CHAPTER 2 MODELLING FOR DISTRIBUTED NETWORK SYSTEMS: THE CLIENT-SERVER MODEL

This chapter is to introduce the client-server model and its role in the development of distributed network systems. The chapter discusses the cooperation between clients and servers/group servers in distributed network systems, and addresses extensions to the client-server model. Service discovery, which is of crucial importance for achieving transparency in distributed network systems, is also elaborated in this chapter.

2.1 Issues Leading to the Client-Server Model

By amalgamating computers and networks into one single computing system and providing appropriate system software, a distributed computing system has created the possibility of sharing information and peripheral resources. Furthermore, these systems improved performance of a computing system and individual users through parallel execution of programs, load balancing and sharing, and replication of programs and data. Distributed computing systems are also characterised by enhanced availability, and increased reliability.

However, the amalgamation process has also generated some serious challenges and problems. The most important, critical challenge was to synthesise a model of distributed computing to be used in the development of both application and system software. Another critical challenge was to develop ways to hide distribution of resources and build relevant services upon them. The development of distributed computing systems is complicated by the lack of a central clock and centrally available data to manage the whole system. Furthermore, amalgamating computers and networks into one single computing system generates a need to deal with the problems of resource protection, communication security and authentication.

The synthesis of a distributed computing model has been influenced by a need to deal with the issues caused by distribution, such as locating data, programs and peripheral resources, accessing remote data, programs and peripheral resources, supporting cooperation and competition between programs executing on different computers, coordinating distributed programs executing on different computers, maintaining the consistency of replicated data and programs, detecting and recovering from failures, protecting data and programs stored and in transit, and authenticating users, etc.

2.2 The Client-Server Model in a Distributed Computing System

A distributed computing system is a set of application and system programs, and data dispersed across a number of independent personal computers connected by a communication network. In order to provide requested services to users the system and relevant application programs must be executed. Because services are provided as a result of executing programs on a number of computers with data stored on one or more locations, the whole computing activity is called *distributed computing*.

2.2.1 Basic Concepts

The problem is how to formalise the development of distributed computing. The above shows that the main issue of distributed computing is programs in execution, which are called *processes.* The second issue is that these processes cooperate or compete in order to provide the requested services. This means that these processes are synchronised.

A natural model of distributed computing is the client-server model, which is able to deal with the problems generated by distribution, could be used to describe computation processes and their behaviour when providing services to users, and allows design of system and application software for distributed computing systems.

According to this model there are two processes, the *client,* which requests a service from another process, and the *server,* which is the service provider. The server performs the requested service and sends back a response. This response could be a processing result, a confirmation of completion of the requested operation or even a notice about a failure of an operation.

From the user's point of view a distributed computing system can provide the following services: printing, electronic mail, file service, authentication, naming, database service and computing service. These services are provided by appropriate servers. Because of the restricted number of servers (implied by a restricted number of resources on which these servers were implemented), clients compete for these servers.

An association between this abstract model and its physical implementation is shown in Figure 2.1. In particular the basic items of the model: the client and server, and request and response are shown. In this case, the client and server processes execute on two different computers. They communicate at the *virtual* (logical) level by exchanging requests and responses. In order to achieve this virtual communication, physical messages are sent between these two processes. This implies that operating systems of computers and a communication system of a distributed computing system are actively involved in the service provision.

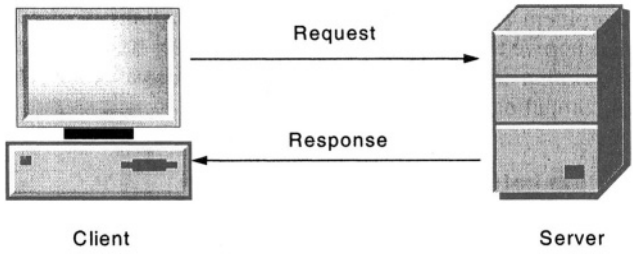

Figure 2.1: The basic client-server model

A more detailed client-server model has three components:

- *Service:* A service is a software entity that runs on one or more machines. It provides an abstraction of a set of well-defined operations in response to applications' requests.
- *Server:* A server is an instance of a particular service running on a single machine.
- *Client:* A client is a software entity that exploits services provided by servers. A client can but does not have to interface directly with a human user.

2.2.2 Features and Problems of the Client-Server Model

The most important features of the client-server model are simplicity, modularity, extensibility and flexibility. Simplicity manifests itself by closely matching the flow of data with the control flow. Modularity is achieved by organising and integrating a group of computer operations into a separate service. Also any set of data with operations on this data can be organised as a separate service. The whole distributed computing system developed based on the client-server model can be easily extended by adding new services in the form of new servers. The servers which do not satisfy user requirements can be easily modified or even removed. Only the interfaces between the clients and servers must be maintained.

There are three major problems of the client-server model:

- The first is due to the fact that the control of individual resources is centralised in a single server. This means that if the computer supporting a server fails, then that element of control fails. Such a solution is not tolerable if a control function of a server is critical to the operation of the system (e.g., a name server, a file server, an authentication server). Thus, the reliability and availability of an operation depending on multiple servers is a product of reliability of all computers and devices, and communication lines.
- The second problem is that each single server is a potential bottleneck. The problem is exacerbated as more computers with potential clients are added to the system.

- The third problem arises when multiple implementations of similar functions are used to improve the performance of a client-server based system because of a need to maintain consistency. Furthermore, this increases the total costs of a distributed computing system.

2.3 Cooperation between Clients and Servers

2.3.1 Cooperation Type and Chained Server

A system in which there is only one server and one client would not be able to provide high performance, and reliable and cost effective services to users. As mentioned in the previous section, it is necessary to use one server to provide services to more than one client. The simplest cooperation between clients and servers based on sharing allows for lowering the costs of the whole system and more effective use of resources. An example of a service based on this cooperation is a printing service. Figure 2.2 shows a printer server providing services to *n* clients, which all are connected by a local area network.

In a distributed computing system there are two different types of cooperation between clients and servers. The first type assumes that a client requests a temporary service. The second one is generated by a client that wants to arrange a number of calls to be directed to a particular serving process. This implies a need for establishing long term bindings between this client and a server.

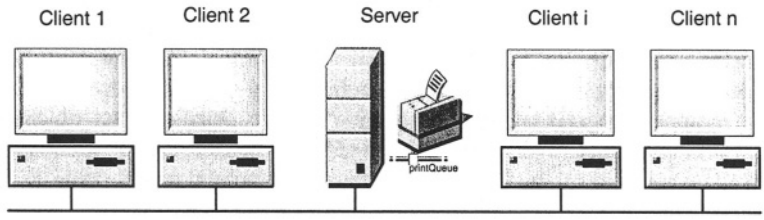

Figure 2.2: Printing service (a service example)

Processes can act as either clients or servers, depending on the context. A file server that receives a request to read a file from a user's client process must check on the access rights of this user. For this purpose the file server sends a request to an authentication server and waits for its response. The response of the file server to the client depends on the response from the authentication server. This implies that the file server acts as a client of the authentication server. Thus, a service provided to the user by a distributed computing system based on the client-server model could require a chain of *cooperating servers.*

2.3.2 Multiple Servers

A distributed computing system has the following functions:

- Improve performance through parallel execution of programs on a cluster (sometimes called network) of workstations,
- Decrease response time of databases through data replication,
- Support synchronous distant meetings,
- Support cooperative workgroups, and
- Increase reliability by service multiplication, etc.

To perform these functions, many servers must contribute to the overall application. This implies a need to invoke multiple services. Furthermore, it would require in some cases simultaneous requests to be sent to a number of servers. Different applications will require different semantics for the cooperation between clients and servers, as illustrated in the following paragraphs.

Cooperation in the systems supporting parallel execution

In a distributed computing system supporting parallel execution, there are some parts of a program which could be executed as individual processes on separate computers. For this purpose a process (parent), which coordinates such parallel processing of individual processes (children), causes them to execute on selected idle computers and waits for computational results. In this case the parent process acts as a client and the child processes as servers, in a one-to-many communication pattern. However, the parent process cannot proceed any further unless all children send back responses.

This example shows that there are two questions which should be answered in order to improve the cooperation between the client and servers: who is responsible for locating idle computers on which servers can run, and who is responsible for setting up those servers on remote computers and coordinating responses.

Cooperation in the systems supporting a distributed database

Similar semantics of cooperation between a client and multiple servers in a distributed computing system occur in supporting a distributed database. To commit a transaction all operations performed on a set of databases must be completed successfully. Thus, a client process which executes a transaction sends operation requests to relevant databases (servers) and waits for the results of the operations. The client process can be involved in other operations, however, responses from all database servers must be received to commit the transaction.

In this case there is no need to set up servers on idle computers — there are servers which already run on dedicated computers. However, there is an issue of who can deal with these database servers, the client process or another entity working on behalf of this client.

Cooperation in the systems supporting a user application

There are different semantics of cooperation between a client and multiple servers in a distributed computing system supporting a user application. This requires identifying a database server which manages relevant data, accesses that database server and carries out some operations on data, and prints that data. There are three servers which must be engaged in supporting the application: a service discovery server, a database server and a printer server. The client process running the application software invokes each server in a sequential order and waits for a response before accessing the next server.

In this case there is again no need to set up servers on idle computers. However, the same issue exists, i.e., who can deal with these database servers, the client process or another entity working on behalf of this client.

Cooperation in the systems supporting mission critical applications

A different form of cooperation between a client and multiple servers is required in reliable distributed computing systems, in particular those which support mission critical applications. In this case a request sent to a group of servers should generate identical responses from all of them. An example of such a system is a redundant computational server on board a space ship, or a fault-tolerant transaction oriented database. In any case only identical operation results are accepted by the client.

These examples show that distributed computing systems have moved from the basic one-to-one client-server model to the one-to-many and chain models in order to improve performance and reliability. Furthermore, the issues identified when discussing the one-to-many communication pattern of the client-server model demonstrate that client and servers cooperation can be strongly influenced and supported by some active entities which are extensions to the client-server model. The following sections address these issues.

2.4 Extensions to the Client-Server Model

The need for extending the client-server model can be specified as an outcome of a study into involvement of other entities in the provision of services, an interface between a client and server, and the behaviour of a client after sending of a request to a server.

2.4.1 Agents and Indirect Client-Server Cooperation

A client and server can cooperate either directly or indirectly. In the former case there is no additional entity that participates in exchanging requests and responses between a client and a server. Indirect cooperation in the client-server model requires two additional entities, called *agents,* to request a service and to be provided with the requested service. Figure 2.3 shows such an extension.

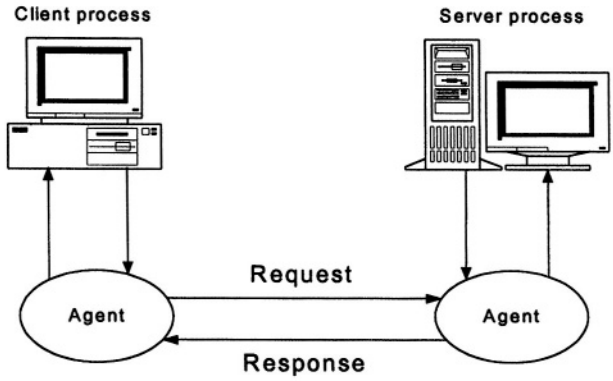

Figure 2.3: Indirect client-server cooperation

The role of these agents can vary from a simple communication module which hides communication network details to an entity which is involved in mediating between clients and servers, resolving heterogeneity issues, and managing resources and cooperating servers. These aspects will be elaborated in other chapters of this book.

As presented in Sections 2.3, a client can invoke desired servers explicitly by sending direct requests to multiple servers. In this case the programmer of a user application must concentrate both on an application and on managing server cooperation and communication. Writing resource management and communication software is expensive, time consuming and error prone. The interface between the client and the server is complicated, differs from one application to another, and the whole service provided is not transparent to the client process (user).

Clients can also request multiple services implicitly. This requires the client to send only one request to a general server. A requested service will be composed by this invoked server cooperating with other servers, based on information provided in the request. After completion of necessary operations by involved servers, the invoked server sends a response back to the client. This coordination operation can be performed by a properly designed agent. Despite the fact that such an agent is quite complicated, the cooperation between the client and the server is based on a single, well-defined interface. Furthermore, transparency is provided to the client which reduces the complexity of the application.

Cooperation between a client and multiple servers can be supported by a simple communication system which employs a direct, one-to-one message protocol. Although this communication model is simple, its performance is poor because each server involved must be invoked by sending a separate message. The overall performance of a communication system supporting message delivery in a client-server based distributed computing system can be dramatically improved if a one-to-many communication pattern is used. In this case a single request is sent by the client process to all servers, specified by a single group name. The use of multicast at the physical/data link layer does improve this system, but it is not essential.

2.4.2 The Three-Tier Client-Server Architecture

Agents and servers acting as clients can generate different architectures of distributed computing systems. The three-tier client-server architecture extends the basic client-server model by adding a middle tier to support the application logic and common services. In this architecture, a distributed application consists of the following three types of components:

- *User interface and presentation processing*. These components are responsible for accepting inputs and presenting the results. They belong to the client tier;
- *Computational function processing*. These components are responsible for providing transparent, reliable, secure, and efficient distributed computing. They are also responsible for performing necessary processing to solve a particular application problem. We say these components belong to the application tier;

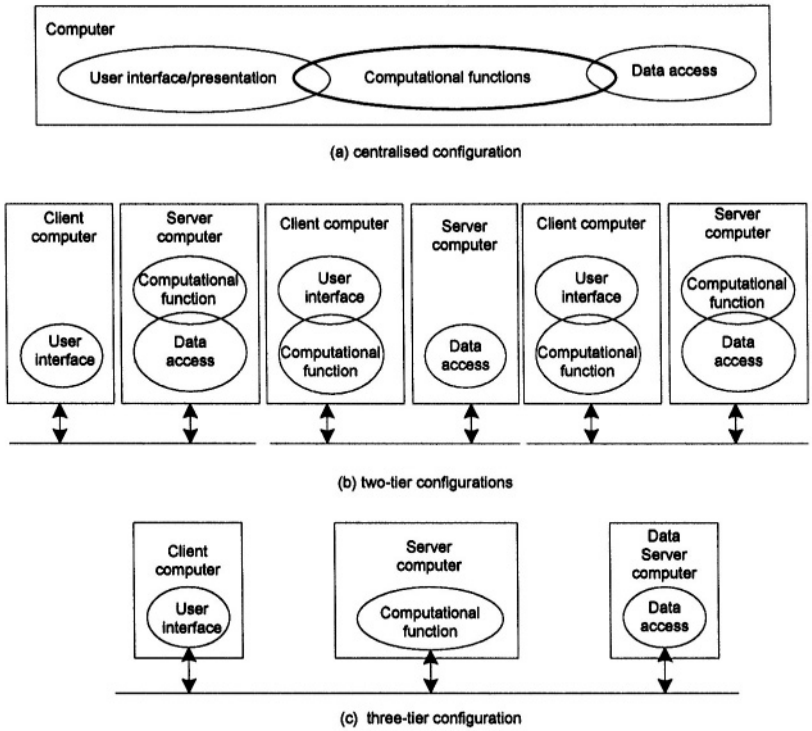

Figure 2.4: Examples of three-tier configurations

- *Data access processing.* These components are responsible for accessing data stored on external storage devices (such as disk drives). They belong to the back-end tier.

These components can be combined and distributed in various ways to create different configurations with varying complexity. Figure 2.4 shows some examples of such configurations ranging from centralised processing to three-tier distribution. In particular, Figure 2.4(a) shows a centralised configuration where all the three types of components are located in a single computer. Figure 2.4(b) shows three two-tier configurations where the three types of components are distributed on two computers. Figure 2.4(c) shows a three-tier configuration where all the three types of components are distributed on different computers.

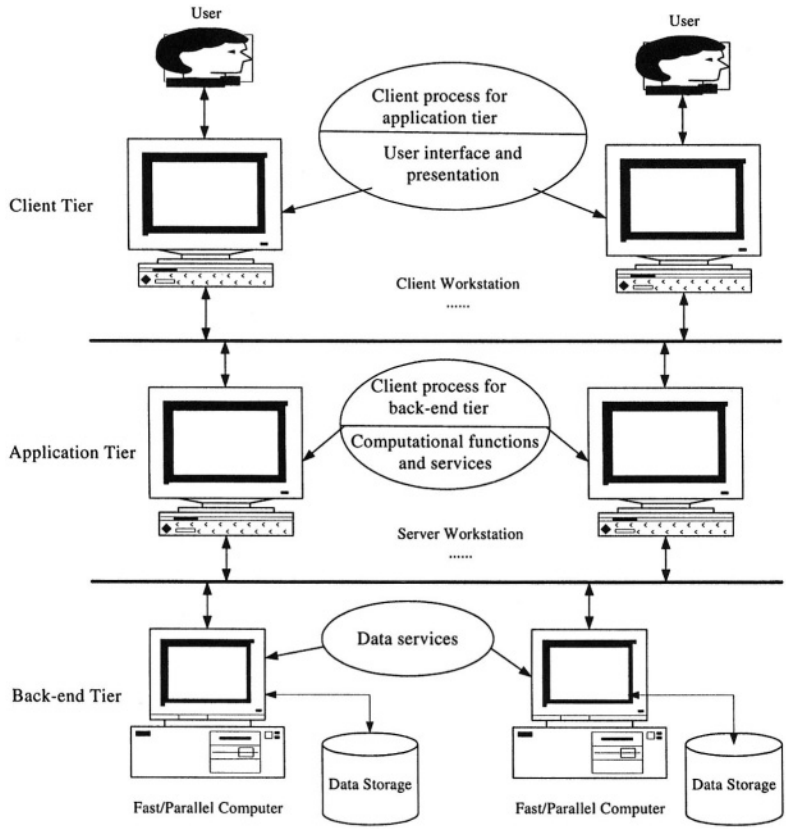

Figure 2.5: An example implementation of the three-tier architecture

Figure 2.5 illustrates an example of implementation of the three-tier architecture. In this example, the upper tier consists of client computers that run user interface processing software. The middle tier contains computers that run computational function processing software. The bottom tier includes back-end data servers. In a

three-tier client-server architecture, application clients usually do not interact directly with the data servers, instead, they interact with the middle tier servers to obtain services. The middle tier servers will then either fulfil the requests themselves, sending the result back to the clients, or more commonly, if additional resources are required, servers in the middle tier will act (as clients themselves) on behalf of the application clients to interact with the data servers in the bottom tier or other servers within the middle tier.

Compared with a normal two-tier client-server architecture, the three-tier client-server architecture has the following two important advantages:

- *Better transparency.* The servers within the application tier of the three-tier architecture allow an application to detach user interface from back-end resources and therefore provide better location and migration transparency. That is, the location or implementation of back-end resources can be changed without affecting the programs within the client tier;
- *Better scalability.* The centralised and two-tier architectures do not scale well to support large applications. The servers within the application tier of the three-tier architecture, however, inject another dimension of scalability into the client-server environment.

Other benefits that the three-tier client-server architecture may achieve include better *concurrency, flexibility, reusability, load balancing,* and *reliability.*

In addition to providing services for business logic and applications, the application tier should provide the following key services (they sometimes are called *middleware*)*:*

- *Directory services.* These services are required to locate application services and resources and route messages there.
- *Security services.* An integrated security service is needed to provide a comprehensive inter-application client-server security mechanism.
- *Time services.* These services provide a universal format for representing time on different platforms running in different countries in various time zones. This is critical in maintaining error logs and timestamps, and in keeping synchronisation among application processes.
- *Transaction services.* These services provide transaction semantics to support commit, rollback, and recovery mechanisms. These mechanisms are critical to ensure that updates across one or more databases are handled correctly and that data integrity is not jeopardised.

2.5 Service Discovery

To invoke a desired service a client must know whether there is a server which is able to provide this service and its characteristics if it exists, and its name and location. This is the issue of *service discovery.* In the case of a simple distributed computing system, where there are only a few servers, there is no need to identify

an existence of a desired server. Information about all available servers is a priori. This implies that service discovery is restricted to locating the server, which provides the desired service. On the other hand, in a large distributed computing system which is a federation of a set of distributed computing systems, with many service providers who offer and withdraw these services dynamically, there is a need to learn both whether a proper service (e.g., a very fast colour printer of high quality) is available at a given time, and if so its name and location.

It is worth mentioning that a client in a distributed computing system managed by a distributed operating system, which provides *transparency* (one of the most important features of a distributed computing system), should only know a name of either a server or an agent working on behalf of the server. On the other hand, a client in a distributed computing system managed by a set of centralised operating systems and their extensions to access remote resources and services must know both its name and location. The reason is that transparency is not provided.

Service discovery is achieved through the following modes:

- Server computer address is hardwired into client code;
- Broadcast is used to locate servers;
- Name server is used to locate services; and
- Brokers are used to locate servers.

We discuss them in detail next.

2.5.1 Hardwiring Computer Address

This approach requires only a location of the server, in the form of computer address, to be provided. However, it is only applicable in very small and simple systems, where there is only one server process running on the destination computer. Thus, an operating system knows where to deliver an incoming request.

Another version of this approach is based on a much more advanced naming system, where requests are sent to processes rather than to computers. In this case each process is named by a pair <*computer_address, process_name*>. A client is provided with not only a name of a server, but also with the address of a server computer. This solution is not location transparent as the user is aware of the location of the server. The lack of transparency can create a problem when there is a need to move a server to another computer, and a pair <*computer_address, process_name*> has been hardwired in client code.

2.5.2 Broadcast Approach

According to this approach each process has a unique name (e.g., a very long identifier can be used for this purpose). In order to send a request a client knows a name of a destination, in particular of a server. However this is not enough because an operating system of a computer where the server runs must know an address of

the server's computer. For this purpose the client's operating system broadcasts a special locate request containing the name of the server, which will be received by all computers on a network. An operating system that finds the server's name in the list of its processes, which means that the named server runs on its computer, sends back a 'here I am' response containing its address (location). The client's operating system receives the response and can store (cache) the server's computer address for future communication. This approach is transparent, however the broadcast overhead is high as all computers on a network are involved in the processing of the locate request.

The cooperation between clients, servers and operating systems supporting them in a distributed computing system using the broadcast approach to locate servers is illustrated in Figure 2.6.

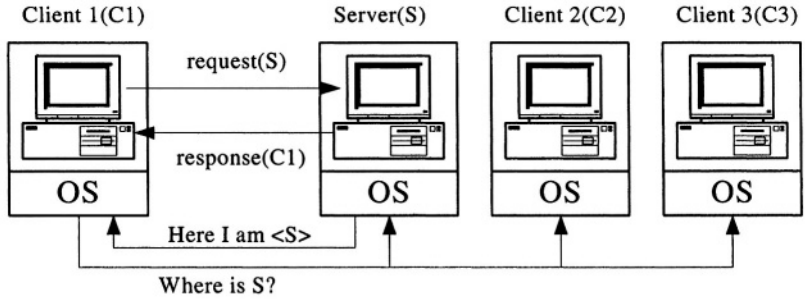

Figure 2.6: Service discovery -- broadcast approach

2.5.3 Name Server Approach

This approach is very similar to the broadcast based approach, however it reduces the broadcast overhead. In order to learn the address of a desired server, an operating system of the client's computer sends a 'where is' request to a special system server, called a *name server,* asking for the address of a computer where the desired server runs. This means that the name and location (computer address) of the name server are known to all operating systems. The name server sends back a response containing an address of the desired server. The client's operating system receives the response and can cache the server's computer address for future communication.

This approach is transparent and much more efficient than the broadcast based approach. However, because the name server is centralised, the overall performance of a distributed computing system could be degraded, as the name server could be a bottleneck. Furthermore, reliability of this approach is low; if a name server computer crashes, a distributed computing system cannot work.

Figure 2.7 illustrates the cooperation between clients, servers and operating systems supporting them in a distributed computing system using the approach of server

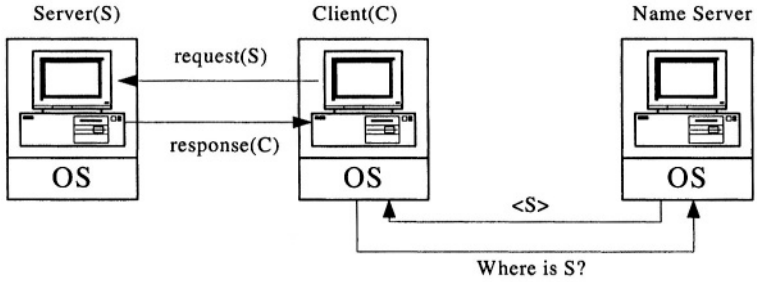

Figure 2.7: Service discovery -- name server and server location lookup

2.5.4 Broker-Based Location Lookup

A client in a distributed computing system supported by any of the above server location approaches must know all servers available in that system and their names (also the server locations (addresses) in systems which do not provide transparency) in order to invoke a desired service. In a large distributed computing system there could be a large number of servers. Moreover, servers of the same type can be characterised by different attributes describing the services they provide (e.g., one laser printer is a colour printer, another is a black and white printer). Furthermore, servers can be offered by some users and revoked dynamically. A user is not able to know names and attributes of all these servers, and their dynamically changing availability. There must be a server which could support users to deal with these problems. This server is called a broker. Thus, a *broker* is a server that

- allows a client to identify available servers which can be characterised by a set of attributes that describe the properties of a desired service;
- mediates cooperation between clients and servers;
- allows service providers to register the services they support by providing their names, locations and features in the form of attributes;
- advertises registered services and makes them available to clients; and
- withdraws services dynamically.

A broker-based approach is very similar to the server location lookup performed via a name server approach. However, there are real conceptual differences between a broker and a name server which frees clients from remembering ASCII names or path names of all servers (and eventually the server locations), and allows clients to identify attributes of servers and learn about their availability. Thus, a broker is a server, which embodies both service management and naming services. There are two basic broker classes, which form two different forms of cooperation between clients and servers:

Forwarding broker

Cooperation between a client and a server mediated by this broker is as follows:

- Step 1: the broker receives from a client a service enquiry in a form of a set of attributes that characterise a desired service, and a server operation request;
- Step 2: if a matching server is available, the broker sends the server operation request to that found server; otherwise, it sends a failure response to the client;
- Step 3: the server sends back a response to the broker;
- Step 4: the broker passes the response to the client.

The forwarding broker possesses advantages and disadvantages of a system with a centralised server. This means that all requests to servers and their responses are going through this broker.

Direct broker

Cooperation between a client and a server mediated by this broker is as follows:

- Step 1: the broker receives from a client a service enquiry in a form of a set of attributes that characterise a desired service;
- Step 2: if a matching server is available, the broker sends back a name and a server computer address to the client; otherwise, it sends a failure response;
- Step 3: the client sends the server operation request to the server;
- Step 4: the server sends back a response to the client.

Despite the fact that the direct broker also possesses advantages and disadvantages of a system with a centralised server, its performance is better than that of the forwarding broker, because only service enquiry messages are sent to this broker.

2.6 Client-Server Interoperability

Reusability of servers is a critical issue for both users and software manufacture due to the high cost of software writing. This issue could be easily resolved in a homogeneous environment because accessing mechanisms of clients may be made compatible with software interfaces, with static compatibility specified by types and dynamic compatibility by protocols.

Cooperation between heterogeneous clients and servers is much more difficult as they are not fully compatible. Thus, the issue is how to make them interoperable. Wegner [Wegner 1996] defines *interoperability* as the ability of two or more software components to cooperate despite differences in language, interface, and execution platform.

There are two aspects of client-server interoperability: a unit of interoperation, and interoperation mechanisms. The basic unit of interoperation is a procedure [Wegner 1996]. However, larger-granularity units of interoperation may be required by software components. Furthermore, preservation of temporal and functional properties may also be required.

There are two major mechanisms for interoperation:

- *Interface standardisation:* the objective of this mechanism is to map client and server interfaces to a common representation. The advantages of this mechanism are: (i) it separates communication models of clients from those of servers, and (ii) it provides scalability, since it only requires $m + n$ mappings, where m and n are the number of clients and servers, respectively. The disadvantage of this mechanism is that it is closed.
- *Interface bridging:* the objective of this mechanism is to provide a two-way mapping between a client and a server. The advantages of this mechanism are: (i) openness, and (ii) flexibility — it can be tailored to the requirements of a given client and server pair. However, this mechanism does not scale as well as the interface standardisation mechanism, as it requires $m * n$ mappings.

2.7 The Relationship

In Section 2.2 we said that in order to allow a client and a server to exchange requests and responses, there is a need to employ a communication network that links computers on which these processes run. We also demonstrated in Section 2.6 that in order to locate a server, the operating systems must be involved. The question is what would be the architecture of a distributed computing system that supports a distributed application developed on the basis of client-server model. Figure 2.8 illustrates the relationship between such a distributed application, the operating system supporting it and communication facility of a distributed computing system.

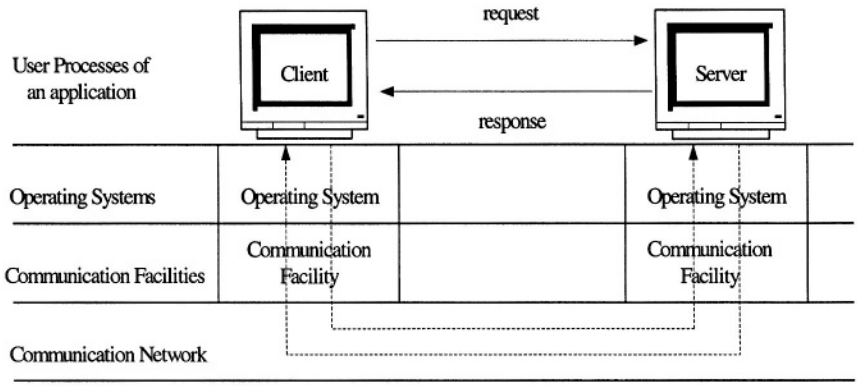

Figure 2.8: A distributed computing system architecture

This figure shows the distributed nature of the operating system, which could be either achieved by:

adding a module to the local centralised operating system of each computer, which allows processes to access remote resources and services; however, in the majority of cases this solution does not fully support transparency, or

- employing a distributed operating system, which hides distribution of resources and services; this solution, although futuristic from the current practice point of view, provides location transparency.

It is clear that the extensions to the basic client-server model, described in the previous sections, are achieved through an operating system. Furthermore, network communication services are invoked by an operating system on behalf of cooperating clients and servers.

The overall performance of a distributed computing system developed on the basis of client-server model depends on the performance of a communication facility. Such a facility is comprised of an inter-process communication system of an operating system and network protocols. Communication between clients and servers is discussed in Chapter 5.

Another question is the role of the client-server model in the development of operating systems and communication facilities for distributed computing systems. It will be demonstrated in later chapters that operating systems and network communication systems can also be developed based on the client-server model. This is the current approach to the design of such complicated software following the results of research and practice of software engineering.

2.8 Summary

In this chapter we introduced the client-server model and some concepts related to this model. Partitioning software into clients and servers allows us to place these components independently on computers in a distributed computing system. Furthermore, it allows these clients and servers to execute on different computers in a distributed computing system in order to complete the processing of an application in an integrated manner. This paves the way to high productivity and high performance in distributed computing. The client-server model is becoming the predominant form of software application design and operation. However, to fully benefit from the client-server model, many issues such as client and server cooperation; agents; service discovery; and client-server interoperability, must be investigated.

Exercises

2.1 What challenges does a distributed computing system have when servicing users who share information and peripheral resources? 2.1

2.2 What are the functions of the client-server model? What features does it have? 2.2.1 – 2.2.2

2.3 What is a chain of servers? Give examples. 2.3.1

2.4 Why are multiple servers necessary? 2.3.2

2.5 Describe the indirect client-server cooperation. 2.4.1

2.6 Describe the functions of each tier in the three-tier client-server architecture. 2.4.2

2.7 What methods are used for achieving service discovery? 2.5

2.8 What is client-server interoperability? 2.6

2.9 Describe the relationship between an application, the OS and the communication facility. 2.7