CLIENT–SERVER SYSTEMS

By amalgamating computers and networks into one single computing system, a distributed computing system has created the possibility of sharing information and peripheral resources. Furthermore, these systems improve 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 synthesis of a model of distributed computing has been influenced by a need to deal with the issues generated by distribution such as

- Locating and accessing remote data, programs, and peripheral resources
- 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
- Authenticating users

The model that has been used to develop application and system software of distributed computing systems is the client-server model. Because of this the current image of computing is client-server distributed computing.

The goal of this article is to introduce and discuss the client-server model and the communication paradigm which supports this model, and to show how this model has influenced the development of different systems and applications. This article contains three major parts. The first part introduces the client-server model and different concepts and extensions to this model. The second part discusses communication supporting distributed computing systems built based on the client-server model. It contains a detailed discussion of two dimensions of the communication paradigm: the communication pattern, one-to-one and group communication, and the techniques, message passing and remote procedure call (RPC), which are used to design and build client-server based applications. The third part presents advanced applications developed based on the client-server model. The first and simplest of the presented applications of the client-server model is the network file system (NFS). It is an extension to centralised (local) operating systems (e.g., Unix, MS-DOS) which allows transparent remote file access. Subsequent sections show the RHODOS distributed operating system and distributed computing environment (DCE), respectively. RHODOS has been built from scratch on top of a bare computer. It employs the concept of a microkernel which is a cornerstone of the whole client-server-based operating system. It provides full transparency to the user. On the other hand, DCE is built on top of existing operating systems such as Unix and VMS, and hides differences among individual computers. However, it does not fully support transparency.

THE CLIENT-SERVER MODEL

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 activity is called distributed computing.

The problem is how to formalize the development of distributed computing. 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.

The client-server model is a natural model of distributed computing, which is able to deal with the problems generated by distribution, could be used to describe these processes and their behavior when providing services to users, and allows the 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.

The client-server model and the association between this model and the physical environment this model is used in are illustrated in Fig. 1. The basic items of the model are the client and server and request and response, and the elements of a distributed computing system are distinguished. This figure firstly shows that the user must send a request to an individual server in order to be provided with a given service. A need for another service requires the user to send a request to another server. Secondly, 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 between these two processes are sent. This implies that operating systems of computers and the communication system of a distributed computing system are actively involved in the service provision.

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 organizing and integrating a group of computer operations into a separate ser-

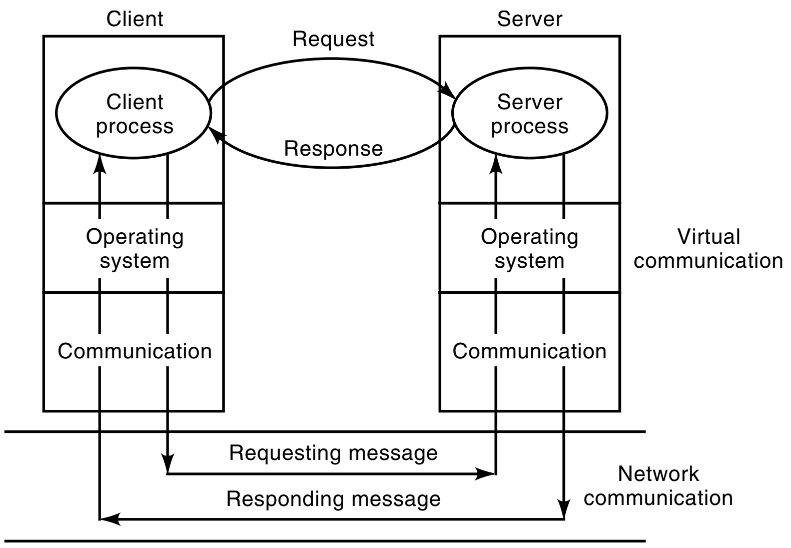

Figure 1. The client–server model and its association with operating systems and a communication facility.

vice. Also, any set of data with operations on this data can be organized 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.

From the user's point of view a distributed computing system can provide services such as: printing, electronic mail, file service, authentication, naming, database service, and computing service. These services are provided by appropriate servers.

Cooperation between Clients and Servers in a Distributed Computing System

A system where there is only one server would not be able to provide high performance and reliable and cost-effective services to the user. As was shown in the previous section, one server is used to provide services to more than one client. The simplest form of 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 which is based on this form of cooperation is a printing service and file service.

Processes can act as either clients or servers. It depends on the context. A file server which 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 it sends a request to an authentication server and waits for a response. Its response to the client depends on a response from the authentication server; the file server acts as a client of the authentication server. Thus, a service provided to the user by a distributed computing system developed based on the client-server model can require a chain of cooperating servers.

Distributed computing systems provide the opportunity to improve performance through parallel execution of programs on a network (sometimes called clusters) of workstations, and decrease the response time of databases through data replication. Furthermore, they can be used to support synchronous distant meetings and cooperative workgroups. They also can increase reliability by service multiplication.

In these cases many servers must contribute to the overall application. Furthermore, it would require in some cases simultaneous requests to be sent to a number of servers. Different application will require different semantics for the cooperation between clients and servers.

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, client and server cooperation can be strongly influenced and supported by some active entities which are extensions to the client-server model.

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

Groups in Distributed Computing Systems

A group is a collection of processes, in particular servers, which share common features (described by a set of attributes) or application semantics. In general, processes are grouped in order to deal with this set of processes as a single abstraction; form a set of servers which can provide an identical service (but not necessary of the same quality); encapsulate the internal state and hide interactions among group members from the clients and provide a uniform interface to the external world; and deliver a single message to multiple receivers thereby reducing the sender and receiving overheads (1).

There are two types of groups: closed and open (2). In a closed group only the members of the group can send and receive messages to access the resources of the group. In an open group not only can the members of the group exchange messages and request services but nonmembers of the group can send messages to group members. Importantly, the nonmembers of the group need not join the group nor have any knowledge that the requested service is provided by a group.

Four group structures are often supported to provide the most appropriate policy for a wide range of user applications.

The peer group is composed of a set of member processes that cooperate for a particular purpose. Fault-tolerant and load-sharing applications dominate this type of group style. The client-server group is made from a potentially large number of client processes with a peer group of server processes. It is an open group. The diffusion group is a special case of the client-server group, where a single request message is sent by a client process to all servers. The hierarchical group is an extension to the client–server group. In large applications with a need for sharing between large numbers of group members, it is important to localize interactions within smaller clusters of components in an effort to increase performance.

According to external behavior, groups can be classified into two major categories: deterministic and nondeterministic. A group is considered deterministic if each member must receive and act on a request. This requires coordination and synchronization between the members of the group. In a deterministic group, all members are considered equivalent. Nondeterministic groups assume their applications do not require consistency in group state and behavior, and they relax the deterministic coordination and synchronization. Each group member is not equivalent and can provide a different response to a group request, or not respond at all, depending on the individual group member's state and function.

In order to act properly and efficiently, each member of the group must exchange messages amongst themselves (above normal application messages) to resolve the current status and membership of the group. Any change in group membership will require all members to be notified to satisfy the requested message requirements. Furthermore, users are provided with primitives to support group membership discovery (3) and group association operations (4). Group membership discovery allows a process to determine the state of the group and its membership. However, as the requesting process has no knowledge of the group members location, a network broadcast is required.

There are four operations to support group association: *create, destroy, join,* and *leave.* Initially a process requiring group communication creates the required group. A process is considered to be a group member after it has successfully issued a group join primitive, and will remain a member of the group until the process issues a leave group primitive. When the last member of the group leaves, the group will be destroyed.

Extensions to the Client-Server Model

A client and server can cooperate either directly or indirectly. In the former case there is no additional entity which 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.

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.

As was presented previously, a client can invoke desired servers explicitly by sending direct requests to these servers. In this case the programmer of a user application must concentrate on both 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.

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 by cooperating, based on information provided in the request, with other servers. After completion of necessary operations by involved servers, the general 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 one-to-one message protocol. Although this communication pattern 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 improves this system even further.

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 three components: user interface and presentation processing component, responsible for accepting inputs and presenting the results (the client tier); computational function processing component, responsible for providing transparent, reliable, secure, and efficient distributed computing—it is also responsible for performing necessary processing to solve a particular application problem (application tier); data access processing component, responsible for accessing data stored on external storage devices, such as disk drives (back-end tier).

These components can be combined and distributed in various ways to create different configurations with varying complexity. Figure 2(a) shows a centralized configuration where all the three types of components are located in a single computer. Figure 2(b) shows three two-tier configurations where the three types of components are distributed on two computers. Figure 2(c) shows a three-tier configuration where all the three types of components are distributed on different computers.

Figure 3 illustrates an example 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 is computers that run computational function processing software. The bottom tier is back-

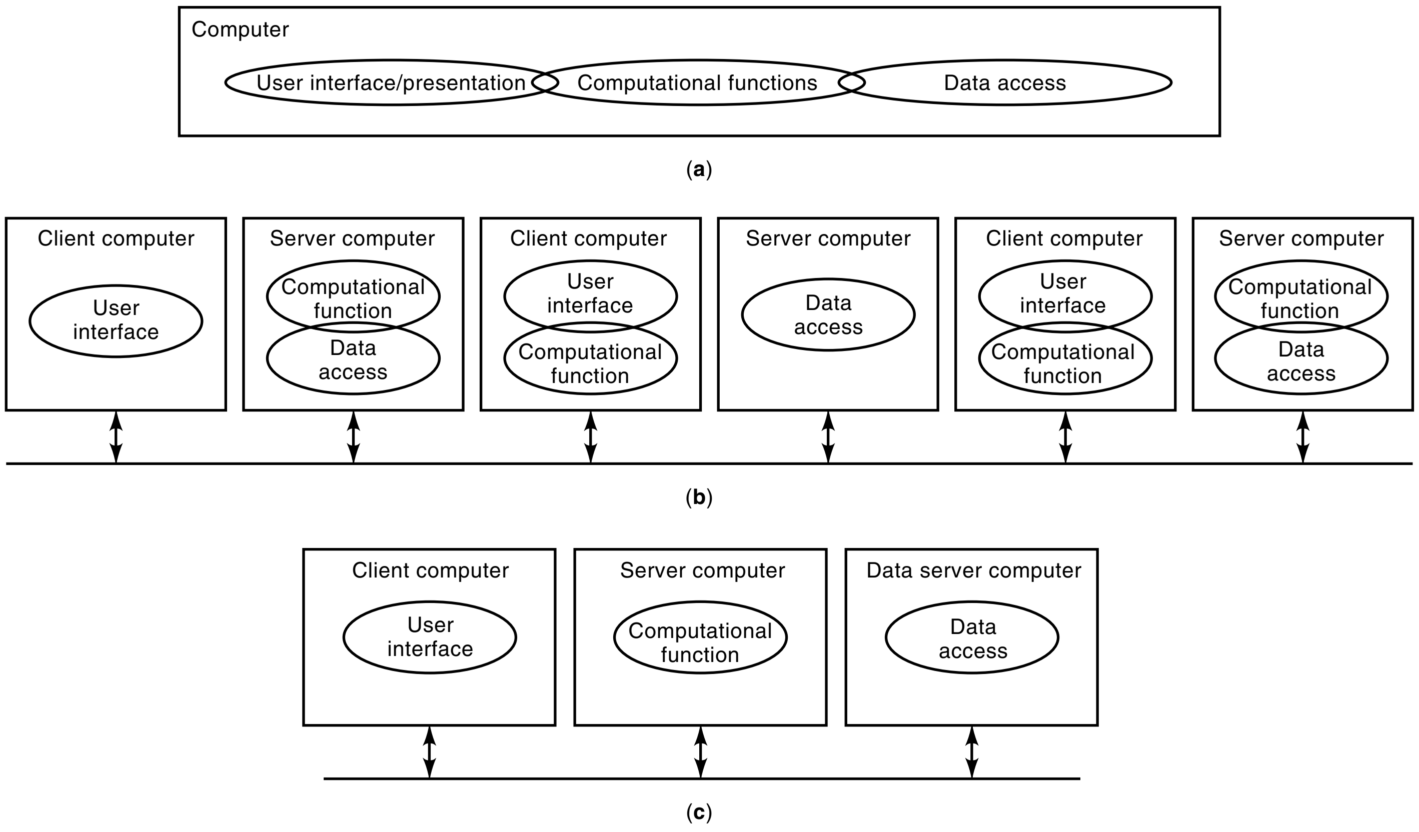

Figure 2. One- (a), two- (b), and three-tier (c) client–server configurations.

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 demonstrates: (1) better transparency, since the servers within the application tier allow an application to detach the user interface from back-end resources, and (2) better scalability, since servers as individual entities can be easily modified, added, or removed.

Service Discovery

To invoke a desired service a client must know whether there is a server which is able to provide this service, its characteristics, 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 the existence of a desired server—information about all available servers is available *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 the potential for 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 color printer of high quality) is available at a given time, and if so its name and location. Service discovery is achieved through the following approaches.

Computer Address Is Hardwired into Client Code. This approach requires the location of the server, in the form of a 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.

Another version of this approach is based on a more advanced naming system, where requests are sent to processes rather than to computers. In this case each process is located using a pair ⟨computer_address, process_name⟩. A client is provided with not only the 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.

Broadcast Is Used to Locate Servers. According to this approach each process has a unique name. In order to send a request a client must know the name of the server. However, this is not enough because the operating system of the computer where the server runs must know the 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 which finds the server's name in the list of its processes 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

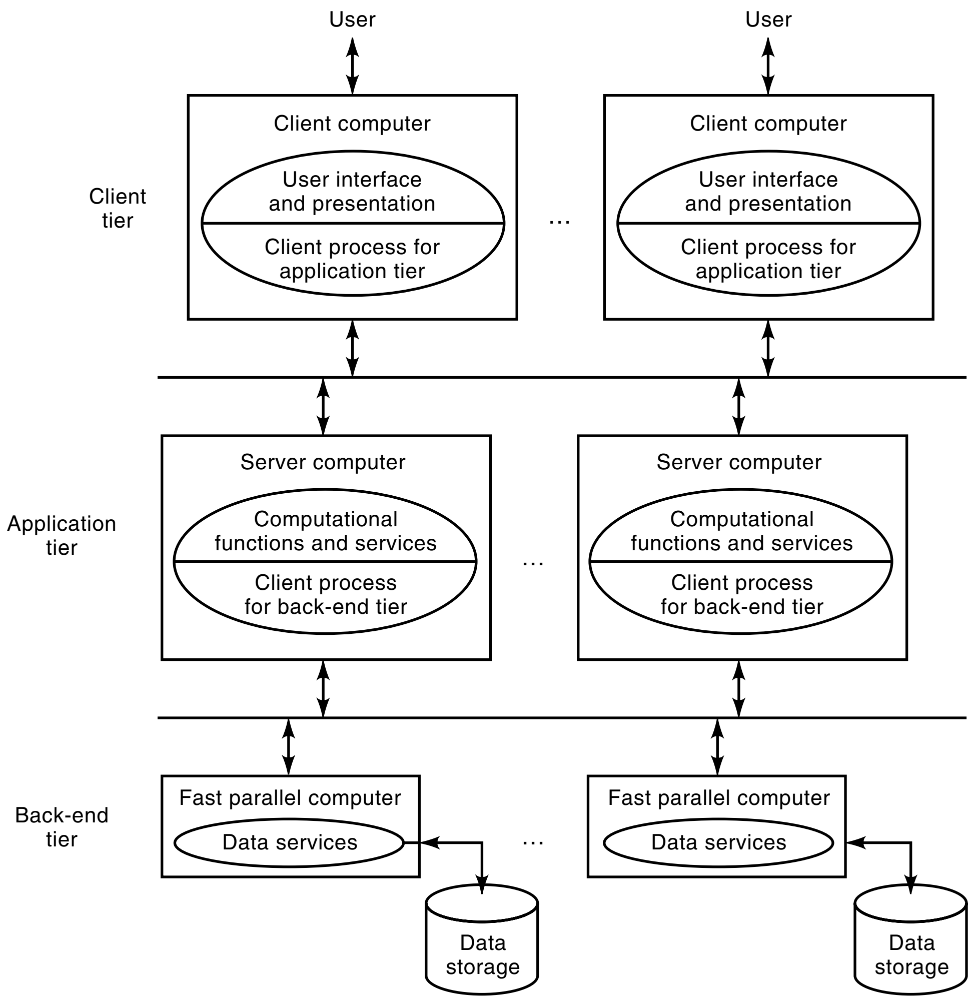
User
User
Client tier
Client computer
User interface and presentation
Client process for application tier
Client computer
User interface and presentation
Client process for application tier
Application tier
Server computer
Computational functions and services
Client process for back-end tier
Server computer
Computational functions and services
Client process for back-end tier
Back-end tier
Fast parallel computer
Data services
Fast parallel computer
Data services
Data storage
Data storage

Figure 3. An application of the three-tier architecture in a distributed computing system.

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 location request.

Server Location Lookup Is Performed via a Name Server. 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 computers. 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 centralized, the overall performance of a distributed computing system could be degraded as the name server can become a bottleneck. Furthermore, the reliability of this approach is low; if a name server computer crashed a distributed computing system cannot work.

In a large distributed computing system there could be a large number of servers. Moreover, servers of the same type can be characterized by different attributes describing the services they provide (e.g., one laser printer is a color 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.

A Broker Is Employed. This 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. A broker is a server which (1) allows a client to identify available servers which can be characterized by a set of attributes which describe the properties of a desired service; (2) mediates cooperation between clients and servers; (3) allows service providers to register the services they support by providing their names, locations, and features in the form of attributes; (4) advertises registered services and makes them available to clients; and (5) withdraws services dynamically. Thus, a broker is a server which embodies both service management and naming services.

Client–Server Interoperability

Reusability of servers is a critical issue for both users and software manufactures due to the high cost of software writing. This issue can be easily resolved in a homogeneous environment because the 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 (5) 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 (5). 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 standardization and bridging. The objective of the former is to map client and server interfaces to a common representation. The advantages of this mechanism are: (1) it separates communication models of clients from those of servers, and (2) it provides scalability, since it only requires $m + n$ maps, where m and n are the number of clients and servers, respectively. The disadvantage of this mechanism is that it is closed. The objective of the latter is to provide a two-way map between client and server. The advantages of this mechanism are: (1) openness, and (2) 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 standardization mechanism, as it requires $m \times n$ maps.

Conclusions

In this section 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 of achieving 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, there is a need to employ an operating system and communication network which links computers on which these processes run. Furthermore, in order to locate a server, the operating system must be involved. The question is what class of an operating system can be used. There are two classes of operating systems which could be employed to develop a distributed computing system: a network operating system and a distributed operating system. A network operating system is constructed by adding a module to the local centralized 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. A distributed operating system is built from scratch, 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.

COMMUNICATION BETWEEN CLIENTS AND SERVERS

Distributed computing systems must be fast in order to instil in users the feeling of a huge powerful computer sitting on their desks. This implies that communication between the clients and servers must be fast. Furthermore, the speed of communication between remote client and server processes should not be highly different from the speed between local processes. The issue is how to build a communication facility within a distributed computing system to achieve high communication performance.

One of the strongest factors which influences the performance of a communication facility is the communication paradigm: that is, the communication model supporting cooperation between clients and servers and the operating system support provided to deal with the cooperation.

There are two issues in the communication paradigm. Firstly, a client can send a request to either a single server or a group of servers. This leads to two patterns of communication: one-to-one and one-to-many, also called group communication (which are operating system abstractions). Secondly, these two patterns of interprocess communication could be developed based on two different techniques: message passing, adopted for distributed computing systems in the late 1970s; and remote procedure call (RPC), adopted for distributed computing systems in mid-1980s. These two techniques are supported by two respective sets of primitives provided by an operating system. Furthermore, communication between processes on different computers can be given the same format as communication between processes on a single computer.

The following topics are discussed in this section: message passing, including communication primitives; semantics of these primitives; direct and indirect communication; blocking and nonblocking primitives; buffered and unbuffered exchange of messages; and reliable and unreliable primitives are considered. Also, RPC is discussed. The basic features of this technique; parameters, results and their marshalling; client-server binding; and reliability issues are presented. Thirdly, group communication is discussed. In particular, the basic concepts of this communication pattern; group structures; different types of groups; group membership; message delivery and response semantics; and message ordering in group communication are presented.

Message Passing—Message-Oriented Communication

We define message-oriented communication as a form of communication in which the user is explicitly aware of the message used in communication and the mechanisms used to deliver and receive messages (6).

Basic Message Passing Primitives. A message is sent and received by executing the following two primitives:

send(dest, src, buffer). The execution of this primitive sends the message stored in *buffer* to a server process named *dest.* The message contains a name of a client process named *src* to be used by the server to send a response back.

receive(client, buffer). The execution of this primitive causes the receiving server process to be blocked until a message arrives. The server process specifies the *client* name of a process from whom a message is desired, and provides a *buffer* to store an incoming message.

It is obvious that the *receive* primitive must be issued before a message arrives; otherwise the request could be declared as lost and must be retransmitted by the client. Of course, when the server process sends any message to the client process, it must use these two primitives also; the server sends a message by executing the primitive *send* and the client receives it by executing the primitive *receive.*

There are several points that should be discussed at this stage. All of them are connected with a problem stated as follows: What semantics should these primitives have. The following alternatives are presented: direct or indirect communication via ports; blocking versus nonblocking primitives; buffered versus unbuffered primitives; reliable versus unreliable primitives; and structured forms of message passing based primitives.

Direct and Indirect Communication via Ports. A very basic issue in message-based communication is where do messages go. Message communication between processes uses one of two techniques: the sender designates either a fixed destination process or a fixed location for receipt of a message. The former technique is called direct communication—it uses direct names; the latter is called indirect communication and it exploits the concept of a port.

In direct communication, each process that wants to send or receive a message must explicitly name the recipient or sender of the communication. In this case, the send and receive primitives have the following form: *send(dest, src, buffer), receive(client, buffer).* The *dest* and *client* are the names of a destination process (server) and sending process (client) from whom the server is prepared to receive a request. This scheme exhibits a symmetry in naming: that is, both the sender and the receiver have to name one another in order to communicate. A variant of this scheme employs asymmetry in naming: only the client names the server, whereas the server is not required to name the client.

Direct communication is easy to implement and to use. It enables a process to control the times at which it receives messages from each process. The disadvantage of the symmetric and asymmetric schemes is the limited modularity of the resulting process definition. Changing the name of the process may necessitate the examination of all other process' destination. All references to the old process must be found, in order to modify them to the new name. This is not desirable from the point of view of separate compilation. Moreover, the *receive* primitive in a server should allow receipt of a message from any client to provide a service to whatever client process calls it.

Direct communication does not allow more than one client. Similarly, direct communication does not make it possible to send one request to more than one identical server. This implies the need for a more sophisticated technique. Such a technique is based on ports.

A port can be abstractly viewed as a protected kernel object into which messages may be placed by processes and from which messages can be removed: that is, the messages are sent to and received from ports. Processes may have ownership, send, and receive rights on a port. Each port has a unique identification (name) that distinguishes it. A process may communicate with other process by a number of different ports. In this case *dest* in the send primitive is a name of a port of the server the request is sent to.

Logically associated with each port is a FIFO queue of finite length. Messages which have been sent to this port but which have not yet been removed from it by a process reside on this queue. Messages may be added to this queue by any process which can refer to the port via a local name (e.g., capability). A port should be declared. A port declaration serves to define a queuing point for messages. A process which wants to remove a message from a port must have the appropriate receive rights. Usually, only one process may have receive access to a port at a time. Messages sent to a port are normally queued in FIFO order. However, an emergency message can be sent to a port and receive special treatment with regard to queuing.

Blocking versus Nonblocking Primitives. One of the most important properties of message passing primitives concerns whether their execution could cause delay. We distinguish blocking and nonblocking primitives. We say that a primitive has nonblocking semantics if its execution never delays its invoker; otherwise, a primitive is said to be blocking. In the former case, a message must be buffered. The previously described primitives have blocking semantics.

It is necessary to distinguish two different forms of the blocking *send* primitives. These forms are generated by different criteria. The first criterion reflects the operating system design and addresses buffer management and message transmission. The blocking and nonblocking send primitives are illustrated in Fig. 4. If the blocking send primitive is used, the sending process (client) is blocked: that is, the instruction following the send primitive is not executed until the message has been completely sent. The blocking receive implies that the process which issued this primitive remains blocked (suspended) until a message arrives, and being put into the buffer specified in the *receive* primitive. If the nonblocking send primitive is used, the sending process (client) is only blocked for the period of copying a message into the kernel buffer. This means that the instruction following the *send* primitive can be executed even before the message is sent. This can lead toward parallel execution of a process and message transmission.

The second criterion reflects the client-server cooperation and the programming language approach to dealing with message communication. In this case the client is blocked until the server (receiver) has accepted the request message and

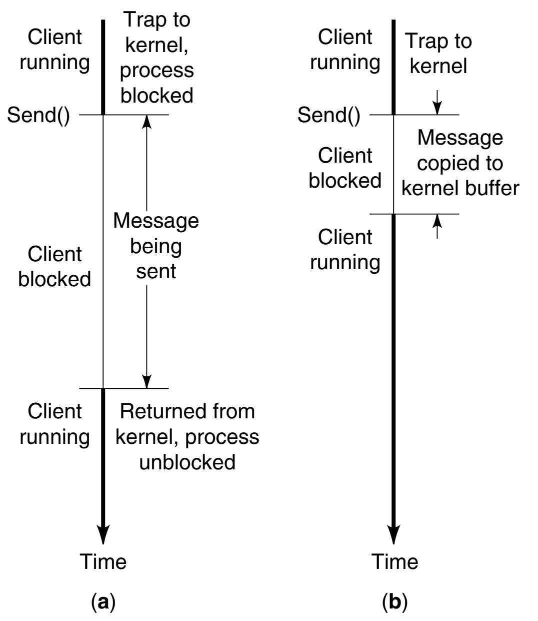

Figure 4. Operating system oriented blocking (a) and unblocking (b) *send* primitives.

the result or acknowledgment has been received by the client, illustrated in Fig. 5.

There are three forms of the *receive* primitive. The blocking receive is the most common, since the receiving process often has nothing else to do while awaiting receipt of a message. There is also a nonblocking *receive* primitive, and a primitive for checking whether a message is available to receive. As a result, a process can receive all messages and then select one to process.

Unbuffered versus Buffered Message Passing Primitives. In some message-based communication systems, messages are buffered between the time they are sent by a client and received by a server. If a buffer is full when a *send* is executed,

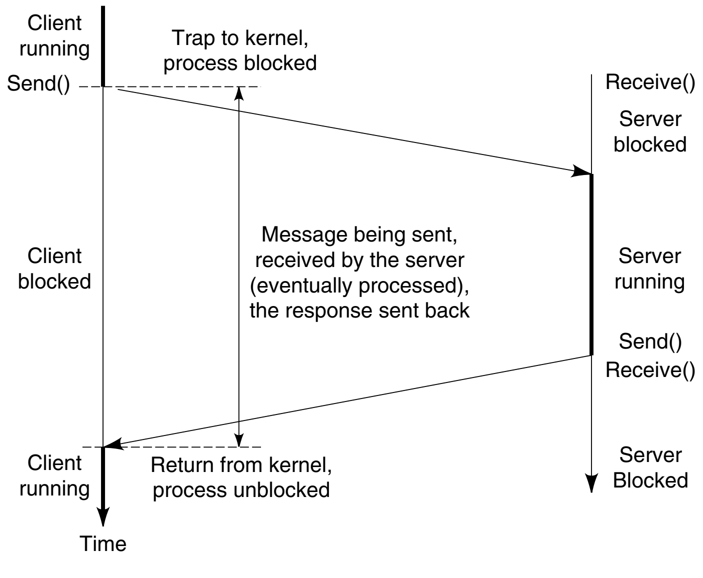
Client running
Trap to kernel, process blocked
Send()
Receive()
Server blocked
Client blocked
Message being sent, received by the server (eventually processed), the response sent back
Server running
Send()
Receive()
Client running
Return from kernel, process unblocked
Server Blocked
Time

Figure 5. Client–server cooperation oriented blocked *send* primitive.

there are two possible solutions: the *send* may delay until there is a space in the buffer for the message, or the *send* might return a code to the client, indicating that because the buffer is full the message could not be sent.

The situation of the receiving server is different. The *receive* primitive informs an operating system about a buffer into which the server wishes to put an arrived message. The problem occurs when the *receive* primitive is issued after the message arrives. The question is what to do with the message. The first possible approach is to discard the message. The client could time out and re-send, and hopefully the *receive* primitive will be invoked in the meantime. Otherwise, the client can give up. The second approach to deal with this problem is to buffer the message in the operating system area for a specified period of time. If during this period the appropriate *receive* primitive is invoked the message is copied to the invoking server space. If the *receive* primitive is not invoked and the timeout expires the message is discarded.

Buffered message passing systems are more complex than unbuffered message passing based systems, since they require creation, destruction, and management of the buffers. Also, they generate protection problems and cause catastrophic event problems when a process owning a port dies or is killed.

Unreliable versus Reliable Primitives. Different catastrophic events, such as a computer crash or a communication system failure can happen in a distributed computing system. These can cause either a requesting message being lost in the network, a response message being lost or delayed in transit, or the responding computer "dying" or becoming unreachable. Moreover, messages can be duplicated, or delivered out of order. The primitives discussed previously cannot cope with these problems. These are called unreliable primitives. The unreliable primitive *send* merely puts a message on the network. There is no guarantee of delivery provided and no automatic retransmission is carried out by the operating system when a message is lost.

Dealing with failure requires providing reliable primitives. In a reliable interprocess communication, the *send* primitive handles lost messages using internal retransmissions and acknowledgments on the basis of timeouts. This implies that when *send* terminates, the process is sure that the message was received and acknowledged.

Reliable and unreliable *receive* differ in that the former automatically sends an acknowledgment confirming message reception, whereas the latter does not. Two-way communication requires the utilization of the basic message passing primitives in a symmetrical way. If the client requested any data, the server sends reply messages (responses) using the *send* primitive. For this reason the client has to set the *receive* primitive up to receive any message from the server. Reliable and unreliable primitives are contrasted in Fig. 6.

Structured Forms of Message Passing Based Communication. A structured form of communication using message passing is achieved by distinguishing requests and replies and providing for bidirectional information flow. This means that the client sends a request message and waits for a response. The set of primitives is as follows.

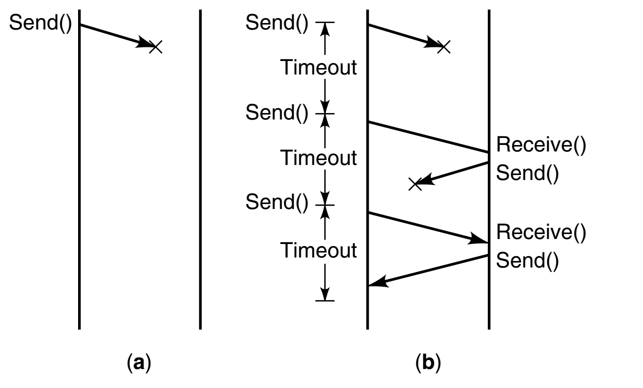

Figure 6. Unreliable (a) and reliable (b) message passing primitives.

send(dest, src, buffer). Sends a request and gets a response; it combines the previous client's *send* to the server with a *receive* to get the server's response.

get_request(client, buffer). Done by the receiver (server) to acquire a message containing work for them to do.

send_response(src, dest, buffer). The receiver (server) uses this primitive to send a reply after completion of the work.

It should be emphasised that the semantics, described in the previous sections, can be linked with these primitives. The result of the *send* and *receive* combination in the structured form of the *send* primitive is one operation performed by the interprocess communication system. This implies that rescheduling overhead is reduced, buffering is simplified (because request data can be left in a client's buffer, and the response data can be stored directly in this buffer), and the transport-level protocol is simplified.

Remote Procedure Call

Message passing between remote and local processes is visible to the programmer. It is a completely untyped technique. Programming message passing based applications is difficult and error prone. An answer to these problems is the RPC technique which is based on the fundamental linguistic concept known as the procedure call. The very general term remote procedure call means a type-checked mechanism that permits a language-level call on one computer to be automatically turned into a corresponding language-level call on another computer. The first and most complete description of the RPC concept was presented in Ref. 7.

Basic Features of Remote Procedure Calls. The idea of remote procedure calls (RPC) is very simple and is based on the observation that a client sends a request and then blocks until a remote server sends a response. This approach is very similar to a well-known and well-understood mechanism referred to as a procedure call. Thus, the goal of a remote procedure call is to allow distributed programs to be written in the same style as conventional programs for centralized computer systems. This implies that RPC must be transparent. This leads to one of the main advantages of this communication approach: the programmer does not have to know that the called procedure is executing on a local or a remote computer.

When remote procedure calls are used a client interacts with a server by means of a call statement

service_name (*value_args, result_args*)

To illustrate that both local and remote procedure calls look identical to the programmer, suppose that a client program requires some data from a file. For this purpose there is a *read* primitive in the program code.

In a system supported by a classical procedure call, the *read* routine from the library is inserted into the program. This procedure, when executing, puts the parameters into registers, and then traps to the kernel as a result of issuing a READ system call. From the programmer point of view there is nothing special; the *read* procedure is called by pushing the parameters onto the stack and is executed.

In a system supported by RPC (Fig. 7), the *read* routine is a remote procedure which runs on a server computer. In this case, another *call* procedure called a client stub from the library is inserted into the program. When executing, it also traps to the kernel. However, rather than placing the parameters into registers, it packs them into a message and issues the *send* primitive, which forces the operating system to send it to the server. Next, it calls the *receive* primitive and blocks itself until the response comes back.

The server's operating system passes the arrived message to a server stub, which is bound to the server. The stub is blocked waiting for messages as a result of issuing the *receive* primitive. The parameters are unpacked from the received message and a procedure is called in a conventional manner. Thus, the parameters and return address are on the stack, and the server does not see that the original call was made on a remote client computer. The server executes the procedure call and returns the results to the virtual caller: that is, the server stub. The stub packs them into a message and issues *send* to return the results. The stub comes back to the beginning of its loop to issue the *receive* primitive, and blocks waiting for the next request message.

The result message on the client computer is copied to the client process (practically to the stub's part of the client) buffer. The message is unpacked, and the results are extracted and copied to the client in a conventional manner. As a result of calling *read,* the client process finds its data available. The client does not know that the procedure was executing remotely.

It is evident that the semantics of remote procedure calls is analogous to local procedure calls: the client is suspended when waiting for results; the client can pass arguments to the remote procedure; and the called procedure can return results. However, since the client's and server's processes are on different computers (with disjoint address spaces), the remote procedure has no access to data and variables of the client's environment.

There is a difference between message passing and remote procedure calls. Whereas in message passing all required values must be explicitly assigned into the fields of a message before transmission, the remote procedure call provides marshalling of the parameters for message transmission: that is, the list of parameters is collected together by the system to form a message.

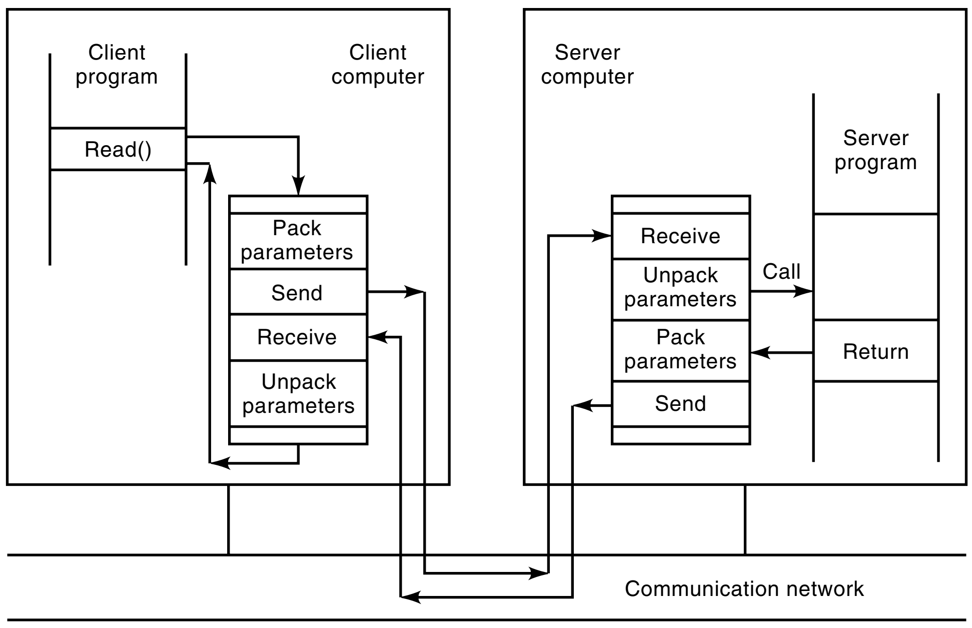

Figure 7. The sequence of operations in RPC.

Parameters and Results in RPCs. One of the most important problems of the remote procedure call is parameter passing and the representation of parameters and results in messages. Parameters can be passed by value or by reference. By-value message systems require that message data be physically copied. Thus, passing value parameters over the network is easy: the stub copies parameters into a message and transmits it. If the semantics of communication primitives allow the client to be suspended until the message has been received, only one copy operation is necessary. Asynchronous message semantics often require that all message data be copied twice: once into a kernel buffer and again into the address space of the receiving process. Data copying costs can dominate the performance of by-value message systems. Moreover, by-value message systems often limit the maximum size of a message, forcing large data transfers to be performed in several message operations reducing performance.

Passing reference parameters (pointers) over a network is more complicated. In general, passing data by-reference requires sharing of memory. Processes may share access to either specific memory areas or entire address spaces. As a result, messages are used only for synchronization and to transfer small amounts of data, such as pointers to shared memory. The main advantage of passing data by-reference is that it is cheap—large messages need not be copied more than once. The disadvantages of this method are that the programming task becomes more difficult, and it requires a combination of virtual memory management and interprocess communication, in the form of distributed shared memory.

Marshalling Parameters and Results. Remote procedure calls require the transfer of language-level data structures between two computers involved in the call. This is generally performed by packing the data into a network buffer on one computer and unpacking it at the other site. This operation is called marshalling.

More precisely, marshalling is the process (performed when sending the request as well as when sending the result back) in which three actions can be distinguished:

Extracting the parameters to be passed to the remote procedure and the results of executing the procedure;

Assembling these two into a form suitable for transmission among computers involved in the remote procedure call; and

Disassembling them on arrival.

The marshalling process must reflect the data structures of the language. Primitive types, structured types, and user-defined types must be considered. In the majority of cases, marshalling procedures for scalar data types and procedures to marshal structured types built from the scalar ones are provided as a part of the RPC software.

Client-Server Binding. Usually, RPC hides all details of locating servers from clients. However, as we stated in a previous section, in a system with more than one server (e.g., file server, print server), the knowledge of location of clients' files or a special type of a printer is important. This implies the need for a mechanism to bind a client and a server, in particular, to bind an RPC stub to the right server and remote procedure. There are two aspects of binding: the way the client specifies what it wants to be bound to (this is the problem of naming), and the ways the client locates the server and the specification of the procedure to be invoked (this is the problem of addressing).

In a distributed computing system there are two different forms of cooperation between clients and servers. The first form assumes that a client requests a temporary service. Another situation is generated by a client which wants to arrange for a number of calls to be directed to a particular serving process. These imply a need for a run-time mechanism for establishing long-term bindings between this client and a server.

In the case of requests for a temporary service, the problem can be solved using broadcast and multicast messages to locate a server. In the case of a solution based on a name server, that solution is not enough, because the process wants to call the located server during a time horizon. This means that a special binding table should be created containing established long-term binding objects (i.e., a client name and a server name), should be registered. The RPC run-time procedure for performing remote calls expects to be provided a binding object as one of its arguments. This procedure directs the call to the binding address received. It should be possible to add new binding objects to the table, remove binding objects from the binding table (which in practice means breaking a binding), and update the binding table. In systems with name servers, broadcasting is replaced by the operation of sending requests to a name server requesting a location of a given server and sending a response with an address of this server. Binding can take place at compile time, link time, or call time.

Error Recovery Issues. Because the client and server are separate processes which run on separate computers, they are prone to failures of themselves, their computers, or the communication system. The remote procedure may not be complete successfully. For example, the result message is not returned to the client as a response to its call message, because one of four events may occur: the request message is lost; the result (response) message is lost; the server computer crashes and is restarted; and the client computer crashes and is restarted. These events form the basis for design of RPC recovery mechanisms.

Three different semantics of RPC and their mechanisms can be identified to deal with problems generated by these four events:

Maybe call semantics. Timeouts are used to prevent a client waiting indefinitely for a response message;

At-least-once call semantics. This mechanism usually includes timeouts and a call retransmission procedure. The client tries to call the remote procedure until it gets a response or can tell that the server has failed;

Exactly once call semantics. In the case of at-least-once call semantics it can happen that the call can be received by the server more than once, because of lost responses. This can have the wrong effect. To avoid this the server sends each time (when retransmitting) as its response the result of the first execution of the called procedure. Thus, the mechanisms for these semantics include, in addition to those used in at-least-once call semantics (i.e., timeouts, retransmissions), call identifications and the server's table of current calls. This table is used to store the calls received first time and procedure execution results for these calls.

Message Passing versus Remote Procedure Calls. A problem arises in deciding which of the two interprocess communication techniques presented is better, if any, and whether there are any suggestions for when, and for what systems, these facilities should be used.

First of all, the syntax and semantics of the remote procedure call are the functions of the programming language being used. On the other hand, choosing a precise syntax and semantics for message passing is more difficult than for RPC because there are no standards for messages. Moreover, neglecting language aspects of RPC and because of the variety of message passing semantics, these two facilities can look very similar. Examples of a message passing system that look like RPC are message passing for the V system (which in Ref. 8 is called now the remote procedure call system) and message passing for Amoeba (9) and RHODOS (10).

By comparing the remote procedure call and message passing, the former has the important advantage that the interface of a remote service can be easily documented as a set of procedures with certain parameter and result types. Moreover, from the interface specification, it is possible to automatically generate code that hides all of the details of messages from a programmer.

On the other hand, a message passing model provides flexibility not found in remote procedure call systems. However, this flexibility is at the cost of difficulty in the preparation of precisely documented behavior of a message passing interface.

The problem is when these facilities should be used. The message passing approach appears preferable when serialization of request handling is required. The RPC approach appears preferable when there are significant performance benefits to concurrent request handling. RPC is particularly efficient for request–response transactions.

Group Communication

Distributed computing systems provide the opportunity to improve the overall performance through parallel execution of programs on a network of workstations, decreasing the response time of databases using data replication, supporting synchronous distant meetings and cooperative workgroups, and increasing reliability by service multiplication. In these cases many servers must contribute to the overall application. This implies a need to invoke multiple services by sending a simultaneous request to a number of servers. This leads toward group communication.

The concept of a process group is not new. The V-system (11), Amoeba (2), Chorus (12), and RHODOS (10) all support this basic abstraction in providing process groups to applications and operating system services with the use of group communication.

Basic Concepts of Group Communication. Group communication is an operating system abstraction which supports the programmer by offering convenience and clarity. This operating system abstraction must be distinguished from the message transmission mechanisms such as multicast (one-to-many physical entities connected by a network) or its special case broadcast (one-to-all physical entities connected by a network).

A request is sent by a client called *src* to a group of servers providing the desired service named *group_name* by executing either *send(group_name, src, buffer)* when the message passing technique is used, or *call* service_name *(value_args, result_args)* when the RPC technique is used.

This request is delivered following the semantics of a primitive used. The primitives should be constructed such that there is no difference between invoking a single server or a group of servers. This means that communication pattern transparency is provided to the programmer.

Thus, groups should be named in the same manner that single processes are named. Each group is treated as one single entity; its internal structure and interactions are not shown to the users. The mapping of group names on multicast addresses is performed by an interprocess communication facility of an operating system and supported by a naming server. However, if multicast or even broadcast is not provided, group communication could be supported by one-to-one communication at the network level.

Communication groups are dynamic. This means that new groups can be created and some groups can be destroyed. A process can be a member of more than one group at the same time. It can leave a group or join another one.

In summary, group communication shares many design features with message passing and RPC. However, there are some issues which are very specific, and their knowledge could be of a great value to the application programmer.

Message Delivery Semantics. Message delivery semantics of a group relates to the successful delivery of a message to processes in a group. There are four choices of delivery semantics:

Single Delivery. Single delivery semantics require that only one of the current group members needs to receive the message for the group communication to be successful.

k-Delivery. In *k*-delivery semantics, at least *k* members of the current group will receive the message successfully.

Quorum Delivery. With quorum delivery semantics, a majority of the current group members will receive the message successfully.

Atomic Delivery. With atomic delivery all current members of the group successfully receive the message or none does. This delivery semantic is the most stringent as processes can and do fail and networks may also partition during the delivery process of the request messages, making some group members unreachable.

Message Response Semantics. By providing a wide range of message response semantics the application programmer is capable of providing flexible group communication to a wider range of applications. The message response semantics specify the number and type of expected message responses. There are five broad categories for response semantics:

No Responses. By providing no response to a delivered request message the group communication facility is only able to provide unreliable group communication.

Single Response. The client process expects (for successful delivery of a message) a single response from one member of the group.

k-Responses. The client process expects to obtain *k* responses for the delivered message from the members of the process group. By using *k* response semantics the groups resilience can be defined (13). The resilience of a group is based on the minimum number of processes that must receive and respond to a message.

Majority Response. The client process expects to receive a majority of responses from the current members of the process group.

Total Response. The client process requires all current members of the group to respond to the delivery of a request message.

Message Ordering in Group Communication. The semantics of message ordering are an important factor in providing good application performance and reduction in the complexity of distributed application programming. The order of message delivery to members of the group will dictate the type of group it is able to support.

There are four possible message ordering semantics:

No Ordering. This semantic implies that all request messages will be sent to the current group of processes in no apparent order.

FIFO Ordering. This semantic implies that all request messages transmitted in the first-in first-out (FIFO) order by a client process to the current members of the group will be delivered in the FIFO order.

Causal Ordering. The causal ordering semantic delivers request messages to all members of the current group such that the causal ordering of message delivery is preserved. This implies that if the sending of a message m′ causally follows the delivery of message m, then each process in the group receives m before m′.

Total Ordering. Total ordering semantic implies that all messages are reliably delivered in sequence to all current members of the group or no member will receive the message. Also, total ordered semantic guarantees that all group members see the same order of messages. Total order is more stringent that FIFO ordering as all message transfers between all current members of the group are in order. This implies that all processes within the current group perceive the same total ordering of messages. In causal ordering we are concerned with the relationship of two messages while in total ordering we are concerned with seeing the same order of messages for all group member processes.

Conclusions

In this section we described two issues of the communication paradigm for the client-server cooperation: firstly, the communication pattern, including one-to-one and one-to-many (group communication); secondly, two techniques, message-passing and RPC, which are used to develop distributed computing systems. The message passing technique allows clients and servers to exchange messages explicitly using the *send* and *receive* primitives. Various semantics, such as direct and indirect, blocking and nonblocking, buffered and unbuffered, reliable and unreliable can be used in message passing. The RPC technique allows clients to request services from servers by following a well-defined procedure call interface. Various issues are important in RPC, such as marshalling and unmarshalling of parameters and results, binding a client to a particular server, and handling exceptions.

SUN'S NETWORK FILE SYSTEM

The first major step in the development of distributed software was made when inexpensive diskless personal computers were connected by inexpensive local networks in order to share a file service or a printer service.

Distributed File Systems

A distributed file system is a key component of any distributed computing system. The main function of such a system is to create a common file system that can be shared by all the clients which run on autonomous computers in the distributed computing system. The common file system should store programs and data and make them available as needed. Since files can be stored anywhere in a distributed computing system, a distributed file system should provide location transparency.

To achieve such a goal a distributed file system usually follows the client-server model. A distributed file system typically provides two types of services: the file service and the directory service, which are implemented by the file server and the directory server, respectively, distributed over the network. These two servers can also be implemented as a single server. The file server provides operations on the contents of files such as read, write, and append. The directory server provides operations such as directory and file creation and deletion, for manipulating directories and file names. The client application program interface (client API, usually in the form of a process or a group of processes) runs on each client computer and provides a uniform user-level interface for accessing file servers. In this section we will present one of the most important achievement of the 1980s, which is still in use now, the Network File System, known as NFS, developed based on the client-server model.

NFS Architecture

NFS was developed by Sun Microsystems and introduced in late 1984 (14). Since then it has been widely used in both industry and academia. NFS was originally developed for use on Unix workstations. Currently, many manufacturers support it for other operating systems (e.g., MS-DOS). Here, NFS is introduced based on the Unix system. To understand the architecture of NFS, we need to define the following terms:

INODE. This is a data structure that represents either an open file or directory within the Unix file system. It is used to identify and locate a file or directory within the local file system.

RNODE. The remote file node is a data structure that represents either an open file or directory within a remote file system.

VNODE. The virtual file node is a data structure that represents either an open file or directory within the virtual file system (VFS).

VFS. The virtual file system is a data structure (linked lists of VNODEs) that contains all necessary information on a real file system that is managed by the NFS. Each VNODE associated with a given file system is included in a linked list attached to the VFS for that file system.

The NFS server integrates functions of both a file server and a directory server and the NFS clients use a uniform interface, the VFS/VNODE interface, to access the NFS server. The VFS/VNODE interface abstraction makes it possible to achieve the goal of supporting multiple file system types in a generic fashion. The VFS and VNODE data structures provide the linkage between the abstract uniform file system interface and the real file system (such as a Unix or MS-DOS file systems) that accesses the data. Further, the VFS/VNODE interface abstraction allows NFS to make remote files and local files appear identical to a client program.

In NFS, a client process transparently accesses files through the normal operating system interface. All operating system calls that manipulate files or file systems are modified to perform operations on VFSs/VNODEs. The VFS/VNODE interface hides the heterogeneity of underlying file systems and the location of these file systems. The steps of processing a user-level file system call can be described as follows (Fig. 8):

1. The user-level client process makes the file system call through the normal operating system interface.
2. The request is redirected to the VFS/VNODE interface. A VNODE is used to describe the file or directory accessed by the client process.
3. If the request is for accessing a file stored in the local file system, the INODE pointed by the VNODE is used. The INODE interface is used and the request is served by the Unix file system interface.
4. If the request is for accessing a file stored locally in other types of file systems (e.g., MS-DOS file system), a proper interface of that file system is used to serve the request.
5. If the request is for accessing a file stored remotely, the RNODE pointed to by the VNODE is used and the request is passed to the NFS client and some RPC messages are sent to the remote NFS server that stores the requested file.
6. The NFS server processes the request by using the VFS/VNODE interface to find the appropriate local file system to serve the request.

The Role of RPC

The communication between NFS clients and servers is implemented as a set of RPC procedures. The RPC interface provided by a NFS server includes operations for directory manipulation, file access, link manipulation, and file system access (15). The actual specifications for these remote procedures are defined in the RPC language, and the data structures used by the procedures are defined in the XDR format. The RPC language is a C-like language used as input into Sun's RPC Protocol Compiler utility. This utility can be used to output the actual C language source code.

NFS servers are designed to be stateless, meaning that there is no need to maintain information (such as whether a file is open or the position of the file pointer) about past requests. The client keeps track of all information required to send requests to the server. Therefore, NFS RPC requests are designed to completely describe the operation to be performed. Also, most NFS RPC requests are idempotent, meaning that an NFS client may send the same request one or more times without any harmful side effects. The net result of these duplicate requests is the same. NFS RPC requests are transported using the unreliable User Datagram Protocol (UDP). NFS servers notify clients when an RPC completes by sending the client an acknowledgment (also using UDP).

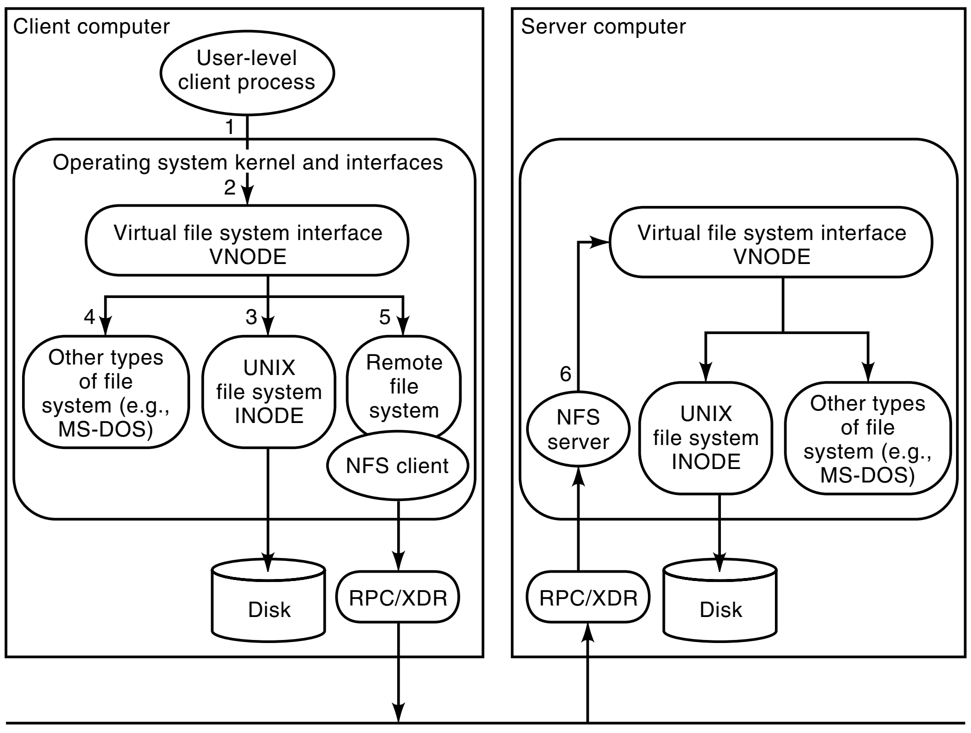
Client computer
User-level client process
1
Operating system kernel and interfaces
2
Virtual file system interface VNODE
4
3
5
Other types of file system (e.g., MS-DOS)
UNIX file system INODE
Remote file system
NFS client
Disk
RPC/XDR
Server computer
Virtual file system interface VNODE
6
NFS server
UNIX file system INODE
Other types of file system (e.g., MS-DOS)
RPC/XDR
Disk

Figure 8. The NFS structure. See text for description.

A NFS client sends its RPC requests to a NFS server one at a time. Although a client computer may have several NFS RPC requests in progress at any time, each of these requests must come from a different client. When a client makes an RPC request, it sets a timeout period during which the server must service and acknowledge it. If the server does not acknowledge during the timeout period, the client retransmits the request. This may happen if the request is lost along the way or if the server is too slow because of overloading. Since the RPC requests are idempotent, there is no harm if the server executes the same request twice. If the client gets a second acknowledgment from the request, the client simply discards it.

Conclusions

In this section we showed an application of the client-server model in the development of a distributed file system based on the Network File System. The NFS server integrates functions of both a file server and a directory server. It has been built as an extension module to a centralized operating system (e.g., Unix or MS-DOS). NFS clients use RPC to communicate with the NFS system. This system allows clients running on diskless computers to access and share files.

THE DEVELOPMENT OF THE RHODOS

The vast majority of design and implementation efforts in the area of distributed computing systems have concentrated on client-server-based applications running on centralized operating systems (e.g., Unix, VMS, OS/2). However, there have been huge research efforts on the development of operating systems built from scratch based on the client-server model (called distributed operating systems). These systems support distributed computing systems developed on a set of personal homogeneous computers connected by local or fast wide area networks.

The results achieved have changed, and are still changing, operating systems of distributed computing systems and the development of applications supported by these systems. The following systems have been developed based on the client-server model: V (8), Amoeba (2), Chorus (12), and RHODOS (16).

Distributed Operating Systems

A distributed operating system is one that looks to its users like a centralized operating system, but runs on multiple, independent computers connected by fast local or wide area networks. There are the following four major goals (the first three are the goals of a centralized operating system) of a distributed operating system:

- Hide details of hardware by creating abstractions: for example, software which provides a set of higher level functions which form a virtual computer;
- Manage resources to allow their use in the most effective way and support user processes in the most efficient way;
- Create a pleasant user computational environment; and
- Hide distribution of resources, information, peripheral and computational resources, in order to provide full transparency to users.

A generic architecture of a distributed operating system which allows these goals to be achieved has the following software levels. Software providing an abstraction sits on bare hardware, and allows the handling of interrupts and context switching. The second level of a distributed operating system is formed by software which manages physical resources such as processor time, memory, input/output, and virtual resources such as processes, remote communication, communication ports, and network protocols. It depends on the support provided by functions of the software abstraction level. The second level provides it services to the system services level. This third software level allows the management of files and object (services, resources) names, and creates a human user interface formed by graphics terminals, command interpreters and authentication systems. This level creates an image of a computer system for users. User processes form the software level sitting on the system services level.

In a client-server based distributed operating system all management functions and services provided to user processes are modelled and developed as individual cooperating server processes. User processes act as clients. However, because servers cooperate in order to achieve the goals of a distributed operating system they also act as clients. As physical memory in a distributed computing system is not shared, remote processes communicate using messages. In order to have a uniform communication model, local processes also communicate using messages. This provides communication transparency in a natural manner.

In this section we will use RHODOS to illustrate the application of the client-server model in the development of a new class of distributed operating systems. For this reason we will mainly concentrate on the kernel servers and microkernel as they form a new image of operating systems for distributed computing systems.

The RHODOS Architecture

RHODOS (research oriented distributed operating system) is a microkernel and message passing based system developed using the client-server model. This operating system is capable of supporting parallel processing on a network of workstations and providing load sharing and balancing in order to provide high-performance services to users (10).

There are three layers of cooperating processes in RHODOS: user processes, system servers, and kernel servers (Fig. 9). Each process executes in user mode and is confined to an individual address space which is controlled and maintained by the RHODOS microkernel.

In RHODOS, software creating abstractions forms a microkernel. The microkernel provides the following functions: context switching, interrupt handling, basic operations on memory pages relating to the hardware, and local interprocess communication. Furthermore, this microkernel is responsible for storing and managing basic data structures.

Kernel servers implement the mechanisms of the RHODOS functionality. Two groups of kernel servers can be distinguished. To the first group belong these servers which provide services which could be identified in any distributed or network operating system: process management, memory management, remote IPC management, communication protocols, and I/O management (drivers in RHODOS have also been developed as individual servers). The second group encompasses servers which provide advanced services which are necessary to support parallel processing on a network of workstations, and load sharing and balancing. These services are: process migration, remote process creation, and data collection.

System servers implement the policy of the RHODOS functionality. They provide services such as naming, file accessing and manipulation (in basic and transaction modes), two-way and m-way authentication, and global scheduling. A broker service has also been developed and will be installed shortly. In order to provide these services, system servers act as clients and invoke relevant kernel servers and the nucleus using standard system calls.

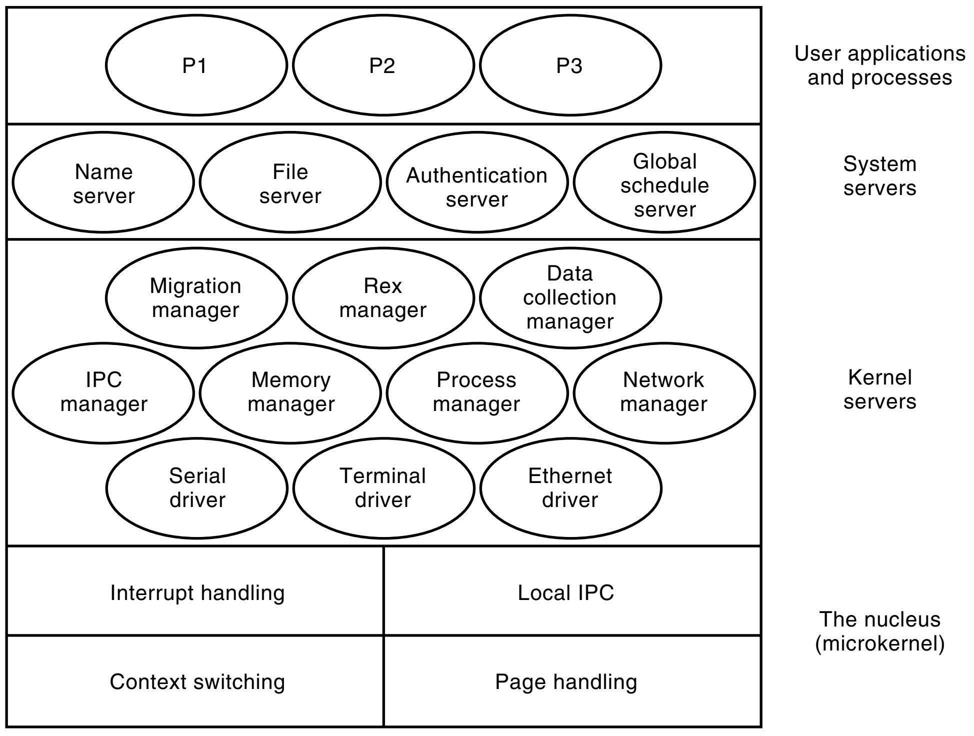

Figure 9. The logical architecture of RHODOS.

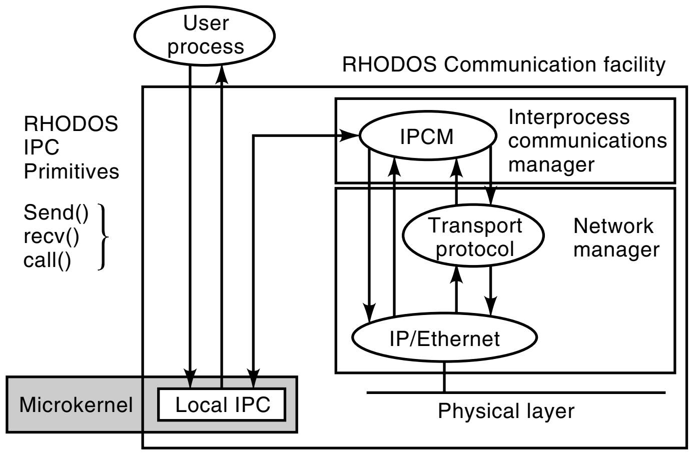

Figure 10. RHODOS interprocess communication facility.

User applications and processes are those developed and allocated to perform task for users. These processes have no special privileges and obtain services by calls to the microkernel and system servers.

Communication in RHODOS

In RHODOS, access to local and remote services is achieved in the same transparent manner, via a system name of that service and uniform interprocess communication, which is provided by the Interprocess Communication (IPC) facility. The facility provides three basic communication primitives: *send*(), *recv*(), and *call*(). Both *send*() and *recv*() provide the basic message passing semantics while the *call*(), *recv*(), and *send*() in combination provide synchronous RPC. In providing both message passing and RPC semantics the programmer is able to select the most appropriate communication technique for a given application.

The functioning of the IPC facility is divided into three sections: local IPC module, the IPC manager, and the network manager (Fig. 10). The local IPC module is an integral part of the RHODOS microkernel and provides local communication between processes on the same personal computer. If the destination process exists on the local computer, the module will complete the transfer. Otherwise, the IPC module sends a request to the IPC manager to provide a remote communication service.

The primary responsibility of the IPC Manager is the receiving and transmitting of remote messages for all processes within the RHODOS distributed computing system. It also supports group communication. This service is achieved with the cooperation of the name server by assigning a single name to a group of names. Furthermore, in order to support one-to-one and group communication the IPC manager is responsible for address resolution. In particular, a message that is sent to an individual process or a group requires the IPC manager to resolve the destination processes' (servers') location and provide the mechanism for the transport of the message to the desired process or group of processes.

In order to deliver a message to a remote process (server), the IPC manager invokes a delivery server, called the network manager. This server consists of a protocol stack employing transport, network, and data link layer protocols. Currently, the transport service is provided by a fast specialized RHODOS Reliable Datagram Protocol (RRDP). Network and data link layer protocols are provided by the IP/Ethernet suit.

RHODOS Kernel Servers and Services

One of the basic features of the RHODOS design is that each resource is managed by a relevant server: the process manager is responsible for processes and basic operations on processes, the space manager for memory, and the IPC manager for remote and group communication and address resolution. A process is a very special resource, because it is constructed based on some basic resources such as spaces, data structures usually called process control blocks, communication ports, and buffers. Thus, in RHODOS advanced operations on processes such as process migration and remote process creation are provided by separate servers: the migration manager and REX manager.

Process Manager. The job of the process manager is to manage the processes that are created in RHODOS. The process manager manipulates the process queues and deals with parent processes waiting for child processes to exit. It cooperates with other kernel servers, for instance with the migration manager to transfer a process' state during migration; and the remote execution manager to set up a process' state when a process is created.

Space Manager. One of the goals of RHODOS is portability across hardware platforms. Thus, RHODOS memory management has been separated into two sections: hardware dependent and hardware independent. The small hardware-dependent section is found in the microkernel and the larger hardware-independent section comprises RHODOS space manager. This server deals with spaces, logical units of memory, independent of physical units (e.g., pages), which are mapped to the physical memory.

The space manager supports two types of page operations: copy_on_write, which allows twin processes to share pages while they are reading them but makes separate copies when either process attempts to write to the page; copy_on_reference, which is used in process migration where only referenced pages are transferred from a source computer to a destination computer the process has been migrated.

Handling exceptions, creating spaces and transferring pages have been extended by adding additional functions in order to provide an operating system built in support for Distributed Shared Memory (DSM). Two consistency models are supported in the RHODOS DSM: invalidation and update based.

Device Manager. Transparency is an important feature of RHODOS. This not only includes interprocess communication between remote hosts, but also a transparent unified interface of physical devices such as serial ports, keyboards, video screens, and disks. Device drivers provide this interface. Device drivers in RHODOS are in their own right processes with the privilege and status of kernel servers. The benefits obtained from implementing device drivers as processes include the ability to enable and disable new drivers dynamically, as well as to use normal process debugging tools whilst the device driver is active. The device manager is the controlling entity that allows users to access a requested physical device.

Migration Manager. The process migration manager is responsible for the migration of running processes from the home computer to a remote computer. To migrate a process in RHODOS involves migrating the process state, address space, communication state, file state, and other resources. Thus, process migration requires the cooperation of all the servers managing these resources, the process, space, IPC managers, and the file server, respectively. The process migration manager only coordinates these servers, and all of them cooperate following the client-server model.

Process migration in RHODOS is a transaction-based operation performed on processes. Thus, the initial request from the source process migration manager to the destination process migration manager to migrate a selected process starts the transaction. The destination process migration manager commits this transaction by sending a response back, if all operations of installation of resources on the destination computer by individual servers, the process, space, IPC managers, and the file server have been completed successfully. Otherwise, an abort response is sent back.

Remote Execution Manager. The function of the remote execution (REX) manager is to provide coordination for creation of processes on local and remote computers. If a process is created on a local computer only a local REX manager is involved. If processes are created on remote computers, the home REX manager cooperates with remote REX managers to ensure processes are created correctly whilst maintaining the link with the process that issued the request. The generic cooperation of the servers is shown in Fig. 11.

Data Collection Manager. The RHODOS Data Collection System is responsible for collection and dissemination of the operational statistics of processes and exchanged messages in the RHODOS environment. The Data Collection System consists of a data collection manager (server) and stubs of code within the microkernel and other servers. The data collection manager is designed to be activated periodically and when special events occur (e.g., a new process was created, a process was killed), and provide a central repository for the accumulation of statistics. It provides accurate process statistics to the global scheduler. These statistics will permit the global scheduler to make the most appropriate decisions concerning process placement within the RHODOS environment.

RHODOS System Servers

The RHODOS system provides direct services to users by employing the following servers: the naming server, file server, authentication server, and the broker server, called the trader. Furthermore, RHODOS provides a special service which improves the overall performance of all services by em-

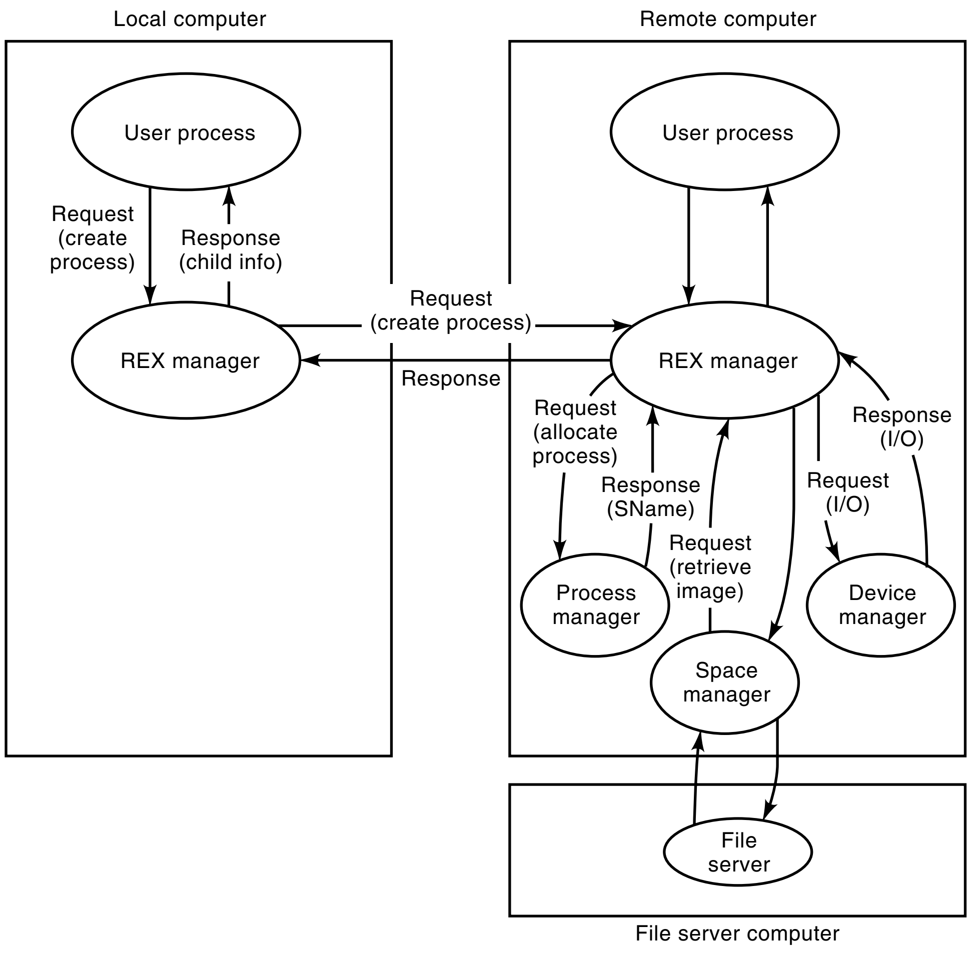

Figure 11. The generic cooperation of servers involved in local or remote process creation.

ploying the global scheduler. The utilization of the client-server model in the development of user-oriented services of a distributed operating system is presented here based on the global scheduler.

RHODOS provides global scheduling services in order to allocate/migrate processes to idle or lightly loaded computers to share computational resources and balance load. Global scheduling employs both static allocation and load balancing. Static allocation is employed when system load remains steadily high and new processes have to be created. Static allocation is making the decision of where to create new processes. Load balancing is employed to react to large fluctuations in system load. Load balancing is making a decision when to migrate a process, which process to migrate, and where to migrate this process. These servers make these decisions based on the information about the current load of the personal computers participating in global scheduling, their load trends, and the process communication pattern.

Conclusions

In this section we showed an application of the client-server model in the development of an advanced distributed operating system, RHODOS. RHODOS consists of a microkernel and two layers of cooperating servers, called kernel servers and system servers. Generally speaking, kernel servers implement the mechanism of the RHODOS functionality, whereas system servers implement the policy of the RHODOS functionality. User processes, sitting on top of the RHODOS software, obtain services from RHODOS servers. When a RHODOS server receives a service request, it may serve the request directly, or it may contact other servers if services from these servers are required.

BUILDING THE DISTRIBUTED COMPUTING ENVIRONMENT ON TOP OF EXISTING OPERATING SYSTEMS

The previous section contains a presentation of RHODOS, an example of a distributed operating system, developed based on the client-server model and the concept of a microkernel. The whole system has been built from scratch on bare hardware. There is another approach to building a distributed computing environment by putting it on top of existing operating systems. Such a software layer hides the differences among the individual computers, and forms a single computing system.

The Role of the Client-Server Model in Building a Distributed Computing Environment

Open Software Foundation's Distributed Computing Environment (DCE) (17) is a vendor-neutral platform for supporting distributed applications. DCE is a standard software structure for distributed computing that is designed to operate across a range of standard Unix, VMS, OS/2, and other operating systems. It includes standards for RPC, name, time, security, and thread services—all sufficient for client–server computing across heterogeneous architectures.

DCE uses the client–server model to support its infrastructure and transparent services. All DCE services are provided through servers. By using DCE, application programmers can avoid considerable work in creating supporting services, such as creating communication protocols for various parts of a distributed program, building a directory service for locating those pieces, and maintaining a service for providing security in their own program.

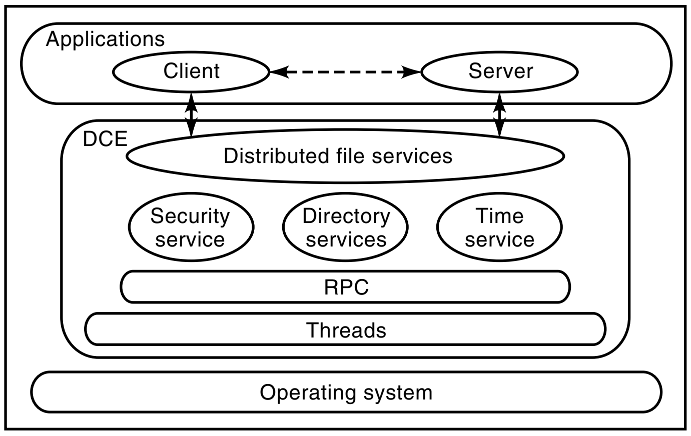

Figure 12. The logical architecture of DCE.

In the previous section we mainly addressed the kernel servers and microkernel of RHODOS as they are the result of the new approach based on the client-server model and the concept of a microkernel to building distributed computing systems. Here, since DCE is a complete extension of centralized operating systems to form a distributed computing system we mainly concentrate on servers which directly provide services to users.

The Architecture of DCE

The architecture of DCE masks the physical complexity of the networked environment by providing a layer of logical simplicity, composed of a set of services that can be used separately or in combination to form a comprehensive distributed computing system. Servers that provide DCE services usually run on different computers; so do clients and servers of a distributed application program that use DCE.

DCE is based on a layered model which integrates a set of fundamental technologies (Fig. 12). To applications, DCE appears to be a single logical system with two broad categories of services (18):

The DCE Core Services. They provide tools with which software developers can create end-user applications and system software products for distributed computing:

- *Threads.* DCE supports multithreaded applications;
- *RPC.* The fundamental communication mechanism which is used in building all other services and applications;
- *Security Service.* Provides the mechanism for writing applications that support secure communication between clients and servers;
- *Cell Directory Services (CDS).* Provides a mechanism for logically naming objects within a DCE cell (a group of client and server computers);
- *Distributed Time Service (DTS).* Provides a way to synchronize the clocks on different computers in a distributed computing system.

DCE Data-Sharing Services. In addition to the core services, DCE provides important data-sharing services, which require no programming on the part of the end user and which facilitate better use of shared information:

Distributed File Service (DFS). Provides a high-performance, scalable, secure method for sharing remote files;

Enhanced File Service (EFS). Provides features which greatly increase the availability and further simplify the administration of DFS.

In a typical distributed environment, most clients perform their communication with only a small set of servers. In DCE, computers that communicate frequently are placed in a single cell. Cell size and geographical location are determined by the people administering the cell. Cells may exist along social, political, or organizational boundaries and may contain up to several thousand computers. Although DCE allows clients and servers to communicate in different cells, it optimizes for the more common case of intra-cell communication. One computer can belong to only one cell at a time.

The Role of RPC

DCE RPC is based on the Apollo Network Computing System (NCA/RPC). The components of DCE RPC can be split into the following two groups according to the stage of their usage:

Used in Development. It includes IDL (Interface Definition Language) and the idl compiler. IDL is a language used to define the data types and operations applicable to each interface in a platform independent manner. idl compiler is the tool used to translate IDL definitions into code which can be used in a distributed application;

Used in Runtime. It includes RPC runtime library, *rpcd* (RPC daemon), and *rpccp* (RPC control program).

To build a basic DCE application, the programmer has to supply the following three files:

The Interface Definition File. It defines the interfaces (data structures, procedure names, and parameters) of the remote procedures that are offered by the server;

The Client Program. It defines the user interfaces, the calls to the remote procedures of the server, and the client side processing functions;

The Server Program. It implements the calls offered by the server.

DCE uses threads to improve the efficiency of RPCs. A thread is a lightweight process that executes a portion of a program, cooperating with other threads concurrently executing in the same address space of a process. Most of the information that is a part of a process can then be shared by all threads executing within the process address space. Sharing reduces significantly the overhead incurred in creating and maintaining the information, and the amount of information that needs to be saved when switching between threads of the same program.

The Servers of DCE

All the higher-level DCE services, such as the directory services, security service, time service, and distributed file services, are provided by relevant servers.

Directory Services. The main job of the directory services is to help clients find the locations of appropriate servers. To let clients access the services offered by a server, the server has to place some binding information into the directory. A *directory* is a hierarchically structured database which stores dynamic system configuration information. The directory is a realization of the naming system. Each name has attributes associated with it, which can be obtained via a query using the name.

Each cell in a DCE distributed computing system has its own directory service, called the Cell Directory Service (CDS), that stores the directory service information for a cell (18). It is optimized for intra-cell access, since most clients communicate with servers in the same cell. Each CDS consists of CDS servers and CDS clerks. A CDS server runs on a computer containing a database of directory information (called the *clearinghouse*). Each clearinghouse contains some number of directories, analogs to but not the same as directories in a file system. Each directory, in turn, can logically contain other directories, object entries, or soft links (an alias that points to something else in CDS).

Each cell may have multiple CDS servers. Nodes which do not run a CDS server must run a CDS clerk. A CDS clerk acts as an intermediary between a distributed application and the CDS server on a node not running a CDS server.

When a server wishes to make its binding information available to clients, it *exports* that information on one of its cell's CDS servers. When a client wishes to locate a server within its own cell, it imports that information from the appropriate CDS server by calling on the CDS clerk on its computer.

DCE uses the Domain Name System (DNS) or Global Directory Service (GDS, based on the X.500 standard) to enable clients to access servers in foreign cells. To access a server in a foreign cell, a client gives the cell's name and the name of the desired server. A CDS component called a Global Directory Agent (GDA) extracts the location of the named cell's CDS server from DNS or GDS, then a query is sent directly to this foreign server.

Security Service. DCE provides the following four security services: authentication, authorization, data integrity, and data privacy. A security server (it may be replicated) is responsible for providing these services within a cell. The security server has the following three components:

Registry Service. It is a database of principal (a user of the cell), group, and organization accounts, their associated secret keys, and administration policies.

Key Distribution Service. It provides tickets to clients. A ticket is a specially encrypted object that contains a conversation key and an identifier that can be presented by one principal to another as a proof of identity.

Privilege Service. It supplies the privileges of a particular principal. It is used in authorization.

The security server must run on a secure computer, since the registry on which it relies contains a secret key, generated from a password, for every principal in the cell. They are based on the Kerberos V5.0, created by the MIT/Project Athena, and DCE extends Kerberos version 5 by providing authorization services.

Time Service. Distributed Time Service (DTS) of DCE is designed to keep a set of clocks on different computers synchronized. DTS uses the usual client-server structure: DTS clients, daemon processes called clerks, request the correct time from some number of servers, receive responses, and then reset their clocks as necessary to reflect this new knowledge.

There are several components that compose the DCE DTS:

Time Clerk. It is the client side of DTS. It runs on a client computer and keeps the computer's local time synchronized by asking a time server for the correct time and adjusting the local time accordingly.

Time Servers. There are three types of time servers. The *local time server* maintains the time synchronization of a given LAN. The *global time server* and *courier time servers* are used to synchronize time among interconnected LANs. A time server synchronizes with other time servers by asking these time servers for correct times and by adjusting its time accordingly.

DTS API. It provides an interface where application programs can access time information provided by the DTS.

Distributed File Services. DCE uses its distributed file services (DFS) to join the file systems of individual computers within a cell into a single file space. A uniform and transparent interface is provided for applications to accessing files located in the network. DFS is derived from the Andrew File System. It uses RPC for client-server communication and threads to enhance parallelism; it relies on the DCE directory to locate servers; and it uses DCE security services to protect from attackers.

DFS is based on the client-server model. DFS clients, called cache managers, communicate with DFS servers using RPC on behalf of user applications. There are two types of DFS servers: a fileset location server which stores the locations of system and user files in DFS, and a file server which manages files.

A typical interaction between various components of DFS is shown in Fig. 13. At first, the application issues a file request call to the cache manager in its computer. If the requested file is located in the local cache, the request is served using the local copy of the file. Otherwise, the cache manager locates the fileset location server through the CDS server, and the location of the file server that stores the requested file is found through the fileset location server. Finally, the cache manager calls the file server and the file data are accessed.

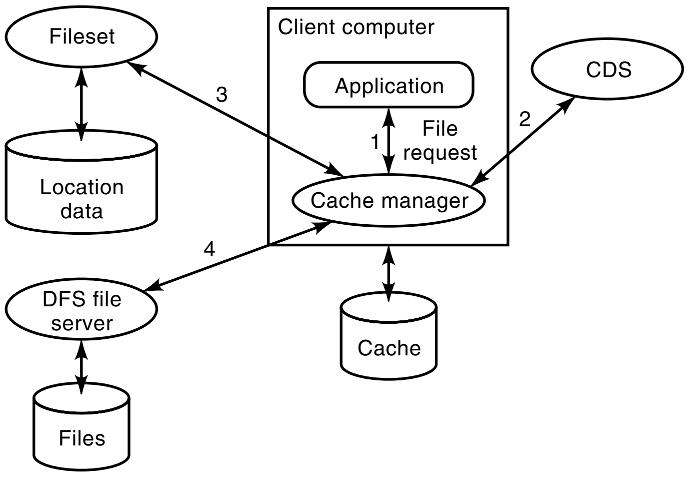

Figure 13. Interactions between DFS components. 1: file request from an application; 2: locate the fileset location server; 3: locate the file server that stores the requested file; 4: access the requested file.

Conclusions

In this section we described an application of the client-server model in the development of an advanced distributed computing environment, DCE. DCE is built on top of existing operating systems and it hides the heterogeneity of underlying computers by providing an integrated environment for distributed computing. DCE consists of many integrated services, such as thread and RPC services, security service, directory service, time service, and distributed file service, that are necessary in performing client–server computing in a heterogeneous environment. Most of these services are implemented as individual servers or groups of cooperating servers. Application processes act as clients of DCE servers. Now in its fifth year (DCE 1.0 was announced in 1991), DCE has gone through several major stages of evolution and enhancement (through DCE 1.1 and DCE 1.2). Because of its operating system independence, DCE has gained significant support from user and vendor communities.

BIBLIOGRAPHY

1. L. Liang, S. T. Chanson, and G. W. Neufeld, Process groups and group communications: Classifications and requirements, *IEEE Computer,* **23** (2): 56–66, 1990.
2. A. S. Tanenbaum, Experiences with the Amoeba distributed operating system, *Commun. ACM,* **33** (12): 46–63, 1990.
3. F. Cristian, Understanding fault tolerant distributed systems, *Commun. ACM,* **34** (2): 56–78, 1991.
4. K. P. Birman and T. A. Joseph, Reliable communication in the presence of failures, *ACM Trans. Comp. Sys.,* **5** (1): 47–76, 1987.
5. P. Wegener, Interoperability, *ACM Comp. Surv.,* **28** (1): 285–287, 1996.
6. A. Goscinski, *Distributed Operating Systems: The Logical Design,* Reading, MA: Addison-Wesley, 1991.
7. A. D. Birrell and B. J. Nelson, Implementing remote procedure calls, *ACM Trans. Comp. Sys.,* **2** (1): 39–59, 1984.
8. D. R. Cheriton, The V distributed system, *Commun. ACM,* **31** (3): 314–333, 1988.
9. A. S. Tanenbaum and R. van Renesse, Distributed operating systems, *ACM Comp. Sur.,* **17** (4): 419–470, 1985.
10. D. De Paoli et al., Microkernel and kernel server support for parallel execution and global scheduling on a distributed system, *Proc. IEEE First Int. Conf. Algorithms Architectures Parallel Process.,* Brisbane, April, 1995.
11. D. R. Cheriton and Zwaenepoel, Distributed process groups in the V distributed system, *ACM Trans. Comp. Sys.,* **3** (2): 77–107, 1985.

12. M. Rozier et al., Chorus distributed operating system, *Comput. Syst.,* **1**: 305–379, 1998.

13. M. F. Kaashoek and A. S. Tanenbaum, Efficient reliable group communication for distributed system, *Department of Mathematics and Computer Science Technical Report,* Vrije Universiteit, Amsterdam, 1994.

14. R. Sandberg et al., Design and implementation of the Sun Network File System, *Proc. Summer USENIX Conf.,* 119–130, 1985.

15. Sun Microsystems, NFS Version 3 Protocol Specification (RFC 1813), *Internet Network Working Group Request for Comments,* No. 1813, Network Information Center, SRI International, June, 1995.

16. G. Gerrity et al., Can we study design issues of distributed operating systems in a generalized way?—RHODOS, *Proc. 2nd Symp. Experiences Distributed Multiprocessor Syst. (SEDMS II),* Atlanta, March, 1991.

17. *Distributed Computing Environment Rationale,* Open Software Foundation, 1990.

18. *The OSF Distributed Computing Environment,* Open Software Foundation, 1992.

ANDRZEJ GOSCINSKI
WANLEI ZHOU
Deakin University