$M_{4,3}$ 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 Thus, P_1 also "learns" that the explicit piggybacked information "$M_{5,1}.Dests = \{P_4\}$" is outdated.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the information "$P_6 \in M_{5,1}.Dests$" piggybacked on $M_{2,3}$, which arrives at P_1, is inferred to be outdated (and hence ignored) using the *implicit* knowledge derived from "$M_{5,1}.Dests = \emptyset$" in Log_1.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.

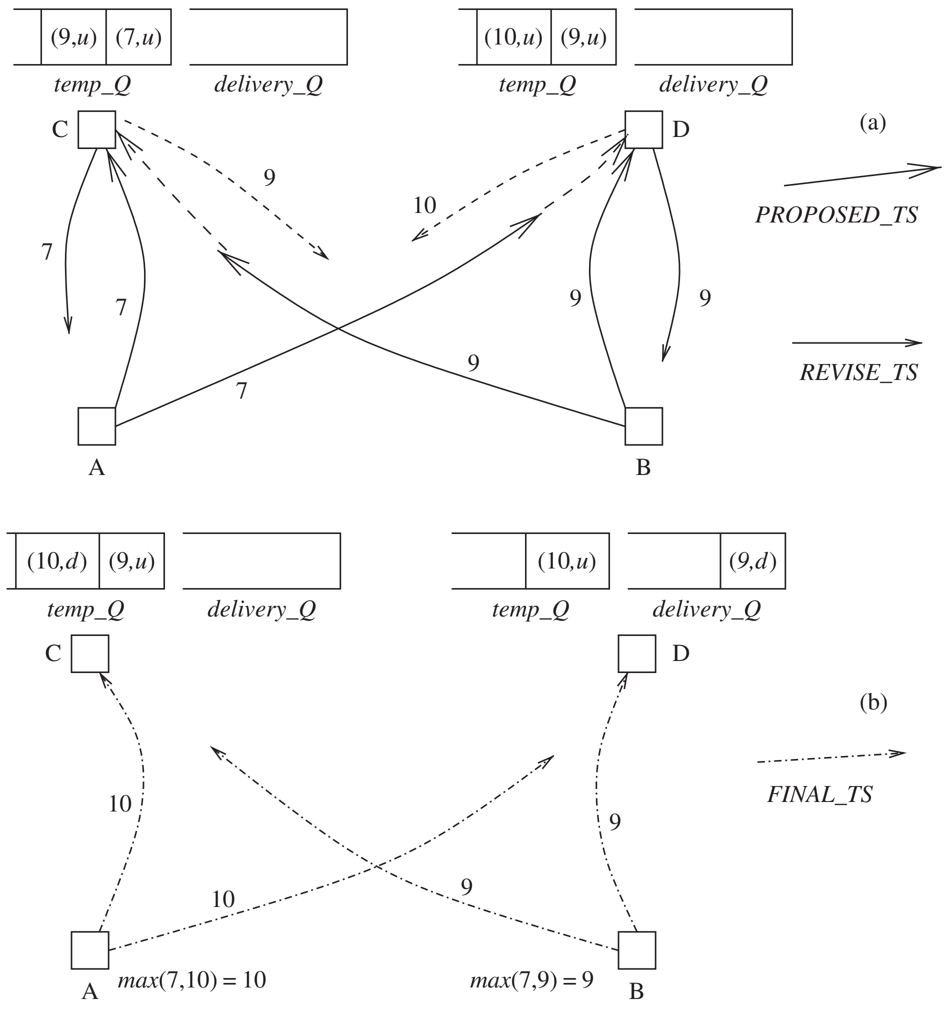

Figure 6.14 An example to illustrate the three-phase total ordering algorithm. (a) A snapshot for *PROPOSED_TS* and *REVISE_TS* messages. The dashed lines show the further execution after the snapshot. (b) The *FINAL_TS* messages in the example.

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