Chapter 4

Reliable, Atomic, and Causal Broadcast

In the previous chapter, we discussed reliable point-to-point communication as one of the basic building blocks. Though point-to-point communication is sufficient for many applications, there are many other applications where a node needs to send a message to many other nodes. In such applications, a one-to-many form of communication is more useful. There are two forms of one-to-many communication: *broadcast* and *multicast*. Broadcast is the communication paradigm where the sender sends a message to all the nodes in the system. In multicast, the sender sends the message to only a subset of the nodes in the system. In this chapter, for ease of exposition, we will focus on broadcasting.

Since the basic communication primitive supported by a network is a one-to-one communication (except in the case of broadcast networks), this communication primitive has to be used to support broadcast and multicast primitives. This makes the implementation of broadcasting protocols susceptible to node and communication failures. It is possible that a sender may fail while broadcasting a message leading to the possibility of only some of the nodes receiving the message. Though this may be acceptable in some applications, this clearly cannot be accepted when building fault tolerant systems. Just like a reliable one-to-one communication primitive is a basic building block, a reliable broadcast primitive is a building block for those fault tolerant systems that employ broadcasting. In this sense, reliable broadcasting is also an abstraction that is not an end in itself but is needed for building fault tolerant applications.

When messages are being broadcast by different nodes in the system, there are three properties of interest: *reliability*, *consistent ordering*, and *causality*

preservation. The reliability property requires that a broadcast message be received by all the operational nodes. The consistent ordering property requires that different messages sent by different nodes be delivered to all the nodes in the same order. Causality preservation requires that the order in which messages are delivered at the nodes is consistent with the causality between the send events of these messages.

These three properties bring in three different types of broadcast primitives: *reliable broadcast*, *atomic broadcast*, and *causal broadcast*. Reliable broadcast supports reliability only, that is, a message that is broadcast is delivered to all alive nodes, even if failures occur in the system. Atomic broadcast, in addition to the reliability property, also supports the ordering property. Causal broadcast ensures that the order in which messages are delivered is consistent with the causal ordering of these messages.

Each of these broadcast primitives has its own applications. If an application sends isolated messages (e.g., an e-mail message, or a news item), reliable broadcast may be enough. However, in database-type applications, it is generally necessary to perform operations in the same order at all the nodes to preserve consistency. Hence, in this case, atomic broadcast would be needed. If the nodes sending the messages also communicate with each other and the contents of the message being broadcast depend on the contents of received messages, then causality must be preserved at the receiving end and so causal broadcast is required. In this chapter, we will discuss all three broadcast paradigms. For each of these we will specify the requirements, and then the methods, for implementing them. We will assume throughout the chapter that failures do not partition the network.

4.1 Reliable Broadcast

In broadcasting, a sender node tries to send a message to all the nodes in the system. Reliable broadcast has one basic property: that a message to be broadcast should be received by all the nodes that are operational. This property should be preserved despite failures. Even if the sender node fails after sending the message to some nodes, this property should be preserved. In this section, we will discuss two protocols that support reliable broadcast.

4.1.1 Using Message Forwarding

We first describe a protocol for reliably broadcasting a message that considers a network as a tree [SGS84]. This tree is used as the basis of disseminating the message to all the nodes. The root of the tree is the original sender (or the initiator)

of the broadcast message. If there is an edge from a node P in the tree to another node Q, it implies that during broadcasting the node P will forward the message to node Q. This rooted tree represents a *broadcast strategy* and could be "flat," or a linear chain, or something else.

This tree is a logical structure used to organize the nodes in the network and has no direct relationship with the physical structure of the network. How the structure of the tree is decided is not an issue for this protocol, though it is clear that it will be more efficient if the neighbor of a node in the tree is a neighbor of the node in the underlying physical network also. The tree is statically defined and is known to all the nodes in the system.

A relation *SUCC* is defined on the given tree (V, E). This relation captures the hierarchy of the tree. For a node P, *SUCC(P)* represents the set of successor nodes of P in the tree. For a set of nodes X, *SUCC(X)* is the set of successors of the nodes in X. Let the root of the tree (the broadcast initiator) be node S (source). The protocol has to ensure that if a message m is broadcast by S, then all nodes that have not failed will receive m.

The set of all failed nodes is represented by *FAILED*. If a node fails, we assume that all other nodes find out about the failure in a finite time. We assume that each node has a copy of the set *FAILED*. In the previous chapter, we discussed how this can be achieved by fault diagnosis.

The basic strategy for broadcasting is as follows [SGS84]. Starting from the root of the tree, the message is forwarded along the edges of the tree. A node i, on receiving a message, forwards it to all its successor nodes, which send an acknowledgment back to i. If the node i does not get an acknowledgment from a node j (in *SUCC(i)*), and finds that j has failed, it assumes that j failed before completing its task and that the successors of j may not have received the message. Hence, i takes over the role of j and forwards the message to nodes in *SUCC(j)*. This may result in duplicates, but duplicates can easily be detected by using sequence numbers.

This strategy works well for handling the failures of all nodes other than the root, since the failure of a node is detected by its parent node which completes the task that had to be performed by the failed node. However, the root node has no parent. Hence, a special situation arises if it fails before forwarding the message to all its successors.

If the root node S does not forward the message to any of its successors, then it is all right, since no alive node has received the message. However, if S had failed after sending the messages to some, but not all, of its successor nodes, then something has to be done. Essentially, some other node that has received the message has to complete the task of S to ensure that the reliable broadcast property is satisfied. Multiple nodes performing this does not cause any problem, since duplicate messages

can be detected.

For this, a node i, on receiving a message m from S, monitors S until it recognizes that S has failed or that its broadcast has been successful. If i detects the failure of S, it takes over the job of S if the broadcast of S was not successful. To help other nodes detect whether S has completed its broadcast successfully, S informs its successors when its broadcast is successful (i.e., it has received acknowledgments from all its successors). If we assume that a node does not initiate any new broadcast until its previous broadcast is successfully completed, a node i can infer successful completion of a broadcast when it receives a new broadcast message from S, that is, it receives a broadcast message which has a larger sequence number.

Each node i executes the same protocol, except for the root node, whose protocol is slightly different (since it initiates the broadcast). Each node i maintains three sets of nodes: *sendto*, *ackfrom*, and *ackto*. These represent the nodes to which a message must be sent, the nodes from which acknowledgments are expected, and the set of nodes to which acknowledgment has to be sent, respectively. For clarity, we use the operation $send(k, ack(m))$ to signify that a message carrying the acknowledgment for the message m is being sent to node k. Similarly, a primitive $receive(ack(a))$ is used to signify the receipt of a message a that is an acknowledgment. The protocol for a node i is shown in Fig. 4.1.

In this protocol, in the first guard, the protocol sends a message to one of the nodes to which it needs to send a message and updates *sendto* and *ackfrom* appropriately. In the second guard, if it finds that some of the nodes from which it is expecting an acknowledgment has failed, it adds the successors of the failed nodes to its $sendto$ and appropriately modifies $ackfrom$. The third guard is straightforward: if an acknowledgment is received, then *ackfrom* is updated. In the fourth guard, if the node finds that it has completed its own broadcast (signified by $ackfrom = sendto = \phi$), and that the root S has failed, then it takes over the role of S. The variable r represents the node whose successors the protocol is trying to cover by broadcasting; usually it is i, but if S fails, it is set to S. The actions the protocol performs upon receiving a new message are shown in the next guard. The actions depend on the sequence number. If the sequence number is the same as the sequence number of the current message, then it is recorded that an acknowledgment also has to be sent to the sender of the message. If the sequence number of the new message is smaller, this means that it is an old message, and an acknowledgment is sent to the sender. If the sequence number of the new message is larger, this means it is a new message, and *ackto* is set to the sender of the message and *sendto* is set to the successors. Finally, in the last guard, if a node i has no message to send and is not expecting any *acks*, but has some *acks* to send, it sends those *acks*.

The protocol given in Fig. 4.1 is for a node i that is not the root of the tree. For

```
m.sender := S
r := i
sendto, ackfrom, ackto := φ
*[
        sendto ≠ φ →
                from sendto chose a node i
                sendto := sendto − {i}
                send(i, m)
                ackfrom := ackfrom ∪ {i}
        □ackfrom∩ FAILED ≠ φ →
                t := ackfrom∩ FAILED
                sendto := sendto∪ SUCC(t)
                ackfrom := ackfrom − t
        □receive(ack(a)) →
                if a.seqno = m.seqno then ackfrom := ackfrom − {a.sender}
        □S ∈ FAILED ∧r ≠ S ∧ ackfrom = φ ∧ sendto = φ →
                r := S
                sendto := SUCC(S)
        □receive(new) →
                if new.seqno = m.seqno then ackto := ackto ∪ {new.sender}
                if new.seqno < m.seqno then send(new.sender, ack(new))
                if new.seqno > m.seqno then
                        forall p ∈ ackto do send(p, ack(m))
                        m := new
                        r := i
                        ackto := {m.sender}
                        sendto := SUCC(i)
                        ackfrom := φ
        □ackto ≠ φ ∧ (r = b ∨ (sendto = φ ∧ ackfrom = φ)) →
                for all p ∈ ackto do send(p, ack(m))
                ackto := φ
]
```

Figure 4.1: The broadcast protocol for a node i [SGS84]

the root node S, the guard beginning with $S \in FAILED$ is replaced by another where it accepts a message if its $ackfrom$ and $sendto$ are empty, prepares the message m, and sets $sendto = SUCC(S)$.

This protocol ensures that if a message sent by the root node S has reached even one node that has not failed, then the message reaches all the nodes that have not failed. Formal proofs of many properties are given in [SGS84]. For messages to be sent by different nodes, copies of the protocols, but with different root nodes, will have to be executed. That is, for each node, a rooted tree is defined, and a protocol for that rooted tree operates at different nodes. However, the different protocols for different trees can be easily combined into a single process that will perform all the tasks.

4.1.2 An Approach by Piggybacking Acknowledgments

Now we describe the *Trans protocol* for reliable broadcasting [MMA90, MSM94]. The Trans protocol uses a combination of positive and negative acknowledgments to achieve the reliable broadcast property. By piggybacking acknowledgments (acks) and negative acknowledgements (nacks) on messages that are being broadcast by nodes, it simplifies the detection of missed messages, and minimizes the need for explicit acknowledgments.

The protocol assumes that when a node broadcasts a message, some nodes receive it and some nodes miss it. This type of physical communication medium exists, for example, in the Ethernet. An underlying unreliable broadcast protocol in a point-to-point network will also have the same effect. The Trans protocol builds a reliable broadcast primitive from the unreliable broadcast primitive which it assumes is available to it.

The basic idea of the protocol is to piggyback acknowledgments and negative acknowledgments on a broadcast message. Each broadcast message carries the identity of the sender node, and a unique sequence number for the message. From the acknowledgments and negative acknowledgments, a receiving node knows which message it does not need to acknowledge, or which messages it has missed and must request other processor to retransmit. The idea behind the protocol is illustrated by this sequence of events in a system consisting of three nodes (or processes): P, Q, and R [MMA90]:

1. Node P broadcasts a message m_1.

2. Node Q receives the message and piggybacks a positive acknowledgment on the next message that Q broadcasts, say, m_2.

3. On receiving m_2:

 - If R had received m_1, it realizes that it does not need to send an acknowledgment for it, as Q had acknowledged it.
 - If R had not received m_1, it knows about this loss by the acknowledgment on m_2, and requests retransmission by sending a negative acknowledgment in the next message it broadcasts.

From this it is clear that by the effective use of acknowledgments and negative acknowledgments, the Trans protocol can support reliable broadcast efficiently. When a retransmission is requested, any node (not just the original sender of the message) may retransmit it. A retransmitted message is identical to the original message and contains the same contents.

To support the protocol, each node maintains an *ack-list*, a *nack-list*, a *received-list*, and a *Pending Retransmissions list (PR-list)*. The ack-list contains the message identifiers of messages for which this node has to send an acknowledgment; the nack-list contains the message identifiers of messages for which this node has to send a negative acknowledgment. The received list contains the messages that this node has received or has sent recently, which may need to be retransmitted. Messages are deleted from this list when no retransmission of the message could be needed by any processor. The PR-list contains the message identifiers of the messages whose retransmission has been requested by some node.

To support the Trans protocol, special actions have to be performed by a node when sending or receiving a broadcast message. *Sending a message* by a node is straightforward. When a node has a new message m to broadcast, it executes the following steps:

1. Append ack-list to m (as $m.acks$); reset ack-list to empty.
2. Append nack-list to m (as $m.nacks$).
3. Broadcast m.

In addition to sending its own messages, a node also sends messages in its PR-list. Furthermore, if a node does not receive a positive acknowledgment of a message within some time interval, it adds the message to the PR-list (and later broadcasts it again). On receiving a message m, a node performs the actions shown in Fig. 4.2 ($m.id$ refers to the message-id of m).

On receiving a message m, it is saved in the received-list and its id is added to the ack-list, representing that m has to be acknowledged. If $m.id$ is in the nack-list, it is deleted, as the message has been received and there is no need to send any negative

```
add m.id to ack-list
add m to received list
if m.id ∈ nack-list then delete it.
if m ∈ PR-list then delete it.

for all id, such that id ∈ m.acks do
    if id ∈ ack-list then delete it
    if message corresponding to id ∉ received-list then add id to nack-list

for all id such that id ∈ m.nacks do
    if message corresponding to id ∈ received-list
        then add message corresponding to id to PR-list
    else add id to nack-list
```

Figure 4.2: Receiving a message in the Trans protocol

acknowledgment. Similarly, if m was in the PR-list, it is deleted, since the arrival of the message means that some other node has satisfied the retransmission request, hence this node need not send the message.

After this, the acknowledgments and the negative acknowledgments in the message m are processed. All the messages for which acknowledgments are in m need not be acknowledged by this node now, and hence are deleted from the ack-list. If a message that is acknowledged in m is in the ack-list, it is deleted, as no acknowledgment need be sent by the node. If the message acknowledged in m is not in the received-list, then the node has missed the message and its id is added to the nack-list. If a message is negatively acknowledged in m, then if that message has been received by the node, it is added to the PR-list, as the sender of m has requested retransmission. If the negatively acknowledged message has not been received by the node, its id is added to nack-list.

Besides these actions, if the sequence number $m.id$ of m indicates that messages from the sender have been missed, then the missing sequence numbers are also added to nack-list. Sequence numbers can also be used to detect duplicates, if needed.

Let us consider a few examples to illustrate the working of the Trans protocol. In the examples, we will use A, B, C, D, etc. to represent messages, a, b, c, d, etc. to represent acknowledgments for the messages, and $\bar{a}, \bar{b}, \bar{c}, \bar{d}$, etc. to represent negative acknowledgments for the messages. In the examples, we do not specify the source of a message, as it is not particularly significant. Consider the following

sequence of messages that is broadcast [MMA90]:

$$A \ Ba \ Cb \ Dc \ E\bar{c}d \ Cb \ Fec.$$

First message A is sent, which is acknowledged by the sender of B by piggybacking acknowledgment a with B. On seeing this acknowledgment, no other node that receives B will send an acknowledgment for A. The message C carries an acknowledgment for B, and the message D carries an acknowledgment for C. The sender node for the message E has not received C, but by receiving D and seeing the acknowledgment c, it knows it has missed C, and so sends $\bar{c}$ as well as d along with its message E. On receiving this message, some node retransmits C. Note that the retransmitted message is the same as the original message and is not used to acknowledge recent messages. The sender of F acknowledges both E and C in the message. By these acknowledgments it implicitly acknowledges messages D and B (whose acknowledgments came with messages E and C) as well. Thus each message typically contains a few acknowledgments, but implicitly acknowledges many more messages.

This example illustrates how the acknowledgments work and how a message missed by a node is requested and retransmitted. Now let us see what happens if a series of messages is missed. Consider the following sequence of messages [MMA90]:

$$A \ Ba \ Cb \ Dc \ E\bar{c}d \ Cb \ F\bar{b}ec \ Ba \ Gfb.$$

In this example, we consider a situation in which the sender of E received neither C nor B, but received D. From the message for D, it detects that it has missed C, but still does not know about missing B. Hence with E, it sends a negative acknowledgment for C. When C is retransmitted and received by this node, the positive acknowledgment for B in this message will alert the node about missing B, a negative acknowledgment which can then be included in the next message (F in this example). Hence, if a sequence of messages is missed, they are not all detected together, but in a transitive fashion. Loss of some messages is first detected by the receipt of acknowledgments on the message that has been received. These messages are then retransmitted. Some other missed messages are detected when these retransmitted messages are received. This transitivity ensures that all messages that are missed are detected and finally retransmitted. In some cases, more than one message may acknowledge a message. But this does not cause any problem in the working of the protocol.

It is clear that for any message, if some working node has received a message and it transmits messages in the future, then all the working nodes will eventually receive it, assuming that all nodes have messages to send or they send dummy messages

to transfer information about their ack-list and nack-list to others. This happens, since on the receipt of a message, some missing messages are detected, which causes further detection of missing messages, until all such messages are detected (since for each missed message there is some node that has received it). It should also be clear that the different nodes may receive the messages in a different order (e.g., one node may receive the message in the original transmission, while the other may receive it in a retransmission after receiving other messages). As we will see later, this protocol can be extended to ensure that all operational nodes deliver the messages in the same order.

4.2 Atomic Broadcast

The atomic broadcast paradigm for one-to-many communication is stricter than the reliable broadcast paradigm. Not only does it require that a message sent by a node be received by all the operational nodes, but it also requires that if multiple messages are broadcast by different nodes, then the different messages must be delivered at all the nodes in the same order. Hence, in addition to reliability, the same order of delivery at all nodes is an additional requirement. And both of these requirements should be satisfied even during the occurrence of failures. We have already seen how reliability can be satisfied. The focus in atomic broadcast is ensuring the same ordering of messages at different nodes.

With atomic broadcast, we need to distinguish between a node *receiving* a message and a node *delivering* it. Receiving a message means that the node has received the message using its network interface. Typically, a message sent to a node is meant for some process (probably a user process) running on the node. Hence, after receiving a message, the node (or rather the operating system on the node) has to deliver the message to the process. With reliable broadcast, it was implicitly assumed that when a node received a message, it delivered it to the higher layers for consumption. However, in atomic broadcast, after receiving a message, it is first buffered, and is later delivered. An atomic broadcast protocol has to ensure that the messages are delivered in a consistent order, which may not be the same as the order in which the messages are received. Frequently, a node has no control over the order in which messages are received by it, but it can exercise control on the order in which they are delivered. Hence, in atomic broadcast, the order in which the messages are delivered is significant, and may be different from the order in which messages are received.

Atomic broadcast is frequently needed in managing process groups, replicated data, replicated processes, etc. It is a very useful primitive for constructing fault

tolerant systems. Consequently, a large number of protocols have been proposed for atomically broadcasting a message. In this section, we will discuss a few of these protocols.

4.2.1 Using Piggybacked Acknowledgments

Earlier, we discussed the Trans protocol for reliable broadcast. The protocol ensures that a message is successfully received by all operational nodes, but does not guarantee that different messages are received by different nodes in the same order. Here we will see how that protocol can be extended to satisfy the ordering property as well.

In the Trans protocol, since acknowledgments and negative acknowledgments are appended in the message itself, and the message is broadcast, a node can determine if another node has received a message. We define an *Observable Predicate for Delivery*, denoted by $OPD(P, A, C)$, where P is a node, and A and C are messages. We denote the sender of a message A by P_A. If $OPD(P, A, C)$ is true, it states that the node P is certain that P_C has received and acknowledged (directly or indirectly) the message A at the time of broadcasting C. The node P can evaluate the predicate based on the messages it receives. This predicate is true if, and only if, from the sequence of all the messages received, by deleting some of the messages, P can form a sequence S_M of messages such that [MMA90]:

1. S_M commences with message A and ends with message C.
2. Every message of S_M, other than A, positively acknowledges the predecessor in S_M, or is broadcast by the same node as its predecessor.
3. No message in S_M is negatively acknowledged by C.

Essentially these properties say that P has received a sequence of messages (not necessarily consecutively) in which the acknowledgments, starting from the acks in C, transitively acknowledge A. For example, suppose that the sequence of messages that are transmitted by four different processors is [MMA90]:

$$B_1 \; D_1 \; A_1 d_1 \; C_1 \bar{d}_1 b_1 a_1 \; D_2 \bar{a}_1 c_1 \; D_1 \; C_2 d_2 d_1 \; B_2 \bar{a}_1 c_2.$$

The acknowledgments and negative acknowledgments of the messages can be represented as a graph, where nodes are the messages and arcs represent the acknowledgments of messages. If a message m_1 acknowledges a message m_2, then there is an arc in the graph from m_1 to m_2. Negative acknowledgments are

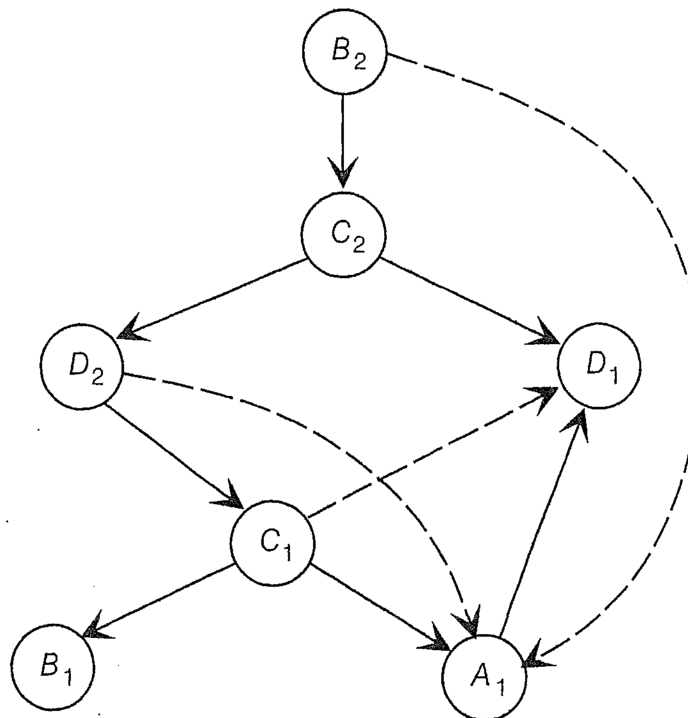

Figure 4.3: Graphical representation of the sequence of messages

represented as dashed arcs. The graph for the sequence of messages given above is shown in Fig. 4.3.

Note that the message D_2 is considered to implicitly acknowledge D_1, as it is sent by the same node. This graph is for the global sequence of messages that are transmitted. At a given time, the graph at a node will depend on the sequence of messages received by that node. However, since Trans supports reliable broadcast, and since a retransmission is exactly the same as the original message, eventually all nodes will have a graph that is the same as the global graph. If a node receives a message m_1 before transmitting m_2, then there will be a path from m_2 to m_1. For example, in the graph in Fig. 4.3, there is a path from the node B_2 to the node C_1. There is an arc from B_2 to C_2, implying that B_2 has acknowledged C_2. But C_2 contains acknowledgments for D_2 and D_1. Hence, at the time of broadcasting B_2, the processor P_B must have received these messages, else it would have included a negative acknowledgment for them. Again, D_2 contains a negative acknowledgment for A_1 and a positive acknowledgment for C_1. Since B_2 does not contain any negative

acknowledgment for C_1, P_B must have received it at the time of sending B_2. Since it has not received A_1, a negative acknowledgment is attached to B_2.

$OPD(P, A, C)$ represents that there is a path from C to A in the graph formed by the messages received by P and there is no negative acknowledgment edge from C to any node in the path from C to A. That is, C transitively acknowledges A. OPD can be used to define a partial order on the sequence of messages, as follows.

In the *partial order* constructed by P, a message C *follows* a message B if, and only if, $OPD(P, B, C)$ and for all messages A, $OPD(P, A, B)$ implies $OPD(P, A, C)$.

In the partial order, if C follows A, it implies that C acknowledges (directly or indirectly) the message A and also all the messages that A acknowledges. If C is included in the partial order, it means that at the time of transmitting C, the processor P_C had received and acknowledged, directly or indirectly, all messages that precede C in the partial order. For the graph shown in Fig. 4.3, the partial order is shown in Fig. 4.4.

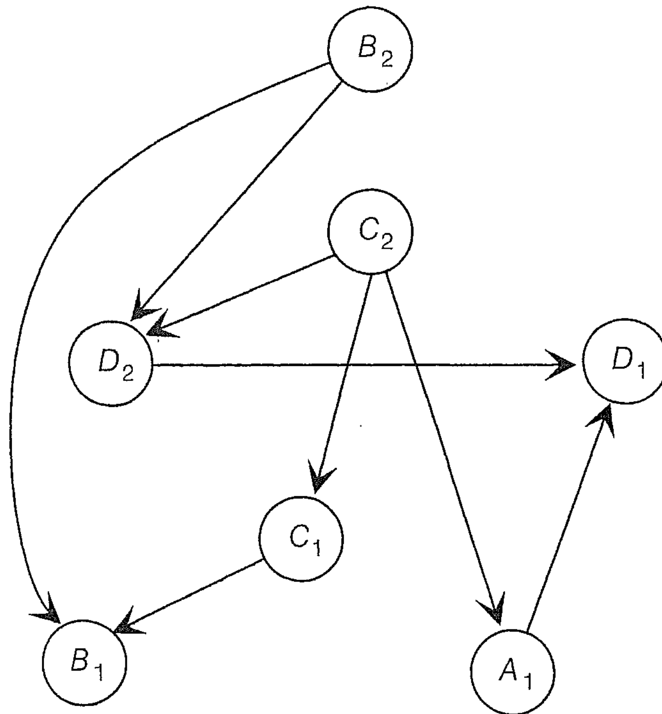

Figure 4.4: Partial order corresponding to the graph

Note that in this partial order, message C_1 does not follow A_1 (even though it contains an ack a_1) because A_1 follows D_1, but C_1 has a negative acknowledgment for D_1. Similarly, B_2 does not follow C_2 because of A_1. However, C_2 follows A_1, since there is a path from C_2 to A_1, and also a path to any node to which there is a path from A_1 (which is only the node D_1).

Since the Trans protocol ensures that all operational nodes eventually receive each broadcast message, all operational nodes will have the same partial order eventually. However, at any given time, the partial orders at different nodes may be different, as they may have received a different set of messages. Typically, the partial orders of nodes will differ only in the recent messages. Node failures can cause further transient discrepancies between the partial orders of different nodes at a given time. This partial order can be converted into a total order by the *Total protocol* [MMA90, MMSA93].

The Total protocol needs no additional messages beyond those required by the Trans protocol. However, a message is not placed in the total order by a node immediately after it is received. A node must wait to receive more messages from other nodes before it can add a message to the total order. That is, the protocol incrementally extends the total order. The Total protocol is also resilient to node failures. Though it can handle multiple node failures, in our discussion we will limit attention to the single-node failure case only.

A message that follows in the partial order only those messages that are already in the total order (or follows no message) is a *candidate* message for inclusion in the total order. Each set of candidate messages (called a *candidate set*) is voted by the messages that precede the candidates in the partial order. This "voting" is not an actual voting involving messages; rather it is an evaluation based on the messages received. Hence the decision of a node about including a message in the total order is only dependent on the sequence of messages it receives.

Voting on a candidate set (CS) takes place in stages. The number of voting stages depends on the candidate set and the partial order. In stage 0, the vote of a message is based on the precedence in the partial order. In stage i ($i > 0$) it also depends on which messages voted in stage $i - 1$. Stage i voting also requires the parameter N_v, which is $(n - 1)/2$ (for a resiliency of 1). A message m votes for a candidate set CS as follows [MMA90]:

Stage 0:

- m votes for CS if CS contains only m.
- m *votes for* CS, if (i) no message from the sender of m that precedes m has voted for CS, (ii) m follows every message in CS, and (iii) m follows no other

candidate message. It *votes against* CS if it follows some other candidate message.

Stage i:

- m votes for CS if (i) no message from the sender of m that precedes m has voted for CS, (ii) the number of messages that had voted for CS in stage $i-1$ that m follows in the partial order is at least N_v, and (iii) it follows fewer messages that voted against CS than voted for CS in stage $i-1$.

- m votes against CS if the number of messages that had voted against CS in stage $i-1$ that m follows in the partial order is at least N_v, and it does not vote for CS in stage i.

It is clear from these rules of "voting" that a node can determine its votes from the messages it receives. From these votes a node decides how and when to add messages to the total order. The voting rules and the decision criteria together ensure that all nodes form the same total order from the partial order. The decision criteria for a node P is given below. The decision criteria requires the parameter N_d which is $(n+2)/2$ (for a 1-resilient system).

The Decision Criteria:
In stage i where $i > 0$

- P decides for CS if the number of messages in its partial order that voted for CS in stage i is at least N_d, and for each proper subset of CS, P has decided against it.

- P decides against CS if the number of messages in its partial order that voted against CS in stage i is at least N_d.

Once a decision is made in favor of a candidate set, the messages of that set are included in the total order in some deterministic order. The whole process is then repeated with the new set of candidate messages. Note that by the way the partial order is constructed, a node can always determine the vote of a message in its partial order, since all messages that precede this message in the partial order must have been received by the node. The Total protocol ensures [MMA90] that: (i) If a node decides for (against) a candidate set, then all nodes decide for (against) that set. (ii) If a node includes a candidate set as its jth extension to its total order, then all nodes include that set as their jth extension. Consequently, the total orders of all nodes are the same. (iii) The total order is consistent with the partial order.

Let us illustrate the Total protocol by use of examples. First, let us consider a 1-resilient, six node system. In this system, four votes are needed for a decision to include a candidate set in the total order. Assume that the transmissions are all reliable and all nodes receive all the transmitted messages. The partial order of such a system is a linear chain, and is shown for five messages in Fig. 4.5.

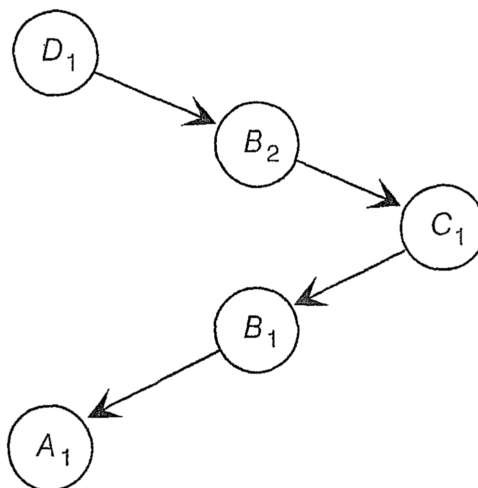

Figure 4.5: A simple partial order that converts easily into a total order

The graph with acknowledgments will also be the same, as there are no negative acknowledgments in the system. At the start, there is only one candidate message, A_1, and so only one candidate set $\{A_1\}$. The messages A_1, B_1, C_1, and D_1 vote for this candidate set (B_2 does not, since B_1 is voting). Four votes are sufficient for a decision. Hence, whenever a node receives the message D_1, it can decide to include A_1 in the total order. Note that until D_1 is received, a node cannot decide on A_1.

Let us now consider the partial order shown earlier in Fig. 4.4. Assume that there are four nodes in the system. Hence, three votes are needed for a decision. The candidate messages are B_1 and D_1 and the candidate sets are $\{B_1\}$, $\{D_1\}$, and $\{B_1, D_1\}$. Only messages C_1 and B_1 vote for $\{B_1\}$, which is not sufficient for a decision. Similarly, A_1 and D_1 vote for $\{D_1\}$, which is not sufficient for a decision. Hence, both these candidate sets are rejected. For the candidate set $\{B_1, D_1\}$, the messages C_1, A_1, D_2 and B_2 vote, which is sufficient to decide. Hence, this candidate set is chosen for inclusion in the total order. After these messages have been added, the candidate messages are A_1, C_1, and D_2. It can be seen that no candidate set of these messages will get the required three votes. Hence, no further addition to the

total order will be made till enough new messages are received to obtain the required number of votes.

4.2.2 A Centralized Method

Now we discuss an approach in which consistent ordering of messages is guaranteed by conceptually funneling each message through a centralized message exchange [CM84]. Like the Total protocol, this protocol also assumes that the underlying communication is broadcast. The protocol converts the unreliable broadcast communication available to it into an atomic broadcast primitive.

The basic idea behind the protocol is simple. If multiple nodes broadcast messages, there is no guarantee of the order in which the messages will reach the different destinations. However, if all messages are broadcast through a centralized message exchange, then the order in which the messages will be received at different nodes will be the same as the order in which the centralized message exchange sends them. This order may be the same as the order in which the message exchange receives the messages from the different original senders of the broadcast messages. This centralized approach can guarantee consistent ordering at all nodes, but it is not resilient; the failure of the message exchange can disrupt the process. Hence, in this protocol, the centralized message exchange is rotated between different nodes.

In the protocol, the senders do not actually send the message through the message exchange (called the *token site* in the protocol). Instead, a sender node directly transmits the message using the (unreliable) broadcast primitive available. On receiving these messages, nodes save them in a buffer queue Q_B. The token site, which is one of the receiver nodes, assigns the message a unique global sequence number, $gseq$, and transmits it to all nodes through an acknowledgment message. This global sequence number is used to determine the order in which the messages in Q_B are delivered by nodes, and also to detect missing messages. The token site is rotated among a set of nodes called the *token list*.

Normal Phase

The protocol has two phases: a *normal phase* and a *reformation phase* [CM84]. The normal phase consists of activities performed if no failure occurs, while the protocol gets into the reformation phase when some nodes fail. For the normal phase, each node i maintains the following information:

- $M_i[j]$: The sequence number of the next broadcast message it expects from a node j. A node assigns sequence numbers consecutively, and hence receipt of a message from j with a higher sequence number than expected tells the receiver that it has missed some messages that were earlier sent by j.

- $gseq_i$: The next global sequence number it expects. Again, if a node receives a global sequence number from the token site that is higher than what it expects, it knows that it has missed some global sequence numbers sent earlier.

Initially, all nodes have the same $gseq$ (say 0), and the same sequence number it expects from other nodes (say 0). During the normal phase, there are three major activities that are performed for atomically broadcasting the messages: transmitting, assigning global sequence numbers, and committing [CM84].

Transmitting. A node transmits (i.e., broadcasts) a message. The current token site, if it receives the message, sends an acknowledgment to the sender node. The sender node repeatedly transmits the message until it receives an acknowledgment from the token site. Each broadcast message contains $< j, n >$, where j is the node identifier and n is the sequence number j has assigned to the message. This sequence number represents the number of messages that have been broadcast by node j, hence successive messages from j have sequential numbers.

Assigning global sequence number. The token site acknowledges messages broadcast by nodes. If the token site i receives a message with $< j, n >$ such that $M_i[j] = n$, it assumes that this message has not been acknowledged, and transmits an acknowledgment $ACK(gseq_i, < j, n >)$. Each ACK message, besides acknowledging the sender, also signifies the transfer of the token site to the next node in the token list. After transmitting an acknowledgment, it increments its $gseq_i$ and $M_i[j]$.

The nodes save the messages they receive in Q_B. A node processes the ACK messages $ACK(seq, < j, n >)$ in the order they arrive. An ACK message is processed by a node k only if $seq = gseq_k$ and the message corresponding to $< j, n >$ is in Q_B. When an acknowledgment is processed, $gseq_k$ is incremented, and $M_k[j]$ is set to $n + 1$. If for an ACK message, the node k does not have the message corresponding to $< j, n >$ in its Q_B, it implies that it has missed this message earlier, and it transmits a request for its retransmission. If $seq < gseq_k$, this acknowledgment is a duplicate and is not processed. If $seq > gseq_k$, it means that the node k has missed some previous ACK messages, and it transmits a request for retransmission of messages between $gseq_k$ and seq. All retransmission requests are satisfied by the token site, and a node repeatedly transmits its request for retransmission until it gets the requested message or ACK.

Committing. After the message has been assigned a global sequence number and the token has been successfully transferred L times, it is certain that at least the $L + 1$ token sites have successfully received the broadcast message. At this time, the message is "committed." As long as L of fewer nodes in the token list fail, all

committed messages can be recovered. As the token is transferred with an ACK message, a null ACK is sent to transfer the token site in case there are no broadcast messages. The committed messages are delivered by nodes in the order of their global sequence numbers. This ensures that all nodes will deliver the messages in the same order.

In this method, the token site is responsible for acknowledging a message and thereby assigning a global sequence number to it, and also for transferring the token site. Since the token is also transferred as a part of the ACK message, token transfer does not require any additional messages. A token site accepts the transfer from the previous token site, if the global sequence number in the ACK message is the same as the node expects, and the corresponding message is in the queue. Otherwise, it waits till it receives all the required messages. Therefore, when a token site accepts a transfer of token, it has received all the messages and acknowledgments that may be requested for retransmission later. This property makes the token site capable of satisfying a retransmission request.

Reformation Phase

The protocol enters the *reformation phase* when a failure or recovery is detected [CM84]. The token list initially consists of all the nodes. Failure of some of these nodes can disrupt the token-passing mechanism. Hence, the reformation process redefines the token list. The new list will consist of only operational nodes. When a new list is formed, the protocol resumes normal operation.

Any site that detects a failure initiates a reformation, and is called the *originator*. Since there could be different token lists at different times, a *version number* is associated with a token list. A new token list will always have a higher version number than the old token list. Since multiple sites may initiate reformation, the reformation protocol has to ensure that after the reformation there is exactly one *valid token list*. The process also has to make sure that none of the messages that was committed from the old token list are lost.

An originator first asks the other nodes to join in forming a list. The originator also chooses the version number of the new list to be one more than the version number of the previous list. During the reformation, a node can join only one list. The lists formed during the reformation started by a node become the *new token list* only if they satisfy the *majority test* and the *sequence test*.

The *majority test* requires that a valid list has a majority of the nodes. This test ensures that there is only one valid list at a given time. The *sequence test* requires that a site only joins a list with a higher version number than the list it previously belonged to. The originator always passes this test, since it chooses the version number to be one more than its previous version number. If any of the nodes fail this test, it tells

the originator its version number. The originator adopts the higher version number, and uses one more than this version number the next time that it tries to form a list. The combination of the majority test and the sequence test ensures that all valid lists have increasing version numbers. This happens because any two valid lists must have some nodes in common, which ensures that a new valid list always has a higher version number than the previous valid list.

In addition to these, the protocol has to ensure that none of the messages that was committed with the old list are lost. It should also make sure that the old list cannot be used by nodes that are unaware of the new list. This is done by the *resiliency test*. In the normal phase, for L resiliency, a message is committed only after the token has been passed L times after the message is acknowledged by the current token site. That is, $L+1$ sites have the message before it is committed. If the new list consists of any one of these sites for the message that was assigned the highest global sequence number in the old list, then the committed messages cannot be lost.

To ensure this, when a site i agrees to join a list, it tells the originator the next global sequence number that would have been assigned to a message with the old list, and the old list version number. The list with the largest known version number is considered to be the old list.

The reformation protocol of [CM84] is a three-phase protocol. In Phase I, the originator forms a new list. The procedure for forming a list is described above. The originator enters Phase II when all nodes have either responded or have been detected to have failed. To prevent the reformation process from being blocked due to the failure of the originator, a site leaves a list if no messages are received from the originator during some specified timeout period. A node can respond only if it belongs to the list (i.e., has not left it) and has recovered all the missing messages. A site that has missed some messages first requests the missed messages from the new token site.

In Phase II, the new list is formed. The new list consists of all the nodes that have responded. The majority and resiliency tests are applied to the new list. If the list is not valid, the originator aborts the reformation process. If the list is found to be valid, it is announced to all the nodes in the new list. A new token site is elected and the starting global sequence number determined. The new token site has all the messages up to the starting global sequence number of the new list.

In Phase III, the originator generates a new token and passes it to the new token site. The new token site accepts the token and starts acknowledging the message, and the reformation process is complete.

4.2.3 The Three-Phase Protocol

A straightforward way to achieve consistent ordering of messages at different nodes is to assign priorities to messages, and then deliver the messages in the order of priority, say the lowest priority message is delivered first. With different nodes assigning priorities, the problem in this approach is how a node should ascertain that no other message with lower priority will arrive later. This is particularly harder if the communication delays can vary.

The three-phase protocol solves this problem by all the nodes explicitly agreeing to a priority of a message and then only assigning higher priorities to later messages [BJ87]. This agreement protocol for assigning priorities works in three rounds of message exchange. That is why this protocol is called the three-phase protocol. The protocol presented here can work for both multicasting and broadcasting, though we will discuss only the broadcasting case.

Assume that a message is atomically broadcast by a node by using the $abcast(m, p)$ primitive, where m is the message and p is the priority (an integer) assigned to the message locally by the node broadcasting m. Each node maintains a queue, in which it keeps the messages it has received, till it delivers them. Each message is tagged *deliverable* or *undeliverable*. The protocol works as follows [BJ87]:

1. **(Phase I.)** The sender transmits the message (m, p) to all the nodes.

2. **(Phase II.)**Each receiver adds the message to its queue, and tags it as *undeliverable*. It then assigns this message a priority higher than the priority of any message that was placed in the buffer. It then informs the sender of the priority that has been assigned to the message.

3. **(Phase III.)** The sender collects the responses from all the nodes that have not failed. It then computes the maximum value of all the priorities it has received, and sends this value back to the receivers.

4. Each receiver changes the priority of the message to the priority received from the sender, and tags the message as *deliverable*. It sorts the queue in the order of the priority of the messages. Starting from the lowest priority message, it delivers all the messages which have been marked as *deliverable*, in order of priority. The delivery of messages stops at the first encounter of a message that is still not marked *deliverable*.

It is clear how the unique priority is being assigned and how a node determines that it will not receive a message with a lower priority. Consider a message m which

a node S wants to atomically broadcast. The final unique priority to m is assigned by S itself in Phase III. Since the initial priority to m was assigned by each receiver and it was higher than any priority the receiver was aware of, in Phase III, the priority that the S assigns to m will be the highest global priority in the system. Since these messages are sent to all the nodes, all nodes will later assign priorities that are higher than the final priority assigned to m. So, if the message m is marked as *deliverable*, any later message will have a higher priority and hence will be delivered after m.

However, an older message with a lower priority may be in the queue, whose final priority has not yet been determined, and whose final priority may finally turn out to be lower than the final priority of m. Since message delays are arbitrary, we cannot make any assumptions about the order in which different messages from different nodes arrive at a node, and this scenario is possible. To avoid this, a message is not delivered if there is any message of lower priority that has not yet been delivered. That is, in the last step of the protocol, the message that is first delivered is the one that has the lowest priority *and* is marked *deliverable*. When one such message is delivered and is removed from the queue, this process repeats itself till either all the messages have been delivered, or till we encounter a message which has the lowest priority but is not marked *deliverable*. Since only the messages that are marked *deliverable* are actually delivered, and the messages are marked *deliverable* only after they have been assigned their final priority, we can see that all nodes will deliver the messages in the same order, which is the order of the final priorities of the messages. This also implies that a message is delivered only after the three phases have been executed for it.

Let us consider an example of messages concurrently being broadcast. Suppose two messages m_1 and m_2 are broadcast by two different nodes. Let us consider two nodes 1 and 2. Suppose node 1 gets $m_1; m_2$, and sends the initial priorities p and p' back to the senders of m_1 and m_2 respectively (we know that $p' > p$). Suppose node 2 gets $m_2; m_1$ and sends q and q' to the senders of m_2 and m_1 respectively (with $q' > q$). The sender for m_1 computes the final priority $P_1 = max(p, q')$ and the sender for m_2 computes the final priority $P_2 = max(p', q)$. These priorities are communicated to nodes 1 and 2. Let $P_1 < P_2$. We then want both nodes 1 and 2 to deliver the messages in the order $m_1; m_2$. If $P_1 < p'$, then as soon as P_1 is received by node 1, its m_1 will be ready for delivery, since it will have the lowest priority and will be marked *deliverable*. Let us see what happens if node 2 gets P_2 before it gets P_1. When it gets P_2, it will set the priority of m_2 to P_2, and mark it *deliverable*. The other message it has is m_1 which is marked *undeliverable* and still has the priority q'. Since $P_1 > q'$, $P_2 > p'$, and by assumption $P_1 < p'$, we have $P_2 > q'$. That is, even though m_2 has been marked *deliverable*, it still does not have the lowest priority, and hence cannot be delivered. When P_1 is received, then both messages

can be delivered, but m_1 will be delivered first, as $P_1 < P_2$. Hence, the protocol ensures that even though the messages may be received in a different order, they are actually delivered in the same order at all nodes.

Let us now consider failures. In this protocol, the failure of a receiver node causes no problem. Since the sender only tries to communicate with operational nodes, and since a sender also detects the failure of a node, the failed node is simply ignored by the sender. However, if a sender node fails during the protocol, it can cause problems if at least one of the receivers has its message tagged as *undeliverable*. The presence of an *undeliverable* message implies that the protocol was not completed and some nodes may have the final priority, while others may not. In such a case, a node with the *undeliverable* message takes over the role of the failed sender, on detecting its failure, and becomes the *coordinator*. After becoming the coordinator, it first asks all the nodes the status of their messages. If any of the nodes has a message tagged *deliverable*, it sends the message and its priority. The presence of a *deliverable* message implies that the sender had computed the final priority and had communicated it to some of the nodes before failing. The new coordinator then uses this as the final priority and distributes it to other nodes. If this is not the case, the new coordinator first checks if any node has missed the initial message (sent in Phase I). If so, it sends its copy of the message to those nodes. After this, Phase II and Phase III are executed, as they would have been executed by the original sender. This approach requires that even after a message is delivered by a node, it cannot be actually removed until all nodes have received the message. Hence, the "garbage collection" of unneeded messages is done separately by a separate process.

4.2.4 Using Synchronized Clocks

Now we describe a protocol that employs synchronized clocks to guarantee ordering [CAS85]. Many protocols were presented in [CAS85] to satisfy the ordering property in the face of different types of faults. We will limit our discussion to fail stop failures only. It is assumed that node and link failures do not partition the network.

The protocol assumes that the network delay is bounded. A message m sent by a node P to another node Q takes, at most, δ time to arrive at the destination, if both P and Q remain operational. For a fixed upper bound on the number of node and link failures, the worst-case message delay from one operational node to another can easily be determined for a given network topology [CAS85]. Let this worst-case message delay be D. That is, even in the presence of a maximum number of nodes and link failures, a message sent by a nonfaulty node arrives at another nonfaulty node within D time. This worst-case delay D depends on the maximum transmission delay δ, and the maximum number of nodes and links that can fail.

The clocks of the different nodes are assumed to be synchronized, so that the clocks of different nodes are within β of each other. With this, we can say that if a message is sent by a node P at time t by its clock, the message will arrive at its destination by time $t + D + \beta$, by the clock of the destination node. The *termination time* of the protocol is $\Delta = D + \beta$. The goal of the protocol is to ensure (i) every message broadcast by a node at time t (by local clock) is delivered to every nonfaulty node at time $t + \Delta$ (by the receiver's clock), and (ii) all delivered messages are delivered in the same order to all nonfaulty nodes.

The protocol uses message diffusion to achieve its goals. When a node initiates the broadcast of a message, it timestamps the message with the value of its local clock, and also attaches its own unique node-id. A node sends the message along all outgoing links. When an intermediate node receives the message, it forwards it on all links other than the link on which the message arrived. To satisfy the ordering property, the messages are delivered by all nodes in the order of the timestamps of the messages. If the timestamps are the same, ties are broken based on node-ids of the sending nodes. Bounded message delays and synchronized clocks are used by a node to determine how long it should wait before it can be sure that no message with a smaller timestamp can arrive.

The protocol consists of three tasks. The *start* task initiates the broadcasting by sending a message on all of its outgoing links, the *relay* task forwards the message to adjacent nodes, and the *end* task delivers the message. The start task is given in Fig. 4.6.

obtain message m to be transmitted
attach local timestamp and node-id to m
send m to all neighbors
append m to history H
$schedule(End, t + \Delta, t)$

Figure 4.6: The start task

This task is performed by a node whenever it gets a message (from a process) to atomically broadcast. The node attaches the local clock and the node-id to the message and sends it to all the neighbors. The fact that m has been broadcast is recorded in a history variable H. This history H keeps track of ongoing broadcasts. In the end, the task schedules the *End* task to start at time $t + \Delta$, by which m will be received by all nodes. The current time at which the broadcast is initiated (i.e., t) is passed as a parameter to the *End* task when it is executed.

The relay task is shown in Fig. 4.7. This task is initiated by a node when it

receive (m) $\rightarrow$
  τ = current time
  if $\tau > m.t + \Delta$ **then exit** /* late message */
  if $m \in H$ **then exit** /* An old message */
  send m on all outgoing links except on which m came
  append m to history H
  $schedule(End, m.t + \Delta, m.t)$

Figure 4.7: The relay task

receives a message. If the current time is more than the time at which the message was sent plus Δ, the message is late and is discarded. Similarly, if the message is already in history H, it is discarded as a duplicate. Otherwise, m is forwarded to other neighbors and is appended to H. The *End* task of this node is scheduled at time $m.t + \Delta$. It should be noted that the *End* task is scheduled based on the time when the message was originated, not based on the time of arrival at the node.

Each message that is broadcast by some node at time t (by its local clock), must be received by all the nodes in the network by time $t + \Delta$ by their local clocks. As messages pass by the nodes, the *End* task is scheduled to execute at time $t + \Delta$, where t is the time at which the broadcast of the message was originated (as this time is recorded in the message, all nodes know about it). When the *End* task is executed, it delivers all the messages whose broadcast was initiated at time t by the local clock of the originator node. If there are multiple such messages, then the messages are delivered in the order of the node-ids of the originator nodes (the node-id of the originator is also appended in the message). All the messages it delivers are also deleted from the local history H.

It is clear that a message that is broadcast by any source node at time t_s is delivered by a node when its local clock is $t_s + \Delta$. It is also clear that all the nodes deliver the messages in the same order. This order is the same as the order of the timestamps of the messages, with those messages sent at the same local time by different nodes being delivered in the order of their node-ids.

4.2.5 A Protocol for CSMA/CD Networks

Now we describe a protocol for atomic broadcast for Carrier Sense Multiple Access with Collision Detection (CSMA/CD) networks that exploits the broadcast and

collision detection property of such networks [Jal92]. Consider a system which uses a common, shared medium for communication, and follows the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol [Tan88, Sta87]. Ethernet [MB76] is a prime example of such a network. In such networks, a node in the system usually has the Network Interface (NI) hardware that executes the Medium Access Control layer (MAC layer) protocol for the CSMA/CD network [Tan88]. The NI is usually connected to the common medium through a passive device called a transceiver, whose job is to pass the bits to the NI as they arrive. It also contains the collision detection hardware, and when a transceiver detects a collision, it sends a short jamming signal to enhance collision detection at other nodes.

A packet to be broadcast is given to the NI for transmission. The NI receives/sends a packet bit-by-bit from/to the transceiver and is responsible for all the activities of the MAC layer protocol, such as sensing the channel, transmitting the packet if the channel is free, aborting the transmission and later retransmitting a packet if a collision occurs, buffering of packets, etc. Above the NI are the communication software for higher-level communication protocols and user processes.

Even though the medium in such networks is inherently a broadcast medium, it does not guarantee that all the nodes will receive a broadcast message. Some of the nodes in the network may not receive a broadcast message. This can happen due to transient problems of buffer, network connection, etc. resulting in *missed messages*. Even though such networks are very reliable, this rare event where some nodes might miss some messages requires a special protocol for supporting reliable broadcasts. The protocol makes the following assumptions about the network:

1. The number of nodes that miss a broadcast message is less than half of the total nodes. This assumption is likely to be satisfied for most LANs.

2. The NI of a node can cause a collision at any time, even while receiving a message being transmitted by some other node, by sending a "jamming signal." Current network interfaces of CSMA/CD protocols do not provide this facility, since it is not needed, and the logic of NI has to be extended to support this. However, since the NI receives a packet bit-by-bit as it arrives on the cable, it can send a jamming signal, if needed, during the receiving of the packet. Further details about how to satisfy this assumption are given in [Jal92].

The NI of each node maintains a counter, initialized to 0. The counter at a node i, is referred to as C_i. When the NI transmits an atomic broadcast message, it attaches the *current value* of the counter as the sequence number of the message (for

a message m, the sequence number is represented by $m.seq$). If no collision occurs (i.e., the broadcast is successful), then the counter is incremented.

On receiving a message, if the sequence number of the message is greater than or equal to the counter value of the node, the counter is set to one more than the sequence number of the incoming message. If the sequence number is less than the counter value, the counter remains unchanged. Proper use of sequence numbers is central to the scheme. Only atomic broadcast messages have sequence numbers and affect the counters. Other messages, such as messages to particular destinations, or control messages, have no effect on the counters.

If there are no missed messages, all the counters will have the same value, and the messages will be sequentially numbered in the order in which they are actually transmitted. A *global sequence number* is defined as the number of broadcast messages that have been successfully transmitted. The counter of a node is said to be *correct* if its value is the same as the global sequence number. Due to missed messages, the counter value of some nodes may be less than the global sequence number. This property is used for detecting missed messages. At any given time, the set of alive nodes can be partitioned into two sets: a set of nodes whose counter value is the same as the global sequence number, and a set of nodes whose counter value is less than the global sequence number. No node can have a counter value greater than the global sequence number. Note that a node does not know by itself which set it belongs to. A node knows that it has missed some messages if it receives a message with a sequence number greater than its counter value. However, if a node receives a message whose sequence number is the same as its counter, it does not imply that the node has not missed any messages. If a node that has missed some messages transmits a message to others, other nodes which have also missed the same number of messages will have a counter value that is the same as the sequence number.

This creates the possibility that a node that has missed some messages may broadcast a message which can be received and delivered by another node. This will make the order of delivering the messages by this receiving node different from, say, a node that has not missed any messages. If we can ensure that a node with an incorrect counter is not allowed to successfully broadcast a message, then if the counter value at a node is the same as the sequence number at a node, it implies that the node has the correct counter value.

The problem is to stop a node with an incorrect counter from broadcasting. In the protocol, a simple approach is used. If, while receiving a message, the NI of a node discovers that the incoming message has a sequence number less than its counter value, *it immediately sends a jamming signal, thereby causing a collision.* This collision has the same effect as it does in a regular CSMA/CD protocol; the message is not successfully received by any node. This checking of the sequence

number and jamming has to occur while the message is being received and before the transmission ends. For this reason, it has to be done in the NI. The communication software above the NI will not be able to do this, as the NI passes the packet only after it has successfully received the entire packet. The effect of this feature on timing and minimum packet size is discussed in [Jal92].

The sending and receiving procedures at a node have to be modified to accomplish all this. The modified procedures for a node i are given in Fig. 4.8 [Jal92].

The sending procedure (lines 1–7) is similar to a regular procedure for sending a message in a CSMA/CD network. The only difference is the addition of the sequence number, and the management of the counter.

In the receiving procedure, different actions are taken depending on the relation of the sequence number of the incoming message to the counter value. If $m.seq = C_i$ and the previous message in the buffer is marked (i.e., has been delivered), then the message is immediately delivered and marked, the counter is incremented, and the messages are buffered (lines 12–18). If the previous message is not marked (i.e., has not yet been delivered), the message is simply buffered and is not delivered (lines 17–18). The jamming by a node of any incoming message m for which $m.seq < C_i$ (lines 10–11) ensures that when $m.seq = C_i$, then all the messages with a sequence number less than $m.seq$ have either been received or have been requested. If $m.seq > C_i$, a retransmit_request for the messages with the sequence number $C_i + 1$ through $m.seq - 1$ is broadcast, and the counter is set from the sequence number of the message (lines 19–23). Note that retransmit_request is not an atomic broadcast message, and may not be received by all the nodes. Also, it has no effect on counters.

If a retransmit_request is received by a node, it transmits any message in the requested sequence that it has in its buffer (lines 24–25). Sending of messages in response to a retransmit_request is considered as a reliable point-to-point, positive-acknowledgment-based scheme. Counters are not modified for these messages. Many nodes may respond to a retransmit_request, and the requesting node may receive many copies of the message from many different nodes. However, all these will have the same sequence number and the duplicates can easily be discarded. Once all the messages have been received, the node marks and delivers all the unmarked messages (lines 26–30).

The protocol uses the fact that a message is received by more than half of the nodes (the set of nodes that receive a message is called the *receiving set* of that message). Consequently, the intersection of the receiving sets of any two messages is non-empty. The protocol ensures that after any successful broadcast, more than half the alive nodes have counter values which are the same as the global sequence number, and any node with a counter value less than the global sequence number is

```
1. send(m) →
2.      m.seq = C_i
3.      Attempt broadcast
4.      if successful
5.              Increment C_i
6.              Buffer m
7.      else wait for random time and try again

8. receiving(m) →
9.      if m is a ordered broadcast message
10.             if m.seq < C_i
11.                     Jam the transmission
12.             if m.seq = C_i
13.                     Receive the entire message
14.                     if msg with seq C_i − 1 is marked
15.                             Mark m
16.                             Deliver m
17.                     Increment C_i
18.                     Buffer m
19.             if m.seq > C_i
20.                     Receive the entire message
21.                     Buffer m
22.                     Send retransmit_request [C_i + 1..m.seq − 1]
23.                     C_i = m.seq + 1
24.     if m is a retransmit_request
25.             Send requested messages from buffer to sender
26.     if m is a retransmitted_message
27.             Buffer m
28.             if all requested messages received
29.                     Deliver all unmarked messages in order
30.                     Mark all these messages
```

Figure 4.8: Modified send and receive procedures

prevented from successfully transmitting any message. This leads to the property that if a node receives two messages numbered s and s+1, then it has not missed any message between these two messages.

Due to the non-null intersection of the receiving sets of different messages, all the nodes that receive the retransmit_request together have all the messages a node has missed. Hence, every retransmit request is satisfied. There are other properties of the protocol which can be used to reduce the number of messages that need to be stored in the buffer by a node [Jal93]. Further details about implementing this protocol are also given in [Jal93].

4.3 Causal Broadcast

In atomic broadcast, the order in which messages are delivered is not important. Though this is sufficient in many applications, it is not strong enough in others. In some situations there are some restrictions on the order in which different requests sent by different nodes should be delivered to the resource. For example, consider the situation of a distributed database system in which a node i broadcasts a request, and then sends a message to another node j. After receiving this message, node j sends its own request for some operation on the database. It is conceivable that the request made by j uses the information given to it by i, and may depend on the fact that i has already performed some operation on the database. In such a situation, it is desirable that the request made by j be performed on the copies of the database after the request by node i has been performed. With atomic broadcast, this restriction cannot be enforced. For this, we require *causal broadcast* in which the causal ordering of messages is preserved.

As we have seen in Chapter 2, a "happened before" relation (represented by $\rightarrow$) can be defined by the events in a distributed system. This relation defines the potential causality between events in a distributed system. That is, $e_1 \rightarrow e_2$ means that the event e_1 can causally affect the event e_2. This relation specifies a partial ordering of the set of events in the system. We will also refer to the ordering defined by $\rightarrow$ as *causal ordering*. Causal broadcast requires that the order in which the messages are delivered by a node is consistent with the partial ordering defined by $\rightarrow$ on the events corresponding to the sending of the messages. Let the event corresponding to the sending of a message m be represented by $s(m)$. Causal broadcast requires that if $s(m_1) \rightarrow s(m_2)$, then m_1 is delivered before m_2 by all nodes. In the rest of this section, $m_1 \rightarrow m_2$ means that $s(m_1) \rightarrow s(m_2)$.

The causal ordering of messages can be preserved by two different approaches, leading to two different sets of requirements for causal broadcast. The weaker

requirement only states that the causality should be preserved. The stricter requirement states that the consistency with the causal relationship is over and above the total ordering requirement. That is, the messages should be delivered in the same order at all the sites, but the order should be such that it preserves the causal relationship between messages. We will discuss two different protocols for causal broadcast, one for the weaker requirement, and one for the stricter requirement.

As in atomic broadcast, we distinguish between a node *receiving* a message and *delivering* a message. A message is received by a node when it arrives over the network and the node has no control over the order in which messages are received. A received message is delivered (to the processes executing on the node) whenever the node wishes to do that. Hence, the order of delivery is entirely in the control of a node. Causal broadcast protocols have to ensure that the order in which the messages are delivered is consistent with the causal ordering.

4.3.1 Causal Broadcast without Total Ordering

We first discuss a protocol that delivers messages to nodes in the system in an order that preserves the causal ordering. However, it does not guarantee the same total ordering of all messages at all nodes. That is, if two messages m and m' are such that $m \rightarrow m'$, then m is delivered before m' at all nodes. However, if there is no causal relation between m and m', that is, $m \not\rightarrow m'$ and $m' \not\rightarrow m$, then m and m' can be delivered in any order, and that order may not be the same at different nodes.

The protocol described here is distributed, and can support causal multicasting also [BJ87]. For causally broadcasting a message, a process executes the broadcast primitive $CBCAST(m, l, dests)$, where m is the message, l is a label, and $dests$ is the set of destination nodes to which the message is to be sent (where not needed, we will ignore the last two parameters). The label is a timestamp based on logical clocks (discussed earlier in Chapter 2) which preserve causal ordering. Hence, two labels l and l' are comparable if there is a causal relationship between the sending of the corresponding messages m and m', and $l < l'$ if $CBCAST(m) \rightarrow CBCAST(m')$. If there is no causal relationship between $CBCAST(m)$ and $CBCAST(m')$, then the labels l and l' are not comparable. We say that a message m *precedes* another message m' if $CBCAST(m) \rightarrow CBCAST(m')$.

For supporting *CBCAST*, a node P contains a buffer BUF_P which contains all messages sent to and from P, as well as copies of messages that arrive at P *en route* to other destinations. From the buffer, messages are put on a *delivery queue* from which the application processes can receive the messages. The protocol for *CBCAST* ensures that the messages are added to the delivery queue in an order which is consistent with the causal order of messages.

When a node P performs a $CBCAST(m, l, dests)$, the message m (along with other related information) is added to BUF_P. If P is one of the destination nodes, then m is added to the delivery queue of P. Messages in BUF_P are later scheduled for transmission so that they can reach other destinations. It is assumed that each message in the buffer is transmitted in a finite amount of time.

When a message m is to be transmitted from P to Q, only m is not sent. P sends all messages in BUF_P that precede m. In other words, when P wants to send m to Q, it sends a *transfer packet* consisting of a sequence of messages $m_1, m_2, ...$ This transfer packet includes all messages in BUF_P that precede m. The messages are kept in a sorted order in the transfer packet so that if $m_i \rightarrow m_j$, then m_i comes before m_j in the transfer packet (i.e., $i < j$).

When a node Q receives a packet $m_1, m_2, ...$, it processes each message m_i in the transfer packet in the order in which it appears in the packet. For an m_i, Q first checks if this message is a duplicate (this is done easily by use of sequence numbers). If it is a duplicate, the message is discarded. If Q is one of the destination nodes of the message m_i, then m_i is put on the delivery queue of Q. Otherwise m_i is simply added to BUF_Q.

Let us see how this supports causal broadcasting. Suppose a message m was sent by a node P and a message m' was sent by a node Q (nodes Q and P also could be the same). If m precedes m', it implies that there is a sequence of broadcasts $m_0, m_1, ..., m_n$, such that $m = m_0$ and $m' = m_n$, and for all $i, 0 < i \leq n$, m_{i-1} is received by the sender of m_i before m_i is sent. Since in each broadcast, besides the message, all messages in the buffer that precede the message are also sent to the destination, it is clear that when m_n is sent, the transmission packet will contain all messages $m_0, m_1, ..., m_{n-1}$. Each node that receives this packet will deliver m before delivering m', since a node processes the messages in the packet in the order they appear. Hence, the causal broadcast property is satisfied. Failures do not affect this as long as the network does not get partitioned.

4.3.2 Causal Broadcast with Total Ordering

Now we describe a protocol that delivers messages to the different nodes in the same order, such that the order preserves the causality between messages [Jal93]. The protocol is based on the primary site approach. One of the nodes in the system is designated as primary, and k other nodes are designated as backups for a k-resilient system. If the primary fails, a backup takes over the role of the primary site. If a node wants to broadcast a message, it sends it to the designated primary site (PS), which then broadcasts it to other nodes. This approach can easily satisfy the atomic broadcast properties. However, supporting causal broadcast is not straightforward,

since messages sent by two different nodes, regardless of any causal dependency between them, can arrive at the PS in any order. For causal broadcasting, the PS has to order the incoming messages in a manner consistent with the causal ordering between them. This approach uses a method based on sequence numbers and counters for capturing and disseminating information about dependency between nodes.

Each node i keeps a counter C_i which represents the number of messages that this node has sent to PS for broadcast (initialized to 0). The counter is used to assign sequence numbers to messages whose broadcast is requested by node i. A node i also maintains an array $seq[]$. This array keeps a sequence number for each node, such that $seq[j]$ is the sequence number of the last message sent by the node j to PS for broadcasting, as known by the node i. Clearly, for a node i, $seq[i]$ represents the sequence number assigned by the node i (i.e., the value of C_i) to the last message it sent to PS for broadcasting.

Whenever a node j sends a message to another node i, it sends a copy of the array $seq[]$ along with the message. On receiving a message, a node sets its $seq[]$ such that each element is the maximum of its local value and the value received in the message. That is, if a node receives a new message m it sets:

$$\forall j : seq[j] = max(seq[j], m.seq[j]).$$

At a node i, at a given time, the array seq[] represents the sequence number of the last message from each node on which events of this node causally depend. In other words, any send command executed by the node i causally depends only on those broadcast messages of node j which were assigned by j a sequence number less than or equal to seq[j] at the node i.

The sequence number assigned by the PS to a broadcast message is represented as $gseq$, and is different from the sequence numbers assigned by the nodes using their local counters. The sequence number $gseq$ is used for globally ordering the delivery of messages. Each node maintains a variable *last-msg*, initialized to zero, which is the sequence number of the last broadcast message received by this node from the PS. This is used to identify and discard duplicate messages.

The major logical participants in the broadcast protocol are a simple node which is a sender and receiver of messages, the primary site, and a backup site (these are logical sites; a node may act as a simple node as well as the PS). Here we specify the actions that need to be performed by these different parties to support atomic broadcast. The actions performed by a simple node are shown in Fig. 4.9. When the node i has a message to broadcast, it assigns to seq[i] the value of the local counter C_i, adds seq[] to the message, and sends it to the PS. The message is also saved in the buffer, in case it is needed for retransmission later on.

Node i:

msg-to-send(msg) $\rightarrow$
  $seq[i] = C_i$
  $C_i = C_i + 1$
  msg.seq[] = seq[]
  send(PS, msg)
  save msg in buffer

□ request-to-resend(n) $\rightarrow$
  PS = sender
  send(PS, messages numbered n .. C_i - 1)

□ receive(msg) $\rightarrow$
  PS = msg.src
  if (msg.gseq $\leq$ last-msg) then ignore the message
  else if (msg.original-src = i) delete msg from buffer
    last-msg = msg.gseq
    deliver(msg)

Figure 4.9: Actions of a sender/receiver node

If the primary site fails and a backup takes over as the primary, the backup may have missed some messages sent by the node to the primary site. In that case, when the backup assumes the role of the primary and detects that it has missed some messages, it sends a request for resending the missed messages. If that request comes, the sender retransmits the missed messages to the new primary site (second guard).

Finally, if the node receives a broadcast message (the node is also one of the recipients), and if the sequence number assigned to the message by the PS ($gseq$) is less than or equal to the sequence number of the last message received by the node from the PS, it implies that this message is a duplicate, and is discarded by the node. The primary site is identified as the source of the coming message (represented by $msg.src$). If the original source of the message is this node, then the copy of the message is deleted from the buffer, since it will not be needed for retransmission. In Fig. 4.9, msg.original-src represents the originator of the message.

Now let us consider the actions that need to be performed by the primary site.

Primary Site:

receive(msg) $\rightarrow$
    if expected[msg.src] > msg.seq[msg.src] $\rightarrow$
        discard the message as duplicate
    else save *msg* in buffer

$\Box\exists msg$ in buffer such that ($\forall j \neq msg.src$: expected[j] > msg.seq[j])
        and (expected[msg.src] = msg.seq[msg.src]) $\rightarrow$
    add *msg* to Q_B
    increment expected[msg.src]

$\Box Q_B$ not empty $\rightarrow$
    take *msg* at the head of Q_B
    msg.gseq = Ctr
    to backups in sequence, send (msg and expected[])
    to other receivers, send(msg)
    Ctr = Ctr + 1

Figure 4.10: Actions performed by the primary site

The primary site keeps an array called *expected[]*, where *expected[i]* is the expected sequence number of the next message to be received from the node i. In other words, the PS has already received messages numbered 0 .. i (expected[i] - 1) from the node i for broadcasting. In addition, the PS maintains a counter Ctr, which is used to assign sequence numbers ($gseq$) to the broadcast messages. The primary site maintains a queue Q_B, in which the messages to be broadcast are kept in order. It takes the messages from Q_B and broadcasts them to other sites. It also has a buffer in which messages are kept till they can be added to Q_B. The actions of the PS are shown in Fig. 4.10.

When the PS receives a message from a node for broadcasting, it compares its sequence number from what it expects. If the sequence number is less than expected, it means that this message is a duplicate (which may arise because of requests to resend messages), and is discarded, otherwise the message is put in a buffer. If at any time there exists some message *msg* in the buffer for which msg.seq[msg.src] is the same as expected[msg.src] (implying that all previous messages from this node have been received by PS), and msg.seq[j] is less than expected[j] for all other nodes j (implying that messages from all other nodes that were "sent before" this

Primary Site:

```
receive(msg) →
        if expected[msg.src] > msg.seq[msg.src] →
                discard the message as duplicate
        else save msg in buffer

□∃msg in buffer such that (∀j ≠ msg.src: expected[j] > msg.seq[j])
                    and (expected[msg.src] = msg.seq[msg.src]) →
        add msg to Q_B
        increment expected[msg.src]

□Q_B not empty →
        take msg at the head of Q_B
        msg.gseq = Ctr
        to backups in sequence, send (msg & expected[])
        to other receivers, send(msg)
        Ctr = Ctr + 1
```

Figure 4.11: Actions performed by the primary site

message have been received by PS), then *msg* is added to Q_B for broadcasting, and expected[msg.src] is incremented. Finally, if Q_B is not empty, PS sends the message expected[], first to the backups in order, and then to the other nodes.

Now let us consider the activities which a backup node should do. If the primary site and all earlier backups fail, then this node has to become the primary site. So, first it sets the value of its counter Ctr from the *gseq* of the last message received from the previous PS. The messages that were there in the buffer of the primary site at the time of failure are lost and have to be recovered by the backup. For this, the backup requests every node j to resend messages with a sequence number more than or equal to expected[j]. Note that expected[] at the backup is as obtained from the last message by PS to this backup. This ensures that any message from j with a sequence number less than expected[j] must have been broadcast by PS earlier. Since the backup does not know whether the PS was able to complete broadcasting of the last message it received (PS must have completed broadcasting of earlier messages), the backup also broadcasts the last message it received from the PS. This may result in duplicate messages, but they will be discarded by the receivers by the use of *gseq*, as discussed earlier. It should be pointed out that any failure during the recovery of

a backup poses no problems, as long as the assumption about one alive node with the message holds. Failure during recovery simply means that the next backup will start its own recovery and take over as the primary.

In this protocol, the PS broadcasts messages one by one in the order in which they appear in Q_B. The Ctr value assigned by the PS is incremented at each broadcast. If PS fails, the backup completes the last broadcast in which PS was involved, and starts assigning Ctr values to messages after that. Since the message service ensures ordering between a sender and receiver, and all nodes discard duplicates based on $gseq$, all nodes will receive and deliver the messages in the order defined by $gseq$.

Hence, it is clear that the protocol does deliver the messages in the same order to all nodes. Now let us see how the delivery order of messages is consistent with the causal ordering. Suppose m_i is the message sent by node i and m_j is the message sent by node j. If $m_i \rightarrow m_j$, it means that either the event for sending m_i, or some later send event in i, and some receive event in j before sending m_j are related by a sequence of send-receives. On receiving a message m from another node, since a node j sets $seq[k] = max(seq[k], m.seq[k])$, for all k, it implies that $m_i.seq[i] \leq m_j.seq[i]$). As m_j is added to Q_B only after all messages m from i with $m.seq[i]$ less than or equal to $m_j.seq[i]$ have been added, clearly m_i is added before m_j in Q_B. Since the messages are delivered by the nodes in the order in which the PS broadcasts them, and the PS broadcasts messages in the order they appear in Q_B, the causality condition is satisfied.

4.4 Summary

Just like reliable point-to-point communication is a building block for building fault tolerant systems that require point-to-point communication, reliable broadcast is a building block for those fault tolerant applications that require one process to send a message to multiple destinations. The topic of this chapter is the abstraction of reliably broadcasting a message in a distributed system.

For broadcasting, the three properties of interest are *reliability, ordering*, and *causality*. Reliability requires that a broadcast message reach all the nodes, ordering requires that different messages sent by different nodes be delivered to all the nodes in the same order, and causality requires that the messages be delivered in an order that is consistent with the causality between them. For satisfying these three properties, three different broadcast primitives are needed: *reliable broadcast, atomic broadcast*, and *causal broadcast*. These primitives and the means of supporting them in a system that only supports reliable point-to-point communication, or unreliable broadcast communication, are the topic of this chapter. The protocols for these primitives