CHAPTER 3

Message Passing

3.1 INTRODUCTION

A *process* is a program in execution. When we say that two computers of a distributed system are communicating with each other, we mean that two processes, one running on each computer, are in communication with each other. In a distributed system, processes executing on different computers often need to communicate with each other to achieve some common goal. For example, each computer of a distributed system may have a *resource manager* process to monitor the current status of usage of its local resources, and the resource managers of all the computers might communicate with each other from time to time to dynamically balance the system load among all the computers. Therefore, a distributed operating system needs to provide interprocess communication (IPC) mechanisms to facilitate such communication activities.

Interprocess communication basically requires information sharing among two or more processes. The two basic methods for information sharing are as follows:

1. Original sharing, or shared-data approach
2. Copy sharing, or message-passing approach

In the shared-data approach, the information to be shared is placed in a common memory area that is accessible to all the processes involved in an IPC. The shared-data

paradigm gives the conceptual communication pattern illustrated in Figure 3.1(*a*). On the other hand, in the message-passing approach, the information to be shared is physically copied from the sender process's address space to the address spaces of all the receiver processes, and this is done by transmitting the data to be copied in the form of messages (a *message* is a block of information). The message-passing paradigm gives the conceptual communication pattern illustrated in Figure 3.1(*b*). That is, the communicating processes interact directly with each other.

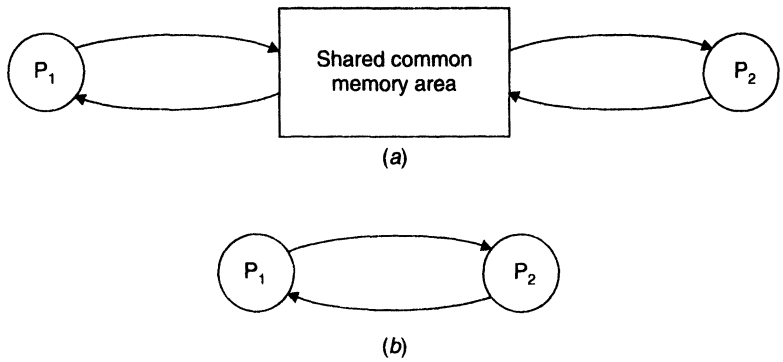


Fig. 3.1 The two basic interprocess communication paradigms: (*a*) The shared-data approach. (*b*) The message-passing approach.

Since computers in a network do not share memory, processes in a distributed system normally communicate by exchanging messages rather than through shared data. Therefore, message passing is the basic IPC mechanism in distributed systems.

A *message-passing system* is a subsystem of a distributed operating system that provides a set of message-based IPC protocols and does so by shielding the details of complex network protocols and multiple heterogeneous platforms from programmers. It enables processes to communicate by exchanging messages and allows programs to be written by using simple communication primitives, such as *send* and *receive*. It serves as a suitable infrastructure for building other higher level IPC systems, such as remote procedure call (RPC; see Chapter 4) and distributed shared memory (DSM; see Chapter 5).

3.2 DESIRABLE FEATURES OF A GOOD MESSAGE-PASSING SYSTEM

3.2.1 Simplicity

A message-passing system should be simple and easy to use. It must be straightforward to construct new applications and to communicate with existing ones by using the primitives provided by the message-passing system. It should also be possible for a

programmer to designate the different modules of a distributed application and to send and receive messages between them in a way as simple as possible without the need to worry about the system and/or network aspects that are not relevant for the application level. Clean and simple semantics of the IPC protocols of a message-passing system make it easier to build distributed applications and to get them right.

3.2.2 Uniform Semantics

In a distributed system, a message-passing system may be used for the following two types of interprocess communication:

1. *Local communication*, in which the communicating processes are on the same node
2. *Remote communication*, in which the communicating processes are on different nodes

An important issue in the design of a message-passing system is that the semantics of remote communications should be as close as possible to those of local communications. This is an important requirement for ensuring that the message-passing system is easy to use.

3.2.3 Efficiency

Efficiency is normally a critical issue for a message-passing system to be acceptable by the users. If the message-passing system is not efficient, interprocess communication may become so expensive that application designers will strenuously try to avoid its use in their applications. As a result, the developed application programs would be distorted. An IPC protocol of a message-passing system can be made efficient by reducing the number of message exchanges, as far as practicable, during the communication process. Some optimizations normally adopted for efficiency include the following:

- Avoiding the costs of establishing and terminating connections between the same pair of processes for each and every message exchange between them
- Minimizing the costs of maintaining the connections
- Piggybacking of acknowledgment of previous messages with the next message during a connection between a sender and a receiver that involves several message exchanges

3.2.4 Reliability

Distributed systems are prone to different catastrophic events such as node crashes or communication link failures. Such events may interrupt a communication that was in progress between two processes, resulting in the loss of a message. A reliable IPC protocol can cope with failure problems and guarantees the delivery of a message. Handling of lost

messages usually involves acknowledgments and retransmissions on the basis of timeouts.

Another issue related to reliability is that of duplicate messages. Duplicate messages may be sent in the event of failures or because of timeouts. A reliable IPC protocol is also capable of detecting and handling duplicates. Duplicate handling usually involves generating and assigning appropriate sequence numbers to messages.

A good message-passing system must have IPC protocols to support these reliability features.

3.2.5 Correctness

A message-passing system often has IPC protocols for group communication that allow a sender to send a message to a group of receivers and a receiver to receive messages from several senders. Correctness is a feature related to IPC protocols for group communication. Although not always required, correctness may be useful for some applications. Issues related to correctness are as follows [Navratnam et al. 1988]:

- Atomicity
- Ordered delivery
- Survivability

Atomicity ensures that every message sent to a group of receivers will be delivered to either all of them or none of them. Ordered delivery ensures that messages arrive at all receivers in an order acceptable to the application. Survivability guarantees that messages will be delivered correctly despite partial failures of processes, machines, or communication links. Survivability is a difficult property to achieve.

3.2.6 Flexibility

Not all applications require the same degree of reliability and correctness of the IPC protocols. For example, in adaptive routing, it may be necessary to distribute the information regarding queuing delays in different parts of the network. A broadcast protocol could be used for this purpose. However, if a broadcast message is late in coming, due to communication failures, it might just as well not arrive at all as it will soon be outdated by a more recent one anyway. Similarly, many applications do not require atomicity or ordered delivery of messages. For example, a client may multicast a request message to a group of servers and offer the job to the first server that replies. Obviously, atomicity of message delivery is not required in this case. Thus the IPC protocols of a message-passing system must be flexible enough to cater to the various needs of different applications. That is, the IPC primitives should be such that the users have the flexibility to choose and specify the types and levels of reliability and correctness requirements of their applications. Moreover, IPC primitives must also have the flexibility to permit any kind of control flow between the cooperating processes, including synchronous and asynchronous *send/receive*.

3.2.7 Security

A good message-passing system must also be capable of providing a secure end-to-end communication. That is, a message in transit on the network should not be accessible to any user other than those to whom it is addressed and the sender. Steps necessary for secure communication include the following:

- Authentication of the receiver(s) of a message by the sender
- Authentication of the sender of a message by its receiver(s)
- Encryption of a message before sending it over the network

These issues will be described in detail in Chapter 11.

3.2.8 Portability

There are two different aspects of portability in a message-passing system:

1. The message-passing system should itself be portable. That is, it should be possible to easily construct a new IPC facility on another system by reusing the basic design of the existing message-passing system.

2. The applications written by using the primitives of the IPC protocols of the message-passing system should be portable. This requires that heterogeneity must be considered while designing a message-passing system. This may require the use of an external data representation format for the communications taking place between two or more processes running on computers of different architectures. The design of high-level primitives for the IPC protocols of a message-passing system should be done so as to hide the heterogeneous nature of the network.

3.3 ISSUES IN IPC BY MESSAGE PASSING

A message is a block of information formatted by a sending process in such a manner that it is meaningful to the receiving process. It consists of a fixed-length header and a variable-size collection of typed data objects. As shown in Figure 3.2, the header usually consists of the following elements:

- *Address.* It contains characters that uniquely identify the sending and receiving processes in the network. Thus, this element has two parts—one part is the sending process address and the other part is the receiving process address.
- *Sequence number.* This is the message identifier (ID), which is very useful for identifying lost messages and duplicate messages in case of system failures.

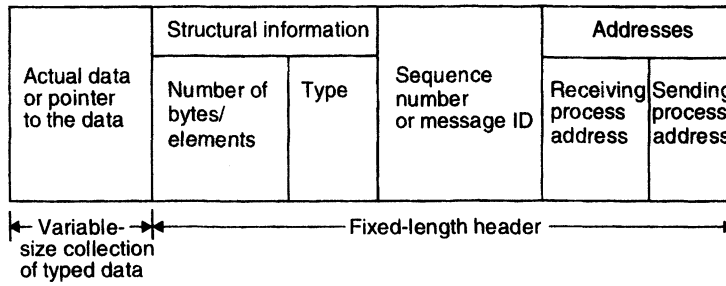


Fig. 3.2 A typical message structure.

- *Structural information.* This element also has two parts. The *type* part specifies whether the data to be passed on to the receiver is included within the message or the message only contains a pointer to the data, which is stored somewhere outside the contiguous portion of the message. The second part of this element specifies the length of the variable-size message data.

In a message-oriented IPC protocol, the sending process determines the actual contents of a message and the receiving process is aware of how to interpret the contents. Special primitives are explicitly used for sending and receiving the messages. Therefore, in this method, the users are fully aware of the message formats used in the communication process and the mechanisms used to send and receive messages.

In the design of an IPC protocol for a message-passing system, the following important issues need to be considered:

- Who is the sender?
- Who is the receiver?
- Is there one receiver or many receivers?
- Is the message guaranteed to have been accepted by its receiver(s)?
- Does the sender need to wait for a reply?
- What should be done if a catastrophic event such as a node crash or a communication link failure occurs during the course of communication?
- What should be done if the receiver is not ready to accept the message: Will the message be discarded or stored in a buffer? In the case of buffering, what should be done if the buffer is full?
- If there are several outstanding messages for a receiver, can it choose the order in which to service the outstanding messages?

These issues are addressed by the semantics of the set of communication primitives provided by the IPC protocol. A general description of the various ways in which these issues are addressed by message-oriented IPC protocols is presented below.

3.4 SYNCHRONIZATION

A central issue in the communication structure is the synchronization imposed on the communicating processes by the communication primitives. The semantics used for synchronization may be broadly classified as *blocking* and *nonblocking* types. A primitive is said to have nonblocking semantics if its invocation does not block the execution of its invoker (the control returns almost immediately to the invoker); otherwise a primitive is said to be of the blocking type. The synchronization imposed on the communicating processes basically depends on one of the two types of semantics used for the *send* and *receive* primitives.

In case of a blocking *send* primitive, after execution of the *send* statement, the sending process is blocked until it receives an acknowledgment from the receiver that the message has been received. On the other hand, for nonblocking *send* primitive, after execution of the *send* statement, the sending process is allowed to proceed with its execution as soon as the message has been copied to a buffer.

In the case of a blocking *receive* primitive, after execution of the *receive* statement, the receiving process is blocked until it receives a message. On the other hand, for a nonblocking *receive* primitive, the receiving process proceeds with its execution after execution of the *receive* statement, which returns control almost immediately just after telling the kernel where the message buffer is.

An important issue in a nonblocking *receive* primitive is how the receiving process knows that the message has arrived in the message buffer. One of the following two methods is commonly used for this purpose:

1. *Polling.* In this method, a *test* primitive is provided to allow the receiver to check the buffer status. The receiver uses this primitive to periodically poll the kernel to check if the message is already available in the buffer.

2. *Interrupt.* In this method, when the message has been filled in the buffer and is ready for use by the receiver, a software interrupt is used to notify the receiving process. This method permits the receiving process to continue with its execution without having to issue unsuccessful *test* requests. Although this method is highly efficient and allows maximum parallelism, its main drawback is that user-level interrupts make programming difficult [Tanenbaum 1995].

A variant of the nonblocking *receive* primitive is the conditional *receive* primitive, which also returns control to the invoking process almost immediately, either with a message or with an indicator that no message is available.

In a blocking *send* primitive, the sending process could get blocked forever in situations where the potential receiving process has crashed or the sent message has been lost on the network due to communication failure. To prevent this situation, blocking *send* primitives often use a timeout value that specifies an interval of time after which the *send* operation is terminated with an error status. Either the timeout value may be a default value or the users may be provided with the flexibility to specify it as a parameter of the *send* primitive.

A timeout value may also be associated with a blocking *receive* primitive to prevent the receiving process from getting blocked indefinitely in situations where the potential sending process has crashed or the expected message has been lost on the network due to communication failure.

When both the *send* and *receive* primitives of a communication between two processes use blocking semantics, the communication is said to be *synchronous;* otherwise it is *asynchronous*. That is, for synchronous communication, the sender and the receiver must be synchronized to exchange a message. This is illustrated in Figure 3.3. Conceptually, the sending process sends a message to the receiving process, then waits for an acknowledgment. After executing the *receive* statement, the receiver remains blocked until it receives the message sent by the sender. On receiving the message, the receiver sends an acknowledgment message to the sender. The sender resumes execution only after receiving this acknowledgment message.

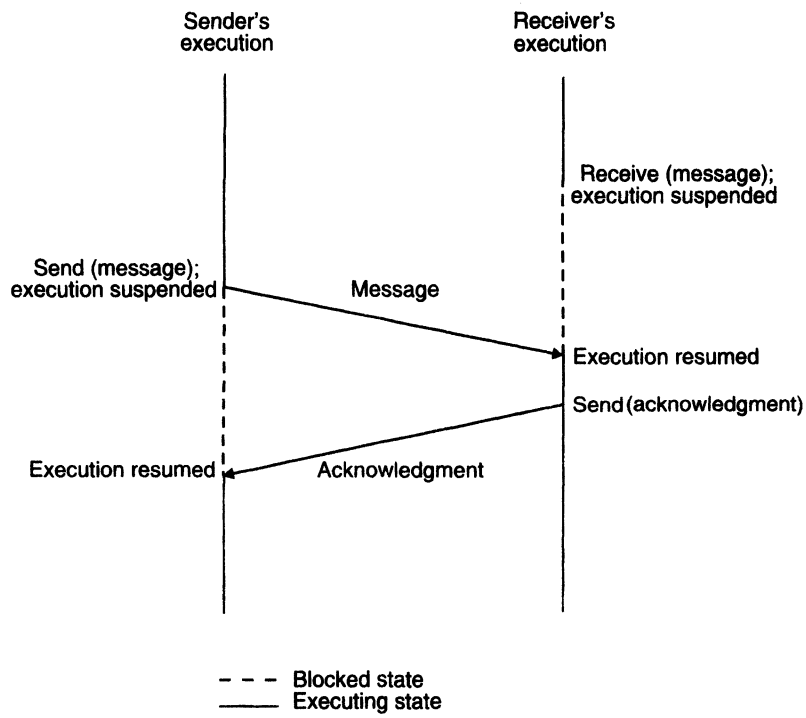


Fig. 3.3 Synchronous mode of communication with both *send* and *receive* primitives having blocking-type semantics.

As compared to asynchronous communication, synchronous communication is simple and easy to implement. It also contributes to reliability because it assures the sending process that its message has been accepted before the sending process resumes execution. As a result, if the message gets lost or is undelivered, no backward error recovery is

necessary for the sending process to establish a consistent state and resume execution [Shatz 1984]. However, the main drawback of synchronous communication is that it limits concurrency and is subject to communication deadlocks (communication deadlock is described in Chapter 6). It is less flexible than asynchronous communication because the sending process always has to wait for an acknowledgment from the receiving process even when this is not necessary. In a system that supports multiple threads in a single process (see Chapter 8), the blocking primitives can be used without the disadvantage of limited concurrency. How this is made possible is explained in Chapter 8.

A flexible message-passing system usually provides both blocking and nonblocking primitives for *send* and *receive* so that users can choose the most suitable one to match the specific needs of their applications.

3.5 BUFFERING

Messages can be transmitted from one process to another by copying the body of the message from the address space of the sending process to the address space of the receiving process (possibly via the address spaces of the kernels of the sending and receiving computers). In some cases, the receiving process may not be ready to receive a message transmitted to it but it wants the operating system to save that message for later reception. In these cases, the operating system will rely on the receiver having a buffer in which messages can be stored prior to the receiving process executing specific code to receive the message.

In interprocess communication, the message-buffering strategy is strongly related to synchronization strategy. The synchronous and asynchronous modes of communication correspond respectively to the two extremes of buffering: a *null buffer*, or *no buffering*, and a *buffer with unbounded capacity*. Other two commonly used buffering strategies are *single-message* and *finite-bound*, or *multiple-message*, buffers. These four types of buffering strategies are described below.

3.5.1 Null Buffer (or No Buffering)

In case of no buffering, there is no place to temporarily store the message. Hence one of the following implementation strategies may be used:

1. The message remains in the sender process's address space and the execution of the *send* is delayed until the receiver executes the corresponding *receive*. To do this, the sender process is backed up and suspended in such a way that when it is unblocked, it starts by reexecuting the *send* statement. When the receiver executes *receive*, an acknowledgment is sent to the sender's kernel saying that the sender can now send the message. On receiving the acknowledgment message, the sender is unblocked, causing the *send* to be executed once again. This time, the message is successfully transferred from the sender's address space to the receiver's address space because the receiver is waiting to receive the message.

2. The message is simply discarded and the timeout mechanism is used to resend the message after a timeout period. That is, after executing *send*, the sender process waits for

an acknowledgment from the receiver process. If no acknowledgment is received within the timeout period, it assumes that its message was discarded and tries again hoping that this time the receiver has already executed *receive*. The sender may have to try several times before succeeding. The sender gives up after retrying for a predecided number of times.

As shown in Figure 3.4(*a*), in the case of no buffering, the logical path of message transfer is directly from the sender's address space to the receiver's address space, involving a single copy operation.

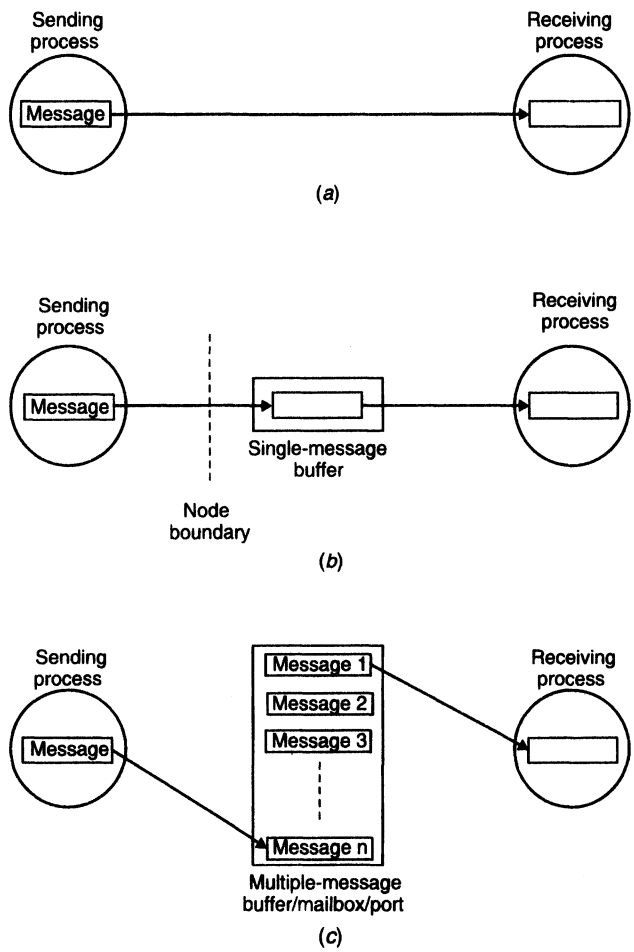


Fig. 3.4 The three types of buffering strategies used in interprocess comunication mechanisms: (*a*) Message transfer in synchronous send with no buffering strategy (only one copy operation is needed). (*b*) Message transfer in synchronous send with single-message buffering strategy (two copy operations are needed). (*c*) Message transfer in asynchronous send with multiple-message buffering strategy (two copy operations are needed).

3.5.2 Single-Message Buffer

The null buffer strategy is generally not suitable for synchronous communication between two processes in a distributed system because if the receiver is not ready, a message has to be transferred two or more times, and the receiver of the message has to wait for the entire time taken to transfer the message across the network. In a distributed system, message transfer across the network may require significant time in some cases. Therefore, instead of using the null buffer strategy, synchronous communication mechanisms in network/distributed systems use a single-message buffer strategy. In this strategy, a buffer having a capacity to store a single message is used on the receiver's node. This is because in systems based on synchronous communication, an application module may have at most one message outstanding at a time. The main idea behind the single-message buffer strategy is to keep the message ready for use at the location of the receiver. Therefore, in this method, the request message is buffered on the receiver's node if the receiver is not ready to receive the message. The message buffer may either be located in the kernel's address space or in the receiver process's address space. As shown in Figure 3.4(*b*), in this case the logical path of message transfer involves two copy operations.

3.5.3 Unbounded-Capacity Buffer

In the asynchronous mode of communication, since a sender does not wait for the receiver to be ready, there may be several pending messages that have not yet been accepted by the receiver. Therefore, an unbounded-capacity message buffer that can store all unreceived messages is needed to support asynchronous communication with the assurance that all the messages sent to the receiver will be delivered.

3.5.4 Finite-Bound (or Multiple-Message) Buffer

Unbounded capacity of a buffer is practically impossible. Therefore, in practice, systems using asynchronous mode of communication use finite-bound buffers, also known as multiple-message buffers. When the buffer has finite bounds, a strategy is also needed for handling the problem of a possible buffer overflow. The buffer overflow problem can be dealt with in one of the following two ways:

1. *Unsuccessful communication.* In this method, message transfers simply fail whenever there is no more buffer space. The *send* normally returns an error message to the sending process, indicating that the message could not be delivered to the receiver because the buffer is full. Unfortunately, the use of this method makes message passing less reliable.

2. *Flow-controlled communication.* The second method is to use flow control, which means that the sender is blocked until the receiver accepts some messages, thus creating

space in the buffer for new messages. This method introduces a synchronization between the sender and the receiver and may result in unexpected deadlocks. Moreover, due to the synchronization imposed, the asynchronous send does not operate in the truly asynchronous mode for all *send* commands.

The amount of buffer space to be allocated in the bounded-buffer strategy is a matter of implementation. In the most often used approach, a *create_buffer* system call is provided to the users. This system call, when executed by a receiver process, creates a buffer (sometimes called a *mailbox* or *port*) of a size specified by the receiver. The receiver's mailbox may be located either in the kernel's address space or in the receiver process's address space. If it is located in the kernel's address space, mailboxes are a system resource that must be allocated to processes as and when required. This will tend to limit the number of messages that an individual process may keep in its mailbox. On the other hand, if the mailbox is located in the receiver process's address space, the operating system will have to rely on the process allocating an appropriate amount of memory, protecting the mailbox from mishaps, and so on.

As shown in Figure 3.4(*c*), in the case of asynchronous *send* with bounded-buffer strategy, the message is first copied from the sending process's memory into the receiving process's mailbox and then copied from the mailbox to the receiver's memory when the receiver calls for the message. Therefore, in this case also, the logical path of message transfer involves two copy operations.

Although message communication based on multiple-message-buffering capability provides better concurrency and flexibility as compared to no buffering or single-message buffering, it is more complex to design and use. This is because of the extra work and overhead involved in the mechanisms needed for the creation, deletion, protection, and other issues involved in buffer management.

3.6 MULTIDATAGRAM MESSAGES

Almost all networks have an upper bound on the size of data that can be transmitted at a time. This size is known as the *maximum transfer unit* (*MTU*) of a network. A message whose size is greater than the MTU has to be fragmented into multiples of the MTU, and then each fragment has to be sent separately. Each fragment is sent in a packet that has some control information in addition to the message data. Each packet is known as a *datagram*. Messages smaller than the MTU of the network can be sent in a single packet and are known as *single-datagram messages*. On the other hand, messages larger than the MTU of the network have to be fragmented and sent in multiple packets. Such messages are known as *multidatagram messages*. Obviously, different packets of a multidatagram message bear a sequential relationship to one another. The disassembling of a multidatagram message into multiple packets on the sender side and the reassembling of the packets on the receiver side is usually the responsibility of the message-passing system.

3.7 ENCODING AND DECODING OF MESSAGE DATA

A message data should be meaningful to the receiving process. This implies that, ideally, the structure of program objects should be preserved while they are being transmitted from the address space of the sending process to the address space of the receiving process. This obviously is not possible in a heterogeneous system in which the sending and receiving processes are on computers of different architectures. However, even in homogeneous systems, it is very difficult to achieve this goal mainly because of two reasons:

1. An absolute pointer value loses its meaning when transferred from one process address space to another. Therefore, such program objects that use absolute pointer values cannot be transferred in their original form, and some other form of representation must be used to transfer them. For example, to transmit a tree object, each element of the tree must be copied in a leaf record and properly aligned in some fixed order in a buffer before it can be sent to another process. The leaf records themselves have no meaning in the address space of the receiving process, but the tree can be regenerated easily from them. To facilitate such regeneration, object-type information must be passed between the sender and receiver, indicating not only that a tree object is being passed but also the order in which the leaf records are aligned. This process of flattening and shaping of tree objects also extends to other structured program objects, such as linked lists.

2. Different program objects occupy varying amount of storage space. To be meaningful, a message must normally contain several types of program objects, such as long integers, short integers, variable-length character strings, and so on. In this case, to make the message meaningful to the receiver, there must be some way for the receiver to identify which program object is stored where in the message buffer and how much space each program object occupies.

Due to the problems mentioned above in transferring program objects in their original form, they are first converted to a stream form that is suitable for transmission and placed into a message buffer. This conversion process takes place on the sender side and is known as *encoding* of a message data. The encoded message, when received by the receiver, must be converted back from the stream form to the original program objects before it can be used. The process of reconstruction of program objects from message data on the receiver side is known as *decoding* of the message data.

One of the following two representations may be used for the encoding and decoding of a message data:

1. *In tagged representation* the type of each program object along with its value is encoded in the message. In this method, it is a simple matter for the receiving process to check the type of each program object in the message because of the self-describing nature of the coded data format.

2. *In untagged representation* the message data only contains program objects. No information is included in the message data to specify the type of each program object. In

this method, the receiving process must have a prior knowledge of how to decode the received data because the coded data format is not self-describing.

The untagged representation is used in Sun XDR (eXternal Data Representation) [Sun 1990] and Courier [Xerox 1981], whereas the tagged representation is used in the ASN.1 (Abstract Syntax Notation) standard [CCITT 1985] and the Mach distributed operating system [Fitzgerald and Rashid 1986].

In general, tagged representation is more expensive than untagged representation, both in terms of the quantity of data transferred and the processing time needed at each side to encode and decode the message data. No matter which representation is used, both the sender and the receiver must be fully aware of the format of data coded in the message. The sender possesses the encoding routine for the coded data format and the receiver possesses the corresponding decoding routine. The encoding and decoding operations are perfectly symmetrical in the sense that decoding exactly reproduces the data that was encoded, allowing for differences in local representations. Sometimes, a receiver may receive a badly encoded data, such as encoded data that exceeds a maximum-length argument. In such a situation, the receiver cannot successfully decode the received data and normally returns an error message to the sender indicating that the data is not intelligible.

3.8 PROCESS ADDRESSING

Another important issue in message-based communication is addressing (or naming) of the parties involved in an interaction: To whom does the sender wish to send its message and, conversely, from whom does the receiver wish to accept a message? For greater flexibility, a message-passing system usually supports two types of process addressing:

1. *Explicit addressing.* The process with which communication is desired is explicitly named as a parameter in the communication primitive used. Primitives (*a*) and (*b*) of Figure 3.5 require explicit process addressing.

2. *Implicit addressing.* A process willing to communicate does not explicitly name a process for communication. Primitives (*c*) and (*d*) of Figure 3.5 support implicit process addressing. In primitive (*c*), the sender names a service instead of a process. This type of primitive is useful in client-server communications when the client is not concerned with which particular server out of a set of servers providing the service desired by the client actually services its request. This type of process addressing is also known as *functional addressing* because the address used in the communication primitive identifies a service rather than a process.

On the other hand, in primitive (*d*), the receiver is willing to accept a message from any sender. This type of primitive is again useful in client-server communications when the server is meant to service requests of all clients that are authorized to use its service.

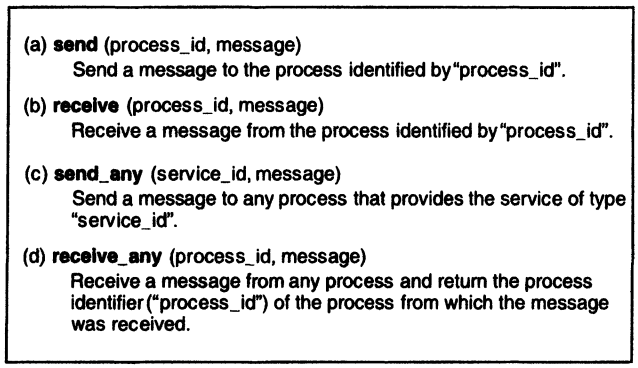


Fig. 3.5 Primitives for explicit and implicit addressing of processes.

With the two basic types of process addressing used in communication primitives, we now look at the commonly used methods for process addressing.

A simple method to identify a process is by a combination of *machine_id* and *local_id*, such as *machine_id@local_id*. The *local_id* part is a process identifier, or a port identifier of a receiving process, or something else that can be used to uniquely identify a process on a machine. A process willing to send a message to another process specifies the receiving process's address in the form *machine_id@local_id*. The *machine_id* part of the address is used by the sending machine's kernel to send the message to the receiving process's machine, and the *local_id* part of the address is then used by the kernel of the receiving process's machine to forward the message to the process for which it is intended. This method of process addressing is used in Berkeley UNIX with 32-bit Internet addresses for *machine_id* and 16-bit numbers for *local_id*.

An attractive feature of this method is that no global coordination is needed to generate systemwide unique process identifiers because *local_ids* need to be unique only for one machine and can be generated locally without consultation with other machines. However, a drawback of this method is that it does not allow a process to migrate from one machine to another if such a need arises. For instance, one or more processes of a heavily loaded machine may be migrated to a lightly loaded machine to balance the overall system load.

To overcome the limitation of the above method, processes can be identified by a combination of the following three fields: *machine_id, local_id, and machine_id.*

1. The first field identifies the node on which the process is created
2. The second field is a local indentifier generated by the node on which the process is created
3. The third field identifies the last known location (node) of the process.

During the lifetime of a process, the values of the first two fields of its identifier never change; the third field, however, may. This method of process addressing is known as *link-based process addressing*. For this method to work properly, when a process is

migrated from its current node to a new node, a link information (process identifier with the value of its third field equal to the *machine_id* of the process's new node) is left on its previous node, and on the new node, a new *local_id* is assigned to the process, and its process identifier and the new *local_id* is entered in a mapping table maintained by the kernel of the new node for all processes created on another node but running on this node. Note that the value of the third field of a process identifier is set equal to its first field when the process is created.

A process willing to send a message to another process specifies the receiving process's address in the form, say, *machine_id@local_id@machine_id*. The kernel of the sending machine delivers the message to the machine whose *machine_id* is specified in the third field of the receiving process's address. If the value of the third field is equal to the first field, the message will be sent to the node on which the process was created. If the receiving process was not migrated, the message is delivered to it by using the *local_id* information in the process identifier. On the other hand, if the receiving process was migrated, the link information left for it on that node is used to forward the message to the node to which the receiving process was migrated from this node. In this manner, the message may get forwarded from one node to another several times before it reaches the current node of the receiving process. When the message reaches the current node of the receiving process, the kernel of that node extracts the process identifiers of the sending and receiving processes from the message. The first two fields of the process identifier of the receiving process are used as its unique identifier to extract its *local_id* from the mapping table and then to deliver the message to the proper process. On the other hand, the process identifier of the sending process is used to return to it the current location of the receiving process. The sending process uses this information to update the value of the third field of the receiving process's identifier, which it caches in a local cache, so that from the next time the sending process can directly send a message for the receiving process to this location of the receiving process instead of sending it via the node on which the receiving process was created. A variant of this method of process addressing is used in DEMOS/MP [Miller et al. 1987] and Charlotte [Artsy et al. 1987]. Although this method of process addressing supports the process migration facility, it suffers from two main drawbacks:

1. The overload of locating a process may be large if the process has migrated several times during its lifetime.
2. It may not be possible to locate a process if an intermediate node on which the process once resided during its lifetime is down.

Both process-addressing methods previously described are nontransparent due to the need to specify the machine identifier. The user is well aware of the location of the process (or at least the location on which the process was created). However, we saw in Chapter 1 that location transparency is one of the main goals of a distributed operating system. Hence a location-transparent process-addressing mechanism is more desirable for a message-passing system. A simple method to achieve this goal is to ensure that the systemwide unique identifier of a process does not contain an embedded machine identifier. A centralized process identifier allocator that maintains a counter can be used

for this purpose. When it receives a request for an identifier, it simply returns the current value of the counter and then increments it by 1. This scheme, however, suffers from the problems of poor reliability and poor scalability.

Another method to achieve the goal of location transparency in process addressing is to use a two-level naming scheme for processes. In this method, each process has two identifiers: a high-level name that is machine independent (an ASCII string) and a low-level name that is machine dependent (such as *machine_id@local_id*). A *name server* is used to maintain a mapping table that maps high-level names of processes to their low-level names. When this method of process addressing is used, a process that wants to send a message to another process specifies the high-level name of the receiving process in the communication primitive. The kernel of the sending machine first contacts the name server (whose address is well known to all machines) to get the low-level name of the receiving process from its high-level name. Using the low-level name, the kernel sends the message to the proper machine, where the receiving kernel delivers the message to the receiving process. The sending kernel also caches the high-level name to low-level name-mapping information of the receiving process in a local cache for future use, so that the name server need not be contacted when a message has to be sent again to the receiving process.

Notice that the name server approach allows a process to be migrated from one node to another without the need to change the code in the program of any process that wants to communicate with it. This is because when a process migrates its low-level identifier changes, and this change is incorporated in the name server's mapping table. However, the high-level name of the process remains unchanged.

The name server approach is also suitable for functional addressing. In this case, a high-level name identifies a service instead of a process, and the name server maps a service identifier to one or more processes that provide that service.

The name server approach also suffers from the problems of poor reliability and poor scalability because the name server is a centralized component of the system. One way to overcome these problems is to replicate the name server. However, this leads to extra overhead needed in keeping the replicas consistent.

3.9 FAILURE HANDLING

While a distributed system may offer potential for parallelism, it is also prone to partial failures such as a node crash or a communication link failure. As shown in Figure 3.6, during interprocess communication, such failures may lead to the following problems:

1. *Loss of request message.* This may happen either due to the failure of communication link between the sender and receiver or because the receiver's node is down at the time the request message reaches there.

2. *Loss of response message.* This may happen either due to the failure of communication link between the sender and receiver or because the sender's node is down at the time the response message reaches there.

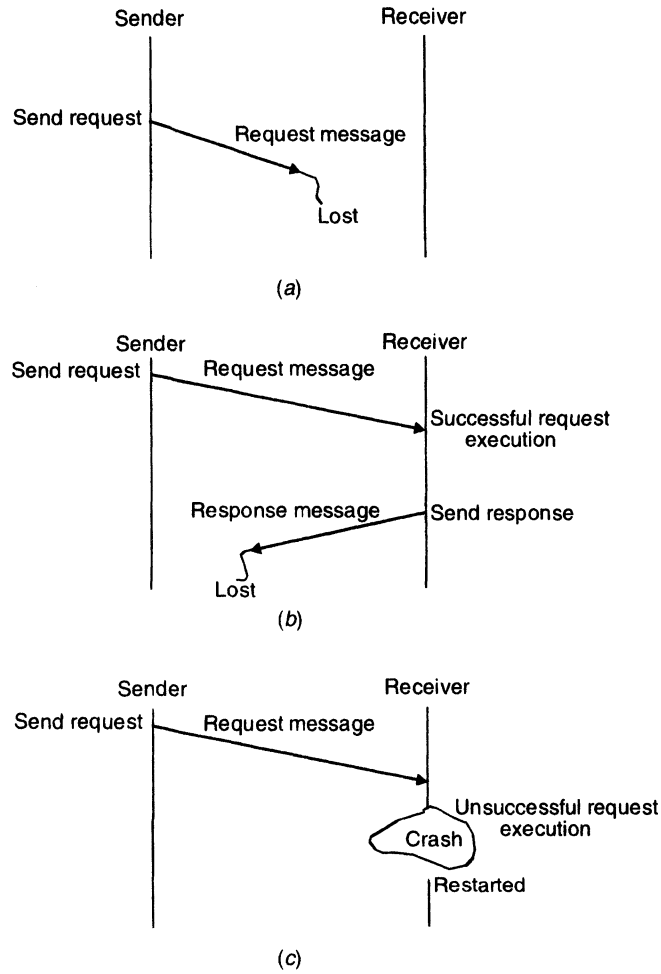


Fig. 3.6 Possible problems in IPC due to different types of system failures. (*a*) Request message is lost. (*b*) Response message is lost. (*c*) Receiver's computer crashed.

3. *Unsuccessful execution of the request.* This happens due to the receiver's node crashing while the request is being processed.

To cope with these problems, a reliable IPC protocol of a message-passing system is normally designed based on the idea of internal retransmissions of messages after timeouts and the return of an acknowledgment message to the sending machine's kernel by the receiving machine's kernel. That is, the kernel of the sending machine is responsible for retransmitting the message after waiting for a timeout period if no acknowledgment is received from the receiver's machine within this time. The kernel of the sending machine frees the sending process only when the acknowledgment is received. The time duration

for which the sender waits before retransmitting the request is normally slightly more than the approximate round-trip time between the sender and the receiver nodes plus the average time required for executing the request.

Based on the above idea, a four-message reliable IPC protocol for client-server communication between two processes works as follows (see Fig. 3.7):

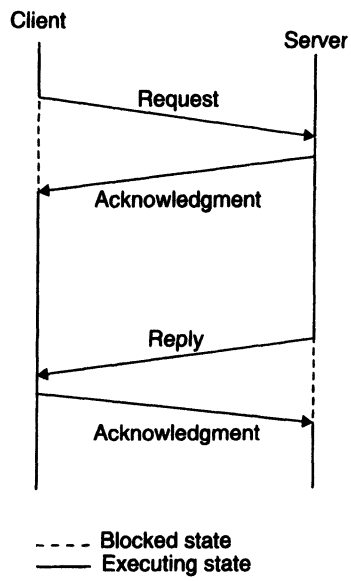


Fig. 3.7 The four-message reliable IPC protocol for client-server communication between two processes.

1. The client sends a request message to the server.

2. When the request message is received at the server's machine, the kernel of that machine returns an acknowledgment message to the kernel of the client machine. If the acknowledgment is not received within the timeout period, the kernel of the client machine retransmits the request message.

3. When the server finishes processing the client's request, it returns a reply message (containing the result of processing) to the client.

4. When the reply message is received at the client's machine, the kernel of that machine returns an acknowledgment message to the kernel of the server machine. If the acknowledgment message is not received within the timeout period, the kernel of the server machine retransmits the reply message.

In client-server communication, the result of the processed request is sufficient acknowledgment that the request message was received by the server. Based on this idea, a three-message reliable IPC protocol for client-server communication between two processes works as follows (see Fig. 3.8):

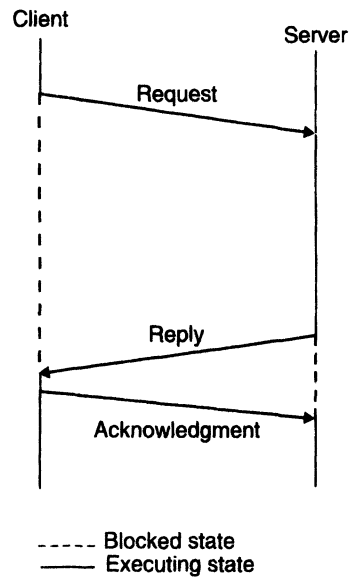


Fig. 3.8 The three-message reliable IPC protocol for client-server communication between two processes.

1. The client sends a request message to the server.

2. When the server finishes processing the client's request, it returns a reply message (containing the result of processing) to the client. The client remains blocked until the reply is received. If the reply is not received within the timeout period, the kernel of the client machine retransmits the request message.

3. When the reply message is received at the client's machine, the kernel of that machine returns an acknowledgment message to the kernel of the server machine. If the acknowledgment message is not received within the timeout period, the kernel of the server machine retransmits the reply message.

In the protocol of Figure 3.8, a problem occurs if a request processing takes a long time. If the request message is lost, it will be retransmitted only after the timeout period, which has been set to a large value to avoid unnecessary retransmissions of the request message. On the other hand, if the timeout value is not set properly taking into consideration the long time needed for request processing, unnecessary retransmissions of

the request message will take place. The following protocol may be used to handle this problem:

1. The client sends a request message to the server.

2. When the request message is received at the server's machine, the kernel of that machine starts a timer. If the server finishes processing the client's request and returns the reply message to the client before the timer expires, the reply serves as the acknowledgment of the request message. Otherwise, a separate acknowledgment is sent by the kernel of the server machine to acknowledge the request message. If an acknowledgment is not received within the timeout period, the kernel of the client machine retransmits the request message.

3. When the reply message is received at the client's machine, the kernel of that machine returns an acknowledgment message to the kernel of the server machine. If the acknowledgment message is not received within the timeout period, the kernel of the server machine retransmits the reply message.

Notice that the acknowledgment message from client to server machine in the protocol of Figure 3.8 is convenient but not a necessity. This is because if the reply message is lost, the request message will be retransmitted after timeout. The server can process the request once again and return the reply to the client. Therefore, a message-passing system may be designed to use the following two-message IPC protocol for client-server communication between two processes (see Fig. 3.9):

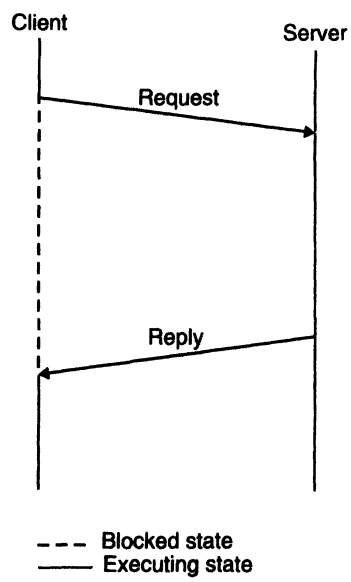


Fig. 3.9 The two-message IPC protocol used in many systems for client-server communication between two processes.

1. The client sends a request message to the server and remains blocked until a reply is received from the server.
2. When the server finishes processing the client's request, it returns a reply message (containing the result of processing) to the client. If the reply is not received within the timeout period, the kernel of the client machine retransmits the request message.

Based on the protocol of Figure 3.9, an example of failure handling during communication between two processes is shown in Figure 3.10. The protocol of Figure 3.9 is said to obey *at-least-once* semantics, which ensures that at least one execution of the receiver's operation has been performed (but possibly more). It is more appropriate to call

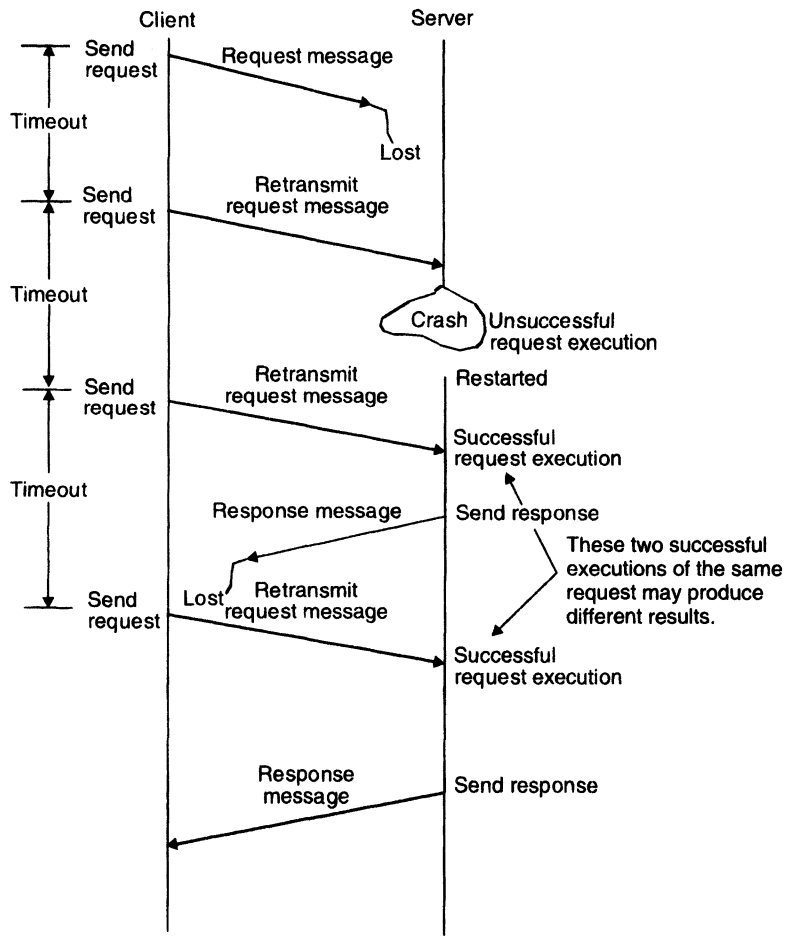


Fig. 3.10 An example of fault-tolerant communication between a client and a server.

this semantics the *last-one* semantics because the results of the last execution of the request are used by the sender, although earlier (abandoned) executions of the request may have had side effects that survived the failure. As explained later, this semantics may not be acceptable to several applications.

3.9.1 Idempotency and Handling of Duplicate Request Messages

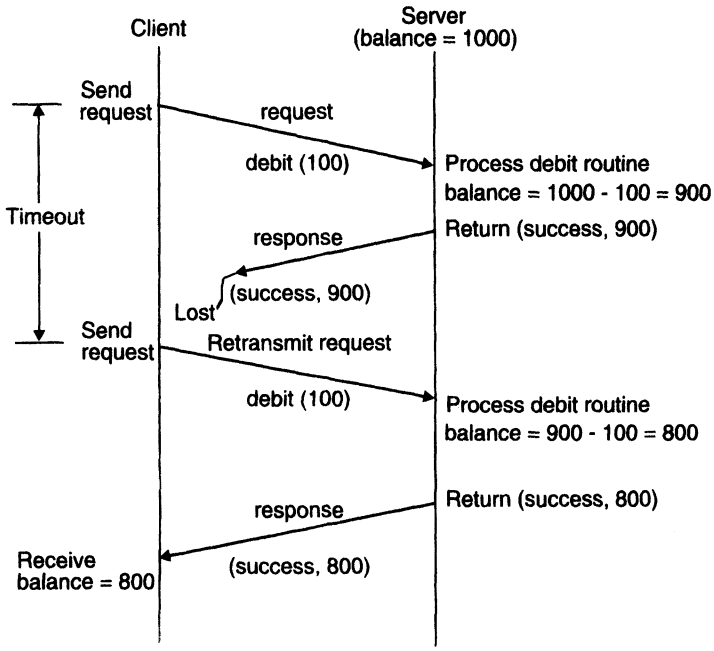
Idempotency basically means "repeatability." That is, an idempotent operation produces the same results without any side effects no matter how many times it is performed with the same arguments. An example of an idempotent routine is a simple *GetSqrt* procedure for calculating the square root of a given number. For example, *GetSqrt*(64) always returns 8.

On the other hand, operations that do not necessarily produce the same results when executed repeatedly with the same arguments are said to be nonidempotent. For example, consider the following routine of a server process that debits a specified amount from a bank account and returns the balance amount to a requesting client:

```
debit (amount)
if (balance ≥ amount)
   {balance = balance – amount;
   return ("success", balance);}
else return ("failure", balance);
end;
```

Figure 3.11 shows a sequence of *debit*(100) requests made by a client for processing the *debit* routine. The first request asks the server to debit an amount of 100 from the balance. The server receives the request and processes it. Suppose the initial balance was 1000, so the server sends a reply ("*success*," 900) to the client indicating that the balance remaining is 900. This reply, for some reason, could not be delivered to the client. The client then times out waiting for the response of its request and retransmits the *debit*(100) request. The server processes the *debit*(100) request once again and sends a reply ("*success*," 800) to the client indicating that the remaining balance is 800, which is not correct. Therefore, we see from this example that multiple executions of nonidempotent routines produce undesirable results.

Clearly, when no response is received by the client, it is impossible to determine whether the failure was due to a server crash or the loss of the request or response message. Therefore, as can be seen in Figure 3.10, due to the use of timeout-based retransmission of requests, the server may execute (either partially or fully) the same request message more than once. This behavior may or may not be tolerable depending on whether multiple executions of the request have the same effect as a single execution (as in idempotent routines). If the execution of the request is nonidempotent, then its repeated execution will destroy the consistency of information. Therefore such "orphan" executions must, in general, be avoided. The orphan phenomenon has led to the identification and use of *exactly-once* semantics, which ensures that only one execution of the server's operation is performed. Primitives based on exactly-once semantics are most desired but difficult to implement.




```
debit (amount)
{
    if (balance >= amount)
    {  balance = balance - amount;
       return ("success", balance);
    }
    else return ("failure", balance);
}
```

Fig. 3.11 A nonidempotent routine.

One way to implement exactly-once semantics is to use a unique identifier for every request that the client makes and to set up a reply cache in the kernel's address space on the server machine to cache replies. In this case, before forwarding a request to a server for processing, the kernel of the server machine checks to see if a reply already exists in the reply cache for the request. If yes, this means that this is a duplicate request that has already been processed. Therefore, the previously computed result is extracted from the reply cache and a new response message is sent to the client. Otherwise, the request is a new one. In this case, the kernel forwards the request to the appropriate server for processing, and when the processing is over, it caches the request identifier along with the result of processing in the reply cache before sending a response message to the client.

An example of implementing exactly-once semantics is shown in Figure 3.12. This figure is similar to Figure 3.11 except that requests are now numbered, and a reply cache has been added to the server machine. The client makes *request-1;* the server machine's kernel receives *request-1* and then checks the reply cache to see if there is a cached reply

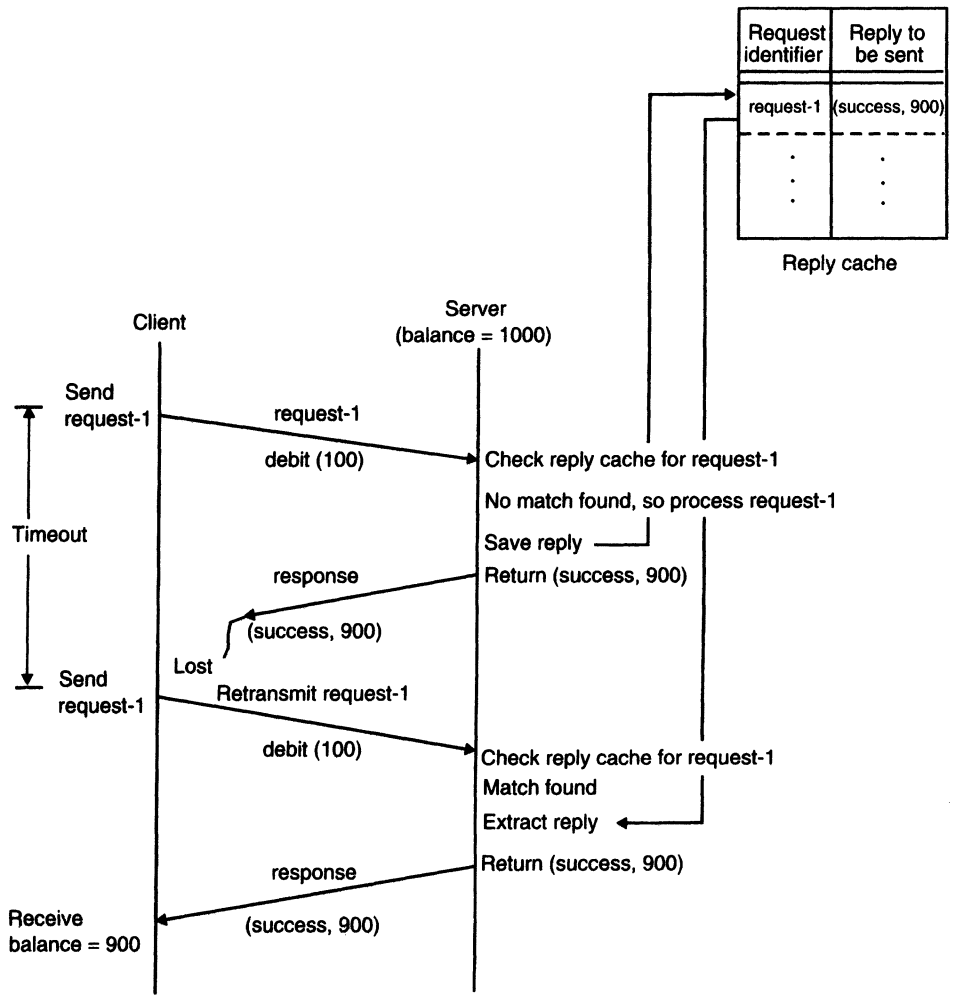


Fig. 3.12 An example of exactly-once semantics using request identifiers and reply cache.

for *request-1*. There is no match, so it forwards the request to the appropriate server. The server processes the request and returns the result to the kernel. The kernel copies the request identifier and the result of execution to the reply cache and then sends the result in the form of a response message to the client. This reply is lost, and the client times out on *request-1* and retransmits *request-1*. The server machine's kernel receives *request-1* once again and checks the reply cache to see if there is a cached reply for *request-1*. This time a match is found so it extracts the result corresponding to *request-1* from the reply cache and once again sends it to the client as a response message. Thus the reprocessing of a duplicate request is avoided. Note that the range of the request identifiers should be much larger than the number of entries in the cache.

It is important to remember that the use of a reply cache does not make a nonidempotent routine idempotent. The cache is simply one possible way to implement nonidempotent routines with exactly-once semantics.

3.9.2 Keeping Track of Lost and Out-of-Sequence Packets in Multidatagram Messages

In the case of multidatagram messages, the logical transfer of a message consists of physical transfer of several packets. Therefore, a message transmission can be considered to be complete only when all the packets of the message have been received by the process to which it is sent. For successful completion of a multidatagram message transfer, reliable delivery of every packet is important. A simple way to ensure this is to acknowledge each packet separately (called *stop-and-wait protocol*). But a separate acknowledgment packet for each request packet leads to a communication overhead. Therefore, to improve communication performance, a better approach is to use a single acknowledgment packet for all the packets of a multidatagram message (called *blast protocol*). However, when this approach is used, a node crash or a communication link failure may lead to the following problems:

- One or more packets of the multidatagram message are lost in communication.
- The packets are received out of sequence by the receiver.

An efficient mechanism to cope with these problems is to use a bitmap to identify the packets of a message. In this mechanism, the header part of each packet consists of two extra fields, one of which specifies the total number of packets in the multidatagram message and the other is the bitmap field that specifies the position of this packet in the complete message. The first field helps the receiving process to set aside a suitably sized buffer area for the message and the second field helps in deciding the position of this packet in that buffer. Since all packets have information about the total number of packets in the message, so even in the case of out-of-sequence receipt of the packets, that is, even when the first packet is not received first, a suitably sized buffer area can be set aside by the receiver for the entire message and the received packet can be placed in its proper position inside the buffer area. After timeout, if all packets have not yet been received, a bitmap indicating the unreceived packets is sent to the sender. Using the bitmap information, the sender retransmits only those packets that have not been received by the receiver. This technique is called *selective repeat*. When all the packets of a multidatagram message are received, the message transfer is complete, and the receiver sends an acknowledgment message to the sending process. This method of multidatagram message communication is illustrated with an example in Figure 3.13 in which the multidatagram message consists of five packets.

3.10 GROUP COMMUNICATION

The most elementary form of message-based interaction is *one-to-one communication* (also known as *point-to-point*, or *unicast*, *communication*) in which a single-sender process sends a message to a single-receiver process. However, for performance and ease

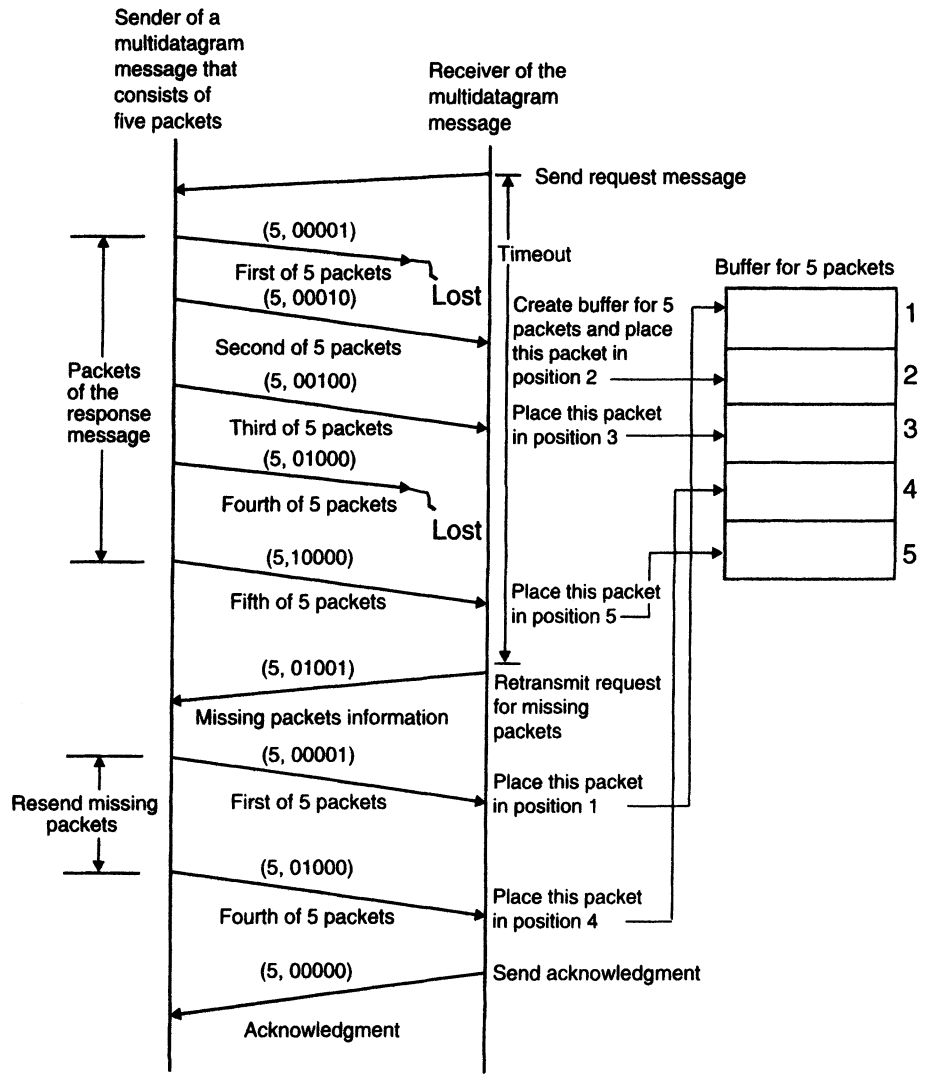


Fig. 3.13 An example of the use of a bitmap to keep track of lost and out of sequence packets in a multidatagram message transmission.

of programming, several highly parallel distributed applications require that a message-passing system should also provide group communication facility. Depending on single or multiple senders and receivers, the following three types of group communication are possible:

1. One to many (single sender and multiple receivers)
2. Many to one (multiple senders and single receiver)

3. Many to many (multiple senders and multiple receivers)

The issues related to these communication schemes are described below.

3.10.1 One-to-Many Communication

In this scheme, there are multiple receivers for a message sent by a single sender. One-to-many scheme is also known as *multicast communication*. A special case of multicast communication is *broadcast communication*, in which the message is sent to all processors connected to a network.

Multicast/broadcast communication is very useful for several practical applications. For example, consider a server manager managing a group of server processes all providing the same type of service. The server manager can multicast a message to all the server processes, requesting that a free server volunteer to serve the current request. It then selects the first server that responds. The server manager does not have to keep track of the free servers. Similarly, to locate a processor providing a specific service, an inquiry message may be broadcast. In this case, it is not necessary to receive an answer from every processor; just finding one instance of the desired service is sufficient.

Group Management

In case of one-to-many communication, receiver processes of a message form a group. Such groups are of two types—closed and open. A *closed group* is one in which only the members of the group can send a message to the group. An outside process cannot send a message to the group as a whole, although it may send a message to an individual member of the group. On the other hand, an *open group* is one in which any process in the system can send a message to the group as a whole.

Whether to use a closed group or an open group is application dependent. For example, a group of processes working on a common problem need not communicate with outside processes and can form a closed group. On the other hand, a group of replicated servers meant for distributed processing of client requests must form an open group so that client processes can send their requests to them. Therefore, a flexible message-passing system with group communication facility should support both types of groups.

A message-passing system with group communication facility provides the flexibility to create and delete groups dynamically and to allow a process to join or leave a group at any time. Obviously, the message-passing system must have a mechanism to manage the groups and their membership information. A simple mechanism for this is to use a centralized *group server* process. All requests to create a group, to delete a group, to add a member to a group, or to remove a member from a group are sent to this process. Therefore, it is easy for the group server to maintain up-to-date information of all existing groups and their exact membership. This approach, however, suffers from the problems of poor reliability and poor scalability common to all centralized techniques. Replication of the group server may be done to solve these problems to some extent. However, replication leads to the extra overhead involved in keeping the group information of all group servers consistent.

Group Addressing

A two-level naming scheme is normally used for group addressing. The high-level group name is an ASCII string that is independent of the location information of the processes in the group. On the other hand, the low-level group name depends to a large extent on the underlying hardware. For example, on some networks it is possible to create a special network address to which multiple machines can listen. Such a network address is called a *multicast address*. A packet sent to a multicast address is automatically delivered to all machines listening to the address. Therefore, in such systems a multicast address is used as a low-level name for a group.

Some networks that do not have the facility to create multicast addresses may have broadcasting facility. Networks with broadcasting facility declare a certain address, such as zero, as a *broadcast address*. A packet sent to a broadcast address is automatically delivered to all machines on the network. Therefore, the broadcast address of a network may be used as a low-level name for a group. In this case, the software of each machine must check to see if the packet is intended for it. If not, the packet is simply discarded. Since all machines receive every broadcast packet and must check if the packet is intended for it, the use of a broadcast address is less efficient than the use of a multicast address for group addressing. Also notice that in a system that uses a broadcast address for group addressing, all groups have the same low-level name, the broadcast address.

If a network does not support either the facility to create multicast addresses or the broadcasting facility, a one-to-one communication mechanism has to be used to implement the group communication facility. That is, the kernel of the sending machine sends the message packet separately to each machine that has a process belonging to the group. Therefore, in this case, the low-level name of a group contains a list of machine identifiers of all machines that have a process belonging to the group.

Notice that in the first two methods a single message packet is sent over the network, whereas in the third method the number of packets sent over the network depends on the number of machines that have one or more processes belonging to the group. Therefore the third method generates more network traffic than the other two methods and is in general less efficient. However, it is better than the broadcasting method in systems in which most groups involve only a few out of many machines on the network. Moreover the first two methods are suitable for use only on a single LAN. If the network contains multiple LANs interconnected by gateways and the processes of a group are spread over multiple LANs, the third method is simpler and easier to implement than the other two methods.

Message Delivery to Receiver Processes

User applications use high-level group names in programs. The centralized group server maintains a mapping of high-level group names to their low-level names. The group server also maintains a list of the process identifiers of all the processes for each group.

When a sender sends a message to a group specifying its high-level name, the kernel of the sending machine contacts the group server to obtain the low-level name of the group

and the list of process identifiers of the processes belonging to the group. The list of process identifiers is inserted in the message packet. If the low-level group name is either a multicast address or a broadcast address, the kernel simply sends the packet to the multicast/broadcast address. On the other hand, if the low-level group name is a list of machine identifiers, the kernel sends a copy of the packet separately to each machine in the list.

When the packet reaches a machine, the kernel of that machine extracts the list of process identifiers from the packet and forwards the message in the packet to those processes in the list that belong to its own machine. Note that when the broadcast address is used as a low-level group name, the kernel of a machine may find that none of the processes in the list belongs to its own machine. In this case, the kernel simply discards the packet.

Notice that a sender is not at all aware of either the size of the group or the actual mechanism used for group addressing. The sender simply sends a message to a group specifying its high-level name, and the operating system takes the responsibility to deliver the message to all the group members.

Buffered and Unbuffered Multicast

Multicasting is an asynchronous communication mechanism. This is because multicast *send* cannot be synchronous due to the following reasons [Gehani 1984]:

1. It is unrealistic to expect a sending process to wait until all the receiving processes that belong to the multicast group are ready to receive the multicast message.
2. The sending process may not be aware of all the receiving processes that belong to the multicast group.

How a multicast message is treated on a receiving process side depends on whether the multicast mechanism is buffered or unbuffered. For an *unbuffered multicast*, the message is not buffered for the receiving process and is lost if the receiving process is not in a state ready to receive it. Therefore, the message is received only by those processes of the multicast group that are ready to receive it. On the other hand, for a *buffered multicast*, the message is buffered for the receiving processes, so each process of the multicast group will eventually receive the message.

Send-to-All and Bulletin-Board Semantics

Ahamad and Bernstein [1985] described the following two types of semantics for one-to-many communications:

1. *Send-to-all semantics.* A copy of the message is sent to each process of the multicast group and the message is buffered until it is accepted by the process.

2. *Bulletin-board semantics.* A message to be multicast is addressed to a channel instead of being sent to every individual process of the multicast group. From a logical

point of view, the channel plays the role of a bulletin board. A receiving process copies the message from the channel instead of removing it when it makes a *receive* request on the channel. Thus a multicast message remains available to other processes as if it has been posted on the bulletin board. The processes that have *receive* access right on the channel constitute the multicast group.

Bulletin-board semantics is more flexible than send-to-all semantics because it takes care of the following two factors that are ignored by send-to-all semantics [Ahamad and Bernstein 1985]:

1. The relevance of a message to a particular receiver may depend on the receiver's state.
2. Messages not accepted within a certain time after transmission may no longer be useful; their value may depend on the sender's state.

To illustrate this, let us once again consider the example of a server manager multicasting a message to all the server processes to volunteer to serve the current request. Using send-to-all semantics, it would be necessary to multicast to all the servers, causing many contractors to process extraneous messages. Using bulletin-board semantics, only those contractors that are idle and in a state suitable for serving requests will make a *receive* request on the concerned channel, and thus only contractors in the correct state will process such messages [Ahamad and Bernstein 1985]. Furthermore, the message is withdrawn from the channel by the server manager as soon as the bid period is over; that is, the first bidder is selected (in this case). Therefore, the message remains available for being received only as long as the server manager is in a state in which bids are acceptable. While this does not completely eliminate extraneous messages (contractors may still reply after the bid period is over), it does help in reducing them.

Flexible Reliability in Multicast Communication

Different applications require different degrees of reliability. Therefore multicast primitives normally provide the flexibility for user-definable reliability. Thus, the sender of a multicast message can specify the number of receivers from which a response message is expected. In one-to-many communication, the degree of reliability is normally expressed in the following forms:

1. The *0-reliable*. No response is expected by the sender from any of the receivers. This is useful for applications using asynchronous multicast in which the sender does not wait for any response after multicasting the message. An example of this type of application is a time signal generator.

2. The *1-reliable*. The sender expects a response from any of the receivers. The already described application in which a server manager multicasts a message to all the servers to volunteer to serve the current request and selects the first server that responds is an example of 1-reliable multicast communication.

3. The *m-out-of-n-reliable*. The multicast group consists of n receivers and the sender expects a response from m ($1 < m < n$) of the n receivers. Majority consensus algorithms (described in Chapter 9) used for the consistency control of replicated information use this form of reliability, with the value $m = n/2$.

4. *All-reliable*. The sender expects a response message from all the receivers of the multicast group. For example, suppose a message for updating the replicas of a file is multicast to all the file servers having a replica of the file. Naturally, such a sender process will expect a response from all the concerned file servers.

Atomic Multicast

Atomic multicast has an all-or-nothing property. That is, when a message is sent to a group by atomic multicast, it is either received by all the processes that are members of the group or else it is not received by any of them. An implicit assumption usually made in atomic multicast is that when a process fails, it is no longer a member of the multicast group. When the process comes up after failure, it must join the group afresh.

Atomic multicast is not always necessary. For example, applications for which the degree of reliability requirement is 0-reliable, 1-reliable, or m-out-of-n-reliable do not need atomic multicast facility. On the other hand, applications for which the degree of reliability requirement is all-reliable need atomic multicast facility. Therefore, a flexible message-passing system should support both atomic and nonatomic multicast facilities and should provide the flexibility to the sender of a multicast message to specify in the *send* primitive whether atomicity property is required or not for the message being multicast.

A simple method to implement atomic multicast is to multicast a message, with the degree of reliability requirement being all-reliable. In this case, the kernel of the sending machine sends the message to all members of the group and waits for an acknowledgment from each member (we assume that a one-to-one communication mechanism is used to implement the multicast facility). After a timeout period, the kernel retransmits the message to all those members from whom an acknowledgment message has not yet been received. The timeout-based retransmission of the message is repeated until an acknowledgment is received from all members of the group. When all acknowledgments have been received, the kernel confirms to the sender that the atomic multicast process is complete.

The above method works fine only as long as the machines of the sender process and the receiver processes do not fail during an atomic multicast operation. This is because if the machine of the sender process fails, the message cannot be retransmitted if one or more members did not receive the message due to packet loss or some other reason. Similarly, if the machine of a receiver process fails and remains down for some time, the message cannot be delivered to that process because retransmissions of the message cannot be continued indefinitely and have to be aborted after some predetermined time. Therefore, a fault-tolerant atomic multicast protocol must ensure that a multicast will be delivered to all members of the multicast group even in the event of failure of the sender's machine or a receiver's machine. One method to implement such a protocol is described next [Tanenbaum 1995].

In this method, each message has a message identifier field to distinguish it from all other messages and a field to indicate that it is an atomic multicast message. The sender sends the message to a multicast group. The kernel of the sending machine sends the message to all members of the group and uses timeout-based retransmissions as in the previous method. A process that receives the message checks its message identifier field to see if it is a new message. If not, it is simply discarded. Otherwise, the receiver checks to see if it is an atomic multicast message. If so, the receiver also performs an atomic multicast of the same message, sending it to the same multicast group. The kernel of this machine treats this message as an ordinary atomic multicast message and uses timeout-based retransmissions when needed. In this way, each receiver of an atomic multicast message will perform an atomic multicast of the message to the same multicast group. The method ensures that eventually all the surviving processes of the multicast group will receive the message even if the sender machine fails after sending the message or a receiver machine fails after receiving the message.

Notice that an atomic multicast is in general very expensive as compared to a normal multicast due to the large number of messages involved in its implementation. Therefore, a message-passing system should not use the atomicity property as a default property of multicast messages but should provide this facility as an option.

Group Communication Primitives

In both one-to-one communication and one-to-many communication, the sender of a process basically has to specify two parameters: destination address and a pointer to the message data. Therefore ideally the same *send* primitive can be used for both one-to-one communication and one-to-many communication. If the destination address specified in the *send* primitive is that of a single process, the message is sent to that one process. On the other hand, if the destination address is a group address, the message is sent to all processes that belong to that group.

However, most systems having a group communication facility provide a different primitive (such as *send_group*) for sending a message to a group. There are two main reasons for this. First, it simplifies the design and implementation of a group communication facility. For example, suppose the two-level naming mechanism is used for both process addressing and group addressing. The high-level to low-level name mapping for processes is done by the name server, and for groups it is done by the group server. With this design, if a single *send* primitive is used for both one-to-one communication and one-to-many communication, the kernel of the sending machine cannot know whether the destination address specified by a user is a single process address or a group address. Consequently, it does not know whether the name server or the group server should be contacted for obtaining the low-level name of the specified destination address. Implementation methods to solve this problem are possible theoretically, but the design will become complicated. On the other hand, if separate primitives such as *send* and *send_group* are used, the kernel can easily make out whether the specified destination address is a single process address or a group address and can contact the appropriate server to obtain the corresponding low-level name.

Second, it helps in providing greater flexibility to the users. For instance, a separate parameter may be used in the *send_group* primitive to allow users to specify the degree of reliability desired (number of receivers from which a response message is expected), and another parameter may be used to specify whether the atomicity property is required or not.

3.10.2 Many-to-One Communication

In this scheme, multiple senders send messages to a single receiver. The single receiver may be selective or nonselective. A *selective receiver* specifies a unique sender; a message exchange takes place only if that sender sends a message. On the other hand, a *nonselective receiver* specifies a set of senders, and if any one sender in the set sends a message to this receiver, a message exchange takes place.

Thus we see that an important issue related to the many-to-one communication scheme is nondeterminism. The receiver may want to wait for information from any of a group of senders, rather than from one specific sender. As it is not known in advance which member (or members) of the group will have its information available first, such behavior is nondeterministic. In some cases it is useful to dynamically control the group of senders from whom to accept message. For example, a buffer process may accept a request from a producer process to store an item in the buffer whenever the buffer is not full; it may accept a request from a consumer process to get an item from the buffer whenever the buffer is not empty. To program such behavior, a notation is needed to express and control nondeterminism. One such construct is the "guarded command" statement introduced by Dijkstra [1975]. Since this issue is related to programming languages rather than operating systems, we will not discuss it any further.

3.10.3 Many-to-Many Communication

In this scheme, multiple senders send messages to multiple receivers. The one-to-many and many-to-one schemes are implicit in this scheme. Hence the issues related to one-to-many and many-to-one schemes, which have already been described above, also apply to the many-to-many communication scheme. In addition, an important issue related to many-to-many communication scheme is that of *ordered message delivery.*

Ordered message delivery ensures that all messages are delivered to all receivers in an order acceptable to the application. This property is needed by many applications for their correct functioning. For example, suppose two senders send messages to update the same record of a database to two server processes having a replica of the database. If the messages of the two senders are received by the two servers in different orders, then the final values of the updated record of the database may be different in its two replicas. Therefore, this application requires that all messages be delivered in the same order to all receivers.

Ordered message delivery requires message sequencing. In a system with a single sender and multiple receivers (one-to-many communication), sequencing messages to all the receivers is trivial. If the sender initiates the next multicast transmission only after confirming that the previous multicast message has been received by all the members, the

messages will be delivered in the same order. On the other hand, in a system with multiple senders and a single receiver (many-to-one communication), the messages will be delivered to the receiver in the order in which they arrive at the receiver's machine. Ordering in this case is simply handled by the receiver. Thus we see that it is not difficult to ensure ordered delivery of messages in many-to-one or one-to-many communication schemes.

However, in many-to-many communication, a message sent from a sender may arrive at a receiver's destination before the arrival of a message from another sender; but this order may be reversed at another receiver's destination (see Fig. 3.14). The reason why messages of different senders may arrive at the machines of different receivers in different orders is that when two processes are contending for access to a LAN, the order in which messages of the two processes are sent over the LAN is nondeterministic. Moreover, in a WAN environment, the messages of different senders may be routed to the same destination using different routes that take different amounts of time (which cannot be correctly predicted) to the destination. Therefore, ensuring ordered message delivery requires a special message-handling mechanism in many-to-many communication scheme.

The commonly used semantics for ordered delivery of multicast messages are absolute ordering, consistent ordering, and causal ordering. These are described below.

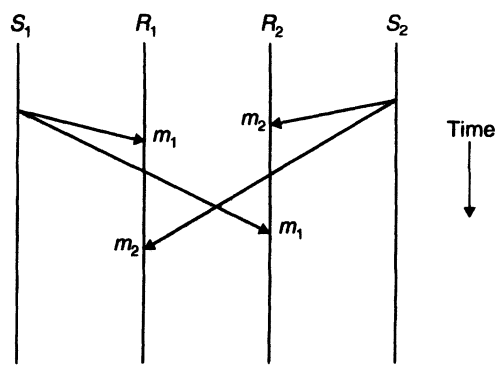


Fig. 3.14 No ordering constraint for message delivery.

Absolute Ordering

This semantics ensures that all messages are delivered to all receiver processes in the exact order in which they were sent (see Fig. 3.15). One method to implement this semantics is to use global timestamps as message identifiers. That is, the system is assumed to have a clock at each machine and all clocks are synchronized with each other, and when a sender sends a message, the clock value (timestamp) is taken as the identifier of that message and embedded in the message.

The kernel of each receiver's machine saves all incoming messages meant for a receiver in a separate queue. A sliding-window mechanism is used to periodically

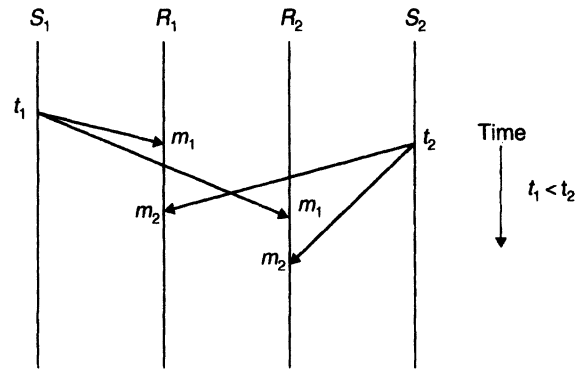


Fig. 3.15 Absolute ordering of messages.

deliver the message from the queue to the receiver. That is, a fixed time interval is selected as the window size, and periodically all messages whose timestamp values fall within the current window are delivered to the receiver. Messages whose timestamp values fall outside the window are left in the queue because of the possibility that a tardy message having a timestamp value lower than that of any of the messages in the queue might still arrive. The window size is properly chosen taking into consideration the maximum possible time that may be required by a message to go from one machine to any other machine in the network.

Consistent Ordering

Absolute-ordering semantics requires globally synchronized clocks, which are not easy to implement. Moreover, absolute ordering is not really what many applications need to function correctly. For instance, in the replicated database updation example, it is sufficient to ensure that both servers receive the update messages of the two senders in the same order even if this order is not the real order in which the two messages were sent. Therefore, instead of supporting absolute-ordering semantics, most systems support consistent-ordering semantics. This semantics ensures that all messages are delivered to all receiver processes in the same order. However, this order may be different from the order in which messages were sent (see Fig. 3.16).

One method to implement consistent-ordering semantics is to make the many-to-many scheme appear as a combination of many-to-one and one-to-many schemes [Chang and Maxemchuk 1985]. That is, the kernels of the sending machines send messages to a single receiver (known as a *sequencer*) that assigns a sequence number to each message and then multicasts it. The kernel of each receiver's machine saves all incoming messages meant for a receiver in a separate queue. Messages in a queue are delivered immediately to the receiver unless there is a gap in the message identifiers, in which case messages after the gap are not delivered until the ones in the gap have arrived.

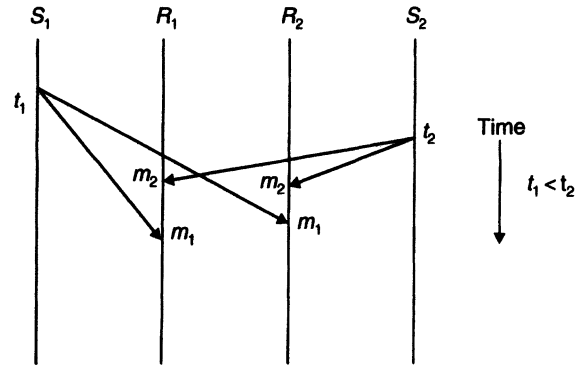


Fig. 3.16 Consistent ordering of messages.

The sequencer-based method for implementing consistent-ordering semantics is subject to single point of failure and hence has poor reliability. A distributed algorithm for implementing consistent-ordering semantics that does not suffer from this problem is the *ABCAST protocol* of the ISIS system [Birman and Van Renesse 1994, Birman 1993, Birman et al. 1991, Birman and Joseph 1987]. It assigns a sequence number to a message by distributed agreement among the group members and the sender and works as follows:

1. The sender assigns a temporary sequence number to the message and sends it to all the members of the multicast group. The sequence number assigned by the sender must be larger than any previous sequence number used by the sender. Therefore, a simple counter can be used by the sender to assign sequence numbers to its messages.

2. On receiving the message, each member of the group returns a proposed sequence number to the sender. A member (i) calculates its proposed sequence number by using the function

$$\max(F_{\max}, P_{\max}) + 1 + i/N$$

where $F_{\max}$ is the largest final sequence number agreed upon so far for a message received by the group (each member makes a record of this when a final sequence number is agreed upon), $P_{\max}$ is the largest proposed sequence number by this member, and N is the total number of members in the multicast group.

3. When the sender has received the proposed sequence numbers from all the members, it selects the largest one as the final sequence number for the message and sends it to all members in a *commit* message. The chosen final sequence number is guaranteed to be unique because of the term i/N in the function used for the calculation of a proposed sequence number.

4. On receiving the *commit* message, each member attaches the final sequence number to the message.

5. Committed messages with final sequence numbers are delivered to the application programs in order of their final sequence numbers. Note that the algorithm for sequence number assignment to a message is a part of the runtime system, not the user processes.

It can be shown that this protocol ensures consistent ordering semantics.

Causal Ordering

For some applications consistent-ordering semantics is not necessary and even weaker semantics is acceptable. Therefore, an application can have better performance if the message-passing system used supports a weaker ordering semantics that is acceptable to the application. One such weaker ordering semantics that is acceptable to many applications is the causal-ordering semantics. This semantics ensures that if the event of sending one message is causally related to the event of sending another message, the two messages are delivered to all receivers in the correct order. However, if two message-sending events are not causally related, the two messages may be delivered to the receivers in any order. Two message-sending events are said to be causally related if they are corelated by the *happened-before* relation (for a definition of *happened-before* relation see Chapter 6). That is, two message-sending events are causally related if there is any possibility of the second one being influenced in any way by the first one. The basic idea behind causal-ordering semantics is that when it matters, messages are always delivered in the proper order, but when it does not matter, they may be delivered in any arbitrary order.

An example of causal ordering of messages is given in Figure 3.17. In this example, sender S_1 sends message m_1 to receivers R_1, R_2, and R_3 and sender S_2 sends message m_2 to receivers R_2, and R_3. On receiving m_1, receiver R_1 inspects it, creates a new message m_3, and sends m_3 to R_2 and R_3. Note that the event of sending m_3 is causally related to the event of sending m_1 because the contents of m_3 might have been derived in part from m_1; hence the two messages must be delivered to both R_2 and R_3 in the proper order, m_1

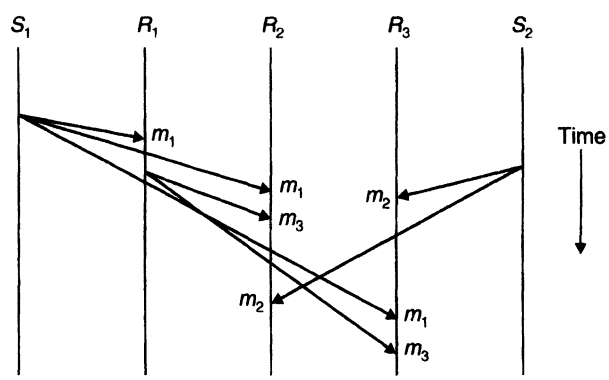


Fig. 3.17 Causal ordering of messages.

before m_3. Also note that since m_2 is not causally related to either m_1 or m_3, m_2 can be delivered at any time to R_2 and R_3 irrespective of m_1 or m_3. This is exactly what the example of Figure 3.17 shows.

One method for implementing causal-ordering semantics is the *CBCAST protocol* of the ISIS system [Birman et al. 1991]. It works as follows:

1. Each member process of a group maintains a vector of n components, where n is the total number of members in the group. Each member is assigned a sequence number from 0 to n, and the ith component of the vectors corresponds to the member with sequence number i. In particular, the value of the ith component of a member's vector is equal to the number of the last message received in sequence by this member from member i.

2. To send a message, a process increments the value of its own component in its own vector and sends the vector as part of the message.

3. When the message arrives at a receiver process's site, it is buffered by the runtime system. The runtime system tests the two conditions given below to decide whether the message can be delivered to the user process or its delivery must be delayed to ensure causal-ordering semantics. Let S be the vector of the sender process that is attached to the message and R be the vector of the receiver process. Also let i be the sequence number of the sender process. Then the two conditions to be tested are

$$S[i] = R[i] + 1 \quad \text{and} \quad S[j] \leq R[j] \quad \text{for all } j \neq i$$

The first condition ensures that the receiver has not missed any message from the sender. This test is needed because two messages from the same sender are always causally related. The second condition ensures that the sender has not received any message that the receiver has not yet received. This test is needed to make sure that the sender's message is not causally related to a message missed by the receiver.

If the message passes these two tests, the runtime system delivers it to the user process. Otherwise, the message is left in the buffer and the test is carried out again for it when a new message arrives.

A simple example to illustrate the algorithm is given in Figure 3.18. In this example, there are four processes A, B, C, and D. The status of their vectors at some instance of time is (3, 2, 5, 1), (3, 2, 5, 1), (2, 2, 5, 1), and (3, 2, 4, 1), respectively. This means that, until now, A has sent three messages, B has sent two messages, C has sent five messages, and D has sent one message to other processes. Now A sends a new message to other processes. Therefore, the vector attached to the message will be (4, 2, 5, 1). The message can be delivered to B because it passes both tests. However, the message has to be delayed by the runtime systems of sites of processes C and D because the first test fails at the site of process C and the second test fails at the site of process D.

A good message-passing system should support at least consistent- and causal-ordering semantics and should provide the flexibility to the users to choose one of these in their applications.

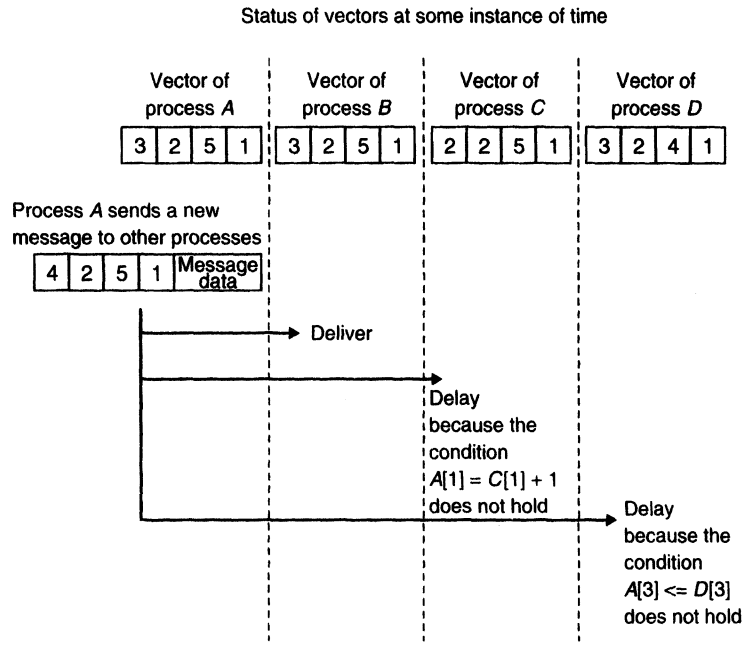


Fig. 3.18 An example to illustrate the CBCAST protocol for implementing causal ordering semantics.

3.11 CASE STUDY: 4.3BSD UNIX IPC MECHANISM

The socket-based IPC of the 4.3BSD UNIX system illustrates how a message-passing system can be designed using the concepts and mechanisms presented in this chapter. The system was produced by the Computer Systems Research Group (CSRG) of the University of California at Berkeley and is the most widely used and well documented message-passing system.

3.11.1 Basic Concepts and Main Features

The IPC mechanism of the 4.3BSD UNIX provides a general interface for constructing network-based applications. Its basic concepts and main features are as follows:

1. It is network independent in the sense that it can support communication networks that use different sets of protocols, different naming conventions, different hardware, and so on. For this, it uses the notion of *communication domain*, which refers to a standard set of communication properties. In this chapter we have seen that there are different methods of naming a communication endpoint. We also have seen that there are different semantics of communication related to synchronization, reliability, ordering, and so on. Different networks often use different naming conventions for naming communication endpoints

and possess different semantics of communication. These properties of a network are known as its *communication properties*. Networks with the same communication properties belong to a common communication domain (or protocol family). By providing the flexibility to specify a communication domain as a parameter of the communication primitive used, the IPC mechanism of the 4.3BSD UNIX allows the users to select a domain appropriate to their applications.

2. It uses a unified abstraction, called *socket*, for an endpoint of communication. That is, a socket is an abstract object from which messages are sent and received. The IPC operations are based on socket pairs, one belonging to each of a pair of communicating processes that may be on the same or different computers. A pair of sockets may be used for unidirectional or bidirectional communication between two processes. A message sent by a sending process is queued in its socket until it has been transmitted across the network by the networking protocol and an acknowledgment has been received (only if the protocol requires one). On the receiver side, the message is queued in the receiving process's socket until the receiving process makes an appropriate system call to receive it.

Any process can create a socket for use in communication with another process. Sockets are created within a communication domain. A created socket exists until it is explicitly closed or until every process having a reference to it exits.

3. For location transparency, it uses a two-level naming scheme for naming communication endpoints. That is, a socket can be assigned a high-level name that is a human-readable string. The low-level name of a socket is communication-domain dependent. For example, it may consist of a local port number and an Internet address. For translation of high-level socket names to their low-level names, 4.3BSD provides functions for application programs rather than placing the translation functions in the kernel. Note that a socket's high-level name is meaningful only within the context of the communication domain in which the socket is created.

4. It is highly flexible in the sense that it uses a typing mechanism for sockets to provide the semantic aspects of communication to applications in a controlled and uniform manner. That is, all sockets are typed according to their communication semantics, such as ordered delivery, unduplicated delivery, reliable delivery, connectionless communication, connection-oriented communication, and so on. The system defines some standard socket types and provides the flexibility to the users to define and use their own socket types when needed. For example, a socket of type *datagram* models potentially unreliable, connectionless packet communication, and a socket of type *stream* models a reliable connection-based byte stream.

5. Messages can be broadcast if the underlying network provides broadcast facility.

3.11.2 The IPC Primitives

The primitives of the 4.3BSD UNIX IPC mechanism are provided as system calls implemented as a layer on top of network communication protocols such as TCP, UDP, and so on. Layering the IPC mechanism directly on top of network communication

protocols helps in making it efficient. The most important available IPC primitives are briefly described below.

s = socket (domain, type, protocol)

When a process wants to communicate with another process, it must first create a socket by using the *socket* system call. The first parameter of this call specifies the communication domain. The most commonly used domain is the Internet communication domain because a large number of hosts in the world support the Internet communication protocols. The second parameter specifies the socket type that is selected according to the communication semantics requirements of the application. The third parameter specifies the communication protocol (e.g., TCP/IP or UDP/IP) to be used for the socket's operation. If the value of this parameter is specified as zero, the system chooses an appropriate protocol. The *socket* call returns a descriptor by which the socket may be referenced in subsequent system calls. A created socket is discarded with the normal *close* system call.

bind (s, addr, addrlen)

After creating a socket, the receiver must bind it to a socket address. Note that if two-way communication is desired between two processes, both processes have to receive messages, and hence both must separately bind their sockets to a socket address. The *bind* system call is used for this purpose. The three parameters of this call are the descriptor of the created socket, a reference to a structure containing the socket address to which the socket is to be bound, and the number of bytes in the socket address. Once a socket has been bound, its address cannot be changed.

It might seem more reasonable to combine the system calls for socket creation and binding a socket to a socket address (name) in a single system call. There are two main reasons for separating these two operations in different system calls. First, with this approach a socket can be useful without names. Forcing users to name every socket that is created causes extra burden on users and may lead to the assignment of meaningless names. Second, some communication domains might require additional, nonstandard information (such as type of service) for binding of a name to a socket. The need to supply this information at socket creation time will further complicate the interface.

connect (s, server_addr, server_addrlen)

The two most commonly used communication types in the 4.3BSD UNIX IPC mechanism are connection-based (stream) communication and connectionless (datagram) communication. In connection-based communication, two processes first establish a connection between their pairs of sockets. The connection establishment process is asymmetric because one of the processes keeps waiting for a request for a connection and the other makes a request for a connection. Once connection has been established, data can be transmitted between the two processes in either direction. This type of communication is useful for implementing client-server applications. A server creates a socket, binds a name

to it, and makes the name publicly known. It then waits for a connection request from client processes. Clients send connection requests to the server. Once the connection is established, they can exchange request and reply messages. Connection-based communication supports reliable exchange of messages.

In connectionless communication, a socket pair is identified each time a communication is made. For this, the sending process specifies its local socket descriptor and the socket address of the receiving process's socket each time it sends a message. Connectionless communication is potentially unreliable.

The *connect* system call is used in connection-based communication by a client process to request a connection establishment between its own socket and the socket of the server process with which it wants to communicate. The three parameters of this call are the descriptor of the client's socket, a reference to a structure containing the socket address of the server's socket, and the number of bytes in the socket address. The *connect* call automatically binds a socket address (name) to the client's socket. Hence prior binding is not needed.

listen (s, backlog)

The *listen* system call is used in case of connection-based communication by a server process to listen on its socket for client requests for connections. The two parameters of this call are the descriptor of the server's socket and the maximum number of pending connections that should be queued for acceptance.

snew = accept (s, client_addr, client_addrlen)

The *accept* system call is used in a connection-based communication by a server process to accept a request for a connection establishment made by a client and to obtain a new socket for communication with that client. The three parameters of this call are the descriptor of the server's socket, a reference to a structure containing the socket address of the client's socket, and the number of bytes in the socket address. Note that the call returns a descriptor (*snew*) that is the descriptor of a new socket that is automatically created upon execution of the *accept* call. This new socket is paired with the client's socket so that the server can continue to use the original socket with descriptor *s* for accepting further connection requests from other clients.

Primitives for Sending and Receiving Data

A variety of system calls are available for sending and receiving data. The four most commonly used are:

nbytes = read (snew, buffer, amount)
write (s, "message," msg_length)
amount = recvfrom (s, buffer, sender_address)
sendto (s, "message," receiver_address)

The *read* and *write* system calls are most suitable for use in connection-based communication. The *write* operation is used by a client to send a message to a server. The socket to be used for sending the message, the message, and the length of the message are specified as parameters to the call. The *read* operation is used by the server process to receive the message sent by the client. The socket of the server to which the client's socket is connected and the buffer for storing the received message are specified as parameters to the call. The call returns the actual number of characters received. The socket connection establishment between the client and the server behaves like a channel of stream data that does not contain any message boundary indications. That is, the sender pumps data into the channel and the receiver reads them in the same sequence as written by the corresponding write operations. The channel size is limited by a bounded queue at the receiving socket. The sender blocks if the queue is full and the receiver blocks if the queue is empty.

On the other hand, the *recvfrom* and *sendto* system calls are most suitable for use in case of connectionless communication. The *sendto* operation is used by a sender to send a message to a particular receiver. The socket through which the message is to be sent, the message, and a reference to a structure containing the socket address of the receiver to which the message is to be sent are specified as parameters to this call. The *recvfrom* operation is used by a receiver to receive a message from a particular sender. The socket through which the message is to be received, the buffer where the message is to be stored, and a reference to a structure containing the socket address of the sender from which the message is to be received are specified as parameters to this call. The *recvfrom* call collects the first message in the queue at the socket. However, if the queue is empty, it blocks until a message arrives.

Figure 3.19 illustrates the use of sockets for connectionless communication between two processes. In the *socket* call, the specification of *AF_INET* as the first parameter indicates that the communication domain is the Internet communication domain, and the specification of *SOCK_DGRAM* as the second parameter indicates that the socket is of the datagram type (used for unreliable, connectionless communication).

Alternatively, Figure 3.20 illustrates the use of sockets for connection-based communication between a client process and a server process. The specification of *SOCK_STREAM* as the second parameter of the *socket* call indicates that the socket is of the stream type (used for reliable, connection-based communication).

3.12 SUMMARY

Interprocess communication (IPC) requires information sharing among two or more processes. The two basic methods for information sharing are original sharing (shared-data approach) and copy sharing (message-passing approach). Since computers in a network do not share memory, the message-passing approach is most commonly used in distributed systems.

A message-passing system is a subsystem of a distributed operating system that provides a set of message-based protocols, and it does so by shielding the details of complex network protocols and multiple heterogeneous platforms from programmers.

```
.
.
s = socket (AF_INET, SOCK_DGRAM, 0);
.
.
bind (s, sender_address, server_address_length);
.
.
sendto (s, "message", receiver_address);
.
.
close (s);
```

(*a*)

```
.
.
s = socket (AF_INET, SOCK_DGRAM, 0);
.
.
bind (s, receiver_address, receiver_address_length)
.
.
amount = recvfrom (s, buffer, sender_address);
.
.
close (s);
```

(*b*)

Fig. 3.19 Use of sockets for connectionless communication between two processes. (*a*) Socket-related system calls in sender's program. (*b*) Socket-related system calls in receiver's program.

Some of the desirable features of a good message-passing system are simplicity, uniform semantics, efficiency, reliability, correctness, flexibility, security, and portability.

The sender and receiver of a message may communicate either in the synchronous or asynchronous mode. As compared to the synchronous mode of communication, the asynchronous mode provides better concurrency, reduced message traffic, and better flexibility. However, the asynchronous communication mode is more complicated to implement, needs message buffering, and requires programmers to deviate from the traditional centralized programming paradigm.

The four types of buffering strategies that may be used in the design of IPC mechanisms are a null buffer, or no buffering; a simple-message buffer; an unbounded-capacity buffer; and a finite-bound, or multiple-message, buffer.

Messages are transmitted over a transmission channel in the form of packets. Therefore, for transmitting a message that is greater than the maximum size of a packet, the logical message has to be separated (disassembled) and transmitted in multiple packets. Such a message is called a multipacket or a multidatagram message.

Encoding is the process of converting the program objects of a message to a stream form that is suitable for transmission over a transmission channel. This process takes place on the sender side of the message. The reverse process of reconstructing program objects from message data on the receiver side is known as decoding of the message data.

```
.
.
.
s = socket (AF_INET, SOCK_STREAM, 0);
.
.
.
connect (s, server_address, server_address_length);
.
.
.
write (s, "message", msg_length);
.
.
.
close (s);
```

(*a*)

```
.
.
.
s = socket (AF_INET, SOCK_STREAM, 0);
.
.
.
bind (s, server_address, server_address_length);
listen (s, backlog);
.
.
.
snew = accept (s, client_address, client_address_length);
.
.
.
nbytes = read (snew, buffer, amount);
.
.
.
close (snew);
close(s);
```

(*b*)

Fig. 3.20 Use of sockets for connection-based communication between a client and a server. (*a*) Socket-related system calls in client's program. (*b*) Socket-related system calls in server's program.

Another major issue in message passing is addressing (or naming) of the parties involved in an interaction. A process may or may not explicitly name a process with which it wants to communicate depending upon whether it wants to communicate only with a specific process or with any process of a particular type. An important goal in process addressing is to provide location transparency. The most commonly used method to achieve this goal is to use a two-level naming scheme for processes and a name server to map high-level, machine-independent process names to their low-level, machine-dependent names.

Failure handling is another important issue in the design of an IPC mechanism. The two commonly used methods in the design of a reliable IPC protocol are the use of internal retransmissions based on timeouts and the use of explicit acknowledgment packets. Two important issues related to failure handling in IPC mechanisms are idempotency and handling of duplicate request messages and keeping track of lost and out-of-sequence packets in multidatagram messages.

The most elementary form of message-based interaction is one-to-one communication in which a single sending process sends a message to a single receiving process. However, for better performance and flexibility, several distributed systems provide group com-