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

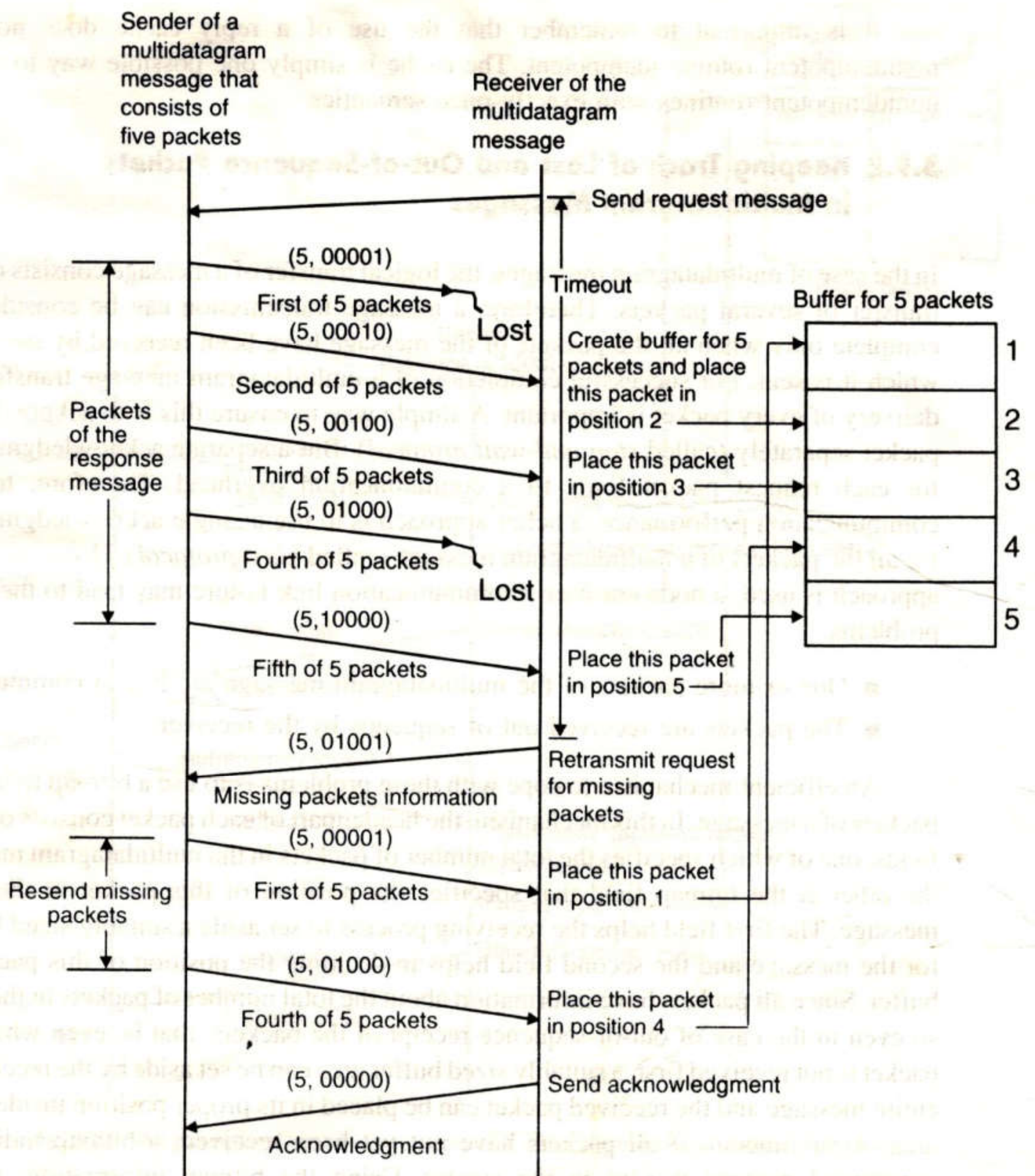
Sender of a multidatagram message that consists of five packets
Receiver of the multidatagram message
Send request message
(5, 00001)
First of 5 packets
Lost
Timeout
Buffer for 5 packets
(5, 00010)
Second of 5 packets
Create buffer for 5 packets and place this packet in position 2
Packets of the response message
(5, 00100)
Third of 5 packets
Place this packet in position 3
(5, 01000)
Fourth of 5 packets
Lost
(5,10000)
Fifth of 5 packets
Place this packet in position 5
1
2
3
4
5
(5, 01001)
Missing packets information
Retransmit request for missing packets
(5, 00001)
Resend missing packets
First of 5 packets
Place this packet in position 1
(5, 01000)
Fourth of 5 packets
Place this packet in position 4
(5, 00000)
Send acknowledgment
Acknowledgment

**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 exafnple, 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.

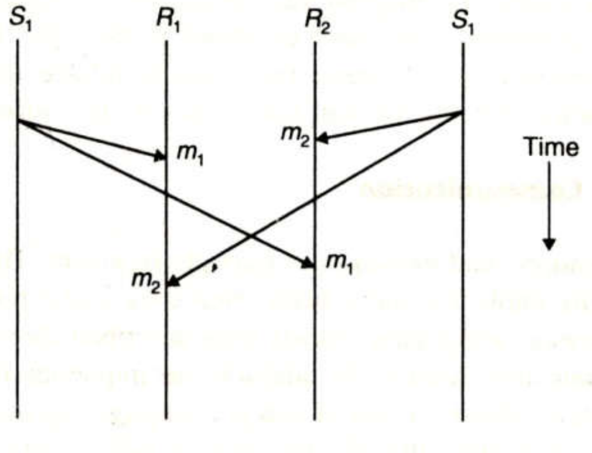

**Fig. 3.14** No ordering constraint for message delivery.

## 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

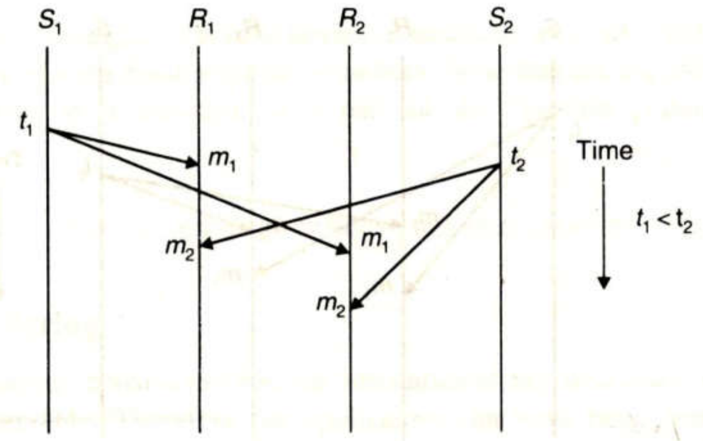

**Fig. 3.15** Absolute ordering of messages.

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.

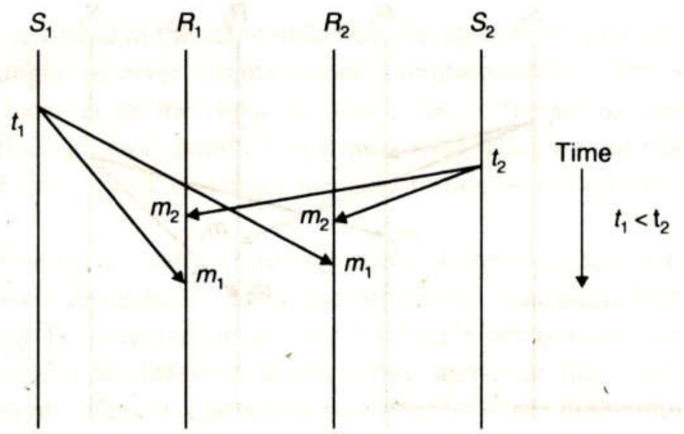

**Fig. 3.16** Consistent ordering of messages.

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, $m_1$

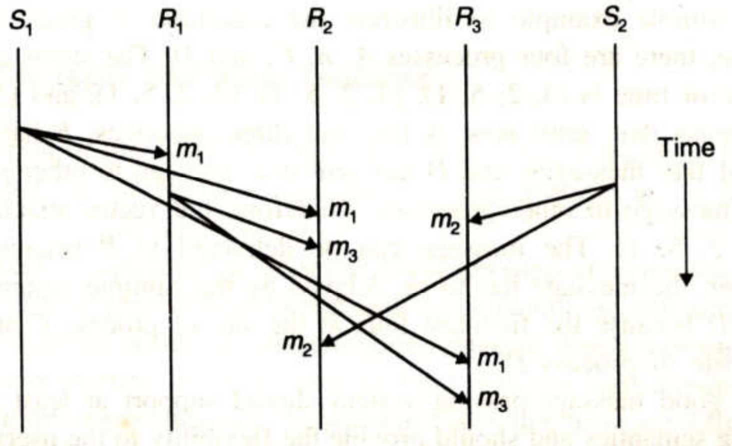

**Fig. 3.17** Causal ordering of messages.

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 $i$th component of the vectors corresponds to the member with sequence number $i$. In particular, the value of the $i$th 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.

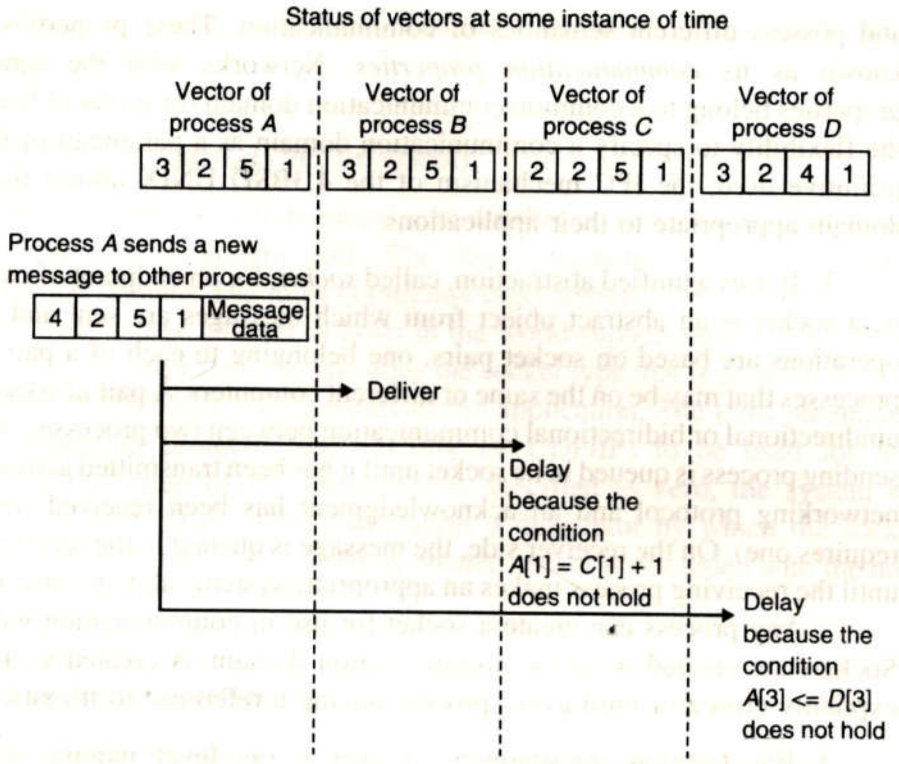
Status of vectors at some instance of time
Vector of process *A*
3 2 5 1
Vector of process *B*
3 2 5 1
Vector of process *C*
2 2 5 1
Vector of process *D*
3 2 4 1
Process *A* sends a new message to other processes
4 2 5 1 Message data
Deliver
Delay because the condition $A[1] = C[1] + 1$ does not hold
Delay because the condition $A[3] <= D[3]$ does not hold

**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