1

Group Communication

CMPS 4760/6760: Distributed Systems

Acknowledgement: slides adapted from Indranil Gupta's lecture notes:
https://courses.engr.illinois.edu/cs425/fa2019/index.html

Coordination in Distribution Systems

- Distributed Mutual Exclusion (15.2)
- Leader Election (15.3)
- Group communication (15.4)
- Consensus (15.5)

Communication Forms

- Multicast: message sent to a group of processes
  - By issuing a single multicast operation
- Broadcast: message sent to to all processes
- Unicast: message sent to a single 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 vs. Unicast

- Much than a convenience for the programmer
- More efficient use of bandwidth, minimizing the delay
  - Each message sent no more than once over any communication link
  - a distribution tree and hardware multicast support
- Delivery guarantees
  - If the sender fails halfway through sending, then some members of the group may receive the message while others do not.
  - The relative ordering of two messages delivered to any two group members is undefined

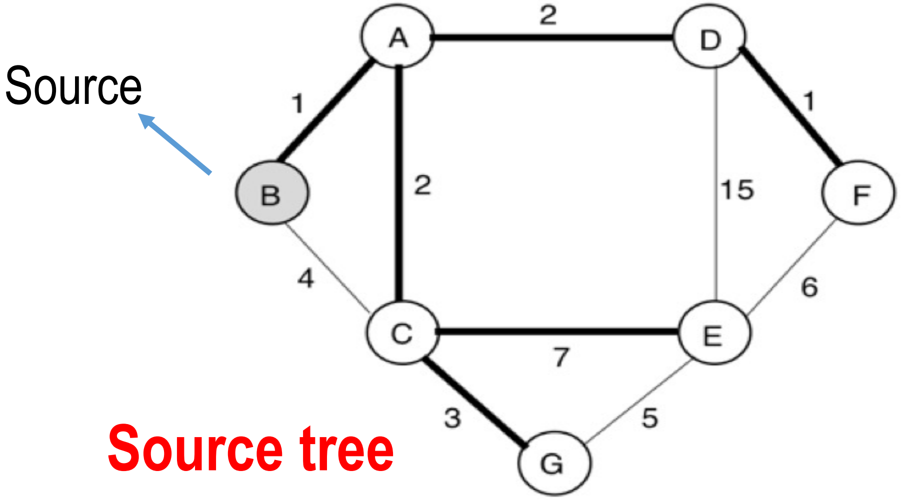
Multicast vs. Unicast

- Example: sending the same message from a computer in London to two computers on the same Ethernet in Palo Alto

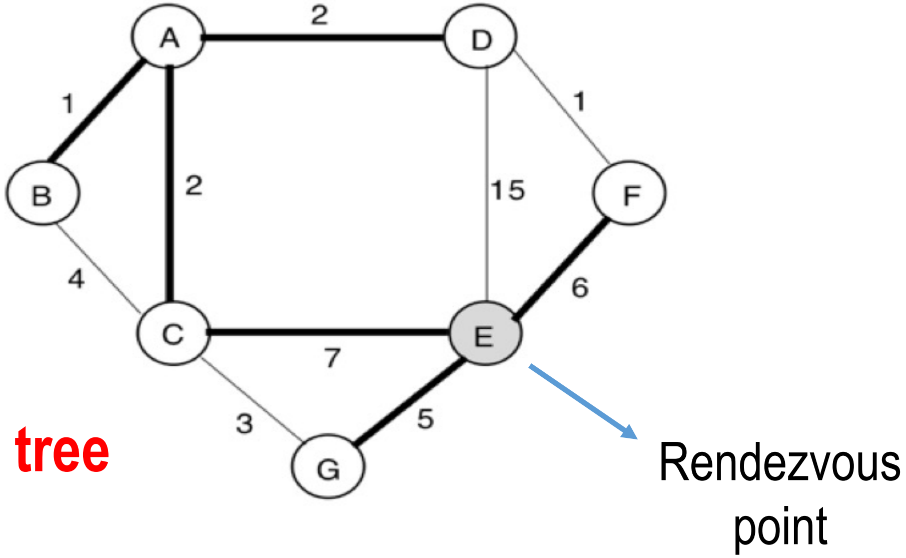
  (a) by two separate UDP sends

  (b) by a single IP multicast operation: a single copy sent from London to a router in Palo Alto, followed by a hardware multicast via the Ethernet to destinations

Multicast Trees



Source tree



Shared tree

- A shortest path tree rooted at source B
- The tree will be different for a difference source
- Routers replicate a packet and forward it to each of their neighbors in the tree

- All routers forward traffic to RP, which forwards them to the appropriate destinations via a common shortest path tree rooted at the RP

Group Communication

- IP Multicast
  - Unreliable multicast
  - Weak membership management
- Group Communication
  - Reliability and ordering guarantees (15.4)
  - Membership management (18.2)
- Group communication vs. IP multicast is like TCP vs. IP

Group Communication

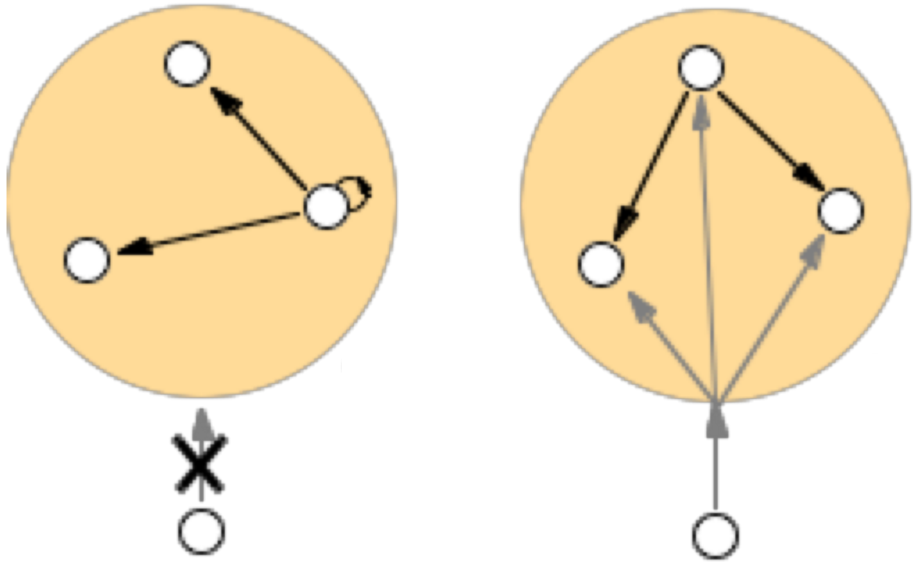
- Programming Model (6.2.1-6.2.2)
- Case study: JGroups (6.2.3)
- Reliable and ordered multicast (15.4)
- View-synchronous group communication (18.2)

Programming Model

- Process Groups
  - Messages sent to the processes and no further support for dispatching provided
  - Messages are unstructured byte arrays with no support for marshalling
  - Similar to services provided by sockets
  - Example: JGroups toolkit
- Object Groups
  - A collection of objects (normally instances of the same class)
  - Each has a local proxy for the group
  - Example: CORBA Group RMI
    - transparent mode: local proxy returns the first available response to client
    - non-transparent mode: the client object can access all the responses returned by the group members

Programming Model

- Closed vs. open groups
- Overlapping vs. non-overlapping groups



Closed group   Open group

Reliable Multicast

- integrity: message received is the same as the one sent and no duplicates
- validity: any outgoing message is eventually delivered
- agreement: if the message is delivered to one process, it is delivered to all processes

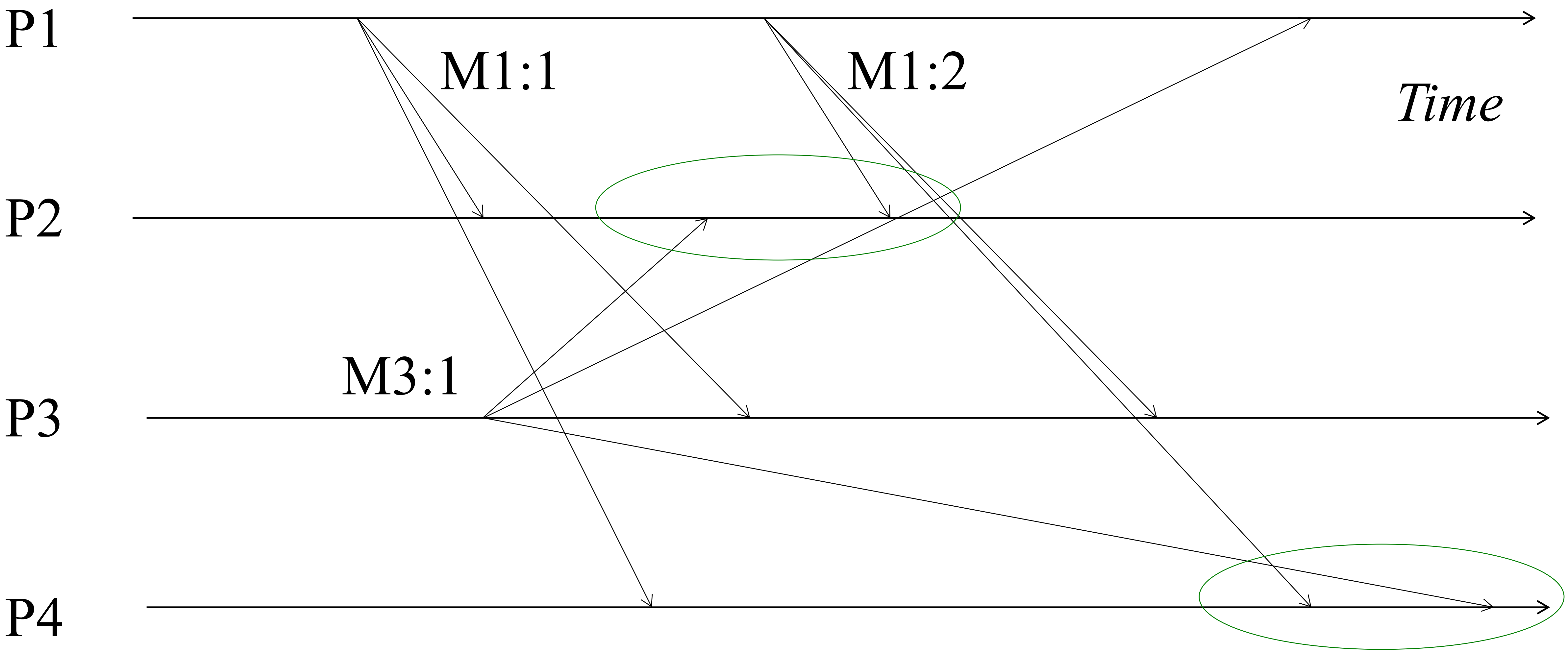
Ordered Multicast

- 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

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



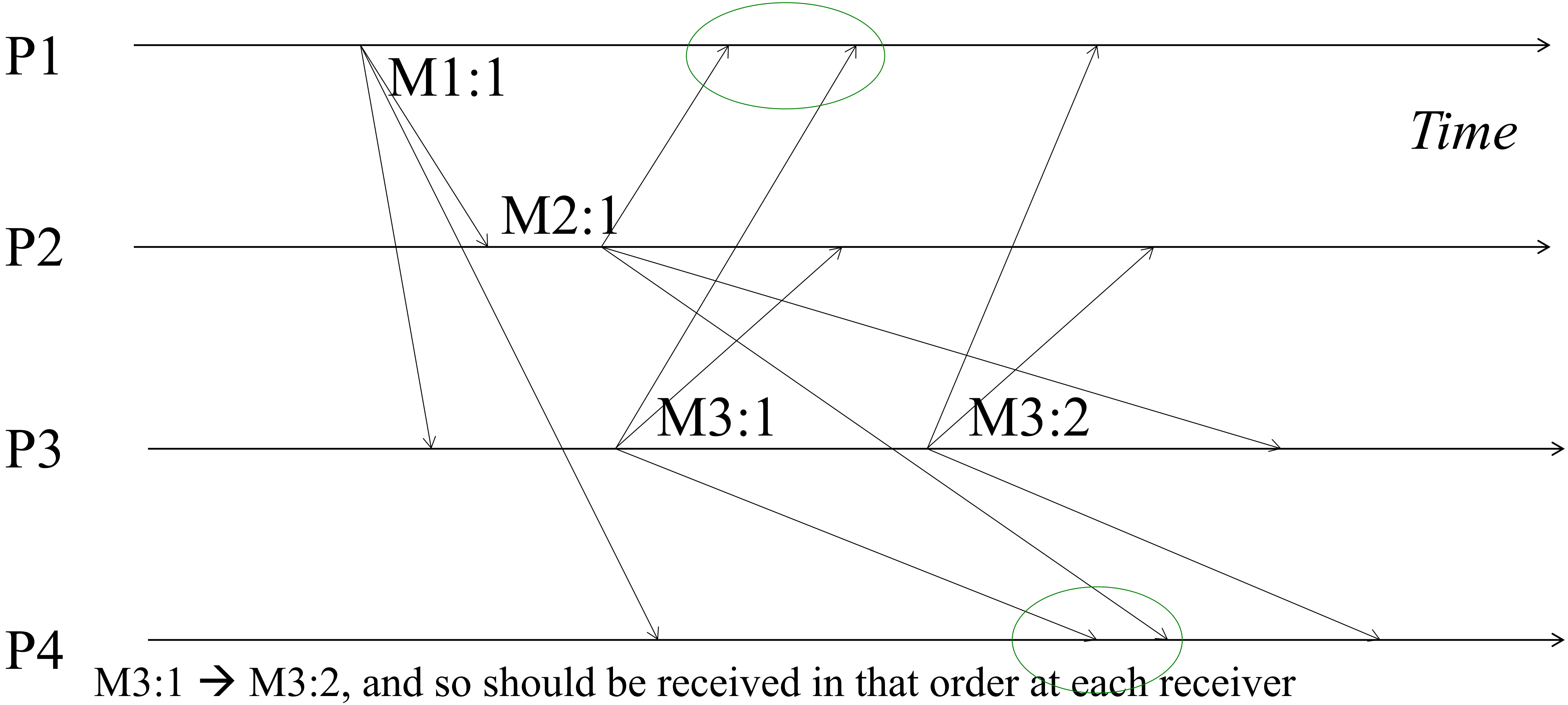
M1:1 and M1:2 should be received in that order at each receiver

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

Causal Ordering

- Multicasts whose send events are causally related, must be received in the same causality-obeying order at all receivers
- Formally
  - *If* $multicast(g, m) \rightarrow multicast(g, m')$ *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 $\rightarrow$ M'
  - Then a multicast protocol that implements causal ordering will obey FIFO ordering since M $\rightarrow$ M'
- Reverse is not true! FIFO ordering does not imply causal ordering.

Ordered Multicast Example: a bulletin board

Bulletin board: os.interesting

Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

FIFO ordering

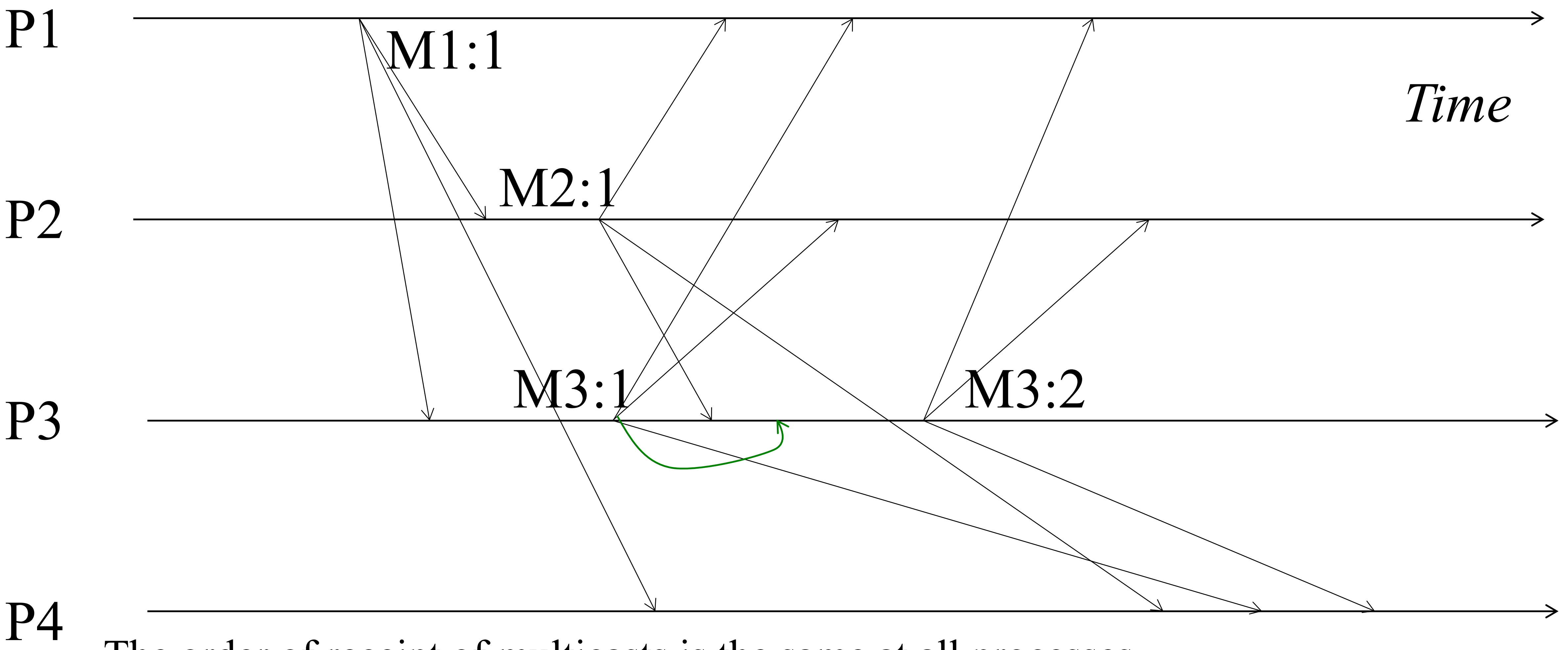
Total ordering

Causal ordering

Total Ordering

- 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.

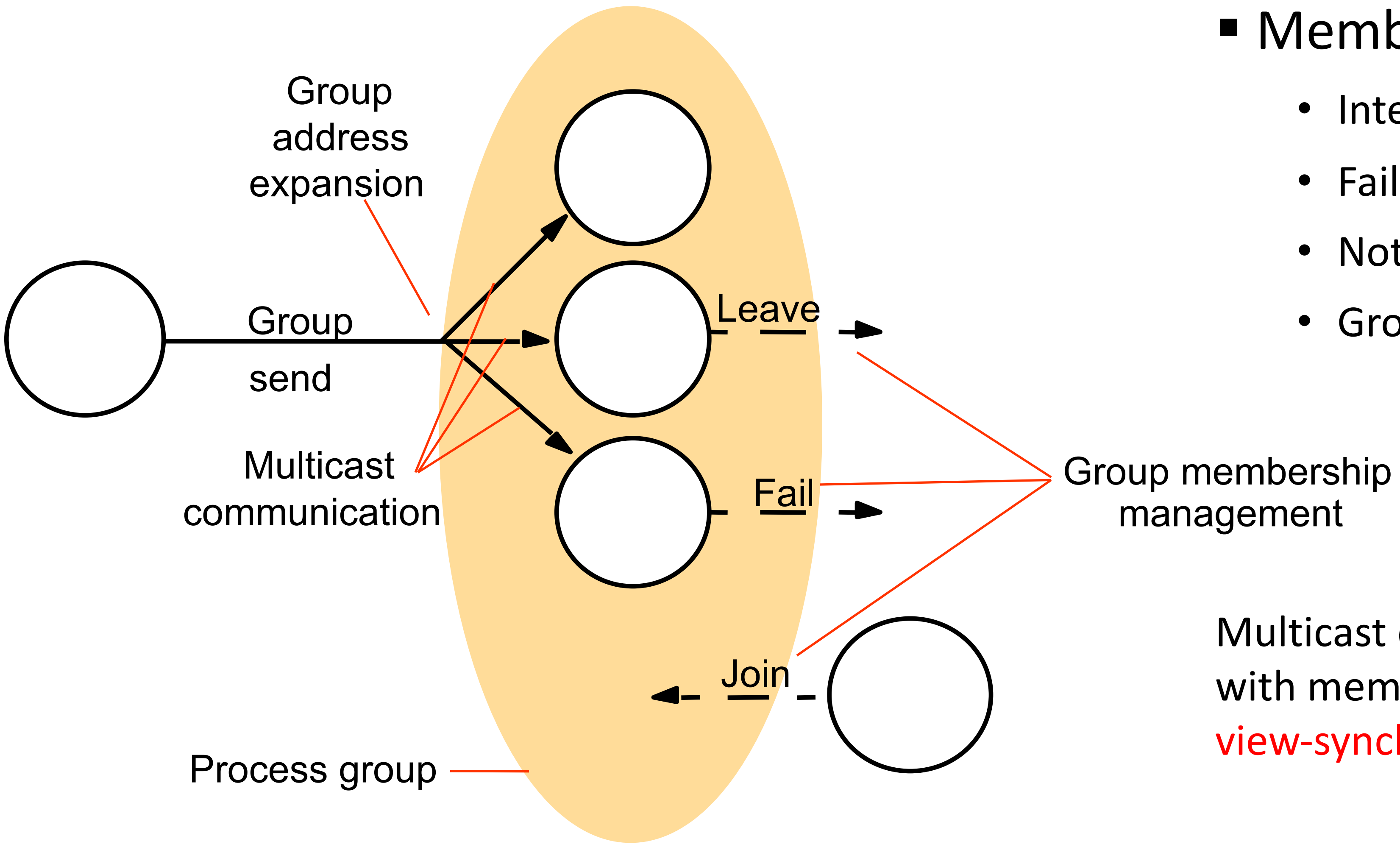
M1:1, then M2:1, then M3:1, then M3:2

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

Group Membership Management



- Membership service provides
  - Interface for membership changes
  - Failure detection
  - Notification of membership changes
  - Group address expansion

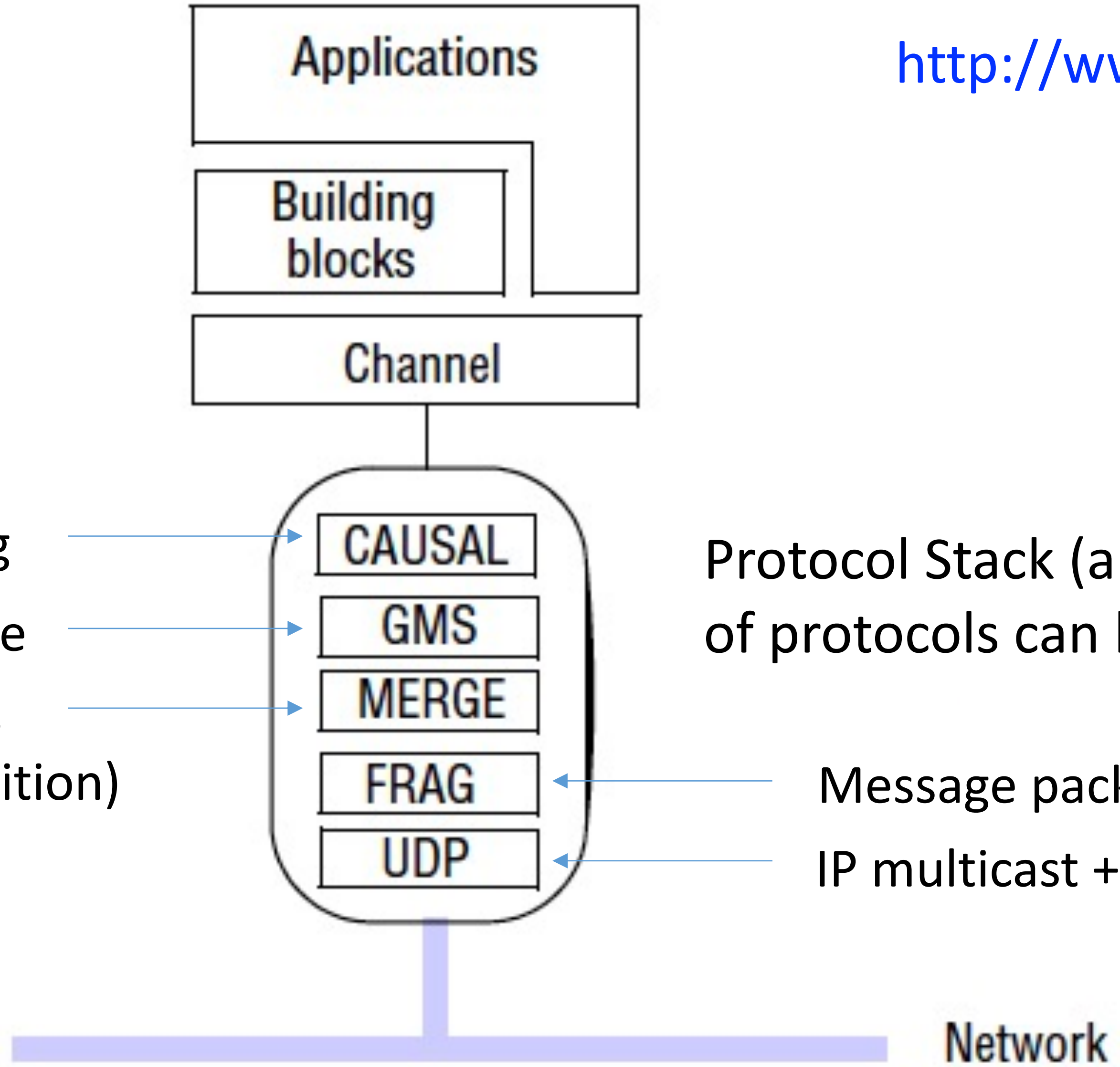
Multicast delivery should be coordinated with membership change => view-synchronous group communication

Group Communication

- Programming Model (6.2.1-6.2.2)
- Case study: JGroups (6.2.3)
- Reliable and ordered multicast (15.4)
- View-synchronous group communication (18.2)

Case study: the JGroups toolkit

http://www.jgroups.org/



Causal ordering

Membership service

Merging subgroups (after network partition)

Protocol Stack (a wide range of protocols can be combined)

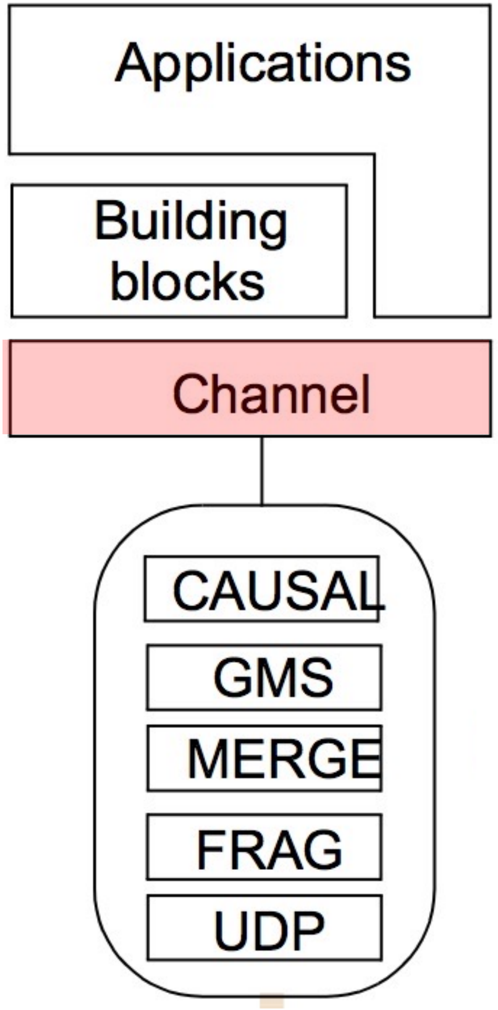
Message packetization

IP multicast + UDP

Java class *FireAlarmJG*

```
import org.jgroups.JChannel;

public class FireAlarmJG {
    public void raise() {
        try {
            JChannel channel = new JChannel();
            channel.connect("AlarmChannel");
            Message msg = new Message(null, null, "Fire!");
            channel.send(msg);
        }
        catch(Exception e) {
        }
}
```

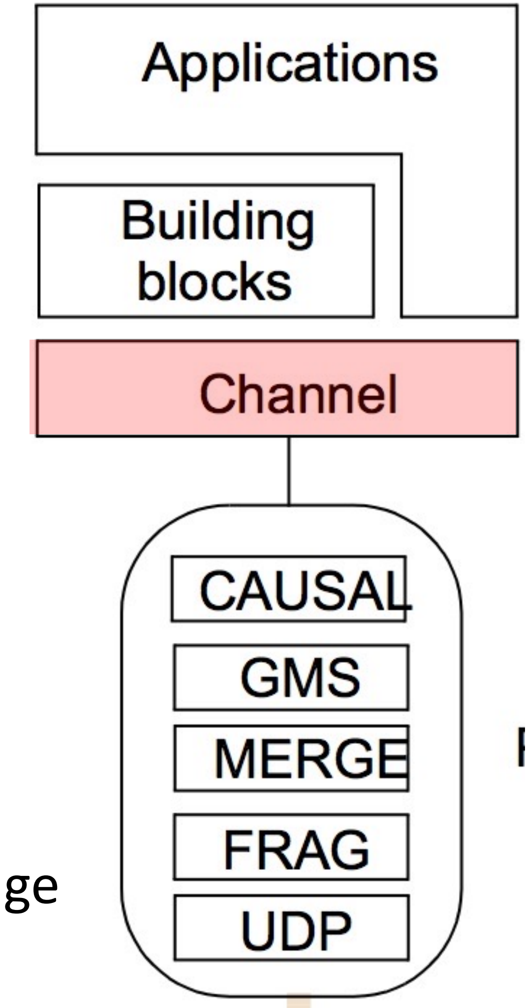
destination source



Java class *FireAlarmConsumerJG*

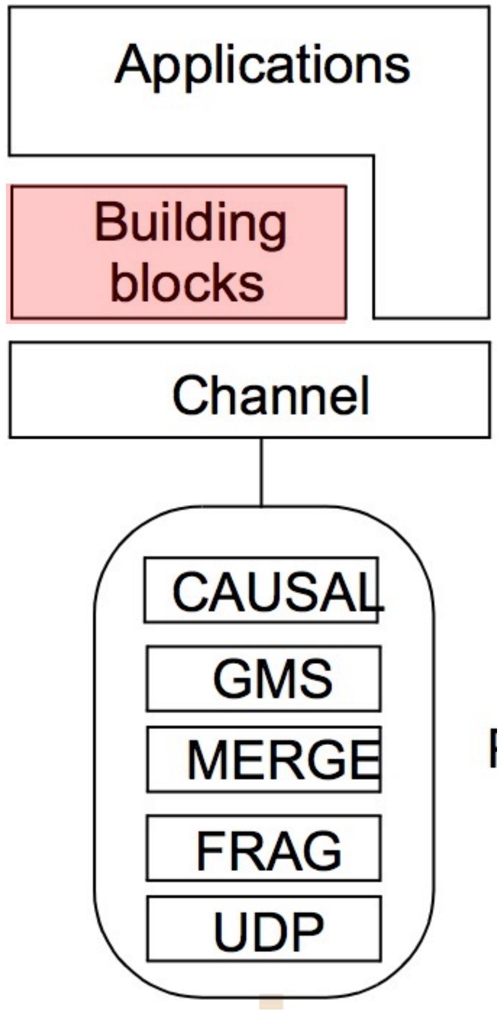
```
import org.jgroups.JChannel;

  public class FireAlarmConsumerJG {
      public String await() {
      try {
              JChannel channel = new JChannel();
              channel.connect("AlarmChannel");
              Message msg = (Message) channel.receive(0);
              return (String) msg.GetObject();
      } catch(Exception e) {
              return null;
      }
    }
}
```

Timeout; 0 means block until a message is received



JGroups – Building Blocks

- MessageDispatcher: send a message to a group and waits for some or all of the replies
- RpcDispatcher: invoke a method on all objects associated with a group
- ReplicatedHashMap: allow members in a group to share common state
- …



Group Communication

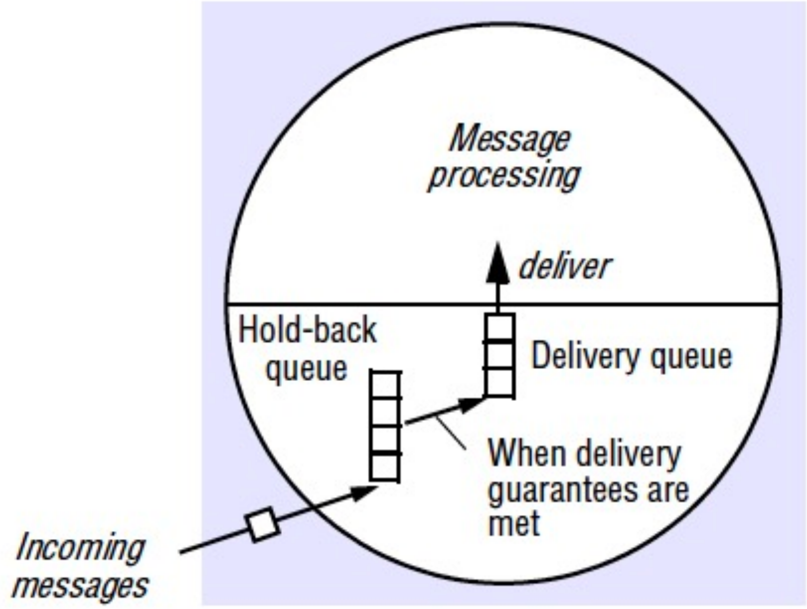
- Programming Model (6.2.1-6.2.2)
- Case study: JGroups (6.2.3)
- Reliable and ordered multicast (15.4)
- View-synchronous group communication (18.2)

Assumptions

- Processes can fail only by crash, reliable one-to-one channels
- Static groups with known membership
- Each process is a member of at most one group
- Closed groups

Group Communication

- $multicast(g, m)$ sends a message m to all members of group g
  - $m.sender$: the unique id of the process that sent
  - $m.group$: the unique destination group id
- $deliver(m)$ delivers a message m sent by a multicast to the calling process
  - A multicast message is not always handed to the application layer inside the process as soon as it is received at the process's node



Reliable Multicast

- integrity: every correct process delivers a message at most once, only if some process in the group multicasts that message
- validity: if a correct process multicasts a message, it will eventually deliver it
- agreement: if a correct process delivers message m, then all other correct processes in the group will eventually deliver m

Reliable Multicast via Reliable Unicast

$\boldsymbol{P_i}$::

var

$Received$= {};

$\text{multicast}(g, m)$:

for each $q \in g$, send(q, m);

reliable unicast

$\text{receive}(m)$:

if $(m \notin Received)$

$Received = Received \cup \{m\}$;

if($P_i \neq m.sender$)

for each $q \in m.group$, send(q, m);

deliver m to the application layer;

Message complexity: $O(N^2)$

Reliable Multicast over IP Multicast

P_i::

var

hold-back $=\{\}$;

$S = 0$; // seq no of last sent msg

$R[1 \ldots N]$; // $R[q]$: seq no of last delivered msg from q

$\text{multicast}(g, m)$:

IP-multicast $(g, m, S, \{<q, R[q]>\})$;

$S = S + 1$;

piggybacked acknowledgements

$\text{receive}(m, S, <q, R'[q]>)$:

$p = m.sender$;

if $(S == R[p] + 1)$

deliver message; $R[p] = R[p] + 1$;

if $(S \leq R[p])$

message is discarded

if $(S > R[p] + 1)$

put m in the *hold-back* queue

send NACK to p

for $q \in [1 \ldots N]$

if $(R'[q] > R[q])$

send NACK to p or q

negative acknowledgement

Reliable Multicast over IP Multicast

- Integrity
  - duplicate detection
  - error checking in IP-multicast
- Validity
  - negative acknowledgement
- Agreement
  - missing message always detected if there are infinite multicast messages
  - there is always an available copy of a missing message if processes retain copies they have delivered indefinitely

Ordered Multicast

- FIFO ordering: if a correct process issues *multicast*(g, m) and then *multicast*(g, m'), then every correct process that delivers m' would already have delivered m
- Causal ordering: if *multicast*$(g, m) \rightarrow$ *multicast*(g, m'), then any correct process that delivers m' would already have delivered m
- Total ordering: if a correct process delivers message m before it delivers m', then any other correct process that delivers m' would already have delivered m
- Hybrid ordering: FIFO-total ordering, causal-total ordering

Ordered Multicast and Reliable Multicast

- Ordered multicast does not assume or imply reliability
- Hybrids of ordered and reliable protocols
  - reliable totally ordered multicast (*atomic multicast*)
  - reliable FIFO multicast
  - reliable causal multicast
  - …

Implement FIFO Ordering

- Our algorithm for reliable multicast over IP multicast guarantees FIFO ordering
- If we don't need reliability:

$\boldsymbol{P_i}$::

var

hold-back $=\{\}$;

$S = 0$; // seq no of last sent msg

$R[1 \ldots N]$; // $R[q]$: seq no of last delivered msg from q

$\text{multicast}(g, m)$:

IP-multicast(g, m, S);

$S = S + 1$;

$\text{receive}(m, S)$:

$p = m.sender$;

if $(S == R[p] + 1)$

deliver message; $R[p] = R[p] + 1$;

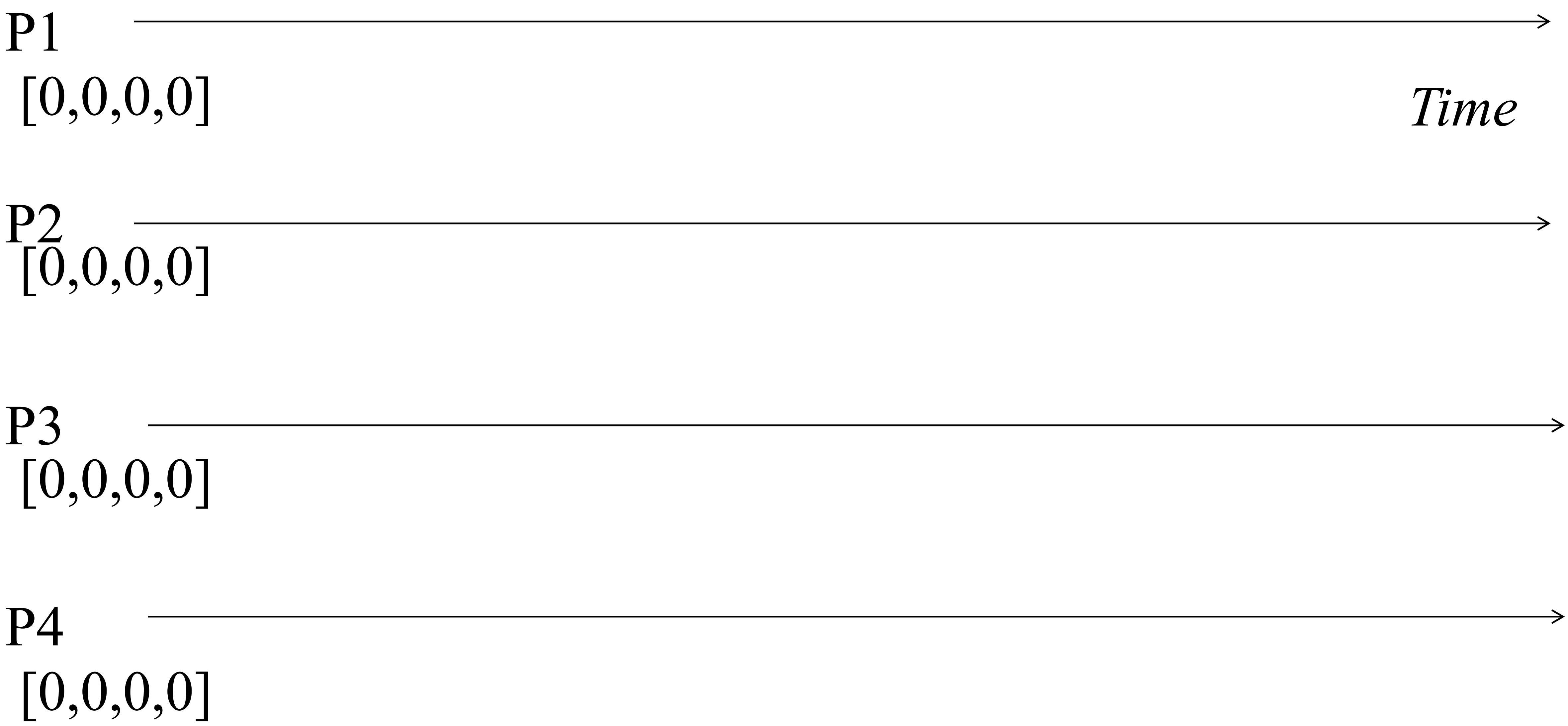
if $(S \leq R[p])$

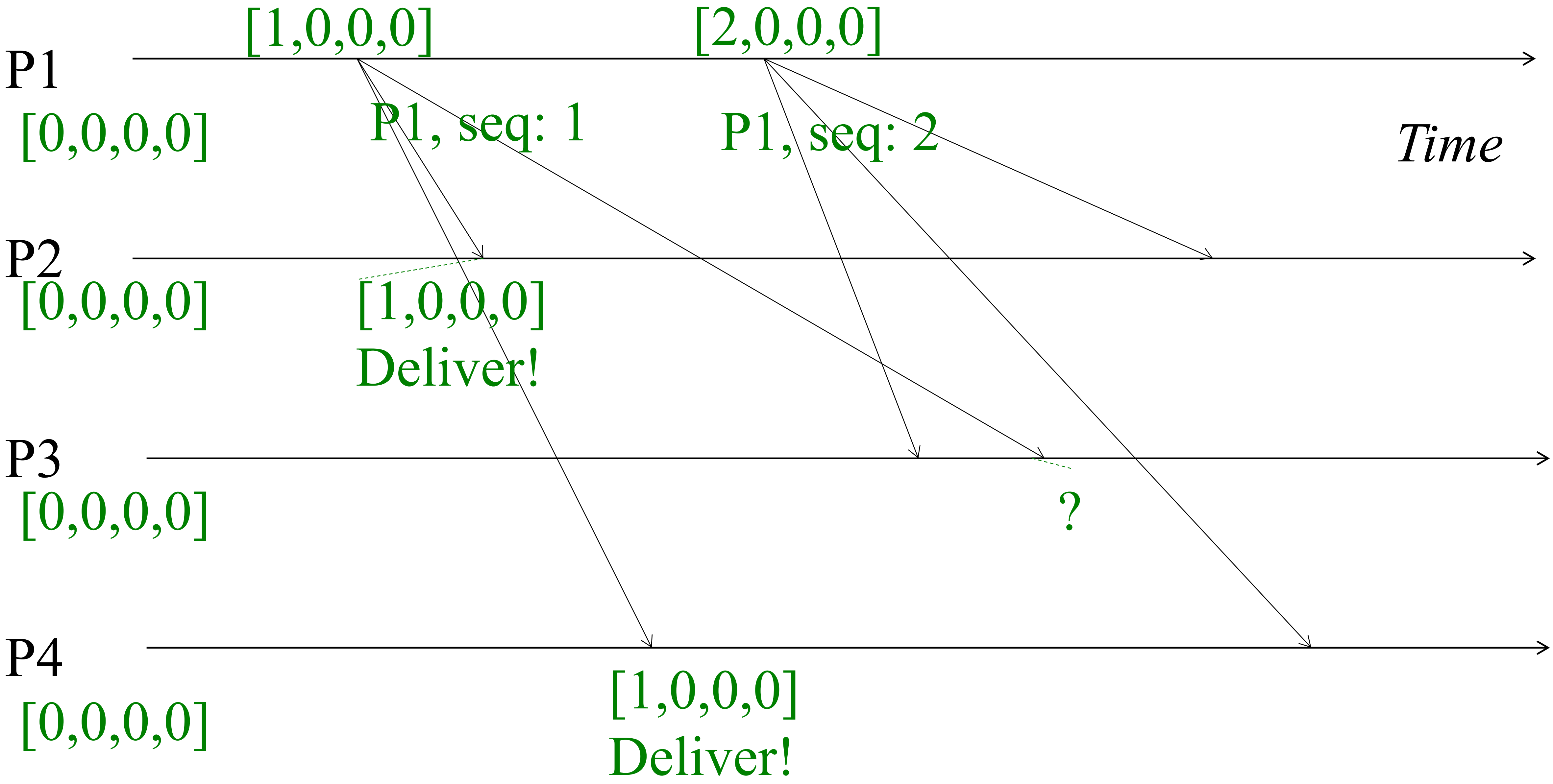
message is discarded

if $(S > R[p] + 1)$

put m in the *hold-back* queue;

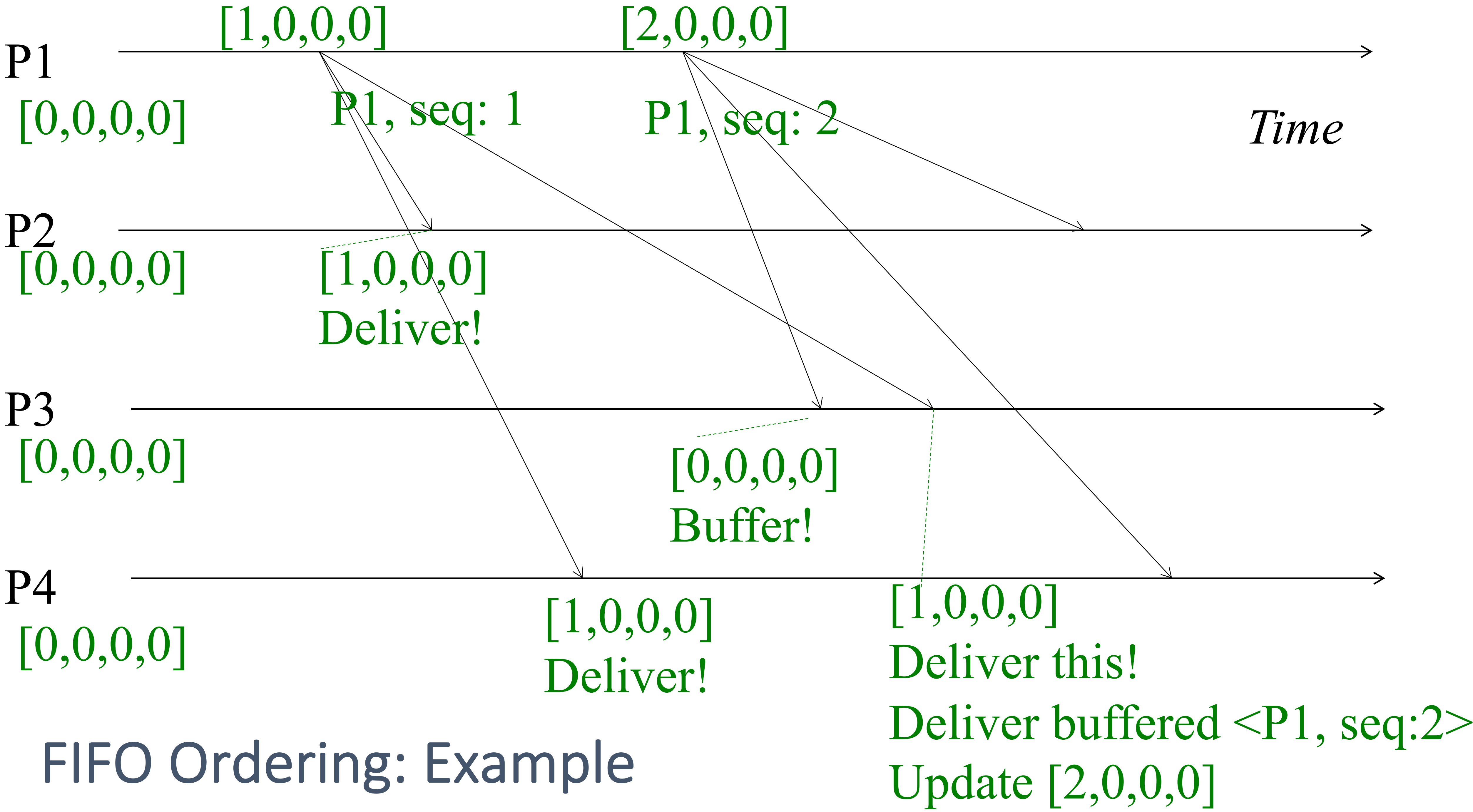
FIFO Ordering: Example



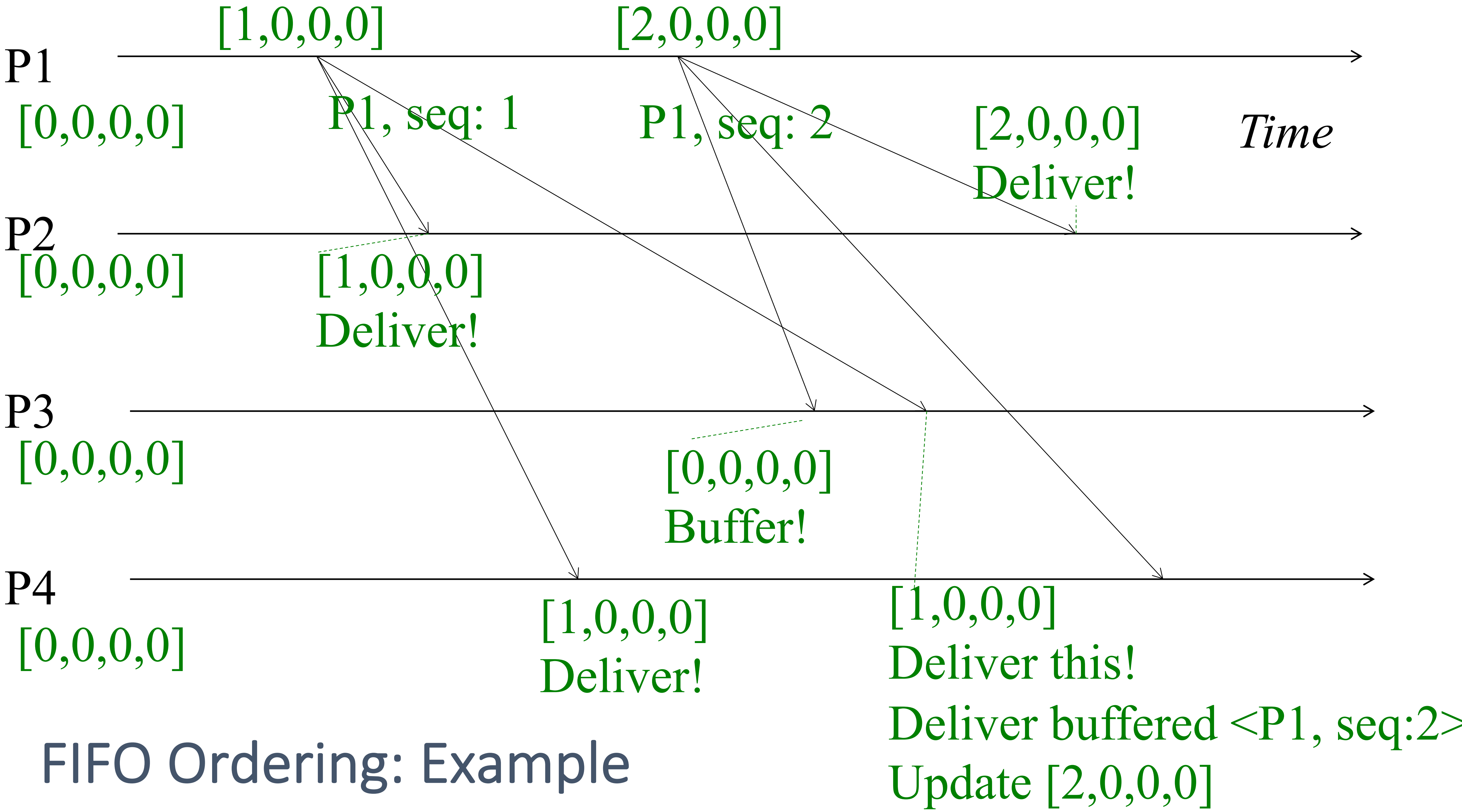


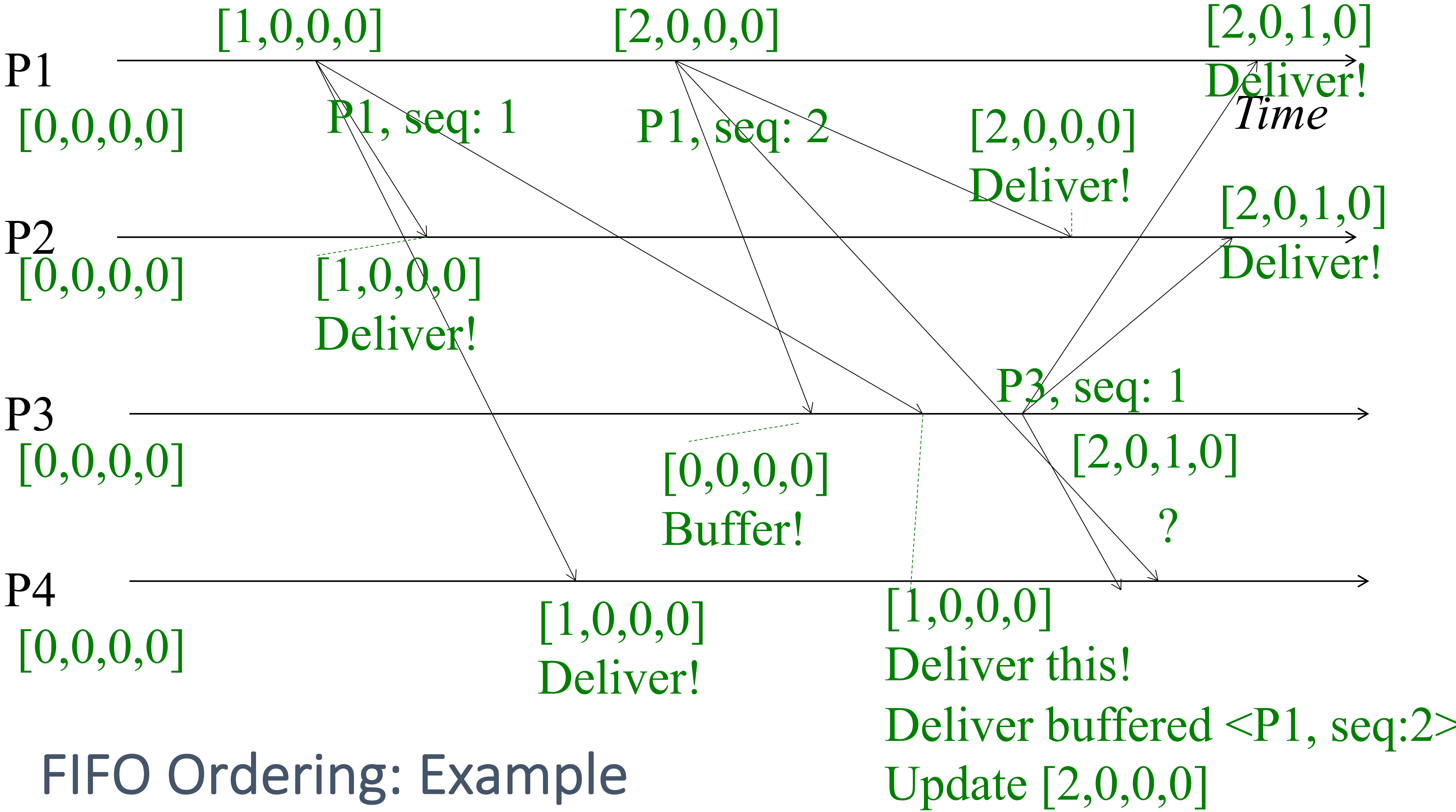
FIFO Ordering: Example

FIFO Ordering: Example

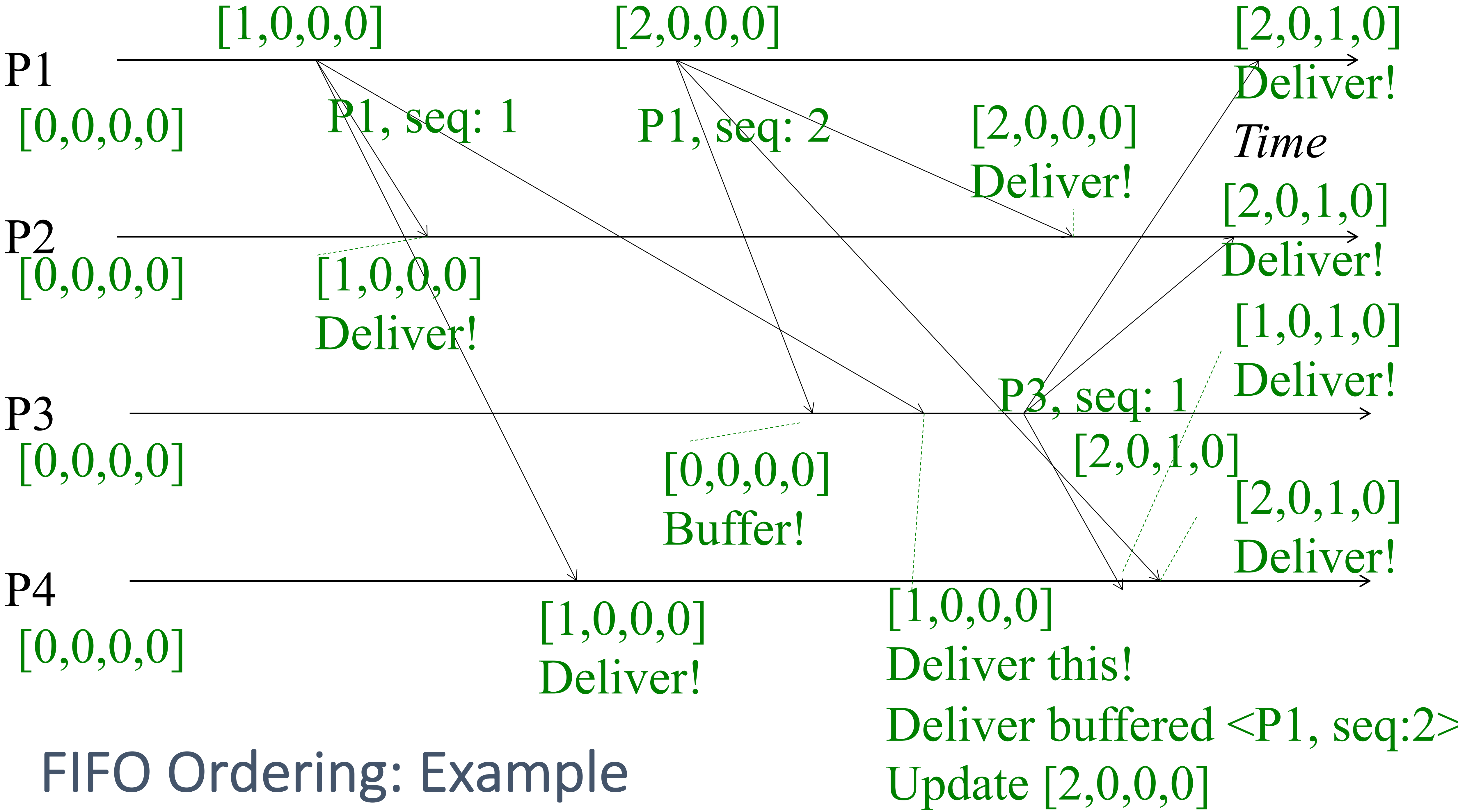


FIFO Ordering: Example



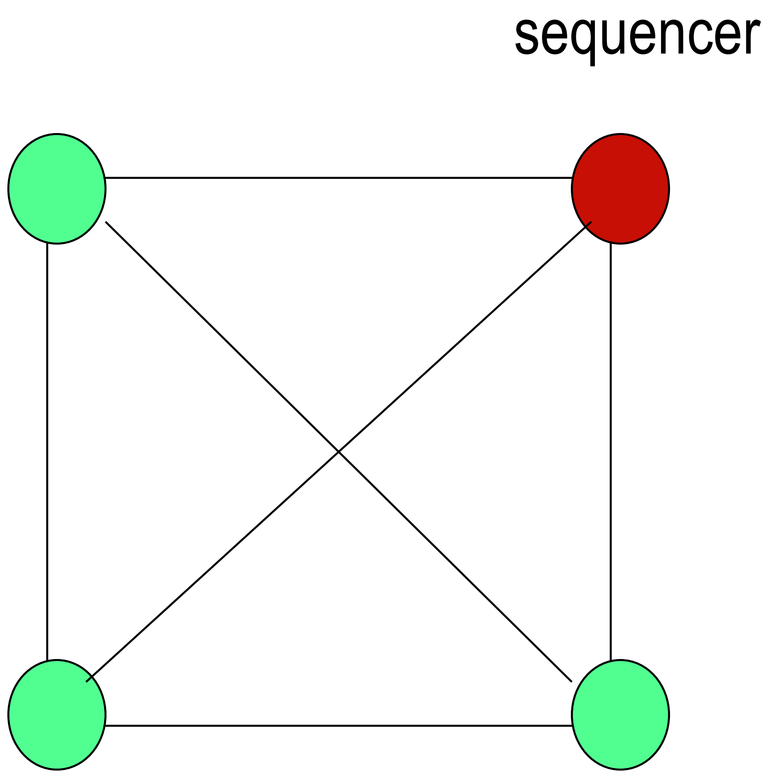


FIFO Ordering: Example



FIFO Ordering: Example

Total ordering using a sequencer



Total ordering using a sequencer

P_i::

var

hold-back = {};

$r = 0;$

multicast(g, m) :

IP-multicast ($g \cup \{sequencer$(g)$\}$, <m, id>);

receive(<m, id>):

place *m* in the *hold-back* queue

receive(m_{order} =<“order”, id, s>)

wait until <m, id> in *hold-back* queue and $s = r$;

deliver m;

$r = s + 1;$

Sequencer::

var

$s = 0;$

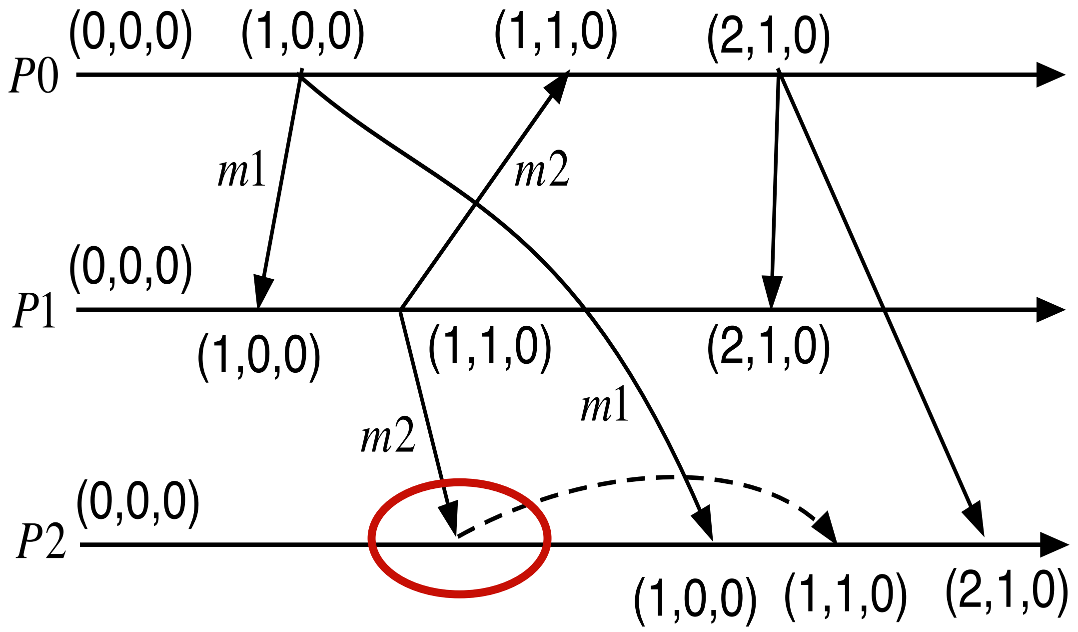
receive(<m, id>):

IP-multicast (g, <“order”, id, s>);

deliver m;

$s = s + 1;$

Causal ordering using vector timestamps



Causal ordering using vector timestamps

P_i::

var

hold-back = {};

VC: array[$1..N$] of integer;

$\text{multicast}(g, m)$:

$VC[i] = VC[i] + 1$;

IP-multicast (g, <m, VC>);

$\text{receive}(<m, t>)$:

$j = m.sender$;

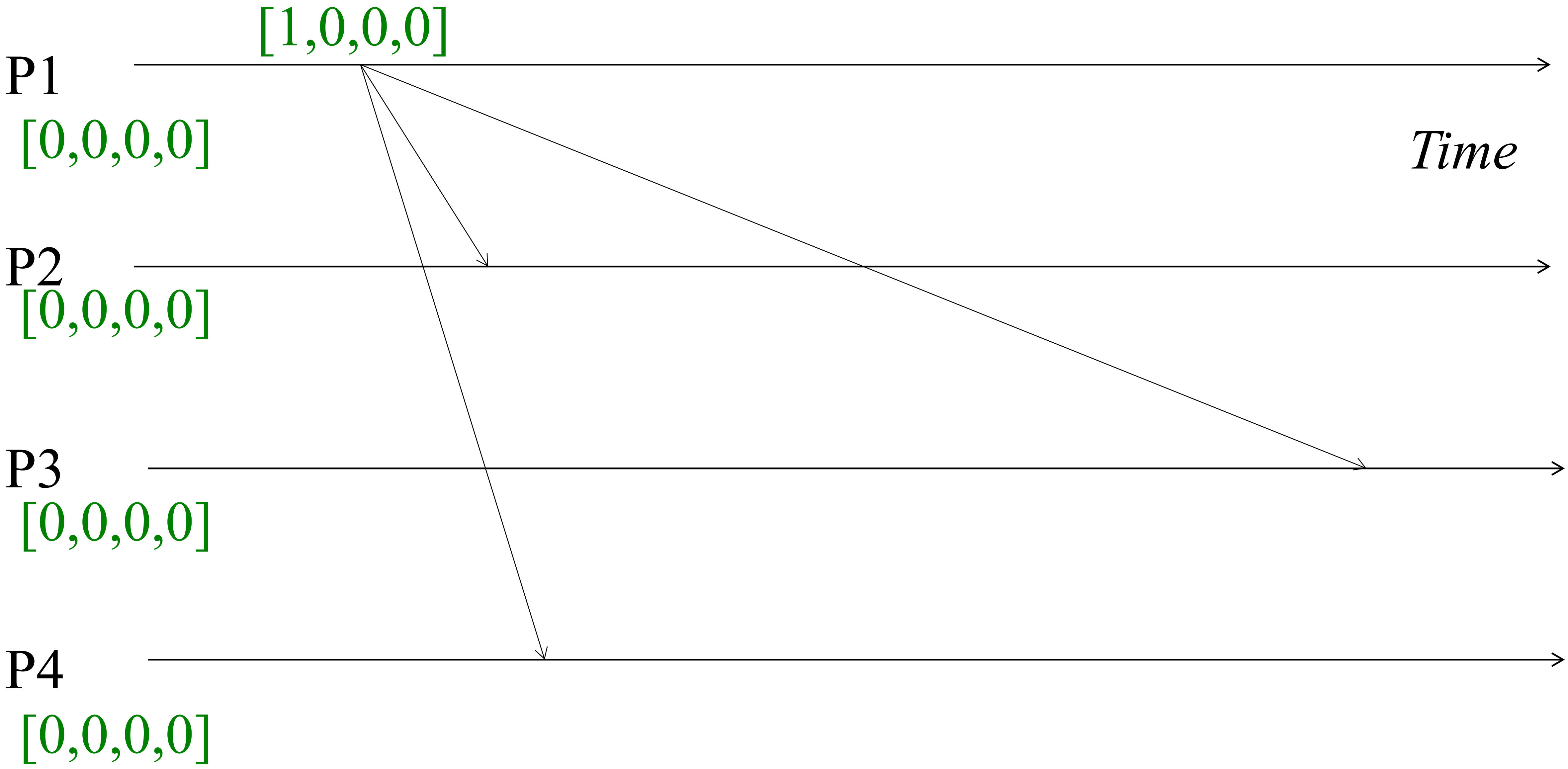
place m in the *hold-back* queue;

wait until $t[j] = VC[j] + 1$ *and* $t[k] \le VC[k]$ $(\forall k \neq j)$;

deliver m;

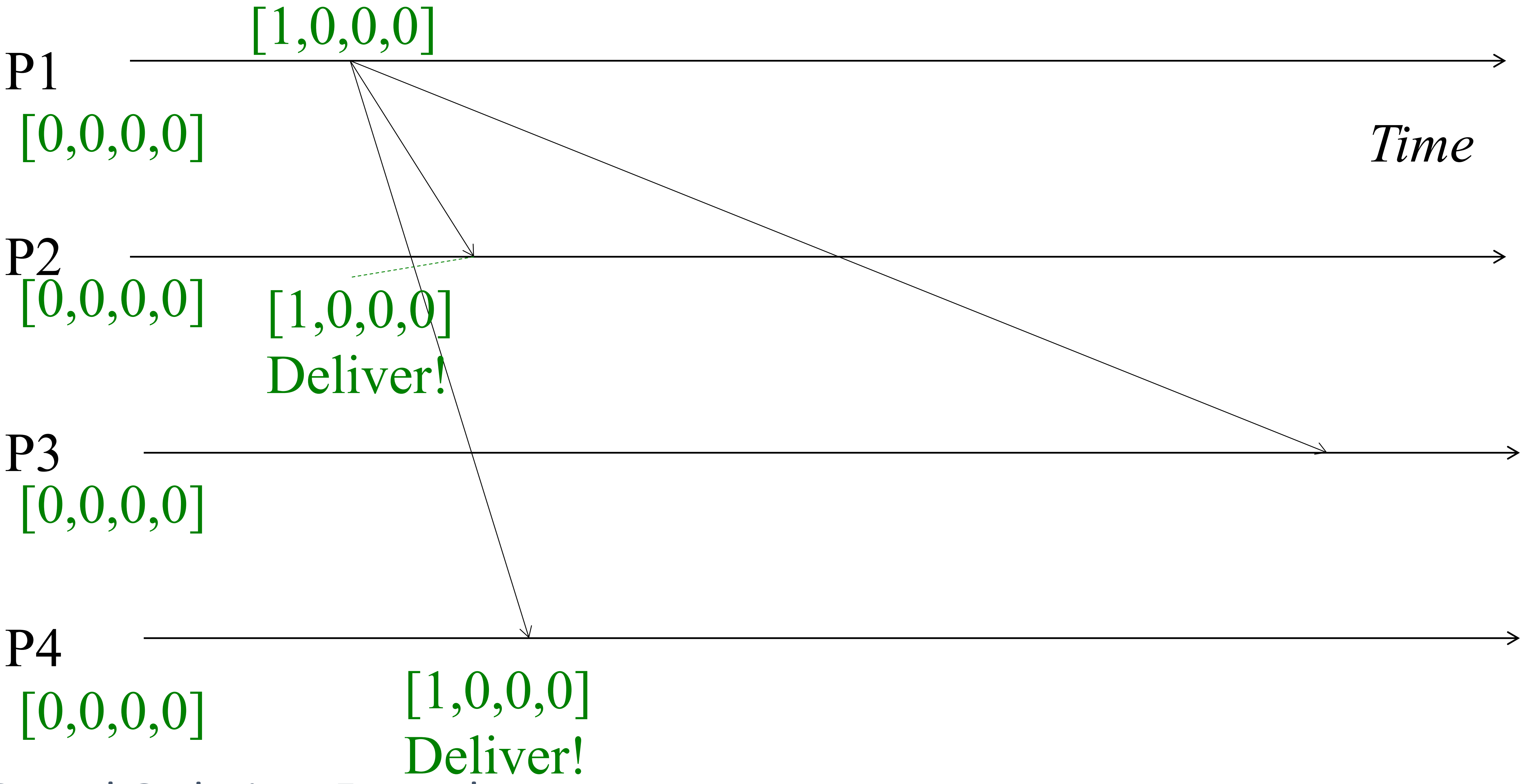
$VC[j] = VC[j] + 1$;

- Causal multicast + reliable multicast ⇒ reliable causally ordered multicast
- Causal multicast + sequencer-based protocol ⇒ causally and totally ordered multicast

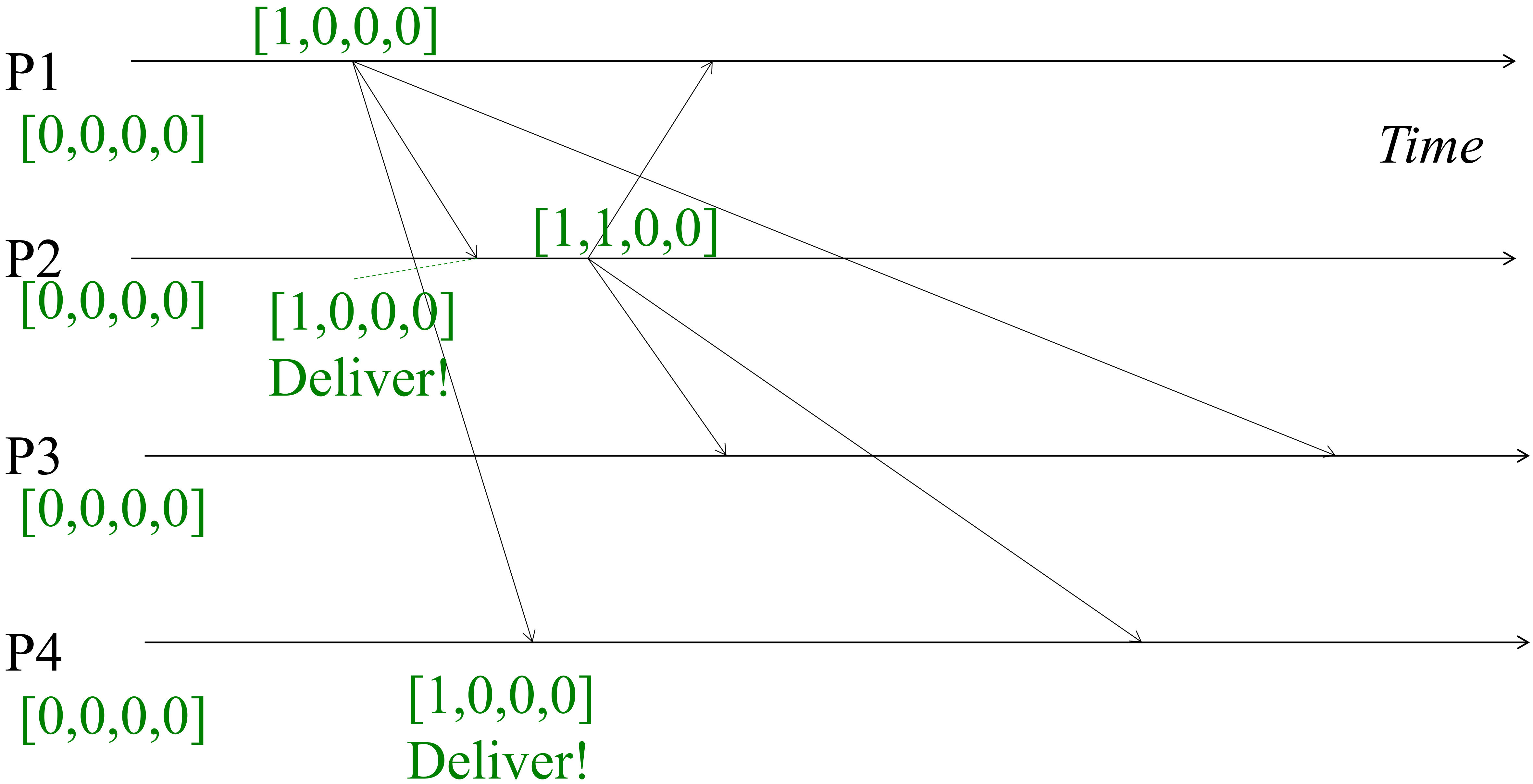


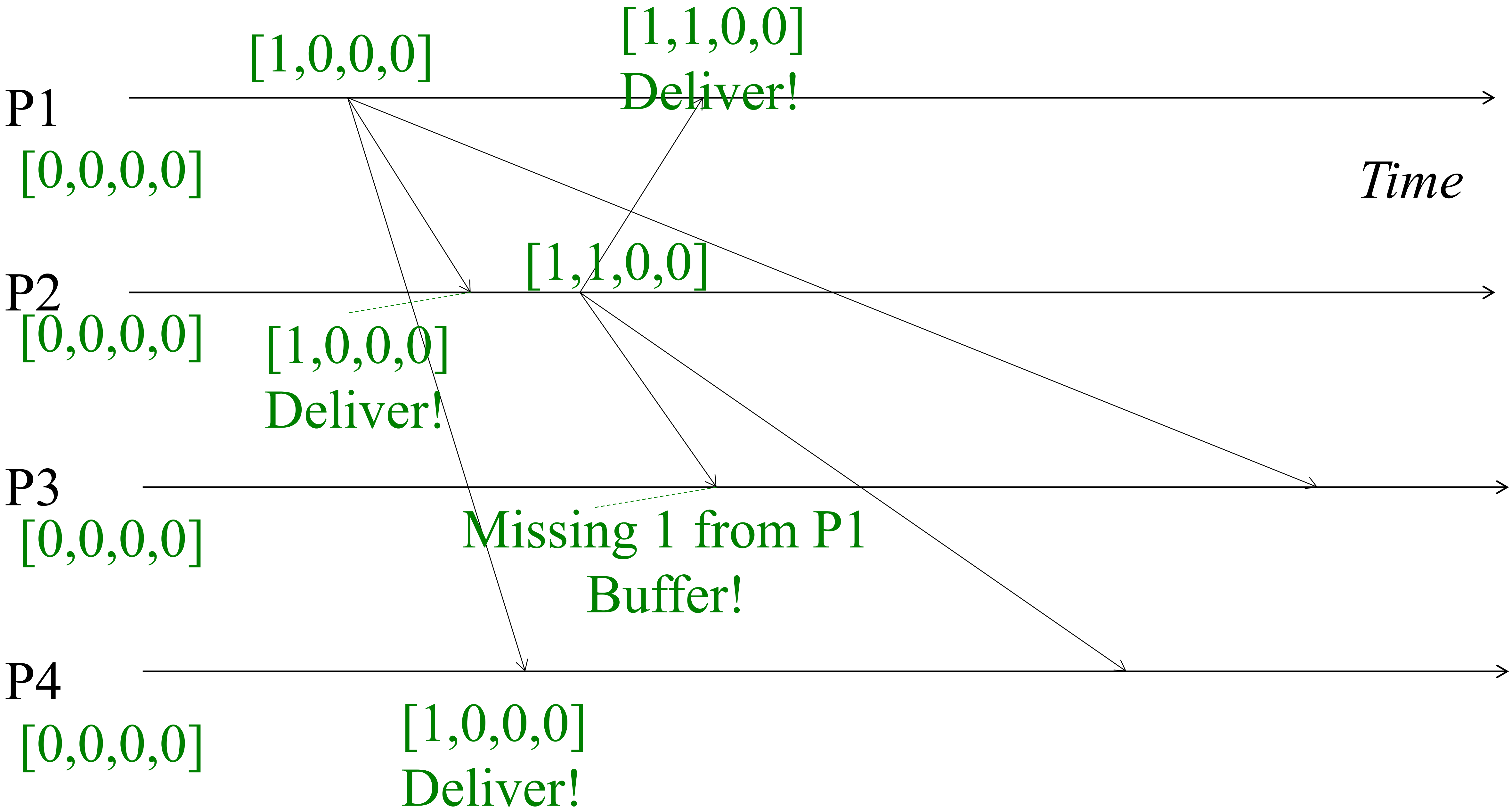
Causal Ordering: Example

Causal Ordering: Example

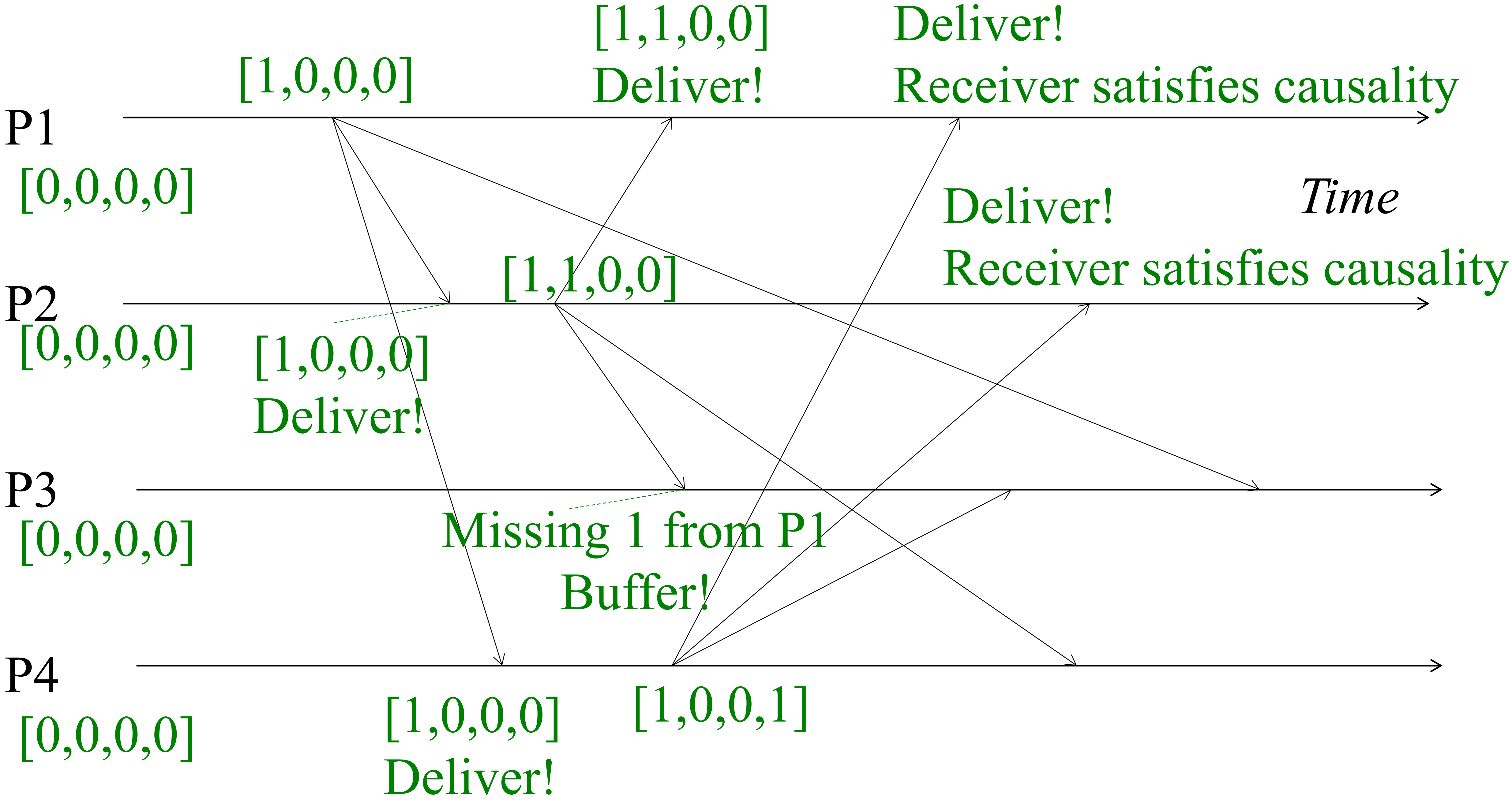


Causal Ordering: Example

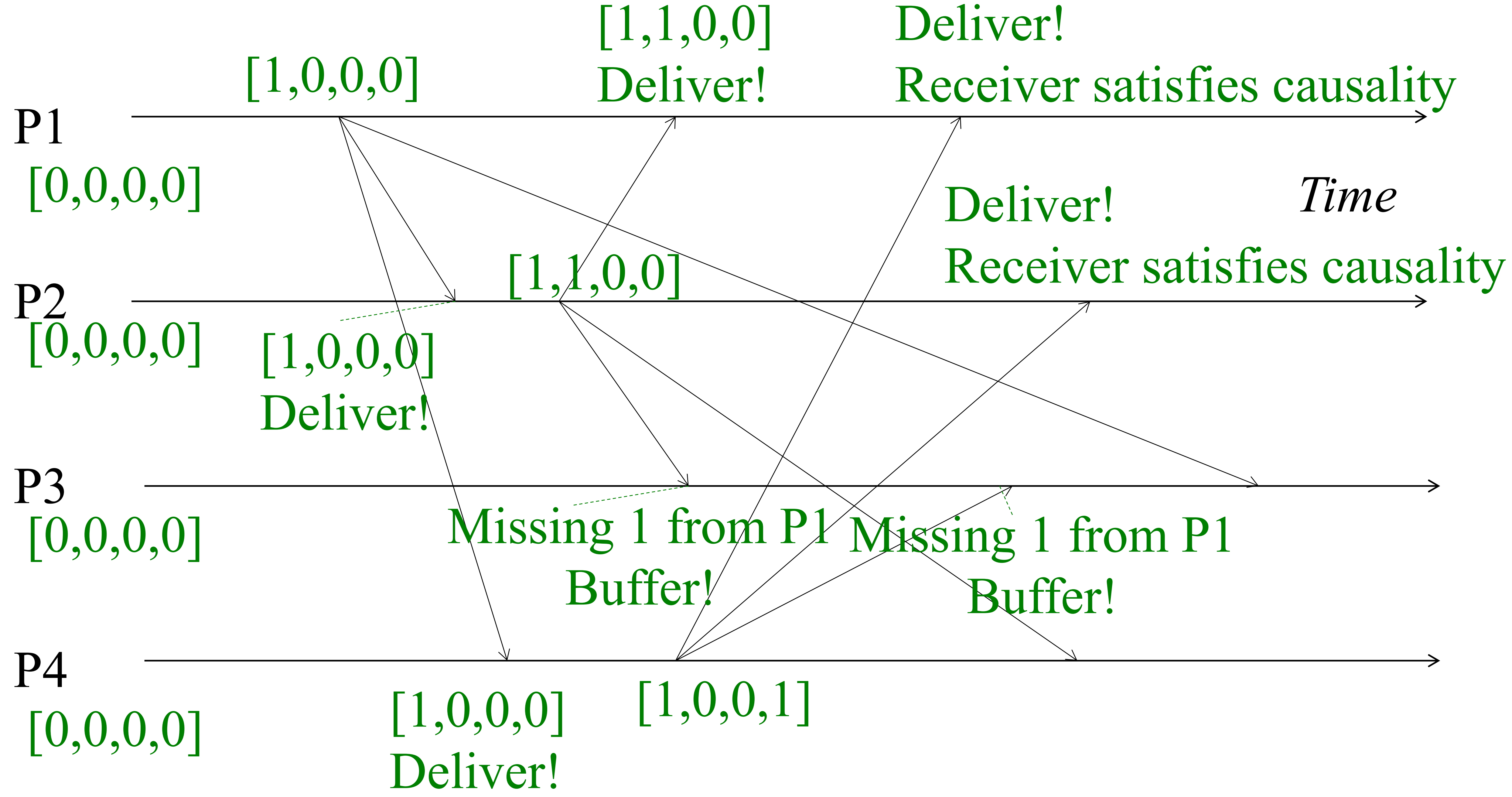




Causal Ordering: Example

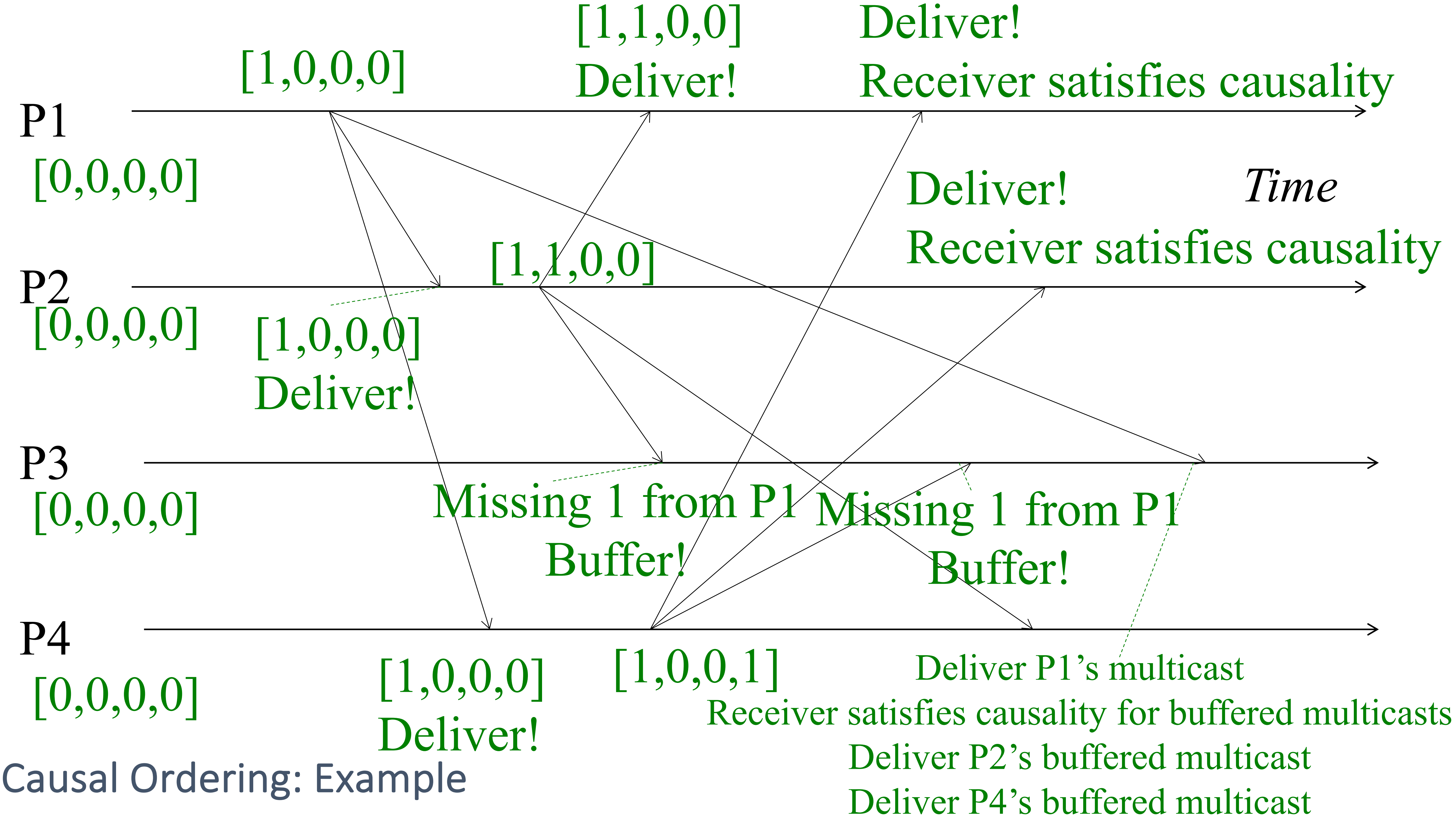


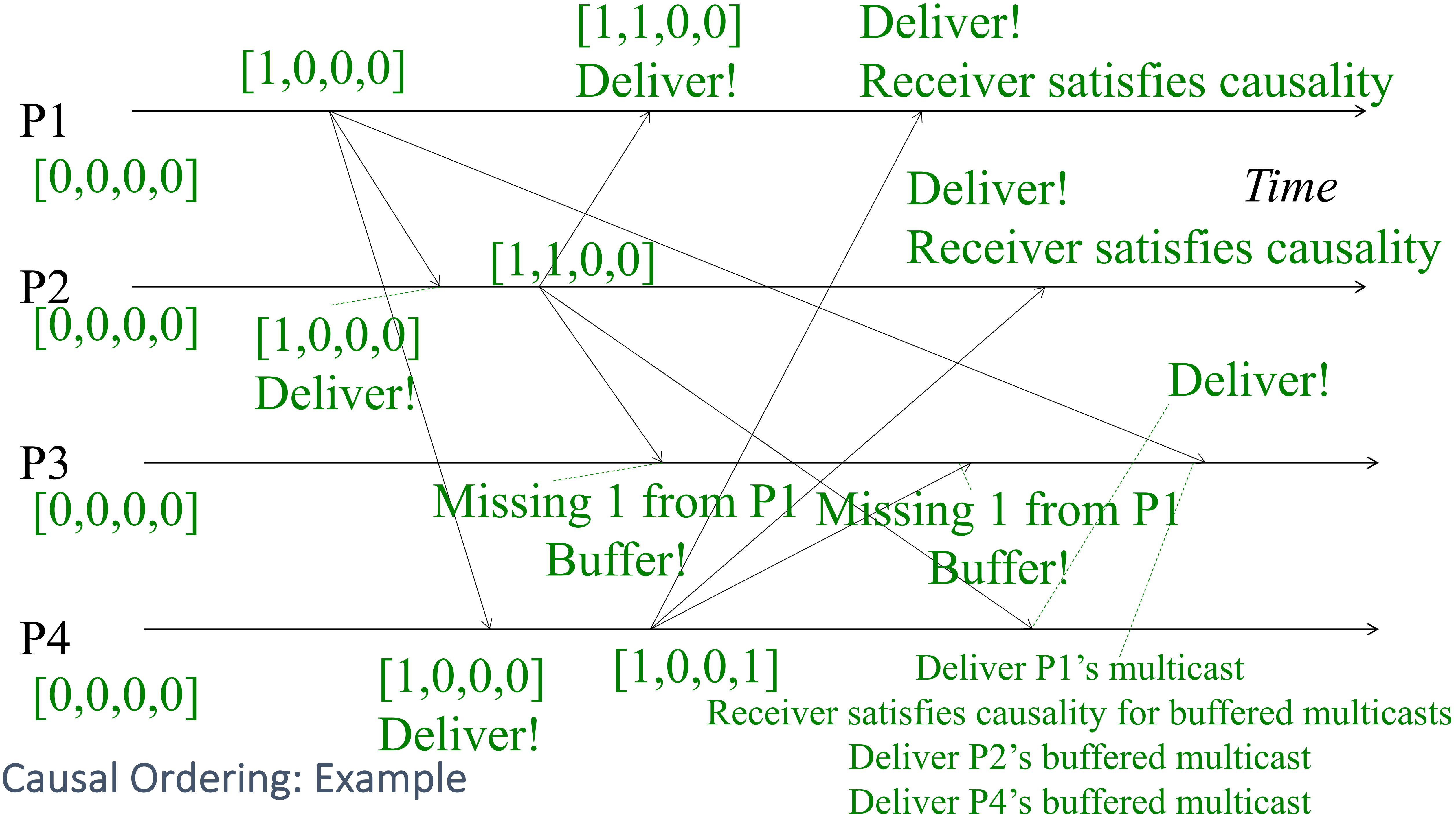
Causal Ordering: Example



Causal Ordering: Example

Causal Ordering: Example





Causal Ordering: Example

Overlapping groups

- Global *FIFO* ordering: If a correct process issues *multicast*(g, m) and then *multicast*(g', m'), every correct process in $g \cap g'$ that delivers m' would already have delivered m
- One can define global *causal* ordering and global *total* ordering similarly
- A simple approach to implement global ordering
  - Multicast each message m to all the processes in the system
  - Each process either discards or delivers m according to whether belongs to *group*(*m*)

Multicast in synchronous and asynchronous systems

- We have described algorithms for
  - Reliable unordered multicast
  - Reliable FIFO-ordered multicast
  - Reliable causally ordered multicast
  - Totally ordered multicast
  - Causally and totally ordered multicast
  - FIFO and totally ordered multicast

Multicast in synchronous and asynchronous systems

- Can we get reliable and totally ordered multicast (atomic multicast)?
  - Yes for synchronous system
  - No for asynchronous system even with a single process crash failure
  - Equivalent to consensus with crash failures (FLP impossibility result)

Group Communication

- Programming Model (6.2.1-6.2.2)
- Case study: JGroups (6.2.3)
- Reliable and ordered multicast (15.4)
- View-synchronous group communication (18.2)

Virtual Synchrony/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 have been used in NYSE, French Air Traffic Control System, Swiss Stock Exchange

Views

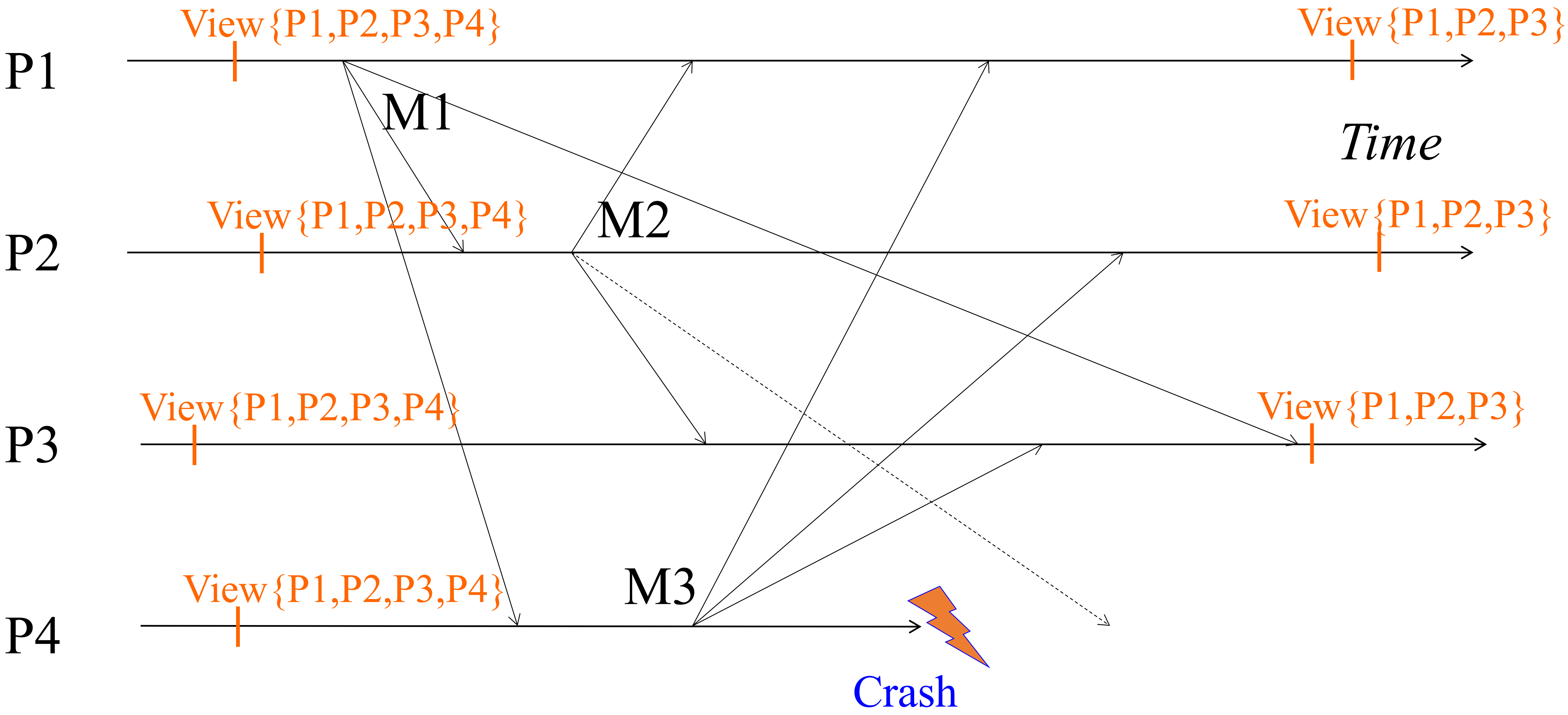
- Each process maintains a membership list
- The membership list is called a *View*
  - i.e., lists of the current group members, identified by their unique process ids
  - The list is ordered, e.g., according to when the members joined the group
- An update to the membership list is called a *View Change*
  - Process join, leave, or failure

Virtual Synchrony

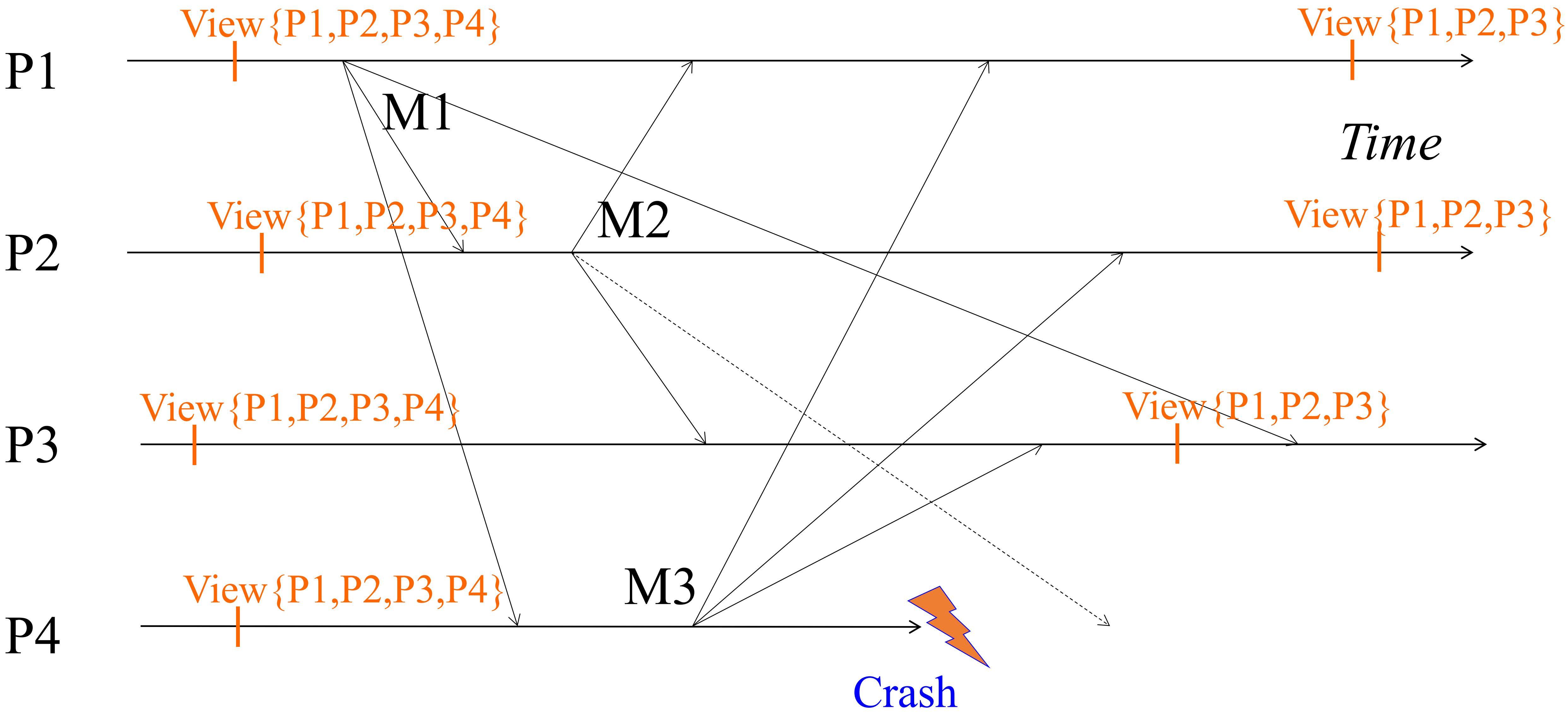
- Virtual synchrony guarantees that all view changes are delivered in the same order at all correct processes
  - If a correct P1 process receives views, say {P1}, {P1, P2, P3}, {P1, P2}, {P1, P2, P4} then
  - Any other correct process receives the *same sequence* of view changes (after it joins the 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 (i.e., total ordering)

VSync Multicasts

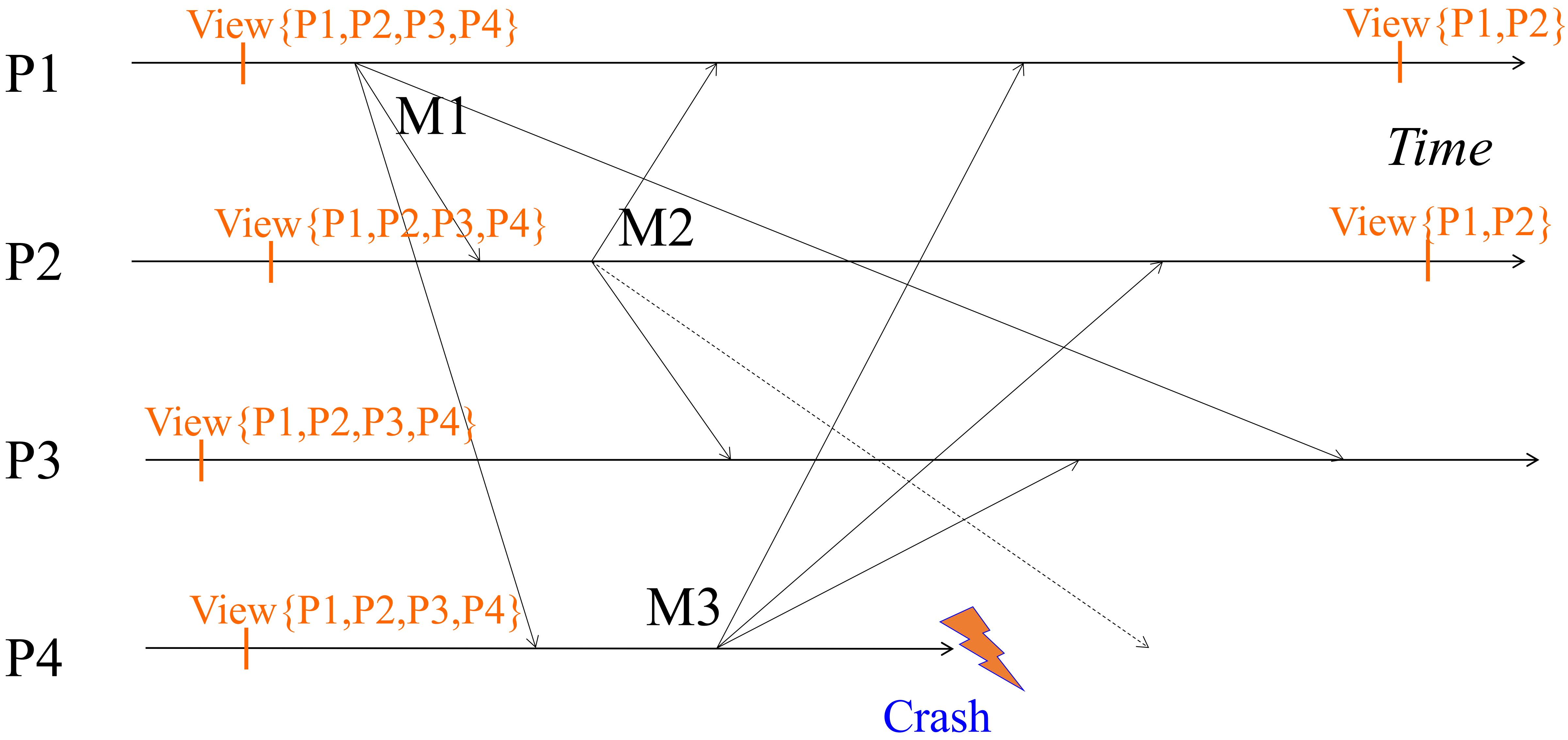
- A multicast M is said to be "delivered in a view V at process P_i" if
  - P_i receives view V, and then sometime before P_i receives the next view it delivers multicast M
- Virtual synchrony ensures that
  - The set of multicasts delivered in a given view is the same set at all correct processes that were in that view
    - What happens in a View, stays in that View
  - The sender of the multicast message also belongs to that view
  - 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



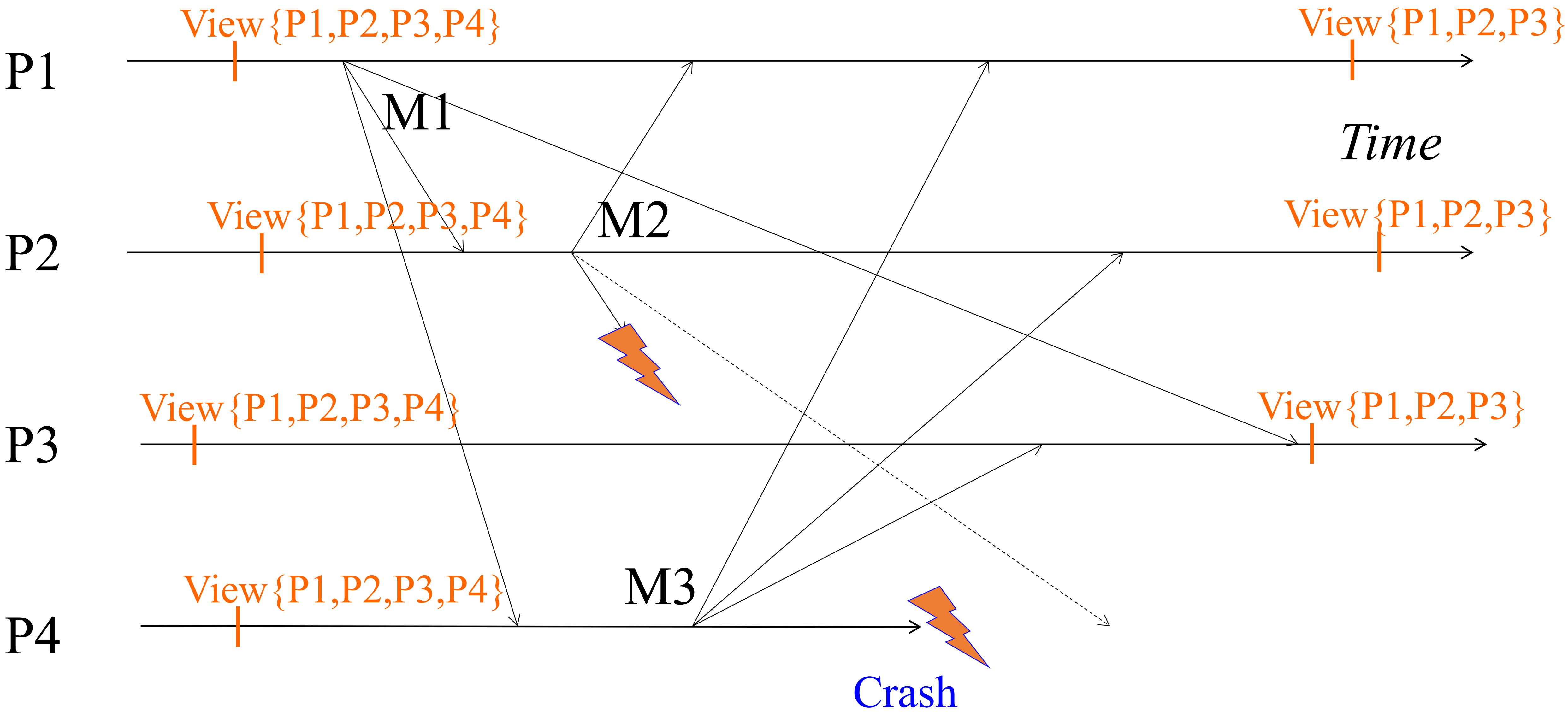
Satisfies virtual synchrony



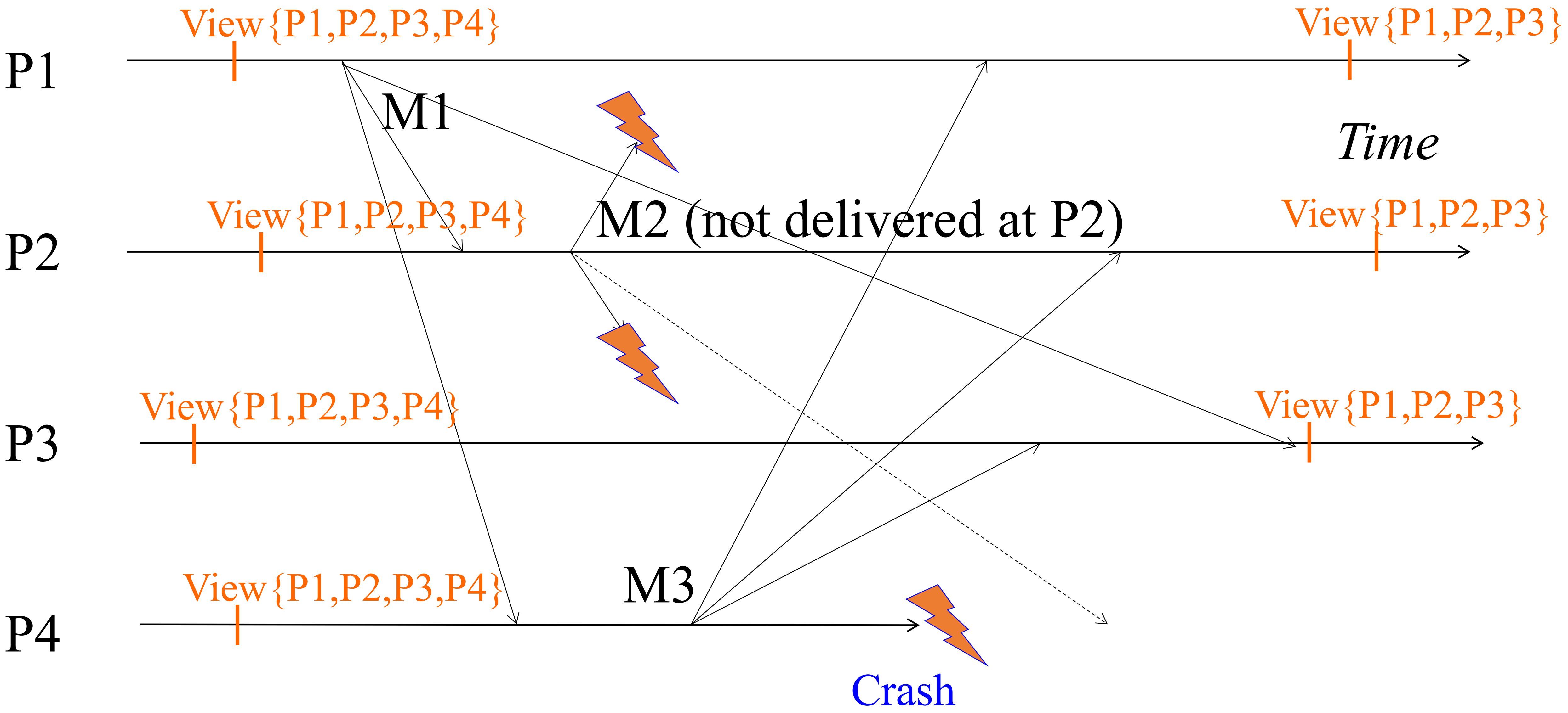
Does not satisfy virtual synchrony



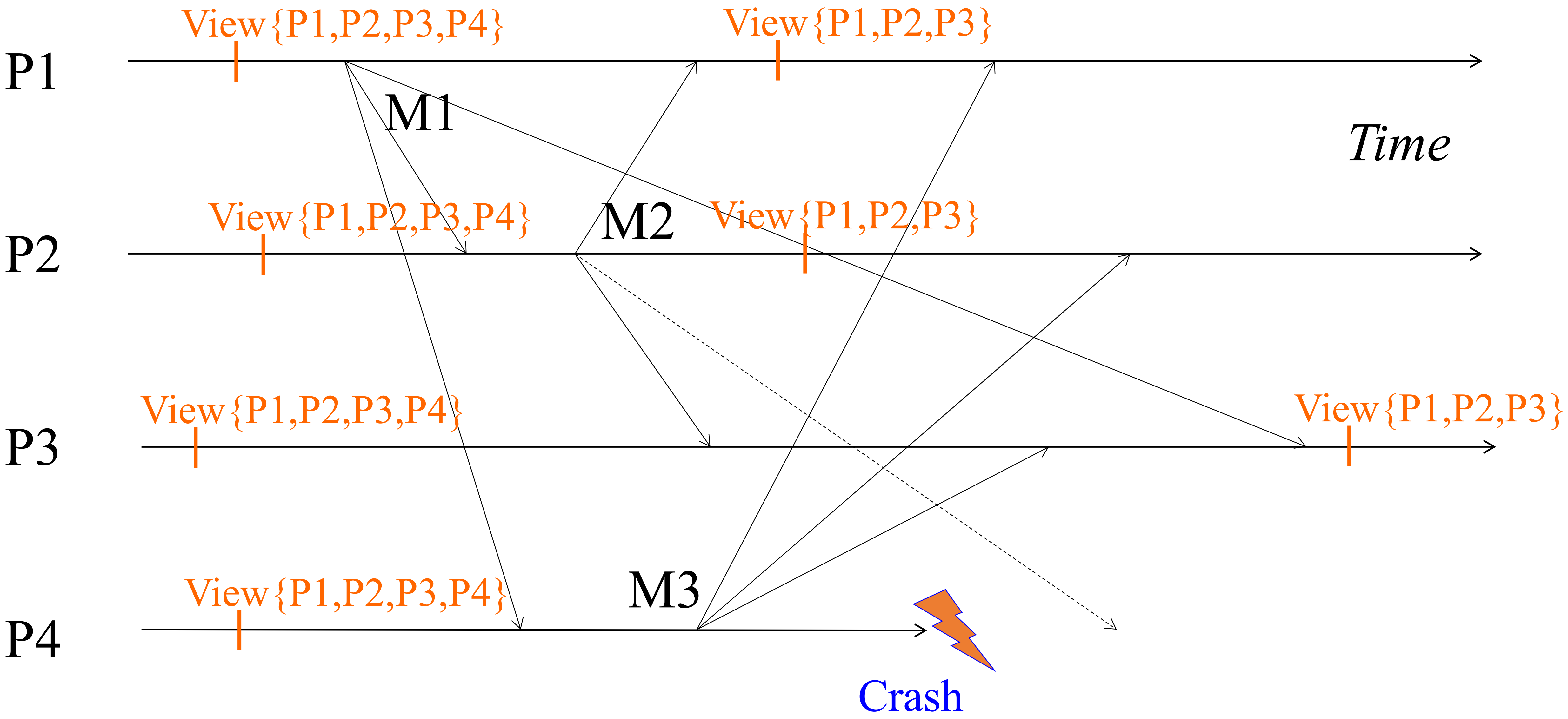
Satisfies virtual synchrony



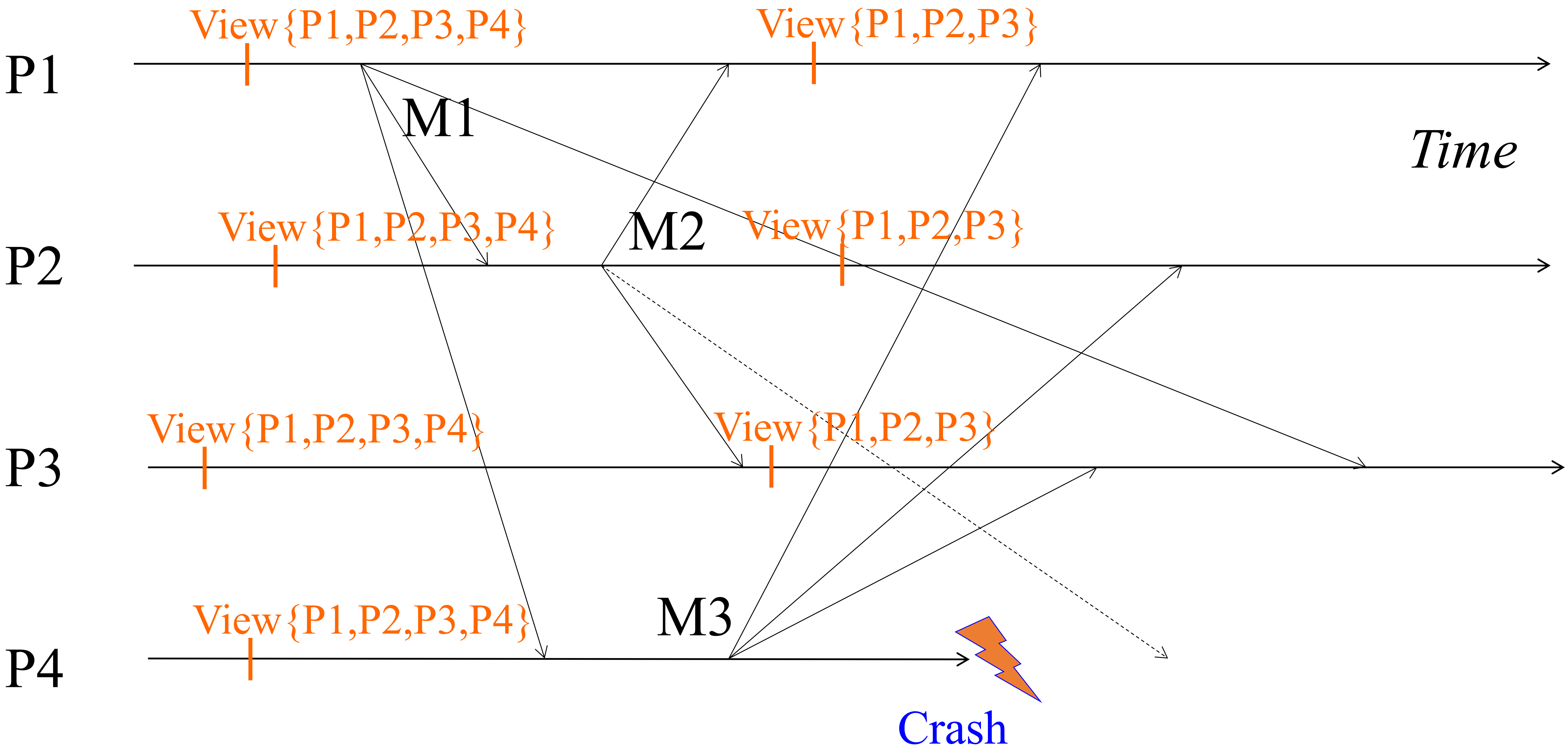
Does not satisfy virtual synchrony



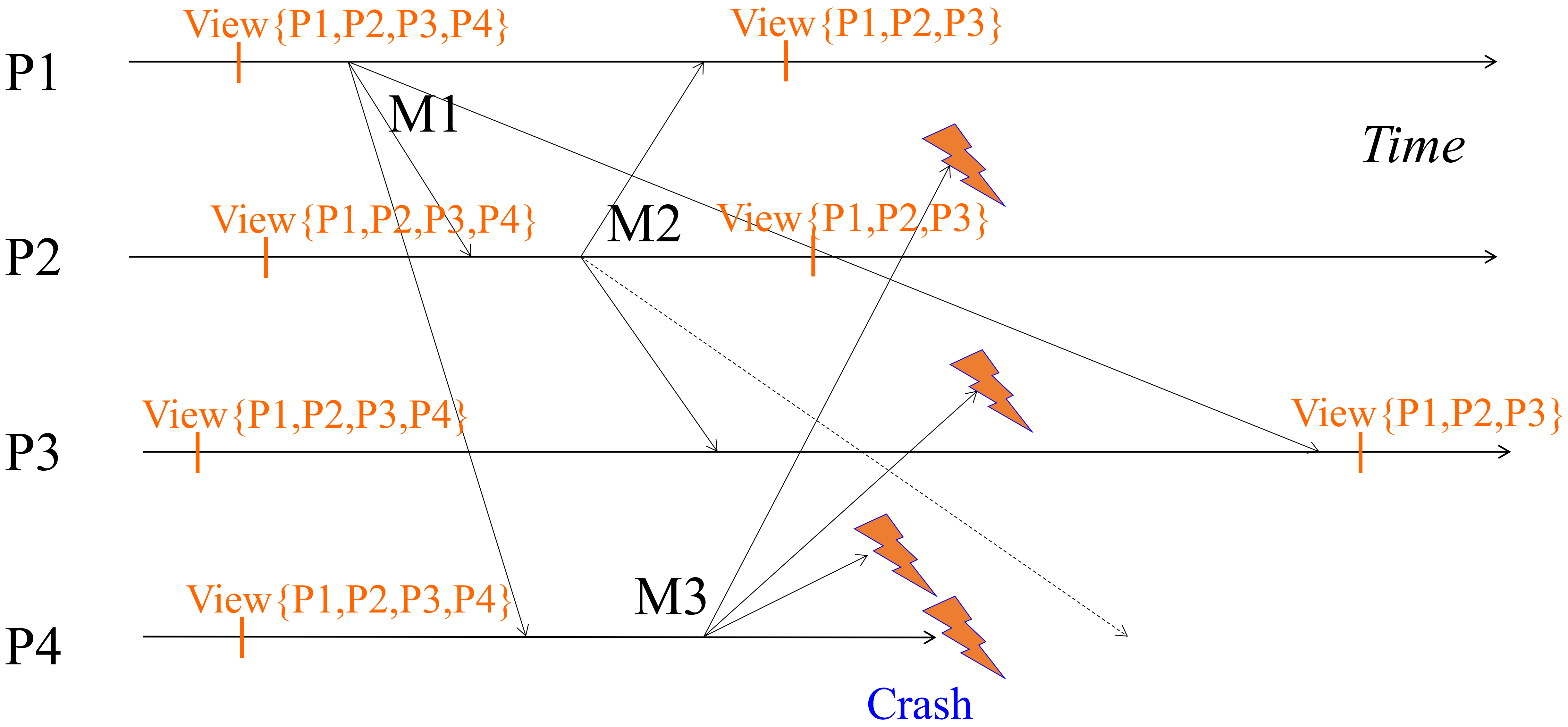
Satisfies virtual synchrony



Does not satisfy virtual synchrony



Does not satisfy virtual synchrony



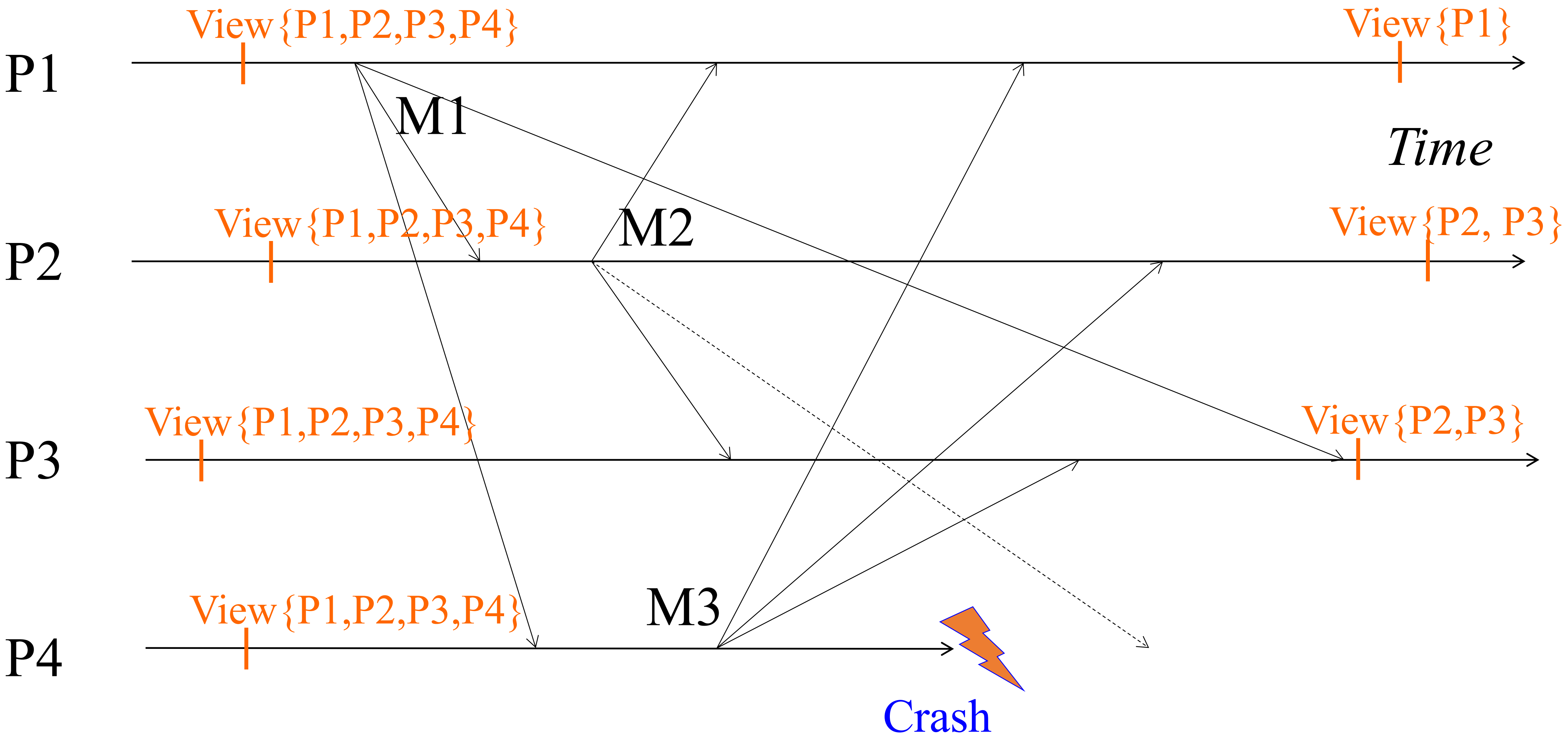
Satisfies virtual synchrony

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