CS 425 / ECE 428
Distributed Systems
Fall 2016

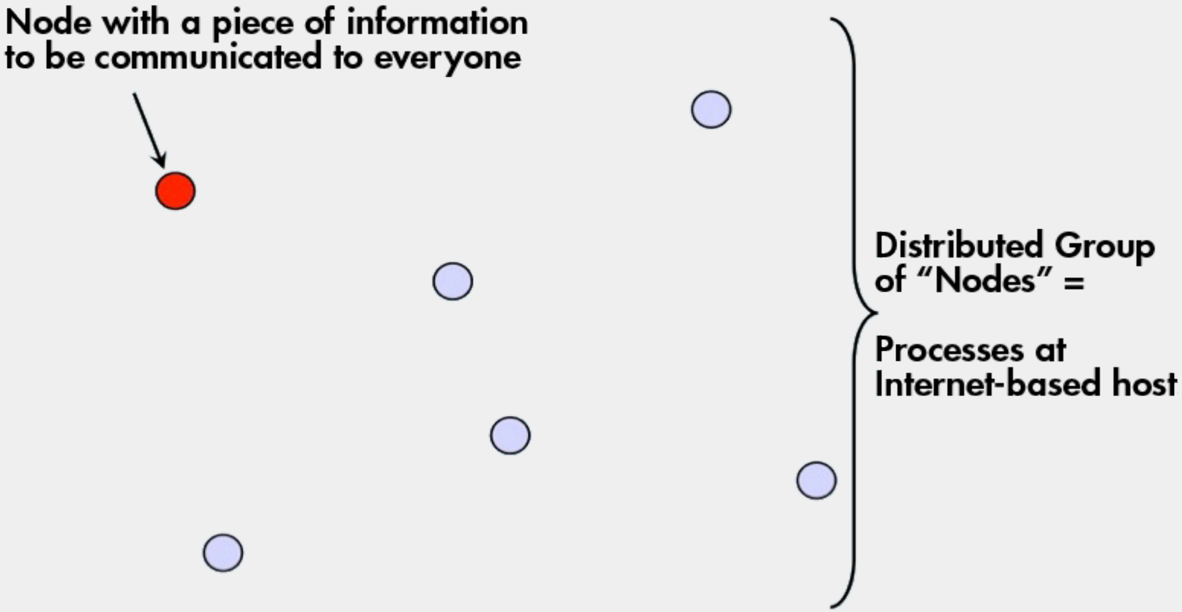
Indranil Gupta (Indy)

Lecture 9: Multicast

Oct 6, 2016

All slides © IG

MULTICAST PROBLEM



Other Communication Forms

- **Multicast** → message sent to a group of processes
- **Broadcast** → message sent to all processes (anywhere)
- **Unicast** → message sent from one sender process to one receiver process

Who Uses Multicast?

- A widely-used abstraction by almost all cloud systems
- Storage systems like Cassandra or a database
  - Replica servers for a key: Writes/reads to the key are multicast within the replica group
  - All servers: membership information (e.g., heartbeats) is multicast across all servers in cluster
- Online scoreboards (ESPN, French Open, FIFA World Cup)
  - Multicast to group of clients interested in the scores
- Stock Exchanges
  - Group is the set of broker computers
  - Groups of computers for High frequency Trading
- Air traffic control system
  - All controllers need to receive the same updates in the same order

Multicast Ordering

- Determines the meaning of "same order" of multicast delivery at different processes in the group
- Three popular flavors implemented

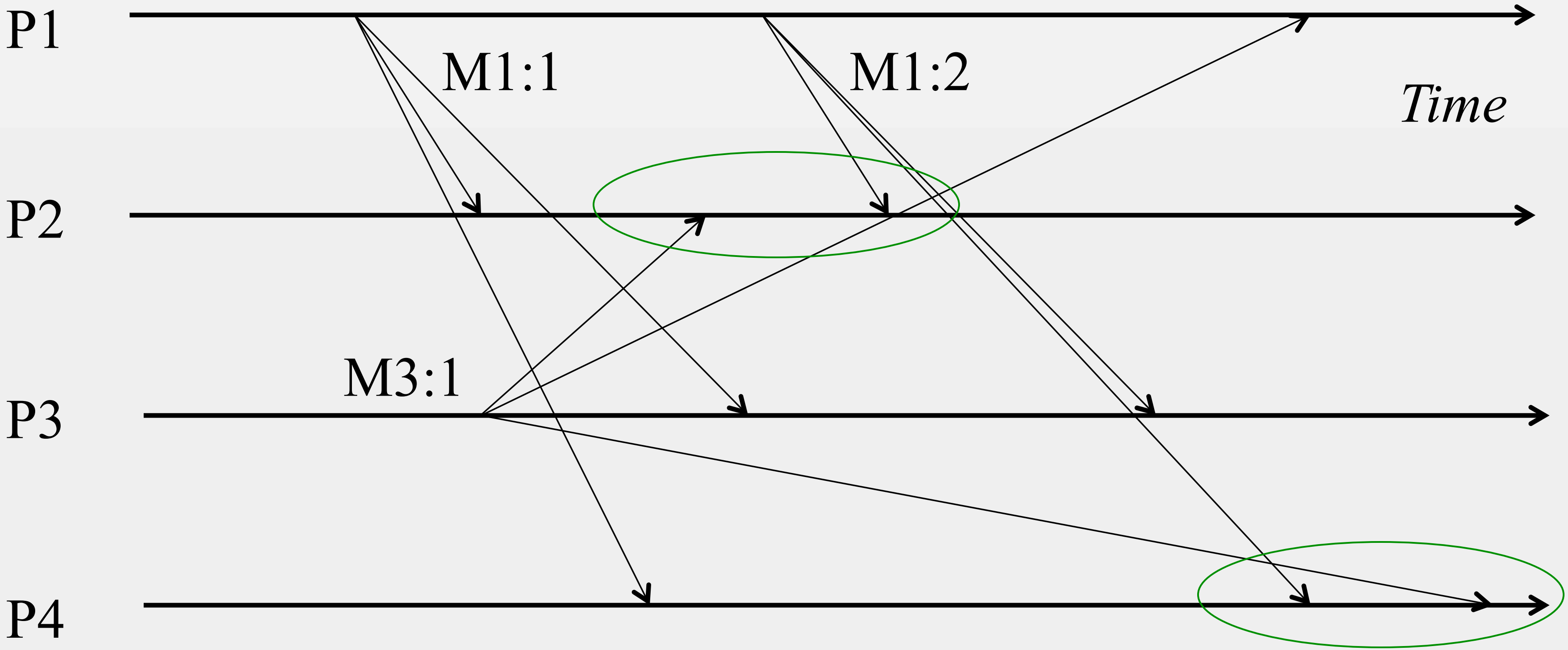
  by several multicast protocols

  1. FIFO ordering
  2. Causal ordering
  3. Total ordering

1. FIFO ORDERING

- Multicasts from each sender are received in the order they are sent, at all receivers
- Don’t worry about multicasts from different senders
- More formally
  - *If a correct process issues (sends) multicast(g,m) to group g and then multicast(g,m’), then every correct process that delivers m’ would already have delivered m.*

FIFO Ordering: Example



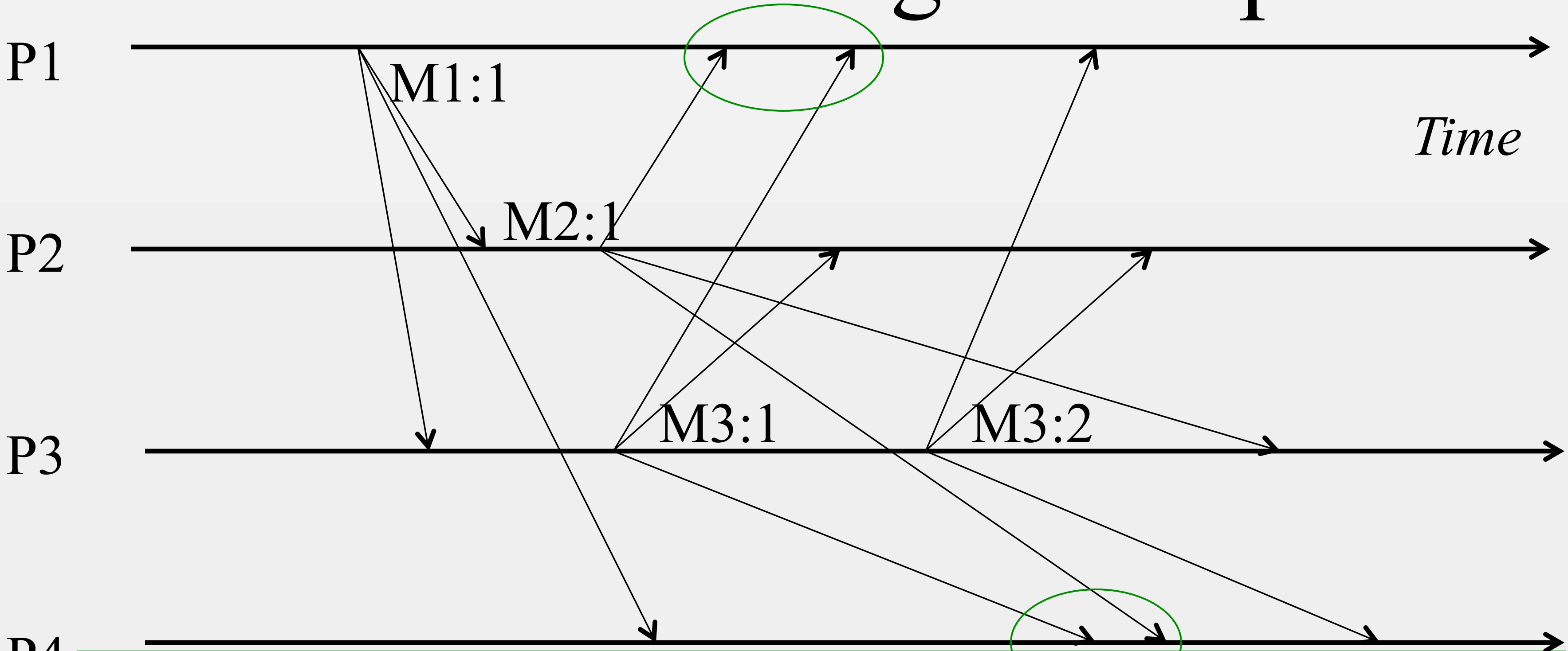
M1:1 and M1:2 should be received in that order at each receiver

Order of delivery of M3:1 and M1:2 could be different at different receivers

2. Causal Ordering

- Multicasts whose send events are causally related, must be received in the same causality-obeying order at all receivers
- Formally
  - *If multicast(g,m) → multicast(g,m')* *then any correct process that delivers m' would already have delivered m.*
  - *(→ is Lamport's happens-before)*

Causal Ordering: Example



M3:1 → M3:2, and so should be received in that order at each receiver

M1:1 → M3:1, and so should be received in that order at each receiver

M3:1 and M2:1 are concurrent and thus ok to be received in different orders at different receivers

Causal vs. FIFO

- Causal Ordering => FIFO Ordering
- Why?
  - If two multicasts M and M' are sent by the same process P, and M was sent before M', then M → M'
  - Then a multicast protocol that implements causal ordering will obey FIFO ordering since M → M'
- Reverse is not true! FIFO ordering does not imply causal ordering.

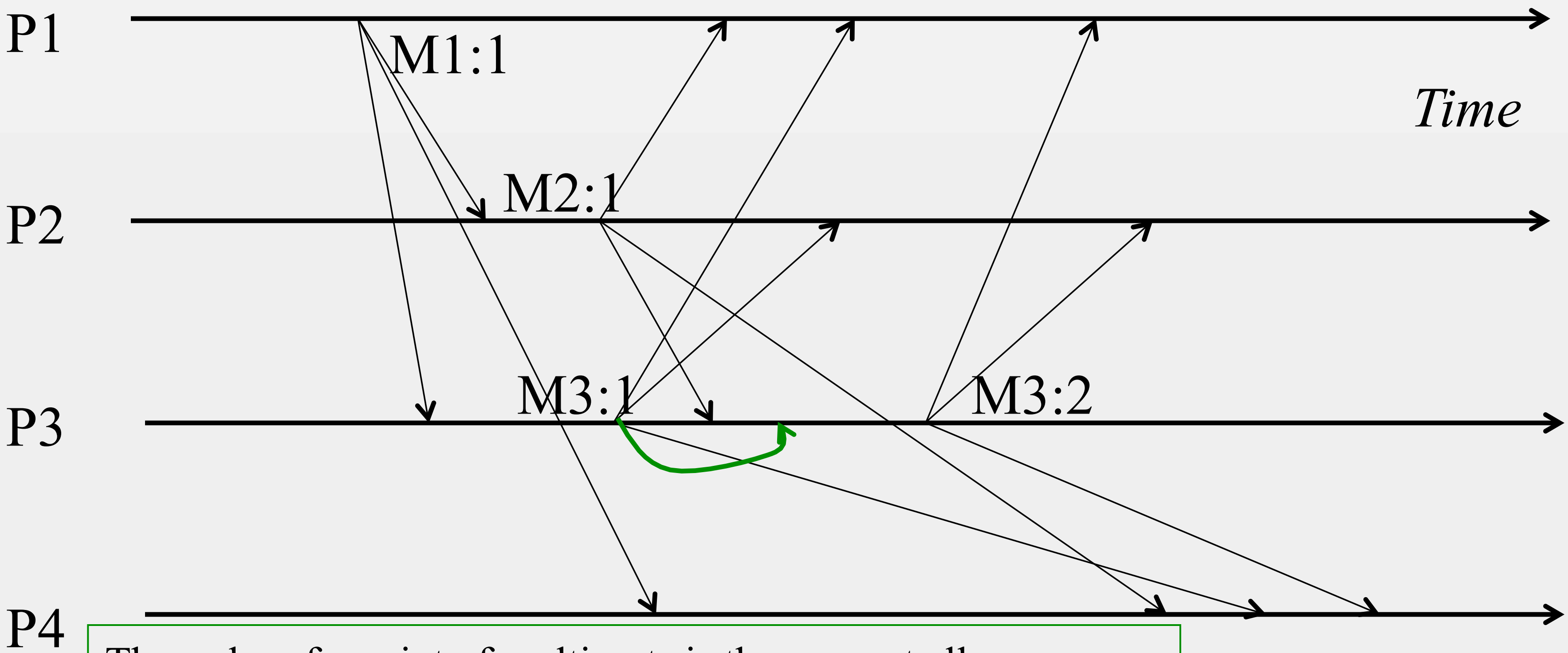
Why Causal at All?

- 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.

3. Total Ordering

- Also known as “Atomic Broadcast”
- Unlike FIFO and causal, this does not pay attention to order of multicast sending
- Ensures all receivers receive all multicasts in the same order
- Formally
  - *If a correct process P delivers message m before m’ (independent of the senders), then any other correct process P’ that delivers m’ would already have delivered m.*

Total Ordering: 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

Hybrid Variants

- Since FIFO/Causal are orthogonal to Total, can have hybrid ordering protocols too
  - FIFO-total hybrid protocol satisfies both FIFO and total orders
  - Causal-total hybrid protocol satisfies both Causal and total orders

Implementation?

- That was *what* ordering is
- But *how* do we implement each of these orderings?

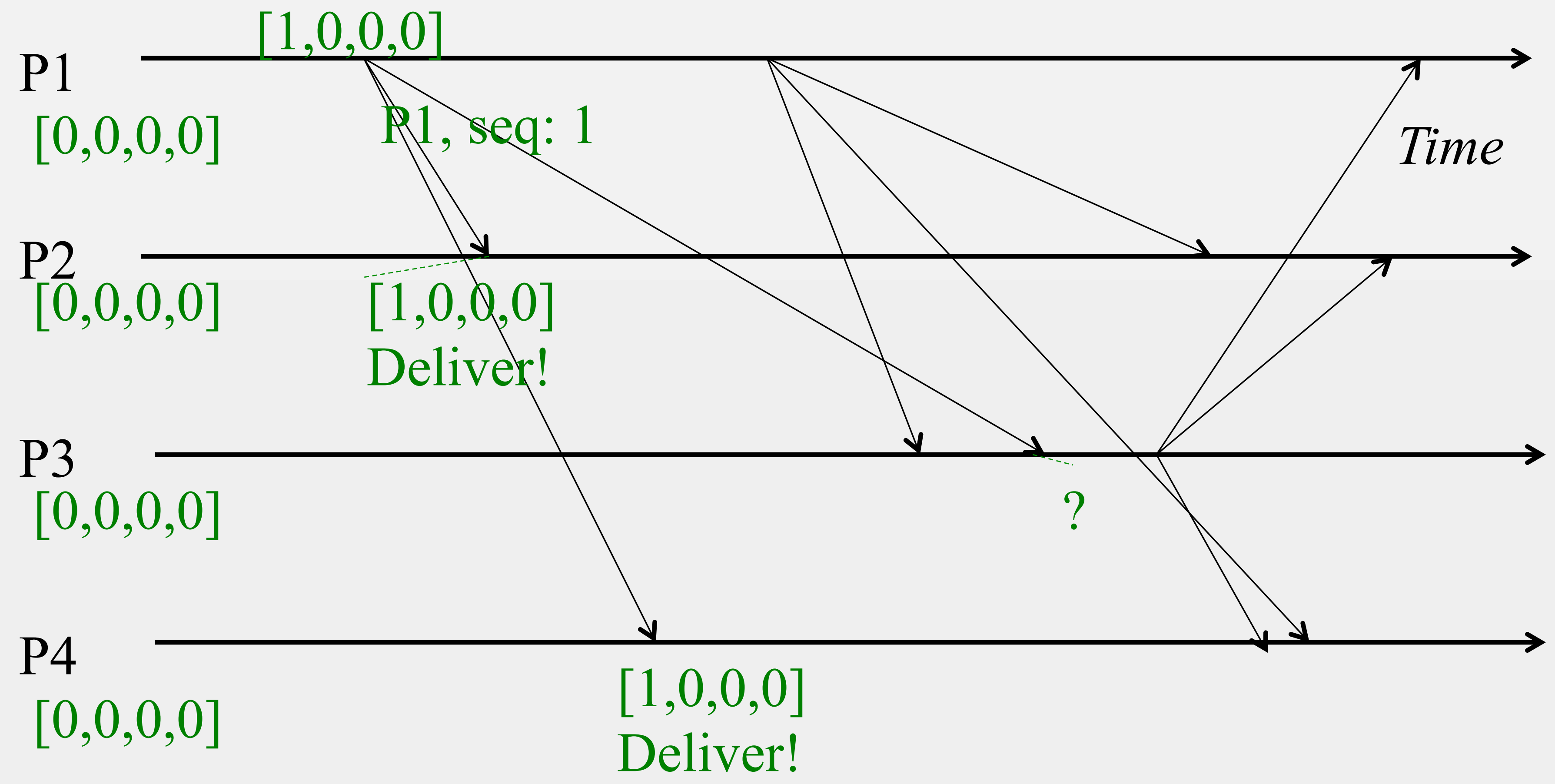
FIFO Multicast: Data Structures

- Each receiver maintains a per-sender sequence number (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*

FIFO Multicast: Updating Rules

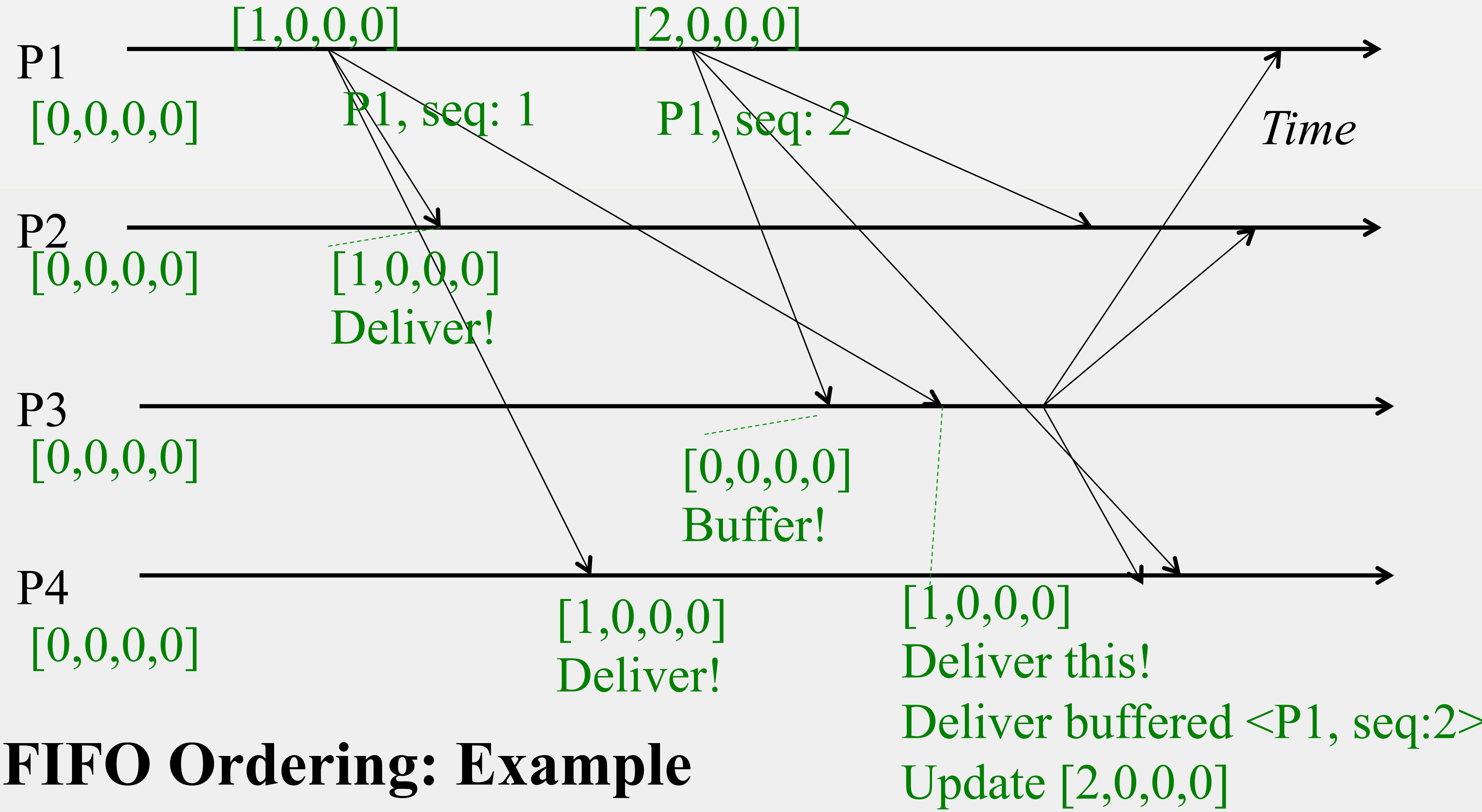
- Send multicast at process Pj:
  - Set Pj[j] = Pj[j] + 1
  - Include new Pj[j] in multicast message as its sequence number
- Receive multicast: If Pi receives a multicast from Pj with sequence number S in message
  - if (S == Pi[j] + 1) then
    - deliver message to application
    - Set Pi[j] = Pi[j] + 1
  - else buffer this multicast until above condition is true

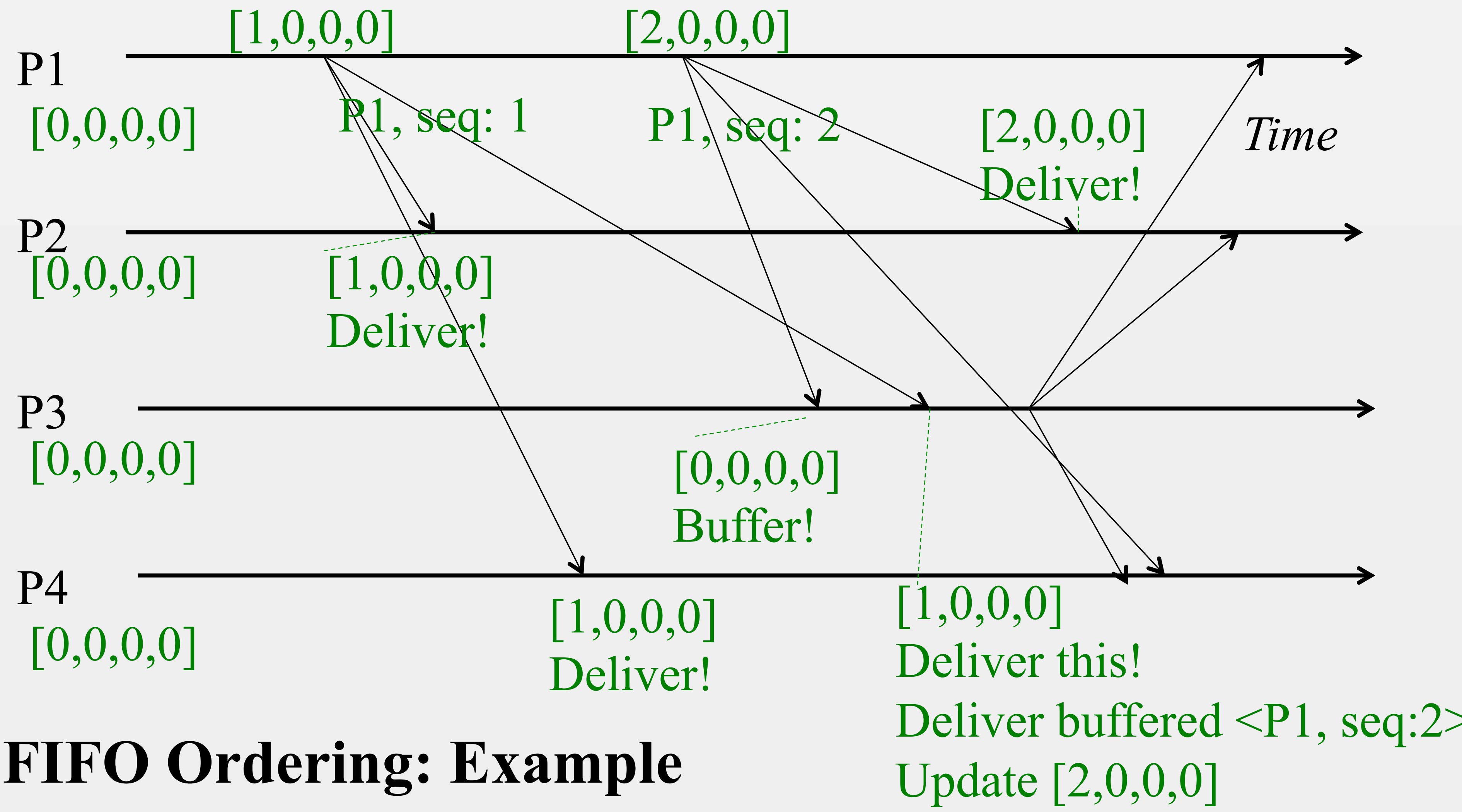
FIFO Ordering: Example



FIFO Ordering: Example

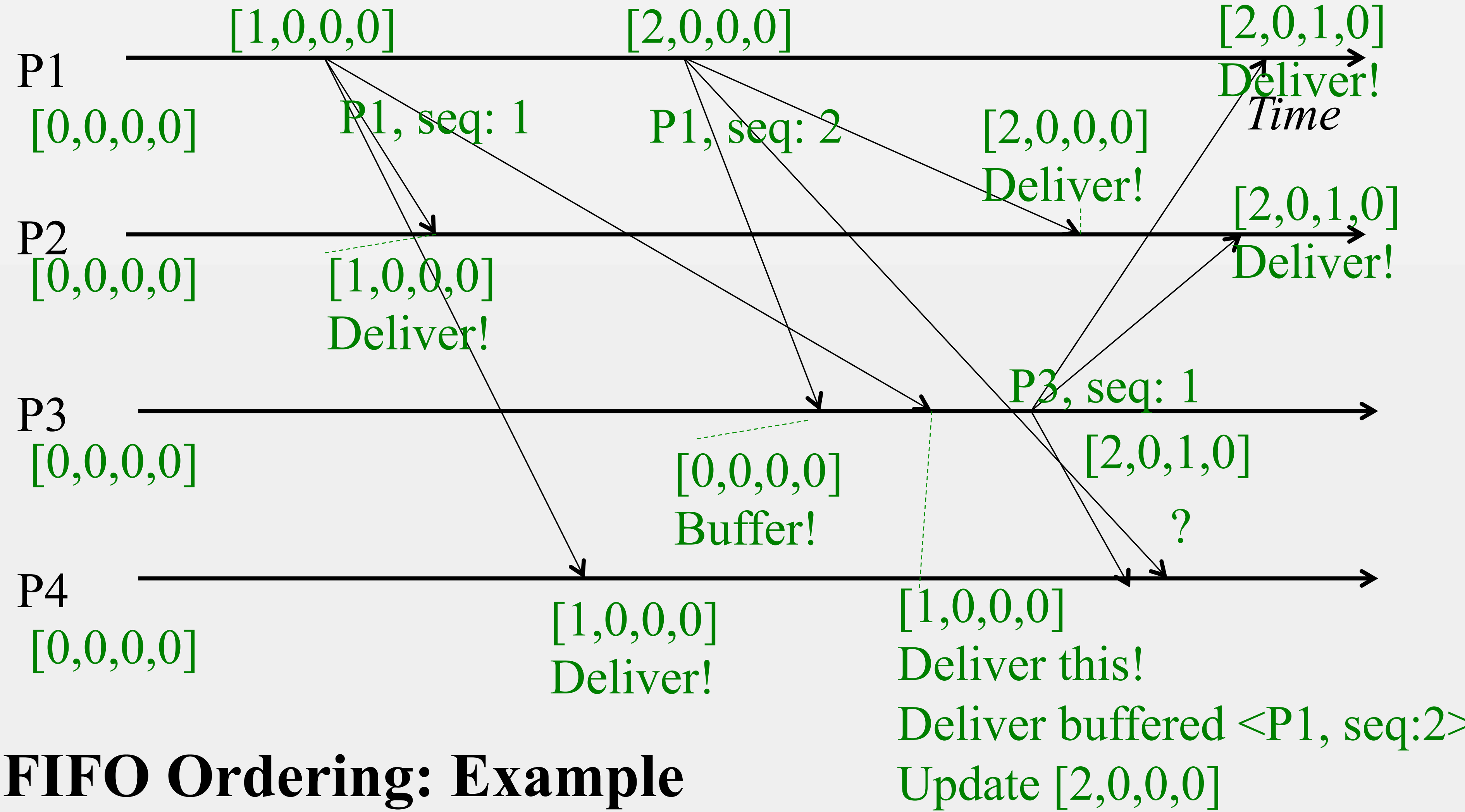
FIFO Ordering: Example



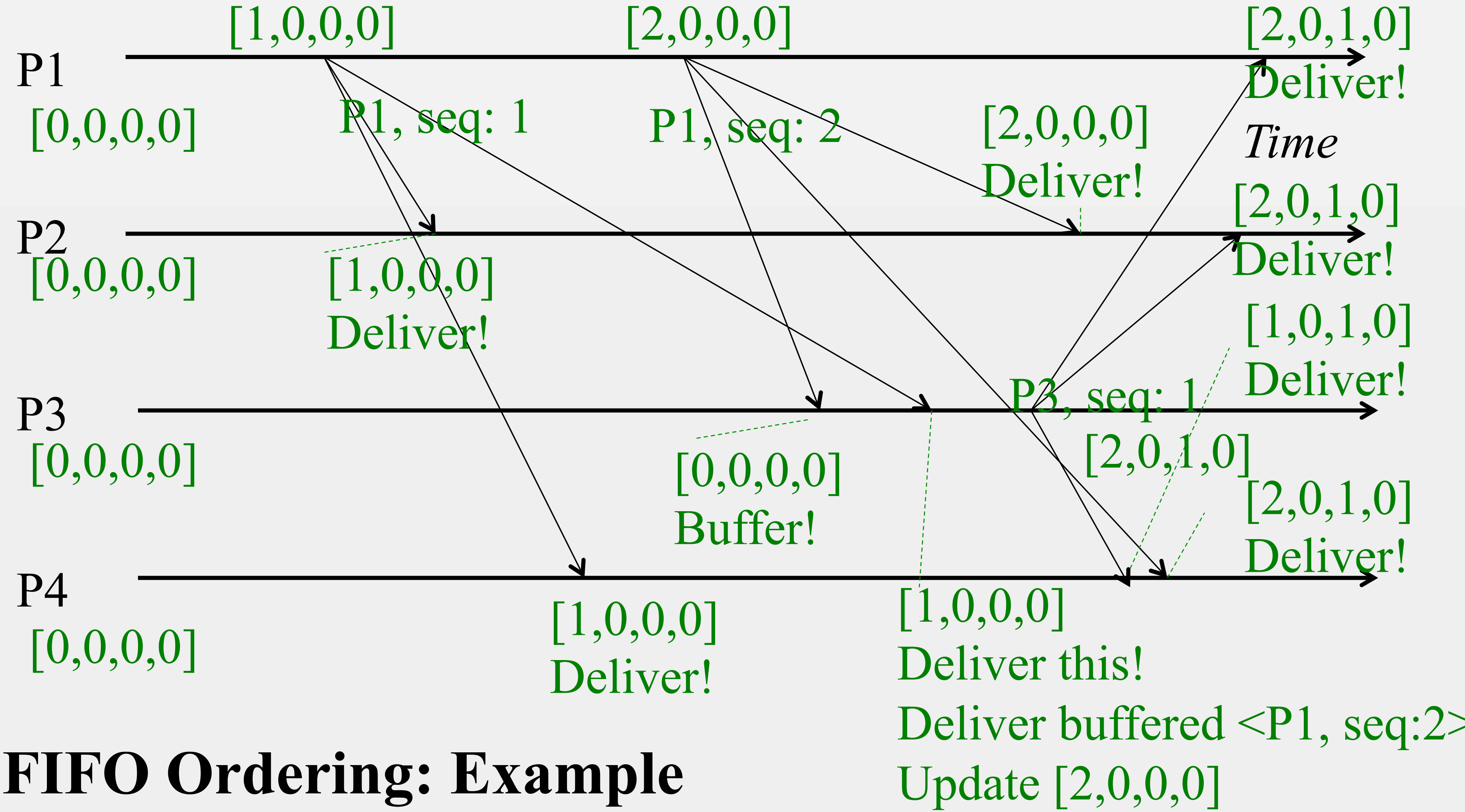


FIFO Ordering: Example

FIFO Ordering: Example



FIFO Ordering: Example



Total Ordering

- Ensures all receivers receive all multicasts in the same order
- Formally
  - *If a correct process P delivers message m before m' (independent of the senders), then any other correct process P' that delivers m' would already have delivered m.*

Sequencer-based Approach

- Special process elected as leader or sequencer
- Send multicast at process Pi:
  - Send multicast message M to group and sequencer
- Sequencer:
  - Maintains a global sequence number S (initially 0)
  - When it receives a multicast message M, it sets S = S + 1, and multicasts <M, S>
- Receive multicast at process Pi:
  - Pi maintains a local received global sequence number Si (initially 0)
  - If Pi receives a multicast M from Pj, it buffers it until it both
    1. Pi receives <M, S(M)> from sequencer, and
    2. Si + 1 = S(M)
    - Then deliver it message to application and set Si = Si + 1

Causal Ordering

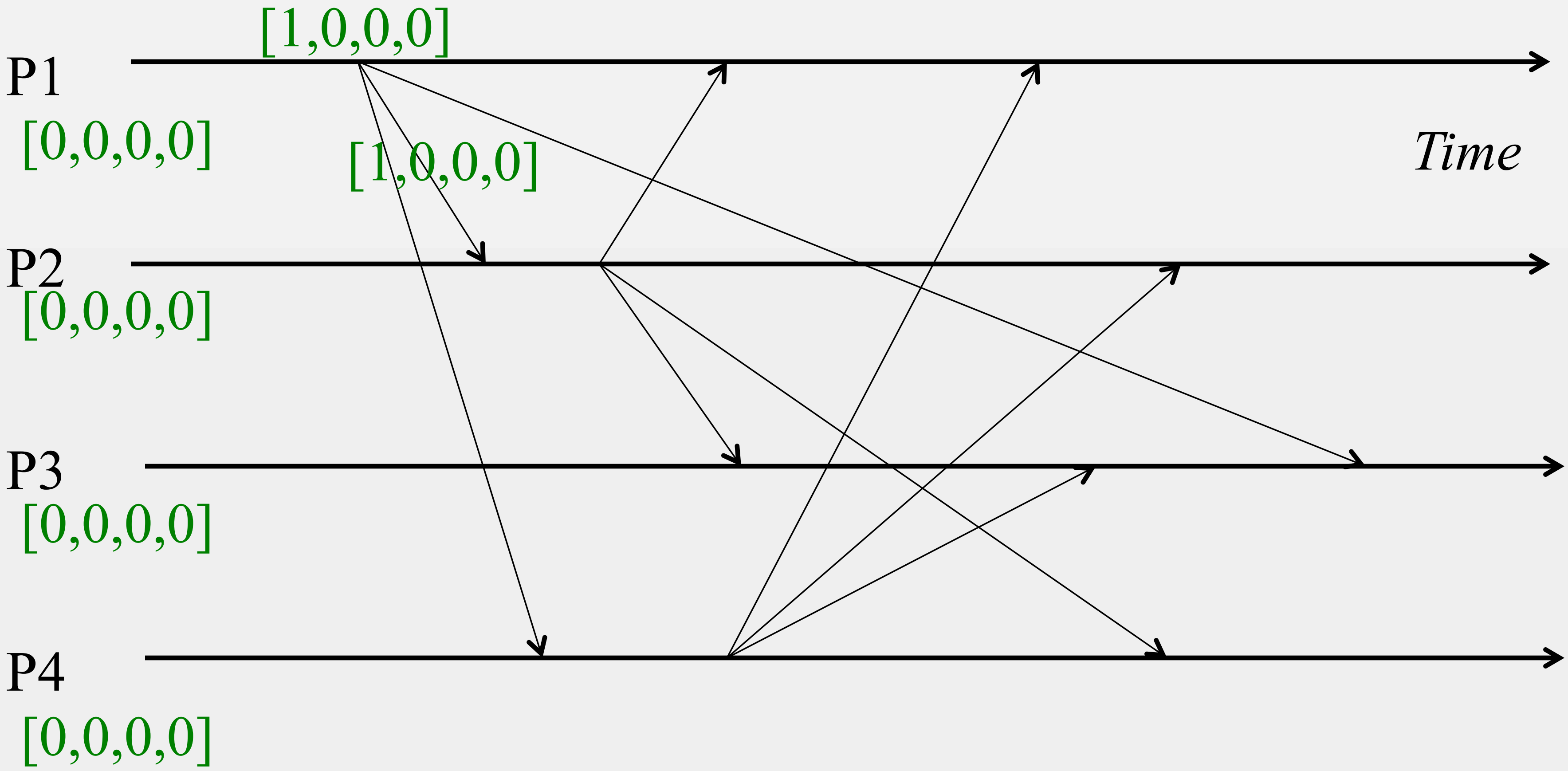
- Multicasts whose send events are causally related, must be received in the same causality-obeying order at all receivers
- Formally
  - *If multicast(g,m) → multicast(g,m') then any correct process that delivers m' would already have delivered m.*
  - *(→ is Lamport's happens-before)*

Causal Multicast: Datastructures

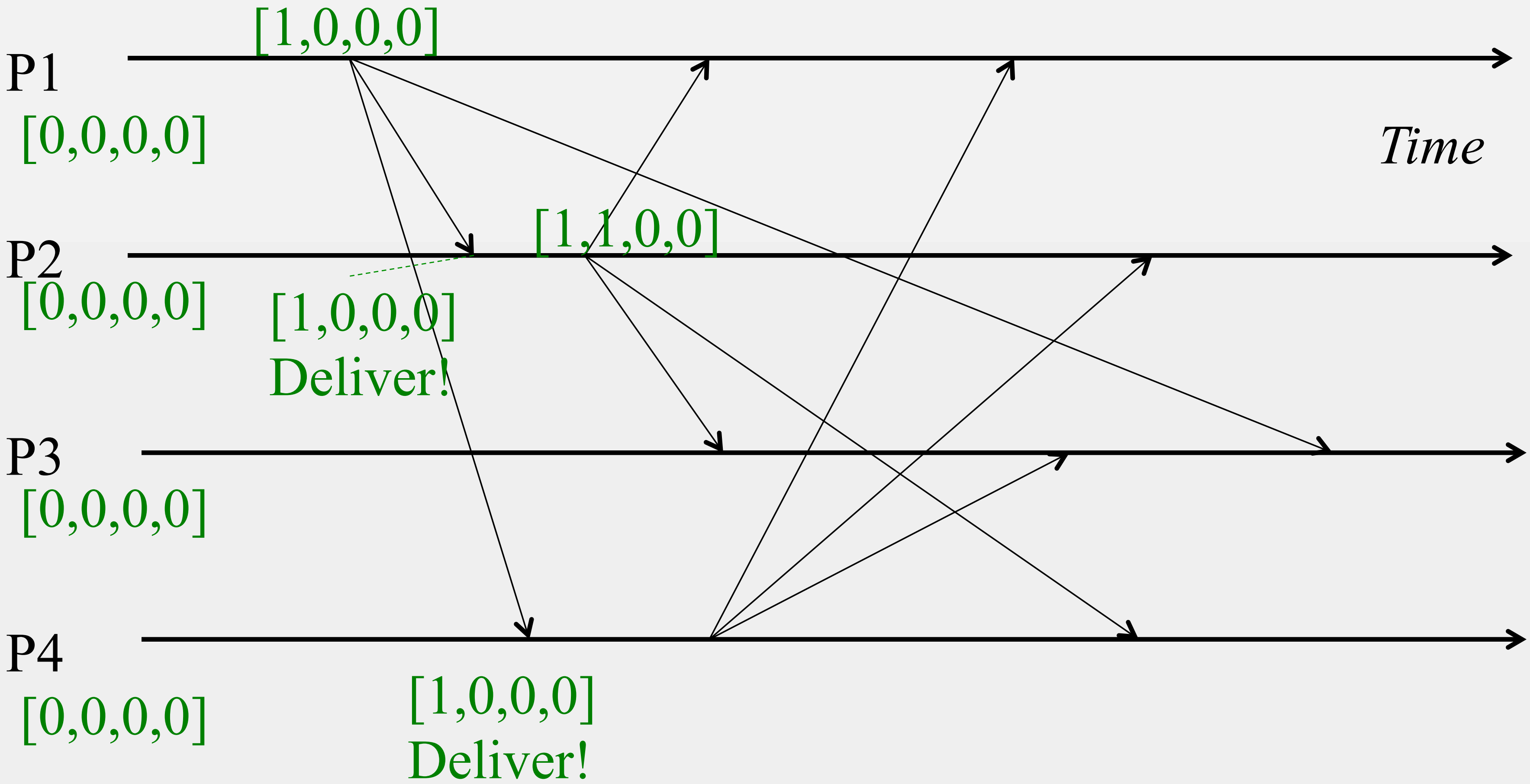
- Each receiver maintains a vector of per-sender sequence numbers (integers)
  - Similar to FIFO Multicast, but updating rules are different
  - Processes P*1* through P*N*
  - P*i* maintains a vector P*i*[1…N] (initially all zeroes)
  - P*i*[*j*] is the latest sequence number P*i* has received from P*j*

Causal Multicast: Updating Rules

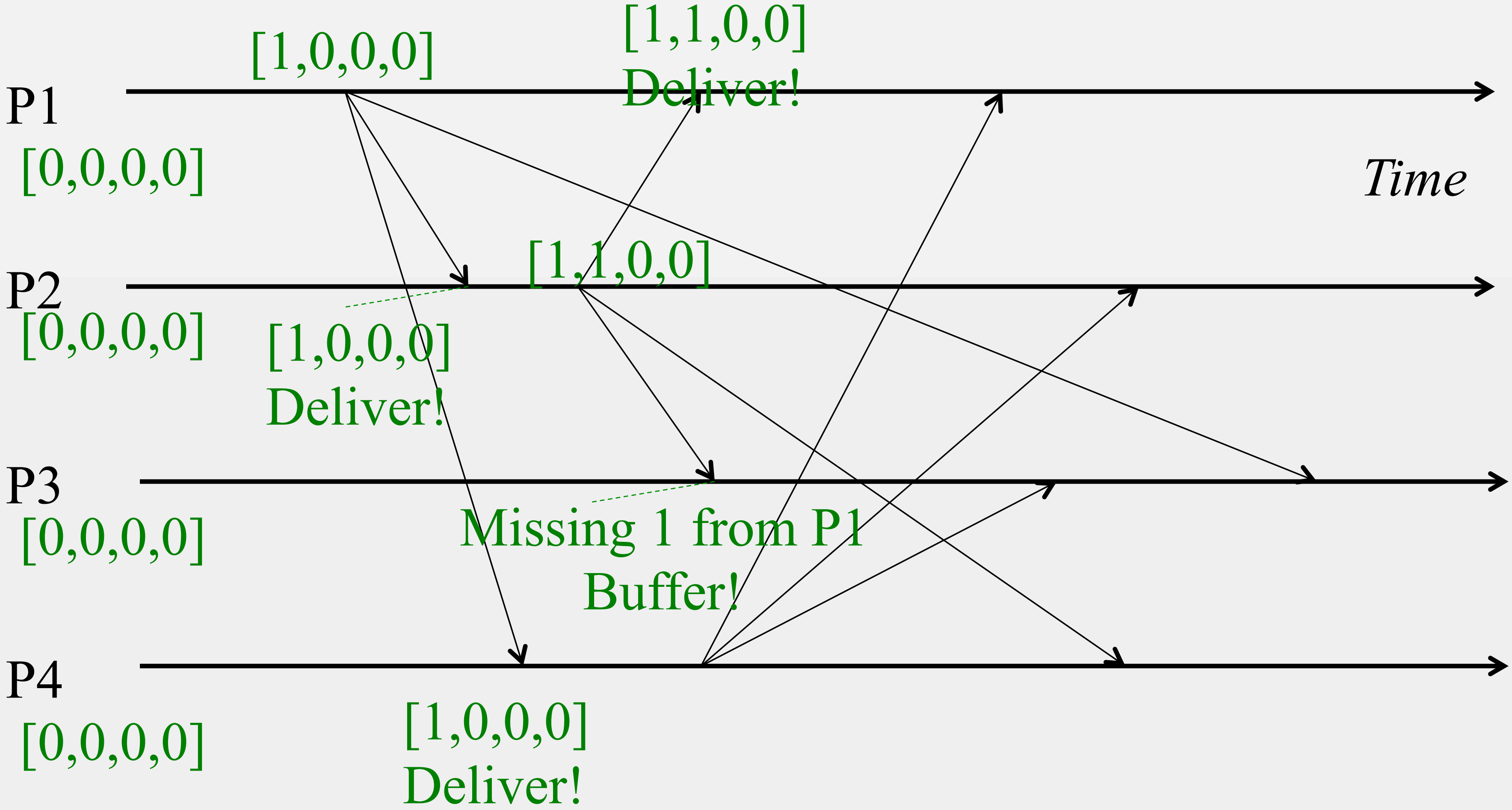
- Send multicast at process Pj:
  - Set Pj[j] = Pj[j] + 1
  - Include new entire vector Pj[1…N] in multicast message as its sequence number
- Receive multicast: If Pi receives a multicast from Pj with vector M[1…N] (= Pj[1…N]) in message, buffer it until both:
  1. This message is the next one Pi is expecting from Pj, i.e.,
     - M[j] = Pi[j] + 1
  2. All multicasts, anywhere in the group, which happened-before M have been received at Pi, i.e.,
     - For all $k \neq j$: M[k] $\leq$ Pi[k]
     - i.e., ***Receiver satisfies causality***
  3. When above two conditions satisfied, deliver M to application and set Pi[j] = M[j]



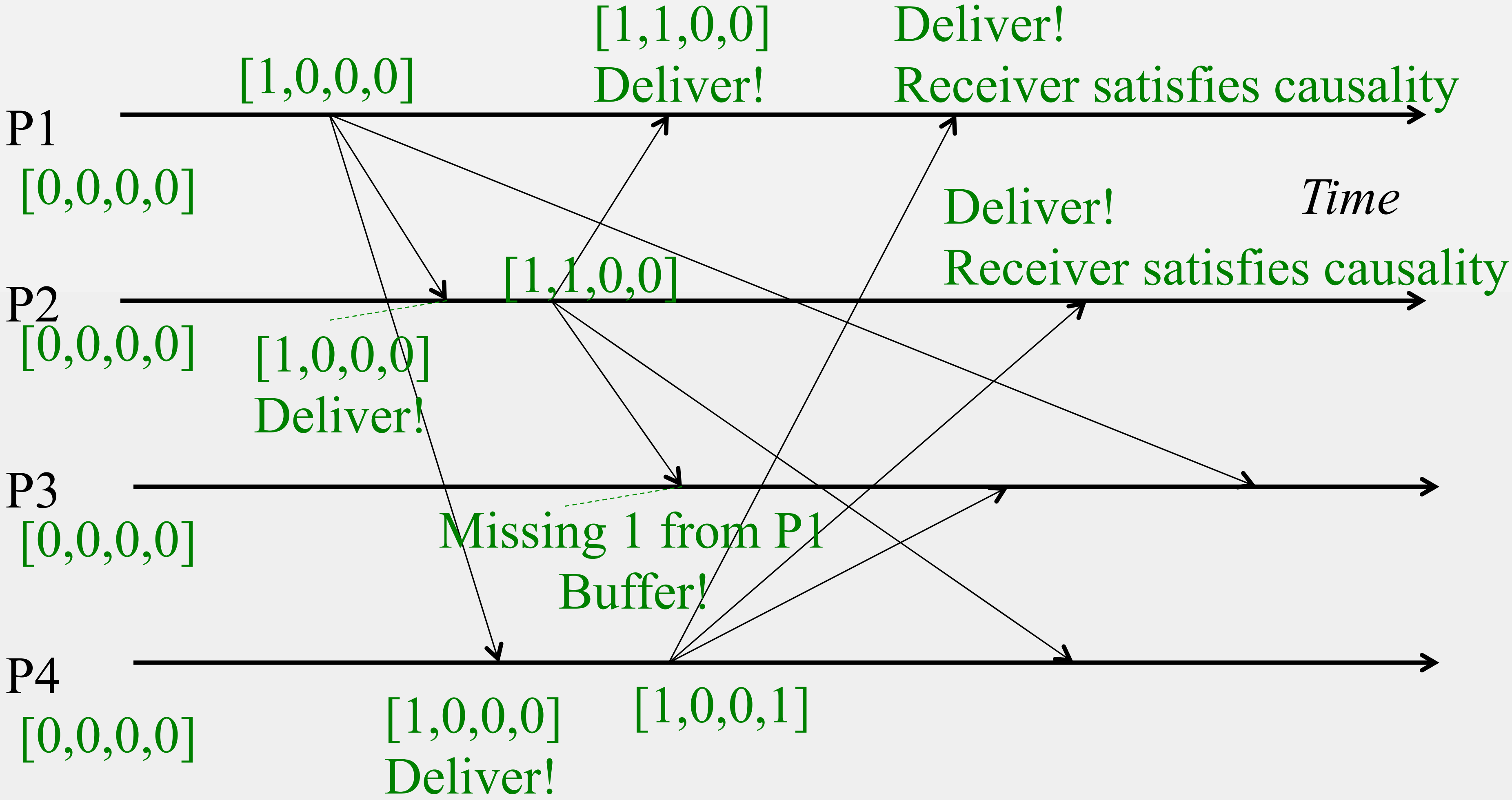
Causal Ordering: Example



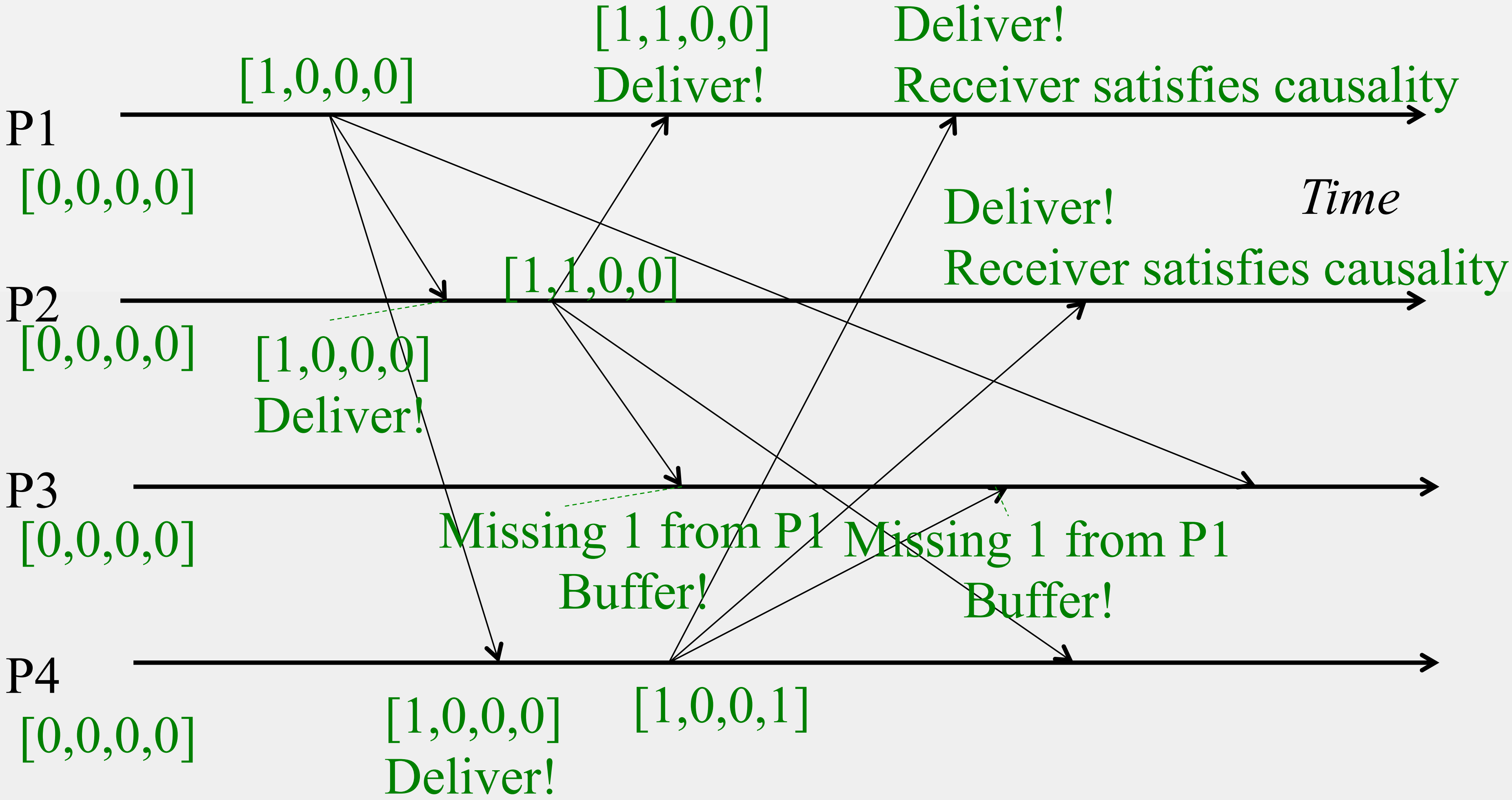
Causal Ordering: Example



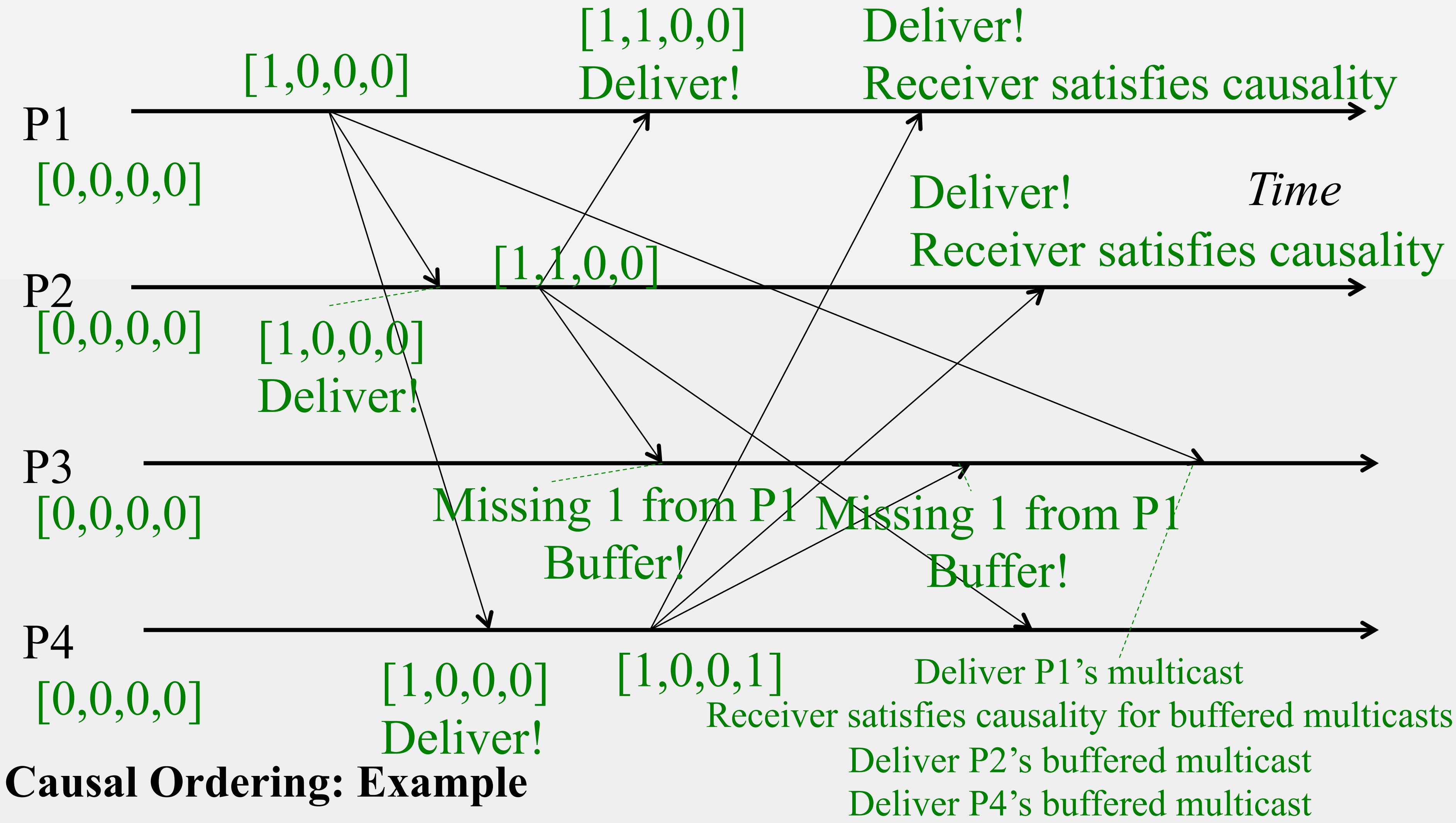
Causal Ordering: Example



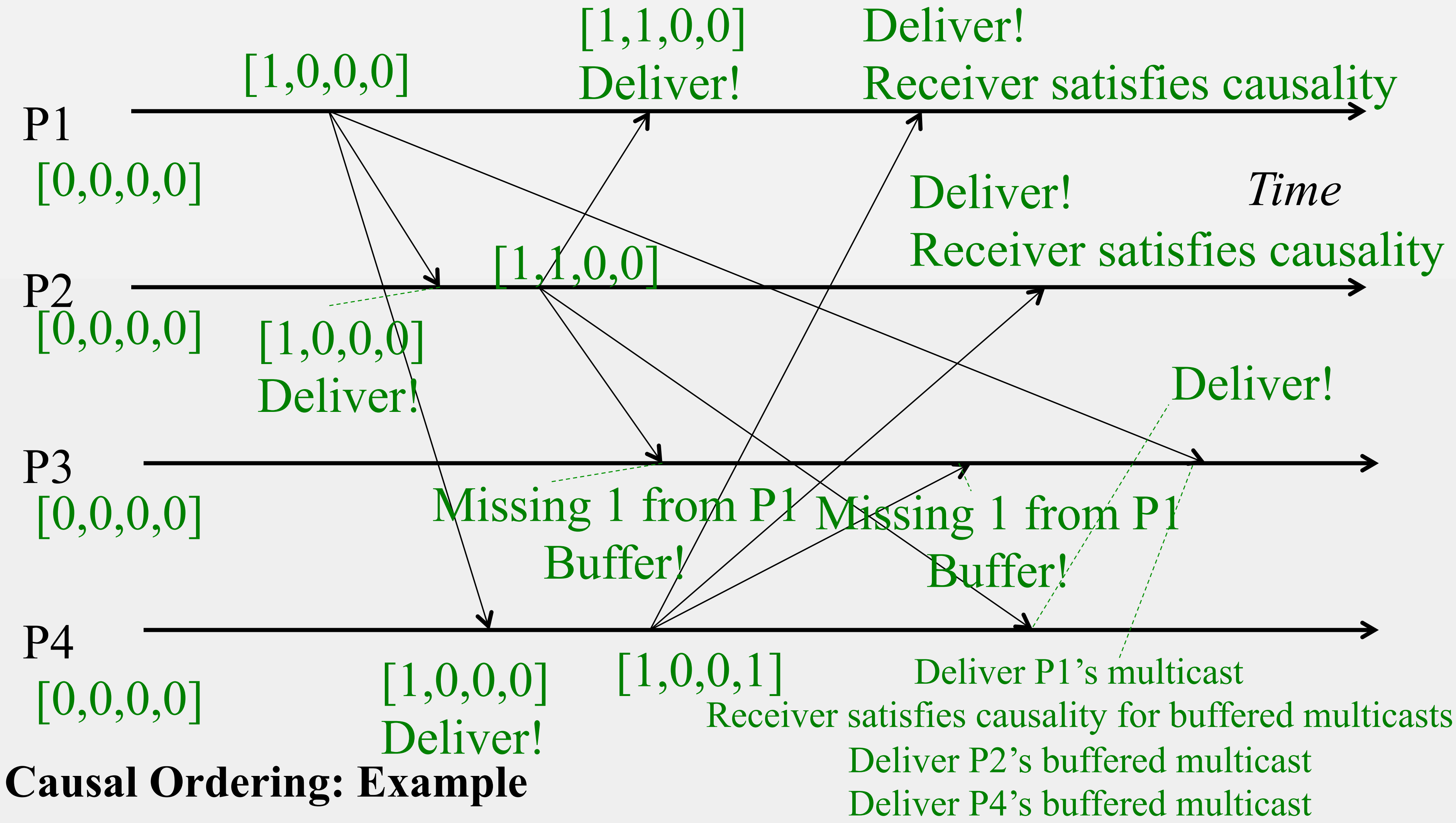
Causal Ordering: Example



Causal Ordering: Example



Causal Ordering: Example



Causal Ordering: Example

Summary: Multicast Ordering

- Ordering of multicasts affects correctness of distributed systems using multicasts
- Three popular ways of implementing ordering
  - FIFO, Causal, Total
- And their implementations
- What about reliability of multicasts?
- What about failures?

Reliable Multicast

- Reliable multicast loosely says that every process in the group receives all multicasts
  - Reliability is orthogonal to ordering
  - Can implement Reliable-FIFO, or Reliable-Causal, or Reliable-Total, or Reliable-Hybrid protocols
- What about process failures?
- Definition becomes vague

Reliable Multicast (under failures)

- Need all *correct* (i.e., non-faulty) processes to receive the same set of multicasts as all other correct processes
  - Faulty processes stop anyway, so we won't worry about them

Implementing Reliable Multicast

- Let's assume we have reliable unicast (e.g., TCP) available to us
- First-cut: Sender process (of each multicast M) sequentially sends a reliable unicast message to all group recipients
- First-cut protocol does not satisfy reliability
  - If sender fails, some correct processes might receive multicast M, while other correct processes might not receive M

REALLY Implementing Reliable Multicast

- Trick: Have receivers help the sender

1. Sender process (of each multicast M) sequentially sends a reliable unicast message to all group recipients
2. When a receiver receives multicast M, it also sequentially sends M to all the group's processes

Analysis

- Not the most efficient multicast protocol, but reliable
- Proof is by contradiction
- Assume two correct processes P*i* and P*j* are so that P*i* received a multicast M and P*j* did not receive that multicast M
  - Then P*i* would have sequentially sent the multicast M to all group members, including P*j*, and P*j* would have received M
  - A contradiction
  - Hence our initial assumption must be false
  - Hence protocol preserves reliability

Virtual Synchrony or View Synchrony

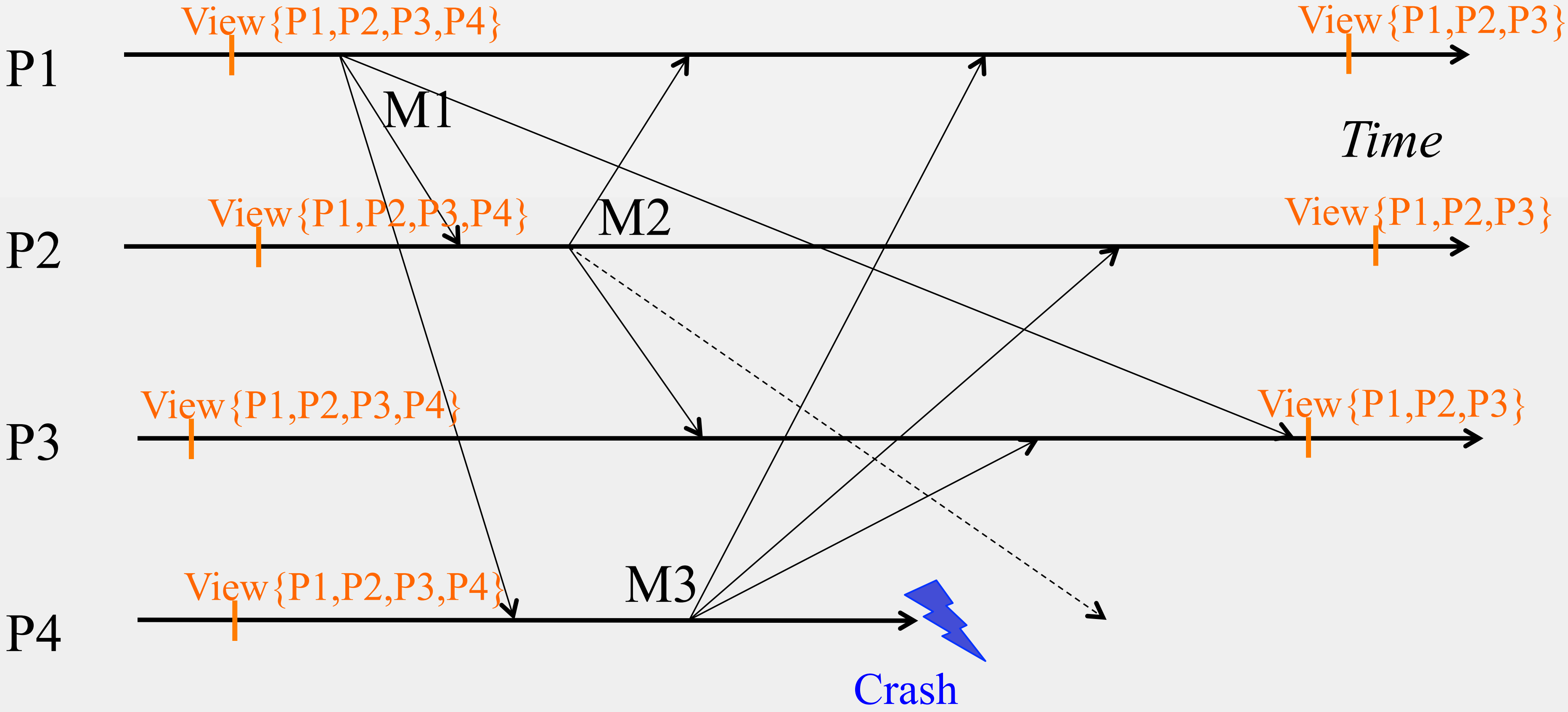
- Attempts to preserve multicast ordering and reliability in spite of failures
- Combines a membership protocol with a multicast protocol
- Systems that implemented it (like Isis Systems) have been used in NYSE, French Air Traffic Control System, Swiss Stock Exchange

Views

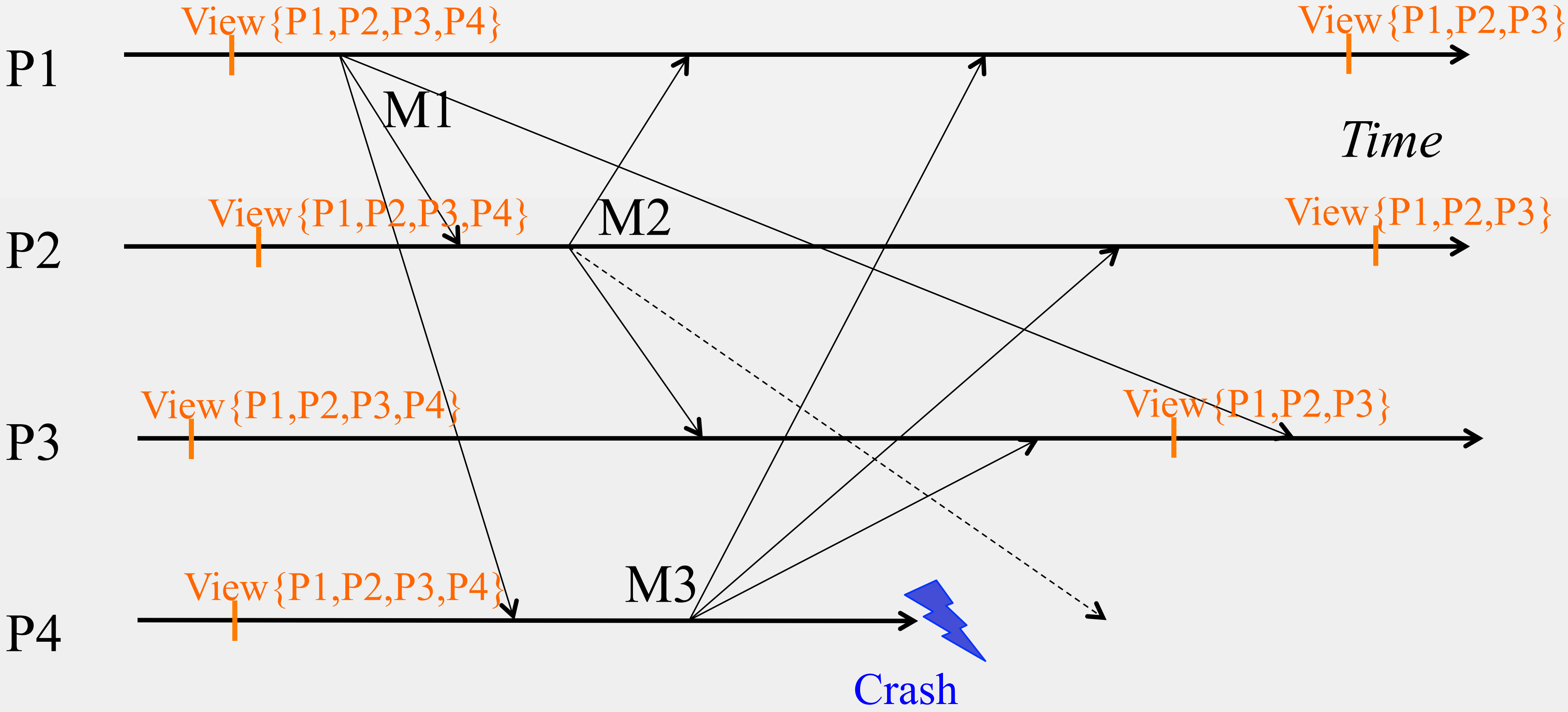
- Each process maintains a membership list
- The membership list is called a *View*
- An update to the membership list is called a *View Change*
  - Process join, leave, or failure
- Virtual synchrony guarantees that all **view changes are delivered in the same order at all correct processes**
  - If a correct P1 process receives views, say {P1}, {P1, P2, P3}, {P1, P2}, {P1, P2, P4} then
  - Any other correct process receives the *same sequence* of view changes (after it joins the group)
    - P2 receives views {P1, P2, P3}, {P1, P2}, {P1, P2, P4}
- Views may be delivered at different physical times at processes, but they are delivered in the same order

VSync Multicasts

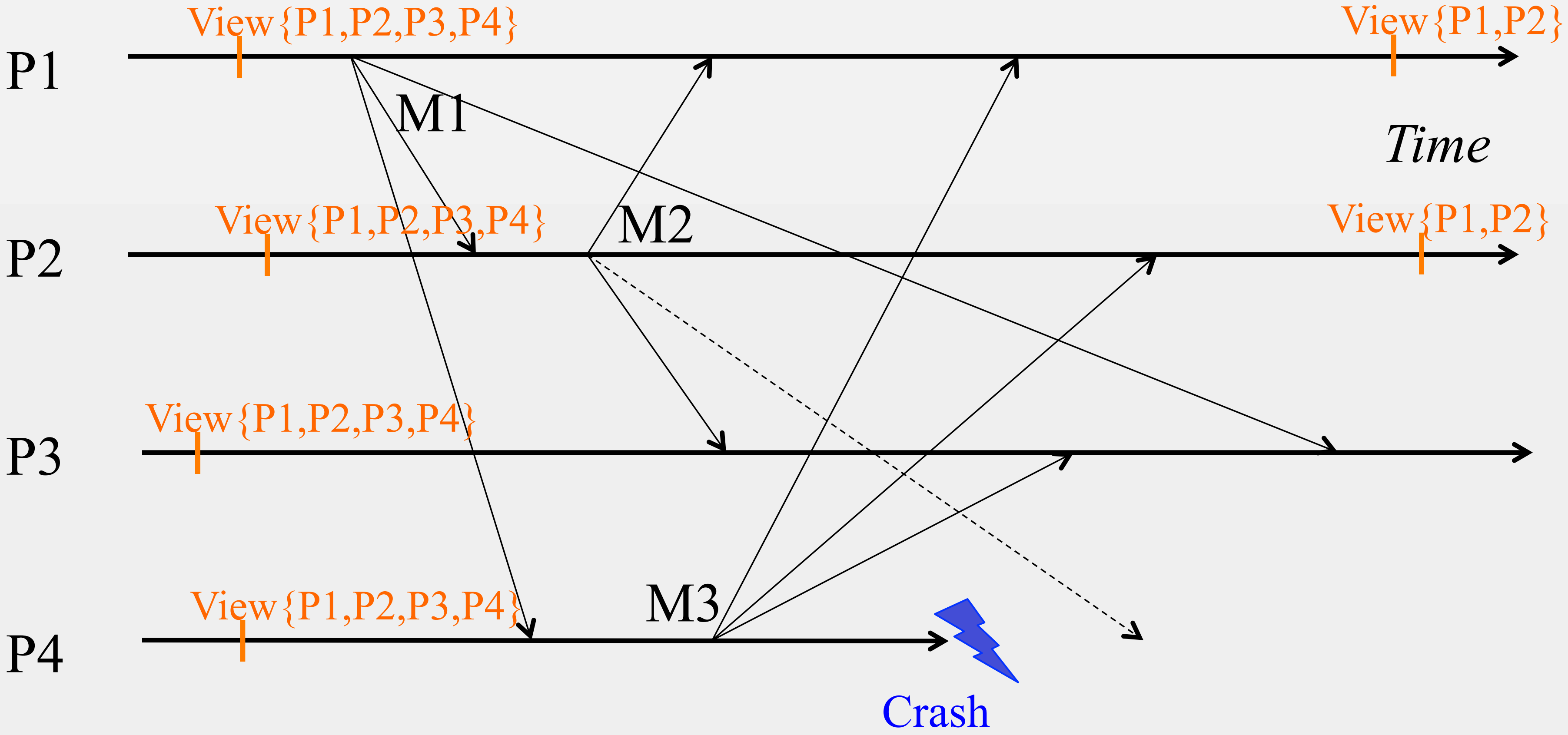
- A multicast M is said to be “delivered in a view V at process P*i*” if
  - P*i* receives view V, and then sometime before P*i* receives the next view it delivers multicast M
- Virtual synchrony ensures that
  1. **The set of multicasts delivered in a given view is the same set at all correct processes that were in that view**
     - *What happens in a View, stays in that View*
  2. The sender of the multicast message also belongs to that view
  3. If a process P*i* does not deliver a multicast M in view V while other processes in the view V delivered M in V, then P*i* will be *forcibly removed* from the next view delivered after V at the other processes



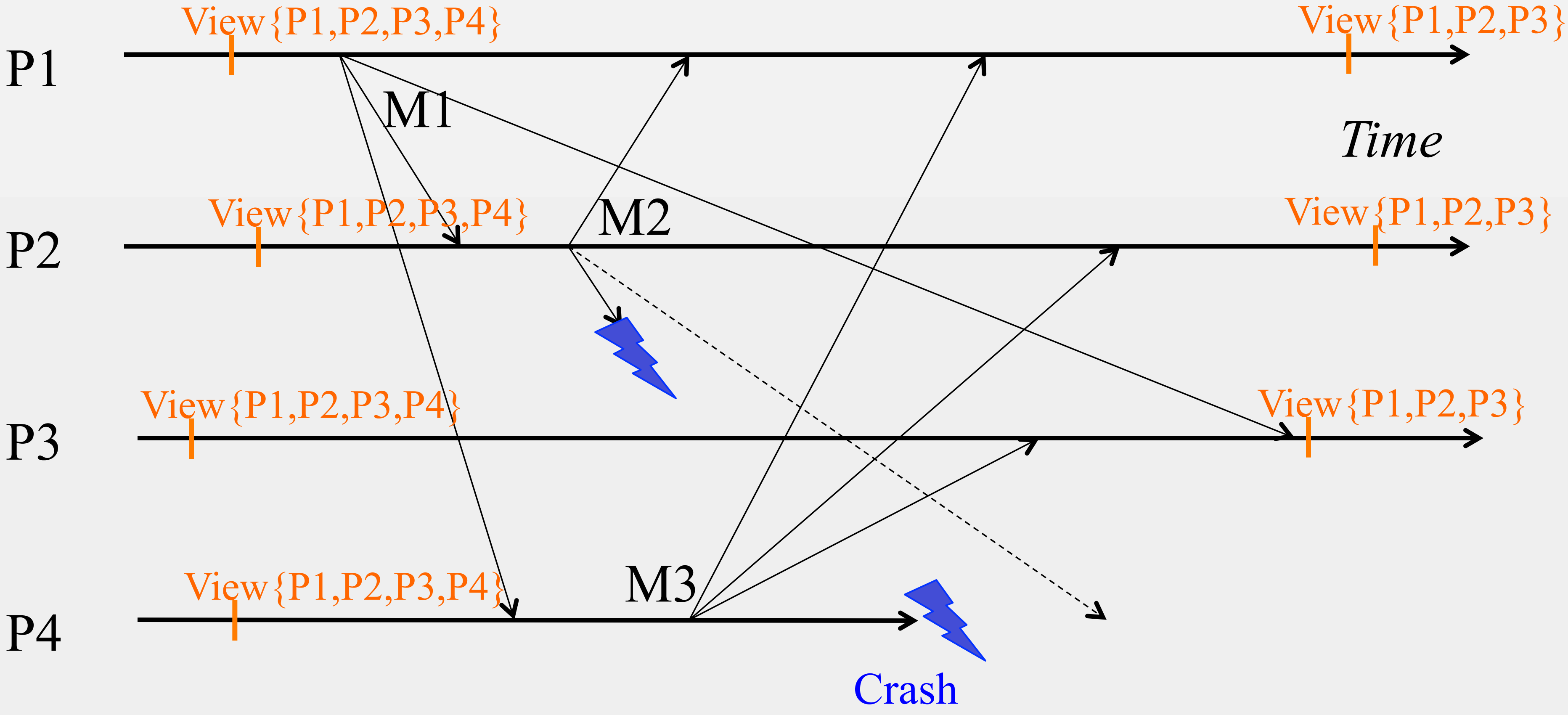
Satisfies virtual synchrony



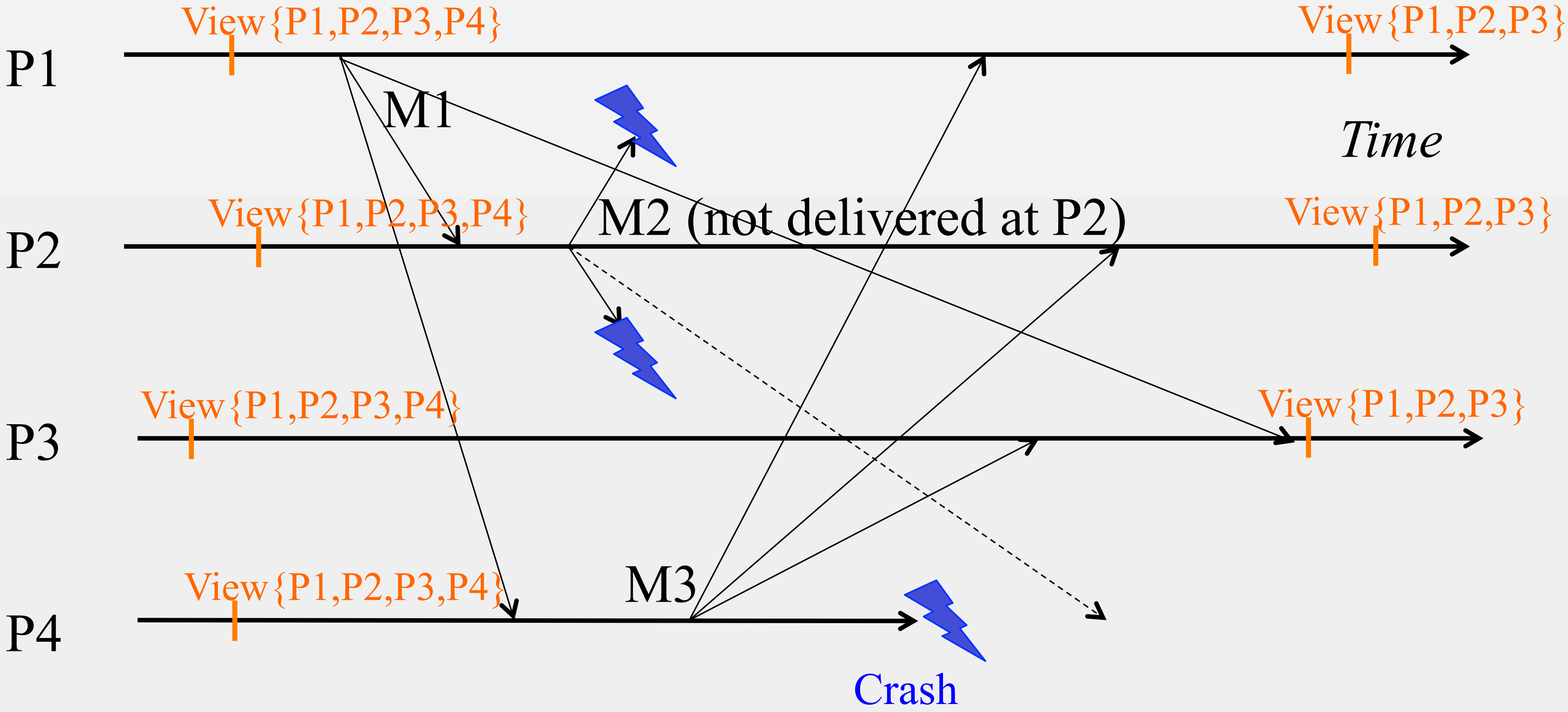
Does not satisfy virtual synchrony



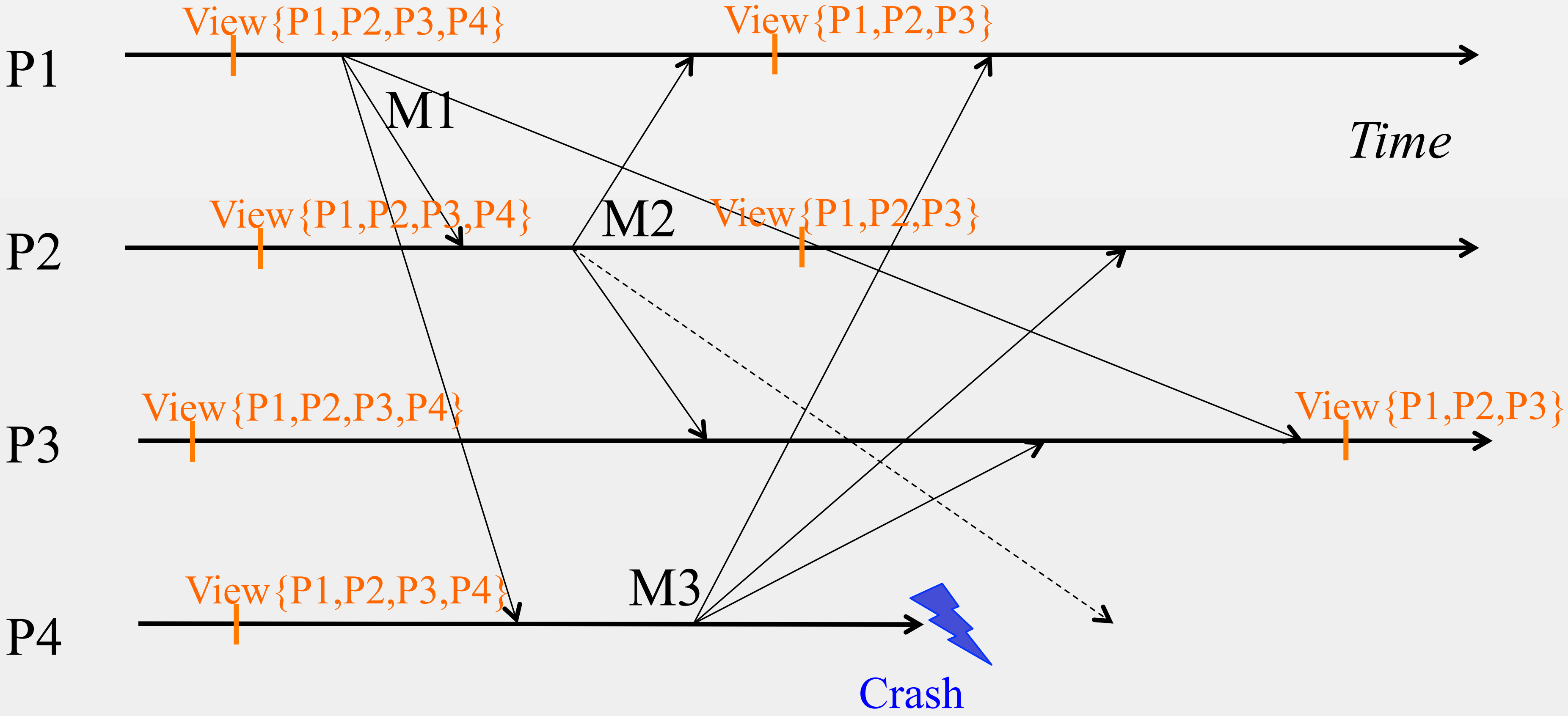
Satisfies virtual synchrony



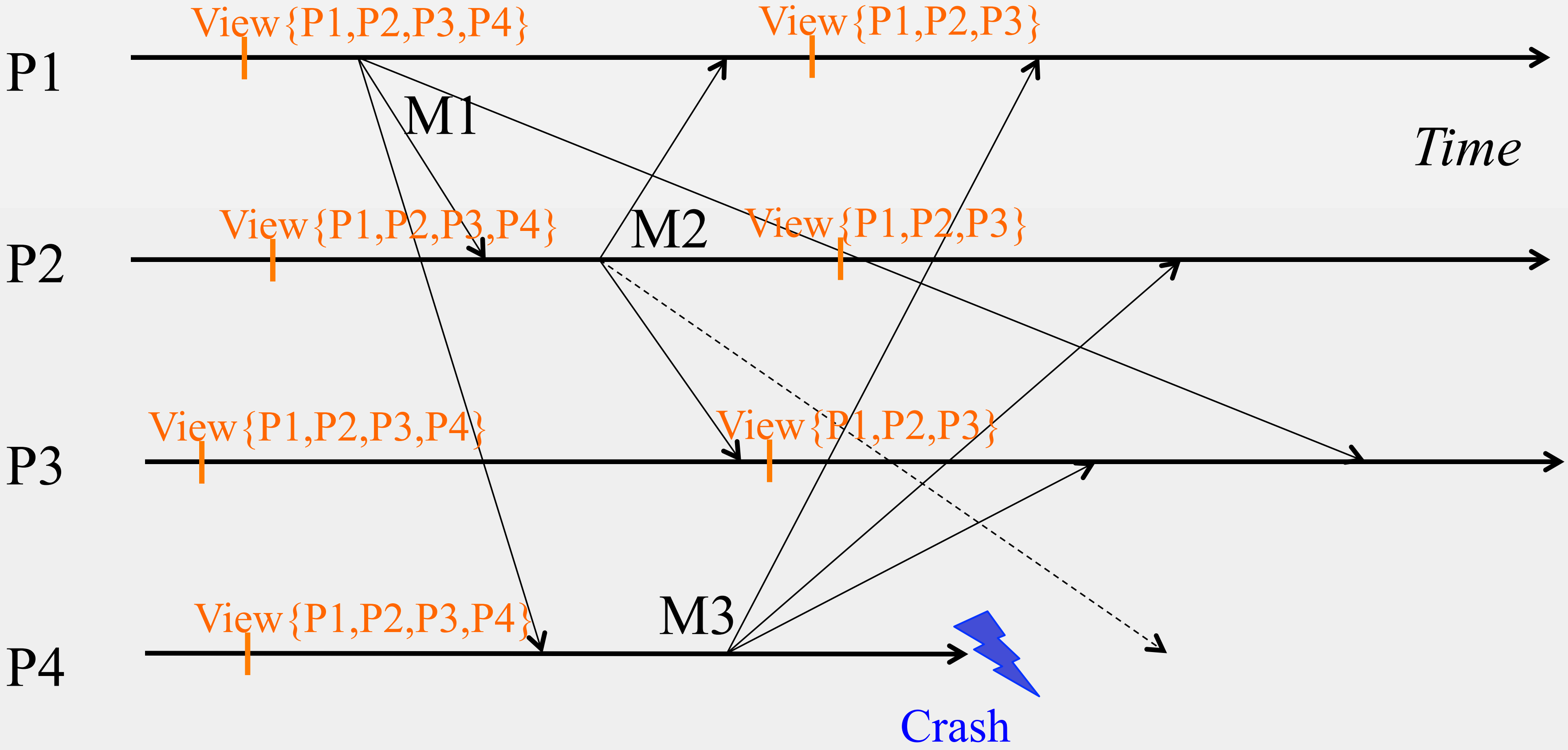
Does not satisfy virtual synchrony



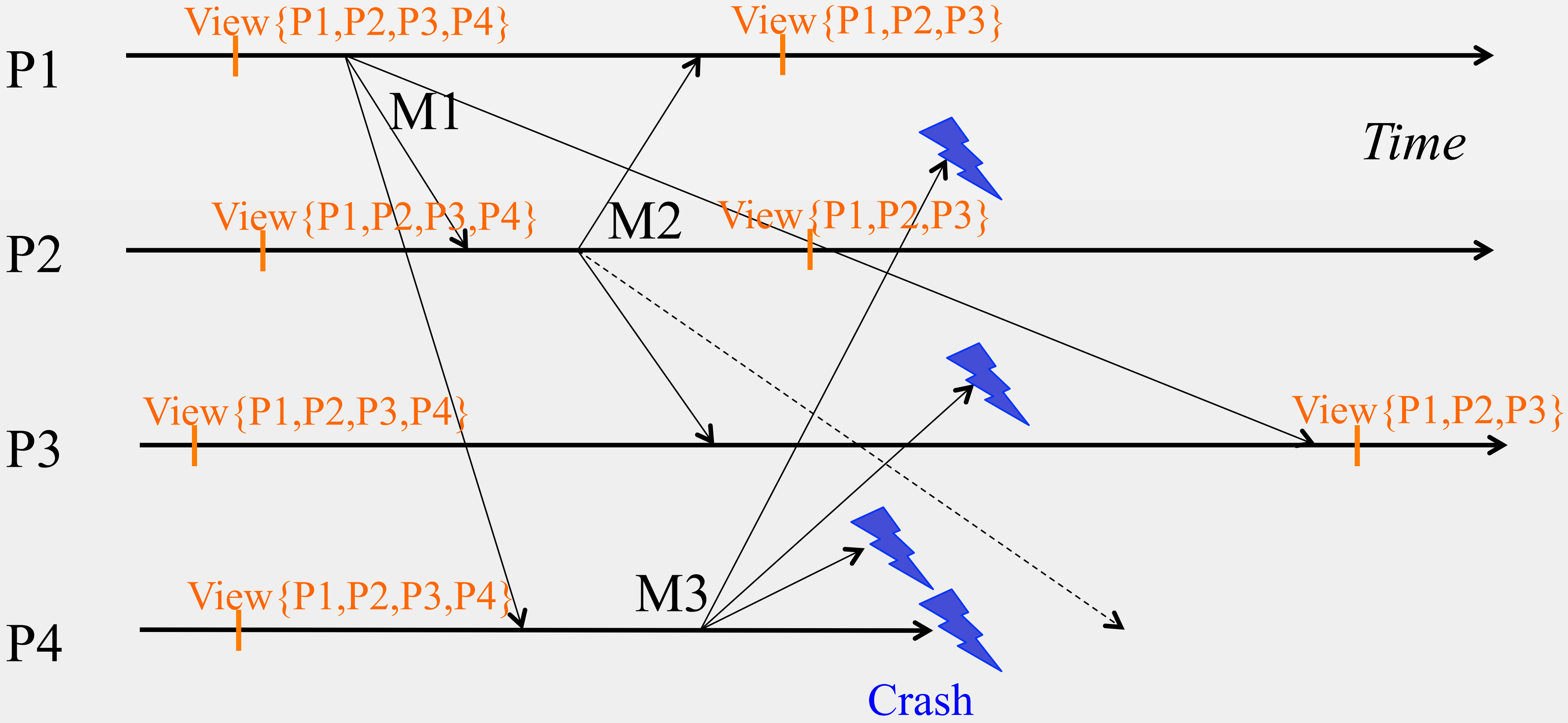
Satisfies virtual synchrony



Does not satisfy virtual synchrony



Does not satisfy virtual synchrony



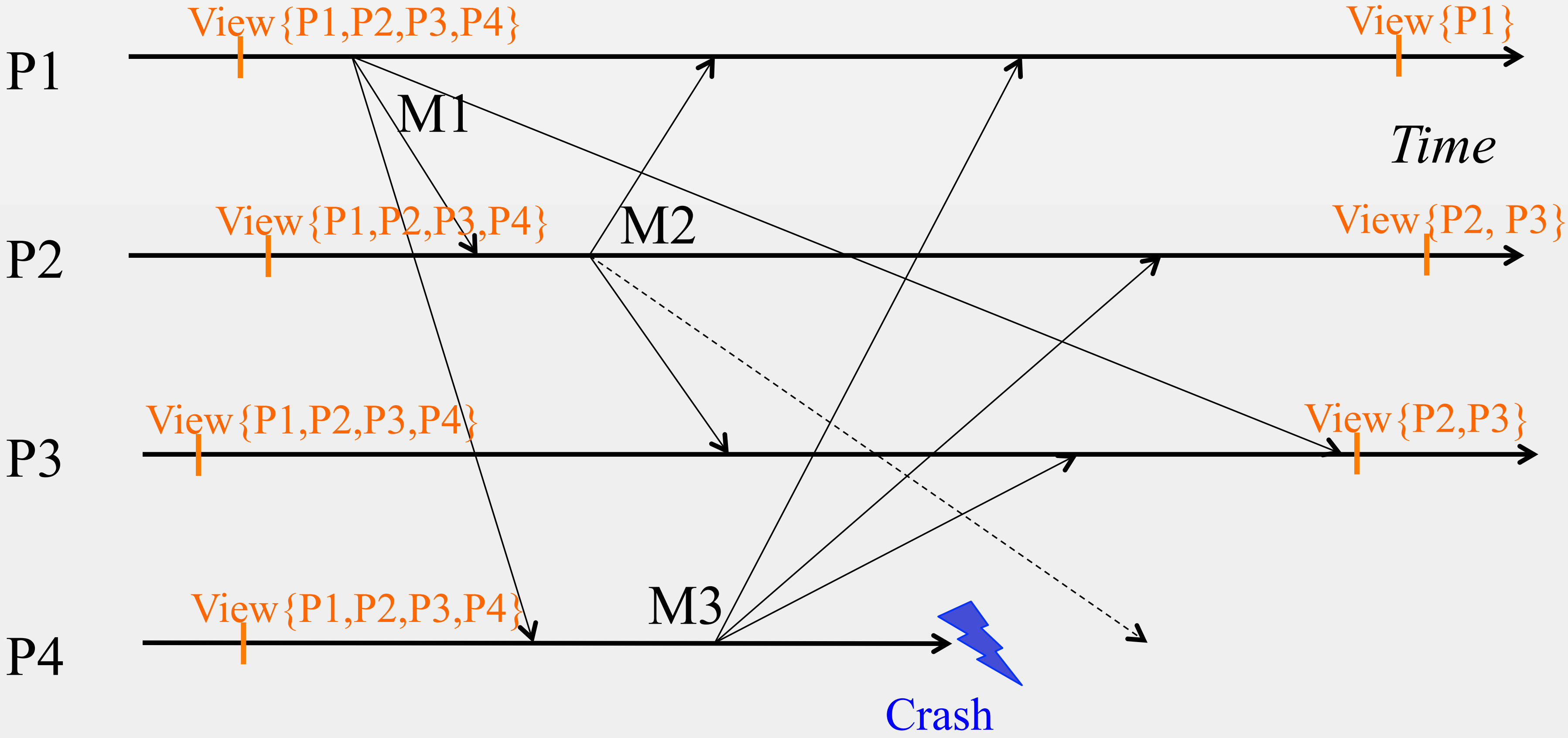
Satisfies virtual synchrony

What about Multicast Ordering?

- Again, orthogonal to virtual synchrony
- The set of multicasts delivered in a view can be ordered either
  - FIFO
  - Or Causally
  - Or Totally
  - Or using a hybrid scheme

About that name

- Called “virtual synchrony” since in spite of running on an asynchronous network, it gives the appearance of a synchronous network underneath that obeys the same ordering at all processes
- So can this virtually synchronous system be used to implement consensus?
- No! VSync groups susceptible to partitioning
  - E.g., due to inaccurate failure detections



Partitioning in View synchronous systems

Summary

- Multicast an important building block for cloud computing systems
- Depending on application need, can implement
  - Ordering
  - Reliability
  - Virtual synchrony

Announcements

- Midterm next Tuesday
- Locations:
  - DCL 1320: if your last name starts with A-Q
  - 1 Noyes 217 (Map): if your last name starts with R-Z
- Material through lecture 12 (Time and Ordering)