Distributed Systems

CS425/ECE428

02/12/2020

Today's agenda

- Finally wrap-up global snapshots
  - Chapter 14.5
- Multicast
  - Chapter 15.4

Recap: Global snapshot

- State of each process (and each channel) in the system at a given instant of time.
- Useful to capture a global snapshot of the system:
  - Checkpointing, distributed debugging, deadlock detection, garbage collection, etc.
- Difficult to capture global state at same instant of time.
- Capture consistent global state.
  - If captured state includes an event **e**, it includes all other events that *happened before* **e**.
- Chandy-Lamport algorithm captures consistent global state.

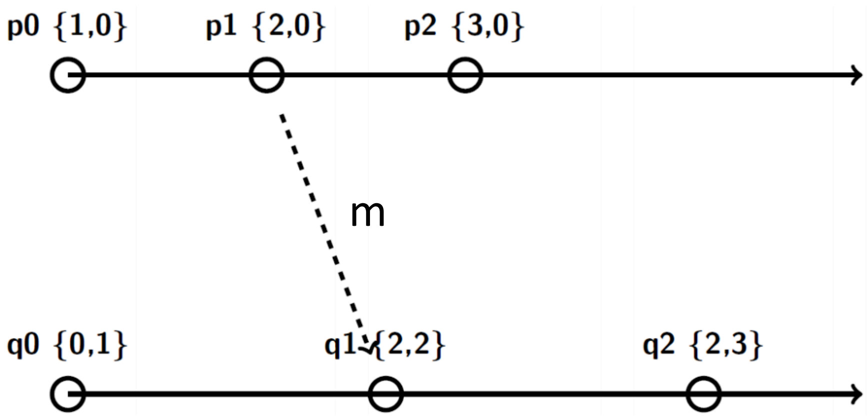
Chandy-Lamport Algorithm: Usefulness

- Consistent global snapshots are useful for detecting global system properties:
  - Safety
  - Liveness

More notations and definitions

- Run
  - a total ordering of events that is consistent with the event ordering at each process.
- Linearization
  - a run consistent with happens-before ($\rightarrow$) relation.
- Linearizations pass through consistent global states.
- A global state $\mathbf{S_k}$ is reachable from global state $\mathbf{S_i}$, if there is a linearization that passes through $\mathbf{S_i}$ and then through $\mathbf{S_k}$.
- The distributed system evolves as a series of transitions between global states S_0 , S_1 , ….

State Transitions: Example



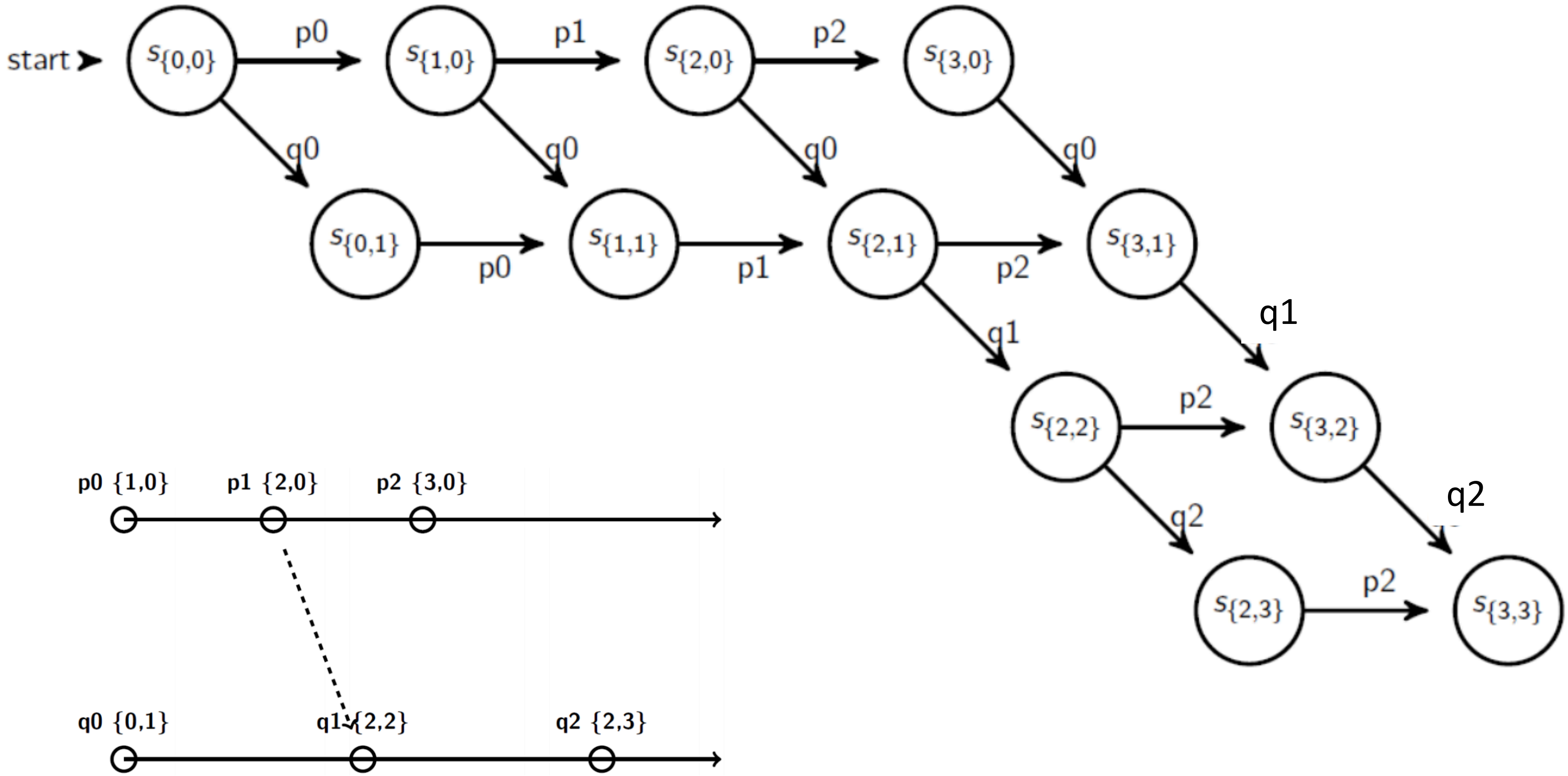
- Causal order:
  - p0 → p1 → p2
  - q0 → q1 → q2
  - p0 → p1 → q1 → q2
- Concurrent:
  - p0 || q0
  - p1 || q0
  - p2 || q0, p2 || q1, p2 || q2

Many linearizations:

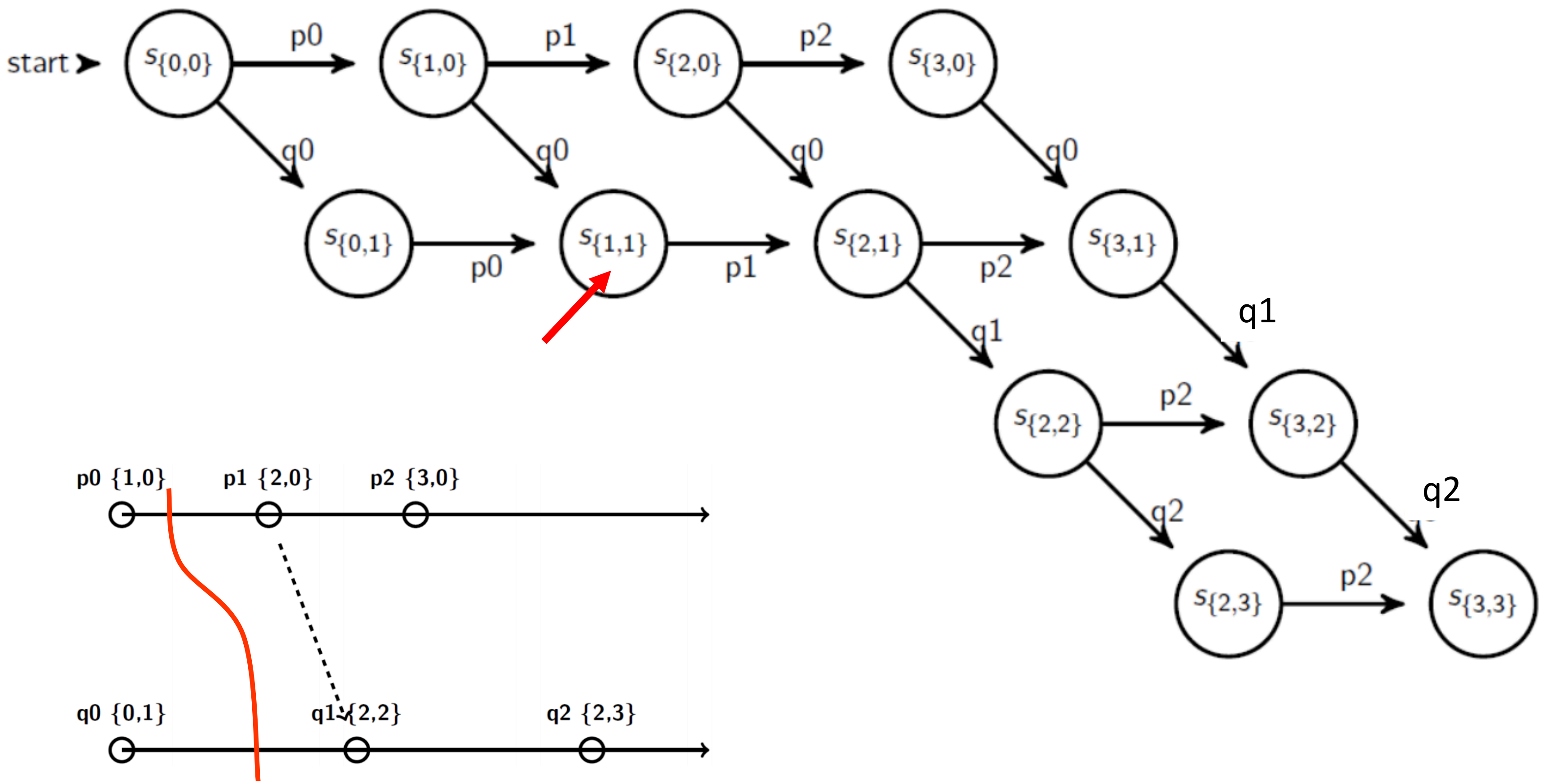
- < p0, p1, p2, q0, q1, q2>
- < p0, q0, p1, q1, p2, q2>
- <q0, p0, p1, q1, p2, q2 >
- <q0, p0, p1, p2, q1,q2 >
- ……

State Transitions: Example

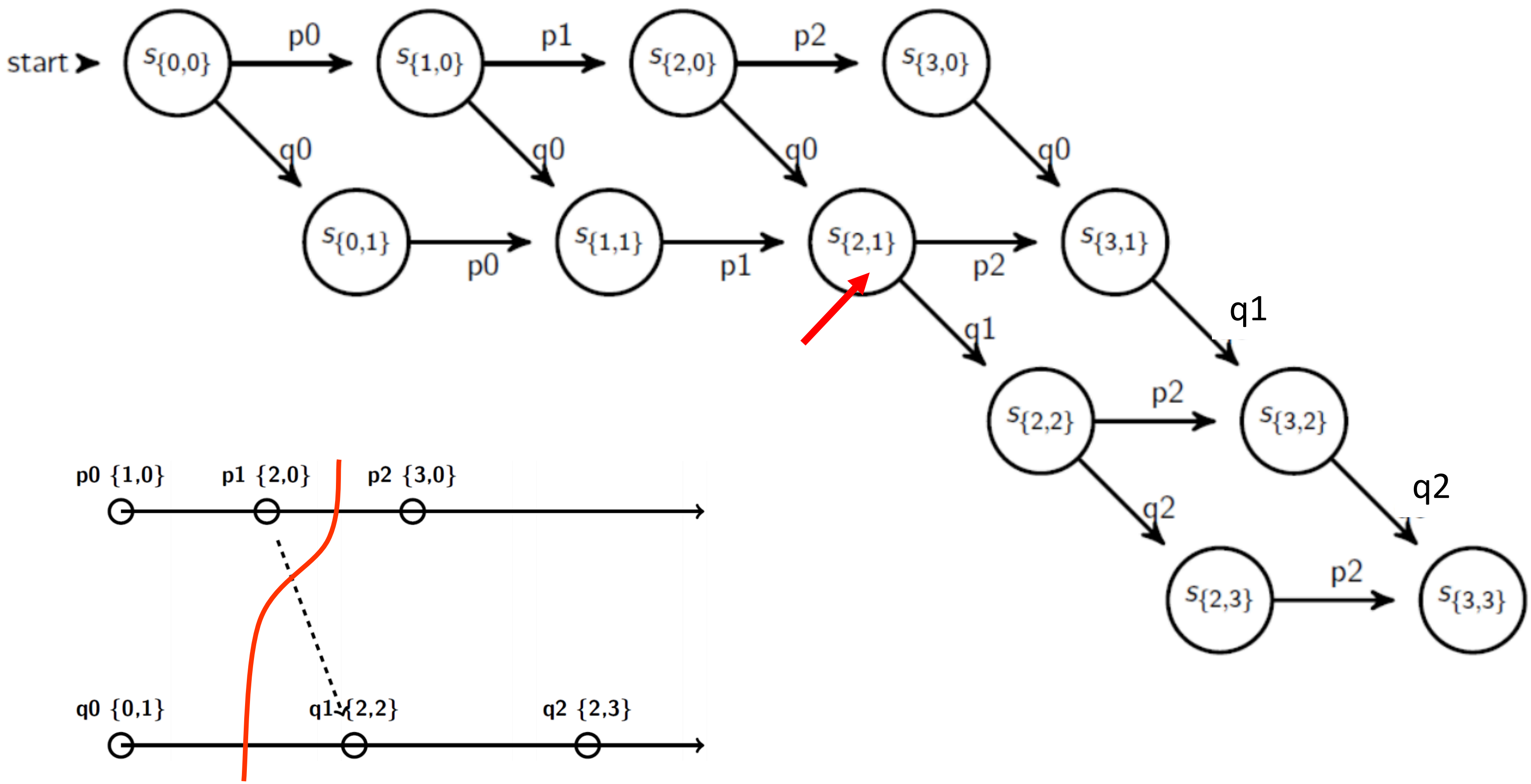
Execution Lattice. Each path is a linear execution of events.



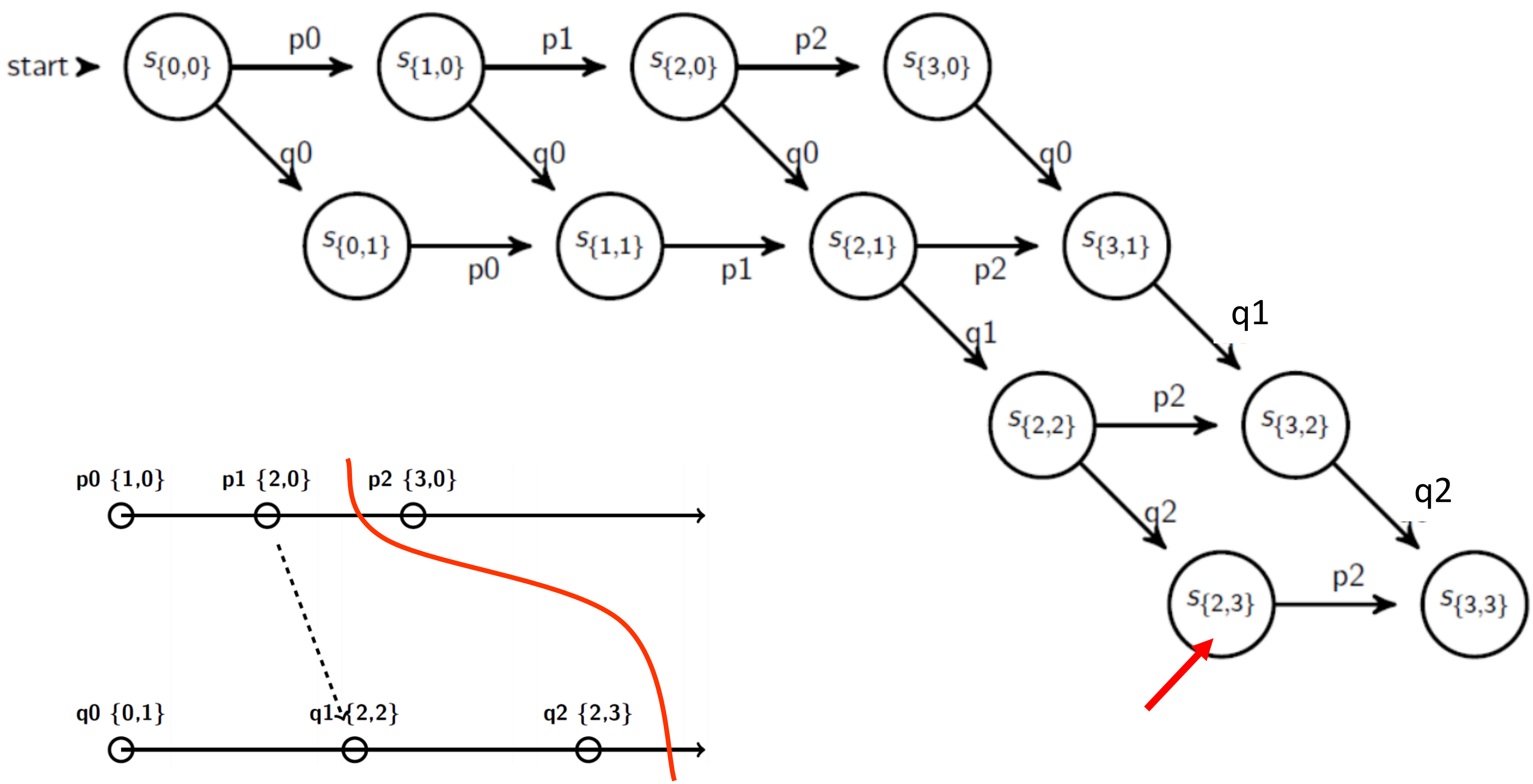
State Transitions: Example



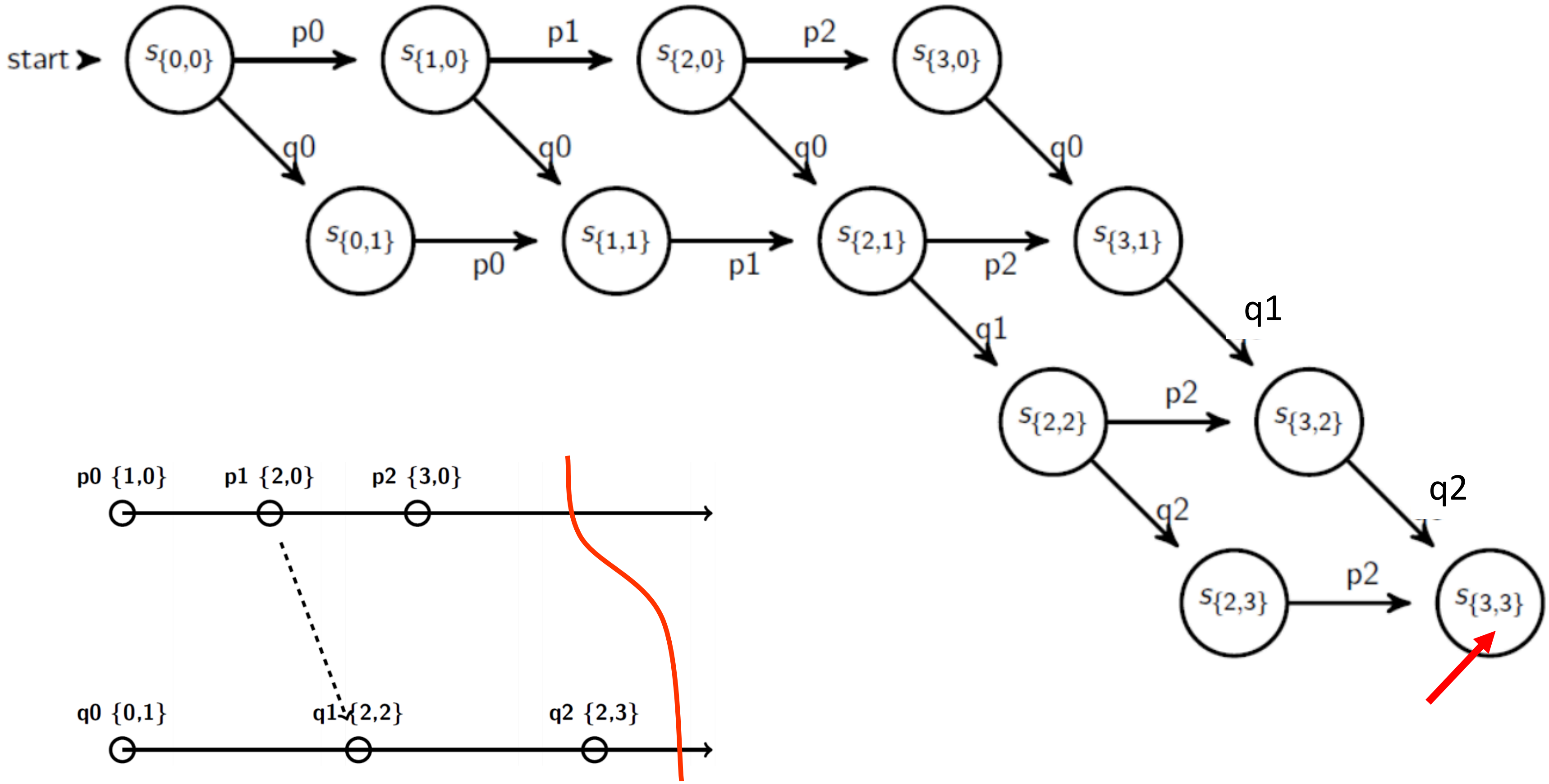
State Transitions: Example



State Transitions: Example



State Transitions: Example



Global State Predicates

- A global-state-predicate is a property that is *true* or *false* for a global state.
  - Is there a deadlock?
  - Has the distributed algorithm terminated?
- Two ways of reasoning about predicates (or system properties) as global state gets transformed by events.
  - Liveness
  - Safety

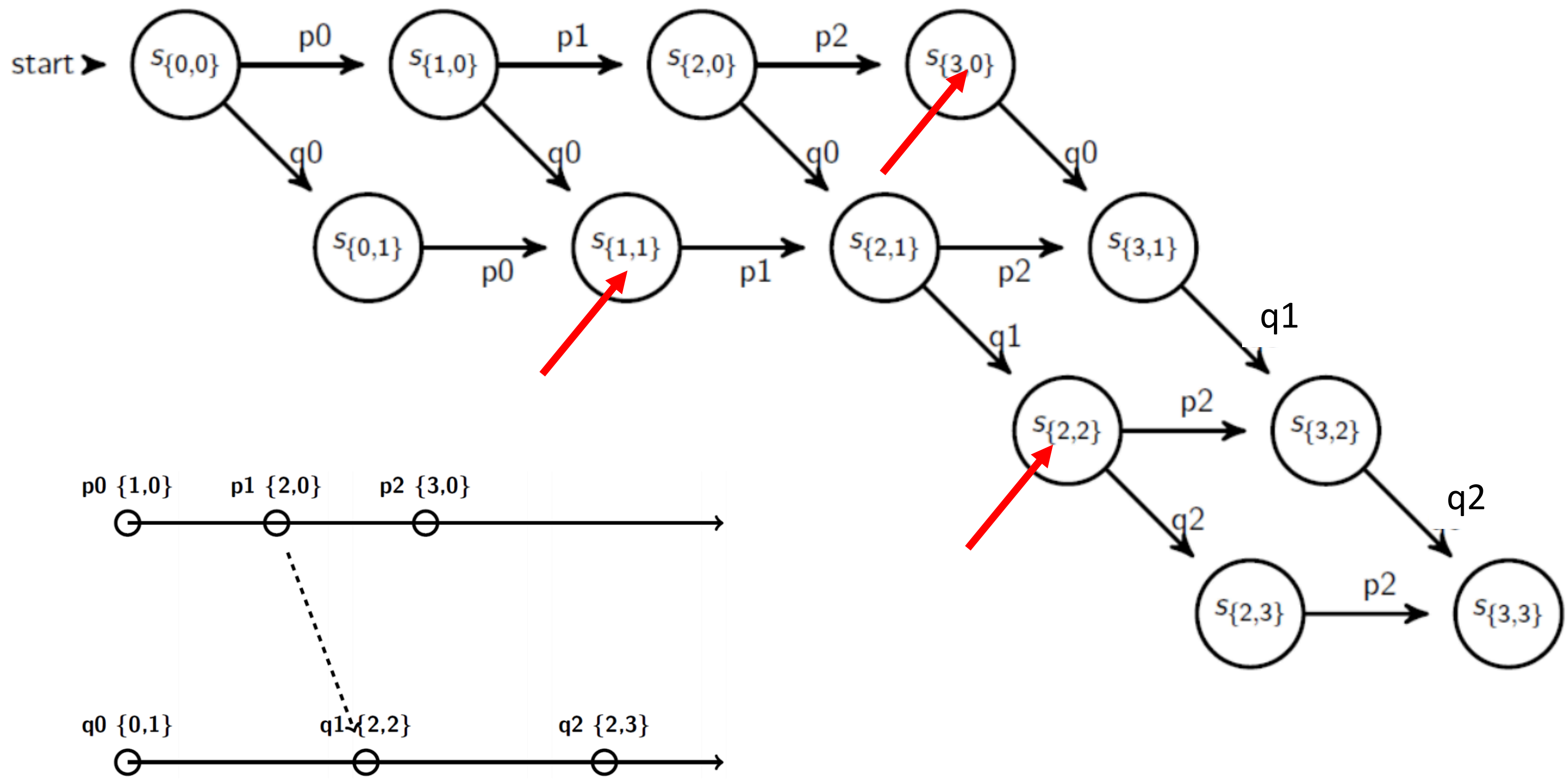
Liveness

- **Liveness = guarantee that something good will happen, eventually**
- **Examples:**
  - Guarantee that a distributed computation will terminate.
  - "Completeness" in failure detectors.
  - All processes eventually decide on a value.
- A global state S_0 satisfies a **liveness** property P iff:
  - $liveness(P(S_0)) \equiv \forall L \in$ linearizations from S_0, L passes through a S_L & $P(S_L)$ = true
  - For any linearization starting from S_0, P is true for some state S_L reachable from S_0.

Liveness Example

If predicate is true only in the marked states, does it satisfy liveness?

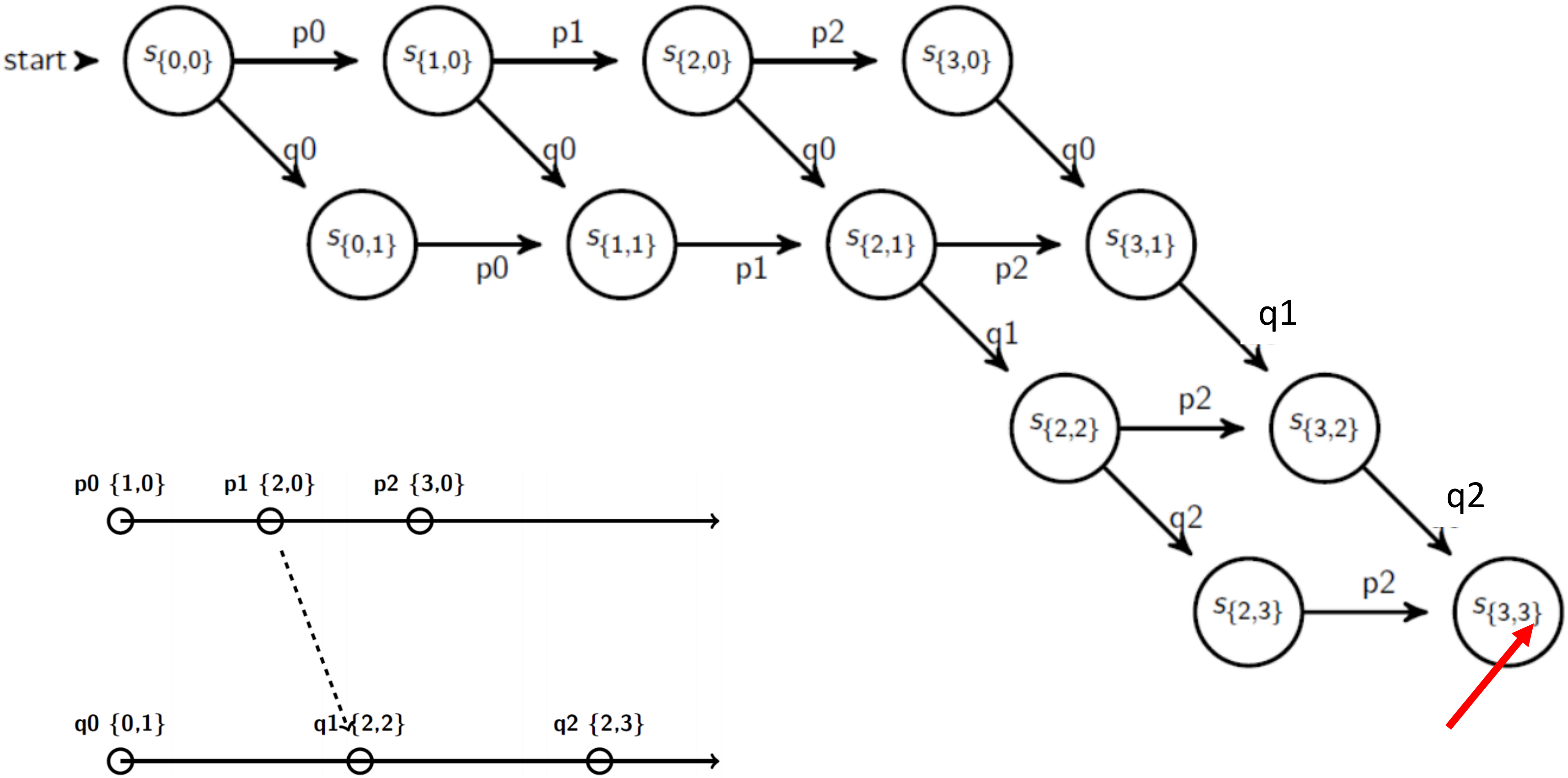
No



Liveness Example

If predicate is true only in the marked states, does it satisfy liveness?

Yes



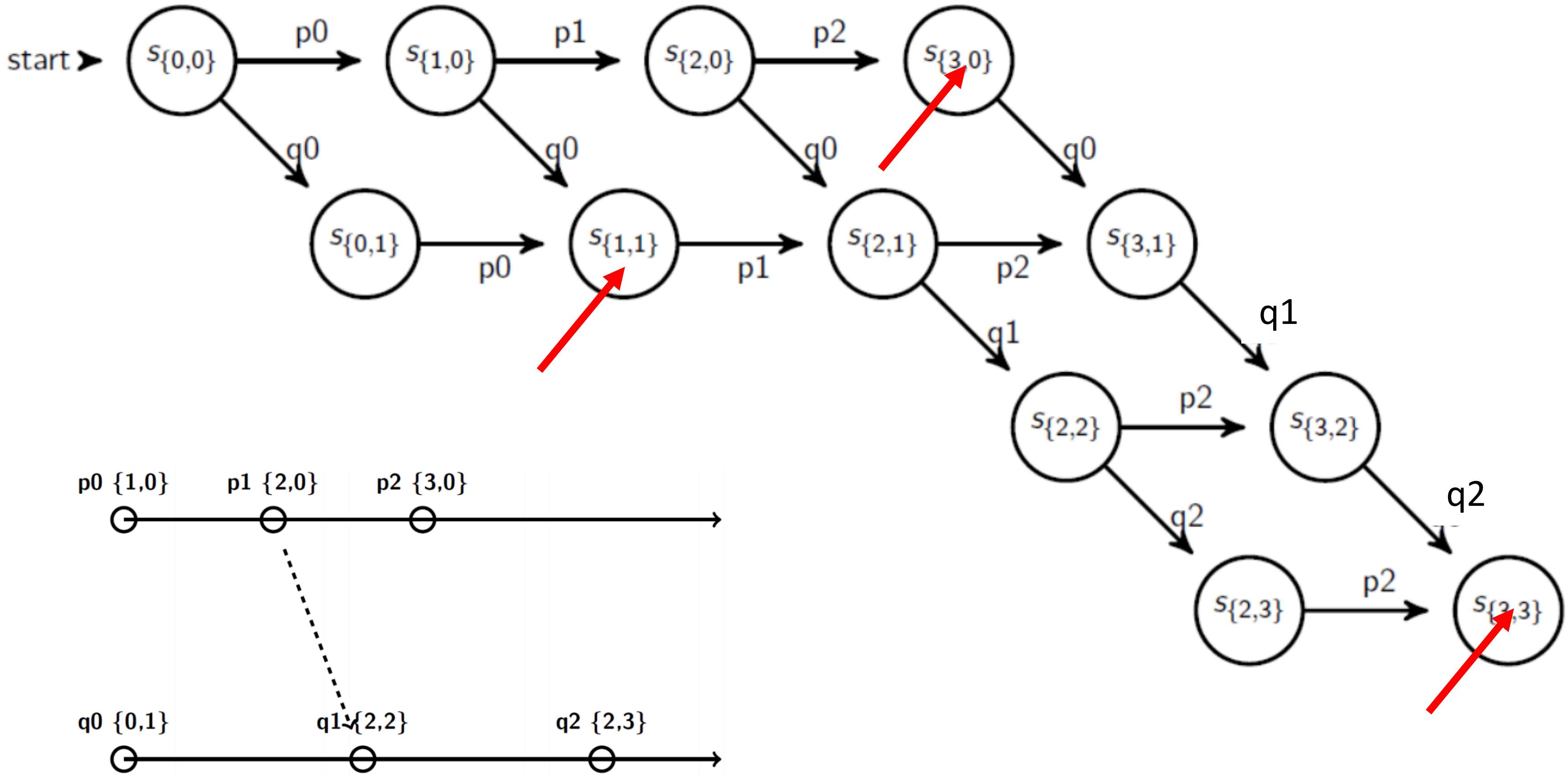
Safety

- **Safety = guarantee that something bad will never happen.**
- **Examples:**
  - There is no deadlock in a distributed transaction system.
  - "Accuracy" in failure detectors.
  - No two processes decide on different values.
- A global state S_0 satisfies a **safety** property P iff:
  - $safety(P(S_0)) \equiv \forall S$ reachable from S_0, $P(S) = true$.
  - For all states S reachable from S_0, P(S) is true.

Safety Example

If predicate is true only in the marked states, does it satisfy safety?

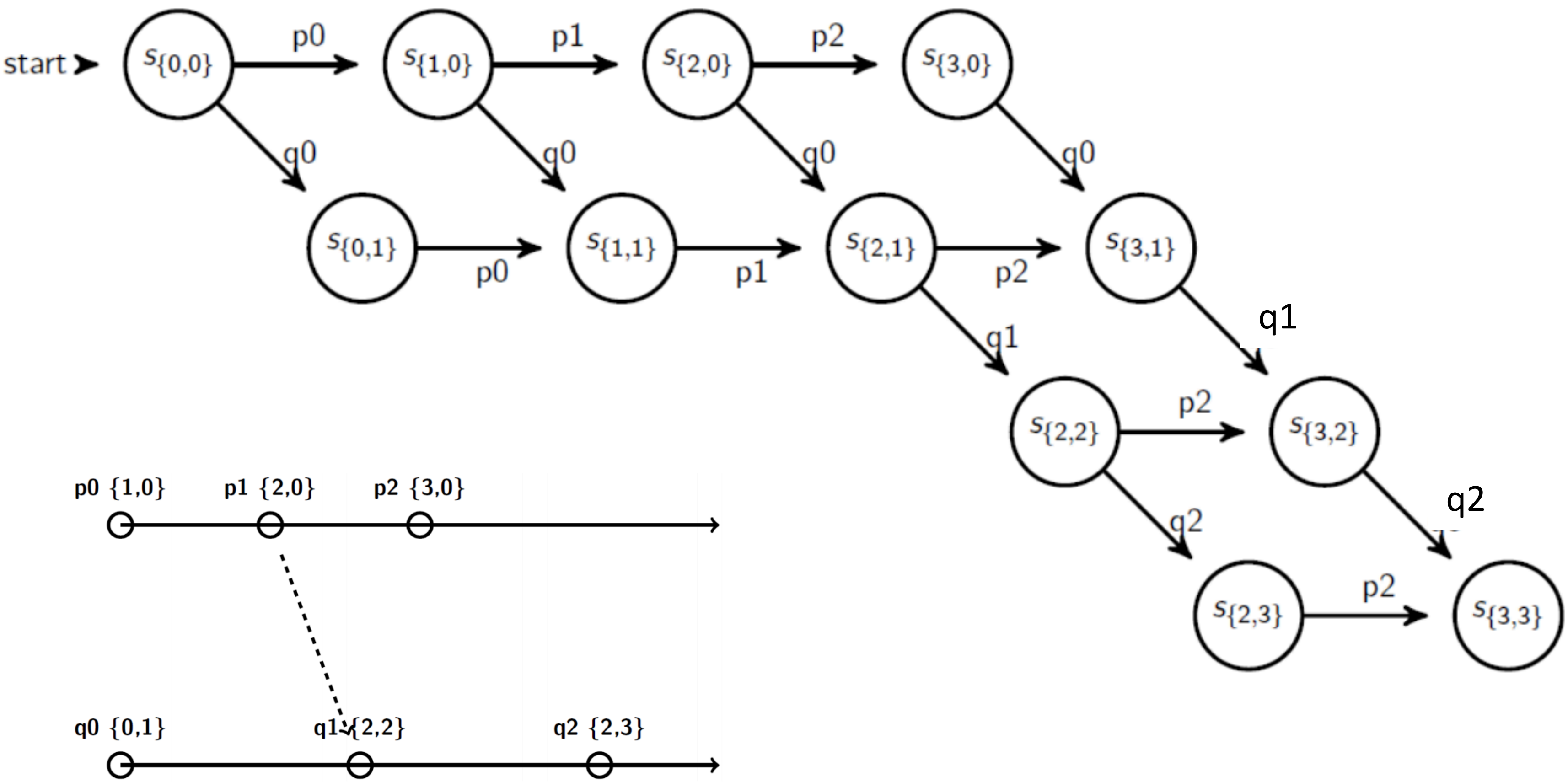
No



Safety Example

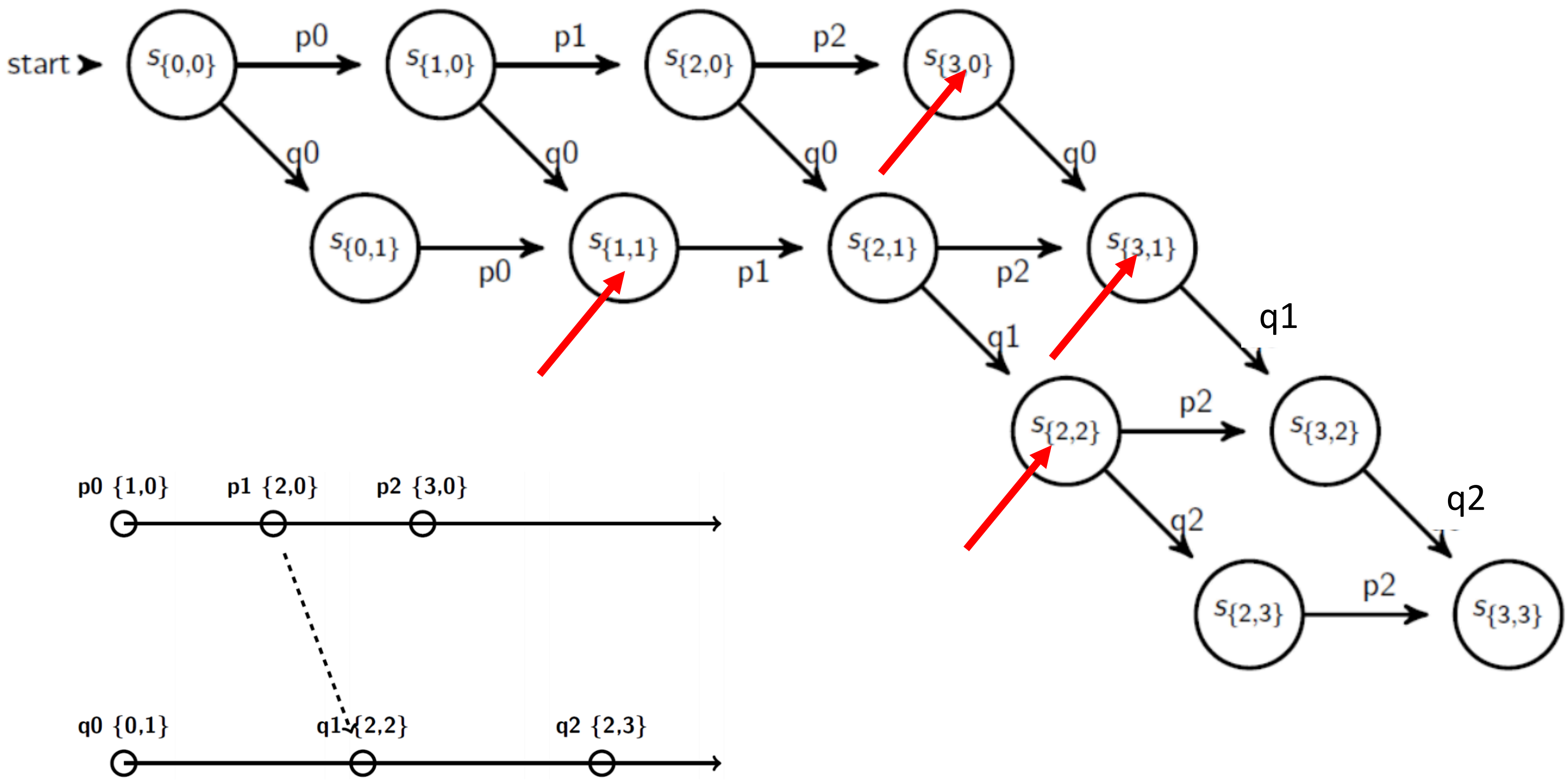
If predicate is true only in the **unmarked** states, does it satisfy safety?

Yes



Liveness Example

Technically satisfies liveness, but difficult to capture or reason about.



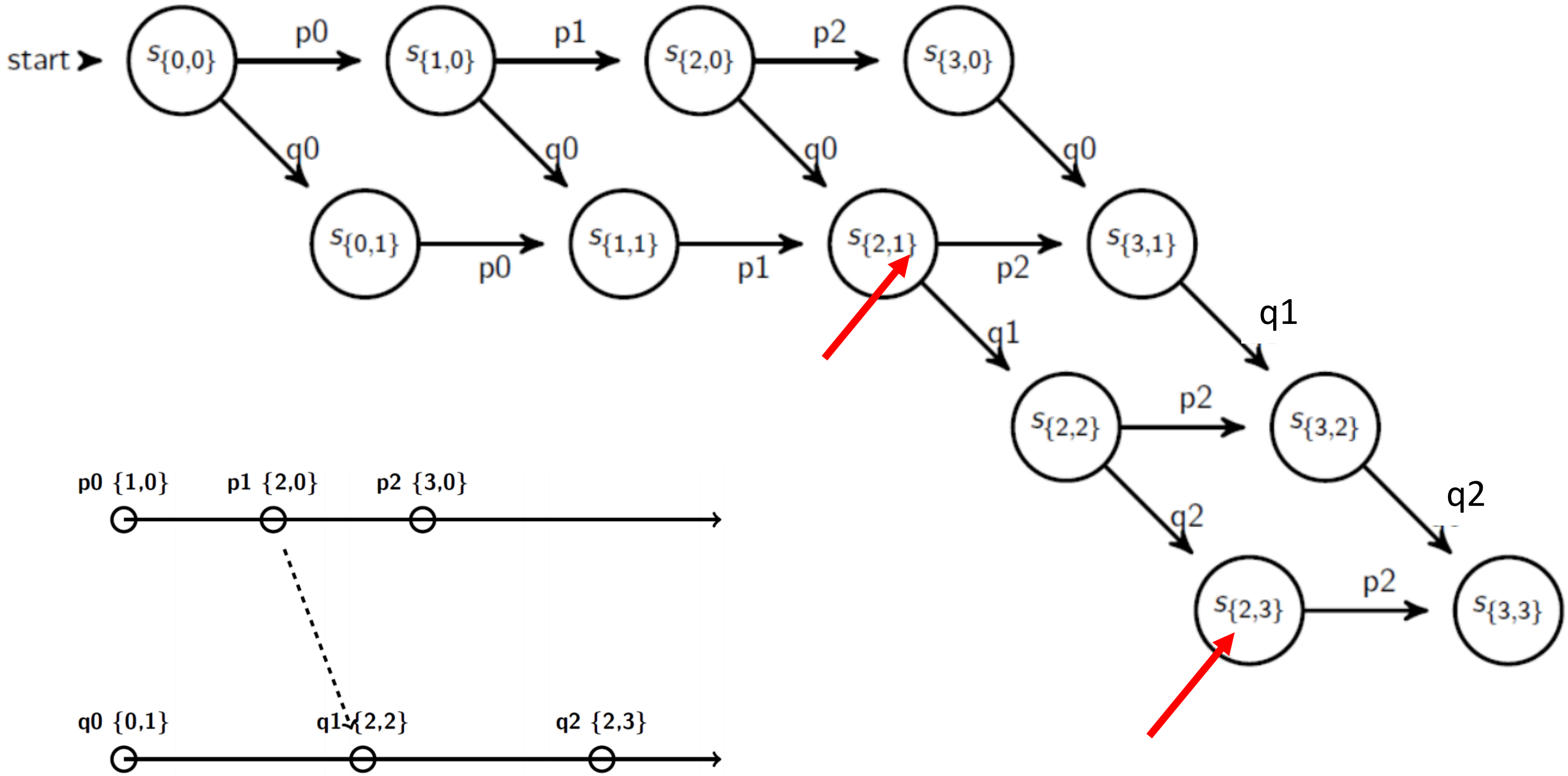
Stable Global Predicates

- Stable = once true, stays true forever afterwards

Stable Global Predicates

If predicate is true only in the marked states, is it stable?

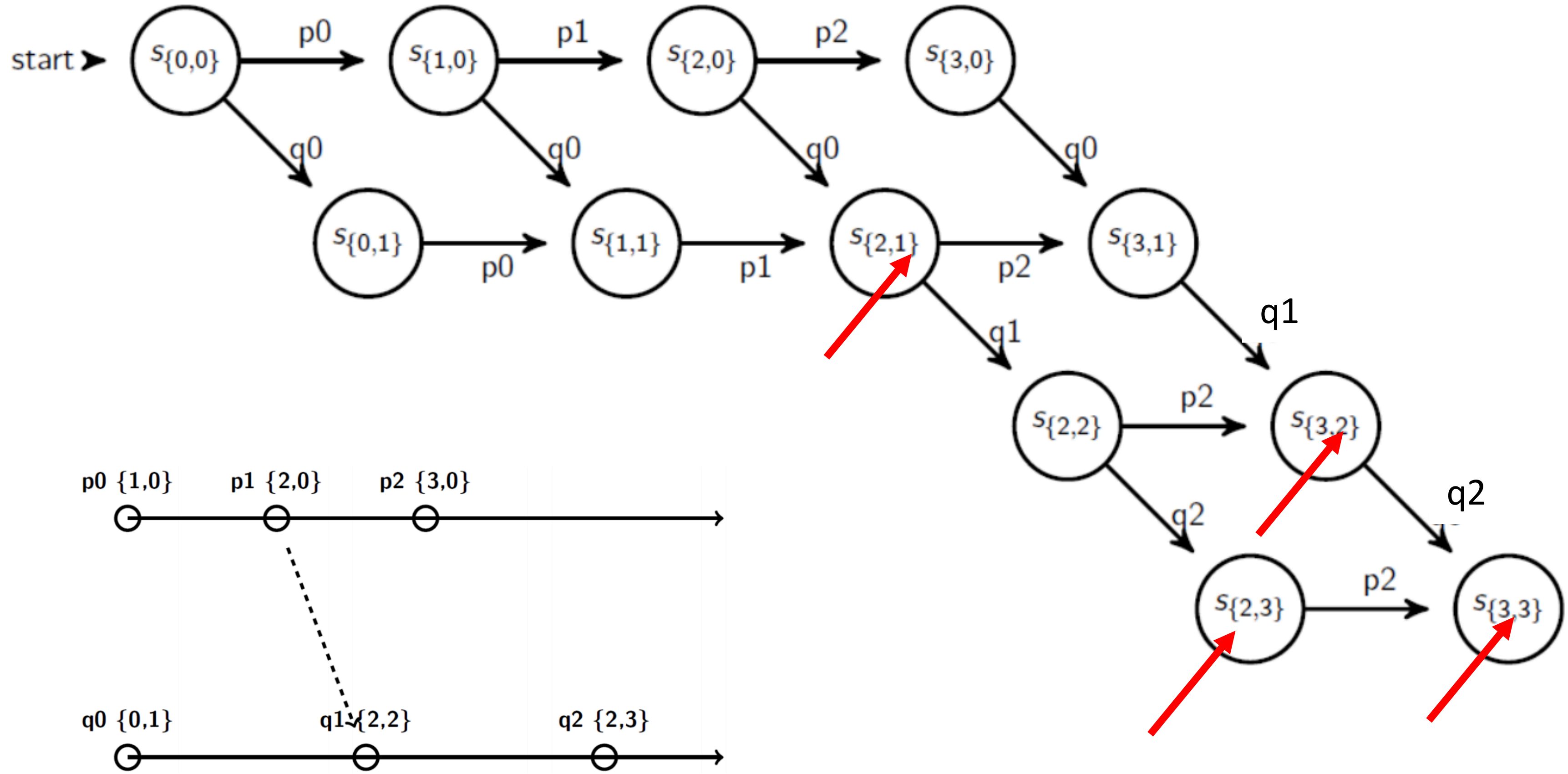
No



Stable Global Predicates

If predicate is true only in the marked states, is it stable?

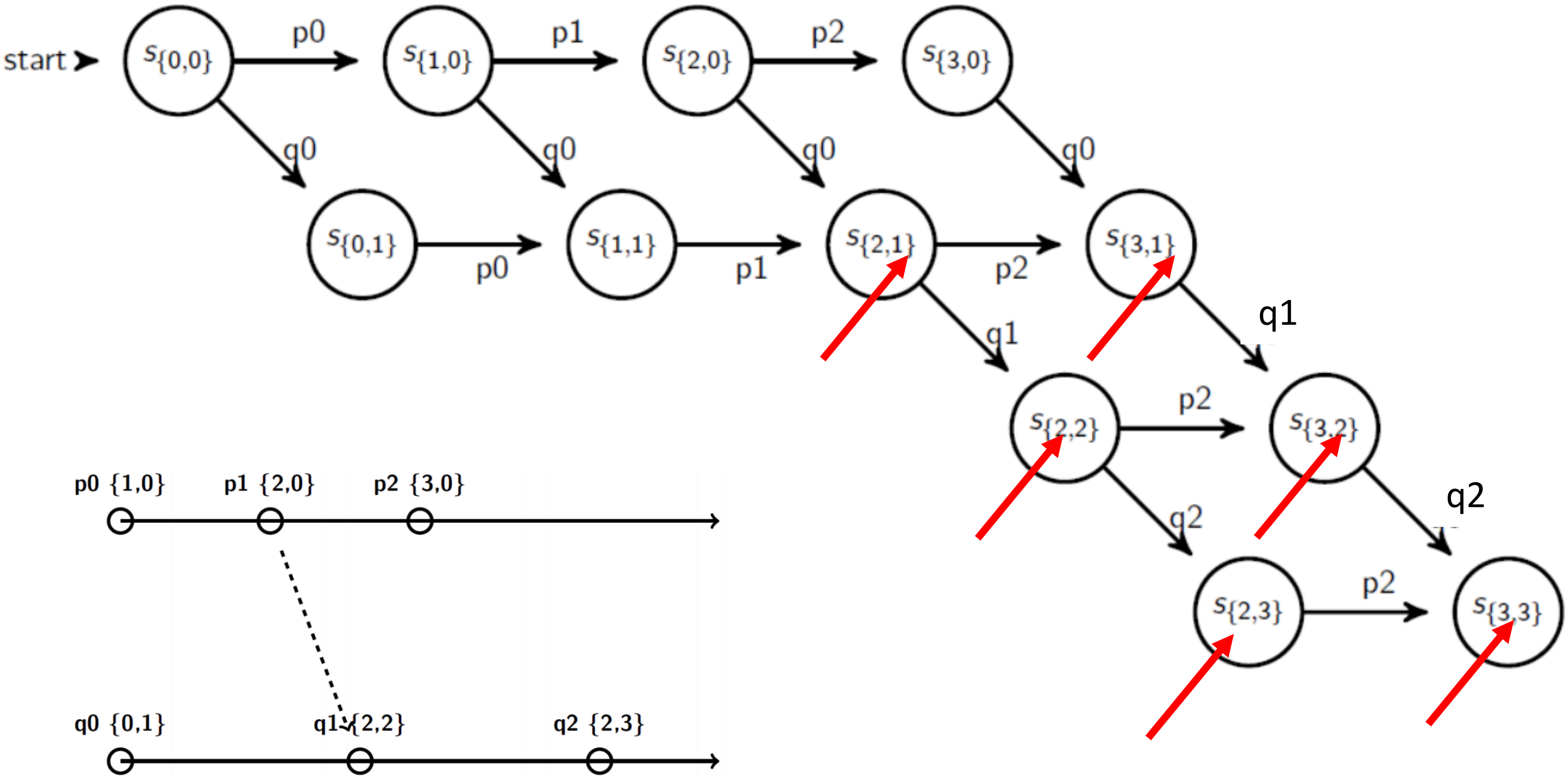
No



Stable Global Predicates

If predicate is true only in the marked states, is it stable?

Yes



Stable Global Predicates

- Stable = once true, stays true forever afterwards
- Stable liveness examples
  - Computation has terminated.
- Stable non-safety examples
  - There is a deadlock.
  - An object is orphaned (no pointers point to it).
- *All stable global properties can be detected using the Chandy-Lamport algorithm.*

Global Snapshot Summary

- The ability to calculate global snapshots in a distributed system is very important.
- But don't want to interrupt running distributed application.
- Chandy-Lamport algorithm calculates global snapshot.
- Obeys causality (creates a consistent cut).
- Can be used to detect stable global properties.
- Safety vs. Liveness.

Today's agenda

- Wrap-up global states and snapshots
  - Chapter 14.5
- Multicast
  - Chapter 15.4

Communication modes

- **Unicast**
  - Messages are sent from exactly one process to one process.
- **Broadcast**
  - Messages are sent from exactly one process to all processes on the network.
- **Multicast**
  - Messages broadcast within a group of processes.
  - A multicast message is sent from any one process to the group of processes on the network.

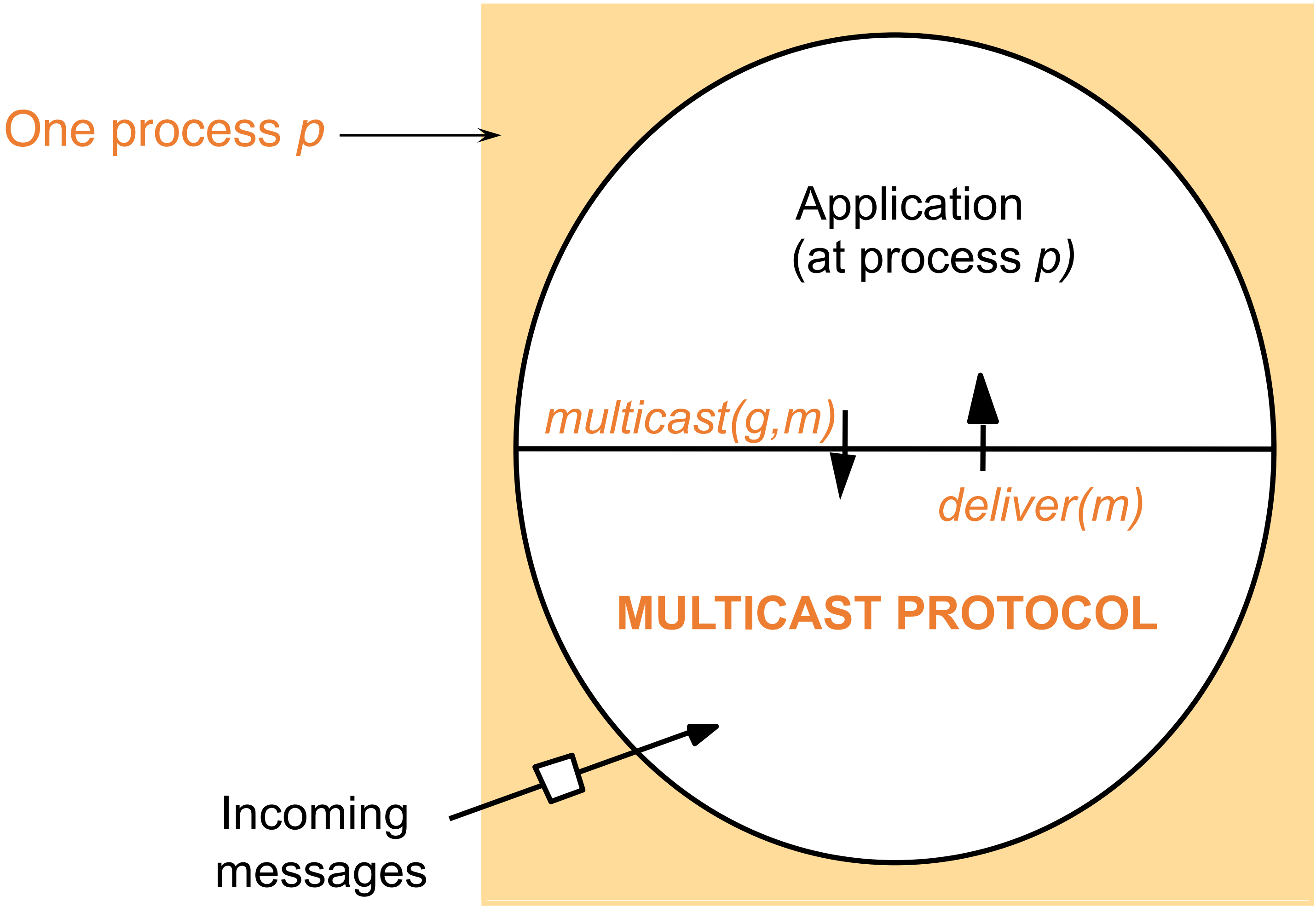
Where is multicast used?

- Distributed storage
  - Write to an object are multicast across replica 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.
- ……

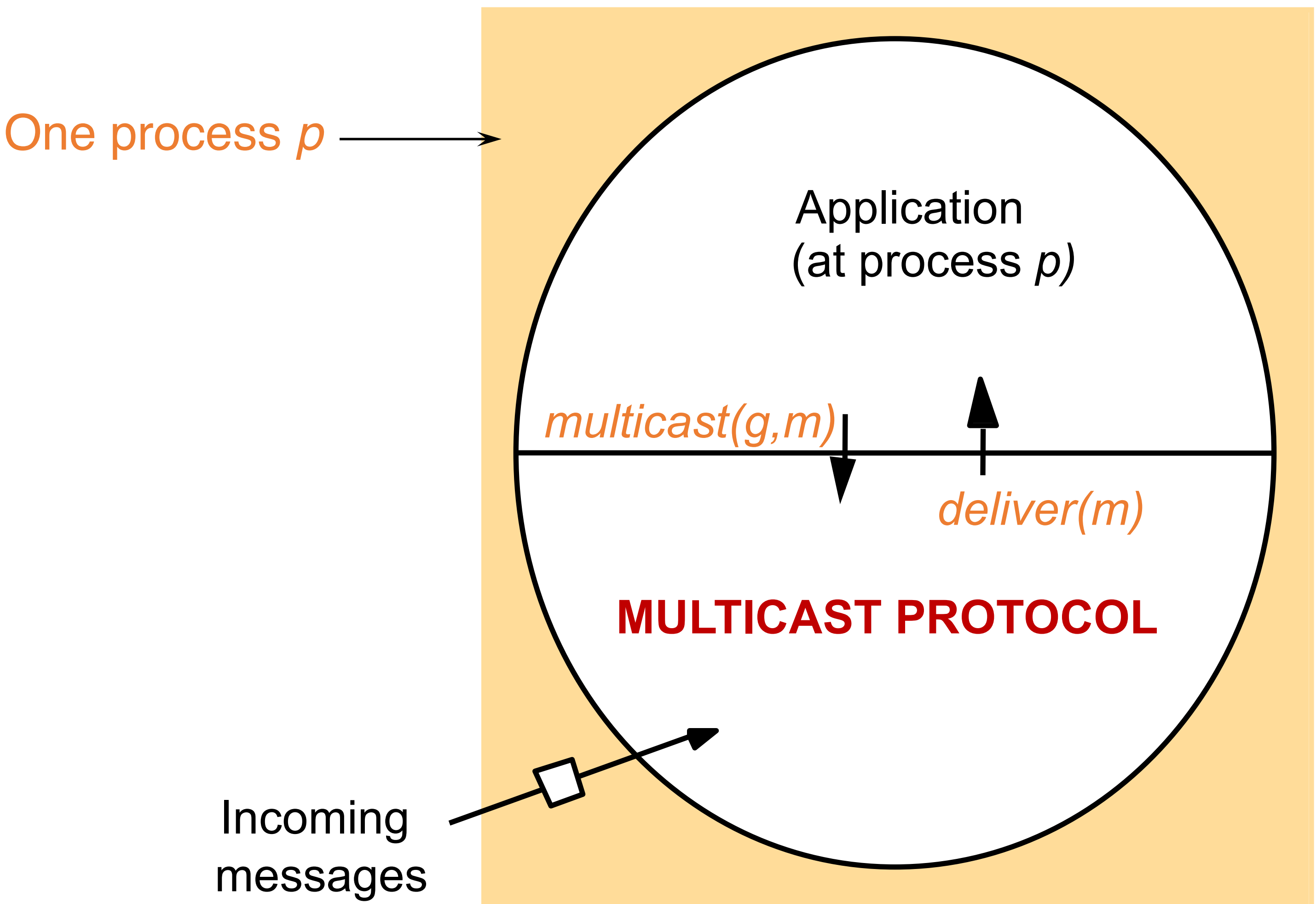
Communication modes

- **Unicast**
  - Messages are sent from exactly one process to one process.
    - *Best effort:* if a message is delivered it would be intact; no reliability guarantees.
    - *Reliable:* guarantees delivery of messages.
    - *In order:* messages will be delivered in the same order that they are sent.
- **Broadcast**
  - Messages are sent from exactly one process to all processes on the network.
- **Multicast**
  - Messages broadcast within a group of processes.
  - A multicast message is sent from any one process to the group of processes on the network.
  - *How do we define (and achieve) reliable or ordered multicast?*

What we are designing in this class?



What we are designing in this class?



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):

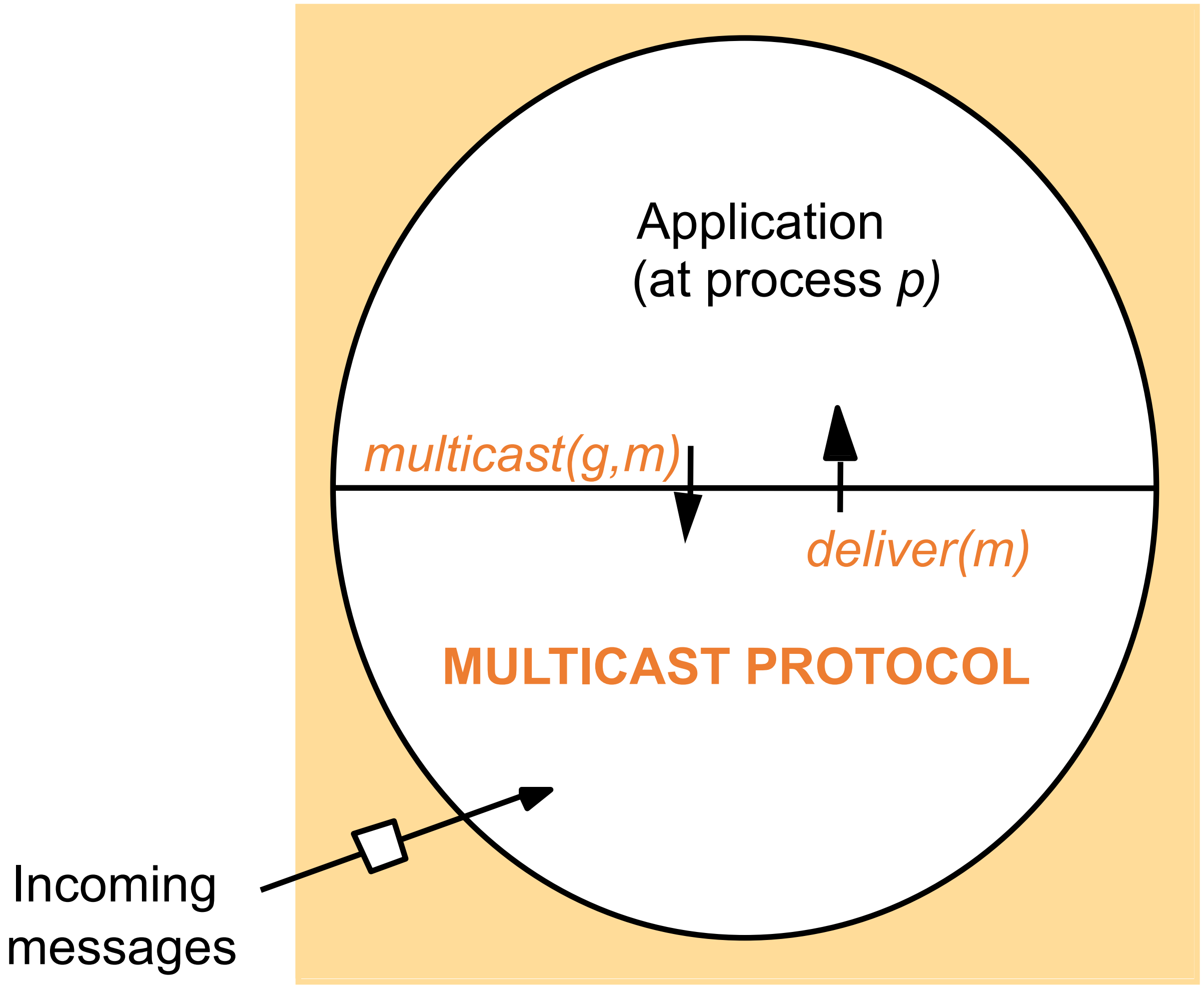
    for each process p in g, send (p,m).

    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?*

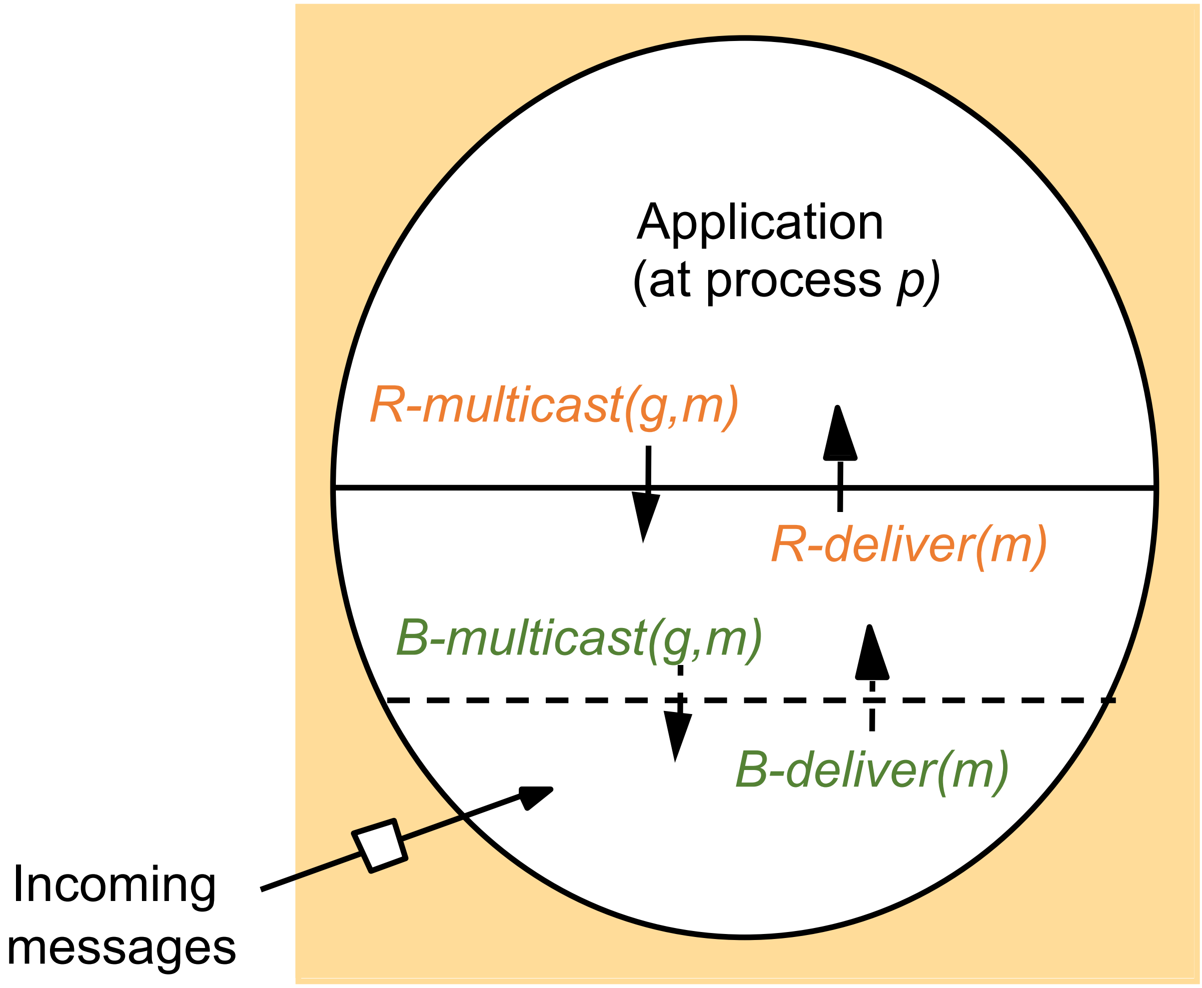
Reliable Multicast (R-Multicast)

- **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



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 with g = group(m)

    if ($m \notin$ Received):

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

        if ($q \neq p$): 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* 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

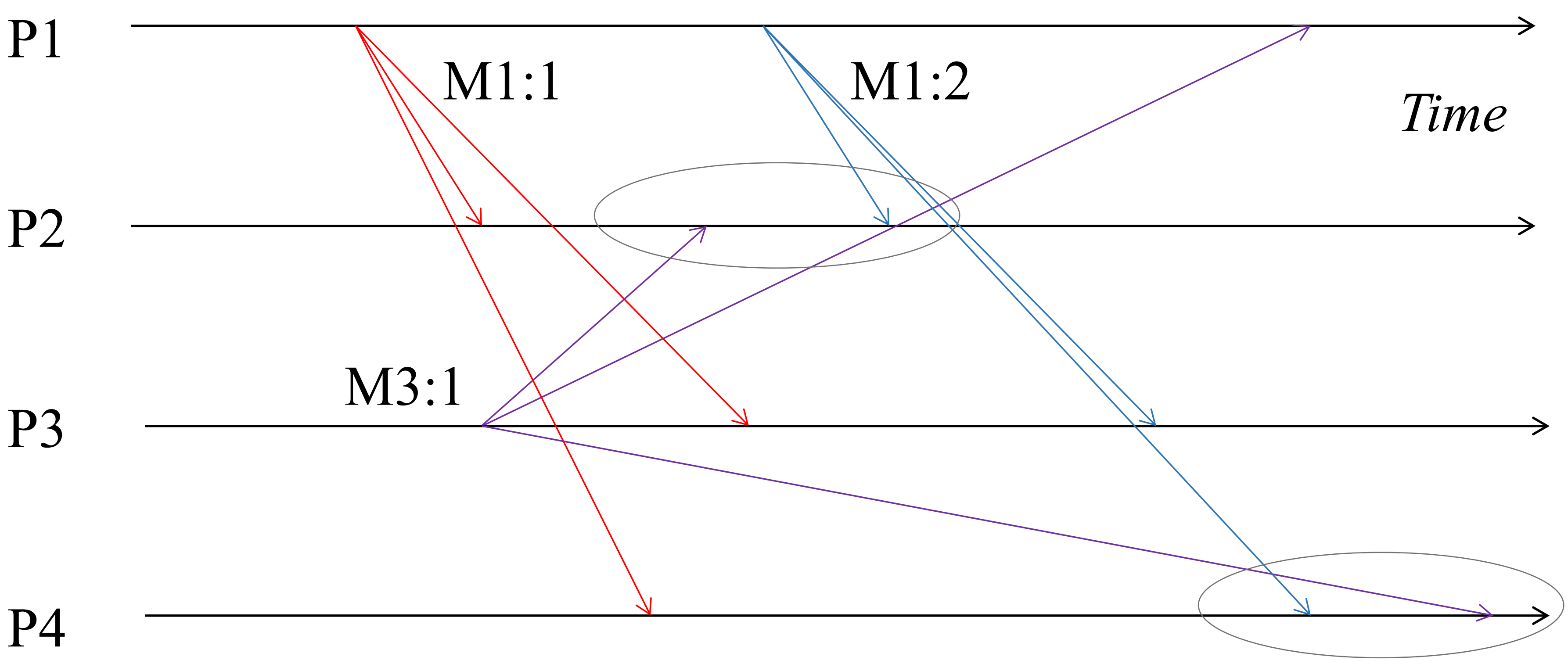
- Three popular flavors implemented by several multicast protocols:
  1. FIFO ordering
  2. Causal ordering
  3. Total ordering

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.

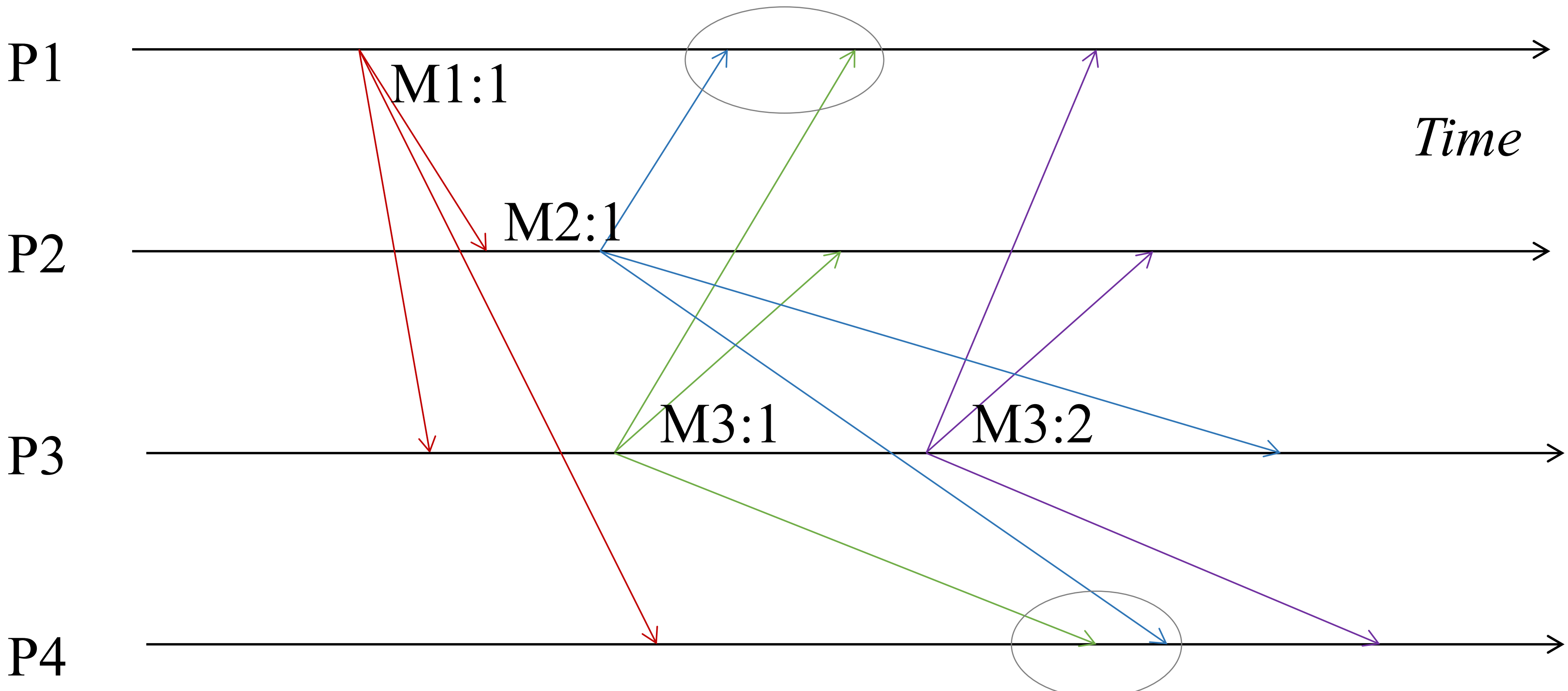
Order of delivery of M3:1 and M1:2 could be different at different receivers.

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.*
  - *(→ is Lamport's happens-before)*
  - (→ counts messages **delivered** to the application, rather than all network messages)

Causal Order: Example



M3:1 → M3:2, M1:1 → M3:1, M1:1 → M2:1, 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 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.

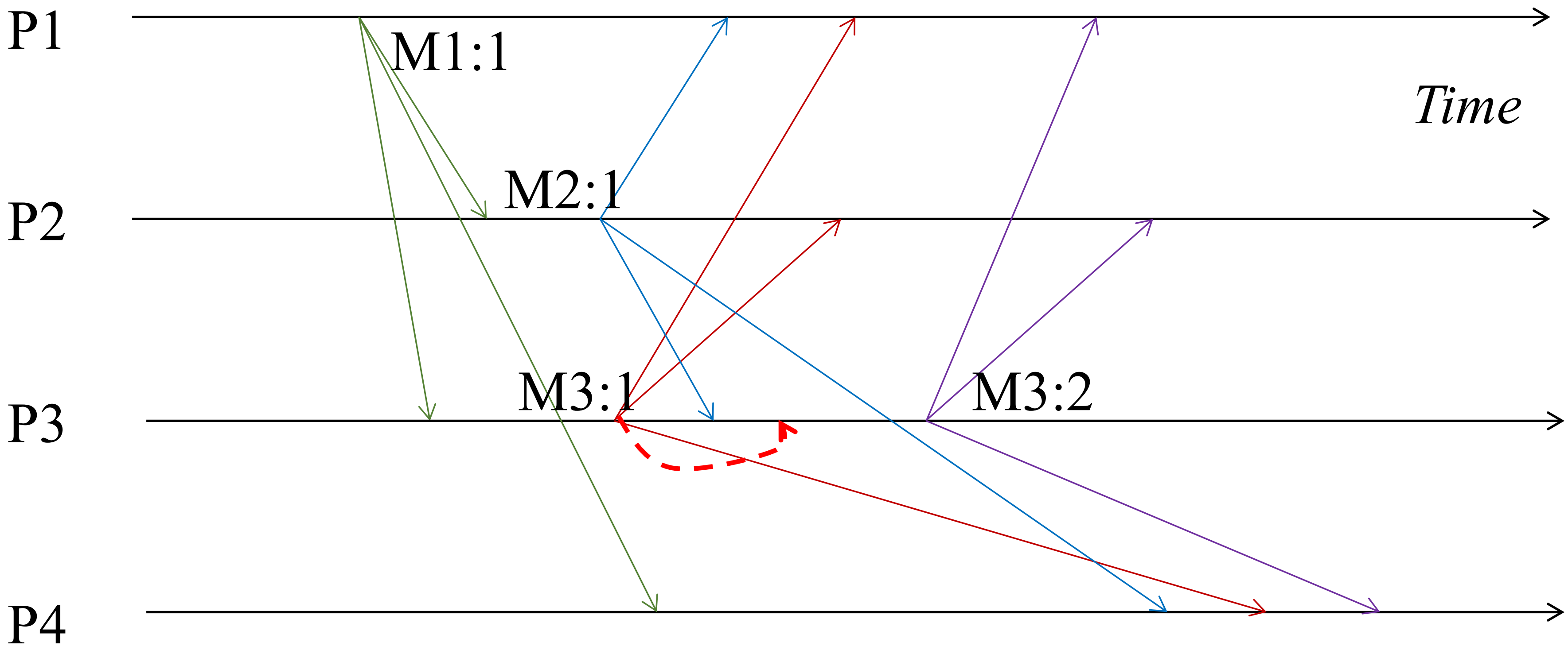
Where is causal ordering useful?

- 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.

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

- 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.

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?