Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Logistics Related

- HW1 has been released.
  - You can solve first 5 questions right away
  - You can solve last two questions hopefully by end of this week.
- MP0 due on Wednesday.

Today's agenda

- Global State
  - Chapter 14.5
  - Goal: reason about how to capture the state across all processes of a distributed system without requiring time synchronization.
- **Multicast**

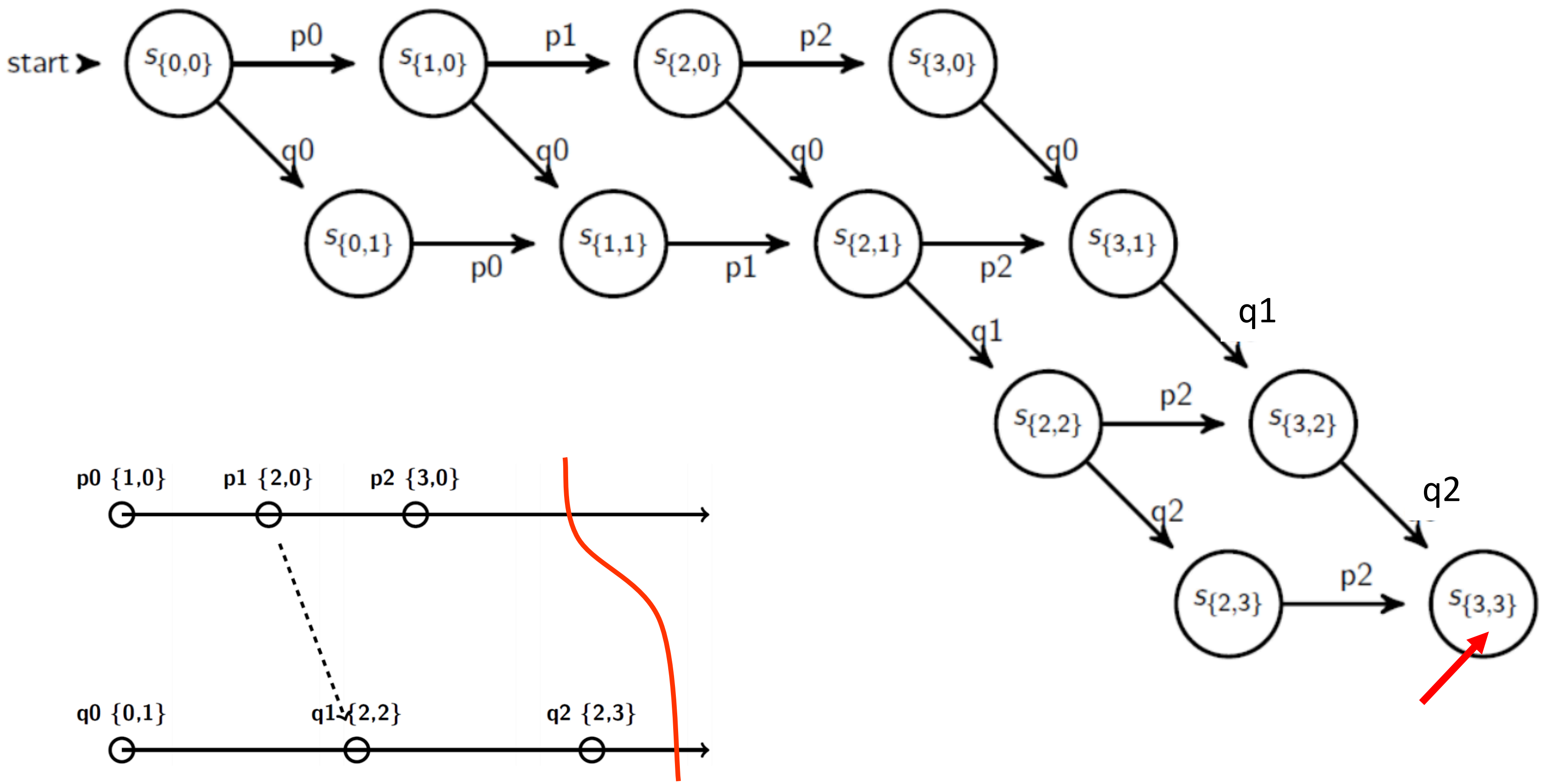
Recap

- State of each process (and each channel) in the system *at a given instant of time.*
  - Difficult to capture -- requires precisely synchronized time.

- Relax the problem: find a consistent global state.

- Chandy-Lamport algorithm to calculate global state.
  - Obeys causality (creates a consistent cut).
  - Does not interrupt the running distributed application.
  - Can be used to detect global properties.

State Transitions: Example



More notations and definitions

- H = set of all events across all processes.
- A run is a total ordering of events in H that is consistent with each $\mathbf{h}_i$'s ordering.

- A linearization is a run consistent with happens-before ($\rightarrow$) relation in H.

- Linearizations pass through consistent global states.

- A global state $\mathbf{S}_k$ is reachable from global state $\mathbf{S}_i$, if there is a linearization that passes through $\mathbf{S}_i$ and then through $\mathbf{S}_k$.

- The distributed system evolves as a series of transitions between global states S_0 , S_1 , ….

Global State Predicates

- A global-state-predicate is a property that is *true* or *false* for a global state.
  - Is there a deadlock?
  - Has the distributed algorithm terminated?

- Two ways of reasoning about predicates (or system properties) as global state gets transformed by events.
  - Liveness
  - Safety

Liveness

- Liveness = **guarantee that something good will happen, eventually**

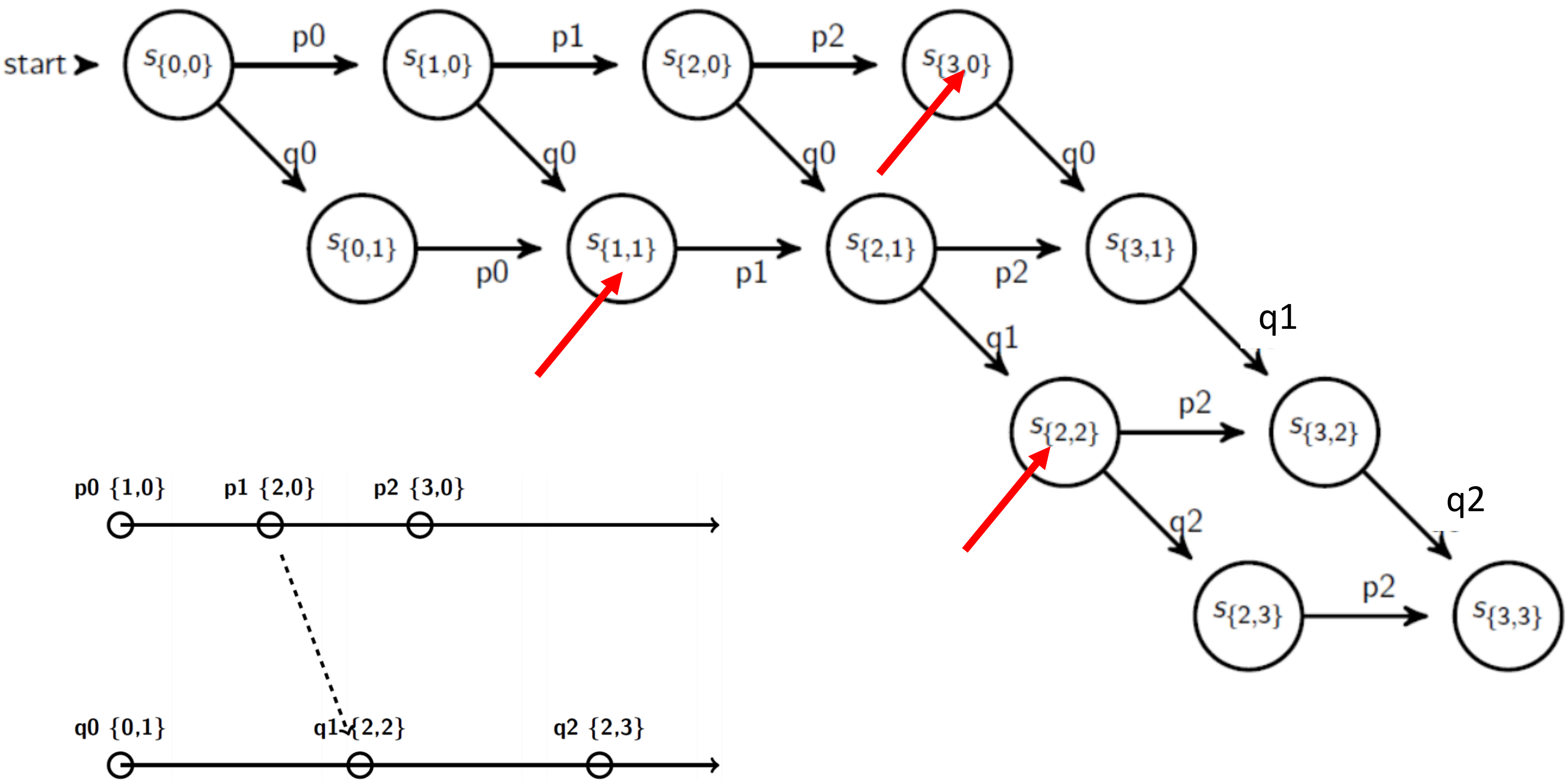
- **Examples:**
  - A distributed computation will terminate.
  - "Completeness" in failure detectors: the failure will be detected.
  - All processes will eventually decide on a value.

- A global state S_0 satisfies a **liveness** property P iff:
  - For all linearizations starting from S_0, P is true for some state S_L reachable from S_0.
  - $liveness(P(S_0)) \equiv \forall L \in$ linearizations from S_0, L passes through a S_L & $P(S_L)$ = true

Liveness Example

If predicate is true only in the marked states, does it satisfy liveness?

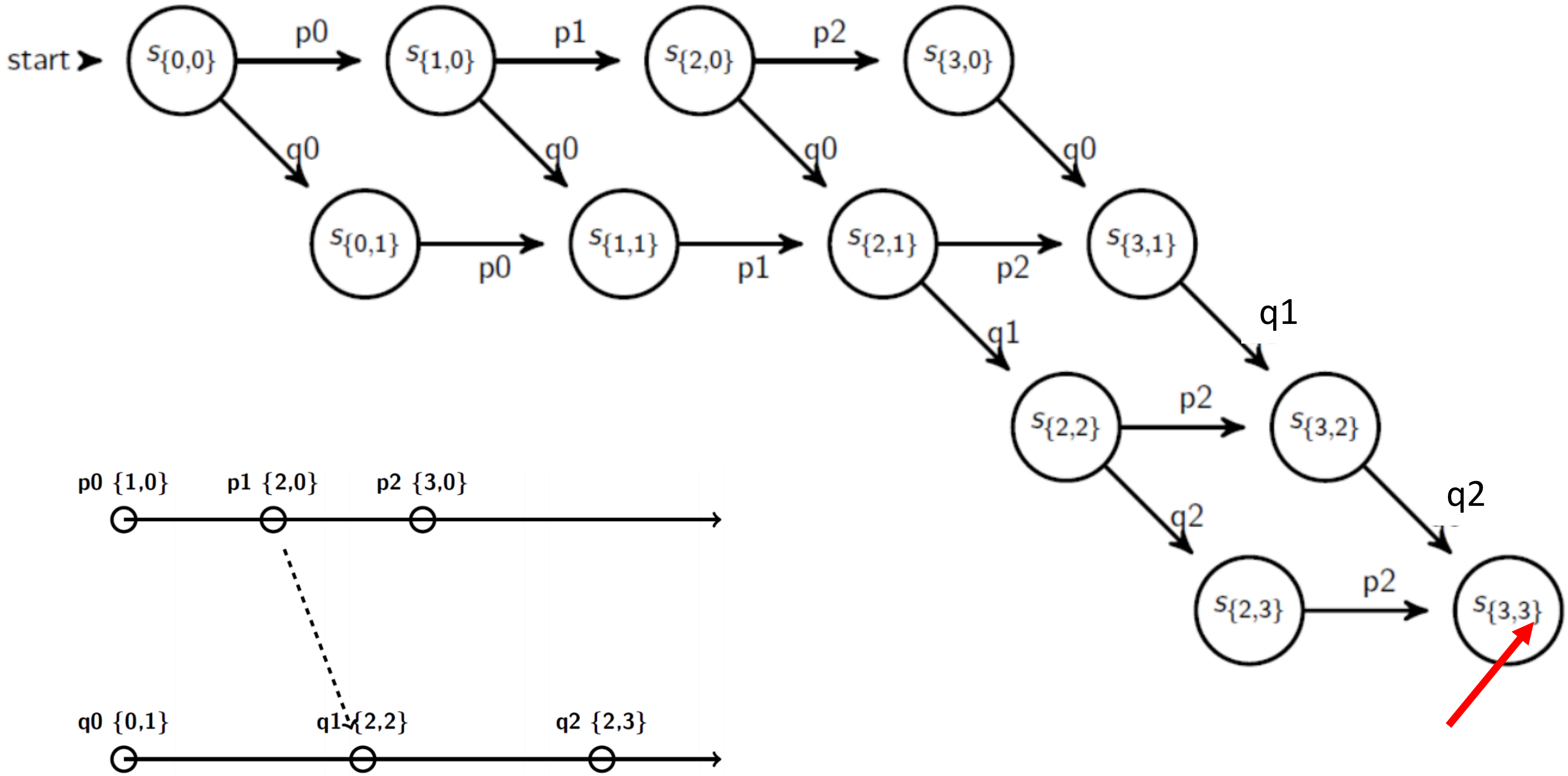
No



Liveness Example

If predicate is true only in the marked states, does it satisfy liveness?

Yes



Liveness

- **Liveness = guarantee that something good will happen, eventually**
- **Examples:**
  - A distributed computation will terminate.
  - "Completeness" in failure detectors: the failure will be detected.
  - All processes will eventually decide on a value.
- A global state S_0 satisfies a **liveness** property P iff:
  - liveness($P(S_0)$) $\equiv$ $\forall L \in$ linearizations from S_0, L passes through a S_L & $P(S_L)$ = true
  - For any linearization starting from S_0, P is true for some state S_L reachable from S_0.

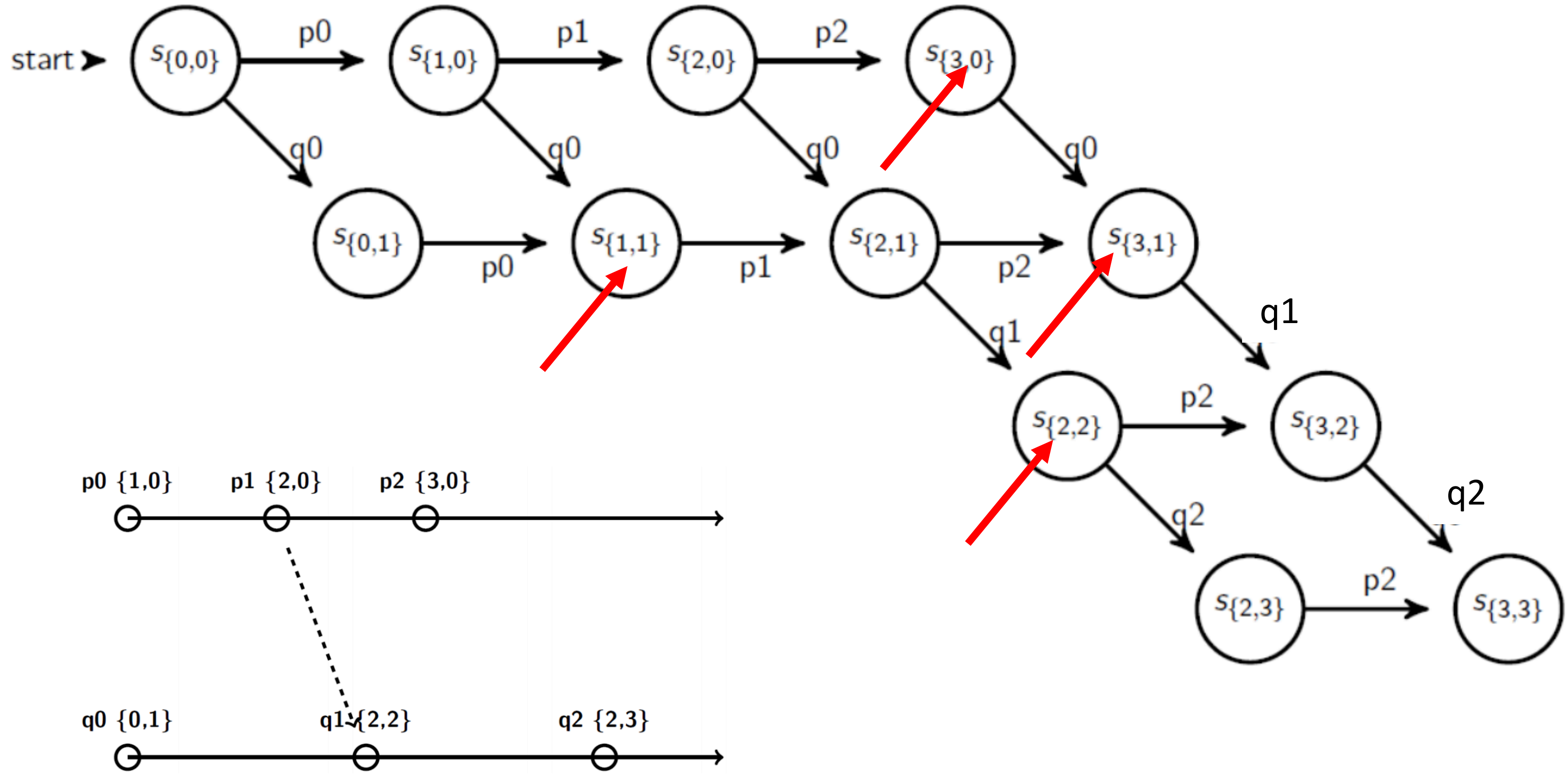
Safety

- **Safety = guarantee that something bad will never happen.**
- **Examples:**
  - There is no deadlock in a distributed transaction system.
  - "Accuracy" in failure detectors: an alive process is not detected as failed.
  - No two processes decide on different values.
- A global state S_0 satisfies a **safety** property P iff:
  - For all states S reachable from S_0, P(S) is true.
  - $\text{safety}(P(S_0)) \equiv \forall S$ reachable from S_0, $P(S) = \text{true}$.

Safety Example

If predicate is true only in the marked states, does it satisfy safety?

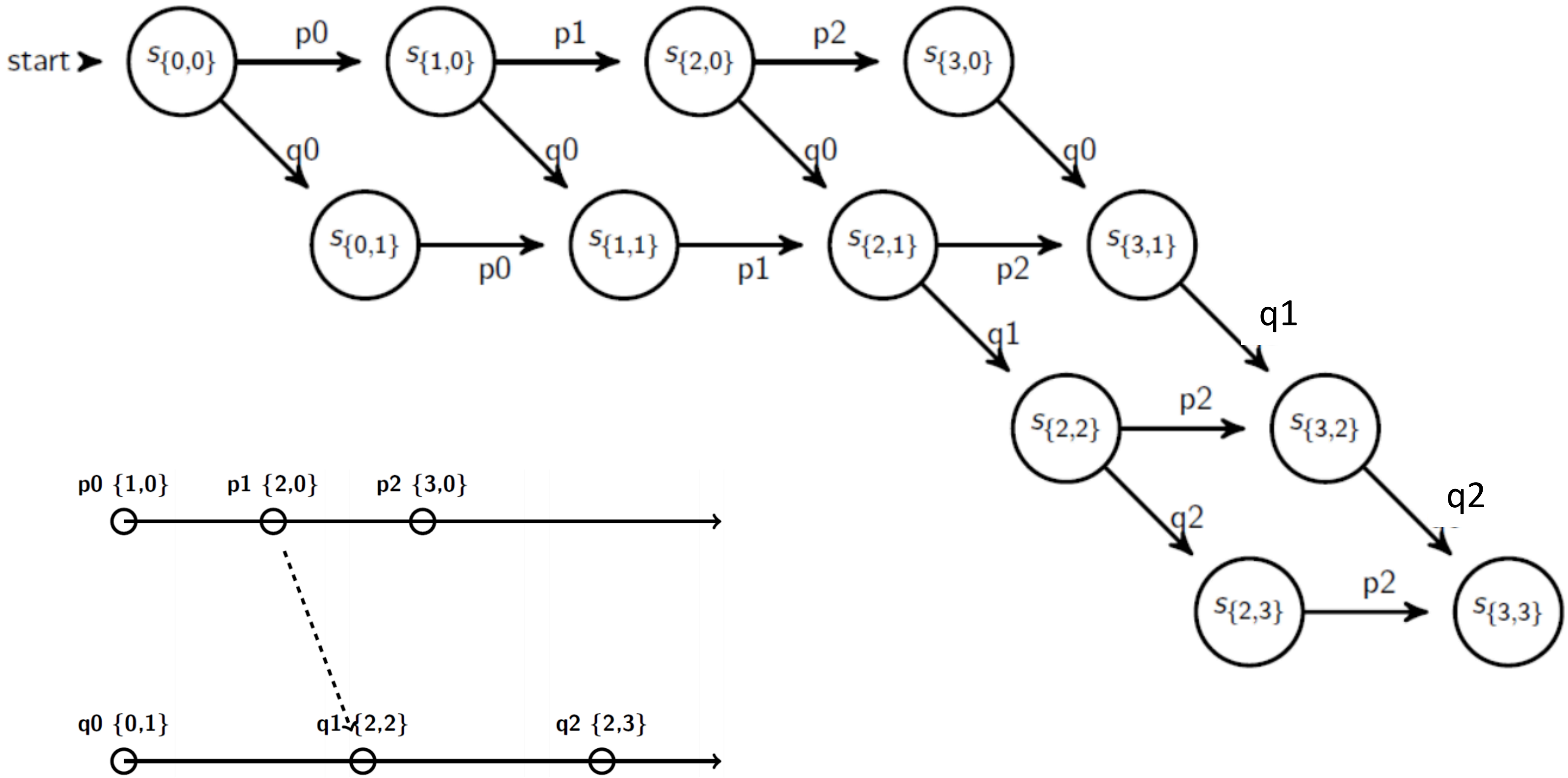
No



Safety Example

If predicate is true only in the **unmarked** states, does it satisfy safety?

Yes



Safety

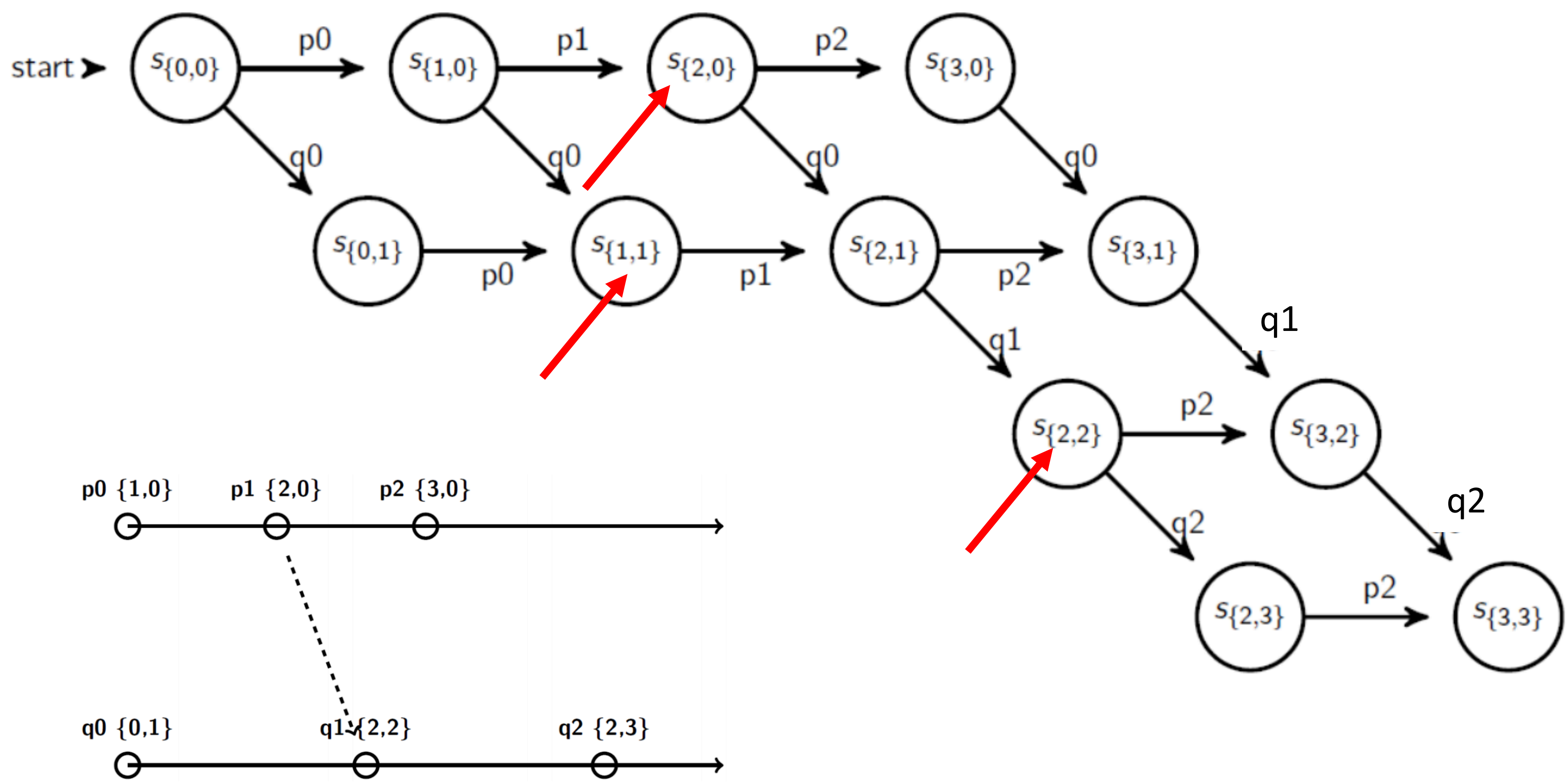
- **Safety = guarantee that something bad will never happen.**

- **Examples:**
  - There is no deadlock in a distributed transaction system.
  - "Accuracy" in failure detectors: an alive process is not detected as failed.
  - No two processes decide on different values.

- A global state S_0 satisfies a **safety** property P iff:
  - safety(P(S_0)) $\equiv \forall S$ reachable from S_0, P(S) = true.
  - For all states S reachable from S_0, P(S) is true.

Liveness Example

Technically satisfies liveness, but difficult to capture or reason about.



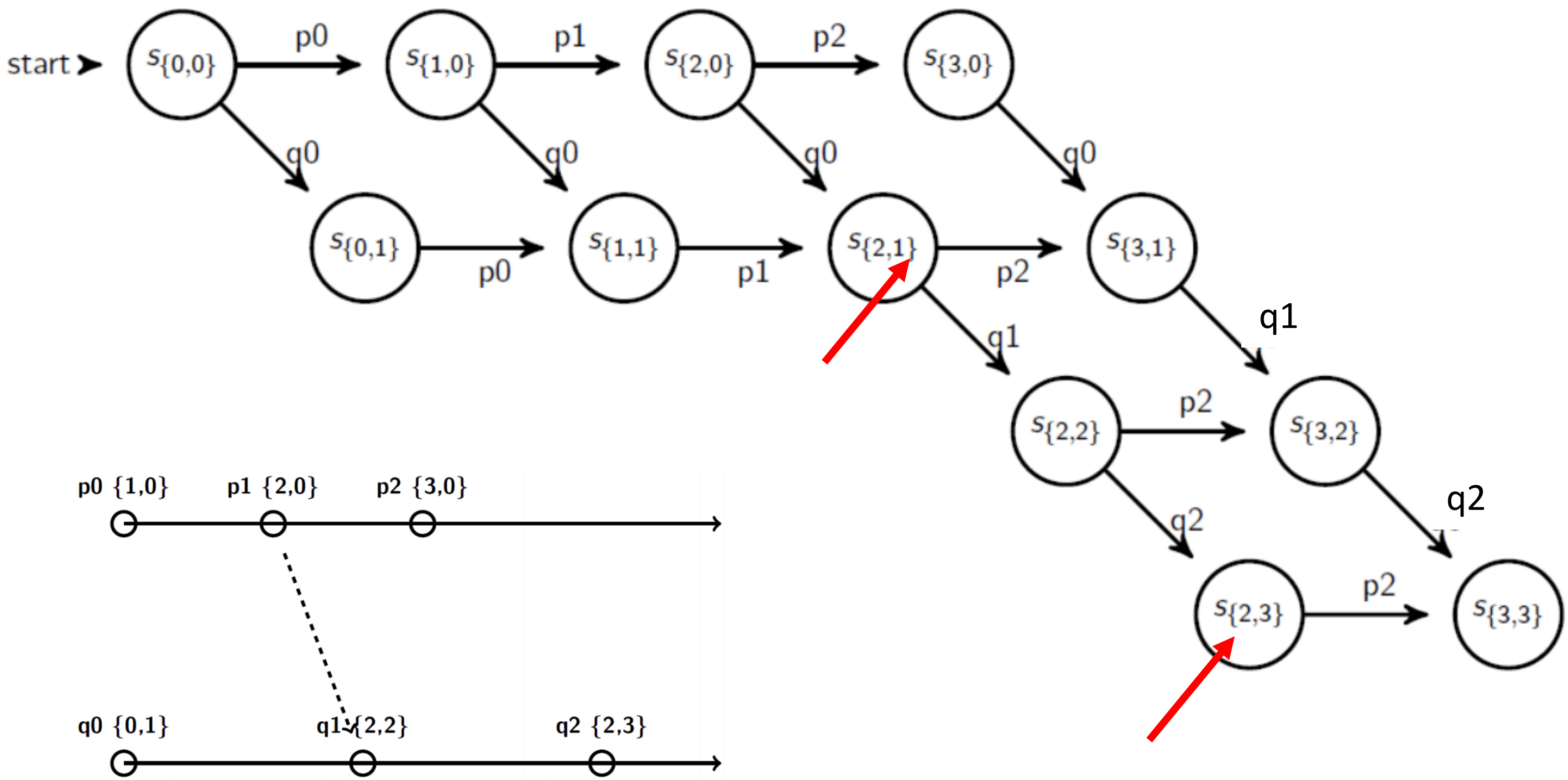
Stable Global Predicates

- once true, stays true forever afterwards (for stable liveness)

Stable Global Predicates

If predicate is true only in the marked states, is it stable?

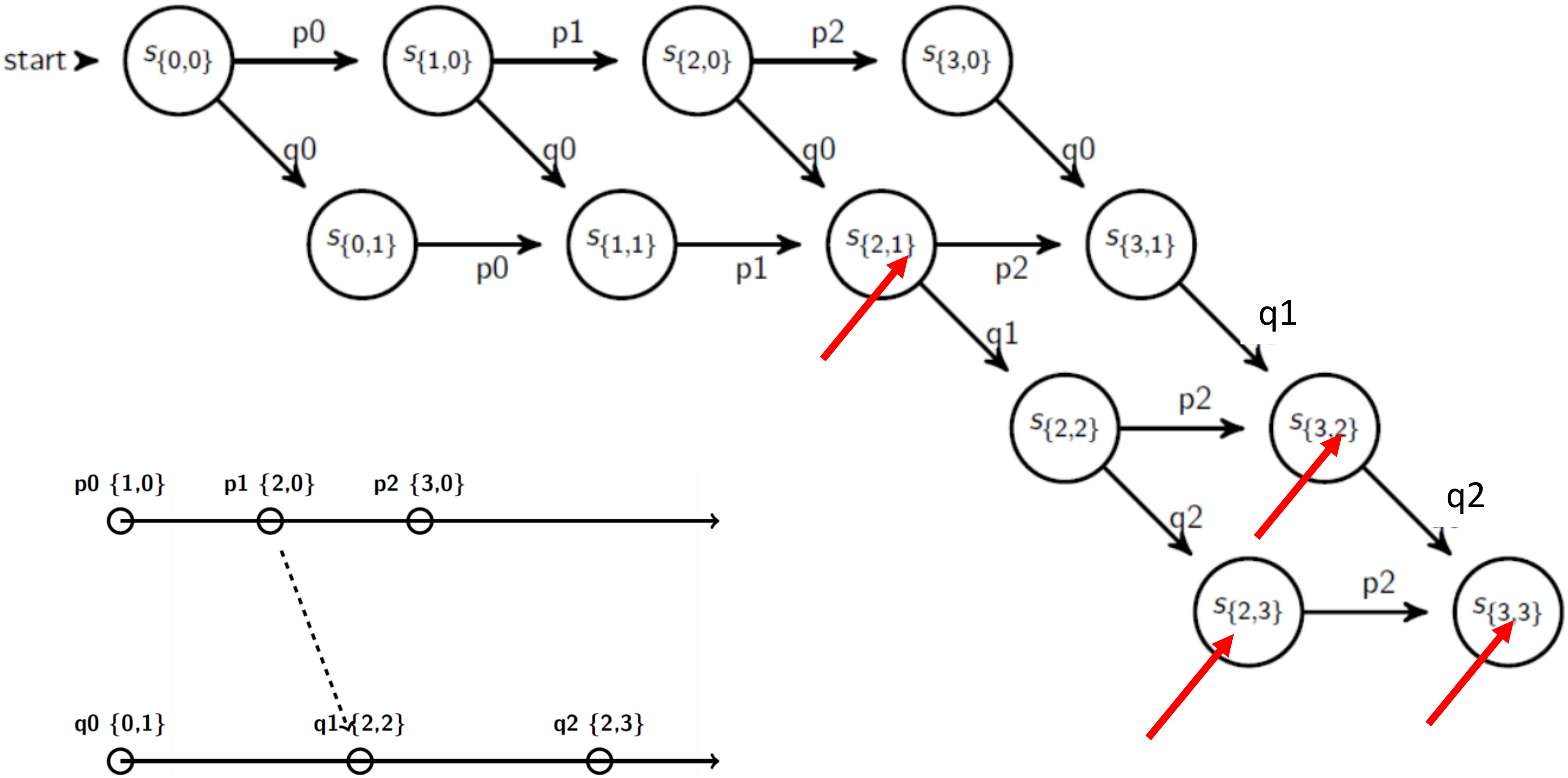
No



Stable Global Predicates

If predicate is true only in the marked states, is it stable?

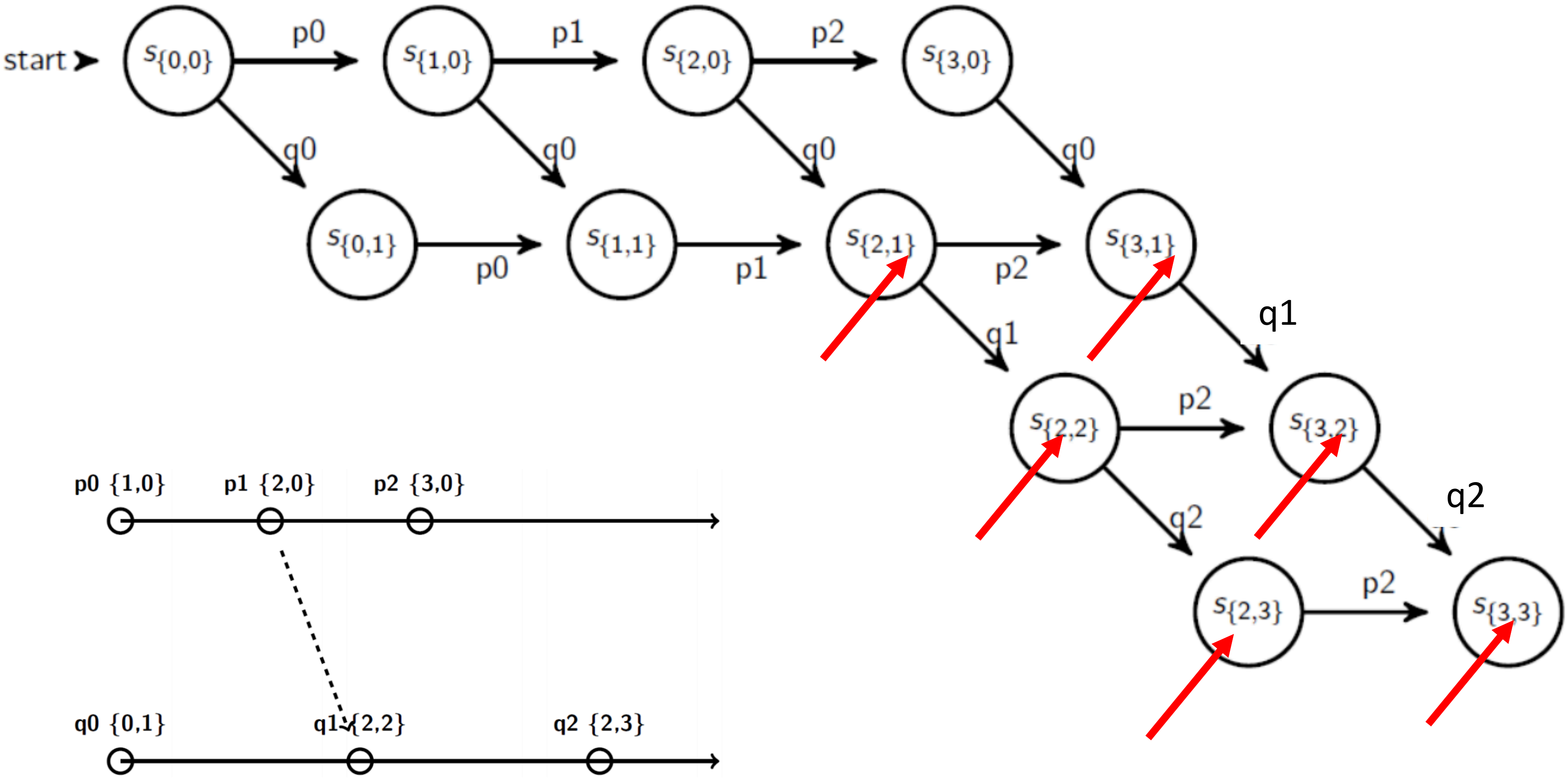
No



Stable Global Predicates

If predicate is true only in the marked states, is it stable?

Yes



Stable Global Predicates

- once true for a state S, stays true for all states reachable from S (for stable liveness)
- once false for a state S, stays false for all states reachable from S (for stable non-safety)
- Stable liveness examples (once true, always true)
  - Computation has terminated.
- Stable non-safety examples (once false, always false)
  - There is no deadlock.
  - An object is not orphaned.
- *All stable global properties can be detected using the Chandy-Lamport algorithm.*

Global Snapshot Summary

- The ability to calculate global snapshots in a distributed system is very important.
- But don't want to interrupt running distributed application.
- Chandy-Lamport algorithm calculates global snapshot.
- Obeys causality (creates a consistent cut).
- Can be used to detect global properties.
- Safety vs. Liveness.

Rest of today's agenda

- **Multicast**
  - Chapter 15.4

- **Goal:** reason about desirable properties for message delivery among a group of processes.

Communication modes

- **Unicast**
  - Messages are sent from exactly <u>one</u> process <u>to</u> <u>one</u> process.
- **Broadcast**
  - Messages are sent from exactly <u>one</u> process <u>to</u> <u>all</u> processes on the network.
- **Multicast**
  - Messages broadcast within a group of processes.
  - A multicast message is sent from any <u>one</u> process <u>to</u> a <u>group</u> of processes on the network.

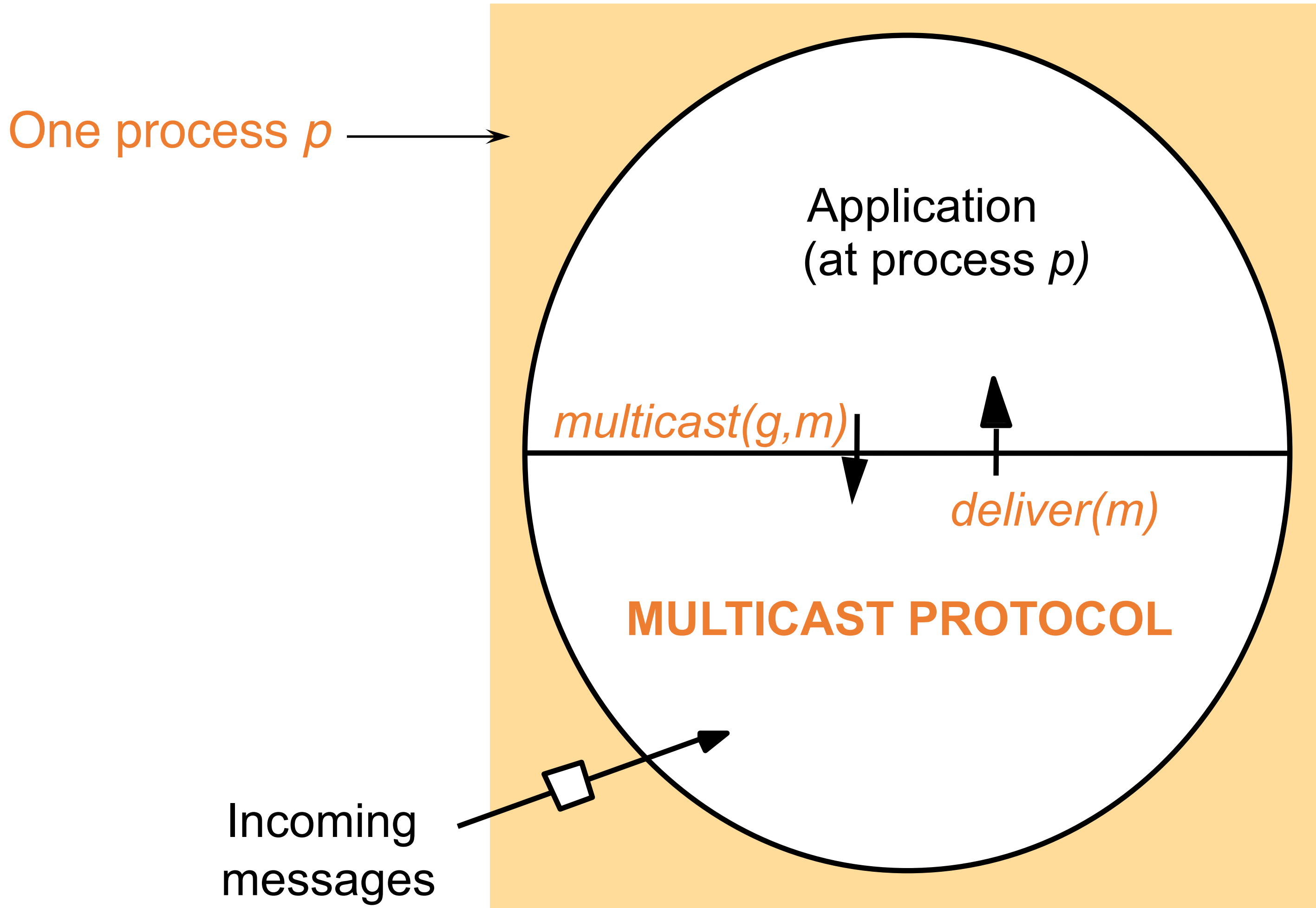
Where is multicast used?

- Distributed storage
  - Write to an object are multicast across replica servers.
  - Membership information (e.g., heartbeats) is multicast across all servers in cluster.
- Online scoreboards (ESPN, French Open, FIFA World Cup)
  - Multicast to group of clients interested in the scores.
- Stock Exchanges
  - Group is the set of broker computers.
- ……

Communication modes

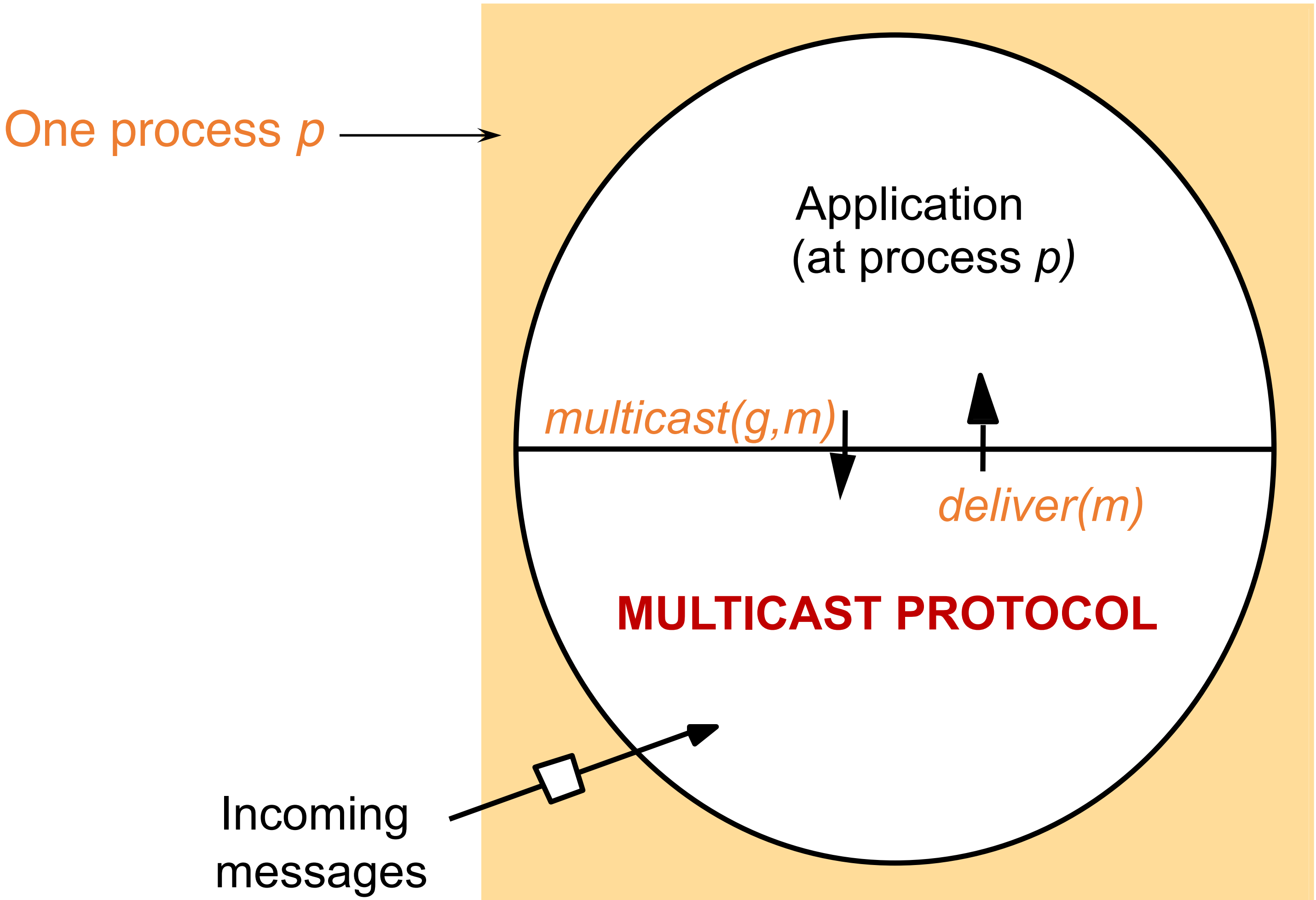
- **Unicast**
  - Messages are sent from exactly one process to one process.
    - *Best effort:* if a message is delivered it would be intact; no reliability guarantees.
    - *Reliable:* guarantees delivery of messages.
    - *In order:* messages will be delivered in the same order that they are sent.
- **Broadcast**
  - Messages are sent from exactly one process to all processes on the network.
- **Multicast**
  - Messages broadcast within a group of processes.
  - A multicast message is sent from any one process to the group of processes on the network.
  - *How do we define (and achieve) reliable or ordered multicast?*

What we are designing in this class?



'g' is a multicast group that also includes the process 'p'.

What we are designing in this class?



'g' is a multicast group that also includes the process 'p'.

Basic Multicast (B-Multicast)

- Straightforward way to implement B-multicast:
  - use a reliable one-to-one send (unicast) operation:

    B-multicast(group g, message m):

    for each process p in g, send (p,m).

    receive(m): B-deliver(m) at p.
- Guarantees: message is eventually delivered to the group if:
  - Processes are non-faulty.
  - The unicast "send" is reliable.
  - *Sender does not crash.*
- *Can we provide reliable delivery even after sender crashes?*
  - *What does this mean?*

Reliable Multicast (R-Multicast)

- **Integrity**: A *correct* (i.e., non-faulty) process *p* delivers a message *m* at most once.
  - *Assumption: no process sends **exactly** the same message twice*
- **Validity**: If a *correct* process multicasts (sends) message *m*, then it will eventually deliver *m* to itself.
  - *Liveness for the sender.*
- **Agreement**: If a *correct* process delivers message *m*, then all the other *correct* processes in group(*m*) will eventually deliver *m*.
  - *All or nothing.*
- Validity and agreement together ensure overall liveness: if some correct process multicasts a message *m*, then, all correct processes deliver *m* too.

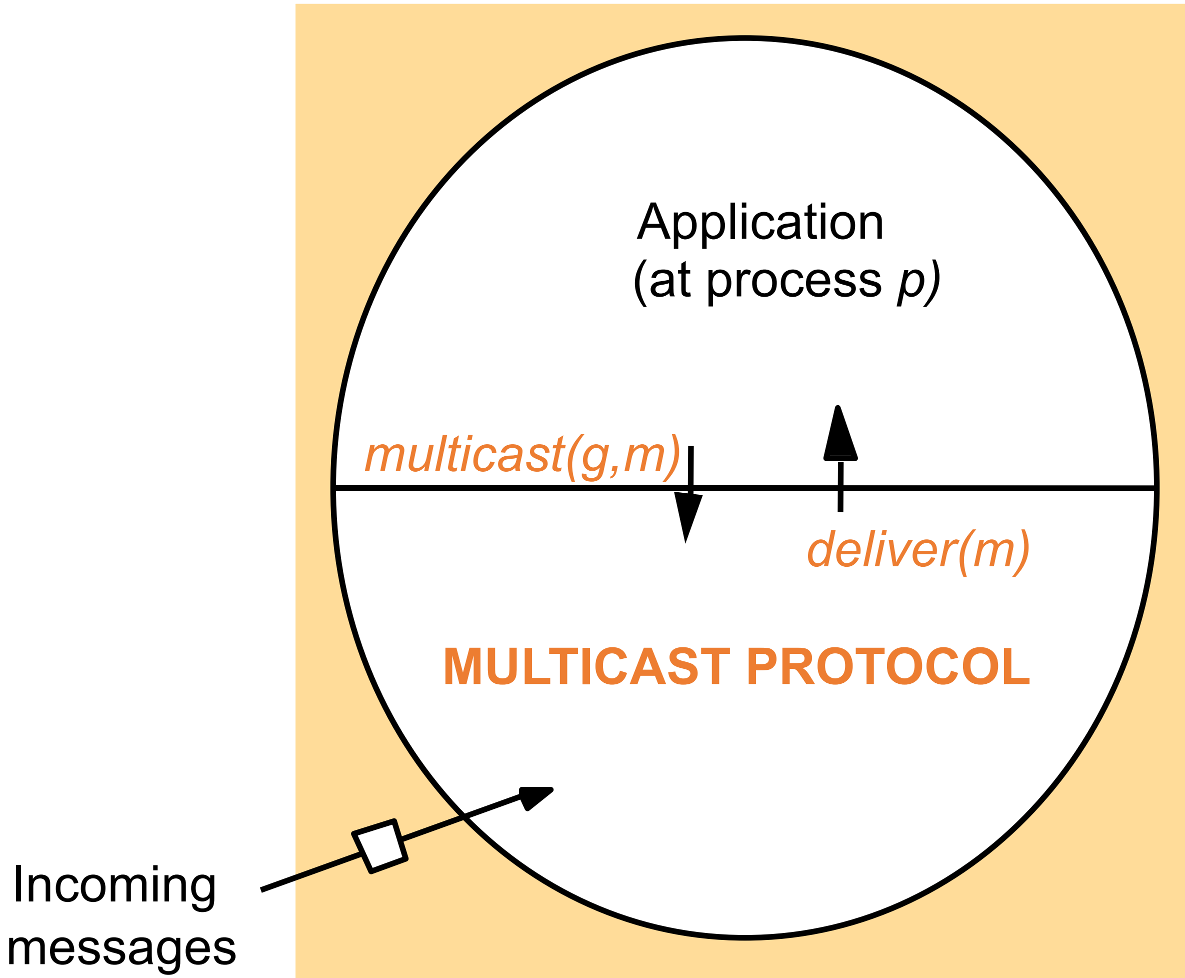
Reliable Multicast (R-Multicast)

- **Integrity**: A *correct* (i.e., non-faulty) process *p* delivers a message *m* at most onc
  - *Assur* *wice*
- **Validity**: I hen it will eventuall
  - *Livene*
- **Agreeme** the other *correct* pr
  - *All or*
- Validity and agreement together ensure overall liveness: if some correct process multicasts a message *m*, then, all correct processes deliver *m* too.

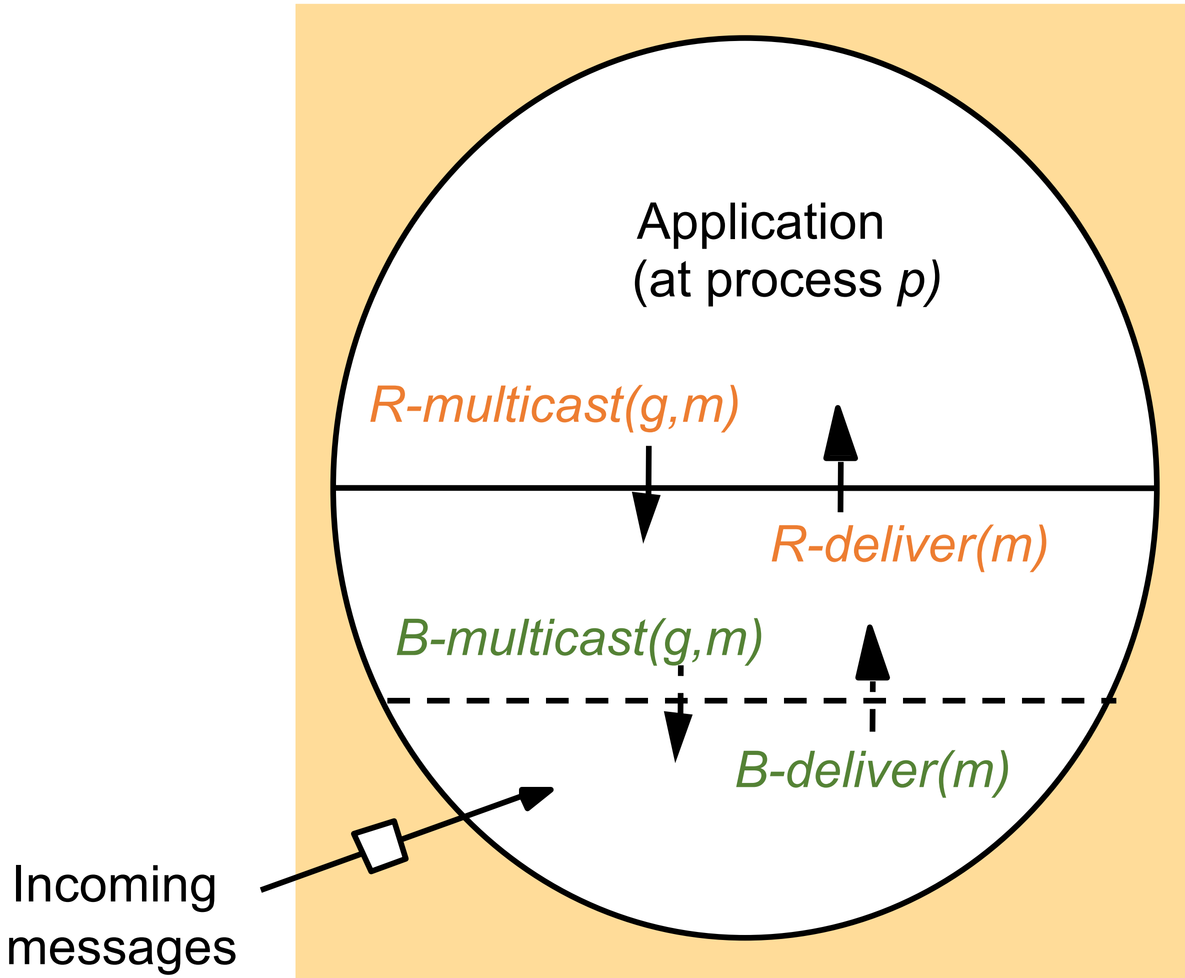
What happens if a process initiates B-multicasts of a message but fails after unicasting to a subset of processes in the group?

Agreement is violated! R-multicast not satisfied.

Implementing R-Multicast



Implementing R-Multicast



Implementing R-Multicast

On initialization

Received := {};

For process p to R-multicast message m to group g

B-multicast(g,m); (p $\in$ g is included as destination)

On B-deliver(m) at process q in g = group(m)

if (m $\notin$ Received):

Received := Received $\cup$ {m};

if (q $\neq$ p): B-multicast(g,m);

R-deliver(m)

Reliable Multicast (R-Multicast)

- **Integrity**: A *correct* (i.e., non-faulty) process *p* delivers a message *m* at most once.
  - *Assumption: no process sends **exactly** the same message twice*
- **Validity**: If a *correct* process multicasts (sends) message *m*, then it will eventually deliver *m* to itself.
  - *Liveness for the sender.*
- **Agreement**: If a *correct* process delivers message *m*, then all the other *correct* processes in group(*m*) will eventually deliver *m*.
  - *All or nothing.*
- Validity and agreement together ensure overall liveness: if some correct process multicasts a message *m*, then, all correct processes deliver *m* too.

Ordered Multicast

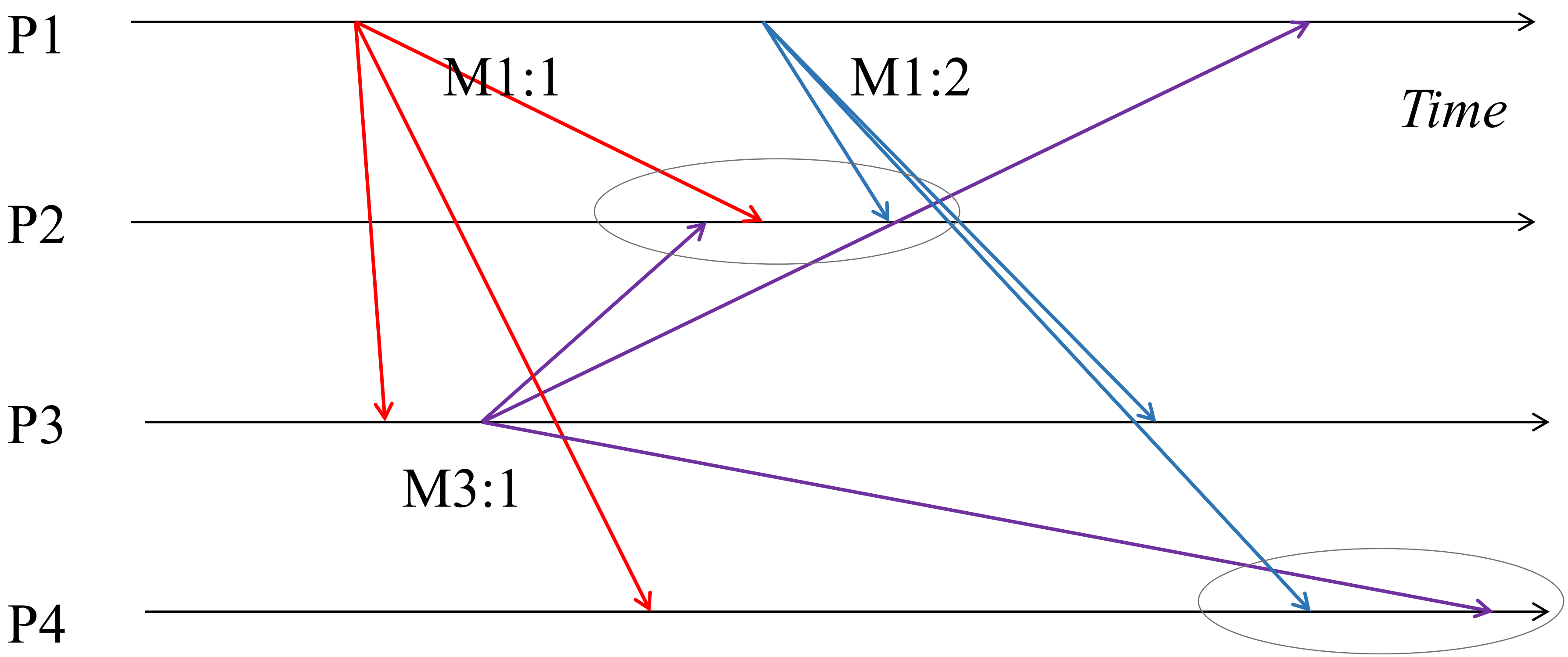
- Three popular flavors implemented by several multicast protocols:
    1. FIFO ordering
    2. Causal ordering
    3. Total ordering

1. FIFO Order

- Multicasts from each sender are delivered in the order they are sent, at all receivers.
- Don't care about multicasts from different senders.

- More formally
  - *If a correct process issues multicast(g,m) and then multicast(g,m'), then every correct process that delivers m' will have already delivered m.*

FIFO Order: Example



M1:1 and M1:2 should be delivered in that order at each receiver.
Order of delivery of M3:1 and M1:2 could be different at different receivers.

2. Causal Order

- Multicasts whose send events are causally related, must be delivered in the same causality-obeying order at all receivers.

- More formally
  - *If multicast(g,m) $\rightarrow$ multicast(g,m') then any correct process that delivers m' will have already delivered m.*
  - $\rightarrow$ *is Lamport's happens-before*
  - $\rightarrow$ is induced only by multicast messages in group g, and when they are **delivered** to the application, rather than all network messages.

Where is causal ordering useful?

- Group = set of your friends on a social network.

- A friend sees your message *m*, and she posts a response (comment) *m'* to it.
  - If friends receive *m'* before *m*, it wouldn't make sense
  - But if two friends post messages m'' and n'' concurrently, then they can be seen in any order at receivers.

- A variety of systems implement causal ordering:
  - social networks, bulletin boards, comments on websites, etc.

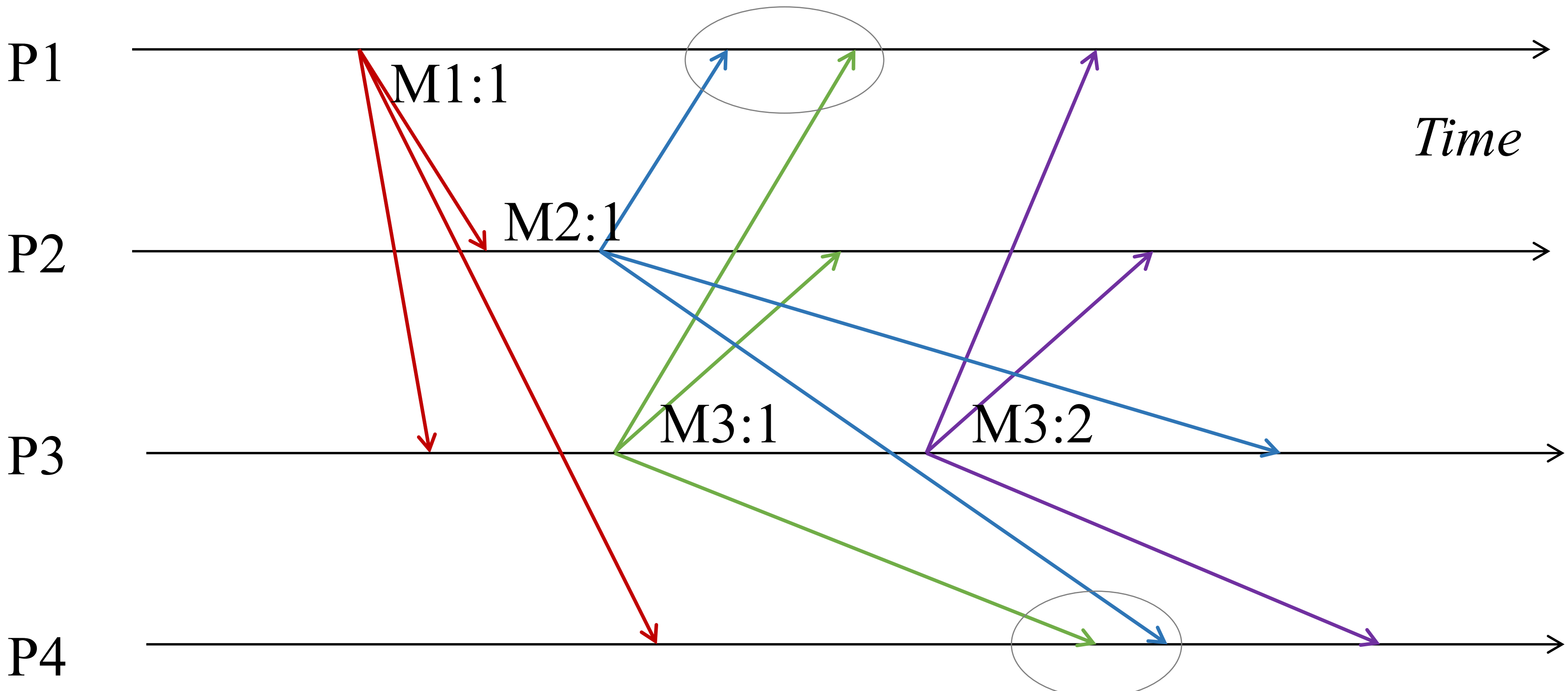
HB Relationship for Causal Ordering

- HB rules in causal ordered multicast:
  - If $\exists$ $\mathbf{p_i}$, **e** $\rightarrow_i$ **e'** then **e** $\rightarrow$ **e'**.
    - If $\exists$ $\mathbf{p_i}$, multicast(g,m) $\rightarrow_i$ multicast(g,m'), then multicast(g,m) $\rightarrow$ multicast(g,m')
    - If $\exists$ $\mathbf{p_i}$, delivery(m) $\rightarrow_i$ multicast(g,m'), then delivery(m) $\rightarrow$ multicast(g,m')
    - …
  - For any message m, **send(m)** $\rightarrow$ **receive(m)**

HB Relationship for Causal Ordering

- HB rules in causal ordered multicast:
  - If $\exists$ $\mathbf{p_i}$, **e** $\rightarrow_i$ **e'** then **e** $\rightarrow$ **e'**.
    - If $\exists$ $\mathbf{p_i}$, multicast(*g*,*m*) $\rightarrow_i$ multicast(*g*,*m'*), then multicast(*g*,*m*) $\rightarrow$ multicast(*g*,*m'*)
    - If $\exists$ $\mathbf{p_i}$, delivery(*m*) $\rightarrow_i$ multicast(*g*,*m'*), then delivery(*m*) $\rightarrow$ multicast(*g*,*m'*)
    - …
  - ~~For any message m, **send(m)** $\rightarrow$ **receive(m)**~~
    - For any *multicast* message m, multicast(*g*,*m*) $\rightarrow$ delivery(m)
  - If **e** $\rightarrow$ **e'** and **e'** $\rightarrow$ **e''** then **e** $\rightarrow$ **e''**
    - multicast(*g*,*m*) at p_i $\rightarrow$ delivery(m) at p_j
    - delivery(*m*) at p_j $\rightarrow$ multicast(*g*,*m'*) at p_j
    - multicast(*g*,*m*) at p_i $\rightarrow$ multicast(*g*,*m'*) at p_j
- *Application can only see when messages are "multicast" by the application and "delivered" to the application, and not when they are sent or received by the protocol.*

Causal Order: Example



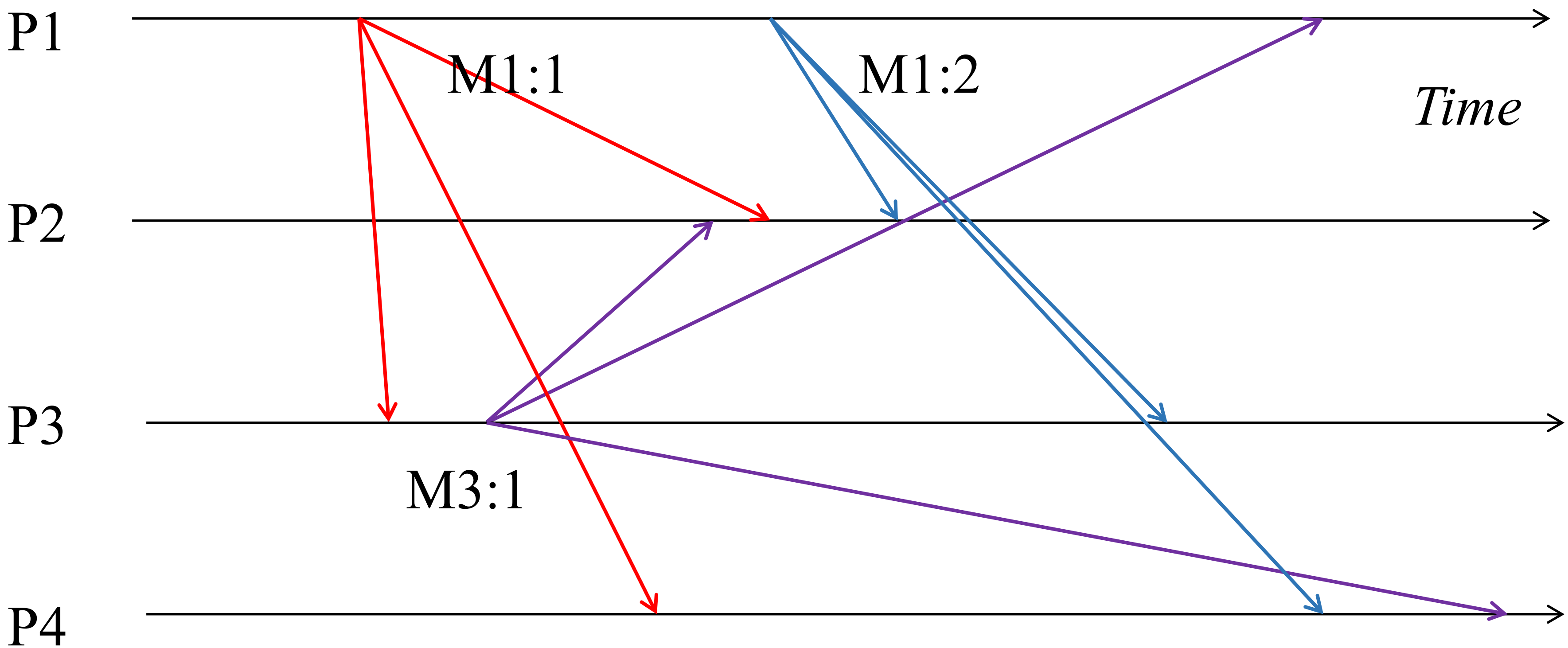
M3:1 → M3:2, M1:1 → M2:1, M1:1 → M3:1 and so should be delivered in that order at each receiver.

M3:1 and M2:1 are concurrent and thus ok to be delivered in any (and even different) orders at different receivers.

Causal vs FIFO

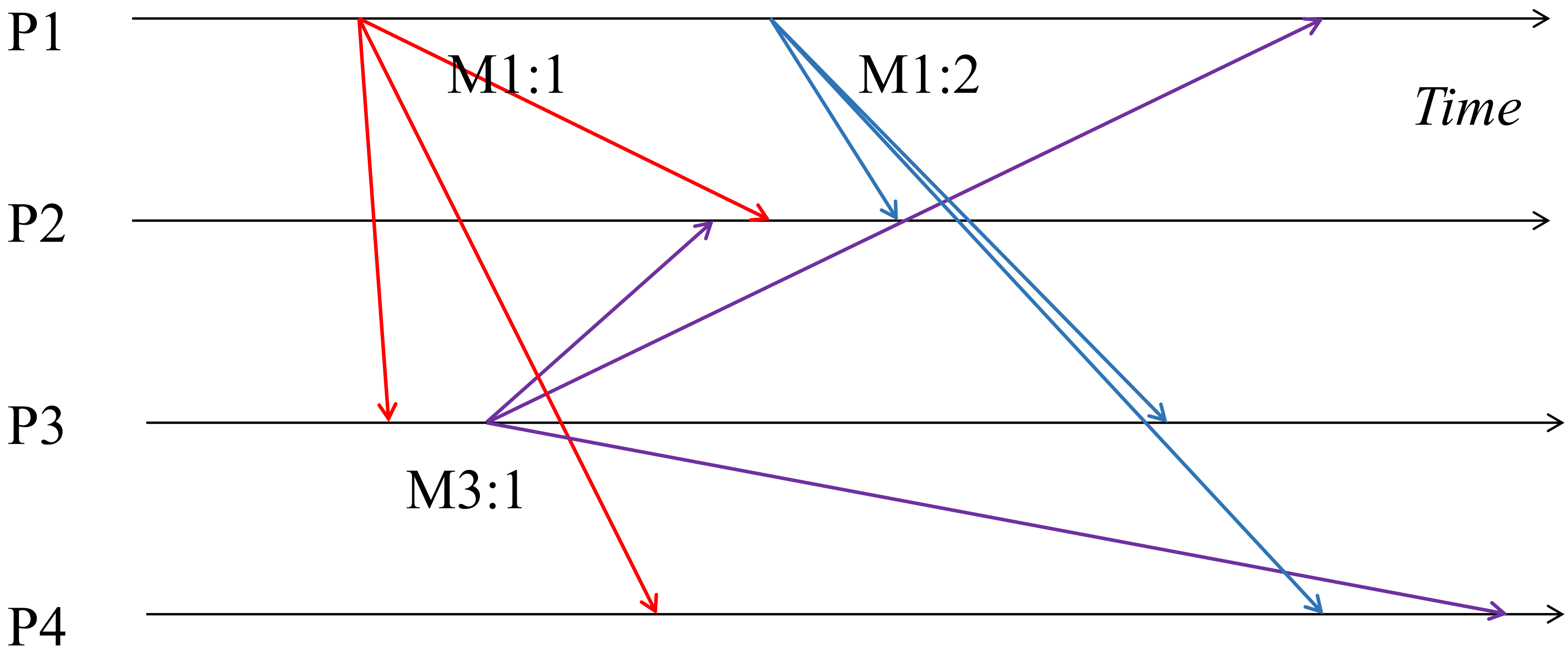
- Does Causal Ordering imply FIFO Ordering?
  - Yes
- Does FIFO Order imply Causal Order?
  - No

Example



Does this satisfy FIFO order?

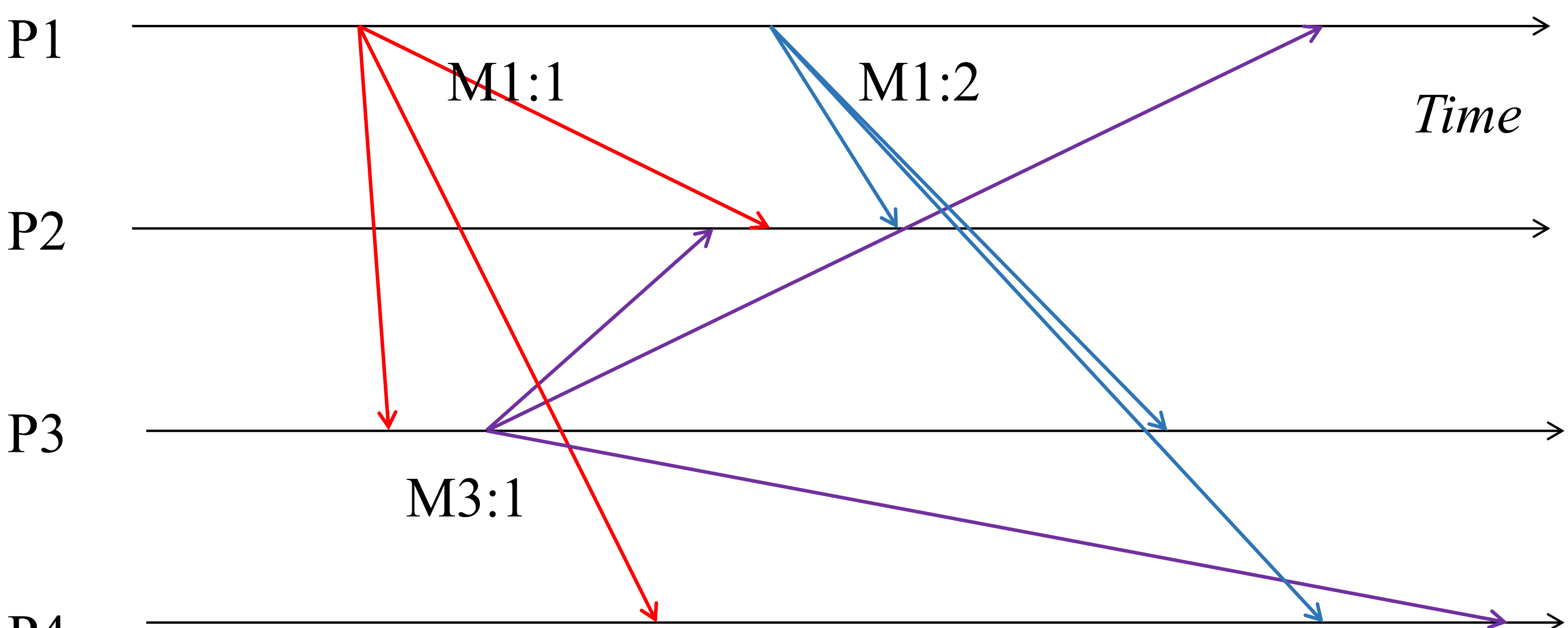
Example



Does this satisfy FIFO order?

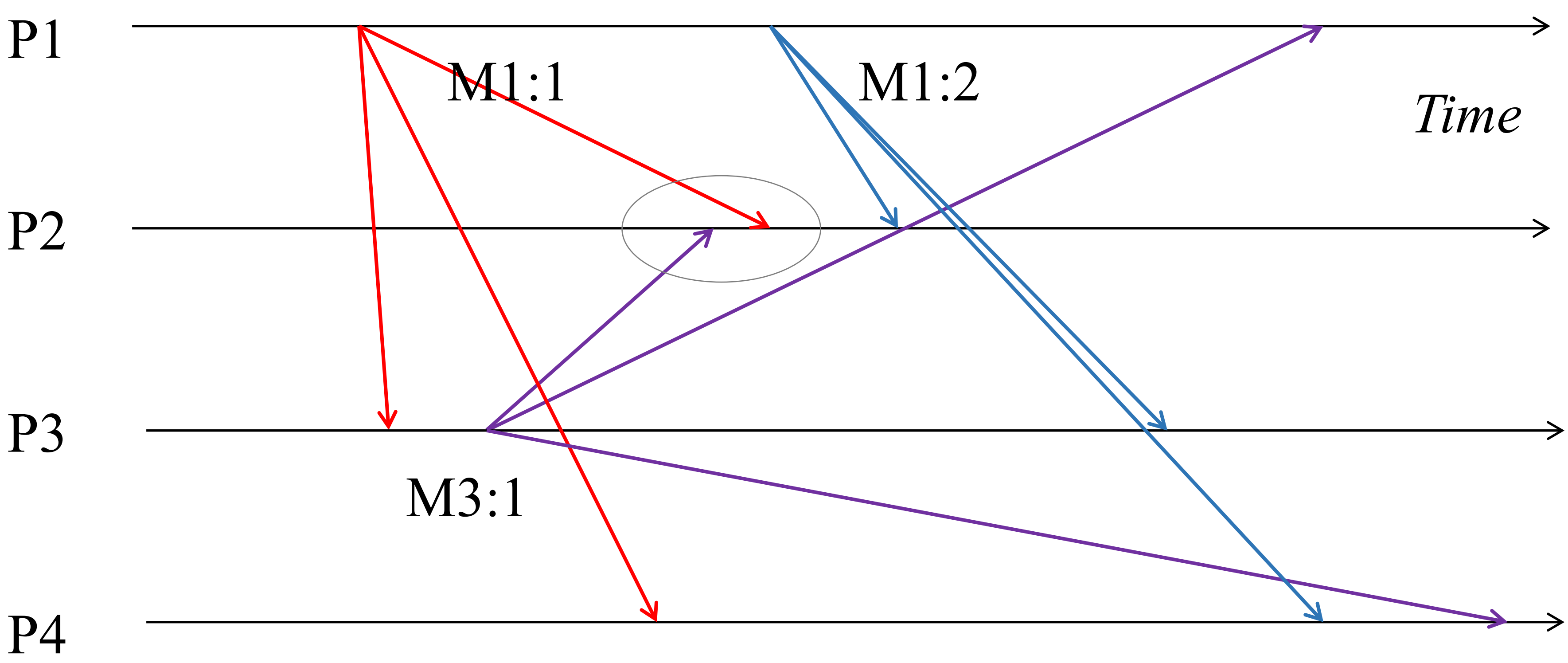
Yes

Example



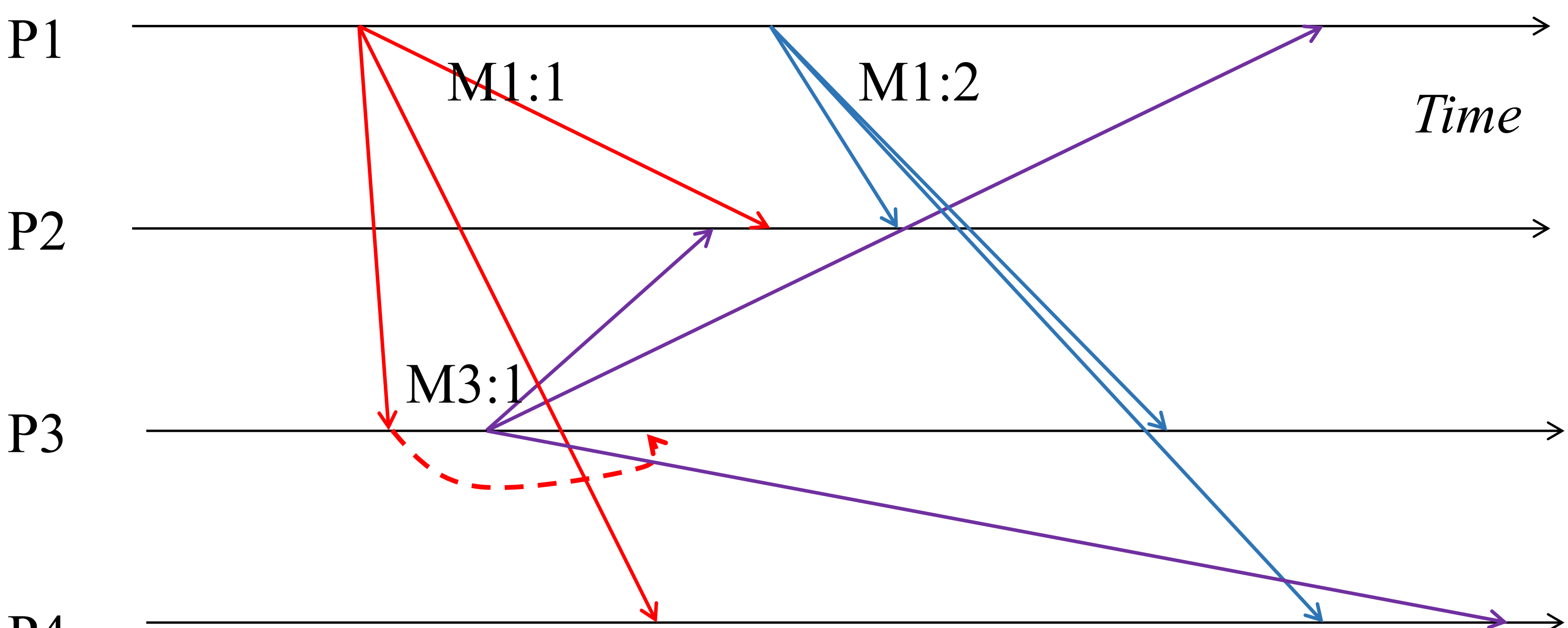
Does this satisfy causal order?

Example



Does this satisfy causal order?
No

Example

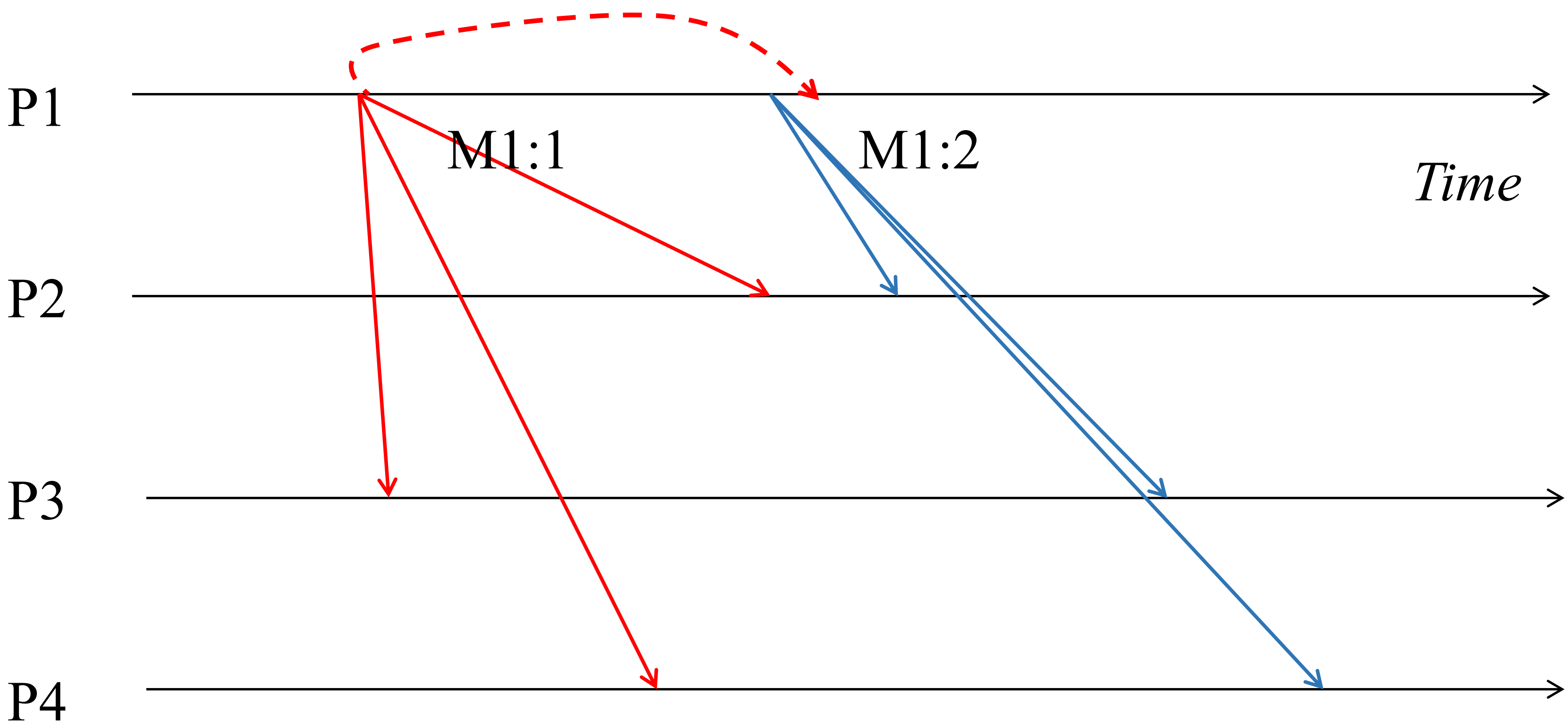


M1:1 is delivered at P3 after M3:1's multicast.

Does this satisfy causal order?

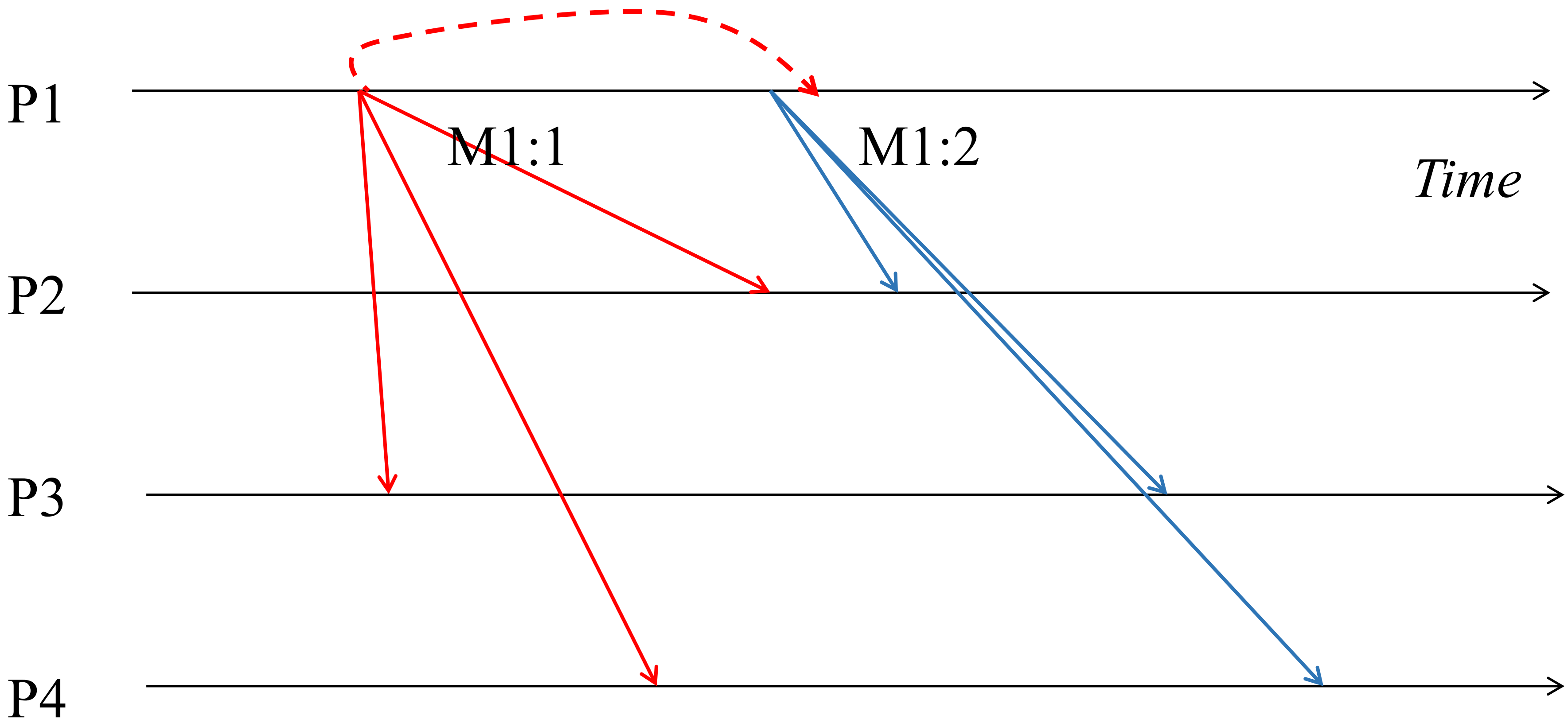
Yes

Example



Does this satisfy causal order?
No

Example

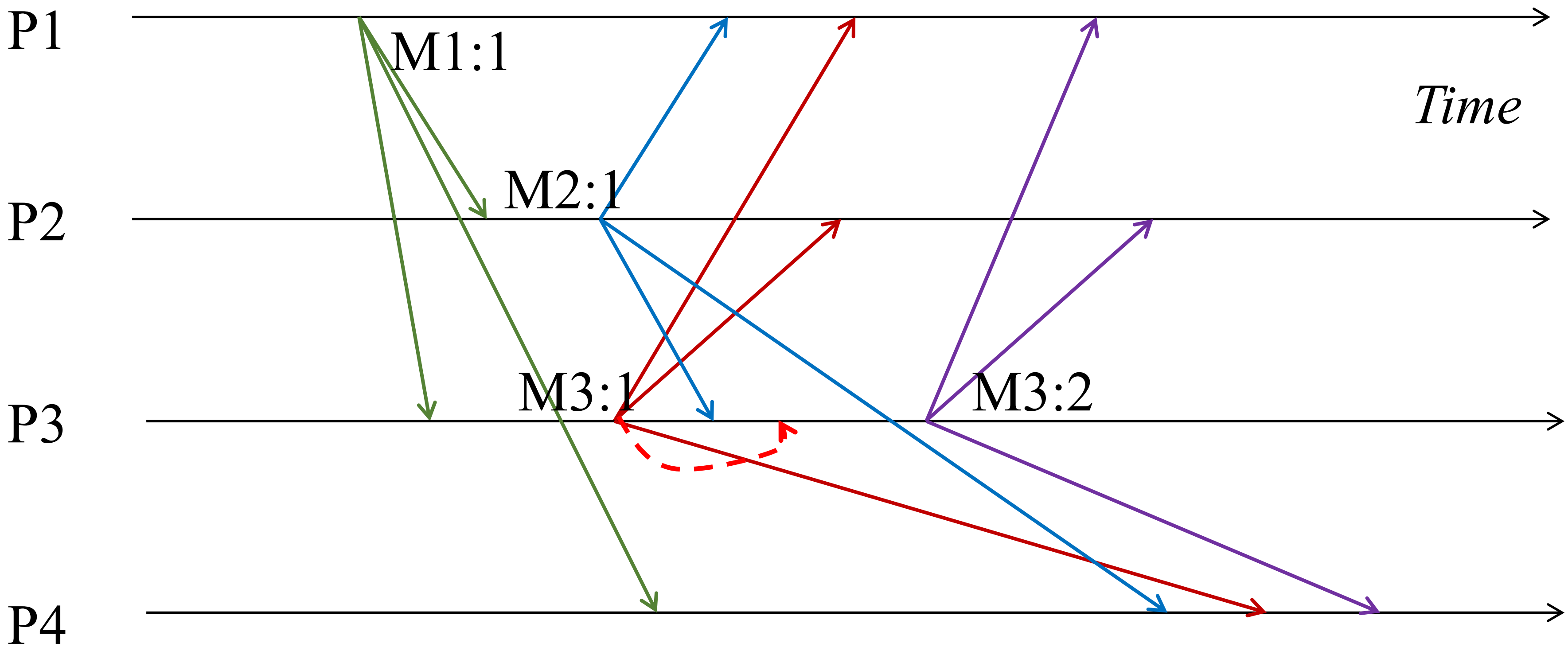


Does this satisfy FIFO order?
No

3. Total Order

- Ensures all processes deliver all multicasts in the same order.
- Unlike FIFO and causal, this does not pay attention to order of multicast sending.
- Formally
  - If a correct process delivers message *m* before *m'* (independent of the senders), then any other correct process that delivers *m'* will have already delivered *m*.

Total Order: Example



The order of receipt of multicasts is the same at all processes.
M1:1, then M2:1, then M3:1, then M3:2
May need to delay delivery of some messages.

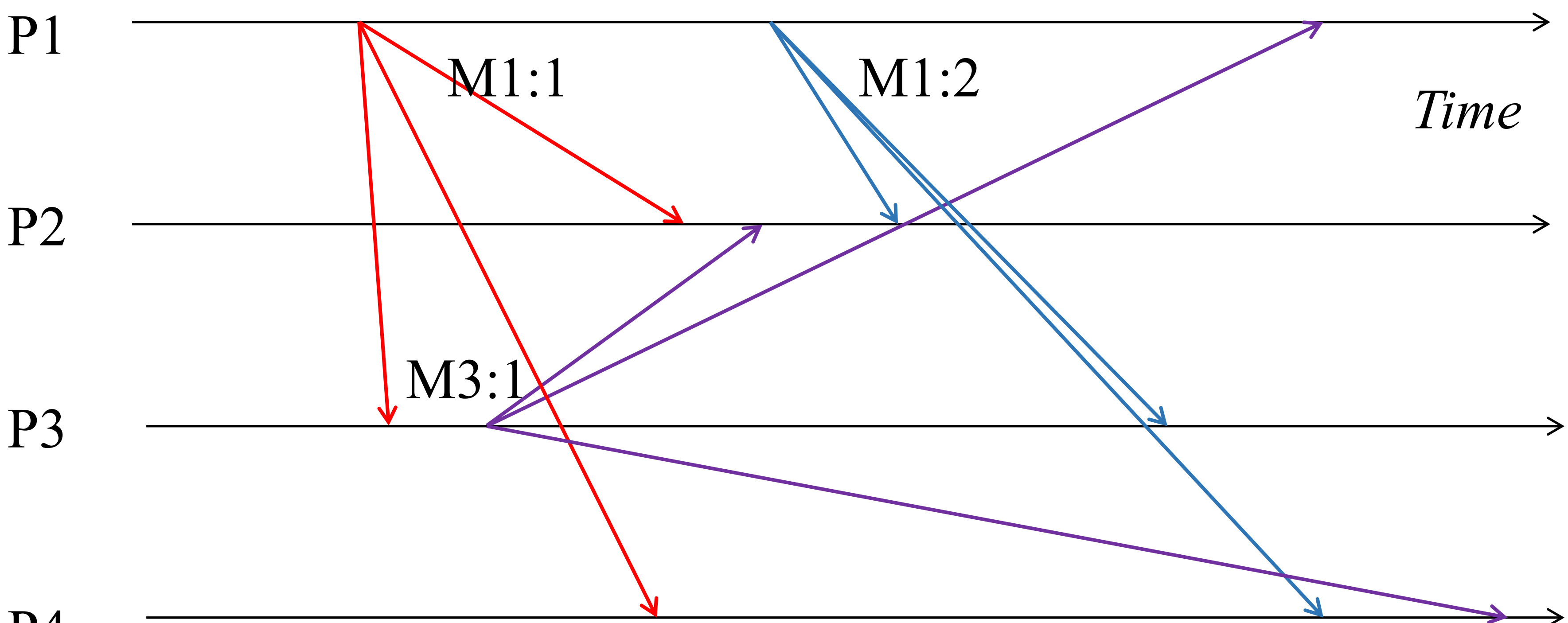
Causal vs Total

- Total ordering does not imply causal ordering.
- Causal ordering does not imply total ordering.

Hybrid variants

- We can have hybrid ordering protocols:
    - Causal-total hybrid protocol satisfies both Causal and total orders.

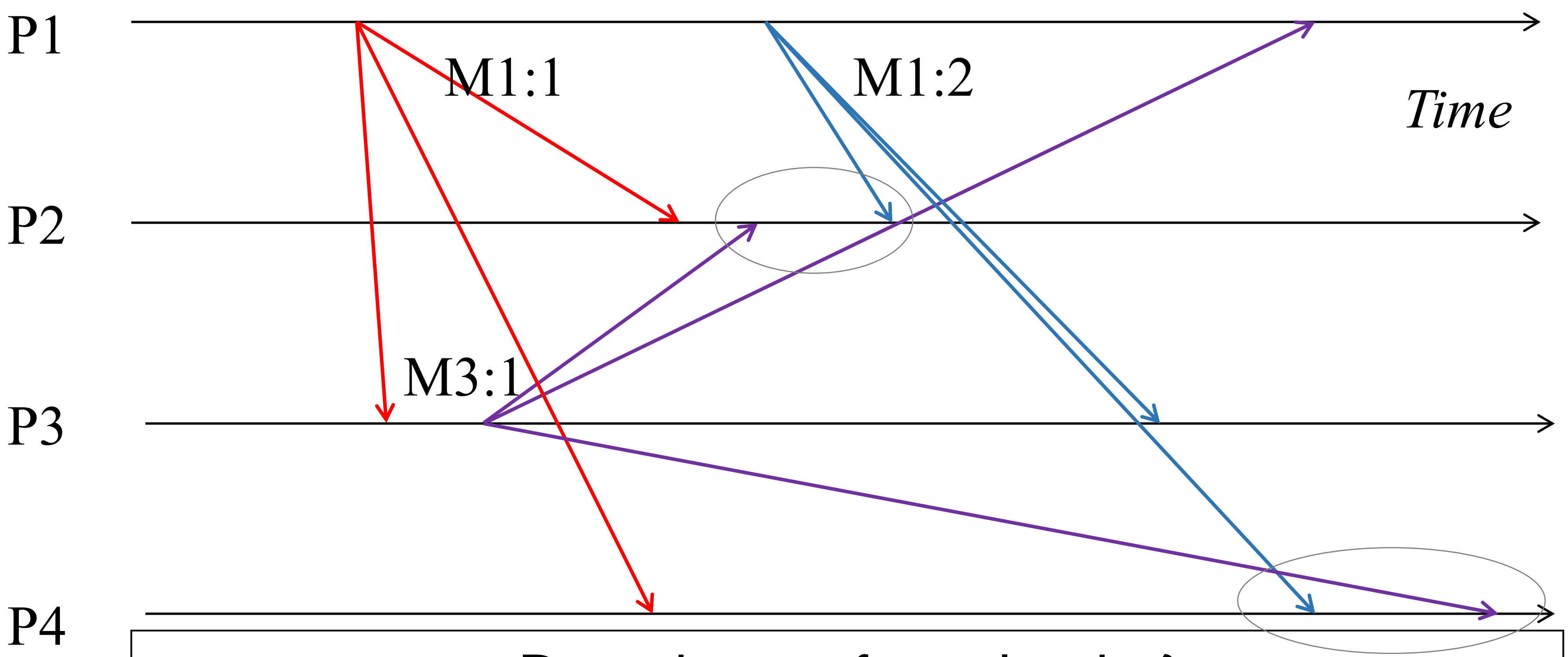
Example



Does this satisfy causal (and FIFO) order?

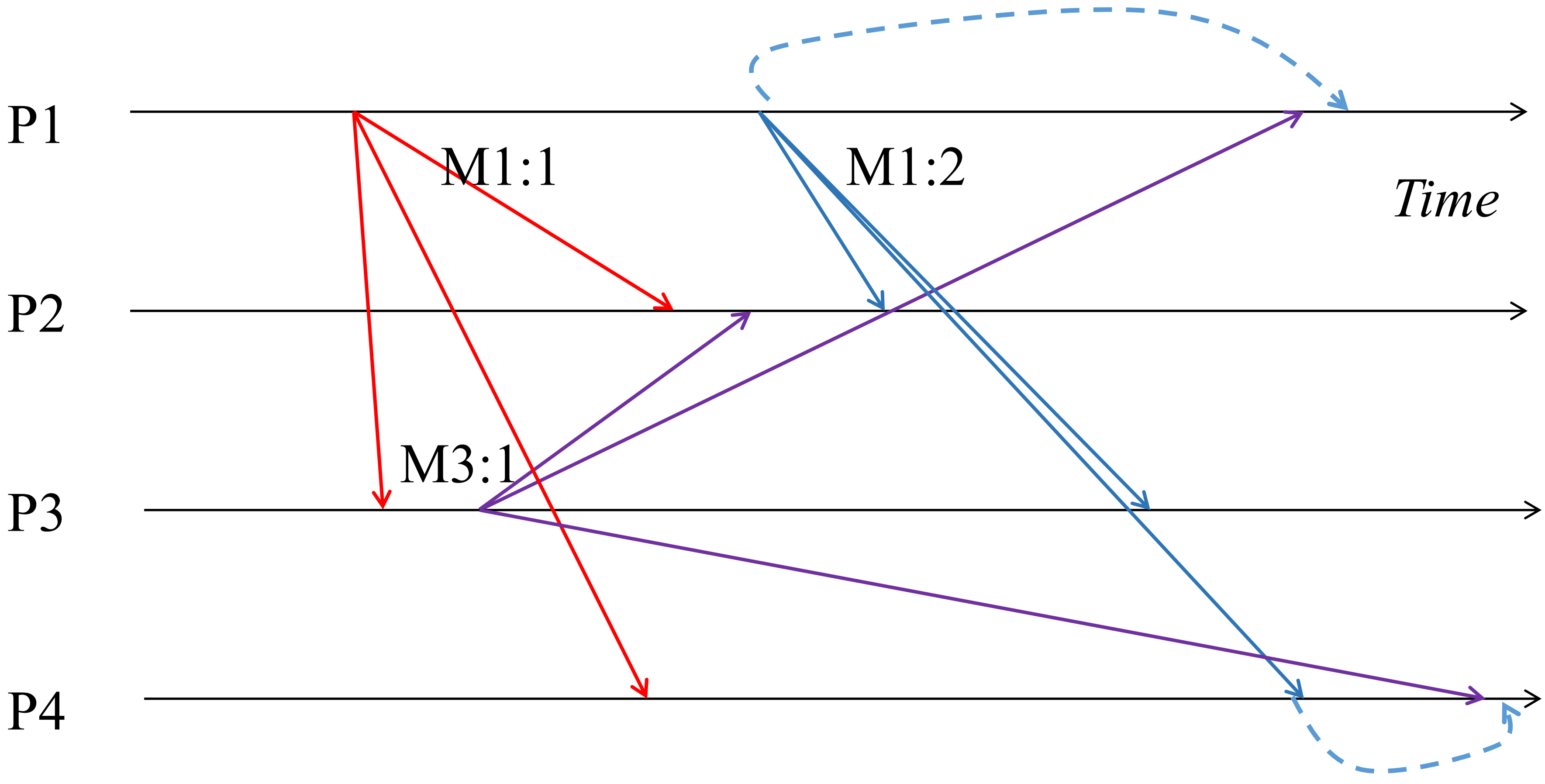
Yes

Example



Does this satisfy total order?
No

Example



Does this satisfy total order?
Yes

Ordered Multicast

- **FIFO ordering:** If a correct process issues multicast(*g*,*m*) and then multicast(*g*,*m'*), then every correct process that delivers *m'* will have already delivered m.

- **Causal ordering:** If multicast(*g*,*m*) → multicast(*g*,*m'*) then any correct process that delivers *m'* will have already delivered *m*.
  - Note that → counts messages **delivered** to the application, rather than all network messages.

- **Total ordering**: If a correct process delivers message *m* before *m'* (independent of the senders), then any other correct process that delivers *m'* will have already delivered *m*.

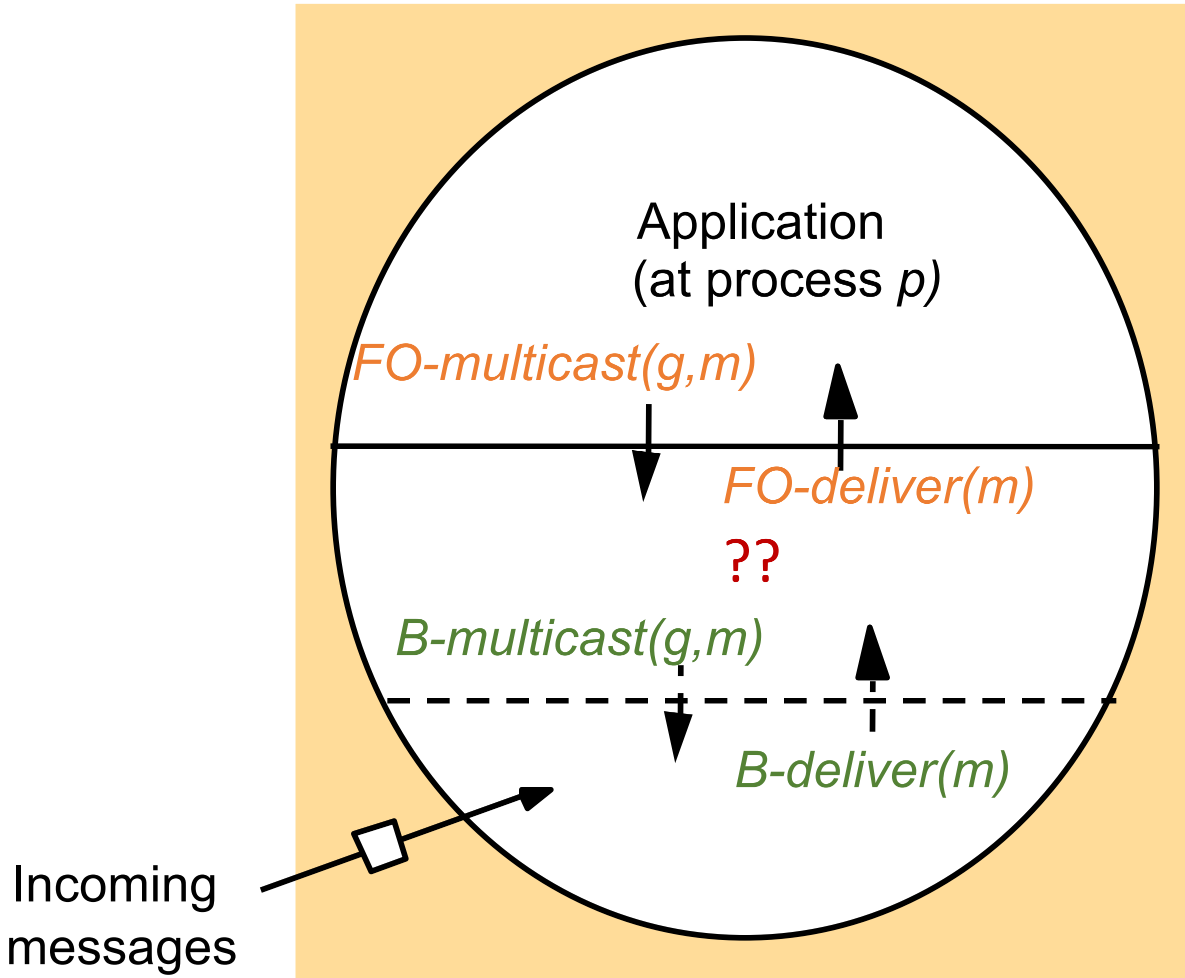
Next Question

How do we implement ordered multicast?

Ordered Multicast

- FIFO ordering
  - If a correct process issues multicast(g,m) and then multicast(g,m'), then every correct process that delivers m' will have already delivered m.
- Causal ordering
  - If multicast(g,m) → multicast(g,m') then any correct process that delivers m' will have already delivered m.
  - Note that → counts messages **delivered** to the application, rather than all network messages.
- Total ordering
  - If a correct process delivers message m before m' (independent of the senders), then any other correct process that delivers m' will have already delivered m.

Implementing FIFO order multicast



Implementing FIFO order multicast

- Each receiver maintains a per-sender sequence number
    - Processes P*1* through P*N*
    - P*i* maintains a vector of sequence numbers P*i*[1…N] (initially all zeroes)
    - P*i*[*j*] is the latest sequence number P*i* has received from P*j*

Implementing FIFO order multicast

- On FO-multicast(g,m) at process P*j*:

  set P*j*[*j*] = P*j*[*j*] + 1

  piggyback P*j*[*j*] with m as its sequence number.

  B-multicast(g,{m, P*j*[*j*]})

- On B-deliver({m, S}) at Pi from Pj: *If Pi receives a multicast from Pj with sequence number S in message*

  if (S == P*i*[*j*] + 1) then

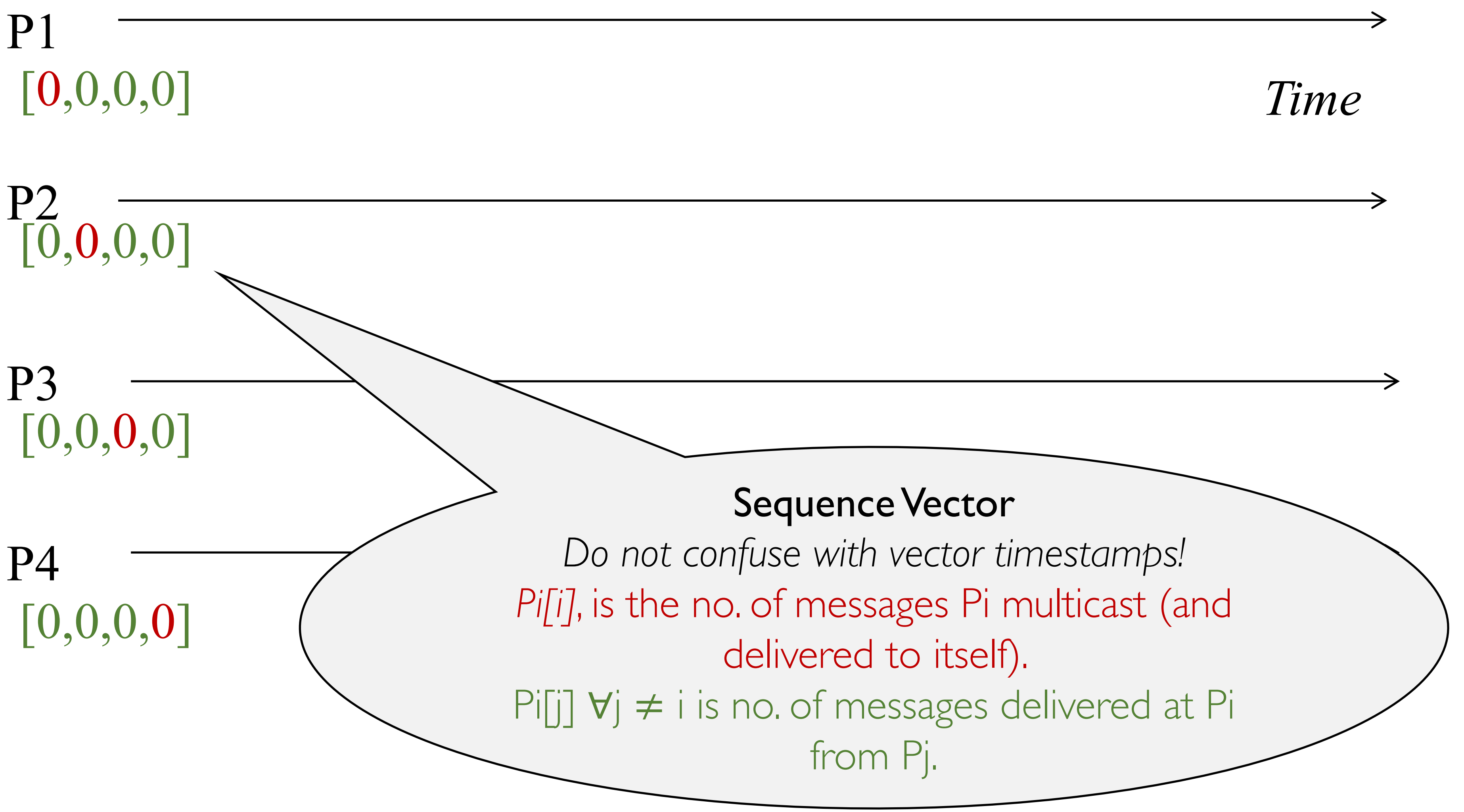
    FO-deliver(m) to application

    set P*i*[*j*] = P*i*[*j*] + 1

  else buffer this multicast until above condition is true

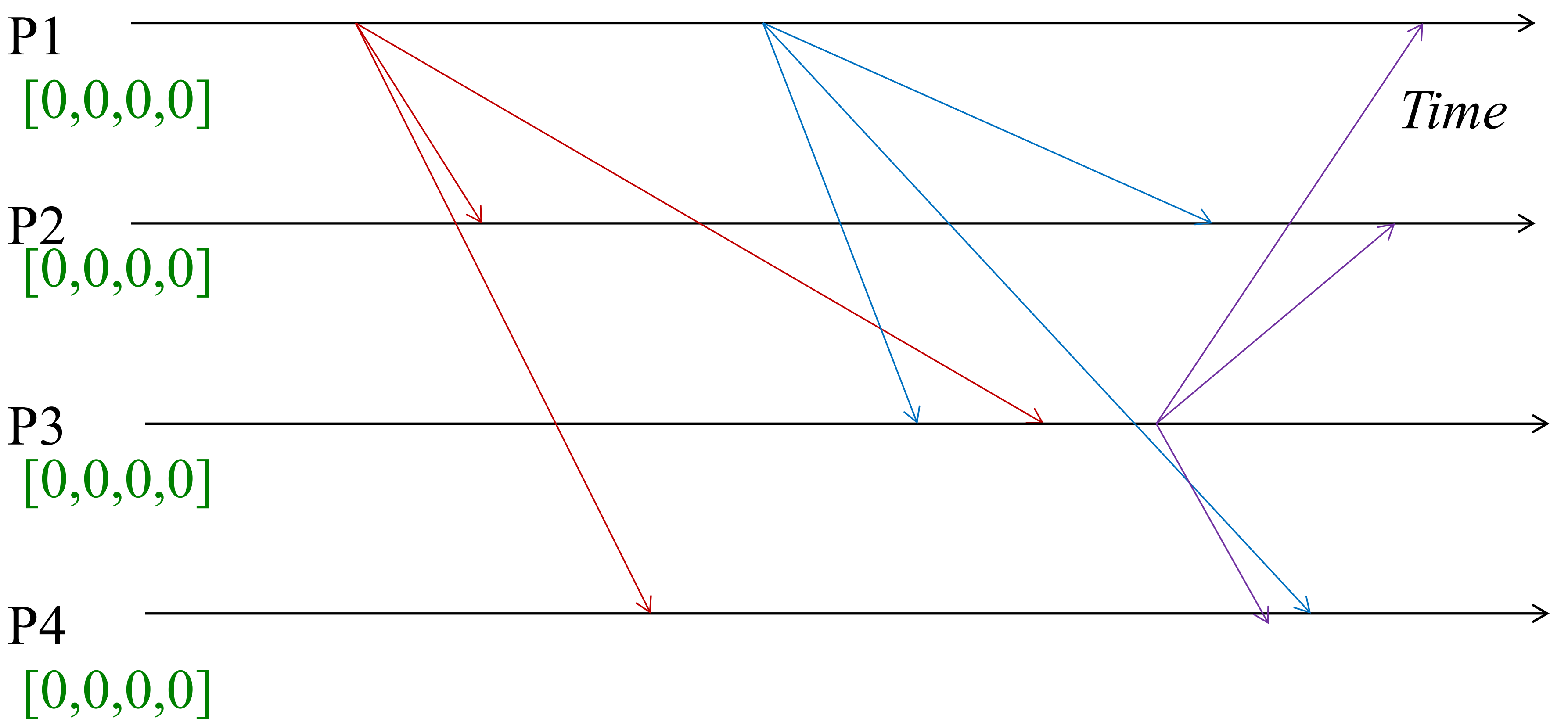
FIFO order multicast execution

FIFO order multicast execution



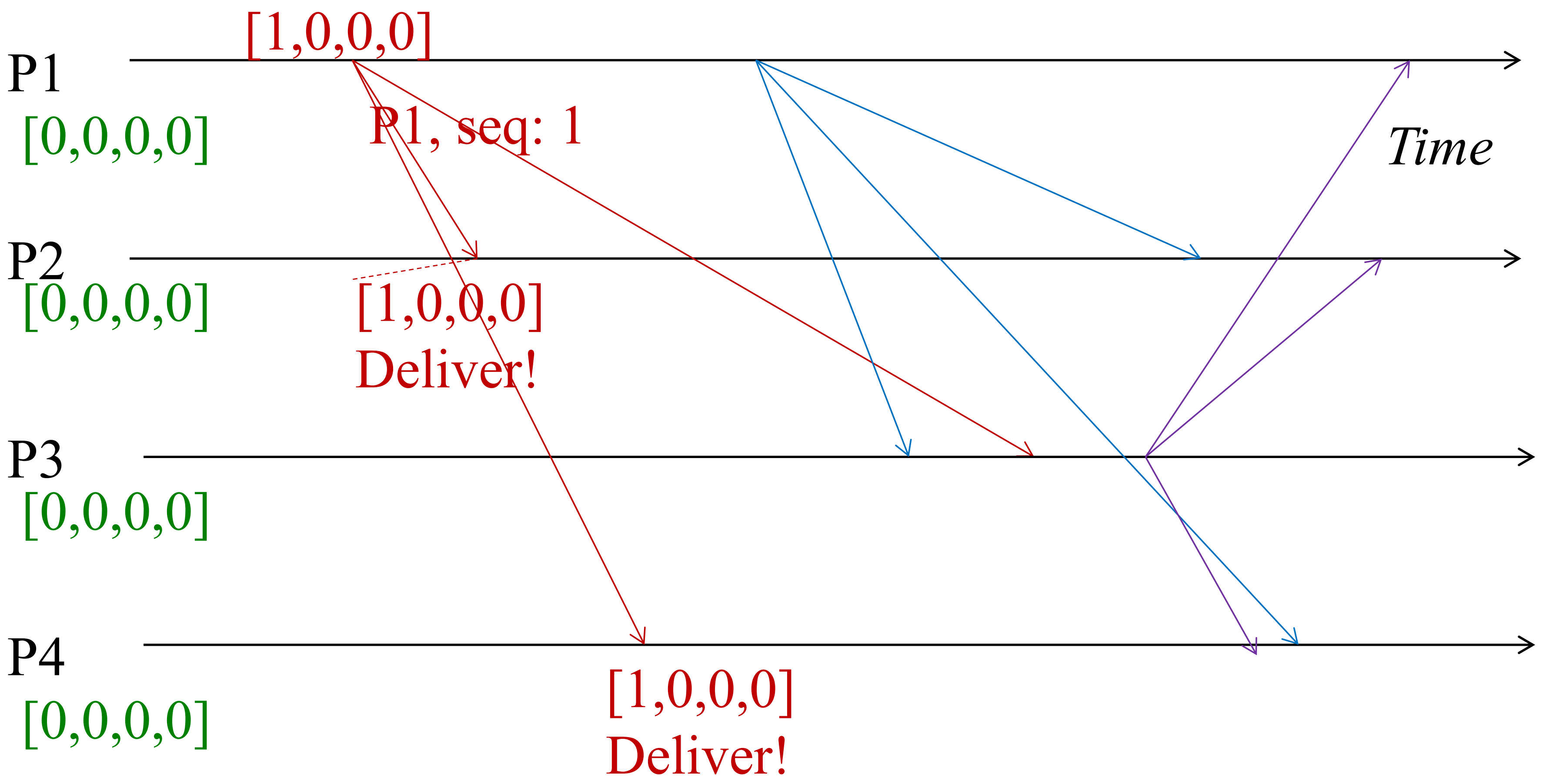
FIFO order multicast execution

FIFO order multicast execution

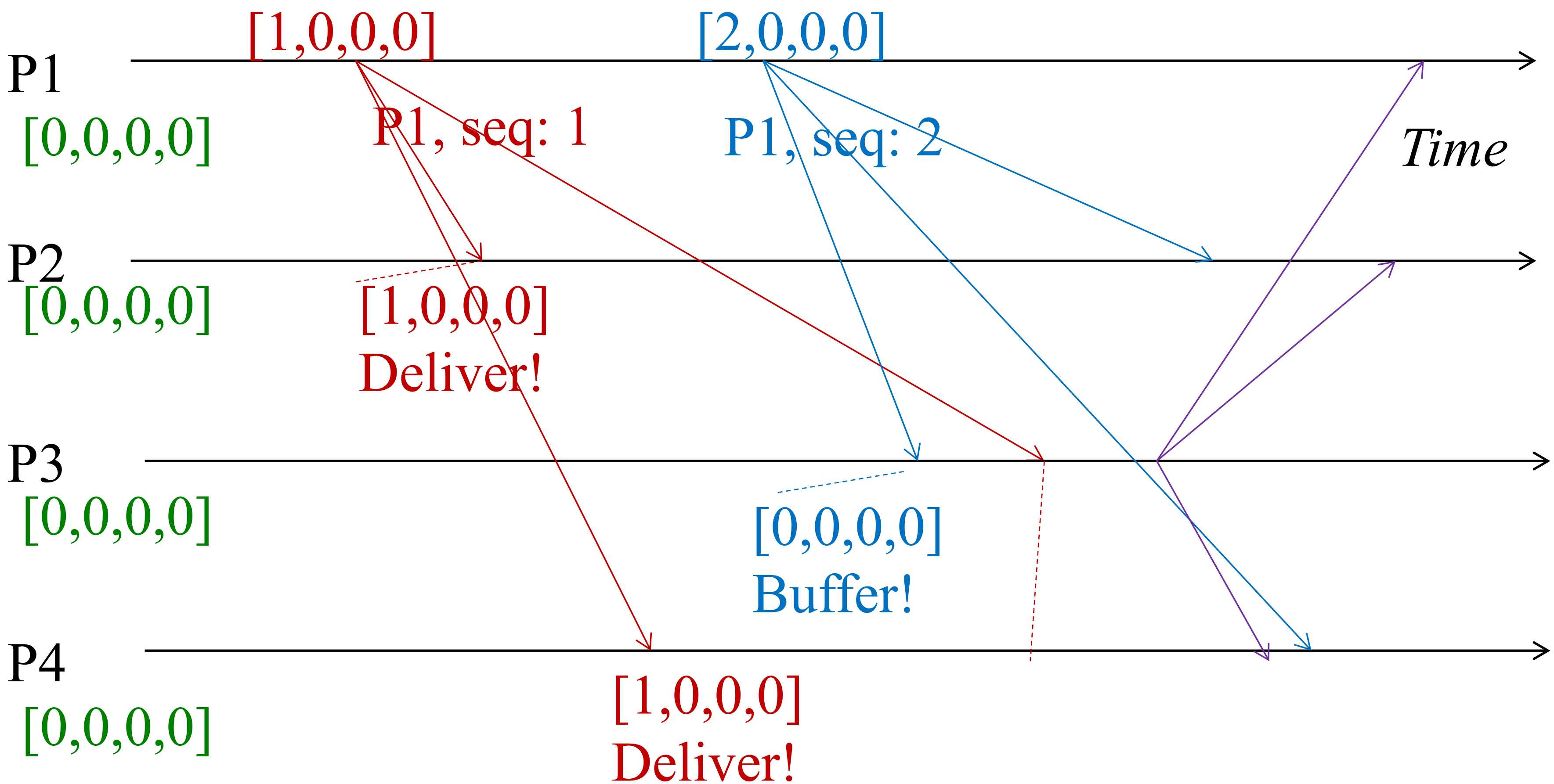


Self-deliveries omitted for simplicity.

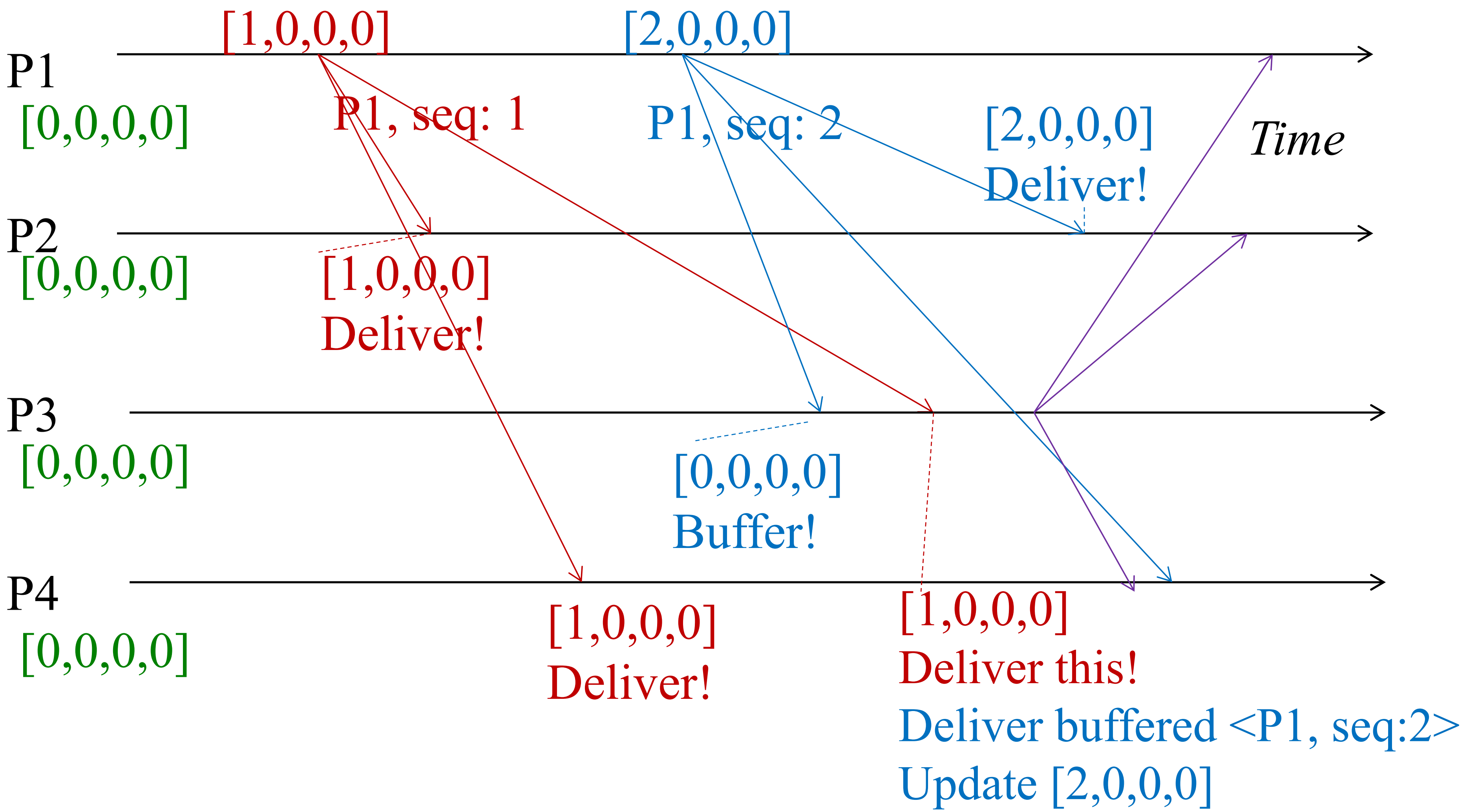
FIFO order multicast execution



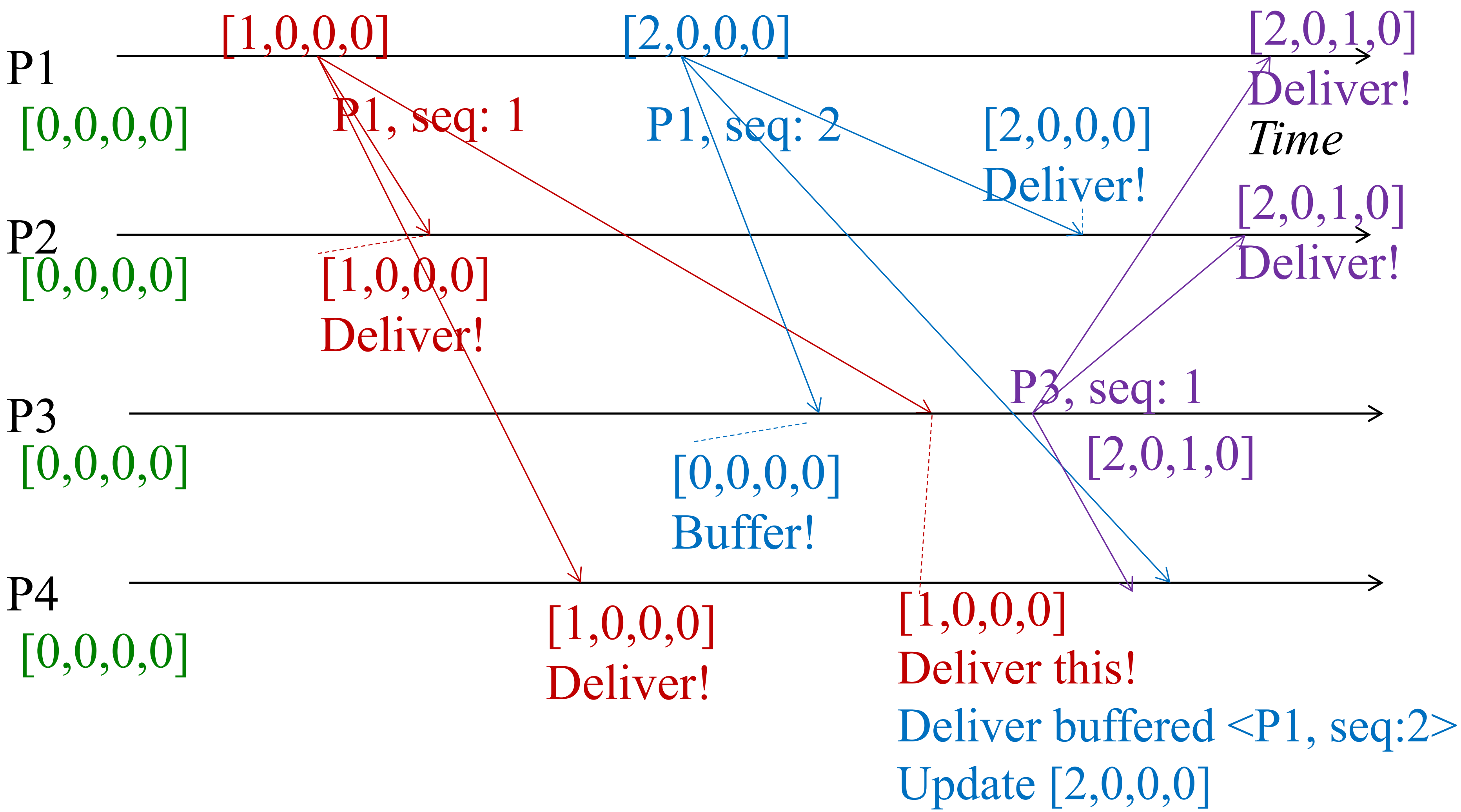
FIFO order multicast execution



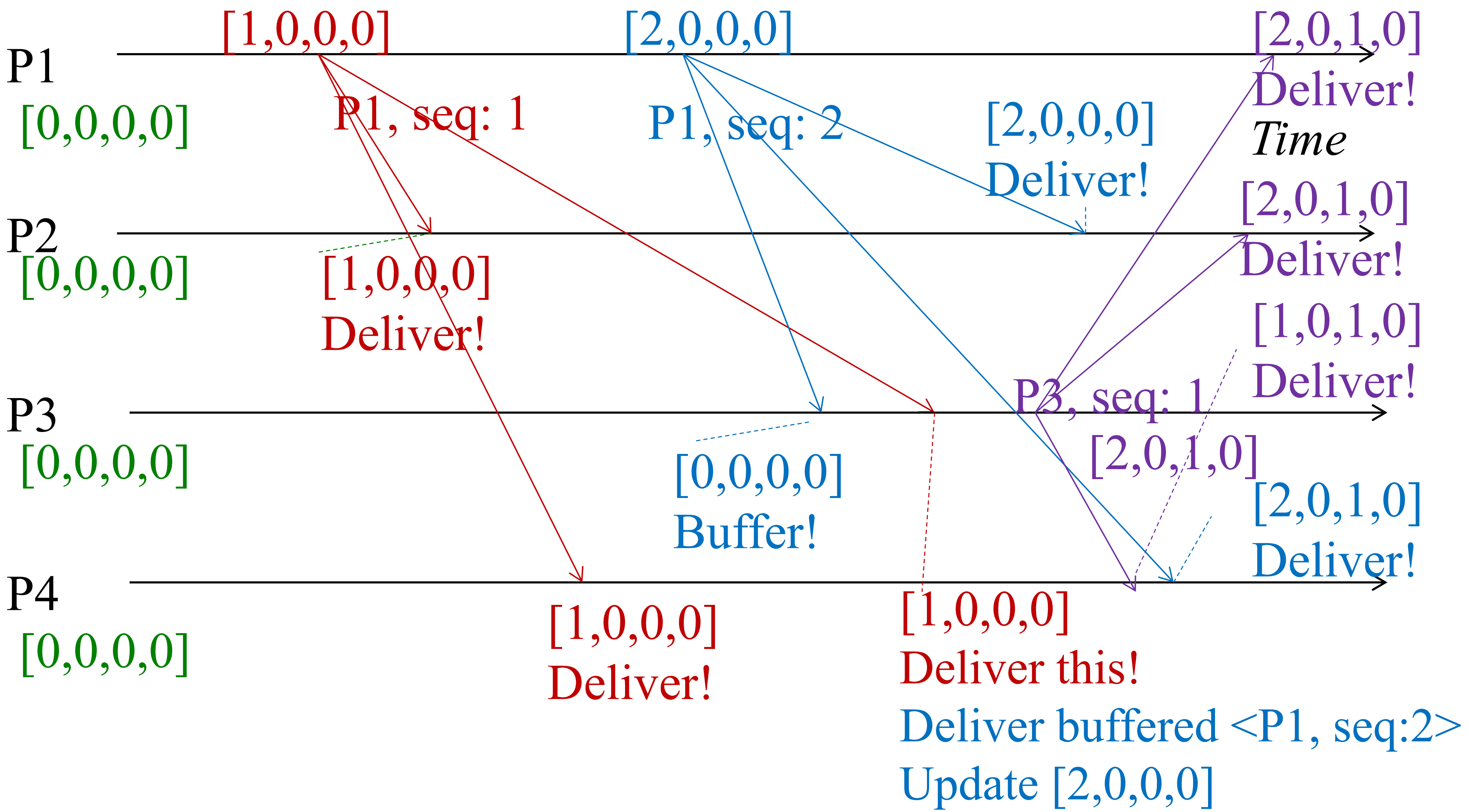
FIFO order multicast execution



FIFO order multicast execution



FIFO order multicast execution



Implementing FIFO order multicast

- On FO-multicast(g,m) at process P*j*:

  set P*j*[*j*] = P*j*[*j*] + 1

  piggyback P*j*[*j*] with m as its sequence number.

  B-multicast(g, {m, P*j*[*j*]})

- On B-deliver({m, S}) at Pi from Pj: *If Pi receives a multicast from Pj with sequence number S in message*

  if (S == P*i*[*j*] + 1) then

  FO-deliver(m) to application

  set P*i*[*j*] = P*i*[*j*] + 1

  else buffer this multicast until above condition is true

Implementing FIFO reliable multicast

- On FO-multicast(g,m) at process P*j*:
  - set P*j*[*j*] = P*j*[*j*] + 1
  - piggyback P*j*[*j*] with m as its sequence number.
  - **R-multicast(g,{m, P*j*[*j*]})**
- On **R-deliver({m, S})** at Pi from Pj: *If Pi receives a multicast from Pj with sequence number S in message*
  - if (S == P*i*[*j*] + 1) then
    - FO-deliver(m) to application
    - set P*i*[*j*] = P*i*[*j*] + 1
  - else buffer this multicast until above condition is true

Ordered Multicast

- FIFO ordering
  - If a correct process issues multicast(*g*,*m*) and then multicast(*g*,*m'*), then every correct process that delivers *m'* will have already delivered m.
- Causal ordering
  - If multicast(*g*,*m*) → multicast(*g*,*m'*) then any correct process that delivers *m'* will have already delivered *m*.
  - Note that → counts messages **delivered** to the application, rather than all network messages.
- Total ordering
  - If a correct process delivers message *m* before *m'* (independent of the senders), then any other correct process that delivers *m'* will have already delivered *m*.

Implementing causal order multicast

- Similar to FIFO Multicast
  - What you send with a message differs.
  - Updating rules differ.

- Each receiver maintains a vector of per-sender sequence numbers (integers)
  - Processes P*1* through P*N*.
  - P*i* maintains a vector of sequence numbers P*i*[1…N] (initially all zeroes).
  - P*i*[*j*] is the latest sequence number P*i* has received from P*j*.

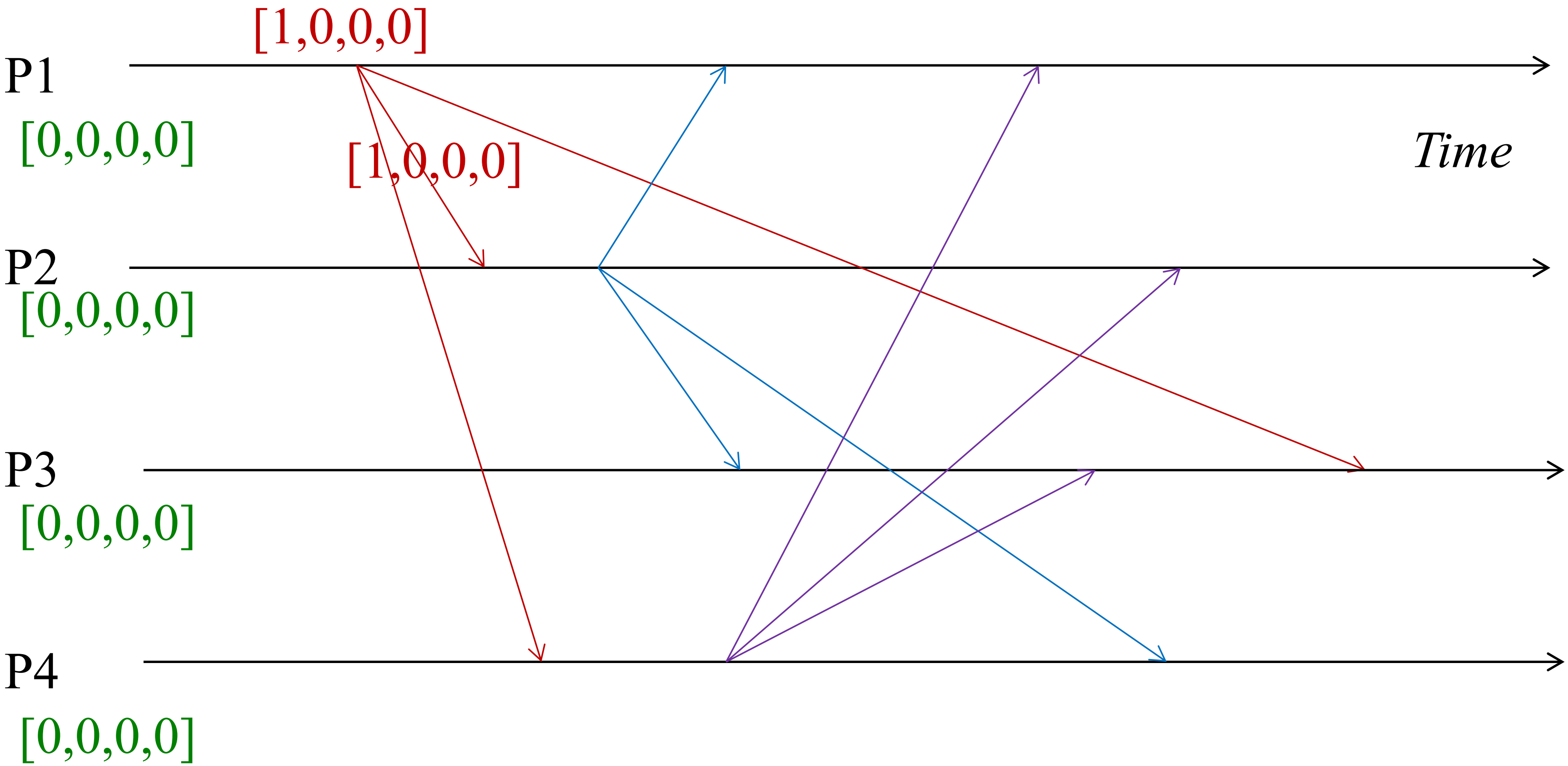
- Ignores other network messages. Only looks at multicast messages delivered to the application.

Implementing causal order multicast

- *CO-multicast(g,m) at Pj:*
  set $Pj[j] = Pj[j] + 1$
  piggyback entire vector $Pj[1\ldots N]$ with m as its sequence no.
  B-multicast(g,{m, $Pj[1\ldots N]$})
- *On B-deliver({m, V[1..N]}) at Pi from Pj:* If Pi receives a multicast from Pj with sequence vector $V[1\ldots N]$, buffer it until both:
  1. This message is the next one Pi is expecting from Pj, i.e.,
     $V[j] = Pi[j] + 1$
  2. All multicasts, anywhere in the group, which happened-before m have been received at Pi, i.e.,
     For all $k \neq j$: $V[k] \leq Pi[k]$

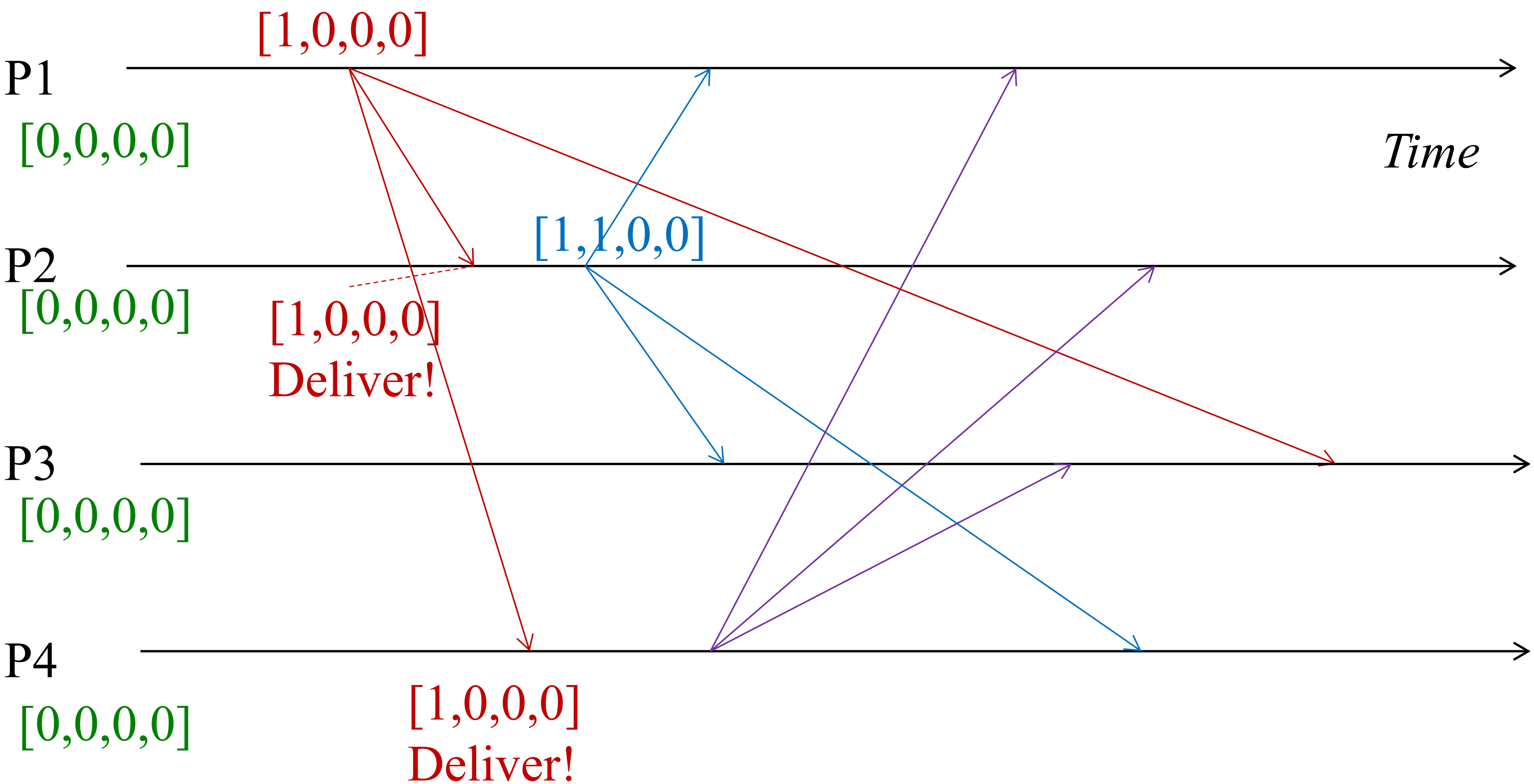
  When above two conditions satisfied,
  CO-deliver(m) and set $Pi[j] = V[j]$

Causal order multicast execution

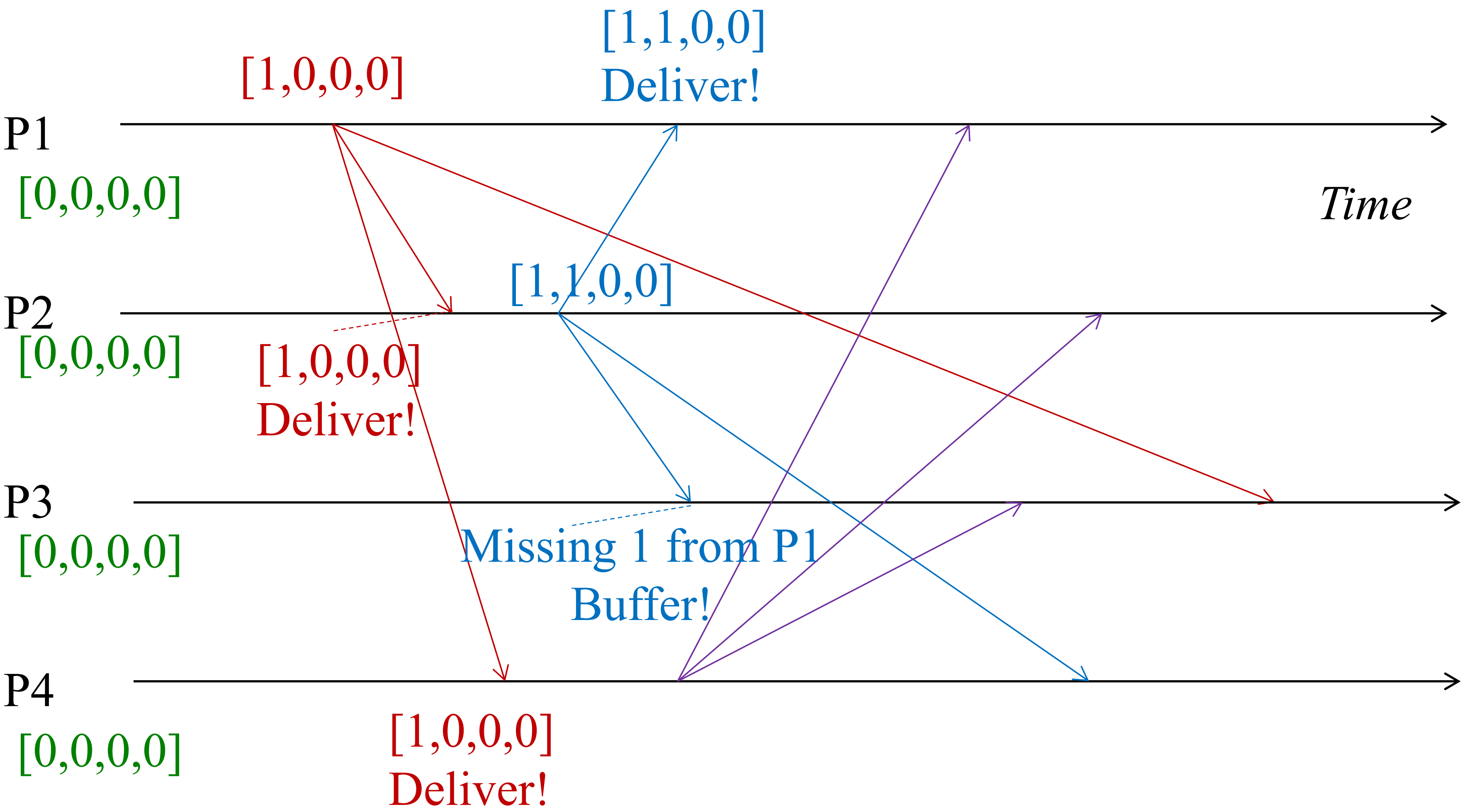


Self-deliveries omitted for simplicity.

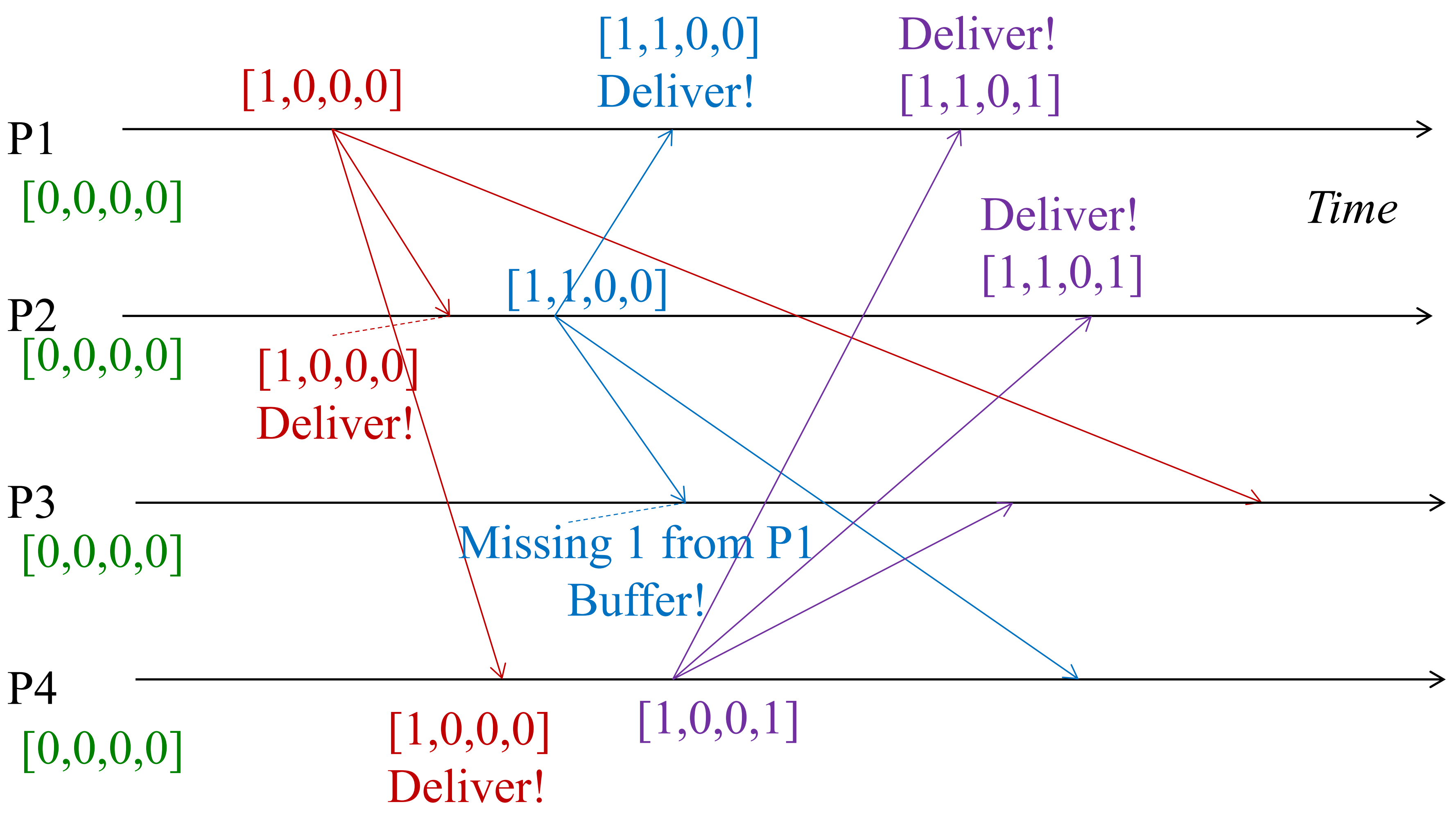
Causal order multicast execution



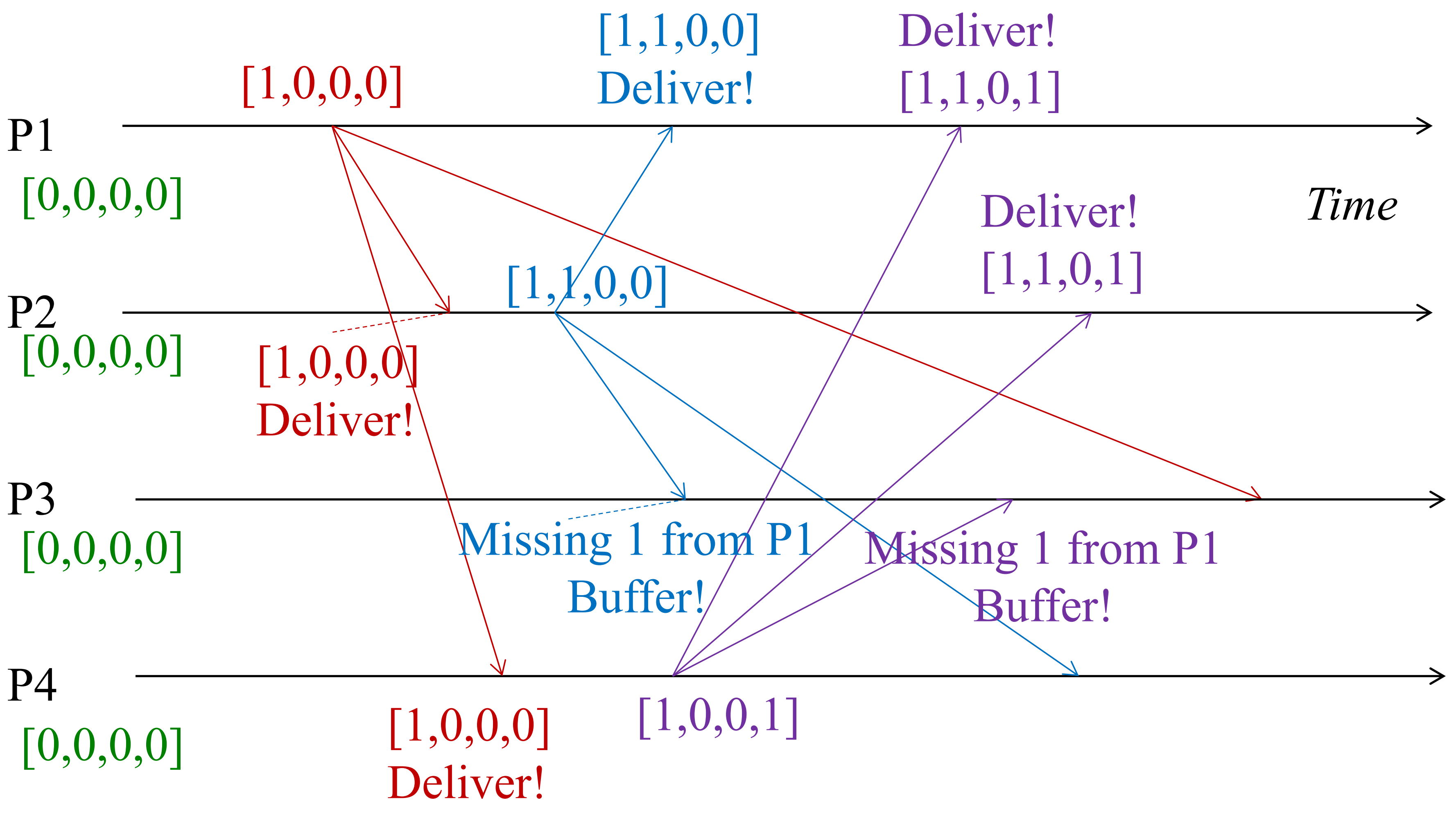
Causal order multicast execution



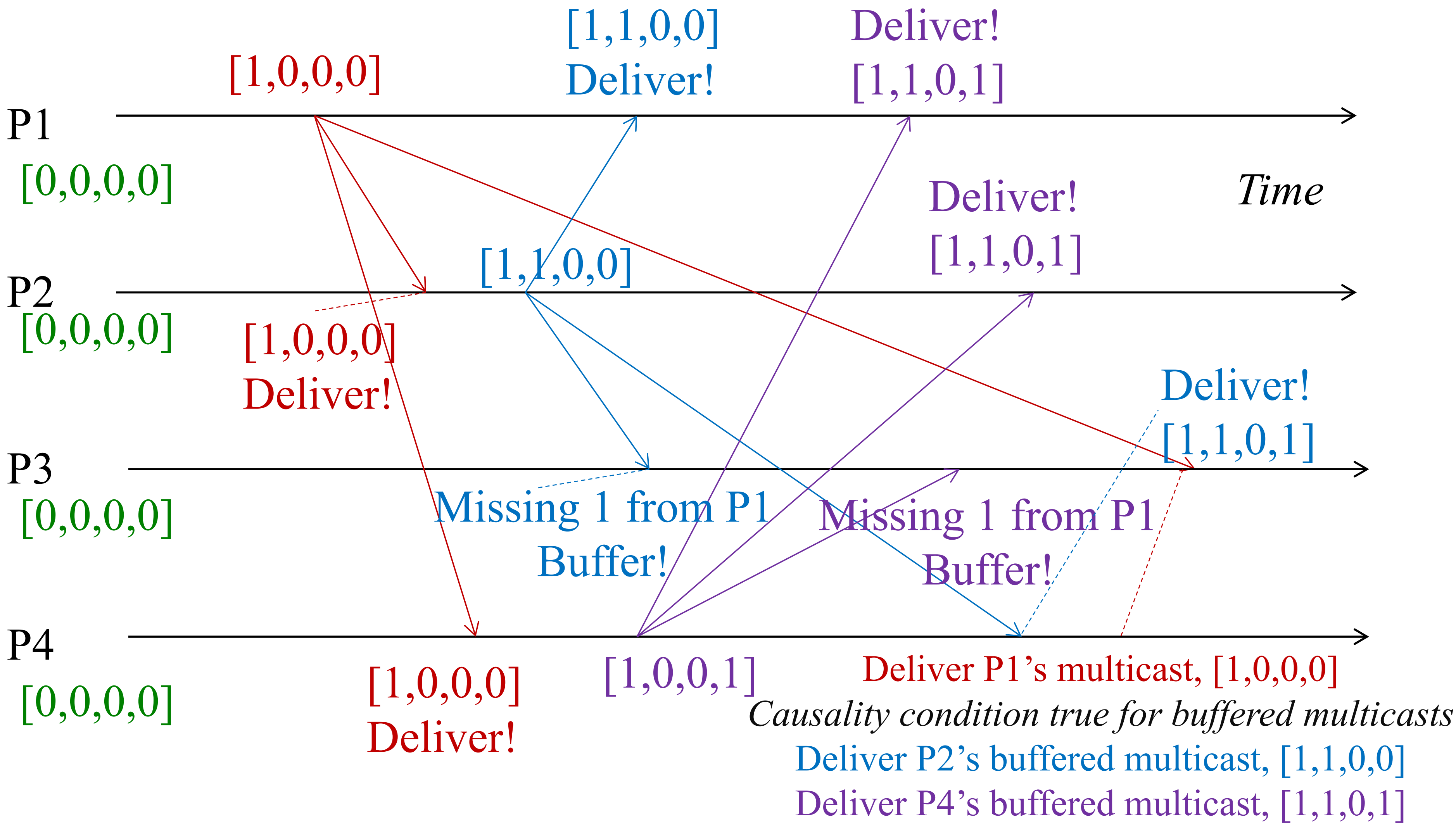
Causal order multicast execution



Causal order multicast execution



Causal order multicast execution



Ordered Multicast

- **FIFO ordering:** If a correct process issues multicast(*g*,*m*) and then multicast(*g*,*m'*), then every correct process that delivers *m'* will have already delivered m.
- **Causal ordering:** If multicast(*g*,*m*) → multicast(*g*,*m'*) then any correct process that delivers *m'* will have already delivered *m*.
  - Note that → counts messages **delivered** to the application, rather than all network messages.
- **Total ordering**: If a correct process delivers message *m* before *m'* (independent of the senders), then any other correct process that delivers *m'* will have already delivered *m*.

Implementing total order multicast

- Basic idea:
    - Same sequence number counter across different processes.
    - Instead of different sequence number counter for each process.
- Two types of approach
    - Using a centralized sequencer
    - A decentralized mechanism (ISIS)

Implementing total order multicast

- Basic idea:
  - Same sequence number counter across different processes.
  - Instead of different sequence number counter for each process.

- Two types of approach
  - **Using a centralized sequencer**
  - A decentralized mechanism (ISIS)

Sequencer based total ordering

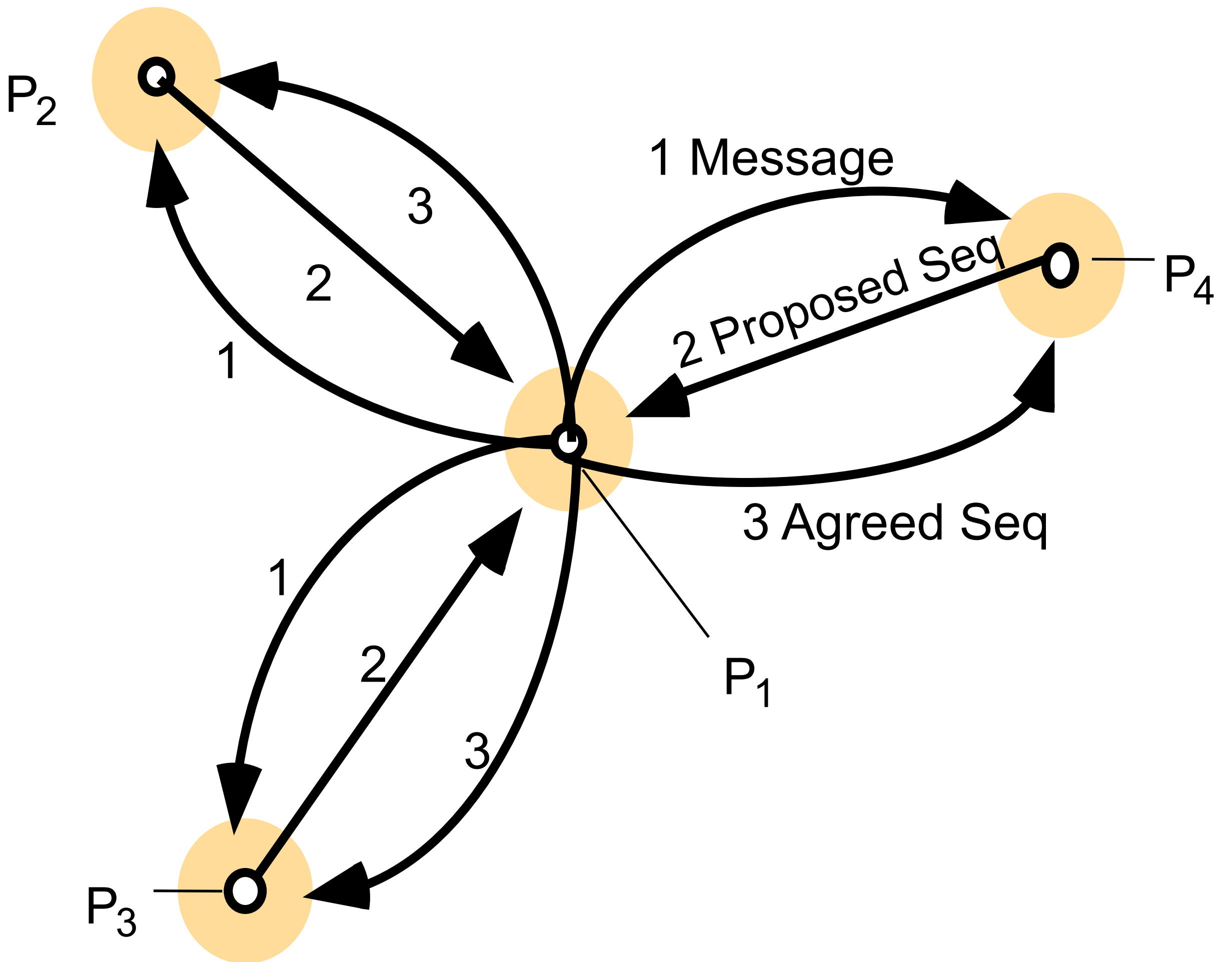
- Special process elected as leader or sequencer.
- TO-multicast(g,m) at P*i*:
  - Send multicast message m to group g *and the sequencer*
- Sequencer:
  - Maintains a global sequence number S (initially 0)
  - When a multicast message m is B-delivered to it:
    - sets S = S + 1, and B-multicast(g,{"order", m, S})
- Receive multicast at process P*i*:
  - P*i* maintains a local received global sequence number S*i* (initially 0)
  - On B-deliver(m) at P*i* from P*j*, it buffers it until both conditions satisfied
    1. B-deliver({"order", m, S}) at P*i* from sequencer, and
    2. S*i* + 1 = S
    - Then TO-deliver(m) to application and set S*i* = S*i* + 1

Implementing total order multicast

- Basic idea:
  - Same sequence number counter across different processes.
  - Instead of different sequence number counter for each process.

- Two types of approach
  - Using a centralized sequencer
  - **A decentralized mechanism (ISIS)**

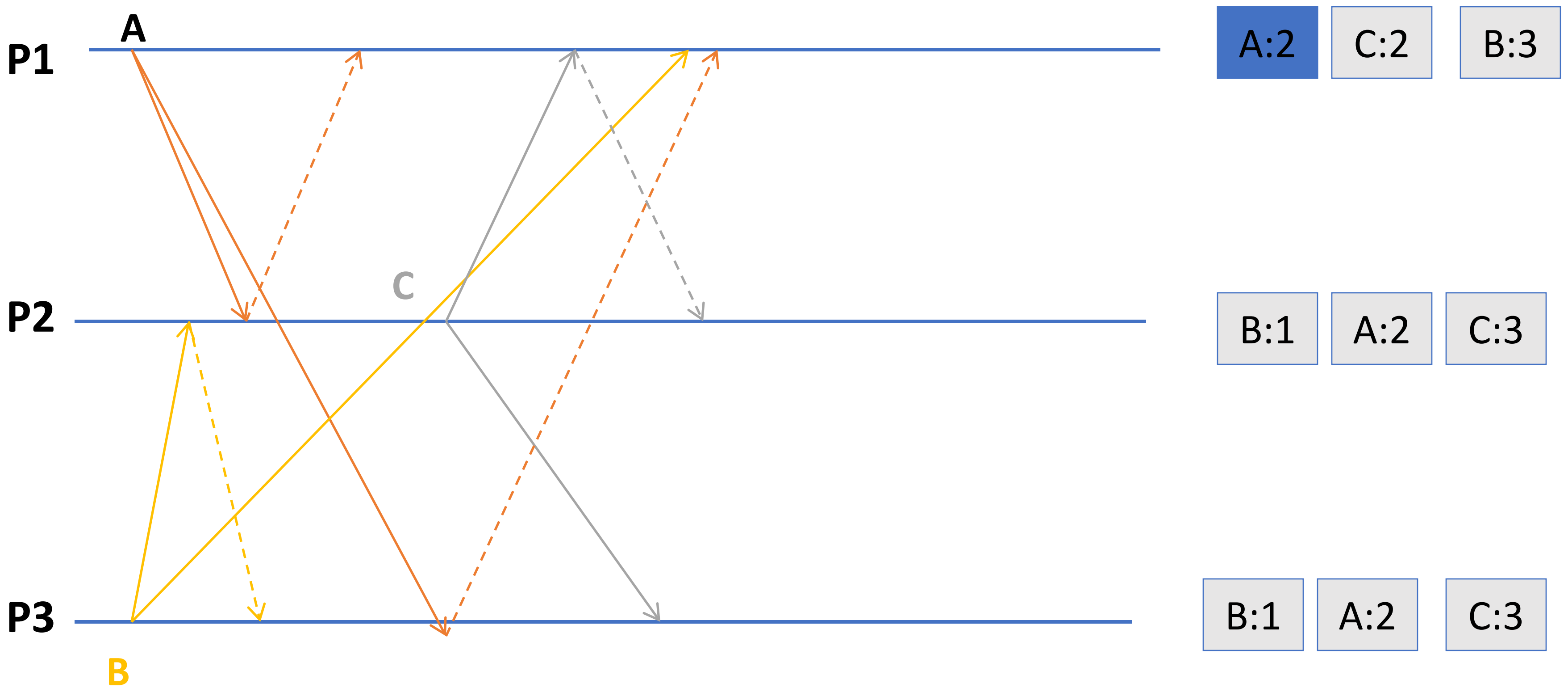
ISIS algorithm for total ordering



ISIS algorithm for total ordering

- Sender multicasts message to everyone.
- Receiving processes:
  - reply with *proposed* priority (sequence no.)
    - larger than all observed *agreed* priorities
    - larger than any previously proposed (by self) priority
  - store message in *priority queue*
    - ordered by priority (proposed or agreed)
  - mark message as undeliverable
- Sender chooses *agreed* priority, re-multicasts message with agreed priority
  - maximum of all proposed priorities
- Upon receiving agreed (final) priority
  - reorder messages based on final priority.
  - mark the message as deliverable.
  - deliver any deliverable messages at front of priority queue.

Example: ISIS algorithm



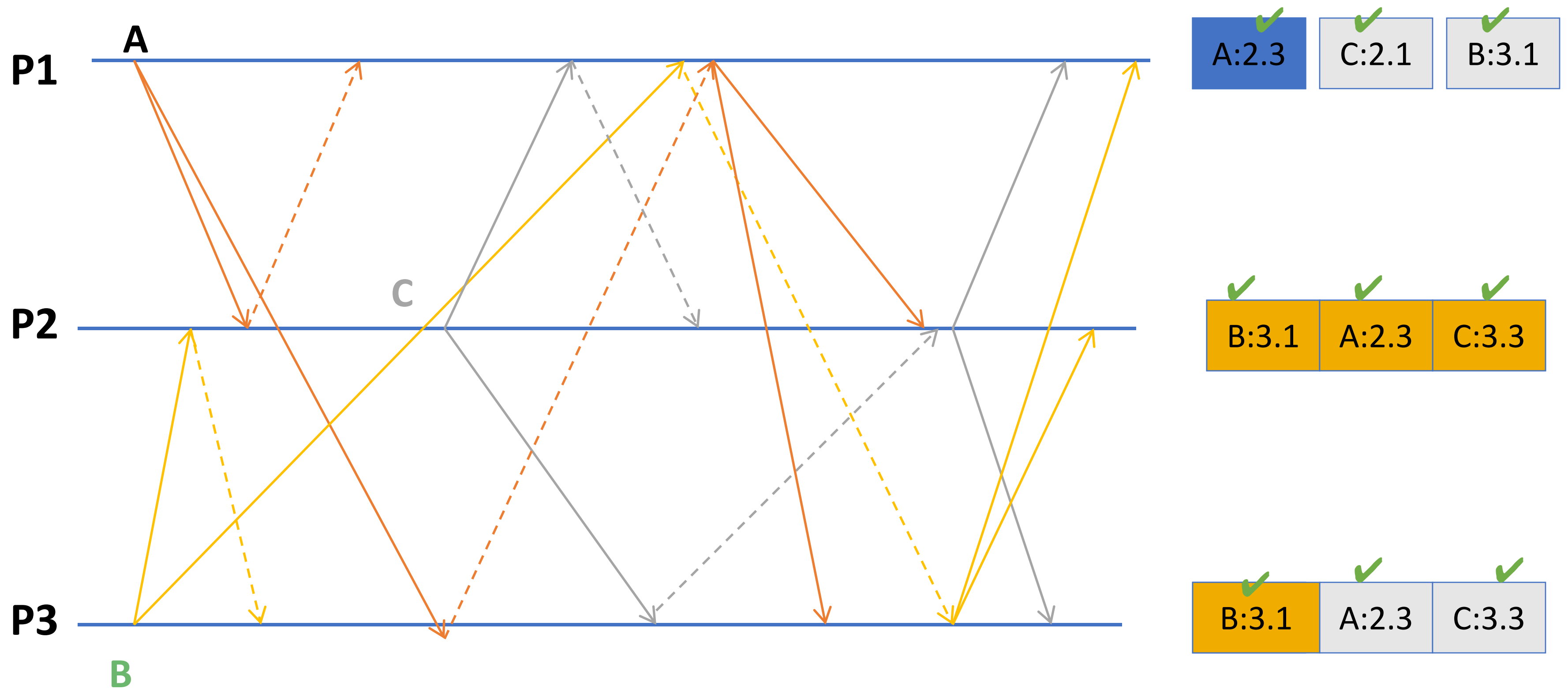
How do we break ties?

- Problem: priority queue requires unique priorities.

- Solution: add process # to suggested priority.
  - priority.(*id of the process that proposed the priority*)
  - i.e., 3.2 == process 2 proposed priority 3

- Compare on priority first, use process # to break ties.
  - 2.1 > 1.3
  - 3.2 > 3.1

Example: ISIS algorithm



Proof of total order with ISIS

- Consider two messages, m_1 and m_2, and two processes, p and p'.
- Suppose that p delivers m_1 before m_2.
- When *p* delivers m_1, it is at the head of the queue. m_2 is either:
  - Already in *p*'s queue, and deliverable, so
    - finalpriority(m_1) < finalpriority(m_2)
  - Already in *p*'s queue, and not deliverable, so
    - finalpriority(m_1) < proposedpriority(m_2) <= finalpriority(m_2)
  - Not yet in *p*'s queue:
    - same as above, since proposed priority > priority of any delivered message
- Suppose *p*' delivers m_2 before m_1, by the same argument:
  - finalpriority(m_2) < finalpriority(m_1)
  - Contradiction!

Ordered Multicast

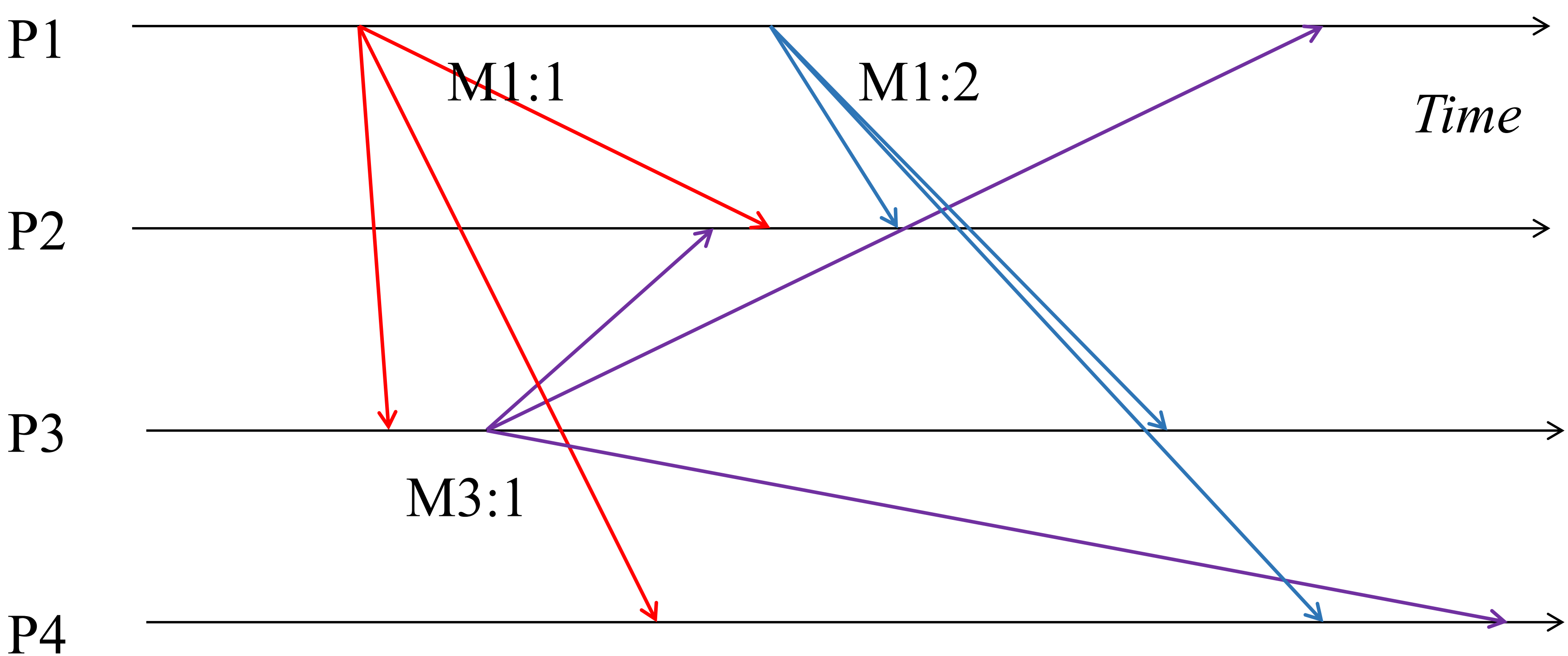
- FIFO ordering
  - If a correct process issues multicast(*g*,*m*) and then multicast(*g*,*m'*), then every correct process that delivers *m'* will have already delivered m.
- Causal ordering
  - If multicast(*g*,*m*) → multicast(*g*,*m'*) then any correct process that delivers *m'* will have already delivered *m*.
  - Note that → counts messages **delivered** to the application, rather than all network messages.
- Total ordering
  - If a correct process delivers message *m* before *m'* (independent of the senders), then any other correct process that delivers *m'* will have already delivered *m*.

Summary

- Multicast is an important communication mode in distributed systems.

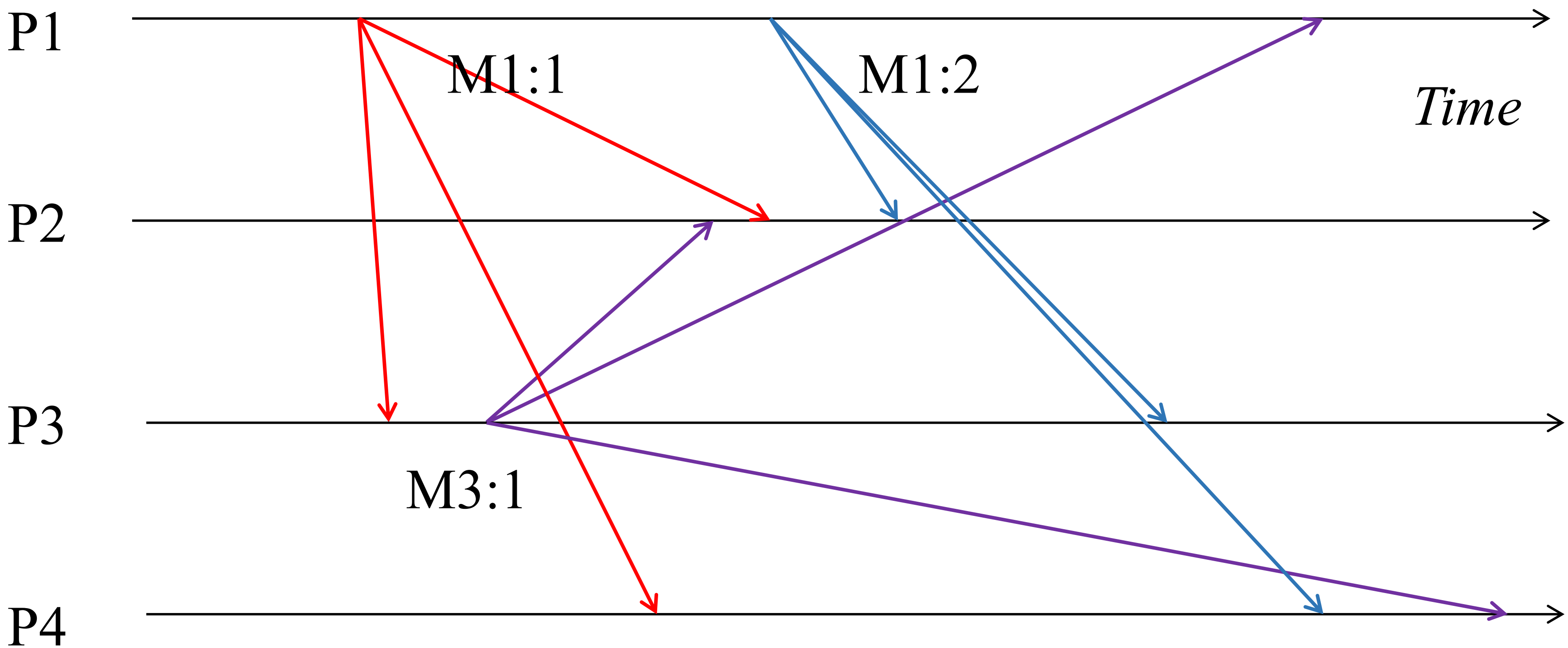
- Applications may have different requirements:
    - Reliability
    - Ordering: FIFO, Causal, Total
    - Combinations of the above.

Example



Does this satisfy FIFO order?

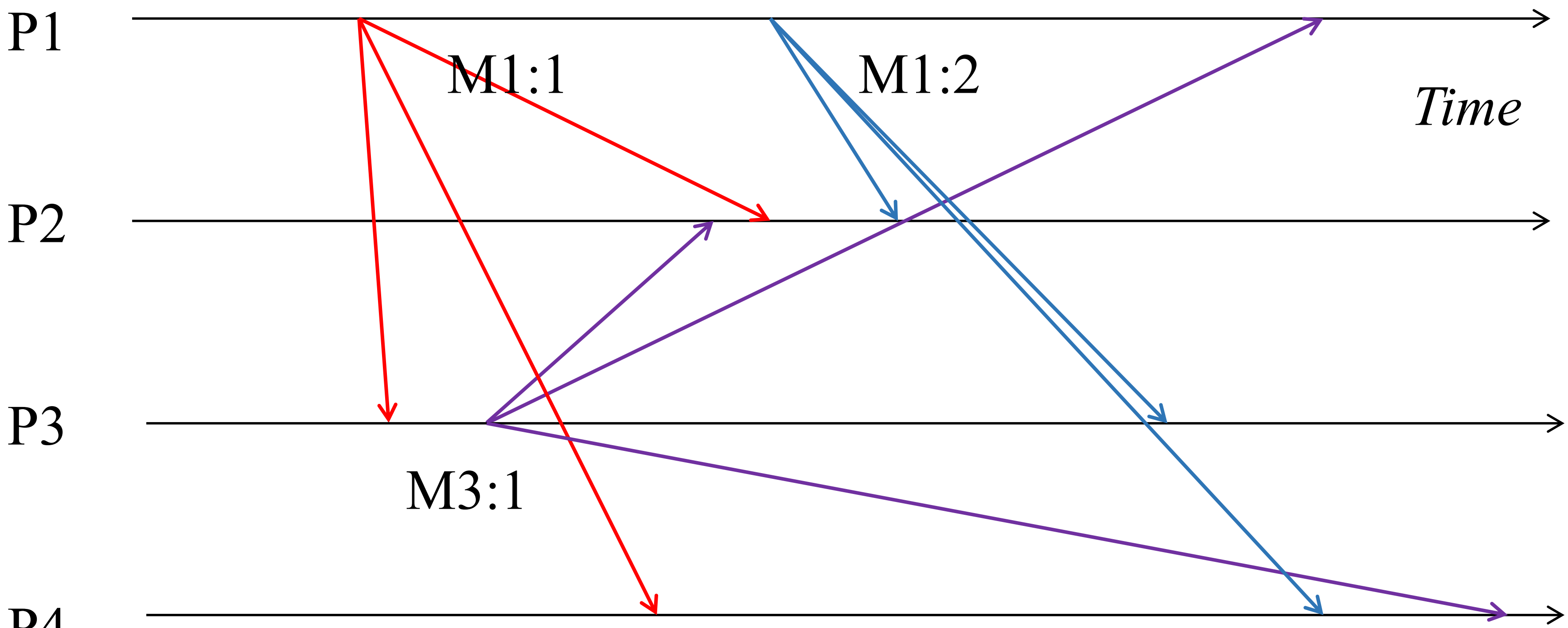
Example



Does this satisfy FIFO order?

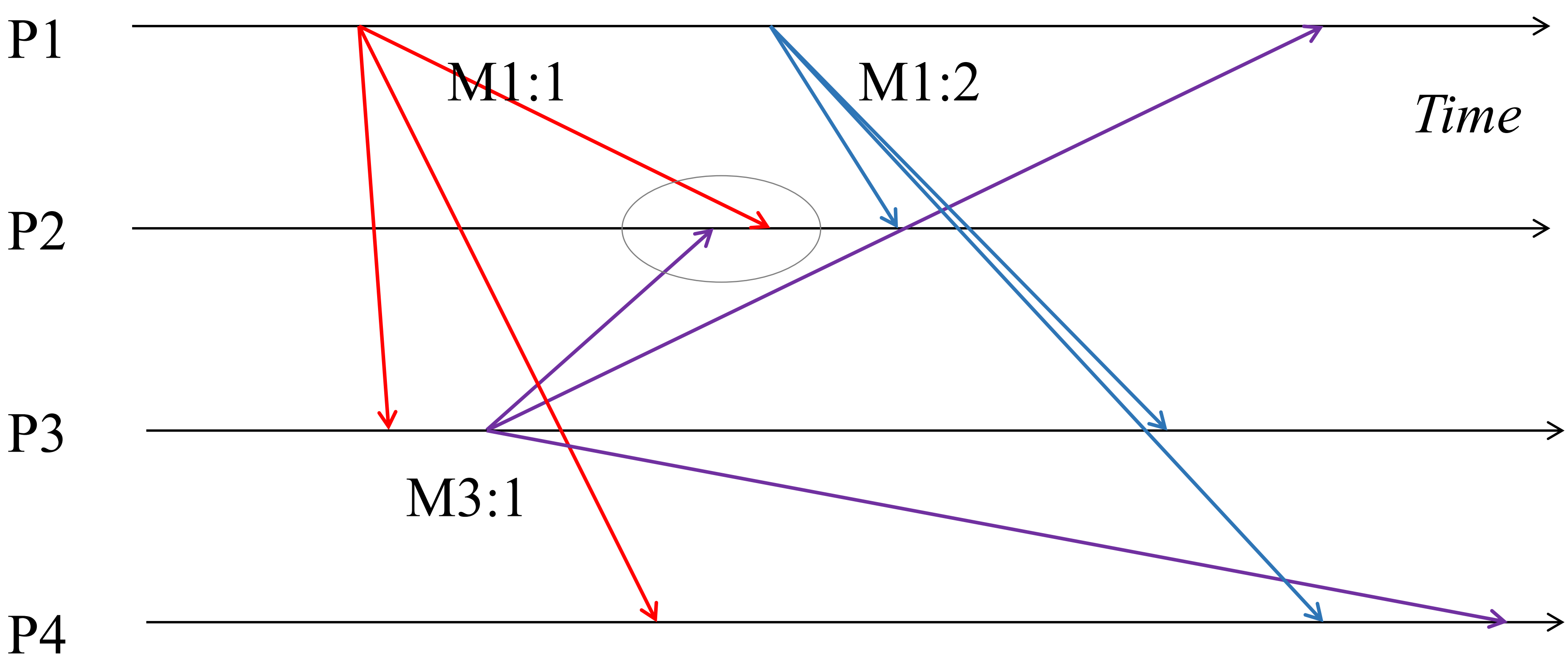
Yes

Example



Does this satisfy causal order?

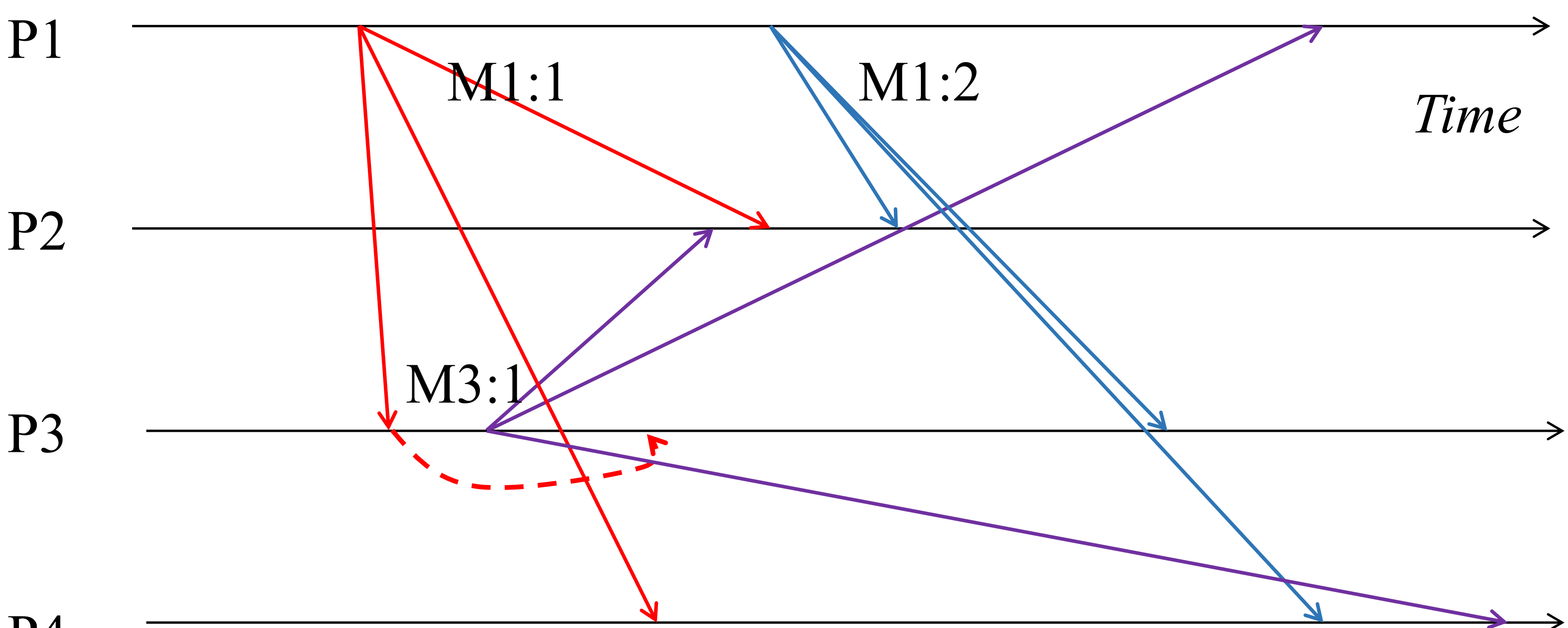
Example



Does this satisfy causal order?

No

Example

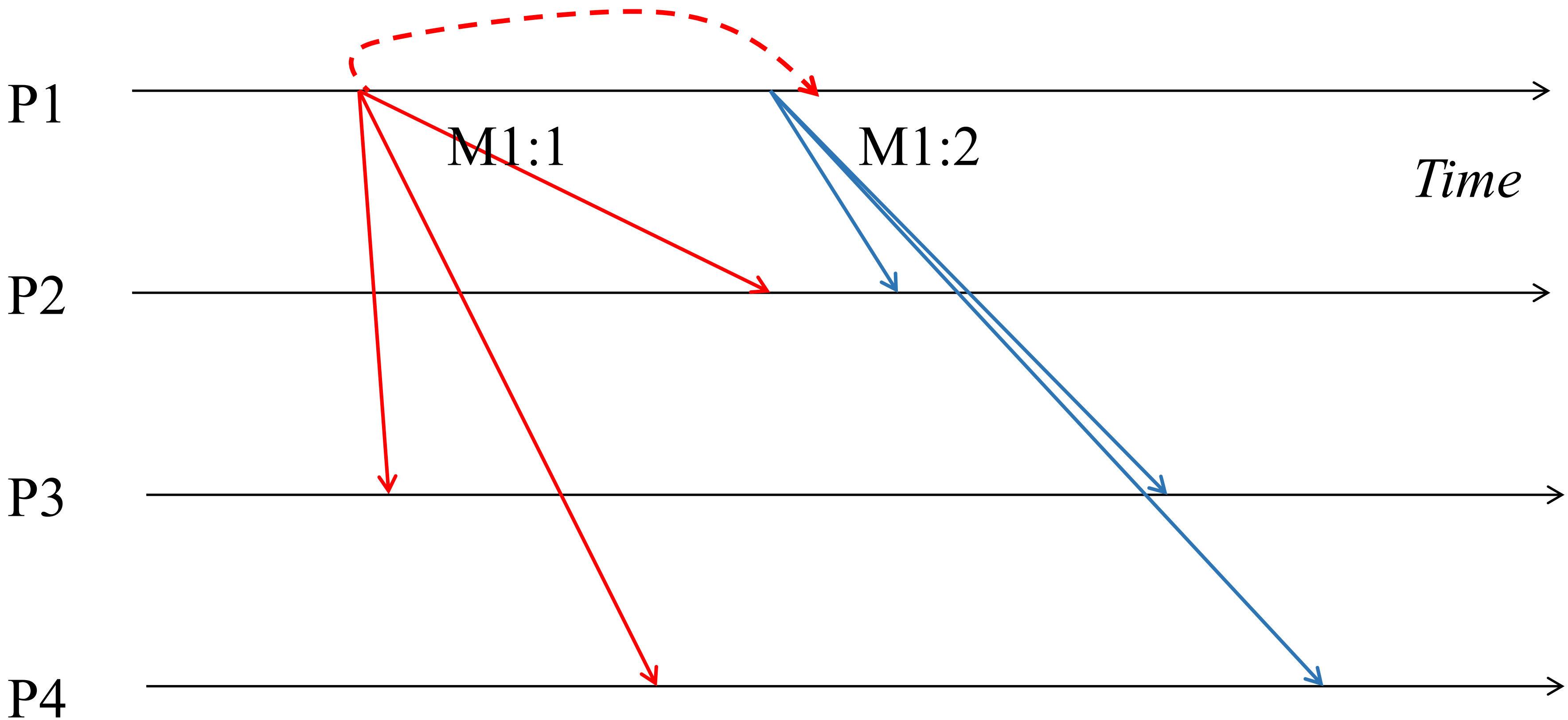


M1:1 is delivered at P3 after M3:1's multicast.

Does this satisfy causal order?

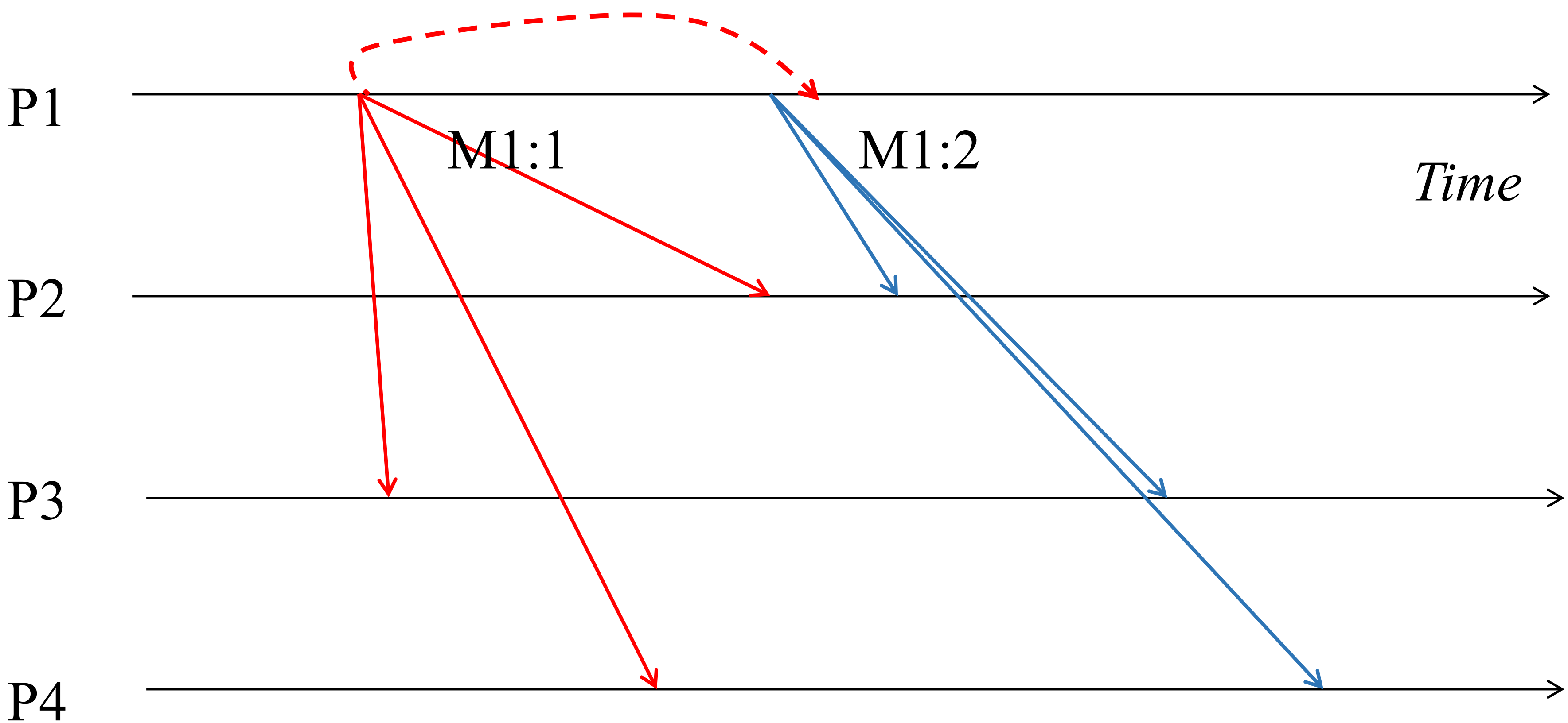
Yes

Example



Does this satisfy causal order?
No

Example



Does this satisfy FIFO order?
No

3. Total Order

Next class!

Also in next class,

How do we implement ordered multicast?