Distributed Systems

CS425/ECE428

Feb 24 2021

Instructor: Radhika Mittal

Acknowledgements for some of materials: Indy Gupta and Nikita Borisov

Logistics

- HW1 solutions have been released.
- MP1 has been released. Due on March 17th.

Today's agenda

- **Multicast**
  - Chapter 15.4

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

Recap: Multicast

- Useful communication mode in distributed systems:
  - Writing an object across replica servers.
  - Group messaging.
  - …..
- Basic multicast (B-multicast): unicast send to each process in the group.
  - Does not guarantee consistent message delivery if sender fails.
- Reliable multicast (R-mulicast):
  - Defined by three properties: *integrity, validity, agreement*.
  - If some correct process multicasts a message **m**, then all other correct processes deliver the **m** (exactly once).
  - When a process receives a message ‘m’ for the first time, it re-multicasts it again to other processes in the group.

Recap: 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 multicast messages **delivered** to the application, rather than all network messages.
- Total ordering
  - Yet to discuss.

Recap: 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 multicast messages **delivered** to the application, rather than all network messages.
- Total ordering
  - Yet to discuss.

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.

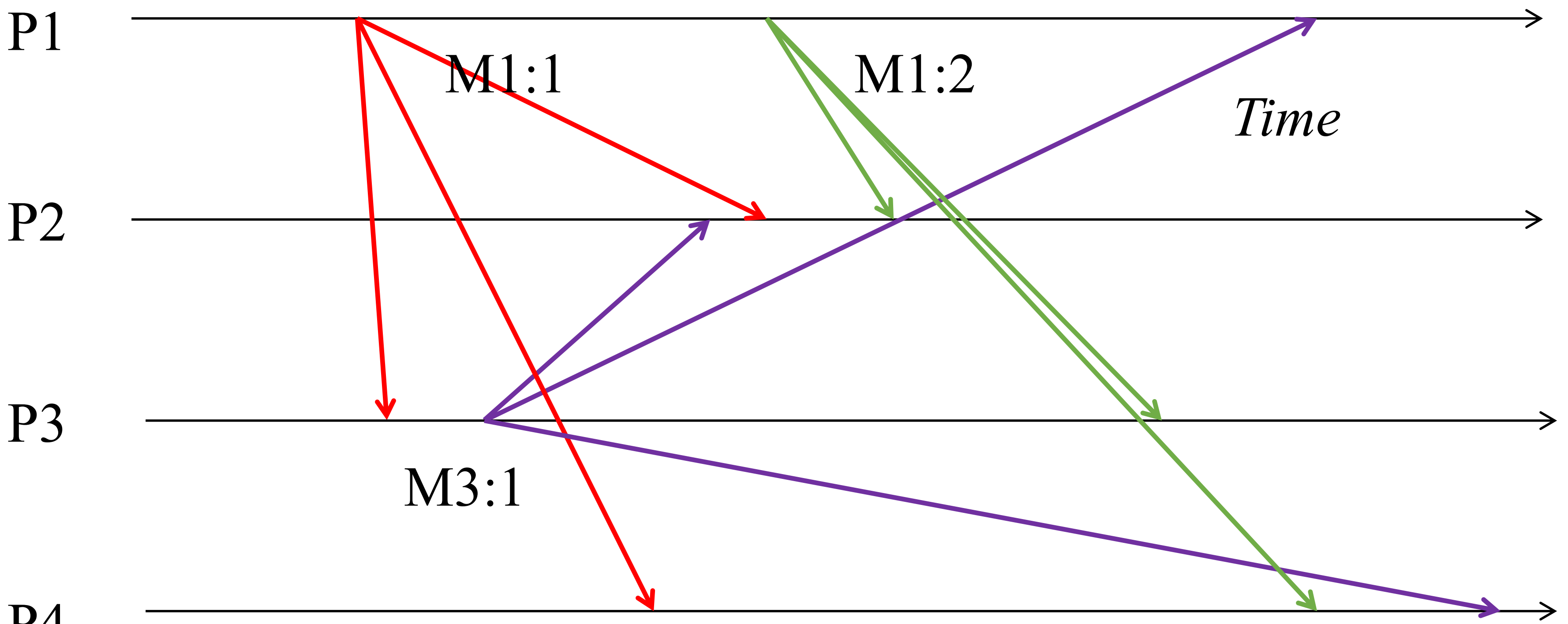
Causal vs FIFO

- Causal Ordering => FIFO Ordering

- Why?
  - If two multicasts M and M' are sent by the same process P, and M was sent before M', then M → M'.
  - Then a multicast protocol that implements causal ordering will obey FIFO ordering since M → M'.

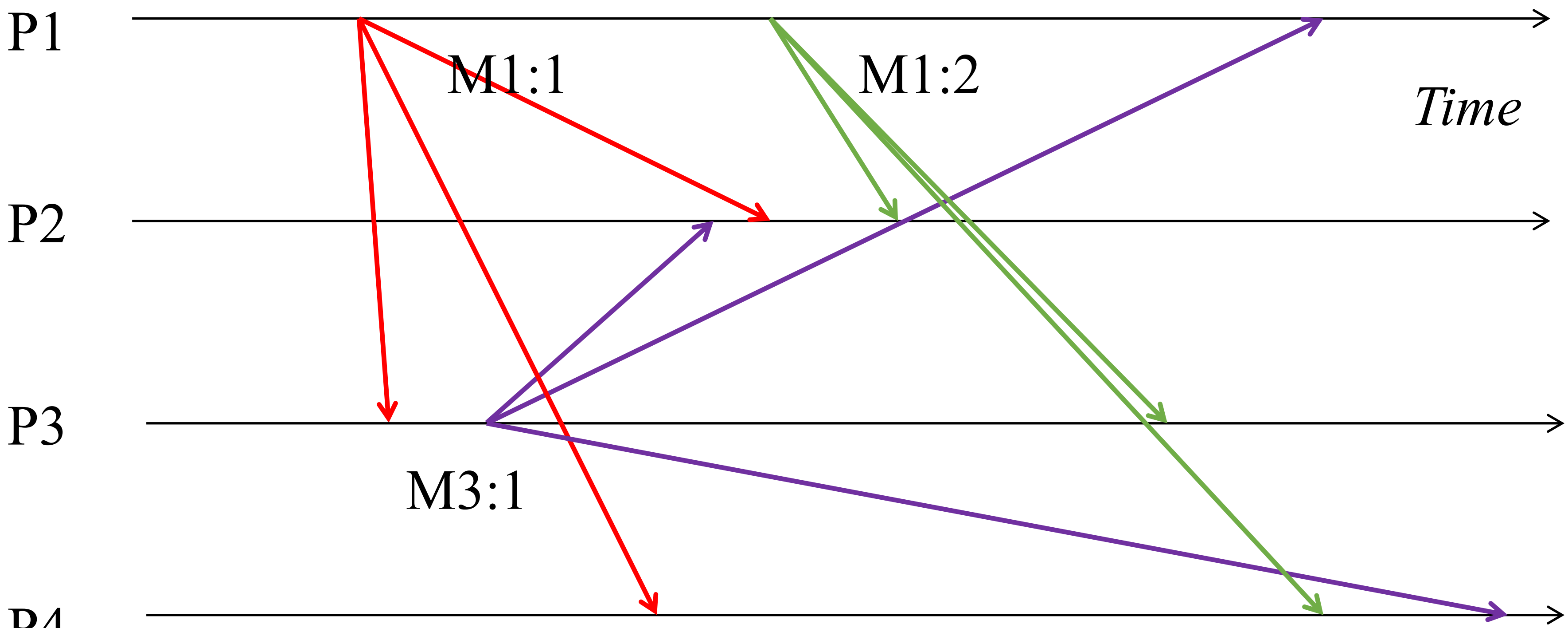
- Reverse is not true! FIFO ordering does not imply causal ordering.

Example



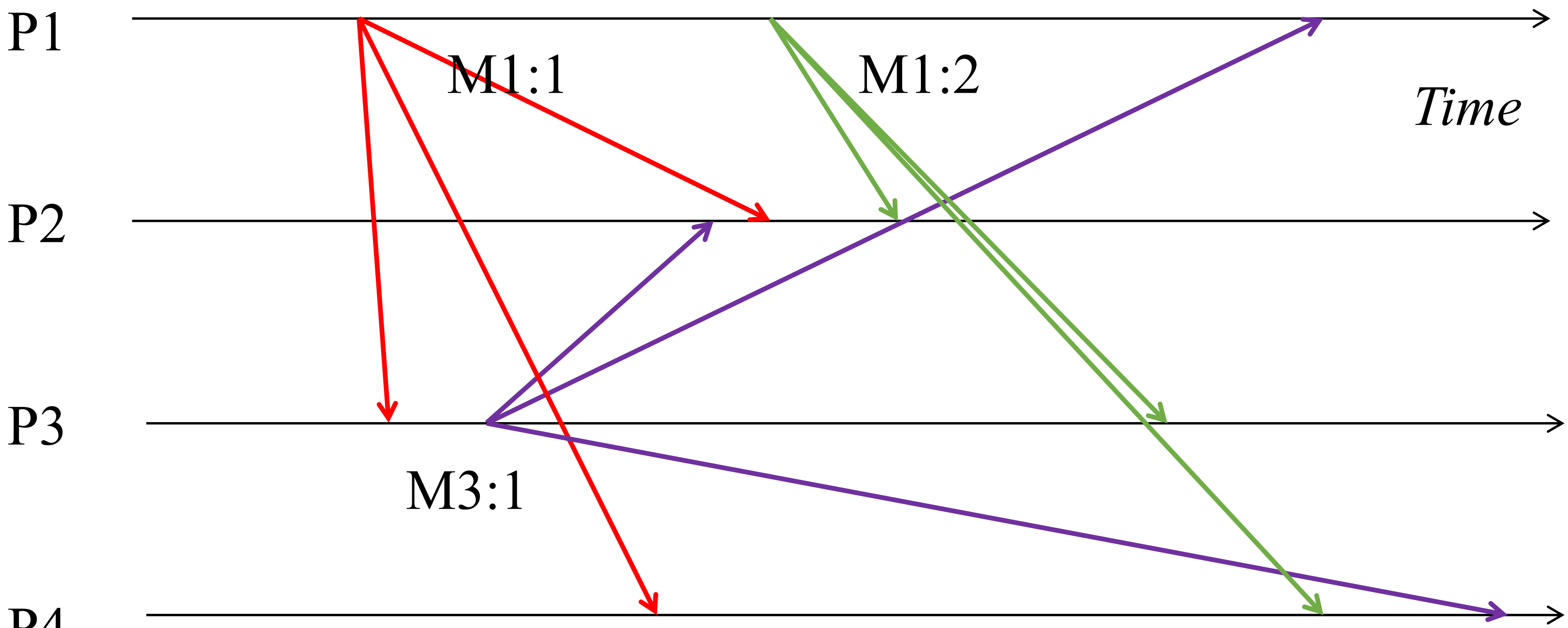
Does this satisfy FIFO order?

Example



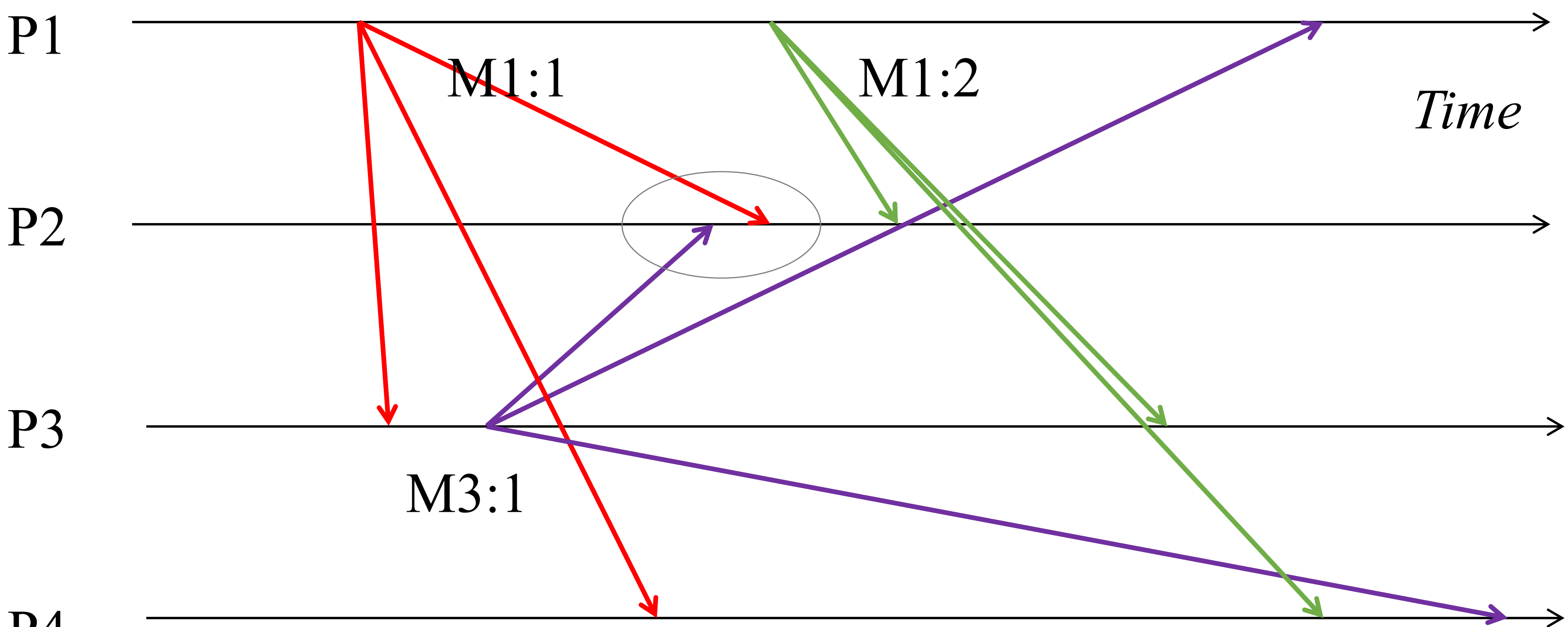
Does this satisfy FIFO order?
Yes

Example



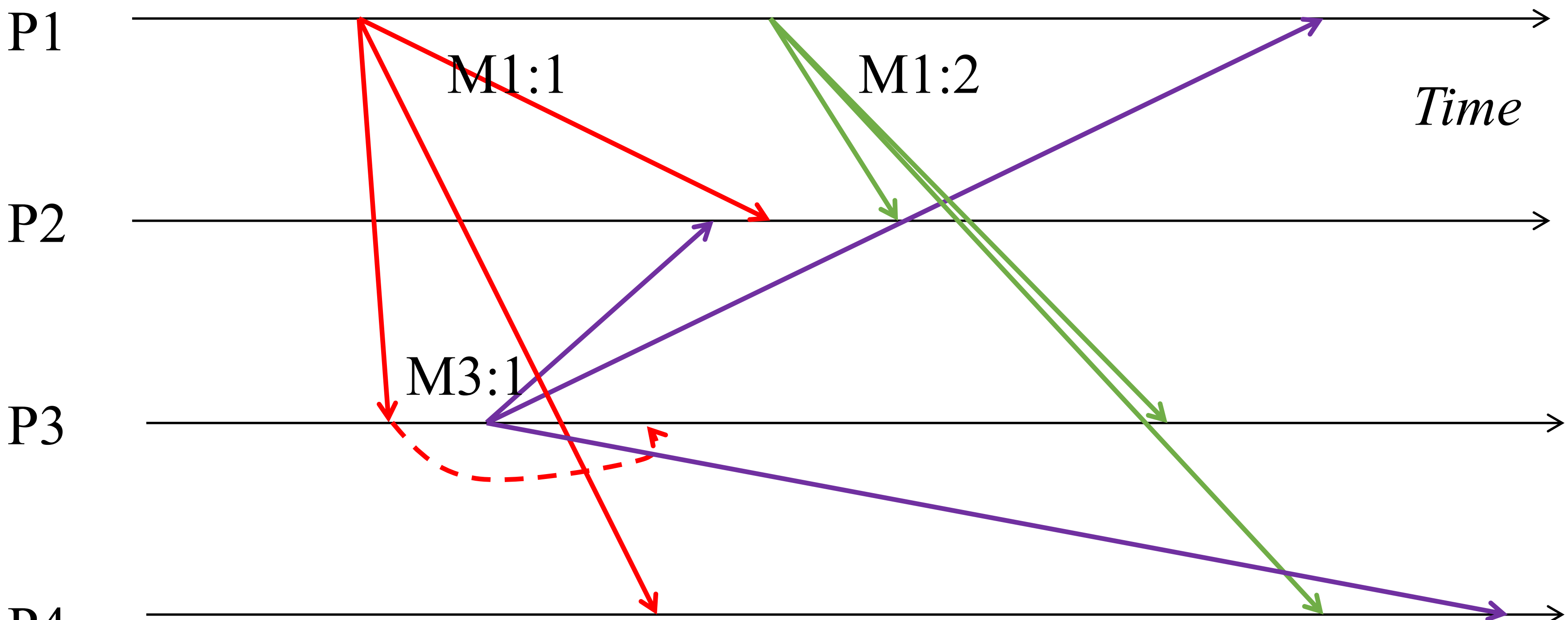
Does this satisfy causal order?

Example



Does this satisfy causal order?
No

Example

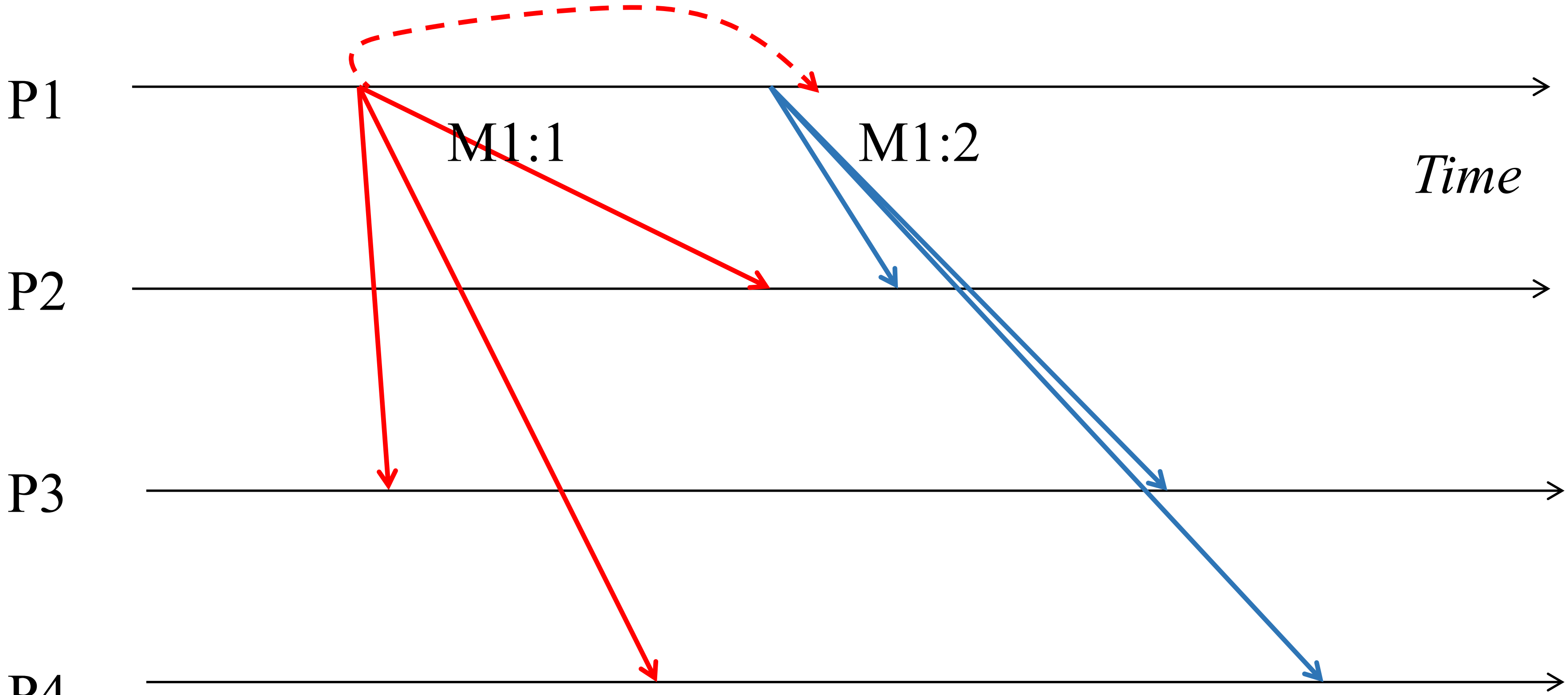


M1:1 is delivered at P3 after M3:1's multicast.

Does this satisfy causal order?

Yes

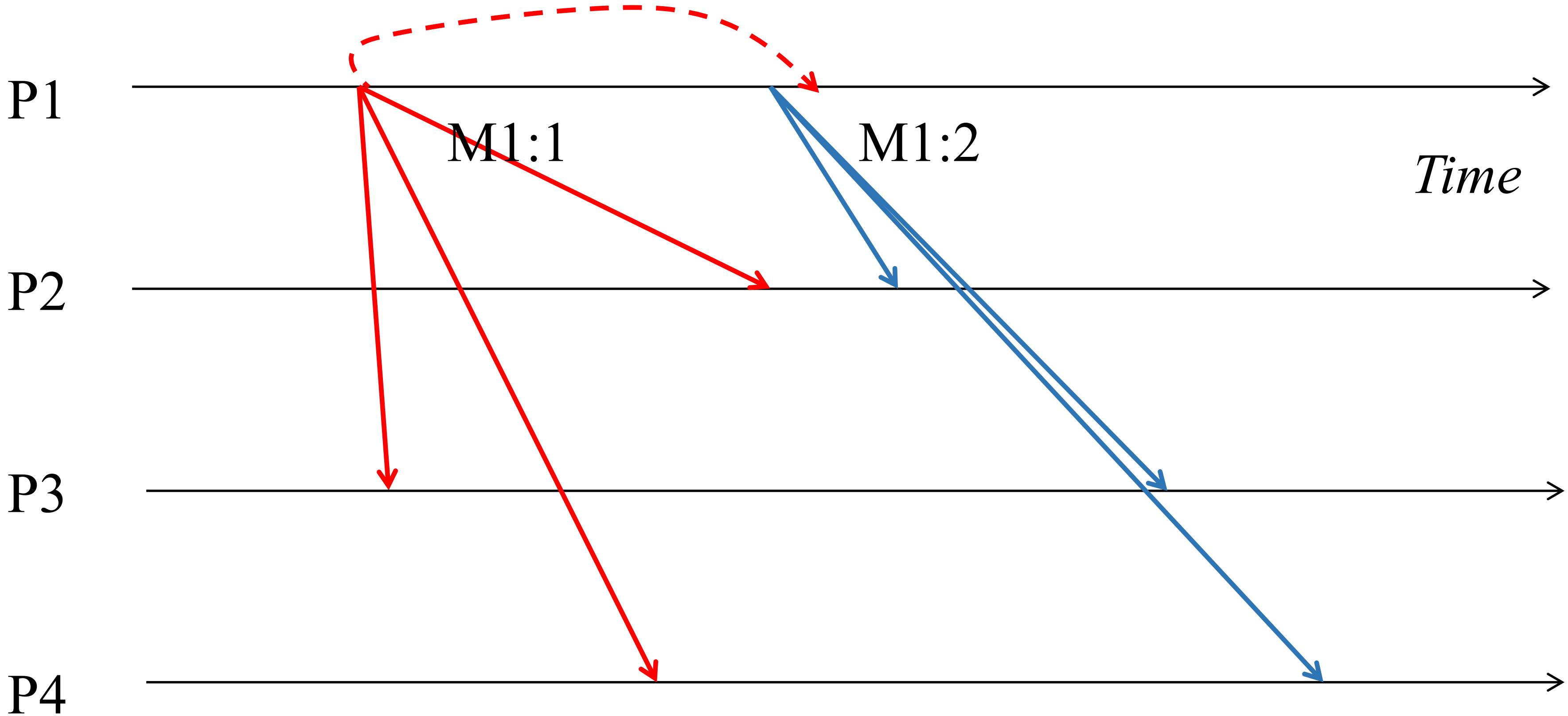
Example



Does this satisfy causal order?

No

Example



Does this satisfy FIFO order?

No

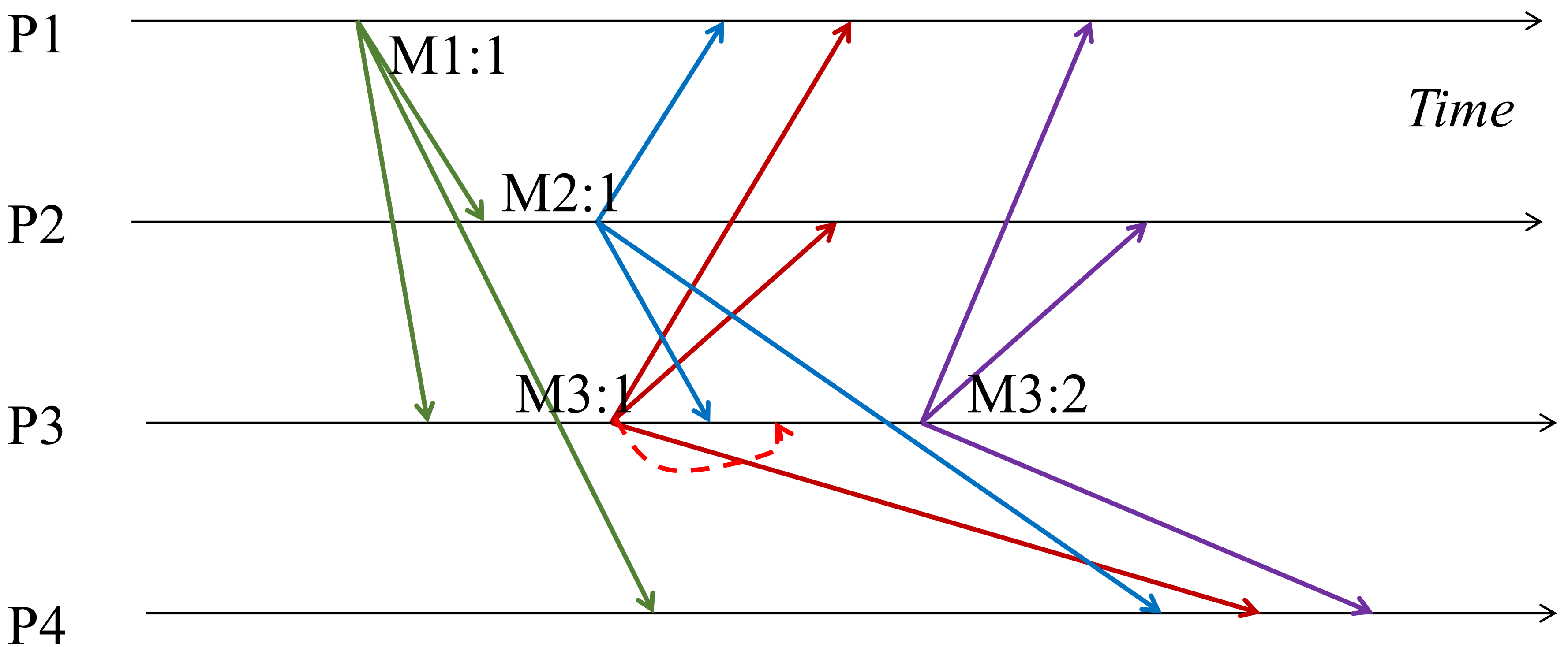
Recap: Ordered Multicast

- **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 multicast messages **delivered** to the application, rather than all network messages.
- Total ordering

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 sending order), 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.

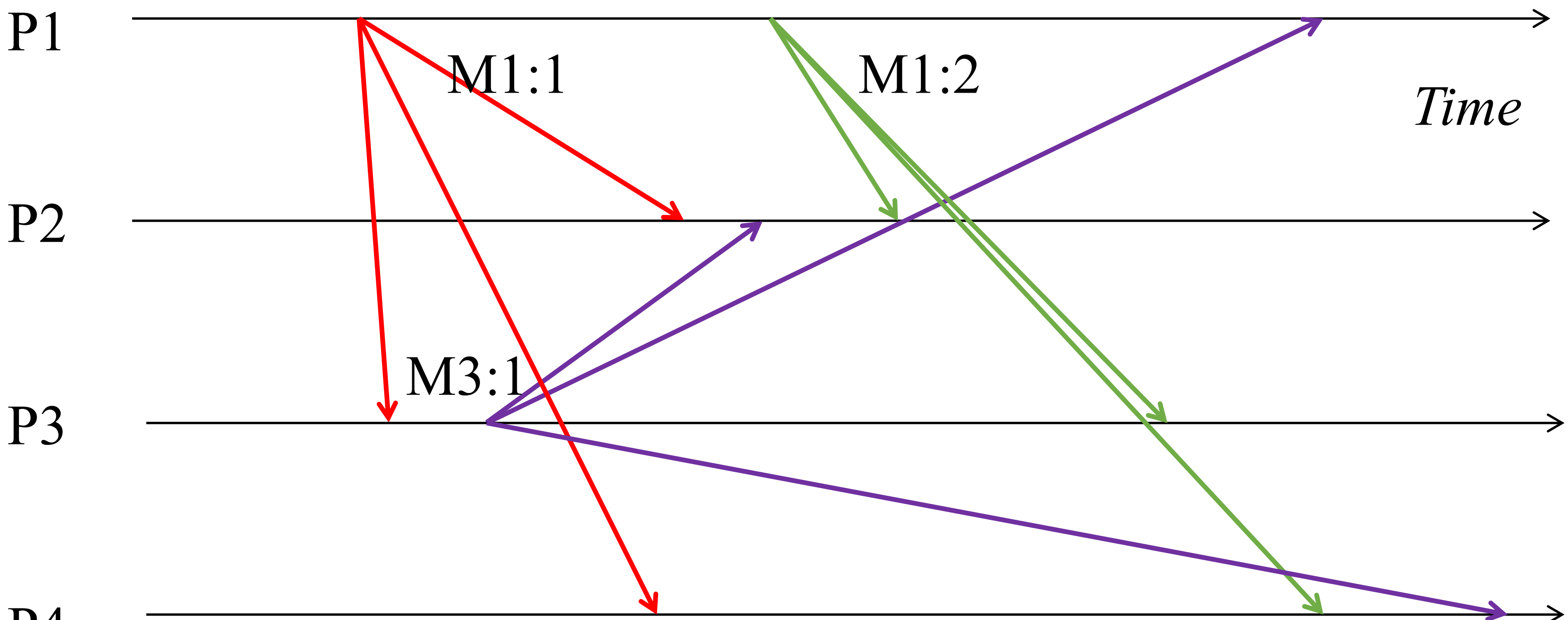
Hybrid variants

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

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'*, then any other correct process that delivers *m'* will have already delivered *m*.

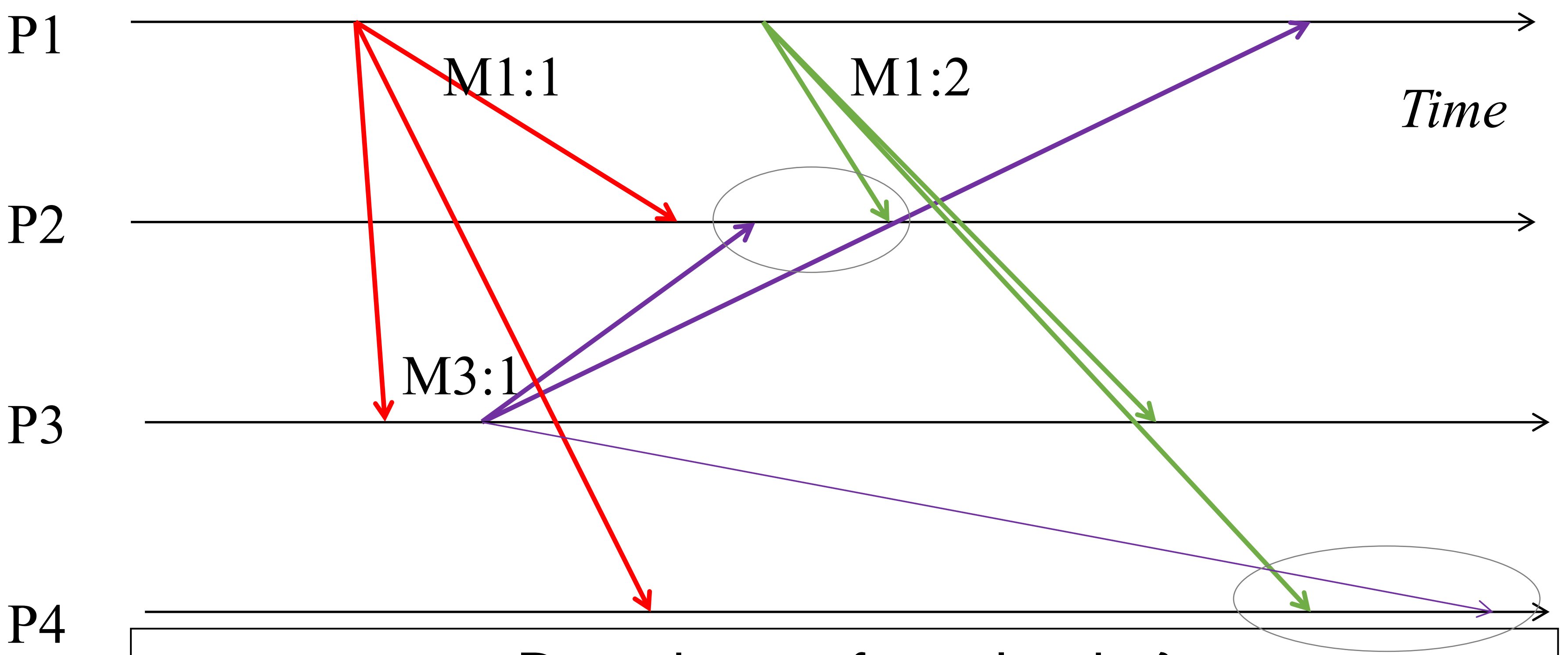
Example



Does this satisfy causal (and FIFO) order?

Yes

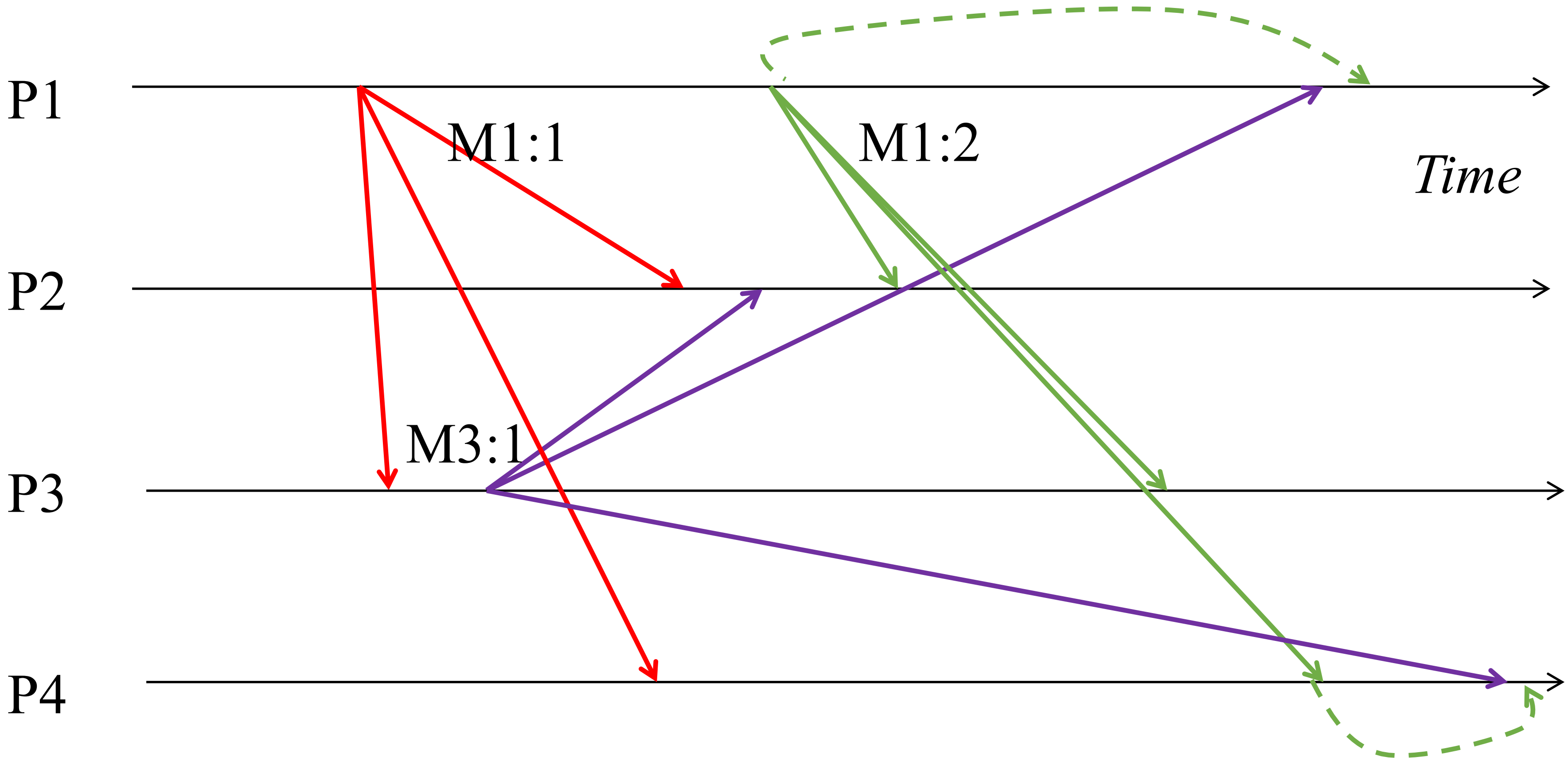
Example



Does this satisfy total order?

No

Example



Does this satisfy total order?

Yes

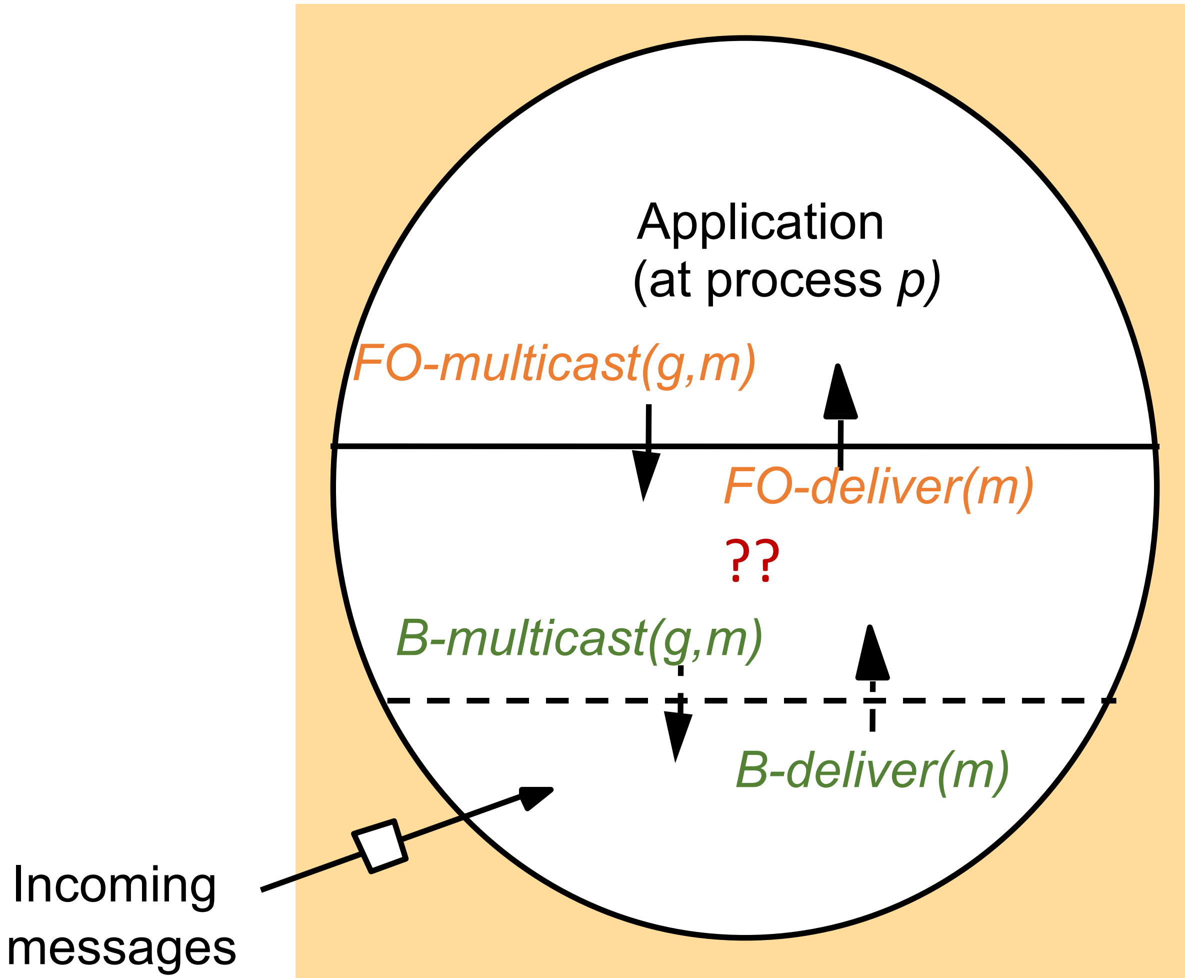
Next Question

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 multicast messages **delivered** to the application, rather than all network messages.
- **Total ordering**
  - If a correct process delivers message *m* before *m'* 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*

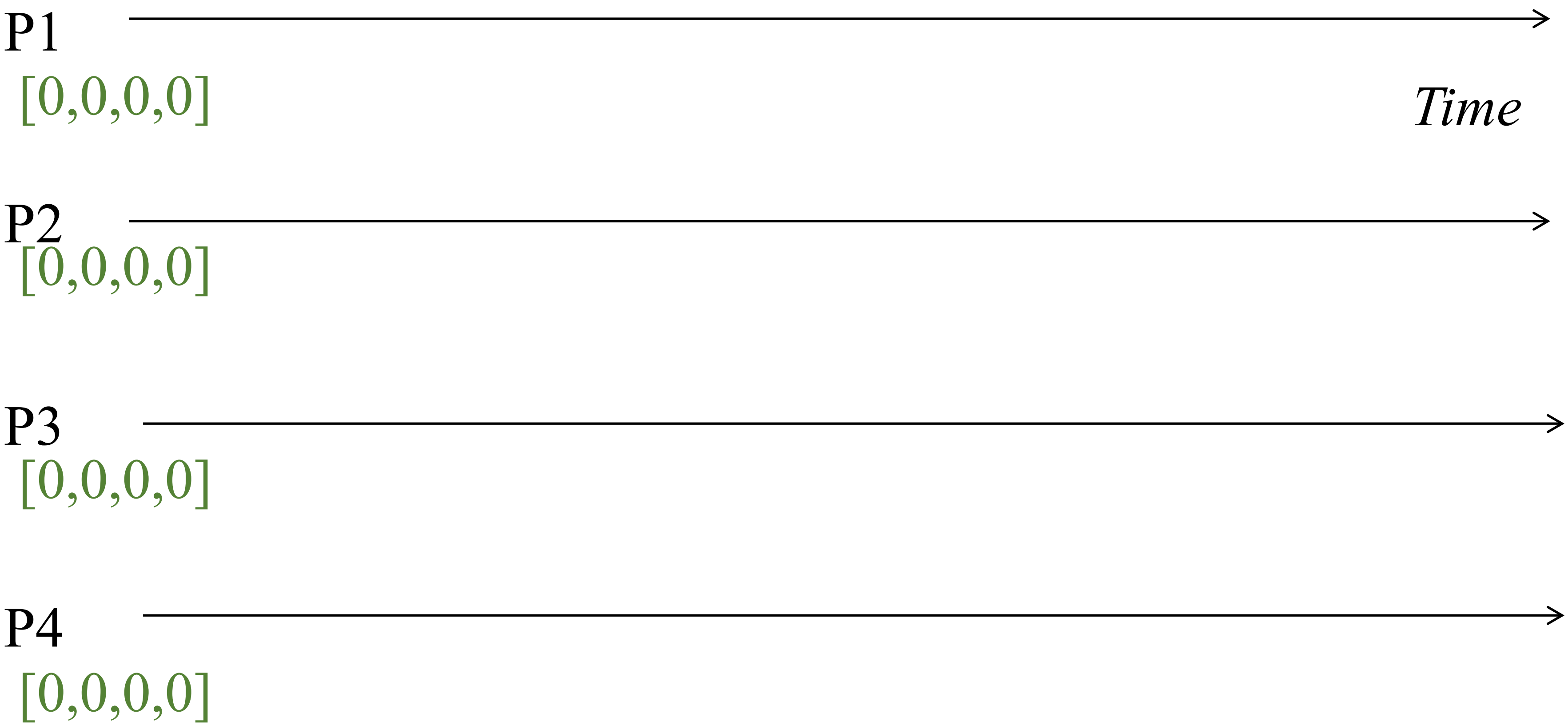
  if (S == P*i*[*j*] + 1) then

      FO-deliver(m) to application

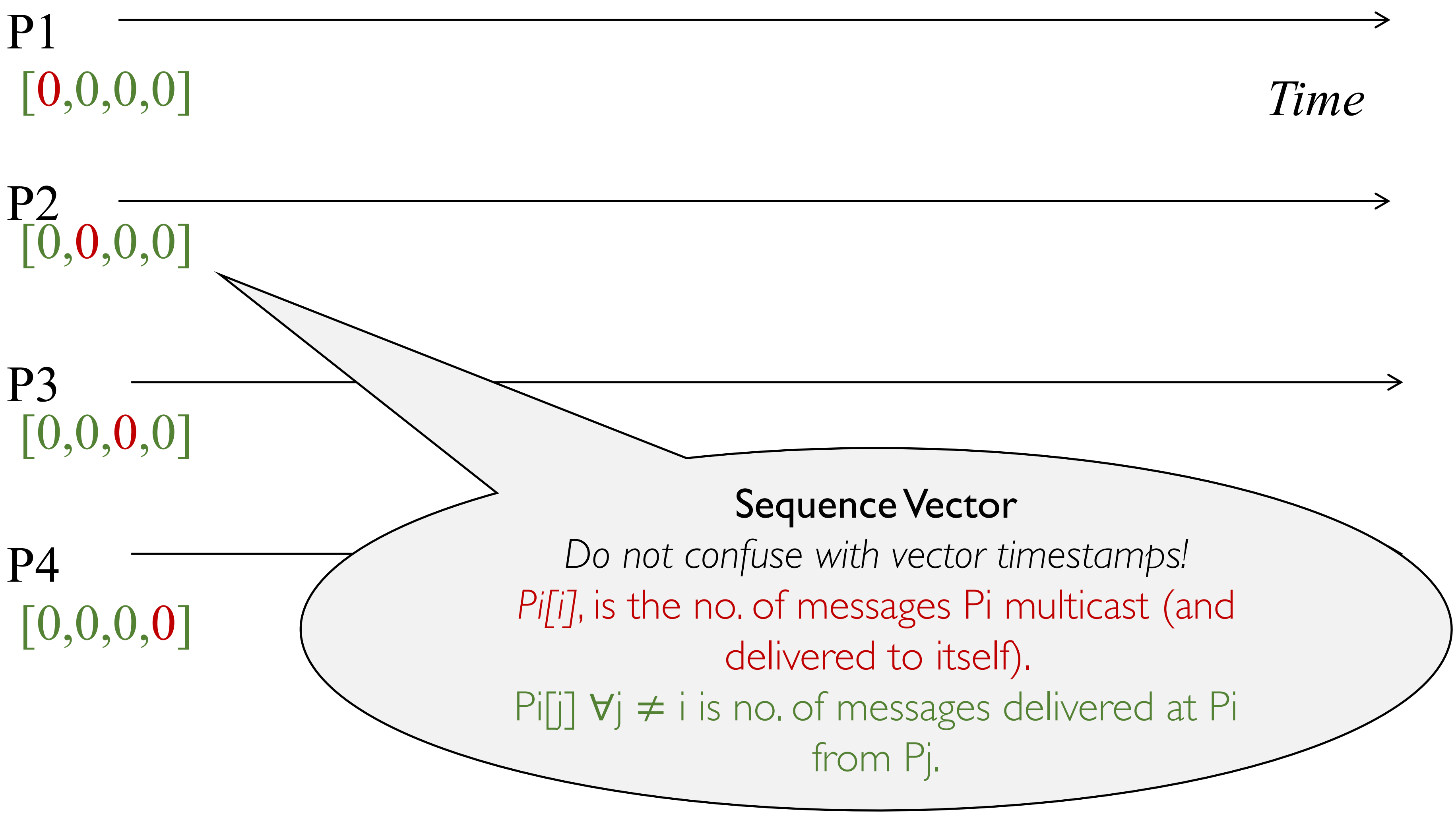
      set P*i*[*j*] = P*i*[*j*] + 1

  else buffer this multicast until above condition is true

FIFO order multicast execution

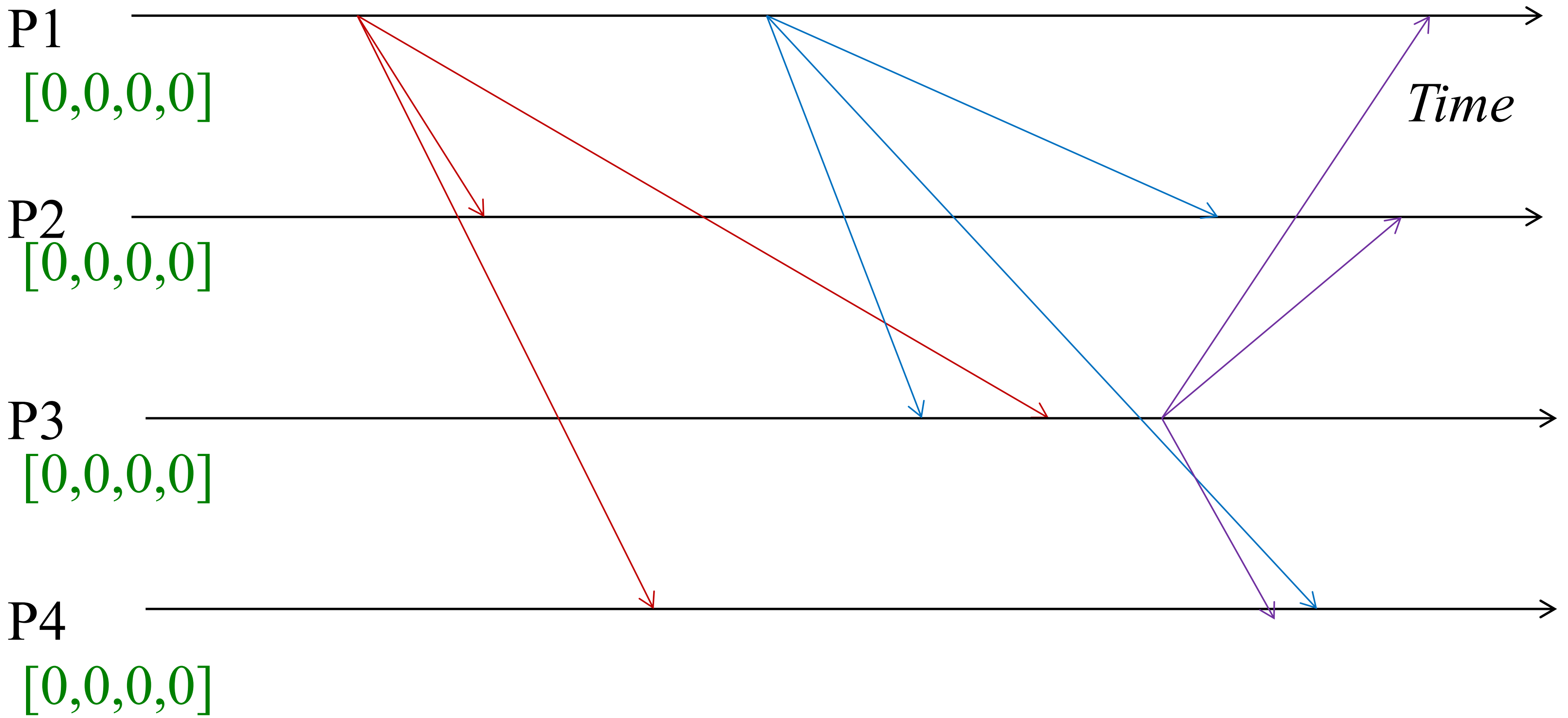


FIFO order multicast execution



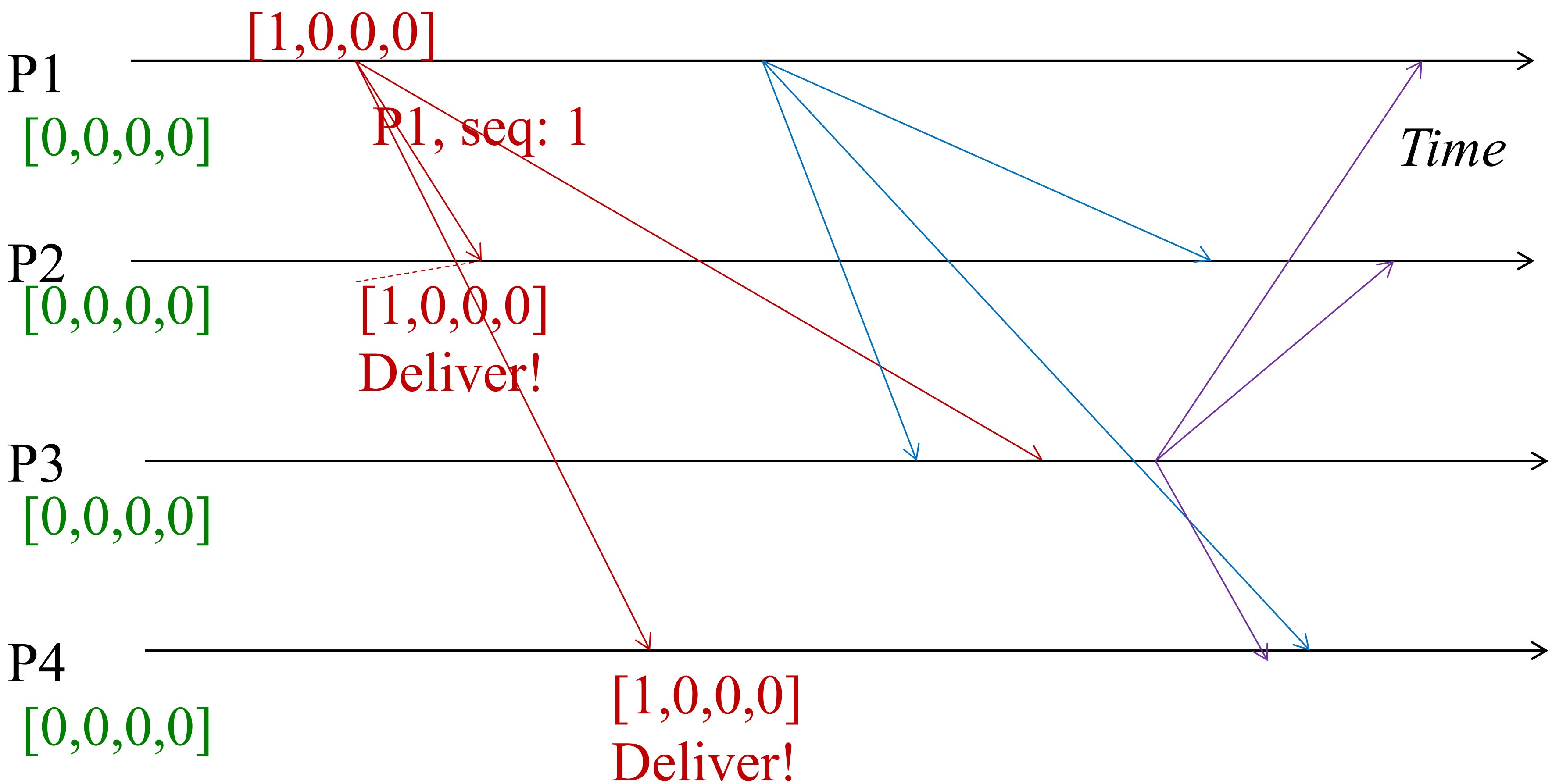
FIFO order multicast execution

FIFO order multicast execution

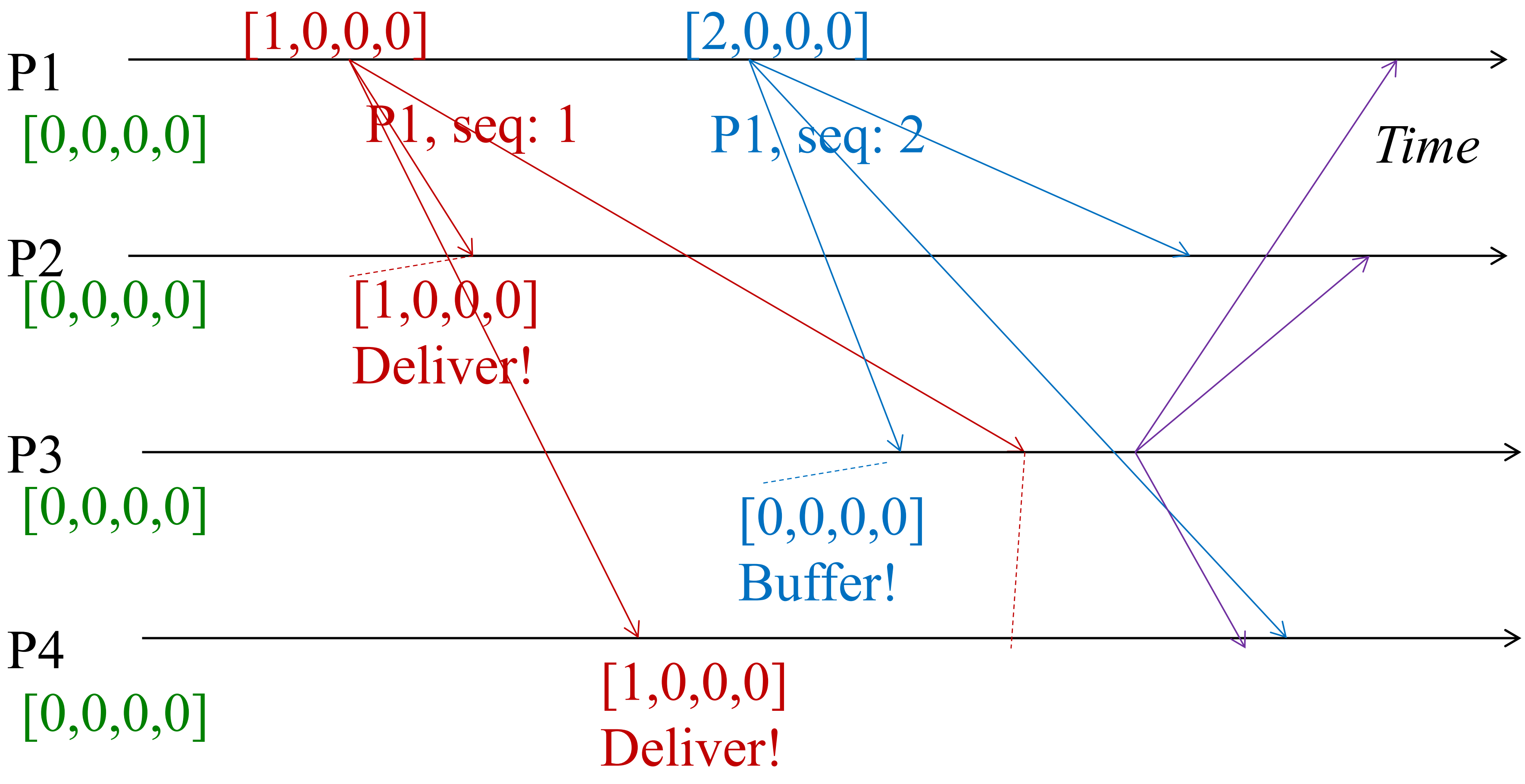


Self-deliveries omitted for simplicity.

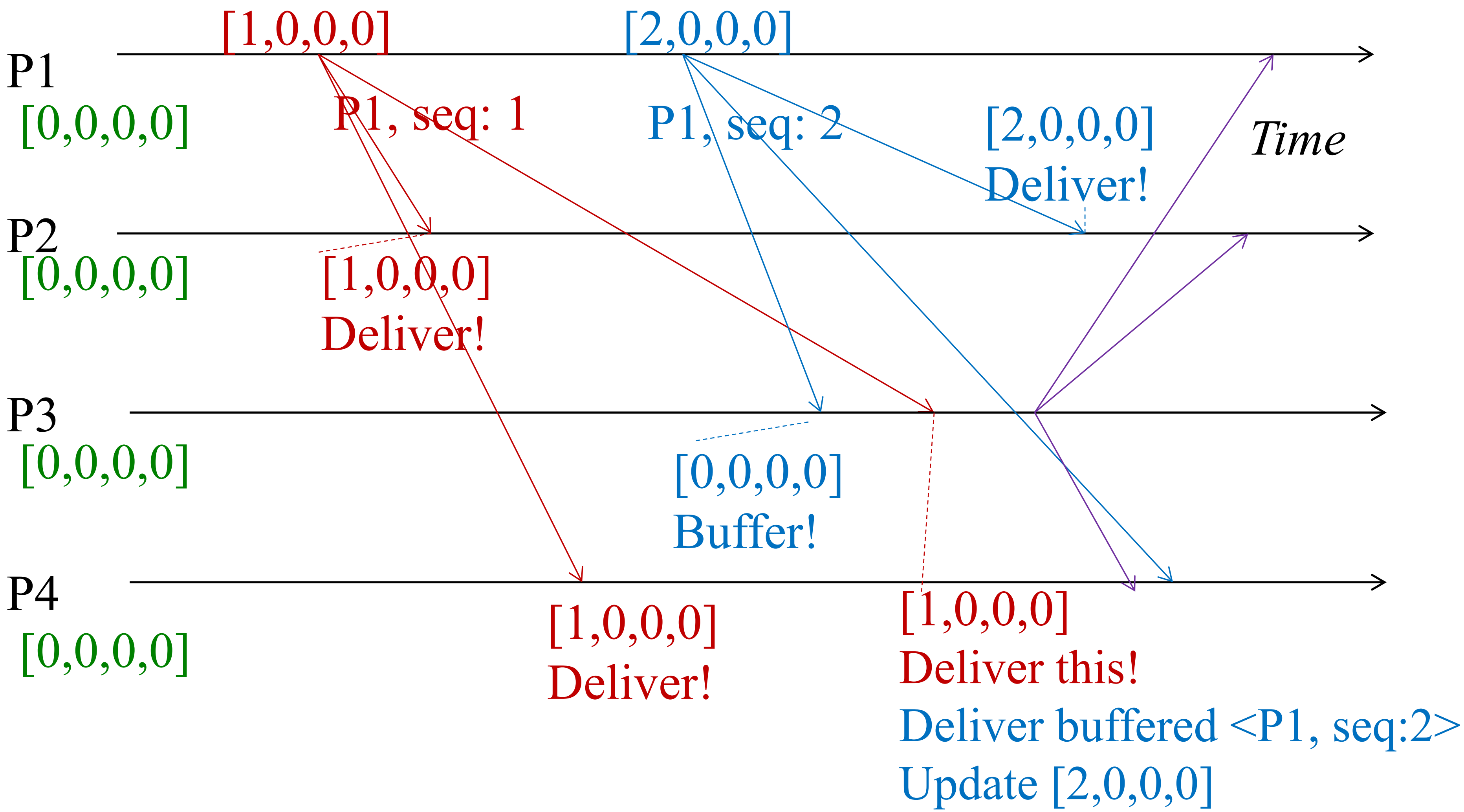
FIFO order multicast execution



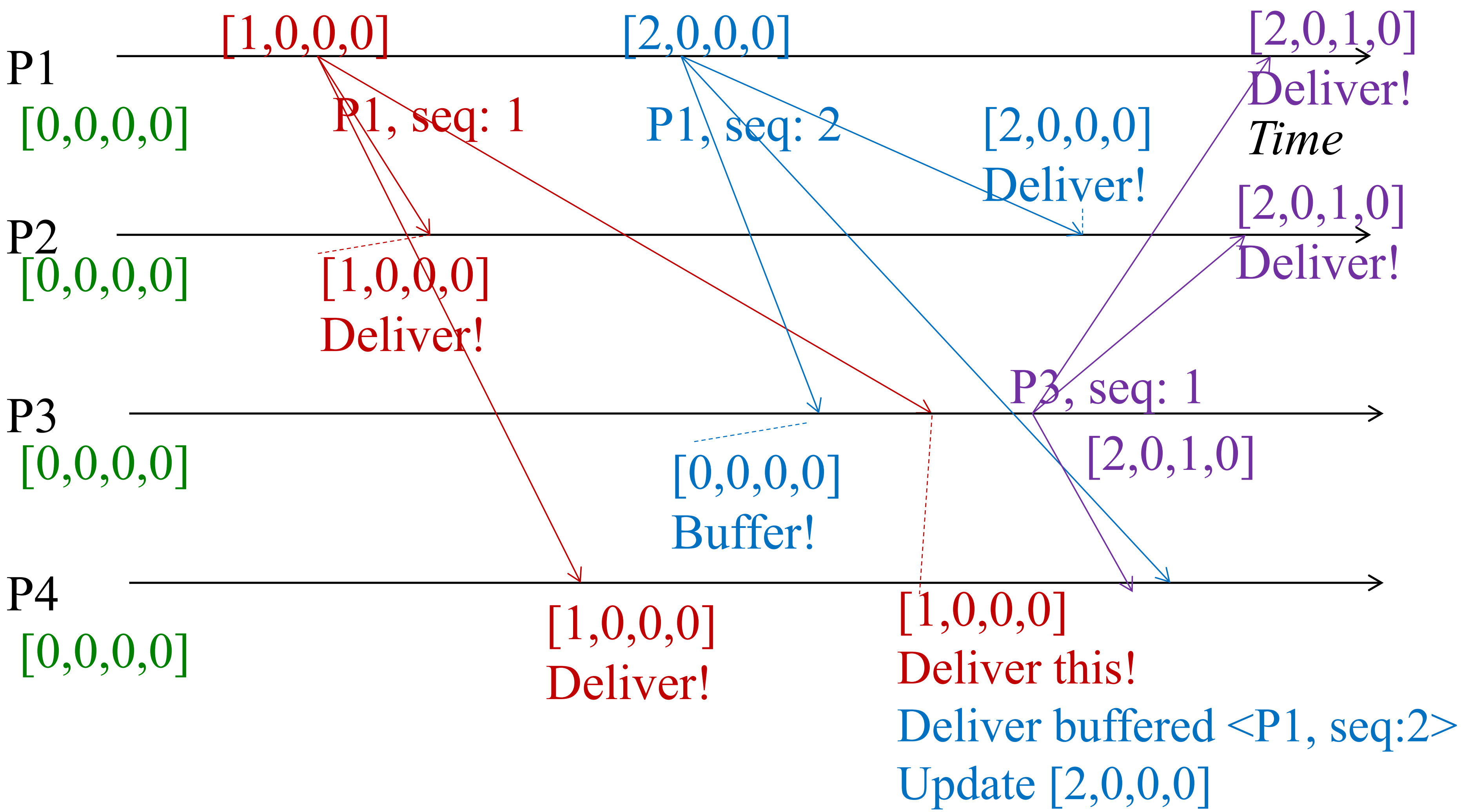
FIFO order multicast execution



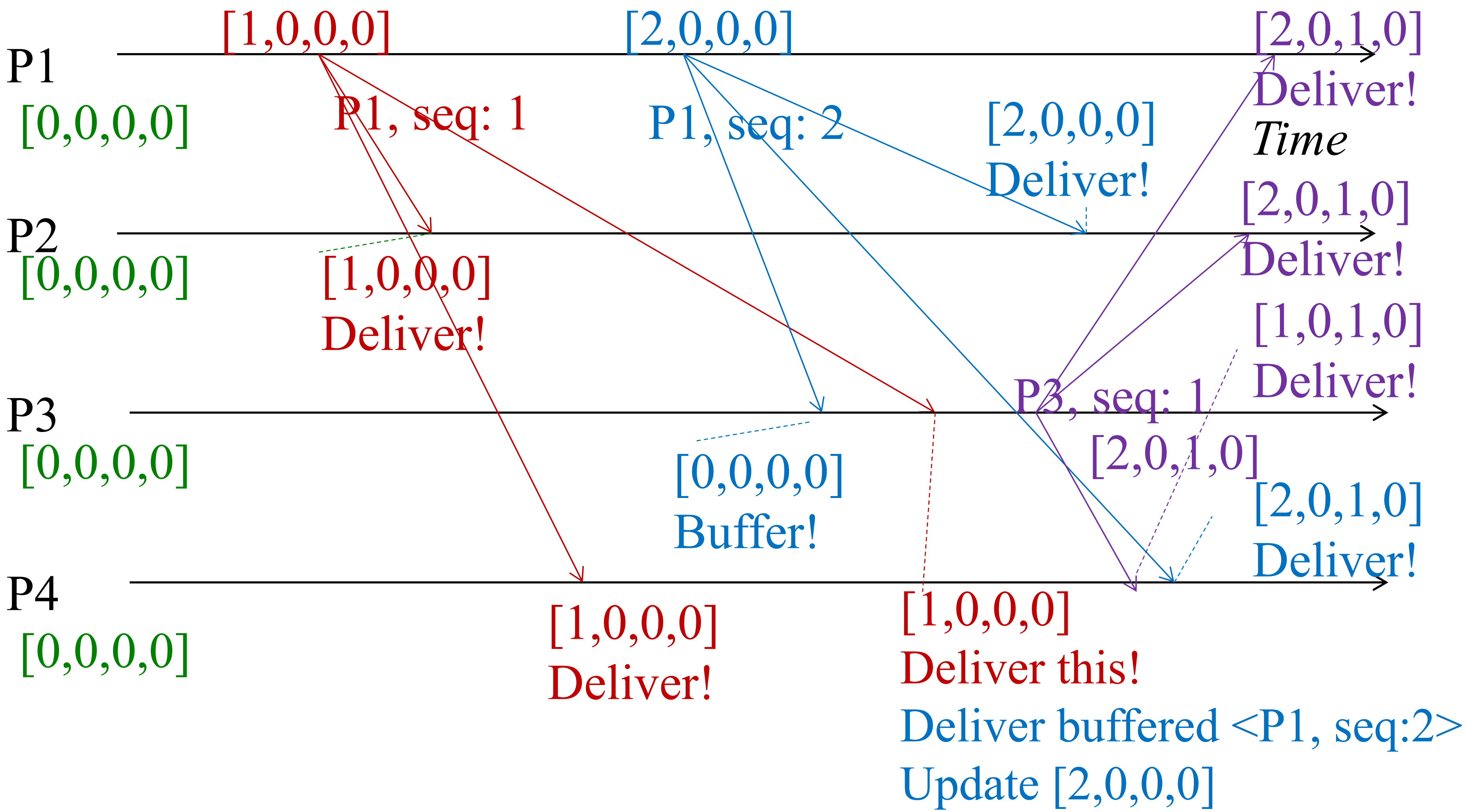
FIFO order multicast execution



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 multicast messages **delivered** to the application, rather than all network messages.

- **Total ordering**: If a correct process delivers message *m* before *m'* 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

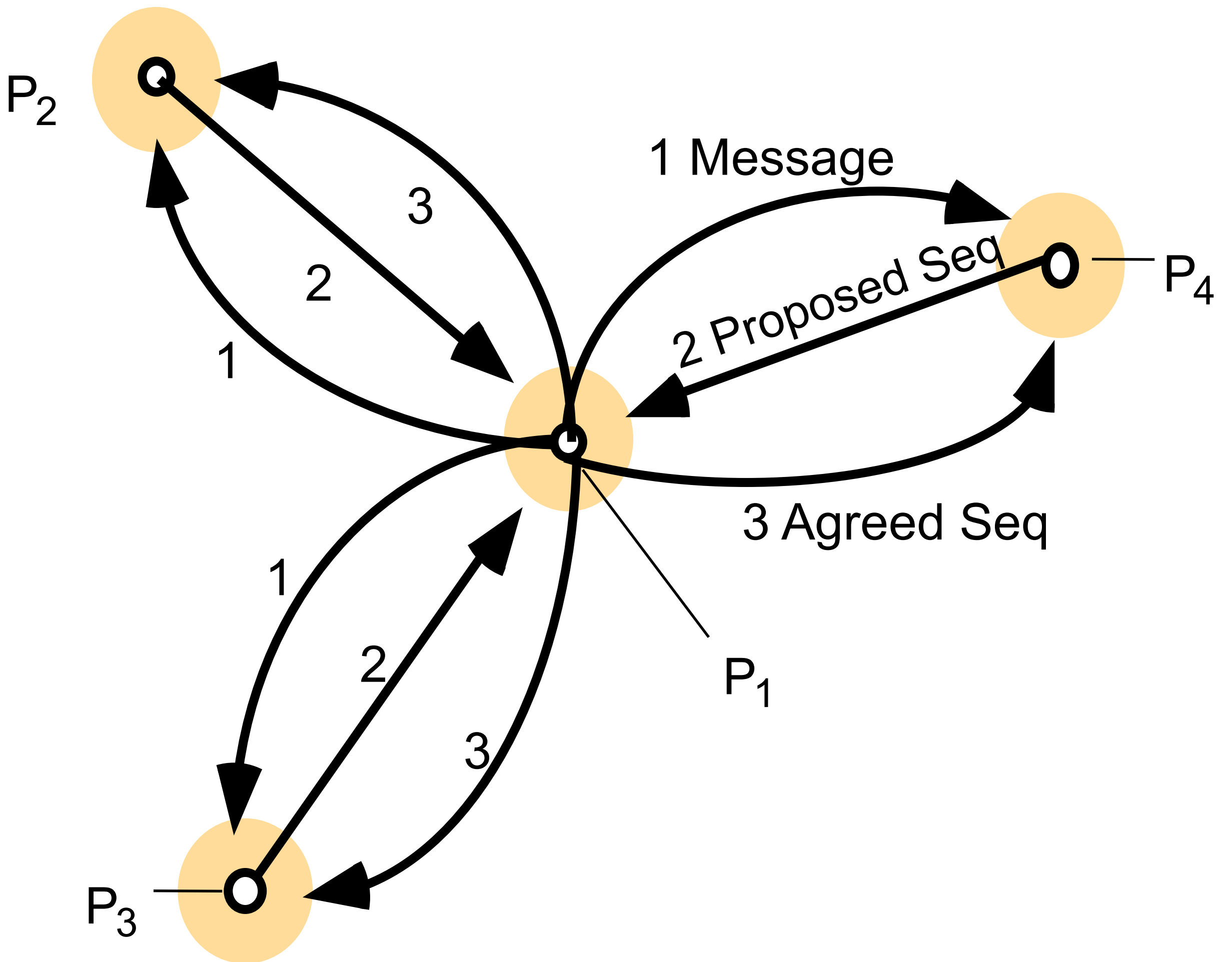
- Special process elected as leader or sequencer.
- TO-multicast(g,m) at P*i*:
  - B-multicast message m to group g *and the sequencer*
- Sequencer:
  - Maintains a global sequence number S (initially 0)
  - When a multicast message m is B-delivered to it:
    - sets S = S + 1, and B-multicast(g,{"order", 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)**

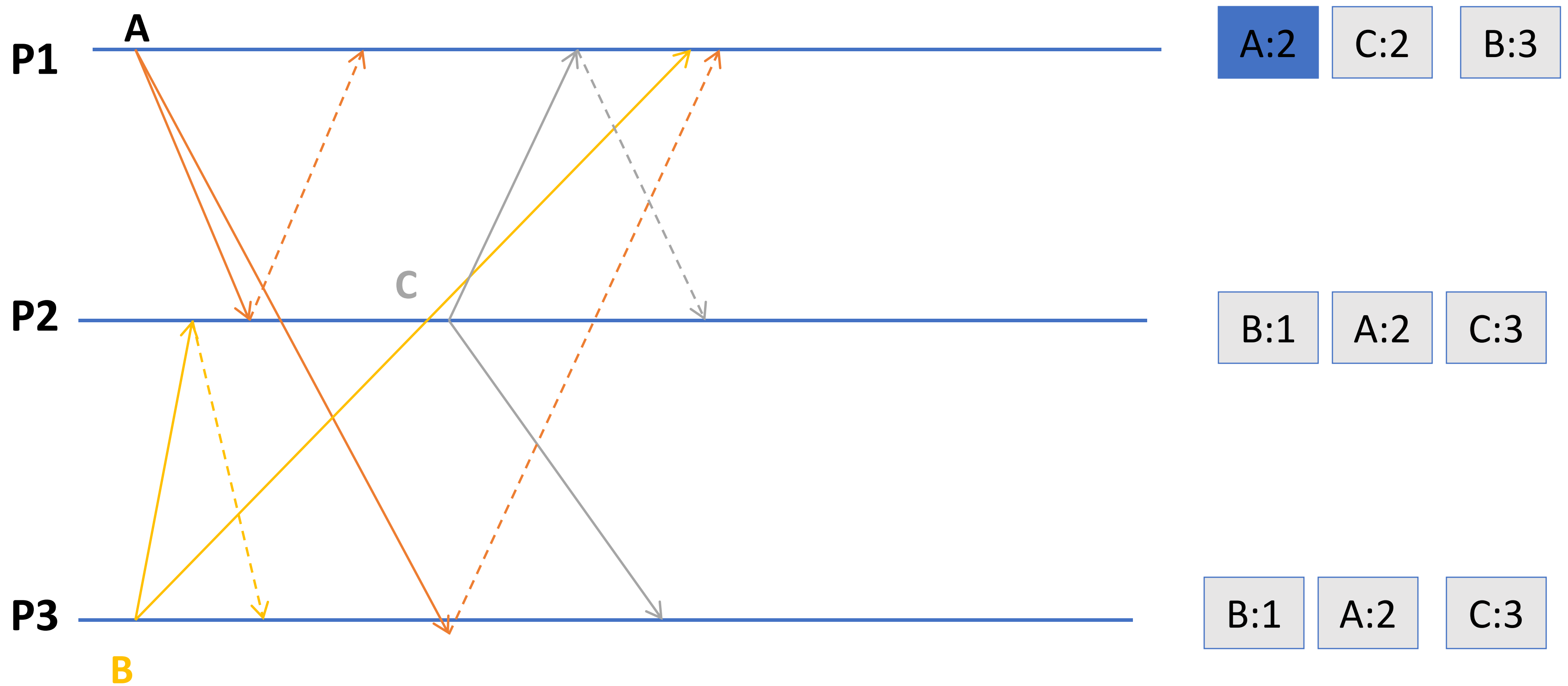
ISIS algorithm for total ordering



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 id with agreed priority
  - maximum of all proposed priorities
- Upon receiving agreed (final) priority for a message 'm'
  - Update m's priority to final, and accordingly reorder messages in queue.
  - mark the message m as deliverable.
  - deliver any deliverable messages at front of priority queue.

Example: ISIS algorithm



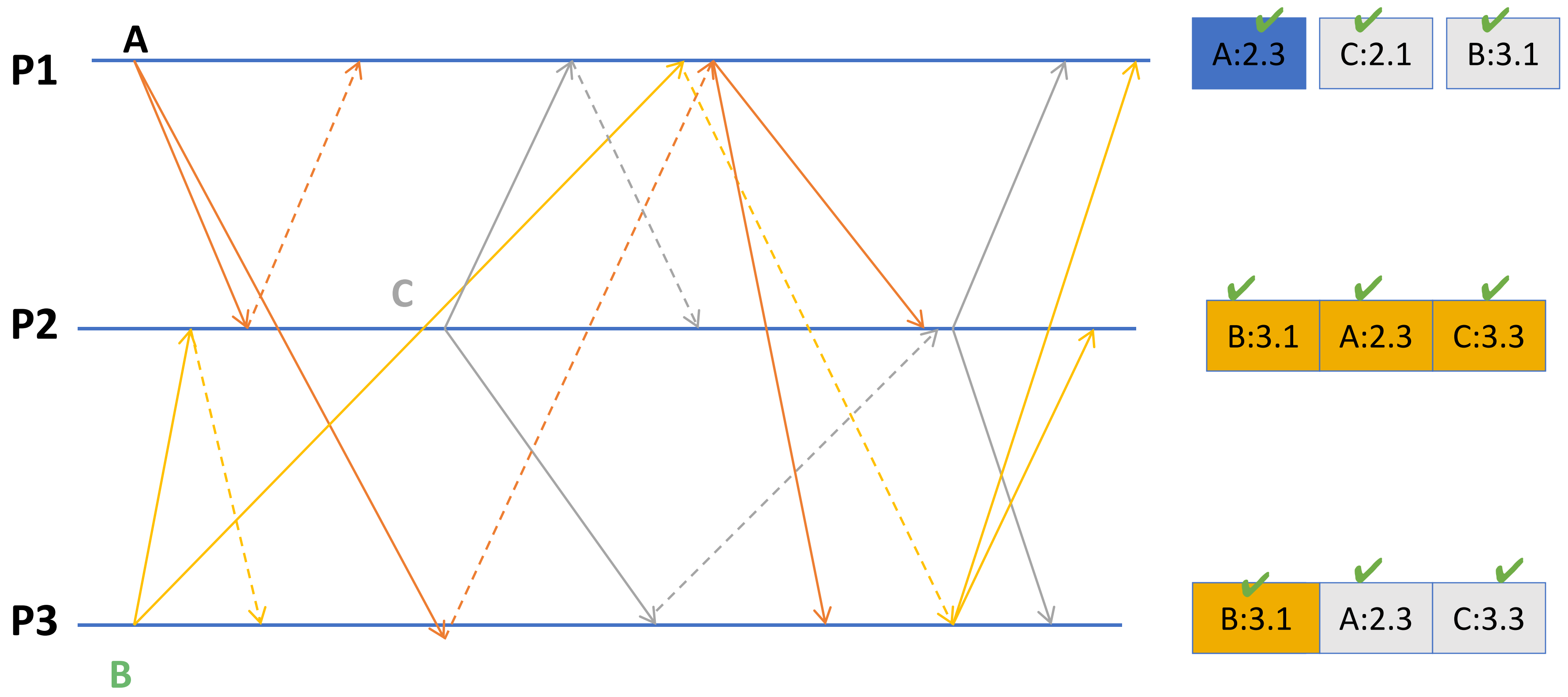
How do we break ties?

- 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



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 multicast messages **delivered** to the application, rather than all network messages.
- **Total ordering**
  - If a correct process delivers message m before m', then any other correct process that delivers m' will have already delivered m.

To be continued in next class

- Proof of total-ordering with ISIS.
- Implementation of causal order multicast.

Summary

- Multicast is an important communication mode in distributed systems.

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

MP1: Event Ordering

- https://courses.grainger.illinois.edu/cs425/sp2021/mps/mp1.html
- Lead TA: Sanchit Vohra

- Task:
  - Collect **transaction** events on distributed **nodes**.
  - **Multicast** transactions to all nodes while maintaining **total order**.
  - Ensure transaction **validity**.
  - Handle **failure** of arbitrary nodes.

- Objective:
  - Build a decentralized multicast protocol to ensure total ordering and handle node failures.

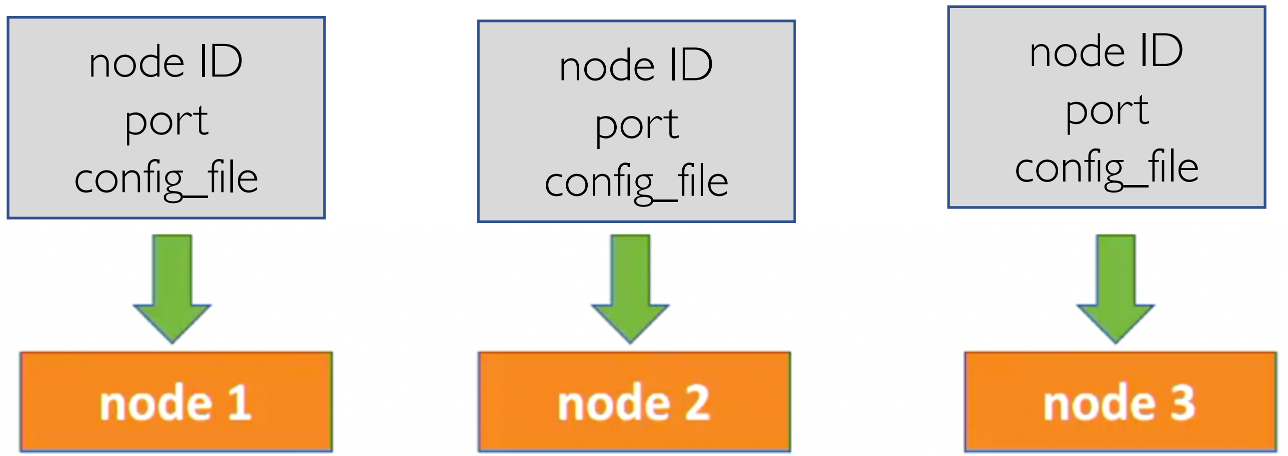
MP1: Event Ordering

- https://courses.grainger.illinois.edu/cs425/sp2021/mps/mp1.html
- Lead TA: Sanchit Vohra

- Task:
  - Collect **transaction** events on distributed **nodes**.
  - **Multicast** transactions to all nodes while maintaining **total order**.
  - Handle **failure** of arbitrary nodes: multicast must be **reliable**.
  - Ensure **transaction validity**.

- Objective:
  - Build a decentralized multicast protocol to ensure total ordering and handle node failures.

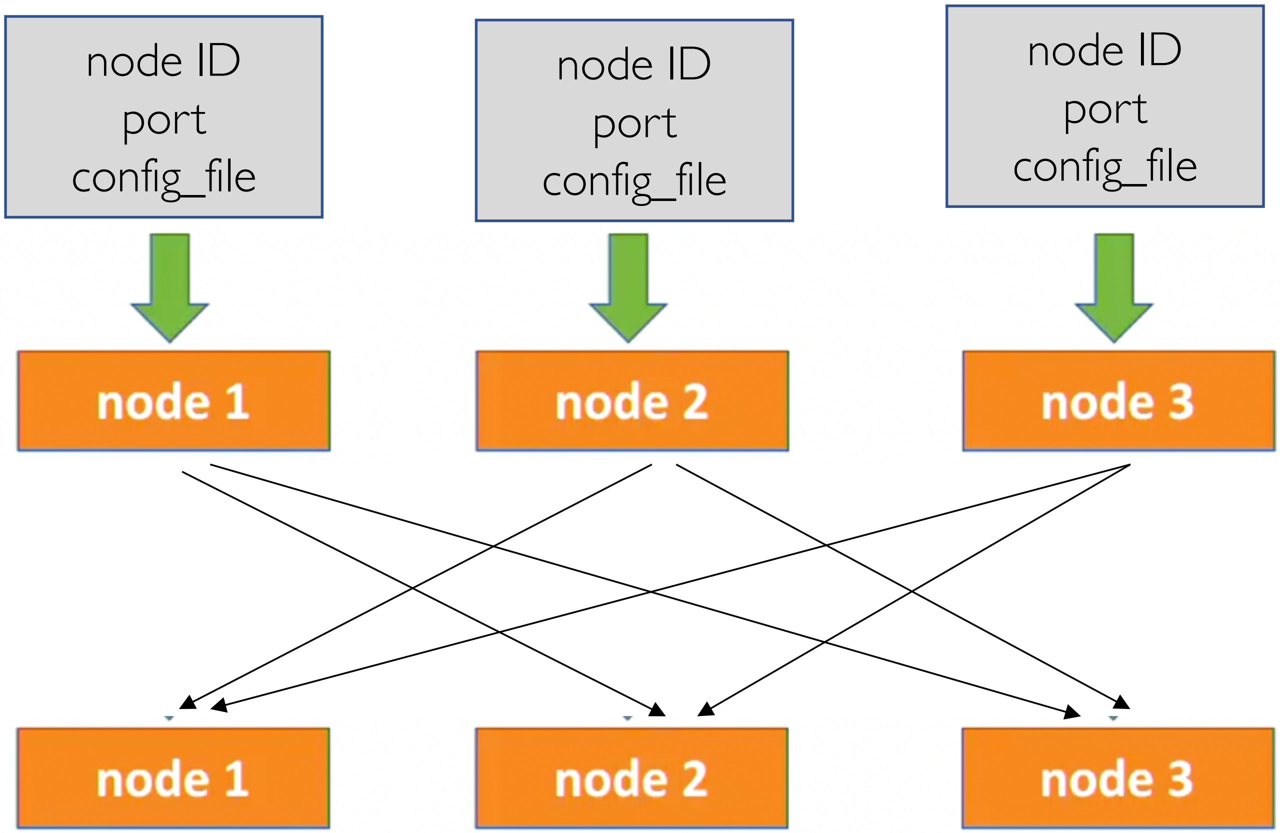
MPI Architecture Setup



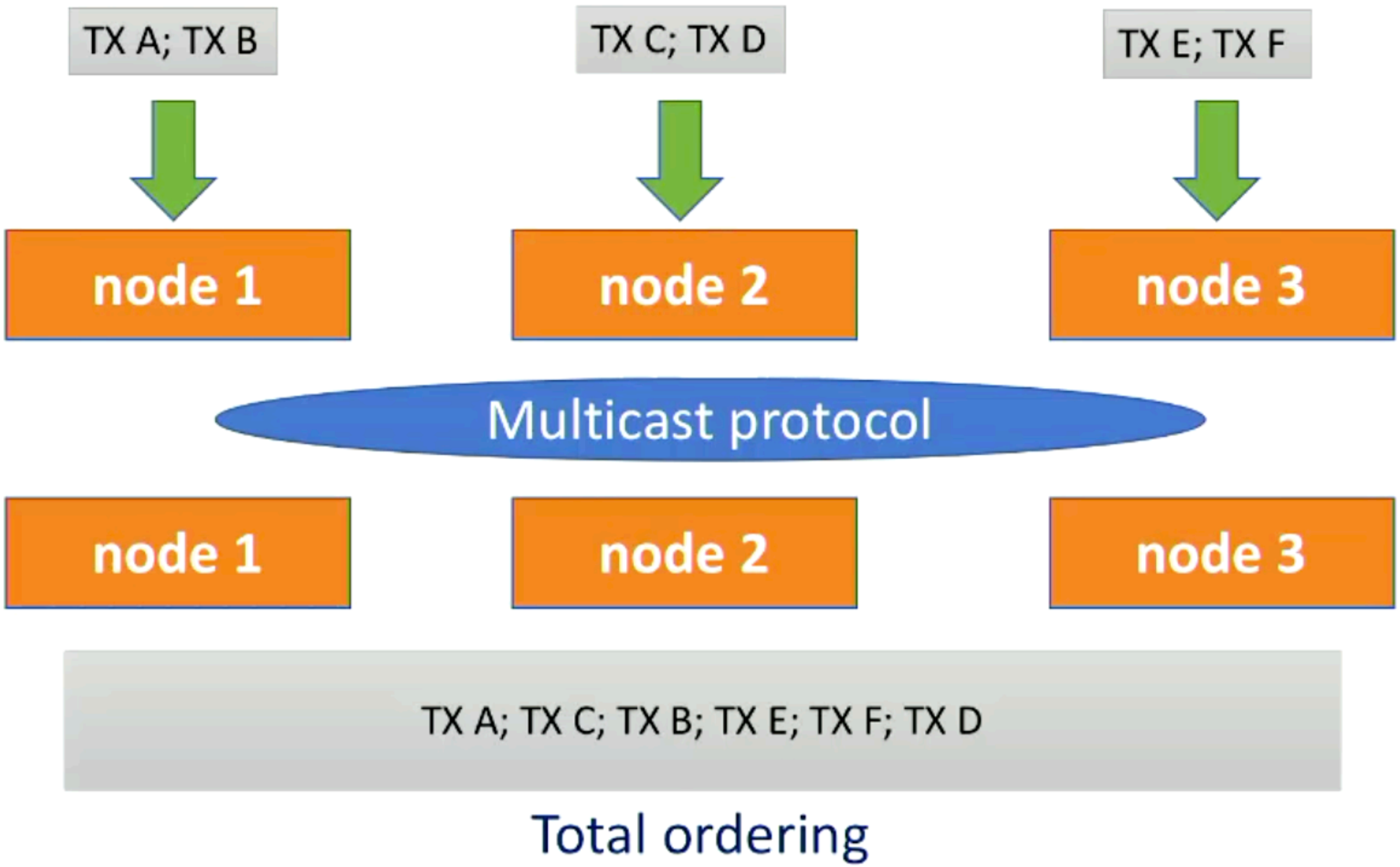
- Example input arguments for first node:
  `./node node1 1234 config1.txt`
- config1.txt looks like this:

```
2
node2 sp21-cs425-g01-02.cs.illinois.edu 1234
node3 sp21-cs425-g01-03.cs.illinois.edu 1234
```

MPI Architecture Setup



MP1 Architecture



Transaction Validity

DEPOSIT abc 100	Adds **100** to account abc (or creates a new abc account)
TRANSFER abc -> def 75	Transfers **75** from account abc to account def (creating if needed)
TRANSFER abc -> ghi 30	Invalid transaction, since abc only has **25** left

Transaction Validity: ordering matters

DEPOSIT xyz 50
TRANSFER xyz -> wqr 40
TRANSFER xyz -> hjk 30
[invalid TX]

BALANCES xyz:10 wqr:40

DEPOSIT xyz 50
TRANSFER xyz -> hjk 30
TRANSFER xyz -> wqr 40
[invalid TX]

BALANCES xyz:20 hjk:30

Graph

- Compute the "processing time" for each transaction:
  - Time difference between when it was generated (read) at a node, and when it was **processed** by the last node.
- Plot the CDF (cumulative distribution function) of the transaction processing time for each evaluation scenario.

MP1: Logistics

- Due on Wednesday, March 17th.
  - Allowed to submit up to 50 hours late, but with 2% penalty for every late hour (rounded up).
- You are allowed to reuse code from MP0.
  - We will release a Go solution for MP0.
  - Note: this MP1 requires all nodes to connect to each other, as opposed to each node connecting to a central logger.
- Read the specification carefully. Start early!!