CHAPTER

# 6 Message ordering and group communication

Inter-process communication via message-passing is at the core of any distributed system. In this chapter, we will study non-FIFO, FIFO, causal order, and synchronous order communication paradigms for ordering messages. We will then examine protocols that provide these message orders. We will also examine several semantics for group communication with multicast – in particular, causal ordering and total ordering. We will then look at how exact semantics can be specified for the expected behavior in the face of processor or link failures. Multicasts are required at the application layer when superimposed topologies or overlays are used, as well as at the lower layers of the protocol stack. We will examine some popular multicast algorithms at the network layer. An example of such an algorithm is the Steiner tree algorithm, which is useful for setting up multi-party teleconferencing and videoconferencing multicast sessions.

## Notation

As before, we model the distributed system as a graph $(N, L)$. The following notation is used to refer to messages and events:

- When referring to a message without regard for the identity of the sender and receiver processes, we use $m^i$. For message $m^i$, its send and receive events are denoted as $s^i$ and $r^i$, respectively.
- More generally, send and receive events are denoted simply as $s$ and $r$. When the relationship between the message and its send and receive events is to be stressed, we also use $M$, $send(M)$, and $receive(M)$, respectively.

For any two events $a$ and $b$, where each can be either a send event or a receive event, the notation $a \sim b$ denotes that $a$ and $b$ occur at the same process, i.e., $a \in E_i$ and $b \in E_i$ for some process $i$. The send and receive event pair for a message is said to be a pair of *corresponding* events. The send event corresponds to the receive event, and vice-versa. For a given execution $E$, let the set of all send–receive event pairs be denoted as $\mathcal{T} = \{(s, r) \in E_i \times E_j \mid s$ corresponds

https://doi.org/10.1017/CBO9780511805318.007 Published online by Cambridge University Press

to $r\}$. When dealing with message ordering definitions, we will consider only send and receive events, but not internal events, because only communication events are relevant.

## 6.1 Message ordering paradigms

The order of delivery of messages in a distributed system is an important aspect of system executions because it determines the messaging behavior that can be expected by the distributed program. Distributed program logic greatly depends on this order of delivery. To simplify the task of the programmer, programming languages in conjunction with the middleware provide certain well-defined message delivery behavior. The programmer can then code the program logic with respect to this behavior.

Several orderings on messages have been defined: (i) non-FIFO, (ii) FIFO, (iii) causal order, and (iv) synchronous order. There is a natural hierarchy among these orderings. This hierarchy represents a trade-off between concurrency and ease of use and implementation. After studying the definitions of and the hierarchy among the ordering models, we will study some implementations of these orderings in the middleware layer. This section is based on Charron-Bost *et al.* [7].

### 6.1.1 Asynchronous executions

**Definition 6.1** (*A*-**execution**) An asynchronous execution (or *A*-execution) is an execution $(E, \prec)$ for which the causality relation is a partial order.

There cannot exist any causality cycles in any real asynchronous execution because cycles lead to the absurdity that an event causes itself. On any logical link between two nodes in the system, messages may be delivered in any order, *not necessarily* first-in first-out. Such executions are also known as *non-FIFO executions*. Although each physical link typically delivers the messages sent on it in FIFO order due to the physical properties of the medium, a logical link may be formed as a composite of physical links and multiple paths may exist between the two end points of the logical link. As an example, the mode of ordering at the Network Layer in connectionless networks such as IPv4 is non-FIFO. Figure 6.1(a) illustrates an *A*-execution under non-FIFO ordering.

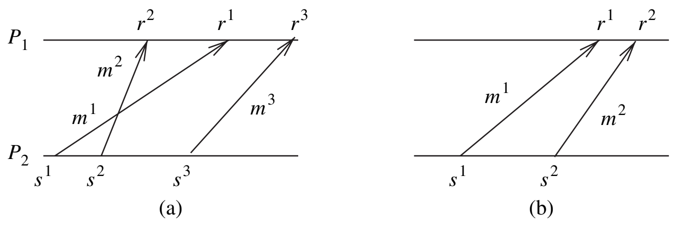


**Figure 6.1** Illustrating FIFO and non-FIFO executions. (a) An *A*-execution that is not a FIFO execution. (b) An *A*-execution that is also a FIFO execution.

### 6.1.2 FIFO executions

**Definition 6.2 (FIFO executions)** A FIFO execution is an $A$-execution in which,
for all $(s, r)$ and $(s', r') \in \mathcal{T}$, $(s \sim s'$ and $r \sim r'$ and $s \prec s') \Longrightarrow r \prec r'$.

On any logical link in the system, messages are necessarily delivered in the order in which they are sent. Although the logical link is inherently non-FIFO, most network protocols provide a connection-oriented service at the transport layer. Therefore, FIFO logical channels can be realistically assumed when designing distributed algorithms. A simple algorithm to implement a FIFO logical channel over a non-FIFO channel would use a separate numbering scheme to sequence the messages on each logical channel. The sender assigns and appends a $\langle sequence\_num, connection\_id \rangle$ tuple to each message. The receiver uses a buffer to order the incoming messages as per the sender's sequence numbers, and accepts only the "next" message in sequence. Figure 6.1(b) illustrates an $A$-execution under FIFO ordering.

### 6.1.3 Causally ordered (CO) executions

**Definition 6.3 (Causal order (CO))** A CO execution is an $A$-execution in which,
for all $(s, r)$ and $(s', r') \in \mathcal{T}$, $(r \sim r'$ and $s \prec s') \Longrightarrow r \prec r'$.

If two send events $s$ and $s'$ are related by causality ordering (not physical time ordering), then a causally ordered execution requires that their corresponding receive events $r$ and $r'$ occur in the same order at all common destinations. Note that if $s$ and $s'$ are not related by causality, then CO is vacuously satisfied because the antecedent of the implication is false.

**Examples**

- **Figure 6.2(a)** shows an execution that violates CO because $s^1 \prec s^3$ and at the common destination $P_1$, we have $r^3 \prec r^1$.
- **Figure 6.2(b)** shows an execution that satisfies CO. Only $s^1$ and $s^2$ are related by causality but the destinations of the corresponding messages are different.

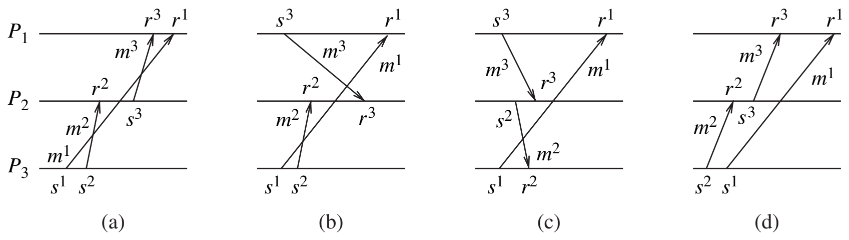


**Figure 6.2** Illustration of causally ordered executions. (a) Not a CO execution. (b), (c), and (d) CO executions.

- **Figure 6.2(c)** shows an execution that satisfies CO. No send events are related by causality.
- **Figure 6.2(d)** shows an execution that satisfies CO. $s^2$ and $s^1$ are related by causality but the destinations of the corresponding messages are different. Similarly for $s^2$ and $s^3$.

Causal order is useful for applications requiring updates to shared data, implementing distributed shared memory, and fair resource allocation such as granting of requests for distributed mutual exclusion. Some of these uses will be discussed in detail in Section 6.5 on ordering message broadcasts and multicasts.

To implement CO, we distinguish between the arrival of a message and its delivery. A message $m$ that arrives in the local OS buffer at $P_i$ may have to be delayed until the messages that were sent to $P_i$ causally before $m$ was sent (the "overtaken" messages) have arrived and are processed by the application. The delayed message $m$ is then given to the application for processing. The event of an application processing an arrived message is referred to as a *delivery* event (instead of as a *receive* event) for emphasis.

**Example** Figure 6.2(a) shows an execution that violates CO. To enforce CO, message $m^3$ should be kept pending in the local buffer after it arrives at $P_1$, until $m^1$ arrives and $m^1$ is delivered.

**Definition 6.4 (Definition of causal order (CO) for implementations)** If $send(m^1) \prec send(m^2)$ then for each common destination $d$ of messages $m^1$ and $m^2$, $deliver_d(m^1) \prec deliver_d(m^2)$ must be satisfied.

Observe that if the definition of causal order is restricted so that $m^1$ and $m^2$ are sent by the same process, then the property degenerates into the FIFO property. In a FIFO execution, no message can be overtaken by another message between the same (sender, receiver) pair of processes. The FIFO property which applies on a per-logical channel basis can be extended globally to give the CO property. In a CO execution, no message can be overtaken by a chain of messages between the same (sender, receiver) pair of processes.

**Example** Figure 6.2(a) shows an execution that violates CO. Message $m^1$ is overtaken by the messages in the chain $\langle m^2, m^3 \rangle$.

CO executions can also be alternatively characterized by Definition 6.5 by simultaneously dropping the requirement from the implicand of Definition 6.3 that the receive events be on the same process, and relaxing the consequence from $(r \prec r')$ to $\neg(r' \prec r)$, i.e., the message $m'$ sent causally later than $m$ is not received causally earlier at the common destination. This ordering is known as message ordering (MO).

**Definition 6.5 (Message order (MO))** A MO execution is an $A$-execution in which,
for all $(s, r)$ and $(s', r') \in \mathcal{T}$, $s \prec s' \Longrightarrow \neg(r' \prec r)$.

**Example** Consider any message pair, say $m^1$ and $m^3$ in Figure 6.2(a). $s^1 \prec s^3$ but $\neg(r^3 \prec r^1)$ is false. Hence, the execution does not satisfy MO.

You are asked to prove the equivalence of MO executions and CO executions in Exercise 6.1. This will show that in a CO execution, a message cannot be overtaken by a chain of messages.

Another characterization of a CO execution in terms of the partial order $(E, \prec)$ is known as the empty-interval (EI) property.

**Definition 6.6 (Empty-interval execution)** An execution $(E, \prec)$ is an empty-interval (EI) execution if for each pair of events $(s, r) \in \mathcal{T}$, the open interval set $\{x \in E \mid s \prec x \prec r\}$ in the partial order is empty.

**Example** Consider any message, say $m^2$, in Figure 6.2(b). There does not exist any event $x$ such that $s^2 \prec x \prec r^2$. This holds for all messages in the execution. Hence, the execution is EI.

You are asked to prove the equivalence of EI executions and CO executions in Exercise 6.1. A consequence of the EI property is that for an empty interval $\langle s, r\rangle$, there exists some *linear* extension[1] $<$ such that the corresponding interval $\{x \in E \mid s < x < r\}$ is also empty. An empty $\langle s, r\rangle$ interval in a linear extension indicates that the two events may be arbitrarily close and can be represented by a vertical arrow in a timing diagram, which is a characteristic of a synchronous message exchange. Thus, an execution $E$ is CO if and only if for each message, there exists *some* space–time diagram in which that message can be drawn as a vertical message arrow. This, however, does not imply that *all* messages can be drawn as vertical arrows in the *same* space–time diagram. If all messages could be drawn vertically in an execution, all the $\langle s, r\rangle$ intervals would be empty in the *same* linear extension and the execution would be synchronous.

Another characterization of CO executions is in terms of the causal past/future of a send event and its corresponding receive event. The following corollary can be derived from the EI characterization above (Definition 6.6).

**Corollary 6.1** *An execution $(E, \prec)$ is CO if and only if for each pair of events $(s, r) \in \mathcal{T}$ and each event $e \in E$,*

- *weak common past*: $e \prec r \Longrightarrow \neg(s \prec e)$;
- *weak common future*: $s \prec e \Longrightarrow \neg(e \prec r)$.

**Example** Corollary 6.1 can be observed for the executions in Figures 6.2(b)–(d).

[1] A linear extension of a partial order $(E, \prec)$ is any total order $(E, <)$ such that each ordering relation of the partial order is preserved.

If we require that the past of both the $s$ and $r$ events are identical (and analogously for the future), viz., $e \prec r \Longrightarrow e \prec s$ and $s \prec e \Longrightarrow r \prec e$, we get a subclass of CO executions, called *synchronous executions*.

### 6.1.4 Synchronous execution (SYNC)

When all the communication between pairs of processes uses synchronous send and receive primitives, the resulting order is the synchronous order. As each synchronous communication involves a handshake between the receiver and the sender, the corresponding send and receive events can be viewed as occuring instantaneously and atomically. In a timing diagram, the "instantaneous" message communication can be shown by bidirectional vertical message lines. Figure 6.3(a) shows a synchronous execution on an asynchronous system. Figure 6.3(b) shows the equivalent timing diagram with the corresponding instantaneous message communication.

The "instantaneous communication" property of synchronous executions requires a modified definition of the causality relation because for each $(s, r) \in \mathcal{T}$, the send event is not causally ordered before the receive event. The two events are viewed as being atomic and simultaneous, and neither event precedes the other.

**Definition 6.7 (Causality in a synchronous execution)** The synchronous causality relation $\ll$ on $E$ is the smallest transitive relation that satisfies the following:

S1: If $x$ occurs before $y$ at the same process, then $x \ll y$.
S2: If $(s, r) \in \mathcal{T}$, then for all $x \in E$, $[(x \ll s \Longleftrightarrow x \ll r)$ and $(s \ll x \Longleftrightarrow r \ll x)]$.
S3: If $x \ll y$ and $y \ll z$, then $x \ll z$.

We can now formally define a synchronous execution.

**Definition 6.8 (Synchronous execution)** A synchronous execution (or $S$-execution) is an execution $(E, \ll)$ for which the causality relation $\ll$ is a partial order.

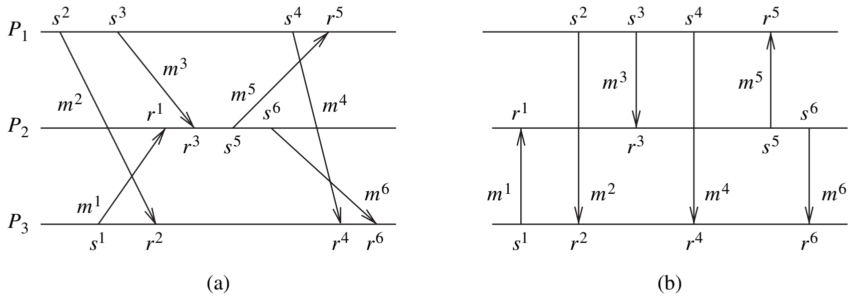


**Figure 6.3** Illustration of a synchronous communication. (a) Execution in an asynchronous system. (b) Equivalent instantaneous communication.

We now show how to timestamp events in synchronous executions.

**Definition 6.9 (Timestamping a synchronous execution)** An execution ($E$, $\prec$) is synchronous if and only if there exists a mapping from $E$ to $T$ (scalar timestamps) such that

- for any message $M$, $T(s(M)) = T(r(M))$;
- for each process $P_i$, if $e_i \prec e'_i$ then $T(e_i) < T(e'_i)$.

By assuming that a send event and its corresponding receive event are viewed atomically, i.e., $s(M) \prec r(M)$ and $r(M) \prec s(M)$, it follows that for any events $e_i$ and $e_j$ that are not the send event and the receive event of the same message, $e_i \prec e_j \Longrightarrow T(e_i) < T(e_j)$.

## 6.2 Asynchronous execution with synchronous communication

When all the communication between pairs of processes is by using synchronous send and receive primitives, the resulting order is synchronous order. The send and receive events of a message appear instantaneous, see the example in Figure 6.3. We now address the following question:

- If a program is written for an asynchronous system, say a FIFO system, will it still execute correctly if the communication is done by synchronous primitives instead? There is a possibility that the program may *deadlock*, as shown by the code in Figure 6.4.

Charron-Bost *et al.* [7] observed that a distributed algorithm designed to run correctly on asynchronous systems (called *A-executions*) may not run correctly on synchronous systems. An algorithm that runs on an asynchronous system may *deadlock* on a synchronous system.

**Examples** The asynchronous execution of Figure 6.4, illustrated in Figure 6.5(a) using a timing diagram, will deadlock if run with synchronous primitives. The executions in Figure 6.5(b)–(c) will also deadlock when run on a synchronous system.

**Figure 6.4** A communication program for an asynchronous system deadlocks when using synchronous primitives.

```
Process i                   Process j
...                         ...
Send(j)                     Send(i)
Receive(j)                  Receive(i)
...                         ...
```

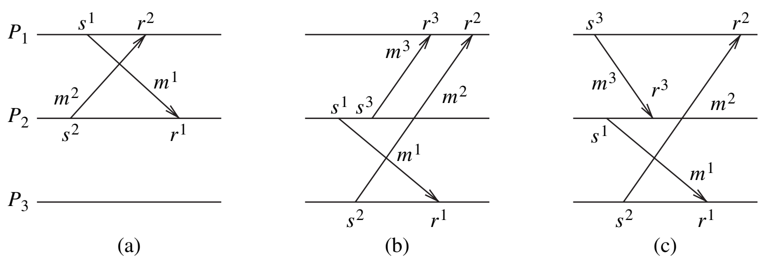


**Figure 6.5** Illustrations of asynchronous executions and of crowns. (a) Crown of size 2. (b) Another crown of size 2. (c) Crown of size 3.

### 6.2.1 Executions realizable with synchronous communication (RSC)

An execution can be modeled (using the interleaving model) as a feasible schedule of the events to give a total order that extends the partial order $(E, \prec)$. In an A-execution, the messages can be made to appear instantaneous if there exists a linear extension of the execution, such that each send event is immediately followed by its corresponding receive event in this linear extension. Such an A-execution can be realized under synchronous communication and is called a *realizable with synchronous communication* (RSC) execution.

**Definition 6.10 (Non-separated linear extension)** A non-separated linear extension of $(E, \prec)$ is a linear extension of $(E, \prec)$ such that for each pair $(s, r) \in \mathcal{T}$, the interval $\{ x \in E \mid s \prec x \prec r \}$ is empty.

**Examples**

- Figure 6.2(d): $\langle s^2, r^2, s^3, r^3, s^1, r^1 \rangle$ is a linear extension that is non-separated. $\langle s^2, s^1, r^2, s^3, r^3, s^1 \rangle$ is a linear extension that is separated.
- Figure 6.3(b): $\langle s^1, r^1, s^2, r^2, s^3, r^3, s^4, r^4, s^5, r^5, s^6, r^6 \rangle$ is a linear extension that is non-separated. $\langle s^1, s^2, r^1, r^2, s^3, s^4, r^4, r^3, s^5, s^6, r^6, r^5 \rangle$ is a linear extension that is separated.

**Definition 6.11 (RSC execution)** [7] An A-execution $(E, \prec)$ is an RSC execution if and only if there exists a non-separated linear extension of the partial order $(E, \prec)$.

In the non-separated linear extension, if the adjacent send event and its corresponding receive event are viewed atomically, then that pair of events shares a common past and a common future with each other. The various other characterizations of S-executions seen in Section 6.1.4 are also seen to hold.

To use Definition 6.11 requires checking for all the linear extensions, incurs exponential overhead. You can verify this by trying to create and examine all the linear extensions of the execution in Figure 6.5(b) or (c). Thus, Definition 6.11 does not provide a practical test to determine whether a program written for a non-synchronous system, say a FIFO system, will still execute correctly if the communication is done by synchronous primitives.

We now study a characterization of the execution in terms of a graph structure called a *crown*; the crown leads to a feasible test for a RSC execution.

**Definition 6.12 (Crown)** Let $E$ be an execution. A crown of size $k$ in $E$ is a sequence $\langle (s^i, r^i), i \in \{0, \ldots, k-1\} \rangle$ of pairs of corresponding send and receive events such that: $s^0 \prec r^1$, $s^1 \prec r^2$, $\ldots$, $s^{k-2} \prec r^{k-1}$, $s^{k-1} \prec r^0$.

**Examples**

- Figure 6.5(a): The crown is $\langle (s^1, r^1), (s^2, r^2) \rangle$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$. This execution represents the program execution in Figure 6.4.
- Figure 6.5(b): The crown is $\langle (s^1, r^1), (s^2, r^2) \rangle$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$.
- Figure 6.5(c): The crown is $\langle (s^1, r^1), (s^3, r^3), (s^2, r^2) \rangle$ as we have $s^1 \prec r^3$ and $s^3 \prec r^2$ and $s^2 \prec r^1$.
- Figure 6.2(a): The crown is $\langle (s^1, r^1), (s^2, r^2), (s^3, r^3) \rangle$ as we have $s^1 \prec r^2$ and $s^2 \prec r^3$ and $s^3 \prec r^1$.

In a crown, the send event $s^i$ and receive event $r^{i+1}$ may lie on the same process (e.g., Figure 6.5(c)) or may lie on different processes (e.g., Figure 6.5(a)). We can also make the following observations:

- In an execution that is not CO (see the example in Figure 6.2(a)), there must exist pairs $(s, r)$ and $(s', r')$ such that $s \prec r'$ and $s' \prec r$. It is possible to generalize this to state that a non-CO execution must have a crown of size at least 2. (Exercise 6.4 asks you to prove that in a non-CO execution, there must exist a crown of size exactly 2.)
- CO executions that are not synchronous, also have crowns, e.g., the execution in Figure 6.2(b) has a crown of size 3.

Intuitively, the cyclic dependencies in a crown indicate that it is not possible to find a linear extension in which all the $(s, r)$ event pairs are adjacent. In other words, it is not possible to schedule entire messages in a serial manner, and hence the execution is not RSC.

To determine whether the RSC property holds in $(E, \prec)$, we need to determine whether there exist any cyclic dependencies among messages. Rather than incurring the exponential overhead of checking all linear extensions of $E$, we can check for crowns by using the test in Figure 6.6. On the set of messages $\mathcal{T}$, we define an ordering $\hookrightarrow$ such that $m \hookrightarrow m'$ if and only if $s \prec r'$.

**Example** By drawing the directed graph $(\mathcal{T}, \hookrightarrow)$ for each of the executions in Figures 6.2, 6.3, and 6.5, it can be seen that the graphs for Figures 6.2(d) and Figure 6.3 are acyclic. The other graphs have a cycle.

**Figure 6.6** The crown test to determine the existence of cyclic dependencies among messages.

1. Define the $\hookrightarrow: \mathcal{T} \times \mathcal{T}$ relation on messages in the execution $(E, \prec)$ as follows. Let $\hookrightarrow ([s, r], [s', r'])$ if and only if $s \prec r'$. Observe that the condition $s \prec r'$ (which has the form used in the definition of a crown) is implied by all the four conditions: (i) $s \prec s'$, or (ii) $s \prec r'$, or (iii) $r \prec s'$, and (iv) $r \prec r'$.
2. Now define a *directed* graph $G_{\hookrightarrow} = (\mathcal{T}, \hookrightarrow)$, where the vertex set is the set of messages $\mathcal{T}$ and the edge set is defined by $\hookrightarrow$.
   Observe that the relation $\hookrightarrow: \mathcal{T} \times \mathcal{T}$ is a partial order if and only if $G_{\hookrightarrow}$ has no cycle, i.e., there must not be a cycle with respect to $\hookrightarrow$ on the set of corresponding $(s, r)$ events.
3. It can be seen from the definition of a crown (Definition 6.12) that $G_{\hookrightarrow}$ has a directed cycle if and only if $(E, \prec)$ has a crown.

This test leads to the following theorem [7].

**Theorem 6.1 (Crown criterion)** *The* crown criterion *states that an A-computation is RSC, i.e., it can be realized on a system with synchronous communication, if and only if it contains no crown.*

**Example** Using the directed graph $(\mathcal{T}, \hookrightarrow)$ for each of the executions in Figures 6.2, 6.3(a), and 6.5, it can be seen that the executions in Figures 6.2(d) and Figure 6.3(a) are RSC. The others are not RSC.

Although checking for a non-separated linear extension of $(E, \prec)$ has exponential cost, checking for the presence of a crown based on the message scheduling test of Figure 6.6 can be performed in time that is linear in the number of communication events (see Exercise 6.3). An execution is not RSC and its graph $G_{\hookrightarrow}$ contains a cycle if and only if in the corresponding space–time diagram, it is possible to form a cycle by (i) moving along message arrows in either direction, but (ii) always going left to right along the time line of any process.

As an RSC execution has a non-separated linear extension, it is possible to assign scalar timestamps to events, as it was assigned for a synchronous execution (Definition 6.9), as follows.

**Definition 6.13 (Timestamps for a RSC execution)** An execution $(E, \prec)$ is RSC if and only if there exists a mapping from $E$ to $T$ (scalar timestamps) such that

- for any message $M$, $T(s(M)) = T(r(M))$;
- for each $(a, b)$ in $(E \times E) \setminus \mathcal{T}$, $a \prec b \Longrightarrow T(a) < T(b)$.

From the acyclic message scheduling criterion (Theorem 6.1) and the timestamping property above, it can be observed that an $A$-execution is RSC if and only if its timing diagram can be drawn such that all the message arrows are vertical.

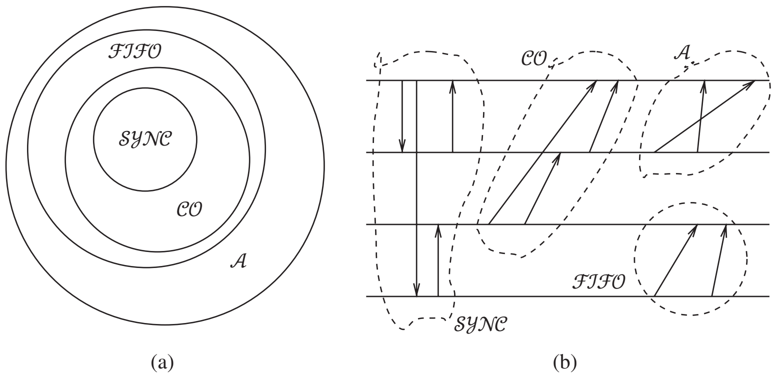


**Figure 6.7** Hierarchy of execution classes. (a) Venn diagram. (b) Example executions.

## 6.2.2 Hierarchy of ordering paradigms

Let $\mathcal{SYNC}$ (or $\mathcal{RSC}$), $\mathcal{CO}$, $\mathcal{FIFO}$, and $\mathcal{A}$ denote the set of all possible executions ordered by synchronous order, causal order, FIFO order, and non-FIFO order, respectively. We have the following results:

- For an $A$-execution, $A$ is RSC if and only if $A$ is an $S$-execution.
- $\mathcal{RSC} \subset \mathcal{CO} \subset \mathcal{FIFO} \subset \mathcal{A}$. This hierarchy is illustrated in Figure 6.7(a), and example executions of each class are shown side-by-side in Figure 6.7(b). Figure 6.1(a) shows an execution that belongs to $\mathcal{A}$ but not to $\mathcal{FIFO}$. Figure 6.2(a) shows an execution that belongs to $\mathcal{FIFO}$ but not to $\mathcal{CO}$. Figures 6.2(b) and (c) show executions that belong to $\mathcal{CO}$ but not to $\mathcal{RSC}$.
- The above hierarchy implies that some executions belonging to a class $X$ will not belong to any of the classes included in $X$. Thus, there are more restrictions on the possible message orderings in the smaller classes. Hence, we informally say that the included classes have less concurrency. The degree of concurrency is most in $\mathcal{A}$ and least in $\mathcal{SYNC}$.
- A program using synchronous communication is easiest to develop and verify. A program using non-FIFO communication, resulting in an $A$-execution, is hardest to design and verify. This is because synchronous order offers the most simplicity due to the restricted number of possibilities, whereas non-FIFO order offers the greatest difficulties because it admits a much larger set of possibilities that the developer and verifier need to account for.

Thus, there is an inherent trade-off between the amount of concurrency provided, and the ease of designing and verifying distributed programs.

## 6.2.3 Simulations

### Asynchronous programs on synchronous systems

Theorem 6.1 indicates that an $A$-execution can be run using synchronous communication primitives if and only if it is an RSC execution. The events in

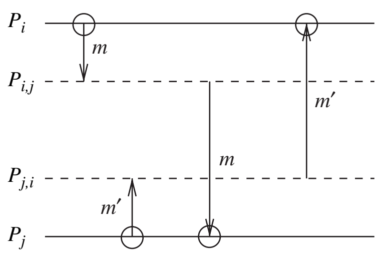


**Figure 6.8** Modeling channels as processes to simulate an execution using asynchronous primitives on an synchronous system.

the RSC execution are scheduled as per some nonseparated linear extension, and adjacent $(s, r)$ events in this linear extension are executed sequentially in the synchronous system. The partial order of the asynchronous execution remains unchanged.

If an $A$-execution is not RSC, then there is no way to schedule the events to make them RSC, without actually altering the partial order of the given $A$-execution. However, the following indirect strategy that does not alter the partial order can be used. Each channel $C_{i,j}$ is modeled by a control process $P_{i,j}$ that simulates the channel buffer. An asynchronous communication from $i$ to $j$ becomes a synchronous communication from $i$ to $P_{i,j}$ followed by a synchronous communication from $P_{i,j}$ to $j$. This enables the decoupling of the sender from the receiver, a feature that is essential in asynchronous systems. This approach is illustrated in Figure 6.8. The communication events at the application processes $P_i$ and $P_j$ are encircled. Observe that it is expensive to implement the channel processes.

### Synchronous programs on asynchronous systems

A (valid) $S$-execution can be trivially realized on an asynchronous system by scheduling the messages in the order in which they appear in the $S$-execution. The partial order of the $S$-execution remains unchanged but the communication occurs on an asynchronous system that uses asynchronous communication primitives. Once a message send event is scheduled, the middleware layer waits for an acknoweldgment; after the ack is received, the synchronous send primitive completes.

## 6.3 Synchronous program order on an asynchronous system

There do not exist real systems with instantaneous communication that allows for synchronous communication to be naturally realized. We need to address the basic question of how a system with synchronous communication can be implemented. We first examine non-determinism in program execution, and CSP as a representative synchronous programming language, before examining an implementation of synchronous communication.

### Non-determinism

The discussions on the message orderings and their characterizations so far assumed a given partial order. This suggests that the distributed programs are *deterministic*, i.e., repeated runs of the same program will produce the same partial order. In many cases, programs are *non-deterministic* in the following senses (we are not considering here the unpredictable message delays that cause different runs to non-deterministically have different global orderings of the events in physical time:)

1. A receive call can receive a message from any sender who has sent a message, if the expected sender is not specified. The receive calls in most of the algorithms in Chapter 5 are non-deterministic in this sense – the receiver is willing to perform a rendezvous with any willing and ready sender.
2. Multiple send and receive calls which are enabled at a process can be executed in an interchangeable order.
   If $i$ sends to $j$, and $j$ sends to $i$ concurrently using blocking synchronous calls, there results a deadlock, similar to the one in Figure 6.4. However, there is no semantic dependency between the send and the immediately following receive at each of the processes. If the receive call at one of the processes can be scheduled before the send call, then there is no deadlock.
   In this section, we consider scheduling synchronous communication events (over an asynchronous system).

## 6.3.1 Rendezvous

One form of group communication is called *multiway rendezvous*, which is a synchronous communication among an arbitrary number of asynchronous processes. All the processes involved "meet with each other," i.e., communicate "synchronously" with each other at one time. The solutions to this problem are fairly complex, and we will not consider them further as this model of synchronous communication is not popular. Here, we study rendezvous between a pair of processes at a time, which is called *binary rendezvous* as opposed to the *multiway rendezvous*.

Support for *binary rendezvous* communication was first provided by programming languages such as CSP and Ada. We consider here a subset of CSP. In these languages, the repetitive command (the $*$ operator) over the alternative command (the $||$ operator) on multiple guarded commands (each having the form $G_i \longrightarrow CL_i$) is used, as follows:

$$*[G_1 \longrightarrow CL_1 \ || \ G_2 \longrightarrow CL_2 \ || \ \cdots \ || \ G_k \longrightarrow CL_k].$$

Each communication command may be a part of a guard $G_i$, and may also appear within the statement block $CL_i$. A guard $G_i$ is a boolean expression. If a guard $G_i$ evaluates to true then $CL_i$ is said to be *enabled*, otherwise $CL_i$ is said to be *disabled*. A send command of local variable $x$ to process $P_k$ is denoted as “$x \,!\, P_k$.” A receive from process $P_k$ into local variable $x$ is denoted as “$P_k \,?\, x$.” Some typical observations about synchronous communication under *binary rendezvous* are as follows:

- For the receive command, the sender must be specified. However, multiple recieve commands can exist. A type check on the data is implicitly performed.
- Send and received commands may be individually disabled or enabled. A command is disabled if it is guarded and the guard evaluates to *false*. The guard would likely contain an expression on some local variables.
- Synchronous communication is implemented by *scheduling* messages under the covers using asynchronous communication. Scheduling involves pairing of matching send and receive commands that are both enabled. The communication events for the control messages under the covers do not alter the partial order of the execution.

The concept underlying *binary rendezvous*, which provides synchronous communication, differs from the concept underlying the classification of synchronous send and receive primitives as blocking or non-blocking (studied in Chapter 1). *Binary rendezvous* explicitly assumes that multiple send and receives are enabled. Any send or receive event that can be “matched” with the corresponding receive or send event can be scheduled. This is dynamically scheduling the ordering of events and the partial order of the execution.

### 6.3.2 Algorithm for binary rendezvous

Various algorithms were proposed to implement *binary rendezvous* in the 1980s [1, 16]. These algorithms typically share the following features. At each process, there is a set of tokens representing the current interactions that are enabled locally. If multiple interactions are enabled, a process chooses one of them and tries to “synchronize” with the partner process. The problem reduces to one of scheduling messages satisfying the following constraints:

- Schedule on-line, atomically, and in a distributed manner, i.e., the scheduling code at any process does not know the application code of other processes.
- Schedule in a deadlock-free manner (i.e., crown-free), such that both the sender and receiver are enabled for a message when it is scheduled.
- Schedule to satisfy the progress property (i.e., find a schedule within a bounded number of steps) in addition to the safety (i.e., correctness) property.

Additional features of a good algorithm are: (i) symmetry or some form of fairness, i.e., not favoring particular processes over others during scheduling, and (ii) efficiency, i.e., using as few messages as possible, and involving as low a time overhead as possible.

We now outline a simple algorithm by Bagrodia [1] that makes the following assumptions:

1. Receive commands are forever enabled from all processes.
2. A send command, once enabled, remains enabled until it completes, i.e., it is not possible that a send command gets disabled (by its guard getting falsified) before the send is executed.
3. To prevent deadlock, process identifiers are used to introduce asymmetry to break potential crowns that arise.
4. Each process attempts to schedule only one *send* event at any time.

The algorithm illustrates how crown-free message scheduling is achieved on-line.

The message types used are: (i) $M$, (ii) $ack(M)$, (iii) $request(M)$, and (iv) $permission(M)$. A process blocks when it knows that it can successfully synchronize the current message with the partner process. Each process maintains a queue that is processed in FIFO order only when the process is unblocked. When a process is blocked waiting for a particular message that it is currently synchronizing, any other message that arrives is queued up.

Execution events in the synchronous execution are only the *send* of the message $M$ and *receive* of the message M. The send and receive events for the other message types – $ack(M)$, $request(M)$, and $permission$(M) which are control messages – are under the covers, and are not included in the synchronous execution. The messages $request(M)$, $ack(M)$, and $permission(M)$ use $M$'s unique tag; the message M is not included in these messages. We use capital SEND(M) and RECEIVE(M) to denote the primitives in the application execution, the lower case send and receive are used for the control messages.

The algorithm to enforce synchronous order is given in Algorithm 6.1. The key rules to prevent cycles among the messages are summarized as follows and illustrated in Figure 6.9:

- To send to a lower priority process, messages $M$ and $ack(M)$ are involved in that order. The sender issues $send(M)$ and blocks until $ack(M)$ arrives. Thus, when sending to a lower priority process, the sender blocks waiting for the partner process to synchronize and send an acknowledgement.
- To send to a higher priority process, messages $request(M)$, $permission(M)$, and $M$ are involved, in that order. The sender issues $send(request(M))$, does not block, and awaits permission. When $permission(M)$ arrives, the sender issues $send(M)$.

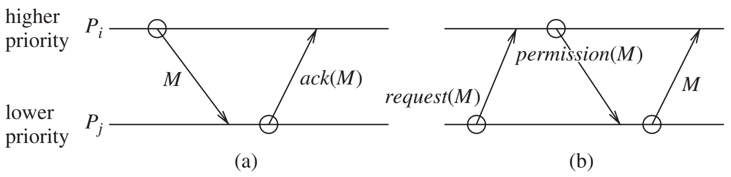


**Figure 6.9** Messages used to implement synchronous order. $P_i$ has higher priority than $P_j$. (a) $P_i$ issues SEND(M). (b) $P_j$ issues SEND(M).

(message types)
$M$, $ack(M)$, $request(M)$, $permission(M)$

(1) **$P_i$ wants to execute SEND($M$) to a lower priority process $P_j$:**
$P_i$ executes $send(M)$ and blocks until it receives $ack(M)$ from $P_j$. The send event SEND($M$) now completes.

Any $M'$ message (from a higher priority processes) and $request(M')$ request for synchronization (from a lower priority processes) received during the blocking period are queued.

(2) **$P_i$ wants to execute SEND($M$) to a higher priority process $P_j$:**

(2a) $P_i$ seeks permission from $P_j$ by executing $send(request(M))$.
// to avoid deadlock in which cyclically blocked processes queue
// messages.

(2b) While $P_i$ is waiting for permission, it remains unblocked.

(i) If a message $M'$ arrives from a higher priority process $P_k$, $P_i$ accepts $M'$ by scheduling a RECEIVE($M'$) event and then executes $send(ack(M'))$ to $P_k$.

(ii) If a $request(M')$ arrives from a lower priority process $P_k$, $P_i$ executes $send(permission(M'))$ to $P_k$ and blocks waiting for the message $M'$. When $M'$ arrives, the RECEIVE($M'$) event is executed.

(2c) When the $permission(M)$ arrives, $P_i$ knows partner $P_j$ is synchronized and $P_i$ executes $send(M)$. The SEND($M$) now completes.

(3) ***request*(*M*) arrival at $P_i$ from a lower priority process $P_j$:**
At the time a $request(M)$ is processed by $P_i$, process $P_i$ executes $send(permission(M))$ to $P_j$ and blocks waiting for the message $M$. When $M$ arrives, the RECEIVE($M$) event is executed and the process unblocks.

(4) **Message $M$ arrival at $P_i$ from a higher priority process $P_j$:**
At the time a message $M$ is processed by $P_i$, process $P_i$ executes RECEIVE($M$) (which is assumed to be always enabled) and then $send(ack(M))$ to $P_j$.

(5) **Processing when $P_i$ is unblocked:**
When $P_i$ is unblocked, it dequeues the next (if any) message from the queue and processes it as a message arrival (as per rules 3 or 4).

**Algorithm 6.1** A simplified implementation of synchronous order. Code shown is for process $P_i$, $1 \le i \le n$.

Thus, when sending to a higher priority process, the sender asks the higher priority process via the $request(M)$ to give permission to send. When the higher priority process gives permission to send, the higher priority process, which is the intended receiver, blocks.

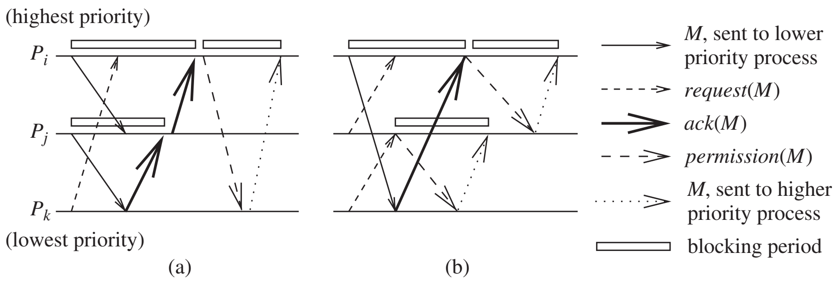


**Figure 6.10** Examples showing how to schedule messages sent with synchronous primitives.

In either case, a higher priority process blocks on a lower priority process. So cyclic waits are avoided.

In more detail, a cyclic wait is prevented because before sending a message $M$ to a higher priority process, a lower priority process requests the higher priority process for permission to synchronize on $M$, in a non-blocking manner. While waiting for this permission, there are two possibilities:

1. If a message $M'$ from a higher priority process arrives, it is processed by a receive (assuming receives are always enabled) and *ack*($M'$) is returned. Thus, a cyclic wait is prevented.
2. Also, while waiting for this permission, if a *request*($M'$) from a lower priority process arrives, a *permission*($M'$) is returned and the process blocks until $M'$ actually arrives.

Note that the *receive*($M'$) event effectively gets permuted before the *send*($M$) event (steps 2(bi) and 2(bii)).

**Examples:** Figure 6.10 shows two examples of how the algorithm breaks cyclic waits to schedule messages. Observe that in all cases in the algorithm, a higher priority process blocks on lower priority processes, irrespective of whether the higher priority process is the intended sender or the receiver of the message being scheduled. In Figure 6.10(a), at process $P_k$, the receive of the message from $P_j$ effectively gets permuted before $P_k$'s own *send*($M$) event due to step 2(bi). In Figure 6.10(b), at process $P_j$, the receive of the *request*($M'$) message from $P_k$ effectively causes $M'$ to be permuted before $P_j$'s own message that it was attempting to schedule with $P_i$, due to step 2(bii).

## 6.4 Group communication

Processes across a distributed system cooperate to solve a joint task. Often, they need to communicate with each other as a group, and therefore there needs to be support for *group communication*. A *message broadcast* is the sending of a message to all members in the distributed system. The notion of a system can be confined only to those sites/processes participating in the joint application. Refining the notion of *broadcasting*, there is *multicasting* wherein a message is sent to a certain subset, identified as a *group*, of the processes in the system. At the other extreme is *unicasting*, which is the familiar point-to-point message communication.

Broadcast and multicast support can be provided by the network protocol stack using variants of the spanning tree. This is an efficient mechanism for distributing information. However, the hardware-assisted or network layer protocol assisted multicast cannot efficiently provide features such as the following:

- Application-specific ordering semantics on the order of delivery of messages.
- Adapting groups to dynamically changing membership.
- Sending multicasts to an arbitrary set of processes at each send event.
- Providing various fault-tolerance semantics.

If a multicast algorithm requires the sender to be a part of the destination group, the multicast algorithm is said to be a *closed group* algorithm. If the sender of the multicast can be outside the destination group, the multicast algorithm is said to be an *open group* algorithm. Open group algorithms are more general, and therefore more difficult to design and more expensive to implement, than closed group algorithms. Closed group algorithms cannot be used in several scenarios such as in a large system (e.g., on-line reservation or Internet banking systems) where client processes are short-lived and in large numbers. It is also worth noting that, for multicast algorithms, the number of groups may be potentially exponential, i.e., $O(2^n)$, and algorithms that have to explicitly track the groups can incur this high overhead.

In the remainder of this chapter we will examine multicast and broadcast mechanisms under varying degrees of strictness of assumptions on the order of delivery of messages. Two popular orders for the delivery of messages were proposed in the context of group communication: *causal order* and *total order*. Much of the seminal work on group communication was initiated by the ISIS project [4,5].

## 6.5 Causal order (CO)

Causal order has many applications such as updating replicated data, allocating requests in a fair manner, and synchronizing multimedia streams. We explain here the use of causal order in updating replicas of a data item in the system. Consider Figure 6.11(a), which shows two processes $P_1$ and $P_2$ that issue updates to the three replicas $R1(d)$, $R2(d)$, and $R3(d)$ of data item $d$. Message $m$ creates a causality between $send(m1)$ and $send(m2)$. If $P_2$ issues its update causally after $P_1$ issued its update, then $P_2$'s update should be seen by the replicas after they see $P_1$'s update, in order to preserve the semantics

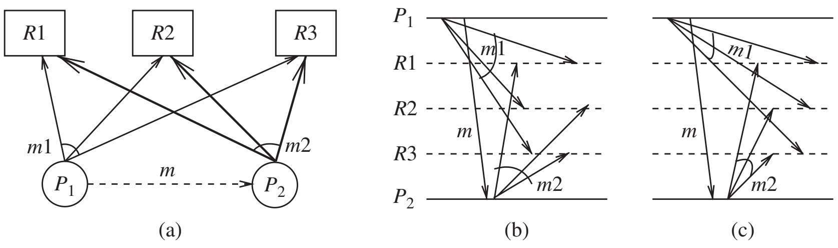


**Figure 6.11** Updates to object replicas are issued by two processes.

of the application. (In this case, CO is satisfied.) However, this may happen at some, all, or none of the replicas. Figure 6.11(b) shows that $R1$ sees $P_2$'s update first, while $R2$ and $R3$ see $P_1$'s update first. Here, CO is violated. Figure 6.11(c) shows that all replicas see $P_2$'s update first. However, CO is still violated. If message $m$ did not exist as shown, then the executions shown in Figure 6.11(b) and (c) would satisfy CO.

Given a system with FIFO channels, causal order needs to be explicitly enforced by a protocol. The following two criteria must be met by a causal ordering protocol:

- **Safety** In order to prevent causal order from being violated, a message $M$ that arrives at a process may need to be buffered until all systemwide messages sent in the causal past of the $send(M)$ event to that same destination have already arrived.
  Therefore, we distinguish between the arrival of a message at a process (at which time it is placed in a local system buffer) and the event at which the message is given to the application process (when the protocol deems it safe to do so without violating causal order). The arrival of a message is transparent to the application process. The delivery event corresponds to the *receive* event in the execution model.
- **Liveness** A message that arrives at a process must eventually be delivered to the process.

Both the algorithms we will study in this section allow each send event to unicast, multicast, or broadcast a message in the system.

### 6.5.1 The Raynal–Schiper–Toueg algorithm [22]

Intuitively, it seems logical that each message $M$ should carry a log of all other messages, or their identifiers, sent causally before $M$'s send event, and sent to the same destination $dest(M)$. This log can then be examined to ensure whether it is safe to deliver a message. All algorithms aim to reduce this log overhead, and the space and time overhead of maintaining the log information at the processes. Algorithm 6.2 gives a canonical algorithm that is representative of several algorithms that try to reduce the size of the local space and message space overhead by various techniques. In order to implement safety, the messages piggyback the control information that helps joint application. Refining the notion of *broadcasting*, there is *multicasting* wherein a message is sent to a certain subset, identified as a *group*, of the processes in the system. At the other extreme is *unicasting*, which is the familiar point-to-point message communication.

Broadcast and multicast support can be provided by the network protocol stack using variants of the spanning tree. This is an efficient mechanism for distributing information. However, the hardware-assisted or network layer protocol assisted multicast cannot efficiently provide features such as the following:

- Application-specific ordering semantics on the order of delivery of messages.
- Adapting groups to dynamically changing membership.
- Sending multicasts to an arbitrary set of processes at each send event.
- Providing various fault-tolerance semantics.

If a multicast algorithm requires the sender to be a part of the destination group, the multicast algorithm is said to be a *closed group* algorithm. If the sender of the multicast can be outside the destination group, the multicast algorithm is said to be an *open group* algorithm. Open group algorithms are more general, and therefore more difficult to design and more expensive to implement, than closed group algorithms. Closed group algorithms cannot be used in several scenarios such as in a large system (e.g., on-line reservation or Internet banking systems) where client processes are short-lived and in large numbers. It is also worth noting that, for multicast algorithms, the number of groups may be potentially exponential, i.e., $O(2^n)$, and algorithms that have to explicitly track the groups can incur this high overhead.

In the remainder of this chapter we will examine multicast and broadcast mechanisms under varying degrees of strictness of assumptions on the order of delivery of messages. Two popular orders for the delivery of messages were proposed in the context of group communication: *causal order* and *total order*. Much of the seminal work on group communication was initiated by the ISIS project [4,5].

## 6.5 Causal order (CO)

Causal order has many applications such as updating replicated data, allocating requests in a fair manner, and synchronizing multimedia streams. We explain here the use of causal order in updating replicas of a data item in the system. Consider Figure 6.11(a), which shows two processes $P_1$ and $P_2$ that issue updates to the three replicas $R1(d)$, $R2(d)$, and $R3(d)$ of data item $d$. Message $m$ creates a causality between $send(m1)$ and $send(m2)$. If $P_2$ issues its update causally after $P_1$ issued its update, then $P_2$'s update should be seen by the replicas after they see $P_1$'s update, in order to preserve the semantics

An optimal CO algorithm stores in local message logs and propagates on messages, information of the form "$d$ is a destination of $M$" about a message $M$ sent in the causal past, *as long as* and *only as long as*:

(*Propagation Constraint I*) it is not known that the message $M$ is delivered to $d$, and

(*Propagation Constraint II*) it is not known that a message has been sent to $d$ in the causal future of $Send(M)$, and hence it is not guaranteed using a reasoning based on transitivity that the message $M$ will be delivered to $d$ in CO.

The Propagation Constraints also imply that if either (I) or (II) is false, the information "$d \in M.Dests$" must *not* be stored or propagated, even to remember that (I) or (II) has been falsified. Stated differently, the information "$d \in M_{i,a}.Dests$" must be available in the causal future of event $e_{i,a}$, but:

- not in the causal future of $Deliver_d(M_{i,a})$, and
- not in the causal future of $e_{k,c}$, where $d \in M_{k,c}.Dests$ and there is no other message sent causally between $M_{i,a}$ and $M_{k,c}$ to the same destination $d$.

In the causal future of $Deliver_d(M_{i,a})$, and $Send(M_{k,c})$, the information is redundant; elsewhere, it is necessary. Additionally, to maintain optimality, no other information should be stored, including information about what messages have been delivered. As information about what messages have been delivered (or are guaranteed to be delivered without violating causal order) is necessary for the Delivery Condition, this information is inferred using a set-operation based logic.

The Propagation Constraints are illustrated with the help of Figure 6.12. The message $M$ is sent by process $i$ at event $e$ to process $d$. The information "$d \in M.Dests$":

- must exist at $e1$ and $e2$ because (I) and (II) are true;
- must not exist at $e3$ because (I) is false;
- must not exist at $e4$, $e5$, $e6$ because (II) is false;
- must not exist at $e7$, $e8$ because (I) and (II) are false.

Information about messages (i) not known to be delivered and (ii) not guaranteed to be delivered in CO, is *explicitly* tracked by the algorithm using (*source, timestamp, destination*) information. The information must be deleted as soon as either (i) or (ii) becomes false. The key problem in designing an optimal CO algorithm is to identify the events at which (i) or (ii) becomes false. Information about messages already delivered and messages guaranteed to be delivered in CO is *implicitly* tracked without storing or propagating it, and is derived from the explicit information. Such implicit information is used for determining when (i) or (ii) becomes false for the explicit information being stored or carried in messages.

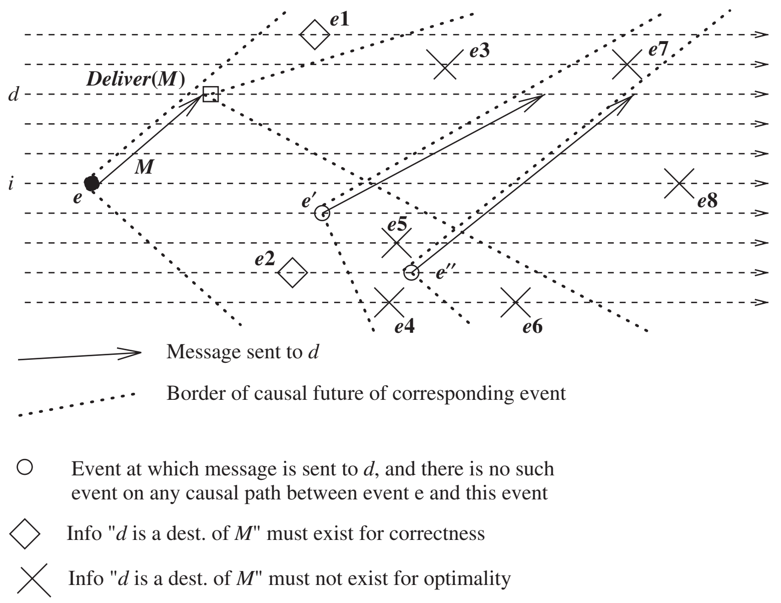

e1
e3
e7
Deliver(M)
d
M
i
e
e′
e8
e5
e2
e″
e4
e6
Message sent to d
Border of causal future of corresponding event
Event at which message is sent to d, and there is no such event on any causal path between event e and this event
Info "d is a dest. of M" must exist for correctness
Info "d is a dest. of M" must not exist for optimality

**Figure 6.12** Illustrating the necessary and sufficient conditions for causal ordering [21].

The algorithm is given in Algorithm 6.3. Procedure SND is executed atomically. Procedure RCV is executed atomically except for a possible interruption in line 2a where a non-blocking wait is required to meet the Delivery Condition. Note that the pseudo-code can be restructured to complete the processing of each invocation of SND and RCV procedures in a single pass of the data structures, by always maintaining the data structures sorted row–major and then column–major.

1. **Explicit tracking** Tracking of (source, timestamp, destination) information for messages (i) not known to be delivered and (ii) not guaranteed to be delivered in CO, is done explicitly using the $l.Dests$ field of entries in local logs at nodes and $o.Dests$ field of entries in messages. Sets $l_{i,a}.Dests$ and $o_{i,a}.Dests$ contain explicit information of destinations to which $M_{i,a}$ is not guaranteed to be delivered in CO and is not known to be delivered. The information about "$d \in M_{i,a}.Dests$" is propagated up to the earliest events on all causal paths from $(i, a)$ at which it is known that $M_{i,a}$ is delivered to $d$ or is guaranteed to be delivered to $d$ in CO.
2. **Implicit tracking** Tracking of messages that are either (i) already delivered, or (ii) guaranteed to be delivered in CO, is performed implicitly.
   The information about messages (i) already delivered or (ii) guaranteed to be delivered in CO is deleted and not propagated because it is redundant as far as enforcing CO is concerned. However, it is useful in determining what information that is being carried in other messages and is being stored in logs at other nodes has become redundant and thus can be purged. The semantics are implicitly stored and propagated. This information about messages that are (i) already delivered or (ii) guaranteed to be delivered in

```
(local variables)
$clock_j \longleftarrow 0$;                                        // local counter clock at node $j$
$SR_j[1 \ldots n] \longleftarrow \overline{0}$;          // $SR_j[i]$ is the timestamp of last msg. from $i$ delivered to $j$
$LOG_j = \{(i, clock_i, Dests)\} \longleftarrow \{\forall i, (i, 0, \emptyset)\}$;
          // Each entry denotes a message sent in the causal past, by $i$ at $clock_i$. $Dests$ is the set of
       // remaining destinations for which it is not known that
                    // $M_{i,clock_i}$ (i) has been delivered, or (ii) is guaranteed to be delivered in CO.

(1) SND: $j$ sends a message $M$ to $Dests$:

   (1a) $clock_j \longleftarrow clock_j + 1$;
   (1b) for all $d \in M.Dests$ do:
            $O_M \longleftarrow LOG_j$;                                    // $O_M$ denotes $O_{M_{j,clock_j}}$
            for all $o \in O_M$, modify $o.Dests$ as follows:
               if $d \notin o.Dests$ then $o.Dests \longleftarrow (o.Dests \setminus M.Dests)$;
               if $d \in o.Dests$ then $o.Dests \longleftarrow (o.Dests \setminus M.Dests) \bigcup \{d\}$;
               // Do not propagate information about indirect dependencies that are
          // guaranteed to be transitively satisfied when dependencies of $M$ are satisfied.
            for all $o_{s,t} \in O_M$ do
               if $o_{s,t}.Dests = \emptyset \bigwedge (\exists o'_{s,t'} \in O_M \mid t < t')$ then $O_M \longleftarrow O_M \setminus \{o_{s,t}\}$;
                     // do not propagate older entries for which $Dests$ field is $\emptyset$
            send $(j, clock_j, M, Dests, O_M)$ to $d$;
   (1c) for all $l \in LOG_j$ do $l.Dests \longleftarrow l.Dests \setminus Dests$;
            // Do not store information about indirect dependencies that are guaranteed
                  // to be transitively satisfied when dependencies of $M$ are satisfied.
        Execute $PURGE\_NULL\_ENTRIES(LOG_j)$;        // purge $l \in LOG_j$ if $l.Dests = \emptyset$
   (1d) $LOG_j \longleftarrow LOG_j \bigcup \{(j, clock_j, Dests)\}$.

(2) RCV: $j$ receives a message $(k, t_k, M, Dests, O_M)$ from $k$:

   (2a) // Delivery Condition: ensure that messages sent causally before M are delivered.
        for all $o_{m,t_m} \in O_M$ do
           if $j \in o_{m,t_m}.Dests$ wait until $t_m \leq SR_j[m]$;
   (2b) Deliver M; $SR_j[k] \longleftarrow t_k$;
   (2c) $O_M \longleftarrow \{(k, t_k, Dests)\} \bigcup O_M$;
        for all $o_{m,t_m} \in O_M$ do $o_{m,t_m}.Dests \longleftarrow o_{m,t_m}.Dests \setminus \{j\}$;
          // delete the now redundant dependency of message represented by $o_{m,t_m}$ sent to $j$
   (2d) // Merge $O_M$ and $LOG_j$ by eliminating all redundant entries.
        // Implicitly track "already delivered" & "guaranteed to be delivered in CO"
        // messages.
        for all $o_{m,t} \in O_M$ and $l_{s,t'} \in LOG_j$ such that $s = m$ do
           if $t < t' \bigwedge l_{s,t} \notin LOG_j$ then mark $o_{m,t}$;
             // $l_{s,t}$ had been deleted or never inserted, as $l_{s,t}.Dests = \emptyset$ in the causal past
           if $t' < t \bigwedge o_{m,t'} \notin O_M$ then mark $l_{s,t'}$;
             // $o_{m,t'} \notin O_M$ because $l_{s,t'}$ had become $\emptyset$ at another process in the causal past
        Delete all marked elements in $O_M$ and $LOG_j$ ;
             // delete entries about redundant information
        for all $l_{s,t'} \in LOG_j$ and $o_{m,t} \in O_M$, such that $s = m \bigwedge t' = t$ do
           $l_{s,t'}.Dests \longleftarrow l_{s,t'}.Dests \bigcap o_{m,t}.Dests$;
                                                   // delete destinations for which Delivery
                         // Condition is satisfied or guaranteed to be satisfied as per $o_{m,t}$
           Delete $o_{m,t}$ from $O_M$;          // information has been incorporated in $l_{s,t'}$
        $LOG_j \longleftarrow LOG_j \bigcup O_M$;     // merge non-redundant information of $O_M$ into $LOG_j$
   (2e) $PURGE\_NULL\_ENTRIES(LOG_j)$.  // Purge older entries $l$ for which $l.Dests = \emptyset$

PURGE_NULL_ENTRIES($Log_j$):    // Purge older entries $l$ for which $l.Dests = \emptyset$ is
                                // implicitly inferred
for all $l_{s,t} \in Log_j$ do
      if $l_{s,t}.Dests = \emptyset \bigwedge (\exists l'_{s,t'} \in Log_j \mid t < t')$ then $Log_j \longleftarrow Log_j \setminus \{l_{s,t}\}$.
```

**Algorithm 6.3** The algorithm by Kshemkalyani–Singhal to optimally implement causal ordering of messages. Code for $P_j$, $1 \leq j \leq n$.

CO is tracked without explicitly storing it. Rather, the algorithm derives it from the existing explicit information about messages (i) not known to be delivered and (ii) not guaranteed to be delivered in CO, by examining only $o_{i,a}.Dests$ or $l_{i,a}.Dests$, which is a part of the explicit information. There are two types of implicit tracking:

- The absence of a node i.d. from destination information – i.e., $\exists d \in M_{i,a}.Dests \mid d \notin l_{i,a}.Dests \bigvee d \notin o_{i,a}.Dests$ – implicitly contains information that the message has been already delivered or is guaranteed to be delivered in CO to $d$. Clearly, $l_{i,a}.Dests = \emptyset$ or $o_{i,a}.Dests = \emptyset$ implies that message $M_{i,a}$ has been delivered or is guaranteed to be delivered in CO to *all* destinations in $M_{i,a}.Dests$. An entry whose $.Dests = \emptyset$ is maintained because of the implicit information in it, viz., that of known delivery or guaranteed CO delivery to all destinations of the multicast, is useful to purge redundant information as per the Propagation Constraints.
- As the distributed computation evolves, several entries $l_{i,a_1}$, $l_{i,a_2}, \ldots$ such that $\forall p$, $l_{i,a_p}.Dests = \emptyset$ may exist in a node's log and a message may be carrying several entries $o_{i,a_1}$, $o_{i,a_2}, \ldots$ such that $\forall p$, $o_{i,a_p}.Dests = \emptyset$. The second implicit tracking uses a mechanism to prevent the proliferation of such entries. The mechanism is based on the following observation: "*For any two multicasts $M_{i,a_1}$, $M_{i,a_2}$ such that $a_1 < a_2$, if $l_{i,a_2} \in LOG_j$, then $l_{i,a_1} \in LOG_j$. (Likewise for any message.)*" Therefore, if $l_{i,a_1}.Dests$ becomes $\emptyset$ at a node $j$, then it can be deleted from $LOG_j$ provided $\exists\ l_{i,a_2} \in LOG_j$ such that $a_1 < a_2$. The presence of such $l_{i,a_1}$s in $LOG_j$ is automatically implied by the presence of entry $l_{i,a_2}$ in $LOG_j$. Thus, for a multicast $M_{i,z}$, if $l_{i,z}$ does not exist in $LOG_j$, then $l_{i,z}.Dests = \emptyset$ implicitly exists in $LOG_j$ iff $\exists\ l_{i,a} \in LOG_j \mid a > z$. As a result of the second implicit tracking mechanism, a node does not keep (and a message does not carry) entries of type $l_{i,a}.Dests = \emptyset$ in its log. However, note that a node must always keep at least one entry of type $l_{i,a}$ (the one with the highest timestamp) in its log for each sender node $i$. The same holds for messages.

The information tracked implicitly is useful in purging information explicitly carried in other $O_{M''}$s and stored in $LOG$ entries about "yet to be delivered to" destinations for the same message $M_{i,a}$ as well as for messages $M_{i,a'}$, where $a' < a$. Thus, whenever $o_{i,a}$ in some $O_{M'}$ propagates to node $j$, in line (2d), (i) the implicit information in $o_{i,a}.Dests$ is used to eliminate redundant information in $l_{i,a}.Dests \in LOG_j$; (ii) the implicit information in $l_{i,a}.Dests \in LOG_j$ is used to eliminate redundant information in $o_{i,a}.Dests$; (iii) the implicit information in $o_{i,a}$ is used to eliminate redundant information $l_{i,a'} \in LOG_j$ if $\nexists\ o_{i,a'} \in O_{M'}$ and $a' < a$; (iv) the implicit information in $l_{i,a}$ is used to eliminate redundant information $o_{i,a'} \in O_{M'}$ if $\nexists\ l_{i,a'} \in LOG_j$ and $a' < a$; and (v) only non-redundant information remains in $O_{M'}$ and $LOG_j$; this is merged together into an updated $LOG_j$.

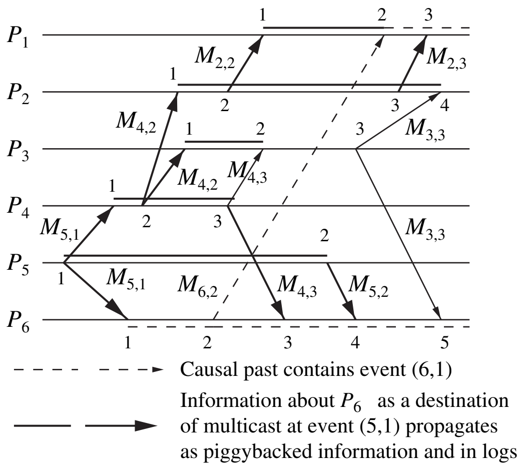
**Example [6]** In the example in Figure 6.13, the timing diagram illustrates (i) the propagation of explicit information "$P_6 \in M_{5,1}.Dests$" and (ii) the inference of implicit information that "$M_{5,1}$ has been delivered to $P_6$, or is guaranteed to be delivered in causal order to $P_6$ with respect to any future messages." A thick arrow indicates that the corresponding message contains the explicit information piggybacked on it. A thick line during some interval of the time line of a process indicates the duration in which this information resides in the log local to that process. The number "*a*" next to an event indicates that it is the *a*th event at that process.

## Multicasts $M_{5,1}$ and $M_{4,2}$

Message $M_{5,1}$ sent to processes $P_4$ and $P_6$ contains the piggybacked information "$M_{5,1}.Dests = \{P_4, P_6\}$." Additionally, at the send event (5, 1), the information "$M_{5,1}.Dests = \{P_4, P_6\}$" is also inserted in the local log $Log_5$. When $M_{5,1}$ is delivered to $P_6$, the (new) piggybacked information "$P_4 \in M_{5,1}.Dests$" is stored in $Log_6$ as "$M_{5,1}.Dests = \{P_4\}$"; information about "$P_6 \in M_{5,1}.Dests$," which was needed for routing, must *not* be stored in $Log_6$ because of constraint I. Symmetrically, when $M_{5,1}$ is delivered to process $P_4$ at event (4, 1), *only* the new piggybacked information "$P_6 \in M_{5,1}.Dests$" is inserted in $Log_4$ as "$M_{5,1}.Dests = \{P_6\}$," which is later propagated during multicast $M_{4,2}$.

## Multicast $M_{4,3}$

At event (4, 3), the information "$P_6 \in M_{5,1}.Dests$" in $Log_4$ is propagated on multicast $M_{4,3}$ only to process $P_6$ to ensure causal delivery using the Delivery Condition. The piggybacked information on message $M_{4,3}$ sent to process $P_3$ must not contain this information because of constraint II. (The piggybacked information contains "$M_{4,3}.Dests = \{P_6\}$." As long as any future message



$P_1$
$P_2$
$P_3$
$P_4$
$P_5$
$P_6$
$M_{2,2}$
$M_{2,3}$
$M_{4,2}$
$M_{3,3}$
$M_{4,3}$
$M_{5,1}$
$M_{6,2}$
$M_{5,2}$
Causal past contains event (6,1)
Information about $P_6$ as a destination of multicast at event (5,1) propagates as piggybacked information and in logs

**Figure 6.13** An example to illustrate the propagation constraints [6].

| Message to dest. | Piggybacked $M_{5,1}$.Dests |
|---|---|
| $M_{5,1}$to $P_4$,$P_6$ | $\{P_4,P_6\}$ |
| $M_{4,2}$to $P_3$,$P_2$ | $\{P_6\}$ |
| $M_{2,2}$to $P_1$ | $\{P_6\}$ |
| $M_{6,2}$to $P_1$ | $\{P_4\}$ |
| $M_{4,3}$to $P_6$ | $\{P_6\}$ |
| $M_{4,3}$to $P_3$ | $\{\}$ |
| $M_{5,2}$to $P_6$ | $\{P_4,P_6\}$ |
| $M_{2,3}$to $P_1$ | $\{P_6\}$ |
| $M_{3,3}$to $P_{2,6}$ | $\{\}$ |

sent to $P_6$ is delivered in causal order w.r.t. $M_{4,3}$ sent to $P_6$, it will also be delivered in causal order w.r.t. $M_{5,1}$ sent to $P_6$.) And as $M_{5,1}$ is already delivered to $P_4$, the information "$M_{5,1}.Dests = \emptyset$" is piggybacked on $M_{4,3}$ sent to $P_3$. Similarly, the information "$P_6 \in M_{5,1}.Dests$" must be deleted from $Log_4$ as it will no longer be needed, because of constraint II. "$M_{5,1}.Dests = \emptyset$" is stored in $Log_4$ to remember that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to all its destinations.

### Learning implicit information at $P_2$ and $P_3$

When message $M_{4,2}$ is received by processes $P_2$ and $P_3$, they insert the (new) piggybacked information in their local logs, as information "$M_{5,1}.Dests = \{P_6\}$." They both continue to store this in $Log_2$ and $Log_3$ and propagate this information on multicasts until they "learn" at events (2, 4) and (3, 2) on receipt of messages $M_{3,3}$ and $M_{4,3}$, respectively, that any future message is guaranteed to be delivered in causal order to process $P_6$, w.r.t. $M_{5,1}$ sent to $P_6$. Hence by constraint II, this information must be deleted from $Log_2$ and $Log_3$. The logic by which this "learning" occurs is as follows:

- When $M_{4,3}$ with piggybacked information "$M_{5,1}.Dests = \emptyset$" is received by $P_3$ at (3, 2), this is inferred to be valid current *implicit* information about multicast $M_{5,1}$ because the log $Log_3$ already contains explicit information "$P_6 \in M_{5,1}.Dests$" about that multicast. Therefore, the explicit information in $Log_3$ is inferred to be old and must be deleted to achieve optimality. $M_{5,1}.Dests$ is set to $\emptyset$ in $Log_3$.
- The logic by which $P_2$ learns this implicit knowledge on the arrival of $M_{3,3}$ is identical.

### Processing at $P_6$

Recall that when message $M_{5,1}$ is delivered to $P_6$, only "$M_{5,1}.Dests = \{P_4\}$" is added to $Log_6$. Further, $P_6$ propagates only "$M_{5,1}.Dests = \{P_4\}$" (from $Log_6$) on message $M_{6,2}$, and this conveys the current *implicit* information "$M_{5,1}$ has been delivered to $P_6$," by its very absence in the explicit information.

- When the information "$P_6 \in M_{5,1}.Dests$" arrives on $M_{4,3}$, piggybacked as "$M_{5,1}.Dests = \{P_6\}$," it is used only to ensure causal delivery of $M_{4,3}$ using the Delivery Condition, and is not inserted in $Log_6$ (constraint I) – further, the presence of "$M_{5,1}.Dests = \{P_4\}$" in $Log_6$ implies the *implicit* information that $M_{5,1}$ has already been delivered to $P_6$. Also, the absence of $P_4$ in $M_{5,1}.Dests$ in the explicit piggybacked information implies the *implicit* information that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to $P_4$, and, therefore, $M_{5,1}.Dests$ is set to $\emptyset$ in $Log_6$.
- When the information "$P_6 \in M_{5,1}.Dests$" arrives on $M_{5,2}$, piggybacked as "$M_{5,1}.Dests = \{P_4, P_6\}$," it is used only to ensure causal delivery of

$M_{4,3}$ using the Delivery Condition, and is not inserted in $Log_6$ because $Log_6$ contains "$M_{5,1}.Dests = \emptyset$," which gives the *implicit* information that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to both $P_4$ and $P_6$. (Note that at event (5, 2), $P_5$ changes $M_{5,1}.Dests$ in $Log_5$ from $\{P_4, P_6\}$ to $\{P_4\}$, as per constraint II, and inserts "$M_{5,2}.Dests = \{P_6\}$" in $Log_5$.)

### Processing at $P_1$

We have the following processing:

- When $M_{2,2}$ arrives carrying piggybacked information "$M_{5,1}.Dests = \{P_6\}$," this (new) information is inserted in $Log_1$.
- When $M_{6,2}$ arrives with piggybacked information "$M_{5,1}.Dests = \{P_4\}$," $P_1$ "learns" *implicit* information "$M_{5,1}$ has been delivered to $P_6$" by the very absence of explicit information "$P_6 \in M_{5,1}.Dests$" in the piggybacked information, and hence marks information "$P_6 \in M_{5,1}.Dests$" for deletion from $Log_1$. Simultaneously, "$M_{5,1}.Dests = \{P_6\}$" in $Log_1$ implies the *implicit* information that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to $P_4$. Thus, $P_1$ also "learns" that the explicit piggybacked information "$M_{5,1}.Dests = \{P_4\}$" is outdated. $M_{5,1}.Dests$ in $Log_1$ is set to $\emptyset$.
- Analogously, the information "$P_6 \in M_{5,1}.Dests$" piggybacked on $M_{2,3}$, which arrives at $P_1$, is inferred to be outdated (and hence ignored) using the *implicit* knowledge derived from "$M_{5,1}.Dests = \emptyset$" in $Log_1$.

## 6.6 Total order

While causal order has many uses, there are other orderings that are also useful. *Total order* is such an ordering [4,5]. Consider the example of updates to replicated data, as shown in Figure 6.11. As the replicas are of just one data item $d$, it would be logical to expect that all replicas see the updates in the same order, whether or not the issuing of the updates are causally related. This way, the issue of coherence and consistency of the replica values goes away. Such a replicated system would still be useful for fault-tolerance, as well as for easy availability for "read" operations. Total order, which requires that all messages be received in the same order by the recipients of the messages, is formally defined as follows:

**Definition 6.14 (Total order)** For each pair of processes $P_i$ and $P_j$ and for each pair of messages $M_x$ and $M_y$ that are delivered to both the processes, $P_i$ is delivered $M_x$ before $M_y$ if and only if $P_j$ is delivered $M_x$ before $M_y$.

**Example** The execution in Figure 6.11(b) does not satisfy total order. Even if the message *m* did not exist, total order would not be satisfied. The execution in Figure 6.11(c) satisfies total order.

### 6.6.1 Centralized algorithm for total order

Assuming all processes broadcast messages, the centralized solution shown in Algorithm 6.4 enforces total order in a system with FIFO channels. Each process sends the message it wants to broadcast to a centralized process, which simply relays all the messages it receives to every other process over FIFO channels. It is straightforward to see that total order is satisfied. Furthermore, this algorithm also satisfies causal message order.

```
(1)   When process $P_i$ wants to multicast a message $M$ to group $G$:
(1a)  send $M(i, G)$ to central coordinator.

(2)   When $M(i, G)$ arrives from $P_i$ at the central coordinator:
(2a)  send $M(i, G)$ to all members of the group $G$.

(3)   When $M(i, G)$ arrives at $P_j$ from the central coordinator:
(3a)  deliver $M(i, G)$ to the application.
```

**Algorithm 6.4** A centralized algorithm to implement total order and causal order of messages.

*Complexity*

Each message transmission takes two message hops and exactly $n$ messages in a system of $n$ processes.

*Drawbacks*

A centralized algorithm has a single point of failure and congestion, and is therefore not an elegant solution.

### 6.6.2 Three-phase distributed algorithm

A distributed algorithm that enforces total and causal order for closed groups is given in Algorithm 6.5. The three phases of the algorithm are first described from the viewpoint of the sender, and then from the viewpoint of the receiver.

**Sender**

**Phase 1** In the first phase, a process multicasts (line 1b) the message $M$ with a locally unique tag and the local timestamp to the group members.

**Phase 2** In the second phase, the sender process awaits a reply from all the group members who respond with a tentative proposal for a revised timestamp for that message $M$. The **await** call in line 1d is non-blocking,

```
record Q_entry
        M: int;                                        // the application message
        tag: int;                                      // unique message identifier
        sender_id: int;                                // sender of the message
        timestamp: int;                // tentative timestamp assigned to message
        deliverable: boolean;            // whether message is ready for delivery
(local variables)
queue of Q_entry: temp_Q, delivery_Q
int: clock                       // Used as a variant of Lamport's scalar clock
int: priority                  // Used to track the highest proposed timestamp
(message types)
REVISE_TS(M, i, tag, ts)
                    // Phase 1 message sent by $P_i$, with initial timestamp ts
PROPOSED_TS(j, i, tag, ts)
             // Phase 2 message sent by $P_j$, with revised timestamp, to $P_i$
FINAL_TS(i, tag, ts)     // Phase 3 message sent by $P_i$, with final timestamp

(1)   When process $P_i$ wants to multicast a message M with a tag tag:
(1a)  $clock \leftarrow clock + 1$;
(1b)  send REVISE_TS(M, i, tag, clock) to all processes;
(1c)  $temp\_ts \leftarrow 0$;
(1d)  await PROPOSED_TS(j, i, tag, $ts_j$) from each process $P_j$;
(1e)  $\forall j \in N$, do $temp\_ts \leftarrow \max(temp\_ts, ts_j)$;
(1f)  send FINAL_TS(i, tag, temp_ts) to all processes;
(1g)  $clock \leftarrow max(clock, temp\_ts)$.

(2)   When REVISE_TS(M, j, tag, clk) arrives from $P_j$:
(2a)  $priority \leftarrow max(priority + 1, clk)$;
(2b)  insert (M, tag, j, priority, undeliverable) in temp_Q;
                                                          // at end of queue
(2c)  send PROPOSED_TS(i, j, tag, priority) to $P_j$.

(3)   When FINAL_TS(j, x, clk) arrives from $P_j$:
(3a)  Identify entry Q_e in temp_Q, where $Q\_e.tag = x$;
(3b)  mark Q_e.deliverable as true;
(3c)  Update Q_e.timestamp to clk and re-sort temp_Q based on the
        timestamp field;
(3d)  if $(head(temp\_Q)).tag = Q\_e.tag$ then
(3e)     move Q_e from temp_Q to delivery_Q;
(3f)     while (head(temp_Q)).deliverable is true do
(3g)               dequeue head(temp_Q) and insert in delivery_Q.

(4)   When $P_i$ removes a message (M, tag, j, ts, deliverable) from
         $head(delivery\_Q_i)$:
(4a)  $clock \leftarrow \max(clock, ts) + 1$.
```

**Algorithm 6.5** A distributed algorithm to implement total order and causal order of messages. Code at $P_i$, $1 \leq i \leq n$.

i.e., any other messages received in the meanwhile are processed. Once all expected replies are received, the process computes the maximum of the proposed timestamps for $M$, and uses the maximum as the final timestamp.

**Phase 3** In the third phase, the process multicasts the final timestamp to the group in line (1f).

**Receivers**

**Phase 1** In the first phase, the receiver receives the message with a tentative/proposed timestamp. It updates the variable *priority* that tracks the highest proposed timestamp (line 2a), then revises the proposed timestamp to the *priority*, and places the message with its tag and the revised timestamp at the tail of the queue *temp_Q* (line 2b). In the queue, the entry is marked as undeliverable.

**Phase 2** In the second phase, the receiver sends the revised timestamp (and the tag) back to the sender (line 2c). The receiver then waits in a non-blocking manner for the final timestamp (correlated by the message tag).

**Phase 3** In the third phase, the final timestamp is received from the multicaster (line 3). The corresponding message entry in *temp_Q* is identified using the tag (line 3a), and is marked as deliverable (line 3b) after the revised timestamp is overwritten by the final timestamp (line 3c). The queue is then resorted using the timestamp field of the entries as the key (line 3c). As the queue is already sorted except for the modified entry for the message under consideration, that message entry has to be placed in its sorted position in the queue. If the message entry is at the head of the *temp_Q*, that entry, and all consecutive subsequent entries that are also marked as deliverable, are dequeued from *temp_Q*, and enqueued in *deliver_Q* in that order (the loop in lines 3d–3g).

*Complexity*

This algorithm uses three phases, and, to send a message to $n-1$ processes, it uses $3(n-1)$ messages and incurs a delay of three message hops.

**Example** An example execution to illustrate the algorithm is given in Figure 6.14. Here, A and B multicast to a set of destinations and C and D are the common destinations for both multicasts.

- **Figure 6.14(a)** The main sequence of steps is as follows:
  1. A sends a *REVISE_TS*(7) message, having timestamp 7. B sends a *REVISE_TS*(9) message, having timestamp 9.
  2. C receives A's *REVISE_TS*(7), enters the corresponding message in *temp_Q*, and marks it as undeliverable; *priority* = 7. C then sends *PROPOSED_TS*(7) message to A.

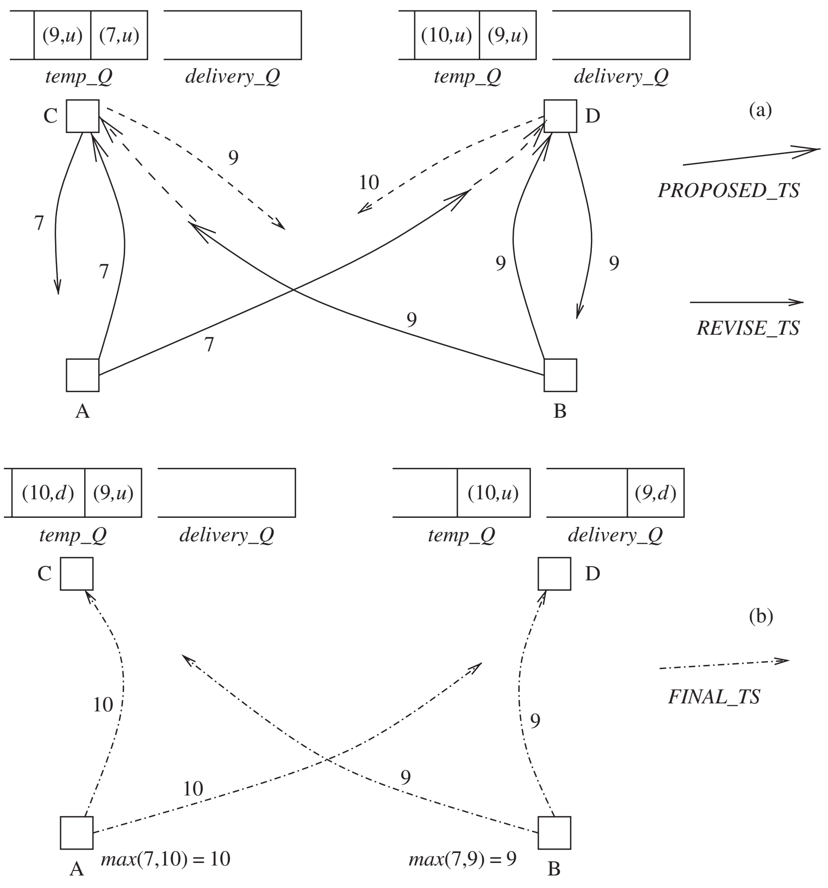


**Figure 6.14** An example to illustrate the three-phase total ordering algorithm. (a) A snapshot for *PROPOSED_TS* and *REVISE_TS* messages. The dashed lines show the further execution after the snapshot. (b) The *FINAL_TS* messages in the example.

3. D receives B's *REVISE_TS*(9), enters the corresponding message in *temp_Q*, and marks it as undeliverable; *priority* = 9. D then sends *PROPOSED_TS*(9) message to B.
4. C receives B's *REVISE_TS*(9), enters the corresponding message in *temp_Q*, and marks it as undeliverable; *priority* = 9. C then sends *PROPOSED_TS*(9) message to B.
5. D receives A's *REVISE_TS*(7), enters the corresponding message in *temp_Q*, and marks it as undeliverable; *priority* = 10. D assigns a tentative timestamp value of 10, which is greater than all of the timestamps on *REVISE_TS*s seen so far, and then sends *PROPOSED_TS*(10) message to A.

The state of the system is as shown in the figure.

- **Figure 6.14(b)** The continuing sequence of main steps is as follows:

6. When A receives *PROPOSED_TS*(7) from C and *PROPOSED_TS*(10) from D, it computes the final timestamp as $max(7, 10) = 10$, and sends *FINAL_TS*(10) to C and D.

7. When B receives *PROPOSED_TS*(9) from C and *PROPOSED_TS*(9) from D, it computes the final timestamp as $max(9,9) = 9$, and sends *FINAL_TS*(9) to C and D.
8. C receives *FINAL_TS*(10) from A, updates the corresponding entry in *temp_Q* with the timestamp, resorts the queue, and marks the message as deliverable. As the message is not at the head of the queue, and some entry ahead of it is still undeliverable, the message is not moved to *delivery_Q*.
9. D receives *FINAL_TS*(9) from B, updates the corresponding entry in *temp_Q* by marking the corresponding message as deliverable, and resorts the queue. As the message is at the head of the queue, it is moved to *delivery_Q*.

This is the system snapshot shown in Figure 6.14(b). The following further steps will occur:

10. When C receives *FINAL_TS*(9) from B, it will update the corresponding entry in *temp_Q* by marking the corresponding message as deliverable. As the message is at the head of the queue, it is moved to the *delivery_Q*, and the next message (of A), which is also deliverable, is also moved to the *delivery_Q*.
11. When D receives *FINAL_TS*(10) from A, it will update the corresponding entry in *temp_Q* by marking the corresponding message as deliverable. As the message is at the head of the queue, it is moved to the *delivery_Q*.

Algorithm 6.5 is closely structured along the lines of Lamport's algorithm for mutual exclusion. We will later see that Lamport's mutual exclusion algorithm has the property that when a process is at the head of its own queue and has received a REPLY from all other processes, the REQUEST of that process is at the head of all the queues. This can be exploited to deliver the message by all the processes in the same total order (instead of entering the critical section).

## 6.7 A nomenclature for multicast

In this section, we systematically classify the various kinds of multicast algorithms possible [9]. Observe that there are four classes of source–destination relationships, as illustrated in Figure 6.15, for open groups:

- **SSSG** Single source and single destination group.
- **MSSG** Multiple sources and single destination group.
- **SSMG** Single source and multiple, possibly overlapping, groups.
- **MSMG** Multiple sources and multiple, possibly overlapping, groups.

The SSSG and SSMG classes are straightforward to implement, assuming the presence of FIFO channels between each pair of processes. Both total

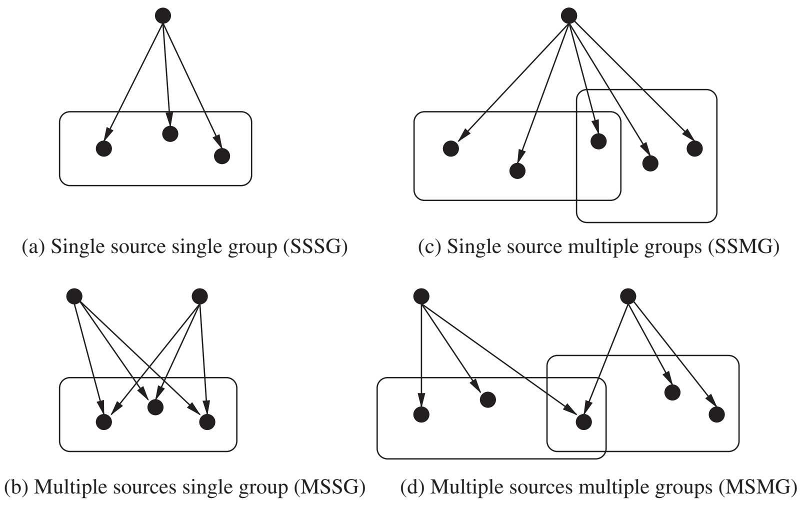

(a) Single source single group (SSSG)
(c) Single source multiple groups (SSMG)
(b) Multiple sources single group (MSSG)
(d) Multiple sources multiple groups (MSMG)

**Figure 6.15** Four classes of source–destination relationships for open-group multicasts. For closed-group multicasts, the sender needs to be part of the recipient group.

order and causal order are guaranteed. The MSSG class is also straightforward to handle; the centralized implementation in Algorithm 6.4 provides both total and causal order. The central coordinator effectively converts this class to the SSSG class.

We now consider a design approach for the MSMG class. This approach, commonly termed as the *propagation tree* approach, uses a semi-centralized structure that adapts the centralized algorithm of Algorithm 6.4 and was proposed by Chiu and Hsaio [9] and Jia [16].

## 6.8 Propagation trees for multicast

To manage the complications of delivery order across multiple overlapping groups $\mathcal{G} = \{G_1 \ldots G_g\}$, the algorithm first identifies a set of *metagroups* $\mathcal{MG} = \{MG_1, \ldots MG_h\}$ with the following properties: (i) each process belongs to a single metagroup, and has the exact same group membership as every other process in that metagroup; (ii) no other process outside that metagroup has that exact group membership.

**Example** Figure 6.16(a) shows some groups and their metagroups. $\langle ABC \rangle$, $\langle AB \rangle$, $\langle AC \rangle$, and $\langle A \rangle$ are the metagroups of user group $\langle A \rangle$.

The definition of metagroups transforms the problem of MSMG multicast to groups, to the problem of MSSG multicast to metagroups, which is easier to solve.

A distinguished node in each metagroup acts as the manager for that metagroup. For each user group $G_i$, one of its metagroups is chosen to be its *primary metagroup (PM)* and denoted as $PM(G_i)$. All the metagroups are

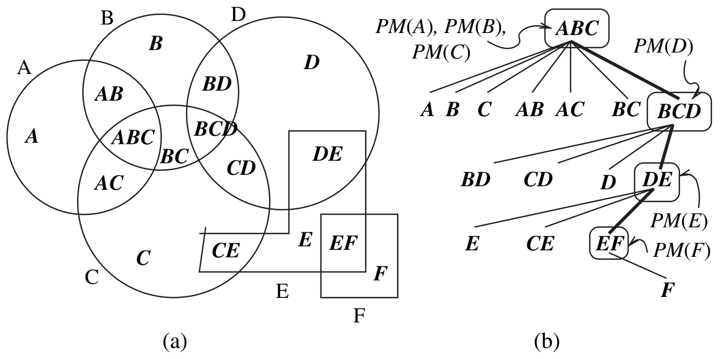


**Figure 6.16** Example illustrating a propagation tree [9]. Metagroups are shown in boldface. (a) Groups A, B, C, D, E, and F, and their metagroups. (b) A *propagation tree*, with the primary meta-groups labeled.

organized in a *propagation forest* or *tree* structure satisfying the following property: for user group $G_i$, its primary metagroup $PM(G_i)$ is at the lowest possible level (i.e., farthest from the root) of the tree such that all the metagroups whose destinations contain any nodes of $G_i$ belong to the subtree rooted at $PM(G_i)$.

**Example** In Figure 6.16, $\langle ABC \rangle$ is the primary metagroup of A, B, and C. $\langle B, C, D \rangle$ is the primary metagroup of D. $\langle D, E \rangle$ is the primary metagroup of E. $\langle E, F \rangle$ is the primary metagroup of F.

The following properties can be seen to be satisfied by the *propagation tree*:

1. The primary metagroup $PM(G)$, is the ancestor of all the other metagroups of $G$ in the propagation tree.
2. $PM(G)$ is uniquely defined.
3. For any metagroup MG, there is a unique path to it from the PM of any of the user groups of which the metagroup MG is a subset.
4. In addition, for any two primary metagroups $PM(G_1)$ and $PM(G_2)$, they should either lie on the same branch of a tree, or be in disjoint trees. In the latter case, their groups membership sets are necessarily disjoint.

## *Key idea*

The metagroup $PM(G_i)$ of user group $G_i$, is useful for multicasts, as follows: *multicasts to $G_i$ are sent first to the metagroup $PM(G_i)$ as only the subtree rooted at $PM(G_i)$ can contain the nodes in $G_i$*. The message is then propagated down the subtree rooted at $PM(G_i)$.

The following definitions are useful to understand and explain the algorithm:

- *$MG_1$ subsumes $MG_2$* (where $MG_1 \neq MG_2$) if for each group $G$ such that a member of $MG_2$ is a member of $G$, we have that some member of $MG_1$ is also a member of $G$. In other words, $MG_1$ is a subset of each user group $G$ of which $MG_2$ is a subset.

**Example** In Figure 6.16, $\langle AB \rangle$ subsumes $\langle A \rangle$. Any member of $MG_2 = \langle A \rangle$ is a member of $A$ and each member of $\langle AB \rangle$ is also a member of $A$. Similarly, $\langle AB \rangle$ subsumes $\langle B \rangle$.

- $MG_1$ *is joint with* $MG_2$ if neither metagroup subsumes the other and there is some group $G$ such that $MG_1, MG_2 \subset G$.

  **Example** In Figure 6.16, $\langle ABC \rangle$ is joint with $\langle CD \rangle$. Neither subsumes the other and both are a subset of $C$.

**Example** Figure 6.16 shows some groups, their metagroups, and their *propagation tree*. Metagroup $\langle ABC \rangle$ is the primary metagroup $PM(A)$, $PM(B)$, $PM(C)$. Meta-group $\langle BCD \rangle$ is the primary metagroup $PM(D)$. Thus, a multicast to group $D$ will be sent to $\langle BCD \rangle$.

We note that the propagation tree is not unique because it depends on the order in which metagroups are processed. Various optimizations on the propagation tree can also be performed, but we require that features (1)–(4) above should be satisfied by the tree. Exercise 6.10 asks you to design an algorithm to construct a propagation tree. A metagroup that has members from multiple user groups is desirable as the root in order to have a tree with low height.

### *Correctness*

The rules for forwarding messages during a multicast are given in Algorithm 6.6. Each process needs to know the propagation tree, computed at a central location. Each metagroup has a distinguished process which acts as the *manager* or representative of that metagroup.

The array $SV[1 \ldots h]$ kept by each process $P_i$ tracks in $SV[k]$, the number of messages multicast by $P_i$ that will traverse through primary metagroup $PM(G_k)$. This array is piggybacked on each message multicast by process $P_i$.

The manager of each primary metagroup keeps an array $RV[1 \ldots n]$ that tracks in $RV[k]$, the number of messages sent by process $P_k$ that have been received by this primary metagroup.

As in the CO algorithms, a message from $P_i$ can be processed by a primary metagroup $j$ if $RV_j[i] = SV_i[j]$; otherwise it buffers the message until this condition is satisfied (lines 2a–2c). At a non-primary metagroup, this check need not be performed because it never receives a message directly from the sender of the multicast. The multicast sender always sends the message to the primary metagroup first. At the non-primary metagroup, the relative order

```
(local variables)
integer: $SV[1 \ldots h]$;          //kept by each process. $h$ is #(primary
                                    //metagroups), $h \leq |\mathcal{G}|$
integer: $RV[1 \ldots n]$;          //kept by each primary metagroup manager.
                                    //$n$ is #(processes)
set of integers: $PM\_set$;         //set of primary metagroups through which
                                    //message must traverse

(1)  When process $P_i$ wants to multicast message $M$ to group $G$:
(1a) send $M(i, G, SV_i)$ to manager of $PM(G)$, primary metagroup of $G$;
(1b) $PM\_set \longleftarrow$ { primary metagroups through which $M$ must traverse };
(1c) for all $PM_x \in PM\_set$ do
(1d)         $SV_i[x] \longleftarrow SV_i[x] + 1$.

(2)  When $P_i$, the manager of a metagroup $MG$ receives $M(k, G, SV_k)$ from $P_j$:
       // Note: $P_i$ may not be a manager of any metagroup
(2a) if $MG$ is a primary metagroup then
(2b)         buffer the message until $(SV_k[i] = RV_i[k])$;
(2c)         $RV_i[k] \longleftarrow RV_i[k] + 1$;
(2d) for each child metagroup that is subsumed by $MG$ do
(2e)         send $M(k, G, SV_k)$ to the manager of that child metagroup;
(2f) if there are no child metagroups then
(2g)         send $M(k, G, SV_k)$ to each process in this metagroup.
```

**Algorithm 6.6** Protocol to enforce total and causal order using propagation trees.

of messages has already been determined by some ancestor metagroup; so it simply forwards the message as per lines 2d–2g.

- The logic behind why total order is maintained is straightforward. For any metagroups $MG_1$ and $MG_2$, and any groups $G_x$ and $G_y$ of which the metagroups are a subset, the primary metagroups $PM(G_x)$ and $PM(G_y)$ both subsume $MG_1$ and $MG_2$, and both lie on the same branch of the *propagation tree* to either $MG_1$ or $MG_2$. The primary metagroup that is lower in the tree will necessarily receive the two multicasts in some order. The assumption of FIFO channels guarantees that all processes in metagroups subsumed by this lower primary metagroup will receive the messages sent to the two groups in a common order.
- Causal order is guaranteed because of the check made by managers of the primary metagroups in lines 2a–2c. Assume that messages $M$ and $M'$ are multicast to $G$ and $G'$, respectively. For nodes in $G \cap G'$, there are two cases, as shown in Figure 6.17. In each case, the sequence numbers next to messages indicate the order in which the messages are sent.

**Case Figure 6.17(a) and (b):** Here, the senders of $M$ and $M'$ are different. $P_k$ sends $M$ to $G$. After $P_i \in G$ receives $M$, $P_i$ sends $M'$ to $G'$.

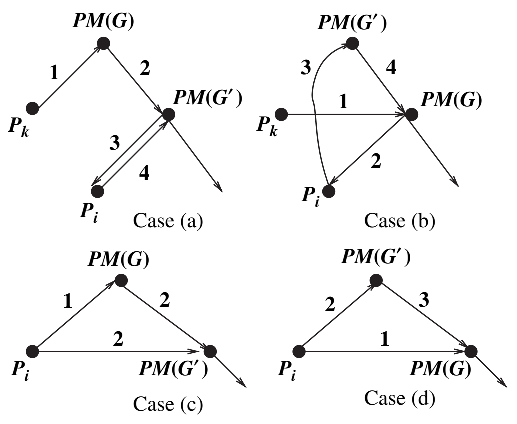


**Figure 6.17** The four cases for the correctness of causal ordering using *propagation trees*. The sequence numbers indicate the order in which the messages are sent.

Thus, we have the causal chain $Send_k(k, M, G)$, $Deliver_i(k, M, G)$, $Send_i(i, M', G')$. For any destination $MG_q$ such that $MG_q \subset G \cap G'$, the primary metagroup of $G$ and $G'$ must both be ancestors of the metagroup of $P_i$ because of the assumption of *closed groups*.

**Case (a):** $PM(G')$ will have already received and processed $M$ (flow 2) before it receives $M'$ (flow 4).

**Case (b):** $PM(G)$ will have already received and processed $M$ (flow 1) before it receives $M'$ (flow 4). Assuming FIFO channels, CO is guaranteed for all processes in $G \cap G'$.

**Case Figure 6.17(c) and (d):** $P_i$ sends $M$ to $G$ and then $P_i$ sends $M'$ to $G'$. Thus, we have the causal chain $Send_i(i, M, G)$, $Send_i(i, M', G')$.

**Case (c):** The check in lines 2a–2c by $PM(G')$ ensures that $PM(G')$ will not process $M'$ before it processes $M$.

**Case (d):** The check in lines 2a–2c by $PM(G)$ ensures that $PM(G)$ will not process $M'$ before it processes $M$. Assuming FIFO channels, CO is guaranteed for all processes in $G \cap G'$.

## 6.9 Classification of application-level multicast algorithms

We have seen some algorithmically challenging techniques in the design of multicast algorithms. The most general scenario allows each process to multicast to an arbitrary and dynamically changing group of processes at each step. As this generality incurs more overhead, algorithms implemented on real systems tend to be more "centralized" in one sense or another: Defago *et al.* give an exhaustive survey and this section is based on this survey [11]. For details of the various protocols, please refer to the survey. Many multicast protocols have been developed and deployed, but they can all be classified as belonging to one of the following five classes.

## Communication history-based algorithms

Algorithms in this class use a part of the communication history to guarantee ordering requirements.

The RST [22] and KS [20, 21] algorithms belong to this class, and provide only causal ordering. They do not need to track separate groups, and hence work for open-group multicasts.

Lamport's algorithm, wherein messages are assigned scalar timestamps and a process can deliver a message only when it knows that no other message with a lower timestamp can be multicast, also belongs to this class. The NewTop protocol [12], which extends Lamport's algorithm to overlapping groups, also guarantees both total and causal ordering. Both these algorithms use closed-group configurations.

## Privilege-based algorithms

The operation of such algorithms is illustrated in Figure 6.18(a). A token circulates among the sender processes. The token carries the sequence number for the next message to be multicast, and only the token-holder can multicast. After a multicast send event, the sequence number is updated. Destination processes deliver messages in the order of increasing sequence numbers. Senders need to know the other senders, hence closed groups are assumed. Such algorithms can provide total ordering, as well as causal ordering using a closed group configuration (see Exercise 6.12).

Examples of specific algorithms are On-Demand, and Totem. They differ in implementation details such as whether a token ring topology is assumed

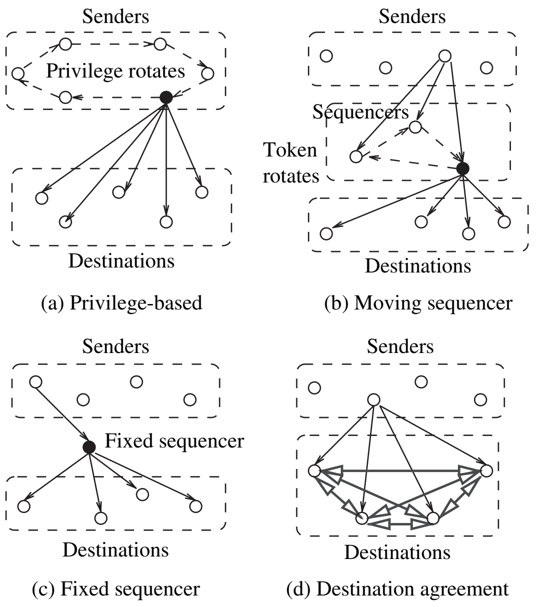


**Figure 6.18** Models for sequencing messages. (a) Privilege-based algorithms. (b) Moving sequencer algorithms. (c) Fixed sequencer algorithms. (d) Destination agreement algorithms.

(Totem) or not (On-Demand). Such algorithms are not scalable because they do not permit concurrent send events. Hence they are of limited use in large systems.

### Moving sequencer algorithms

The operation of such algorithms is illustrated in Figure 6.18(b). The original algorithm was proposed by Chang and Maxemchuck [8]; various variants of it were given by the Pinwheel and RMP algorithms. These algorithms work as follows. (1) To multicast a message, the sender sends the message to all the sequencers. (2) Sequencers circulate a token among themselves. The token carries a sequence number and a list of all the messages for which a sequence number has already been assigned – such messages have been sent already. (3) When a sequencer receives the token, it assigns a sequence number to all received but unsequenced messages. It then sends the newly sequenced messages to the destinations, inserts these messages in to the token list, and passes the token to the next sequencer. (4) Destination processes deliver the messages received in the order of increasing sequence number.

Moving sequencer algorithms guarantee total ordering.

### Fixed sequencer algorithms

The operation of such algorithms is illustrated in Figure 6.18(c). This class is a simplified version of the previous class. There is a single sequencer (unless a failure occurs), which makes this class of algorithms essentially centralized.

The propagation tree approach studied earlier, belongs to this class. Other algorithms are the ISIS sequencer, Amoeba, Phoenix, and Newtop's asymmetric algorithm. Let us look briefly at Newtop's asymmetric algorithm. All processes maintain logical clocks, and each group has an independent sequencer. The unicast from the sender to the sequencer, as well as the multicast from the sequencer are timestamped. A process that belongs to multiple groups must delay the sending of the next message (to the relevant sequencer) until it has received and processed all messages, from the various sequencers, corresponding to the previous messages it sent. Assuming FIFO channels, it can be shown that total order is maintained.

### Destination agreement algorithms

The operation of such algorithms is illustrated in Figure 6.18(d). In this class of algorithms, the destinations receive the messages with some limited ordering information. They then exchange information among themselves to define an order. There are two sub-classes here: (i) the first sub-class uses timestamps (Lamport's three-phase algorithm (Algorithm 6.5) belongs to this sub-class); (ii) the second sub-class uses an agreement or "consensus" protocol among the processes. We will study agreement protocols in Chapter 14.

## 6.10 Semantics of fault-tolerant group communication

A failure-free system can be assumed only in an ideal world. When a system component fails in the midst of the multicast operation, which is a non-atomic operation that spans across time and across multiple links and nodes, the behavior of a multicast protocol must adhere to a well-defined specification, and, correspondingly, the protocol must ensure that the specification under the failure mode is also implemented. This enables well-defined actions during recovery after the failure. This section is based on the results of Hadzilacos and Toueg [15]. Questions such as the following need to be addressed:

- For a multicast, if one correct process delivers the message $M$, what can be said about the other correct processes and faulty processes that also deliver $M$?
- For a multicast, if one faulty process delivers the message $M$, what can be said about the other correct processes and faulty processes that also deliver $M$?
- For causal or total order multicast, if one correct or faulty process delivers $M$, what can be said about other correct processes and faulty processes that also deliver $M$?

There are two broad flavors of the specifications. In the regular flavor, there are no conditions on the messages delivered to faulty processors (because they are faulty). However, assuming the benign failure model, under some conditions, it may be useful to specify and control the behavior of such faulty processes also. Therefore, the second flavor of specifications, termed as the *uniform* specifications, also states the expected behavior of faulty processes. In the following description of the specifications [15], the regular flavor and the uniform flavor are stated. To parse for the regular flavor, the parenthesized words should be omitted. To parse for the uniform flavor, the *italicized* and parenthesized modifiers to the definitions of the regular flavor are included.

(***Uniform***) Reliable multicast of $M$.

**Validity** If a correct process multicasts $M$, then all correct processes will eventually deliver $M$.

(***Uniform***) **agreement** If a correct (*or faulty*) process delivers $M$, then all correct processes will eventually deliver $M$.

(***Uniform***) **integrity** Every correct (*or faulty*) process delivers $M$ at most once, and only if $M$ was previously multicast by $sender(M)$.

The validity property states that once the multicast is initiated by a correct process, it will go to completion. The agreement property states that all correct processes get the same view of a message, irrespective of whether a correct process or a faulty process broadcasts it. The integrity property states that correct processes have non-duplicate delivery of messages, and that they are not delivered spurious messages. While the regular agreement property permits a faulty process to deliver a message that is never delivered to any correct process, this undesirable behavior can be problematic in applications such as atomic commit in database protocols, and is explicitly ruled out by uniform agreement. While the regular Integrity property permits a faulty process to deliver a message multiple times, and to deliver a message that was never sent, this behavior is explicitly ruled out by uniform integrity.

The orderings FIFO order, causal order, and total order are now defined for multicasts, in both the regular and uniform flavors. The uniform flavor requires that even faulty processes do not violate the ordering properties. These definitions of the regular and uniform flavors are superimposed on the basic definition of a (uniform) reliable multicast, given above. The regular flavor and the uniform flavor of each definition is read using the semantics above for parsing the corresponding flavors of multicast. In these definitions which deal with the relative order of messages, it is important that the multicast groups are identical, in which case the messages get broadcast within the common group.

(***Uniform***) **FIFO order** If a process broadcasts $M$ before it broadcasts $M'$, then no correct (*or faulty*) process delivers $M'$ unless it previously delivered $M$.

(***Uniform***) **causal order** If $M$ is broadcast causally before $M'$ is broadcast, then no correct (*or faulty*) process delivers $M'$ unless it previously delivered $M$.

(***Uniform***) **total order** If correct (*or faulty*) processes $a$ and $b$ both deliver $M$ and $M'$, then $a$ delivers $M$ before $M'$ if and only if $b$ delivers $M$ before $M'$.

It is time to remember the folklore result that any protocol or implementation that deals with fault-tolerance incurs a greater cost than what it would in a failure-free environment. In some case, this extra cost can be substantial. Nevertheless, it is important to formally specify the behavior in the face of faults, and to provide the implementations that can realize such behavior. We will not deal with implementations of the above fault-tolerant specifications of multicasts.

Excessive delay in delivering a multicast message can also be viewed as a fault. Applications with real-time constraints require that if a message is delivered, it should be within a bounded period $\Delta$, termed the latency, after it was multicast. This specification can be based on either a global observer's notion of time, or the local time at each process, leading to real-time $\Delta$-timeliness and local-time $\Delta$-timeliness, respectively:

(***Uniform***) **real-time Δ-timeliness** For some known constant $\Delta$, if $M$ is multicast at real-time $t$, then no correct (*or faulty*) process delivers $M$ after real-time $t + \Delta$.

(***Uniform***) **local $\Delta$-timeliness** For some known constant $\Delta$, if $M$ is multicast at local time $t_m$, then no correct (*or faulty*) process $i$ delivers $M$ after its local time $t_m + \Delta$ on $i$'s clock.

Specifying local-time $\Delta$-timeliness requires care because the local clocks at processes can vary. It is assumed that the sender timestamps the message multicast with its local time $t_m$, and any receiver should receive the message within $t_m + \Delta$ on its local clock. The efficacy of this specification depends on how closely the local clocks are synchronized. A protocol to synchronize physical clocks was studied in Chapter 3.

## 6.11 Distributed multicast algorithms at the network layer

Several applications can interface directly with the network layer and the lower hardware-related layers to exploit the physical connectivity and the physical topology for group communication. The network is viewed as a graph $(N, L)$, and various graph algorithms – centralized or distributed – are run to establish and maintain efficient routing structures. For example,

- LANs connected by bridges maintain spanning trees for distributing information and for forward/backward learning of destinations;
- the network layer of the Internet has a rich suite of multicast algorithms.

In this section, we will study the principles underlying several such algorithms. Some of the algorithms in this section may not be distributed. Nevertheless, they are intended for a distributed setting, namely the LAN or the WAN.

### 6.11.1 Reverse path forwarding (RPF) for constrained flooding

As studied in Chapter 5, broadcasting data using flooding in a network $(N, L)$ requires up to $2|L|$ messages. Reverse path forwarding (RPF) is a simple but elegant technique that brings down the overhead significantly at very little cost. Network nodes are assumed to run the distance vector routing (DVR) algorithm (Chapter 5), which was used in the Internet until 1983. (Since 1983, the LSR-based algorithms described in Chapter 5 have been used. These are more sophisticated and provide more information than that required by DVR.)

The simple DVR algorithm assumes that each node knows the next hop on the path to each destination $x$. This path is assumed to be the approximation to the "best" path. Let $Next\_hop(x)$ denote the function that gives the next hop on the "best" path to $x$. The RPF algorithm leverages the DVR algorithm for point-to-point routing, to achieve constrained flooding. The RPF algorithm for constrained flooding is shown in Algorithm 6.7.

```
(1)   When process $P_i$ wants to multicast message $M$ to group $Dests$:
(1a)  send $M(i, Dests)$ on all outgoing links.

(2)   When a node $i$ receives message $M(x, Dests)$ from node $j$:
(2a)  if $Next\_hop(x) = j$ then      // this will necessarily be a new message
(2b)      forward $M(x, Dests)$ on all other incident links besides $(i, j)$;
(2c)  else ignore the message.
```

**Algorithm 6.7** Reverse path forwarding (RPF).

This simple RPF algorithm has been experimentally shown to be effective in bringing the number of messages for a multicast closer to $|N|$ than to $|L|$. Actually, the algorithm does a broadcast to all the nodes, and this broadcast is smartly curtailed to approximate a spanning tree. The curtailed broadcast is effective because, implicitly, an approximation to a tree rooted at the source is identified, without it being computed or stored at any node.

*Pruning* of the implicit broadcast tree can be used to deal with unwanted multicast packets. If a node receives the packets but the application running on it does not need the packets, and all "downstream" (in the implicit tree) nodes also do not need the packets, the node can send a *prune* message to the parent in the tree indicating that packets should not be forwarded on that edge. Implementing this in a dynamic network where the tree periodically changes and the application's node membership also changes dynamically is somewhat tricky (see Exercise 6.14).

### 6.11.2 Steiner trees

The problem of finding an optimal "spanning" tree that spans only all nodes participating in a multicast group, known as the *Steiner tree problem*, is formalized as follows.

#### Steiner tree problem

Given a weighted graph $(N, L)$ and a subset $N' \subseteq N$, identify a subset $L' \subseteq L$ such that $(N', L')$ is a subgraph of $(N, L)$ that connects all the nodes of $N'$.

A *minimal Steiner tree* is a minimal-weight subgraph $(N', L')$. The minimal Steiner tree problem has been well-studied and is known to be NP-complete. When the link weights change, the tree has to be recomputed to obtain the new minimal Steiner tree, making it even more difficult to use in dynamic networks.

Several heuristics have been proposed to construct an approximation to the minimal Steiner tree. A simple heuristic constructs a MST, and deletes edges that are not necessary. This algorithm is given by the first three steps of Algorithm 6.8. The worst case cost of this heuristic is twice the cost of the optimal solution. Algorithm 6.8 can show better performance when using the heuristic by Kou *et al.* [19], given by steps 4 and 5 in the algorithm.

The resulting Steiner tree cost is also at most twice the cost of the minimal Steiner tree, but behaves better on average.

---

Input: weighted graph $G = (N, L)$, and $N' \subseteq N$, where $N'$ is the set of Steiner points

(1) Construct the complete undirected distance graph $G' = (N', L')$ as follows:
$L' = \{(v_i, v_j) \mid v_i, v_j \text{ in } N'\}$, and $wt(v_i, v_j)$ is the length of the shortest path from $v_i$ to $v_j$ in $(N, L)$.
(2) Let $T'$ be the minimal spanning tree of $G'$. If there are multiple minimum spanning trees, select one randomly.
(3) Construct a subgraph $G_s$ of $G$ by replacing each edge of the MST $T'$ of $G'$, by its corresponding shortest path in $G$. If there are multiple shortest paths, select one randomly.
(4) Find the minimum spanning tree $T_s$ of $G_s$. If there are multiple minimum spanning trees, select one randomly.
(5) Using $T_s$, delete edges as necessary so that all the leaves are the Steiner points $N'$. The resulting tree, $T_{Steiner}$, is the heuristic's solution.

---

**Algorithm 6.8** The Kou–Markowsky–Berman heuristic for a minimum Steiner tree.

*Cost* The time complexity of the heuristic algorithm for each of the five steps is as follows: step 1: $O(|N'| \cdot |N|^2)$; step 2: $O(|N'|^2)$; step 3: $O(|N|)$; step 4: $O(|N|^2)$; step 5: $O(|N|)$. Step 1 dominates, hence the time complexity is $O(|N'| \cdot |N|^2)$.

### 6.11.3 Multicast cost functions

Consider a source node $s$ that has to do a multicast to Steiner nodes. As before, we are given the weighted graph $(N, L)$ and the Steiner node set $N'$. We can define several cost functions [3]. For example, let $cost(i)$ be the cost of the path from $s$ to $i$ in the routing scheme $R$.

The *destination cost of* $R$ is defined as $\frac{1}{|N'|} \sum_{i \in N'} cost(i)$. This represents the average cost of the routing. If the cost is measured in time delay, this routing function metric gives the shortest average time for the multicast to reach nodes in $N'$.

As a variant, a link is counted only once even if it is used on the minimum cost path to multiple destinations. This variant reduces to the Steiner tree problem of Section 6.11.2. The sum of the costs of the edges in the Steiner tree routing scheme $R$ is defined as the *network cost*.

### 6.11.4 Delay-bounded Steiner trees

Multimedia networks and interactive applications have given rise to the need for a minimum Steiner tree that also satisfies delay constraints on the transmission. Thus now, the goal is not only to minimize the cost of the tree (measured in terms of a parameter such as the link weight, which models the available bandwidth or a similar cost measure) but also to minimize the delay (propagation delay). The problem is formalized as follows.

#### Delay-bounded minimal Steiner tree problem

Given a weighted graph $(N, L)$, there are two weight functions $\mathcal{C}(l)$ and $\mathcal{D}(l)$ for each edge in $L$. $\mathcal{C}(l)$ is a positive real cost function on $l \in L$ and $\mathcal{D}(l)$ is a positive integer delay function on $l \in L$. For a given delay tolerance $\Delta$, a given source $s$ and a destination set $Dest$, where $\{s\} \cup Dest = N' \subseteq N$, identify a spanning tree $T$ covering all the nodes in $N'$, subject to the constraints below. Here, we let $path(s, v)$ denote the path from $s$ to $v$ in $T$.

- $\sum_{l \in T} \mathcal{C}(l)$ is minimized, subject to
- $\forall v \in N'$, $\sum_{l \in path(s,v)} \mathcal{D}(l) < \Delta$.

Finding such a minimal Steiner tree, subject to another parameter, is at least as difficult as finding a Steiner tree. It can be shown that this problem reduces to the Steiner tree problem. A detailed study of two heuristics to solve this problem is presented by Kompella *et al.* [18]. A *constrained cheapest path* between $x$ and $y$ is the cheapest path between $x$ and $y$ that has delay less than $\Delta$. The cost and delay on such a path are denoted by $\mathcal{C}(x, y)$ and $\mathcal{D}(x, y)$, respectively. If two or more paths have the lowest cost, the lowest delay path is chosen. The steps to compute the constrained Steiner tree are shown in Algorithm 6.9. Step 1 computes the complete closure graph $G'$ on nodes in $N'$. The two heuristics given below are used in Step 2 to greedily build a constrained Steiner tree on $G'$. Step 3 expands the tree edges in $G'$ to their original paths in $G$. An example of a constrained Steiner tree for the input graph in Figure 6.19(a) is given in Figure 6.19(b).

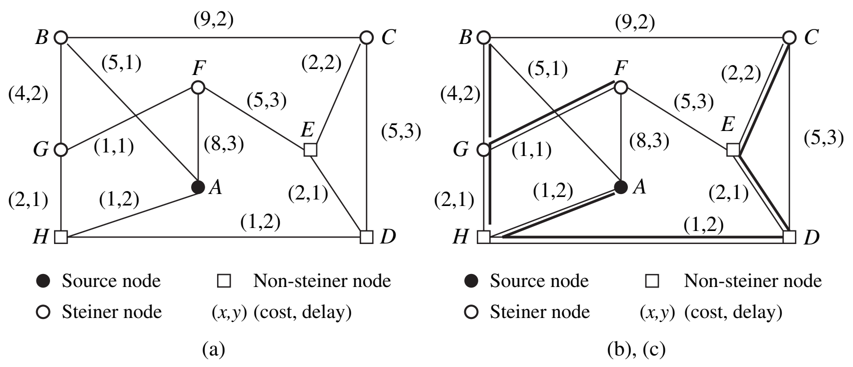


**Figure 6.19** Constrained Steiner tree example [18]. (a) Network graph. (b) and (c) MST and Steiner tree (optimal) are the same and shown in thick lines.

$\mathcal{C}(l)$ // cost of edge $l$
$\mathcal{D}(l)$ // delay of edge $l$
$T$; // constrained spanning tree to be constructed
$\mathcal{P}_C(x, y)$; // cost of constrained cheapest path from $x$ to $y$
$\mathcal{P}_D(x, y)$; // delay on constrained cheapest path from $x$ to $y$
$\mathcal{C}_d(x, y)$; // cost of the cheapest path with delay exactly $d$

Input: weighted graph $G = (N, L)$, and $N' \subseteq N$, where $N'$ is the set of Steiner points, source is $s$, and $\Delta$ is the constraint on the delay.

1. Compute the closure graph $G'$ on $(N', L)$, to be the complete graph on $N'$. The closure graph is computed using the all-pairs constrained cheapest paths using a dynamic programming approach analogous to Floyd's algorithm. For any pair of nodes $x, y \in N'$:

   - $\mathcal{P}_C(x, y) = min_{d<\Delta}\mathcal{C}_d(x, y)$. This selects the cheapest constrained path, satisfying the condition of $\Delta$, among the various paths possible between $x$ and $y$. The various $\mathcal{C}_d(x, y)$ can be calculated using DP as follows:
   - $\mathcal{C}_d(x, y) = min_{z\in N}\{\mathcal{C}_{d-\mathcal{D}(z,y)}(x, z) + \mathcal{C}(z, y)\}$. For a candidate path from $x$ to $y$ passing through $z$, the path with weight exactly $d$ must have a delay of $d - \mathcal{D}(z, y)$ for $x$ to $z$ when the edge $(z, y)$ has delay $\mathcal{D}(z, y)$.

   In this manner, the complete closure graph $G'$ is computed. $\mathcal{P}_D(x, y)$ is the delay on the constrained cheapest path that corresponds to a cost of $\mathcal{P}_C(x, y)$.
2. Construct a constrained spanning tree of $G'$ using a greedy approach that sequentially adds edges to the subtree of the constrained spanning tree $T$ (thus far) until all the Steiner points are included. The initial value of $T$ is the singleton $s$. Consider that node $u$ is in the tree and we are considering whether to add edge $(u, v)$.
   The following two edge selection criteria (heuristics) can be used to decide whether to include edge $(u, v)$ in the tree:

   - $CST_{CD}$: $f_{CD}(u, v) = \begin{cases} \dfrac{\mathcal{C}(u, v)}{\Delta - (\mathcal{P}_D(s, u) + \mathcal{D}(u, v))}, & \text{if } \mathcal{P}_D(s, u) + \mathcal{D}(u, v) < \Delta \\ \infty, & \text{otherwise.} \end{cases}$

     The numerator is the "incremental cost" of adding $(u, v)$ and the denominator is the "residual delay" that could be afforded. The goal is to minimize the incremental cost, while also maximizing the residual delay by choosing an edge that has low delay. Thus, the heuristic picks the neighbor $v$ that minimizes $f_{CD}$, for all $u$ in $T$ and all $v$ adjacent to $T$.
   - $CST_C$: $f_c = \begin{cases} \mathcal{C}(u, v), & \text{if } \mathcal{P}_D(s, u) + \mathcal{D}(u, v) < \Delta \\ \infty, & \text{otherwise.} \end{cases}$

     This heuristic picks the lowest cost edge between the already included tree edges and their nearest neighbor, as long as the total delay is less than $\Delta$.

   The chosen node $v$ is included in $T$. This step 2 is repeated until $T$ includes all $|N'|$ nodes in $G'$.
3. Expand the edges of the constrained spanning tree $T$ on $G'$ into the constrained cheapest paths they represent in the original graph $G$. Delete/break any loops introduced by this expansion.

**Algorithm 6.9** The constrained minimum Steiner tree algorithm using the $CST_{CD}$ and $CST_C$ heuristics.

- **Heuristic** $CST_{CD}$ This heuristic tries to choose low-cost edges, while also trying to pick edges that maximize the remaining allowable delay. The motivation is to try to reduce the tree cost by path sharing, by extending the path beyond the selected edge. This heuristic has the tendency to optimize on delay also, while adding to the cost.
- **Heuristic** $CST_C$ This heuristic simply minimizes the cost while ensuring that the delay bound is met.

*Complexity* Assuming integer-valued $\Delta$, step 1, which finds the constrained cheapest shortest paths over all the nodes, has $O(n^3\Delta)$ time complexity. This is because all pairs of end and intermediate nodes have to be examined, for all integer delay values from 1 to $\Delta$. Step 2, which constructs the constrained MST on the closure graph having $k$ nodes, has $O(k^3)$ time complexity. Step 3, which expands the constrained spanning tree, involves expanding the $k$ edges to up to $n-1$ edges each and then eliminating loops. This has $O(kn)$ time overhead. The dominating step is step 1.

### 6.11.5 Core-based trees

In the core-based tree approach, each group has a center node, or *core* node. A multicast tree is constructed dynamically, and grows on-demand, as follows. (i) A node wishing to join the tree as a receiver sends a unicast "join" message to the core node. (ii) The join message marks the edges as it travels; it either reaches the core node, or some node which is already a part of the multicast tree. The path followed by the "join" message from its source till the core/multicast tree is grafted to the multicast tree, and defines the path to the "core." (iii) A node on the tree multicasts a message by using a flooding on the core tree. (iv) A node not on the tree sends a message towards the core node; as soon as the message reaches any node on the tree, the message is flooded on the tree. In a network with a dynamically changing topology, care needs to be taken to maintain the tree structure and prevent messages from looping. This problem also exists for normal routing algorithms, such as the LSR and DVR algorithms (Chapter 5), in dynamic networks.

Current systems do not widely implement the Steiner tree for group multicast, even though it is more efficient after the initial cost to construct the Steiner tree. They prefer the simpler core-based tree (CBT) approach.

Core-based trees have various variants. A multi-core-based tree has more than one core node. For all CBT algorithms, high-bandwidth links can be specially chosen over others for forming the tree. Core-based trees have a natural analog in wireless networks, wherein it is reasonable to constitute the core tree of high-bandwidth wired links or high-power wireless links.

## 6.12 Chapter summary

At the core of distributed computing is the communication by message-passing among the processes participating in the application. This chapter studied several message ordering paradigms for communication, such as synchronous, FIFO, causally ordered, and non-FIFO orderings. These orders form a hierarchy. The chapter then examined several algorithms to implement these orderings. Group communication is an important aspect of communication in distributed systems. Causal order and total order are the popular forms of ordering when doing group multicasts and broadcasts. Algorithms to implement these orderings in group communication were also studied.

Maintaining communication in the presence of faults is necessary in real-world systems. Faults and their impacts are unpredictable. However, the behavior in the presence of faults needs to be clearly specified so that the application knows what to expect in terms of message delivery and message ordering in the presence of potential faults. The chapter studied some formal specifications of the expected behavior of group communication when faults might occur.

This chapter also studied some distributed multicast algorithms at the network layer. These algorithms include reverse path forwarding, multicast along Steiner trees and delay-bounded Steiner trees, and multicast based on core-based trees over the network graph. The solutions to some of these problems are NP-complete. Hence, only heuristics for polynomial time solutions are examined assuming a centralized setting to perform the computation.

## 6.13 Exercises

**Exercise 6.1** (Characterizing causal ordering)

1. Prove that the CO property (Definition 6.3) and the message order property (Definition 6.5) characterize an identical class of executions.
2. Prove that the CO property (Definition 6.3) and the empty interval property (Definition 6.6) characterize an identical class of executions.

**Exercise 6.2** Draw the directed graph $(\mathcal{T}, \hookrightarrow)$ for each of the executions in Figures 6.2, 6.3, and 6.5.

**Exercise 6.3** Give a linear time algorithm to determine whether an A-execution $(E, \prec)$ is RSC.
Hint: Use the definition of a crown and perform a topological sort on the messages using the $\hookrightarrow$ relation.

**Exercise 6.4** Show that a non-CO execution must have a crown of size 2.

**Exercise 6.5** Synchronous systems were defined in Chapter 5. Synchronous send and receive primitives were also introduced in Chapter 1. Synchronous executions were defined formally in Definition 6.8.

These concepts are closely related. Explain carefully the differences and relationships between: (i) a synchronous execution, (ii) an (asynchronous) execution that uses synchronous communication, and (iii) a synchronous system.

**Exercise 6.6** Rewrite the spanning tree algorithm of Figure 5.3 using CSP-like notation. You can assume a wildcard operator in a receive call to specify that any sender can be matched.

**Exercise 6.7** The algorithm to implement synchronous order by scheduling messages, as given in Algorithm 6.1, uses process identifiers to break cyclic waits.

1. Analyze the fairness of this algorithm.
2. If the algorithm is not fair, suggest some ways to make it fair.
3. Will the use of rotating logical identifiers increase the fairness of the algorithm?

**Exercise 6.8** Show the following containment relationships between causally ordered and totally ordered multicasts (hint: you may use Figure 6.11):

1. Show that a causally ordered multicast need not be a total order multicast.
2. Show that a total order multicast need not be a causal order multicast.

**Exercise 6.9** Assume that all messages are being broadcast. Justify your answers to each of the following:

1. Modify the causal message ordering algorithm (Algorithm 6.2) so that processes use only two vectors of size $n$, rather than the $n \times n$ array.
2. Is it possible to implement total order using a vector of size $n$?
3. Is it possible to implement total order using a vector of size $O(1)$?
4. Is it possible to implement causal order using a vector of size $O(1)$?

**Exercise 6.10** Design a (centralized) algorithm to create a propagation tree satisfying the properties given in Section 6.8.

**Exercise 6.11** For the multicast algorithm based on propagation trees, answer the following:

1. What is a tight upper bound on the number of multicast groups?
2. What is a tight upper bound on the number of metagroups of the multicast groups?
3. Examine and justify in detail, the impact (to the propagation tree) of (i) an existing process departing from one of the multiple groups of which it is a member; (ii) an existing process joining another group; (iii) the formation of a new group containing new processes; (iv) the formation of a new group containing processes that are already part of various other groups.

**Exercise 6.12** For multicast algorithms, show the following.

1. Privilege-based multicast algorithms provide (i) causal ordering if closed groups are assumed, and (ii) total ordering.

2. Moving sequencer algorithms, which work with open groups, provide total ordering.
3. Fixed sequencer algorithms provide total ordering.

**Exercise 6.13** In the example of Figure 6.16, draw the propagation tree that would result if $\langle CE \rangle$ were considered before $\langle BCD \rangle$ as a child of $\langle ABC \rangle$.

**Exercise 6.14** Consider the reverse path forwarding algorithm (Algorithm 6.7) for doing a multicast.

1. Modify the code to perform *pruning* of the multicast tree.
2. Now modify the code of (1) to also deal with dynamic changes to the network topology (use the algorithms in Chapter 5).
3. Now modify the code to deal with dynamic changes in the membership of the application at the various nodes.

**Exercise 6.15** Give a (centralized) algorithm for creating a propagation tree, for any set of groups.

**Exercise 6.16** Prove that the propagation tree for a given set of groups is not unique.

**Exercise 6.17** For the graph in Figure 6.19, compute the following spanning trees:

1. Steiner tree (based on the KMB heuristic).
2. Delay-bounded Steiner (heuristic $CST_{CD}$), with a delay bound of 8 units.
3. Delay-bounded Steiner (heuristic $CST_C$), with a delay bound of 8 units.

**Exercise 6.18** Design a graph for which the $CST_{CD}$ and $CST_C$ heuristics yield different delay-bounded Steiner trees.

**Exercise 6.19** The algorithms for creating the propagation tree, the Steiner tree, and the delay-bounded Steiner tree are centralized. Identify the exact challenges in making these algorithms distributed.

## 6.14 Notes on references

The discussion on synchronous, asynchronous, and RSC-executions is based on Charron-Bost *et al.* [7]. The CSP language for synchronous communication was first proposed and formalized by Hoare [16]. The discussion on implementing synchronous order is based on Bagrodia [1]. The discussion on the group communication paradigm, as well as on total order and causal order is based on Birman and Joseph [4,5]. The algorithm for causal order (Algorithm 6.2) is given by Raynal *et al.* [22]. The space and time optimal algorithm for causal order is given by Kshemkalyani and Singhal [20,21]. The example to illustrate this algorithm is taken from [6]. The algorithm for total order (Algorithm 6.5) is taken from the ISIS project by Birman and Joseph [4,5]. The algorithm for total order using propagation trees is based on Garcia-Molina and Spauster [13], Jia [17], and Chiu and Hsiao [9]. The classification of application-level multicast algorithms was given by Defago *et al.* [11]. The moving sequencer algorithms were proposed by Chang and Maxemchuk [8]. An efficient fault-tolerant group communication protcol is given in [12]. A comprehensive survey of group communication specifications given by Chockler *et al.* [10] as well as the survey in [11] discuss the systems Totem, Pinwheel, RMP, On-Demand, Isis, Amoeba, Phoenix, and Newtop. The Steiner tree problem was named after

Steiner and developed in [14]. The Steiner tree heuristic discussed was proposed by Kou *et al.* [19]. The network cost and destination cost metrics were introduced by [3]. They further showed a detailed analysis of the bounds on the metrics. The discussion on the delay-bounded minimum Steiner tree is based on Kompella *et al.* [18]. The discussion on the semantics of fault-tolerant group communication is given by Hadzilacos and Toueg [15]. Core-based trees were proposed by Ballardie *et al.* [2].

## References

[1] R. Bagrodia, Synchronization of asynchronous processes in CSP, *ACM Transactions in Programming Languages and Systems*, **11**(4), 1989, 585–597.

[2] T. Ballardie, P. Francis, and J. Crowcroft, Core based trees (CBT), *ACM SIGCOMM Computer Communication Review*, **23**(4), 1993, 85–95.

[3] K. Bharath-Kumar and J. Jaffe, Routing to multiple destinations in computer networks, *IEEE Transactions on Communications*, **31**(3) 1983, 343–351.

[4] K. Birman and T. Joseph, Reliable communication in the presence of failures, *ACM Transactions on Computer Systems*, **5**(1), 1987, 47–76.

[5] K. Birman, A. Schiper, and P. Stephenson, Lightweight causal and atomic group multicast, *ACM Transactions on Computer Systems*, **9**(3), 1991, 272–314.

[6] P. Chandra, P. Gambhire, and A. D. Kshemkalyani, Performance of the optimal causal multicast algorithm: a statistical analysis, *IEEE Transactions on Parallel and Distributed Systems*, **15**(1), 2004, 40–52.

[7] B. Charron-Bost, G. Tel, and F. Mattern, Synchronous, asynchronous, and causally ordered communication, *Distributed Computing*, **9**(4), 1996, 173–191.

[8] J.-M. Chang and N. Maxemchuk, Reliable broadcast protocols, *ACM Transactions on Computer Systems*, **2**(3), 1984, 251–273.

[9] G.-M. Chiu and C.-M. Hsiao, A note on total ordering multicast using propagation trees, *IEEE Transactions on Parallel and Distributed Systems*, **9**(2), 1998, 217–223.

[10] G. Chockler, I. Keidar, and R. Vitenberg, Group communication specifications: a comprehensive study, *ACM Computing Surveys*, **33**(4), 2001, 1–43.

[11] X. Defago, A. Schiper, and P. Urban, Total order broadcast and multicast algorithms: taxonomy and survey, *ACM Computing Surveys*, **36**(4), 2004, 372–421.

[12] P. Ezhilchelvan, R. Macdo, and S. Shrivastava, Newtop: a fault-tolerant group communication protocol**,** *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, Vancouver, Canada, May, 1995, 296–306.

[13] H. Garcia-Molina and A. Spauster, Ordered and reliable multicast communication, *ACM Transactions on Computer Systems*, **9**(3), 1991, 242–271.

[14] E. Gilbert and H. Pollack, Steiner minimal trees, *SIAM Journal of Applied Mathematics*, **16**(1), 1968, 1–29.

[15] V. Hadzilacos and S. Toueg, Fault-tolerant broadcasts and related problems in Mullender, S. (ed.), *Distributed Systems*, New York, Addison-Wesley, 1993, 97–146.

[16] C. A. R. Hoare, Communicating sequential processes, *Communications of the ACM*, **21**(8), 1978, 666–677.

[17] X. Jia, A total ordering multicast protocol using propagation trees, *IEEE Transactions on Parallel and Distributed Systems*, **6**(6), 1995, 617–627.

[18] V. Kompella, J. Pasquale, and G. Polyzos, Multcast routing for multimedia communication, *IEEE/ACM Transactions on Networking*, **1**(3), 1993, 86–92.

[19] L. Kou, G. Markowsky, and L. Berman, A fast algorithm for Steiner trees, *Acta Informatica*, **15**, 1981, 141–145.

[20] A. D. Kshemkalyani and M. Singhal, An optimal algorithm for generalized causal message ordering, *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, May 1996, 87.

[21] A. D. Kshemkalyani and M. Singhal, Necessary and sufficient conditions on information for causal message ordering and their optimal implementation, *Distributed Computing*, **11**(2), 1998, 91–111.

[22] M. Raynal, A. Schiper, and S. Toueg, The causal ordering abstraction and a simple way to implement it, *Information Processing Letters*, **39**, 1991, 343–350.