Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements for some of materials: Indy Gupta and Nikita Borisov

Logistics

- MP0 is due today at 11:59pm.

- Please make sure you are on CampusWire
  - Reach out to Sarthak (sm106) if you need access.

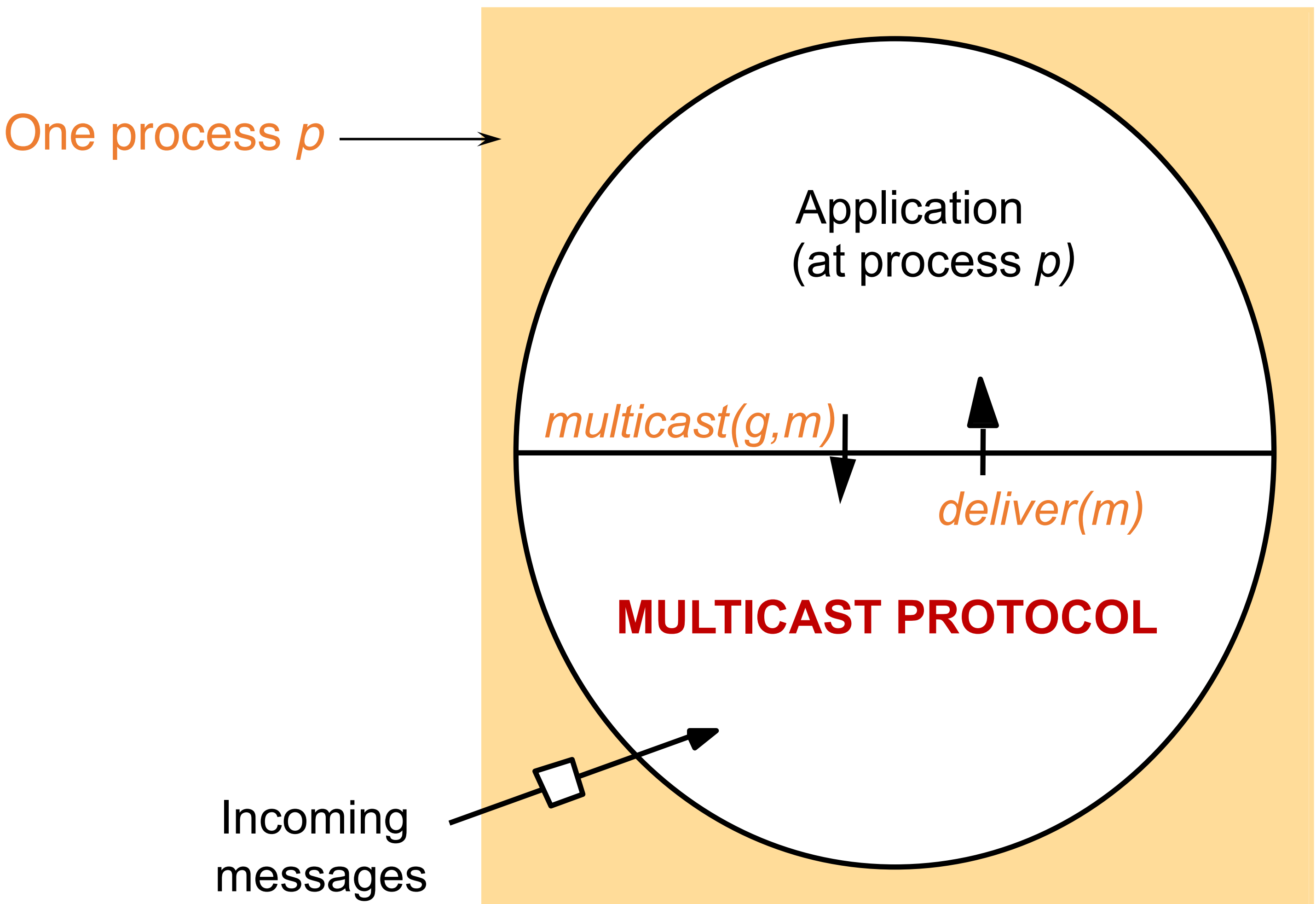
- Reminder to share your name when you speak up in class.

- Note about exams on CampusWire:
  - Midterm 1 (Feb 27-29), Midterm 2 (April 2-4), Finals (May 2-6).
  - Reservation via PrairieTest.
    - You can reserve a slot for Midterm 1 starting Feb 15.
  - If you need DRES accommodations, please upload your Letter of Accommodations on the CBTF website.

Today's agenda

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

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):

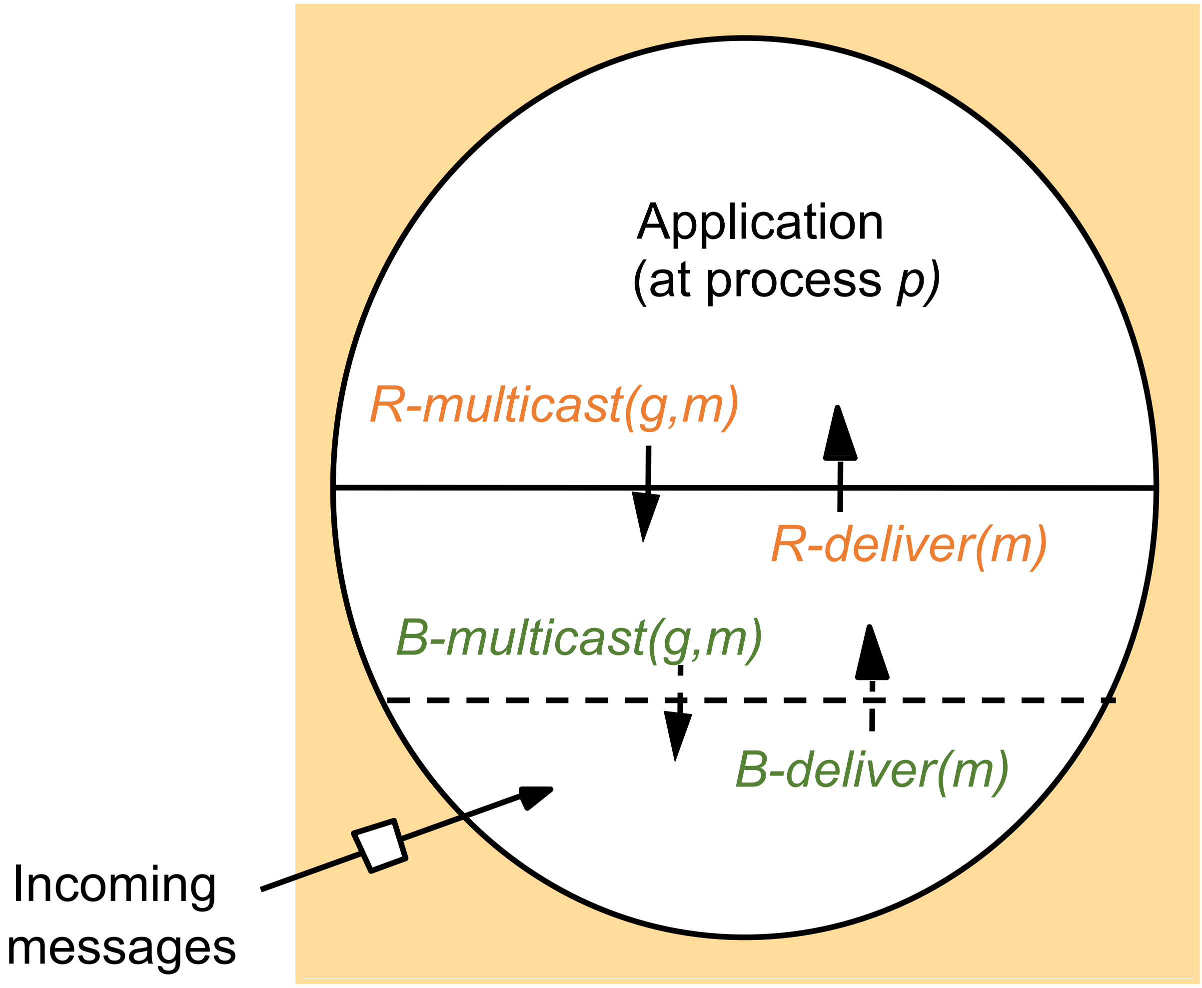
    for each process p in g, send (p,m).

    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* 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.

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)

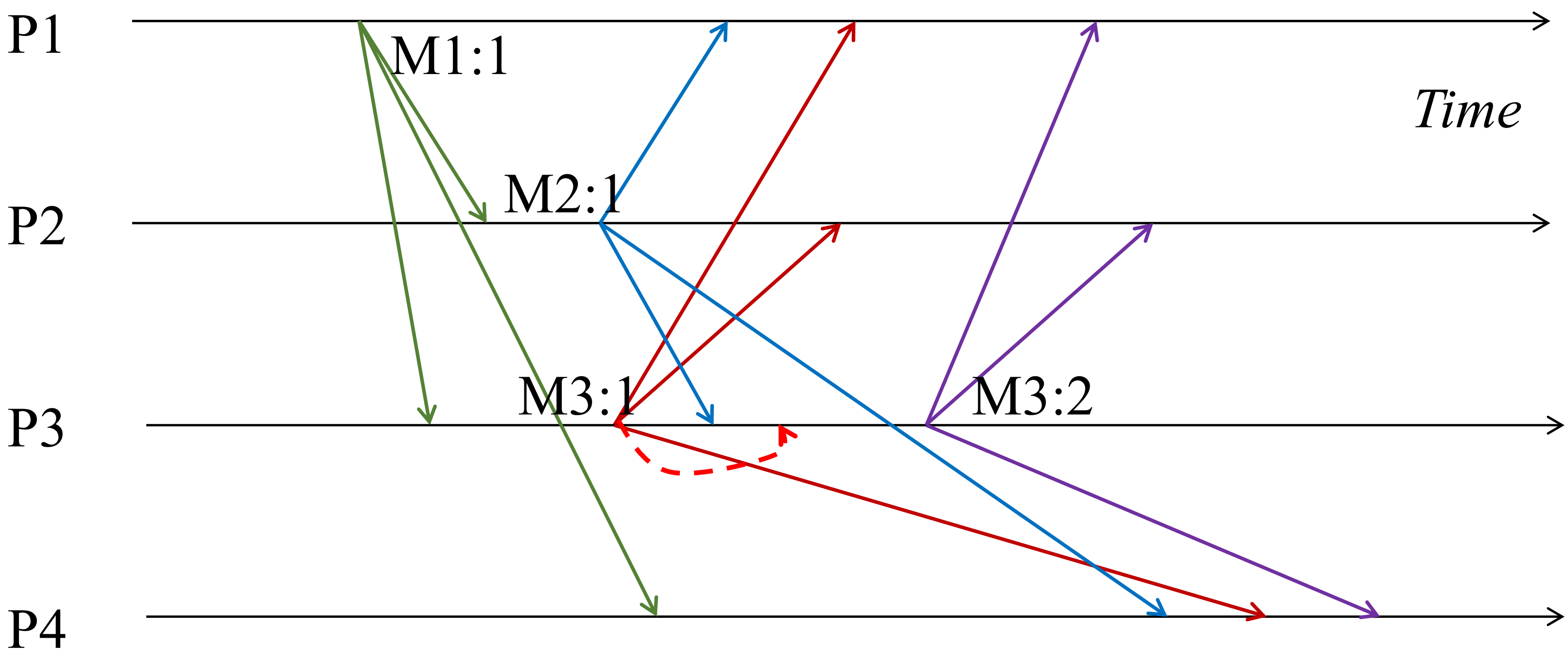
Ordered Multicast

- **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**:

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.

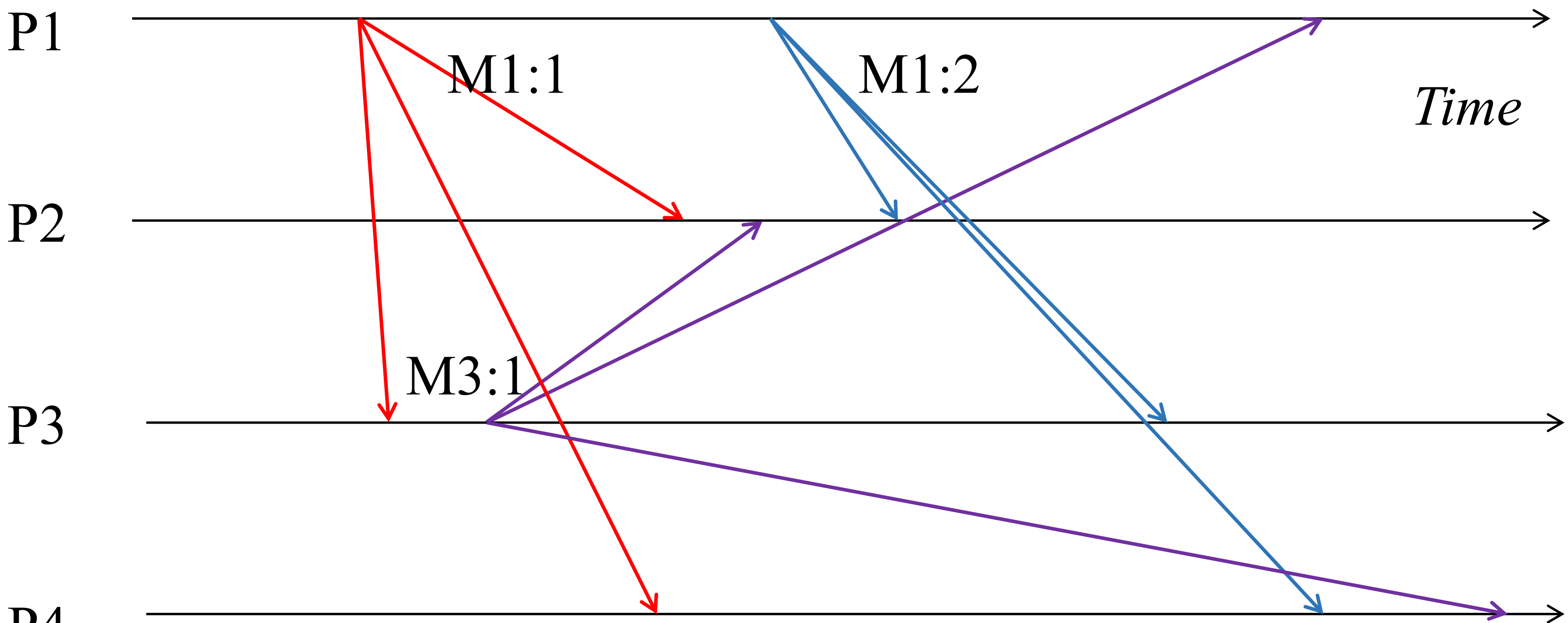
Causal vs Total

- 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.

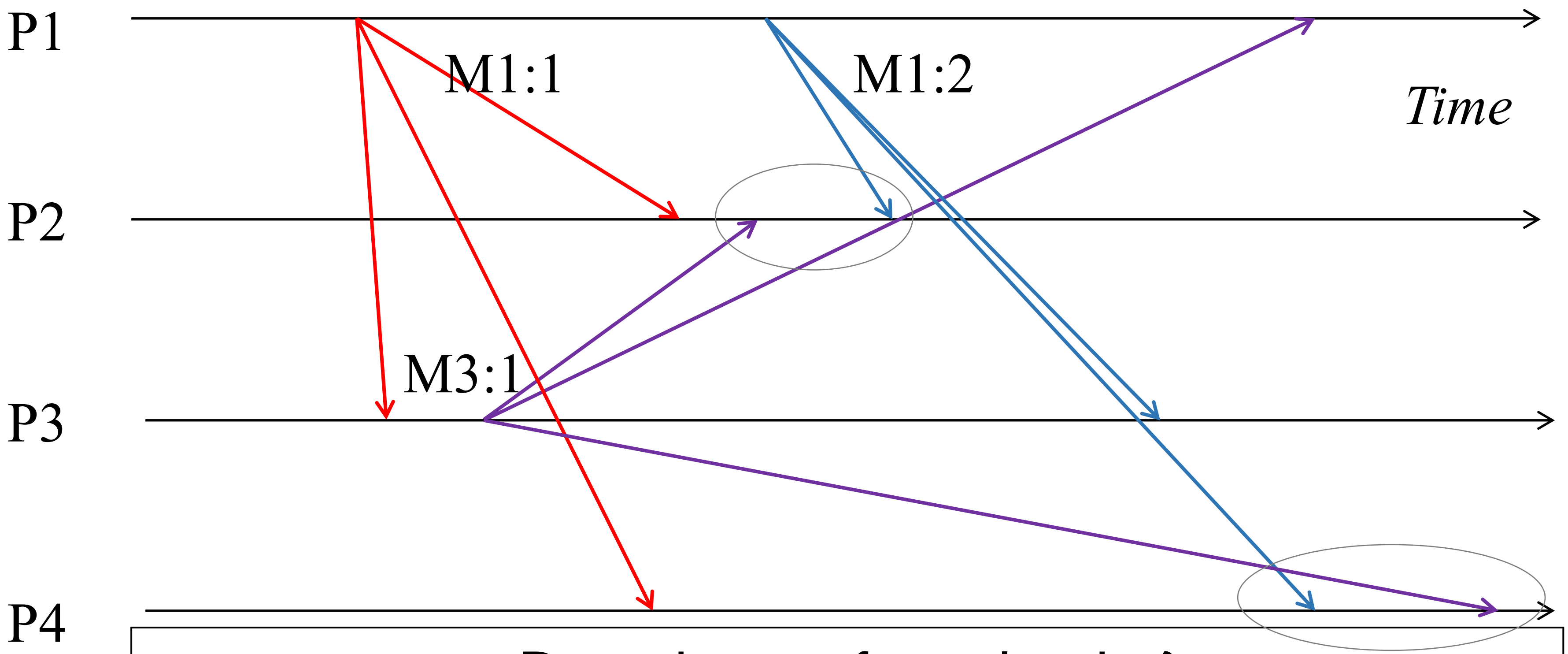
Example



Does this satisfy causal (and FIFO) order?

Yes

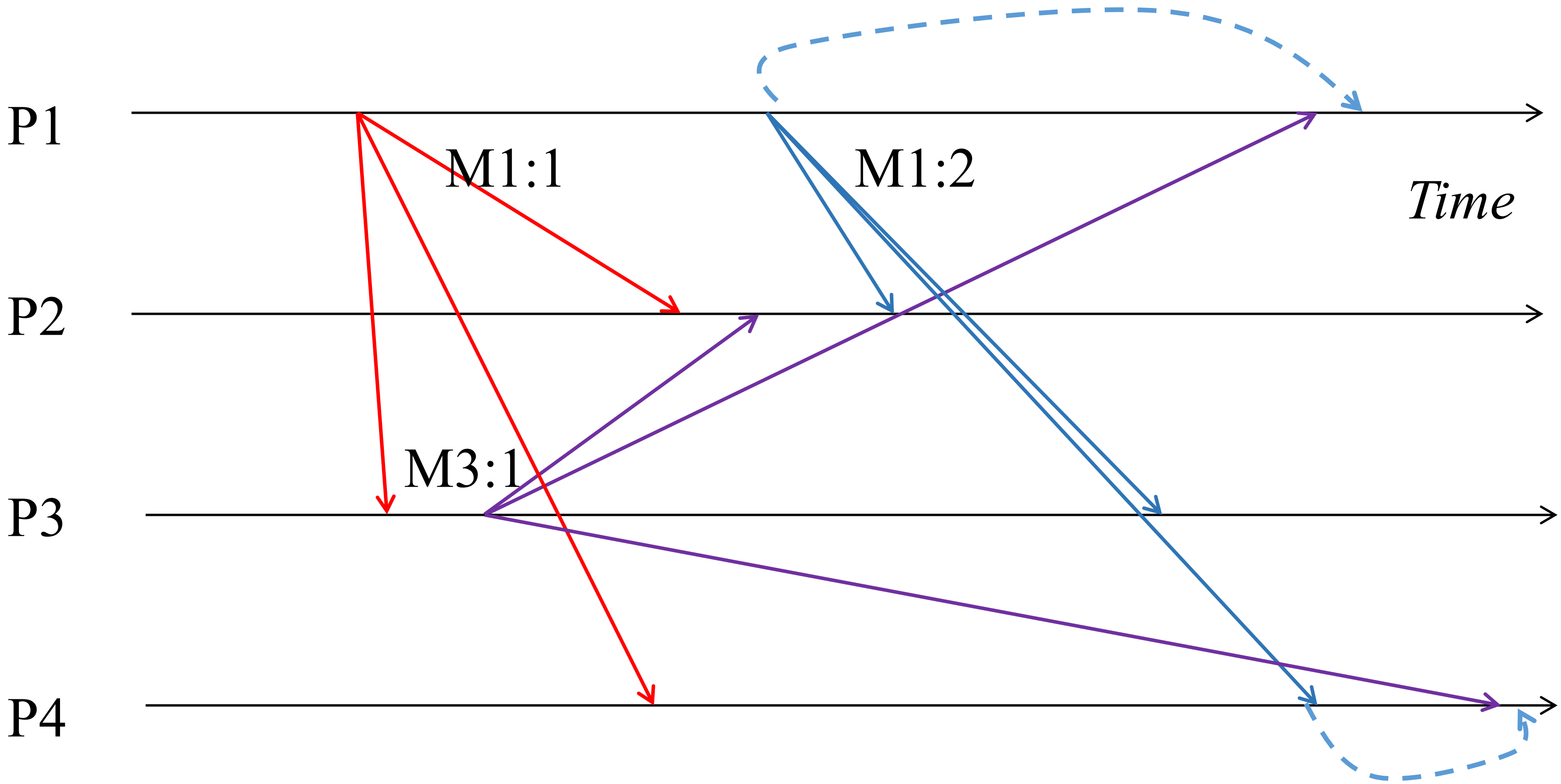
Example



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*.

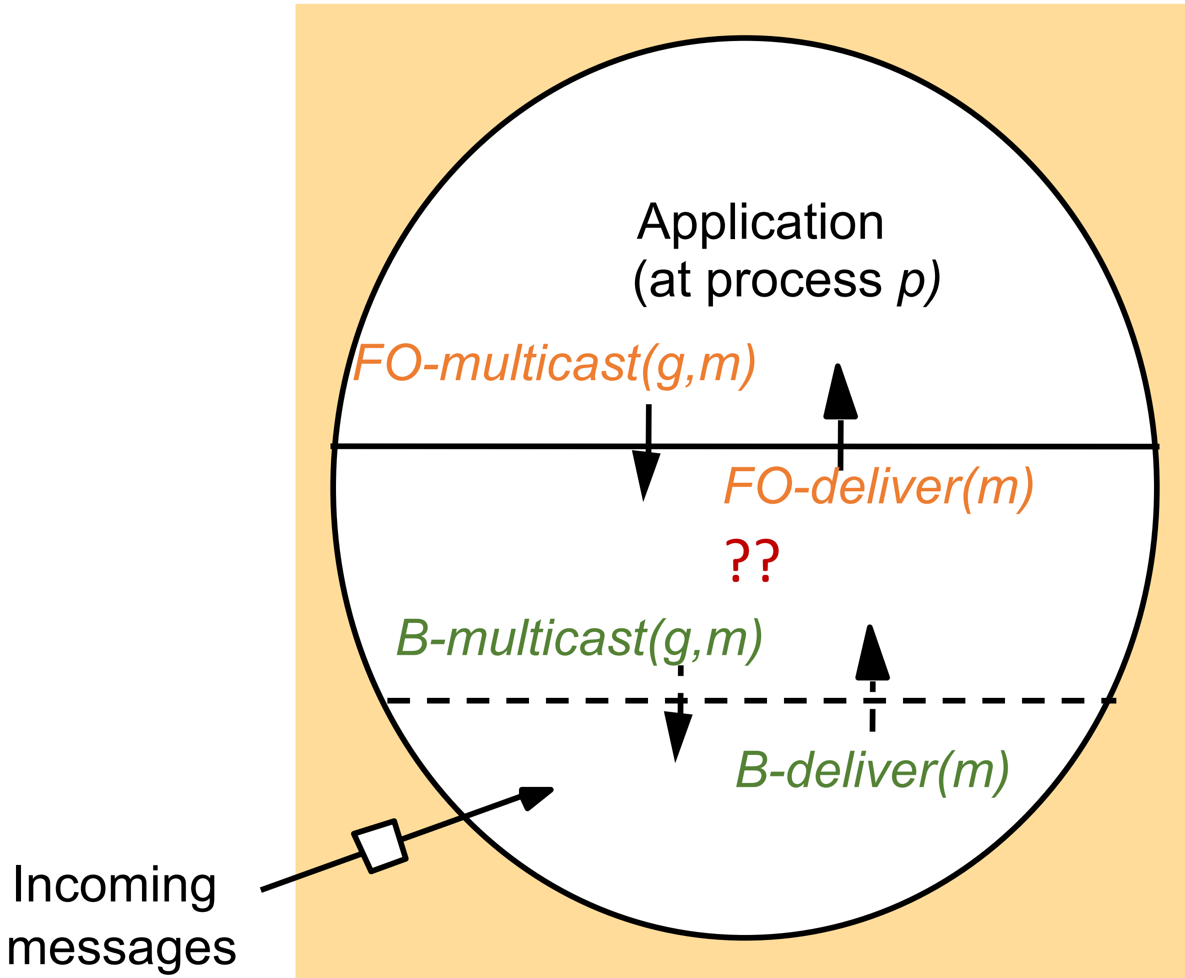
Next Question

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

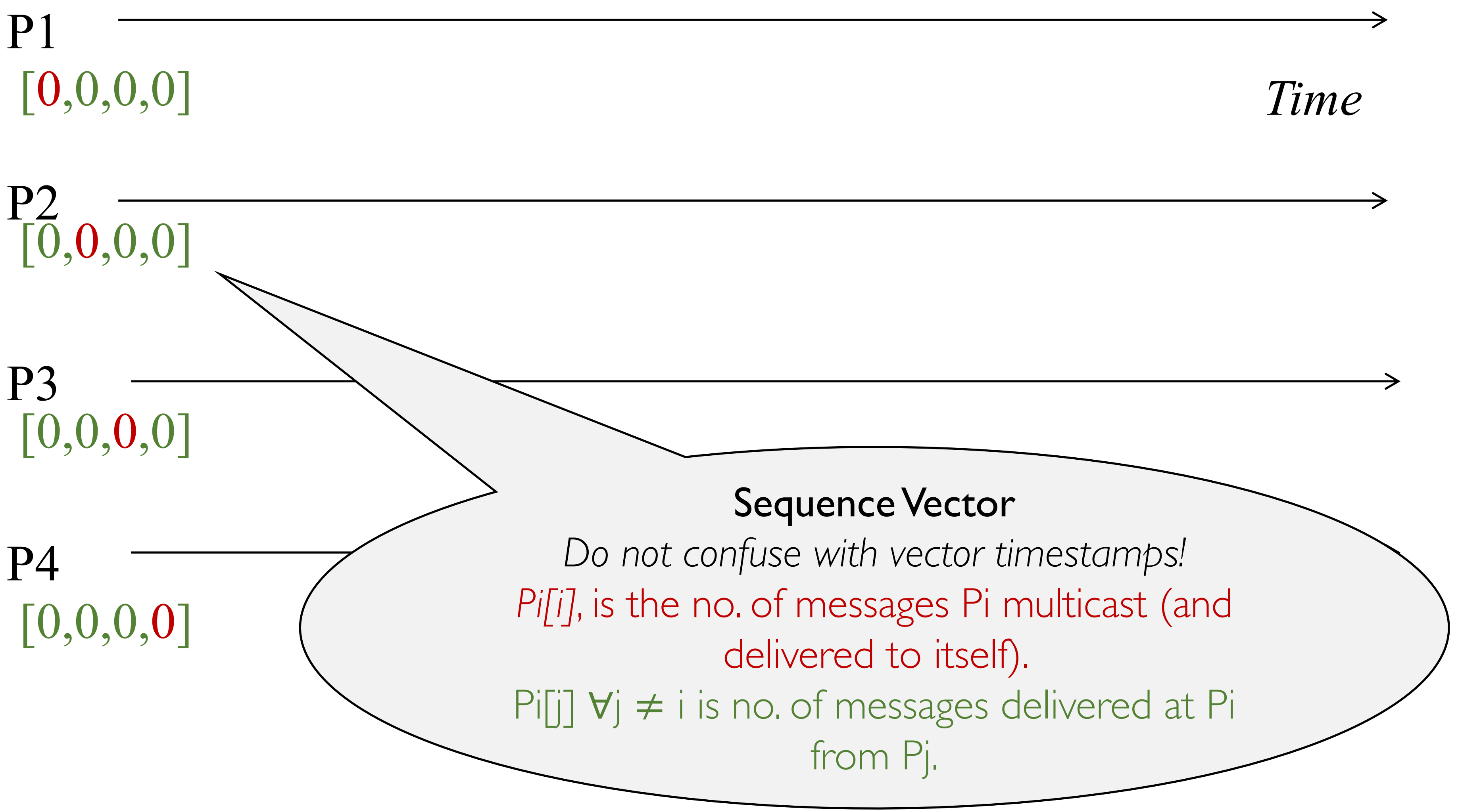
  FO-deliver(m) to application

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

  else buffer this multicast until above condition is true

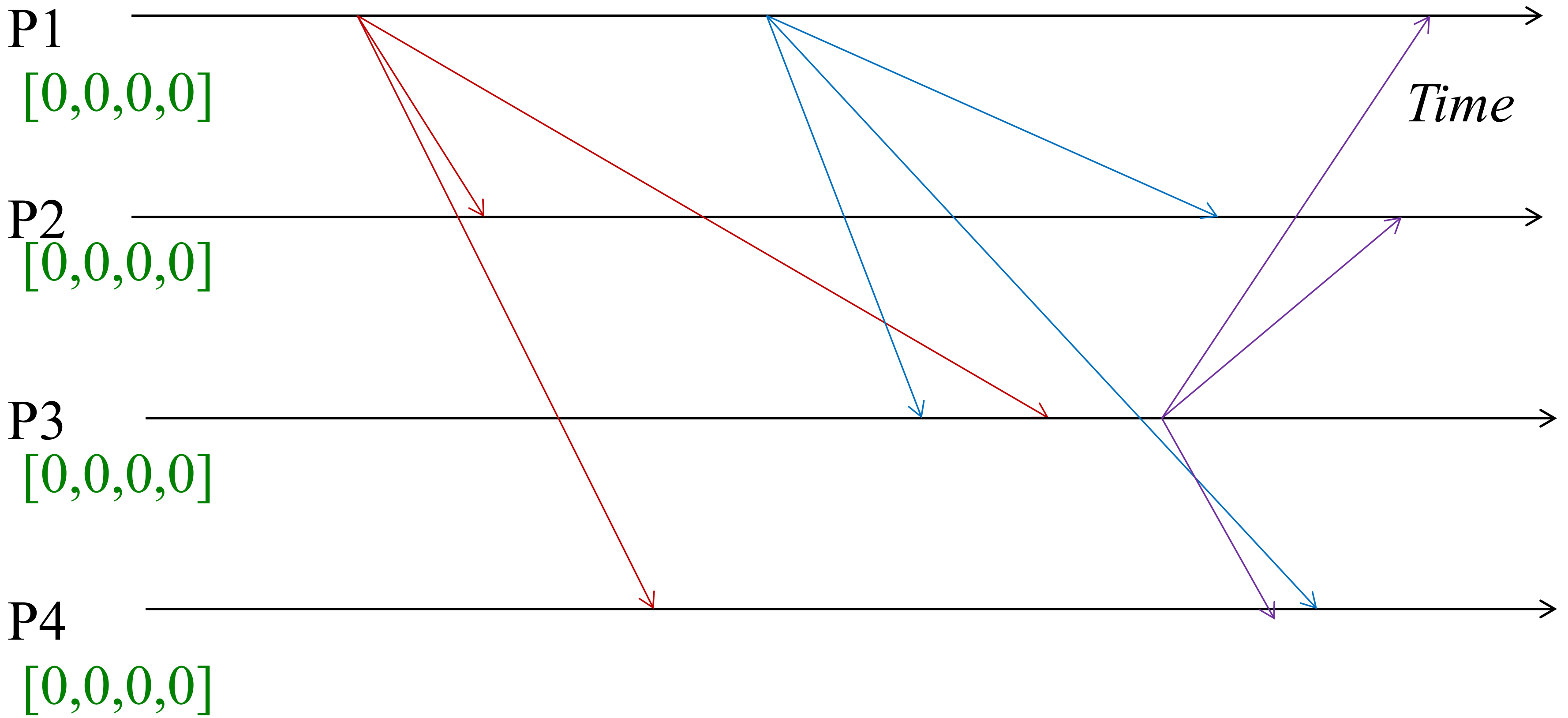
FIFO order multicast execution

FIFO order multicast execution



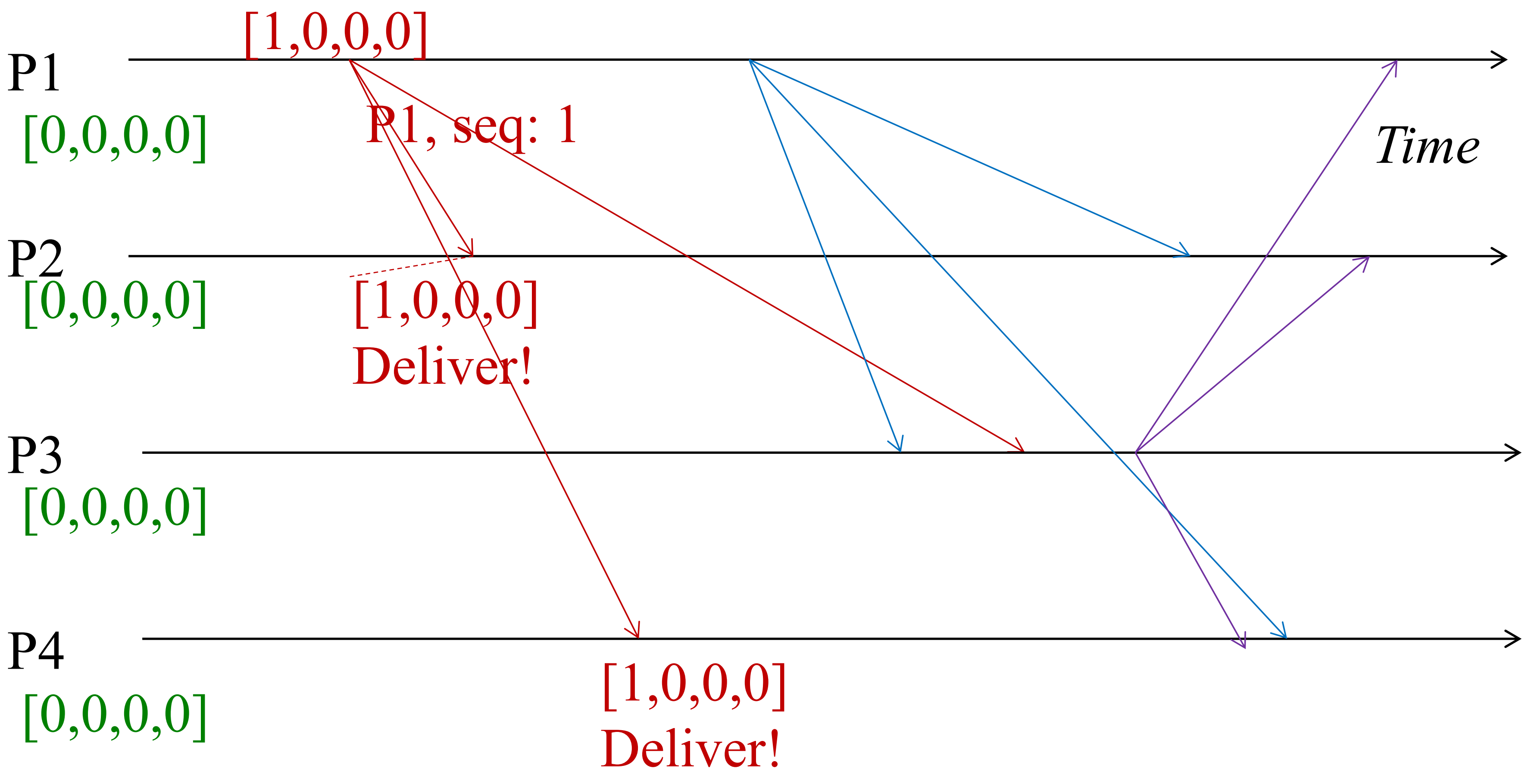
FIFO order multicast execution

FIFO order multicast execution

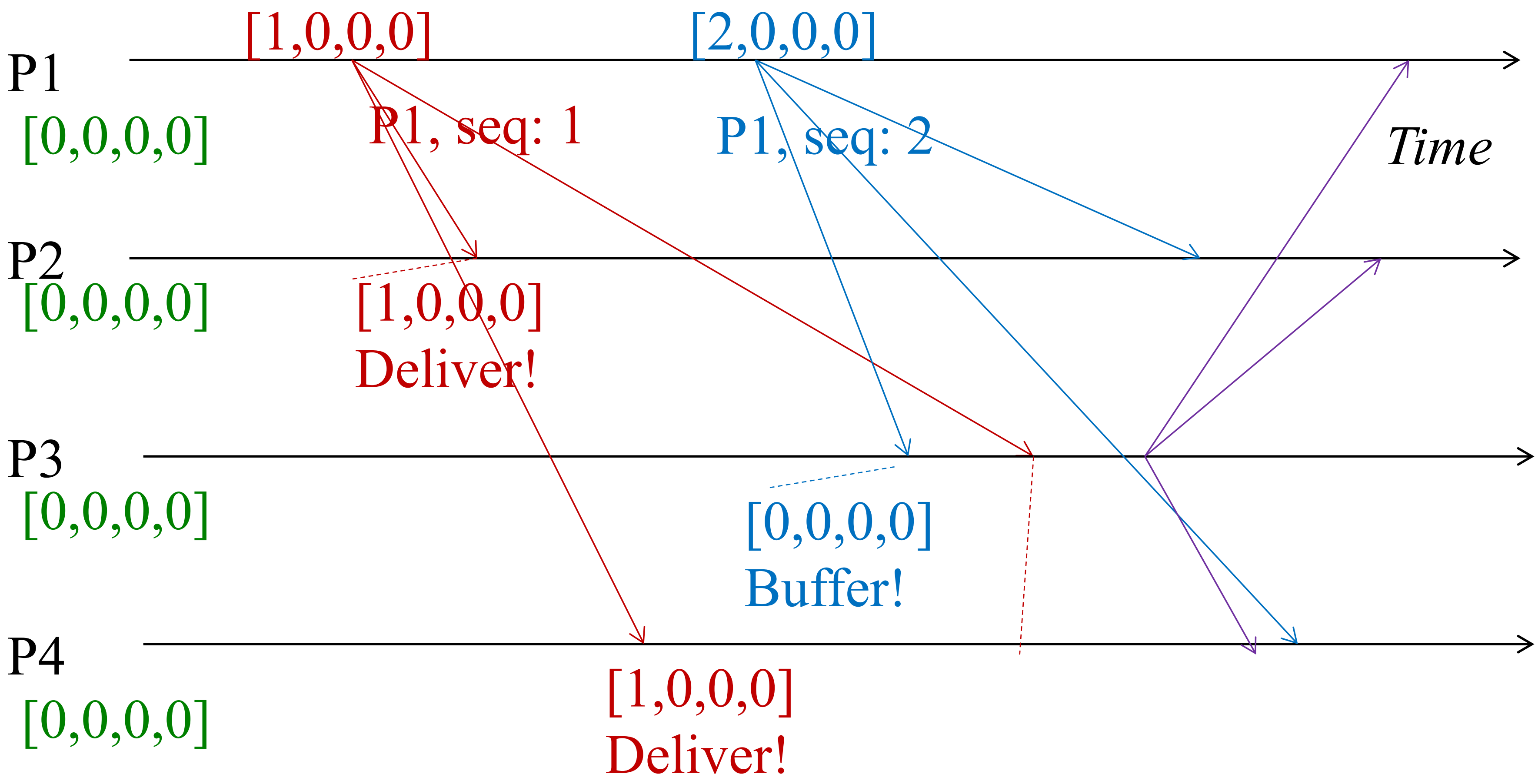


Self-deliveries omitted for simplicity.

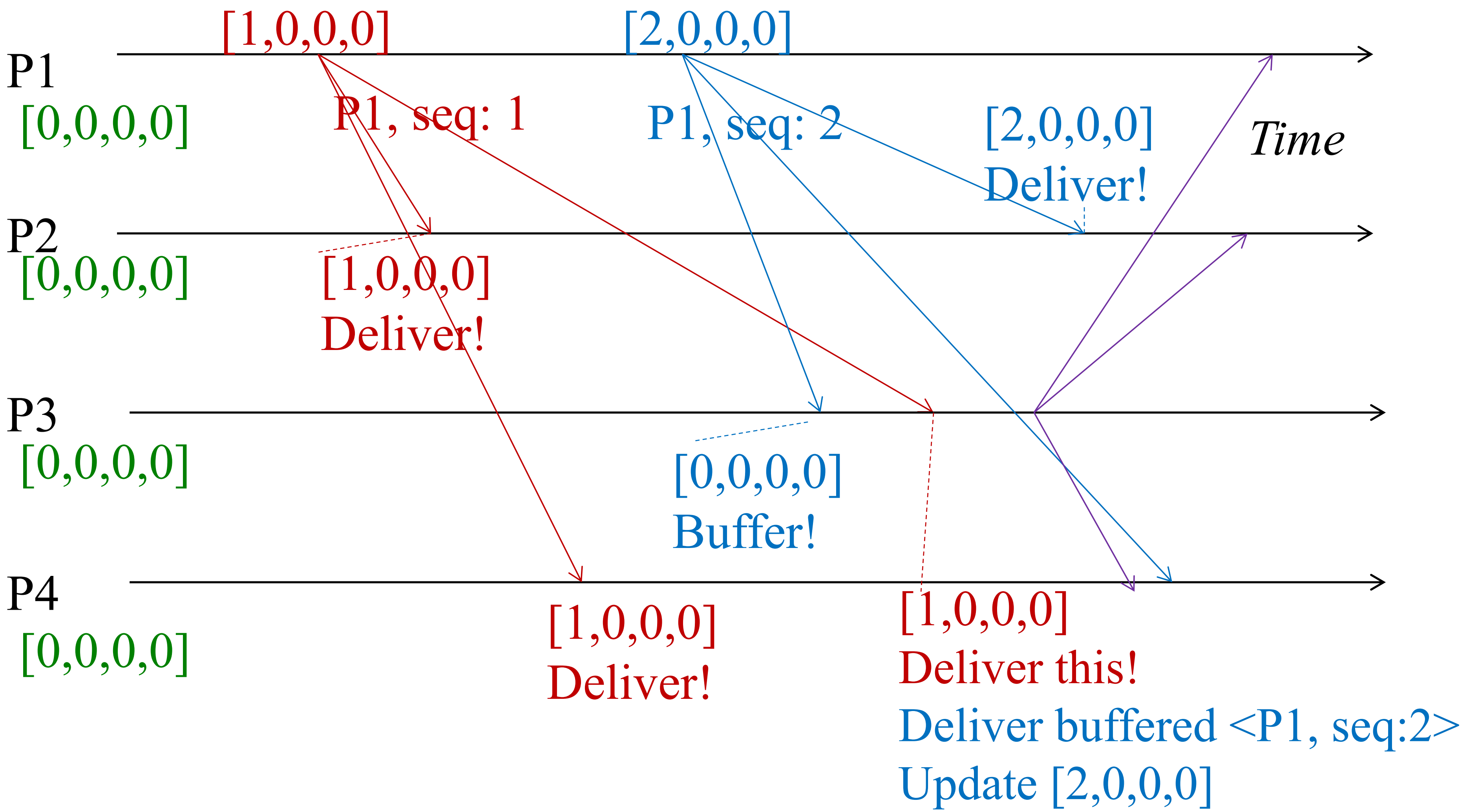
FIFO order multicast execution



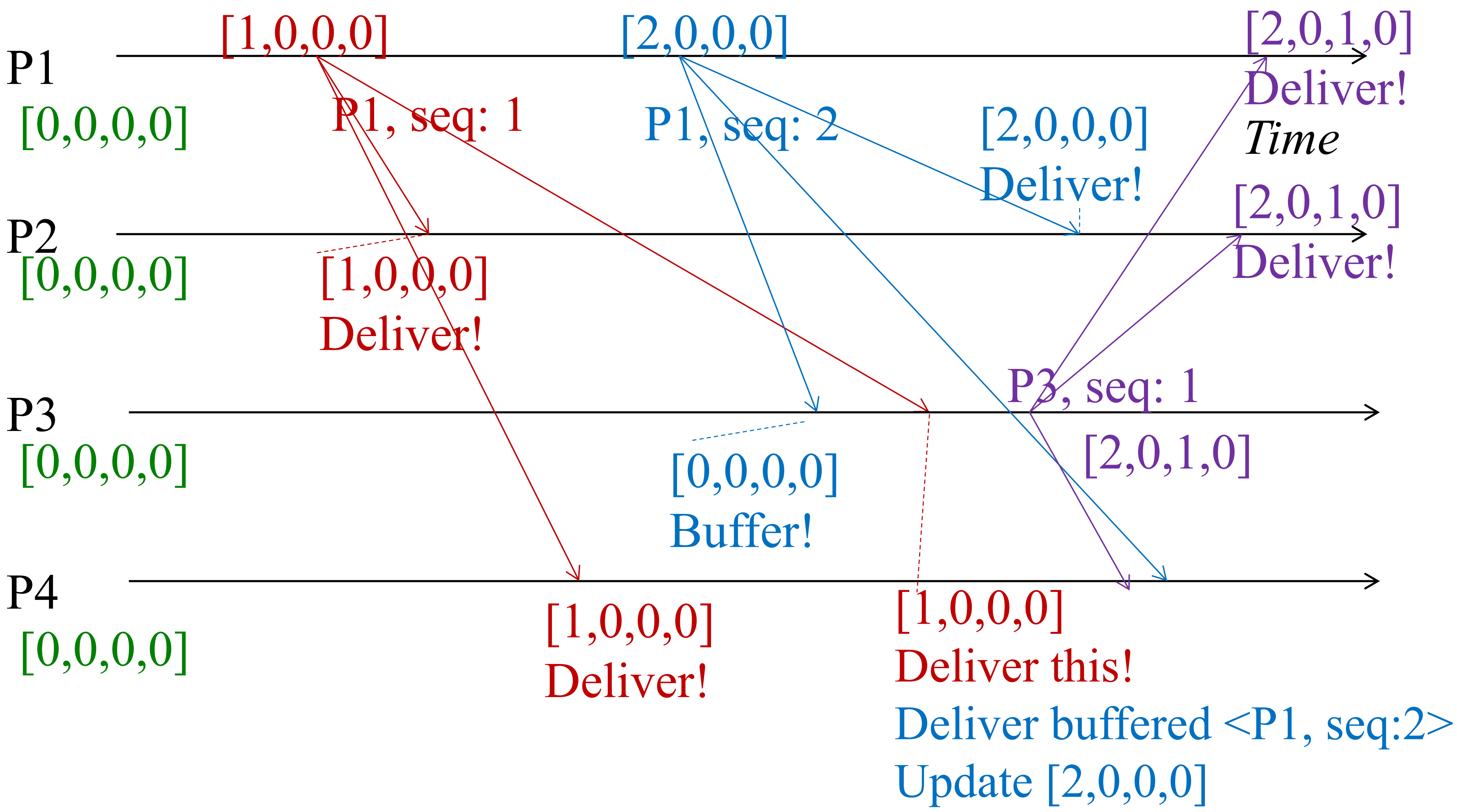
FIFO order multicast execution



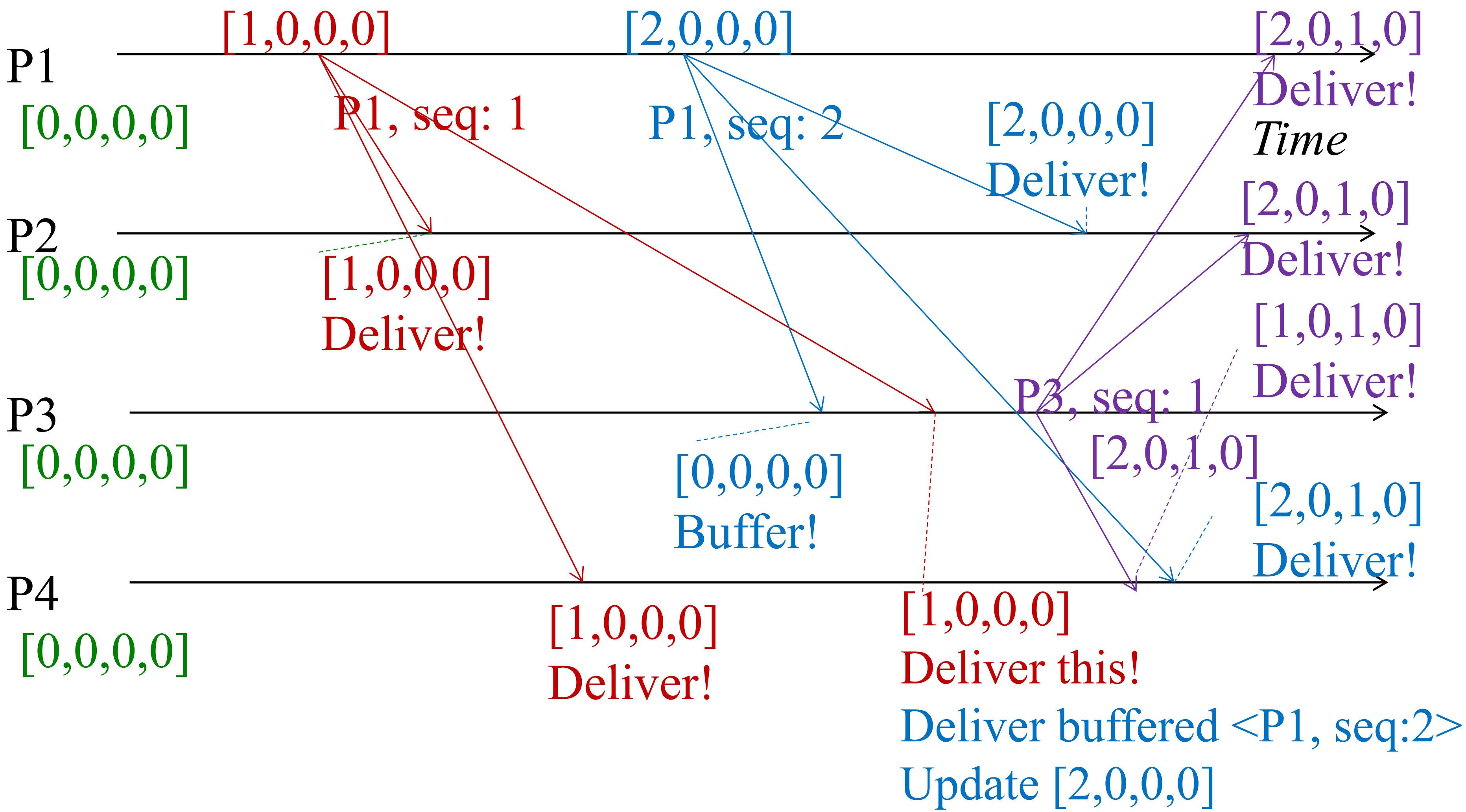
FIFO order multicast execution



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*.

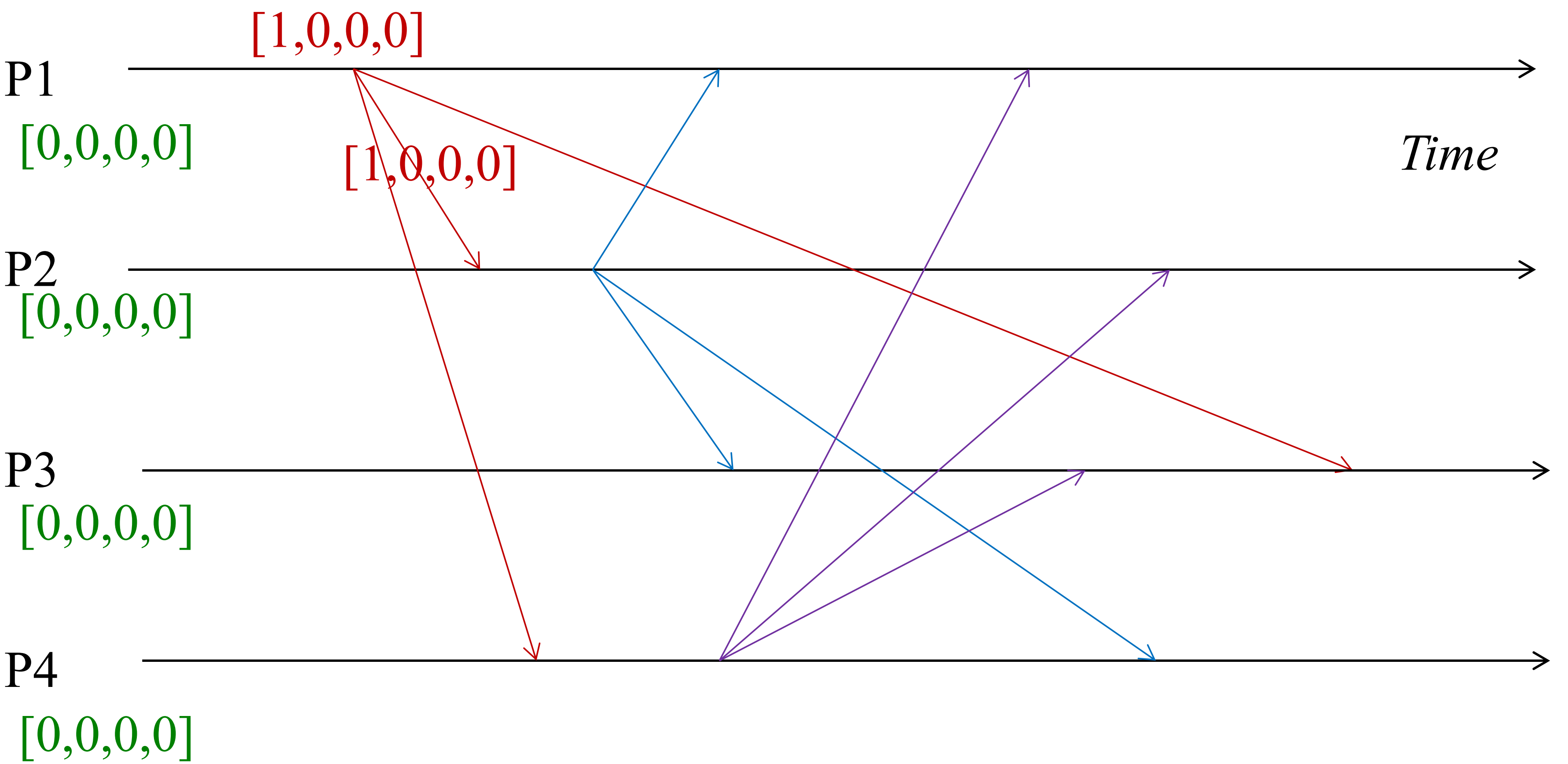
- Ignores other network messages. Only looks at multicast messages delivered to the application.

Implementing causal order multicast

- *CO-multicast(g,m) at Pj:*
  set $Pj[j] = Pj[j] + 1$
  piggyback entire vector Pj[1…N] with m as its sequence no.
  B-multicast(g,{m, Pj[1…N]})
- *On B-deliver({m, V[1..N]}) at Pi from Pj:* If Pi receives a multicast from Pj with sequence vector V[1…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 $k \neq j$: $V[k] \leq Pi[k]$

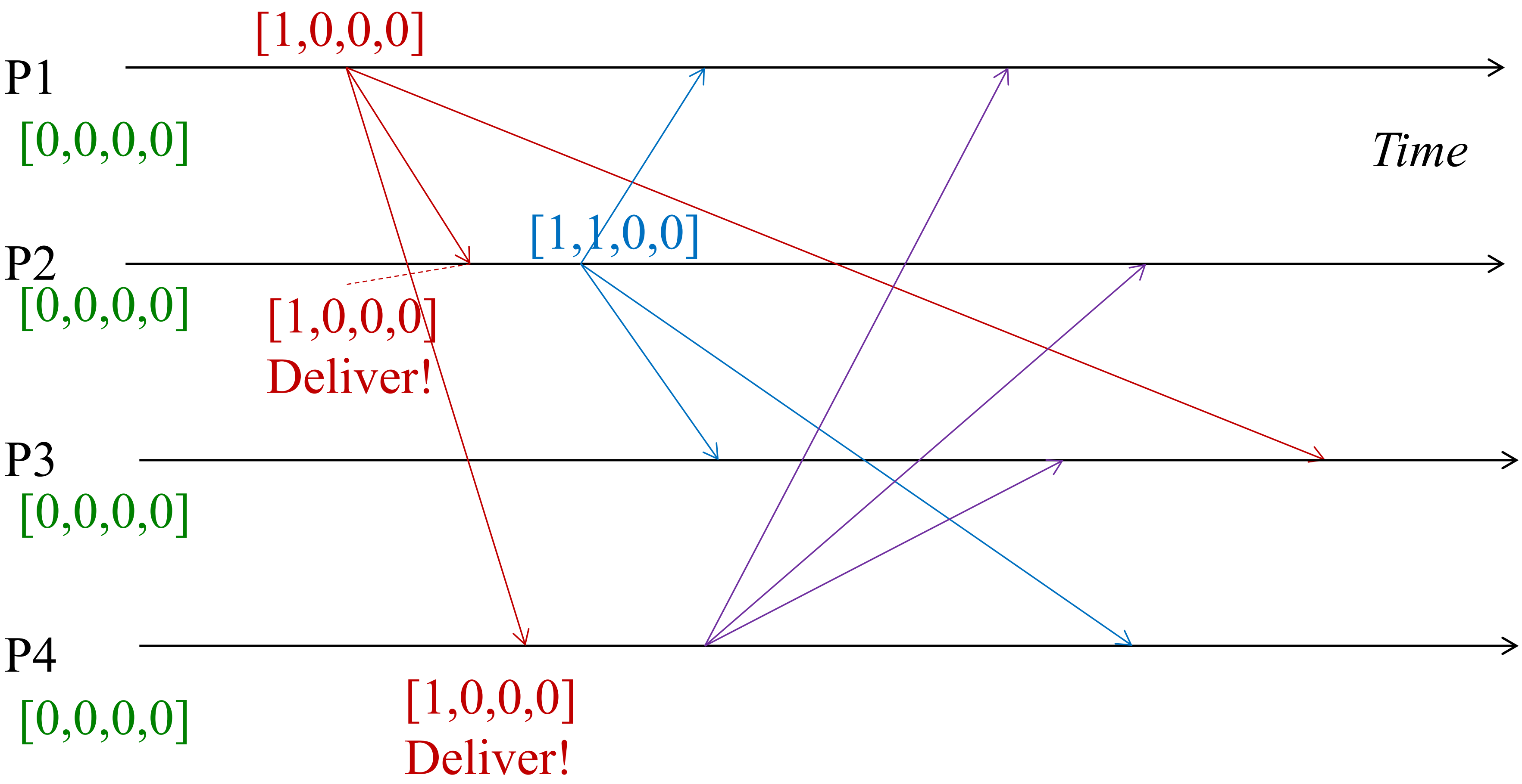
  When above two conditions satisfied,
  CO-deliver(m) and set $Pi[j] = V[j]$

Causal order multicast execution

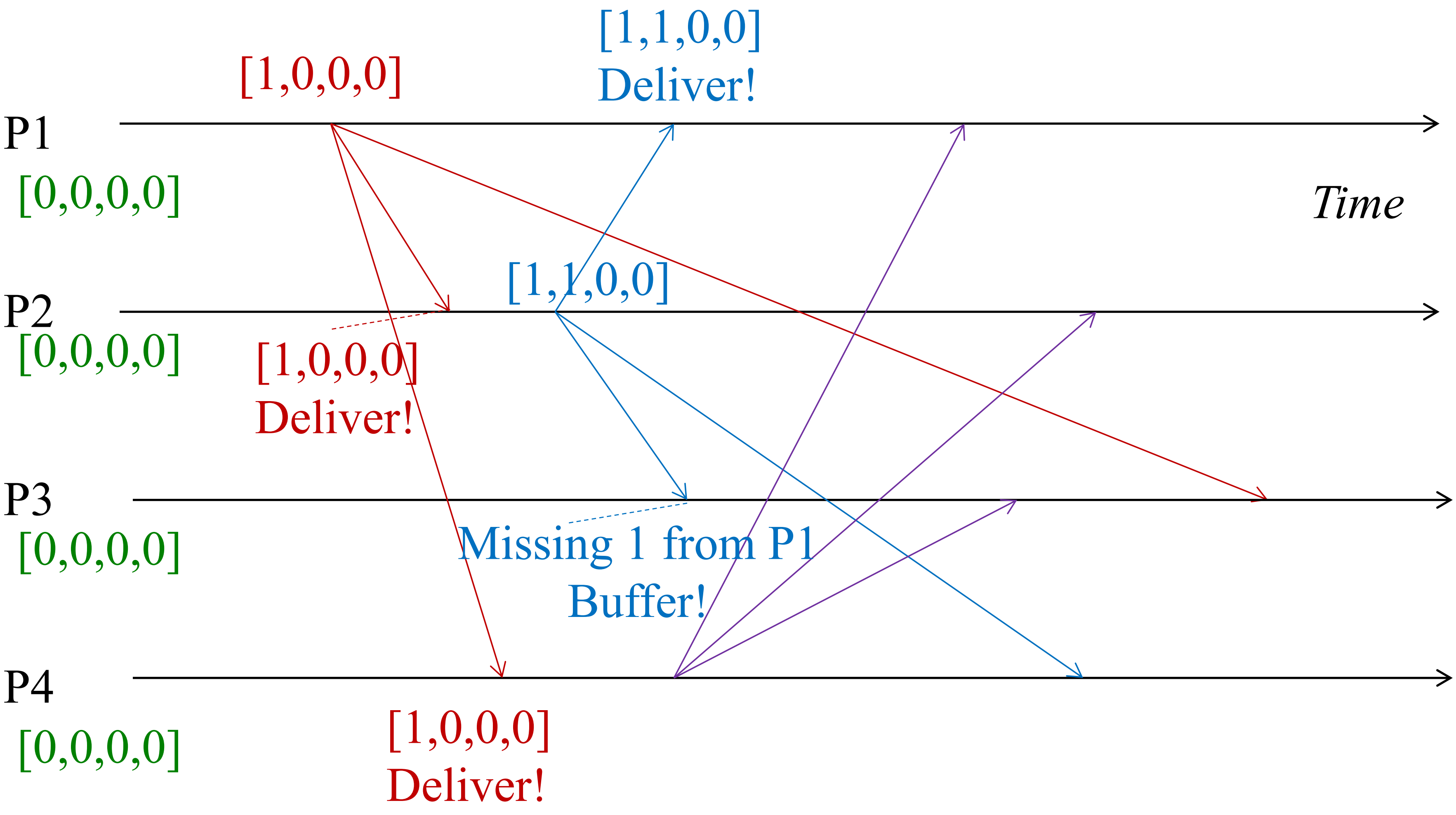


Self-deliveries omitted for simplicity.

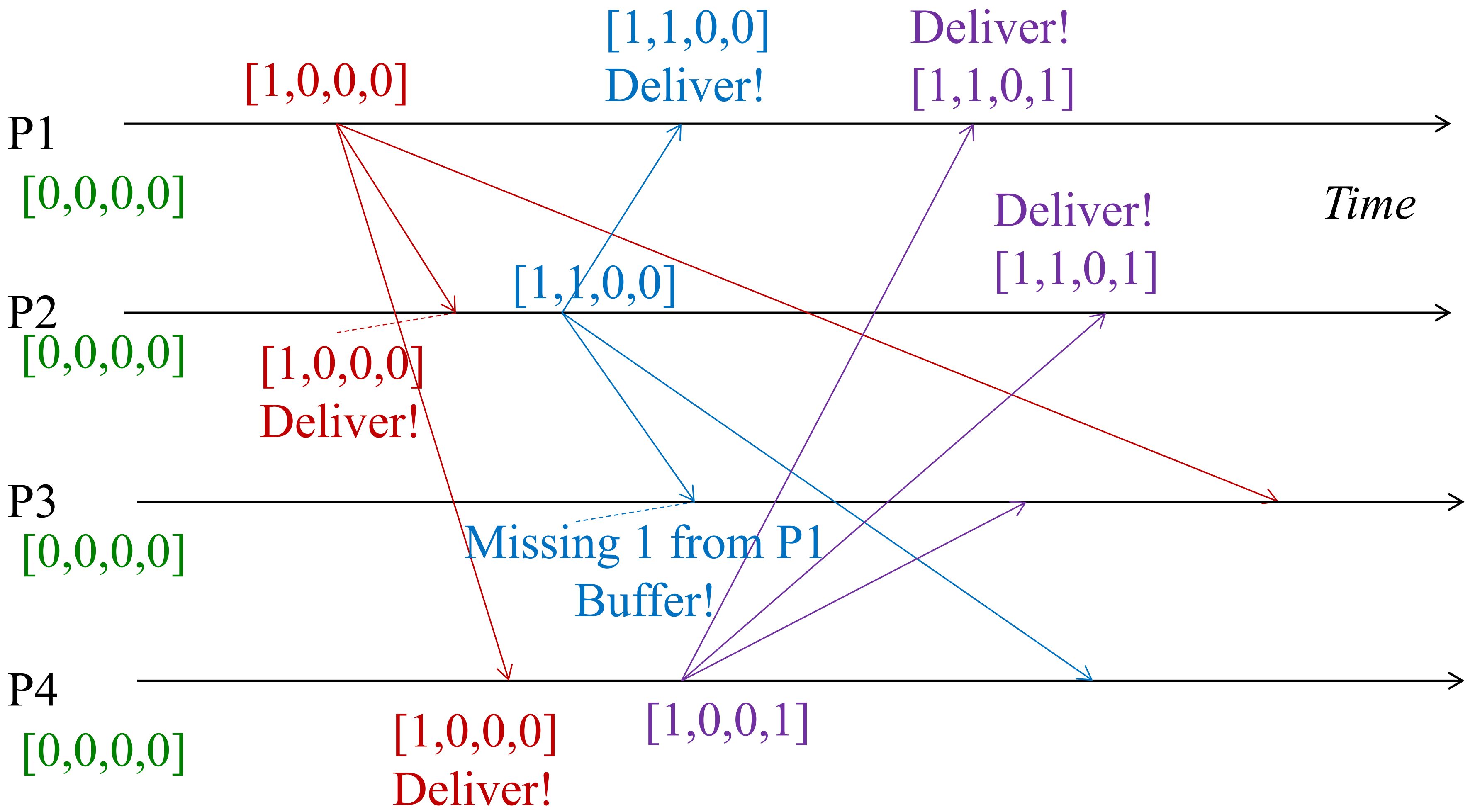
Causal order multicast execution



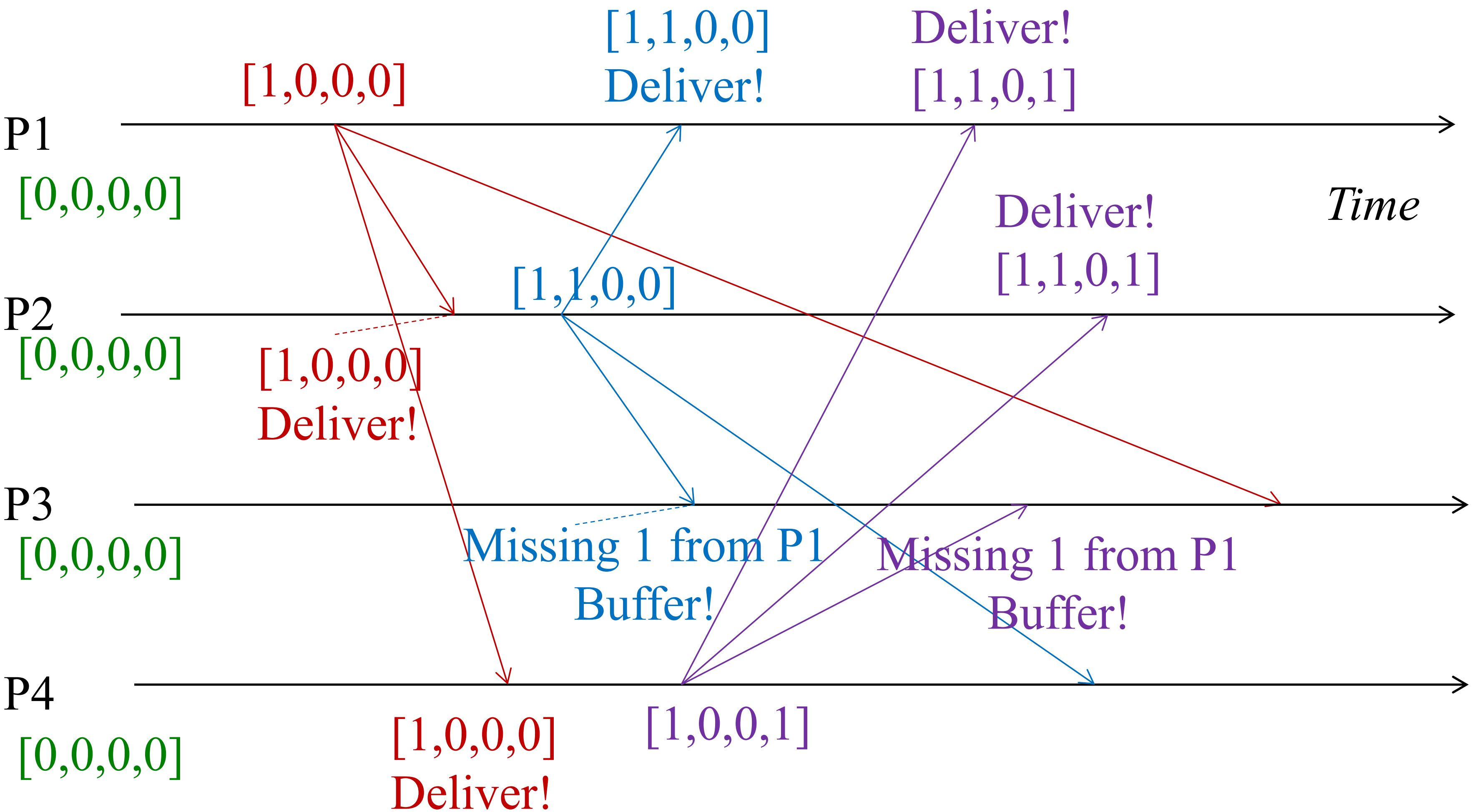
Causal order multicast execution



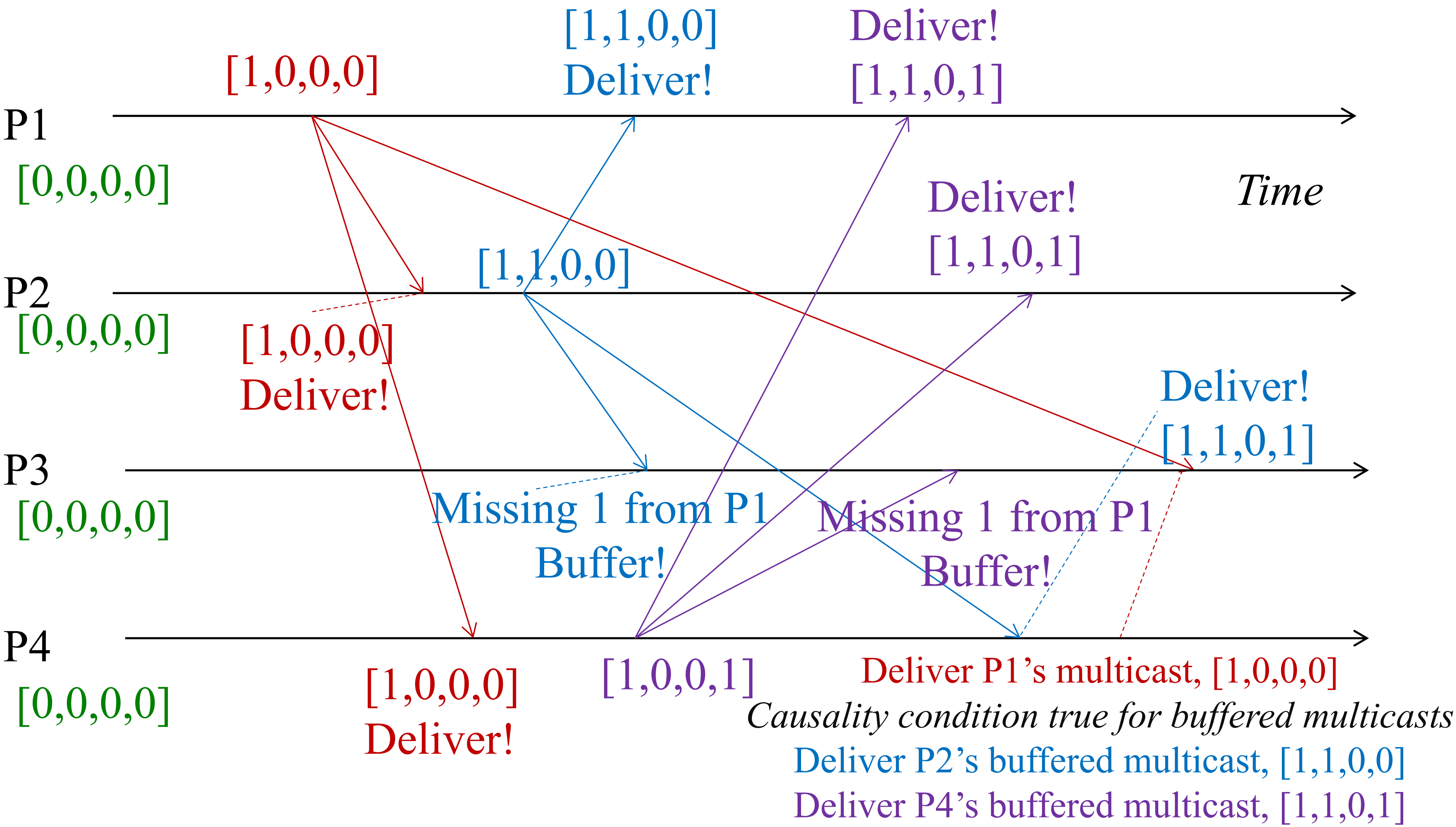
Causal order multicast execution



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)

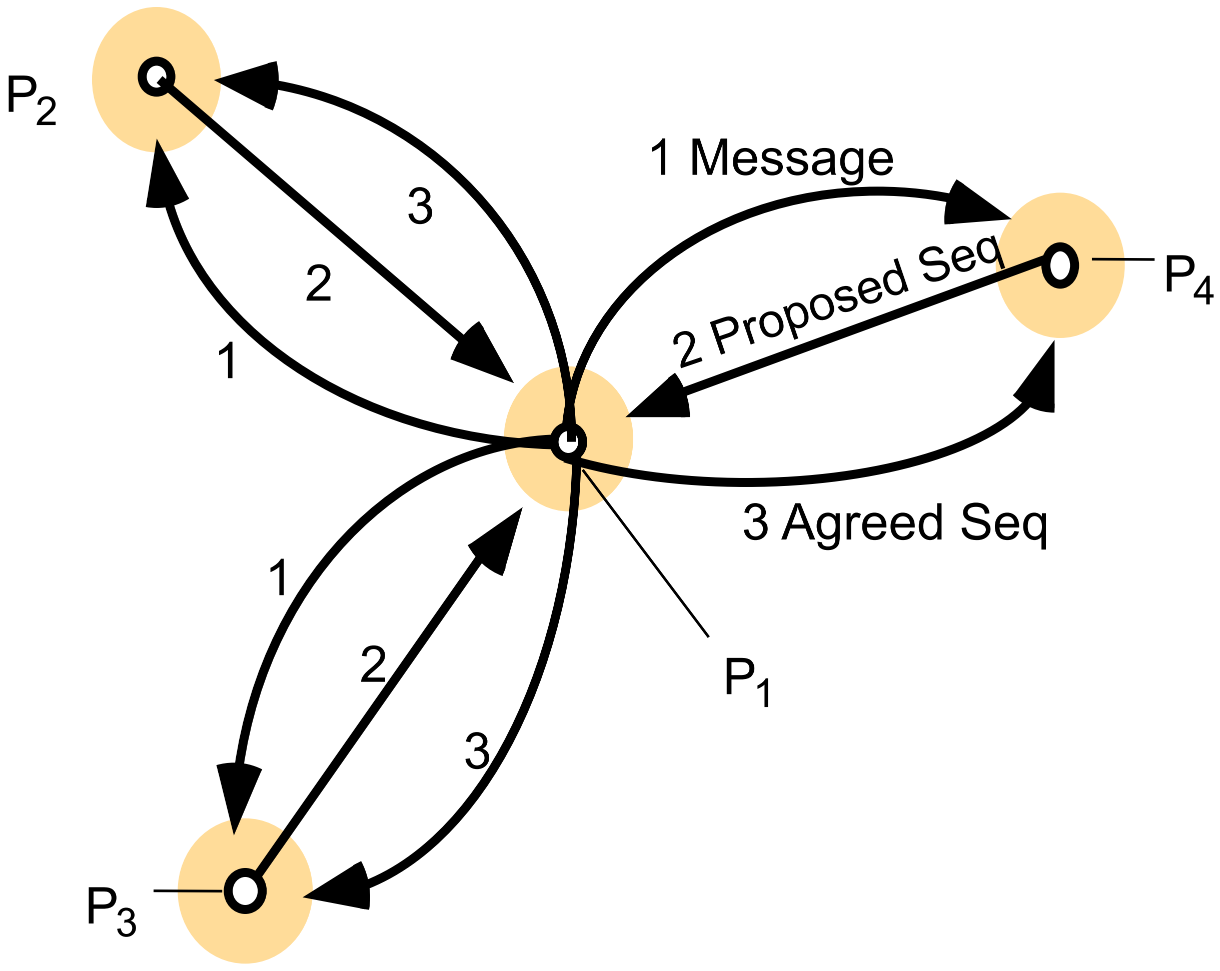
Sequencer based total ordering

- 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

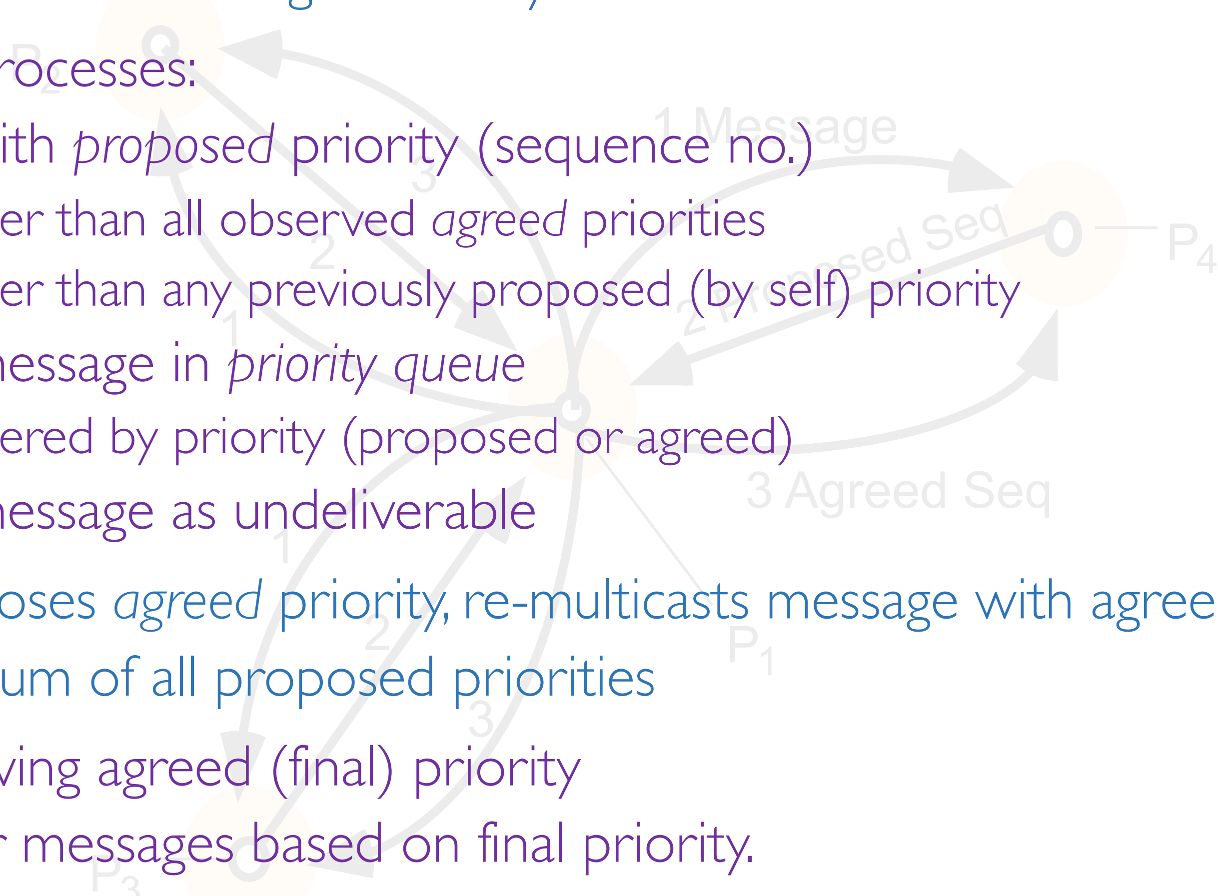
- Basic idea:
  - Same sequence number counter across different processes.
  - Instead of different sequence number counter for each process.
- Two types of approach
  - Using a centralized sequencer
  - **A decentralized mechanism (ISIS)**

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.



To be continued in next class

- Example of ISIS, and why it works.

Summary

- Multicast is an important communication mode in distributed systems.
- Applications may have different requirements:
  - Reliability
  - Ordering: FIFO, Causal, Total
  - Combinations of the above.