. Peer-to-peer architecture, which is used when clients exchange locally stored information and the role of the server is to introduce clients to each other. It may also be used when a large number of independent computations may have to be made.

18.3.1 Master-slave architectures

Master-slave architectures for distributed systems are commonly used in real-time systems where there may be separate processors associated with data acquisition from the system's environment, data processing, and computation and

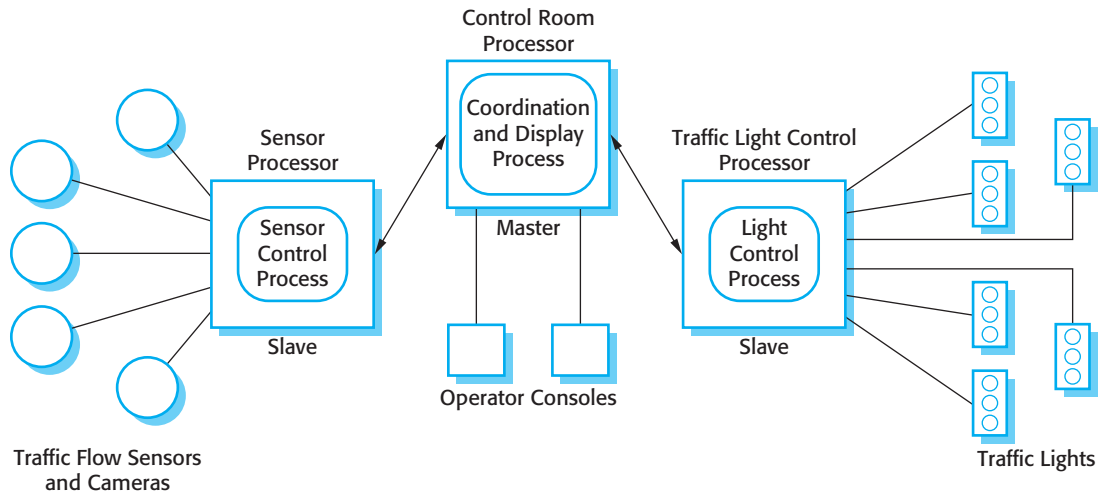


Figure 18.7 A traffic management system with a master-slave architecture

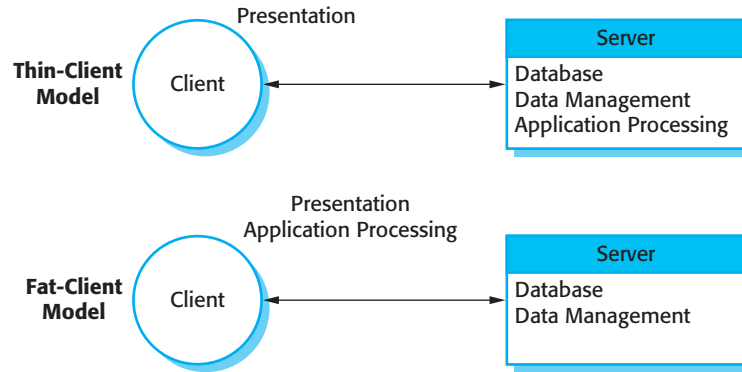
actuator management. Actuators, as I discuss in Chapter 20, are devices controlled by the software system that act to change the system's environment. For example, an actuator may control a valve and change its state from 'open' to 'closed'. The 'master' process is usually responsible for computation, coordination, and communications and it controls the 'slave' processes. 'Slave' processes are dedicated to specific actions, such as the acquisition of data from an array of sensors.

Figure 18.7 illustrates this architectural model. It is a model of a traffic control system in a city and has three logical processes that run on separate processors. The master process is the control room process, which communicates with separate slave processes that are responsible for collecting traffic data and managing the operation of traffic lights.

A set of distributed sensors collects information on the traffic flow. The sensor control process polls the sensors periodically to capture the traffic flow information and collates this information for further processing. The sensor processor is itself polled periodically for information by the master process that is concerned with displaying traffic status to operators, computing traffic light sequences and accepting operator commands to modify these sequences. The control room system sends commands to a traffic light control process that converts these into signals to control the traffic light hardware. The master control room system is itself organized as a client-server system, with the client processes running on the operator's consoles.

You use this master-slave model of a distributed system in situations where you can predict the distributed processing that is required, and where processing can be easily localized to slave processors. This situation is common in real-time systems, where it is important to meet processing deadlines. Slave processors can be used for computationally intensive operations, such as signal processing and the management of equipment controlled by the system.

Figure 18.8 Thin- and fat-client architectural models



18.3.2 Two-tier client–server architectures

In Section 18.2, I discussed the general form of client–server systems in which part of the application system runs on the user’s computer (the client), and part runs on a remote computer (the server). I also presented a layered application model (Figure 18.6) where the different layers in the system may execute on different computers.

A two-tier client–server architecture is the simplest form of client–server architecture. The system is implemented as a single logical server plus an indefinite number of clients that use that server. This is illustrated in Figure 18.8, which shows two forms of this architectural model:

1. A thin-client model, where the presentation layer is implemented on the client and all other layers (data management, application processing, and database) are implemented on a server. The client software may be a specially written program on the client to handle presentation. More often, however, a web browser on the client computer is used for presentation of the data.
2. A fat-client model, where some or all of the application processing is carried out on the client. Data management and database functions are implemented on the server.

The advantage of the thin-client model is that it is simple to manage the clients. This is a major issue if there are a large number of clients, as it may be difficult and expensive to install new software on all of them. If a web browser is used as the client, there is no need to install any software.

The disadvantage of the thin-client approach, however is that it may place a heavy processing load on both the server and the network. The server is responsible for all computation and this may lead to the generation of significant network traffic between the client and the server. Implementing a system using this model may therefore require additional investment in network and server capacity. However, browsers may carry out some local processing by executing scripts (e.g., Javascript) in the web page that is accessed by the browser.

The fat-client model makes use of available processing power on the computer running the client software, and distributes some or all of the application processing

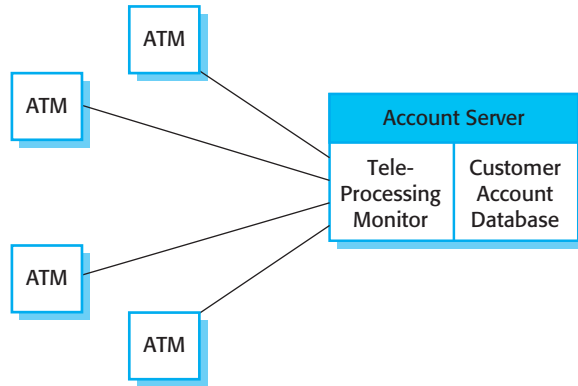


Figure 18.9 A fat-client architecture for an ATM system

and the presentation to the client. The server is essentially a transaction server that manages all database transactions. Data management is straightforward as there is no need to manage the interaction between the client and the application processing system. Of course, the problem with the fat-client model is that it requires additional system management to deploy and maintain the software on the client computer.

An example of a situation in which a fat-client architecture is used is in a bank ATM system, which delivers cash and other banking services to users. The ATM is the client computer and the server is, typically, a mainframe running the customer account database. A mainframe computer is a powerful machine that is designed for transaction processing. It can therefore handle the large volume of transactions generated by ATMs, other teller systems, and online banking. The software in the teller machine carries out a lot of the customer-related processing associated with a transaction.

Figure 18.9 shows a simplified version of the ATM system organization. Notice that the ATMs do not connect directly to the customer database, but rather to a teleprocessing monitor. A teleprocessing (TP) monitor is a middleware system that organizes communications with remote clients and serializes client transactions for processing by the database. This ensures that transactions are independent and do not interfere with one other. Using serial transactions means that the system can recover from faults without corrupting the system data.

Whereas a fat-client model distributes processing more effectively than a thin-client model, system management is more complex. Application functionality is spread across many computers. When the application software has to be changed, this involves reinstallation on every client computer. This can be a major cost if there are hundreds of clients in the system. The system may have to be designed to support remote software upgrades and it may be necessary to shut down all system services until the client software has been replaced.

18.3.3 Multi-tier client–server architectures

The fundamental problem with a two-tier client–server approach is that the logical layers in the system—presentation, application processing, data management, and database—must be mapped onto two computer systems: the client and the server.

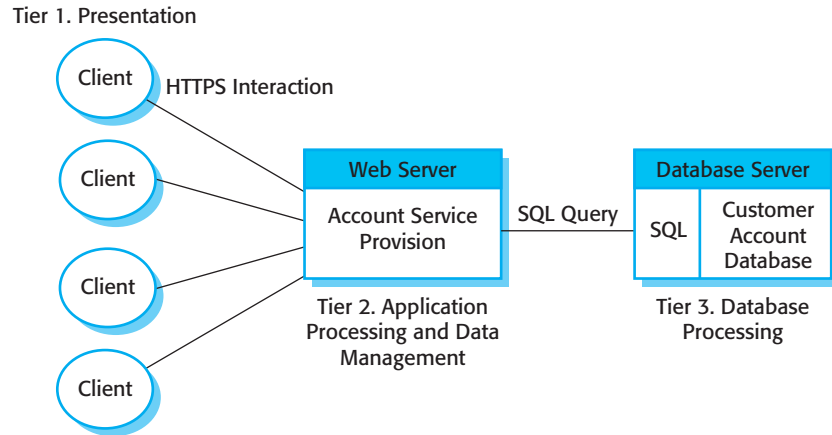


Figure 18.10 Three-tier architecture for an Internet banking system

This may lead to problems with scalability and performance if the thin-client model is chosen, or problems of system management if the fat-client model is used. To avoid some of these problems, a ‘multi-tier client–server’ architecture can be used. In this architecture, the different layers of the system, namely presentation, data management, application processing, and database, are separate processes that may execute on different processors.

An Internet banking system (Figure 18.10) is an example of a multi-tier client–server architecture, where there are three tiers in the system. The bank’s customer database (usually hosted on a mainframe computer as discussed above) provides database services. A web server provides data management services such as web page generation and some application services. Application services such as facilities to transfer cash, generate statements, pay bills, and so on are implemented in the web server and as scripts that are executed by the client. The user’s own computer with an Internet browser is the client. This system is scalable because it is relatively easy to add servers (scale out) as the number of customers increase.

In this case, the use of a three-tier architecture allows the information transfer between the web server and the database server to be optimized. The communications between these systems can use fast, low-level data exchange protocols. Efficient middleware that supports database queries in SQL (Structured Query Language) is used to handle information retrieval from the database.

The three-tier client–server model can be extended to a multi-tier variant, where additional servers are added to the system. This may involve using a web server for data management and separate servers for application processing and database services. Multi-tier systems may also be used when applications need to access and use data from different databases. In this case, you may need to add an integration server to the system. The integration server collects the distributed data and presents it to the application server as if it were from a single database. As I discuss in the following section, distributed component architectures may be used to implement multi-tier client–server systems.

Multi-tier client–server systems that distribute the application processing across several servers are inherently more scalable than two-tier architectures. The

Architecture	Applications
Two-tier client–server architecture with thin clients	<p>Legacy system applications that are used when separating application processing and data management is impractical. Clients may access these as services, as discussed in Section 18.4.</p> <p>Computationally intensive applications such as compilers with little or no data management.</p> <p>Data-intensive applications (browsing and querying) with non-intensive application processing. Browsing the Web is the most common example of a situation where this architecture is used.</p>
Two-tier client-server architecture with fat clients	<p>Applications where application processing is provided by off-the-shelf software (e.g., Microsoft Excel) on the client.</p> <p>Applications where computationally intensive processing of data (e.g., data visualization) is required.</p> <p>Mobile applications where internet connectivity cannot be guaranteed. Some local processing using cached information from the database is therefore possible.</p>
Multi-tier client–server architecture	<p>Large-scale applications with hundreds or thousands of clients.</p> <p>Applications where both the data and the application are volatile.</p> <p>Applications where data from multiple sources are integrated.</p>

Figure 18.11 Use of client–server architectural patterns

application processing is often the most volatile part of the system and it can be easily updated because it is centrally located. Processing, in some cases, may be distributed between the application logic and the data management servers, thus leading to more rapid response to client requests.

Designers of client–server architectures must take a number of factors into account when choosing the most appropriate distribution architecture. Situations in which the client–server architectures discussed here are likely to be appropriate are described in Figure 18.11.

18.3.4 Distributed component architectures

By organizing processing into layers, as shown in Figure 18.6, each layer of a system can be implemented as a separate logical server. This model works well for many types of application. However, it limits the flexibility of system designers in that they have to decide what services should be included in each layer. In practice, however, it is not always clear whether a service is a data management service, an application service, or a database service. Designers must also plan for scalability and so provide some means for servers to be replicated as more clients are added to the system.

A more general approach to distributed system design is to design the system as a set of services, without attempting to allocate these services to layers in the system. Each service, or group of related services, is implemented using a separate component. In a distributed component architecture (Figure 18.12) the system is organized as a set of interacting components or objects. These components provide an interface

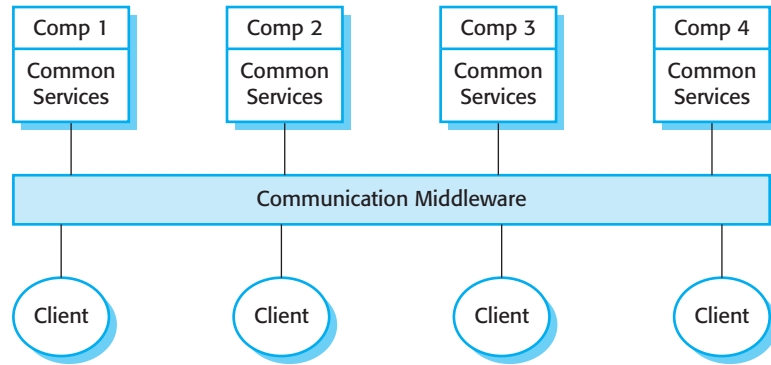


Figure 18.12
A distributed
component
architecture

to a set of services that they provide. Other components call on these services through middleware, using remote procedure or method calls.

Distributed component systems are reliant on middleware, which manages component interactions, reconciles differences between types of the parameters passed between components, and provides a set of common services that application components can use. CORBA (Orfali et al., 1997) was an early example of such middleware but is now not widely used. It has been largely supplanted by proprietary software such as Enterprise Java Beans (EJB) or .NET.

The benefits of using a distributed component model for implementing distributed systems are the following:

1. It allows the system designer to delay decisions on where and how services should be provided. Service-providing components may execute on any node of the network. There is no need to decide in advance whether a service is part of a data management layer, an application layer, etc.
2. It is a very open system architecture that allows new resources to be added as required. New system services can be added easily without major disruption to the existing system.
3. The system is flexible and scalable. New components or replicated components can be added as the load on the system increases, without disrupting other parts of the system.
4. It is possible to reconfigure the system dynamically with components migrating across the network as required. This may be important where there are fluctuating patterns of demand on services. A service-providing component can migrate to the same processor as service-requesting objects, thus improving the performance of the system.

A distributed component architecture can be used as a logical model that allows you to structure and organize the system. In this case, you think about how to provide application functionality solely in terms of services and combinations of services. You then work out how to provide these services using a set of distributed components. For

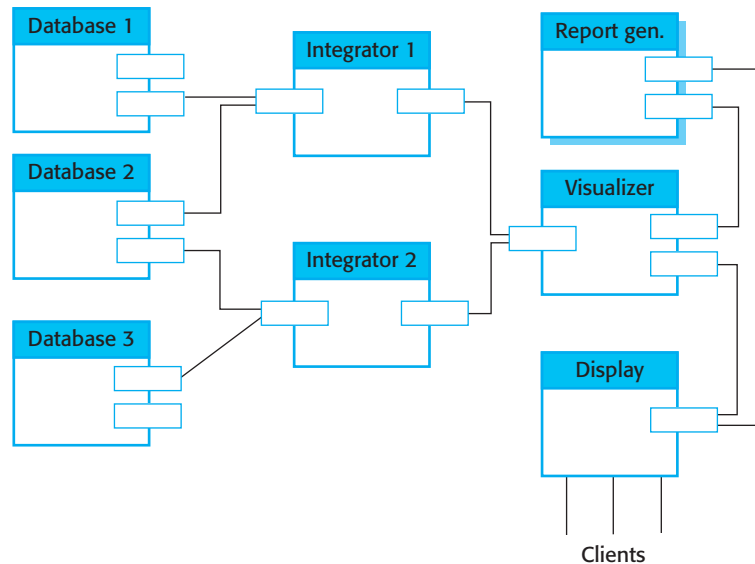


Figure 18.13
A distributed
component
architecture for
a data mining system

example, in a retail application there may be application components concerned with stock control, customer communications, goods ordering, and so on.

Data mining systems are a good example of a type of system in which a distributed component architecture is the best architectural pattern to use. A data mining system looks for relationships between the data that is stored in a number of databases (Figure 18.13). Data mining systems usually pull in information from several separate databases, carry out computationally intensive processing, and display their results graphically.

An example of such a data mining application might be a system for a retail business that sells food and books. The marketing department wants to find relationships between a customer's food and book purchases. For instance, a relatively high proportion of people who buy pizzas might also buy crime novels. With this knowledge, the business can specifically target customers who make specific food purchases with information about new novels when they are published.

In this example, each sales database can be encapsulated as a distributed component with an interface that provides read-only access to its data. Integrator components are each concerned with specific types of relationships, and they collect information from all of the databases to try to deduce the relationships. There might be an integrator component that is concerned with seasonal variations in goods sold, and another that is concerned with relationships between different types of goods.

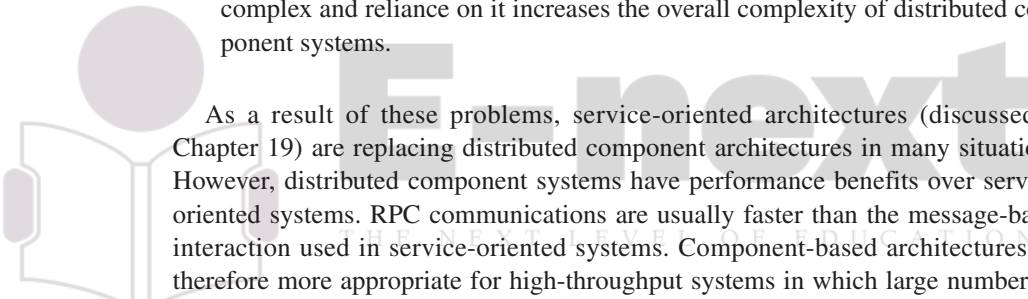
Visualizer components interact with integrator components to produce a visualization or a report on the relationships that have been discovered. Because of the large volumes of data that are handled, visualizer components normally present their results graphically. Finally, a display component may be responsible for delivering the graphical models to clients for final presentation.

A distributed component architecture rather than a layered architecture is appropriate for this type of application because you can add new databases to the system

without major disruption. Each new database is simply accessed by adding another distributed component. The database access components provide a simplified interface that controls access to the data. The databases that are accessed may reside on different machines. The architecture also makes it easy to mine new types of relationship by adding new integrator components.

Distributed component architectures suffer from two major disadvantages:

1. They are more complex to design than client–server systems. Multi-layer client–server systems appear to be a fairly intuitive way to think about systems. They reflect many human transactions where people request and receive services from other people who specialize in providing these services. By contrast, distributed component architectures are more difficult for people to visualize and understand.
2. Standardized middleware for distributed component systems has never been accepted by the community. Rather different vendors, such as Microsoft and Sun, have developed different, incompatible middleware. This middleware is complex and reliance on it increases the overall complexity of distributed component systems.



As a result of these problems, service-oriented architectures (discussed in Chapter 19) are replacing distributed component architectures in many situations. However, distributed component systems have performance benefits over service-oriented systems. RPC communications are usually faster than the message-based interaction used in service-oriented systems. Component-based architectures are therefore more appropriate for high-throughput systems in which large numbers of transactions have to be processed quickly.

18.3.5 Peer-to-peer architectures

The client–server model of computing that I have discussed in previous sections of the chapter makes a clear distinction between servers, which are providers of services and clients, which are receivers of services. This model usually leads to an uneven distribution of load on the system, where servers do more work than clients. This may lead to organizations spending a lot on server capacity while there is unused processing capacity on the hundreds or thousands of PCs that are used to access the system servers.

Peer-to-peer (p2p) systems are decentralized systems in which computations may be carried out by any node on the network. In principle at least, no distinctions are made between clients and servers. In peer-to-peer applications, the overall system is designed to take advantage of the computational power and storage available across a potentially huge network of computers. The standards and protocols that enable communications across the nodes are embedded in the application itself and each node must run a copy of that application.

Peer-to-peer technologies have mostly been used for personal rather than business systems (Oram, 2001). For example, file-sharing systems based on the Gnutella and BitTorrent protocols are used to exchange files on users' PCs. Instant messaging systems such as ICQ and Jabber provide direct communications between users without an intermediate server. SETI@home is a long-running project to process data from radio telescopes on home PCs to search for indications of extraterrestrial life. Freenet is a decentralized database that has been designed to make it easier to publish information anonymously, and to make it difficult for authorities to suppress this information. Voice over IP (VOIP) phone services, such as Skype, rely on peer-to-peer communication between the parties involved in the phone call or conference.

However, peer-to-peer systems are also being used by businesses to harness the power in their PC networks (McDougall, 2000). Intel and Boeing have both implemented p2p systems for computationally intensive applications. This takes advantage of unused processing capacity on local computers. Instead of buying expensive high-performance hardware, engineering computations can be run overnight when desktop computers are unused. Businesses also make extensive use of commercial p2p systems, such as messaging and VOIP systems.

It is appropriate to use a peer-to-peer architectural model for a system in two circumstances:

1. Where the system is computationally intensive and it is possible to separate the processing required into a large number of independent computations. For example, a peer-to-peer system that supports computational drug discovery distributes computations that look for potential cancer treatments by analyzing a huge number of molecules to see if they have the characteristics required to suppress the growth of cancers. Each molecule can be considered separately so there is no need for the peers in the system to communicate.
2. Where the system primarily involves the exchange of information between individual computers on a network and there is no need for this information to be centrally stored or managed. Examples of such applications include file-sharing systems that allow peers to exchange local files such as music and video files, and phone systems that support voice and video communications between computers.

In principle, every node in a p2p network could be aware of every other node. Nodes could connect to and exchange data directly with any other node in the network. In practice, of course, this is impossible, so nodes are organized into 'localities' with some nodes acting as bridges to other node localities. Figure 18.14 shows this decentralized p2p architecture.

In a decentralized architecture, the nodes in the network are not simply functional elements but are also communications switches that can route data and control signals from one node to another. For example, assume that Figure 18.14 represents a decentralized, document-management system. This system is used by a consortium of researchers to share documents, and each member of the consortium maintains his or her own document store. However, when a document is retrieved, the node retrieving that document also makes it available to other nodes.

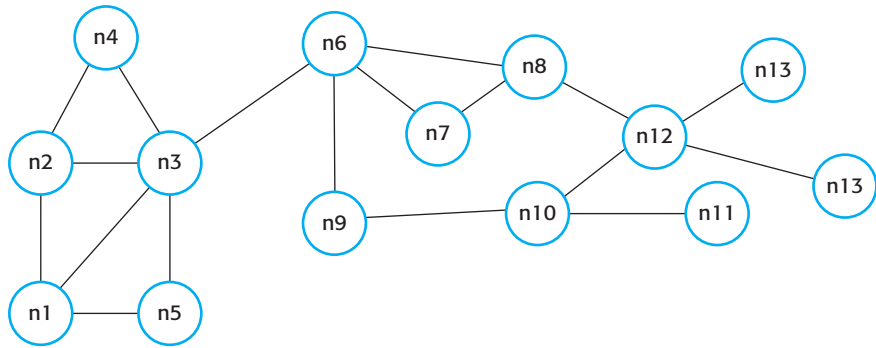


Figure 18.14
A decentralized
p2p architecture

If someone needs a document that is stored somewhere on the network, they issue a search command, which is sent to nodes in their ‘locality’. These nodes check whether they have the document and, if so, return it to the requestor. If they do not have it, they route the search to other nodes. Therefore, if n1 issues a search for a document that is stored at n10, this search is routed through nodes n3, n6, and n9 to n10. When the document is finally discovered, the node holding the document then sends it to the requesting node directly by making a peer-to-peer connection.

This decentralized architecture has advantages in that it is highly redundant and hence both fault-tolerant and tolerant of nodes disconnecting from the network. However, the disadvantages here are that many different nodes may process the same search, and there is also significant overhead in replicated peer communications.

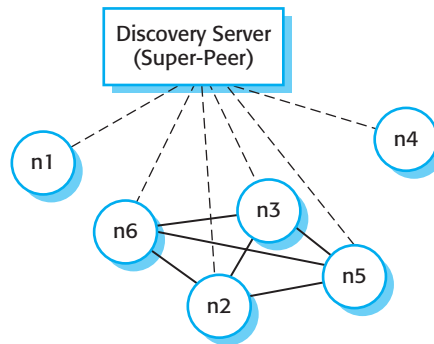
An alternative p2p architectural model, which departs from a pure p2p architecture, is a semicentralized architecture where, within the network, one or more nodes act as servers to facilitate node communications. This reduces the amount of traffic between nodes. Figure 18.15 illustrates this model.

In a semicentralized architecture, the role of the server (sometimes called a super-peer) is to help establish contact between peers in the network, or to coordinate the results of a computation. For example, if Figure 18.15 represents an instant messaging system, then network nodes communicate with the server (indicated by dashed lines) to find out what other nodes are available. Once these nodes are discovered, direct communications can be established and the connection to the server is unnecessary. Therefore nodes n2, n3, n5, and n6 are in direct communication.

In a computational p2p system, where a processor-intensive computation is distributed across a large number of nodes, it is normal for some nodes to be super-peers. Their role is to distribute work to other nodes and to collate and check the results of the computation.

Peer-to-peer architectures allow for the efficient use of capacity across a network. However, the major concerns that have inhibited their use are issues of security and trust. Peer-to-peer communications involve opening your computer to direct interactions with other peers and this means that these systems could, potentially, access any of your resources. To counter this, you need to organize your system so that these resources are protected. If this is done incorrectly, then your system may be insecure.

Figure 18.15
A semicentralized
p2p architecture



Problems may also occur when peers on a network deliberately behave in a malicious way. For example, there have been cases where music companies who believe that their copyright is being abused have deliberately made ‘poisoned peers’ available. When another peer downloads what they think is a piece of music, the actual file delivered is malware that may be a deliberately corrupted version of the music or a warning to the user of copyright infringement.

18.4 Software as a service

In the previous sections, I discussed client–server models and how functionality may be distributed between the client and the server. To implement a client–server system, you may have to install a program on the client computer, which communicates with the server, implements client-side functionality and manages the user interface. For example, a mail client, such as Outlook or Mac Mail, provides mail management features on your own computer. This avoids the problem of some thin-client systems where all of the processing is carried out at the server.

However, the problems of server overload can be significantly reduced by using a modern browser as the client software. Web technologies, such as AJAX (Holdener, 2008), support efficient management of web page presentation and local computation through scripts. This means that a browser can be configured and used as a client, with significant local processing. The application software can be thought of as a remote service, which can be accessed from any device that can run a standard browser. Well-known examples of this are web-based mail systems, such as Yahoo! and Gmail and office applications, such as Google docs.

This notion of SaaS involves hosting the software remotely and providing access to it over the Internet. The key elements of SaaS are the following:

1. Software is deployed on a server (or more commonly a number of servers) and is accessed through a web browser. It is not deployed on a local PC.
2. The software is owned and managed by a software provider, rather than the organizations using the software.

3. Users may pay for the software according to the amount of use they make of it or through an annual or monthly subscription. Sometimes, the software is free for anyone to use but users must then agree to accept advertisements, which fund the software service.

For software users, the benefit of SaaS is that the costs of management of software are transferred to the provider. The provider is responsible for fixing bugs and installing software upgrades, dealing with changes to the operating system platform, and ensuring that hardware capacity can meet demand. Software licence management costs are zero. If someone has several computers, there is no need to licence software for all of these. If a software application is only used occasionally, the pay-per-use model may be cheaper than buying an application. The software may be accessed from mobile devices, such as smart phones, from anywhere in the world.

Of course, this model of software provision has some disadvantages. The main problem, perhaps, is the costs of data transfer to the remote service. Data transfer takes place at network speeds and so transferring a large amount of data takes a lot of time. You may also have to pay the service provider according to the amount transferred. Other problems are lack of control over software evolution (the provider may change the software when they wish) and problems with laws and regulations. Many countries have laws governing the storage, management, preservation, and accessibility of data and moving data to a remote service may breach these laws.

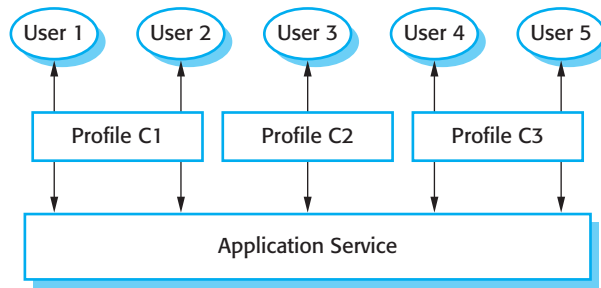
The notion of SaaS and service-oriented architectures (SOAs), discussed in Chapter 19, are obviously related but they are not the same:

1. SaaS is a way of providing functionality on a remote server with client access through a web browser. The server maintains the user's data and state during an interaction session. Transactions are usually long transactions (e.g., editing a document).
2. SOA is an approach to structuring a software system as a set of separate, stateless services. These may be provided by multiple providers and may be distributed. Typically, transactions are short transactions where a service is called, does something, and then returns a result.

SaaS is a way of delivering application functionality to users, whereas SOA is an implementation technology for application systems. The functionality implemented using SOA need not appear to users as services. Similarly, user services do not have to be implemented using SOA. However, if SaaS is implemented using SOA, it becomes possible for applications to use service APIs to access the functionality of other applications. They can then be integrated into more complex systems. These are called mashups and represent another approach to software reuse and rapid software development.

From a software development perspective, the process of service development has much in common with other types of software development. However, service construction is not usually driven by user requirements, but by the service provider's assumptions about what users need. The software therefore needs to be able to

Figure 18.16
Configuration of a
software system
offered as a service



evolve quickly after the provider gets feedback from users on their requirements. Agile development with incremental delivery is therefore a commonly used approach for software that is to be deployed as a service.

When you are implementing SaaS you have to take into account that you may have users of the software from several different organizations. You have to take three factors into account:

1. *Configurability* How do you configure the software for the specific requirements of each organization?
2. *Multi-tenancy* How do you present each user of the software with the impression that they are working with their own copy of the system while, at the same time, making efficient use of system resources?
3. *Scalability* How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?

The notion of product-line architectures, discussed in Chapter 16, is one way of configuring software for users who have overlapping but not identical requirements. You start with a generic system and adapt this according to the specific requirements of each user.

However, this does not work for SaaS as it would mean deploying a different copy of the service for each organization that uses the software. Rather, you need to design configurability into the system and provide a configuration interface that allows users to specify their preferences. You then use these to adjust the behavior of the software dynamically as it is used. Configuration facilities may allow for the following:

1. Branding, where users from each organization, are presented with an interface that reflects their own organization.
2. Business rules and workflows, where each organization defines its own rules that govern the use of the service and its data.
3. Database extensions, where each organization defines how the generic service data model is extended to meet its specific needs.
4. Access control, where service customers create individual accounts for their staff and define the resources and functions that are accessible to each of their users.

Figure 18.17
A multi-tenant
database

Tenant	Key	Name	Address
234	C100	XYZ Corp	43, Anystreet, Sometown
234	C110	BigCorp	2, Main St, Motown
435	X234	J. Bowie	56, Mill St, Starville
592	PP37	R. Burns	Alloway, Ayrshire

Figure 18.16 illustrates this situation. This diagram shows five users of the application service, who work for three different customers of the service provider. Users interact with the service through a customer profile that defines the service configuration for their employer.

Multi-tenancy is a situation in which many different users access the same system and the system architecture is defined to allow the efficient sharing of system resources. However, it must appear to each user that they have the sole use of the system. Multi-tenancy involves designing the system so that there is an absolute separation between the system functionality and the system data. You should, therefore, design the system so that all operations are stateless. Data should either be provided by the client or should be available in a storage system or database that can be accessed from any system instance. Relational databases are not ideal for providing multi-tenancy and large service providers, such as Google, have implemented a simpler database for user data.

A particular problem in multi-tenant systems is data management. The simplest way to provide data management is for each customer to have their own database, which they may use and configure as they wish. However, this requires the service provider to maintain many different database instances (one per customer) and to make these available on demand. This is inefficient in terms of server capacity and increases the overall cost of the service.

As an alternative, the service provider can use a single database with different users being virtually isolated within that database. This is illustrated in Figure 18.17, where you can see that database entries also have a ‘tenant identifier’, which links these entries to specific users. By using database views, you can extract the entries for each service customer and so present users from that customer with a virtual, personal database. This can be extended to meet specific customer needs using the configuration features discussed above.

Scalability is the ability of the system to cope with increasing numbers of users without reducing the overall QoS that is delivered to any user. Generally, when considering scalability in the context of SaaS, you are considering ‘scaling out’, rather than ‘scaling up’. Recall that ‘scaling out’ means adding additional servers and so also increasing the number of transactions that can be processed in parallel. Scalability is a complex topic that I cannot cover in detail here, but some general guidelines for implementing scalable software are:

1. Develop applications where each component is implemented as a simple stateless service that may be run on any server. In the course of a single transaction,

a user may therefore interact with instances of the same service that are running on several different servers.

2. Design the system using asynchronous interaction so that the application does not have to wait for the result of an interaction (such as a read request). This allows the application to carry on doing useful work while it is waiting for the interaction to finish.
3. Manage resources, such as network and database connections, as a pool so that no single server is likely to run out of resources.
4. Design your database to allow fine-grain locking. That is, do not lock out whole records in the database when only part of a record is in use.

The notion of SaaS is a major paradigm shift for distributed computing. Rather than an organization hosting multiple applications on their servers, SaaS allows these applications to be externally provided by different vendors. We are in the midst of a transition from one model to another and, in the future, this is likely to have a very significant effect on the engineering of enterprise software systems.

KEY POINTS

- The benefits of distributed systems are that they can be scaled to cope with increasing demand, can continue to provide user services (even if some parts of the system fail), and they enable resources to be shared.
- Issues to be considered in the design of distributed systems include transparency, openness, scalability, security, quality of service, and failure management.
- Client–server systems are distributed systems in which the system is structured into layers, with the presentation layer implemented on a client computer. Servers provide data management, application, and database services.
- Client–server systems may have several tiers, with different layers of the system distributed to different computers.
- Architectural patterns for distributed systems include master-slave architectures, two-tier and multi-tier client–server architectures, distributed component architectures, and peer-to-peer architectures.
- Distributed component systems require middleware to handle component communications and to allow components to be added to and removed from the system.
- Peer-to-peer architectures are decentralized architectures in which there are no distinguished clients and servers. Computations can be distributed over many systems in different organizations.
- Software as a service is a way of deploying applications as thin client–server systems, where the client is a web browser.

FURTHER READING

‘Middleware: A model for distributed systems services’. Although a little dated in places, this is an excellent overview paper that summarizes the role of middleware in distributed systems and discusses the range of middleware services that may be provided. (P. A. Bernstein, *Comm. ACM*, **39** (2), February 1996.) <http://dx.doi.org/10.1145/230798.230809>.

Peer-to-Peer: Harnessing the Power of Disruptive Technologies. Although this book does not have a lot of information on p2p architectures, it is an excellent introduction to p2p computing and discusses the organization and approach used in a number of p2p systems. (A. Oram (ed.), O’Reilly and Associates Inc., 2001.)

‘Turning software into a service’. A good overview paper that discusses the principles of service-oriented computing. Unlike many papers on this topic, it does not conceal these principles behind a discussion of the standards involved. (M. Turner, D. Budgen and P. Brereton, *IEEE Computer*, **36** (10), October 2003.) <http://dx.doi.org/10.1109/MC.2003.1236470>.

Distributed Systems: Principles and Paradigms, 2nd edition. A comprehensive textbook that discusses all aspects of distributed systems design and implementation. However, it does not include much discussion of the service-oriented paradigm. (A.S. Tanenbaum and M. Van Steen, Addison-Wesley, 2007.)

‘Software as a Service; The Spark that will Change Software Engineering.’ A short paper that argues that the advent of SaaS will push all software development to an iterative model. (G. Goth, *Distributed Systems Online*, **9** (7), July 2008.) <http://dx.doi.org/10.1109/MDSO.2008.21>.

EXERCISES

- 18.1.** What do you understand by ‘scalability’? Discuss the differences between ‘scaling up’ and ‘scaling out’ and explain when these different approaches to scalability may be used.
- 18.2.** Explain why distributed software systems are more complex than centralized software systems, where all of the system functionality is implemented on a single computer.
- 18.3.** Using an example of a remote procedure call, explain how middleware coordinates the interaction of computers in a distributed system.
- 18.4.** What is the fundamental difference between a fat-client and a thin-client approach to client–server systems architectures?
- 18.5.** You have been asked to design a secure system that requires strong authentication and authorization. The system must be designed so that communications between parts of the system cannot be intercepted and read by an attacker. Suggest the most appropriate client–server architecture for this system and, giving reasons for your answer, propose how functionality should be distributed between the client and the server systems.
- 18.6.** Your customer wants to develop a system for stock information where dealers can access information about companies and evaluate various investment scenarios using a simulation

system. Each dealer uses this simulation in a different way, according to his or her experience and the type of stocks in question. Suggest a client–server architecture for this system that shows where functionality is located. Justify the client–server system model that you have chosen.

- 18.7. Using a distributed component approach, propose an architecture for a national theater booking system. Users can check seat availability and book seats at a group of theaters. The system should support ticket returns so that people may return their tickets for last-minute resale to other customers.
- 18.8. Give two advantages and two disadvantages of decentralized and semicentralized peer-to-peer architectures.
- 18.9. Explain why deploying software as a service can reduce the IT support costs for a company. What additional costs might arise if this deployment model is used?
- 18.10. Your company wishes to move from using desktop applications to accessing the same functionality remotely as services. Identify three risks that might arise and suggest how these risks may be reduced.

REFERENCES

- Bernstein, P. A. (1996). 'Middleware: A Model for Distributed System Services'. *Comm. ACM*, **39** (2), 86–97.
- Coulouris, G., Dollimore, J. and Kindberg, T. (2005). *Distributed Systems: Concepts and Design*, 4th edition. Harlow, UK.: Addison-Wesley.
- Holdener, A. T. (2008). *Ajax: The Definitive Guide*. Sebastopol, Calif.: O'Reilly and Associates.
- McDougall, P. (2000). 'The Power of Peer-To-Peer'. *Information Week* (August 28th, 2000).
- Neuman, B. C. (1994). 'Scale in Distributed Systems'. In *Readings in Distributed Computing Systems*. Casavant, T. and Singal, M. (ed.). Los Alamitos, Calif.: IEEE Computer Society Press.
- Oram, A. (2001). 'Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology'.
- Orfali, R. and Harkey, D. (1998). *Client/server Programming with Java and CORBA*. New York: John Wiley & Sons.
- Orfali, R., Harkey, D. and Edwards, J. (1997). *Instant CORBA*. Chichester, UK: John Wiley & Sons.
- Pope, A. (1997). *The CORBA Reference Guide: Understanding the Common Request Broker Architecture*. Boston: Addison-Wesley.
- Tanenbaum, A. S. and Van Steen, M. (2007). *Distributed Systems: Principles and Paradigms*, 2nd edition. Upper Saddle River, NJ: Prentice Hall.