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.
- AI policy:
  - Same as web search. Can't use AI in any way that trivializes the assignment (e.g. auto-generating significant piece code/solution).
  - Ok to use it for boiler-plate code (create TCP sockets, create a Go channel, etc).
- HW1 Q1 clarification: please see Campuswire post #75. Homework PDF on the course website has been updated too.
- Reminder to share your name when you speak up in class.

Today's agenda

- Global State (contd.)
  - Chapter 14.5
- Multicast
  - Chapter 15.4

Today's agenda

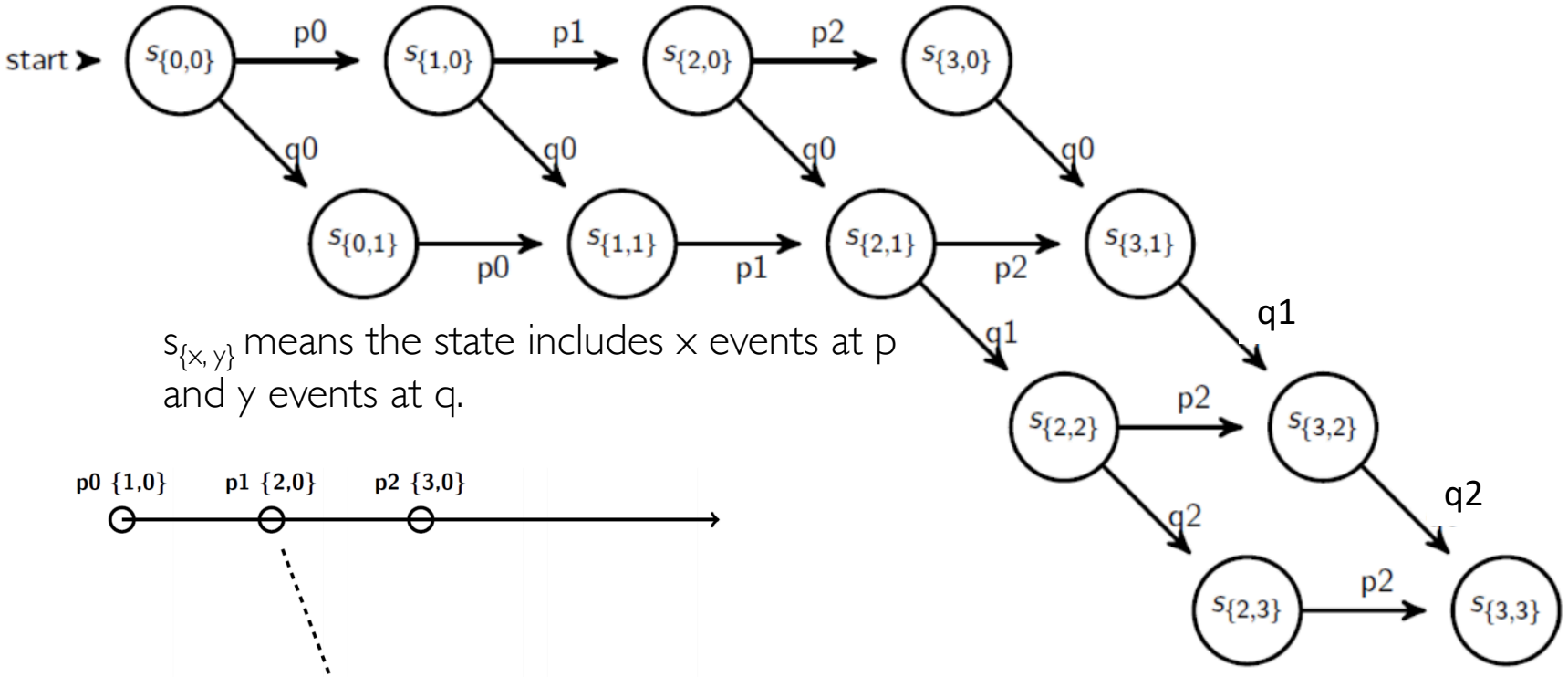
- Global State (contd.)
  - Chapter 14.5
- Multicast
  - Chapter 15.4

More notations and definitions

- A run is a total ordering of events in H that is consistent with each $\mathbf{h_i}$'s ordering.
- A linearization is a run consistent with happens-before ($\rightarrow$) relation in H.
- 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

Execution Lattice. Each path represents a linearization.



$s_{\{x,y\}}$ means the state includes x events at p and y events at q.

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

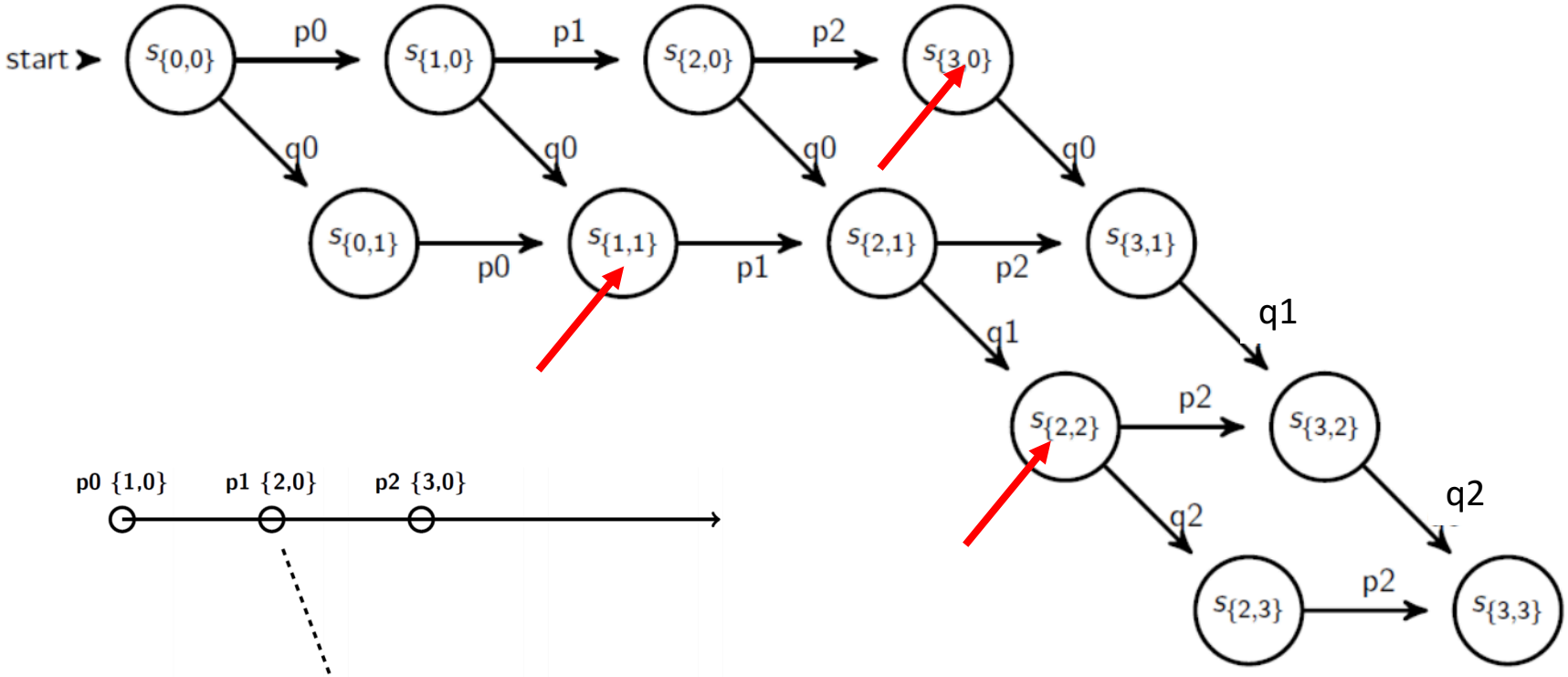
- Examples:
  - A distributed computation will terminate.
  - "Completeness" in failure detectors: the failure will be detected.
  - All processes will eventually decide on a value.

- A global state S_0 satisfies a **liveness** property P iff:
  - For all linearizations starting from S_0, P is true for some state S_L reachable from S_0.
  - liveness($P(S_0)$) ≡ $\forall L \in$ linearizations from S_0, L passes through a S_L & $P(S_L)$ = true

Liveness Example

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

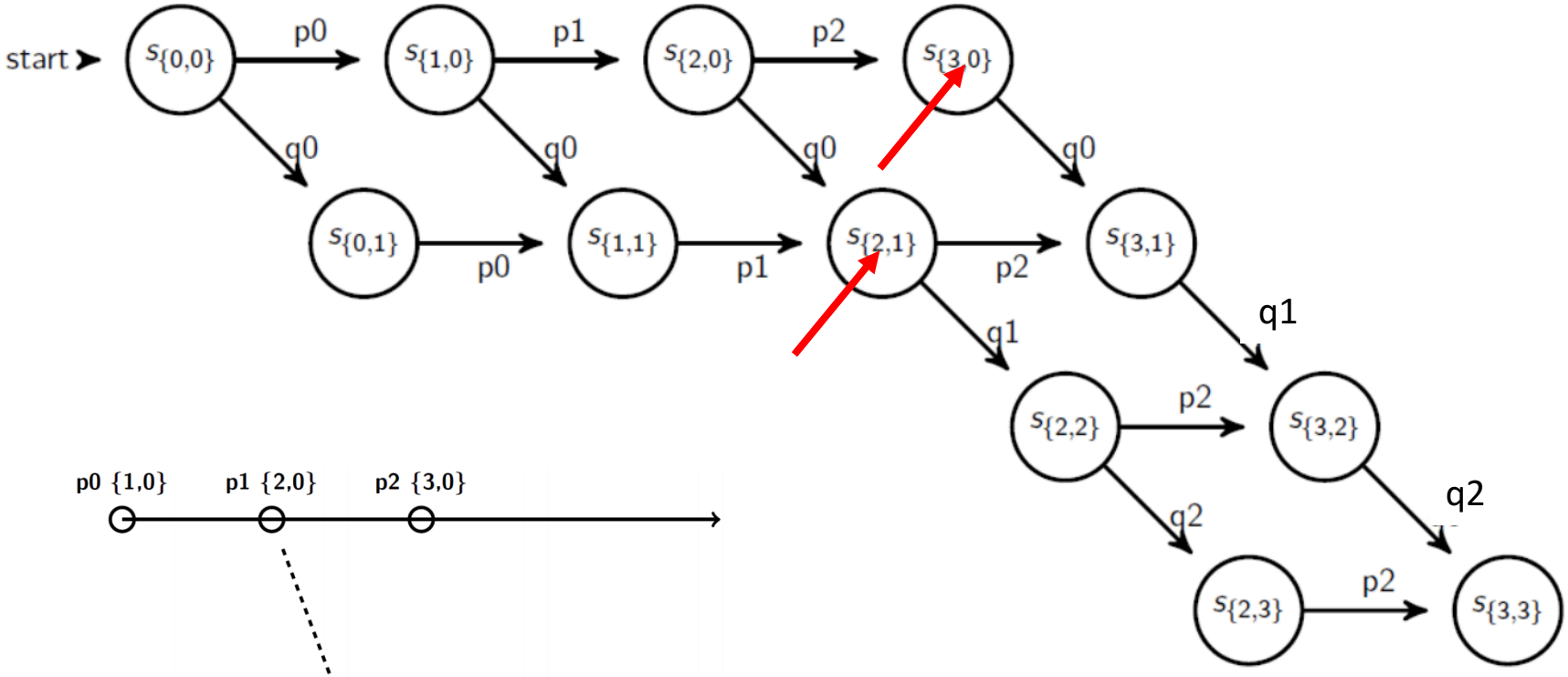
No



Liveness Example

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

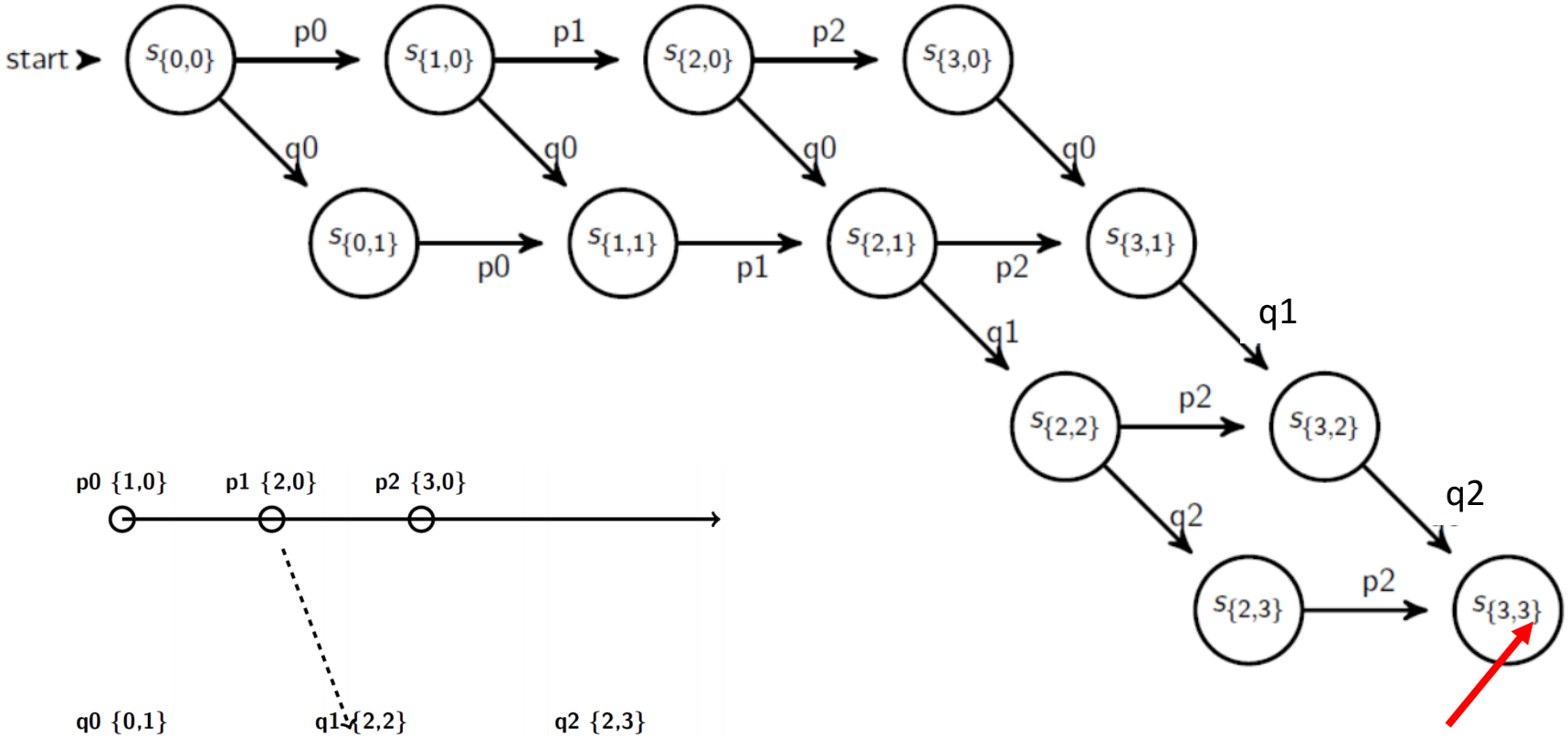
Yes



Liveness Example

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

Yes



Liveness

- Liveness = guarantee that something good will happen, eventually

- Examples:
  - A distributed computation will terminate.
  - "Completeness" in failure detectors: the failure will be detected.
  - All processes will eventually decide on a value.

- A global state S_0 satisfies a **liveness** property P iff:
  - liveness($P(S_0)$) ≡ $\forall L \in$ 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.

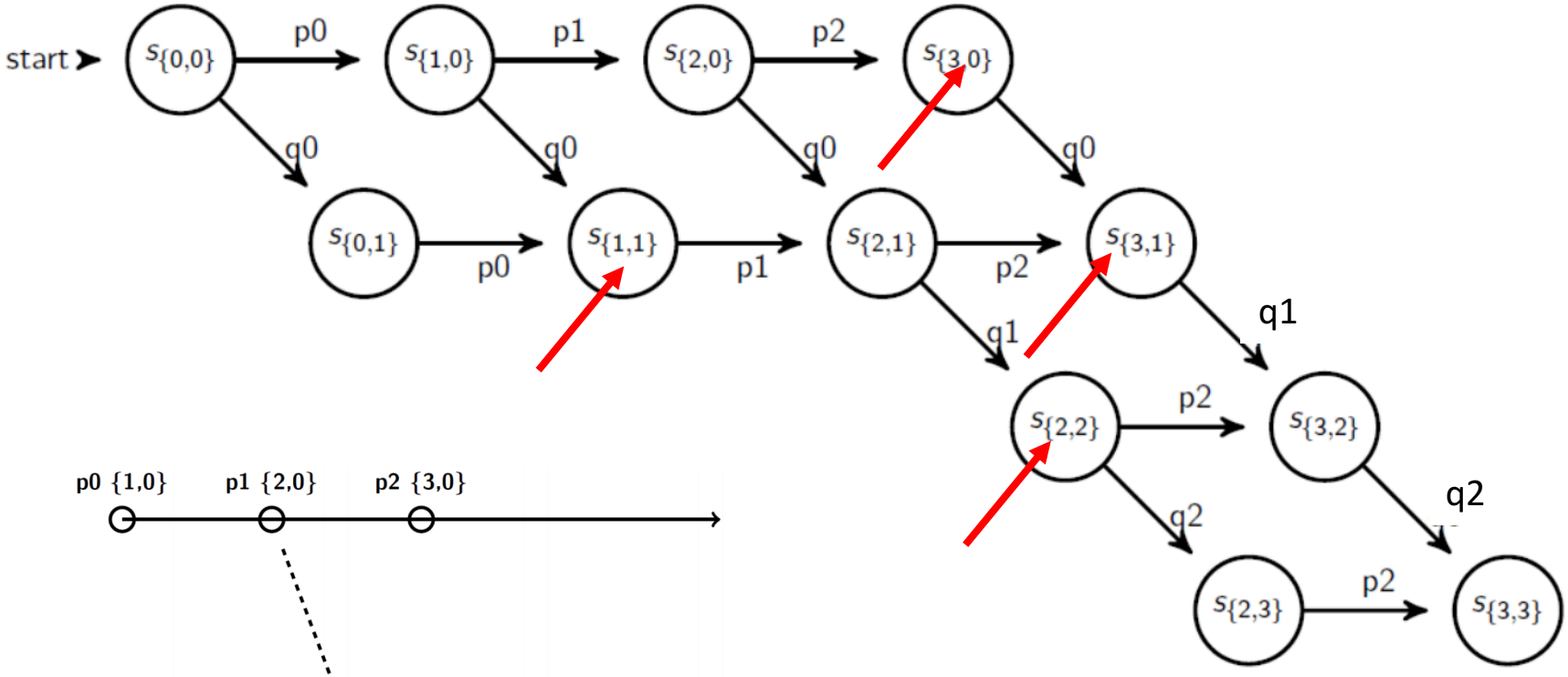
Safety

- Safety = guarantee that something bad will never happen.
- Examples:
  - There is no deadlock in a distributed transaction system.
  - "Accuracy" in failure detectors: an alive process is not detected as failed.
  - No two processes decide on different values.
- A global state S_0 satisfies a **safety** property P iff:
  - For all states S reachable from S_0, P(S) is true.
  - safety($P(S_0)$) ≡ $\forall$S reachable from S_0, P(S) = true.

Safety Example

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

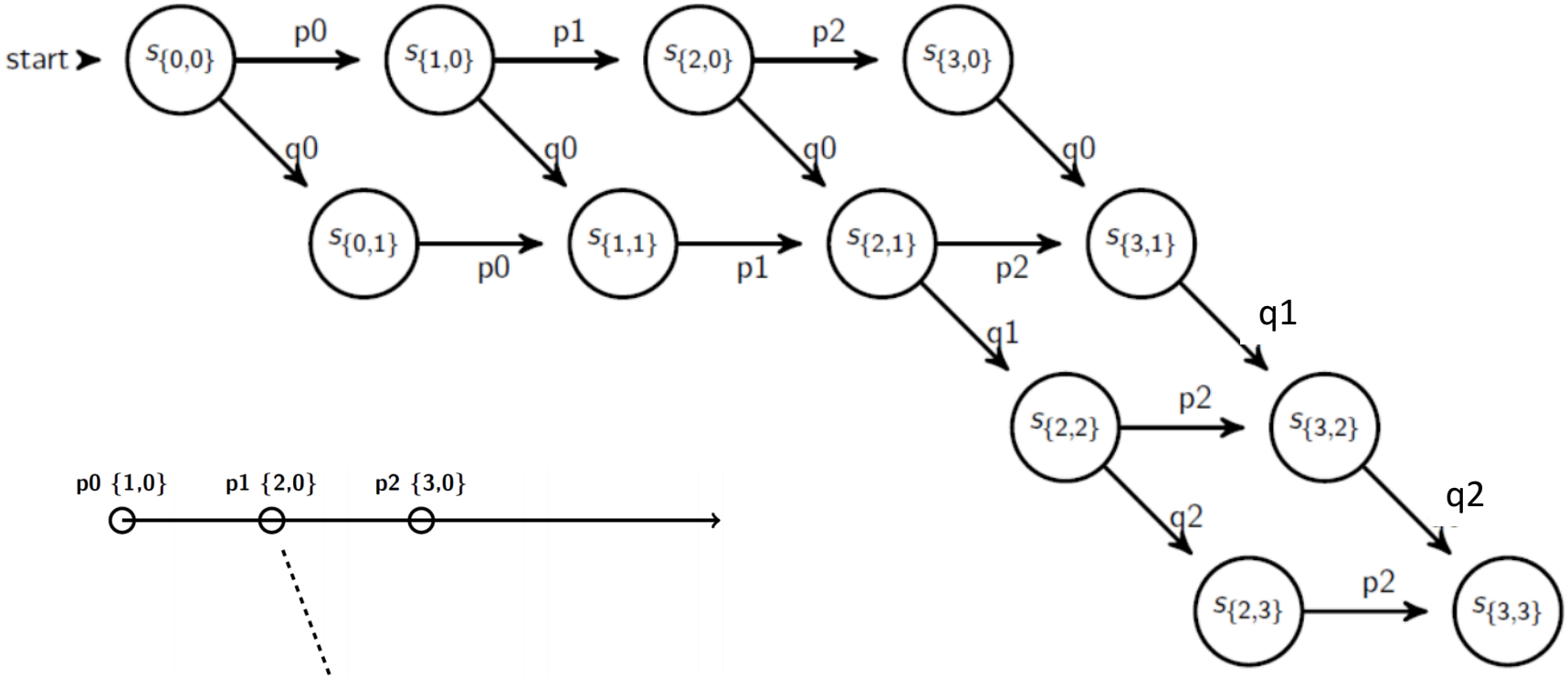
No



Safety Example

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

Yes

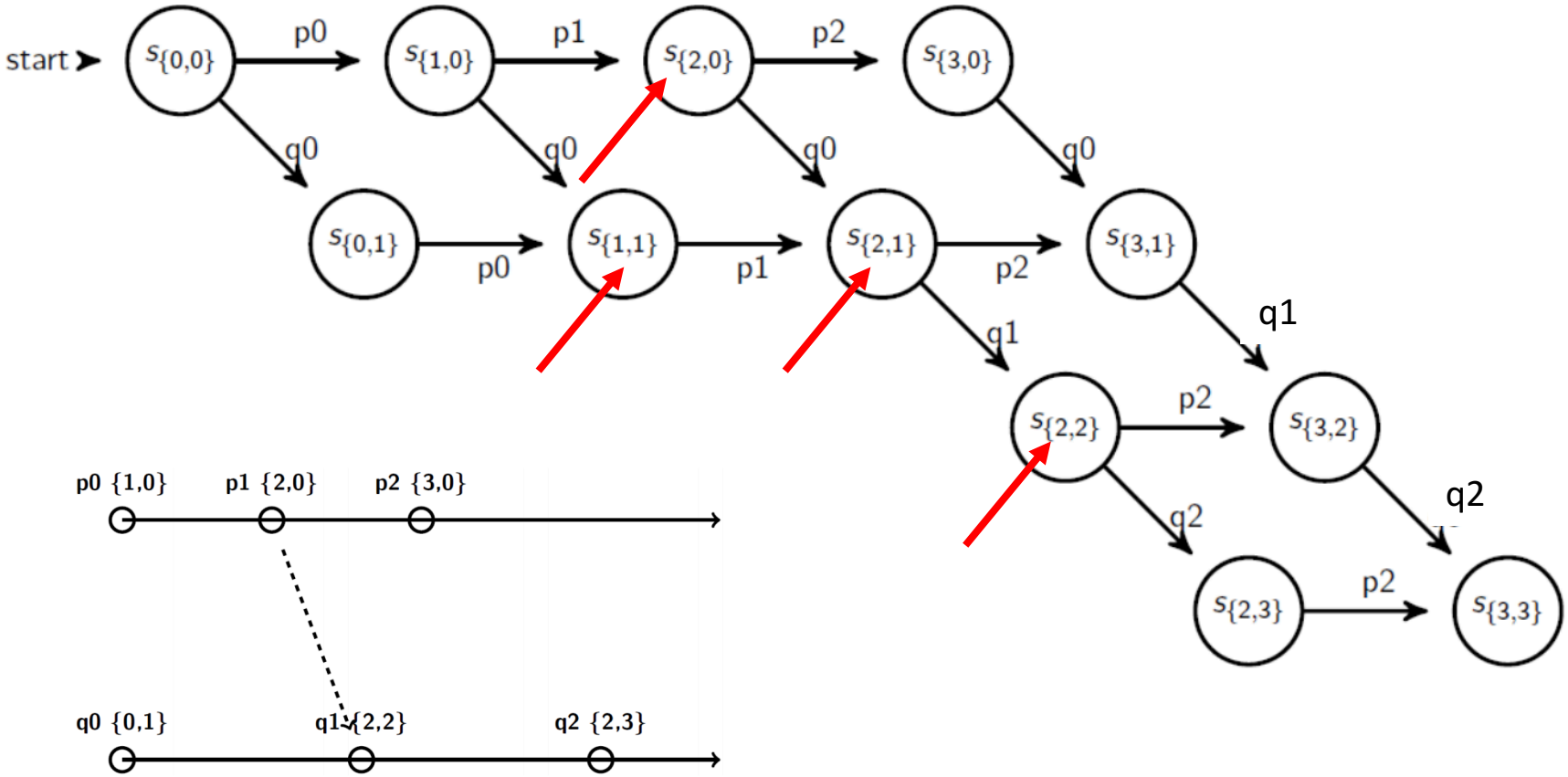


Safety

- Safety = guarantee that something bad will never happen.
- Examples:
  - There is no deadlock in a distributed transaction system.
  - "Accuracy" in failure detectors: an alive process is not detected as failed.
  - No two processes decide on different values.
- A global state S_0 satisfies a **safety** property P iff:
  - safety($P(S_0)$) ≡ $\forall S$ reachable from S_0, $P(S)$ = true.
  - For all states S reachable from S_0, P(S) is true.

Liveness Example

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



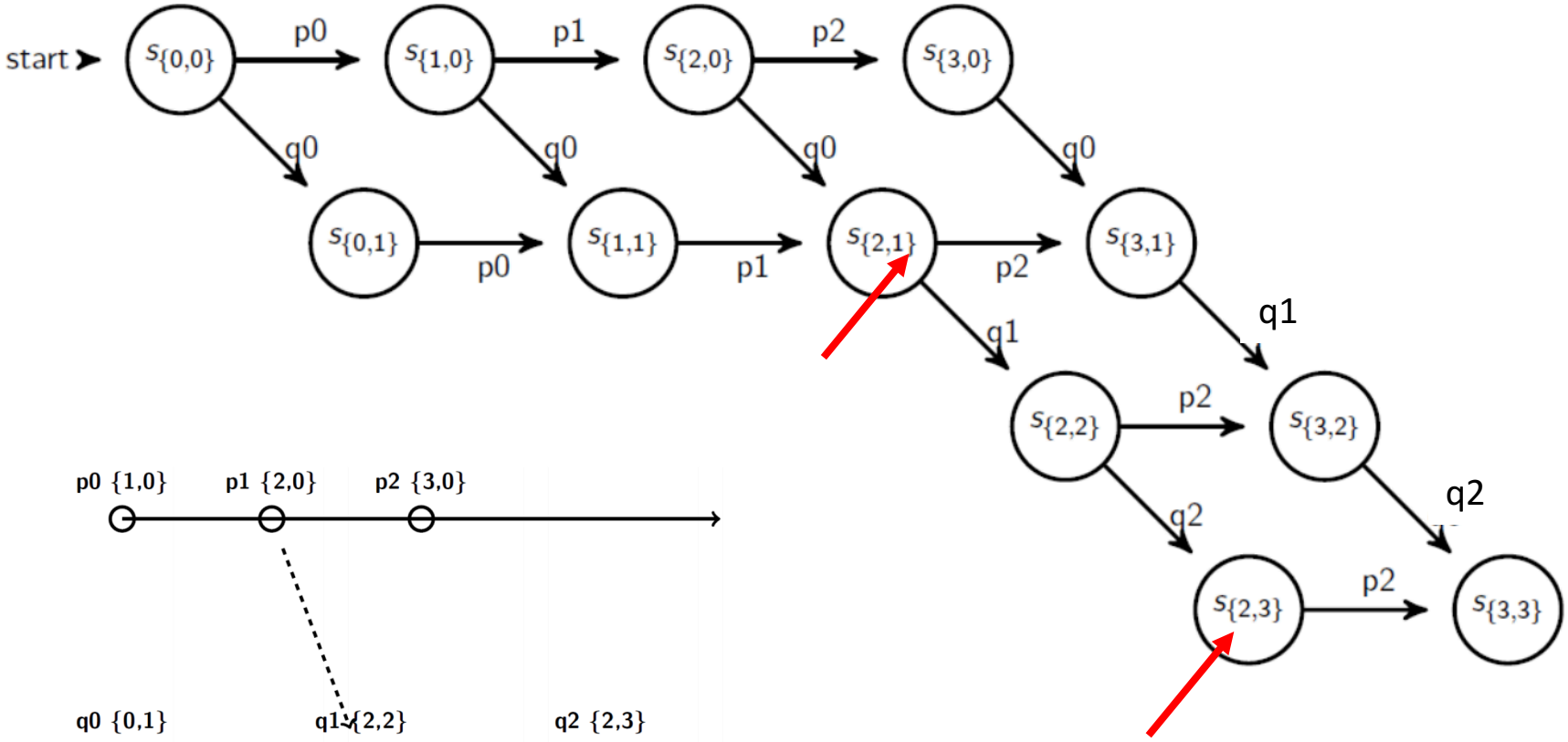
Stable Global Predicates

- once true, stays true forever afterwards (for stable liveness)

Stable Global Predicates

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

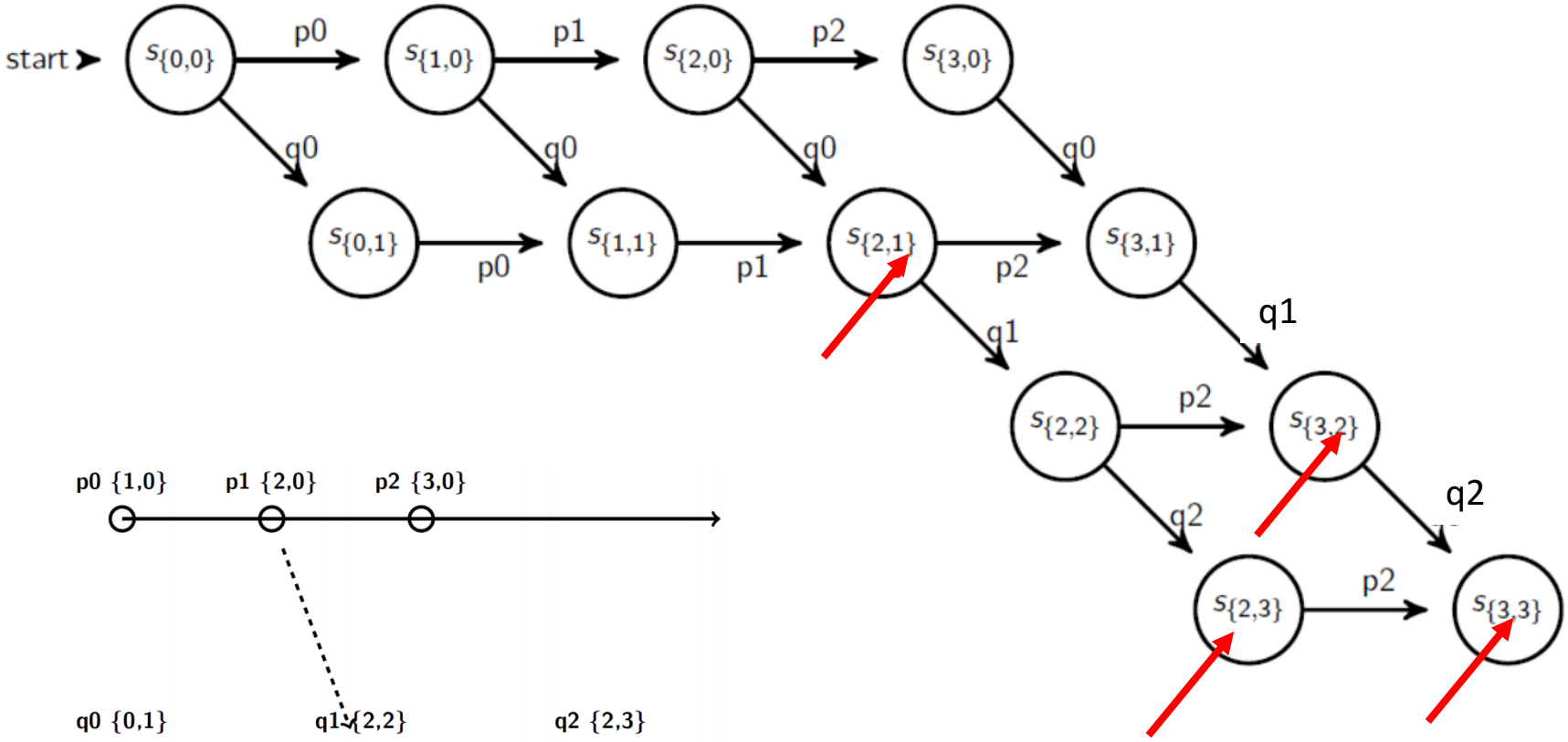
No



Stable Global Predicates

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

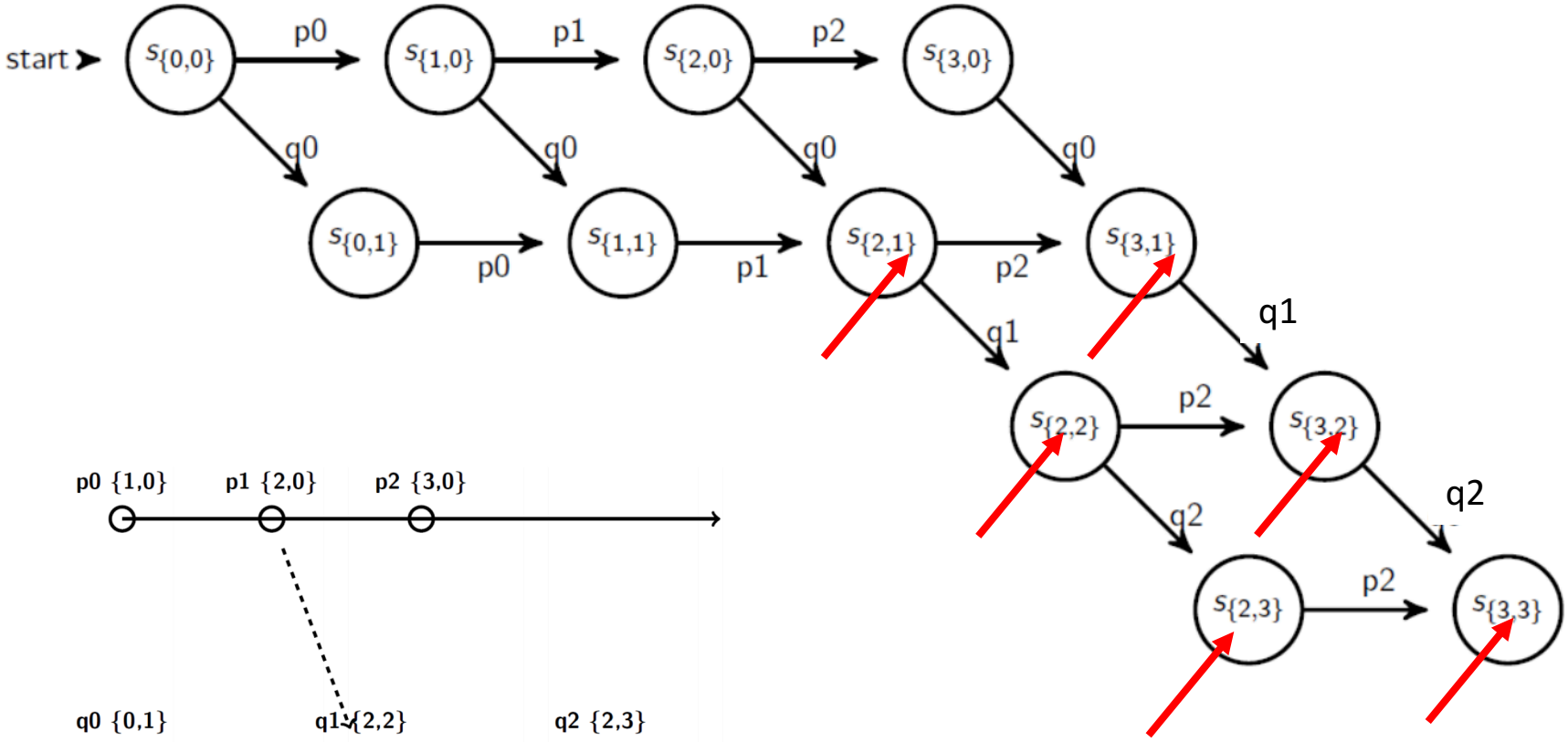
No



Stable Global Predicates

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

Yes



Stable Global Predicates

- once true for a state S, stays true for all states reachable from S (for stable liveness)
- once false for a state S, stays false for all states reachable from S (for stable non-safety)
- Stable liveness examples (once true, always true)
  - Computation has terminated.
- Stable non-safety examples (once false, always false)
  - There is no deadlock.
  - An object is not orphaned.
- *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 global properties.
- Safety vs. Liveness.

Today's agenda

- Global State (contd.)
  - Chapter 14.5

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

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 a group of processes on the network.

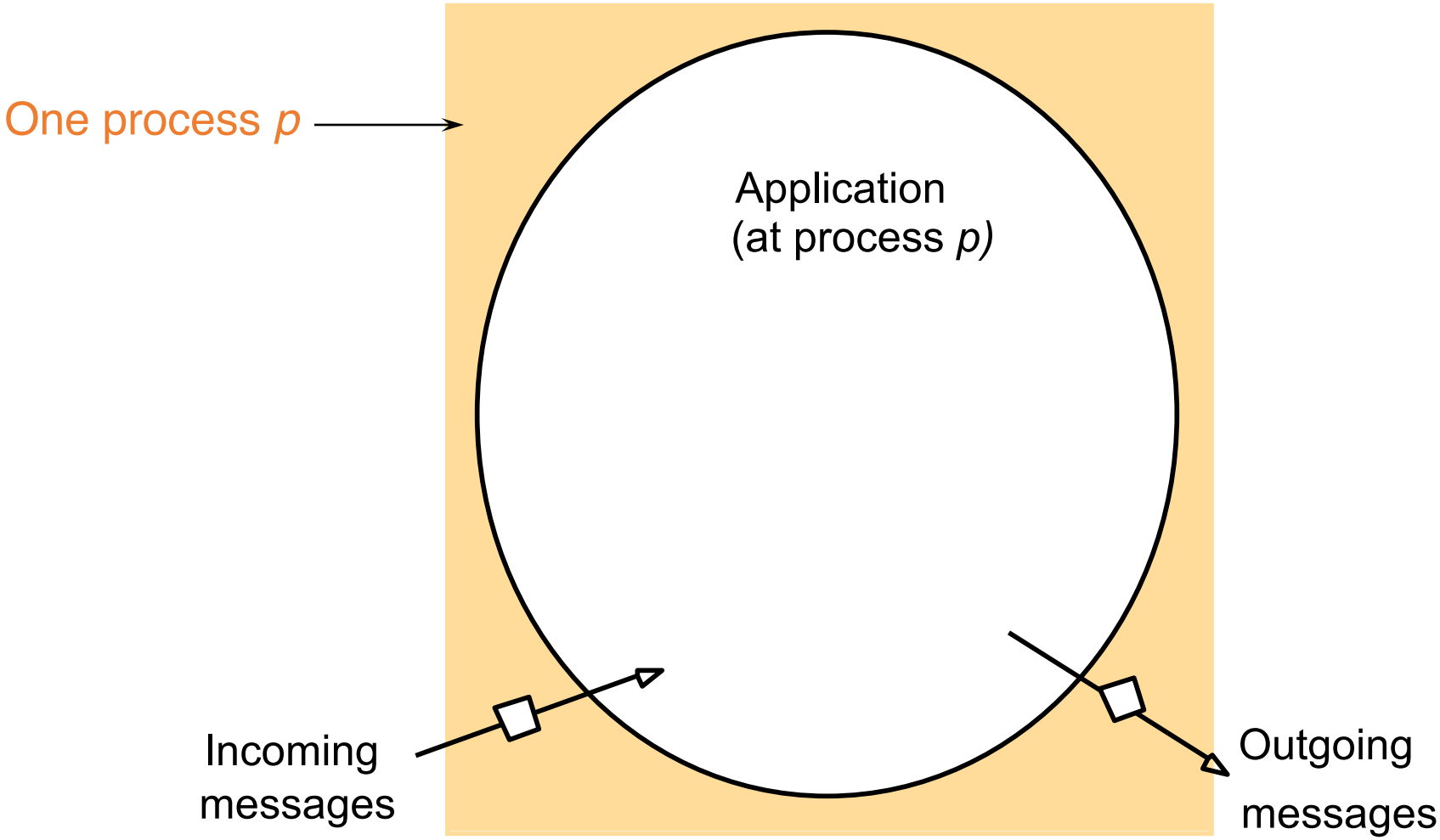
Where is multicast used?

- 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.
- ……

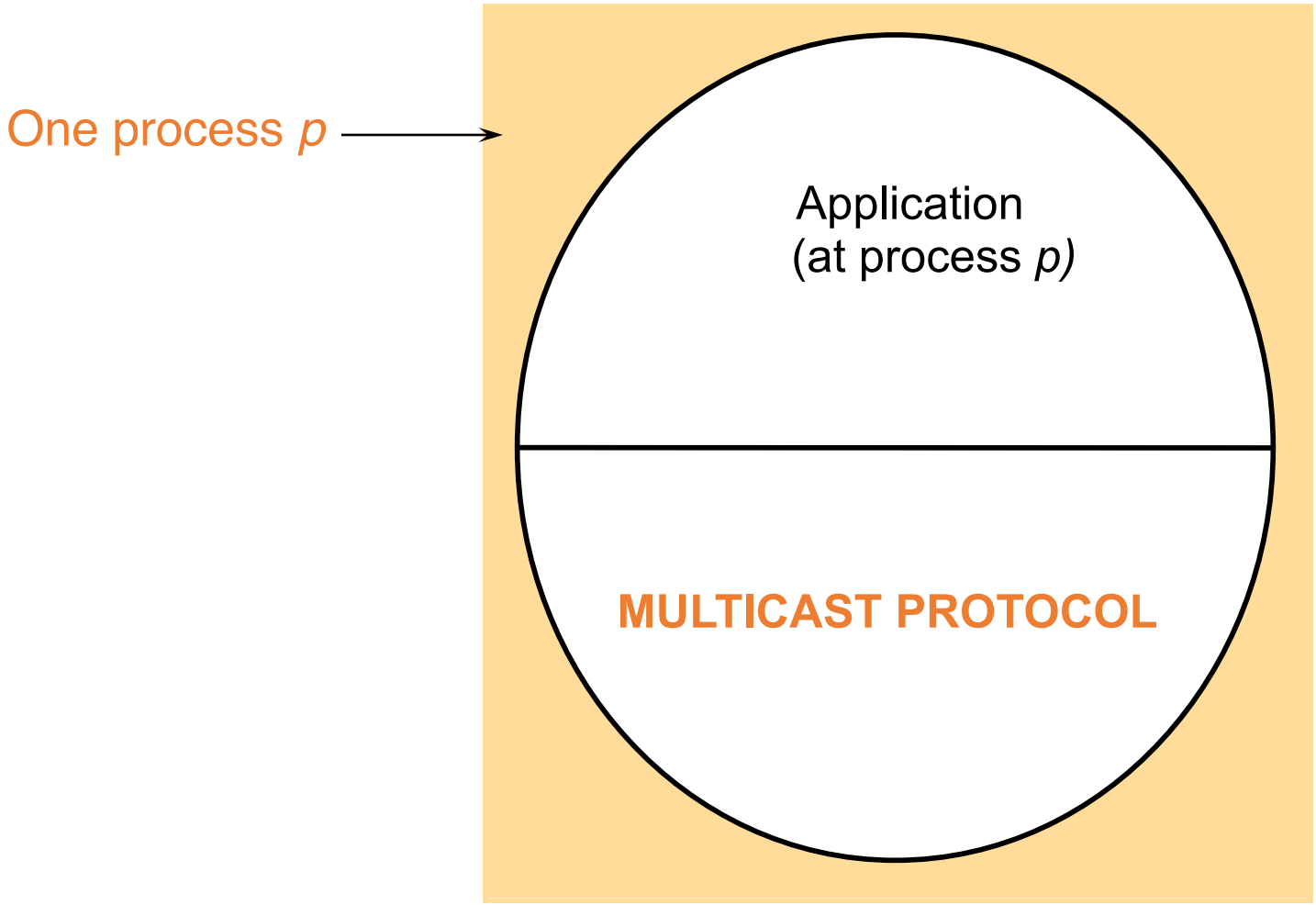
Communication modes

- **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?*

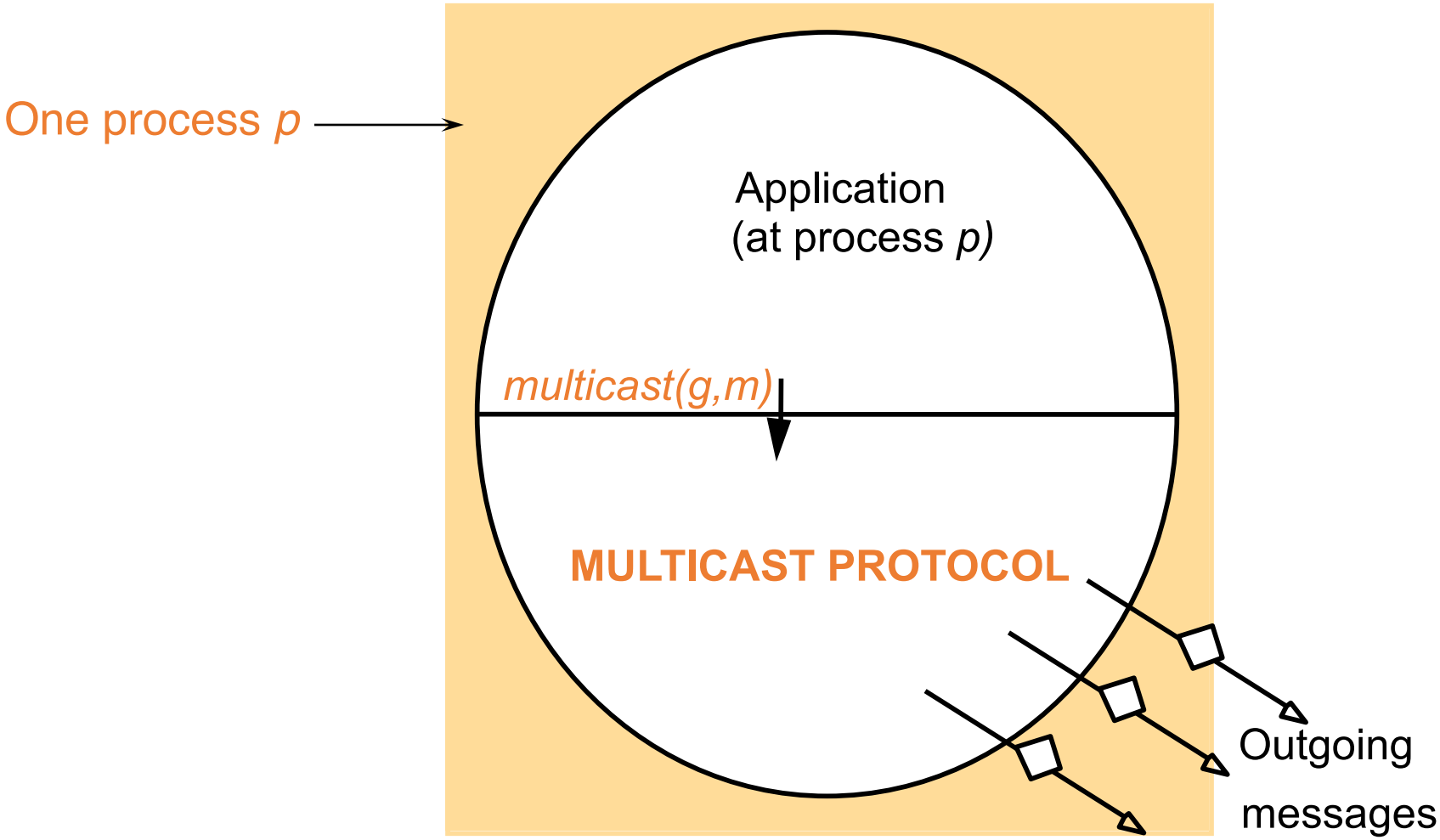
What we are designing in this class?



What we are designing in this class?

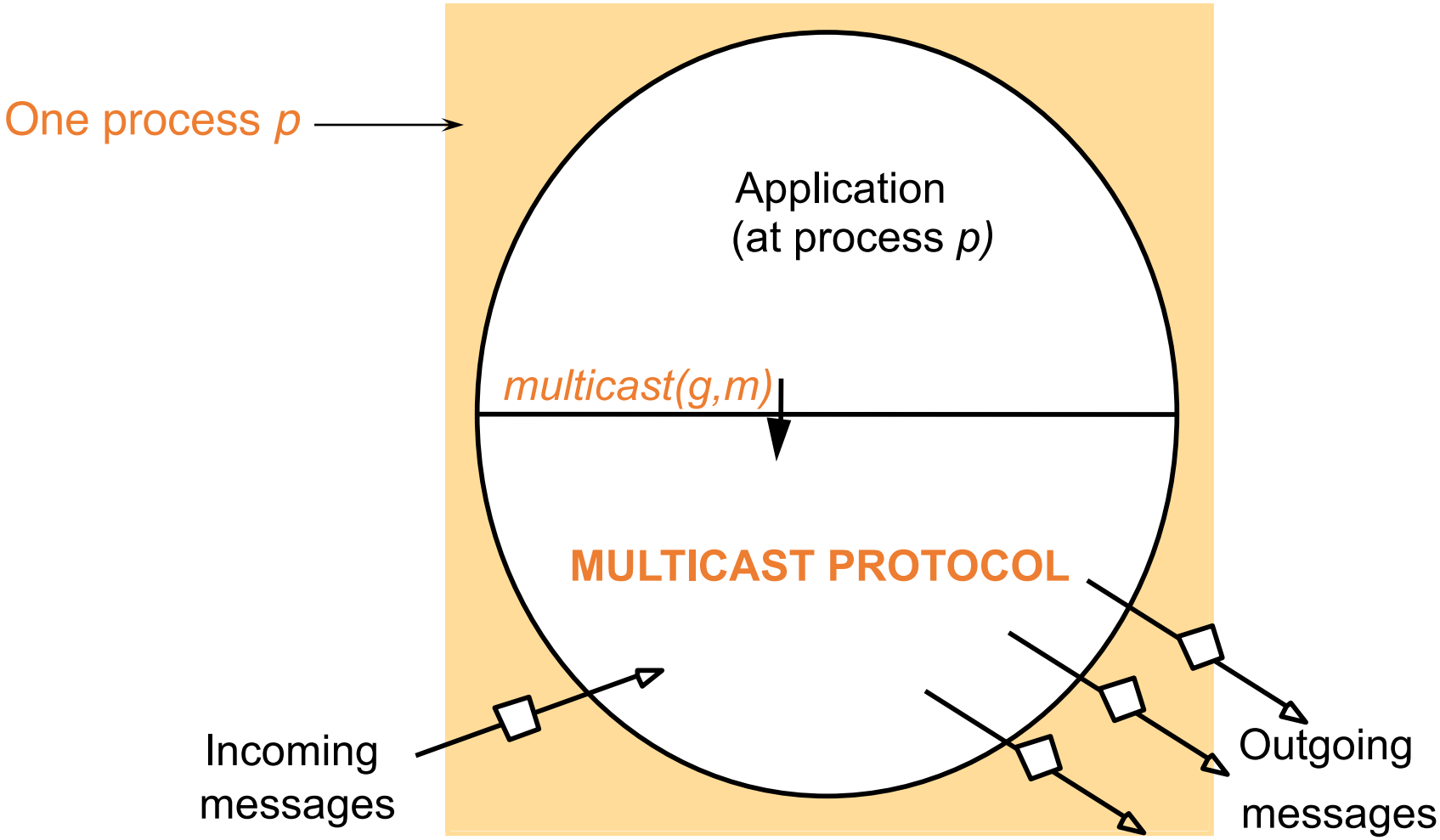


What we are designing in this class?



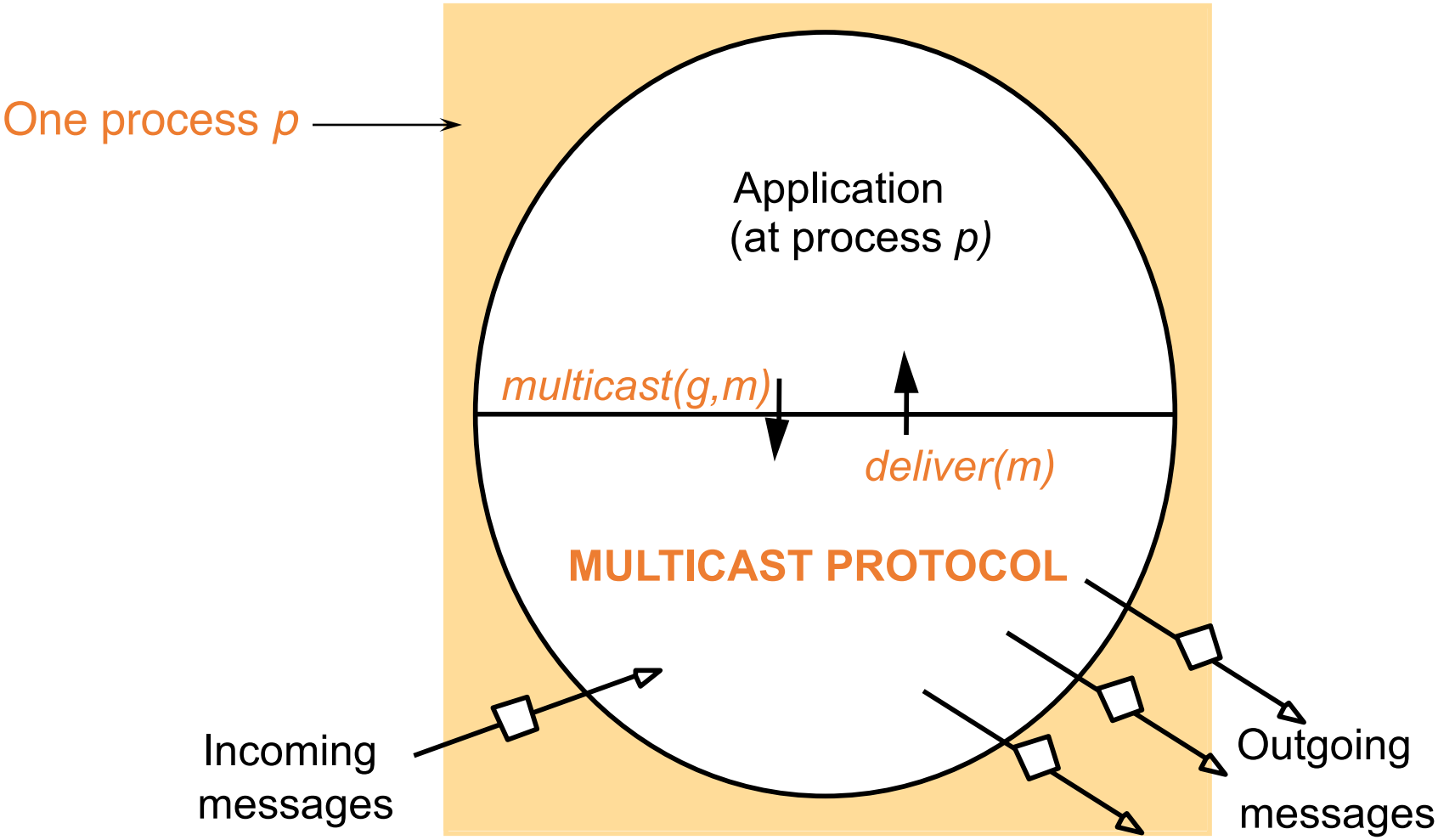
'g' is a multicast group that also includes the process 'p'.

What we are designing in this class?



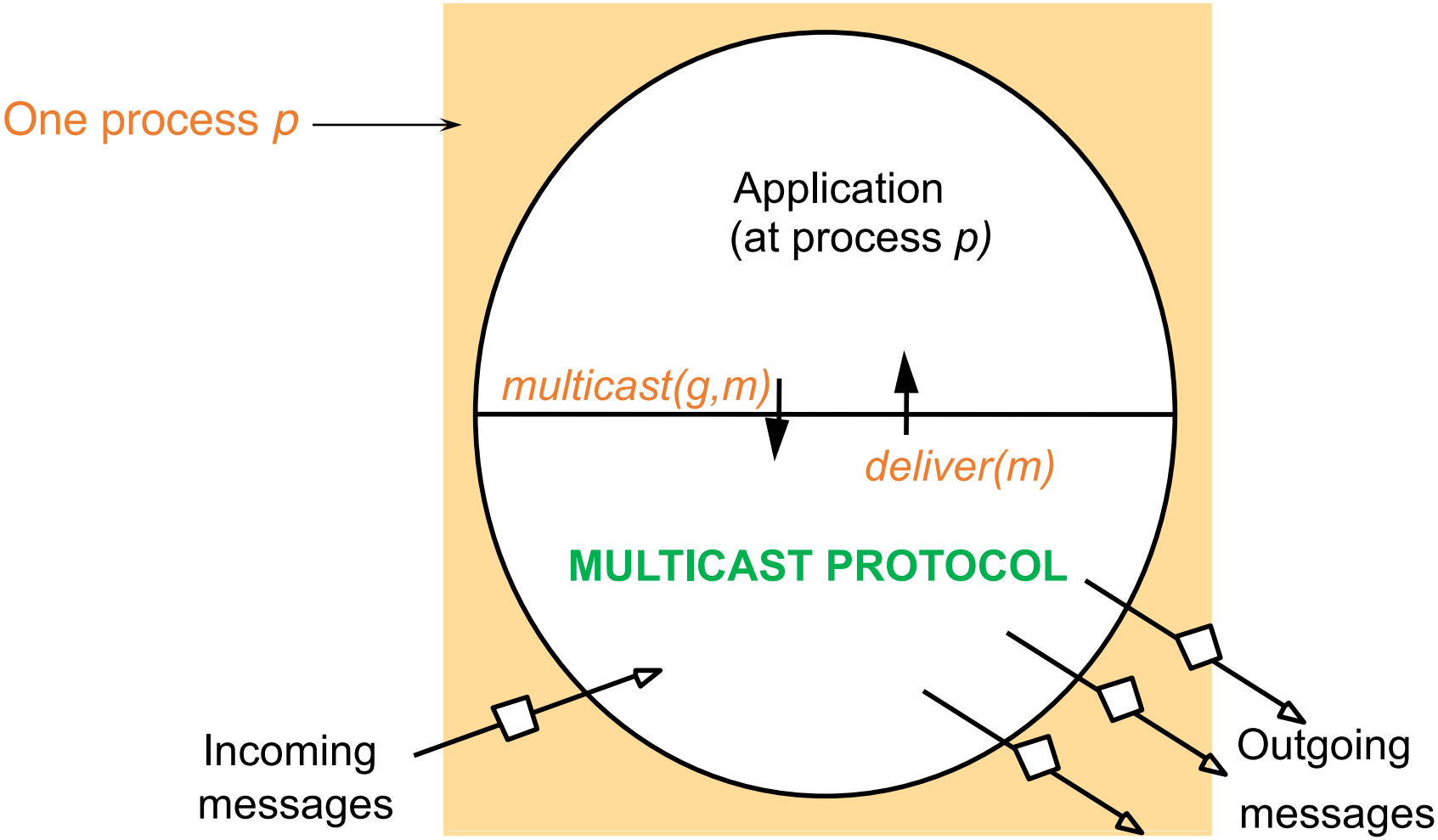
'g' is a multicast group that also includes the process 'p'.

What we are designing in this class?



'g' is a multicast group that also includes the process 'p'.

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* to 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.

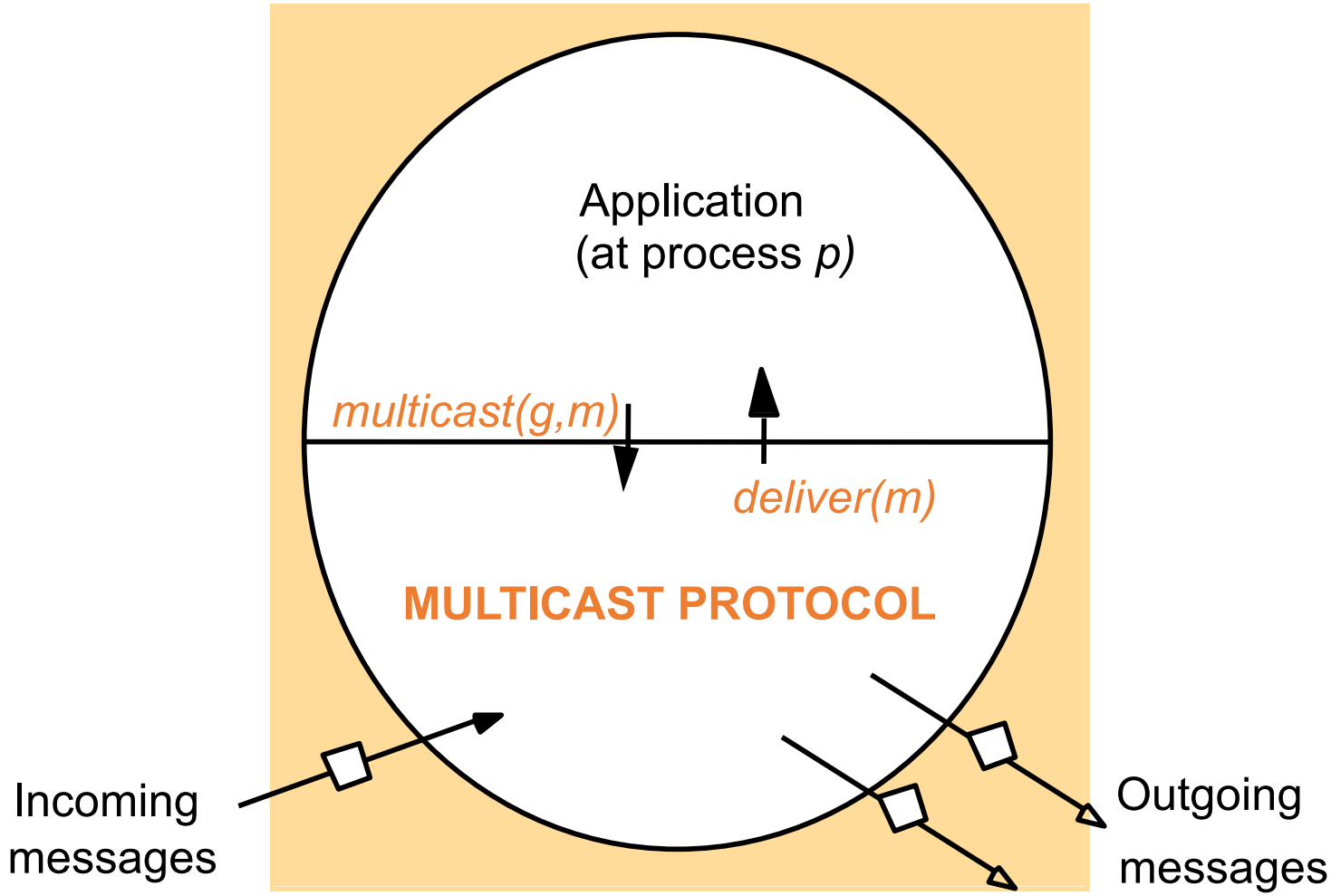
Reliable Multicast (R-Multicast)

- **Integrity**: A *correct* (i.e., non-faulty) process *p* delivers a message *m* at most onc
  - *Assur* *twice*
- **Validity**: I *hen it will* eventuall
  - *Livene*
- **Agreeme** *the other* *correct* pr
  - *All or*
- Validity and agreement together ensure overall liveness: if some correct process multicasts a message *m*, then, all correct processes deliver *m* too.

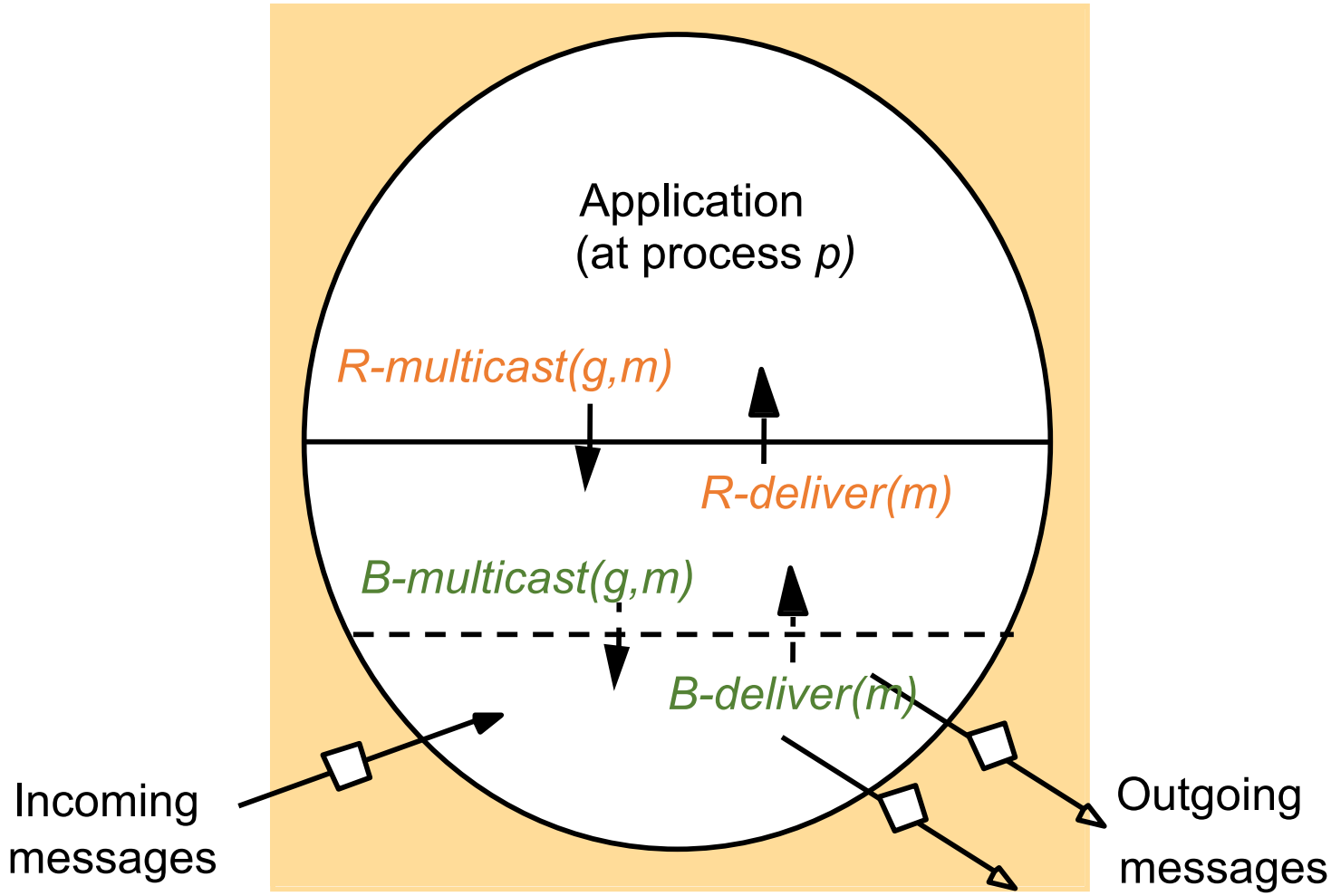
What happens if a process initiates B-multicasts of a message but fails after unicasting to a subset of processes in the group?

Agreement is violated! R-multicast not satisfied.

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 ∈ g is included as destination)

On B-deliver(m) at process q in g = group(m)

    if (m ∉ Received):

        Received := Received ∪ {m};

        if (q ≠ 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* to 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

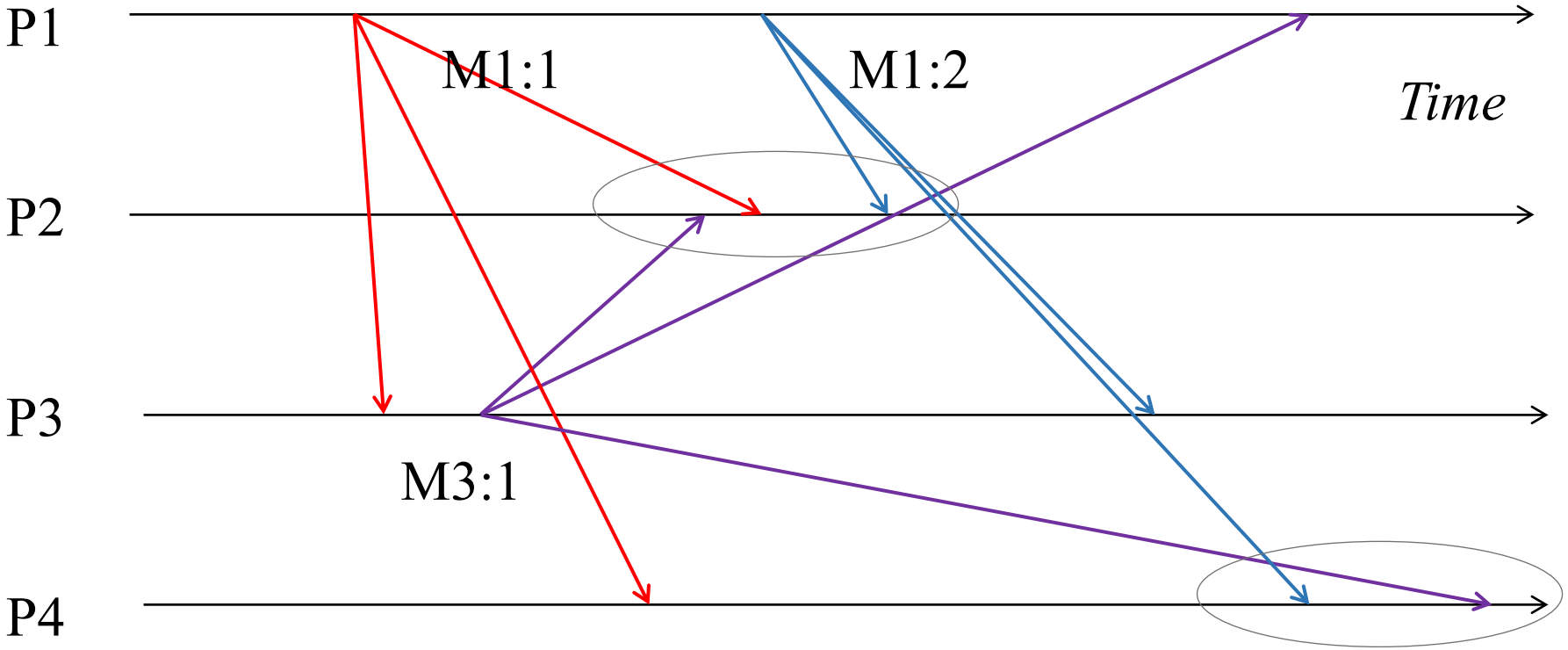
- Three popular flavors implemented by several multicast protocols:
  1. FIFO ordering
  2. Causal ordering
  3. Total ordering

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*
  - → is induced only by multicast messages in group g, and when they are **delivered** to the application, rather than all network messages.

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.

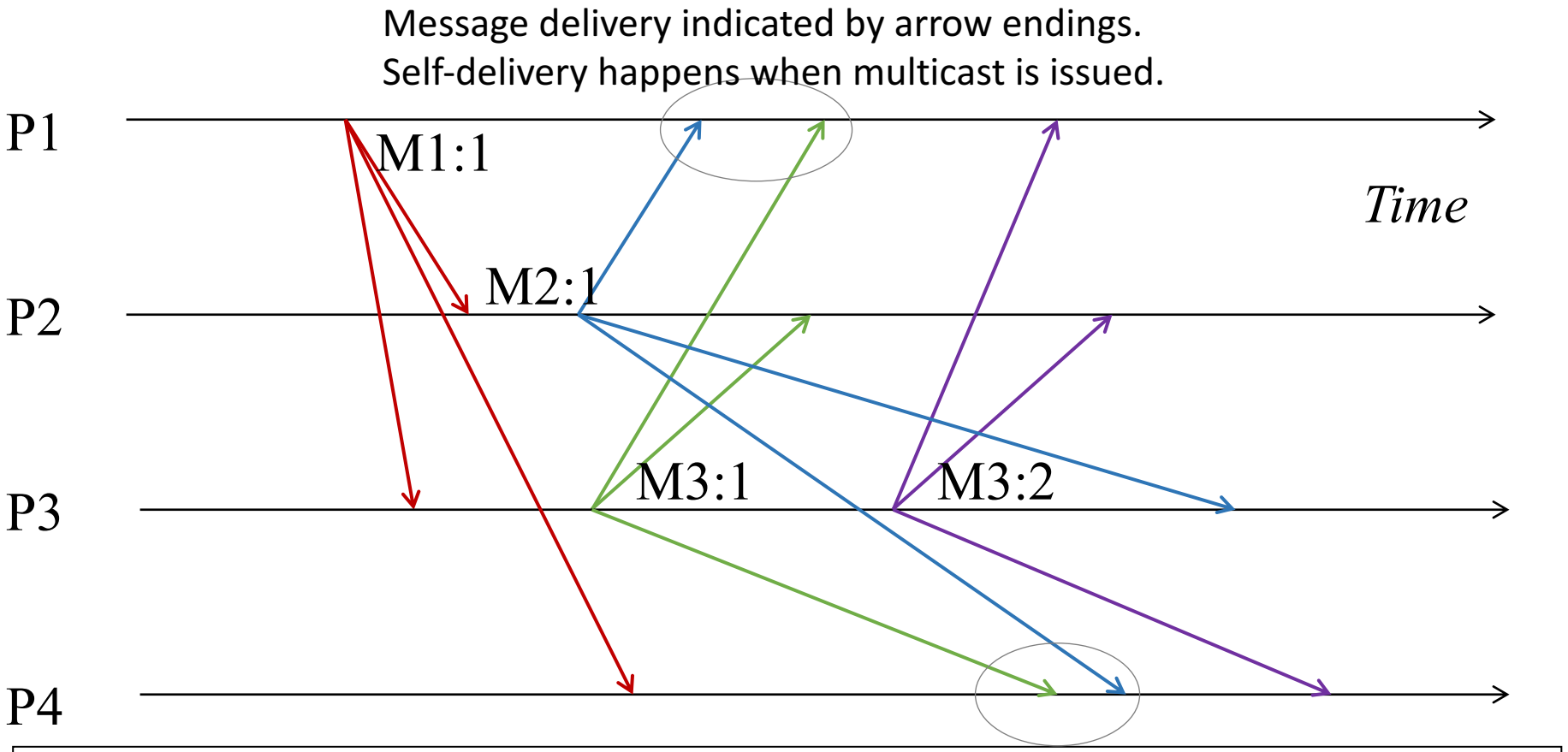
HB Relationship for Causal Ordering

- HB rules in causal ordered multicast:
  - If ∃ p_i , **e** $\rightarrow_i$ **e'** then **e** → **e'**.
    - If ∃ p_i , multicast(*g*,*m*) $\rightarrow_i$ multicast(*g*,*m'*), then multicast(*g*,*m*) → multicast(*g*,*m'*)
    - If ∃ p_i , delivery(*m*) $\rightarrow_i$ multicast(*g*,*m'*), then delivery(*m*) → multicast(*g*,*m'*)
    - …
  - For any message m, **send(m) → receive(m)**

HB Relationship for Causal Ordering

- HB rules in causal ordered multicast:
  - If ∃ p_i , **e** $\rightarrow_i$ **e'** then **e** $\rightarrow$ **e'**.
    - If ∃ p_i , multicast(*g*,*m*) $\rightarrow_i$ multicast(*g*,*m'*), then multicast(*g*,*m*) $\rightarrow$ multicast(*g*,*m'*)
    - If ∃ p_i , delivery(*m*) $\rightarrow_i$ multicast(*g*,*m'*), then delivery(*m*) $\rightarrow$ multicast(*g*,*m'*)
    - …
  - ~~For any message m, **send(m) → receive(m)**~~
    - For any *multicast* message m, **multicast(*g*,*m*) → delivery(m)**
  - If **e** → **e'** and **e'** → **e"** then **e** → **e"**
    - multicast(*g*,*m*) at p_i → delivery(m) at p_j
    - delivery(*m*) at p_j → multicast(*g*,*m'*) at p_j
    - multicast(*g*,*m*) at p_i → multicast(*g*,*m'*) at p_j
- *Application can only see when messages are "multicast" by the application and "delivered" to the application, and not when they are sent or received by the protocol.*

Causal Order: Example



M3:1 → M3:2, M1:1 → M2:1, M1:1 → M3:1 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 any (and even different) orders at different receivers.

Ordered Multicast

- Three popular flavors implemented by several multicast protocols:
  1. FIFO ordering
  2. Causal ordering
     - More examples in next class.
  3. Total ordering
     - Next class