

Lecture 16

Message Ordering and Group Communication



Rajiv Misra

Dept. of Computer Science &
Engineering

Indian Institute of Technology Patna

rajivm@iitp.ac.in

Preface

Recap of Previous Lecture:

- In the previous lecture, we have discussed about '**Termination Detection**' and set of representative termination detection algorithms based on the concepts of **snapshot collection, weight throwing and spanning-tree etc.**

Content of this Lecture:

- In this lecture, we will discuss about:
 - (i) Message ordering
 - (ii) Group communication and
 - (iii) Multicasts

Introduction

- At the core of distributed computing is the communication by message passing among the processes participating in the application.
- In this lecture, we will study **several message ordering paradigms** for communication, such as asynchronous, synchronous, FIFO, causally ordered, and non-FIFO orderings. These orders form a hierarchy. Then we will examine few 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 will also be discussed.

Notations

- Network (N, L) ; event set $(E, \prec)$
- Message m^i : send and receive events s^i and r^i
- Send and receive events: s and r
- Message M , $send(M)$, and $receive(M)$
- Corresponding events: $a \sim b$ denotes a and b occur at the same process
- Send-receive pairs: $T = \{(s, r) \in E_i \times E_j \mid s \text{ corresponds to } r\}$

Message Ordering Paradigms

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.**

Definitions (1)

- 1. Asynchronous execution (A-execution):** It is an execution $(E, \prec)$ for which the causality relation is a partial order.
- 2. FIFO executions:** A FIFO execution is an A-execution in which,
for all (s, r) and $(s', r') \in T$, $(s \sim s' \text{ and } r \sim r' \text{ and } s \prec s') \Rightarrow r \prec r'$
- 3. Causal order (CO):** A CO execution is an A-execution in which,
for all (s, r) and $(s', r') \in T$, $(r \sim r' \text{ and } s \prec s') \Rightarrow r \prec r'$
- 4. 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.

Definitions (2)

5. Message Order (MO): A MO execution is and A-execution in which, for all (s, r) and $(s', r') \in T$, $s \prec s' \Rightarrow \neg(r' \prec r)$

6. Empty-Interval (EI) execution: A execution $(E, \prec)$ is an empty-interval (EI) execution if for each pair of events $(s, r) \in T$, the open interval set $\{x \in E \mid s \prec x \prec r\}$ in the partial order is empty.

7. Synchronous Executions (SYNC): An execution $(E;)$ for which the causality relation is a partial order.

- Handshake between sender and receiver
- Instantaneous communication $\Rightarrow$ modified definition of causality, where $s; r$ are atomic and simultaneous, neither preceding the other.

Hierarchy of execution classes

SYNC, CO, FIFO, and A denote the set of all possible executions ordered by synchronous order, causal order, FIFO order, and non- FIFO order, respectively.

$$\text{SYNC} \subset \text{CO} \subset \text{FIFO} \subset \text{A}$$

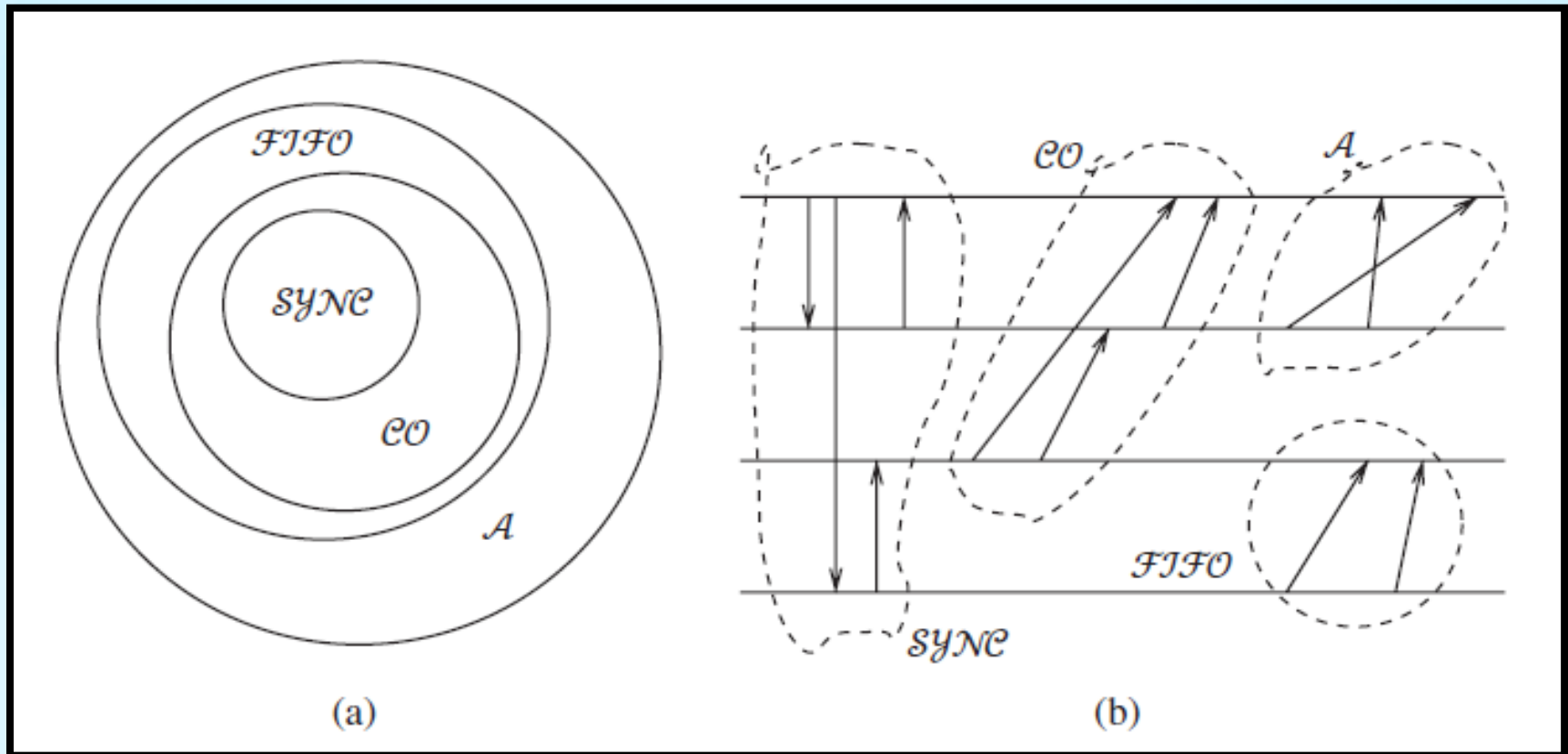


Figure 16.1 Hierarchy of execution classes. (a) Venn diagram. (b) Example executions.

Group Communication

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.

Contd...

- **Network layer or hardware-assist multicast** cannot easily provide:
 - Application-specific semantics on message delivery order
 - Adapt groups to dynamic membership
 - Multicast to arbitrary process set at each send
 - Provide multiple fault-tolerance semantics
- Closed group (source part of group) vs. open group (outside of group)
- # groups can be **$O(2^n)$**

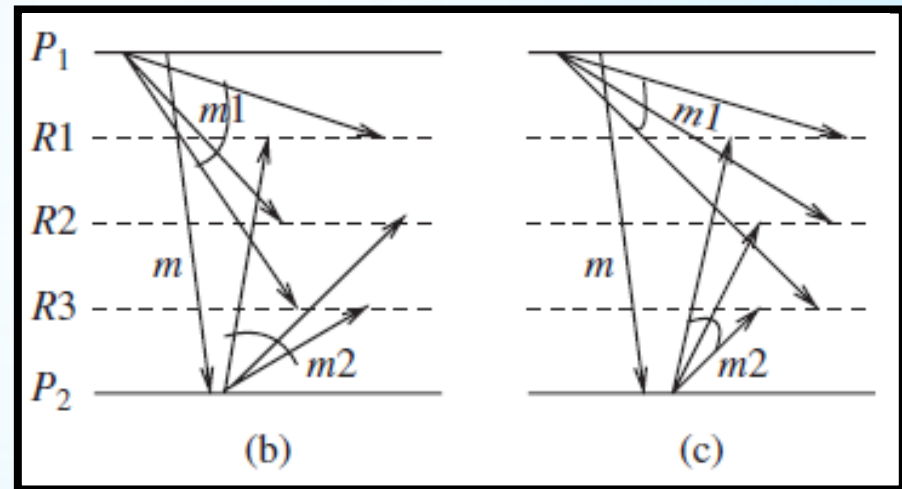
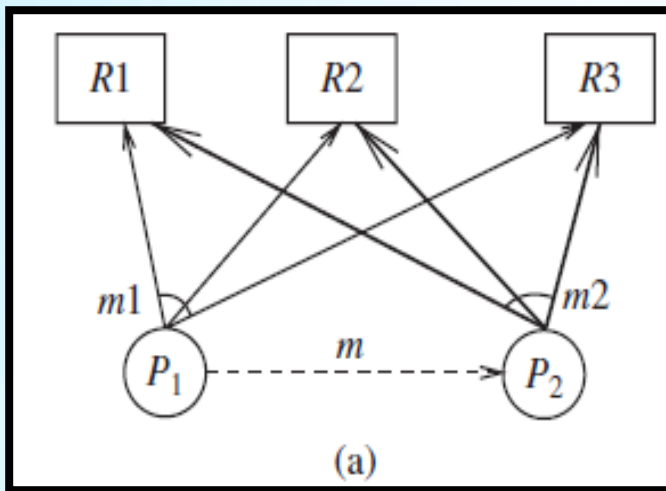


Figure 16.2: (a) Updates to 3 replicas. (b) Causal order (CO) and total order violated. (c) Causal order violated. If m did not exist, (b,c) would not violate CO.

The Raynal–Schiper–Toueg algorithm (1991)

- 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 **$\text{dest}(M)$** .
- 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 16.1** 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.

Raynal-Schiper-Toueg (RST) Algorithm

```
(local variables)
array of int SENT[1...n, 1...n]
array of int DELIV[1...n]           // DELIV[k] = # messages sent by k that are delivered locally

(1) send event, where  $P_i$  wants to send message  $M$  to  $P_j$ :
(1a) send ( $M, SENT$ ) to  $P_j$ ;
(1b)  $SENT[i, j] \leftarrow SENT[i, j] + 1$ .

(2) message arrival, when ( $M, ST$ ) arrives at  $P_i$  from  $P_j$ :
(2a) deliver  $M$  to  $P_i$  when for each process  $x$ ,
(2b)  $DELIV[x] \geq ST[x, i]$ ;
(2c)  $\forall x, y, SENT[x, y] \leftarrow \max(SENT[x, y], ST[x, y])$ ;
(2d)  $DELIV[j] \leftarrow DELIV[j] + 1$ .
```

Algorithm 16.1 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.

Assumptions/Correctness

- FIFO channels.
- Safety: Step (2a,b).
- Liveness: assuming no failures, finite
- propagation times

Complexity

n^2 ints/ process

n^2 ints/ message

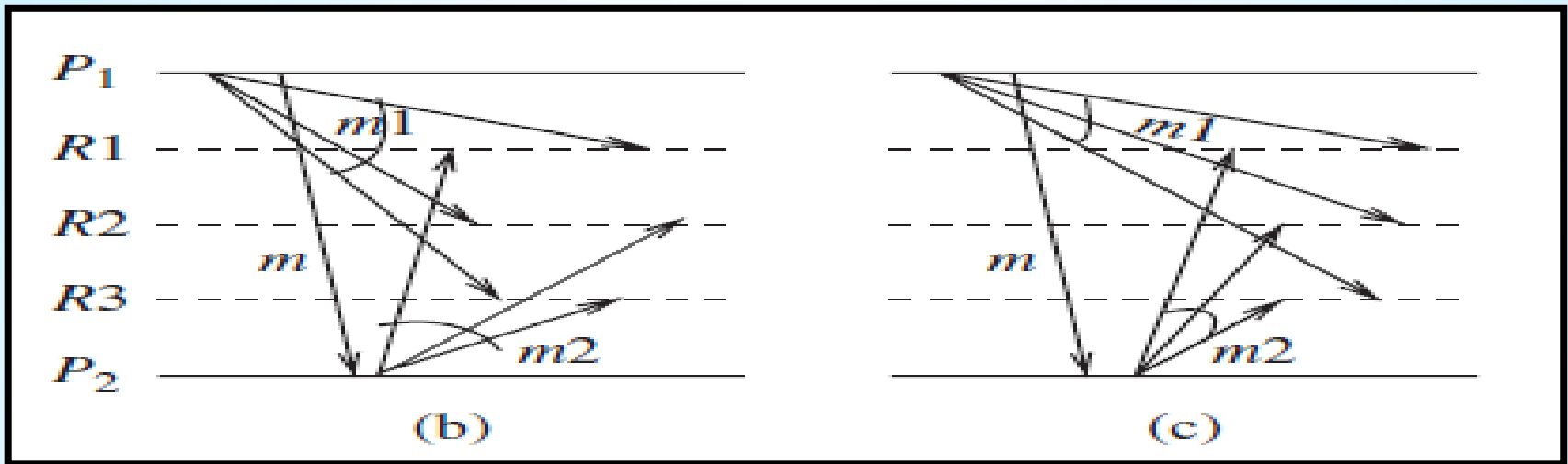
Time per send and receive event: n^2

Total Order

Total Order

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 (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 16.2 (b) does not satisfy total order**. Even if the message m did not exist, total order would not be satisfied. The execution in **Figure 16.2(c) satisfies total order**.

Centralized algorithm for Total Order

- (1) When P_i wants to multicast M to group G :
 - (1a) **send** $M(i, G)$ to coordinator.
- (2) When $M(i, G)$ arrives from P_i at coordinator:
 - (2a) **send** $M(i, G)$ to members of G .
- (3) When $M(i, G)$ arrives at P_j from coordinator:
 - (3a) **deliver** $M(i, G)$ to application.

Algorithm 16.2 A centralized algorithm to implement total order and causal order for messages

- Assuming all processes broadcast messages, the centralized solution shown in Algorithm 16.2 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.

Complexity

- **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.

Three-phase distributed algorithm

- A distributed algorithm that enforces total and causal order for closed groups is given in **Algorithm 16.3**. 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**, 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).

Contd...

- **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).

Contd...

- **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).

Algorithm 16.3 3-phase Algorithm Code

```
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) $clock = clock + 1$;
(1b) send $REVISE_TS(M, i, tag, clock)$ to all processes;
(1c) $temp_ts = 0$;
(1d) await $PROPOSED_TS(j, i, tag, ts_j)$ from each process P_j ;
(1e) $\forall j \in N$, do $temp_ts = \max(temp_ts, ts_j)$;
(1f) send $FINAL_TS(i, tag, temp_ts)$ to all processes;
(1g) $clock = \max(clock, temp_ts)$.

(2) When $REVISE_TS(M, j, tag, clk)$ arrives from P_j :

(2a) $priority = \max(priority + 1, clk)$;
(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 $FINAL_TS(j, tag, clk)$ arrives from P_j :

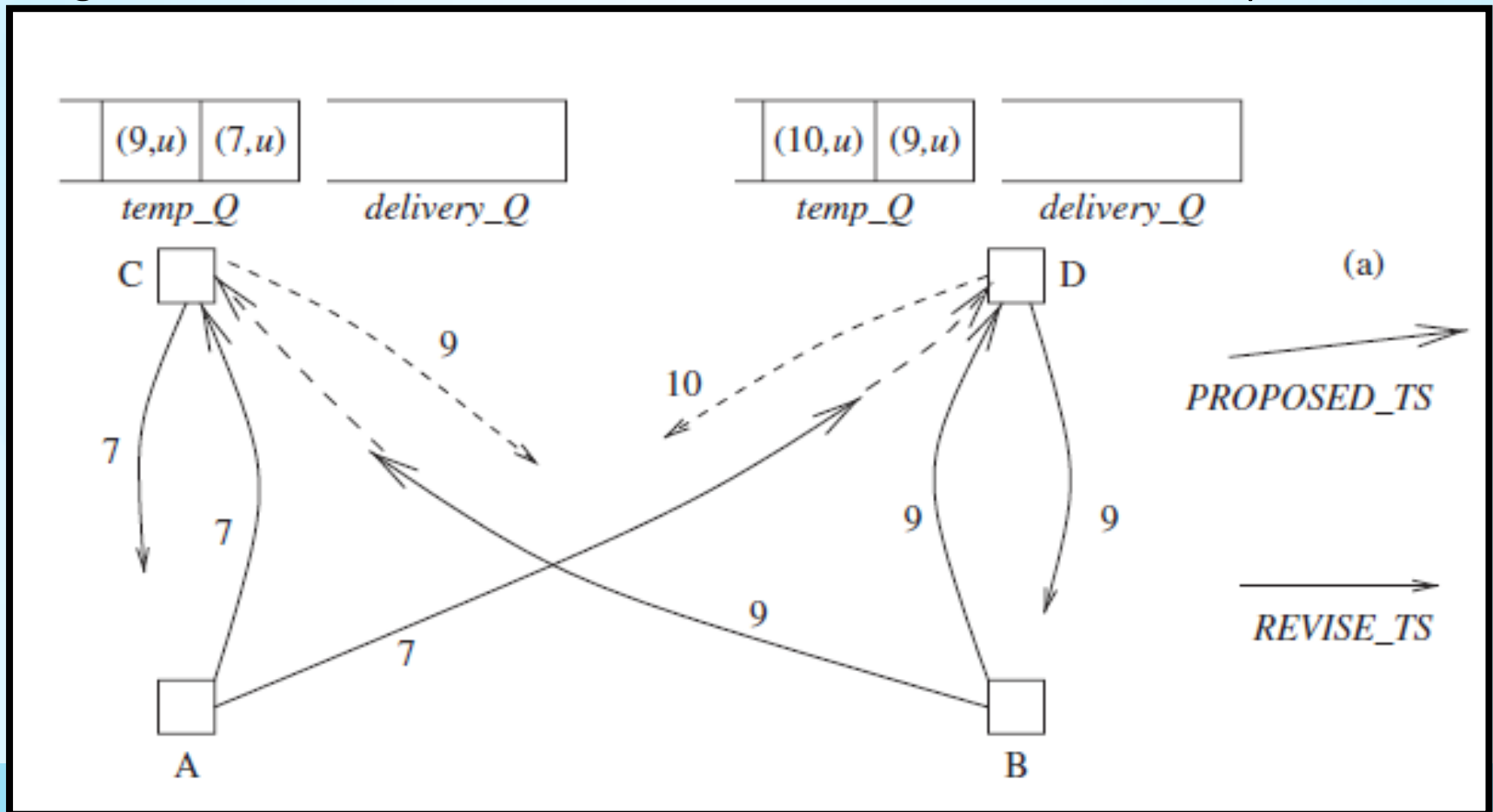
(3a) Identify entry $Q_entry(tag)$ in $temp_Q$, corresponding to tag ;
(3b) mark q_{tag} as deliverable;
(3c) Update $Q_entry.timestamp$ to clk and re-sort $temp_Q$ based on the *timestamp* field;
(3d) if $head(temp_Q) = Q_entry(tag)$ then
(3e) move $Q_entry(tag)$ from $temp_Q$ to $delivery_Q$;
(3f) while $head(temp_Q)$ is deliverable do
(3g) move $head(temp_Q)$ from $temp_Q$ to $delivery_Q$.

(4) When P_i removes a message $(M, tag, j, ts, deliverable)$ from $head(delivery_Q_i)$:

(4a) $clock = \max(clock, ts) + 1$.

Example: a) **PROPOSED_TS** and **REVISE_TS** messages.

Example: An example execution to illustrate the algorithm is given in Figure 16.3(a). Here, A and B multicast to a set of destinations and C and D are the common destinations for both multicasts. A snapshot for **PROPOSED_TS** and **REVISE_TS** messages. The dashed lines show the further execution after the snapshot.



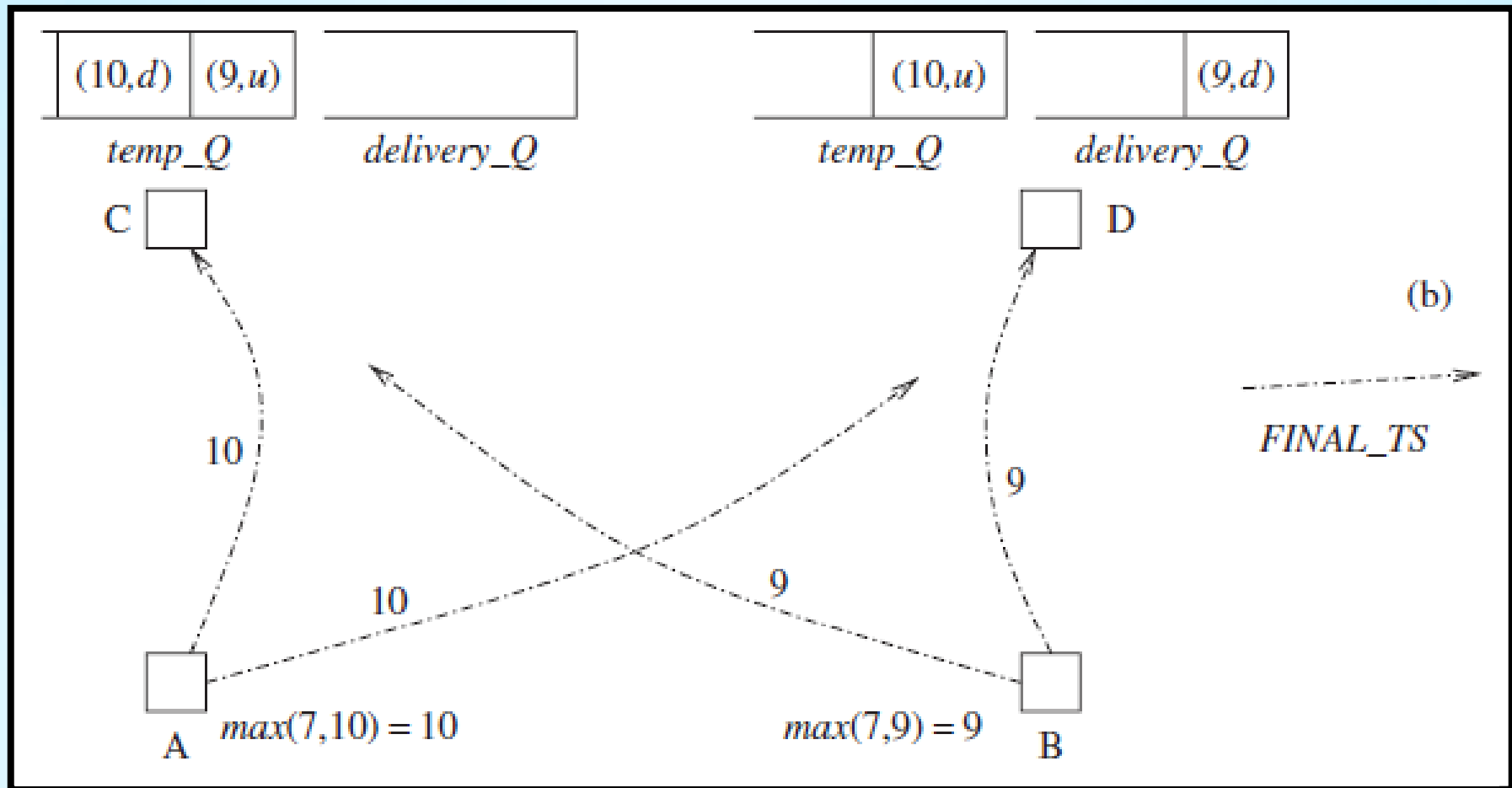
Sequence of Steps

Figure 16.3(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.
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_TSs seen so far, and then sends **PROPOSED_TS(10)** message to A.

Example: (b) The FINAL_TS messages

Figure 16.3 (b) The FINAL_TS messages in the example.



Sequence of Steps

Figure 16.3(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**.

Contd...

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**.

Complexity

- This algorithm uses **three phases**
- **$3(n - 1)$** messages for **$n - 1$** destinations
- Delay: **3 message hops**
- Also implements **causal order**

Note:

1. The **3 phase algorithm** is closely structured along the lines of **Lamport's algorithm** for mutual exclusion.
2. **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.
3. This can be exploited to deliver the message by all the processes in the same total order (instead of entering the critical section).

A Nomenclature for Multicast

- Four classes of source–destination relationships for open groups:

a) SSSG: Single source and single destination group.

b) MSSG: Multiple sources and single destination group.

c) SSMG: Single source and multiple, possibly overlapping, groups.

d) MSMG: Multiple sources and multiple, possibly overlapping, groups.

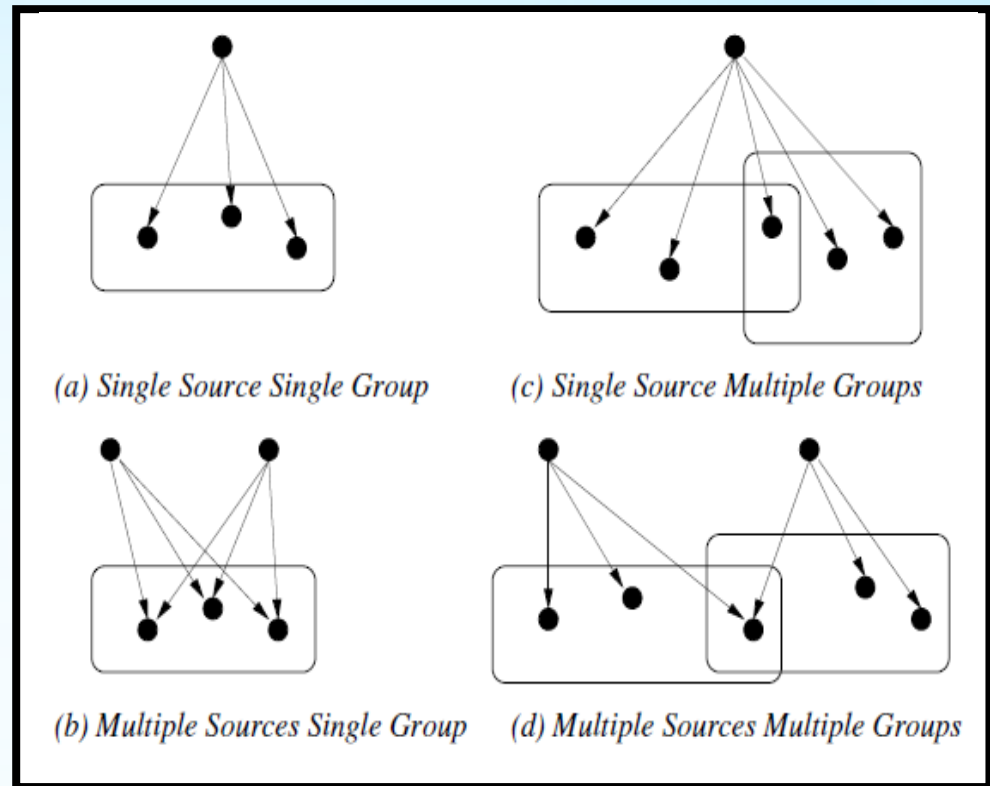


Figure 16.4 : Four classes of source-destinations relationships for open-group multicasts. For closed-group multicasts, the sender needs to be part of the recipient group. SSSG, SSMG: easy to implement MSSG: easy. E.g., Centralized algorithm MSMG: Semi-centralized *propagation tree* approach

Propagation Trees for Multicast: Definitions

- Set of **groups** $G = \{G_1 \dots G_g\}$
- Set of **meta-groups** $MG = \{MG_1, \dots MG_h\}$ with the following properties:
 - Each process belongs to a single meta-group, and has the exact same group membership as every other process in that meta-group.
 - No other process outside that meta-group has that exact group membership.
- MSMG to groups $\rightarrow$ MSSG to meta-groups
- A **distinguished node** in each meta-group acts as its **manager**.
- For each user group G_i , one of its meta-groups is chosen to be its **primary meta-group** (PM), denoted $PM(G_i)$.
- All meta-groups are organized in a **propagation forest/tree** satisfying:
 - For user group G_i , $PM(G_i)$ is at the lowest possible level (i.e., farthest from root) of the tree such that all meta-groups whose destinations contain any nodes of G_i belong to subtree rooted at $PM(G_i)$.

Propagation Trees for Multicast: Example

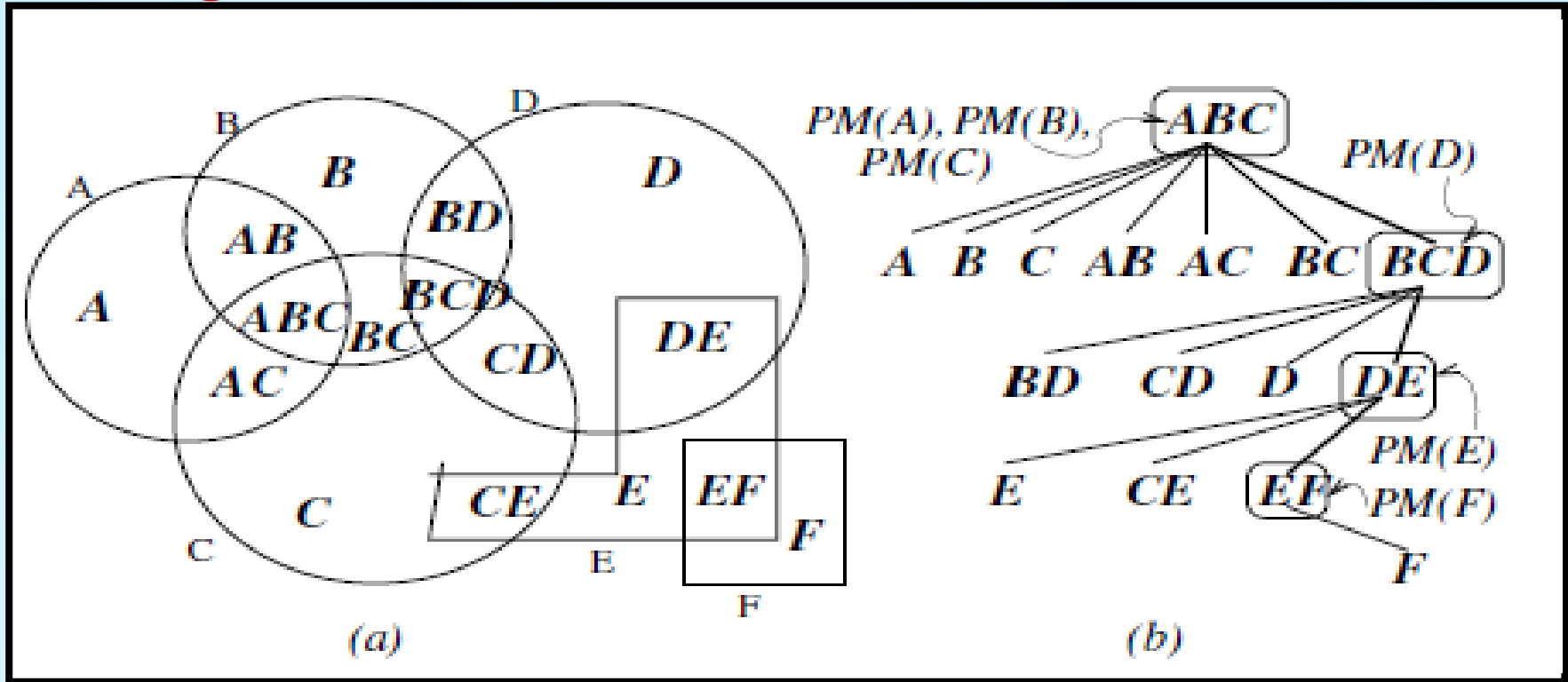


Figure 16.5: Example illustrating a propagation tree. Meta-groups in boldface. (a) Groups A, B, C, D, E and F, and their meta-groups. (b) A *propagation tree*, with the primary meta-groups labeled.

- (ABC) , (AB) , (AC) , and (A) are meta-groups of user group (A) .
- (ABC) is $PM(A)$, $PM(B)$, $PM(C)$. (B, C, D) is $PM(D)$. (D, E) is $PM(E)$. (E, F) is $PM(F)$.
- (ABC) is joint with (CD) . Neither subsumes the other and both are a subset of C .
- Meta-group (ABC) is the primary meta-group $PM(A)$, $PM(B)$, $PM(C)$. Meta-group (BCD) is the primary meta-group $PM(D)$. A multicast to D is sent to (BCD) .

Classification of Application-Level Multicast Algorithms

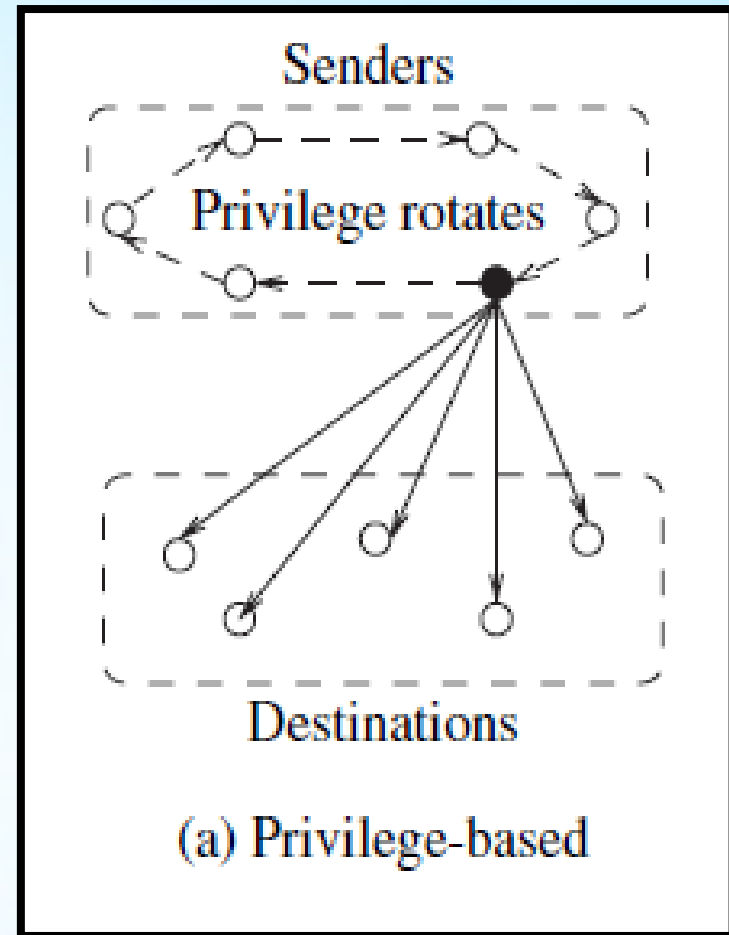
Many multicast protocols have been developed and deployed, but they can all be classified as belonging to one of the following five classes.

1. Communication history-based algorithms:

- Use a part of the **communication history** to guarantee ordering requirements.
- E.g. RST algorithm
- Do not need to track separate groups
- Work for open-group multicasts.

2. Privilege-based:

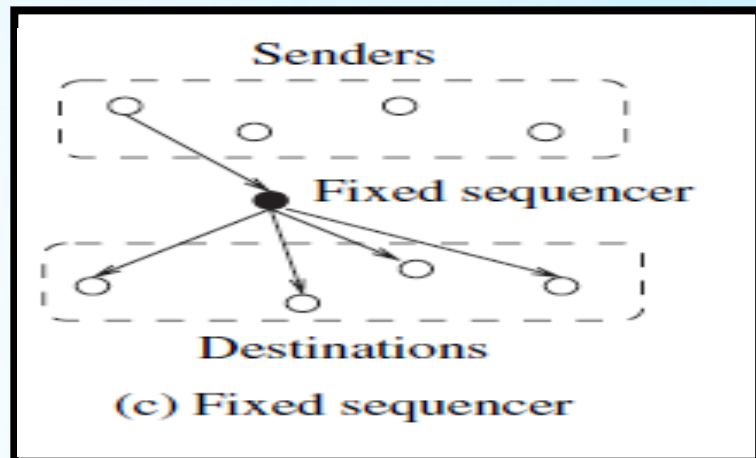
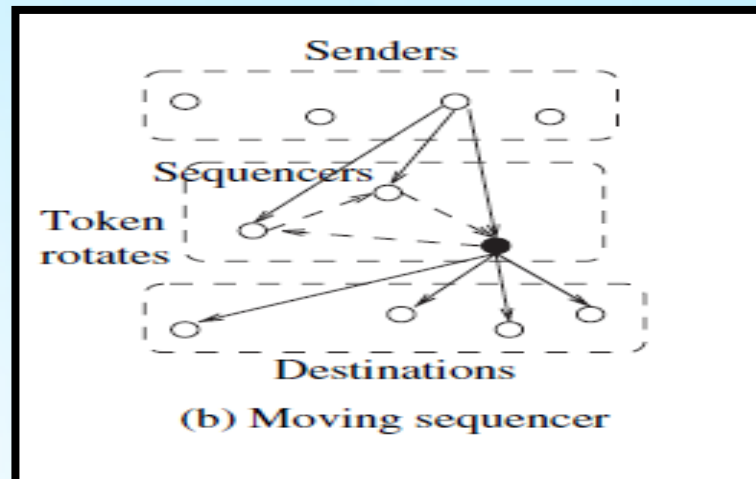
- Token-holder multicasts
- Processes deliver msgs in order of **sequence no.**
- Typically closed groups, and CO & TO.
- E.g., Totem, On-demand.



Contd...

3. Moving sequencer:

- Sequencers' token has **sequence no** and list of messages for which **sequence no** has been assigned (these are sent msgs).
- On receiving token, sequencer assigns **sequence no's** to received but unsequenced messages, and sends the newly sequenced messages to destinations.
- Destinations deliver in order of *sequence no*
- E.g., Chang-Maxemchuk, Pinwheel



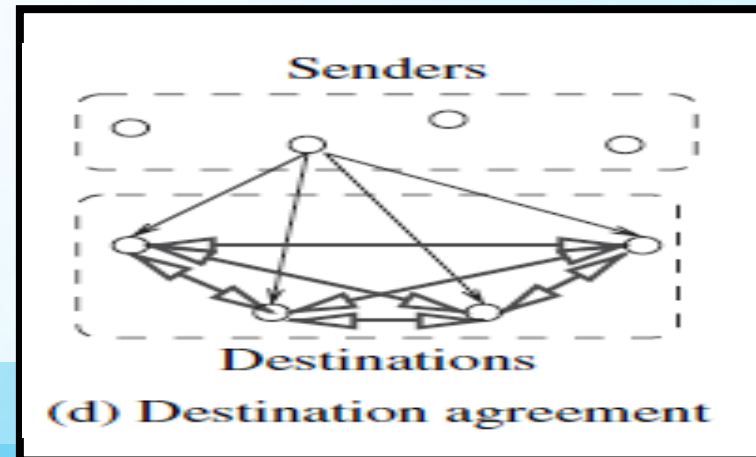
4. Fixed sequencer:

Simplifies moving sequencer approach.

E.g., propagation tree, ISIS, Amoeba, Phoenix

5. Destination agreement:

- Destinations receive limited ordering info.
- Timestamp-based (**Lamport's 3-phase**)
- Agreement-based, among destinations.



Few Other Algorithms for Message Ordering and Group Communication

Author	Algorithm
M. Raynal et al. (1991)	Algorithm for causal order
Kshemkalyani and Singhal (1996,1998)	The space and time optimal algorithm for causal order
K. Birman and T. Joseph (1987)	The algorithm for total order
Garcia-Molina and Spauster (1991) X. Jia (1995), and Chiu and Hsiao (1998)	The algorithm for total order using propagation
Chang and Maxemchuk (1984)	The moving sequencer algorithms
P. Ezhilchelvan et al. (1995)	An efficient fault-tolerant group communication protocol
E. Gilbert and H. Pollack (1968)	Steiner minimal trees
L. Kou et al.	A fast algorithm for Steiner trees
Kompella <i>et al.</i> (1993)	Delay-bounded minimum Steiner
Hadzilacos and Toueg (1993)	Semantics of fault-tolerant group communication
Ballardie <i>et al.</i> (1993)	Core-based trees

Conclusion

- **Inter-process communication via message-passing** is at the core of any distributed system.
- In this lecture, we have discussed **non-FIFO, FIFO, causal order, asynchronous order and synchronous communication paradigms** for ordering messages.
- 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 discussed.
- Then we have explained the **propagation trees for multicast** and classification of application-level multicast algorithms.
- In upcoming lecture, we will discuss about '**Self-Stabilization**'.