Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements for some of the materials: Indy Gupta and Nikita Borisov

Logistics

- MP1 has been released today.
- HW1 is due on Friday. HW2 will be released on Friday.
- Please see CampusWire post #98 on Midterm 1:
  - Mar 4-6. Make reservations on PrairieTest (starting Feb 19th).
  - Syllabus includes everything upto and including Global States.
  - CBTF does not allow cheatsheets – we will provide an abridged version of slides on PrairieLearn
    - The cheatsheet has been linked in post #98 for reference.
  - CBTF will provide scratch papers and calculator.
  - Please checkout CBTF rules.

Today's agenda

- **Multicast (contd)**
  - Chapter 15.4
  - Tree-based multicast and Gossip
- **Mutual Exclusion**
  - Chapter 15.2

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 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*.

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.

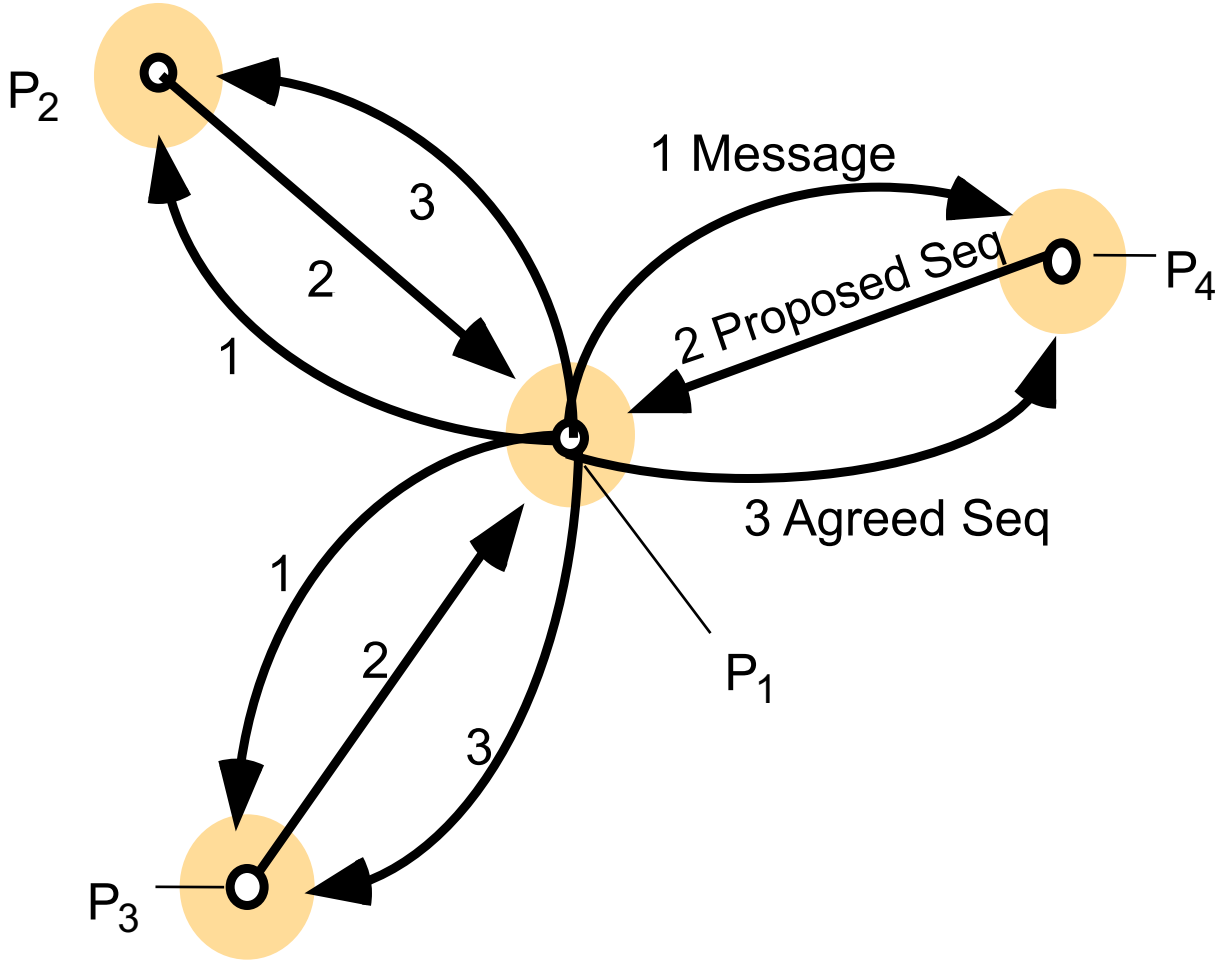
- Two types of approach
  - **Using a centralized sequencer**
  - A decentralized mechanism (ISIS)

Implementing total order multicast

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

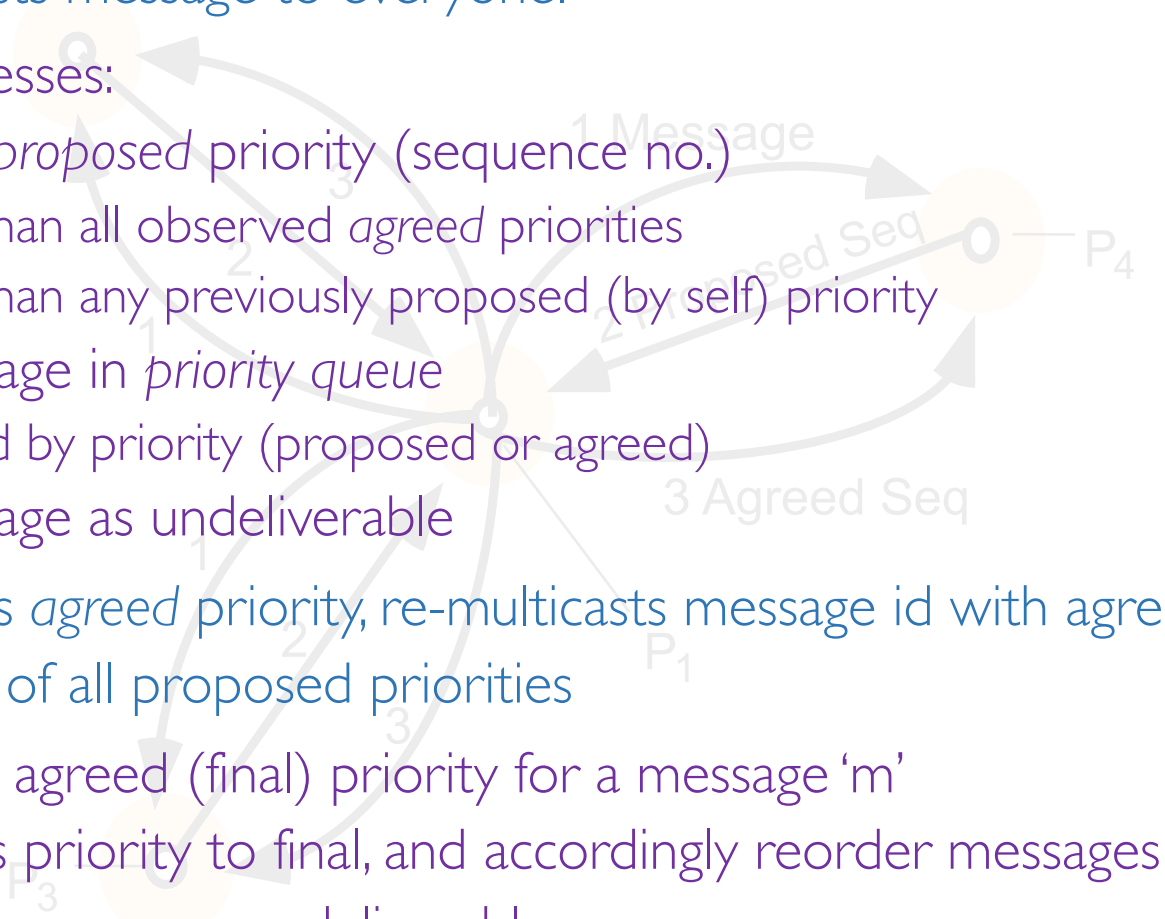
- Two types of approach
  - Using a centralized sequencer
  - **A decentralized mechanism (ISIS)**

ISIS algorithm for total ordering

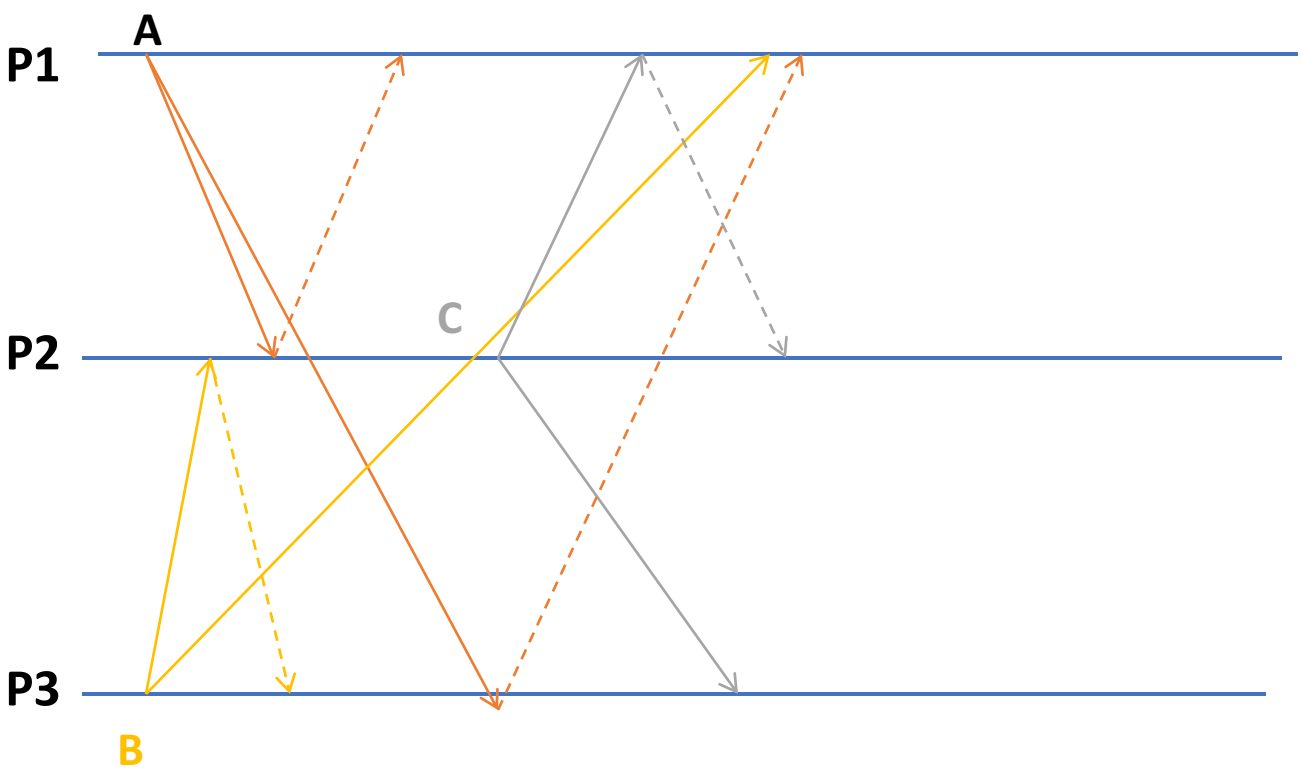


ISIS algorithm for total ordering

- 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

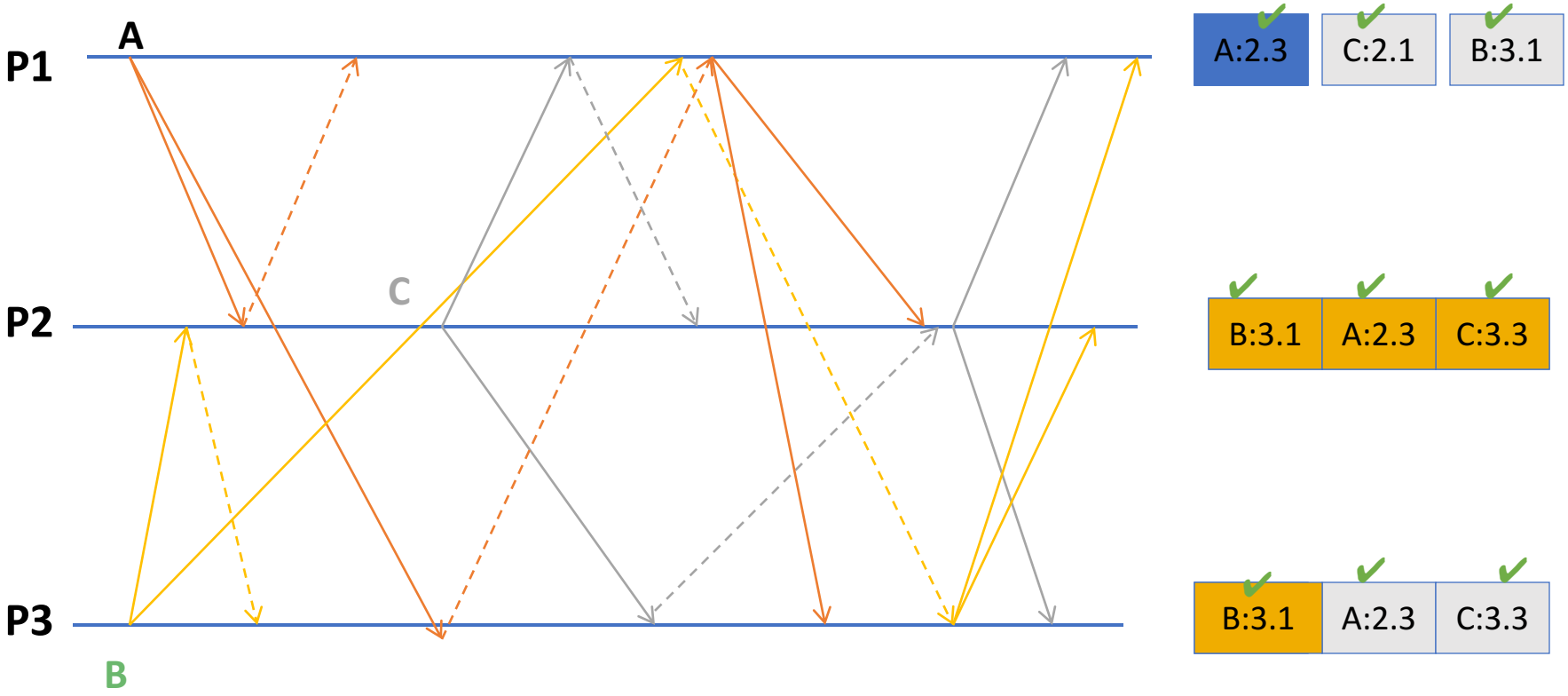


A:2	C:2	B:3
B:1	A:2	C:3
B:1	A:2	C:3

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
  - *The dot (.) does not denote decimal point!*
- Compare on priority first, use process # to break ties.
  - 2.1 > 1.3
  - 3.2 > 3.1

Example: ISIS algorithm



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

- 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*.

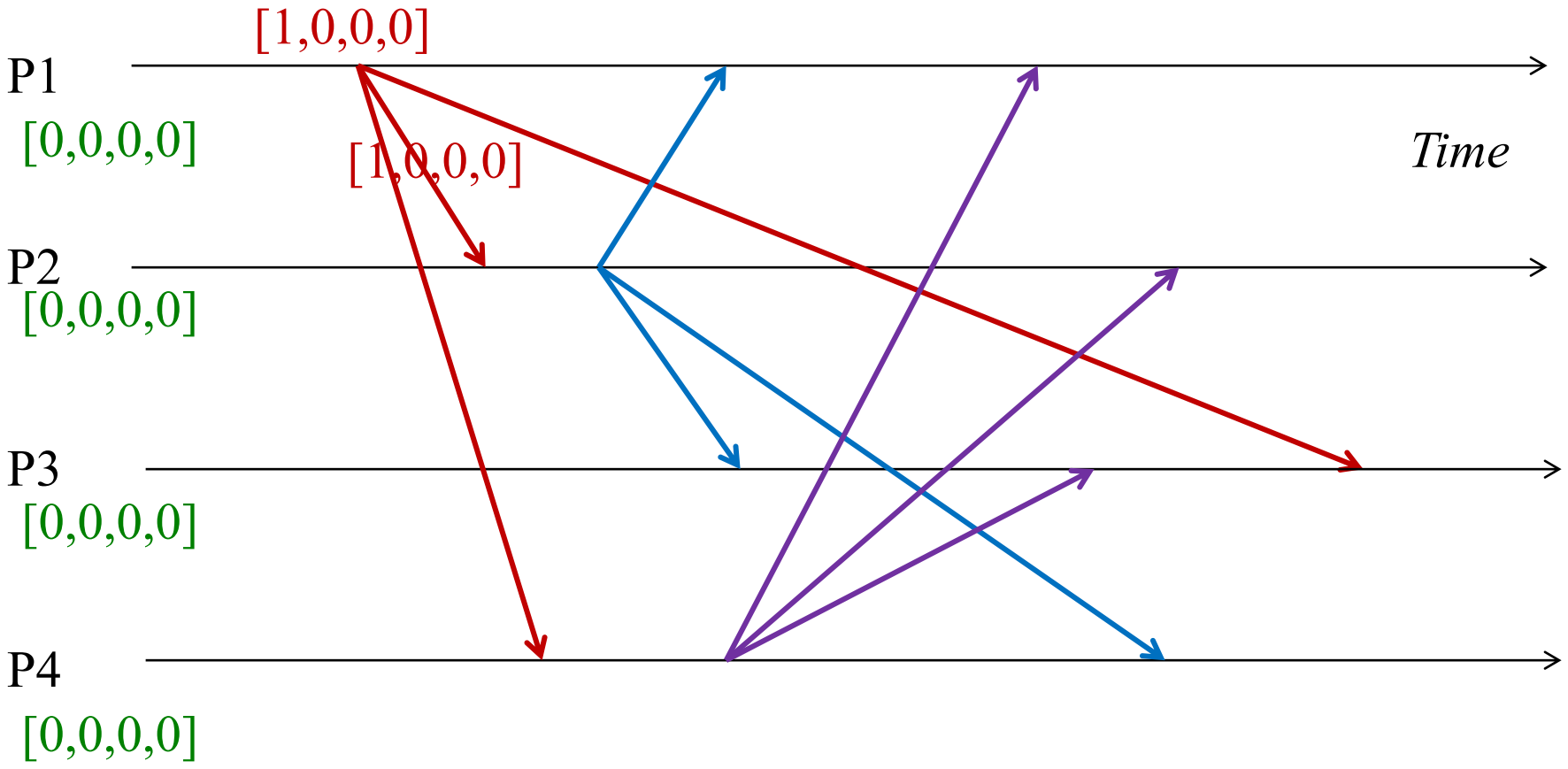
- Ignores other network messages. Only looks at multicast messages delivered to the application.

Implementing causal order multicast

- *CO-multicast(g,m) at Pj:*
  set $P_j[j] = P_j[j] + 1$
  piggyback entire vector $P_j[1 \ldots N]$ with m as its sequence no.
  B-multicast(g,{m, $P_j[1 \ldots N]$})
- *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 P_i is expecting from P_j, i.e.,
     $V[j] = P_i[j] + 1$
  2. All multicasts, anywhere in the group, which happened-before m have been received at P_i, i.e.,
     For all $k \neq j$: $V[k] \leq P_i[k]$

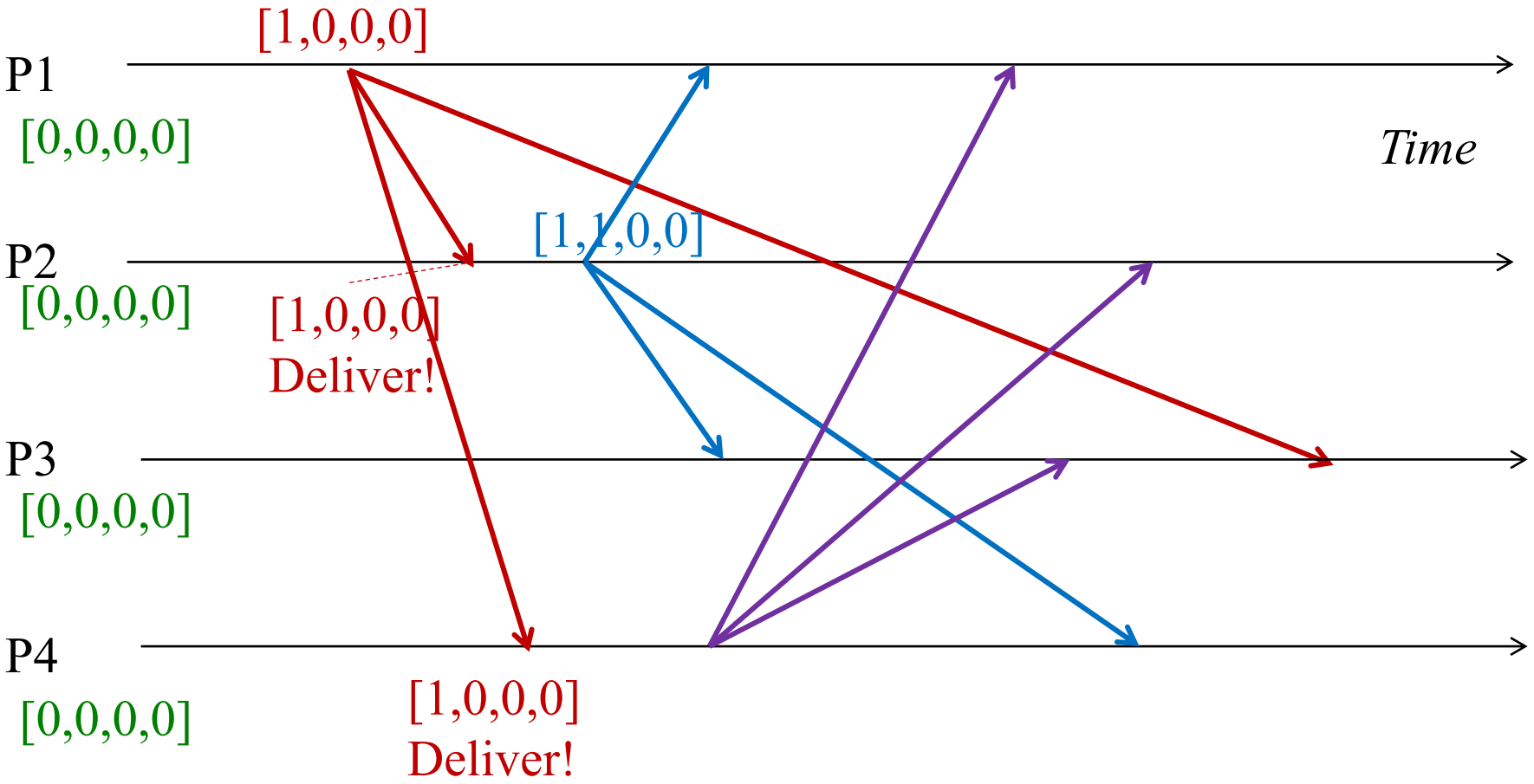
  When above two conditions satisfied,
  CO-deliver(m) and set $P_i[j] = V[j]$

Causal order multicast execution

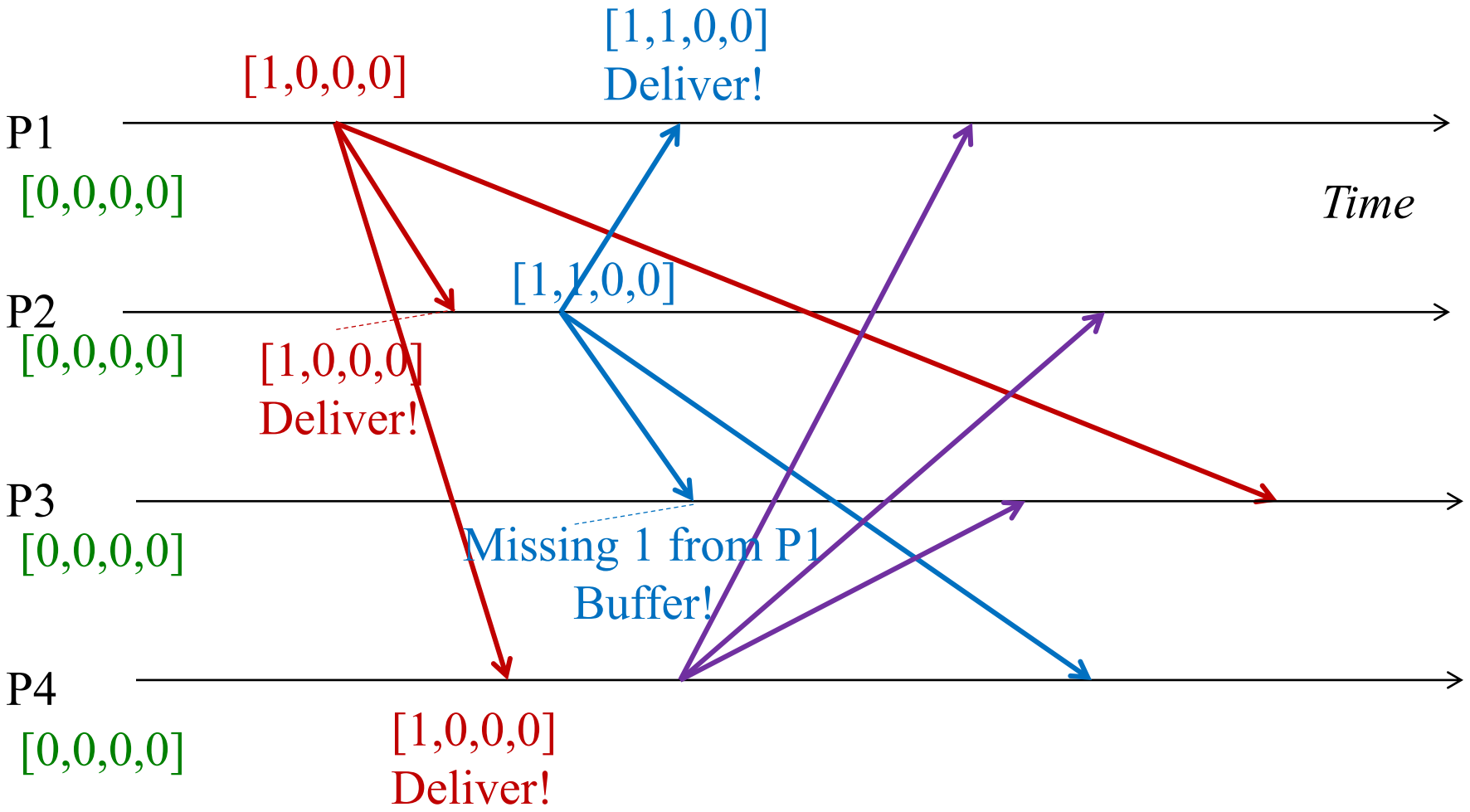


Self-deliveries omitted for simplicity.

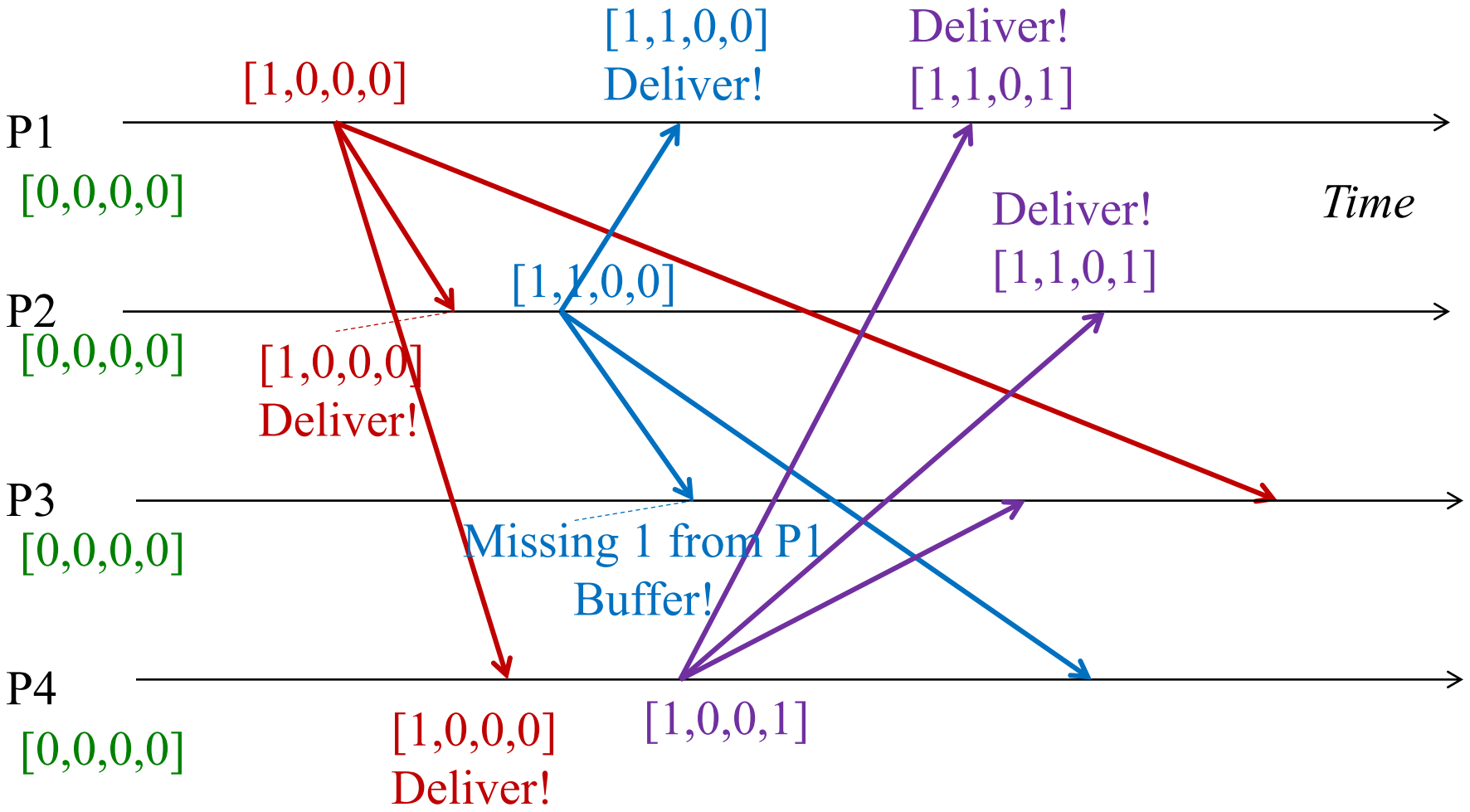
Causal order multicast execution



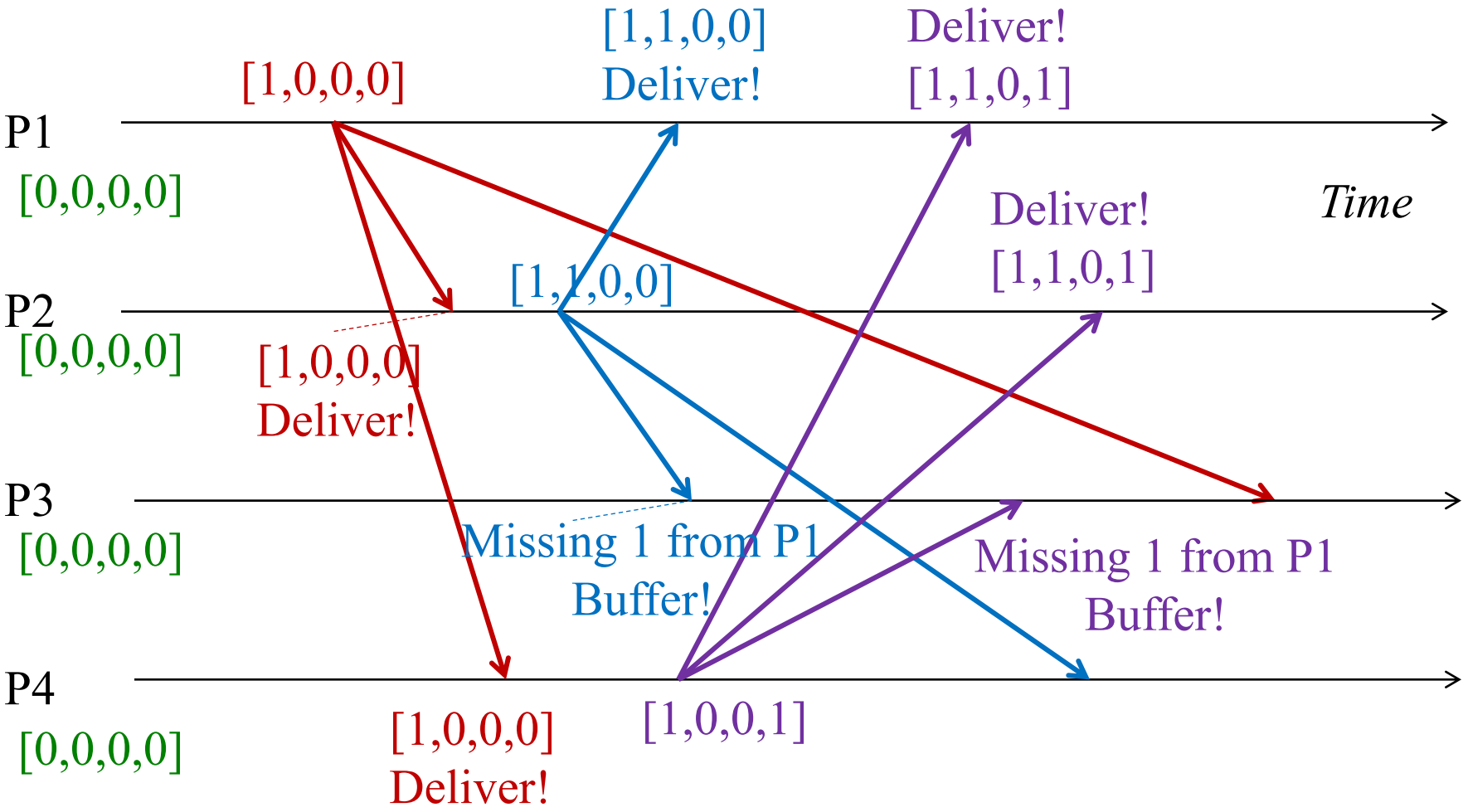
Causal order multicast execution



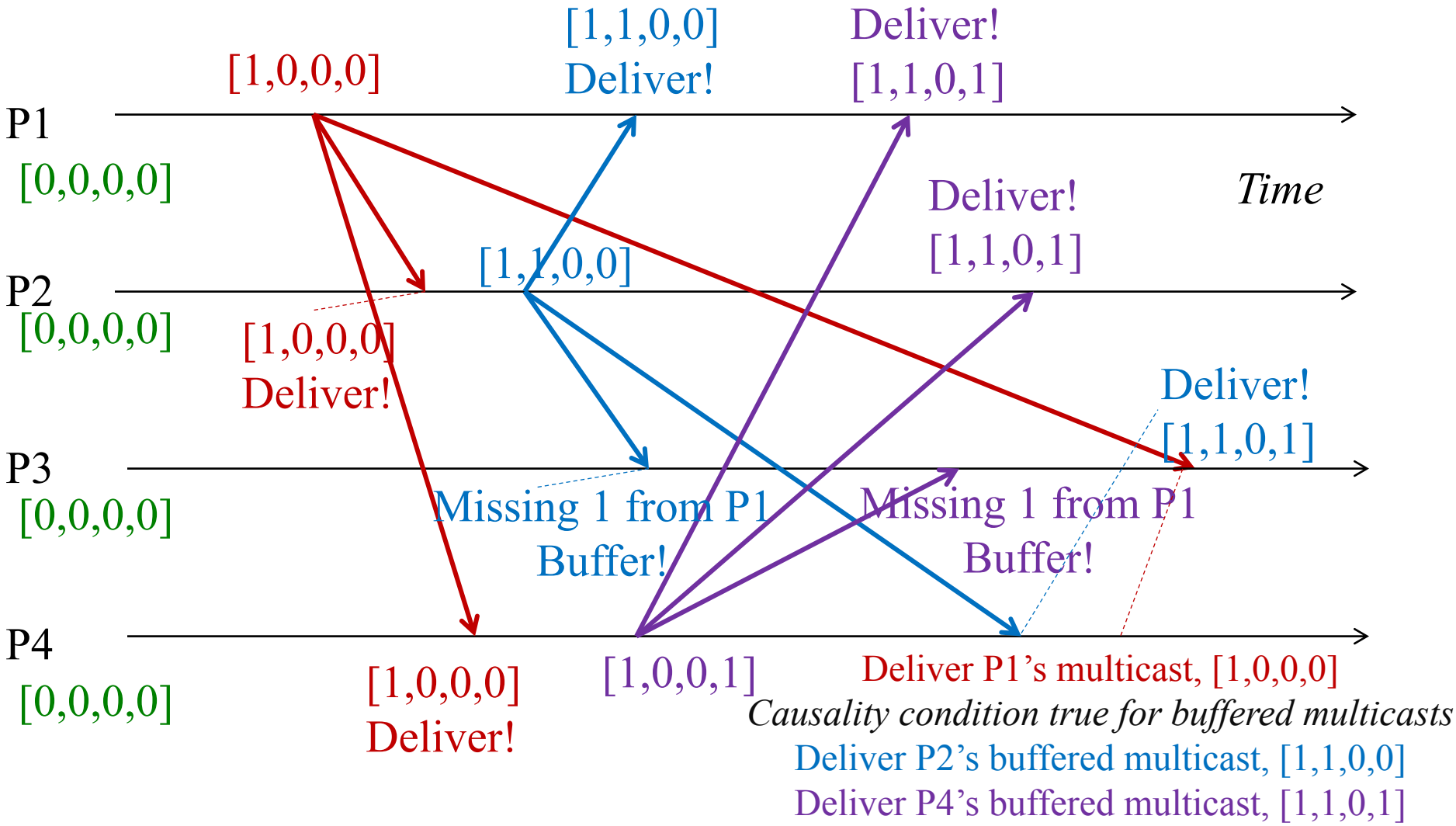
Causal order multicast execution



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 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*.

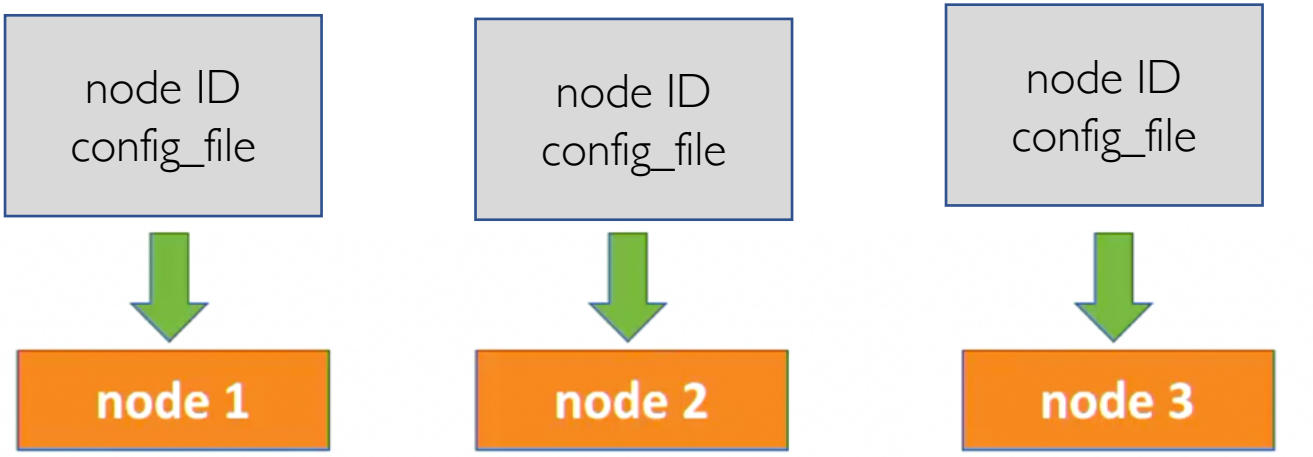
MP1: Event Ordering

- https://courses.grainger.illinois.edu/ece428/sp2026/mps/mp1.html
- Lead TA: Lyuwei Su

- 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.

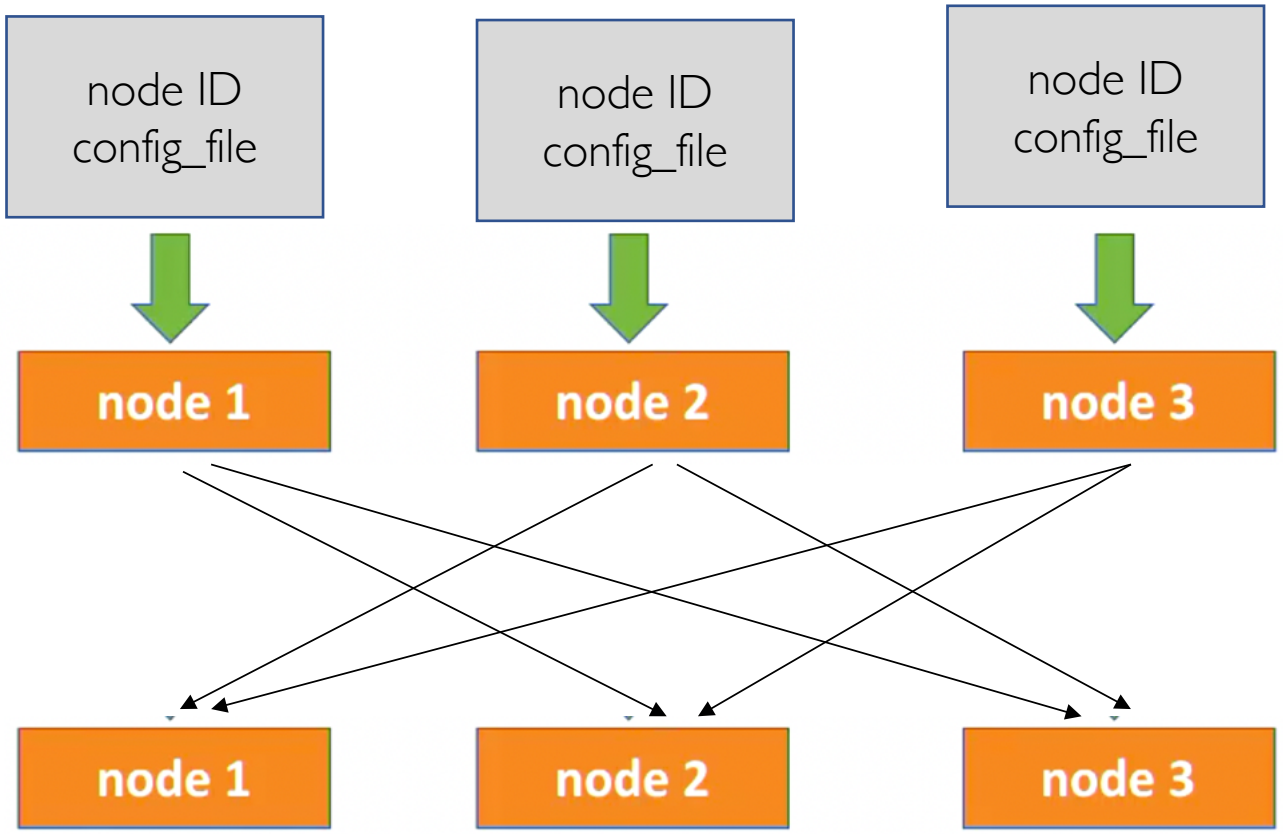
MP1 Architecture Setup



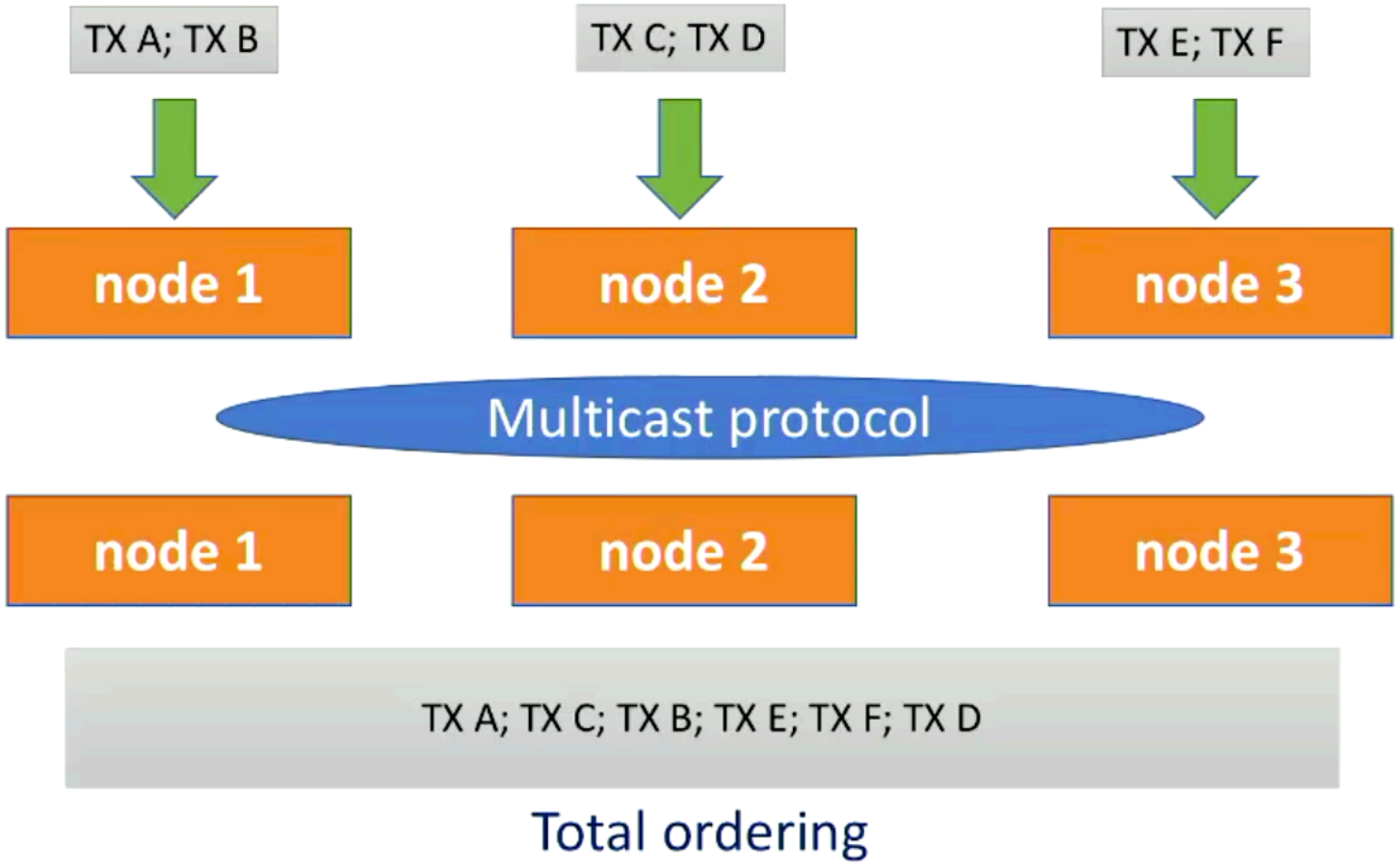
- Example input arguments for first node:
  `./mp1_node node1 config.txt`
- config.txt looks like this:

```
3
node1 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



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 March 13th.
  - 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!!