11 Distributed systems architectures

Objectives

The objective of this chapter is to describe, in more detail, the architectures of distributed software systems. When you have read this chapter, you will:

- understand the main advantages and disadvantages of distributed systems architectures;
- understand different approaches to the development of client–server architectures;
- understand the differences between client–server and distributed object architectures;
- understand the concept of an object request broker and the principles underlying the CORBA standards.

Contents

Virtually all large computer-based systems are now distributed systems. A distributed system is a system where the information processing is distributed over several computers rather than confined to a single machine. Obviously, the engineering of distributed systems has a great deal in common with the engineering of any other software but there are specific issues that have to be taken into account when designing this type of system. I have already introduced some of these issues in the introduction to client–server architectures in Chapter 10 and I cover them in more detail here.

Software engineers must be aware of these design issues because distributed systems are so widely used. Until relatively recently, most large systems were centralised systems that ran on a single mainframe with terminals attached to it. These terminals had little or no processing capability, so all information processing was the responsibility of the mainframe computer. Designers did not have to think about issues of distributed processing. Now, there are three major types of system:

1. Personal systems that are not distributed and that are designed to run on a personal computer or workstation. Examples of this type of system include word processors, spreadsheets, graphics systems, etc.
2. Embedded systems that run on a single processor or on an integrated group of processors. Examples of this type of system include control systems for domestic devices, instrument management systems, etc.
3. Distributed systems where the system software runs on a loosely integrated group of cooperating processors linked by a network. Examples of such systems include bank ATM systems, booking systems, groupware systems, etc.

Currently, there are fairly distinct boundaries between these types of system but these boundaries are likely to blur in future. As high-speed wireless networking becomes widely available, it will be possible to dynamically integrate products with embedded systems such as electronic organisers with more general systems.

Coulouris *et al.* (1994) identify six important characteristics of distributed systems:

1. *Resource sharing* A distributed system allows the sharing of hardware and software resources such as disks, printers, files, compilers, etc. that are associated with different computers on a network. Obviously, resource sharing is also possible on multi-user systems but in this case all resources must be provided and managed by a central computer.
2. *Openness* The openness of a system is the extent to which it can be extended by adding new non-proprietary resources to it. Distributed systems are open systems that normally include hardware and software from different vendors.
3. *Concurrency* In a distributed system, several processes may operate at the same time on different 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, scalability may be limited by the network linking the individual computers in the system. If many new computers are added then the network capacity may be inadequate.
5. *Fault tolerance* The availability of several computers and the potential for replicating information mean that distributed systems can be tolerant of some hardware and software failures (see Chapter 18). In most distributed systems, a degraded service can be provided when failures occur; complete loss of service tends to occur only when there is a network failure.
6. *Transparency* Transparency is the concealment from the user of the distributed nature of the system. A system design goal may be that users have completely transparent access to resources and have no need to know anything about the distribution of the system. However, in many cases giving the users some knowledge of the system organisation allows them to make better use of the resources.

Of course, distributed systems do have some disadvantages:

- *Complexity* Distributed systems are more complex than centralised systems. This makes it more difficult to understand their emergent properties and to test these systems. For example, rather than the performance of the system being dependent on the execution speed of one processor, it depends on the network bandwidth and the speed of the different processors on the network. Moving resources from one part of the system to another can radically affect the system's performance.
- *Security* The system may be accessed from several different computers and the traffic on the network may be subject to eavesdropping. This makes it more difficult to manage the security in a distributed system.
- *Manageability* The different computers in a system may be of different types and may run different versions of the operating system. Faults in one machine may propagate to other machines with unexpected consequences. These mean that more effort is required to manage and maintain the system in operation.
- *Unpredictability* As all users of the WWW know, distributed systems are unpredictable in their response. The response depends on the overall load on the system, its organisation 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.

As well as discussing the advantages and disadvantages of distributed systems, Coulouris *et al.* (1994) identify the critical design issues for distributed systems. These are shown in Figure 11.1. I focus in this chapter on distributed software

Figure 11.1 Issues in distributed systems design

Design issue	Description
Resource identification	The resources in a distributed system are spread across different computers and a naming scheme has to be devised so that users can discover and refer to the resources that they need. An example of such a naming scheme is the URL (Uniform Resource Locator) that is used to identify WWW pages. If a meaningful and universally understood identification scheme is not used then many of these resources will be inaccessible to system users.
Communications	The universal availability of the Internet and the efficient implementation of Internet TCP/IP communication protocols means that, for most distributed systems, these are the most effective way for the computers to communicate. However, where there are specific requirements for performance, reliability, etc. alternative approaches to communications may be used.
Quality of service	The quality of service offered by a system reflects its performance, availability and reliability. It is affected by a number of factors such as the allocation of processes to processes in the system, the distribution of resources across the system, the network and the system hardware and the adaptability of the system.
Software architectures	The software architecture describes how the application functionality is distributed over a number of logical components and how these components are distributed across processors. Choosing the right architecture for an application is essential to achieve the desired quality of service.

architectures as I think that this is most relevant to software engineering. Specialised books on distributed systems cover these other topics.

The challenge for distributed systems designers is to design the software and hardware to provide desirable distributed system characteristics and, at the same time, minimise the problems that are inherent in these systems. To do so, you need to understand the advantages and disadvantages of different distributed systems architectures. I cover two generic types of distributed systems architecture here:

1. *Client–server architectures* In this approach, the system may be thought of as a set of services that are provided to clients that make use of these services. Servers and clients are treated differently in these systems.
2. *Distributed object architectures* In this case, there is no distinction between servers and clients and the system may be thought of as a set of interacting objects whose location is irrelevant. There is no distinction between a service provider and a user of these services.

The different 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 some software that can manage these diverse parts, 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 different distributed components of the system.

Bernstein (1996) summarises different 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 are software for managing communications with databases, transaction managers, data converters, communication controllers, etc. I describe distributed systems frameworks, a very important class of middleware, later in this chapter.

Distributed systems are usually developed using an object-oriented approach. These systems are made up of loosely integrated, independent parts each of which may interact directly with users or with other parts of the system. Parts of the system may have to respond to independent events. Software objects reflect these characteristics so are a natural abstraction for distributed systems components. If you are not familiar with the concept of objects, I recommend that you read Chapter 12 and then come back to this chapter.

11.1 Multiprocessor architectures

The simplest model of a distributed system is a multiprocessor system where the system consists of a number of different processes that may (but need not) execute on separate processors. This model is common in large real-time systems. As I discuss in Chapter 13, these systems collect information, make decisions using this information, then send signals to actuators that modify the system's environment.

Logically, the processes concerned with information collection, decision making and actuator control could all run on a single processor under the control of a scheduler. Using multiple processors improves the performance and resilience of the system. The distribution of processes to processors may be pre-determined (this is common in critical systems) or may be under the control of a despatcher that decides which process to allocate to each processor.

An example of this type of system is shown in Figure 11.2. This is a simplified model of a traffic control system. A set of distributed sensors collect information on the traffic flow and process this locally before sending it to a control room. Operators make decisions using this information and give instructions to a separate traffic light control process. In this example, there are separate logical processes for managing the sensors, the control room and the traffic lights. These logical processes may be single processes or a group of processes. In this example, they run on separate processors.

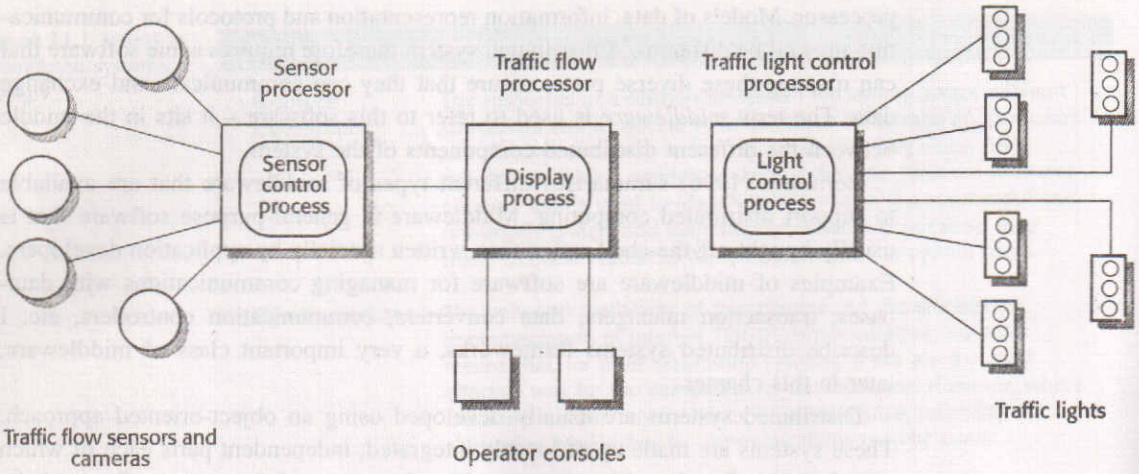


Figure 11.2 A multiprocessor traffic control system

Software systems composed of multiple processes are not necessarily distributed systems. If more than one processor is available then distribution can be implemented but the system designers need not always consider distribution issues during the design process. The design approach for this type of system is essentially that for real-time systems as discussed in Chapter 13.

11.2 Client–server architectures

I have already introduced the concept of client–server architectures in Chapter 10. In a client–server architecture, an application is modelled as a set of services that are provided by servers and a set of clients that use these services (Orfali and

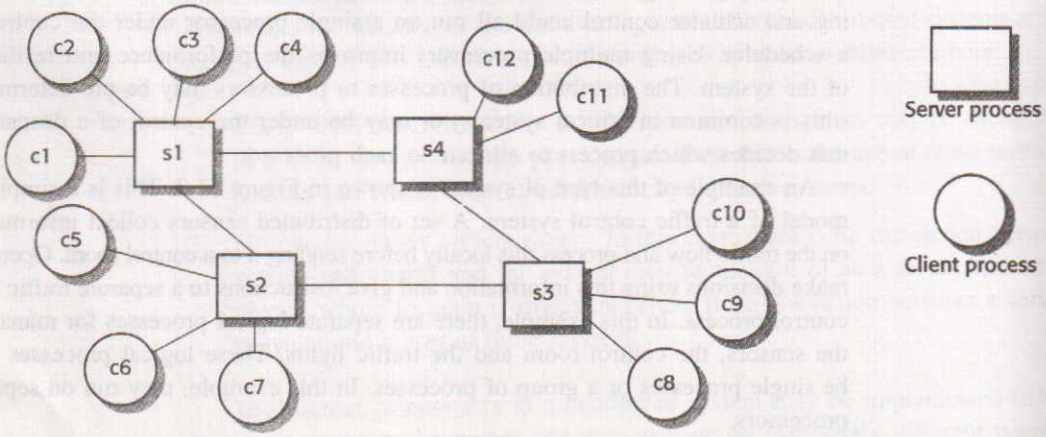


Figure 11.3 A client–server system

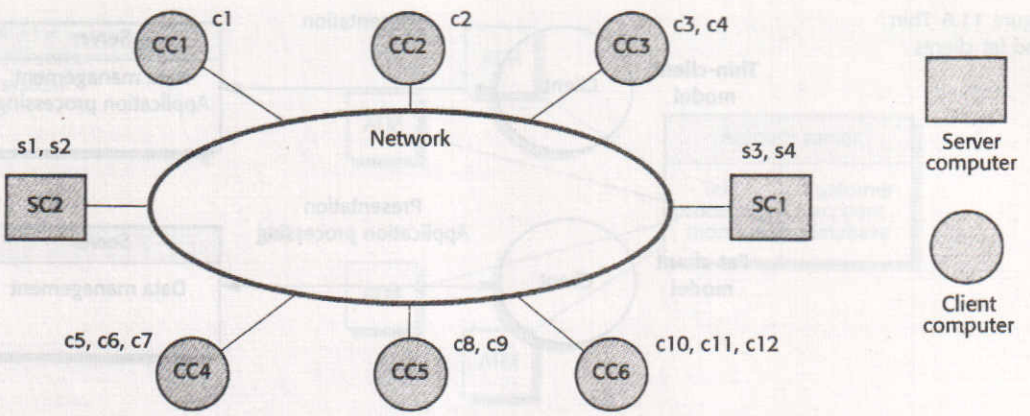


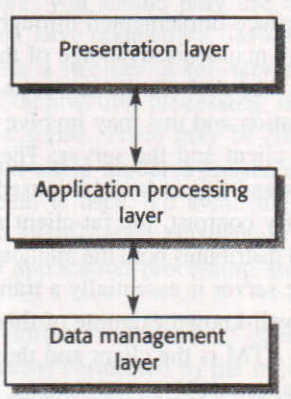
Figure 11.4
Computers in a client–server network

Harkey, 1998). Clients need to be aware of the servers that are available but usually do not know of the existence of other clients. Clients and servers are different processes, as shown in Figure 11.3 which is a logical model of a distributed client–server architecture.

There is not necessarily a 1:1 mapping between processes and processors in the system. Figure 11.4 shows the physical architecture of a system with six client computers and two server computers. These can run the client and server processes shown in Figure 11.3. In general, when I talk of clients and servers I mean these logical processes rather than the physical computers on which these processes execute.

The design of client–server systems should reflect the logical structure of the application that is being developed. One way to look at an application is illustrated in Figure 11.5 which shows an application structured into three layers. The presentation layer is concerned with presenting information to the user and with all user interaction. The application processing layer is concerned with implementing the logic of the application and the data management layer is concerned with all database operations. In centralised systems these need not be clearly separated. However, when you are designing a distributed system, you should make a clear

Figure 11.5
Application layers



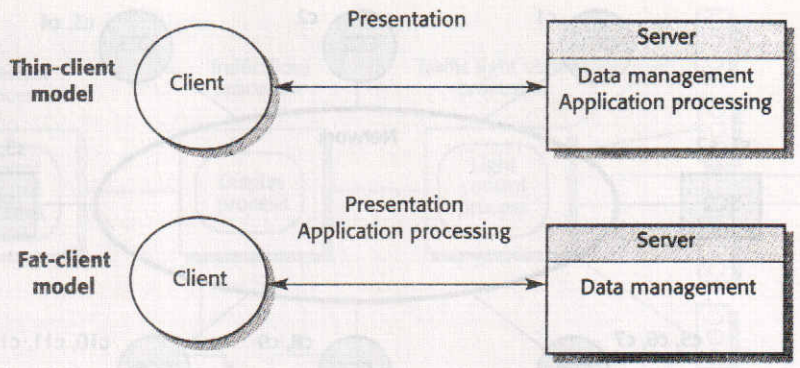


Figure 11.6 Thin and fat clients

distinction between them as it then becomes possible to distribute each layer to a different computer.

The simplest client–server architecture is called a two-tier client–server architecture where an application is organised as a server (or multiple identical servers) and a set of clients. As illustrated in Figure 11.6, two-tier client–server architectures can take two forms:

1. *Thin-client model* In a thin-client model, all of the application processing and data management is carried out on the server. The client is simply responsible for running the presentation software.
2. *Fat-client model* In this model, the server is only responsible for data management. The software on the client implements the application logic and the interactions with the system user.

A thin-client, two-tier architecture is the simplest approach to use when centralised legacy systems (see Chapter 26) are evolved to a client–server architecture. The user interface for these systems is migrated to PCs and the application itself acts as a server and handles all application processing and data management. A thin-client model may also be implemented when the clients are simple network devices rather than PCs or workstations. The network device runs an Internet browser and the user interface implemented through that system.

A major disadvantage of the thin-client model is that it places a heavy processing load on both the server and the network. The server is responsible for all computation and this may involve the generation of significant network traffic between the client and the server. There is a lot of processing power available in modern PCs and this is largely unused in the thin-client approach.

By contrast, the fat-client model makes use of this available processing power and distributes both the application logic processing and the presentation to the client. The server is essentially a transaction server that manages all database transactions. A well-known example of this type of architecture is banking ATM systems where the ATM is the client and the server is a mainframe running the customer account database.

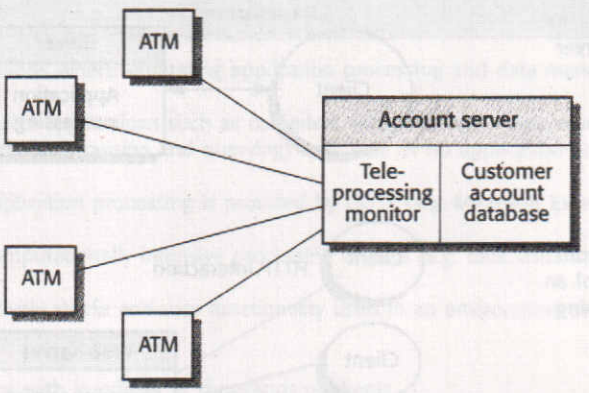


Figure 11.7
A client–server ATM system

This ATM network system is illustrated in Figure 11.7. Notice that the ATMs do not connect directly to the customer database but connect to a teleprocessing monitor. A teleprocessing (TP) monitor is a middleware system that organises communications with remote clients and serialises client transactions for processing by the database. Using serial transactions means that the system can recover from faults without corrupting the system data.

While the fat-client model distributes processing more effectively than a thin-client model, system management is more complex. Application functionality is spread across many different 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 advent of Java and downloadable applets has allowed the development of a client–server model that is somewhere between the thin-client and the fat-client model. Some of the application processing software may be downloaded to the client as Java applets, thus relieving the load on the server. The user interface is built using a web browser that can run Java applets. However, web browsers from different suppliers and even different versions of web browsers from the same supplier do not always display in the same way. Earlier versions on older computers may not be able to run Java. Therefore, you should only use this approach when you are confident that all system users have compatible, Java-enabled browsers.

The essential problem with a two-tier client–server approach is that the three logical layers – presentation, application processing, data management – must be mapped onto two computer systems. There may be either 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 these problems, an alternative approach is to use a three-tier client–server architecture (Figure 11.8). In this architecture, the presentation, the application processing and the data management are logically separate processes.

A three-tier client–server software architecture does not necessarily mean that there are three computer systems connected to the network. A single server computer can run both the application processing and the application data management

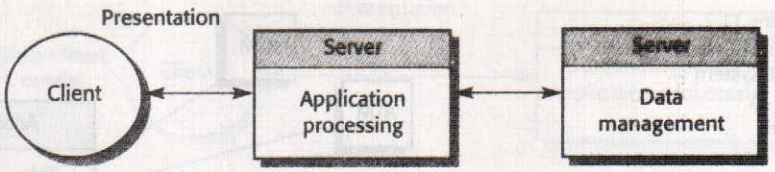


Figure 11.8 A three-tier client–server architecture

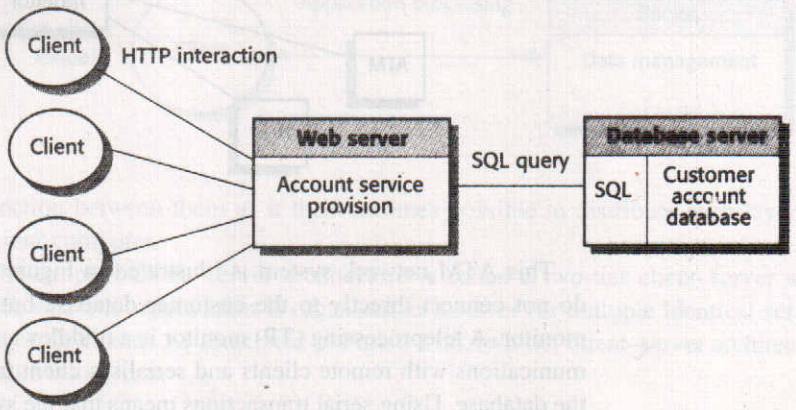


Figure 11.9 The distribution architecture of an Internet banking system

as separate, logical servers. However, if demands rise, it is relatively straightforward to separate the application processing and the data management and execute these on separate processors.

An Internet banking system is an example of the type of system that may be implemented using a three-tier client–server architecture. The bank's customer database (usually hosted on a mainframe computer) provides data management services, a web server provides the application services such as facilities to transfer cash, generate statements, pay bills, etc. and the user's own computer with an Internet browser is the client. This is illustrated in Figure 11.9. This system is scalable as it is relatively easy to add new web servers as the number of customers increase.

The use of a three-tier architecture in this case allows the information transfer between the web server and the database server to be optimised. The communications between these systems do not have to be based on the Internet standards but can use faster, lower-level communications protocols. Efficient middleware that supports database queries in SQL (Structured Query Language) is used to handle information retrieval from the database.

In some cases, it is appropriate to extend the three-tier client–server model to a multi-tier variant where additional servers are added to the system. Multi-tier systems may be used where applications need to access and use data from different databases. In this case, an integration server is positioned between the application server and the database servers that are accessed. The integration server collects the distributed data and presents it to the application as if it was available in a single database.

Three-tier client–server architectures and multi-tier variants that distribute the application processing across several servers are inherently more scalable than two-tier architectures. Network traffic is reduced in contrast with thin-client two-tier architectures.

Architecture	Applications
Two-tier C/S architecture with thin clients	Legacy system applications where separating application processing and data management is impractical Computationally intensive applications such as compilers with little or no data management Data-intensive applications (browsing and querying) with little or no application processing
Two-tier C/S architecture with fat clients	Applications where application processing is provided by COTS (e.g. Microsoft Excel) on the client Applications where computationally intensive processing of data (e.g. data visualisation) is required Applications with relatively stable end-user functionality used in an environment with well-established system management
Three-tier or multi-tier C/S architecture	Large-scale applications with hundreds or thousands of clients Applications where both the data and the application are volatile Applications where data from multiple sources are integrated

Figure 11.10 Use of different client–server architectures

The application processing is 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 architecture. Situations where the client–server architectures that I have discussed are likely to be appropriate are shown in Figure 11.10.

11.3 Distributed object architectures

In the client–server model of a distributed system, clients and servers are different. Clients receive services from servers and not from other clients; servers may act as clients by receiving services from other servers but they do not request services from clients; clients must know the services that are offered by specific servers and must know how to contact these servers. This model works well for many types of application. However, it limits the flexibility of system designers in that they must decide where services are to be provided. They must also plan for scalability and so provide some means for the load on servers to be distributed as more clients are added to the system.

A more general approach to distributed system design is to remove the distinction between client and server and to design the system architecture as a distributed object architecture. In a distributed object architecture (Figure 11.11) the fundamental system components are objects that provide an interface to a set of services that

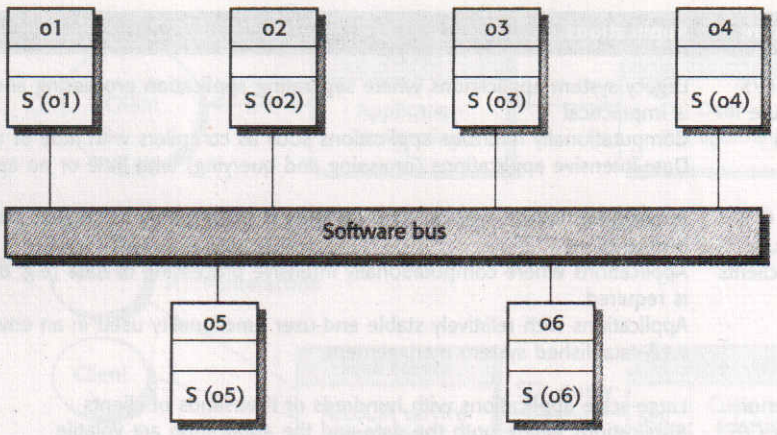


Figure 11.11 Distributed object architecture

they provide. Other objects call on these services with no logical distinction between a client (a receiver of a service) and a server (a provider of a service).

Objects may be distributed across a number of computers on a network and communicate through middleware. By analogy with a hardware bus that allows different cards to be plugged into it and supports communication between hardware devices, this middleware may be thought of as a software bus. It provides a set of services that allow objects to communicate and to be added and removed from the system. This middleware is called an object request broker. Its role is to provide a seamless interface between objects. I discuss object request brokers in section 11.4.

The advantages of this model of a distributed system architecture are:

- It allows the system designer to delay decisions on where and how services should be provided. Service-providing objects may execute on any node of the network. Therefore, the distinction between fat- and thin-client models becomes irrelevant as there is no need to decide in advance where application logic objects are located.
- It is a very open system architecture that allows new resources to be added to it as required. As I discuss in the following section, software bus standards have been developed and implemented that allow objects written in different programming languages to communicate and to provide services to each other.
- The system is flexible and scalable. Different instances of the system with the same service provided by different objects or by replicated objects can be created to cope with different system loads. New objects can be added as the load on the system increases without disrupting other system objects.
- It is possible to reconfigure the system dynamically with objects migrating across the network as required. This may be important where there are fluctuating patterns of demand on services. A service-providing object can migrate to the same processor as service-requesting objects, thus improving the performance of the system.

A distributed object architecture can be used in two ways in system design:

1. As a logical model that allows you to structure and organise 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 number of distributed objects. At this level, the objects that you design are usually large-grain objects (sometimes called business objects) that provide domain-specific services. For example, in a retail application there may be business objects concerned with stock control, customer communications, goods ordering, etc. This logical model can, of course, then be realised as an implementation model.

2. As a flexible approach to the implementation of client–server systems. In this case, the logical model of the system is a client–server model but both clients and servers are realised as distributed objects communicating through a software bus. This makes it possible to change the system easily, for example, from a two-tier to a multi-tier system. In this case, the server or the client may not be implemented as a single distributed object but may consist of a number of smaller objects that each provide specialised services.

An example of a type of system where a distributed object architecture might be appropriate is a data mining system that looks for relationships between the data that is stored in a number of different databases (Figure 11.12). An example of a data mining application might be where a retail business has different types of shop (say food stores and hardware stores) and they try to find relationships between the purchases of different types of food and different types of hardware. For instance, people who buy baby food may also buy particular types of wallpaper. With this knowledge, the business can then specifically target baby-food customers with combined offers.

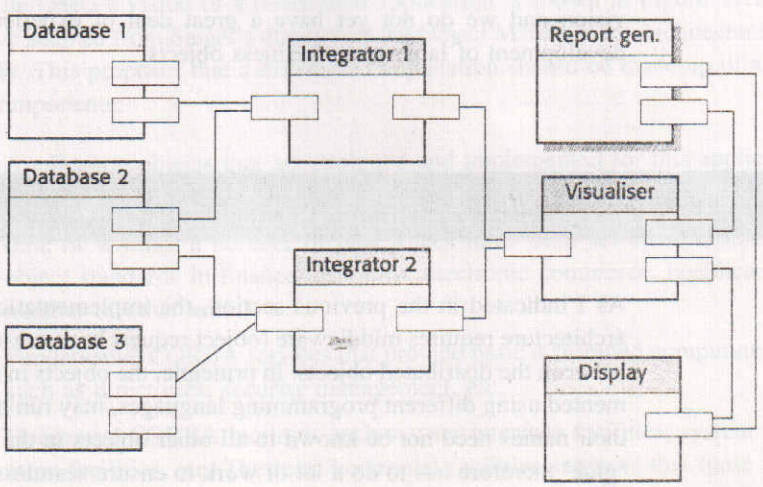


Figure 11.12
The distribution architecture of a data mining system

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

Visualiser objects interact with integrator objects to produce a visualisation or a report on the relationships that have been discovered. Because of the large volumes of data that are handled, visualiser objects will normally use graphical presentations of the relationships that have been discovered. I discuss graphical information presentation in Chapter 15.

A distributed object architecture rather than a client–server architecture is appropriate for this type of application, for three reasons:

1. Unlike a bank ATM system (say), the logical model of the system is not one of service provision where there are distinguished data management services.
2. It allows the number of databases that are accessed to be increased without disrupting the system. Each database is simply another distributed object. The database objects can provide a simplified interface that controls access to the data. The databases that are accessed may reside on different machines.
3. It allows new types of relationship to be mined by adding new integrator objects. Parts of the business that are interested in specific relationships can extend the system by adding integrator objects that operate on their computers without requiring knowledge of any other integrators that are used elsewhere.

The major disadvantage of distributed object architectures is that they are more complex to design than client–server systems. Client–server systems appear to be a fairly natural way to think about systems. They reflect many human transactions where people request and receive services from other people who specialise in providing these services. It is more difficult to think about more general service provision and we do not yet have a great deal of experience with the design and development of large-grain business objects.

11.4 CORBA

As I indicated in the previous section, the implementation of a distributed object architecture requires middleware (object request brokers) to handle communications between the distributed objects. In principle, the objects in the system may be implemented using different programming languages, may run on different platforms and their names need not be known to all other objects in the system. The middleware 'glue' therefore has to do a lot of work to ensure seamless object communications.

At the time of writing, there are two principal standards for middleware to support distributed object computing:

1. *CORBA (Common Object Request Broker Architecture)* CORBA is a set of standards for middleware that has been defined by the Object Management Group (OMG). The OMG is a consortium of companies, including major players such as Sun, Hewlett-Packard and IBM. The CORBA standards define a generic machine-independent approach to distributed object computing. A number of implementations of this standard by different vendors have been developed. CORBA implementations are available for Unix and Microsoft operating systems.
2. *DCOM (Distributed Component Object Model)* DCOM is a standard that has been developed and implemented by Microsoft and that is integrated with Microsoft's operating system. Its model of distributed computing is less general than the CORBA model and DCOM offers more limited support for interoperability. At the time of writing, use of DCOM is limited to Microsoft operating systems.

I focus on CORBA here because it is the more general model. I also think it likely that CORBA, DCOM and other approaches such as Java's RMI (Remote Method Invocation) will converge and this convergence will be based on the CORBA standards. There is no need for more than one standard in this area and different standards hinder long-term progress.

The CORBA standards have been defined by the Object Management Group (OMG) which is made up of over 500 companies to promote object-oriented development. The role of the OMG is to define standards for object-oriented development but not to provide specific implementations of these standards. The defined standards are publicly available free of charge from the OMG's web site. The OMG is not just concerned with CORBA but has defined a wide range of other standards including UML (Rumbaugh *et al.*, 1999a) and common business objects. I have included a link to these standards from the book's web pages.

The OMG's vision of a distributed application is shown in Figure 11.13 which I have adapted from Siegel's diagram of the Object Management Architecture (Siegel, 1998). This proposes that a distributed application should be made up of a number of components:

1. Application objects that are designed and implemented for this application.
2. Standard objects that are defined by the OMG for a specific domain. At the time of writing, a number of task forces have been set up to define domain object standards in finance/insurance, electronic commerce, healthcare, and a number of other areas.
3. Fundamental CORBA services that provide basic distributed computing services such as directories, security management, etc.
4. Horizontal CORBA facilities such as user interface facilities, system management facilities, etc. The term horizontal facilities suggests that these facilities

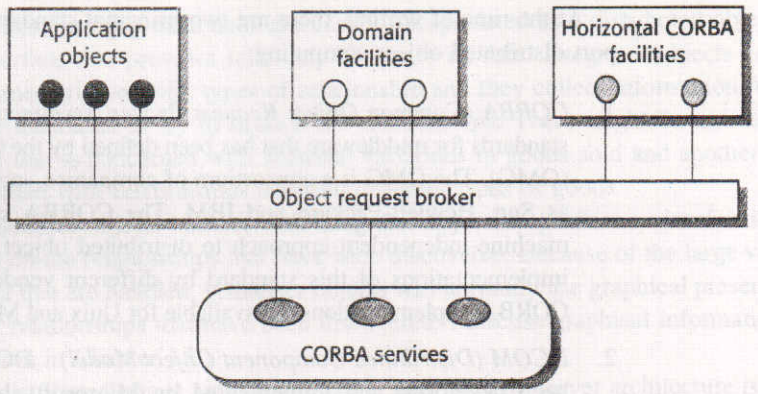


Figure 11.13 The structure of a CORBA-based distributed application

are common to many application domains and the facilities are therefore used in many different applications.

The CORBA standards cover all aspects of this vision. There are four major elements to these standards:

1. An object model for application objects where a CORBA object is an encapsulation of state with a well-defined, language-neutral interface described in an IDL (Interface Definition Language).
2. An object request broker (ORB) that manages requests for object services. The ORB locates the object providing the service, prepares it for the request, sends the service request and returns the results to the requester.
3. A set of object services that are general services likely to be required by many distributed applications. Examples of services are directory services, transaction services and persistence services.
4. A set of common components built on top of these basic services that may be required by applications. These may be vertical domain-specific components or horizontal, general-purpose components that are used by many applications. I discuss components in Chapter 14.

The CORBA object model considers an object to be an encapsulation of attributes and services as is normal for objects. However, CORBA objects must have a separate interface definition that defines the public attributes and operations of the object. CORBA object interfaces are defined using a standard, language-independent interface definition language (IDL). If an object wishes to use services provided by another object then it accesses these services through the IDL interface. CORBA objects have a unique identifier called an Interoperable Object Reference (IOR). This IOR is used when one object requests services from another.

The object request broker knows about the objects that are requesting services and their interfaces. The ORB handles the communication between the objects. The

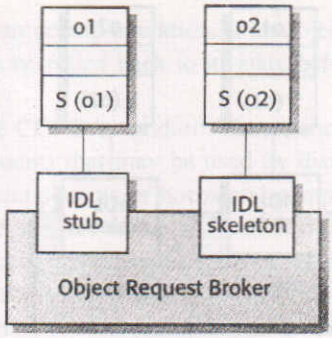


Figure 11.14 Object communications through an ORB

communicating objects do not need to know the location of other objects, nor do they need to know anything about their implementation. As the IDL interface insulates the objects from the ORB, it is possible to change the object implementation in a completely transparent way. The object location can change between invocations and this is transparent to other objects in the system.

This is illustrated in Figure 11.14 which illustrates how two objects o1 and o2 may communicate through an ORB. The calling object (o1) has an associated IDL stub that defines the interface of the object that is providing the required service. The implementor of o1 embeds calls to this stub in his or her object implementation when a service is required. The IDL is a superset of C++ so it is very easy to access this stub if you are programming in C++ and fairly easy in C or Java. Language mappings to IDL have also been defined for other languages such as Ada and COBOL. In these cases, tool support is usually required to support the link to the IDL.

The object that is providing the service has an associated IDL skeleton that links the interface to the implementation of the services. Simplistically, when a service is called through the interface, the IDL skeleton translates this into a call to the service in whatever implementation language has been used. When the method or procedure has been executed, the IDL skeleton translates the results into IDL so that they can be accessed by the calling object. Where an object both provides services to other objects and uses services that are provided elsewhere, it needs both an IDL skeleton and IDL stubs. An IDL stub is required for every object that is used.

Object request brokers are not usually implemented as separate processes but are a framework (see Chapter 14) that can be linked with object implementations. Therefore, in a distributed system, each computer that is running distributed objects will have its own object request broker. This will handle all local invocations of objects. However, when a request is made to a service that is to be provided by a remote object, this requires inter-ORB communications.

This situation is illustrated in Figure 11.15. In this example, if object o1 or o2 requests a service from o3 or o4, this requires the associated ORBs to communicate. A CORBA implementation supports ORB-to-ORB communication by providing all ORBs access to all IDL interface definitions and by implementing the OMG's standard Generic Inter-ORB Protocol (GIOP). This protocol defines standard messages that ORBs can exchange to implement remote object invocation and

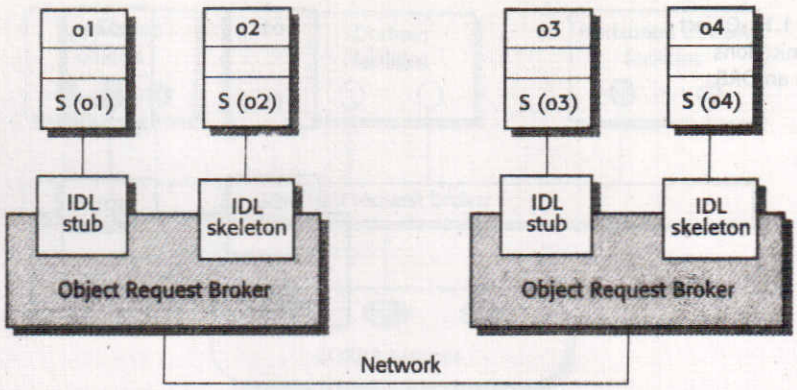


Figure 11.15 Inter-ORB communications

information transfer. When combined with lower-level Internet TCP/IP protocols, the GIOP allows ORBs to communicate across the Internet.

The CORBA initiative has been under way since the 1980s and the early versions of CORBA were simply concerned with supporting distributed objects. However, as the standards have evolved they have become more extensive. As well as a mechanism for distributed object communications, the CORBA standards now define some standard services that may be provided to support distributed object oriented applications.

You can think of CORBA services as those facilities that are likely to be required by many distributed systems. The standards define approximately 15 common services. Some examples of these generic services are:

1. Naming and trading services that allow objects to refer to and discover other objects on the network. The naming service is a directory service that allows objects to be named. Other objects may discover the IOR of objects that they wish to use through the naming service. This is like the white pages of a phone directory. Objects using the service have to know the registered names of other objects. The trading services are like the yellow pages. Objects can find out what other objects have registered with the trader service and can access the specification of these objects.

2. Notification services that allow objects to notify other objects that some event has occurred. Objects may register their interest in a particular event with the service and, when that event occurs, they are automatically notified. For example, say the system includes a print spooler that queues documents to be printed and a number of printer objects. The print spooler registers that it is interested in an 'end of printing' event from a printer object. The notification service informs it when printing is complete and it can then schedule the next document on that printer.

3. Transaction services that support atomic transactions and rollback on failure. Transactions are a fault-tolerance facility (see Chapter 18) that support recovery

from errors during an update operation. If an object update operation fails then the object state can be rolled back to its state before the update was started.

It is intended that the CORBA standards should include interface definitions for a wide range of components that may be used by distributed application builders. These may be vertical components or horizontal components. Vertical components are components that are specific to an application domain. As I have already discussed, various task forces from different industry sectors have been set up to define these components. Horizontal components are general-purpose components such as user interface components. A task force is also responsible for developing these specifications.

At the time of writing, these component specifications are still under development and have not yet been agreed. In my view, this is likely to be the weakest area of the CORBA standards and it may be several years before standard component specifications and implementations are available.

KEY POINTS

- Virtually all new large systems are now distributed systems where the system software runs on a loosely integrated group of networked processors.
- Distributed systems can support resource sharing, openness, concurrency, scalability, fault tolerance and transparency.
- Client–server systems are distributed systems where the system is modelled as a set of services that are provided by servers to client processes.
- In a client–server system, the user interface always runs on a client and data management is always provided by a shared server. Application functionality may be implemented on the client computer or on the server.
- In a distributed object architecture, there is no distinction between clients and servers. Objects provide general services that may be called on by other objects. This approach may be used for implementing client–server systems.
- Distributed object systems require middleware to handle object communications and to allow objects to be added to and removed from the system. Conceptually, you can think of this middleware as a software bus that objects plug into.
- The CORBA standards are a set of standards for middleware that supports distributed object architectures. They include object model definitions, definitions of an object request broker and common service definitions. Various implementations of the CORBA standards are available.

FURTHER READING

Communications of the ACM, October 1998. This special issue includes a number of articles, written for non-specialised readers, on CORBA. I particularly recommend the introductory article by Siegel.

Client–server Programming with Java and CORBA. A how-to guide on developing distributed client–server systems. It focuses on developing these systems in Java and discusses Java's approach to remote method invocation and its relationships with CORBA. (R. Orfali and D. Harkey, 1998, John Wiley and Sons.)

'Frameworks for component-based client–server computing'. This review discusses client–server architectures and both CORBA and DCOM and how they may be used as a basis for implementing client–server systems. (S. M. Lewandowski, *Computer Surveys*, 30(1), March 1998.)

'Middleware: A model for distributed systems services'. This is an excellent overview paper that summarises 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.)

EXERCISES

11.1 Explain why distributed systems are inherently more scalable than centralised systems. What are the likely limits on the scalability of the system?

11.2 What is the fundamental difference between a fat-client and a thin-client approach to client–server systems development? Explain why the use of Java as an implementation language blurs the distinction between these approaches.

11.3 It is proposed to develop a system for stock information where dealers can access information about companies and can evaluate various investment scenarios using a simulation system. Different dealers use this simulation in different ways according to their experience and the type of stocks that they are dealing with. Suggest a client–server architecture for this system that shows where functionality is located. Justify the client–server system model that you have chosen.

11.4 By making reference to the application model shown in Figure 11.5, discuss possible problems that might arise when converting a 1980s mainframe legacy system for insurance policy processing to a client–server architecture.

11.5 Distributed systems based on a client–server model have been developed since the 1980s but it is only recently that system architectures based on distributed objects have been implemented. Suggest three reasons why this should be the case.

11.6 Explain why the use of distributed objects with an object request broker simplifies the implementation of scalable client–server systems. Illustrate your answer with an example.

11.7 How is the CORBA IDL used to support communications between objects that have been implemented in different programming languages? Explain why this approach may cause performance problems if there are radical differences between the languages used for object implementation.

11.8 What are the basic facilities that must be provided by an object request broker?

11.9 It could be argued that developing CORBA standards for horizontal and vertical components is anti-competitive. If these are developed and adopted, it inhibits the development of better components by smaller companies. Discuss the general role of standardisation in supporting and restricting competition in the software market.