CHAPTER 3 COMMUNICATION PARADIGMS FOR DISTRIBUTED NETWORK SYSTEMS

Communication is an important issue in distributed computing systems. This chapter addresses the communication paradigm of distributed network systems, i.e., issues about how to build the communication model for these systems.

3.1 Introduction

Distributed computing systems, in particular those managed by a distributed operating system, must be running fast in order to instil in users the feeling of a huge powerful computer sitting on their desks through hiding distribution of resources. Since we have shown in Chapter 2 that the client-server model is commonly used in distributed network systems, this implies that the communication between clients and servers for cooperating by exchanging requests and responses 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. Distributed systems based on clusters of workstations or PCs do not share physical memory. Thus, requests and responses are sent in the form of messages. The issue is how to build a communication facility within a distributed system to achieve high communication performance.

There is a set of factors which influence the performance of a communication facility. First, the speed of a communication network ranging from slow 10 Mbps to very fast Gbps. Second, the communication protocols that span the connection-oriented protocols such as OSI and TCP, which generate considerable overhead to specialised fast protocols. Third, the communication paradigm, i.e., the communication model supporting cooperation between clients, servers and an operating system support provided to deal with the cooperation. In this chapter we only concentrate on the third factor, i.e., on the communication paradigm.

There are two issues in the communication paradigm. First, as we have shown in Chapter 2, 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,* see Chapter 9), which are operating system abstractions. Second, these two patterns of inter-process communication could be developed based on two different techniques: Message Passing, adopted for distributed systems in the late 1970s; and Remote Procedure Call (RPC), adopted for distributed systems in mid 1980s. These two techniques are supported by two

different sets of primitives provided in 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 two techniques of interprocess communication are based on different basic concepts. Message passing between remote and local processes is visible to the programmer. The flow of information is unidirectional from the client to the server. However, in advanced message passing, such as structured message passing or rendezvous, information flow is bidirectional, i.e., a return message is provided in response to the initial request. Furthermore, message passing is a completely untyped technique.

The RPC technique 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. Message passing is invisible to the programmer of RPC, which requires a transport protocol to support the transmission of its arguments and results. It is important to note that the term remote procedure call is sometimes used to describe just structured message passing. Remote procedure call primitives provide bidirectional flow of information.

The following topics are discussed in this chapter. First of all, message passing, in particular messages in distributed systems; communication primitives; semantics of these primitives; direct and indirect communication; blocking and non-blocking primitives; buffered and unbuffered exchange of messages; and reliable and unreliable primitives are considered. Second, RPC is discussed. In particular, basic features of this technique; execution of RPC; parameters, results and their marshalling; client-server binding; and reliability issues are presented. Third, group communication is discussed. In particular, basic concepts of this communication pattern; message delivery and response semantics; and message ordering in group communication are presented. The detail implementation of a group communication protocol is discussed in Chapter 9.

3.2 Message Passing Communication

There are two critical issues in message-passing communication: the messages used in the communication and the mechanisms used to send and receive messages. A user is explicitly aware of these two issues during this form of communication. In this section, we discuss the message-passing communication based on these two issues.

3.2.1 What is a Message?

A *message* is a collection of data objects consisting of a fixed size header and a variable or constant length body, which can be managed by a process, and delivered to its destination. A type associated with a message provides structural information

on how the message should be identified. A message can be of any size and may contain either data or typed pointers to data outside the contiguous portion of the message. The contents of a message are determined by the sending process. On the other hand, some parts of the header containing system-related information may be supplied by the system.

Messages may be completely unstructured or structured. Unstructured messages have the flexibility that can be interpreted by the communicating pairs. However, the use of unstructured messages that are then interpreted as needed by user processes has problems. That is because some parts of messages (e.g., port names) must be interpreted by the distributed operating system or the communication protocol because they must be translated to be meaningful to another process. Moreover, in heterogeneous networks, only typed information allows the transparent transfer of data items (integers, reals, strings, etc.). Structured messages are also favored for efficiency reasons. Most information transferred between processes is structured in that it represents data items of different types. The use of unstructured messages for such data could be expensive because the encapsulation and decapsulation of structured messages into unstructured linear forms adds a layer of overhead that increases the cost of communication.

In order for any two computers to exchange data value, we need to map data structures and data items to messages. Data structure must be flattened before transmission and rebuilt on arrival. Flattening of structured data into a sequence of basic data is used for the data transmission. Usually a language preprocessor (interface compiler) can be used to generate marshalling/unmarshalling operations automatically. When an IPC primitive is encountered involving data items of the above type, the preprocessor generates code to do the marshalling (for a send) or unmarshalling (for a receive) based on the type description.

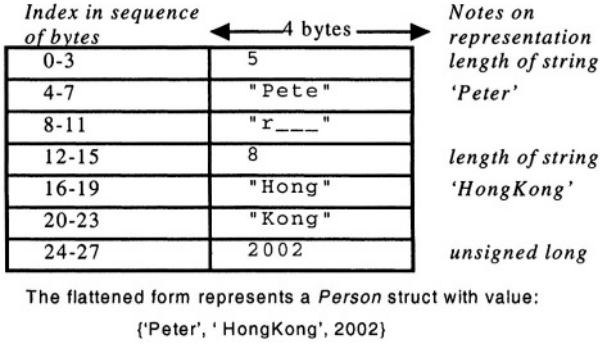

Figure 3.1: CORBA CDR message

Example of structured messages:

External data representation (XDR) provided by SUN XDR and Courier for automatic marshalling. Message consists of a sequence of 4-byte objects: (1)

Cardinal/integer: 4 bytes; (2) Character: 1 byte; (3) represented as sequences of bytes with the length specified. On receiving a data stream, the data structure must be rebuilt. The diagram in Figure 3.1 is an example of CORBA CDR (Common Data Representation) [OMG 1998].

3.2.2 Message-Passing Mechanisms

Another important issue in the message-passing communication is about the mechanisms used to send and receive messages. These mechanisms involve a set of primitives used for communication.

3.2.2.1 Basic Message-Passing Primitives

There are two issues related to message-passing mechanisms: one is, what is the set of communication primitives, and the other is, what are their semantics? In message-passing communication, a message is sent and received by explicitly executing the *send* and *receive* primitives, respectively. The time diagram of the execution of these two primitives is shown in Figure 3.2. 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.

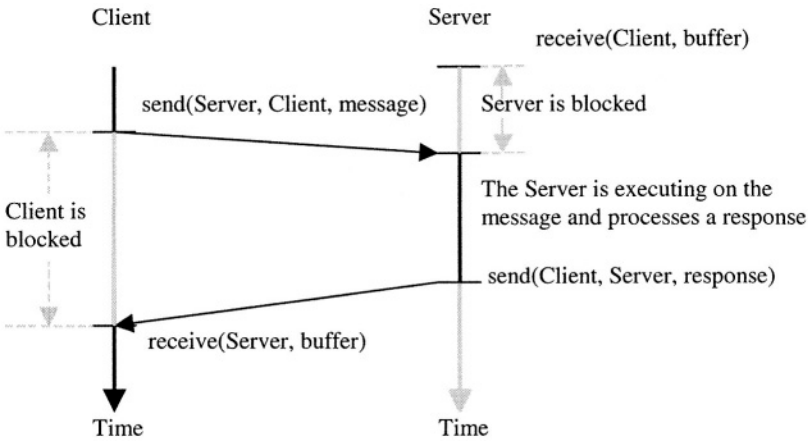

Figure 3.2: Time diagram of the execution of message-passing primitives

Of course, when the server process sends any message to the client process, they have to use these two primitives as well; the server sends a message by executing primitive *send* and the client receives it by executing primitive *receive.*

Several points 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 ports;
- Blocking versus non-blocking primitives;
- Buffered versus unbuffered primitives;
- Reliable versus unreliable primitives; and
- Structured forms of message passing based primitives.

3.2.2.2 Direct and Indirect Communication Ports

The very basic issue in message-based communication is where 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 and 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. 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 this scheme is the limited modularity of the resulting process definition. Changing the name of the process may necessitate the examination of all other process definitions. 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.

Direct communication does not allow more than one client. That is because, at the very least, issuing the receive primitive would be required for each client. The server process cannot reasonably anticipate the names of all potential clients. 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, i.e., the messages are sent to and received from ports. Processes may have ownership, and send and receive rights on a port. Each port has a unique identification (name) that distinguishes it. A process may communicate with other processes by a number of different ports.

Logically associated with each port is a queue of finite length. Messages that have been sent to this port, but 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, that is, the interface between the client and server. A process that wants to remove a message from a port must have the appropriate receive rights. Usually, only one process may 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.

A port can be owned either by a process or by the operating system. If a port is owned by a process, the port is attached to or defined as a part of the process. Ownership of a port can be passed in a message from one process to another. A port can be created by a process which subsequently owns that port. The process also has a receive access to that port. If a single process both owns and has receive access to a port, this process may destroy it. The problem is what can happen to a port when its owner dies. If this process has receive access to that port, the best solution to this problem is automatic destruction of the port. Otherwise, an emergency message is sent to the process which has access rights to it. If the process that is not the owner but has access rights to a port dies, then the emergency message is sent to the owner.

If the operating system owns a port, it provides a mechanism that allows a process to: create a new port (the process is its owner by default); send and receive messages through the port; and destroy a port.

A finite length of the message queues attached to ports is used to prevent a client from queuing more messages to a server than can be absorbed by the system, and as a means for controlling the flow of data between processes of mismatched processing speed. Some implementations can allow the processes owning a port to specify the maximum number of messages which can be queued for that port at any time.

3.2.2.3 Blocking versus Non-blocking Primitives

One of the most important properties of message passing primitives concerns whether their execution could cause delay. We distinguish blocking and non-blocking primitives. We say that a primitive has non-blocking 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.

It is necessary to distinguish two different forms of the blocking primitives, in particular send. These forms are generated by different criteria. The first criterion reflects the operating system design, addresses buffer management and message transmission. The blocking and non-blocking send primitives developed following this criterion are illustrated in Figure 3.3. If the blocking send primitive is used, the sending process (client) is blocked, i.e., 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 non-blocking 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.

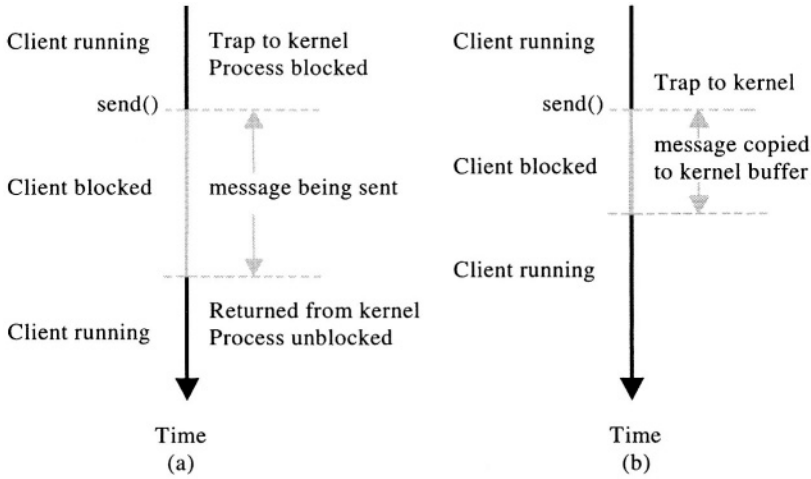

Figure 3.3: Send primitives: (a) blocking; (b) non-blocking

The second criterion reflects the client-server cooperation and the programming language approach to deal with message communication. In this case the client is blocked until the server (receiver) has accepted the request message and the result or acknowledgment has been received by the client. The blocking send primitives developed following this criterion is illustrated in Figure 3.4.

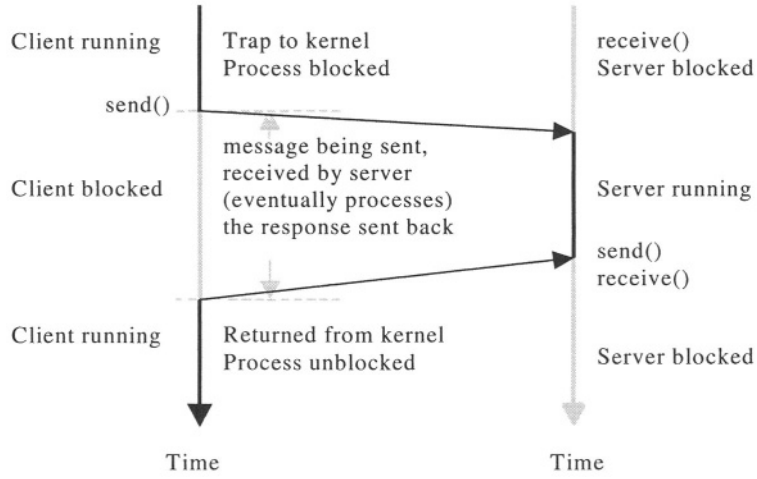

Figure 3.4: Blocked send primitive

There are three forms of 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 are also a non-blocking 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. Blocking primitives provide a simple way to combine the data transfer with the synchronisation function.

With non-blocking primitives:

- Send returns control to the user program as soon as the message has been queued, for subsequent transmission, or a copy made (these alternatives are determined by the method of cooperation between the network interface and the processor);
- When a message has been transmitted (or copied to a safe place for subsequent transmission), the program is interrupted to inform it that the buffer may be reused;
- The corresponding receive primitive signals a willingness to receive a message and provides a buffer into which the massage may be placed; and
- When a message arrives, the program is informed by interrupt.

The advantage of these non-blocking primitives is that they provide maximum flexibility. Moreover, these primitives are useful for real-time applications. The disadvantages of these non-blocking primitives are that they may require buffering to prevent access or change to message contents. These accesses and changes to the message may happen before or during transmission, or while the message is waiting to be received. Buffering may occur on source or destination sites. If a buffer is full, a process must be blocked, which contradicts the original definition of this primitive; make programming tricky and difficult (non-reproducible, timing dependent programs are painful to write and difficult to debug).

3.2.2.4 Buffered versus Unbuffered 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, 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 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. Unbuffered and buffered message passing are illustrated in Figure 3.5, where (a) represents unbuffered message passing (messages are discarded before the server issues the receive primitive); and (b) represents buffered message passing (messages are buffered in the OS area for a limited time).

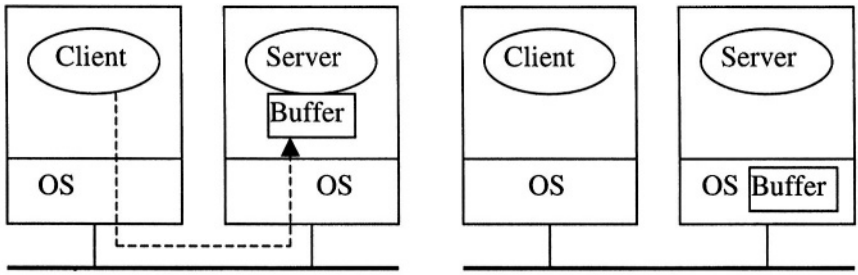

Figure 3.5: Unbuffered and buffered message passing.

Let us consider two extremes: a buffer with unbounded capacity and one with finite bounds. If the buffer has unbounded capacity, then a process is never delayed when executing a send. Systems based on this approach are called systems with asynchronous message passing or systems with no-wait send. The most important feature of asynchronous message passing is that it allows a sender to get arbitrarily far ahead of a receiver. Consequently, when a message is received it contains information about the sender's state that may no longer be valid. If the system has no buffering, execution of send is always delayed until a corresponding receive is executed. Then the message is transferred and both proceed.

When the buffer has finite bounds, we deal with buffered message passing. In this case the client is allowed to get ahead of the server, but not arbitrarily far ahead. In buffered message-passing based systems, the client is allowed to have multiple sends outstanding on the basis of a buffering mechanism (usually in the operating system kernel). In the most often used approach, the user is provided with a system call *create_buffer,* which creates a kernel buffer, of a size specified by the user. This solution implies that the client sends a message to a receiver's port, where it is buffered until requested by the server.

Buffered message-passing systems are characterised by the following features. First, they are more complex than unbuffered message-passing based systems, since they require creation, destruction, and management of the buffers. Second, they generate protection problems, and cause catastrophic event problems, when a process owning a port dies or is killed. In a system with no buffering strategy, processes must be synchronised for a message transfer to take place. This synchronisation is called *rendezvous* see [Gammage and Casey 1985] and [Gammage *et al.* 1987].

3.2.2.5 Unreliable versus Reliable Primitives

Different catastrophic events, such as a computer crash or a communication system failure may happen to a distributed system. These can cause either a request message being lost in the network, or 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 above cannot cope with these problems. These are called *unreliable primitives.* The unreliable send primitive 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 problems requires providing *reliable primitives.* In a reliable inter-process 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.

The question arises of whether reliability should be dealt with at such a high level. Should recovery mechanisms be provided by a network communication facility, in particular either by a transport protocol or lower level protocols? These problems were attacked in [Saltzer *et al.* 1984]. The authors proposed design principles that help guide placement of functions among modules of a distributed system. One of these principles, called end-to-end argument, suggests that "functions placed at a low level of a system may be redundant or of little value when compared with the cost of providing them at that low level". This allows us to suggest that the placement of recovery mechanisms at a process level is sound.

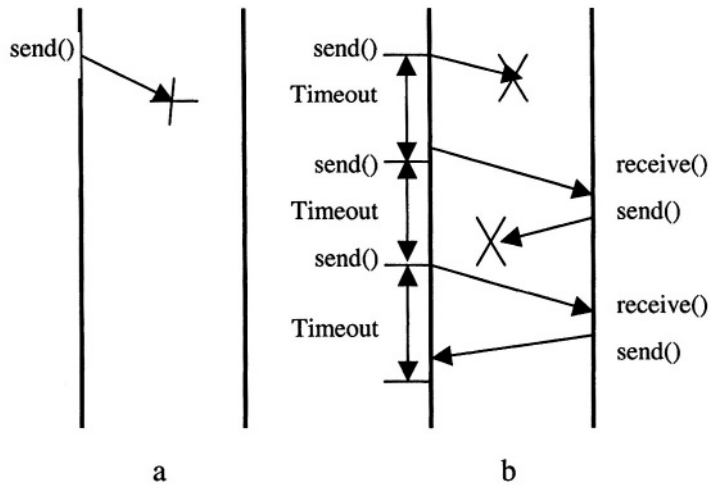

Figure 3.6: Message passing; (a) unreliable; (b) reliable

Reliable and unreliable receives differ in that the former automatically sends an acknowledgment confirming message reception, whereas the latter does not. Two-way communication requires the utilisation of these simple 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 Figure 3.6.

Dealing with multiple requests

Two types of semantics are defined, depending on what the server does if it receives multiple copies of a message. It might repeat its processing of the message, even if only one execution was actually desired. Since in this case the reliable primitives do their best to ensure that the request is executed at least once (Figure 3.7(a)). This is called communication with *at-least-once semantics.*

In many cases, of course, repeated execution of a request could destroy the consistency of information, so it is desirable to have primitives which ensure that a request is executed once and only once. This means that a request is idempotent. A primitive with *exactly-once semantics* makes sure that only one execution of the receiver's operation is performed. It is the most desired semantics, but the most difficult to implement.

Exactly-once primitives can be implemented on the basis of a request list maintained by the responding end of the system (Figure 3.7(b)). In this case, each time a request message is received, the server checks whether the *message_id* is on the request list. If yes, this means that this request has been retransmitted and a response message is lost. The previously computed result is sent in a new response message. Otherwise, the *message_id* is placed on the request list, the requested task is performed, a result is associated with a *message_id* entry, and a response message is sent out.

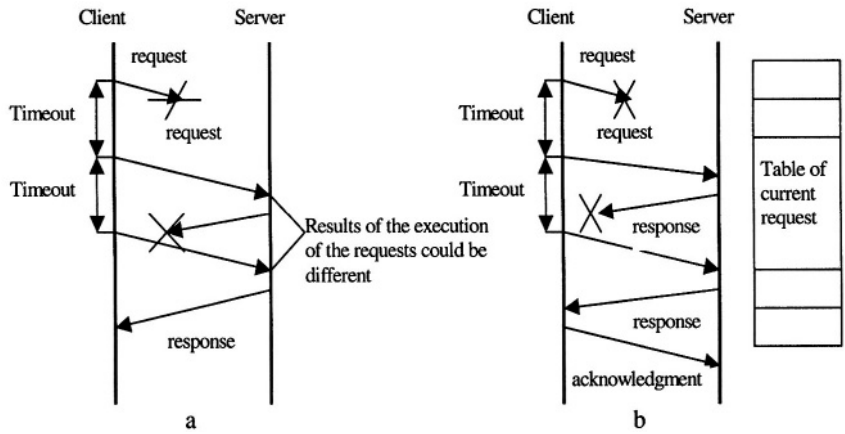

Figure 3.7: Message-passing semantics. (a) at-least-once; (b) exactly-once

3.2.3 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 send primitive combines the previous client's send to the server with a receive to get the server's response. On the other site the receiver (server) acquires a message containing work for them to do and sends the response to the client.

It should be emphasised that different semantics 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 inter-process 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 (because error handling as well as flow control exploit the response to acknowledge a request and authorise a new request) [Cheriton 1988].

When the requesting process is blocked waiting for a reply, it can be blocked indefinitely. This can occur because of a communication failure, a destination computer failure, or simply because the server process does not exist any longer or is too busy to compute a response in a reasonable time. This requires provision of a mechanism to allow the requesting process to withdraw from the commitment to wait for the response.

3.3 Remote Procedure Calls

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 that 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 by [Birrell and Nelson 1984].

3.3.1 Executing Remote Procedure Calls

The idea of remote procedure calls (RPC) is very simple and is based on the model where 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 centralised computer systems. This implies that RPC must be transparent. This leads to one of the main advantages of this communication technique: the programmer need not be aware 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. 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, firstly, puts the parameters into registers, and next 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, 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 the 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, i.e., the server stub. The stub packs them into a message and issues a send to return the results. The stub comes back to the beginning of the 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, the results 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. The whole sequence of operations is illustrated in Figure 3.8.

As we could see above, the RPC mechanism can be used to provide an inter-process communication facility between a single client process and a single server process. Such a mechanism can be extended to a system of many clients and many servers. Furthermore, the RPC facility can be used in homogeneous as well as heterogeneous computer systems.

3.3.2 Basic Features and Properties

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.

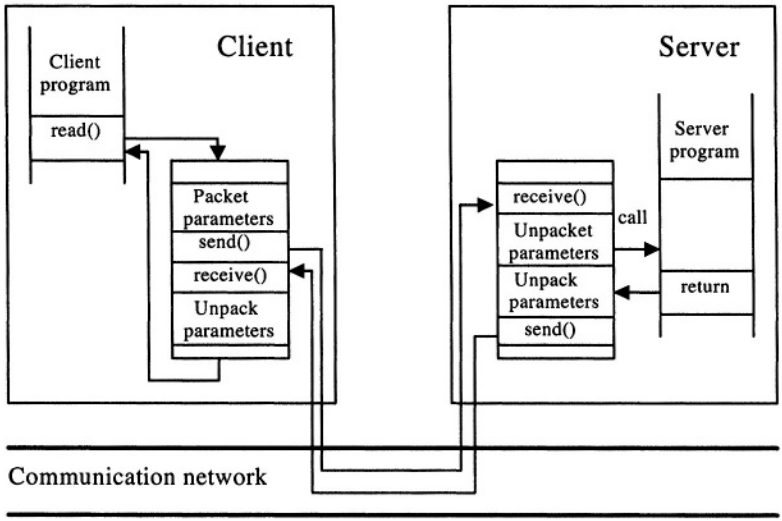

Figure 3.8: An RPC example: a read call

The difference between procedure calls and remote procedure calls is implied by the fact that the client and called procedure are in separate processes, usually running on separate computers. Thus, they are prone to the failures of computers as well as communication systems, they do not share the same address space, and they have separate lifetimes.

There is also 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, i.e., the list of parameters is collected together by the system to form a message.

A remote procedure call mechanism should exhibit the following six properties [Nelson 1981], [LeBlanc 1982], [Hamilton 1984] and [Goscinski 1991]:

- The implementation of a transparent remote procedure call must maintain the same semantics as that used for local procedure calls.

- The level of static type checking (by the compiler) applied to local procedure calls applies equally to remote procedure calls.
- All basic data types should be allowed as parameters to a remote procedure call.
- The programming language that supports RPC should provide concurrency control and exception handling.
- A programming language, which uses RPC, must have some means of compiling, binding, and loading distributed programs onto the network.
- RPC should provide a recovery mechanism to deal with orphans when a remote procedure call fails.

3.3.3 Parameters and Results in RPCs

Parameter passing and the representation of parameters and results in messages are among the most important problems of the remote procedure call.

3.3.3.1 Representation of Parameters and Results

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 synchronisation 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 inter-process communication, in the form of distributed shared memory.

A unique, a system wide pointer is needed for each object so that it can be remotely accessed. For large objects (e.g., files), some kind of capability mechanism could be set up using capabilities as pointers, but for small objects (e.g., integers, booleans) the overhead involved in creating a capability and sending it is too large, so that is highly undesirable. However, the data must be finally copied.

Linking both approaches, passing data by-value and passing data by-reference, can be the most effective solution. The representation of parameters and results in messages is natural for homogeneous systems. The representation is complicated in heterogeneous systems.

3.3.3.2 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 a process performed both when sending the call (request) as well as when sending the result, in which three actions can be distinguished:

- Taking the parameters to be passed to the remote procedure and 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. According to [Nelson 1981], the compiler should always generate in-line marshalling code for every remote call. This permits more efficient marshalling than interpretive schemes but can lead to unacceptably large amounts of code. However, some systems allow the programmer to define marshalling procedures for types that inplude pointers [Bacon and Hamilton 1987].

3.3.4 Client Server Binding

Usually, RPC hides all details of locating servers from clients. However, as we stated in Section 2.3, in a system with more than one server, e.g., a file server and a print server, the knowledge of location of files or a special type of 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.

Naming and addressing

[Birrell and Nelson 1984] identified two aspects of binding:

- The way a client specifies what it wants to be bound to — this is the problem of naming;

- The way of determination by a client (caller) of the computer address for the server and the specification of the procedure to be invoked — this is the problem of addressing.

The first aspect of binding, i.e., naming, was solved in [Birrell and Nelson 1984] in terms of interface names. In their proposal, individual procedures are identified by entry point numbers within an interface. Interface names are user created. The second problem is how to locate a server for a client. This issue was discussed in Section 2.6.

As discussed in Section 2.3.1, in a distributed system there are two different forms of cooperation between clients and servers. The first form assumes that a client requests a temporary service. The second form indicates that a client wants to arrange for a number of calls to be directed to a particular serving process. In the case of requests for a temporary service, the problem can be solved using broadcast and multicast messages to locate a process or a server. In the case that a solution is based on a name server this 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 and registered containing established long term binding objects, i.e., client names and server names. The RPC run-time procedure for performing remote calls expects to be provided with a binding object as one of its arguments. This procedure directs a call to the binding address received. It should be possible to add new binding objects to the binding table, remove binding objects from the table (which in practice means breaking a binding), and update the table as well. In systems with name server(s), 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.

In summary, binding can be performed in two different ways: statically through the third party such as a name server; clients and servers are user processes, and dynamically this binding is between a client channel and a server process, and is controlled by the server which can allocate its server process to active channels.

Binding time

It is important to know when binding can take place. The construction and use of an RPC-based distributed application can be divided into three phases: compile time, link time, and call time [Bershad *et al.* 1987].

Compile time:

- The client and server modules are programmed as if they were intended to be linked together.
- A description of the interface implemented by a server is produced. It yields two stubs: client and server. The client stub, which looks to the client like a server, is linked with the client. The server stub, which looks to the server like a client, is linked with the server.
- The stubs shield the client and server from the details of binding and transport.

- Ideally the stubs are produced mechanically from the definition of the interface, by a stub generator.

Link time:

- A server makes its availability known by exporting (or registering) itself through the RPC routine support mechanism.
- A client binds itself to a specific server by making an import call to this mechanism.
- Calls can take place, once the binding process has been completed.
- It is expected that binding will be performed less frequently than calling.

Call time:

- In providing procedure call semantics by the inter-process communication facility of an operating system, the stubs employ some underlying transport layer protocol to transmit arguments and results reliably between clients and servers.
- The RPC facility should include some control information in each transport packet to track the state of a call.

3.4 Message Passing versus Remote Procedure Calls

The problem arises of deciding which of the two inter-process communication techniques presented above 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 looks like RPC are message passing for the V system (which in [Cheriton 1988] is called now the remote procedure call system), message passing for Amoeba [Tanenbaum and van Renesse 1985] and RHODOS [De Paoli *et al.* 1995].

Secondly, the RPC has an 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 the details of messages from a programmer. Note that a simplified structure that hides message details reduces the range of communication options available to applications programmers. 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 behaviour of a message-passing interface.

The problem is, when these facilities should be used. The message-passing approach appears preferable when serialisation 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.

Inter-process communication is a distributed system facility whose performance has been extensively studied, in contrast to other facilities or issues. Unfortunately, these studies have been carried out mainly for one particular system [Cheriton 1988] [Rashid 1986] [Welch 1986]. This implies that it is very hard to say which form of inter-process communication offers the best performance.

3.5 Group Communication

When there is a need to send a request to a number of servers, group communication should be used for the performance and ease of programming reasons. The details of group communication will be presented in Chapter 9. In this section, we briefly introduce some basic concepts about group communication.

3.5.1 Basic Concepts

Distributed computing systems provide opportunities to improve the overall performance through parallel execution of programs on a cluster of workstations, decrease response time of databases using data replication, support synchronous distant meetings and cooperative workgroups, and increase 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, called a *group.*

The concept of a process group is not new. The V-system [Cheriton and Zwaenepoel 1985], Amoeba [Tanenbaum 1990], Chorus [Rozier *et al.* 1988], and RHODOS [Joyce and Goscinski 1997] all support this basic abstraction in providing process groups to applications and operating system services with the use of one-to-many communication pattern, called *group communication.*

A group is a collection of processes that share common features (described by a set of attributes) or application semantics, for instance file servers and printer servers. In general, processes are grouped in order to [Liang *et al.* 1990]:

- deal with a 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);
- provide a high-level communication abstraction to simplify user level programs in interacting with a group of receivers;
- encapsulate the internal state and hide interactions among group members from the clients, and provide a uniform interface to the external world;

- deliver a single message to multiple receivers thereby reducing the sender and receiving overheads; and
- construct larger systems using groups as their fundamental blocks.

There are several issues which allow us to set up a taxonomy of process groups: *group structure, group behaviour,* and *group types.*

3.5.1.1 Group Structures

Four group structures are often supported to provide the most appropriate policy for a wide range of user applications, as shown in Figure 3.9.

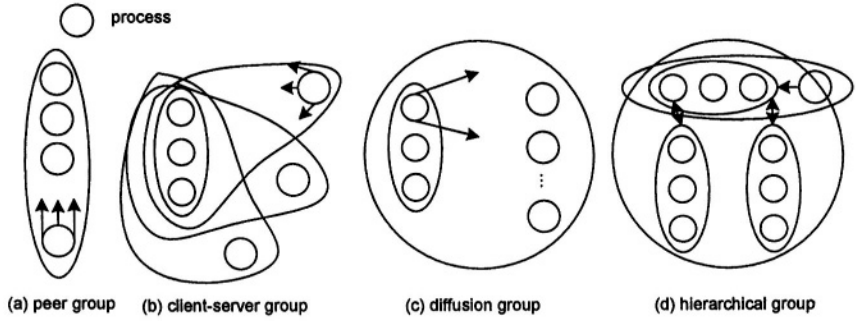

Figure 3.9: Group structures

- The *peer group* is composed of a set of member processes that cooperate for a particular purpose, see Figure 3.9(a). Fault-tolerant and load sharing applications dominate this type of group style. The major problem of the peer group style is that they do not scale very well.
- The *client-server group* is made from a potentially large number of client processes with a peer group of server processes, see Figure 3.9(b).
- The *diffusion group* is a special case of the client-server group, see Figure 3.9(c). Here, a single request message is sent by a client process to a full set of server and client processes.
- The *hierarchical group* is an extension to the client-server group, see Figure 3.9(d). In large applications with a need for sharing between large numbers of group members, it is important to localise interactions within smaller clusters of components in an effort to increase performance. In client-server applications with the hierarchical server, the client is bound transparently to a subgroup that accepts requests on its behalf. The subgroup is responsible for performing the mapping. However, the major problem with hierarchical groups is that they require a base group that may fail leaving the group inoperative.

3.5.1.2 Behaviour Classification of Process Groups

The application programmer must be aware of the behaviour of the application before a suitable policy can be formulated. According to the external behaviour, process groups can be classified into two major categories: deterministic and non-deterministic [Neufield *et al.* 1990] [Joyce and Goscinski 1997].

- *Deterministic groups:* a group is considered deterministic if each member must receive and act on a request. This requires the coordination and synchronisation between the members of the group. In a deterministic group, all members are considered equivalent. When receiving the same request in the same state, all members of the group will execute the same procedure and every member of the group will transfer to the same new state and produce the same response and external events.
- *Non-deterministic groups:* non-deterministic groups assume their applications do not require consistency in group state and behaviour, and they relax the deterministic coordination and synchronisation. 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. Due to the relaxed group consistency requirements of the non-deterministic groups, the overheads associated with this group communication are substantially less than those of the deterministic groups.

3.5.1.3 Closed and Open Groups

In general, there are two group types of groups: closed or open [Tanenbaum 1990]. In the *closed group* only the members of the group can send and receive messages to access the resource(s) of the group. In the *open group* not only can the members of the group exchange messages and request services but non-members of the group can send messages to group members. Importantly, the non-members of the group need not join the group nor have any knowledge that the requested service is provided by a group.

Closed groups are typically used to support parallel processing where a group of process servers work together to formulate a result which does not require the interaction of members outside the group. Closed groups are often implemented in a peer group or diffusion group structure. Conversely, an open group would best suit a replicated file server where process members of the group should have the ability to send messages to the group members. A common group structure for the open group is the client-server or hierarchical group.

3.5.2 Group Membership Discovery and Operations

Group membership is responsible for providing a consistent view to all the members of the current group. Each member of the group must exchange messages amongst themselves to resolve the current status and membership of the group. The

identification of the current members of the group is important in providing the specified message ordering semantics. Any change in group membership will require all members to be notified to satisfy the requested message requirements. For instance, if fault tolerance is required by the application program, not only must membership changes be provided to all group members before the next message, but all group members must be notified of the change at the same point in the message sequence of the group.

An extension to the group membership semantics is the dynamic and static group membership. Static group membership does not allow any members to leave or join the group after initiation of the group. On the other hand, dynamic group membership semantics allow processes to join and leave the group.

There are two different sets of primitives, the first used to support group membership discovery [Cristian 1991], and the second to support group association operations [Amir et al. 1993] [Birman and Joseph 1987] [Jia et al 1996]. Group membership discovery allows a process to determine a state of the group and its membership. However, as the requesting process has no knowledge of the group members location, network broadcast is required. The overheads associated with this method can be prohibitive if the network is large.

There are four operations that address 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.

Essential to the role of group operations is state management for its members. Whenever a process issues and completes a join group primitive, it will be provided with the complete state of the group. The groups' state will be provided from the current group members and will contain a copy of the current groups' view, delivered messages, and current message sequence. The state transferred must not violate the message ordering and reliability of the application messages.

Unfortunately, group membership is greatly complicated by network partitioning and process or computer failure. The group association primitives allow member processes to leave and join at any given time of the life of the group. Following network partitioning members of a group can be un-reachable for some period of time. This is further complicated as in an asynchronous environment, such as a distributed system, where it is almost impossible to distinguish between the slow response of a remote process to that of a partitioned network [Amir *et al.* 1992]. Hence, the group membership facility must provide the flexibility to maintain serviceable performance during network partitioning (although slightly degraded) by allowing a subgroup to form inside the communicating group. When the network partitions disappear the group membership support should allow the subgroups to join back into the original process group. This joining of the subgroups must re-synchronise the state of each member of the group to maintain consistency of the data and messages.

In fault tolerant applications, if a group member process terminates and leaves the group without notifying the current group members, the group will have to identify by its own means the terminated member. Once the group has come to an agreement on the member that has terminated, (i.e., the terminated process no longer responding to any incoming messages) the group membership state can be committed. Therefore, if the application requires consistent data between the members of the group, the terminated member may be removed from the group with the group membership support providing a consistent view of all the messages and events that have occurred. Depending upon the application, the group membership support must maintain the message ordering semantics requested by providing the appropriate state, message, and data consistency for all current group members.

3.6 Distributed Shared Memory

There are two basic paradigms for interprocess communication in distributed systems: one is the message-passing paradigm, which includes message-passing systems and remote procedure call (RPC) introduced above, the other is shared-memory paradigm. The message-passing paradigm uses two basic primitives, *Send* and *Receive* for interprocess communication, while the shared-memory paradigm provides processes in a system with a shared address space. Processes use this address space in the same way they use normal local memory. That is, processes access data in the shared address space through *Read* and *Write* primitives. This section introduces the distributed shared memory system.

3.6.1 What is a Distributed Shared Memory (DSM) System?

Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory. Processes access DSM by reads and updates to what appears to be ordinary memory within their address space. However, an underlying run-time system ensures transparently that processes executing at different computers observe the updates made by one another. It is as if the processes access a single shared memory, but in fact the physical memory is distributed, as depicted in Figure 3.10, where processes running on each computer access the DSM as if they access a single shared memory; each computer is connected with one another through the network.

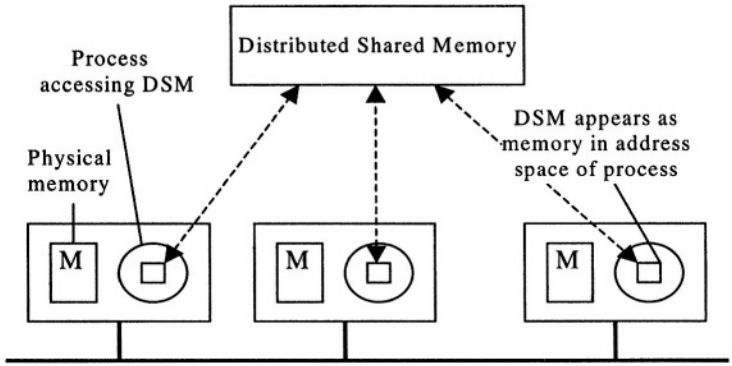

Figure 3.10: The distributed shared memory abstraction

The main point of DSM is that it spares the programmer the concerns of message passing when writing applications that might otherwise have to use it. DSM is primarily a tool for parallel and distributed (group) applications in which individual shared data items can be accessed directly. DSM is in general less appropriate in client-server systems, however, servers can provide DSM that is shared between clients.

As a communication mechanism, DSM is comparable with message-passing rather than with request-reply based communication (client-server model), since its application to parallel processing, in particular, entails the use of asynchronous communication. The DSM and message-passing approaches can be compared as follows:

Programming model: Under the message-passing model, variables have to be marshalled from one process, transmitted and unmarshalled into other variables at the receiving process. By contrast, with shared memory, the processes involved share variables directly, so no marshalling is necessary -- even of pointers to shared variables – and thus no separate communication operations are necessary.

Synchronization between processes is achieved in the message model through message passing primitives themselves, using techniques such as the lock server implementation. In the case of DSM, synchronization is via normal constructs for shared-memory programming such as locks and semaphores. Finally, since DSM can be made persistent, processes communicating via DSM may execute with non-overlapping lifetimes. A process can leave data in an agreed memory location for the other to examine when it runs. By contrast, processes communicating via message passing must execute at the same time.

Efficiency: Experiments show that certain parallel programs developed for DSM can be made to perform about as well as functionally equivalent programs written for message passing platforms on the same hardware [Carter et al 1991] – at least in the case of relatively small numbers of computers (ten or so). However, this result can not be generalized. The performance of a program based on DSM depends upon

many factors, particularly the pattern of data sharing, such as whether an item is updated by several processes.

3.6.2 Design and Implementation Issues

There are several design and implementation issues concerning the main features that characterize a DSM system. These are the structure of data held in DSM; the synchronization model used to access DSM consistently at the application level; the DSM consistency model, which governs the consistency of data values accessed from different computers; the update options for communicating written values between computers; the granularity of sharing in a DSM implementation; and the problem of thrashing.

3.6.2.1 Structure

Structure defines the abstratc view of the shared-memory space to be presented to the application programmers of a DSM system. For example, the shared-memory space of one DSM system may appear to its programmers as a storage for words, while the programmers of another DSM system may view its shared-memory space as a storage for data objects. The three commonly used approaches for structuring the shared-memory space of a DSM system are as follows:

- *No structuring.* Most DSM systems do not structure their shared-memory space. In these systems, the shared-memory space is simply a linear array of words. An advantage of the use of unstructured shared-memory space is that it is convenient to choose any suitable page size as the unit of sharing and a fixed grain size may be used for all applications. Therefore, it is simple and easy to design such a DSM system. It also allows applications to impose whatever data structures they want on the shared memory.

- *Structuring by data type.* In this method, the shared-memory space is structured either as a collection of objects or as a collection of variables in the source language. The granularity in such DSM systems is an object or a variable. But since the sizes of the objects and data types vary greatly, these DSM systems use variable grain size to match the size of the object/variable being accessed by the application. The use of variable grain size complicates the design and implementation of these DSM systems.

- *Structuring as a database.* Another method is to structure the shared memory like a database. In this method, the shared-memory space is ordered as an associative memory (a memory addressed by content rather than by name or address) called a tuple space, which is a collection of immutable tuples with typed data items in their fields. A set of primitives that can be added to any base language (such as C or FORTRAN) are provided to place tuples in the tuple space and to read or extract them from tuple space. To perform updates, old data items in the DSM are replaced by new data items. Processes select tuples by specifying the number of their fields and their values or types.

Although this structure allows the location of data to be separated from its value, it requires programmers to use special access functions to interact with the shared-memory space. Therefore, access to shared data is non-transparent. In most other systems, access to shared data is transparent.

3.6.2.2 Synchronization Model

Many applications apply constraints concerning the values stored in shared memory. In order to use DSM, a distributed synchronization service needs to be provided, which includes familiar constructs such as locks and semaphores. Even when DSM is structured as a set of objects, the implementors of the objects have to be concerned with synchronization. Synchronization constructs are implemented using message passing. DSM implementations take advantage of application-level synchronization to reduce the amount of update transmission. The DSM then includes synchronization as an integrated component.

3.6.2.3 Consistency

A DSM system is a replication system and allows replication of shared data items. In such a system, copies of shared data items may simultaneously be available in the main memories of a number of nodes. In this case, the main problem is to solve the memory coherence that deals with the consistency of a piece of shared data lying in the main memories of two or more nodes. In other words, the issue of consistency arises for a DSM system which replicates the contents of shared memory by caching it at separate computers.

Consistency requirements vary from application to application. A consistency model besically refers to the degree of consistency that has to be maintained for the shared-memory data for the memory to work correctly for a certain set applications. It is defined as a set of rules that application must obey if they want the DSM system to provide the degree of consistency guaranteed by the consistency model. Several consistency models have been proposed in the literature. Of these, the main ones will be introduced in the next section.

3.6.2.4 Update Options

Applicable to a variety of DSM consistency models, two main implementation choices have been devised for propagating updates made by one process to the others: write-update and write-invalidate.

- *Write-update:* The updates made by a process are made locally and multicast to all other replica managers processing a copy of the data item, which immediately modify the data read by local processes. Processes read the local copies of data items without the need for communication. In addition to allowing multiple readers, several processes may write the same data item at the same time; this is known as *multiple-reader/multiple-writer sharing.*

- *Write-invalidate:* This is commonly implemented in the form of multiple-reader/single-writer sharing. At any time, a data item may either be accessed in read-only mode by one or more processes, or it may be read and written by a single process. An item that is currently accessed in read-only mode can be copied indefinitely to other processes. When a process attempts to write to it, a multicast message is first sent to all other copies to invalidate them and this is acknowledged before the write can take place; the other processes are thereby prevented from reading stale data (that is, data that are not up to date). Any processes attempting to access the data item are blocked if a writer exists. Eventually, control is transferred from the writing process, and other accesses may take place once the update has been sent. The effect is to process all accesses to the item on a first-come, first-served basis.

3.6.2.5 Granularity

An issue that is related to the structure of DSM is the granularity of sharing. Conceptually, all processes share the entire contents of a DSM. As processes sharing DSM execute, however, only certain parts of the data are actually shared and then only for certain times during the execution. It would be clearly very wasteful for the DSM implementation always to transmit the entire contents of DSM as processes access and update it. What should be the unit of sharing in a DSM implementation? That is, when a process has written to DSM, which data does the DSM run-time send in order to provide consistent values elsewhere?

In many cases, DSM implementations are page-based implementations. There are a few factors that influence the choice of unit (block size), such as paging overhead, directory size, thrashing and false sharing etc. The relative advantages and disadvantages of small and large unit sizes make it difficult for a DSM designer to decide on a proper block size. Therefore, a suitable compromise in granularity, adopted by several existing DSM systems, is to use the typical page size of a conventional virtual memory implementation as the block size of a DSM system. Using page size as the block size (unit) of a DSM system has the following advantages:

1. It allows the use of existing page-fault schemes (i.e., hardware mechanisms) to trigger a DSM page fault. Thus memory coherence problems can be resolved in page-fault handlers.
2. It allows the access right control (needed for each shared entity) to be readily integrated into the functionality of the memory management unit of the system
3. As long as a page can fit into a packet, page sizes do not impose undue communication overhead at the time of network page fault.
4. Experience has shown that a page size is a suitable data entity unit with respect to memory contention.

3.6.2.6 Thrashing

A potential problem with write-invalidate protocols is thrashing. Thrashing is said to occur where the DSM run-time spends an inordinate amount of time invalidating and transfering shared data compared with the time spent by application processes doing useful work. It occurs when several processes compete for the same data item, or for falsely shared data items. If, for example, one process repeatly reads a data item that another is regularly updating, then this item will be constantly transferred from the writer and invalidated at the reader. This is an example of a sharing pattern for which write-invalidate is inappropriate and write-update would be better.

3.6.3 Consistency Models

This section describes several most common used consistency models for DSM systems.

3.6.3.1 Sequential Consistency Model

The sequential consistency model was proposed by [Lamport 1978]. A shared-memory system is said to support the sequential consistency model if all processes see the same order of all memory access operations on the shared memory. The exact order in which the memory access operations are interleaved does not matter. That is, if the three operations read (r1), write (w1), read (r2) are performed on a memory address in that order, any of the orderings (r1, w1, r2), (r1, r2, w1), ... of the three operations is acceptable provided all processes see the same ordering. If one process sees one of the orderings of the three operations and another process sees a different one, the memory is not a sequentially consistent memory.

The consistency requirement of the sequential consistency model is weaker than that of the strict consistency model because the sequential consistency model does not guarantee that a read operation on a particular memory address always returns the same value as written by the most recent write operation to that address. As a consequence, with a sequentially consistent memory, running a program twice may not give the same result in the absence of explicit synchronization operations.

A DSM system supporting the sequential consistency model can be implemented by ensuring that no memory operation is started until all the previous ones have been completed. A sequentially consistent memory provides one-copy/single-copy semantics because all the processes sharing a memory location always see exactly the same contents stored in it. This is the most intuitively expected semantics for memory coherence. Therefore, sequential consistency is acceptable by most applications.

3.6.3.2 Weak Consistency Model

The weak consistency model is designed to attempt to avoid the costs of sequential consistency on multiprocessors, while retaining the effect of sequential consistency. This model takes advantage of the following two characteristics common to many applications:

1. It is not necessary to show the change in memory done by every write operation to other processes. The results of several write operations can be combined and sent to other processes only when they need it. For example, when a process executes in a critical section, other processes are not supposed to see the changes made by the process to the memory until the process exits from the critical section. In this case, all changes made to the memory by the process while it is in its critical section need be made visible to other processes only at the time when the process exits from the critical section.
2. Isolated accesses to shared variables are rare. That is, in many applications, a process makes several accesses to a set of shared variables and then no access at all to the variables in this set for a long time.

Both characteristics imply that better performance can be achieved if consistency is enforced on a group of memory reference operations rather than on individual memory reference operations. This is exactly the basic idea behind the weak consistency model.

The main problem in implementing this idea is determining how the system can know that it is time to show the changes performed by a process to other processes since this time is different for different applications. Since there is no way for the system to know this on its own, the programmers are asked to tell this to the system for their applications. For this, a DSM system that supports the weak consistency model uses a special variable called a synchronization variable. The operations on it are used to synchronize memory. That is, when a synchronization variable is accessed by a process, the entire (shared) memory is synchronized by making all the changes to the memory made by all processes visible to all other processes. Note that memory synchronization in a DSM system will involve propagating memory updates done at a node to all other nodes having a copy of the same memory addresses.

3.6.3.3 Release Consistency Model

In the weak consistency model the entire (shared) memory is synchronized when a synchronization variable is accessed by a process, and memory synchronization basically involves the following operations:

1. All changes made to the memory by the process are propagated to other nodes.

2. All changes made to the memory by other processes are propagated from other nodes to the process's node.

A closer observation shows that this is not really necessary because the first operation need only be performed when the process exits from a critical section and the second operation need only be performed when the process enters a critical section. Since a single synchronization variable is used in the weak consistency model, the system cannot know whether a process accessing a synchronization variable is entering a critical section or exiting from a critical section. Therefore, both the first and second operations are performed on every access to a synchronization variable by a process. For better performance and development of the weak consistency model, the *release consistency model* was designed to provide a mechanism to clearly tell the system whether a process is entering a critical section or exiting from a critical section so that the system can decide and perform only either the first or the second operation when a synchronization variable is accessed by a process. This is achieved by using two synchronization variables (called *acquire* and *release*) instead of a single synchronization variable. *Acquire* is used by a process to tell the system that it is about to enter a critical section, so that the system performs only the second operation when this variable is accessed. On the other hand, *release* is used by a process to tell the system that it has just exited from a critical section, so that the system only performs the first operation when the variable is accessed. Programmers are responsible for putting acquire and release at suitable places in their programs.

3.6.3.4 Discussion

There still are other consistency models including:

Causal consistency: Reads and writes may be related by the happened-before relationship. This is defined to hold between memory operations and when either (a) they are made by the same process; (b) a process reads a value written by another process; or (c) there exists a sequence of such operations linking the two operations. The model's constraint is that the value returned by a read must be consistent with the happened-before relationship.

Processor consistency: The memory is both coherent and adheres to the pipelined RAM model (see beblow). The simplest way to think of processor consistency is that the memory is coherent and that all processes agree on the ordering of any two write accesses made by the same process – that is, they agree with its program order.

Pipelined RAM: All processors agree on the order of writes issued by any given processor.

Among the consistency models described above, the most commonly used model in DSM systems is the sequential consistency model because it can be implemented, it supports the most intuitively expected semantics for memory coherence, and it does not impose any extra burden on the programmers. Another important reason for its popularity is that a sequential consistent DSM system allows existing

multiprocessor programs to be run on multicomputer architectures without modification. This is because programs written for multiprocessors normally assume that memory is sequentially consistent. However, it is very restrictive and hence suffers from the drawback of low consistency. Therefore, several DSM systems are designed to use other consistency models that are weaker than sequential consistency.

Weak consistency, release consistency and some of the other consistency models weaker than sequential consistency and using explicit sunchronization variables appear to be more promising for use in DSM design because they provide better concurrency and also support the intuitively expected semantics. It does not seem to be a significant disadvantage of these consistency models that synchronization operations need to be known to the DSM run-time – as long as those supplied by the system are sufficiently powerful to meet the needs of programmers. Hence the only problem with these consistency models is that they require the programmers to use the synchronization variables properly. This imposes some burden on the programmers.

3.7 Summary

In this chapter 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 non-blocking, 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 raising exceptions. Two very important aspects are presented for group communication: semantics of message delivery and message response, and message ordering. These aspects strongly influence quality of and cost of programming and application development.

In addition to message-passing paradigm, we also discussed the distributed shared memory systems. Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory. DSM is primarily a tool for parallel applications or for any distributed application or group of applications in which individual shared data items can be accessed directly. Various design and implementation issues about DSM have been discussed in this chapter. Among them, we focused on consistency models since a DSM system is a replication system and consistency is the most important concern in such a system. Several consistency models are presented, of these, the sequential consistency model is most commonly used.

Exercises

3.1. What factors influence the performance of a communication facility? 3.1

3.2. What are structured and unstructured messages? 3.2.1

3.3. What are the basic message-passing primitives? Describe how they work. 3.2.2

3.4. Discuss advantages and disadvantages of direct communication. 3.2.2

3.5. Why is the message queue associated with a port finite length? 3.2.2

3.6. What is the difference between blocking and nonblocking send primitives based on the first criterion? 3.2.2

3.7. What are the buffered and unbuffered primitives? Compare their features. 3.2.2

3.8. Describe the difference between at-least-once semantics and exactly-once semantics when dealing with multiple requests. 3.2.2

3.9. What is the RPC? What features does it have? 3.3.1

3.10. What are called by-value parameter passing and by-reference parameter passing? 3.3.3

3.11. Describe how to marshal data from one computer to another computer. 3.3.3

3.12. What are the basic aspects of client server binding? 3.3.4

3.13. Compare the difference between message passing and RPC. 3.4

3.14. Why should we use a group? What structures do groups have in distributed computing? 3.5.1

3.15. What operations can be used in group memebership management? What happens if failures occur in a group? 3.5.2

3.16. What's a DSM system? What is its purpose? 3.6.1

3.17. Why do we use page size as the block size of a DSM system? 3.6.2

3.18. What problem may a DSM system with sequential consistency model have? 3.6.3