2.5. GROUP COMMUNICATION

An underlying assumption intrinsic to RPC is that communication involves only *two* parties, the client and the server. Sometimes there are circumstances in which communication involves multiple processes, not just two. For example, consider a group of file servers cooperating to offer a single, fault-tolerant file service. In such a system, it might be desirable for a client to send a message to all the servers, to make sure that the request could be carried out even if one of them crashed. RPC cannot handle communication from one sender to many receivers, other than by performing separate RPCs with each one. In this section we will discuss alternative communication mechanisms in which a message can be sent to multiple receivers in one operation.

2.5.1. Introduction to Group Communication

A group is a collection of processes that act together in some system or user-specified way. The key property that all groups have is that when a message is sent to the group itself, all members of the group receive it. It is a form of **one-to-many** communication (one sender, many receivers), and is contrasted with **point-to-point** communication in Fig. 2-30.

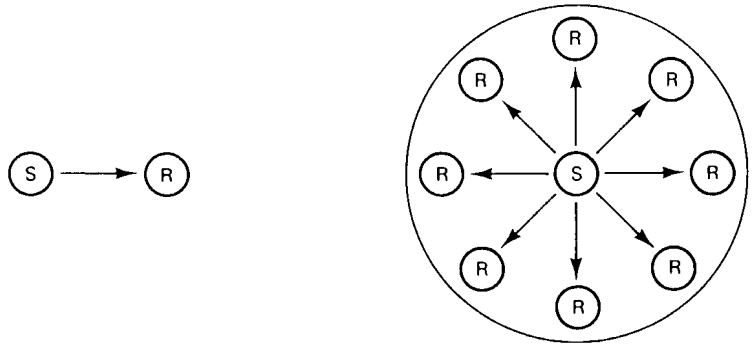

Fig. 2-30. (a) Point-to-point communication is from one sender to one receiver. (b) One-to-many communication is from one sender to multiple receivers.

Groups are dynamic. New groups can be created and old groups can be destroyed. A process can join a group or leave one. A process can be a member of several groups at the same time. Consequently, mechanisms are needed for managing groups and group membership.

Groups are roughly analogous to social organizations. A person might be a member of a book club, a tennis club, and an environmental organization. On a particular day, he might receive mailings (messages) announcing a new birthday cake cookbook from the book club, the annual Mother's Day tennis tournament from the tennis club, and the start of a campaign to save the Southern groundhog from the environmental organization. At any moment, he is free to leave any or all of these groups, and possibly join other groups.

Although in this book we will study only operating system (i.e., process) groups, it is worth mentioning that other groups are also commonly encountered in computer systems. For example, on the USENET computer network, there are hundreds of news groups, each about a specific subject. When a person sends a message to a particular news group, all members of the group receive it, even if there are tens of thousands of them. These higher-level groups usually have looser rules about who is a member, what the exact semantics of message delivery are, and so on, than do operating system groups. In most cases, this looseness is not a problem.

The purpose of introducing groups is to allow processes to deal with collections of processes as a single abstraction. Thus a process can send a message to a group of servers without having to know how many there are or where they are, which may change from one call to the next.

How group communication is implemented depends to a large extent on the hardware. On some networks, it is possible to create a special network address (for example, indicated by setting one of the high-order bits to 1), to which multiple machines can listen. When a packet is sent to one of these addresses, it is automatically delivered to all machines listening to the address. This technique is called **multicasting**. Implementing groups using multicast is straightforward: just assign each group a different multicast address.

Networks that do not have multicasting sometimes still have **broadcasting**, which means that packets containing a certain address (e.g., 0) are delivered to all machines. Broadcasting can also be used to implement groups, but it is less efficient. Each machine receives each broadcast, so its software must check to see if the packet is intended for it. If not, the packet is discarded, but some time is wasted processing the interrupt. Nevertheless, it still takes only one packet to reach all the members of a group.

Finally, if neither multicasting nor broadcasting is available, group communication can still be implemented by having the sender transmit separate packets to each of the members of the group. For a group with n members, n packets are required, instead of one packet when either multicasting or broadcasting is used. Although less efficient, this implementation is still workable, especially if most groups are small. The sending of a message from a single sender to a single receiver is sometimes called **unicasting** (point-to-point transmission), to distinguish it from multicasting and broadcasting.

2.5.2. Design Issues

Group communication has many of the same design possibilities as regular message passing, such as buffered versus unbuffered, blocking versus nonblocking, and so forth. However, there are also a large number of additional choices that must be made because sending to a group is inherently different from sending to a single process. Furthermore, groups can be organized in various ways internally. They can also be addressed in novel ways not relevant in point-to-point communication. In this section we will look at some of the most important design issues and point out the various alternatives.

Closed Groups versus Open Groups

Systems that support group communication can be divided into two categories depending on who can send to whom. Some systems support **closed groups**, in which only the members of the group can send to the group. Outsiders cannot send messages to the group as a whole, although they may be able to send messages to individual members. In contrast, other systems support **open groups**, which do not have this property. When open groups are used, any process in the system can send to any group. The difference between closed and open groups is shown in Fig. 2-31.

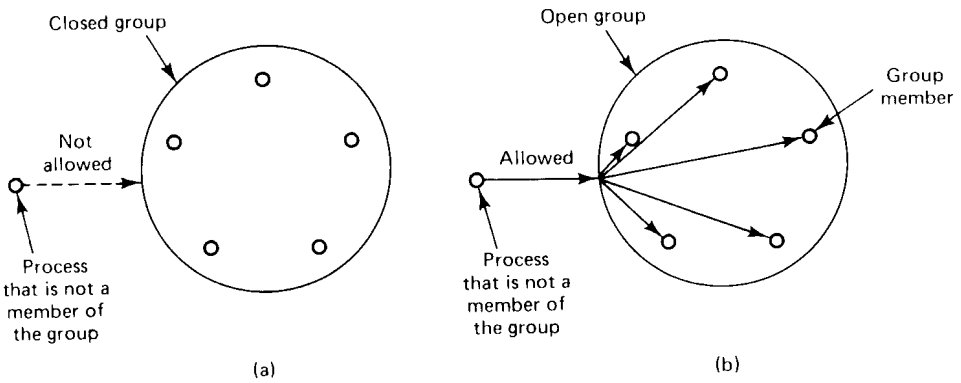

Fig. 2-31. (a) Outsiders may not send to a closed group. (b) Outsiders may send to an open group.

The decision as to whether a system supports closed or open groups usually relates to the reason groups are being supported in the first place. Closed groups are typically used for parallel processing. For example, a collection of processes working together to play a game of chess might form a closed group. They have their own goal and do not interact with the outside world.

On the other hand, when the idea of groups is to support replicated servers, it is important that processes that are not members (clients) can send to the group. In addition, the members of the group may also need to use group communication, for example to decide who should carry out a particular request. The distinction between closed and open groups is often made for implementation reasons.

Peer Groups versus Hierarchical Groups

The distinction between closed and open groups relates to who can communicate with the group. Another important distinction has to do with the internal structure of the group. In some groups, all the processes are equal. No one is boss and all decisions are made collectively. In other groups, some kind of hierarchy exists. For example, one process is the coordinator and all the others are workers. In this model, when a request for work is generated, either by an external client or by one of the workers, it is sent to the coordinator. The coordinator then decides which worker is best suited to carry it out, and forwards it there. More complex hierarchies are also possible, of course. These communication patterns are illustrated in Fig. 2-32.

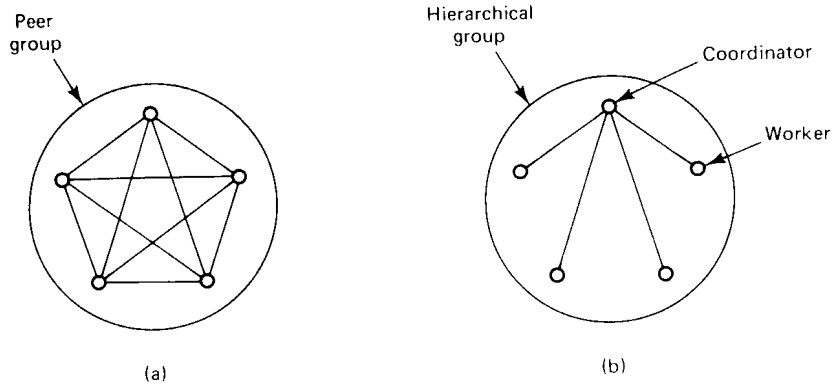

Fig. 2-32. (a) Communication in a peer group. (b) Communication in a simple hierarchical group.

Each of these organizations has its own advantages and disadvantages. The peer group is symmetric and has no single point of failure. If one of the processes crashes, the group simply becomes smaller, but can otherwise continue. A disadvantage is that decision making is more complicated. To decide anything, a vote has to be taken, incurring some delay and overhead.

The hierarchical group has the opposite properties. Loss of the coordinator brings the entire group to a grinding halt, but as long as it is running, it can make

decisions without bothering everyone else. For example, a hierarchical group might be appropriate for a parallel chess program. The coordinator takes the current board, generates all the legal moves from it, and farms them out to the workers for evaluation. During this evaluation, new boards are generated and sent back to the coordinator to have them evaluated. When a worker is idle, it asks the coordinator for a new board to work on. In this manner, the coordinator controls the search strategy and prunes the game tree (e.g., using the alpha-beta search method), but leaves the actual evaluation to the workers.

Group Membership

When group communication is present, some method is needed for creating and deleting groups, as well as for allowing processes to join and leave groups. One possible approach is to have a **group server** to which all these requests can be sent. The group server can then maintain a complete data base of all the groups and their exact membership. This method is straightforward, efficient, and easy to implement. Unfortunately, it shares with all centralized techniques a major disadvantage: a single point of failure. If the group server crashes, group management ceases to exist. Probably most or all groups will have to be reconstructed from scratch, possibly terminating whatever work was going on.

The opposite approach is to manage group membership in a distributed way. In an open group, an outsider can send a message to all group members announcing its presence. In a closed group, something similar is needed (in effect, even closed groups have to be open with respect to joining). To leave a group, a member just sends a goodbye message to everyone.

So far, all of this is straightforward. However, there are two issues associated with group membership that are a bit trickier. First, if a member crashes, it effectively leaves the group. The trouble is, there is no polite announcement of this fact as there is when a process leaves voluntarily. The other members have to discover this experimentally by noticing that the crashed member no longer responds to anything. Once it is certain that the crashed member is really down, it can be removed from the group.

The other knotty issue is that leaving and joining have to be synchronous with messages being sent. In other words, starting at the instant that a process has joined a group, it must receive all messages sent to that group. Similarly, as soon as a process has left a group, it must not receive any more messages from the group, and the other members must not receive any more messages from it. One way of making sure that a join or leave is integrated into the message stream at the right place is to convert this operation into a message sent to the whole group.

One final issue relating to group membership is what to do if so many machines go down that the group can no longer function at all. Some protocol is

needed to rebuild the group. Invariably, some process will have to take the initiative to start the ball rolling, but what happens if two or three try at the same time? The protocol will have to be able to withstand this.

Group Addressing

In order to send a message to a group, a process must have some way of specifying which group it means. In other words, groups need to be addressed, just as processes do. One way is to give each group a unique address, much like a process address. If the network supports multicast, the group address can be associated with a multicast address, so that every message sent to the group address can be multicast. In this way, the message will be sent to all those machines that need it, and no others.

If the hardware supports broadcast but not multicast, the message can be broadcast. Every kernel will then get it and extract from it the group address. If none of the processes on the machine is a member of the group, the broadcast is simply discarded. Otherwise, it is passed to all group members.

Finally, if neither multicast nor broadcast is supported, the kernel on the sending machine will have to have a list of machines that have processes belonging to the group. The kernel then sends each one a point-to-point message. These three implementation methods are shown in Fig. 2-33. The thing to notice is that in all three cases, a process just sends a message to a group address and it is delivered to all the members. How that happens is up to the operating system. The sender is not aware of the size of the group or whether communication is implemented by multicasting, broadcasting, or unicasting.

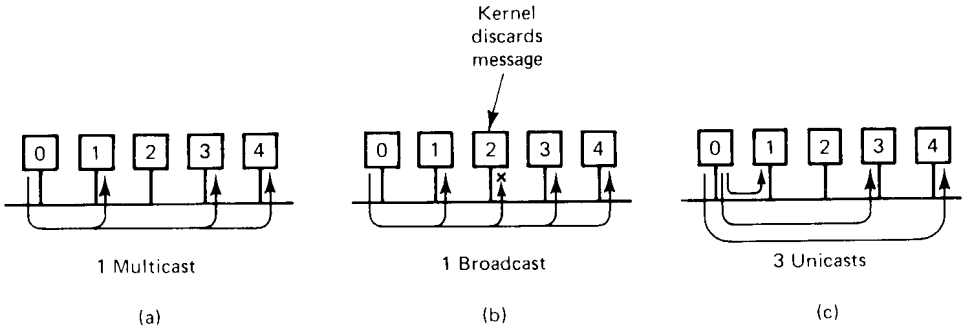

Fig. 2-33. Process 0 sending to a group consisting of processes 1, 3, and 4. (a) Multicast implementation. (b) Broadcast implementation. (c) Unicast implementation.

A second method of group addressing is to require the sender to provide an explicit list of all destinations (e.g., IP addresses). When this method is used,

the parameter in the call to *send* that specifies the destination is a pointer to a list of addresses. This method has the serious drawback that it forces user processes (i.e., the group members) to be aware of precisely who is a member of which group. In other words, it is not transparent. Furthermore, whenever group membership changes, the user processes must update their membership lists. In Fig. 2-33, this administration can easily be done by the kernels to hide it from the user processes.

Group communication also allows a third, and quite novel method of addressing as well, which we will call **predicate addressing**. With this system, each message is sent to all members of the group (or possibly the entire system) using one of the methods described above, but with a new twist. Each message contains a predicate (Boolean expression) to be evaluated. The predicate can involve the receiver's machine number, its local variables, or other factors. If the predicate evaluates to TRUE, the message is accepted. If it evaluates to FALSE, the message is discarded. Using this scheme it is possible, for example, to send a message to only those machines that have at least 4M of free memory and which are willing to take on a new process.

Send and Receive Primitives

Ideally, point-to-point and group communication should be merged into a single set of primitives. However, if RPC is the usual user communication mechanism, rather than raw *send* and *receive*, it is hard to merge RPC and group communication. Sending a message to a group cannot be modeled as a procedure call. The primary difficulty is that with RPC, the client sends one message to the server and gets back one answer. With group communication there are potentially n different replies. How can a procedure call deal with n replies? Consequently, a common approach is to abandon the (two-way) request/reply model underlying RPC and go back to explicit calls for sending and receiving (one-way model).

The library procedures that processes call to invoke group communication may be the same as for point-to-point communication or they may be different. If the system is based on RPC, user processes never call *send* and *receive* directly anyway, so there is less incentive to merge the point-to-point and group primitives. If user programs directly call *send* and *receive* themselves, there is something to be said for doing group communication with these existing primitives instead of inventing a new set.

Suppose, for the moment, that we wish to merge the two forms of communication. To send a message, one of the parameters of *send* indicates the destination. If it is a process address, a single message is sent to that one process. If it is a group address (or a pointer to a list of destinations), a message is sent to all members of the group. A second parameter to *send* points to the message.

The call can be buffered or unbuffered, blocking or nonblocking, reliable or not reliable, for both the point-to-point and group cases. Generally, these choices are made by the system designers and are fixed, rather than being selectable on a per message basis. Introducing group communication does not change this.

Similarly, *receive* indicates a willingness to accept a message, and possibly blocks until one is available. If the two forms of communication are merged, *receive* completes when either a point-to-point message or a group message arrives. However, since these two forms of communication are frequently used for different purposes, some systems introduce new library procedures, say, *group_send* and *group_receive*, so a process can indicate whether it wants a point-to-point or a group message.

In the design just described, communication is one-way. Replies are independent messages in their own right and are not associated with previous requests. Sometimes this association is desirable, to try to achieve more of the RPC flavor. In this case, after sending a message, a process is required to call *getreply* repeatedly to collect all the replies, one at a time.

Atomicity

A characteristic of group communication that we have alluded to several times is the all-or-nothing property. Most group communication systems are designed so that when a message is sent to a group, it will either arrive correctly at all members of the group, or at none of them. Situations in which some members receive a message and others do not are not permitted. The property of all-or-nothing delivery is called **atomicity** or **atomic broadcast**.

Atomicity is desirable because it makes programming distributed systems much easier. When any process sends a message to the group, it does not have to worry about what to do if some of them do not get it. For example, in a replicated distributed data base system, suppose that a process sends a message to all the data base machines to create a new record in the data base, and later sends a second message to update it. If some of the members miss the message creating the record, they will not be able to perform the update and the data base will become inconsistent. Life is just a lot simpler if the system guarantees that every message is delivered to all the members of the group, or if that is not possible, that it is not delivered to any, and that failure is reported back to the sender so it can take appropriate action to recover.

Implementing atomic broadcast is not quite as simple as it looks. The method of Fig. 2-33 fails because receiver overrun is possible at one or more machines. The only way to be sure that every destination receives every message is to require them to send back an acknowledgement upon message receipt. As long as machines never crash, this method will do.

However, many distributed systems aim at fault tolerance, so for them it is essential that atomicity also holds even in the presence of machine failures. In this light, all the methods of Fig. 2-33 are inadequate because some of the initial messages might not arrive due to receiver overrun, followed by the sender's crashing. Under these circumstances, some members of the group will have received the message and others will not have, precisely the situation that is unacceptable. Worse yet, the group members that have not received the message do not even know they are missing anything, so they cannot ask for a retransmission. Finally, with the sender now down, even if they did know, there is no one to provide the message.

Nevertheless, there is hope. Here is a simple algorithm that demonstrates that atomic broadcast is at least possible. The sender starts out by sending a message to all members of the group. Timers are set and retransmissions sent where necessary. When a process receives a message, if it has not yet seen this particular message, it, too, sends the message to all members of the group (again with timers and retransmissions if necessary). If it has already seen the message, this step is not necessary and the message is discarded. No matter how many machines crash or how many packets are lost, eventually all the surviving processes will get the message. Later we will describe more efficient algorithms for ensuring atomicity.

Message Ordering

To make group communication easy to understand and use, two properties are required. The first one is atomic broadcast, as discussed above. It ensures that a message sent to the group arrives at either all members or at none of them. The second property concerns message ordering. To see what the issue is here, consider Fig. 2-34, in which we have five machines, each with one process. Processes 0, 1, 3, and 4 belong to the same group. Processes 0 and 4 want to send a message to the group simultaneously. Assume that multicasting and broadcasting are not available, so that each process has to send three separate (unicast) messages. Process 0 sends to 1, 3, and 4; process 4 sends to 0, 1, and 3. These six messages are shown interleaved in time in Fig. 2-34(a).

The trouble is that when two processes are contending for access to a LAN, the order in which the messages are sent is nondeterministic. In Fig. 2-34(a) we see that (by accident), process 0 has won the first round and sends to process 1. Then process 4 wins three rounds in a row and sends to processes 0, 1, and 3. Finally, process 0 gets to send to 3 and 4. The order of these six messages is shown in different ways in the two parts of Fig. 2-34.

Now consider the situation as viewed by processes 1 and 3 as shown in Fig. 2-34(b). Process 1 first receives a message from 0, then immediately

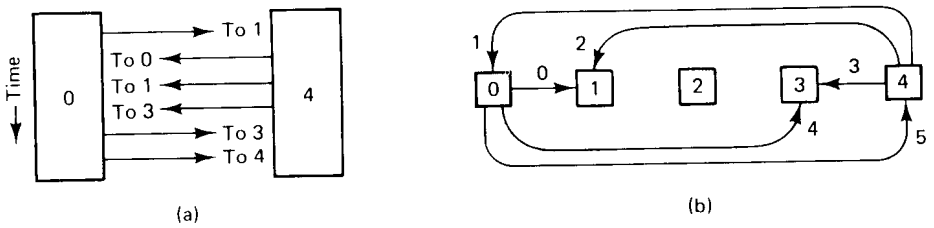

Fig. 2-34. (a) The three messages sent by processes 0 and 4 are interleaved in time. (b) Graphical representation of the six messages, showing the arrival order.

afterward it receives one from 4. Process 3 does not receive anything initially, then it receives messages from 4 and 0, in that order. Thus the two messages arrive in a different order. If processes 0 and 4 are both trying to update the same record in a data base, 1 and 3 end up with different final values. Needless to say, this situation is just as bad as one in which a (true hardware multicast) message sent to the group arrives at some members and not at others (atomicity failure). Thus to make programming reasonable, a system has to have well-defined semantics with respect to the order in which messages are delivered.

The best guarantee is to have all messages delivered instantaneously and in the order in which they were sent. If process 0 sends message *A* and then slightly later, process 4 sends message *B*, the system should first deliver *A* to all members of the group, and then deliver *B* to all members of the group. That way, all recipients get all messages in exactly the same order. This delivery pattern is something that programmers can understand and base their software on. We will call this **global time ordering**, since it delivers all messages in the exact order in which they were sent (conveniently ignoring the fact that according to Einstein's special theory of relativity there is no such thing as absolute global time).

Absolute time ordering is not always easy to implement, so some systems offer various watered-down variations. One of these is **consistent time ordering**, in which if two messages, say *A* and *B*, are sent close together in time, the system picks one of them as being "first" and delivers it to all group members, followed by the other. It may happen that the one chosen as first was not really first, but since no one knows this, the argument goes, system behavior should not depend on it. In effect, messages are guaranteed to arrive at all group members in the same order, but that order may not be the real order in which they were sent.

Even weaker time orderings have been used. We will study one of these, based on the idea of causality, when we come to ISIS later in this chapter.

Overlapping Groups

As we mentioned earlier, a process can be a member of multiple groups at the same time. This fact can lead to a new kind of inconsistency. To see the problem, look at Fig. 2-35, which shows two groups, 1 and 2. Processes *A*, *B*, and *C* are members of group 1. Processes *B*, *C*, and *D* are members of group 2.

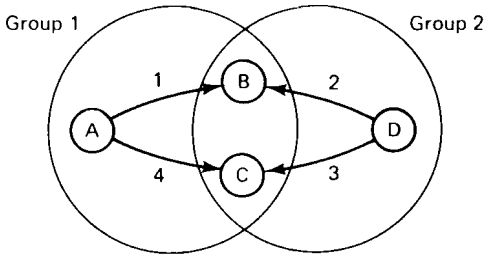

Fig. 2-35. Four processes, *A*, *B*, *C*, and *D*, and four messages. Processes *B* and *C* get the messages from *A* and *D* in a different order.

Now suppose that processes *A* and *D* each decide simultaneously to send a message to their respective groups, and that the system uses global time ordering within each group. As in our previous example, unicasting is used. The message order is shown in Fig. 2-35 by the numbers 1 through 4. Again we have the situation where two processes, in this case *B* and *C*, receive messages in a different order. *B* first gets a message from *A* followed by a message from *D*. *C* gets them in the opposite order.

The culprit here is that although there is a global time ordering within each group, there is not necessarily any coordination among multiple groups. Some systems support well-defined time ordering among overlapping groups and others do not. (If the groups are disjoint, the issue does not arise.) Implementing time ordering among different groups is frequently difficult to do, so the question arises as to whether it is worth it.

Scalability

Our final design issue is scalability. Many algorithms work fine as long as all the groups only have a few members, but what happens when there are tens, hundreds, or even thousands of members per group? Or thousands of groups? Also, what happens when the system is so large that it no longer fits on a single LAN, so multiple LANs and gateways are required? And what happens when the groups are spread over several continents?

The presence of gateways can affect many properties of the implementation. To start with, multicasting becomes more complicated. Consider, for example,

the internetwork shown in Fig. 2-36. It consists of four LANs and four gateways, to provide protection against the failure of any gateway.

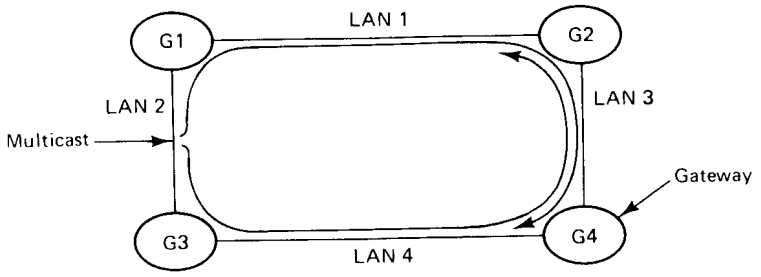

Fig. 2-36. Multicasting in an internetwork causes trouble.

Imagine that one of the machines on LAN 2 issues a multicast. When the multicast packet arrives at gateways *G1* and *G3*, what should they do? If they discard it, most of the machines will never see it, destroying its value as a multicast. If, however, the algorithm is just to have gateways forward all multicasts, then the packet will be copied to LAN 1 and LAN 4, and shortly thereafter to LAN 3 twice. Worse yet, gateway *G2* will see *G4*'s multicast and copy it to LAN 2, and vice versa. Clearly, a more sophisticated algorithm involving keeping track of previous packets is required to avoid exponential growth in the number of packets multicast.

Another problem with an internetwork is that some methods of group communication take advantage of the fact that only one packet can be on a LAN at any instant. In effect, the order of packet transmission defines an absolute global time order, which as we have seen, is frequently crucial. With gateways and multiple networks, it is possible for two packets to be "on the wire" simultaneously, thus destroying this useful property.

Finally, some algorithms may not scale well due to their computational complexity, their use of centralized components, or other factors.

2.5.3. Group Communication in ISIS

As an example of group communication, let us look at the ISIS system developed at Cornell (Birman, 1993; Birman and Joseph, 1987a, 1987b; and Birman and Van Renesse, 1994). ISIS is a toolkit for building distributed applications, for example, coordinating stock trading among all the brokers at a Wall Street securities firm. ISIS is not a complete operating system but rather, a set of programs that can run on top of UNIX or other existing operating systems. It is interesting to study because it has been widely described in the literature and

has been used for numerous real applications. In Chap. 7 we will study group communication in Amoeba, which takes a quite different approach.

The key idea in ISIS is **synchrony** and the key communication primitives are different forms of atomic broadcast. Before looking at how ISIS does atomic broadcast, it is necessary first to examine the various forms of synchrony it distinguishes. A **synchronous system** is one in which events happen strictly sequentially, with each event (e.g., a broadcast) taking essentially zero time to complete. For example, if process *A* sends a message to processes *B*, *C*, and *D*, as shown in Fig. 2-37(a), the message arrives instantaneously at all the destinations. Similarly, a subsequent message from *D* to the others also takes zero time to be delivered everywhere. As viewed by an outside observer, the system consists of discrete events, none of which ever overlap the others. This property makes it easy to understand system behavior.

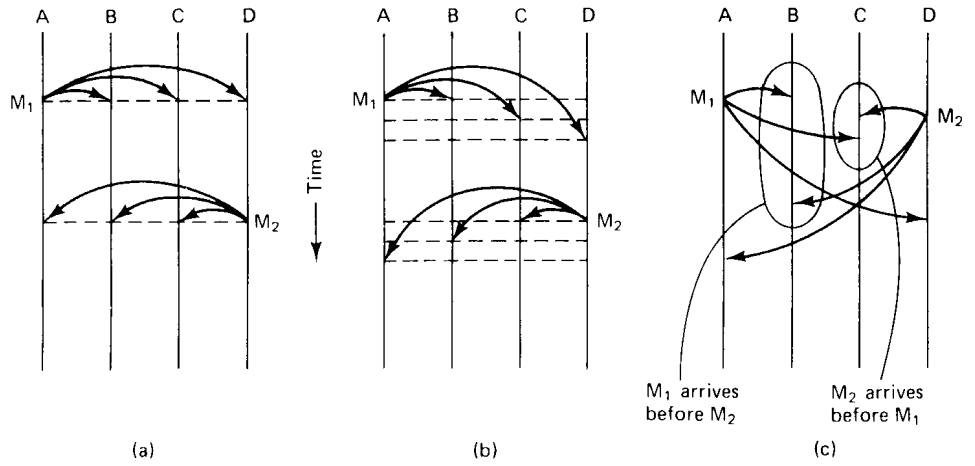

Fig. 2-37. (a) A synchronous system. (b) Loose synchrony. (c) Virtual synchrony.

Synchronous systems are impossible to build, so we need to investigate other types of systems, with weaker requirements on time. A **loosely synchronous system** is one like that of Fig. 2-37(b), in which events take a finite amount of time but all events appear in the same order to all parties. In particular, all processes receive all messages in the same order. Earlier, we discussed essentially the same idea under the name consistent time ordering.

Such systems are possible to build, but for some applications even weaker semantics are acceptable, and the hope is to be able to capitalize on these weak semantics to gain performance. Fig. 2-37(c) shows a **virtually synchronous system**, one in which the ordering constraint has been relaxed, but in such a way that under carefully selected circumstances, it does not matter.

Let us look at these circumstances. In a distributed system, two events are said to be **causally related** if the nature or behavior of the second one might have been influenced in any way by the first one. Thus if *A* sends a message to *B*, which inspects it and then sends a new message to *C*, the second message is causally related to the first one, since its contents might have been derived in part from the first one. Whether this actually happened is irrelevant. The relation holds if there *might* have been an influence.

Two events that are unrelated are said to be **concurrent**. If *A* sends a message to *B*, and about the same time, *C* sends a message to *D*, these events are concurrent because neither can influence the other. What virtual synchrony really means is that if two messages are causally related, all processes *must* receive them in the same (correct) order. If, however, they are concurrent, no guarantees are made, and the system is free to deliver them in a different order to different processes if this is easier. Thus when it matters, messages are always delivered in the same order, but when it does not matter, they may or may not be.

Communication Primitives in ISIS

Now we come to the broadcast primitives used in ISIS. Three of them have been defined: ABCAST, CBCAST, and GBCAST, all with different semantics. ABCAST provides loosely synchronous communication and is used for transmitting data to the members of a group. CBCAST provides virtually synchronous communication and is also used for sending data. GBCAST is somewhat like ABCAST, except that it is used for managing group membership rather than for sending ordinary data.

Originally, ABCAST used a form of two-phase commit protocol that worked like this. The sender, *A*, assigned a timestamp (actually just a sequence number) to the message and sent it to all the group members (by explicitly naming them all). Each one picked its own timestamp, larger than any other timestamp number it had sent or received, and sent it back to *A*. When all of these arrived, *A* chose the largest one and sent a *Commit* message to all the members again containing it. Committed messages were delivered to the application programs in order of the timestamps. It can be shown that this protocol guarantees that all messages will be delivered to all processes in the same order.

It can also be shown that this protocol is complex and expensive. For this reason, the ISIS designers invented the CBCAST primitive, which guarantees ordered delivery only for messages that are causally related. (The ABCAST protocol just described has subsequently been replaced, but even the new one is much slower than CBCAST.) The CBCAST protocol works as follows. If a group has n members, each process maintains a vector with n components, one per group member. The ith component of this vector is the number of the last

message received in sequence from process i. The vectors are managed by the runtime system, not the user processes themselves, and are initialized to zero, as shown at the top of Fig. 2-38.

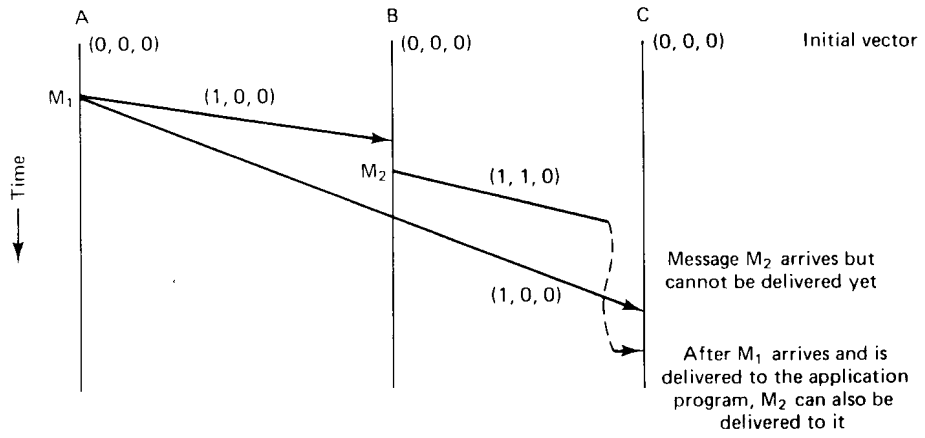

Fig. 2-38. Messages can be delivered only when all causally earlier messages have already been delivered.

When a process has a message to send, it increments its own slot in its vector, and sends the vector as part of the message. When M_1 in Fig. 2-38 gets to B, a check is made to see if it depends on anything that B has not yet seen. The first component of the vector is one higher than B's own first component, which is expected (and required) for a message from A, and the others are the same, so the message is accepted and passed to the group member running on B. If any other component of the incoming vector had been larger than the corresponding component of B's vector, the message could not have been delivered yet.

Now B sends a message of its own, M_2, to C, which arrives before M_1. From the vector, C sees that B had already received one message from A before M_2 was sent, and since it has not yet received anything from A, M_2 is buffered until a message from A arrives. Under no conditions may it be delivered before A's message.

The general algorithm for deciding whether to pass an incoming message to the user process or delay it can now be stated. Let V_i be the ith component of the vector in the incoming message, and L_i be the ith component of the vector stored in the receiver's memory. Suppose that the message was sent by j. The first condition for acceptance is $V_j = L_j + 1$. This simply states that this is the next message in sequence from j, that is, no messages have been missed. (Messages from the same sender are always causally related.) The second condition for acceptance is $V_i \leq L_i$ for all $i \neq j$. This condition simply states that the

sender has not seen any message that the receiver has missed. If an incoming message passes both tests, the runtime system can pass it to the user process without delay. Otherwise, it must wait.

In Fig. 2-39 we show a more detailed example of the vector mechanism. Here process 0 has sent a message containing the vector (4, 6, 8, 2, 1, 5) to the other five members of its group. Process 1 has seen the same messages as process 0 except for message 7 just sent by process 1 itself, so the incoming message passes the test, is accepted, and can be passed up to the user process. Process 2 has missed message 6 sent by process 1, so the incoming message must be delayed. Process 3 has seen everything the sender has seen, and in addition message 7 from process 1, which apparently has not yet gotten to process 0, so the message is accepted. Process 4 missed the previous message from 0 itself. This omission is serious, so the new message will have to wait. Finally, process 5 is also slightly ahead of 0, so the message can be accepted immediately.

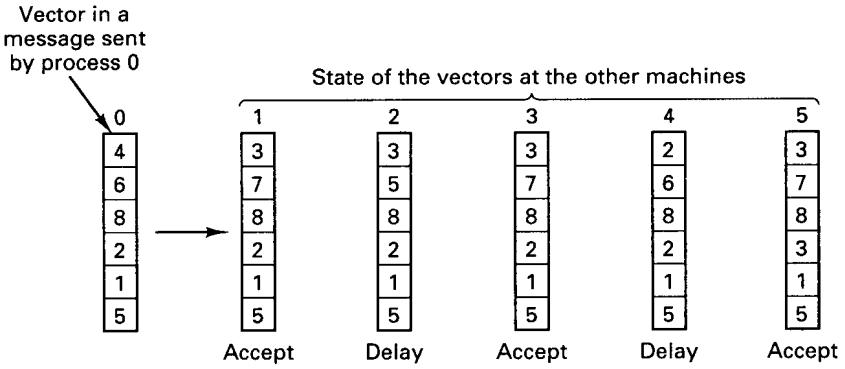

Fig. 2-39. Examples of the vectors used by CBCAST.

ISIS also provides fault tolerance and support for message ordering for overlapping groups using CBCAST. The algorithms used are somewhat complicated, though. For details, see (Birman et al., 1991).

2.6. SUMMARY

The key difference between a centralized operating system and a distributed one is the importance of communication in the latter. Various approaches to communication in distributed systems have been proposed and implemented. For relatively slow, wide-area distributed systems, connection-oriented layered protocols such as OSI and TCP/IP are sometimes used because the main problem to be overcome is how to transport the bits reliably over poor physical lines.