Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements for some of materials: Indy Gupta and Nikita Borisov

Logistics

- Note about exams on CampusWire:
  - Midterm 1 (Mar 4-6), Midterm 2 (April 8-10), Finals (May 8-16).
  - Reservation via PrairieTest.
    - You can reserve a slot for Midterm 1 starting Feb 20.
  - If you need DRES accommodations, please upload your Letter of Accommodations on the CBTF website.

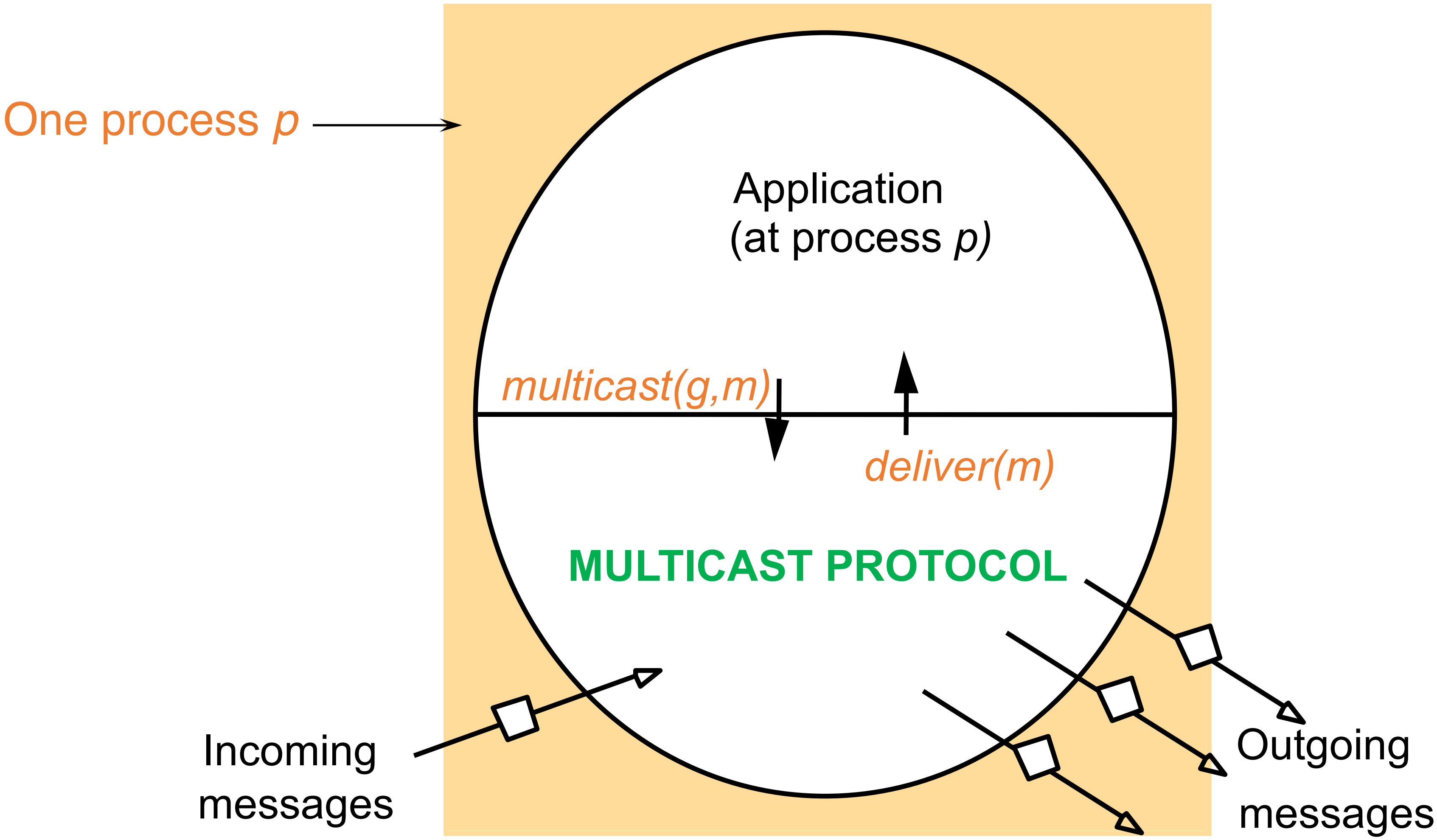
- MP1 has been (will be?) released today.
  - Due on March 14th, 11:59pm.

- HW1 is due next week Friday.
  - You should be able to solve all questions by now.

Today’s agenda

- **Multicast**
  - Chapter 15.4

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.

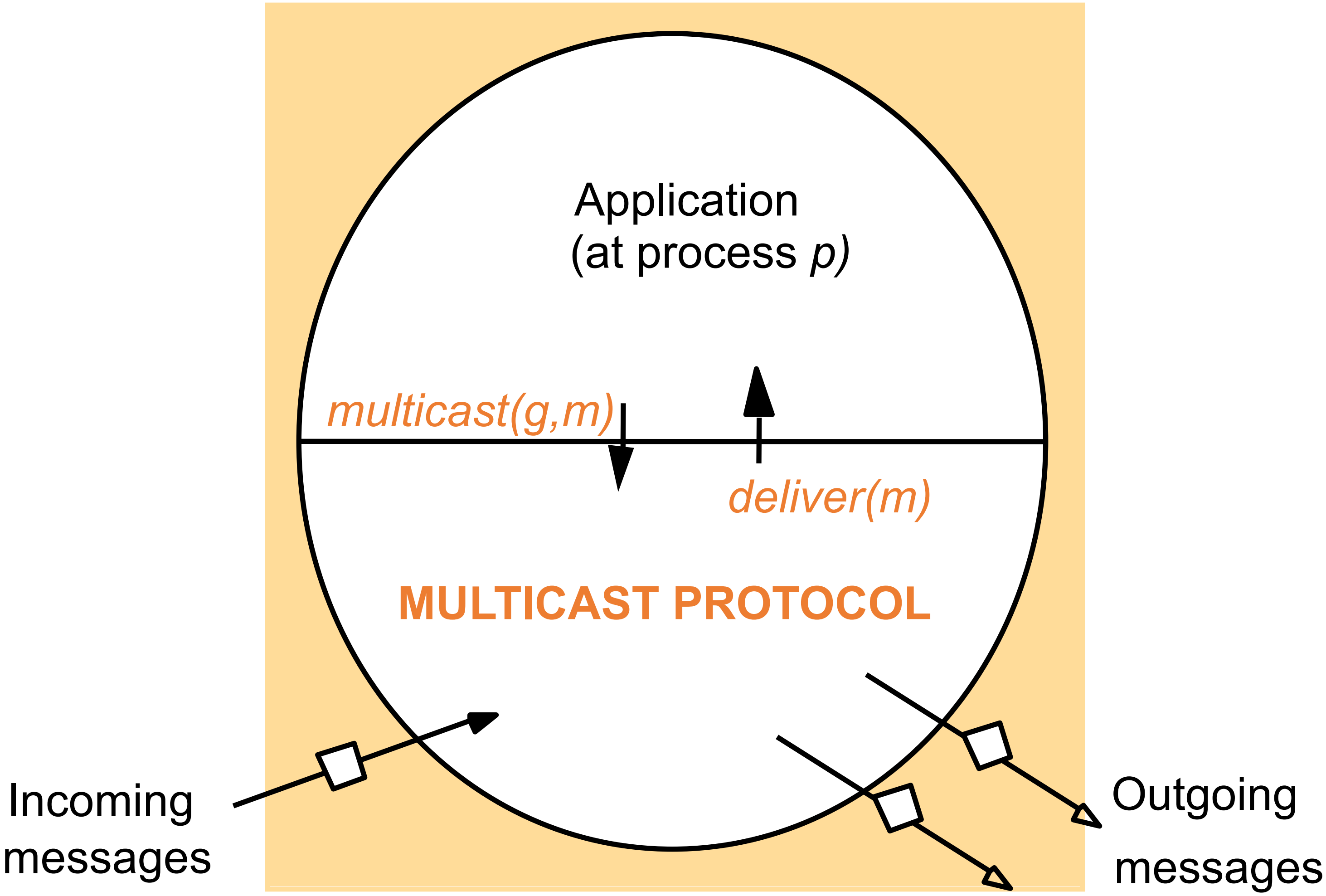
Reliable Multicast (R-Multicast)

- **Integrity**: A *correct* (i.e., non-faulty) process *p* delivers a message *m* at most onc
  - *Assur* *wice*
- **Validity**: I hen it will eventuall
  - *Liven*
- **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



Application

(at process *p*)

multicast(g,m)

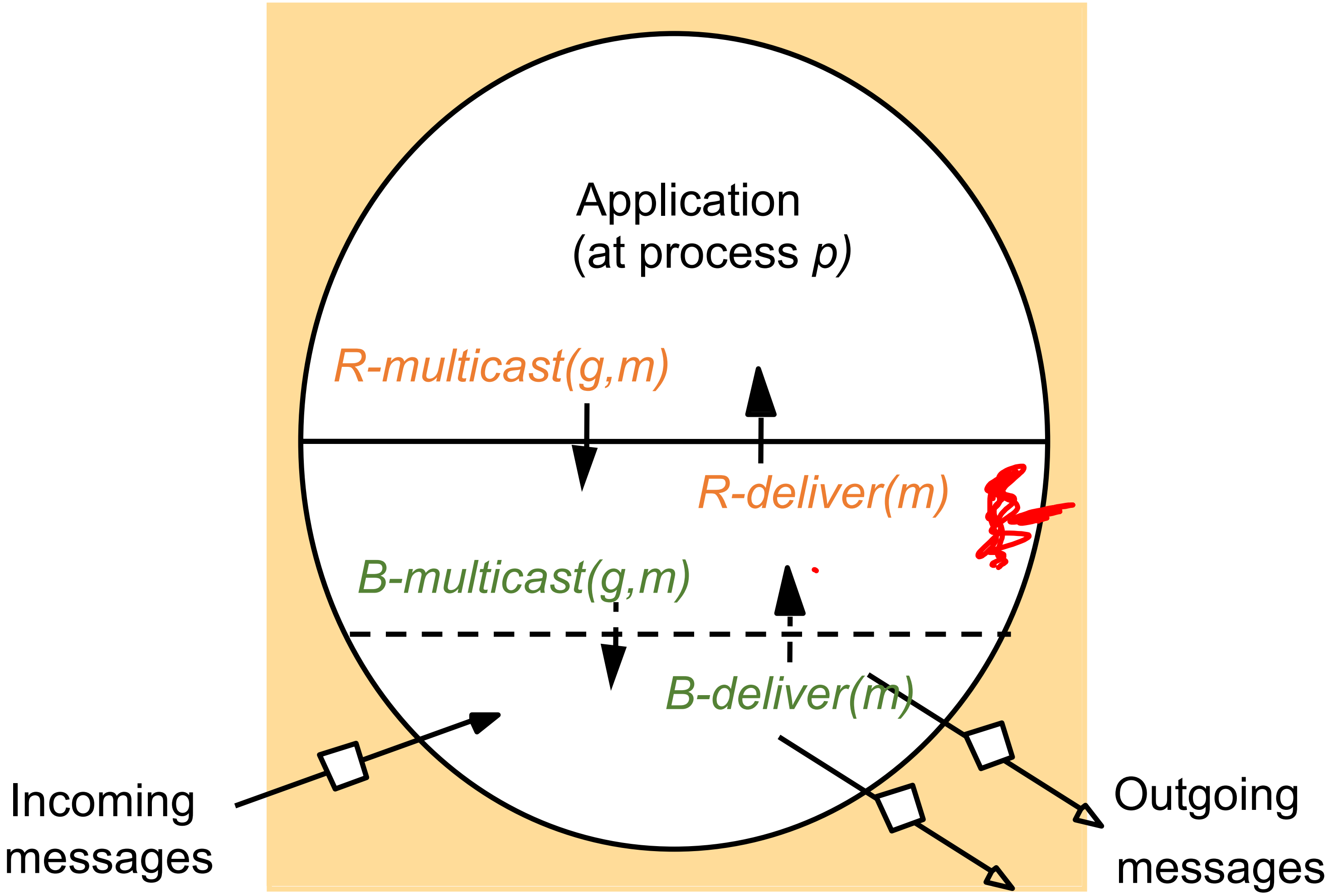
deliver(m)

MULTICAST PROTOCOL

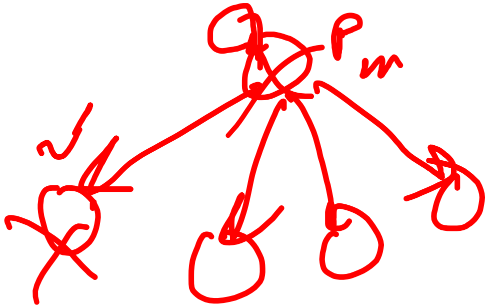
Incoming messages

Outgoing messages

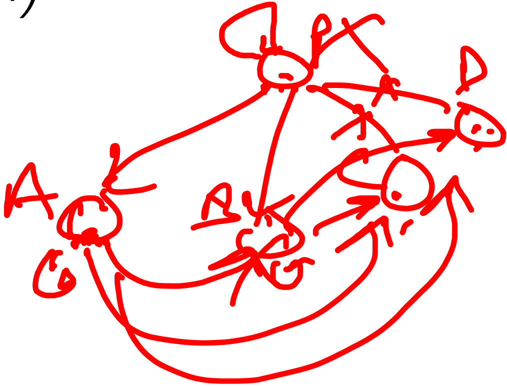
Implementing R-Multicast



Implementing R-Multicast



```
On initialization
    Received := {};
For process p to R-multicast message m to group g
    B-multicast(g,m); (p ∈ g is included as destination)
On B-deliver(m) at process q in g = group(m)
    if (m ∉ Received):
        Received := Received ∪ {m};
        if (q ≠ 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

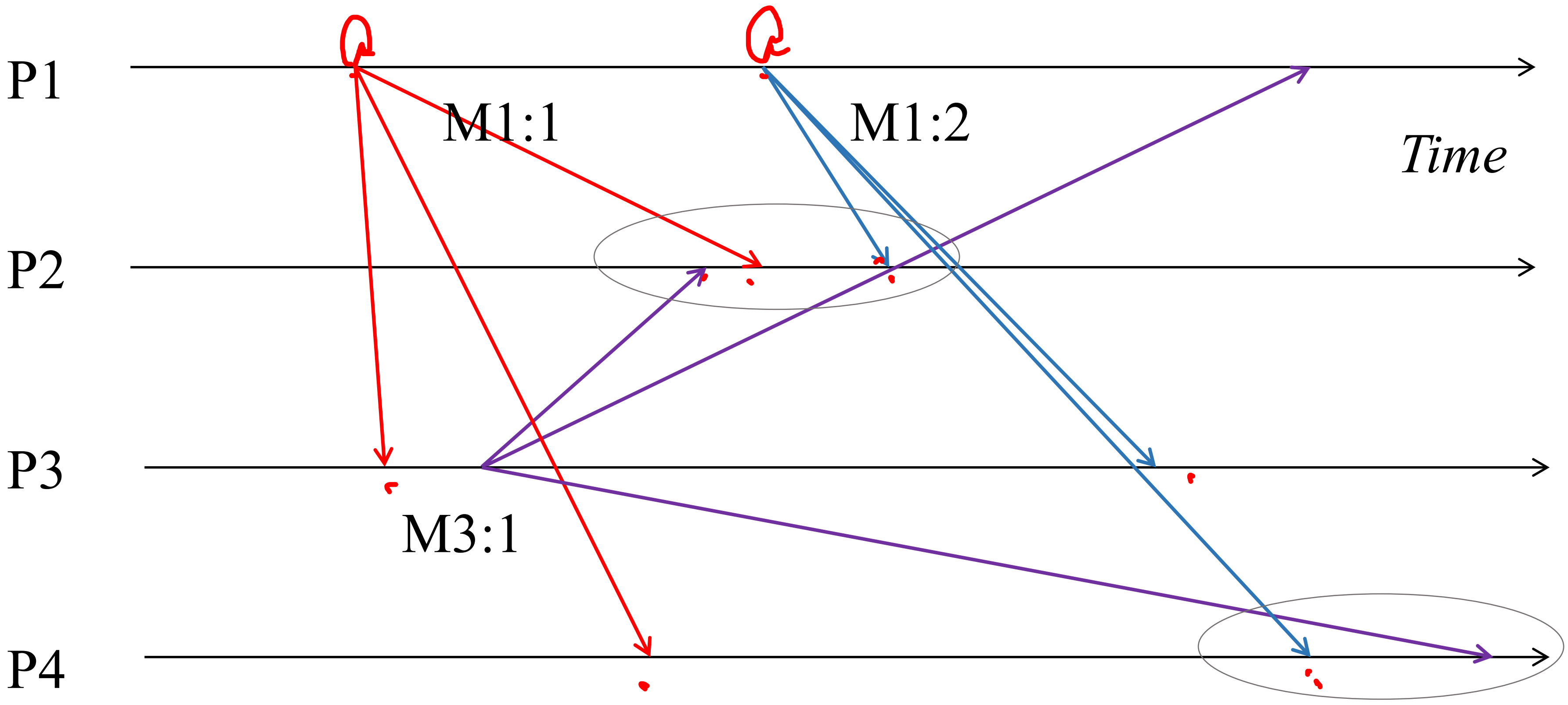
- Three popular flavors implemented by several multicast protocols:
  1. FIFO ordering
  2. Causal ordering
  3. Total ordering

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) → multicast(g,m') then any correct process that delivers m' will have already delivered m.*

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.

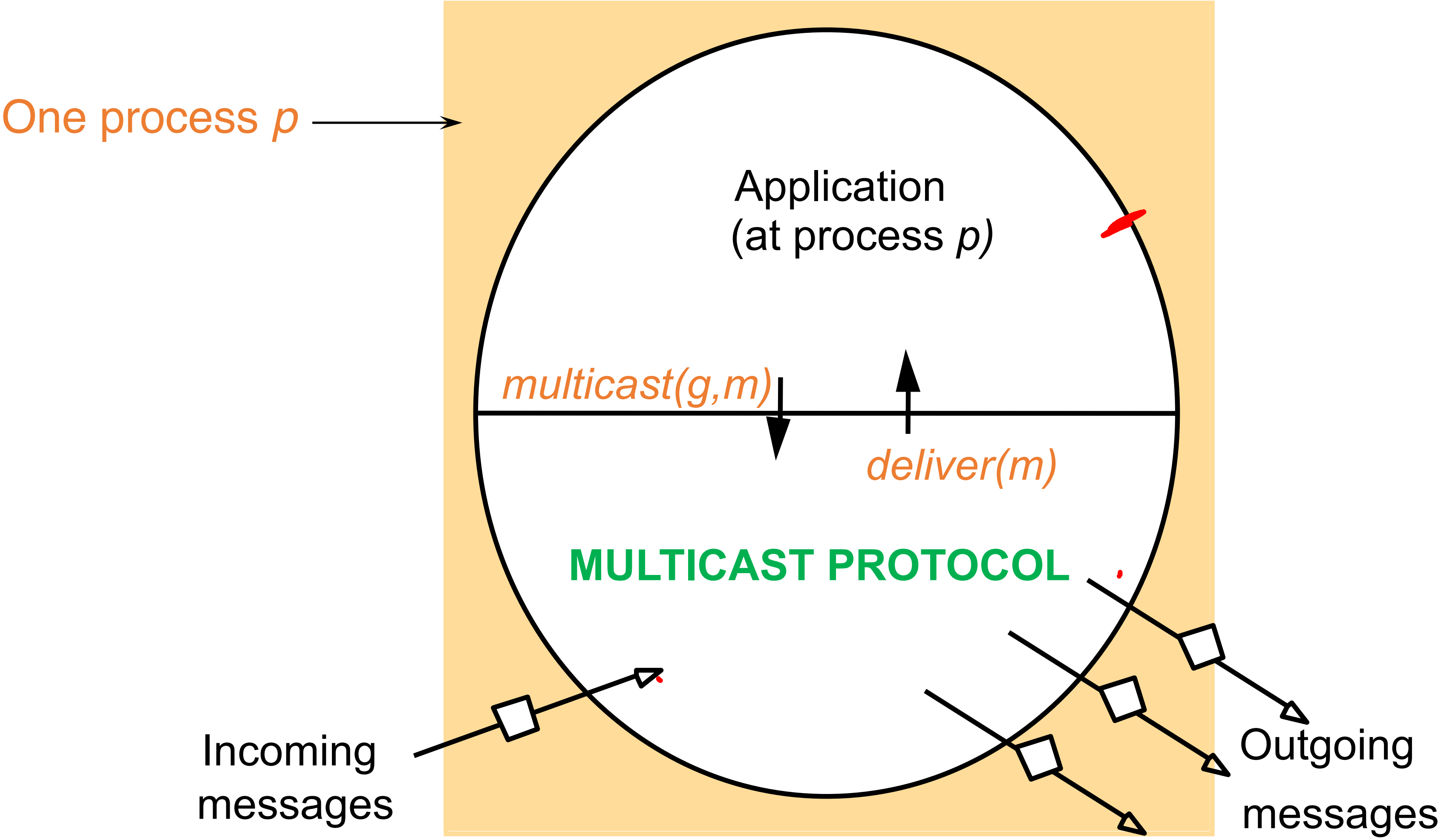
- A variety of systems implement causal ordering:
  - social networks, bulletin boards, comments on websites, etc.

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

What we are designing in this class?



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

HB Relationship for Causal Ordering

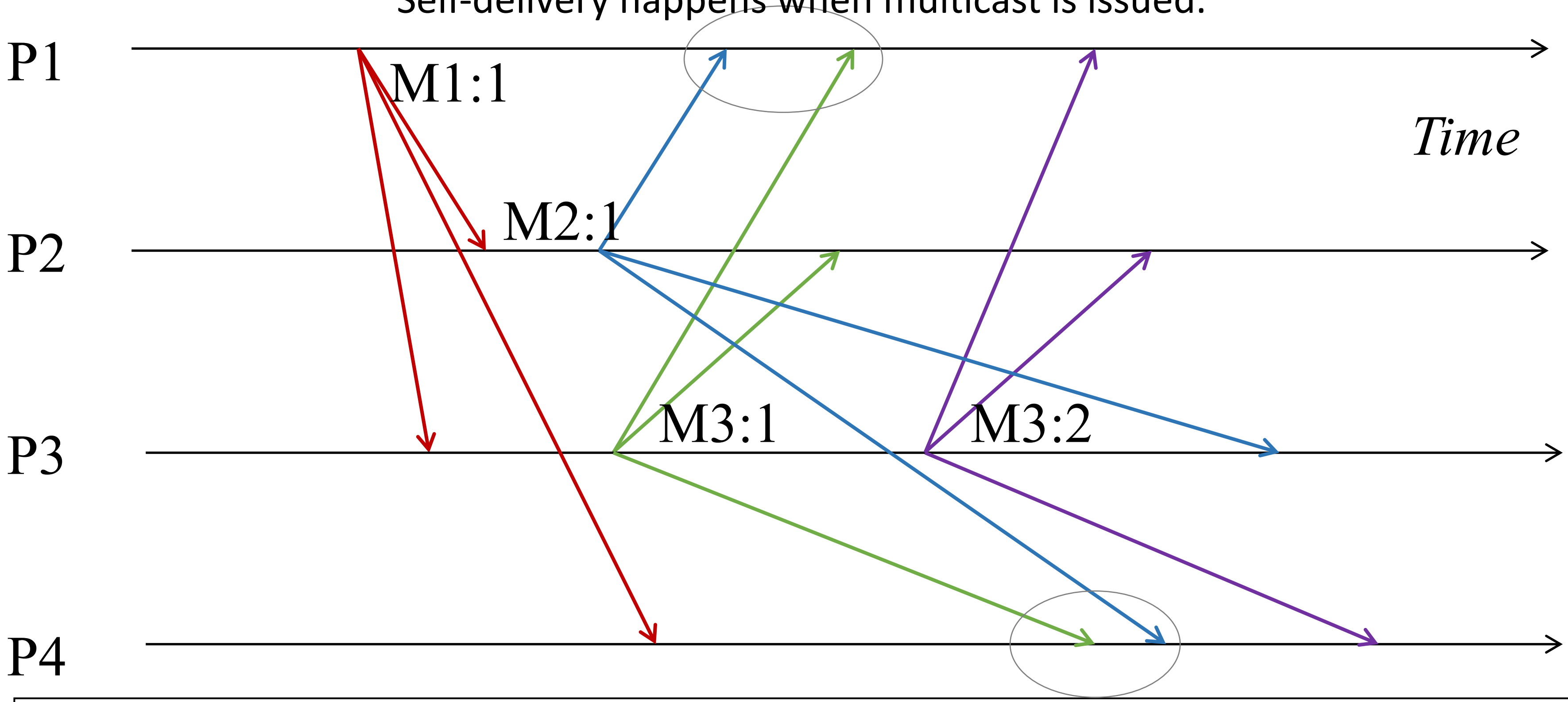
- HB rules in causal ordered multicast:
  - If ∃ **p_i** , **$e \rightarrow_i e'$** then **$e \rightarrow e'$**.
    - If ∃ **p_i** , multicast(*g*,*m*) $\rightarrow_i$ multicast(*g*,*m'*), then multicast(*g*,*m*) → multicast(*g*,*m'*)
    - If ∃ **p_i** , delivery(*m*) $\rightarrow_i$ multicast(*g*,*m'*),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

HB Relationship for Causal Ordering

- HB rules in causal ordered multicast:
  - If $\exists\ \mathbf{p_i}$, **e** $\rightarrow_i$ **e’** then **e** → **e’**.
    - If $\exists\ \mathbf{p_i}$, multicast(*g*,*m*) $\rightarrow_i$ multicast(*g*,*m’*), then multicast(*g*,*m*) → multicast(*g*,*m’*)
    - If $\exists\ \mathbf{p_i}$, delivery(*m*) $\rightarrow_i$ multicast(*g*,*m’*), then delivery(*m*) → multicast(*g*,*m’*)
    - …
  - ~~For any message m, **send(m)** → **receive(m)**~~
    - For any *multicast* message m, **multicast(g,*m*)** → **delivery(m)**
  - If **e** → **e’** and **e’** → **e”** then **e** → **e”**
    - multicast(*g*,*m*) at p_i → delivery(m) at p_j
    - delivery(*m*) at p_j → multicast(*g*,*m’*) at p_j
    - multicast(*g*,*m*) at p_i → 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

Message delivery indicated by arrow endings.
Self-delivery happens when multicast is issued.



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.

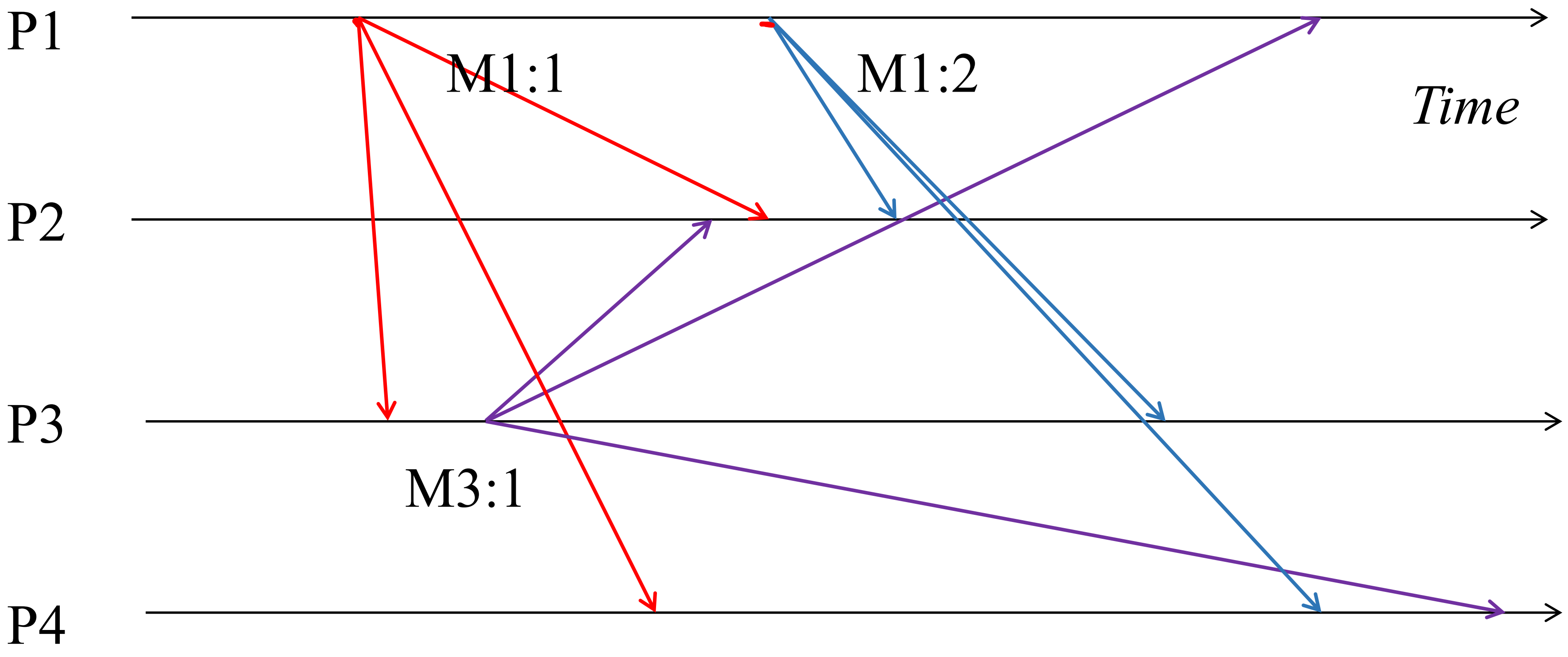
Causal vs FIFO

- Does Causal Ordering imply FIFO Ordering?
    - Yes

- Does FIFO Order imply Causal Order?
    - No

Example

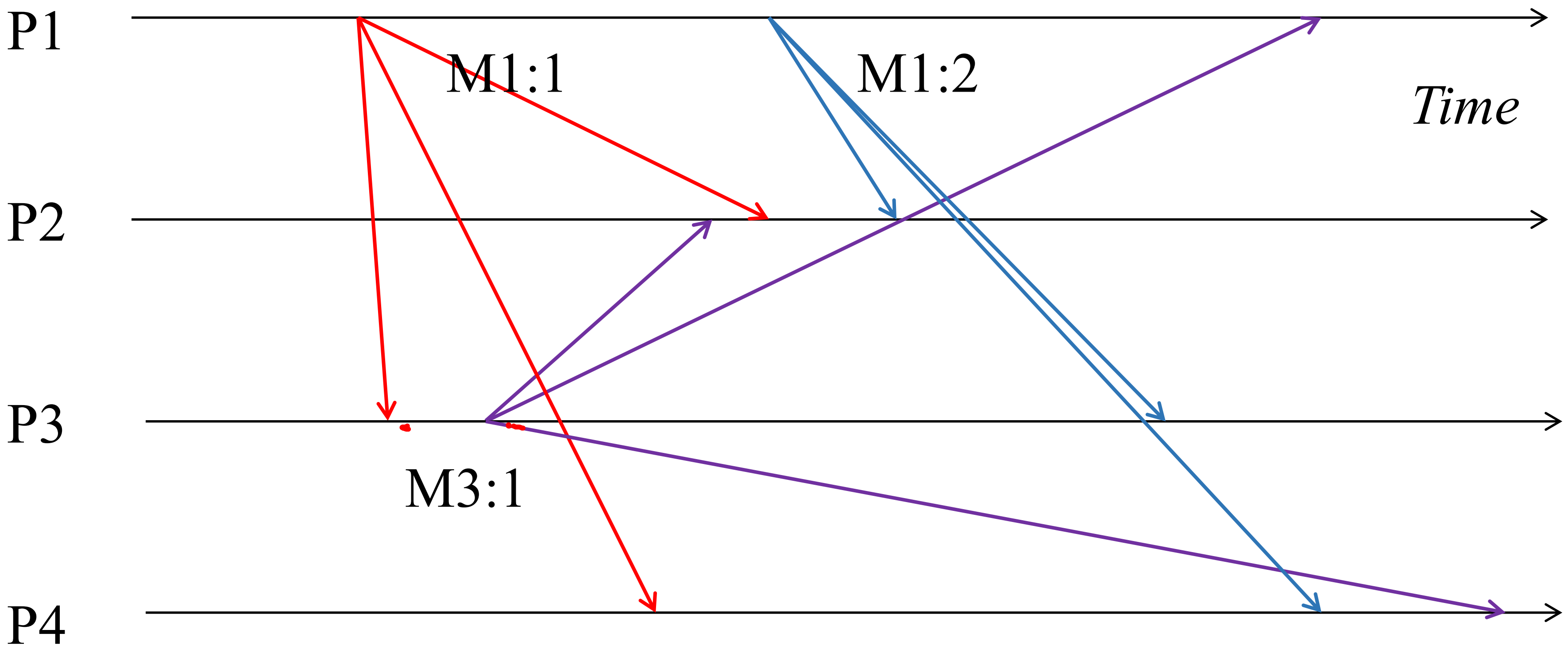
Message delivery indicated by arrow endings.
Self-delivery happens when multicast is issued.



Does this satisfy FIFO order?

Example

Message delivery indicated by arrow endings.
Self-delivery happens when multicast is issued.

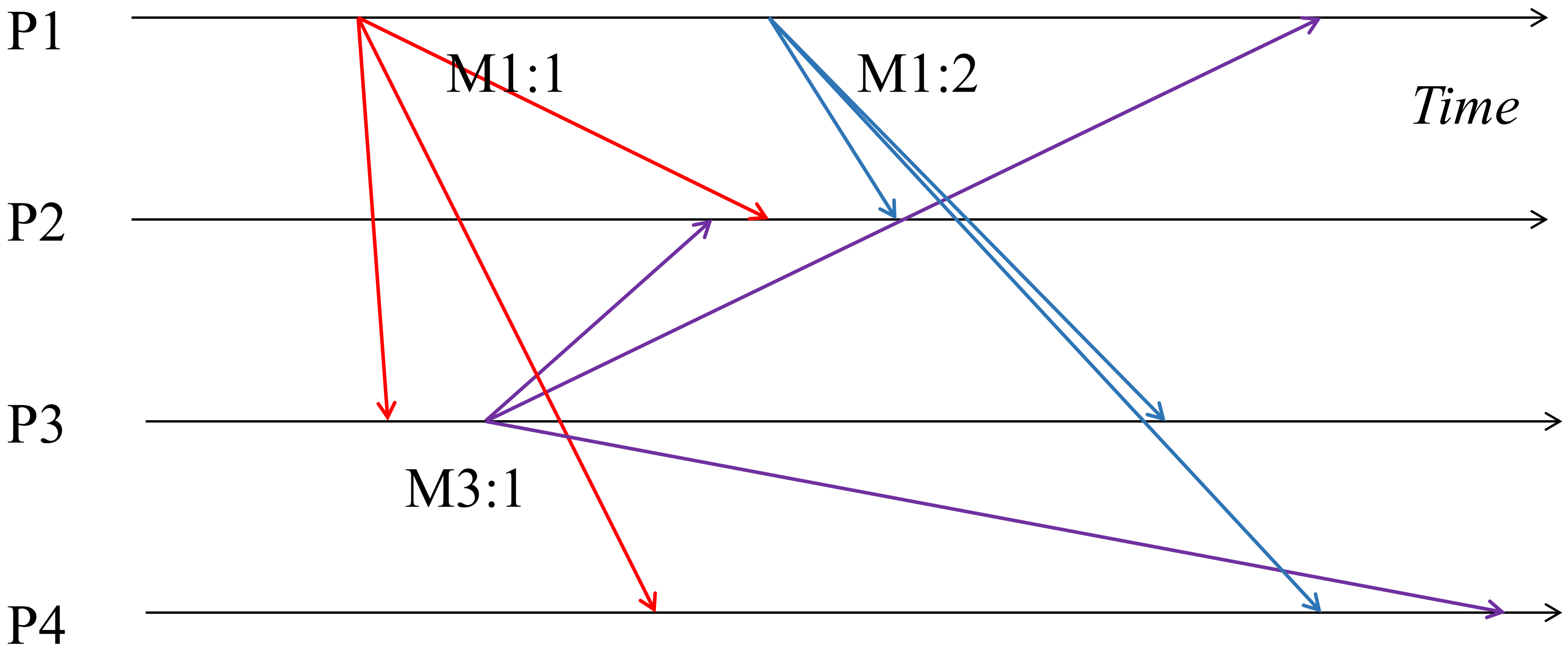


Does this satisfy FIFO order?

Yes

Example

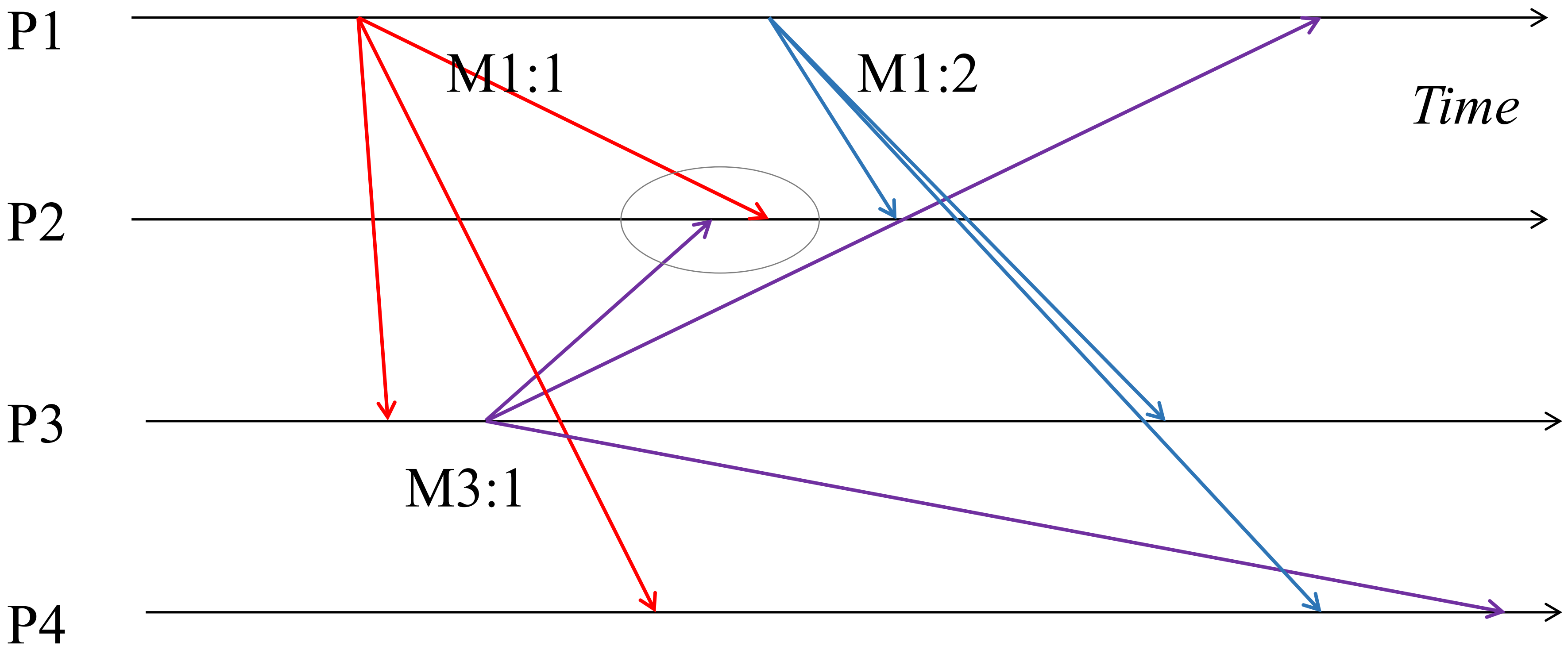
Message delivery indicated by arrow endings.
Self-delivery happens when multicast is issued.



Does this satisfy causal order?

Example

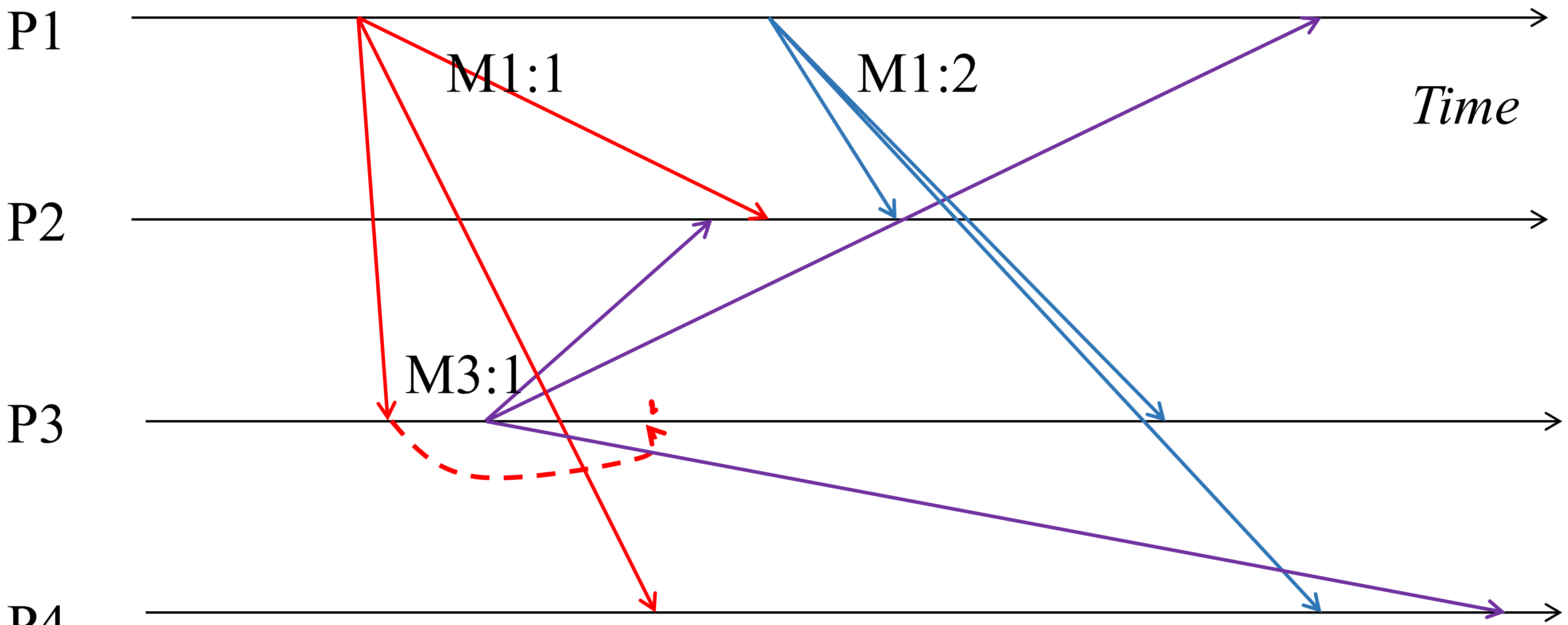
Message delivery indicated by arrow endings.
Self-delivery happens when multicast is issued.



Does this satisfy causal order?
No

Example

Message delivery indicated by (extended) arrow endings.
Self-delivery happens when multicast is issued.



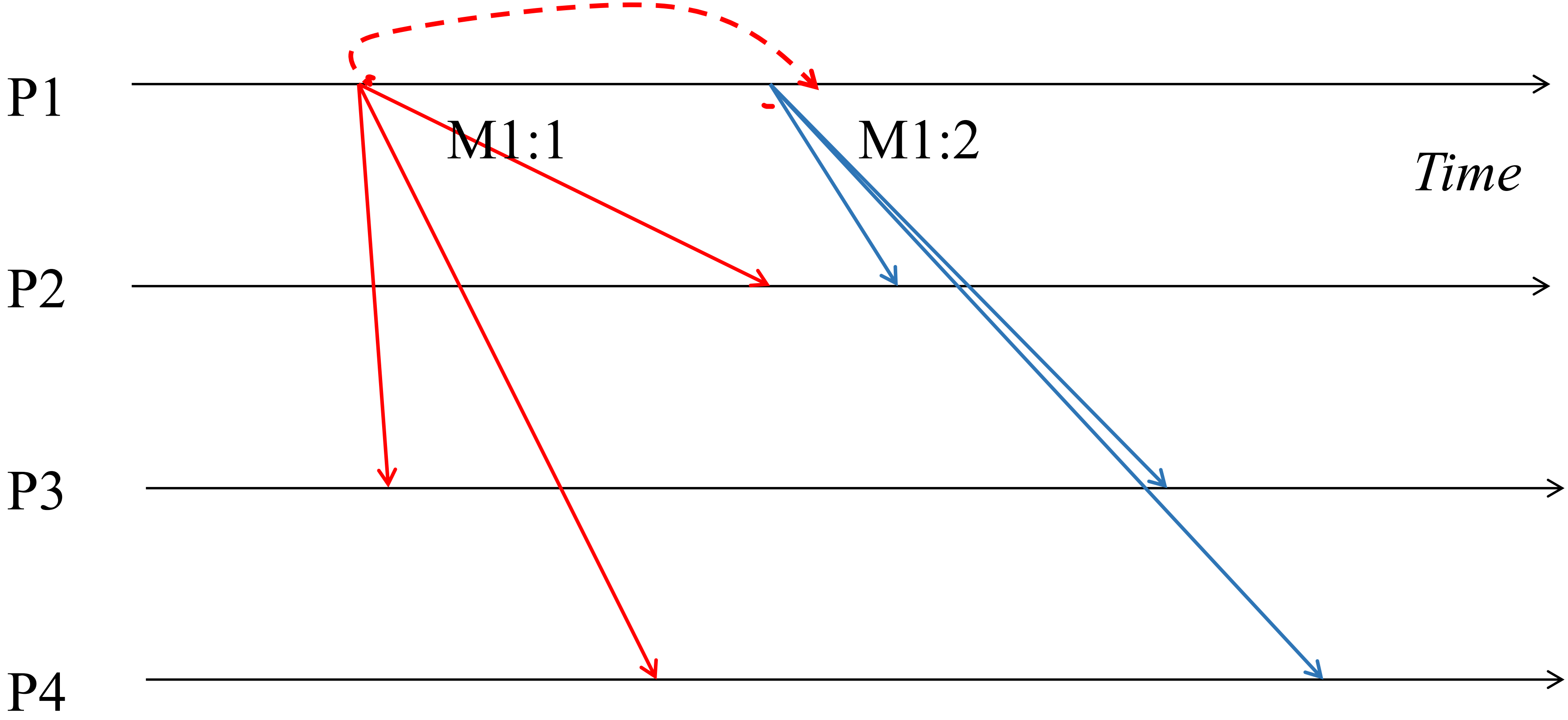
M1:1 is delivered at P3 after M3:1's multicast.

Does this satisfy causal order?

Yes

Example

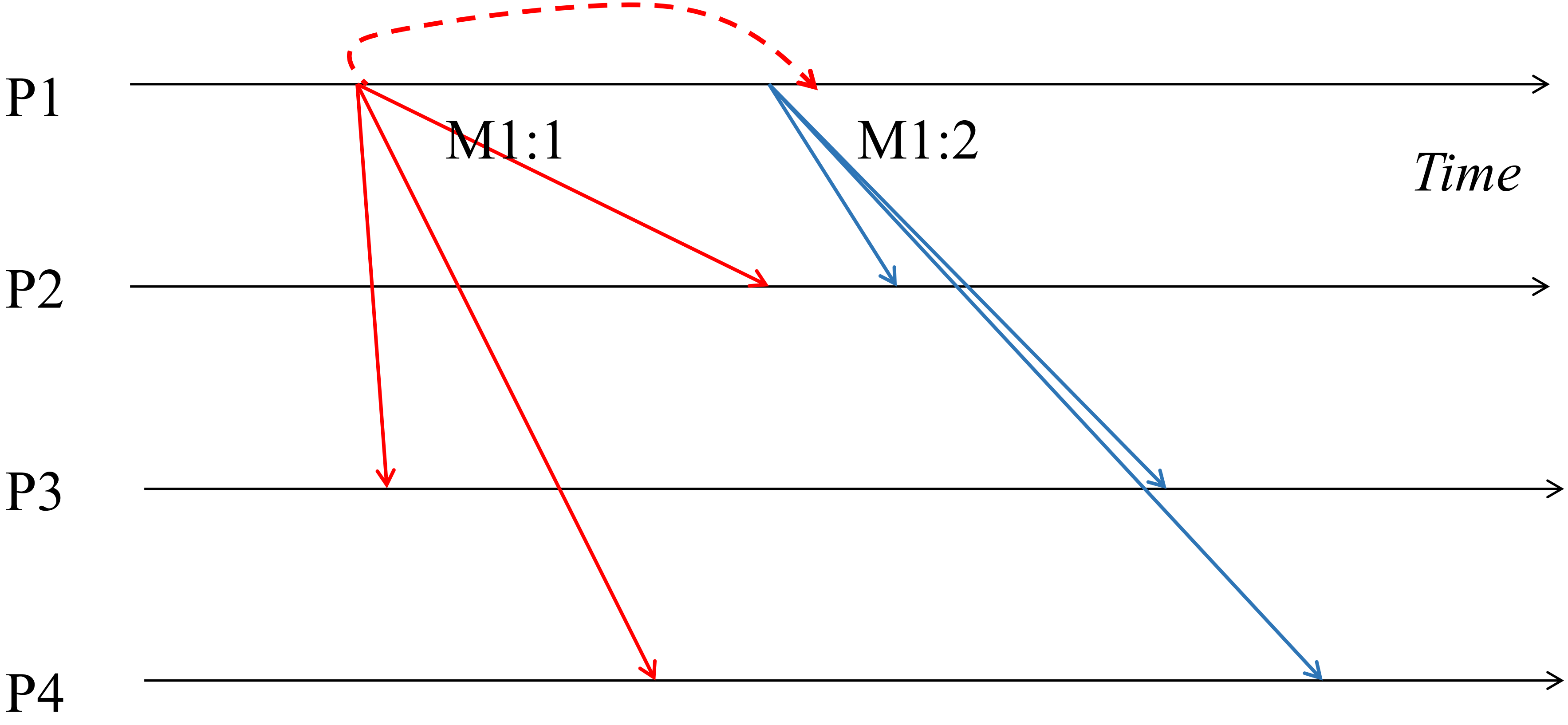
Message delivery indicated by (extended) arrow endings.
Self-delivery happens when multicast is issued when no extra arrow.



Does this satisfy causal order?
No

Example

Message delivery indicated by (extended) arrow endings.
Self-delivery happens when multicast is issued when no extra arrow.



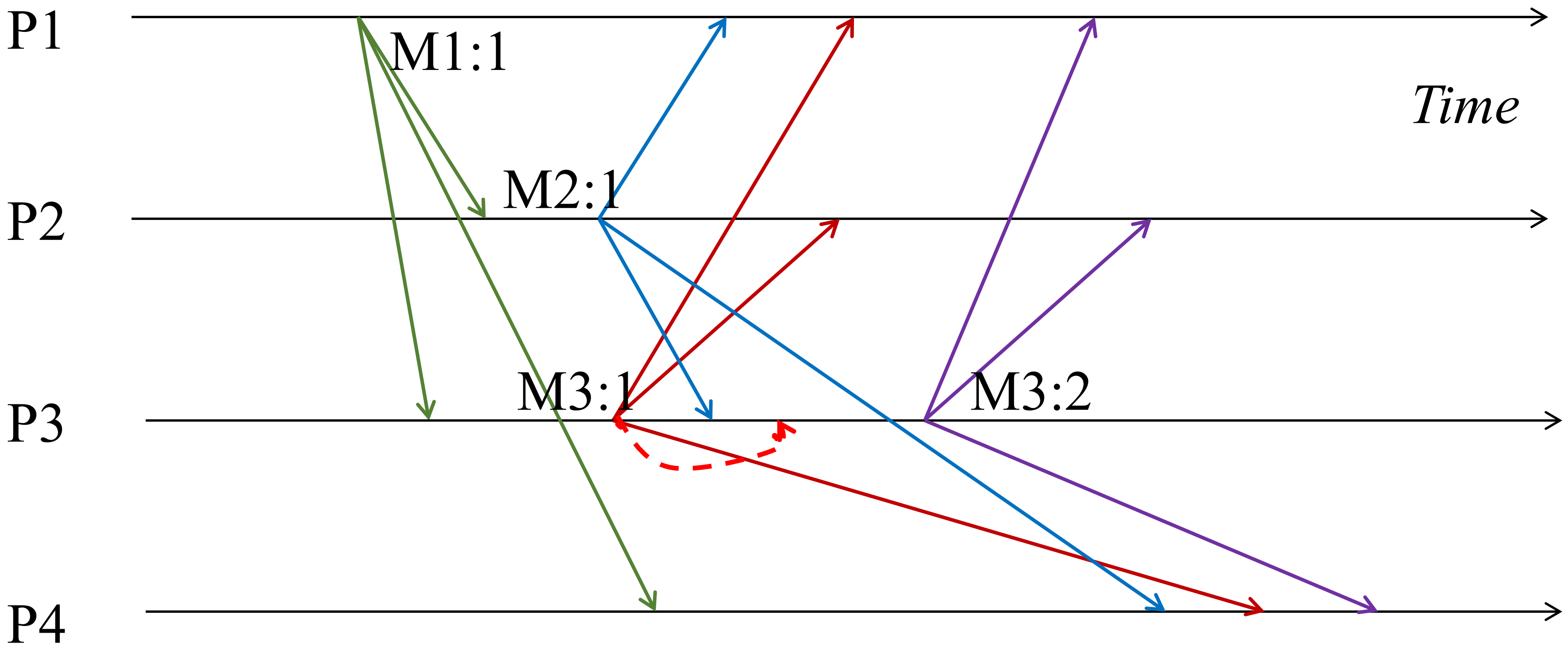
Does this satisfy FIFO order?
No

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

Message delivery indicated by (extended) arrow endings.
Self-delivery happens when multicast is issued (when no extra arrow).



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

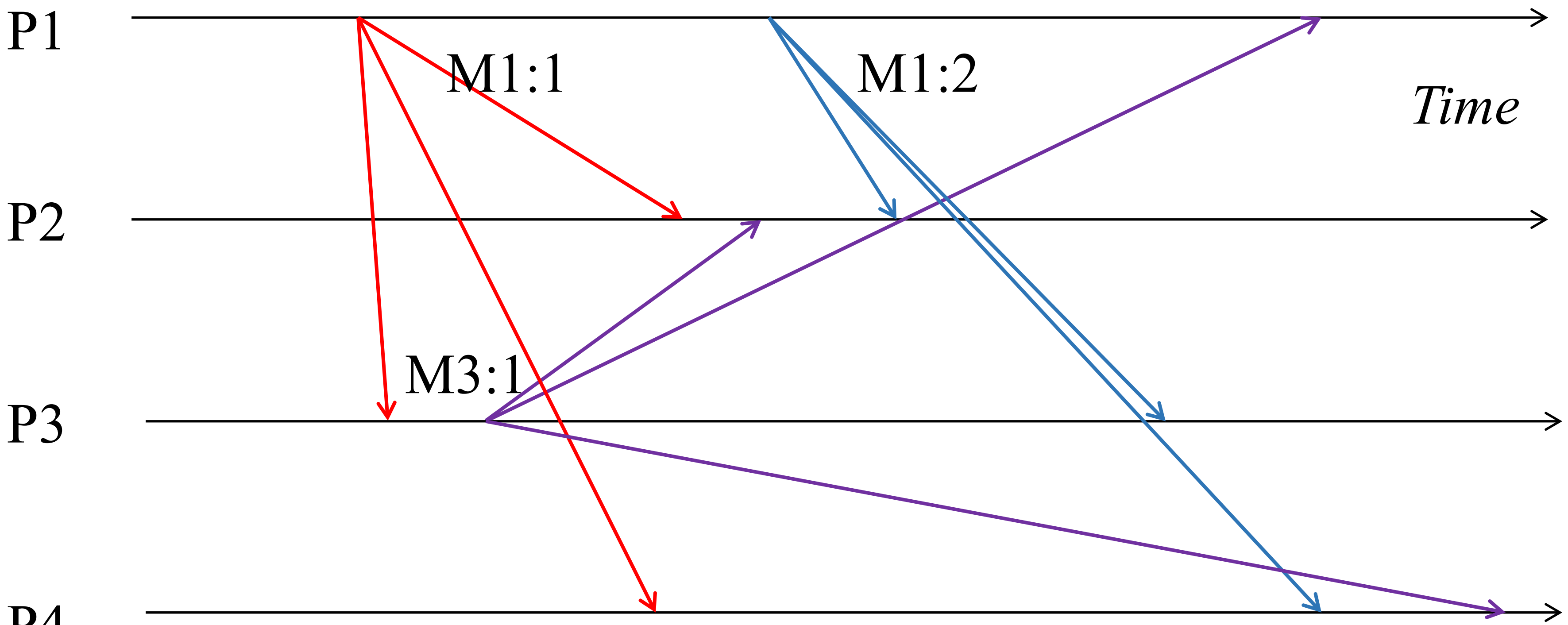
- Total ordering does not imply causal ordering.
- Causal ordering does not imply total ordering.

Hybrid variants

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

Example

Message delivery indicated by (extended) arrow endings.
Self-delivery happens when multicast is issued (when no extra arrow).

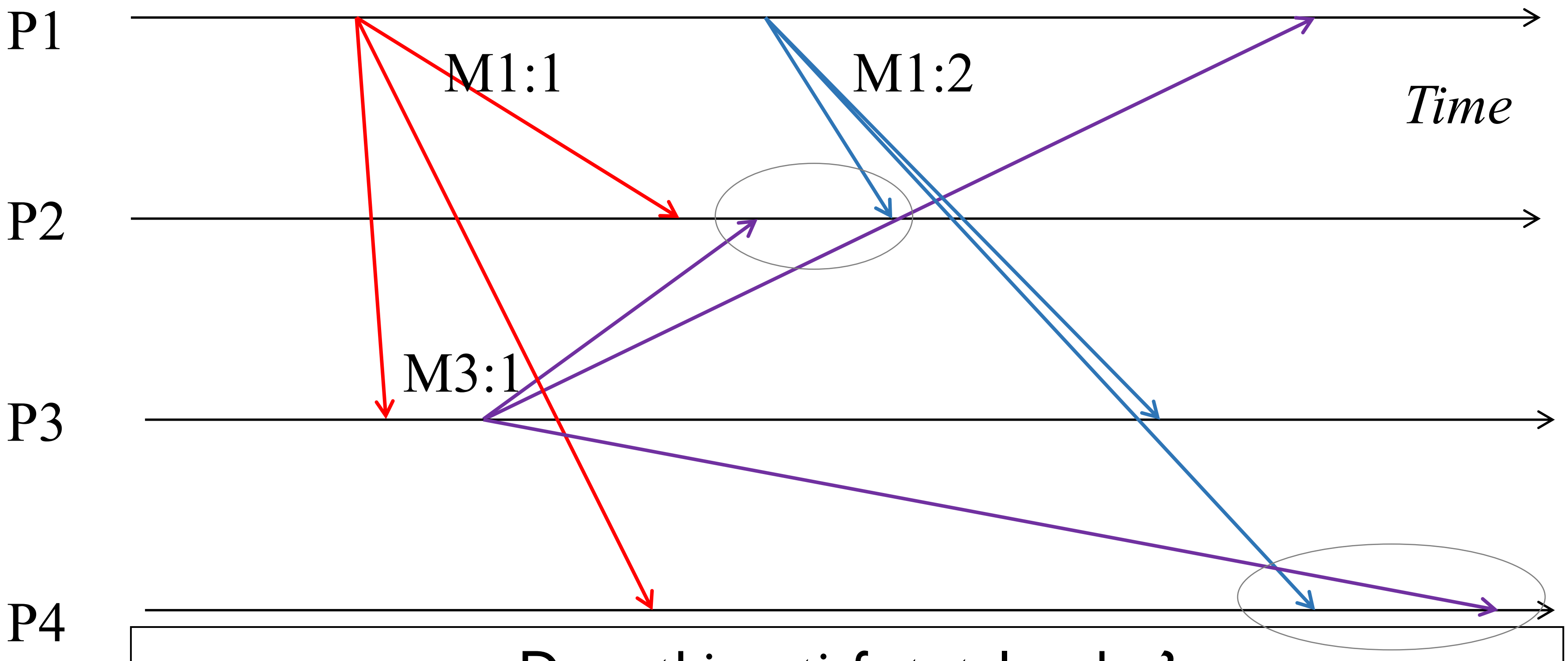


Does this satisfy causal (and FIFO) order?

Yes

Example

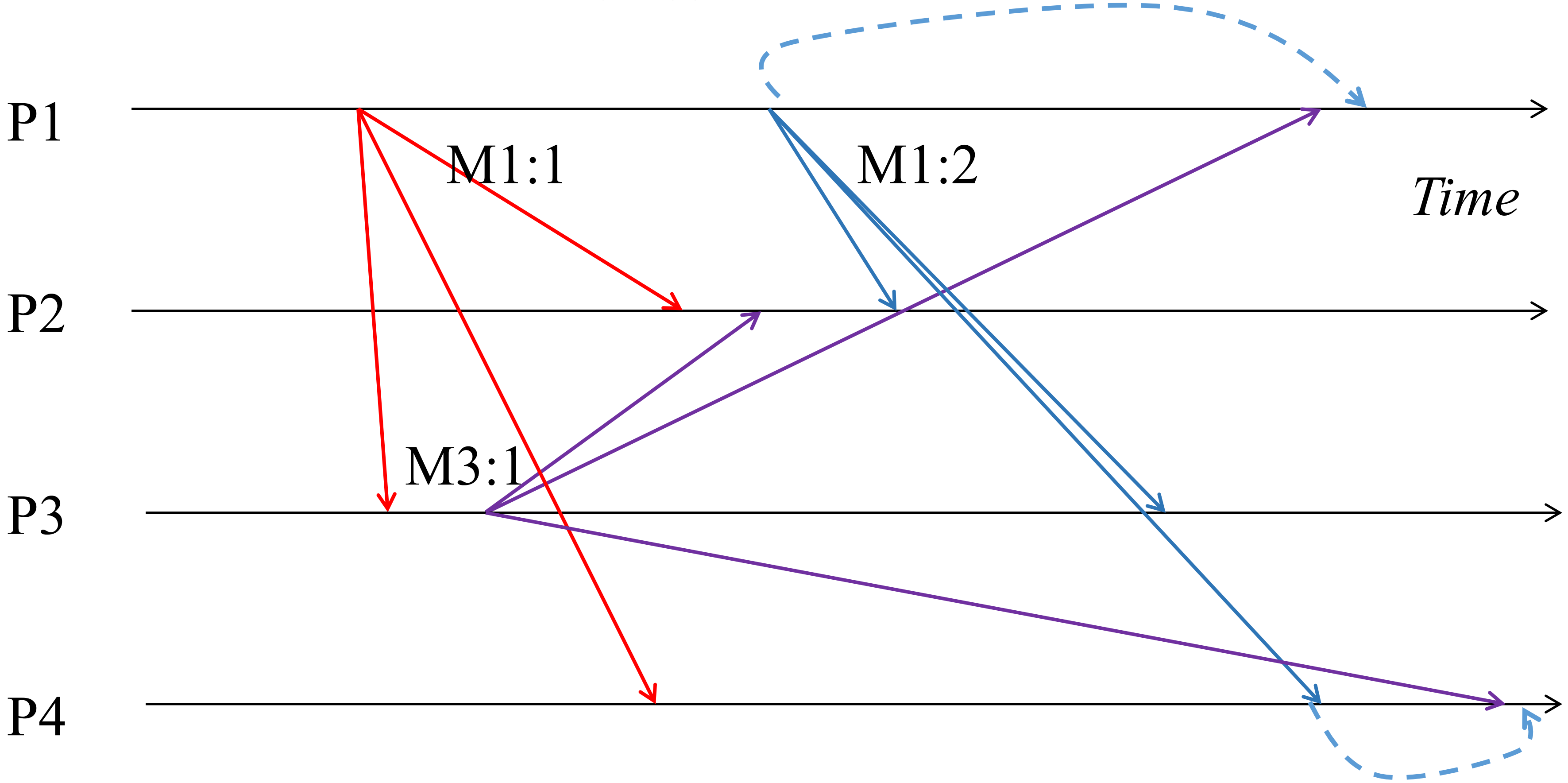
Message delivery indicated by (extended) arrow endings.
Self-delivery happens when multicast is issued (when no extra arrow).



Does this satisfy total order?
No

Example

Message delivery indicated by (extended) arrow endings.
Self-delivery happens when multicast is issued (when no extra arrow).



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*.

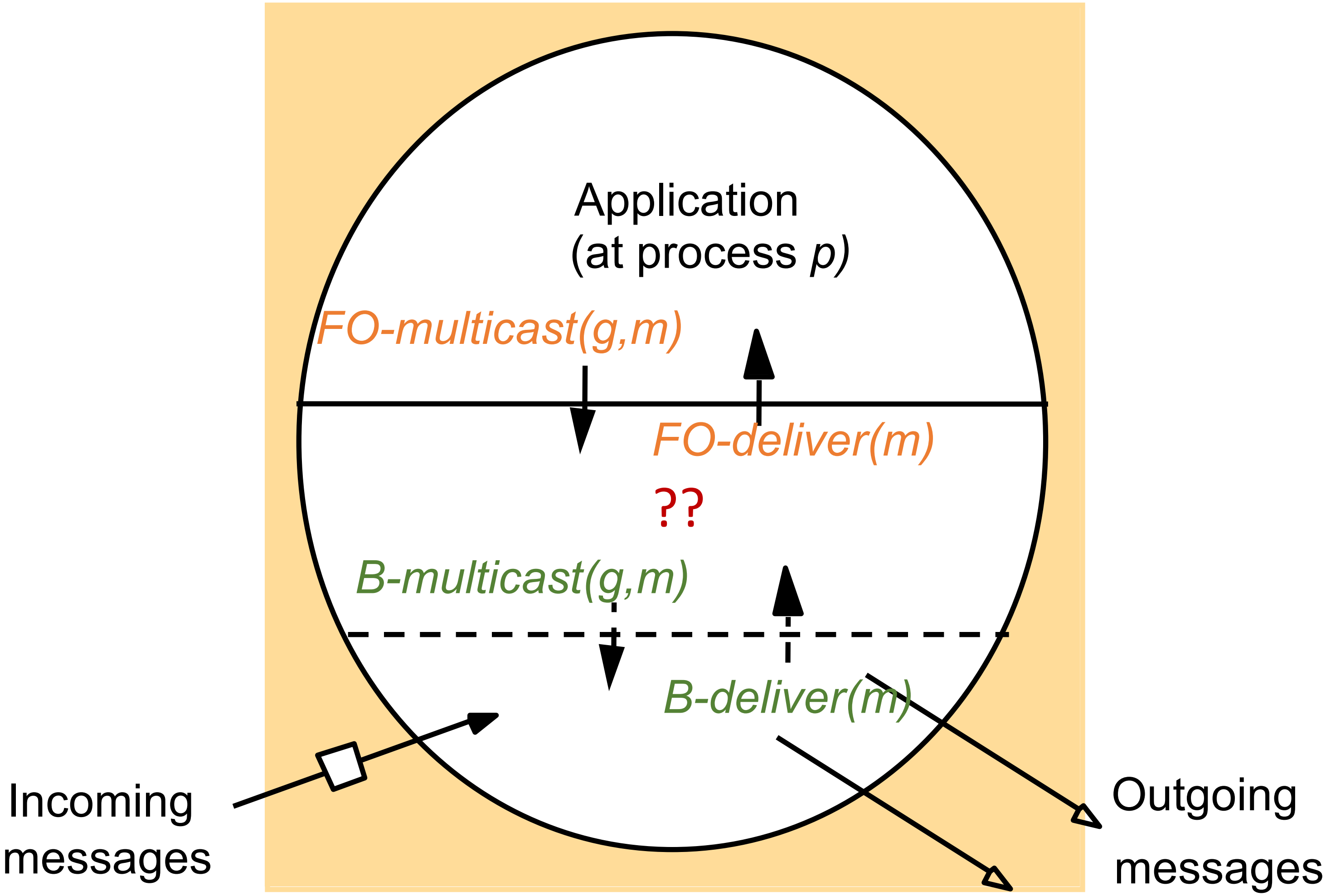
Next Question

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

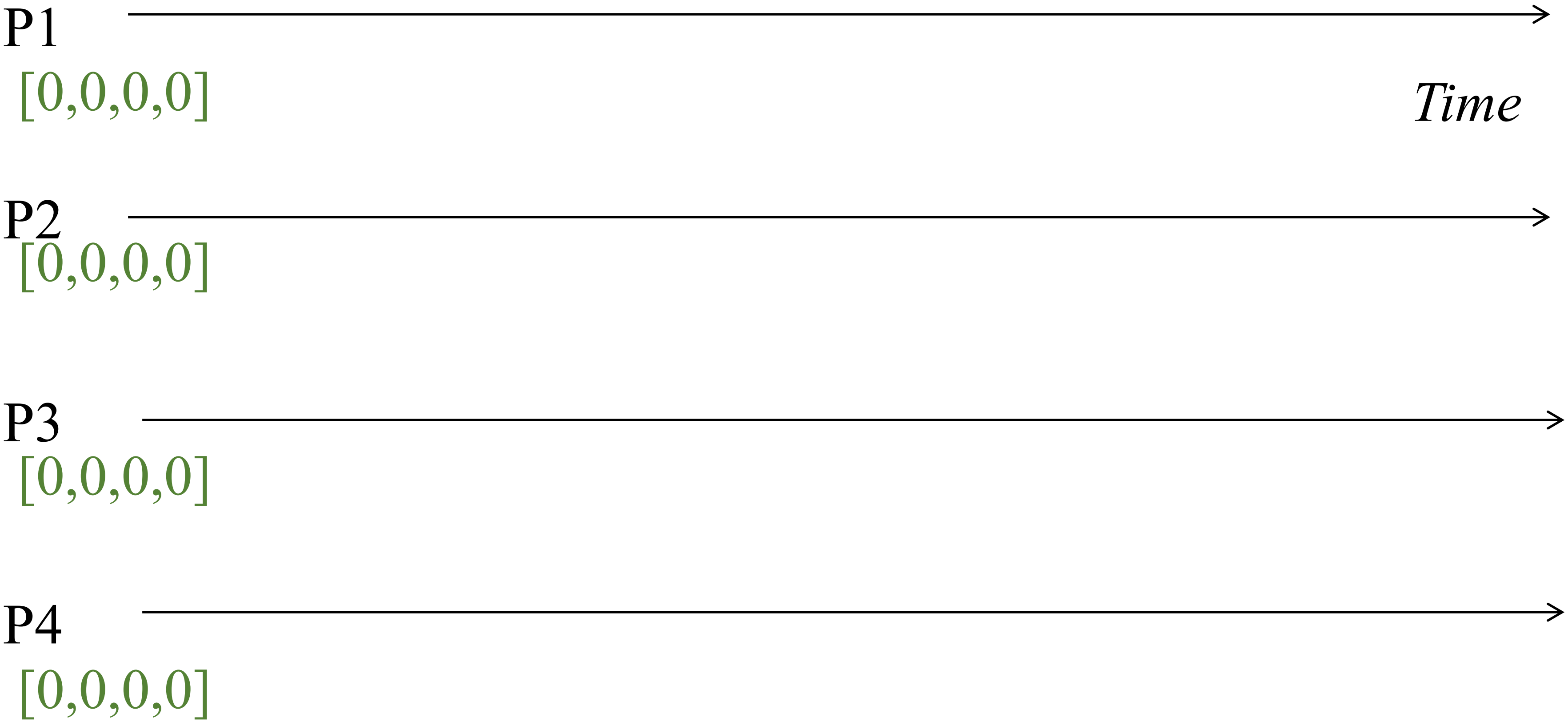
- On FO-multicast(g,m) at process P*j*:

```
set 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]

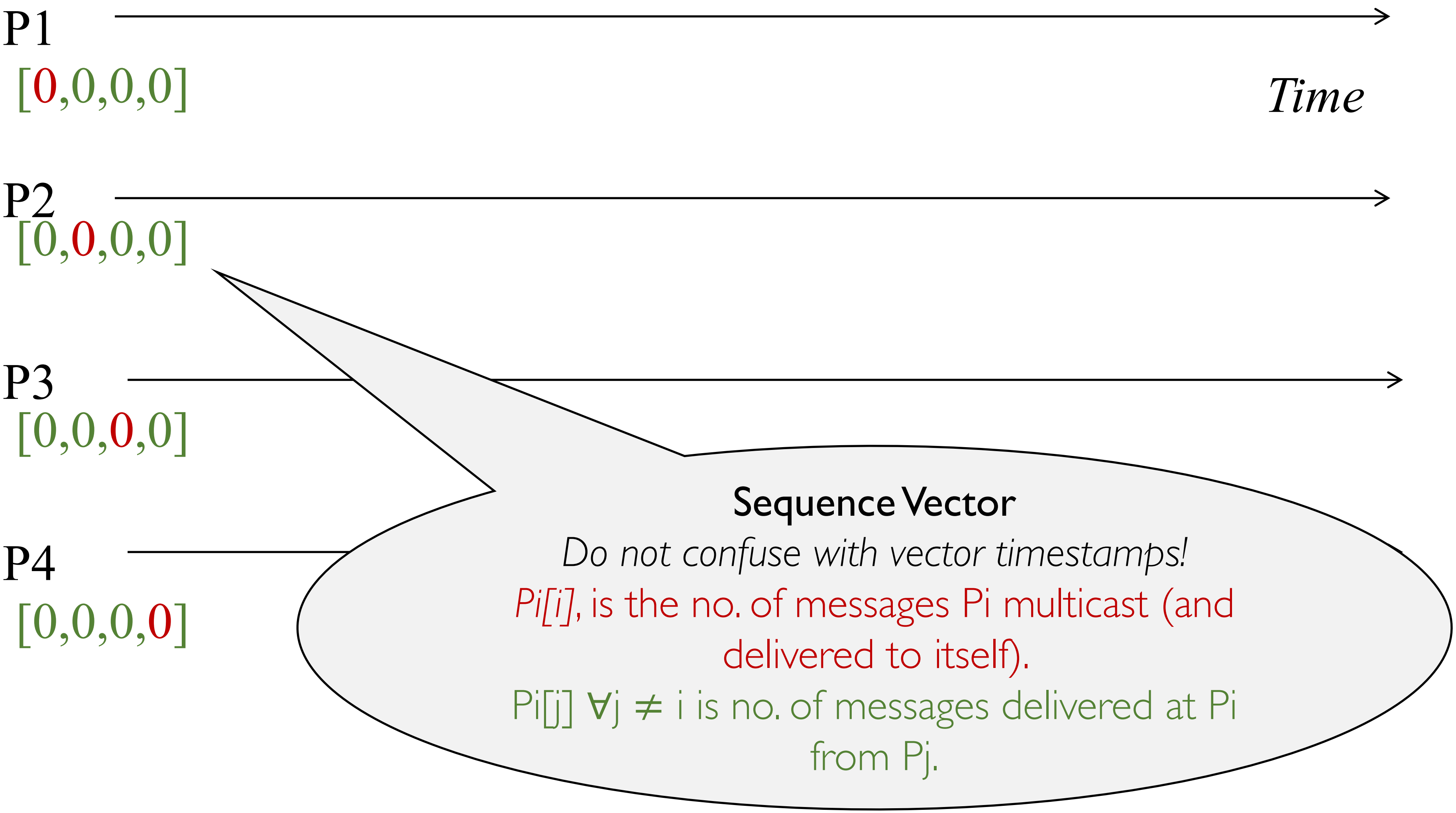
P3

[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]

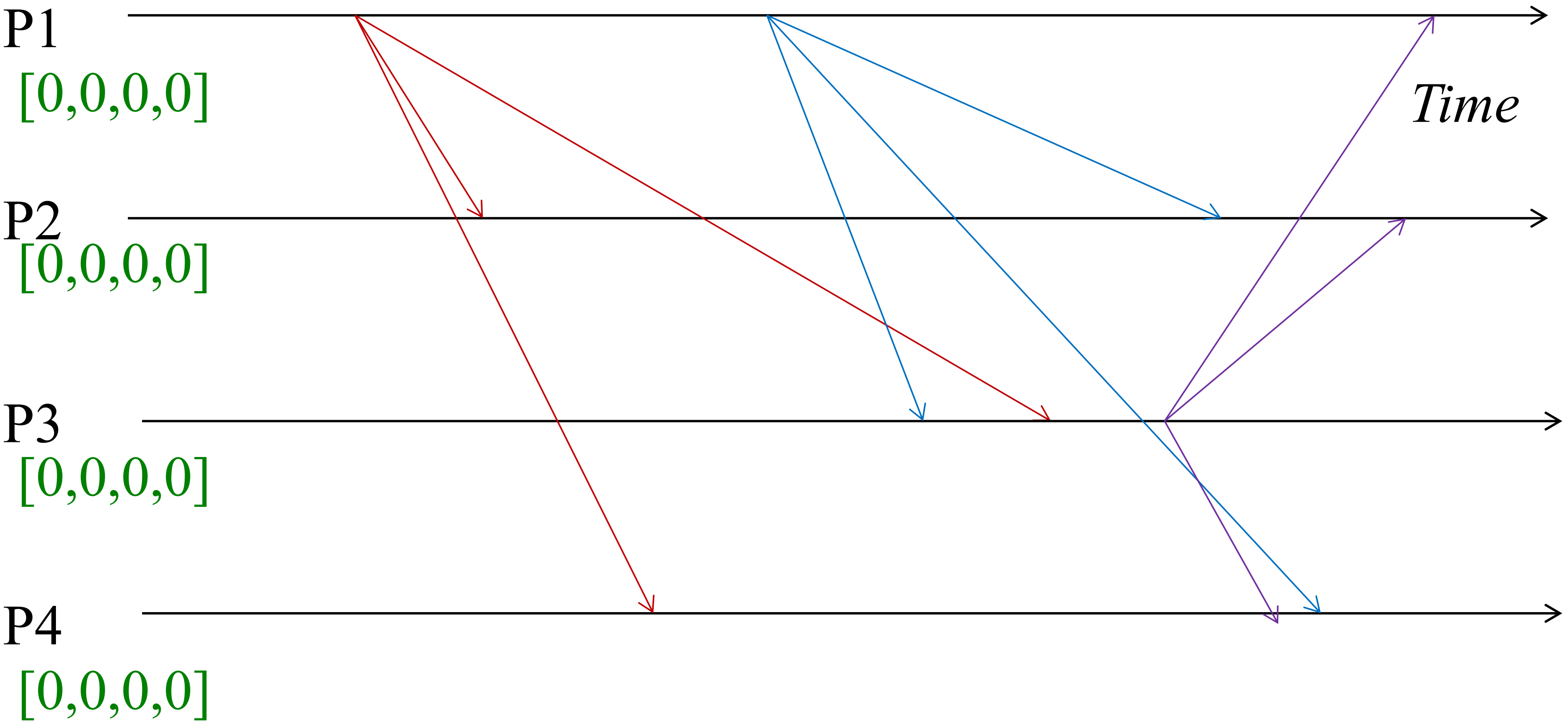
P3

[0,0,0,0]

P4

[0,0,0,0]

FIFO order multicast execution



P1

[0,0,0,0]

P2

[0,0,0,0]

P3

[0,0,0,0]

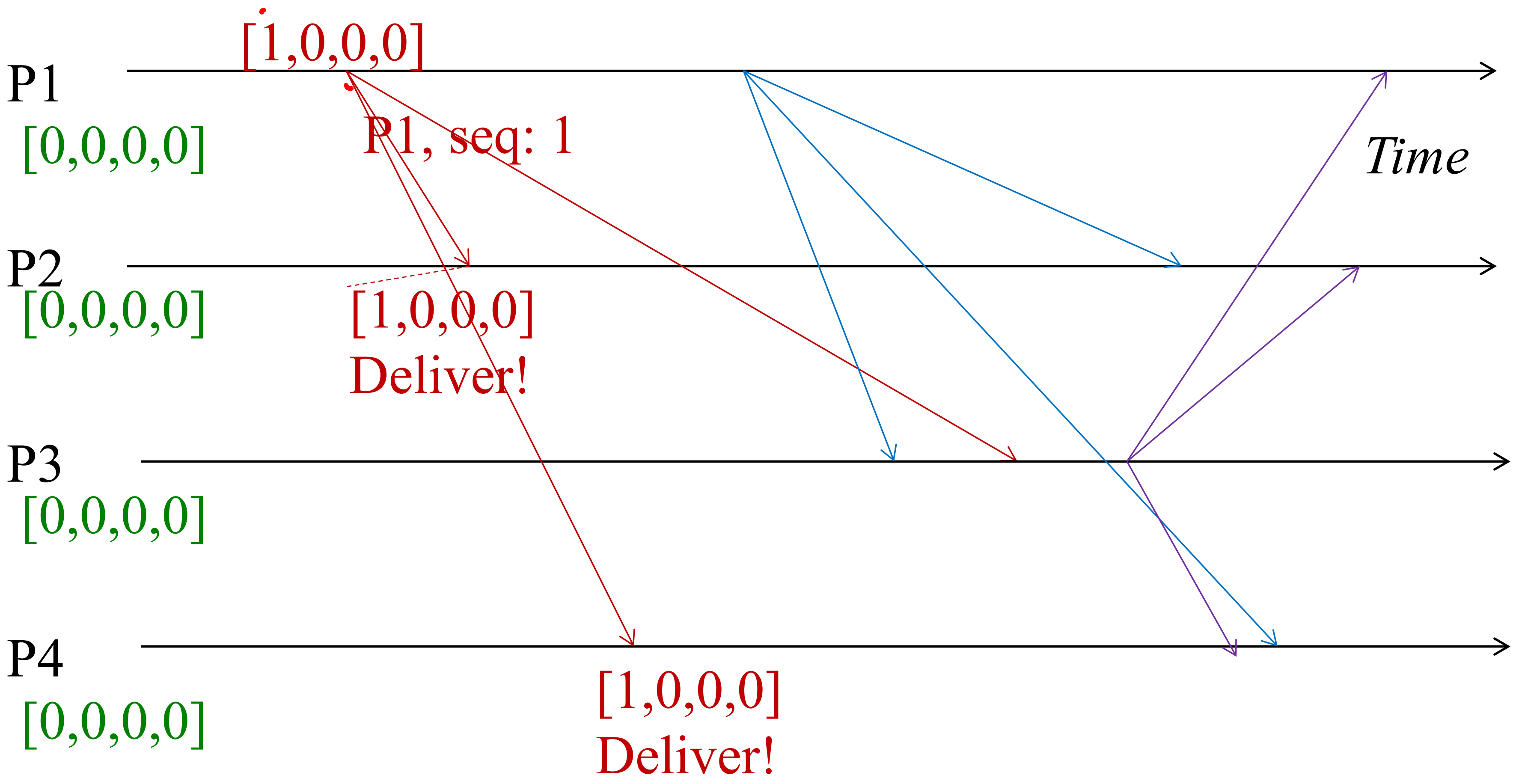
P4

[0,0,0,0]

Time

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]

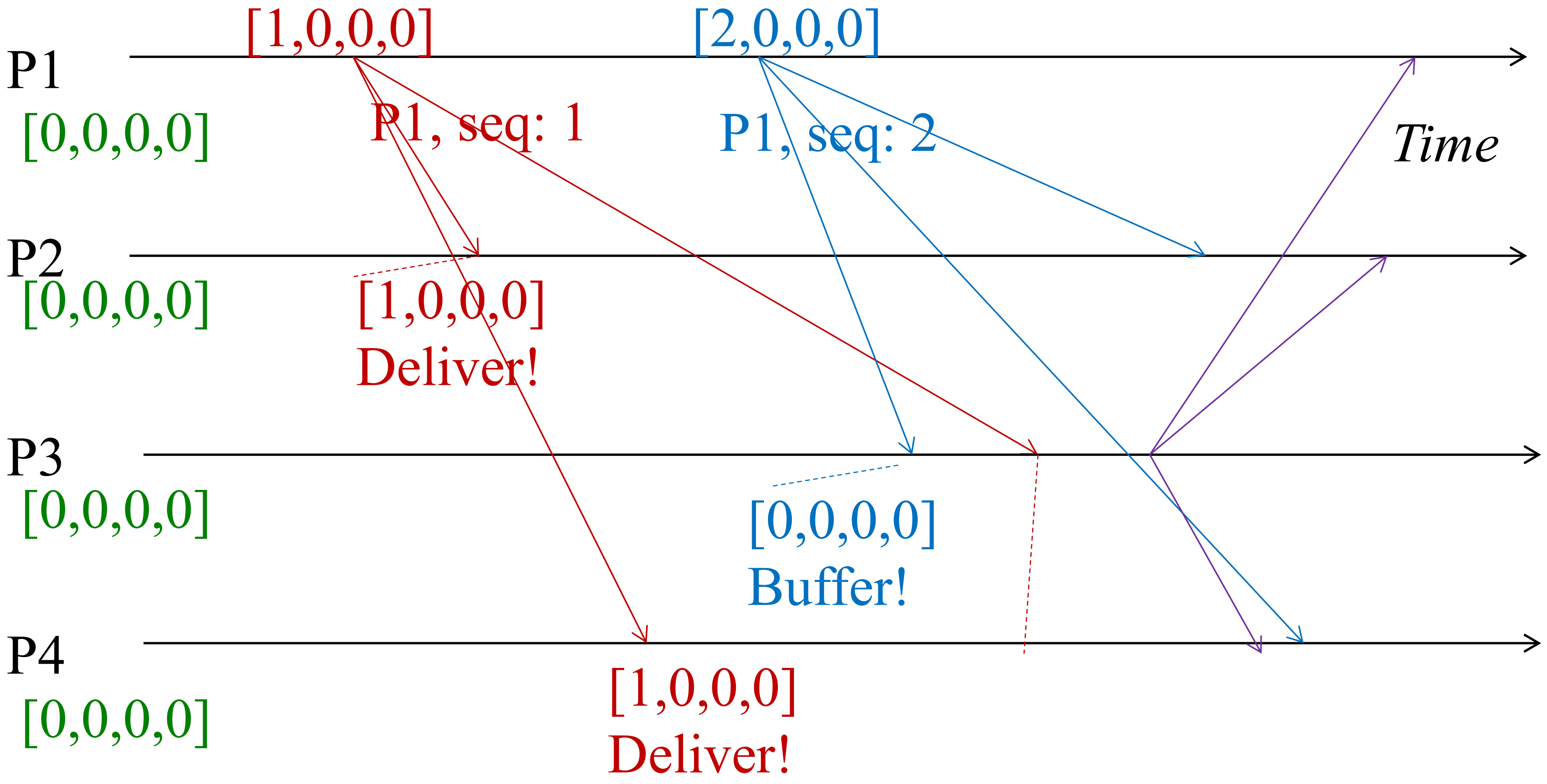
P4

[0,0,0,0]

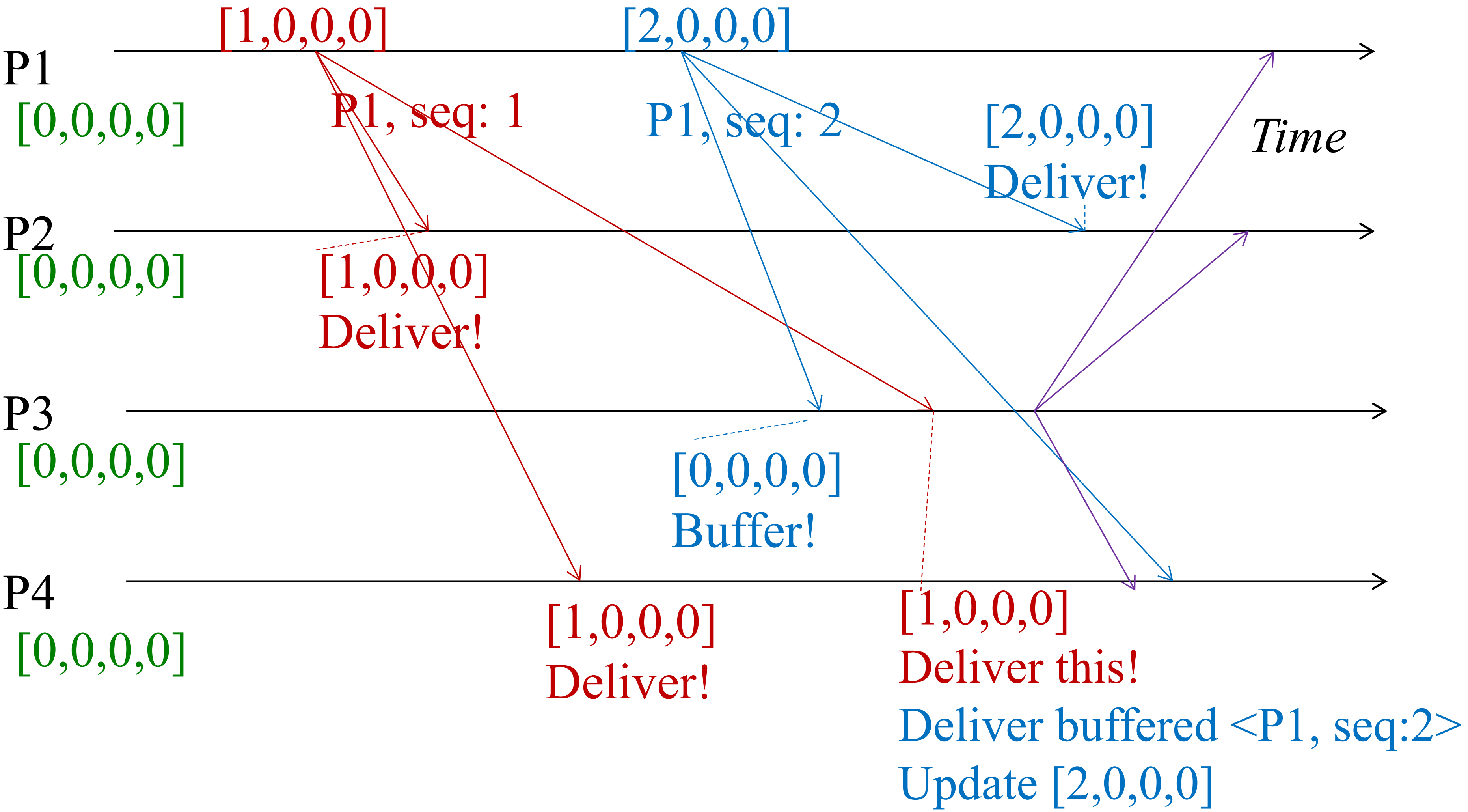
[1,0,0,0]

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]

Time

[1,0,0,0]

P1, seq: 1

[2,0,0,0]

P1, seq: 2

[1,0,0,0]
Deliver!

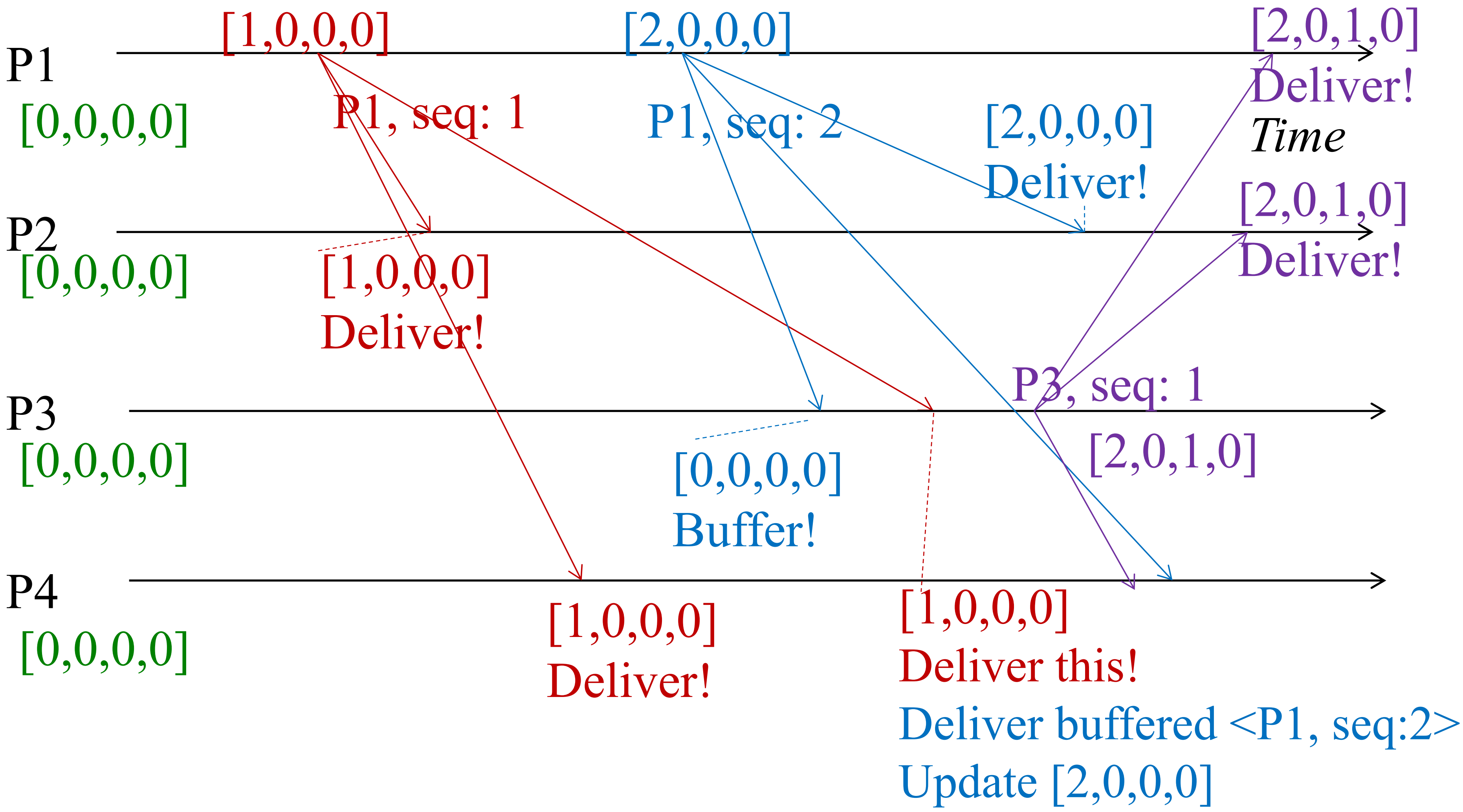
[2,0,0,0]
Deliver!

[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



[1,0,0,0]

[2,0,0,0]

[2,0,1,0]

Deliver!

P1

[0,0,0,0]

P1, seq: 1

P1, seq: 2

[2,0,0,0]

Deliver!

Time

P2

[0,0,0,0]

[1,0,0,0]

Deliver!

[2,0,1,0]

Deliver!

P3

[0,0,0,0]

P3, seq: 1

[2,0,1,0]

[0,0,0,0]

Buffer!

P4

[0,0,0,0]

[1,0,0,0]

Deliver!

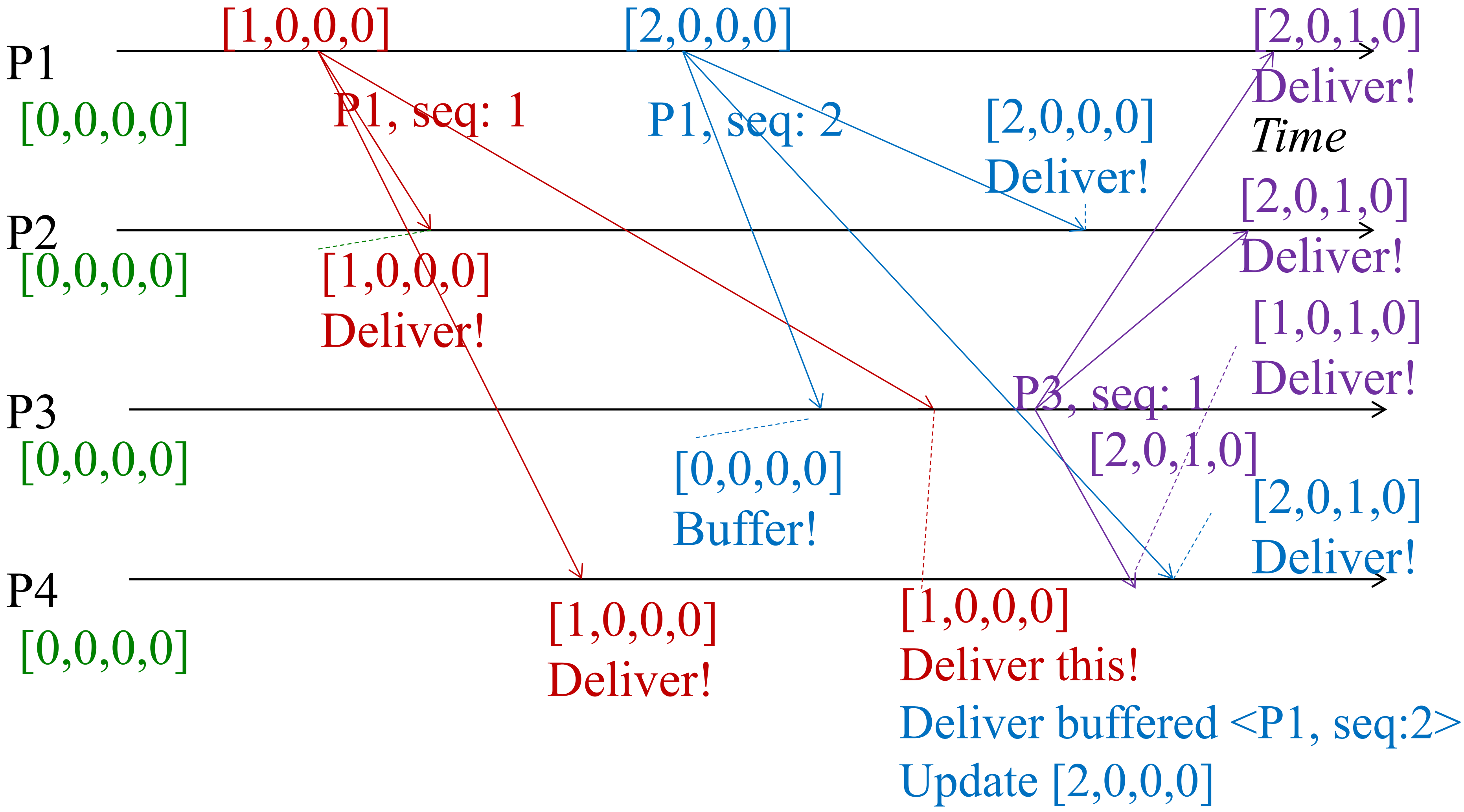
[1,0,0,0]

Deliver this!

Deliver buffered <P1, seq:2>

Update [2,0,0,0]

FIFO order multicast execution



P1
[0,0,0,0]

[1,0,0,0]
P1, seq: 1

[2,0,0,0]
P1, seq: 2

[2,0,0,0]
Deliver!

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

P2
[0,0,0,0]

[1,0,0,0]
Deliver!

[2,0,1,0]
Deliver!

P3
[0,0,0,0]

[0,0,0,0]
Buffer!

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

[1,0,1,0]
Deliver!

[2,0,1,0]
Deliver!

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]

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 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

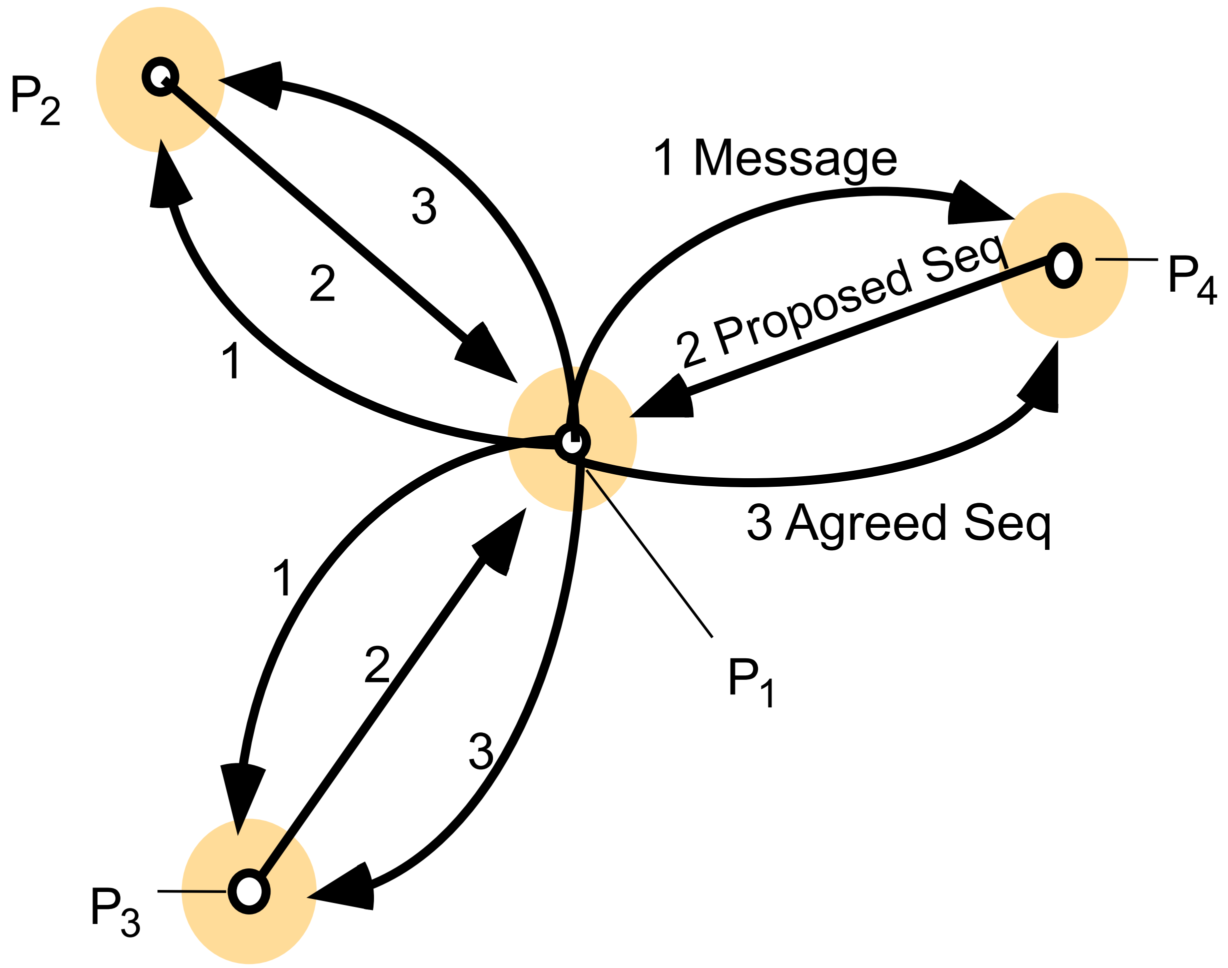
- Special process elected as leader or sequencer.
- TO-multicast(g,m) at P*i*:
  - Send multicast message m to group g *and the sequencer*
- Sequencer:
  - Maintains a global sequence number S (initially 0)
  - When a multicast message m is B-delivered to it:
    - sets S = S + 1, and B-multicast(g,{"order",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

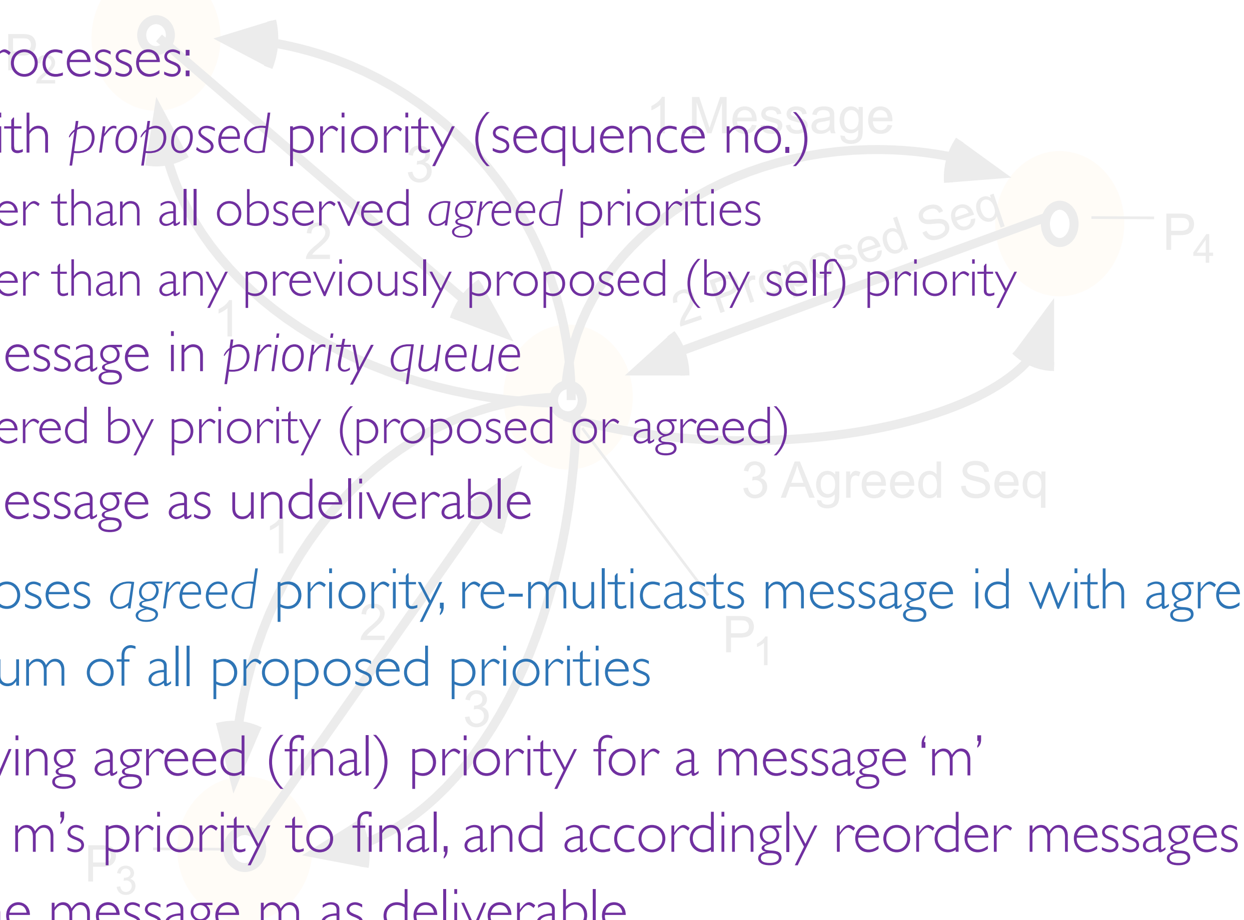
- Basic idea:
  - Same sequence number counter across different processes.
  - Instead of different sequence number counter for each process.

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

ISIS algorithm for total ordering



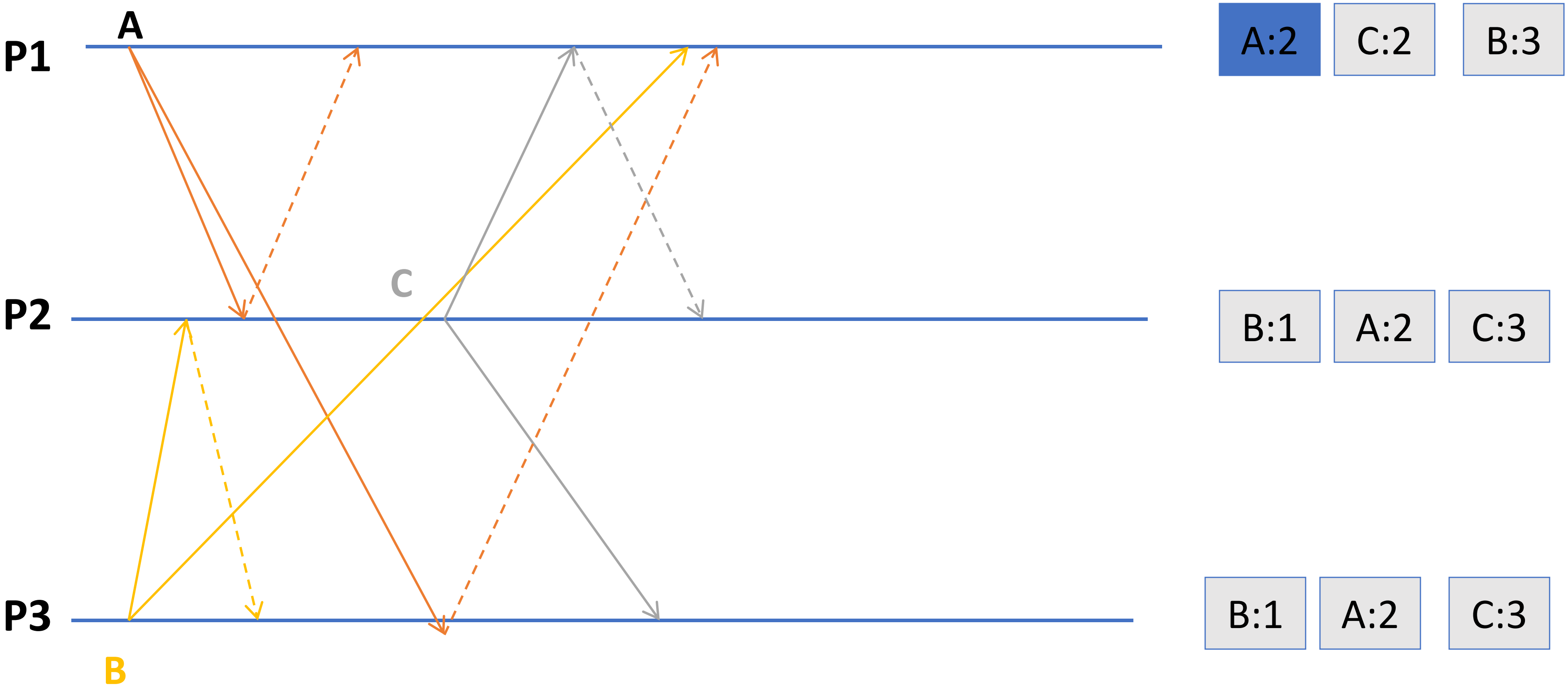
ISIS algorithm for total ordering



- 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 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.

- Will continue ISIS in next class.
- Additional slides provided for early reference.

Example: ISIS algorithm



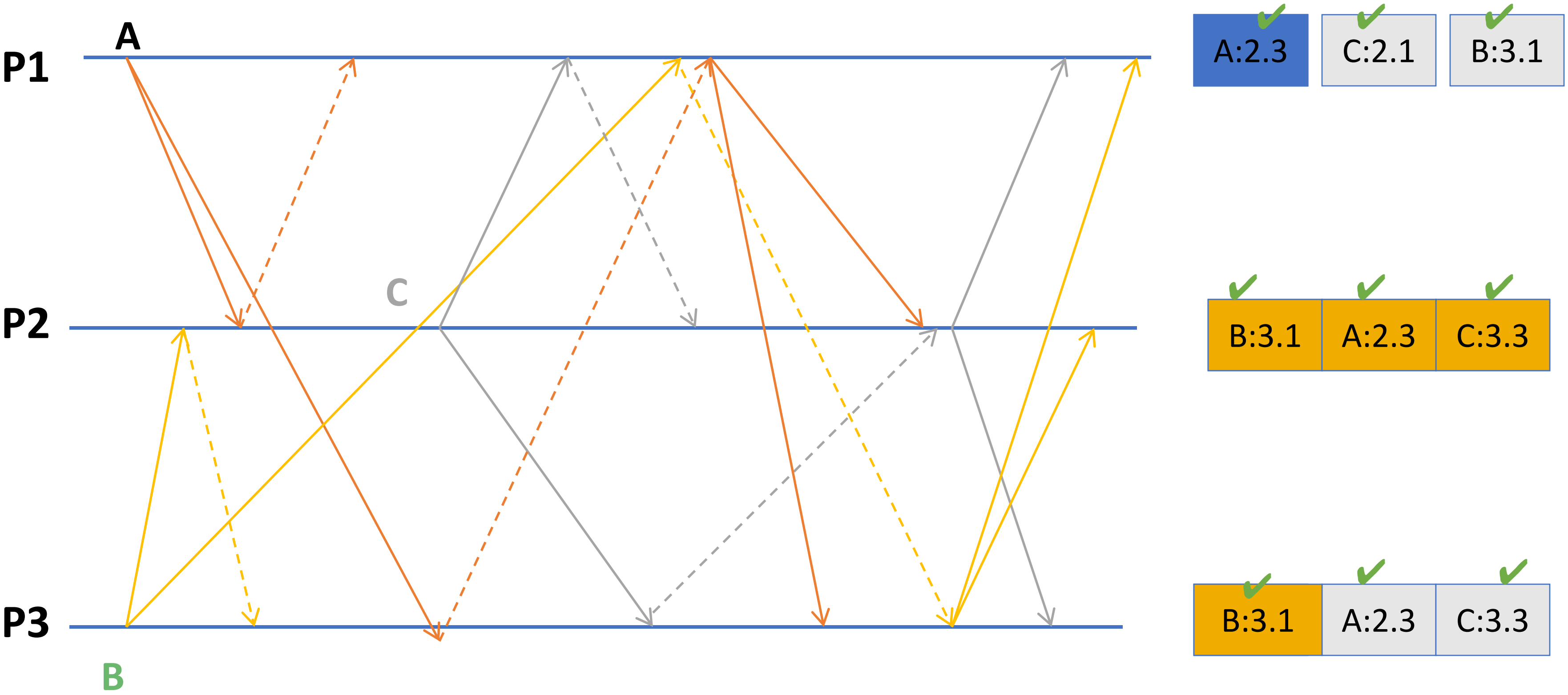
How do we break ties?

- 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



✓A:2.3 ✓C:2.1 ✓B:3.1

✓B:3.1 ✓A:2.3 ✓C:3.3

✓B:3.1 ✓A:2.3 ✓C:3.3

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*.

Implementing causal order multicast

Next class!

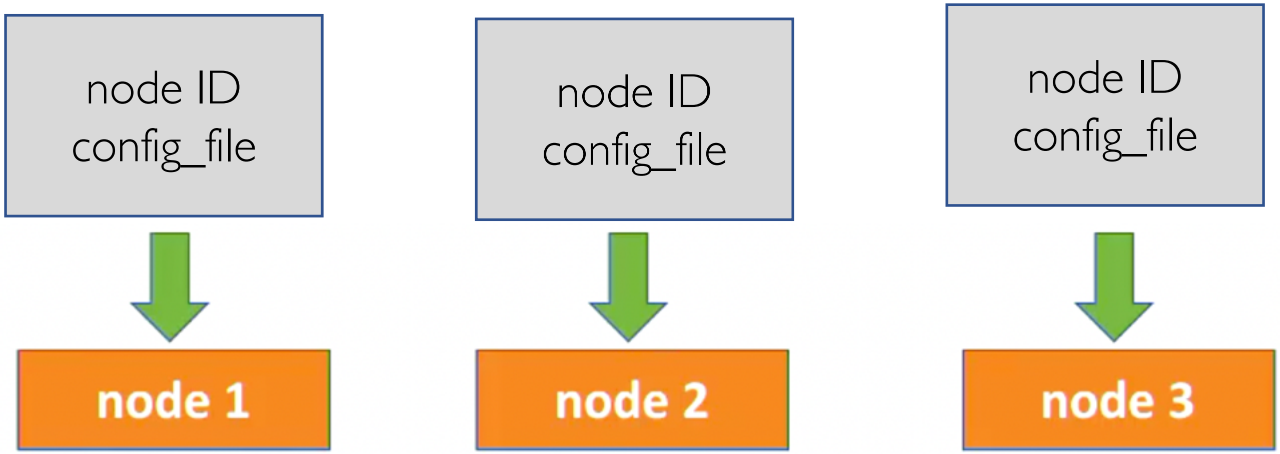
MP1: Event Ordering

- https://courses.grainger.illinois.edu/ece428/sp2025/mps/mp1.html
- Lead TA: Neel Dani

- 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

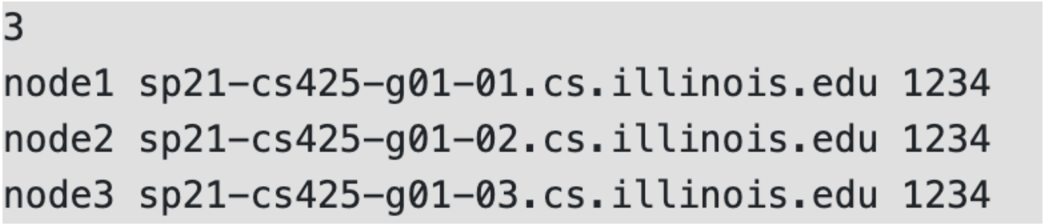
node ID
config_file

node 1

node 2

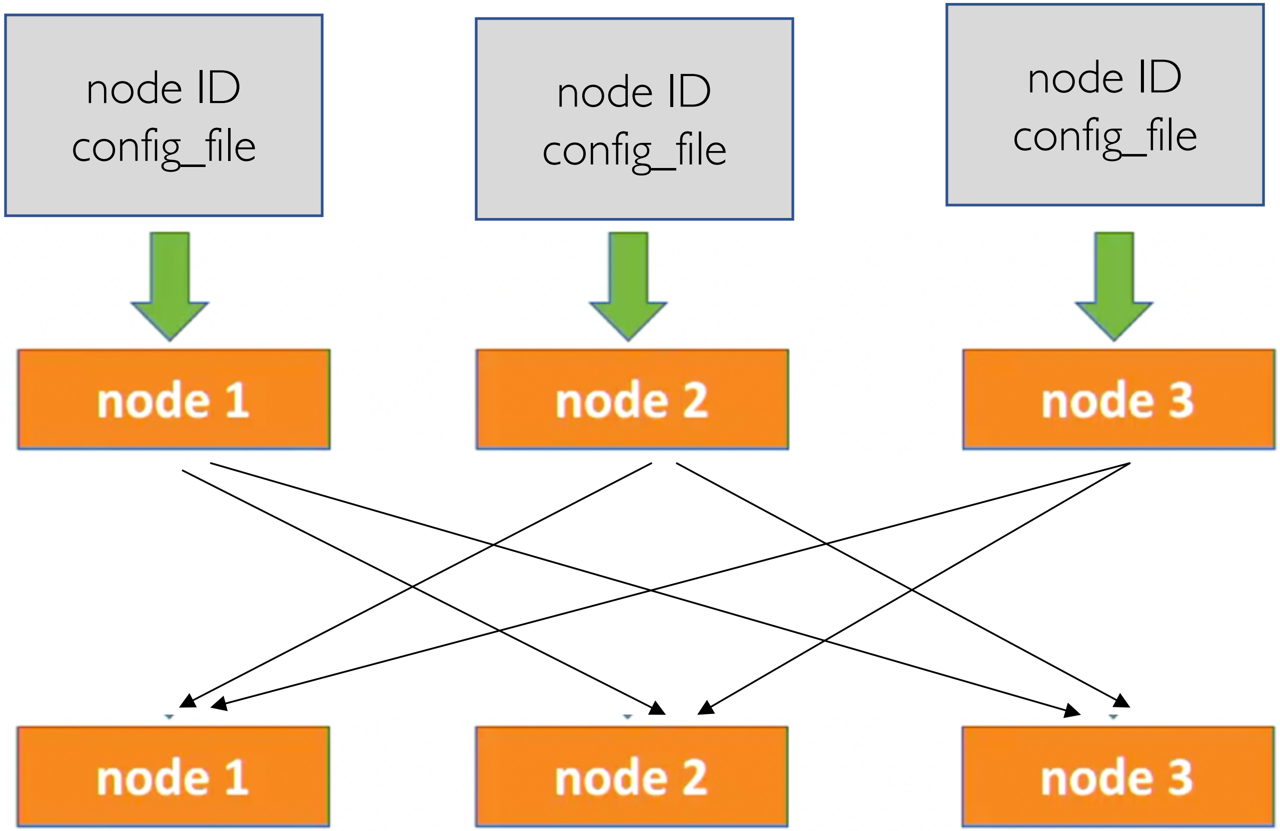
node 3

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

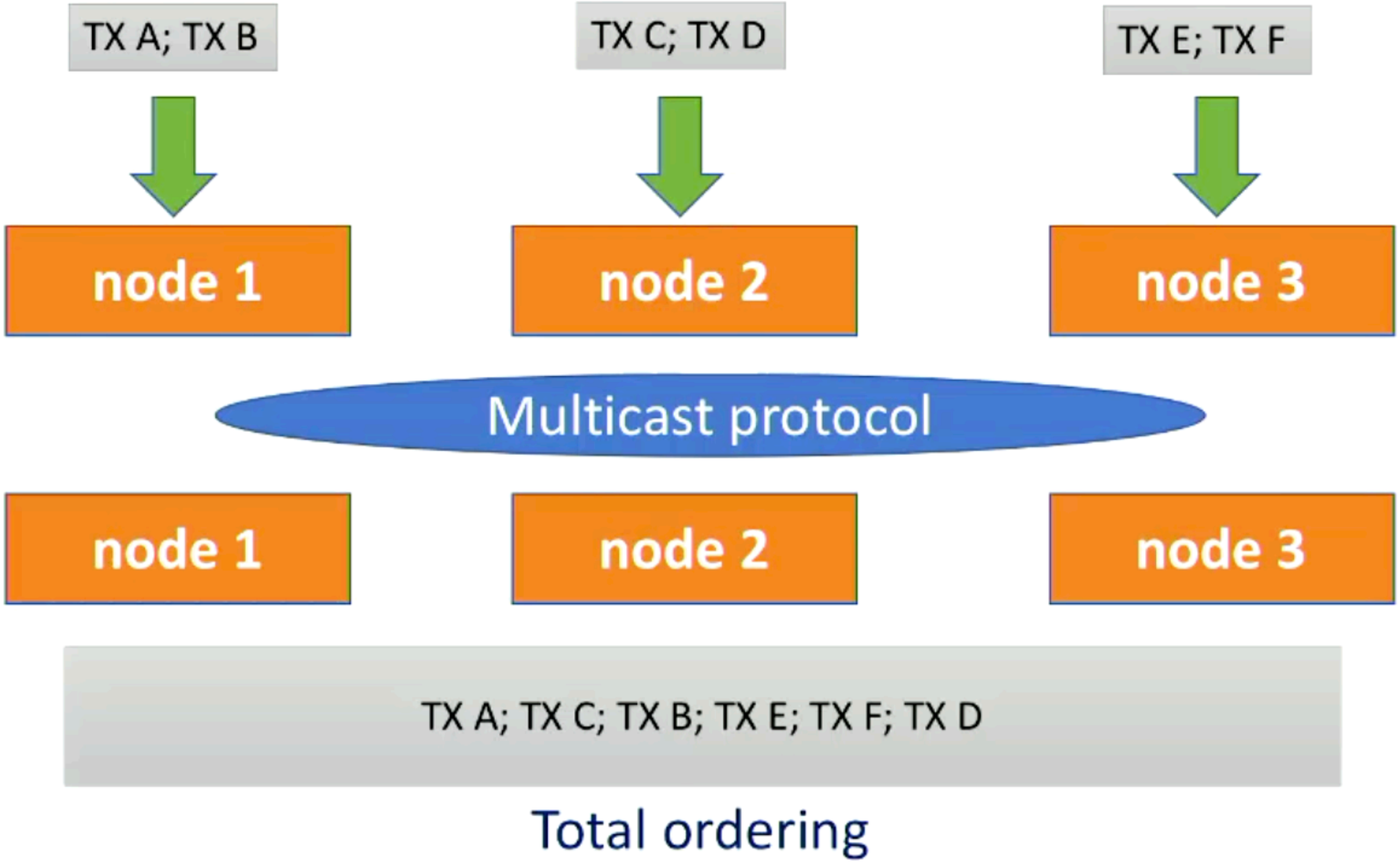


```
3
node1 sp21-cs425-g01-01.cs.illinois.edu 1234
node2 sp21-cs425-g01-02.cs.illinois.edu 1234
node3 sp21-cs425-g01-03.cs.illinois.edu 1234
```

MPI Architecture Setup



MPI 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

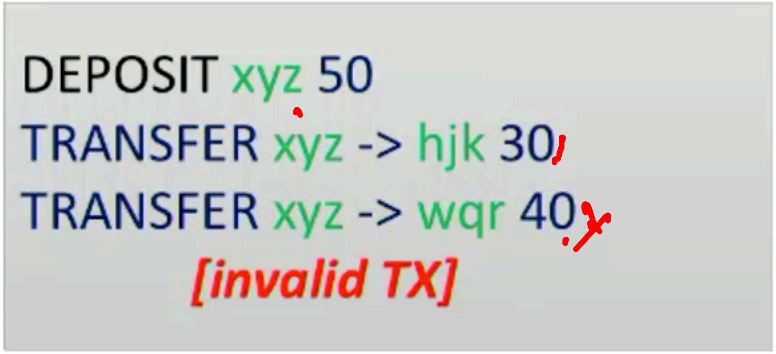
Total ordering

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]



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

BALANCES xyz:10 wqr:40

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 March 14th.
    - 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!!