Distributed Systems

CS425/ECE428

Feb 10 2022

Instructor: Radhika Mittal

Acknowledgements for some of materials: Indy Gupta and Nikita Borisov

Logistics

- HW2 has been released.
  - Due on Feb 23, 11:59pm.

- MP1 has been released.
  - Due on March 3rd, 11:59pm.

- If you have unavoidable conflict for midterm, fill out the form on Campuswire.
  - Use the "additional comments" section of the form to inform us of any special accommodations you might need.
  - Please let us know of your conflicts/needs by Feb 24th.

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*.

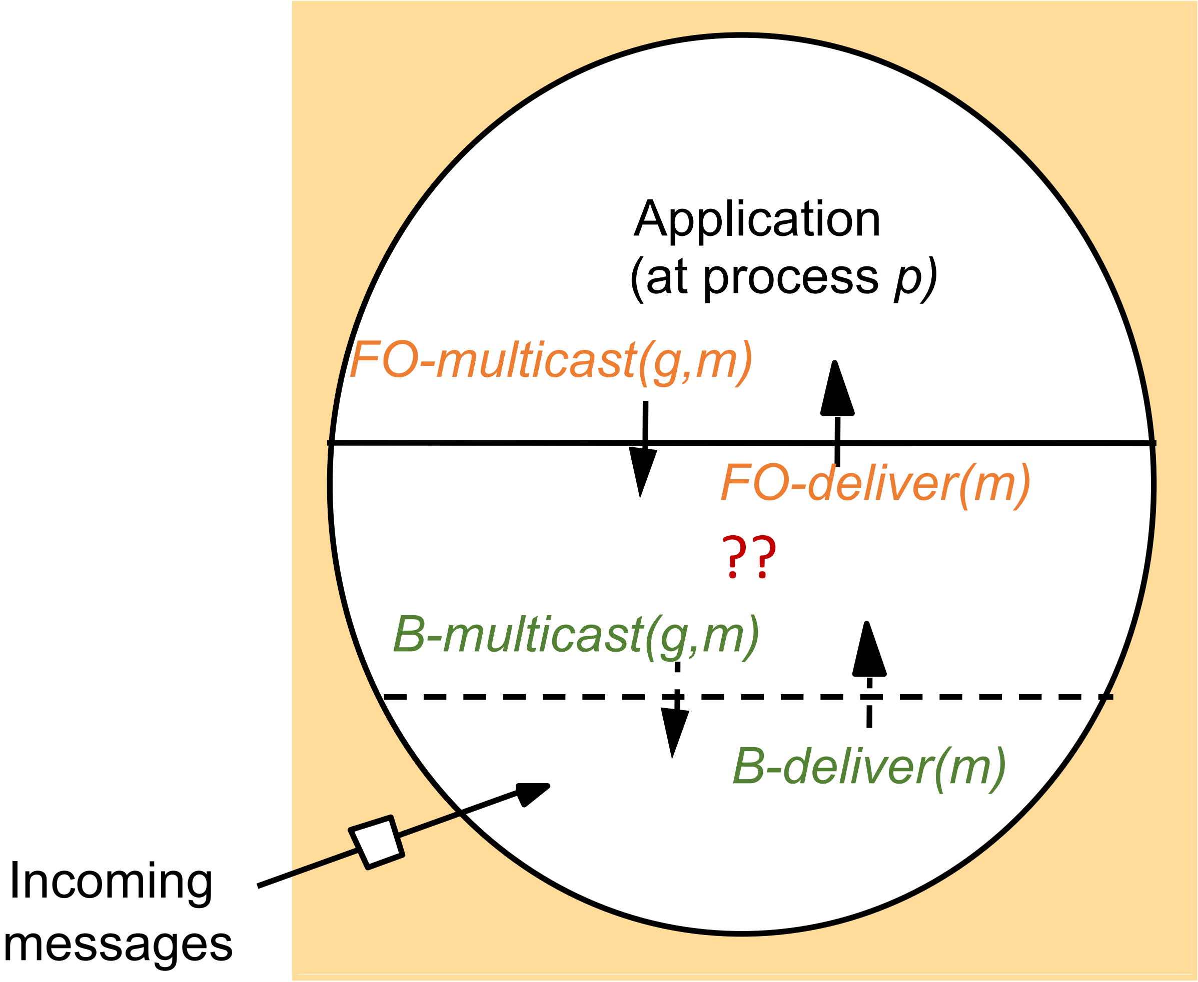
Next Question

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



Application
(at process *p*)

FO-multicast(g,m)

FO-deliver(m)

??

B-multicast(g,m)

B-deliver(m)

Incoming
messages

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

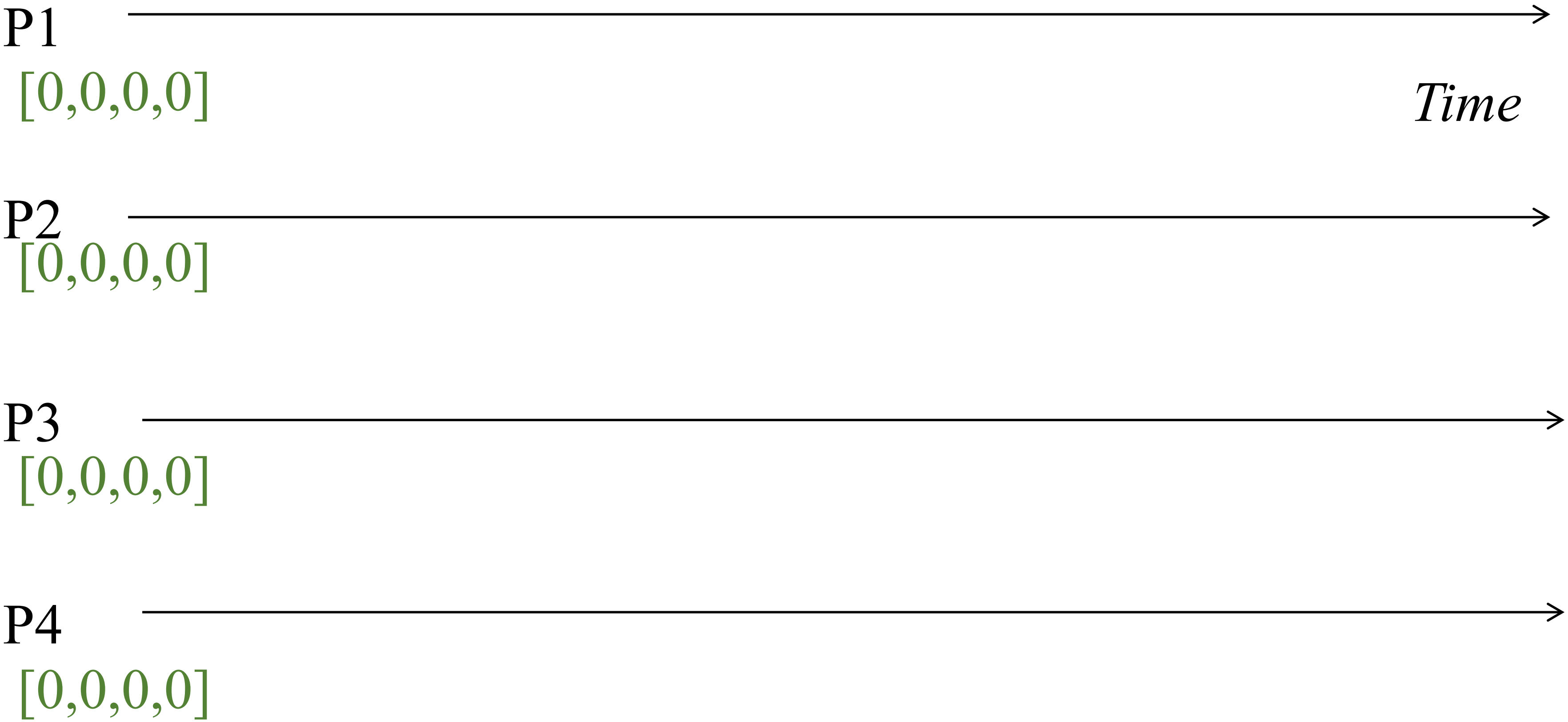
- On FO-multicast(g,m) at process P*j*:

  ```
  set Pj[j] = Pj[j] + 1
  piggyback Pj[j] with m as its sequence number.
  B-multicast(g,{m, Pj[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 == Pi[j] + 1) then
      FO-deliver(m) to application
      set Pi[j] = Pi[j] + 1
  else buffer this multicast until above condition is true
  ```

FIFO order multicast execution



P1

[0,0,0,0]

Time

P2

[0,0,0,0]

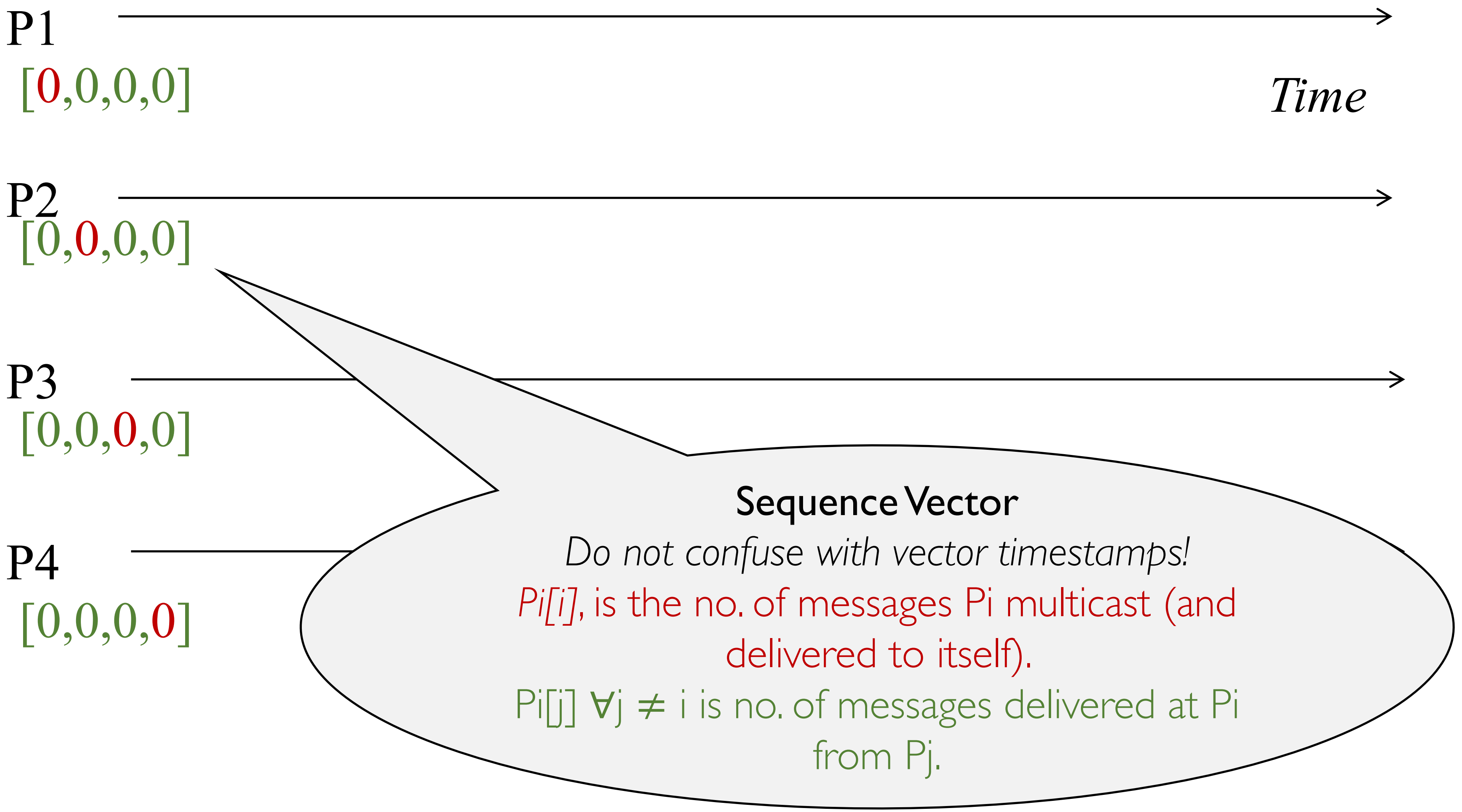
P3

[0,0,0,0]

P4

[0,0,0,0]

FIFO order multicast execution



P1
[0,0,0,0]

Time

P2
[0,0,0,0]

P3
[0,0,0,0]

P4
[0,0,0,0]

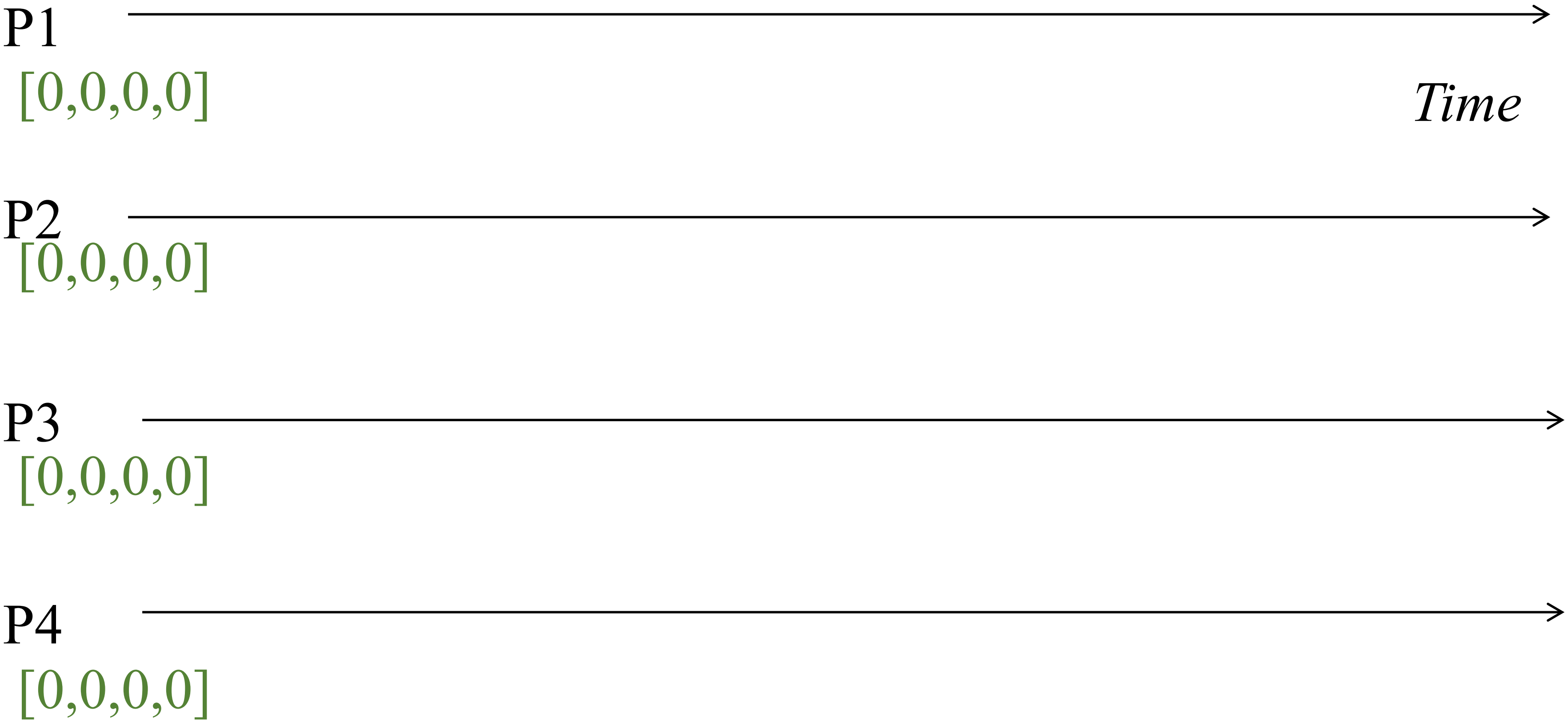
Sequence Vector

Do not confuse with vector timestamps!

Pi[i], is the no. of messages Pi multicast (and delivered to itself).

Pi[j] ∀j ≠ i is no. of messages delivered at Pi from Pj.

FIFO order multicast execution



P1

[0,0,0,0]

Time

P2

[0,0,0,0]

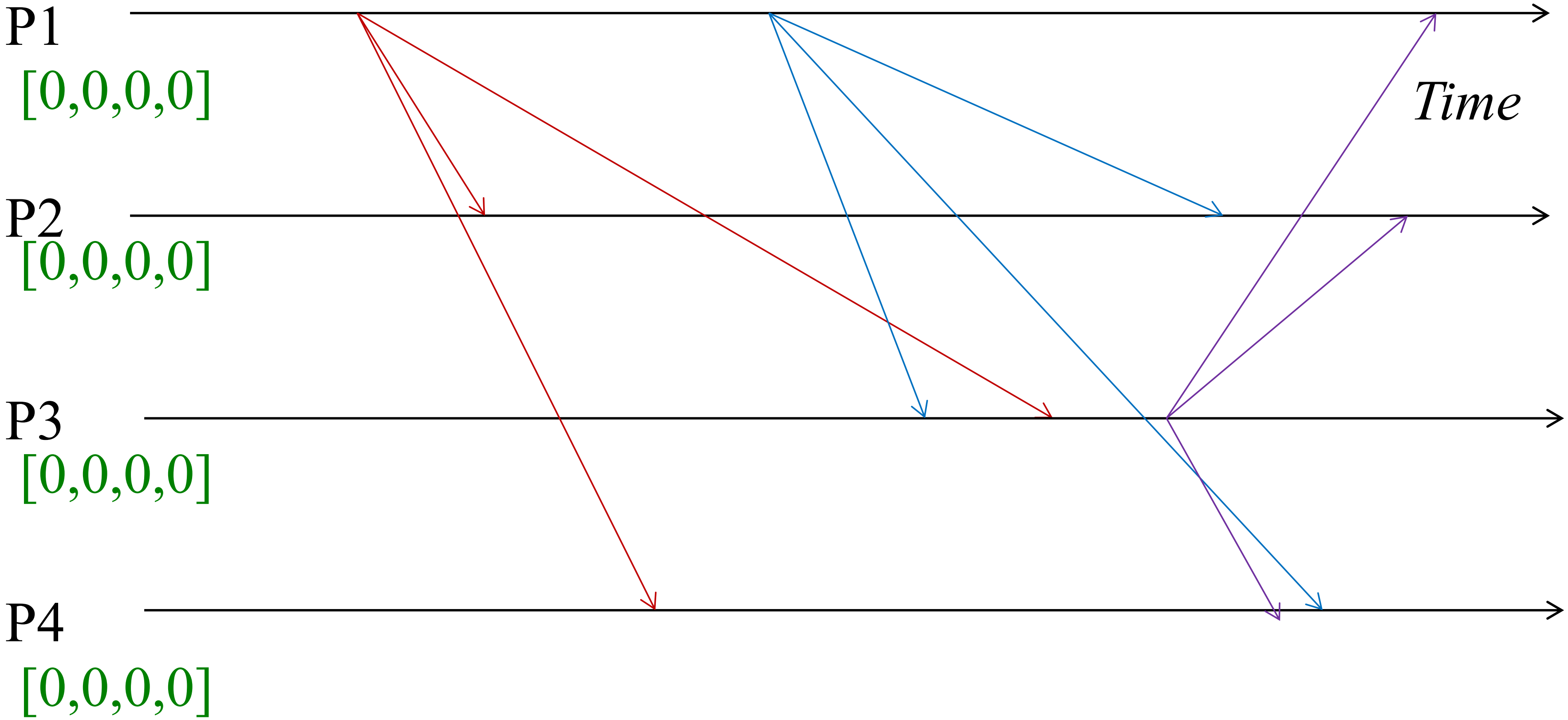
P3

[0,0,0,0]

P4

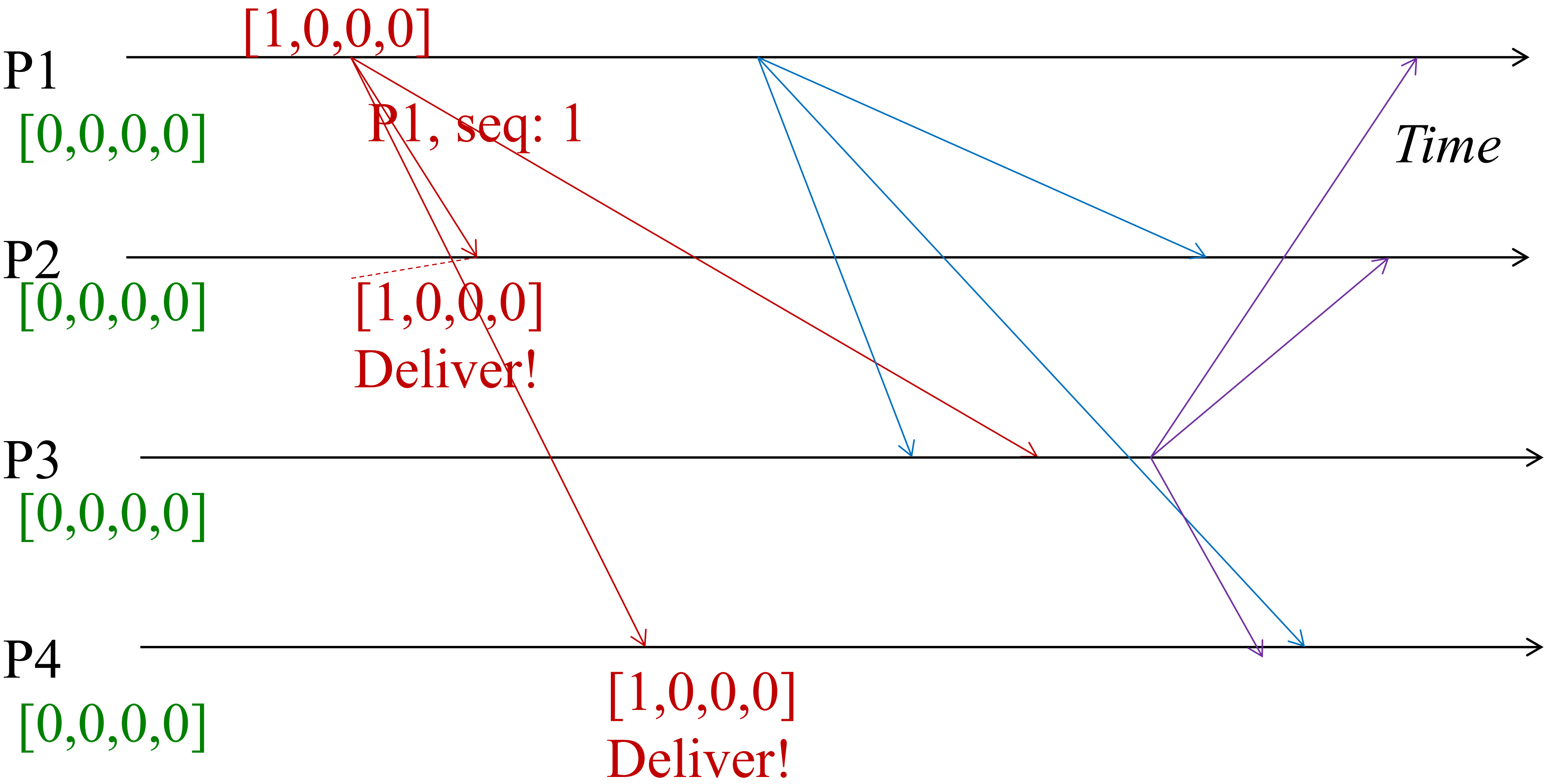
[0,0,0,0]

FIFO order multicast execution



Self-deliveries omitted for simplicity.

FIFO order multicast execution



[1,0,0,0]

P1

[0,0,0,0]

P1, seq: 1

Time

P2

[0,0,0,0]

[1,0,0,0]

Deliver!

P3

[0,0,0,0]

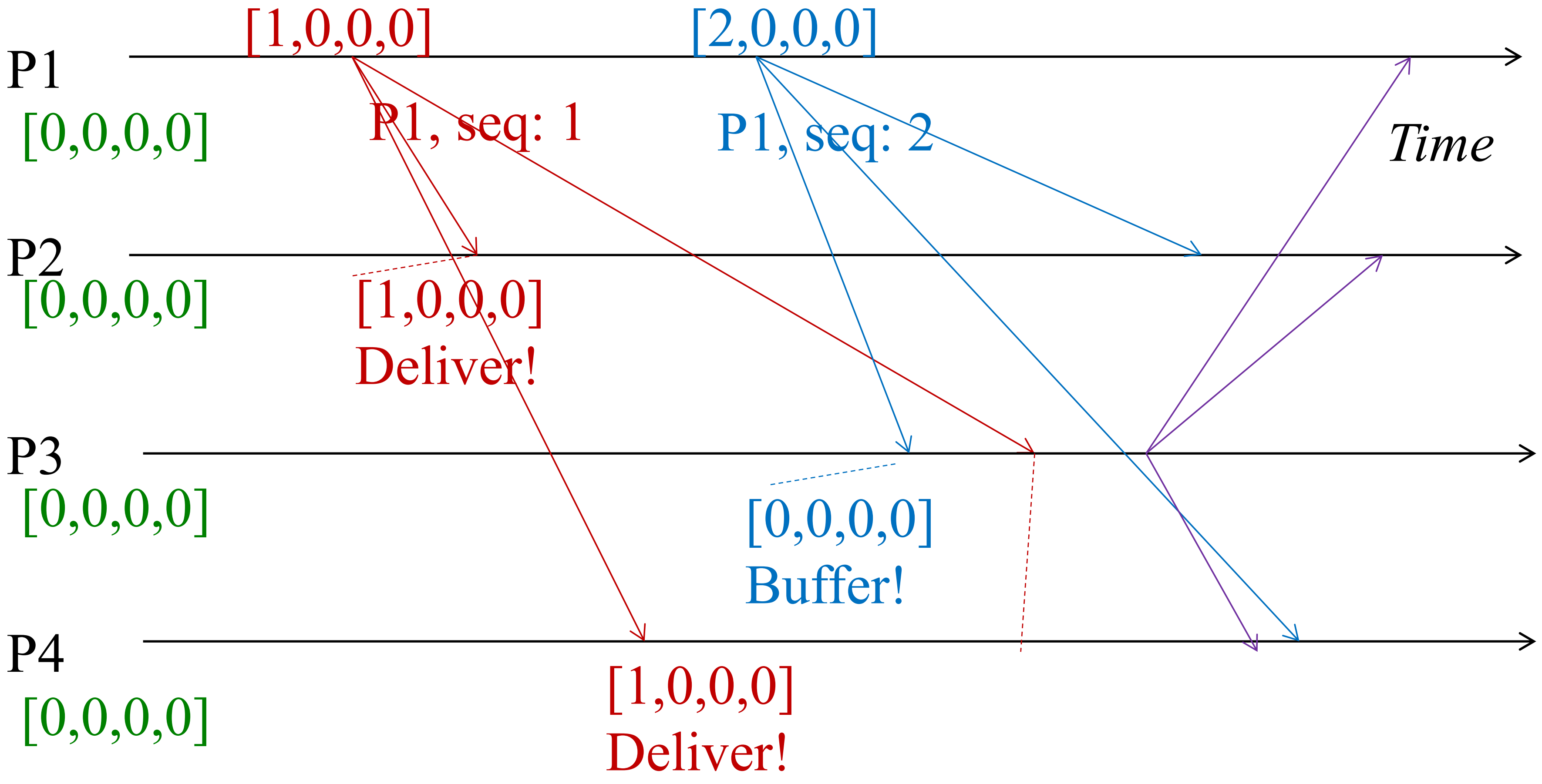
P4

[0,0,0,0]

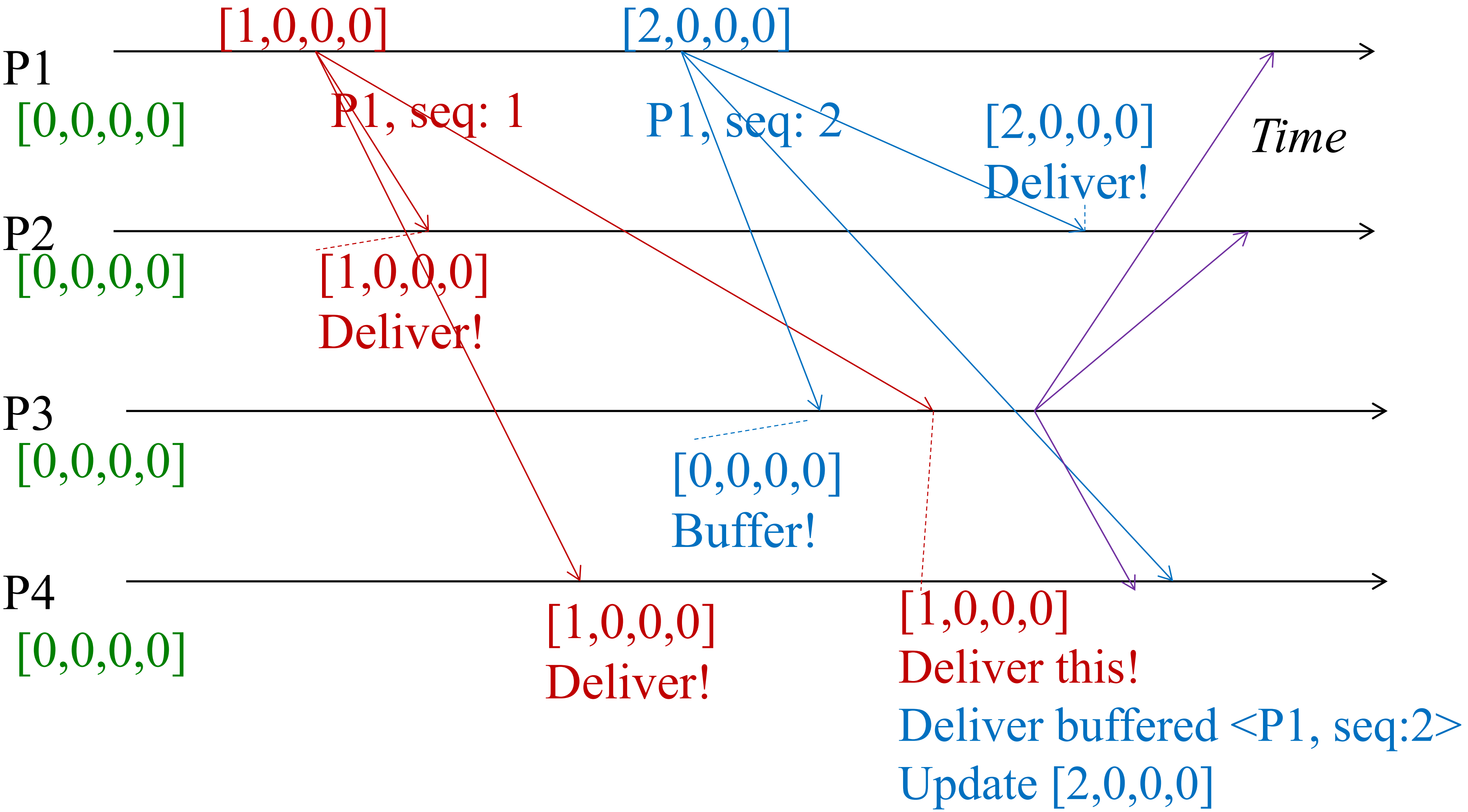
[1,0,0,0]

Deliver!

FIFO order multicast execution



FIFO order multicast execution



P1
[0,0,0,0]

P2
[0,0,0,0]

P3
[0,0,0,0]

P4
[0,0,0,0]

[1,0,0,0]

P1, seq: 1

[2,0,0,0]

P1, seq: 2

Time

[1,0,0,0]
Deliver!

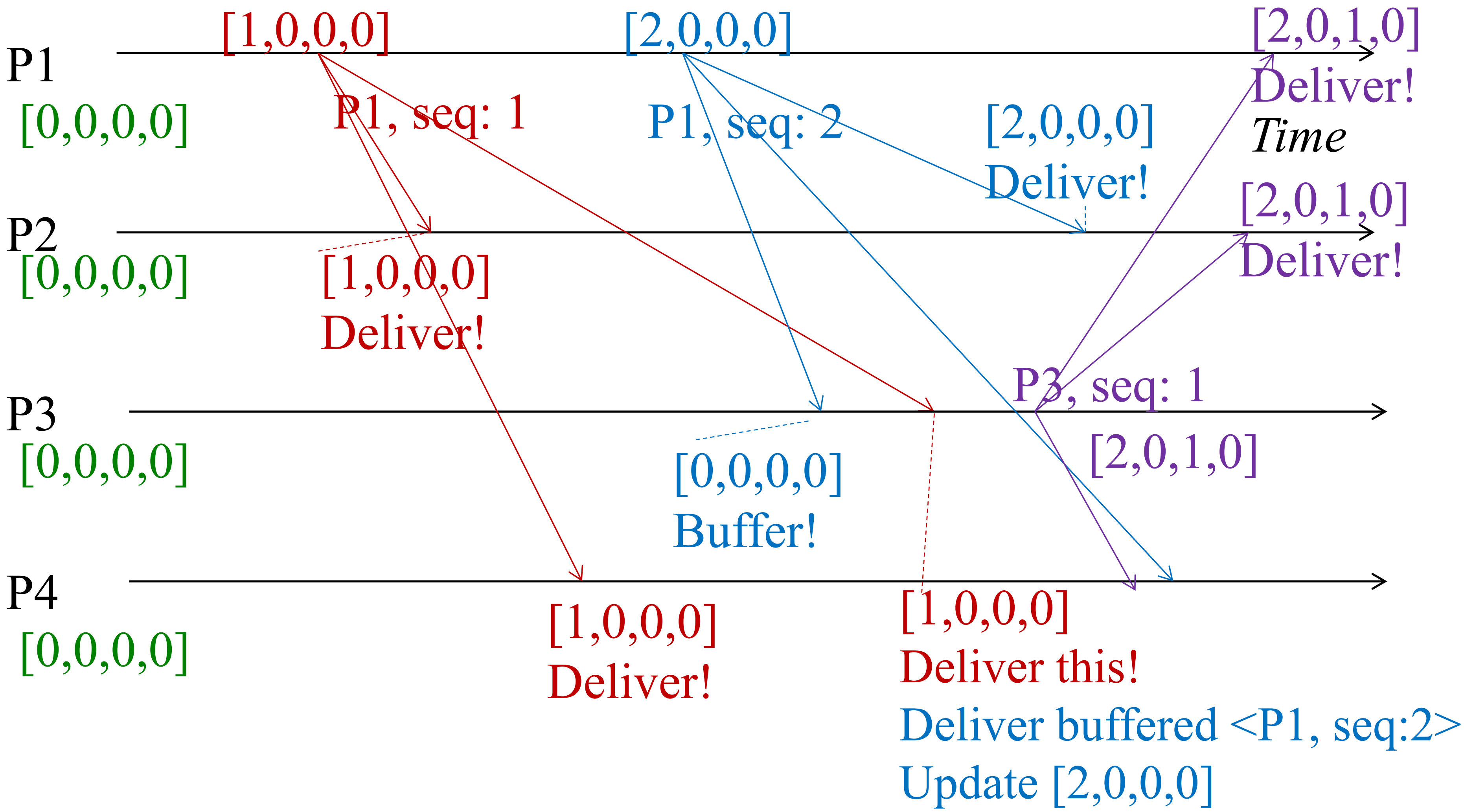
[2,0,0,0]
Deliver!

[0,0,0,0]
Buffer!

[1,0,0,0]
Deliver!

[1,0,0,0]
Deliver this!
Deliver buffered <P1, seq:2>
Update [2,0,0,0]

FIFO order multicast execution



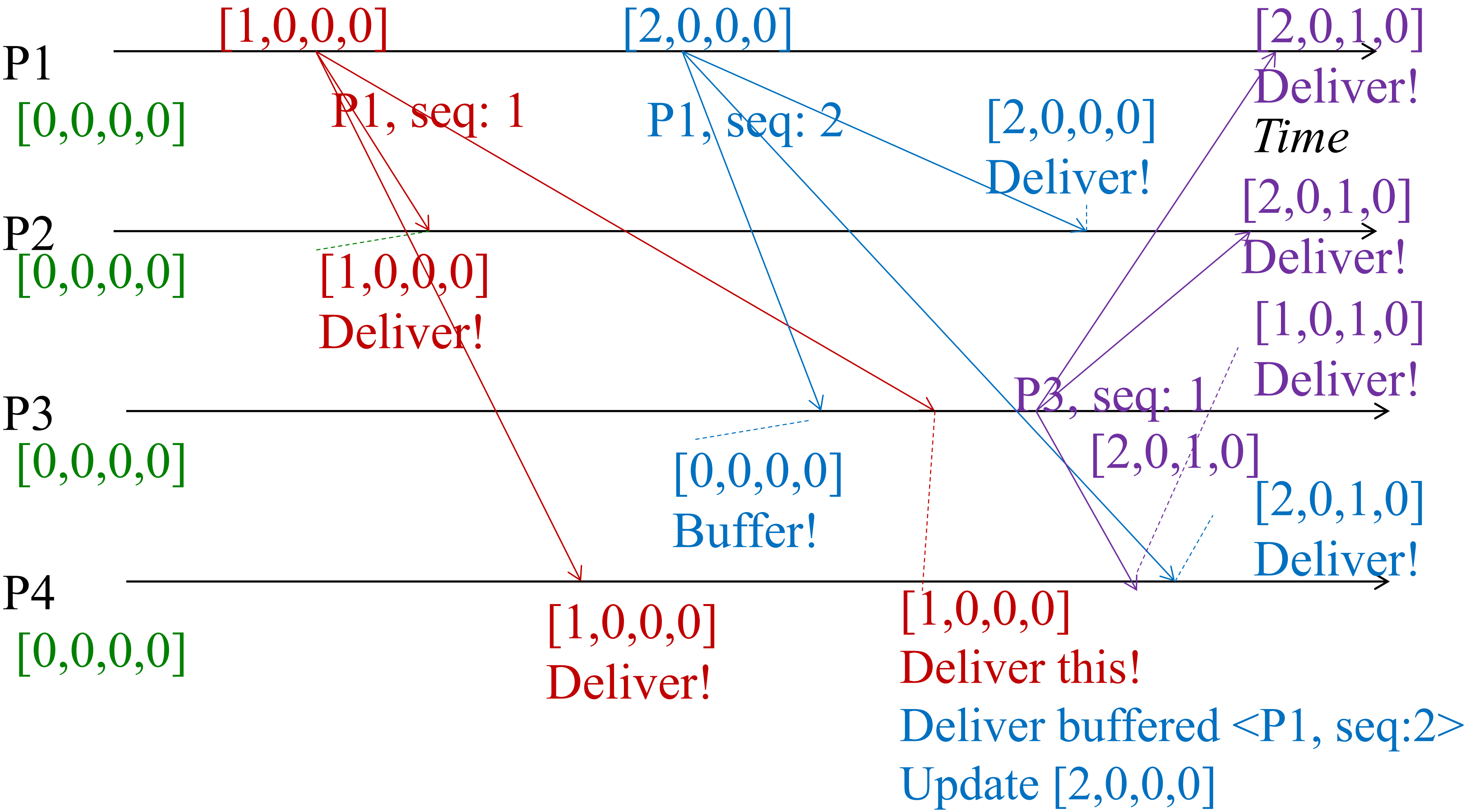
P1
[0,0,0,0]
[1,0,0,0]
P1, seq: 1
[2,0,0,0]
P1, seq: 2
[2,0,1,0]
Deliver!
Time

P2
[0,0,0,0]
[1,0,0,0]
Deliver!
[2,0,0,0]
Deliver!
[2,0,1,0]
Deliver!

P3
[0,0,0,0]
[0,0,0,0]
Buffer!
P3, seq: 1
[2,0,1,0]

P4
[0,0,0,0]
[1,0,0,0]
Deliver!
[1,0,0,0]
Deliver this!
Deliver buffered <P1, seq:2>
Update [2,0,0,0]

FIFO order multicast execution



P1
[0,0,0,0]

P2
[0,0,0,0]

P3
[0,0,0,0]

P4
[0,0,0,0]

[1,0,0,0]

P1, seq: 1

[1,0,0,0]
Deliver!

[1,0,0,0]
Deliver!

[2,0,0,0]

P1, seq: 2

[0,0,0,0]
Buffer!

[2,0,0,0]
Deliver!

[1,0,0,0]
Deliver this!
Deliver buffered <P1, seq:2>
Update [2,0,0,0]

P3, seq: 1

[2,0,1,0]

[2,0,1,0]
Deliver!

Time

[2,0,1,0]
Deliver!

[1,0,1,0]
Deliver!

[2,0,1,0]
Deliver!

Implementing FIFO order multicast

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

```
set Pj[j] = Pj[j] + 1
piggyback Pj[j] with m as its sequence number.
B-multicast(g, {m, Pj[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 == Pi[j] + 1) then
    FO-deliver(m) to application
    set Pi[j] = Pi[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 Pj[j] = Pj[j] + 1
    piggyback Pj[j] with m as its sequence number.
    R-multicast(g,{m, Pj[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 == Pi[j] + 1) then
        FO-deliver(m) to application
        set Pi[j] = Pi[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

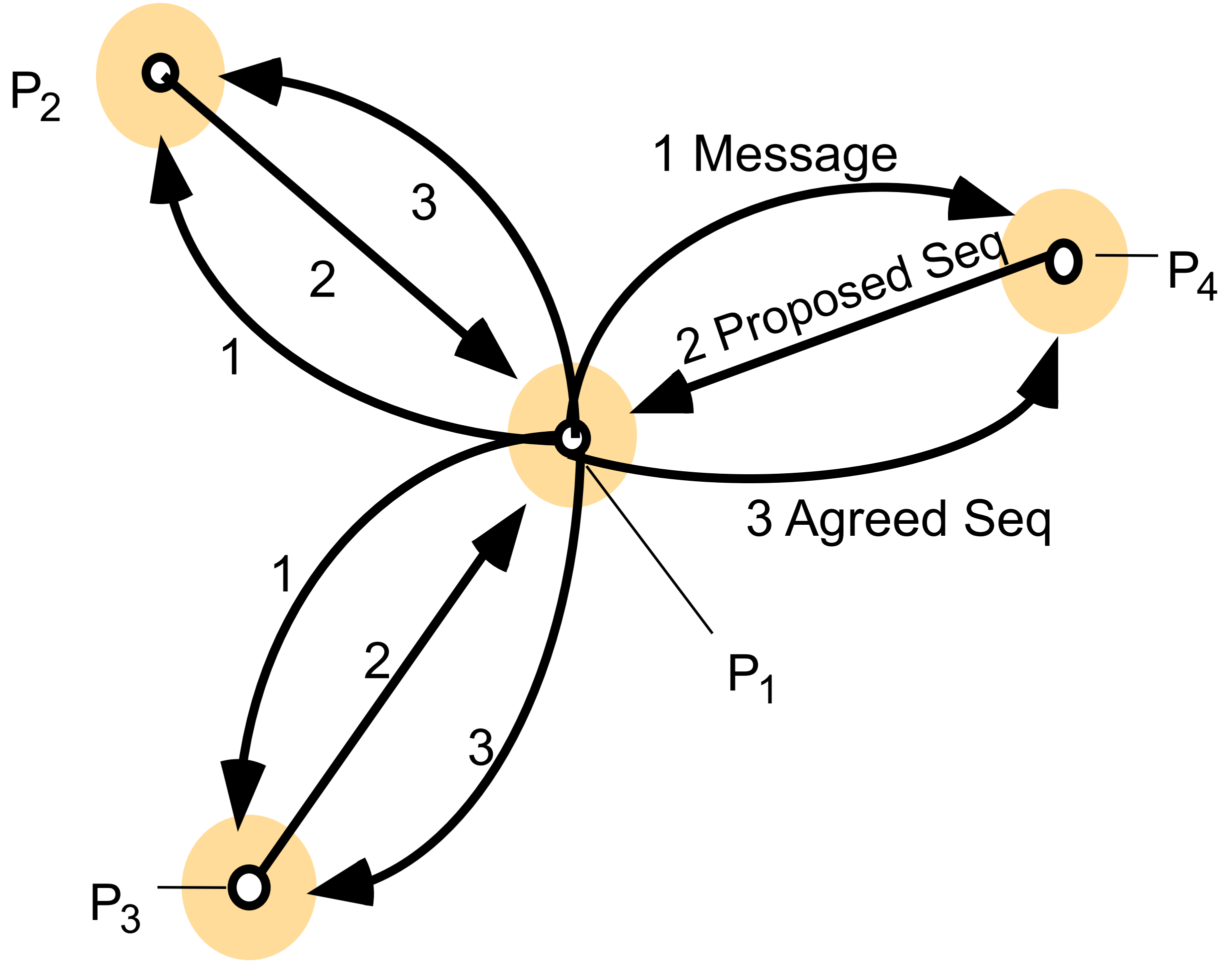
- Special process elected as leader or sequencer.
- TO-multicast(g,m) at P*i*:
  - B-multicast message m to group g *and the sequencer*
- Sequencer:
  - Maintains a global sequence number S (initially 0)
  - When a multicast message m is B-delivered to it:
    - sets S = S + 1, and B-multicast(g,{"order",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

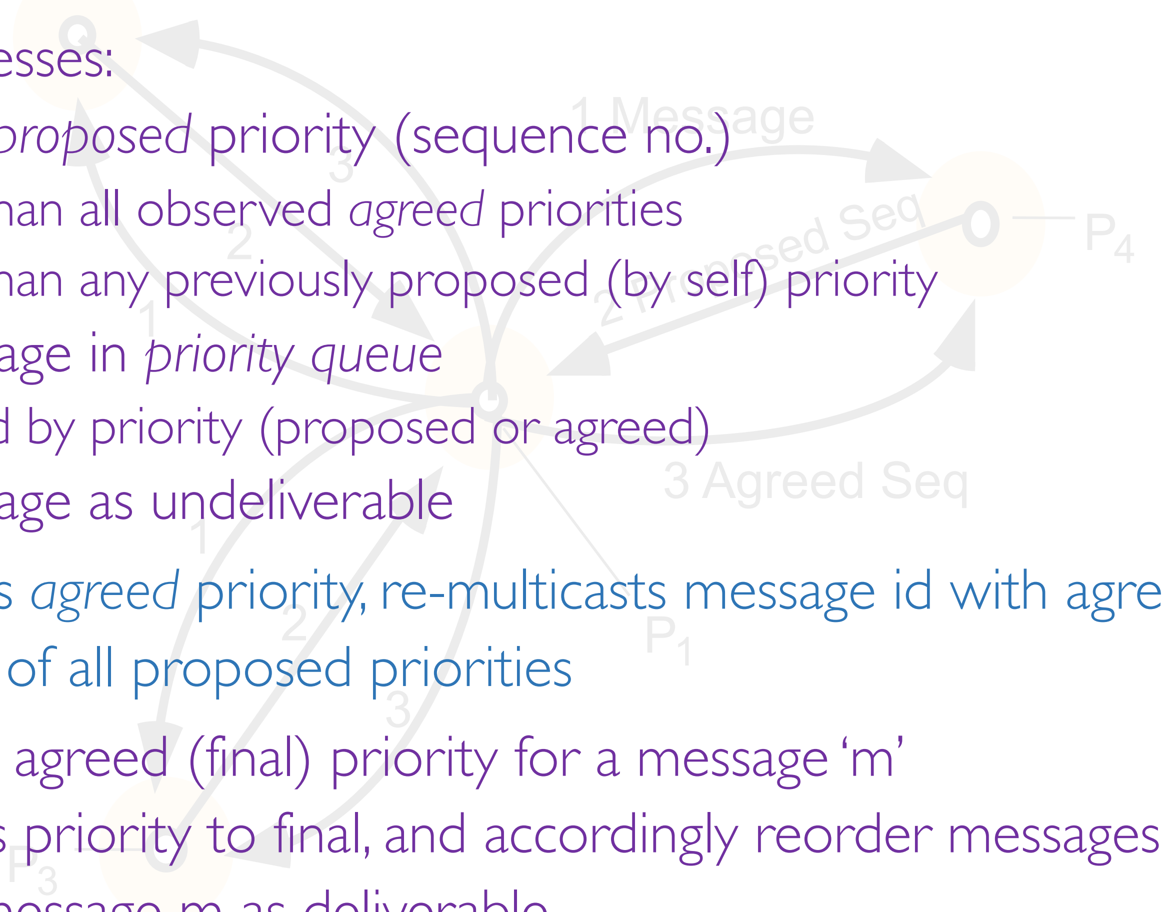
- Basic idea:
  - Same sequence number counter across different processes.
  - Instead of different sequence number counter for each process.

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

ISIS algorithm for total ordering

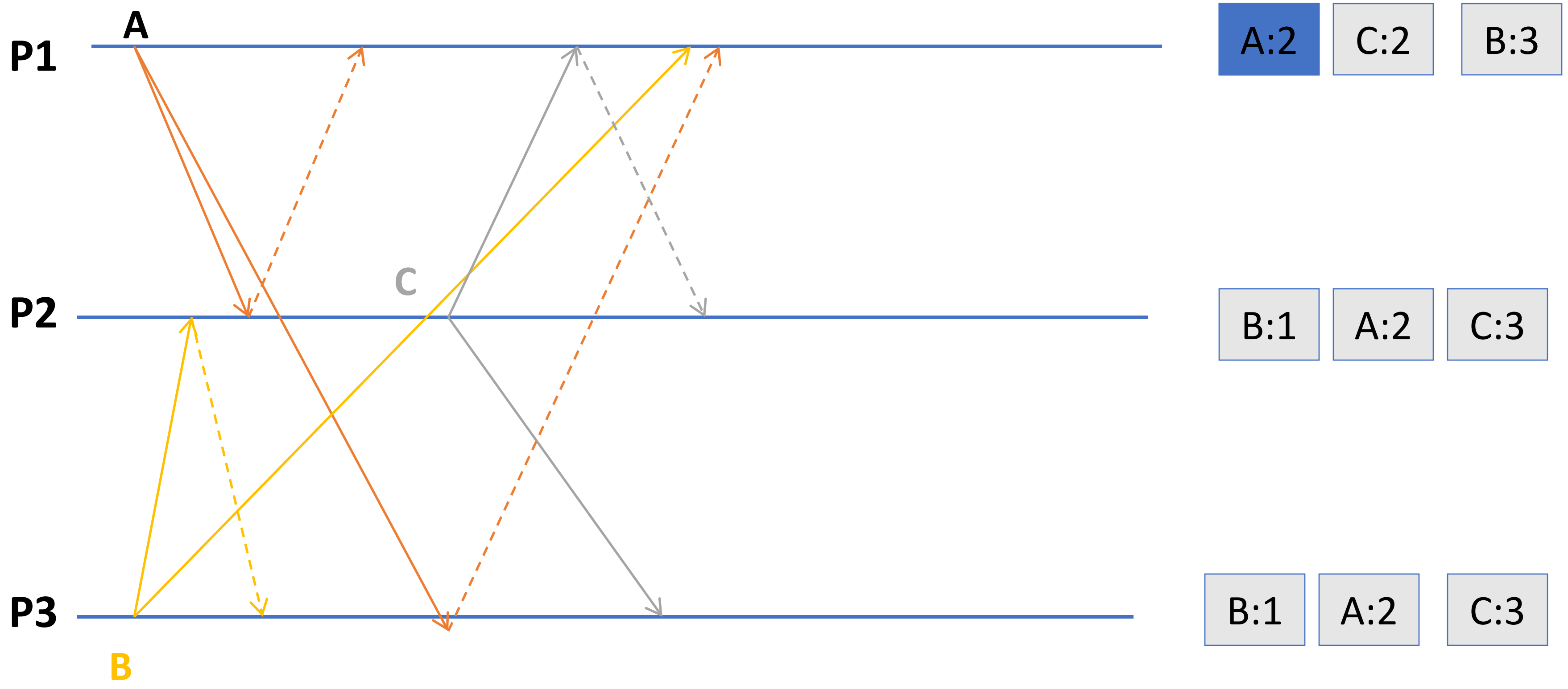


ISIS algorithm for total ordering



- 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 look at lecture recordings for the correct (animated) version of this example.

How do we break ties?

- Problem: priority queue requires unique priorities.

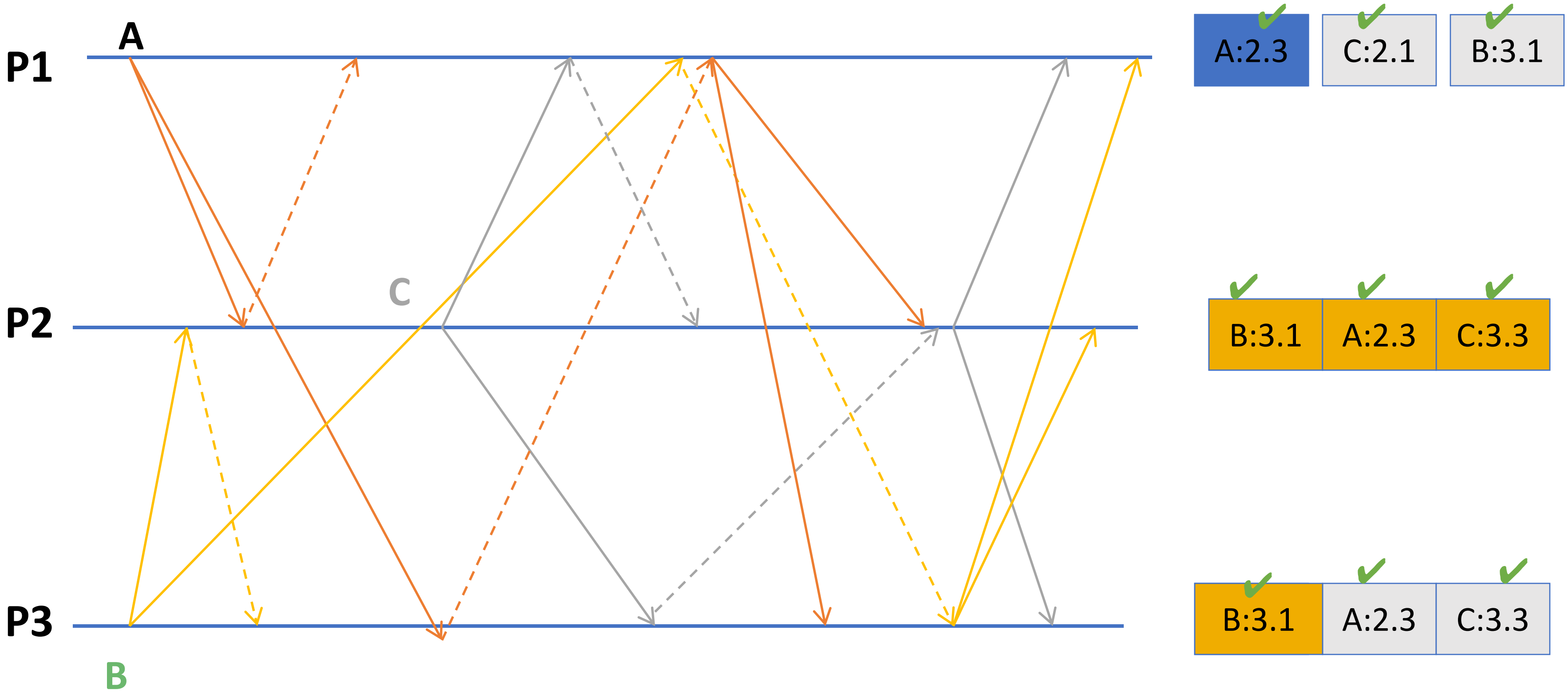
- Solution: add process # to suggested priority.
  - priority.(*id of the process that proposed the priority*)
  - i.e., 3.2 == process 2 proposed priority 3

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

↑ not a decimal point.

(3, 2) > (3, 1 2)

Example: ISIS algorithm



Please look at lecture recordings for the correct (animated) version of this example.

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!

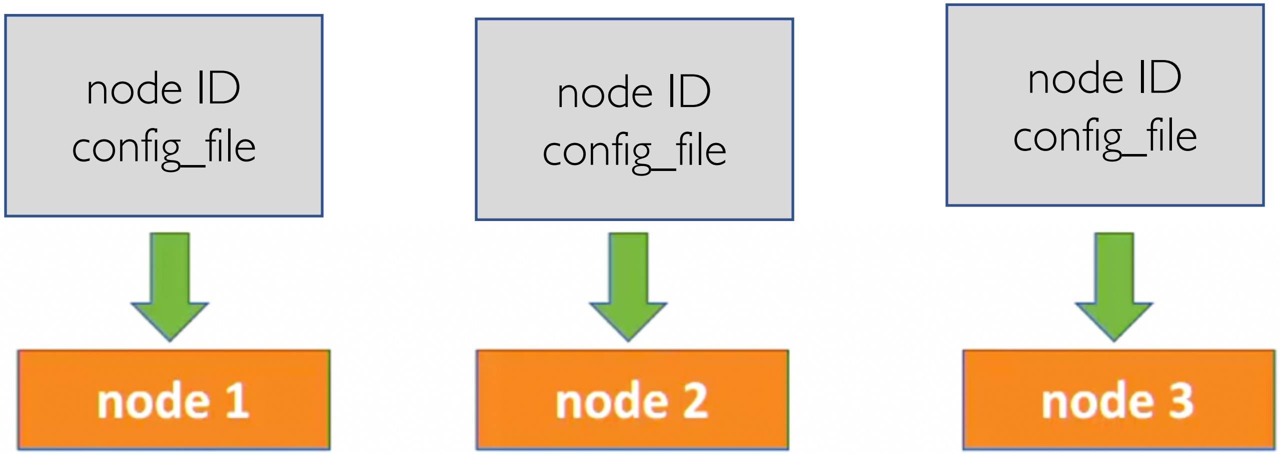
MP1: Event Ordering

- https://courses.grainger.illinois.edu/cs425/sp2022/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 Architecture Setup



node ID
config_file

node ID
config_file

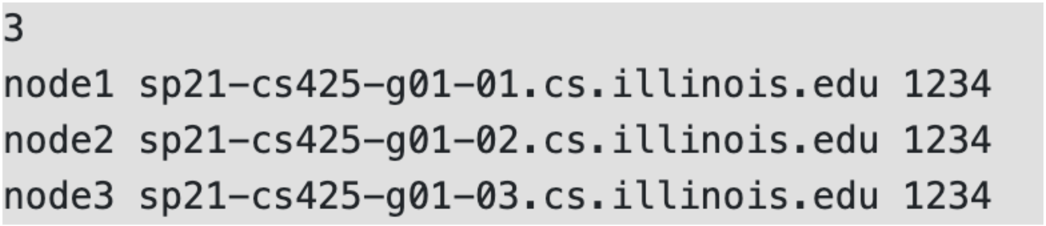
node ID
config_file

node 1

node 2

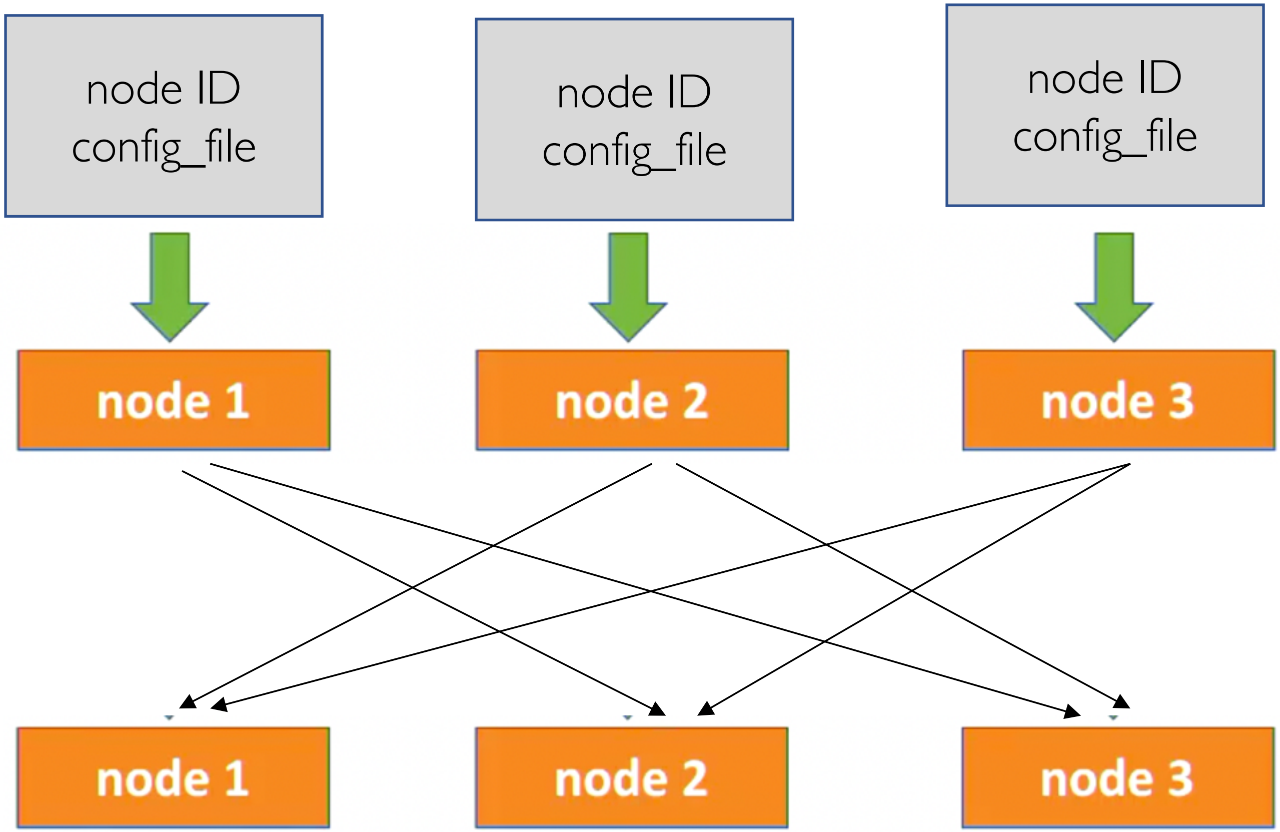
node 3

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



```
3
node1 sp21-cs425-g01-01.cs.illinois.edu 1234
node2 sp21-cs425-g01-02.cs.illinois.edu 1234
node3 sp21-cs425-g01-03.cs.illinois.edu 1234
```

MPI Architecture Setup



node ID
config_file

node ID
config_file

node ID
config_file

node 1

node 2

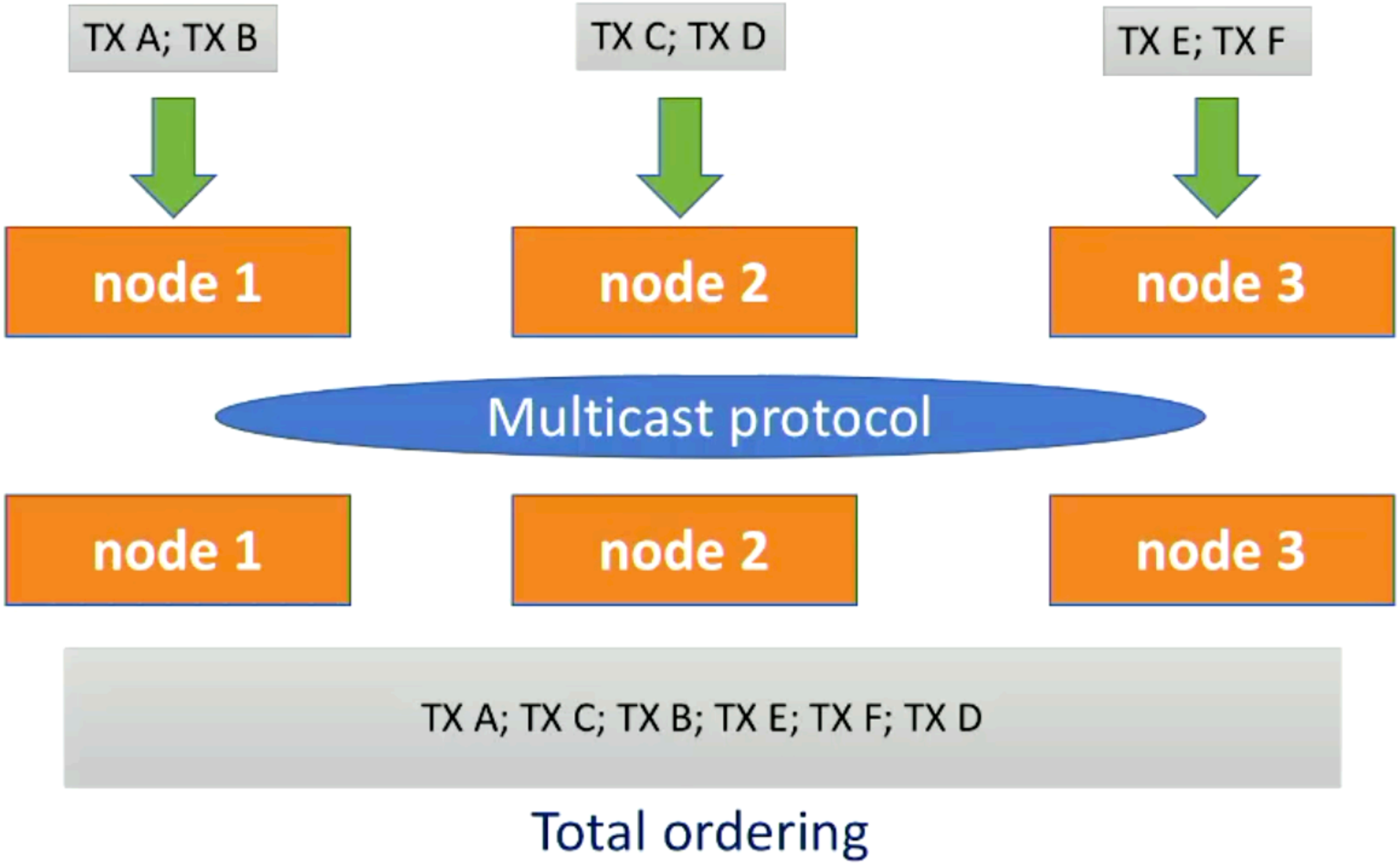
node 3

node 1

node 2

node 3

MP1 Architecture



TX A; TX B

TX C; TX D

TX E; TX F

node 1

node 2

node 3

Multicast protocol

node 1

node 2

node 3

TX A; TX C; TX B; TX E; TX F; TX D

Total ordering

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 Thursday, March 3rd.
    - 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 have released 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!!