BITS Pilani
Hyderabad Campus

Distributed Computing (CS 4 – M4)

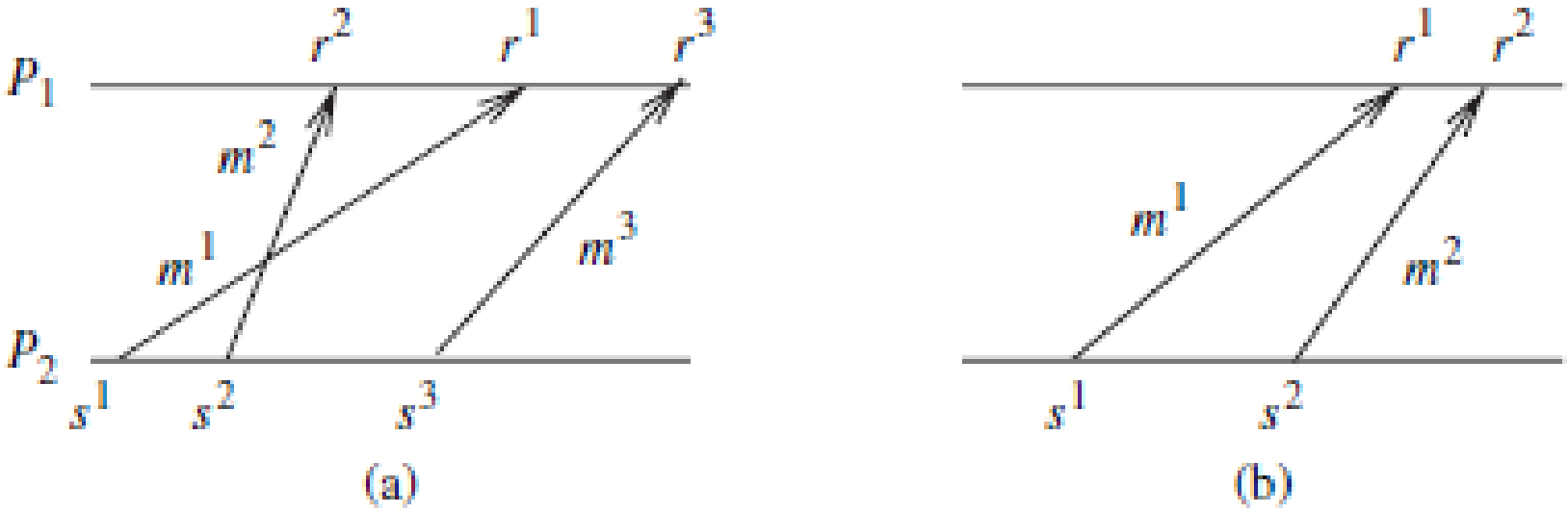
# Message Ordering and Termination Detection

Prof. Geetha
Associate Professor, Dept. of Computer Sc. & Information Systems
BITS Pilani Hyderabad Campus
geetha@hyderabad.bits-pilani.ac.in

# Message ordering paradigms

- **FIFO**
  - each channel acts as a first-in first-out message queue
  - message ordering is preserved by channel
- **non-FIFO**
  - channel acts like a set
  - sender process adds messages to channel
  - receiver process removes messages from it
- **Causal Order** - for $m_{ij}$ and $m_{kj}$, if $send(m_{ij}) \rightarrow send(m_{kj})$,
- then $rec(m_{ij}) \rightarrow rec(m_{kj})$
- **Synchronous Order** - all the communication between pairs of processes uses synchronous send and receive

# Asynchronous executions (A-executions)




(a) An A-execution that is not a FIFO execution (b) An A-execution that is also a FIFO

# FIFO execution

$$\text{for all } (s, r) \text{ and } (s', r') \in \mathcal{T}, (s \sim s' \text{ and } r \sim r' \text{ and } s \prec s') \Longrightarrow r \prec r'.$$

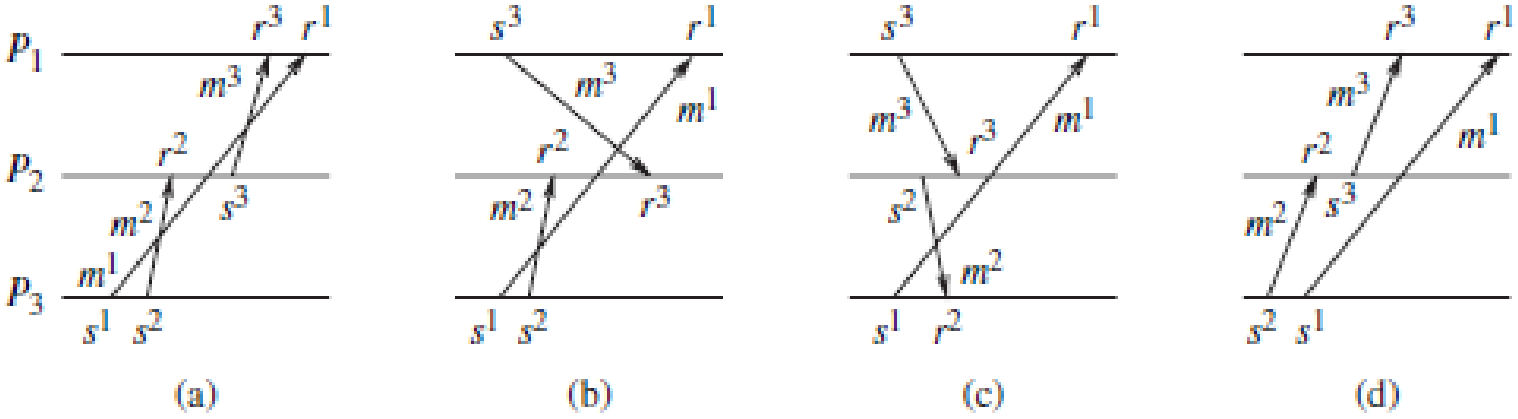
- FIFO logical channel over a non-FIFO channel would use a separate numbering scheme to sequence the messages on each logical channel
- The sender assigns and appends a *sequence_num, connection_id* tuple to each message
- The receiver uses a buffer to order the incoming messages as per the sender's sequence numbers, and accepts only the "next" message in sequence

# Causally Ordered (CO) executions

$$\text{for all } (s, r) \text{ and } (s', r') \in \mathcal{T}, (r \sim r' \text{ and } s \prec s') \Longrightarrow r \prec r'.$$

- 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.
- If s and s' are not related by causality, then CO is vacuously satisfied because the antecedent of the implication is false

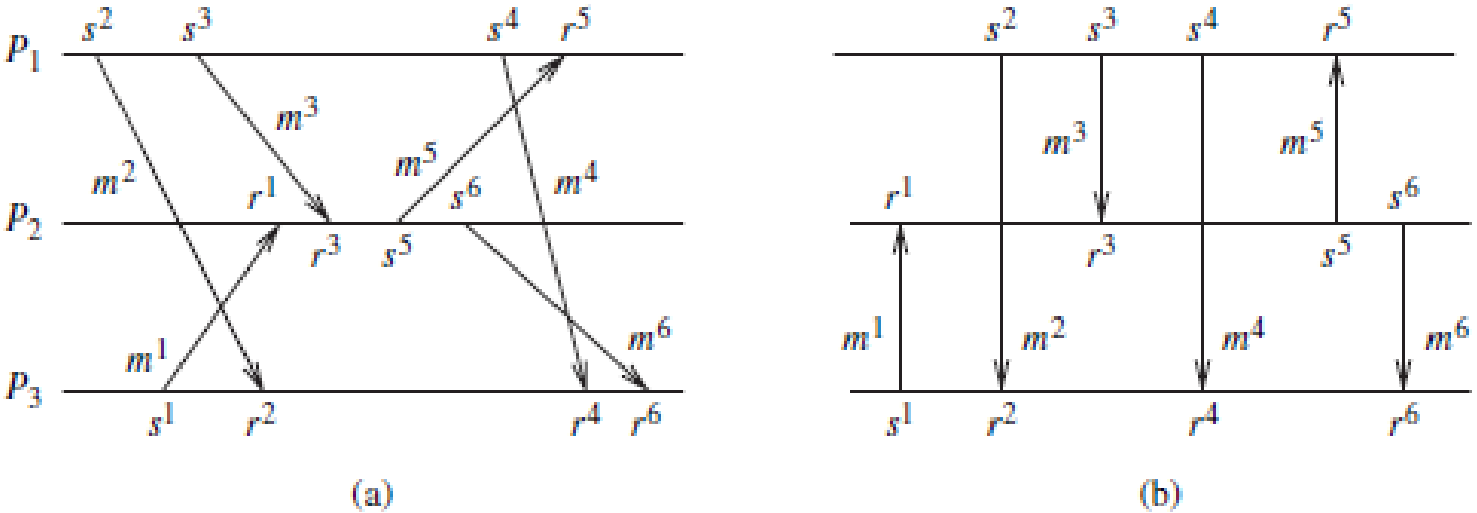
# Illustration of causally ordered executions




(a) Not a CO execution. (b),(c), and (d) CO executions.

# Synchronous (SYNC) execution – S executiom

**A synchronous execution (or S-execution) is an execution (E, <<) for which the causality relation << is a partial order**




(a) Execution in an asynchronous system (b) Equivalent instantaneous communication

- 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 occurring instantaneously and atomically
- In a timing diagram, the “instantaneous” message communication can be shown by bidirectional vertical message lines

# Group Communication

- Group communication needs to be supported
- message broadcast - sending a message to all members in the distributed system
- multicasting - a message is sent to a certain subset, identified as a group, of the processes in the system
- unicasting - point-to-point message communication

# Group Communication

- **closed group algorithm** –
  - when a multicast algorithm requires the sender to be a part of the destination group
- **open group algorithm** –
  - when the sender of the multicast can be outside the destination group

- multicast algorithms
  - number of groups may be potentially exponential, i.e., $O(2^n)$

# Causal Order

- Two criteria must be satisfied by a causal ordering protocol:
  - Safety
  - Liveness

# Causal Order

- **Safety**
- a message M arriving at a process may need to be buffered until all system-wide messages sent in the causal past of the send(M) event to the same destination have already arrived
- distinction is made between
  - arrival of a message at a process
  - event at which the message is given to the application process
- **Liveness**
- A message that arrives at a process must eventually be delivered to the process

# Raynal–Schiper–Toueg Algorithm

- 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)
- log can be examined to ensure whether it is safe to deliver a message
- channels are FIFO

Allows each send event to unicast, multicast, or broadcast a message in the system.

# Raynal–Schiper–Toueg Algorithm contd..

**<u>local variables:</u>**

- **array of int** SENT[1 …… n, 1 …… n] (n x n array)
  - $SENT_i[j, k]$ = no. of messages sent by $P_j$ to $P_k$ as known to $P_i$
- **array of int** DELIV [1 …… n]
  - $DELIV_i[j]$ = no. of messages from $P_j$ that have been delivered to $P_i$
  - DELIV[j] = no. of messages sent by $P_j$ 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] = SENT[i, j] + 1

# Raynal–Schiper–Toueg Algorithm

(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, $SENT[x, y] = \max(SENT[x, y], ST[x, y])$

(2d) $DELIV[j] = DELIV[j] + 1$

# Raynal–Schiper–Toueg Algorithm

<u>Complexity:</u>

- space requirement at each process: $O(n^2)$ integers
- space overhead per message: $n^2$ integers
- time complexity at each process for each send and deliver event: $O(n^2)$

# Raynal–Schiper–Toueg Algorithm

- $P_1$ sent 4 messages to $P_2$
- $P_2$ sent 3 messages to $P_1$
- $P_1 \rightarrow DELIV_1[0\ \ 2]$ // 2 messages from $P_2$ have been delivered to $P_1$
- $P_2 \rightarrow DELIV_2[4\ \ 0]$ // all 4 messages from $P_1$ have been delivered to $P_2$
- Now if,
  - $P_1$ sends a message to $P_2$ → no buffering required
  - $P_2$ sends a message to $P_1$ → buffering required

# Birman-Schiper-Stephenson Protocol

- A message is delivered to a process only if the message preceding it has been delivered
- buffer is needed for pending deliveries
- each message has a vector that contains information for the recipient to determine if another message preceded it
- all messages are broadcast

# Birman-Schiper-Stephenson Protocol

- $C_i$ = vector clock of $P_i$
- $C_i[j]$ = $j^{th}$ element of $C_i$
  - contains $P_i$'s latest value for the current time in $P_j$
- tm = vector timestamp for message m
- stamped after local clock is incremented

# Birman-Schiper-Stephenson Protocol

- $P_i$ sends a message m to $P_j$
- $P_i$ increments $C_i[i]$
- $P_i$ sets the timestamp tm = $C_i$ for message m

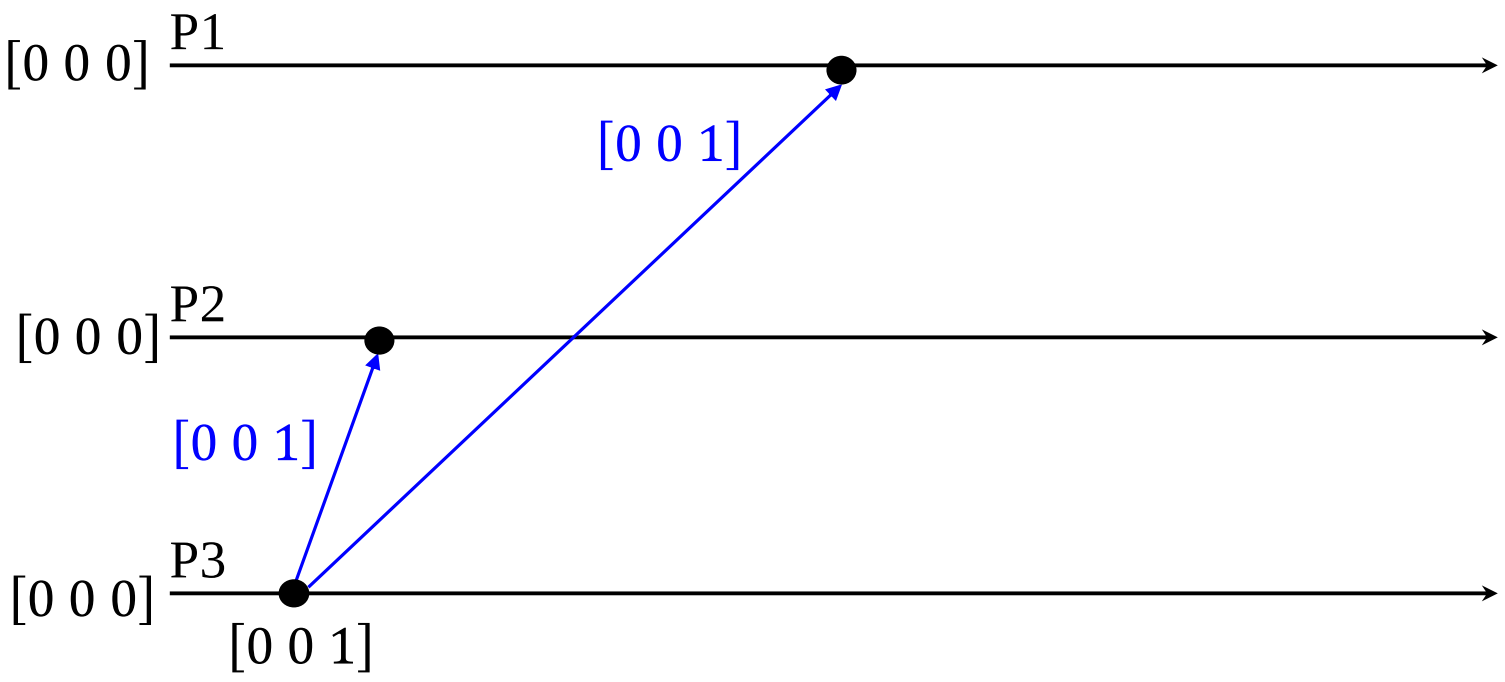
# Birman-Schiper-Stephenson Protocol

<u>$P_j$ receives a message m from $P_i$</u>

- when $P_j$ ($j \neq i$) receives m with timestamp tm, it delays message delivery until:
  - $C_j[i] = tm[i] - 1$ //$P_j$ has received all preceeding messages sent by $P_i$
  - $\forall$ k <= n and $k \neq i$, $C_j[k] \geq tm[k]$ //$P_j$ has received all the messages that were received at $P_i$ from other processes before $P_i$ sent m
- when m is delivered to $P_j$, update $P_j$'s vector clock

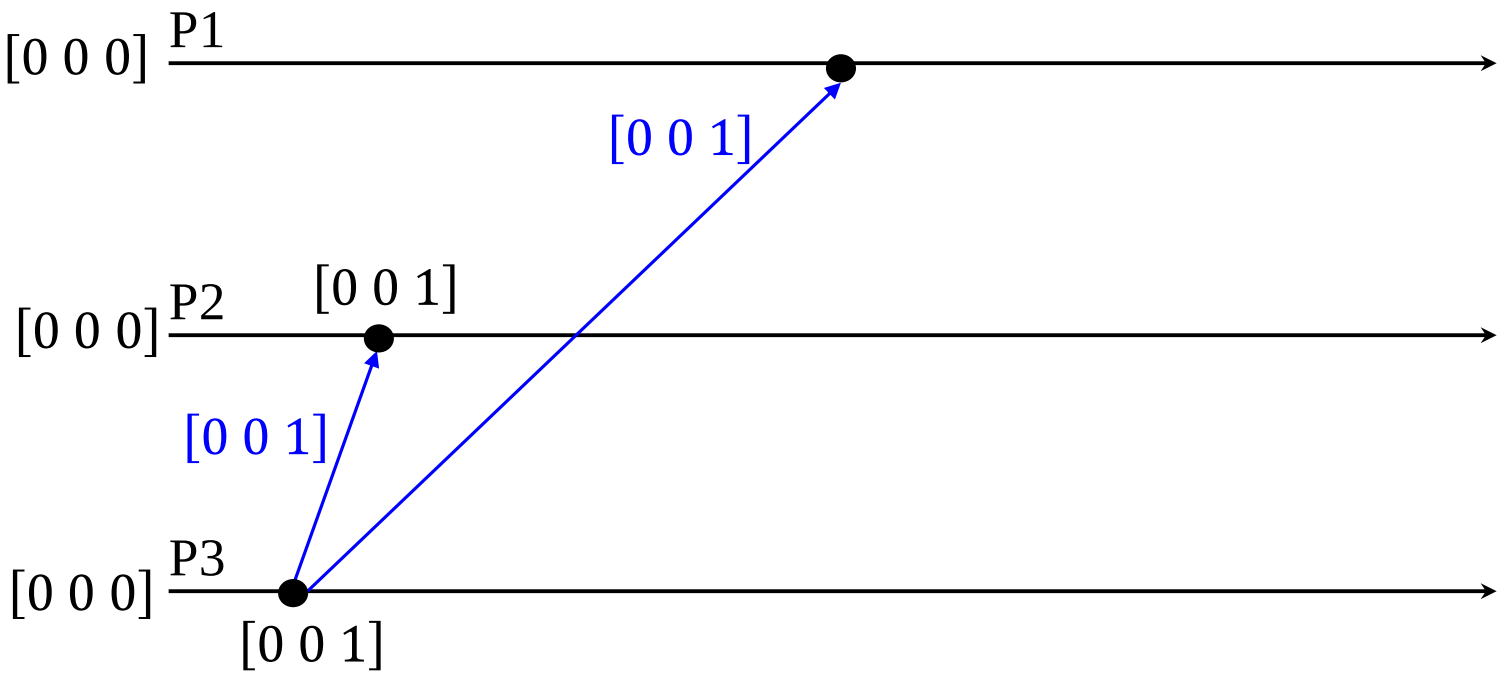
  $\forall i,\ C_j[i] = \max(C_j[i], tm[i])$
- check buffered messages to see if any can be delivered

# Birman-Schiper-Stephenson Protocol

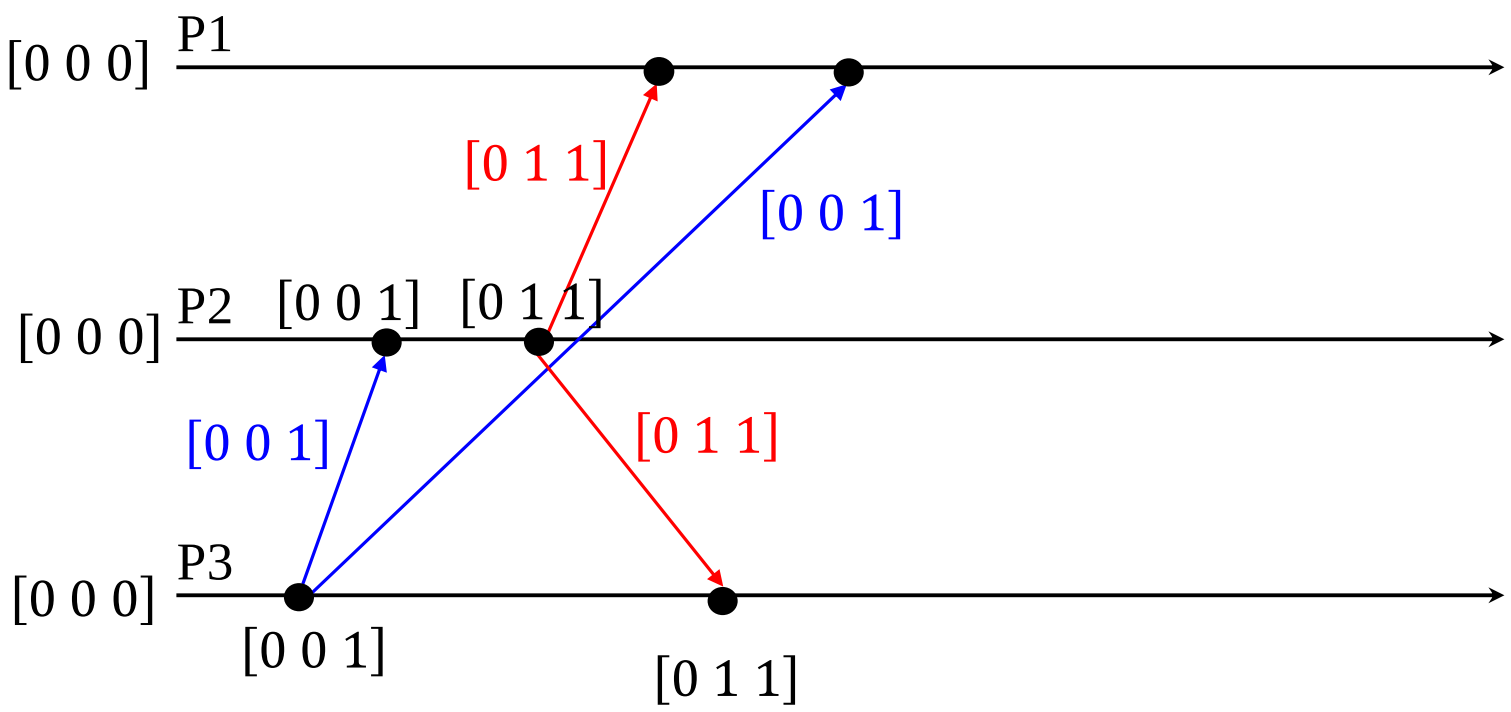



vectors in blue indicate timestamps of messages

# Birman-Schiper-Stephenson Protocol

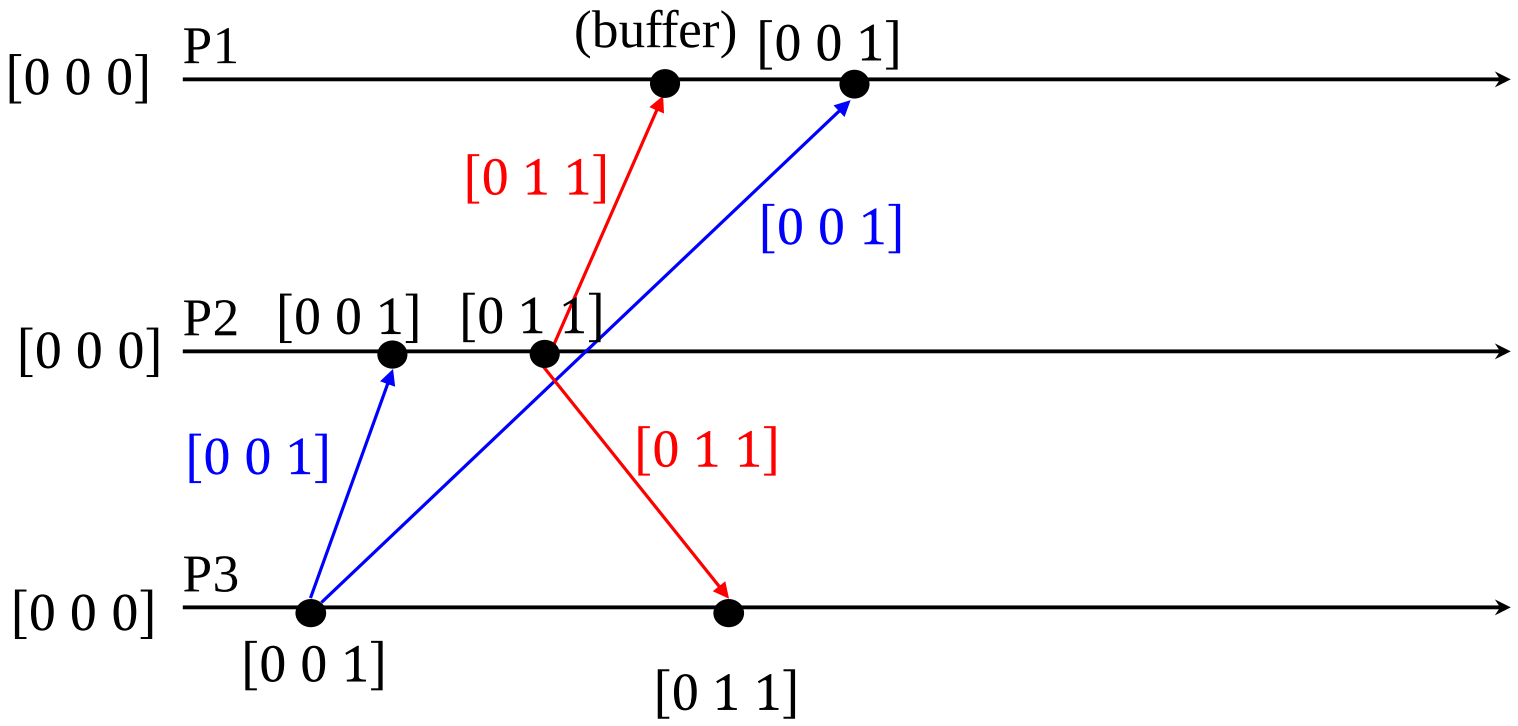


# Birman-Schiper-Stephenson Protocol

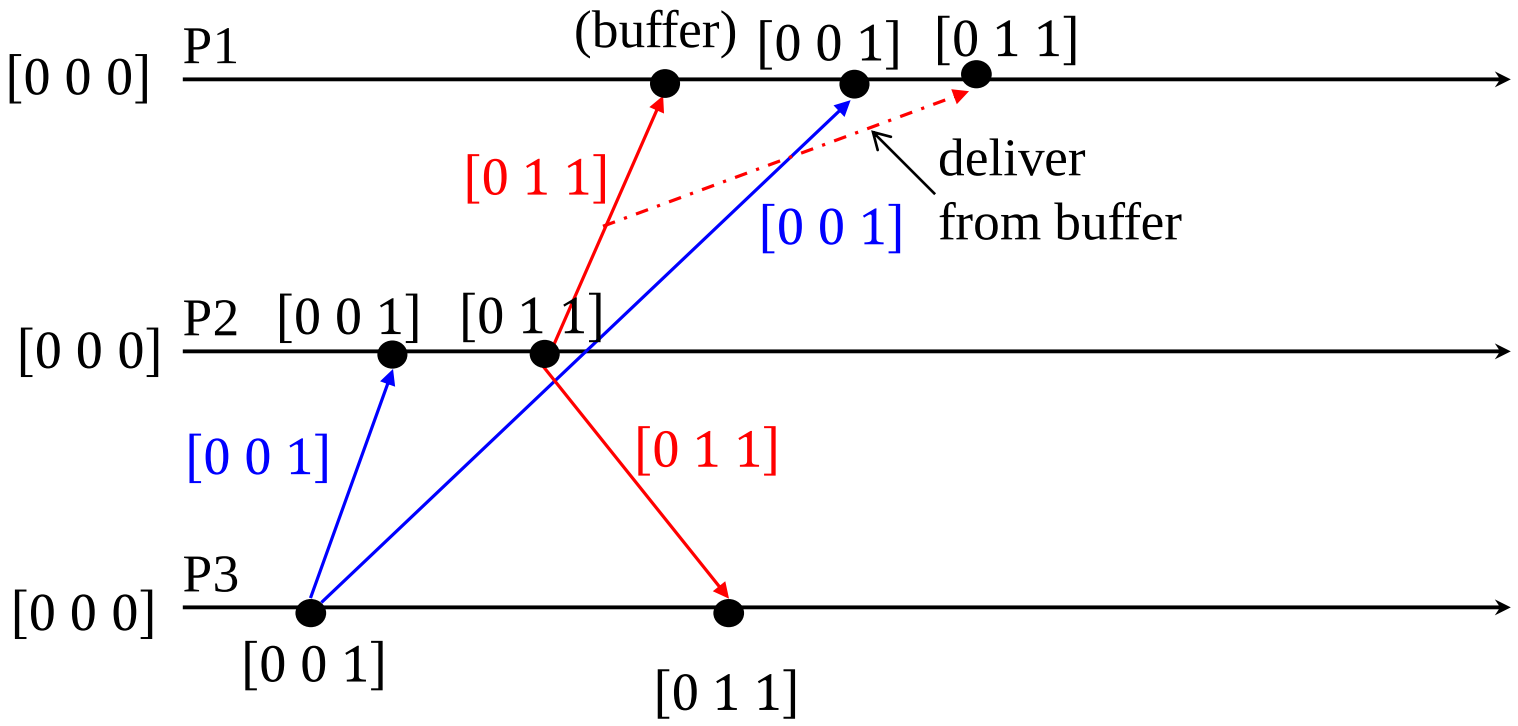



vectors in red indicate timestamps of messages

# Birman-Schiper-Stephenson Protocol



# Birman-Schiper-Stephenson Protocol



# Total Order

- requires that all messages be received in the same order by the recipients of the messages

- 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 $M_y$

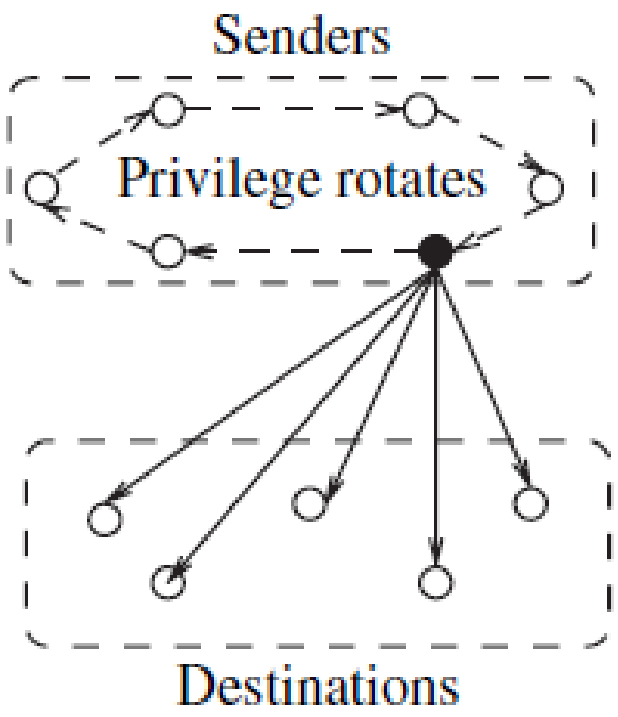
# Classification of Application-Level Multicast Algorithms

- **Communication history-based algorithms**
- use a part of the communication history to guarantee ordering requirements
- eg.,
  - RST (Raynal, Schiper. and Toueg), KS (Kshemkalyani and Singhal) algorithms -> causal ordering
  - Lamport's algorithm -> scalar timestamps
  - NewTop protocol – extension of Lamport's algorithm

# Classification of Application-Level Multicast Algorithms

- **Privilege-based algorithms**
- A token circulates among sender processes
- The token carries the sequence number for the next message to be multicast
- 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
- can provide total ordering and causal ordering using closed group configuration
- not scalable as do not permit concurrent send events

# Classification of Application-Level Multicast Algorithms




**Privilege-based algorithm**

# Classification of Application-Level Multicast Algorithms

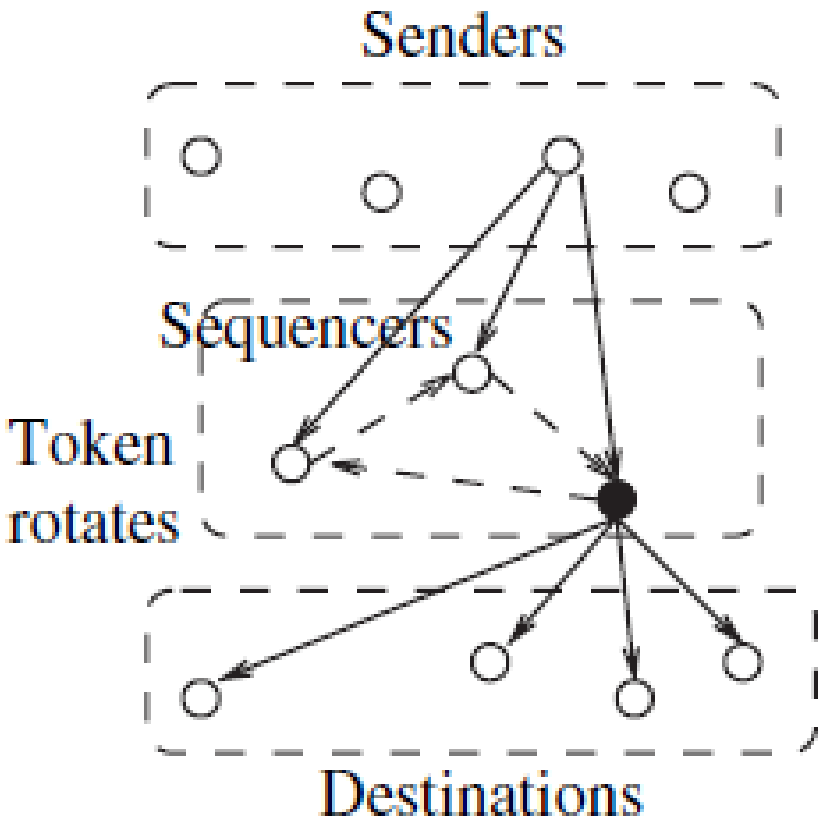
## Moving sequencer algorithms

- sender sends the message to all the sequencers to multicast a message
- sequencers circulate a token among themselves
- token carries
  - a sequence number
  - a list of all the messages for which a sequence number has already been assigned – such messages have been sent already
- when a sequencer receives the token
  - it assigns a sequence number to all received but un-sequenced messages
  - it sends the newly sequenced messages to the destinations
  - inserts these messages into the token list
  - passes the token to the next sequencer

# Classification of Application-Level Multicast Algorithms

## Moving sequencer algorithms contd..

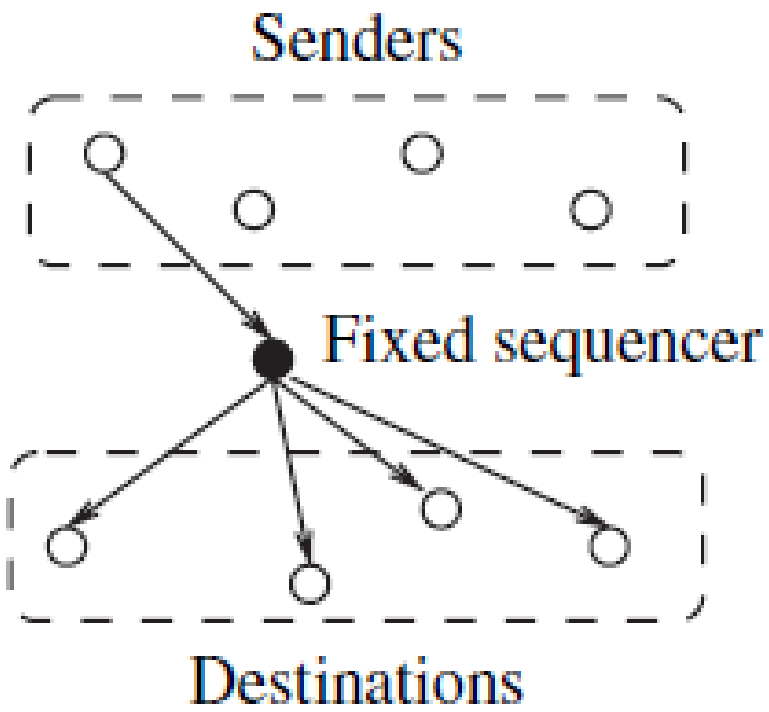
- destination processes deliver messages received in the order of increasing sequence number
- guarantee total ordering



# Classification of Application-Level Multicast Algorithms

## Fixed sequencer algorithms

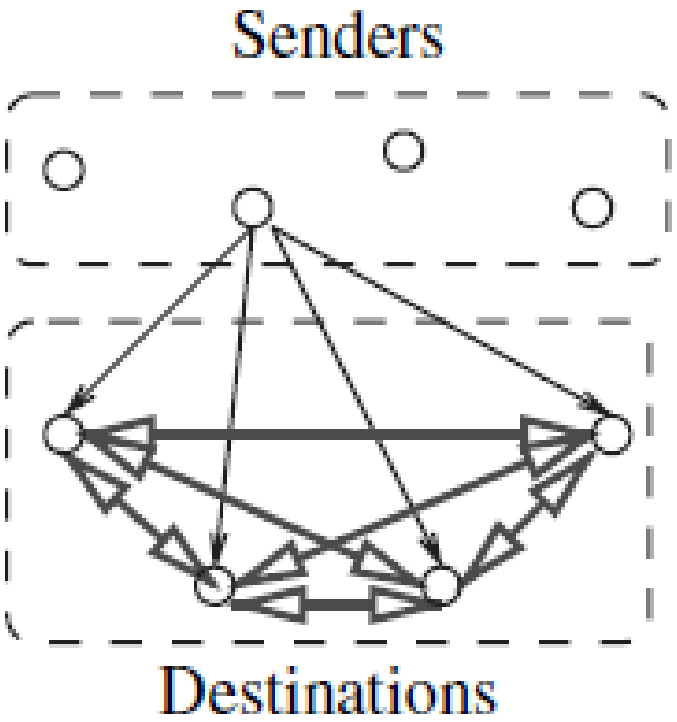
- simplified version of the previous class of algorithms
- a single sequencer is present
- centralized algorithms
- eg., propagation tree approach, ISIS sequencer, Amoeba, Phoenix, Newtop's asymmetric algorithm



# Classification of Application-Level Multicast Algorithms

## Destination agreement algorithms

- destinations receive the messages with some limited ordering information
- destination processes then exchange information among themselves to define an order
- Two sub-classes
  - uses timestamps
  - uses an agreement or consensus protocol among the processes



# Termination detection using distributed snapshots

- consistent snapshot of a distributed system captures stable properties
- termination of a distributed computation is a stable property
- if a consistent snapshot of a distributed computation is taken after the distributed computation has terminated, the snapshot will capture the termination of the computation
- all processes will have become idle
- there will be no in-transit messages

# Termination detection by weight throwing

- a process called controlling agent monitors the computation
- communication channel exists between each of the processes and the controlling agent and also between every pair of processes
- initially, all processes are in the idle state
- weight at each process is zero and the weight at the controlling agent is 1
- computation starts when the controlling agent sends a basic message to one of the processes
- process becomes active and the computation starts
- non-zero weight W ($0 < W \leq 1$) is assigned to each process in the active state

# Termination detection by weight throwing contd..

- when a process sends a message, it sends a part of its weight in the message
- when a process receives a message, it add the weight received in the message to its weight
- sum of weights on all the processes and on all the messages in transit is always 1
- when a process becomes passive, it sends its weight to the controlling agent in a control message, which the controlling agent adds to its weight
- controlling agent concludes termination if its weight becomes 1

# Spanning-tree-based termination detection

- Assumes N processes $P_i$, $0 \leq i \leq N$
- processes → nodes, channels → edges
- uses a fixed spanning tree of the graph with process $P_0$ at its root which is responsible for termination detection
- $P_0$ communicates with other processes to determine their states
- messages used for this purpose are called signals
- all leaf nodes report to their parents, if they have terminated
- an intermediate node will report to its parent when it has completed processing and all of its immediate children have terminated, and so on……
- root concludes that termination has occurred, if it has terminated and all of its immediate children have also terminated

# Recap Quiz

1. In the multicast algorithms, the number of groups may grow to the order of
(a) logn (b) $\log^2 n$ (c) $n^2$ (d) $2^n$

2. In which of the following message ordering paradigms, the channel acts as a set?
(a) Synchronous (b) causal order (c) FIFO (d) non-FIFO

3. In the Raynal–Schiper–Toueg Algorithm, the channel is organized as
(a) Synchronous (b) causal order (c) FIFO (d) non-FIFO

4. Which of the following is not a classification of application-level multicast algorithms?
(a) Fixed sequencer (b) moving sequencer (c) source agreement (d) destination agreement

5. The messages used in the spanning tree based termination detection algorithms are called
(a) Markers (b) signals (c) triggers (d) events

# Recap Quiz - key

| Q1 | Q2 | Q3 | Q4 | Q5 |
|---|---|---|---|---|
| d | d | c | c | b |

# References


- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 6, “Distributed Computing: Principles, Algorithms, and Systems”, Cambridge University Press, 2008 (Reprint 2013).
- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 7, “Distributed Computing: Principles, Algorithms, and Systems”, Cambridge University Press, 2008(Reprint 2013).
- https://courses.csail.mit.edu/6.006/fall11/rec/rec14.pdf