Distributed Systems

CS425/ECE428

Feb 13 2023

Instructor: Radhika Mittal

Acknowledgements for some of materials: Indy Gupta and Nikita Borisov

Logistics

- MP1 has been released.
  - Due on March 6th, 11:59pm.

- HW1 is due on Wednesday.

Today's agenda

- **Multicast**
  - Chapter 15.4
- **Goal:** reason about desirable properties for message delivery among a group of processes.

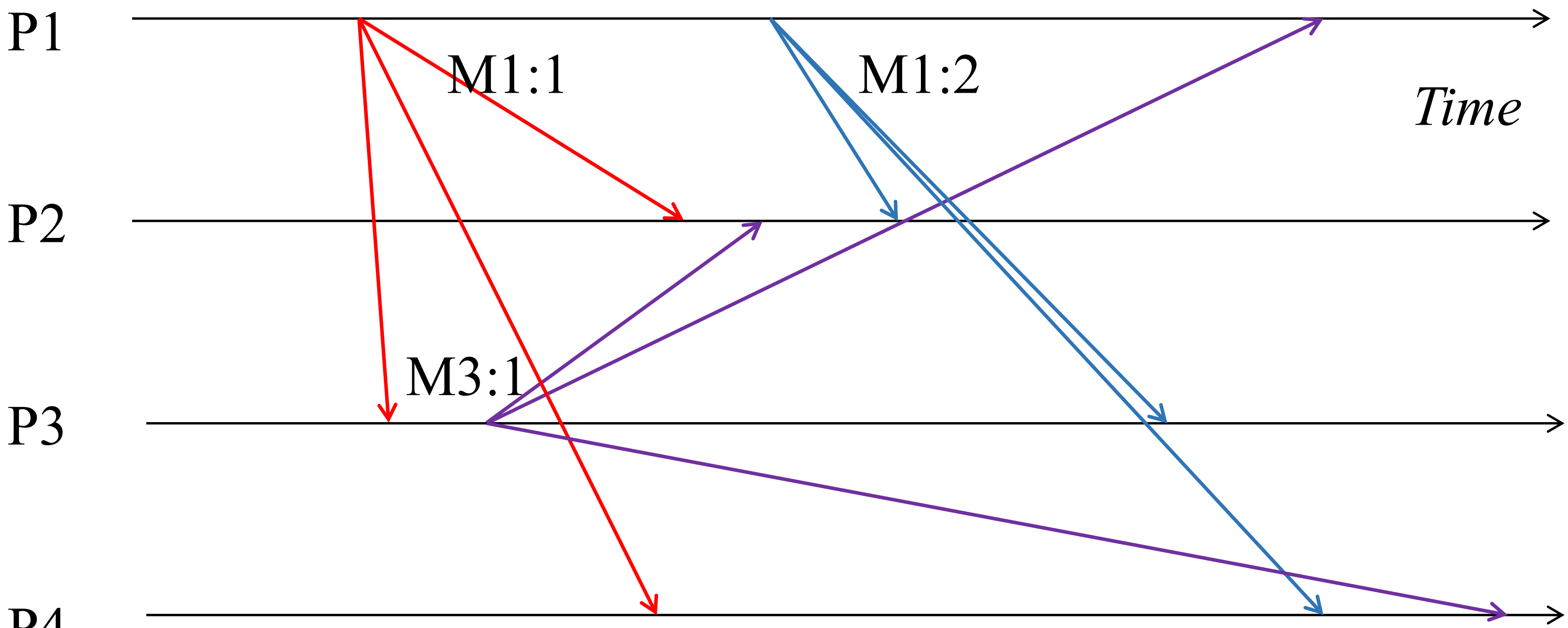
Recap: Multicast

- 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 **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**: If a correct process delivers message *m* before *m'*, then any other correct process that delivers *m'* will have already delivered *m*.

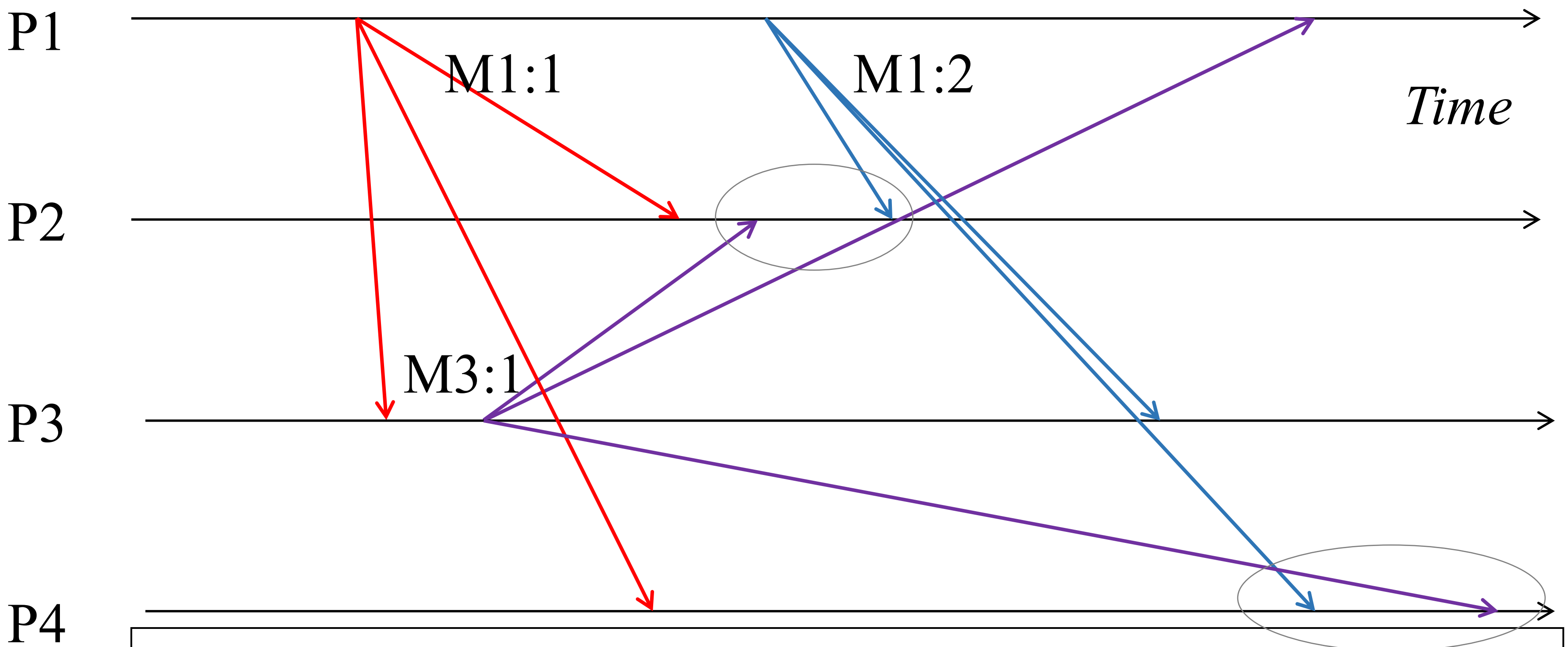
Example



Does this satisfy causal (and FIFO) order?

Yes

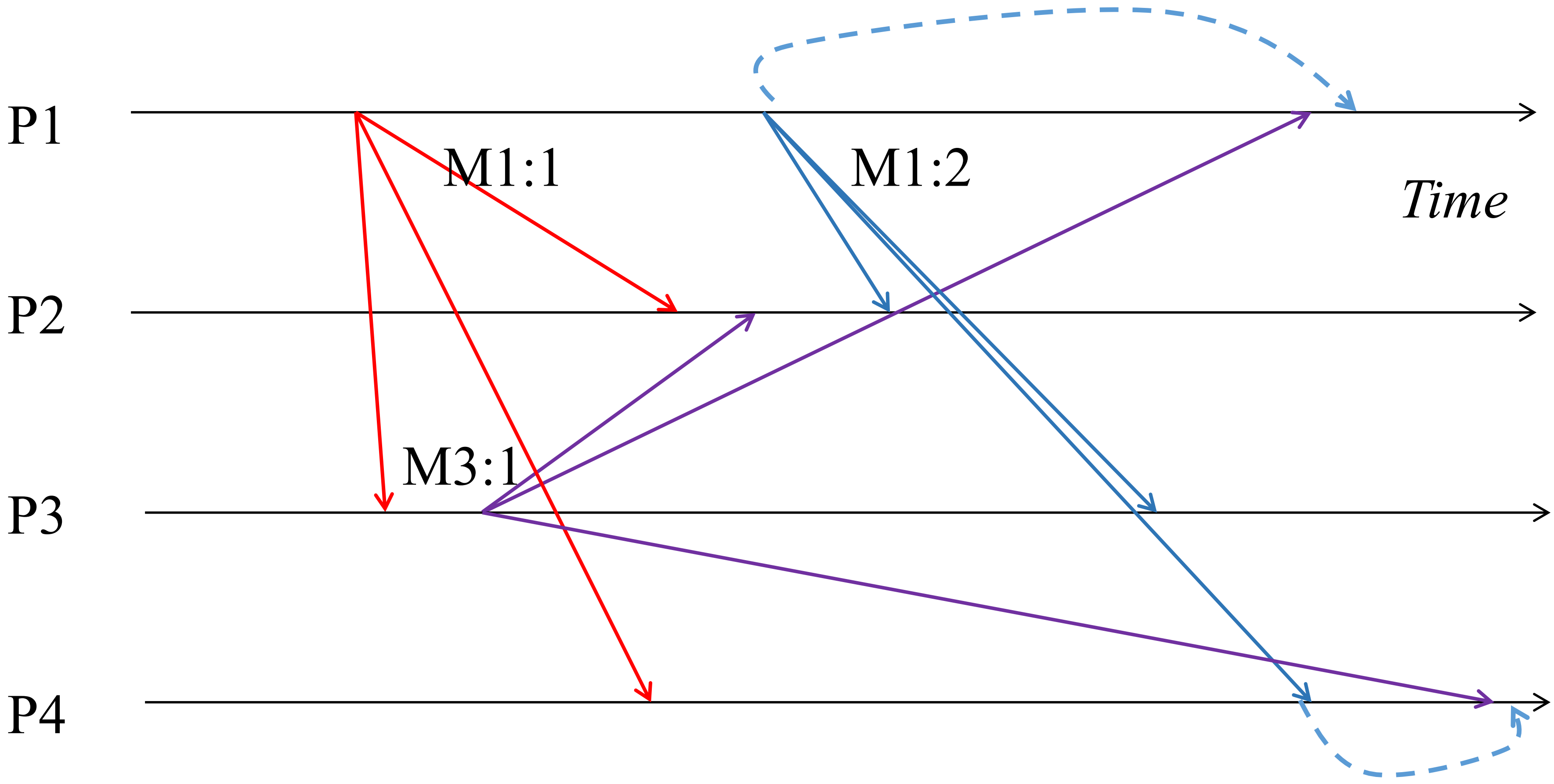
Example



Does this satisfy total order?

No

Example



Does this satisfy total order?
Yes

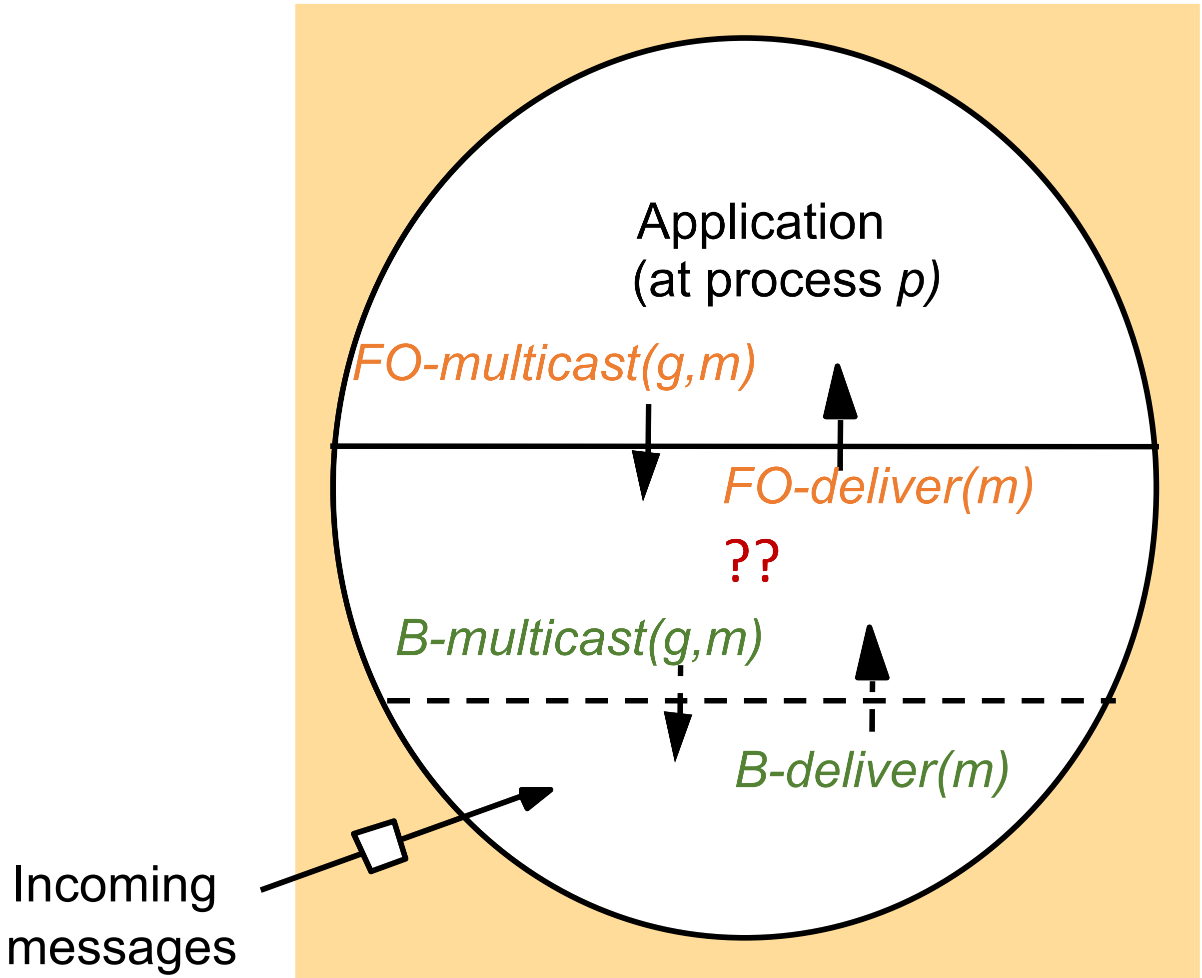
Next Question

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*.

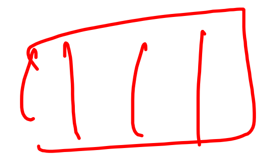
Implementing FIFO order multicast



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*

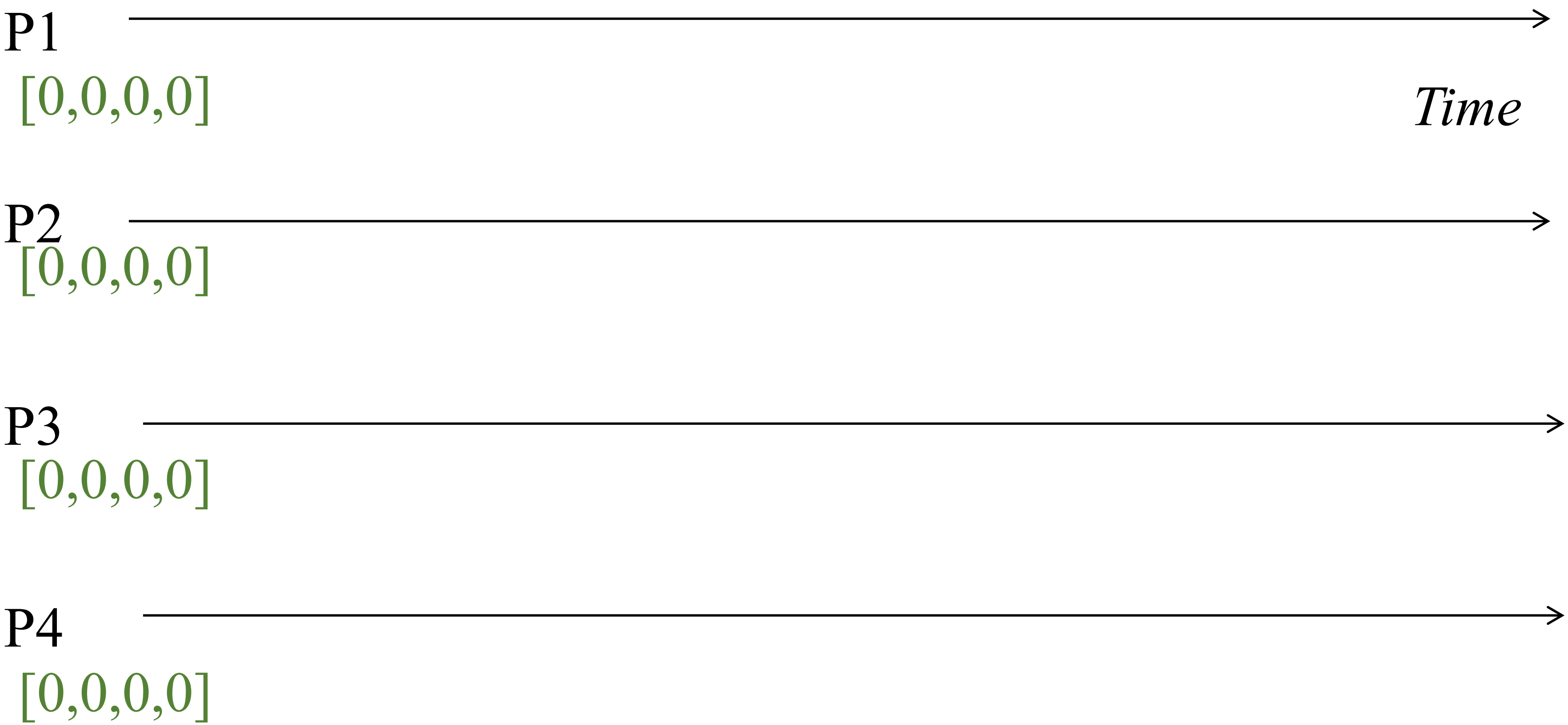
  if (S == P*i*[*j*] + 1) then

      FO-deliver(m) to application

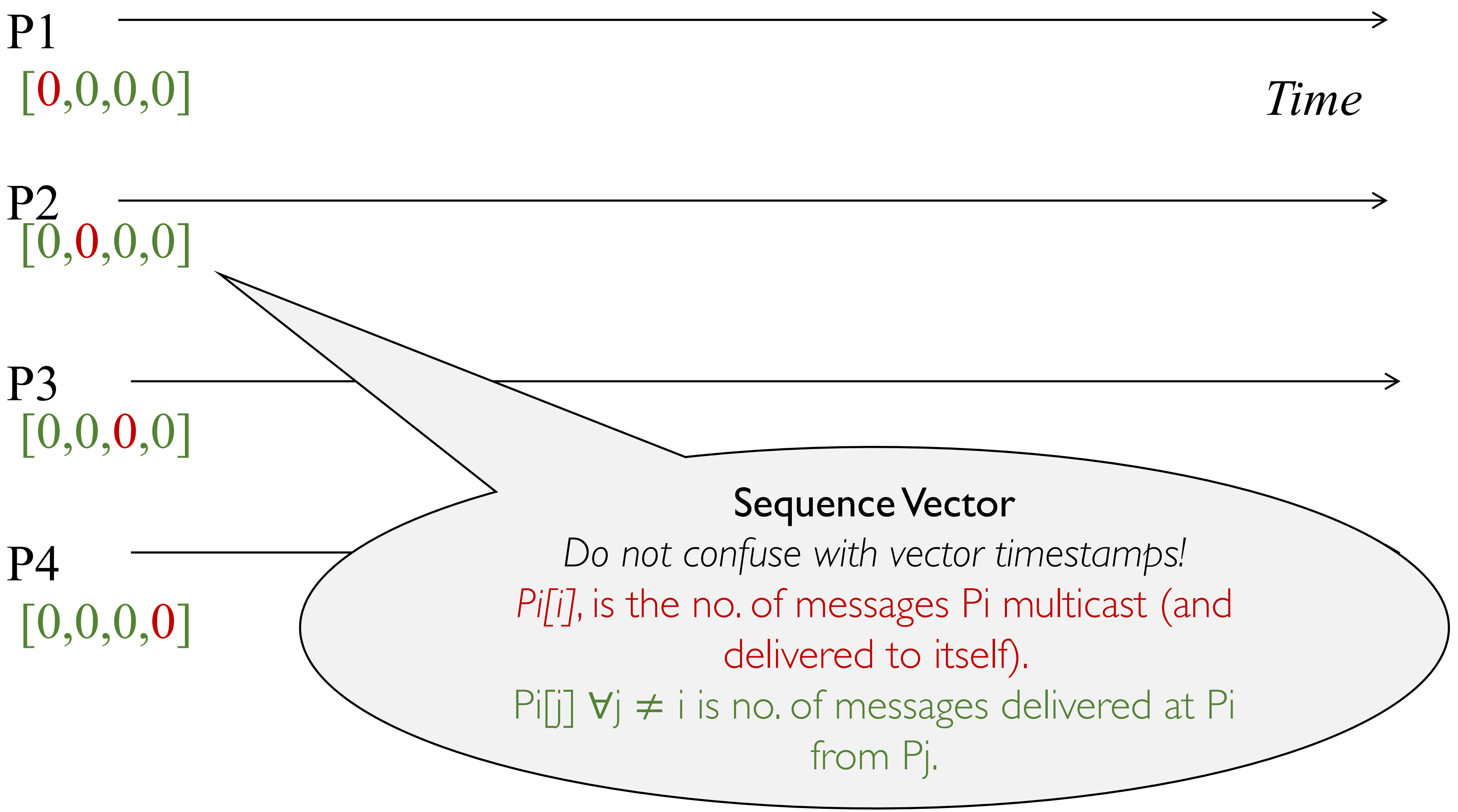
      set P*i*[*j*] = P*i*[*j*] + 1

  else buffer this multicast until above condition is true

FIFO order multicast execution

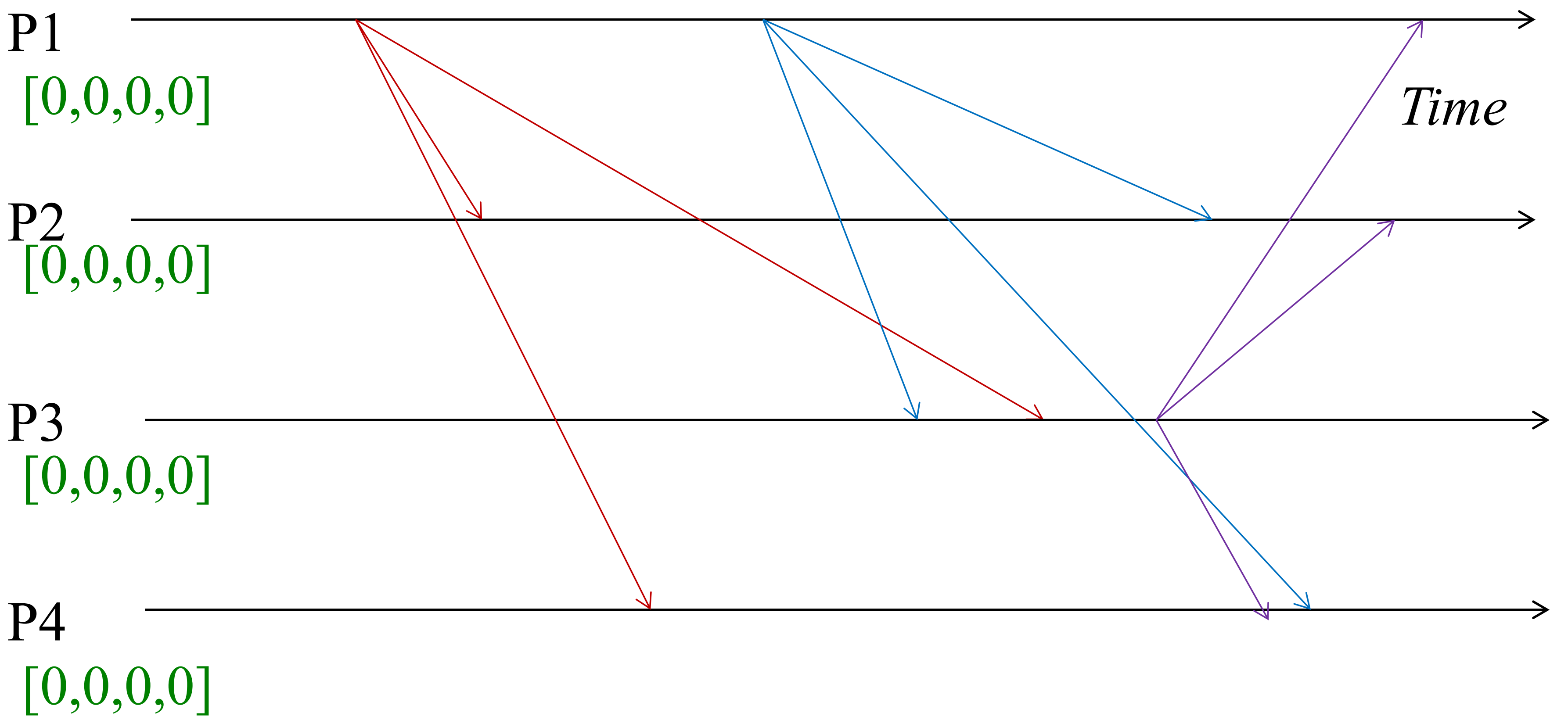


FIFO order multicast execution



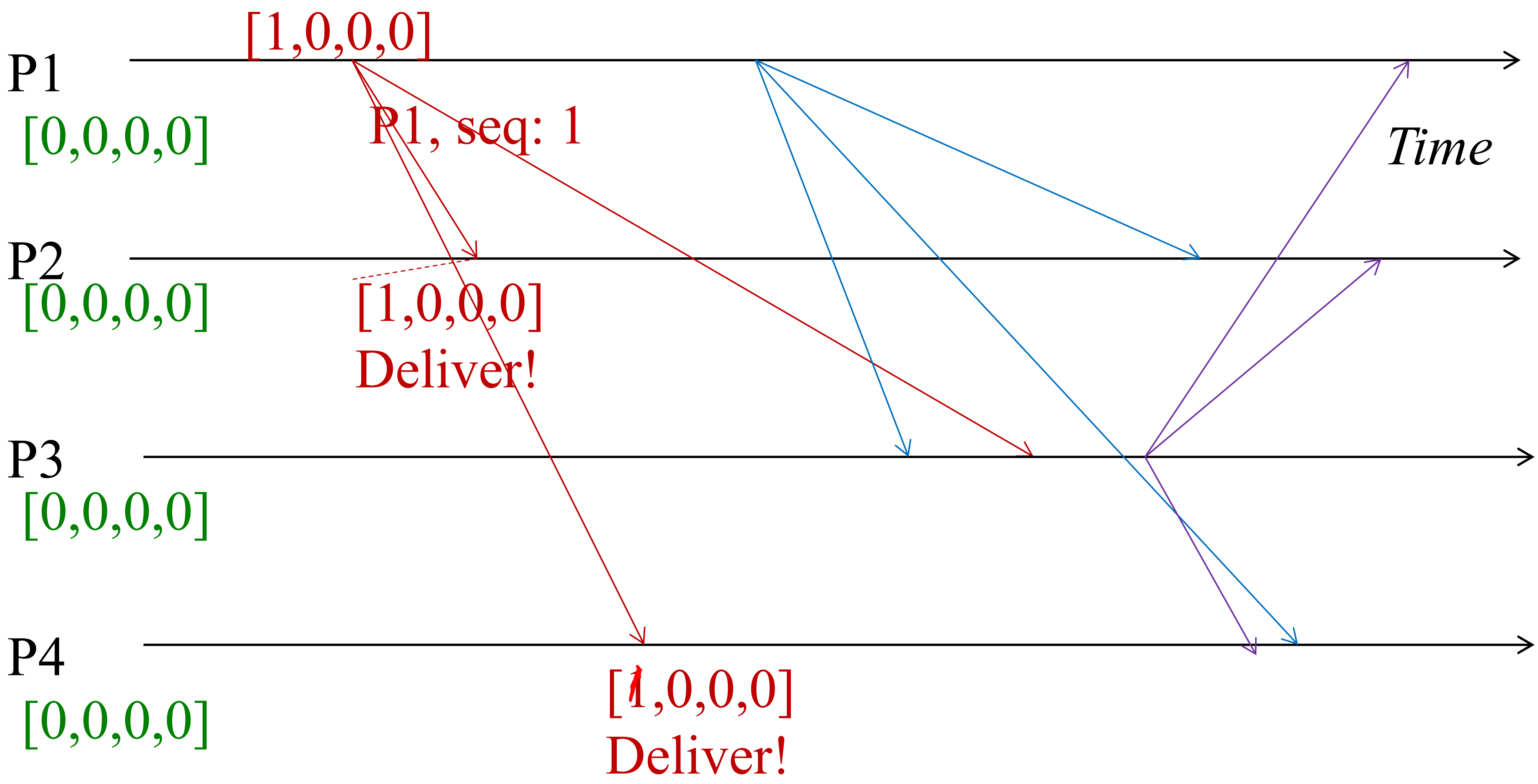
FIFO order multicast execution

FIFO order multicast execution

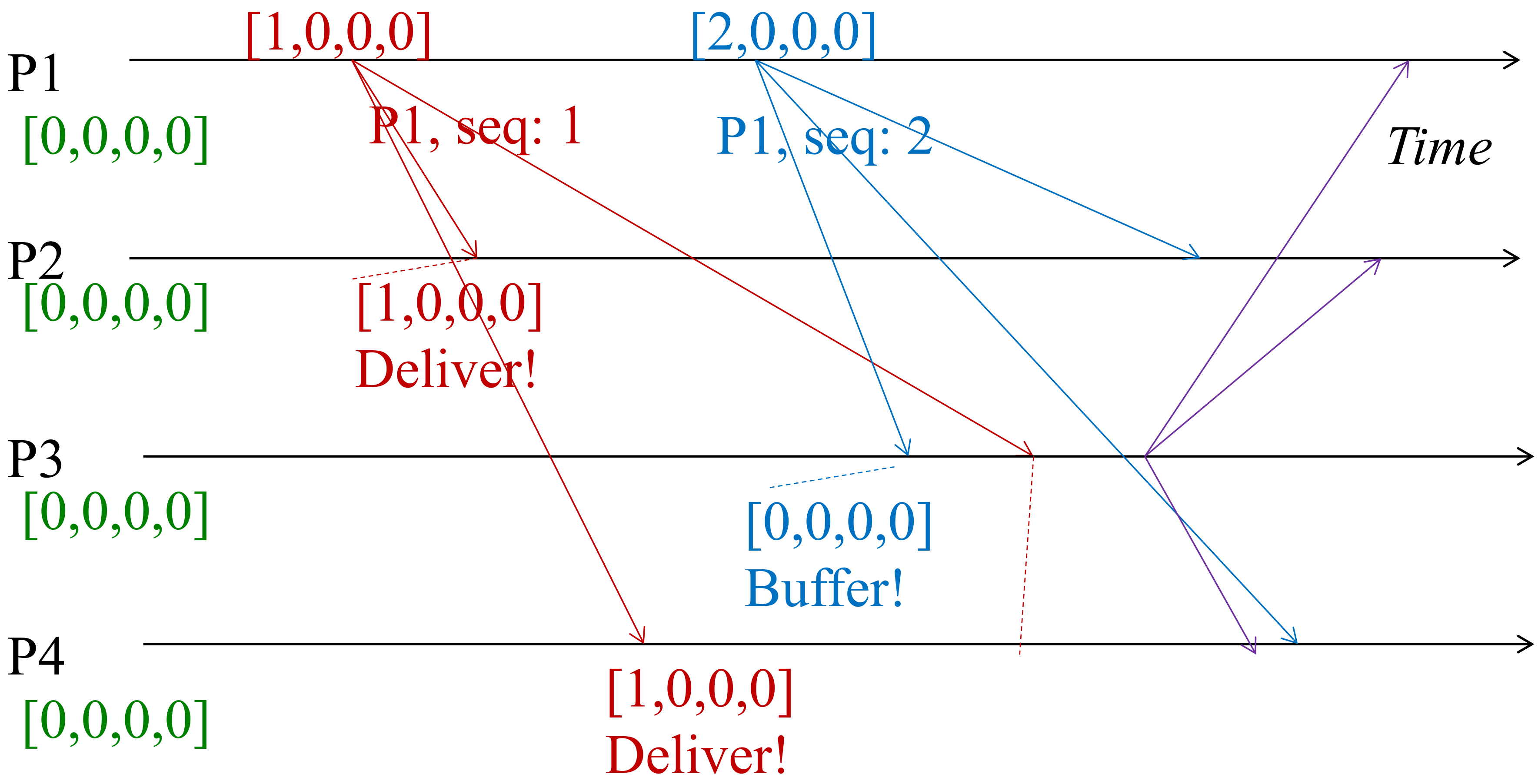


Self-deliveries omitted for simplicity.

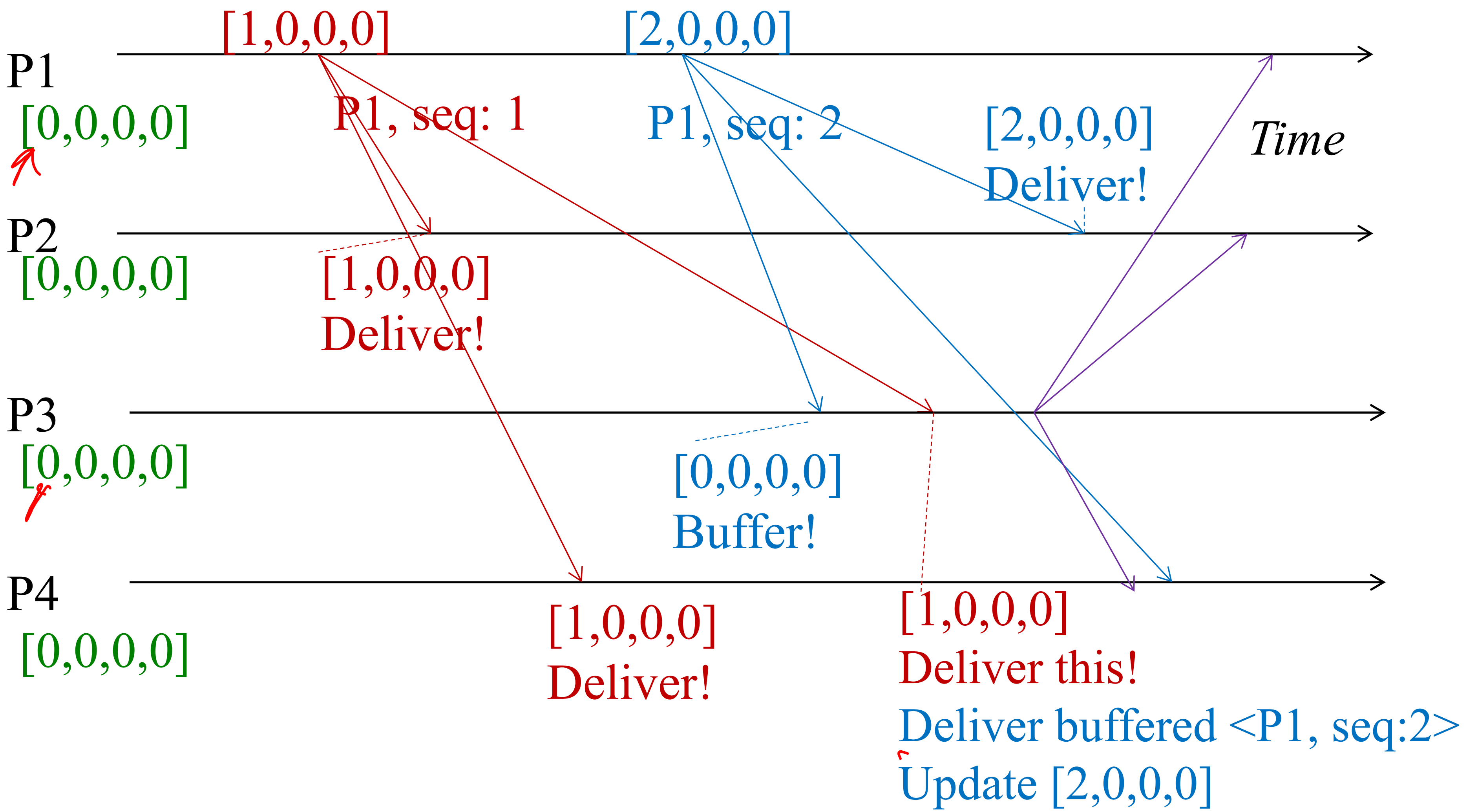
FIFO order multicast execution



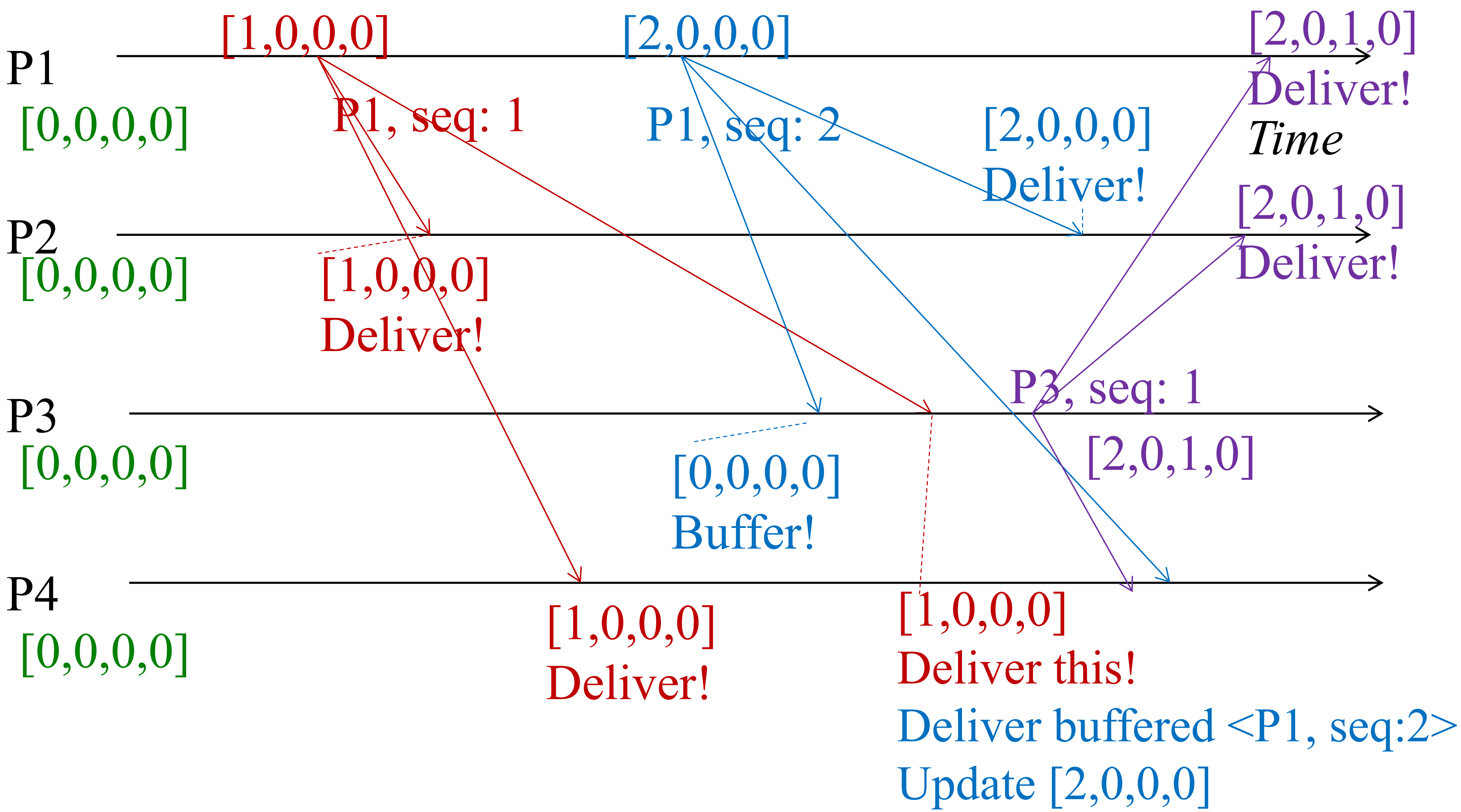
FIFO order multicast execution



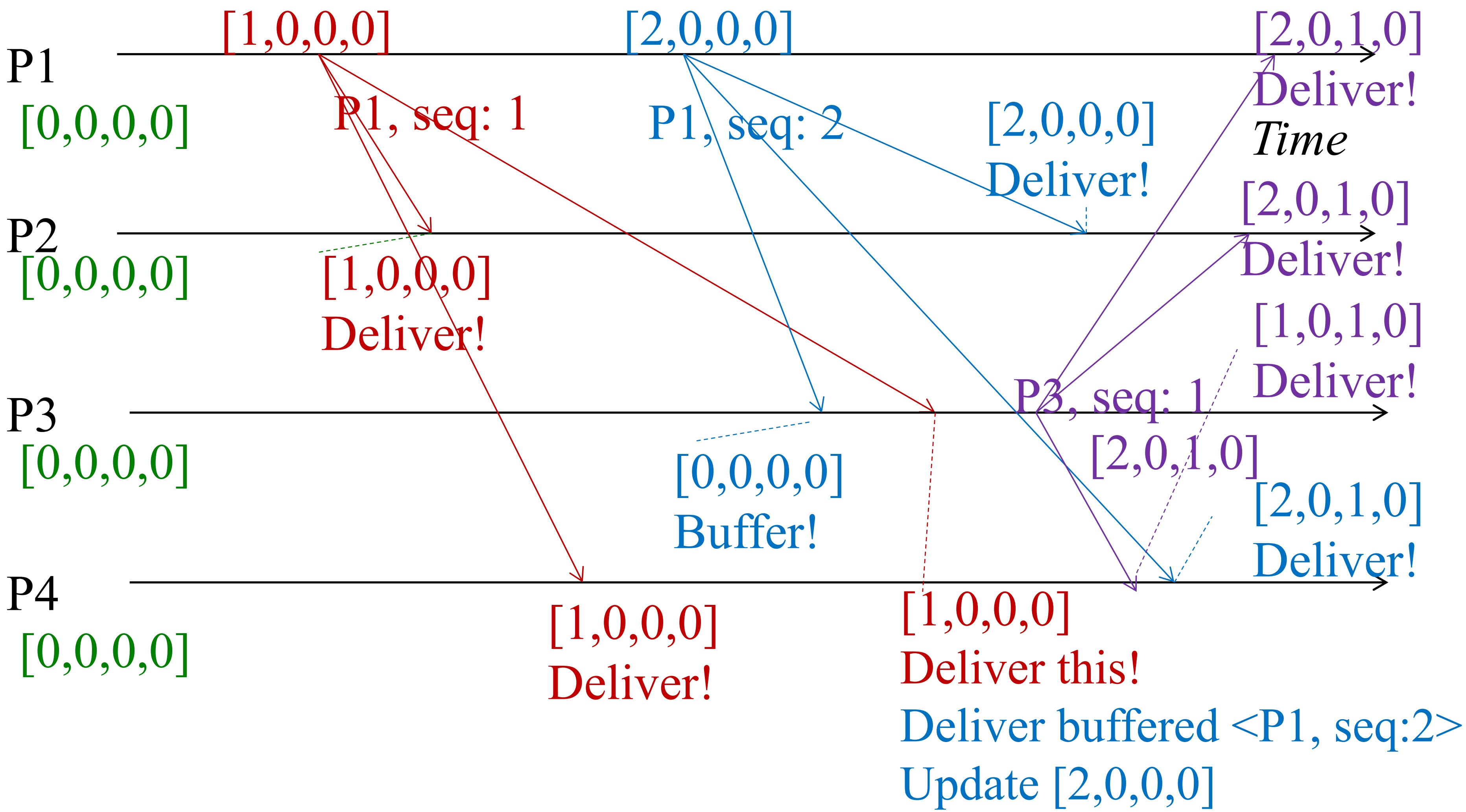
FIFO order multicast execution



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

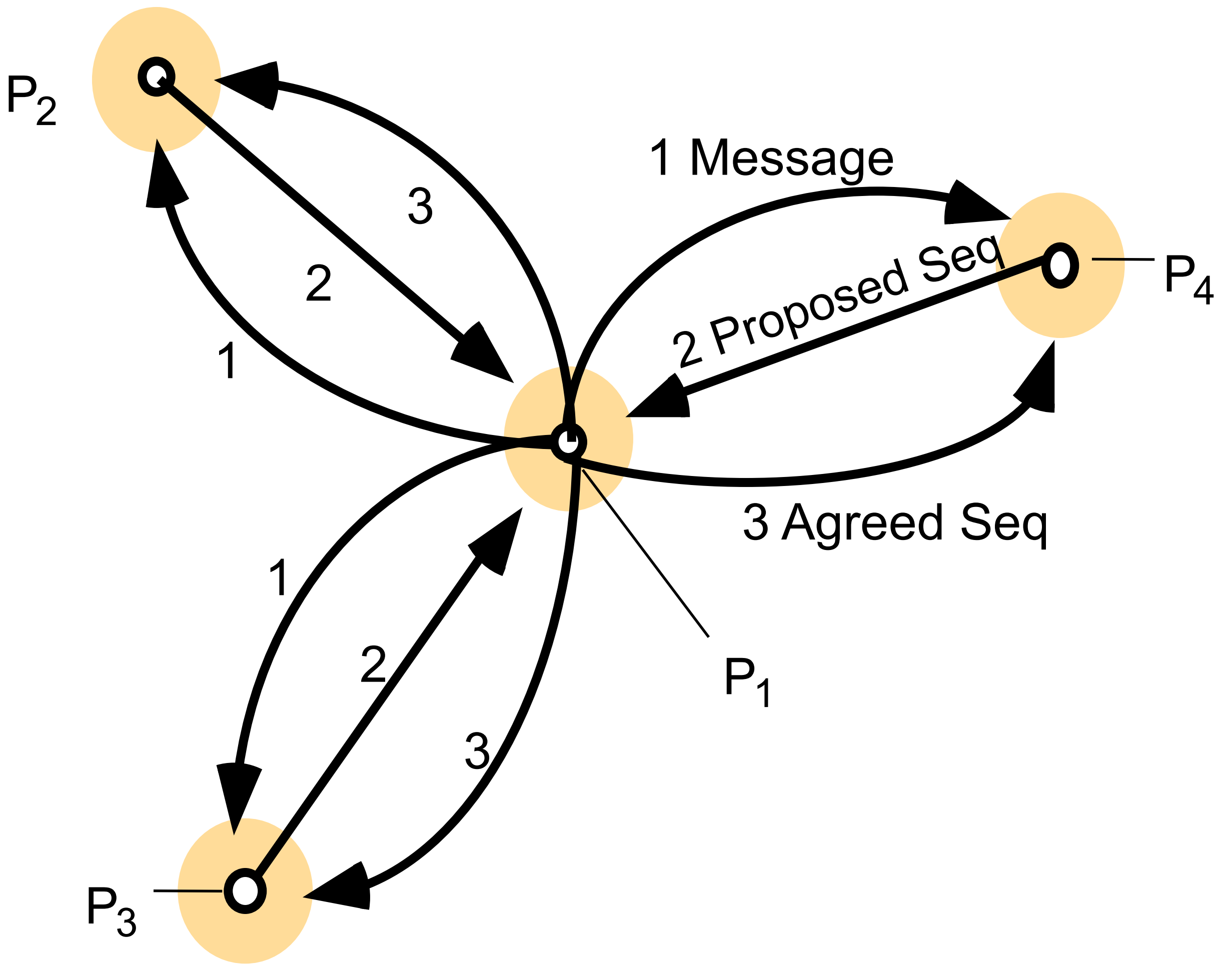
- Special process elected as leader or sequencer.
- TO-multicast(g,m) at P*i*:
  - B-multicast message m to group g *and the sequencer*
- Sequencer:
  - Maintains a global sequence number S (initially 0)
  - When a multicast message m is B-delivered to it:
    - sets S = S + 1, and B-multicast(g,{"order",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

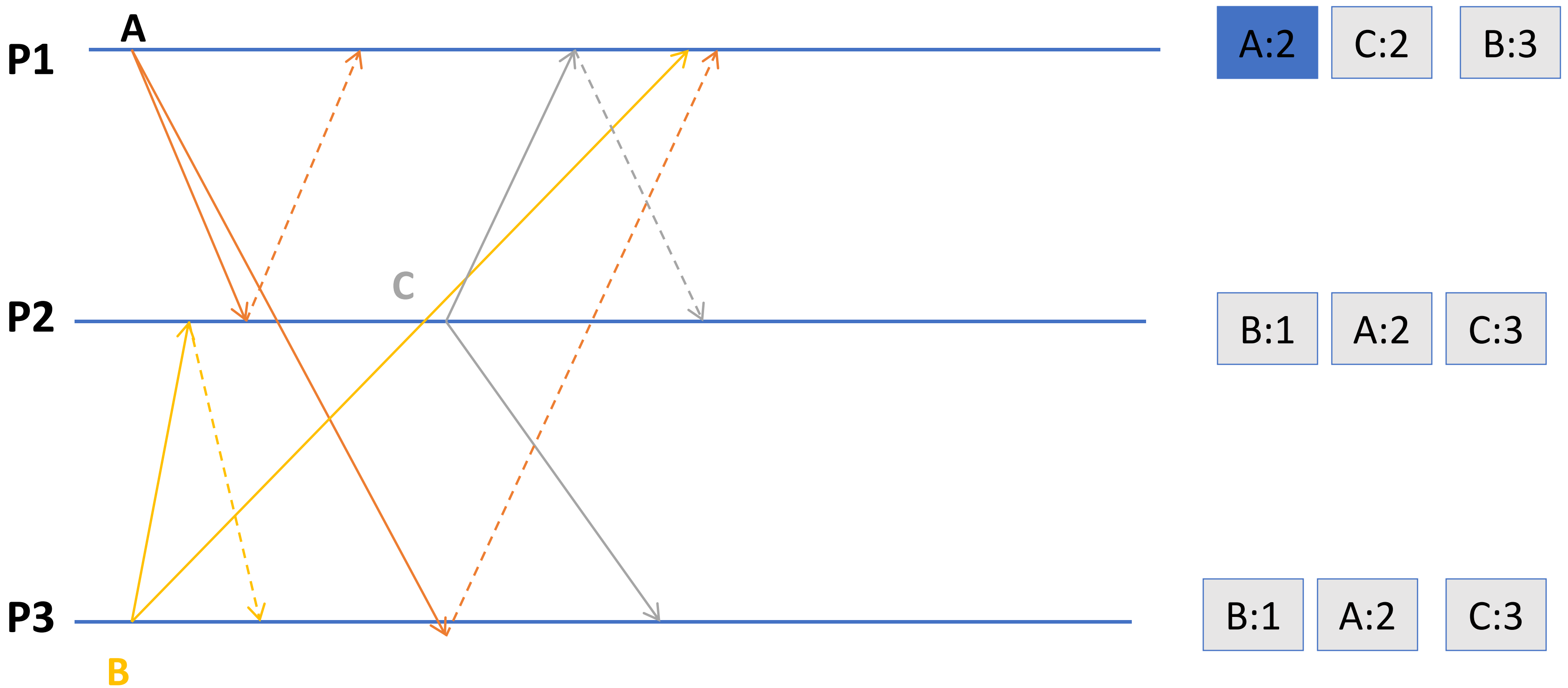
ISIS algorithm for total ordering



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

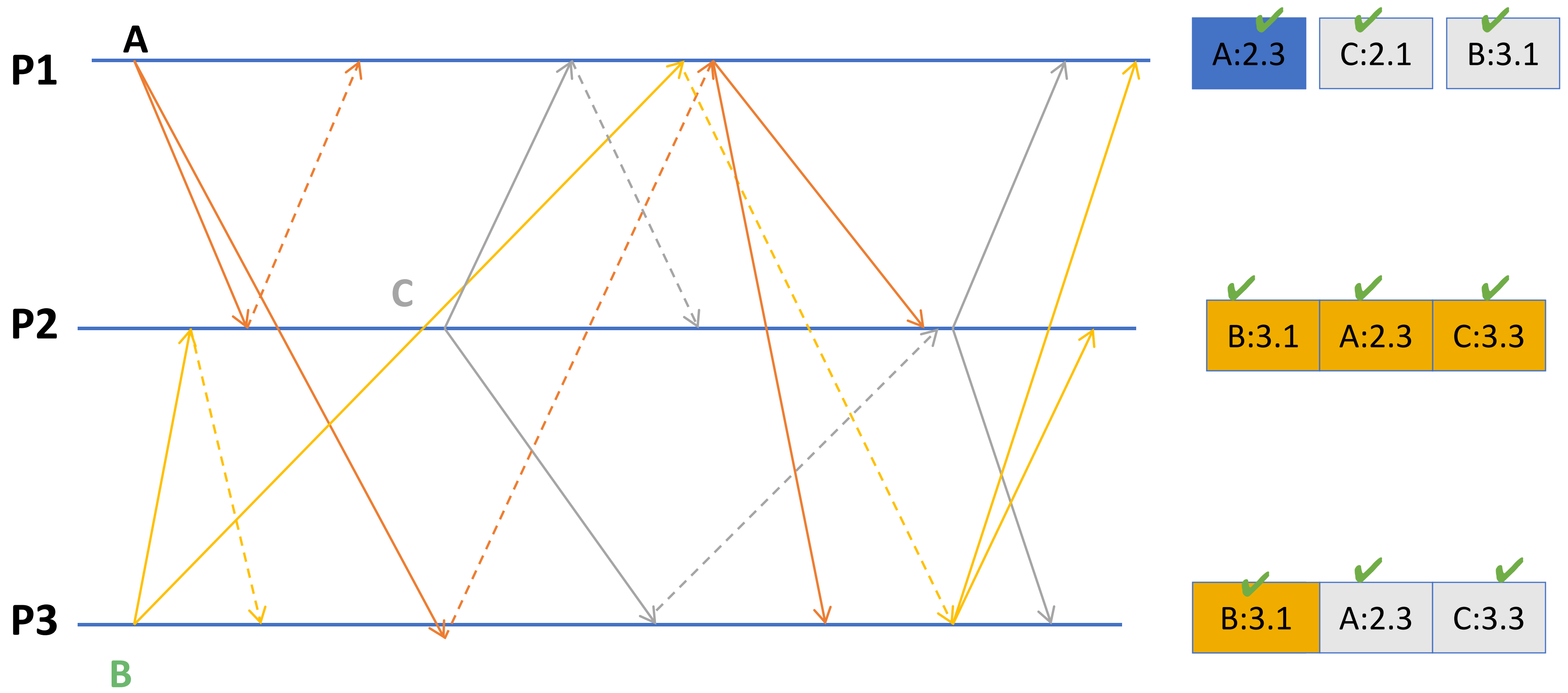


Please refer to lecture recordings/pptx shared over CampusWire for the correct, animated version of this slide.

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



Please refer to lecture recordings/pptx shared over CampusWire for the correct, animated version of this slide.

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!

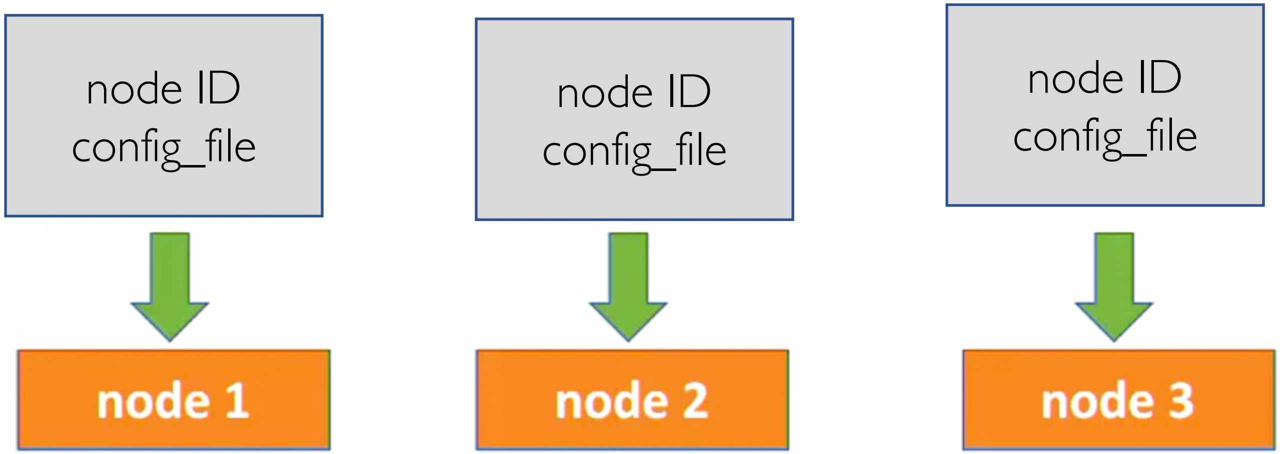
MP1: Event Ordering

- https://courses.grainger.illinois.edu/ece428/sp2023/mps/mp1.html
- Lead TA: Eashan Gupta

- 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.

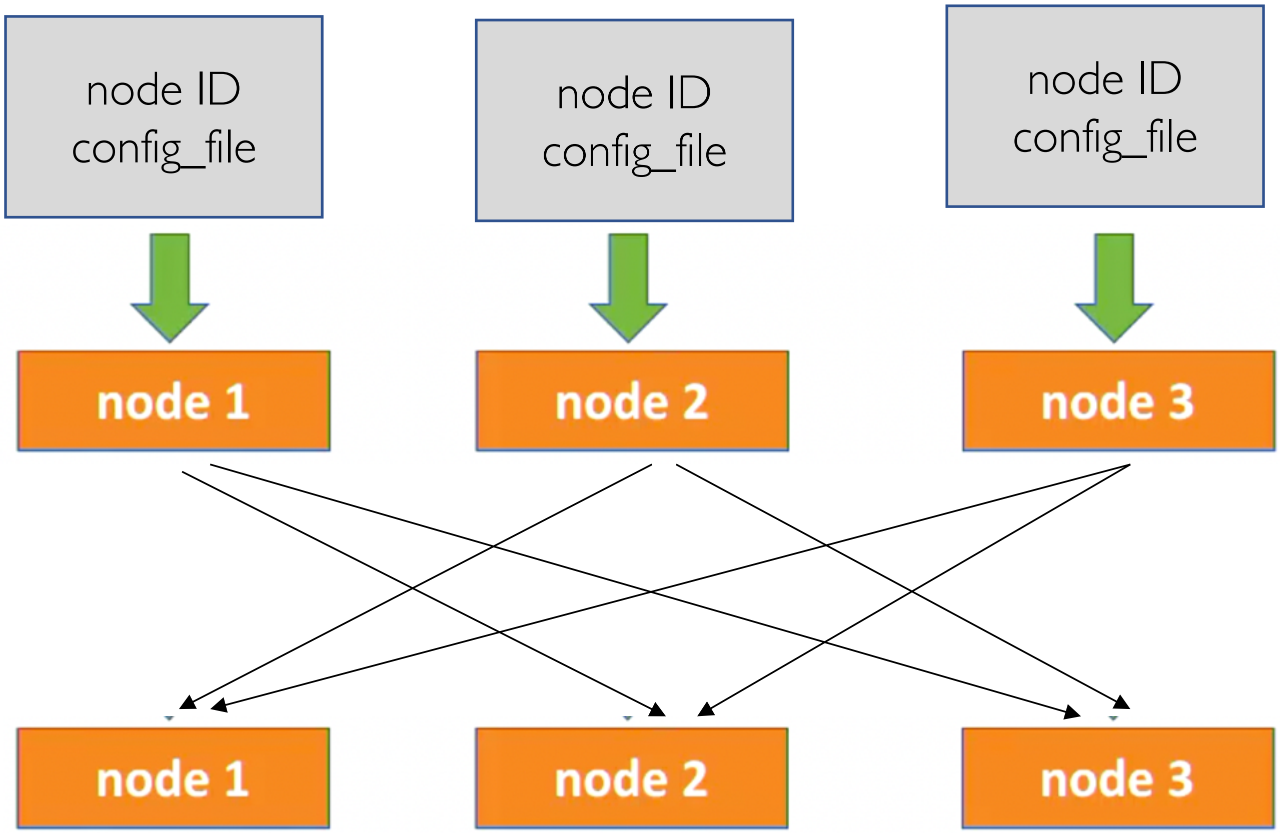
MP1 Architecture Setup



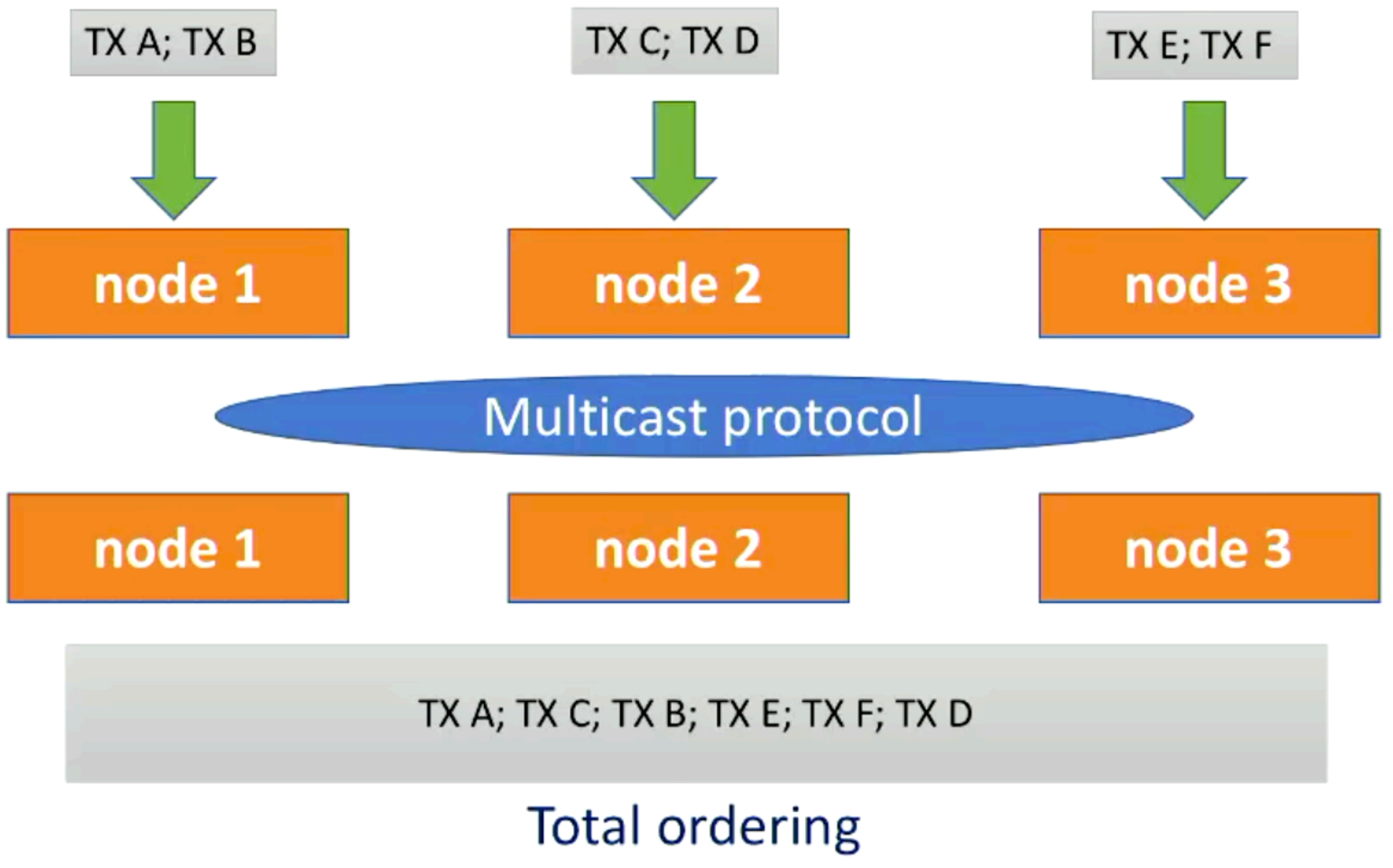
- Example input arguments for first node:
  `./mp1_node node1 config.txt`
- config.txt looks like this:

```
3
node1 sp23-cs425-0101.cs.illinois.edu 1234
node2 sp23-cs425-0102.cs.illinois.edu 1234
node3 sp23-cs425-0103.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 (alive) node.
- Plot the CDF (cumulative distribution function) of the transaction processing time for each evaluation scenario.

MP1: Logistics

- Due on Monday, March 6th.
  - Late policy: Can use part of your 168hours of grace period accounted per student over the entire semester.
- You are allowed to reuse code from MP0.
  - Note: 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!!