Distributed Systems

ECE428

Lecture 6

Adopted from Spring 2021

Some revision while we wait

- For a process p_i , where events $e_i^0, e_i^1, \ldots$ occur:

  history(p_i) = $h_i = \langle e_i^0, e_i^1, \ldots \rangle$

  prefix history(p_i^k) = $h_i^k = \langle e_i^0, e_i^1, \ldots, e_i^k \rangle$

  s_i^k : p_i's state immediately after k^{th} event.

- For a set of processes $\langle p_1, p_2, p_3, \ldots, p_n \rangle$:

  global history: $H = \cup_i (h_i)$

  a cut $C \subseteq H = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_n}$

  the frontier of $C = \{e_i^{c_i}, i = 1,2, \ldots n\}$

  global state S that corresponds to cut $C = \cup_i (s_i^{c_i})$

- A cut C is consistent if and only if $\forall e \in C$ (if $f \rightarrow e$ then $f \in C$)
  - A global state S is consistent if and only if it corresponds to a consistent cut.

Today's agenda

- Global State
  - Chapter 14.5
  - Goal: reason about how to capture the state across all processes of a distributed system without requiring time synchronization.

Recap: How to capture global state?

- State of each process (and each channel) in the system *at a given instant of time.*
  - Difficult to capture -- requires precisely synchronized time.

- Relax the problem
  - For a system with n processes $<p_1, p_2, p_3, \ldots., p_n>$, capture the state of the system after the c_i–th event at process p_i.
    - State corresponding to the cut defined by frontier events $\{e_i^{c_i}$, for i = 1,2, … n$\}$.
  - We want the state to be consistent.
    - Must correspond to a consistent cut.
      - If event e belongs to the cut, all events that "happened before" e must also belong to the cut.

Recap: Chandy-Lamport Algorithm

- *Goal: Record consistent state by identifying a consistent cut.*
- *System model and assumptions:*
  - System of n processes: <p_1, p_2, p_3, …., p_n>.
  - There are two uni-directional channels between each process pair : p_j to p_i and p_i to p_j. Channels are FIFO.
  - All messages arrive intact, and are not duplicated.
  - No failures: neither channel nor processes fail.
- *Requirements:*
  - Snapshot should not interfere with normal application actions, nor require application to stop sending messages.
  - Any process can require global snapshot i.e. initiate the algorithm.

Chandy-Lamport Algorithm Intuition

- First, initiator p_i:
  - records its own state.
  - creates a special marker message.
  - sends the marker to all other process.
  - start recording messages received on other channels.
    - until a marker is received on a channel.
- When a process receives a marker.
  - If marker is received for the first time.
    - records its own state.
    - sends marker on all other channels.
    - start recording messages received on other channels.
      - until a marker is received on a channel.

Chandy-Lamport Algorithm

- First, initiator p_i:
  - records its own state.
  - creates a special marker message.
  - for *j=1 to n* except *i*
    - p_i sends a marker message on outgoing channel c_{ij}
    - starts recording the incoming messages on each of the incoming channels at p_i : c_{ji} (for *j=1 to n* except *i*).

Chandy-Lamport Algorithm

Whenever a process p_i receives a marker message from p_k on incoming channel c_{ki}

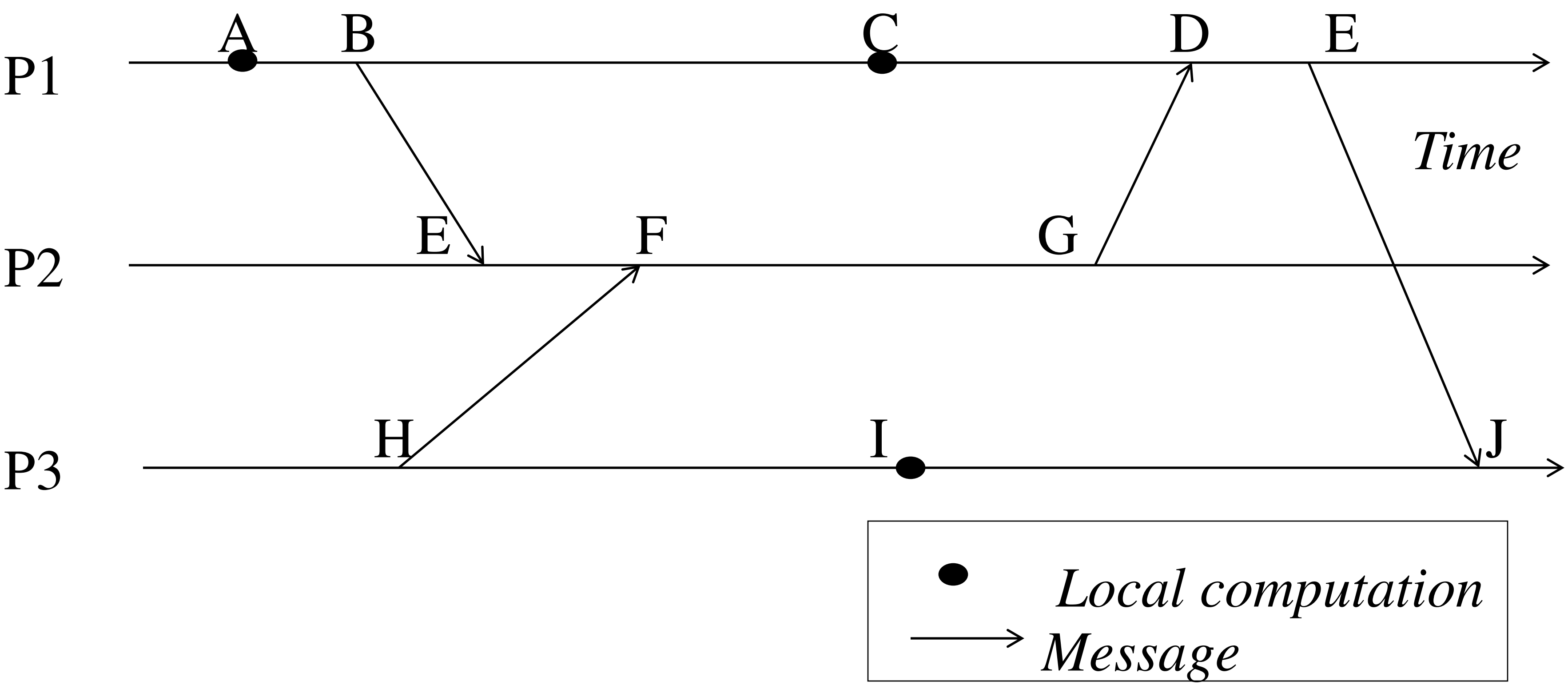
- **if** this is the first marker p_i is seeing, **then**
  - p_i records its own state first
  - marks the state of channel c_{ki} as "empty"
  - for j=1 to n except i
    - p_i sends out a marker message on outgoing channel c_{ij}
  - starts recording the incoming messages on each of the incoming channels at p_i : c_{ji} (for *j=1 to n* except *i and k*).
- **else** // already seen a marker message
  - mark the state of channel c_{ki} as all the messages that have arrived on it since recording was turned on for c_{ki}

Chandy-Lamport Algorithm

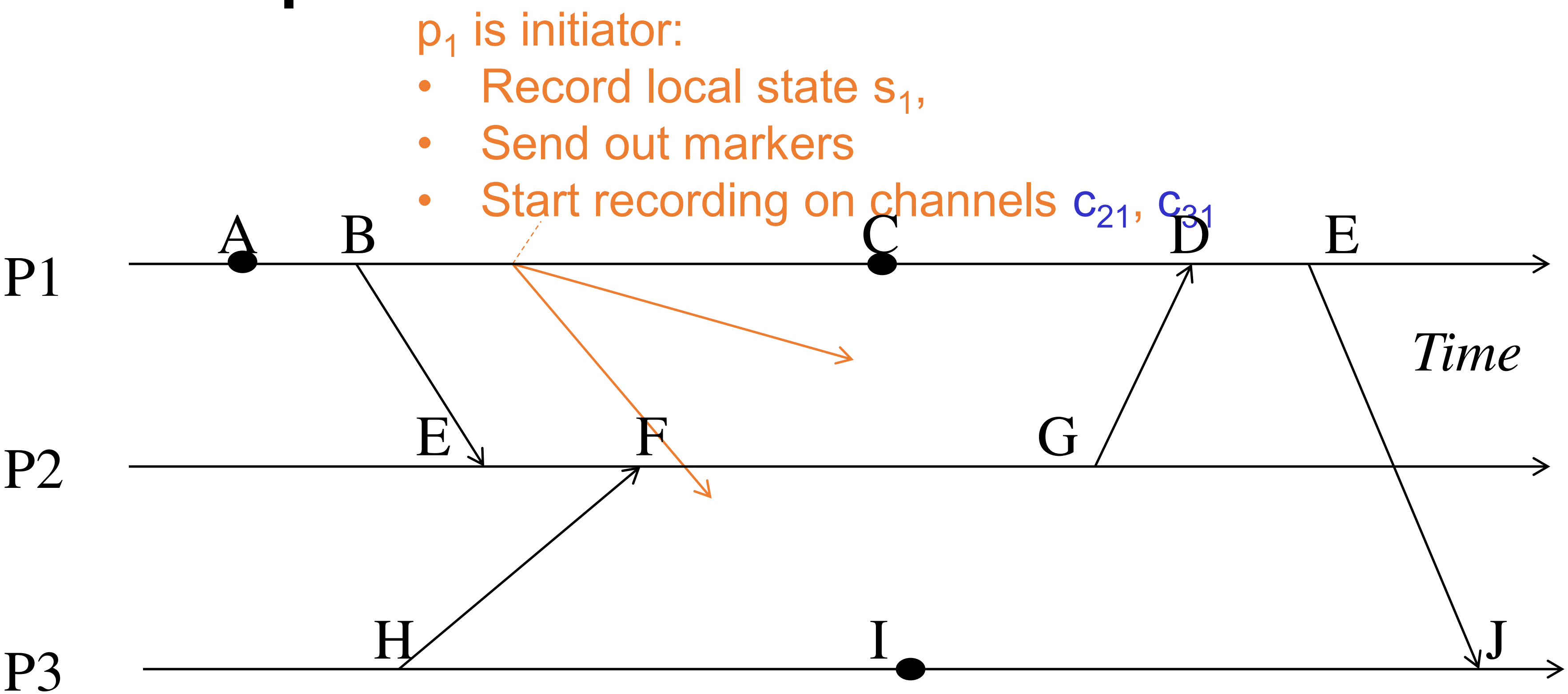
The algorithm terminates when

- All processes have received a marker
  - To record their own state
- All processes have received a marker on all other (*n-1*) incoming channels
  - To record the state of all channels

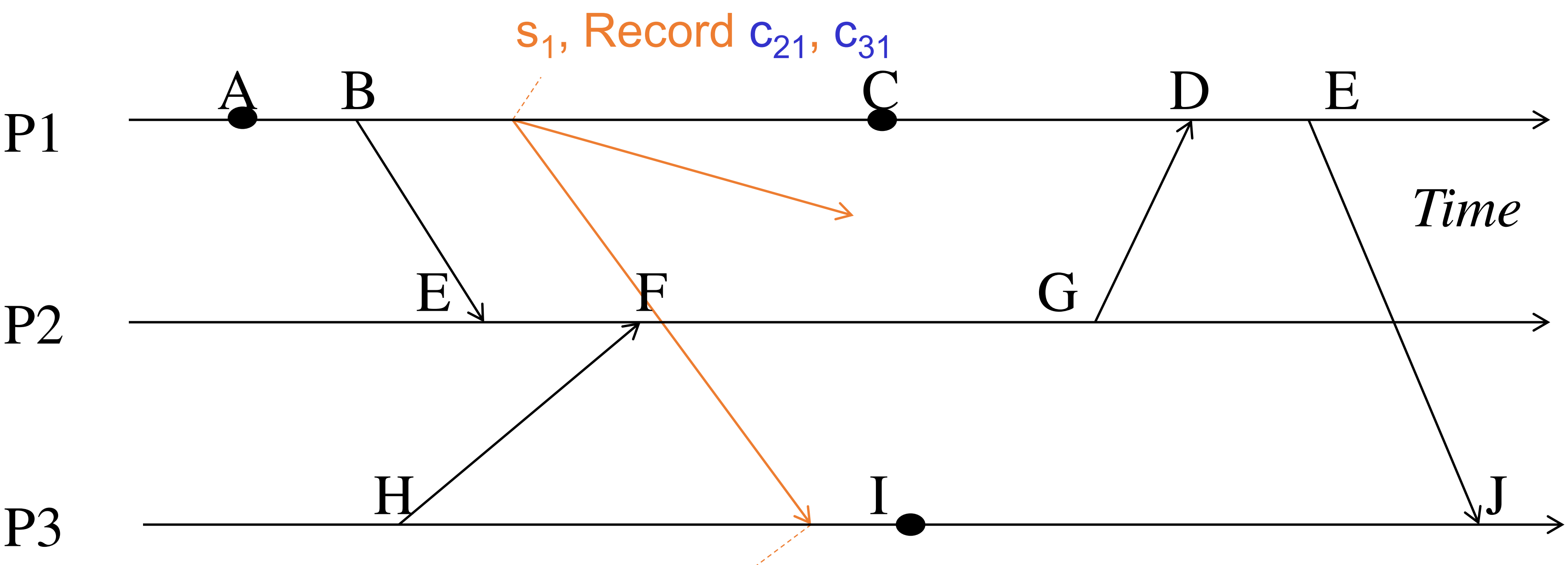
Example



Example


p$_1$ is initiator:
• Record local state s$_1$,
• Send out markers
• Start recording on channels c$_{21}$, c$_{31}$
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3

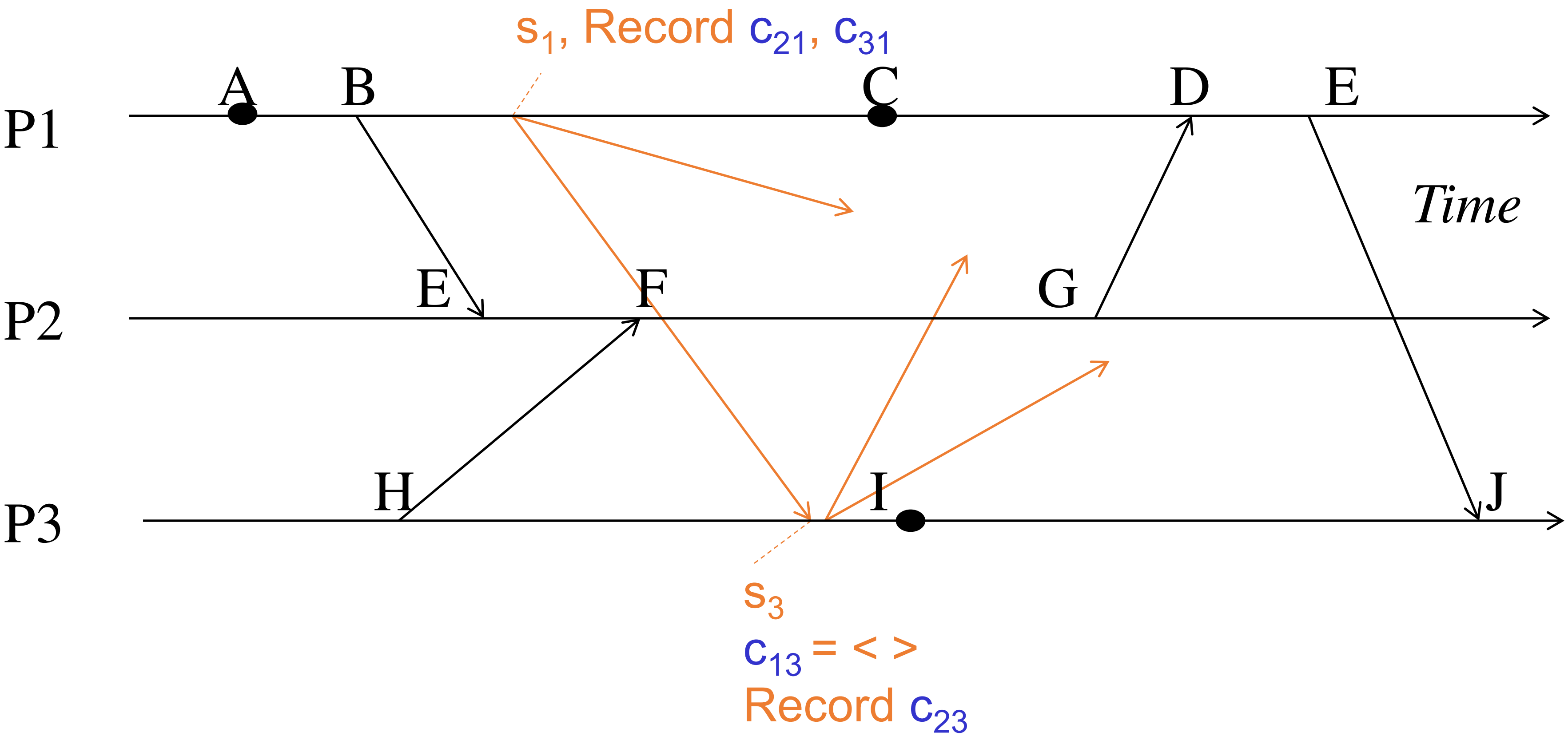
Example



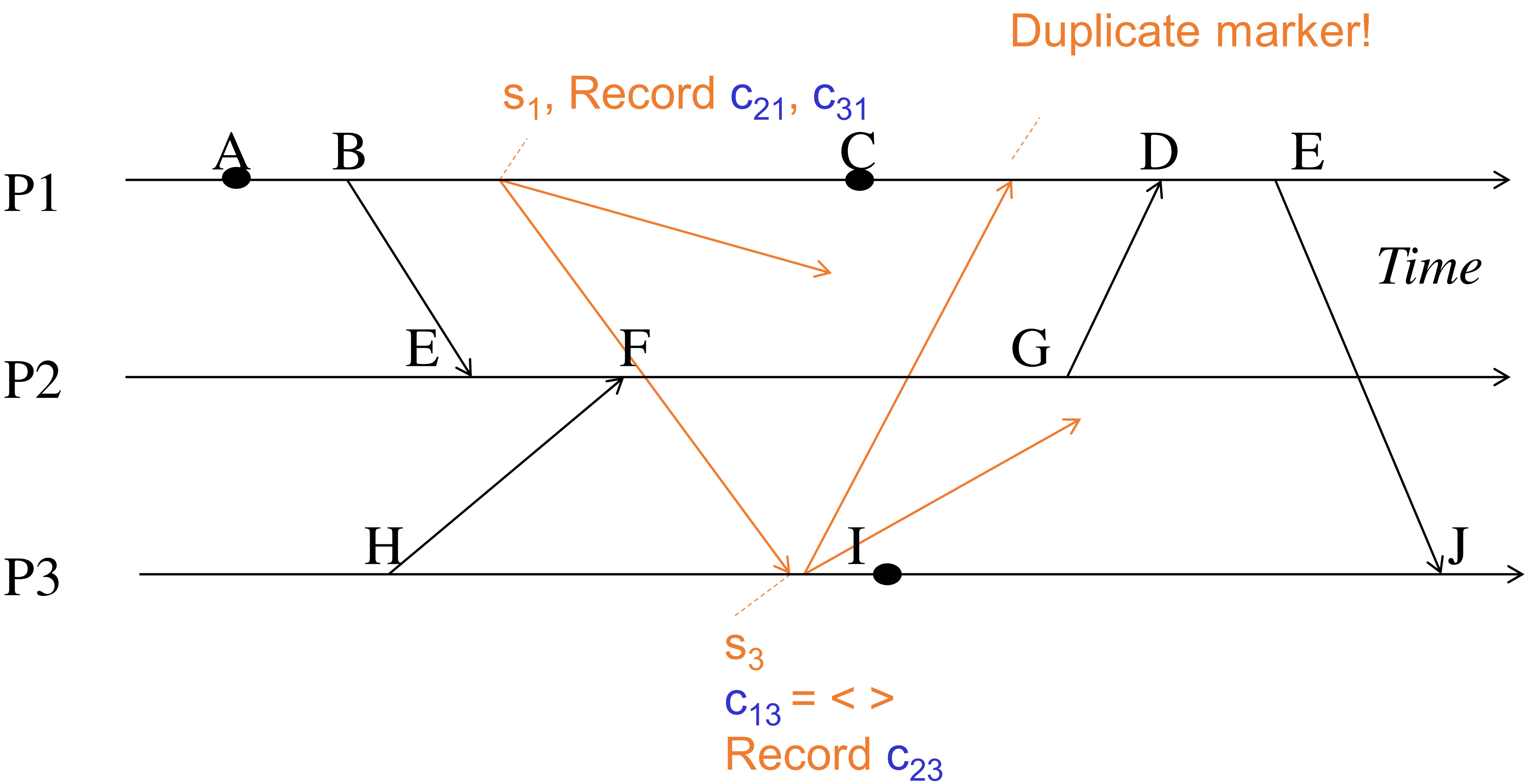
First marker!

- Record own state as s_3
- Mark c_{13} state as empty
- Start recording on other incoming c_{23}
- Send out markers

Example

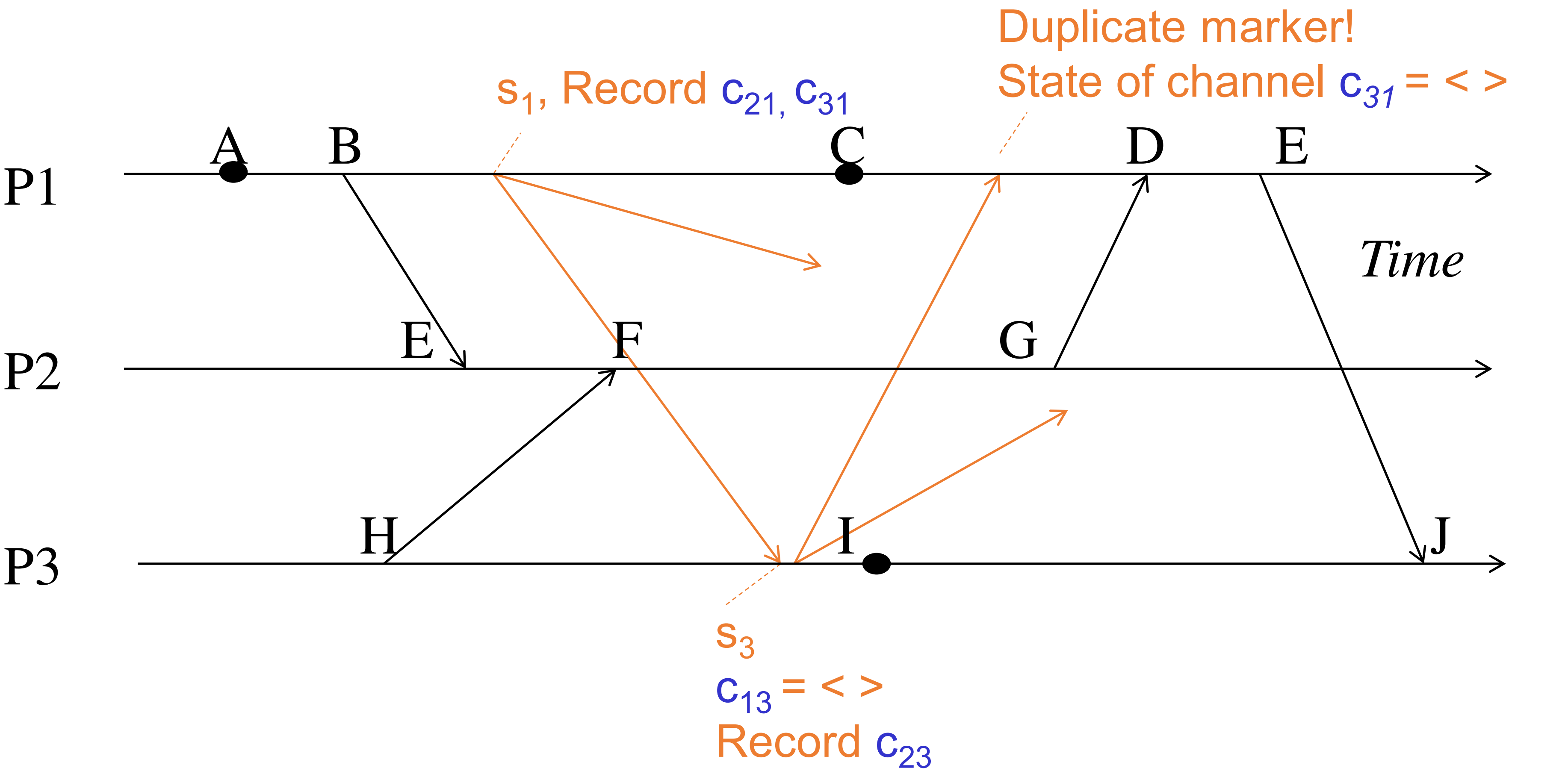


Example



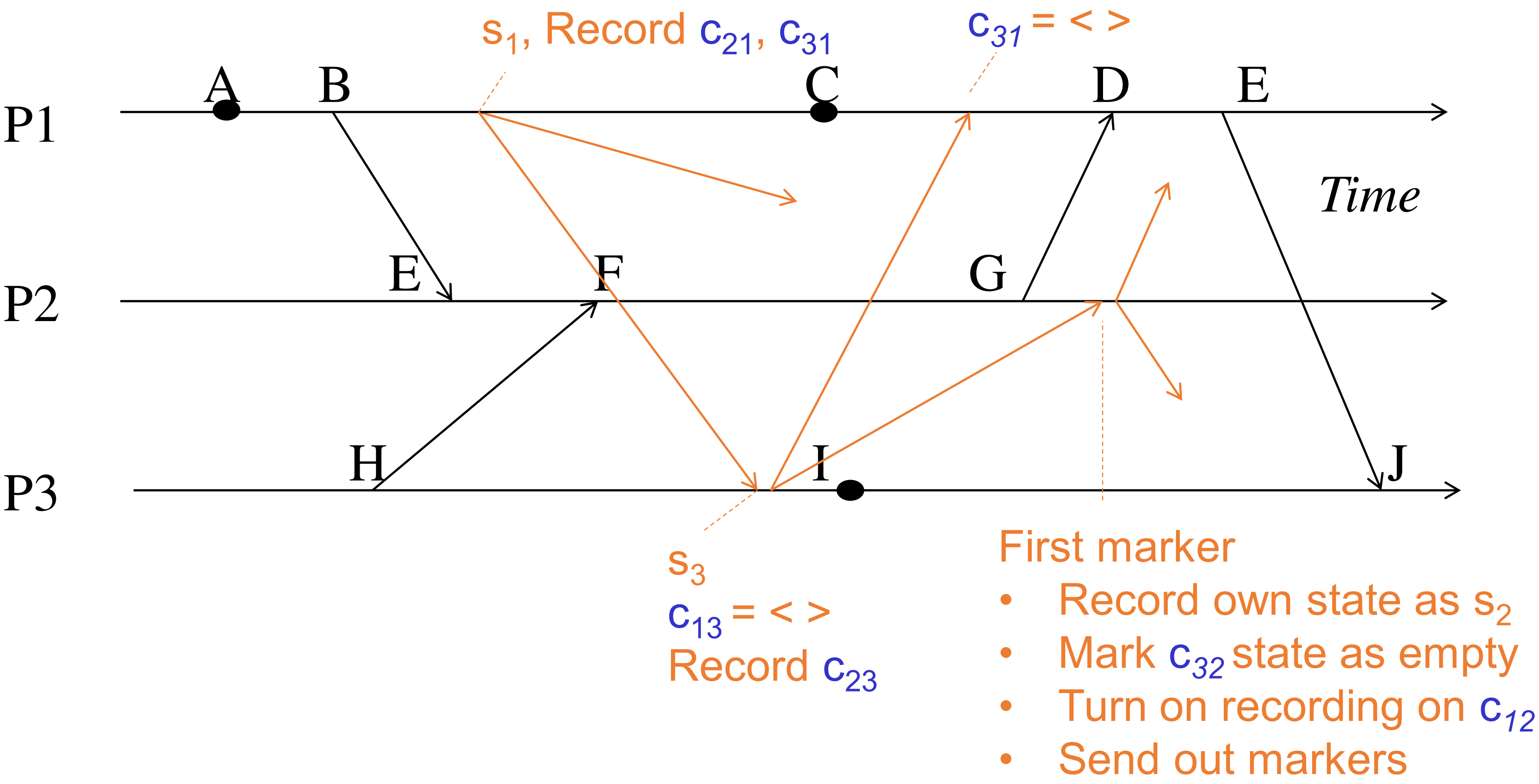
Duplicate marker!
s1, Record c21, c31
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s3
c13 = < >
Record c23

Example



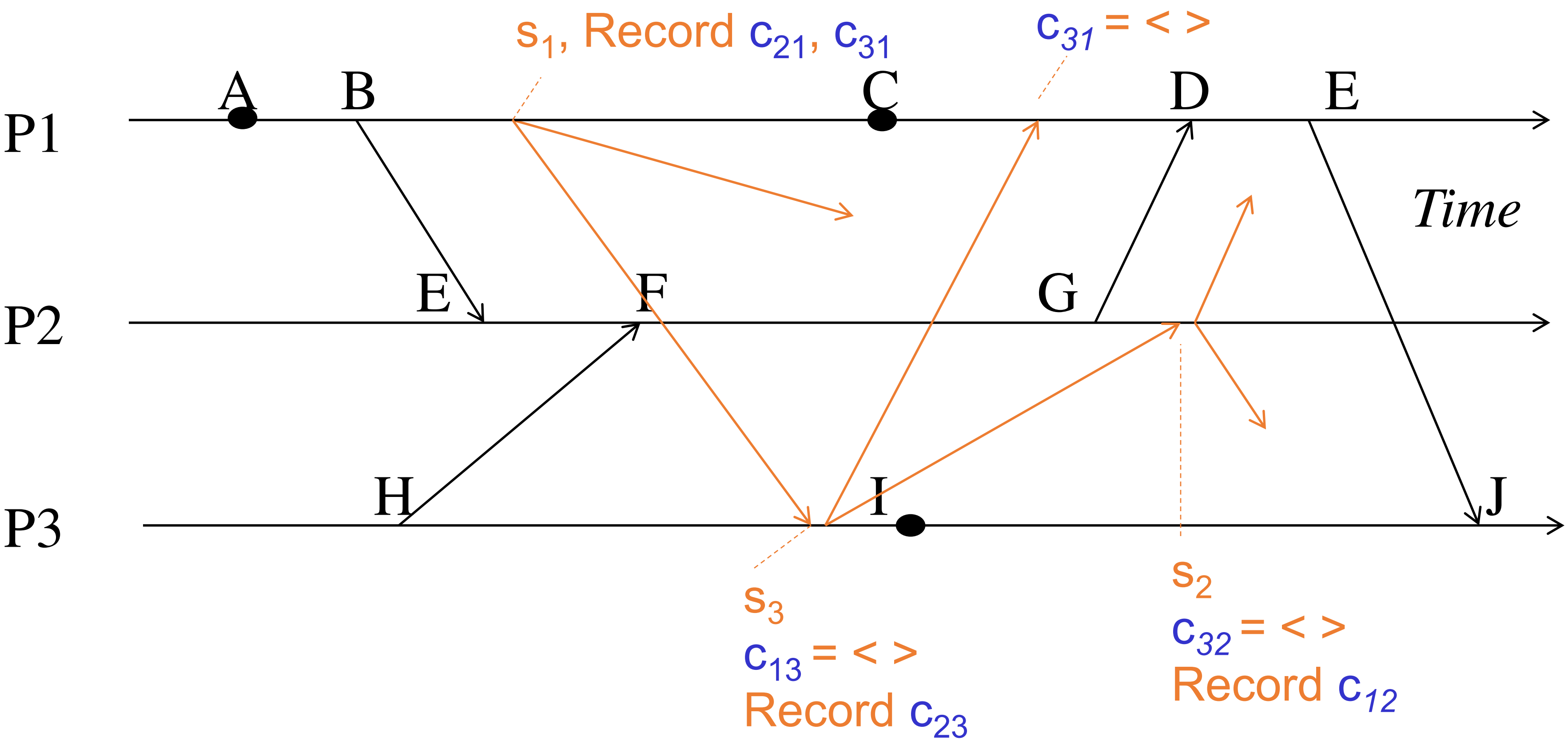
Duplicate marker!
State of channel c_{31} = < >
s_1, Record c_{21}, c_{31}
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s_3
c_{13} = < >
Record c_{23}

Example

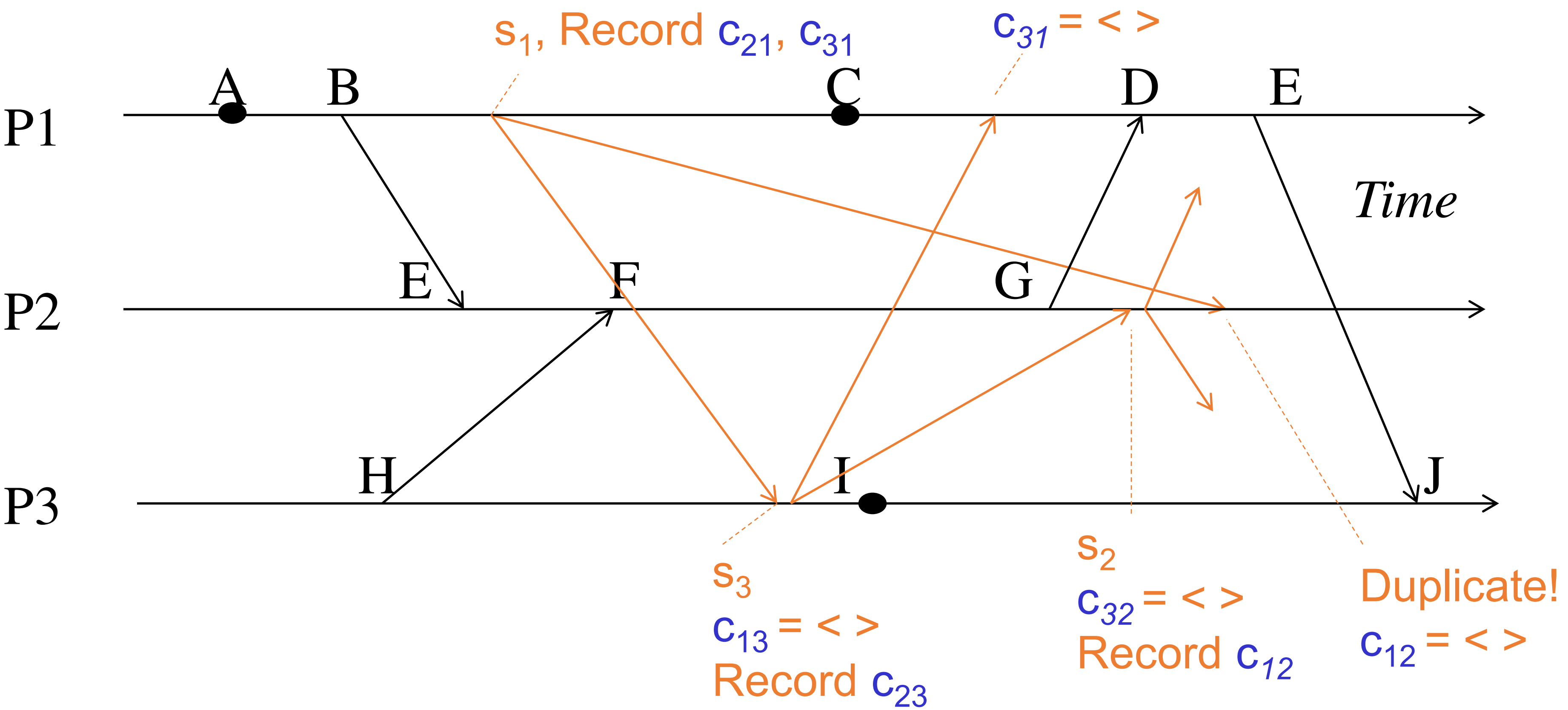


s_1, Record c_{21}, c_{31}
c_{31} = < >
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s_3
c_{13} = < >
Record c_{23}
First marker
• Record own state as s_2
• Mark c_{32} state as empty
• Turn on recording on c_{12}
• Send out markers

Example

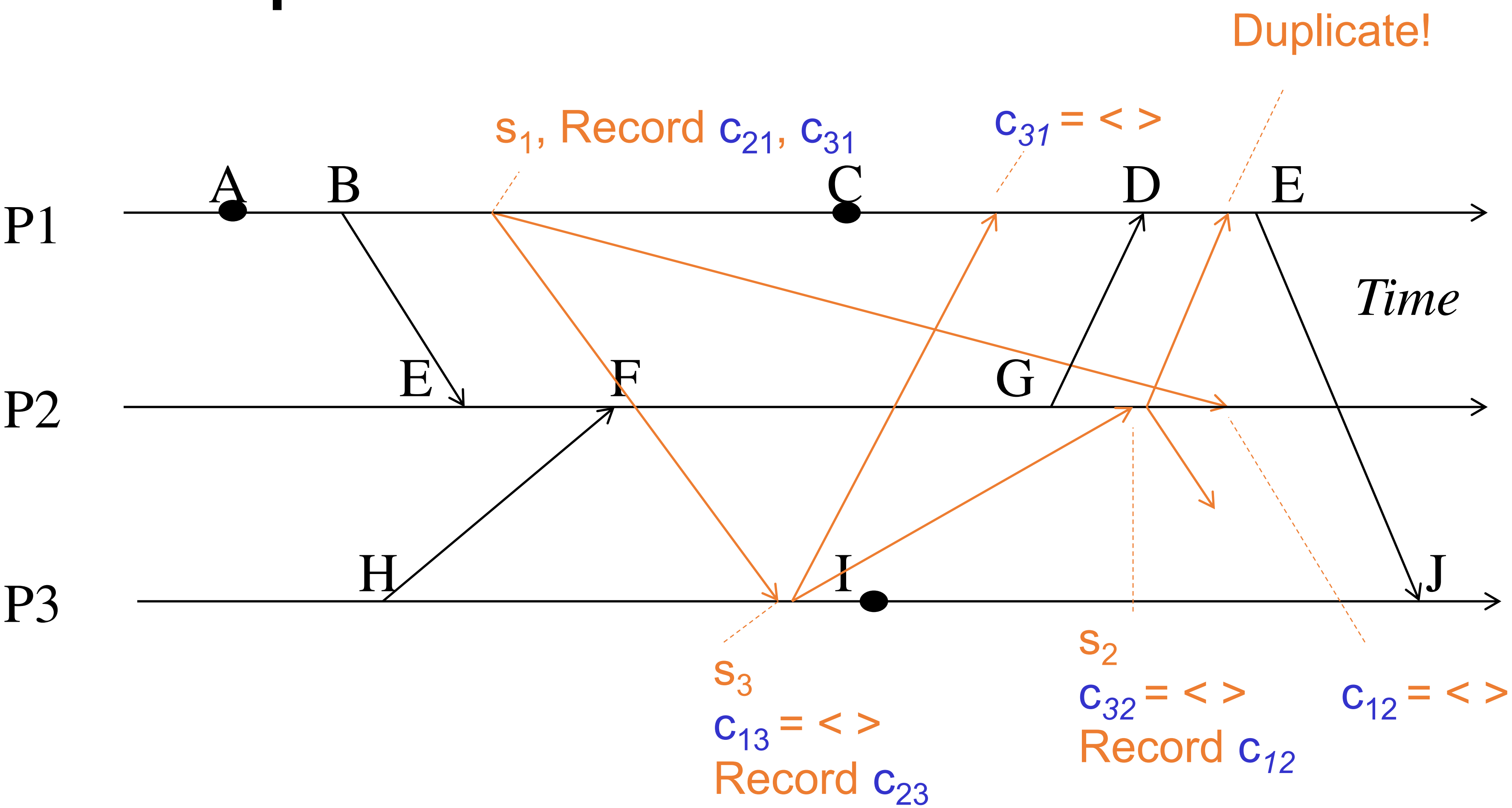

s_1, Record c_{21}, c_{31}
c_{31} = < >
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s_3
c_{13} = < >
Record c_{23}
s_2
c_{32} = < >
Record c_{12}

Example



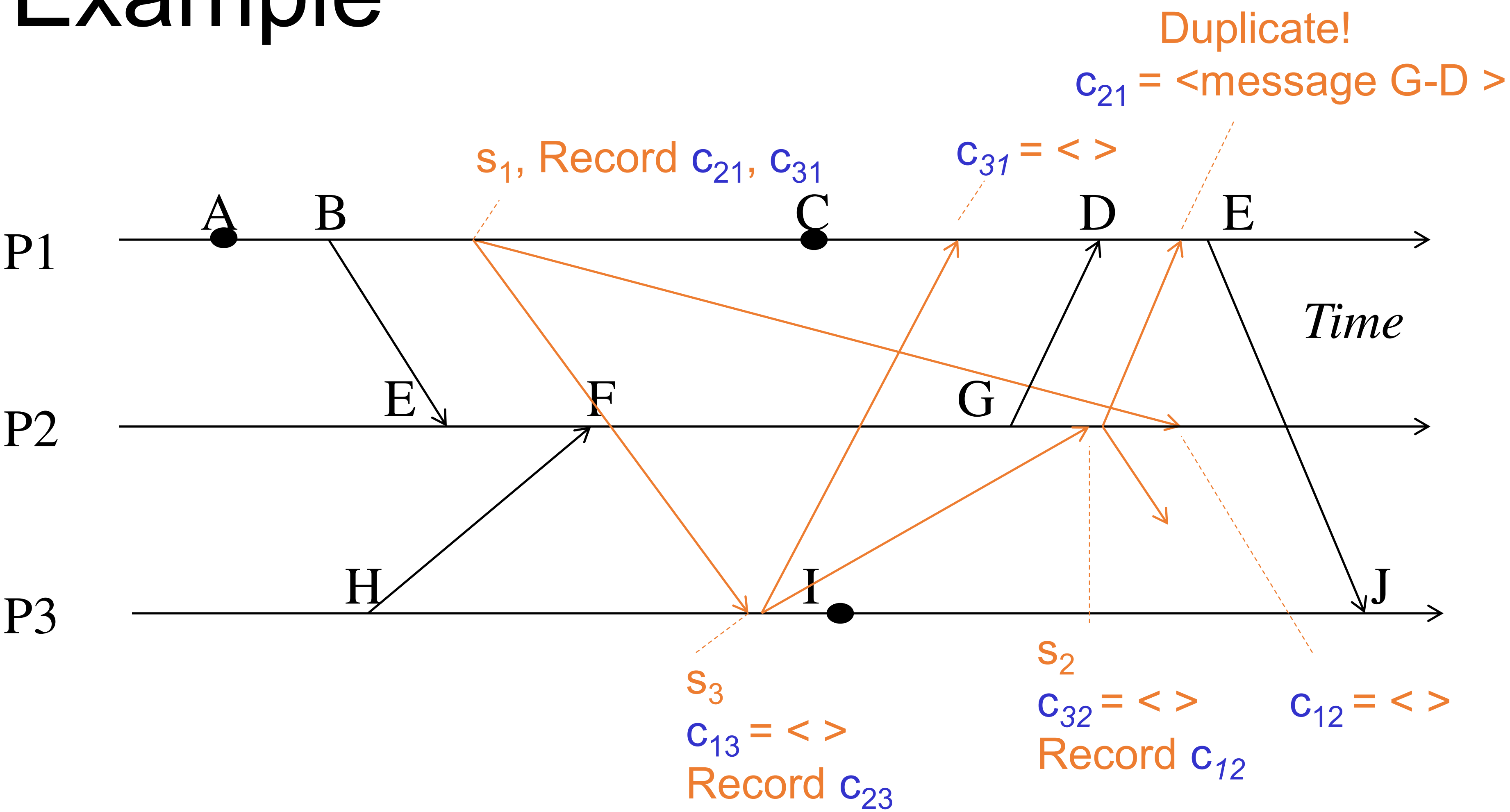
s1, Record c21, c31
c31 = < >
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s3
c13 = < >
Record c23
s2
c32 = < >
Record c12
Duplicate!
c12 = < >

Example



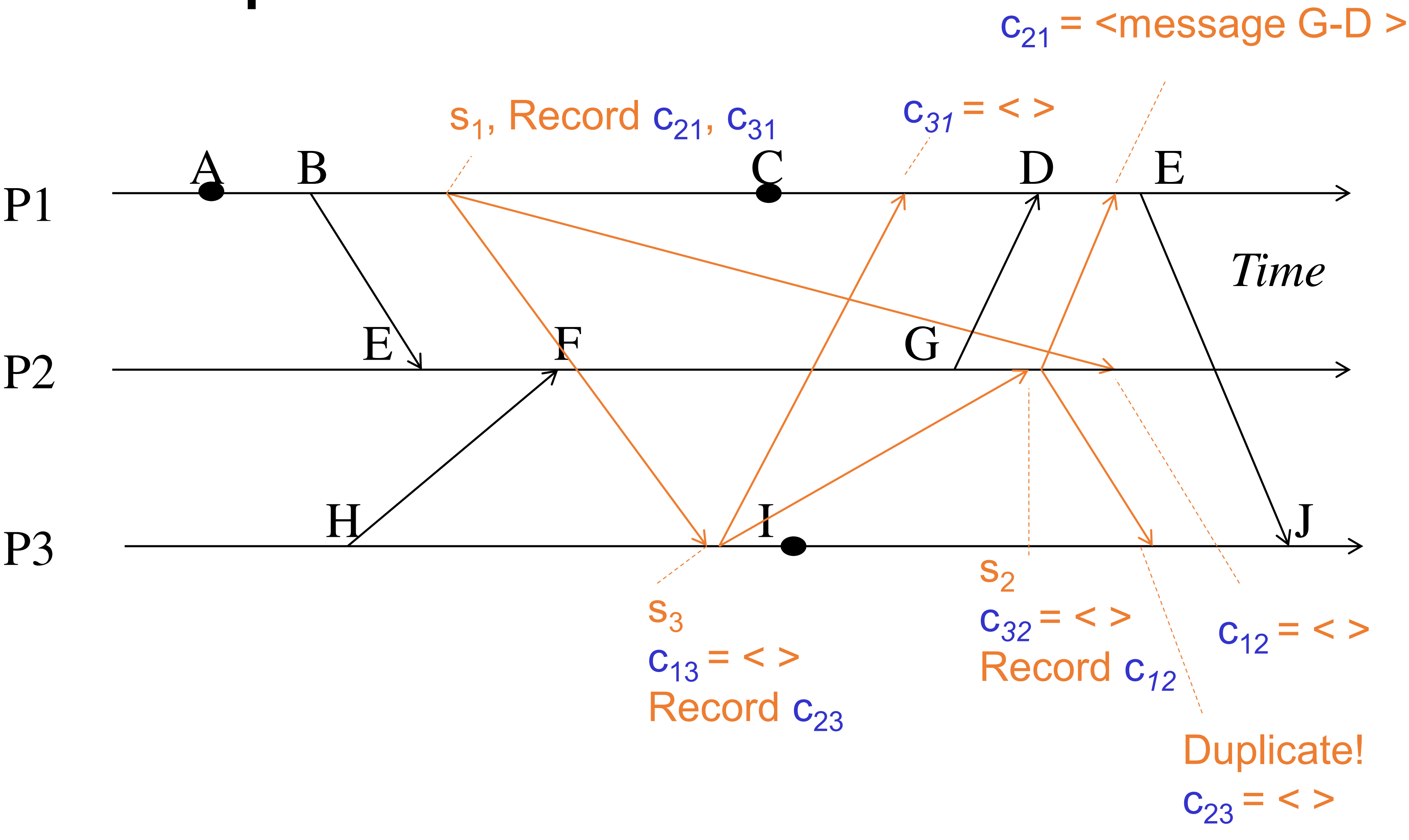
Duplicate!
s1, Record c21, c31
c31 = < >
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s3
c13 = < >
Record c23
s2
c32 = < >
Record c12
c12 = < >

Example



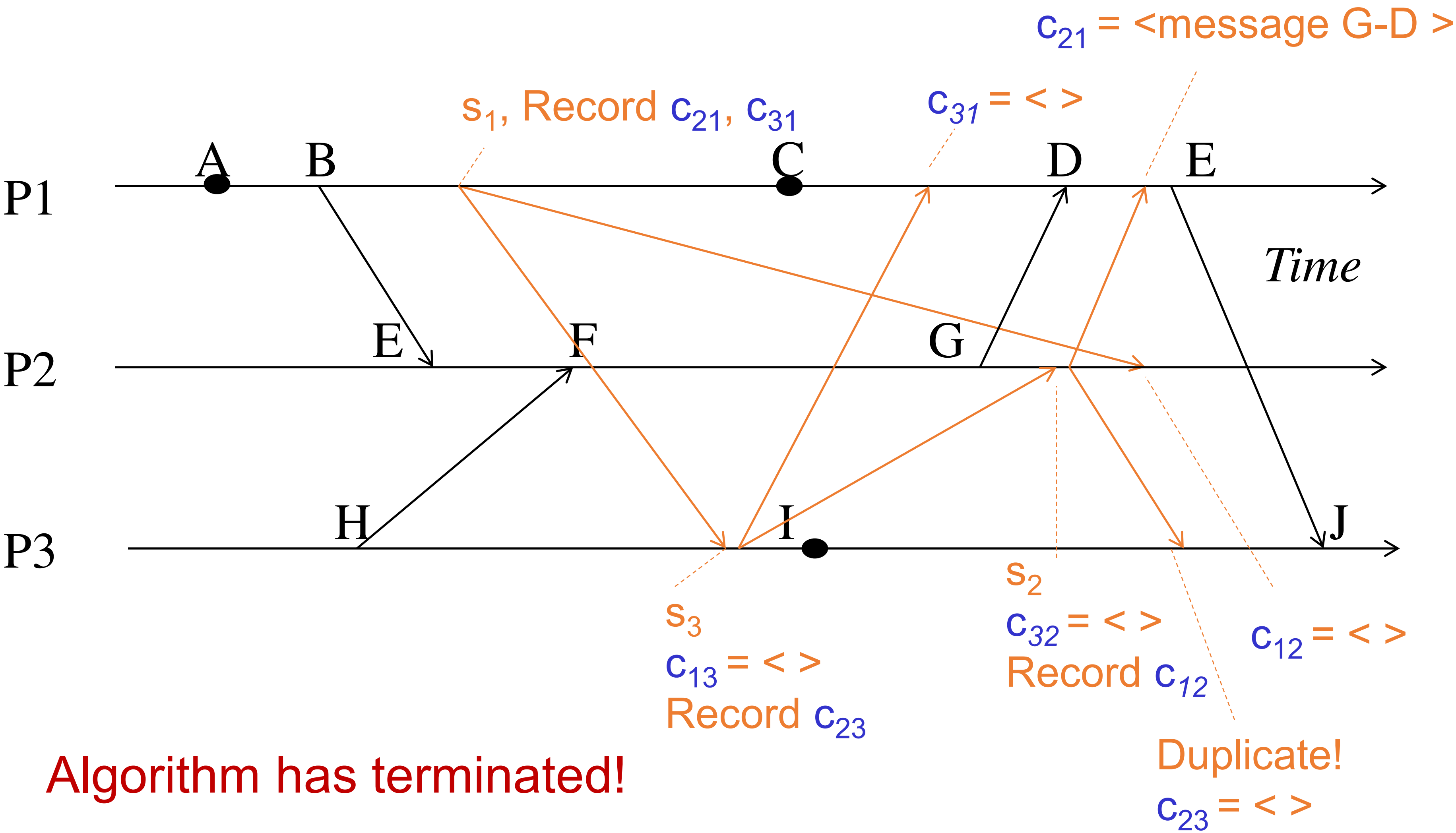
Duplicate!
c_{21} = <message G-D >
s_1, Record c_{21}, c_{31}
c_{31} = < >
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s_3
c_{13} = < >
Record c_{23}
s_2
c_{32} = < >
Record c_{12}
c_{12} = < >

Example



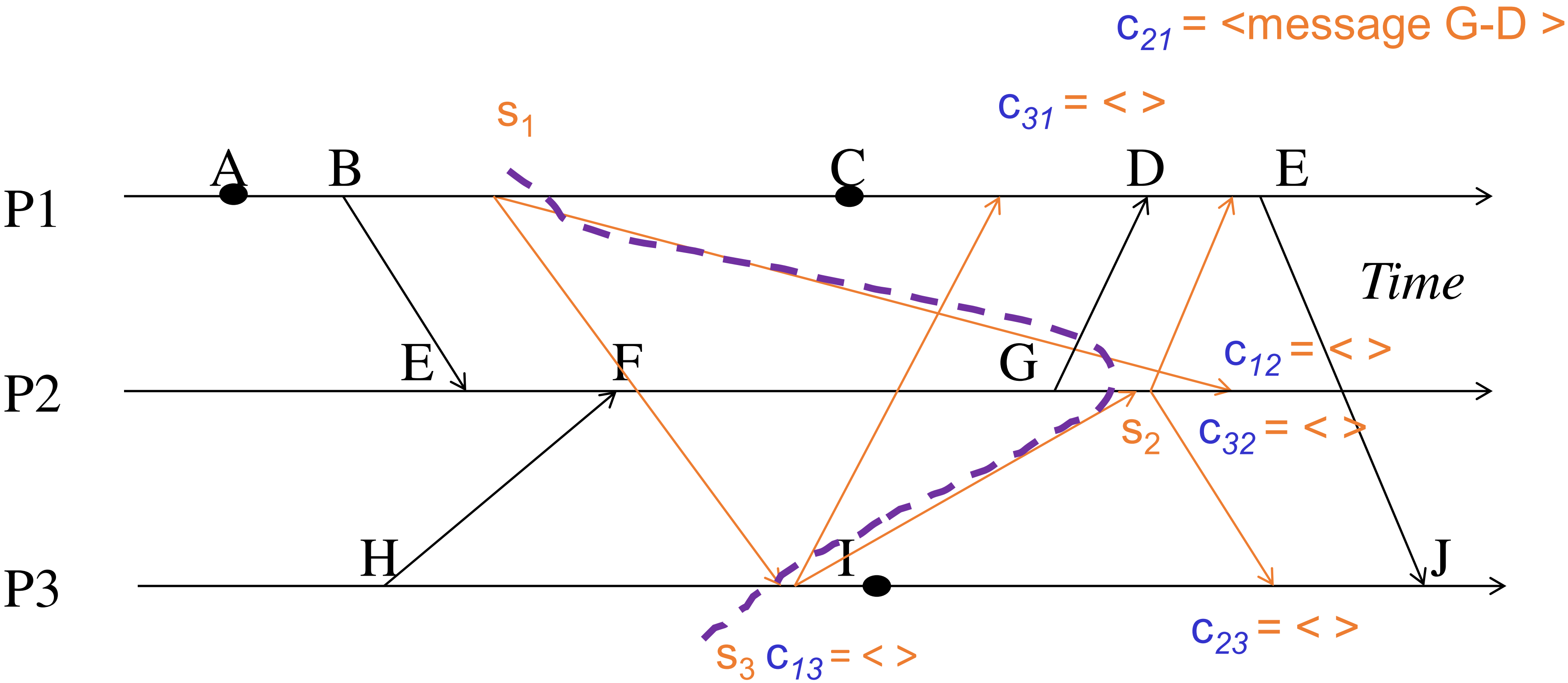
c_{21} = <message G-D >
s_1, Record c_{21}, c_{31}
c_{31} = < >
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s_3
c_{13} = < >
Record c_{23}
s_2
c_{32} = < >
Record c_{12}
c_{12} = < >
Duplicate!
c_{23} = < >

Example



Algorithm has terminated!

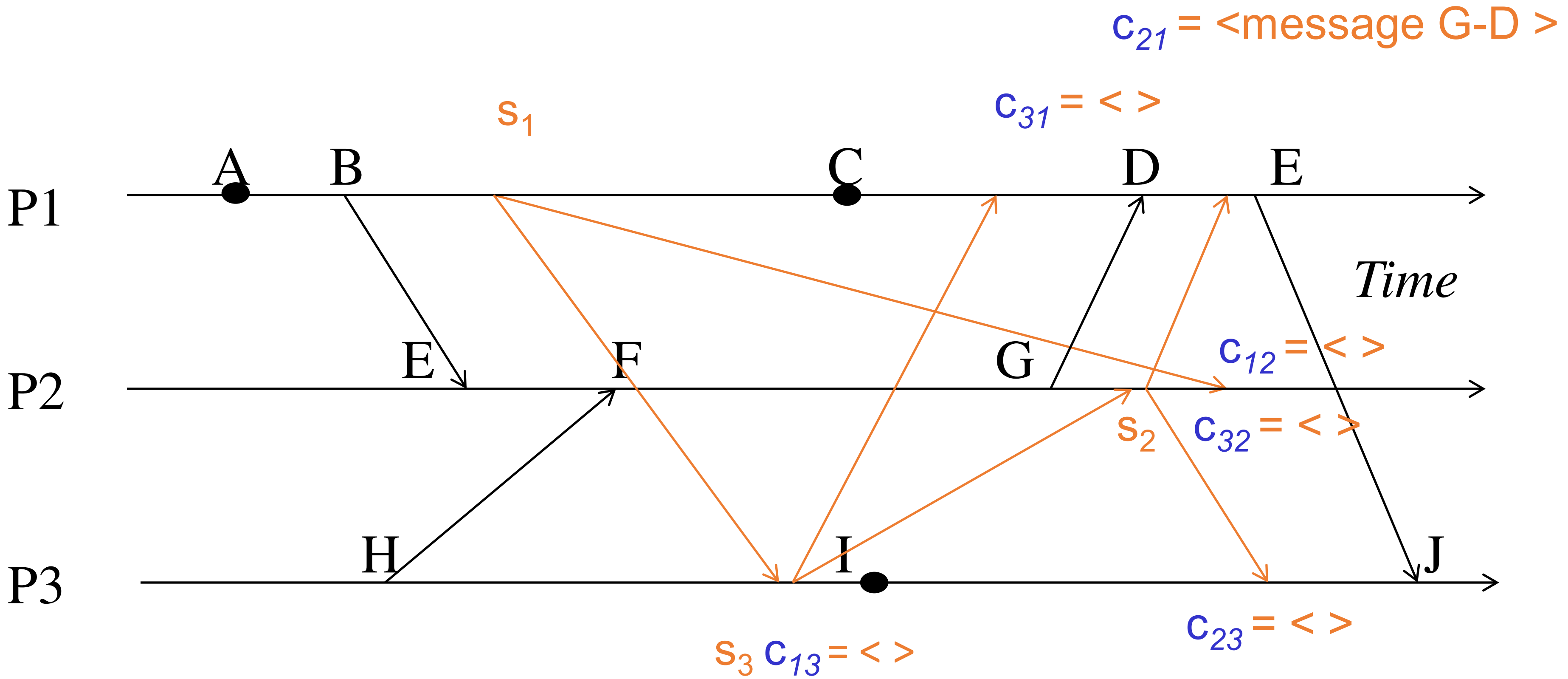
Example



Frontier for the resulting cut:
{B, G, H}

Channel state for the cut:
Only c_{21} has a pending message.

Example



Global snapshots pieces can be collected at a central location.

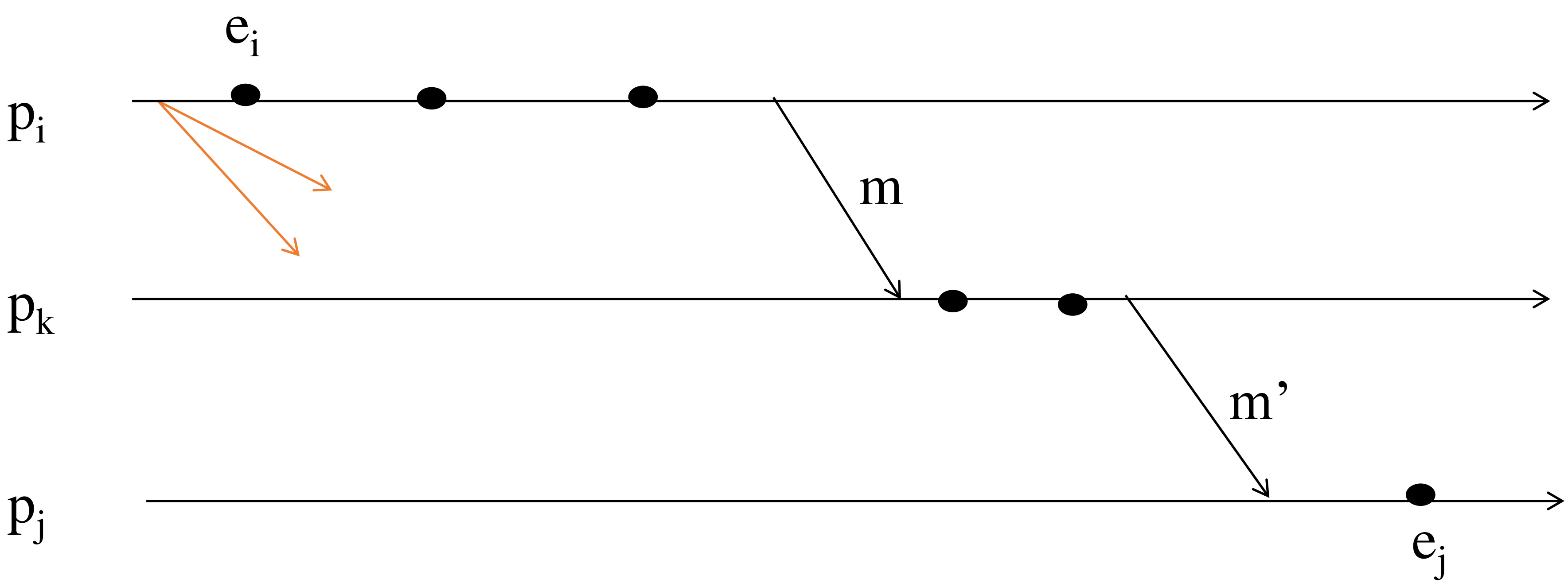
Chandy-Lamport Algorithm: Properties

- Any run of the Chandy-Lamport Global Snapshot algorithm creates a consistent cut.
- Let e_i and e_j be events occurring at p_i and p_j, respectively such that
  - $e_i \rightarrow e_j$ (e_i happens before e_j)
- The snapshot algorithm ensures that

  if e_j is in the cut then e_i is also in the cut.
- That is: if $e_j \rightarrow$ < p_j records its state>, then it must be true that $e_i \rightarrow$ <p_i records its state>.

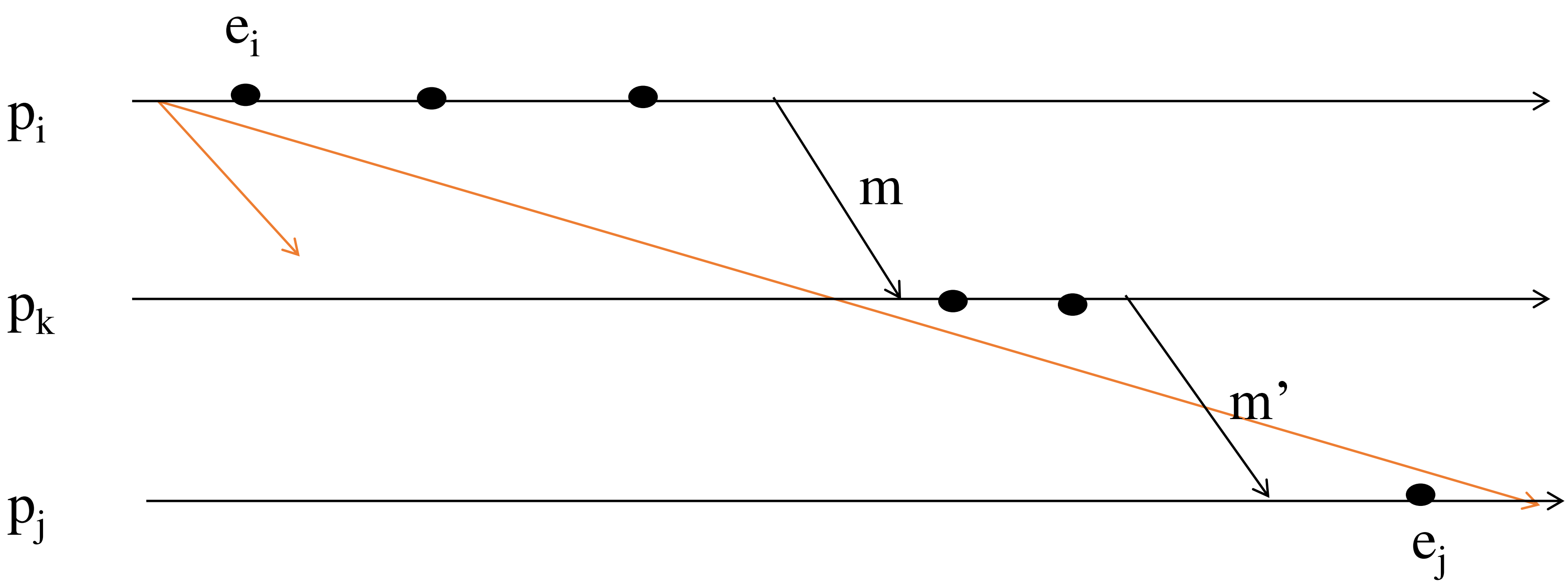
Chandy-Lamport Algorithm: Properties

- Given $e_i \rightarrow e_j$. If $e_j \rightarrow$ < p_j records its state>, then it must be true that $e_i \rightarrow$ <p_i records its state>.
- By contradiction, *suppose $e_j \rightarrow$ < p_j records its state>, and <p_i records its state> $\rightarrow e_i$.*



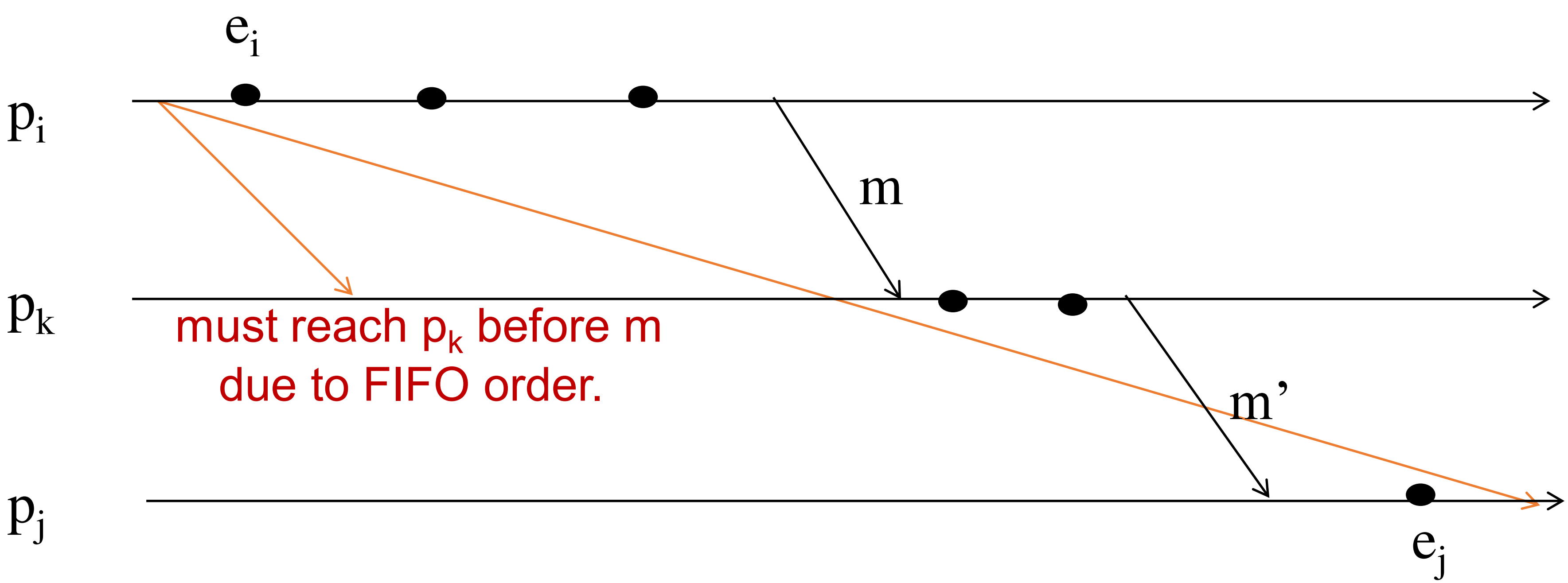
Chandy-Lamport Algorithm: Properties

- Given $e_i \rightarrow e_j$. If $e_j \rightarrow$ < p_j records its state>, then it must be true that $e_i \rightarrow$ <p_i records its state>.
- By contradiction, *suppose $e_j \rightarrow$ < p_j records its state>, and <p_i records its state> $\rightarrow e_i$.*



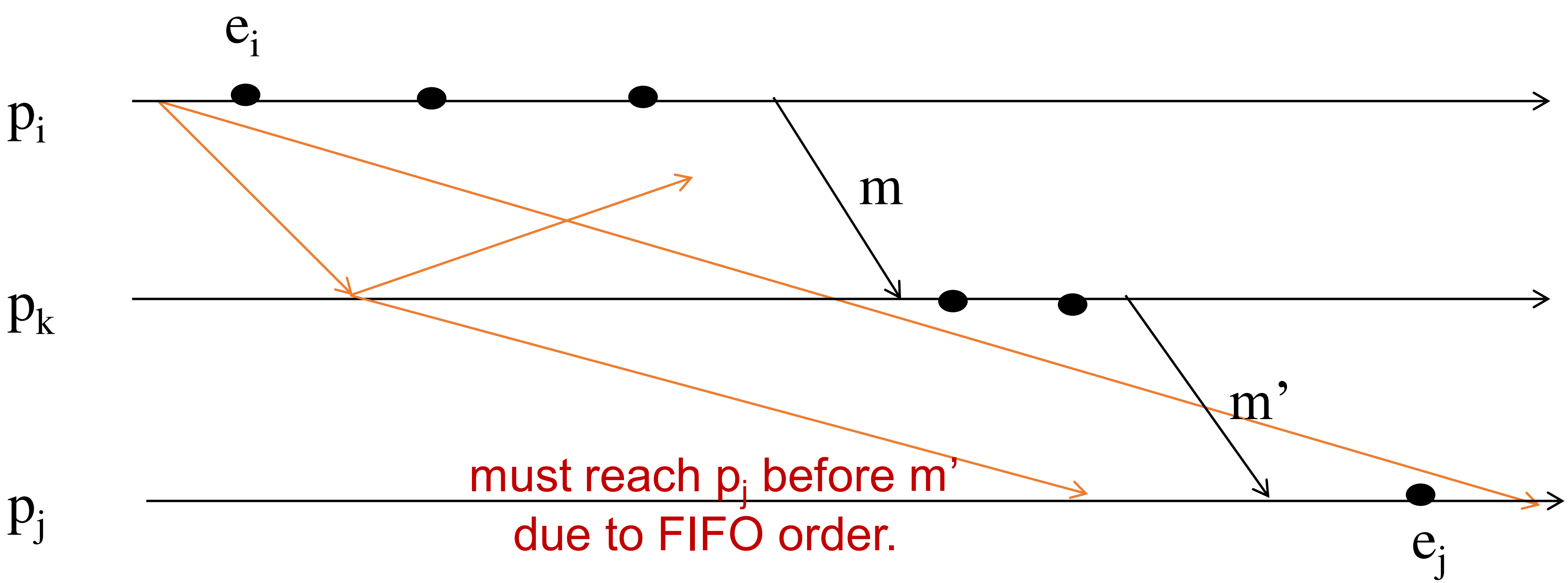
Chandy-Lamport Algorithm: Properties

- Given $e_i \rightarrow e_j$. If $e_j \rightarrow$ < p_j records its state>, then it must be true that $e_i \rightarrow$ <p_i records its state>.
- By contradiction, *suppose $e_j \rightarrow$ < p_j records its state>, and <p_i records its state> $\rightarrow e_i$.*



Chandy-Lamport Algorithm: Properties

- Given $e_i \rightarrow e_j$. If $e_j \rightarrow$ < p_j records its state>, then it must be true that $e_i \rightarrow$ <p_i records its state>.
- By contradiction, *suppose $e_j \rightarrow$ < p_j records its state>, and <p_i records its state> $\rightarrow e_i$.*



Chandy-Lamport Algorithm: Properties

- Given $e_i \rightarrow e_j$. If $e_j \rightarrow$ < p_j records its state>, then it must be true that $e_i \rightarrow$ <p_i records its state>.
- By contradiction, *suppose $e_j \rightarrow$ < p_j records its state>, and <p_i records its state> $\rightarrow e_i$.*
- Consider the path of app messages (through other processes) that go from e_i to e_j .
- Due to FIFO ordering, markers on each link in above path will precede regular app messages.
- Thus, since <p_i records its state> $\rightarrow e_i$, it must be true that p_j received a marker before e_j.
- Thus e_j is not in the cut => contradiction.

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 and Liveness.

Chandy-Lamport Algorithm: Usefulness

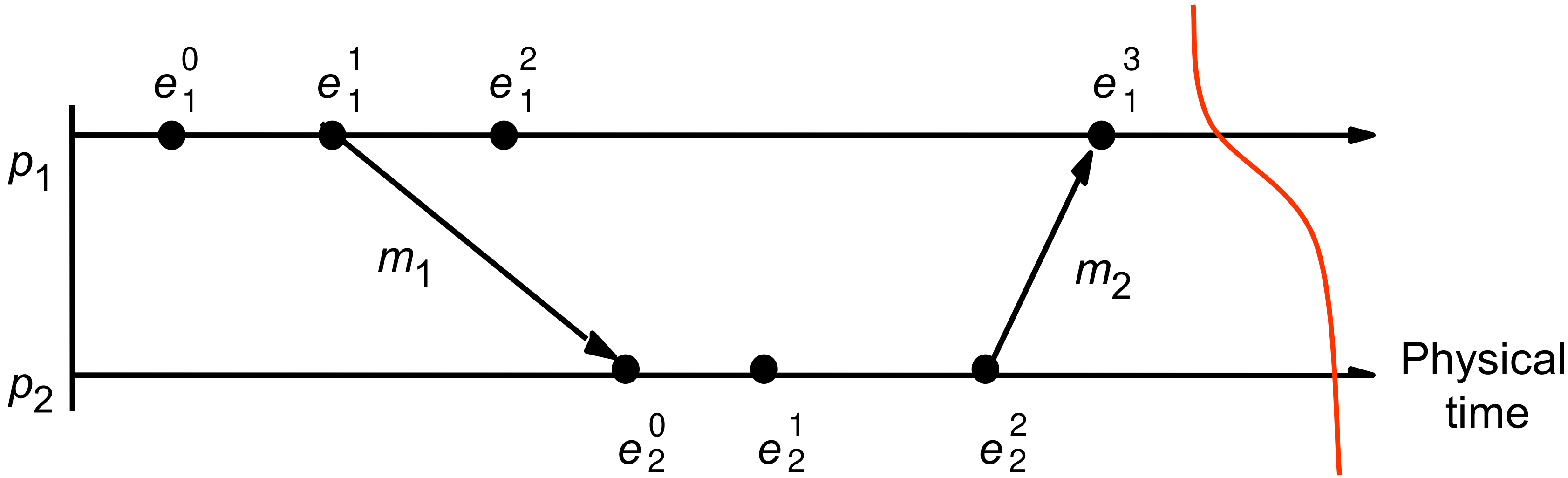
- Consistent global snapshots are useful for detecting global system properties:
  - Safety
  - Liveness

More notations and definitions

- history(p_i) = h_i = <e_i^0, e_i^1, … >
- global history: $H = \cup_i (h_i)$

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

Example



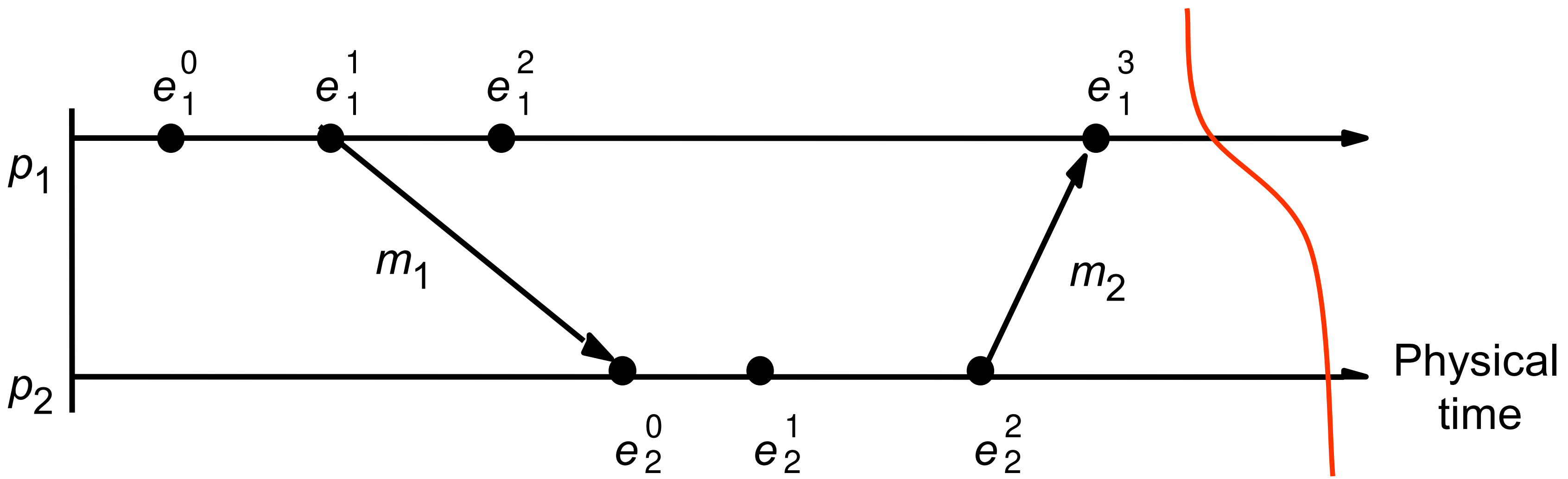
Order at p_1: < e_1^0, e_1^1, e_1^2, e_1^3 > Order at p_2: < e_2^0, e_2^1,e_2^2>

Causal order across p_1 and p_2: < e_1^0, e_1^1, e_2^0, e_2^1, e_2^2 , e_1^3 >

Run: < e_1^0, e_1^1, e_1^2, e_1^3 , e_2^0, e_2^1 e_2^2 >

Linearization: < e_1^0, e_1^1, e_1^2, e_2^0, e_2^1 e_2^2 , e_1^3 >

Example



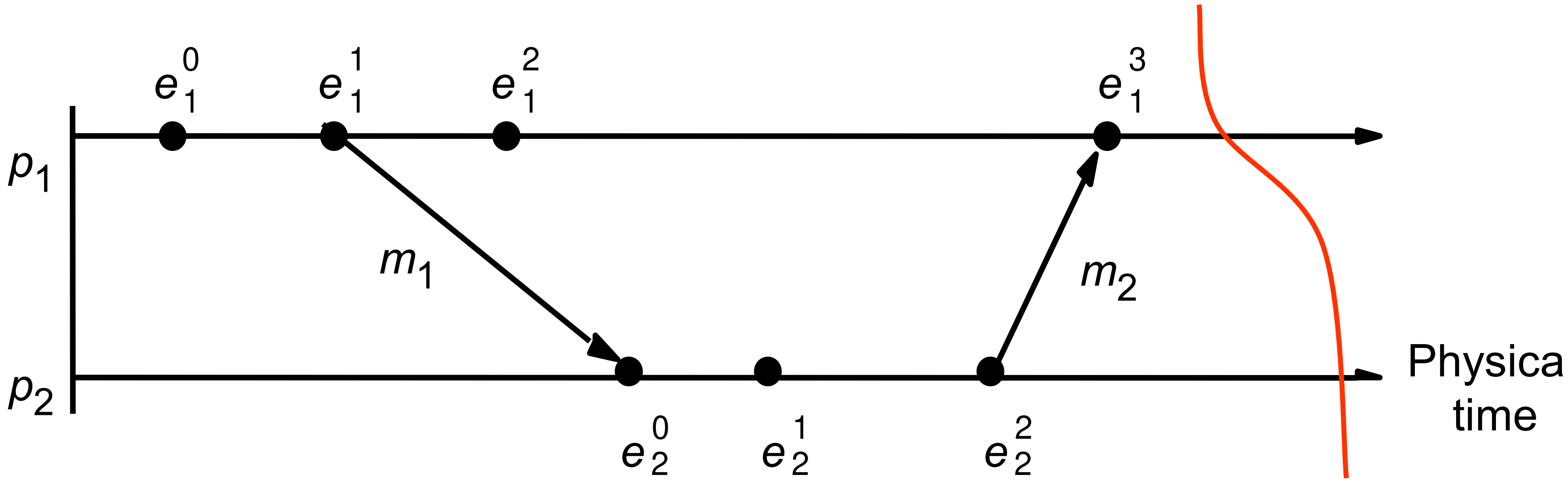
Order at p_1: < e_1^0, e_1^1, e_1^2, e_1^3 > Order at p_2: < e_2^0, e_2^1,e_2^2>

Causal order across p_1 and p_2: < e_1^0, e_1^1, e_2^0, e_2^1, e_2^2 , e_1^3 >

Run: < e_1^0, e_1^1, e_1^2, e_1^3 , e_2^0, e_2^1 e_2^2 >

Linearization: < e_1^0, e_1^1, e_1^2, e_2^0, e_2^1 e_2^2 , e_1^3 >

Example

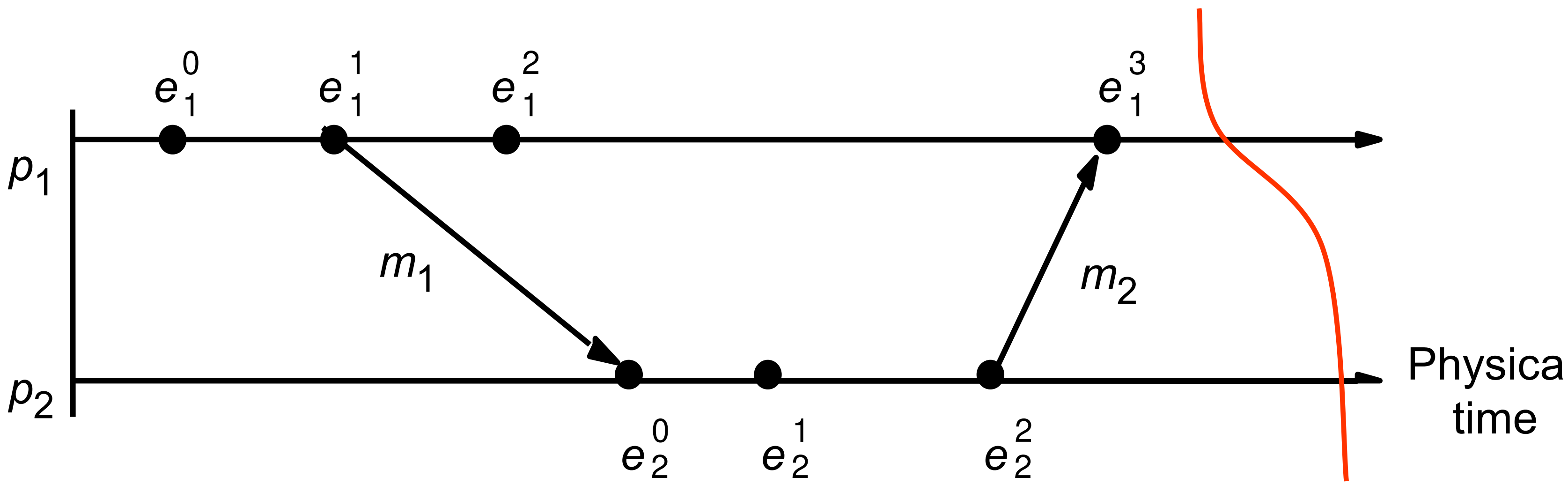


Order at p_1: $< e_1^0, e_1^1, e_1^2, e_1^3 >$ Order at p_2: $< e_2^0, e_2^1, e_2^2 >$

Causal order across p_1 and p_2: $< e_1^0, e_1^1, e_2^0, e_2^1, e_2^2, e_1^3 >$

$< e_1^0, e_1^1, e_2^0, e_2^1, e_1^2, e_2^2, e_1^3 >$

Example



Order at p_1: $< e_1^0, e_1^1, e_1^2, e_1^3 >$ Order at p_2: $< e_2^0, e_2^1, e_2^2 >$

Causal order across p_1 and p_2: $< e_1^0, e_1^1, e_2^0, e_2^1, e_2^2, e_1^3 >$

$< e_1^0, e_1^1, e_2^0, e_2^1, e_1^2, e_2^2, e_1^3 >$: Linearization

$< e_1^0, e_2^1, e_2^0, e_1^1, e_1^2, e_2^2, e_1^3 >$: Not even a run

More notations and definitions

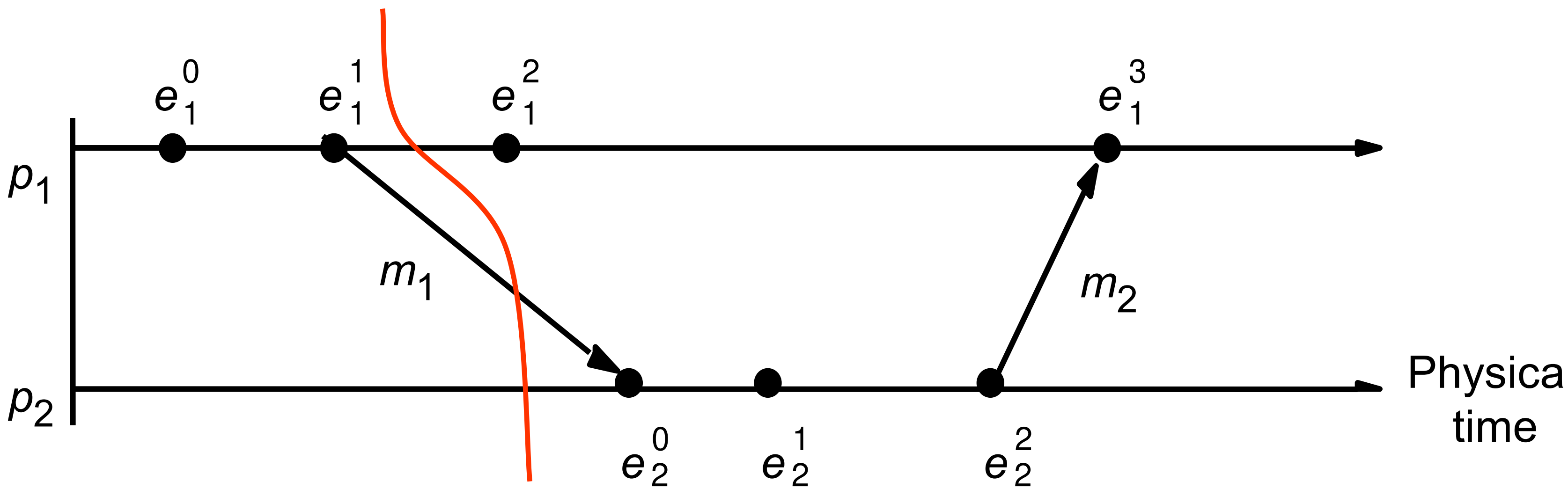
- history(p_i) = $h_i = <e_i^0, e_i^1, \ldots >$
- global history: $H = \cup_i (h_i)$

- A run is a total ordering of events in H that is consistent with each h_i's ordering.

- A linearization is a run consistent with happens-before ($\rightarrow$) relation in H.

- Linearizations pass through consistent global states.

Example

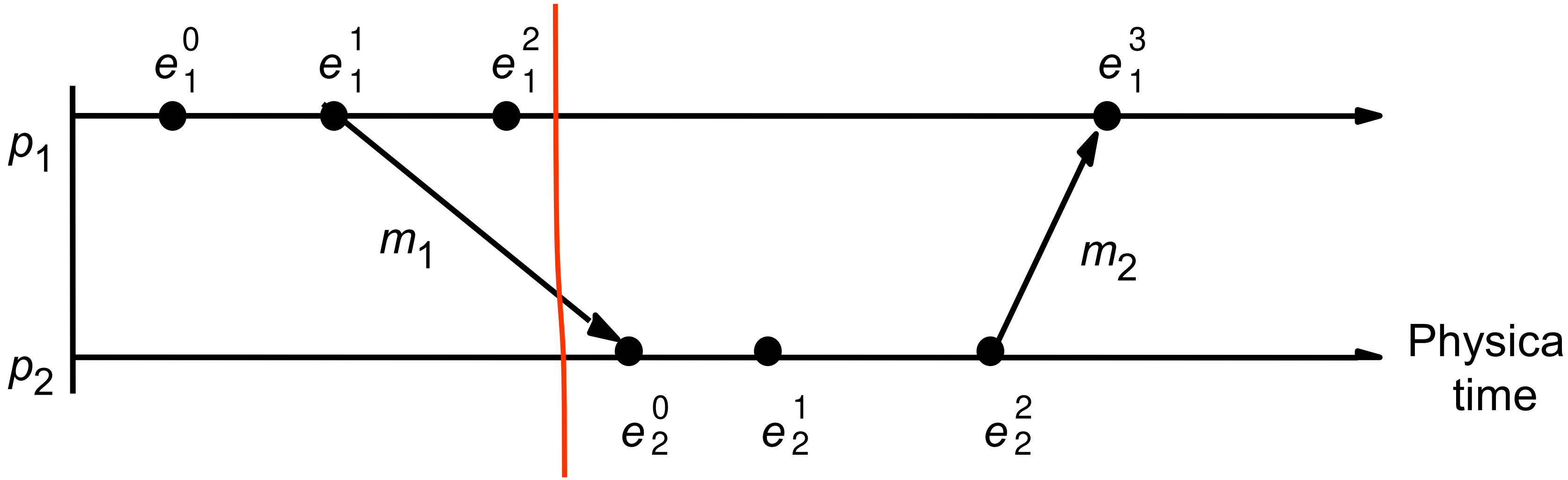


Order at p_1: < e_1^0, e_1^1, e_1^2, e_1^3 > Order at p_2: < e_2^0, e_2^1,e_2^2>

Causal order across p_1 and p_2: < e_1^0, e_1^1, e_2^0, e_2^1, e_2^2 , e_1^3 >

Linearization: < e_1^0, e_1^1, e_1^2, e_2^0, e_2^1 e_2^2 , e_1^3 >

Example

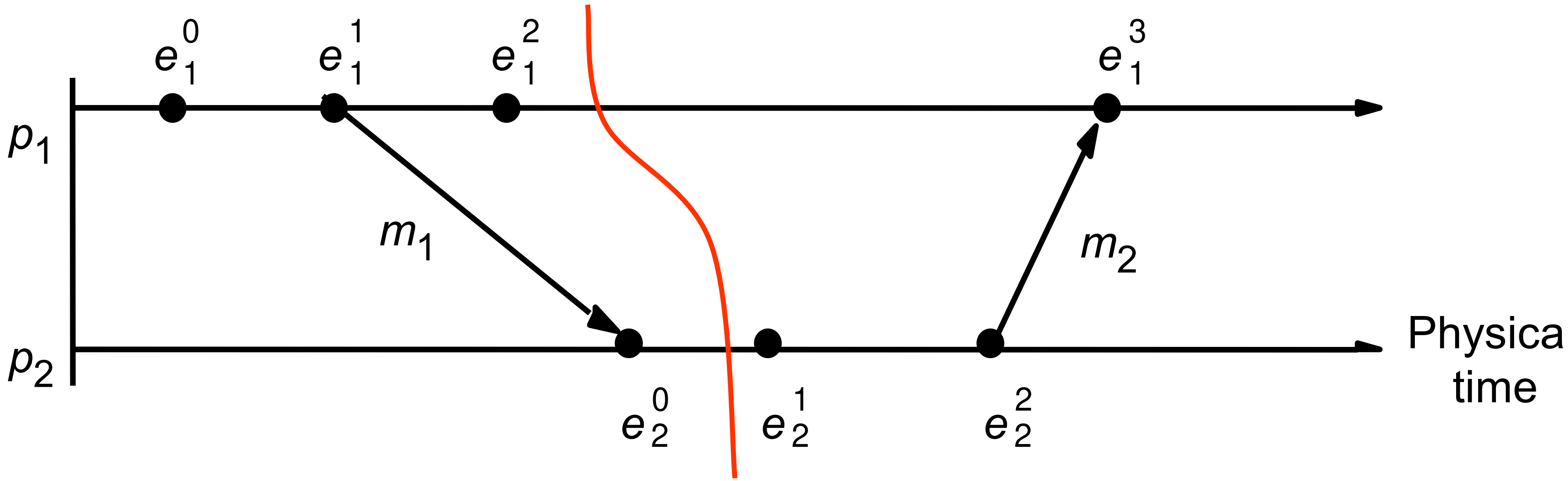


Order at p_1: < e_1^0, e_1^1, e_1^2, e_1^3 > Order at p_2: < e_2^0, e_2^1,e_2^2>

Causal order across p_1 and p_2: < e_1^0, e_1^1, e_2^0, e_2^1 e_2^2 , e_1^3 >

Linearization: < e_1^0, e_1^1, e_1^2, e_2^0, e_2^1 e_2^2 , e_1^3 >

Example

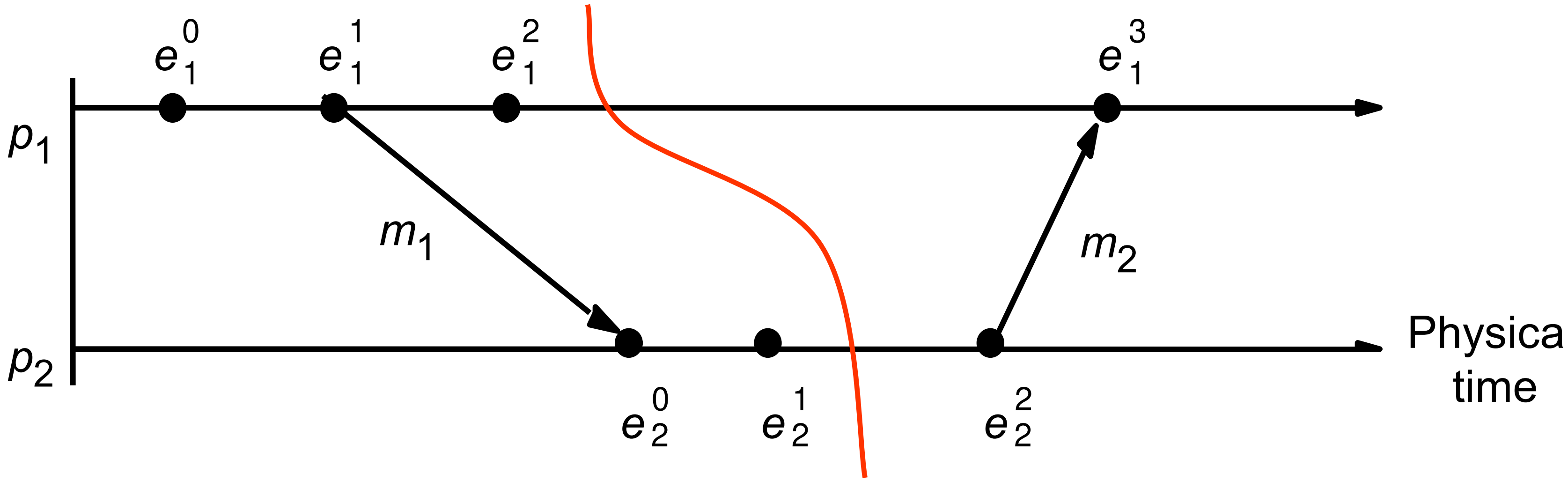


Order at p_1: < e_1^0, e_1^1, e_1^2, e_1^3 > Order at p_2: < e_2^0, e_2^1,e_2^2>

Causal order across p_1 and p_2: < e_1^0, e_1^1, e_2^0, e_2^1 e_2^2 , e_1^3 >

Linearization: < e_1^0, e_1^1, e_1^2, e_2^0, e_2^1 e_2^2 , e_1^3 >

Example

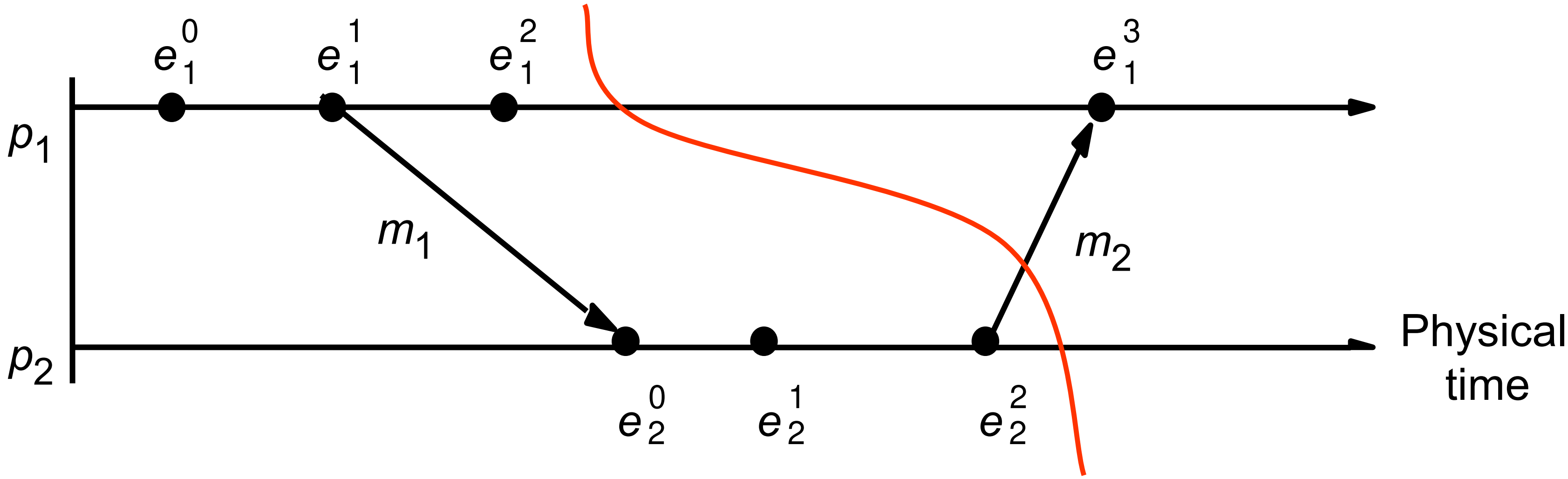


Order at p_1: < e_1^0, e_1^1, e_1^2, e_1^3 > Order at p_2: < e_2^0, e_2^1,e_2^2>

Causal order across p_1 and p_2: < e_1^0, e_1^1, e_2^0, e_2^1 e_2^2 , e_1^3 >

Linearization: < e_1^0, e_1^1, e_1^2, e_2^0, e_2^1, e_2^2 , e_1^3 >

Example

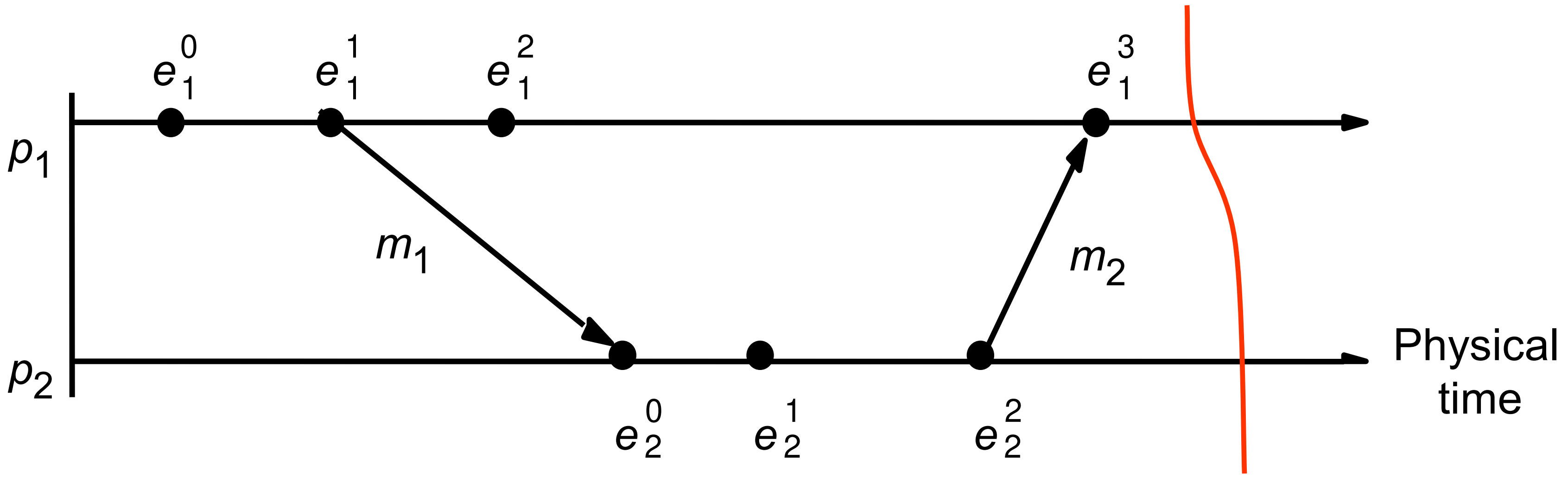


Order at p_1: < e_1^0, e_1^1, e_1^2, e_1^3 >   Order at p_2: < e_2^0, e_2^1,e_2^2>

Causal order across p_1 and p_2: < e_1^0, e_1^1, e_2^0, e_2^1 e_2^2 , e_1^3 >

Linearization: < e_1^0, e_1^1, e_1^2, e_2^0, e_2^1 e_2^2 , e_1^3 >

Example

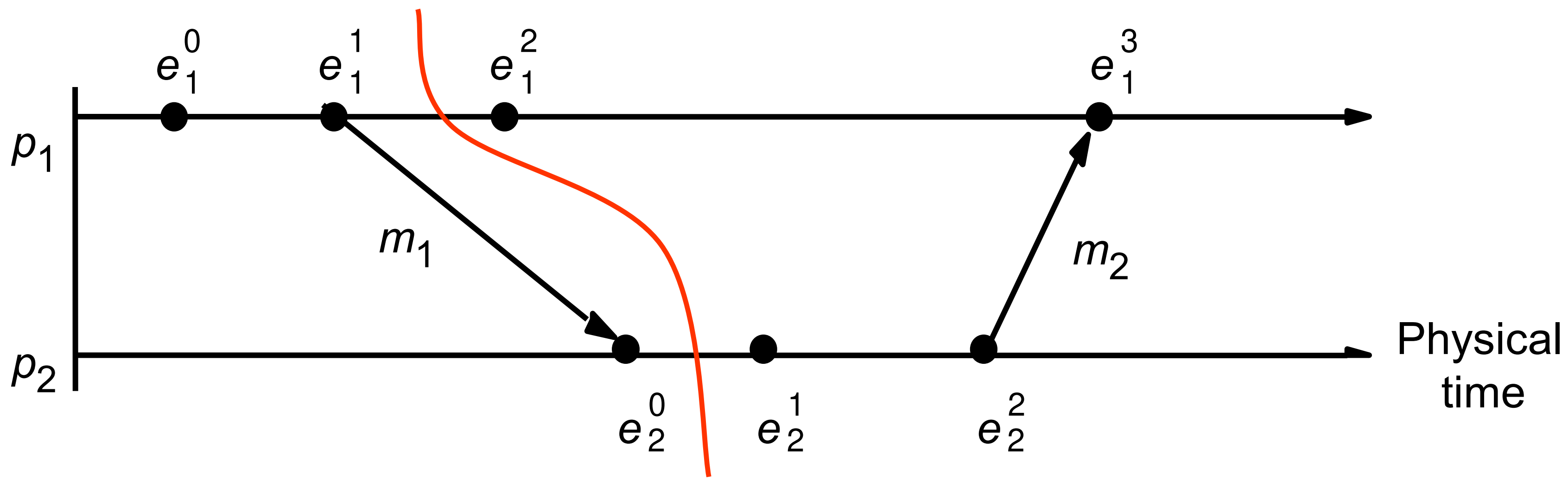


Order at p_1: < e_1^0, e_1^1, e_1^2, e_1^3 > Order at p_2: < e_2^0, e_2^1,e_2^2>

Causal order across p_1 and p_2: < e_1^0, e_1^1, e_2^0, e_2^1 e_2^2 , e_1^3 >

Linearization: < e_1^0, e_1^1, e_1^2, e_2^0, e_2^1 e_2^2 , e_1^3 >

Example



Order at p_1: $< e_1^0, e_1^1, e_1^2, e_1^3 >$ Order at p_2: $< e_2^0, e_2^1, e_2^2 >$

Causal order across p_1 and p_2: $< e_1^0, e_1^1, e_2^0, e_2^1\ e_2^2, e_1^3 >$

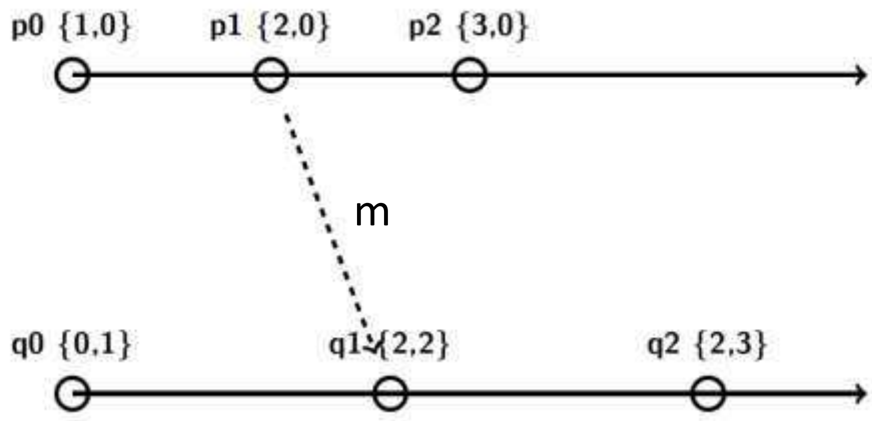
Linearization: $< e_1^0, e_1^1, e_1^2, e_2^0, e_2^1\ e_2^2, e_1^3 >$

Linearization $< e_1^0, e_1^1, e_2^0, e_2^1, e_1^2, e_2^2, e_1^3 >$

More notations and definitions

- Linearizations pass through consistent global states.
- A global state S_k is reachable from global state S_i, if there is a linearization that passes through S_i and then through S_k.
- The distributed system evolves as a series of transitions between global states S_0 , S_1 , ….

State Transitions: Example



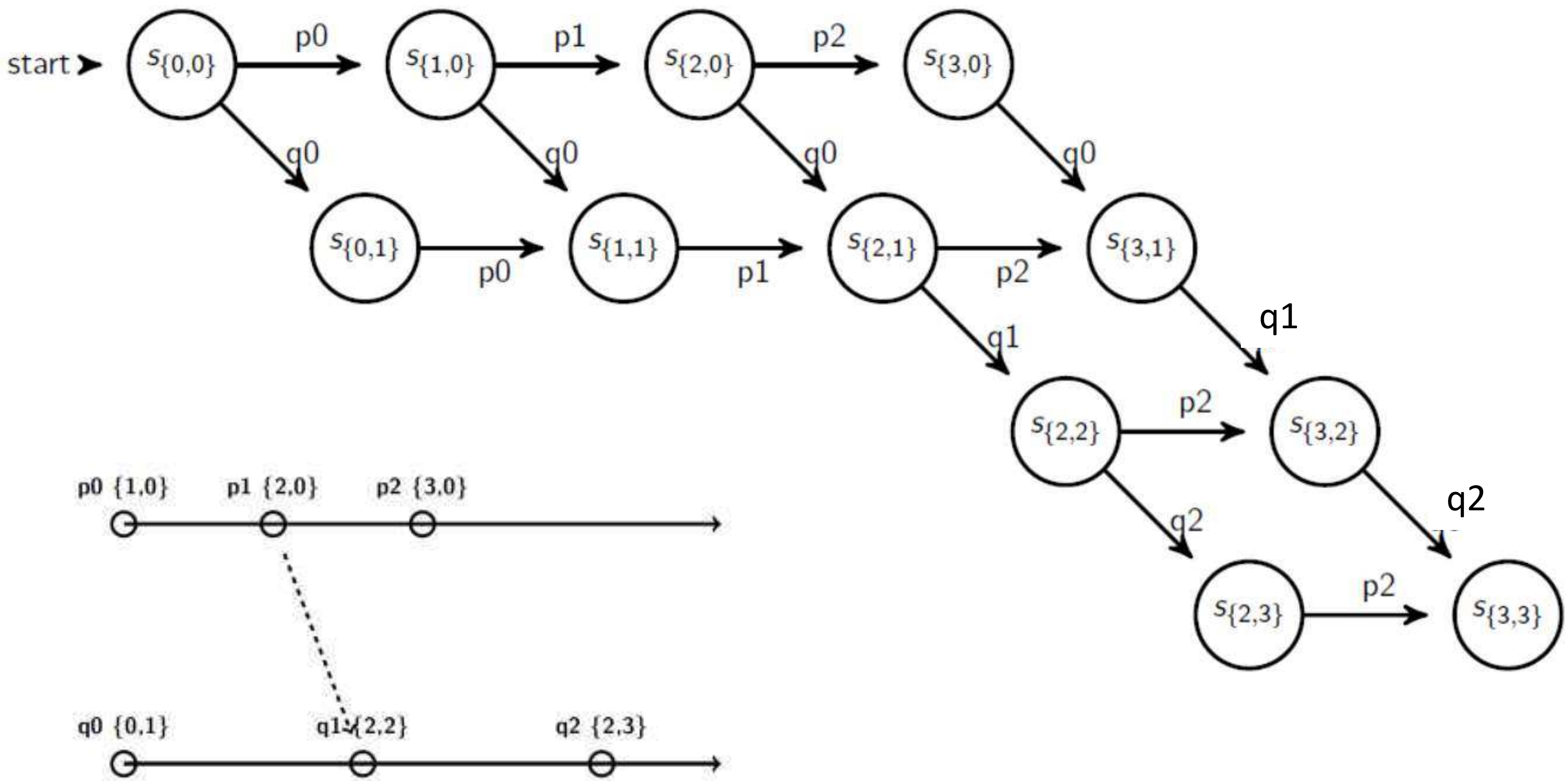
- Causal order:
  - p0 → p1 → p2
  - q0 → q1 → q2
  - p0 → p1 → q1 → q2
- Concurrent:
  - p0 || q0
  - p1 || q0
  - p2 || q0, p2 || q1, p2 || q2

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 >
- ……

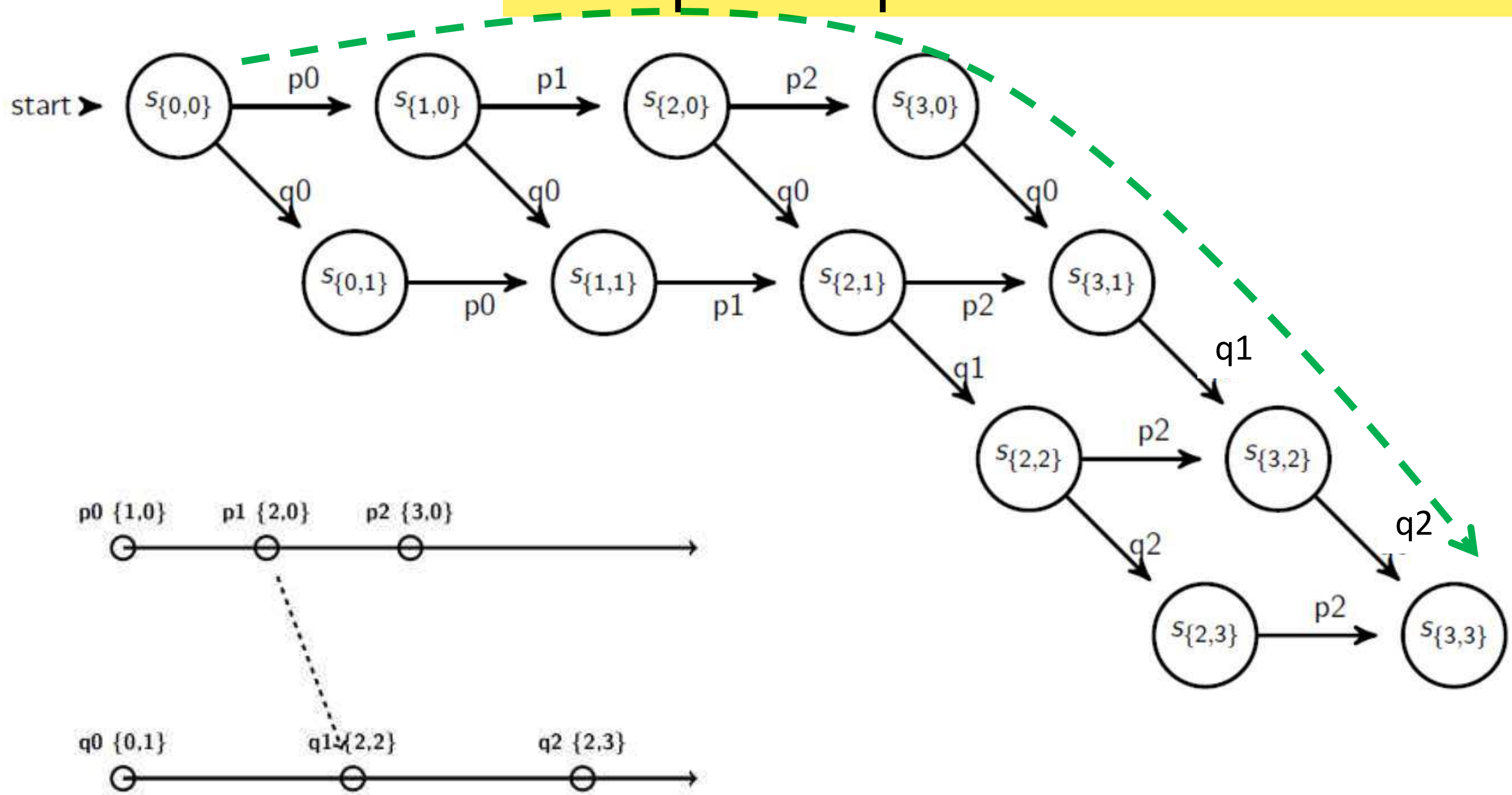
State Transitions: Example

Execution Lattice. Each path represents a linearization.



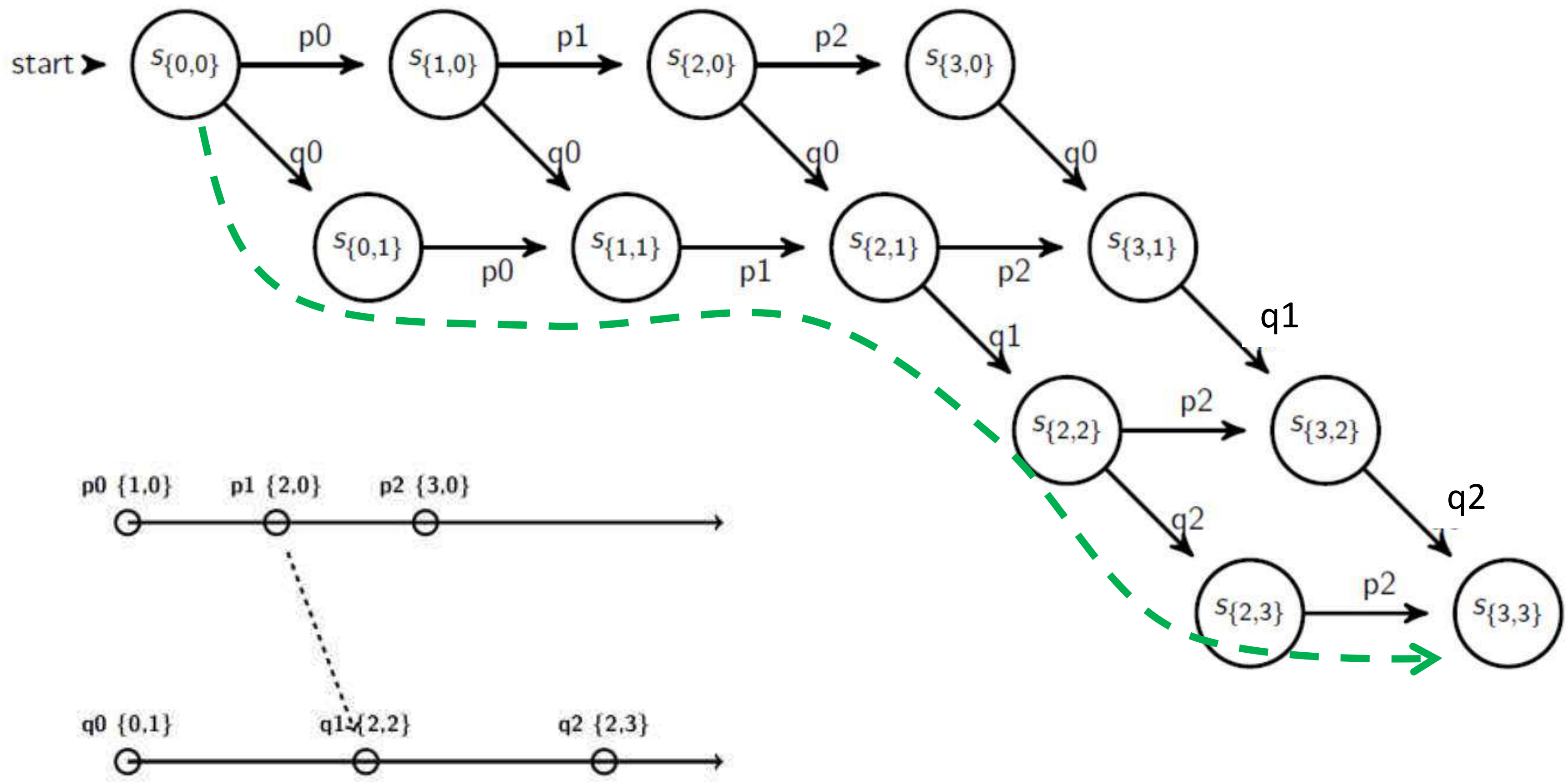
State Transitions: Example

Execution Lattice. Each path represents a linearization.



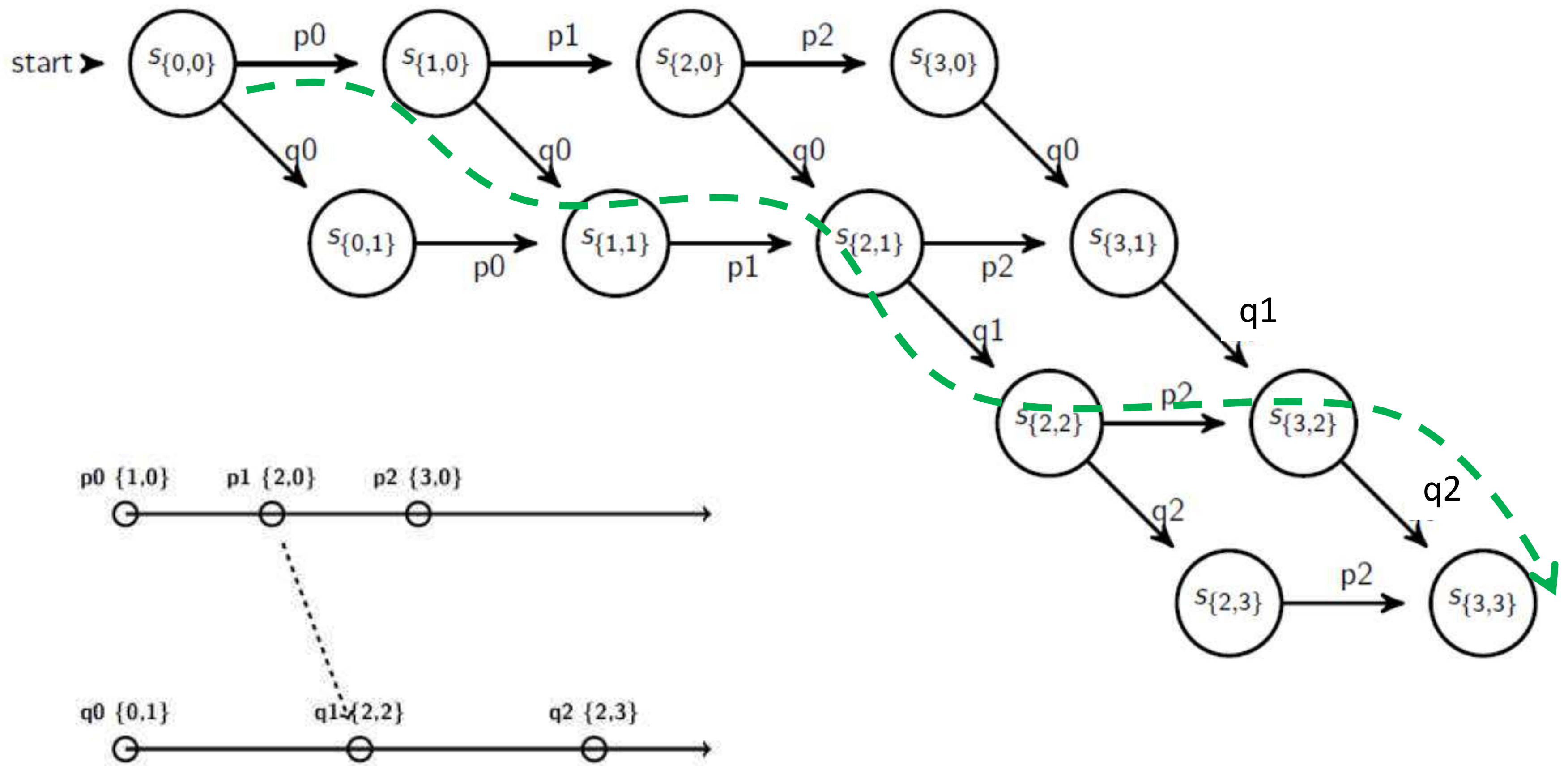
State Transitions: Example

Execution Lattice. Each path represents a linearization.

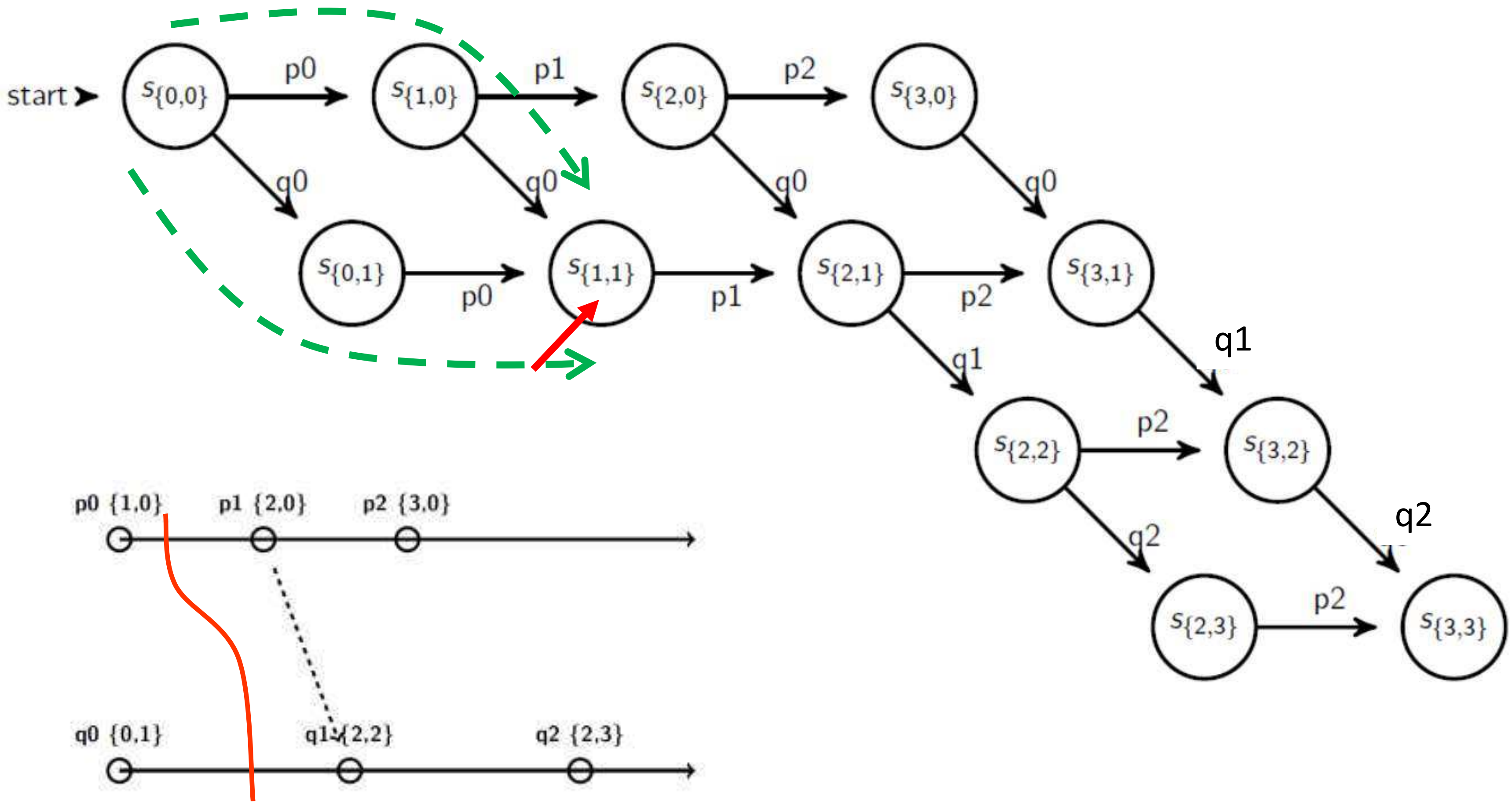


State Transitions: Example

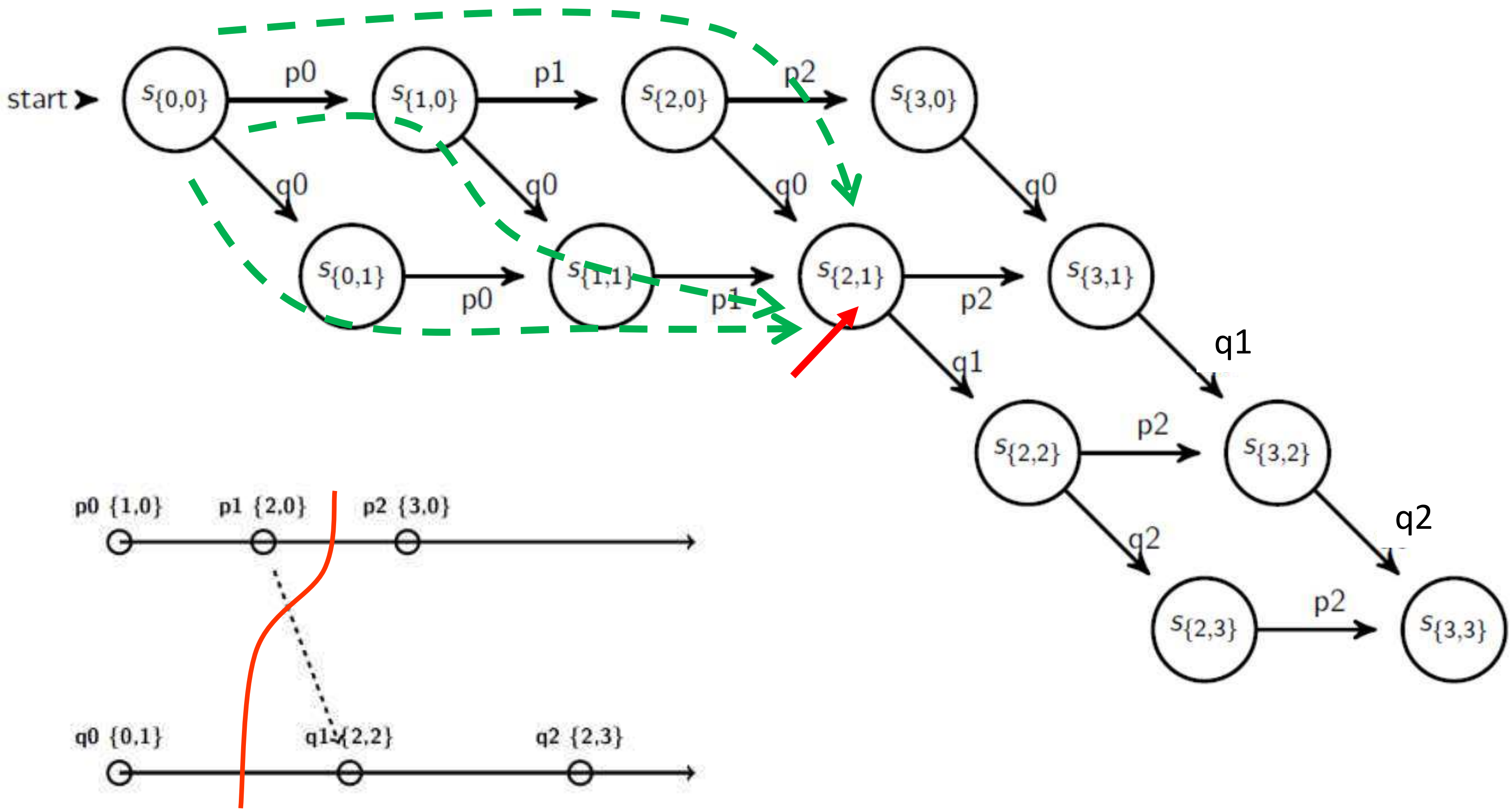
Execution Lattice. Each path represents a linearization.



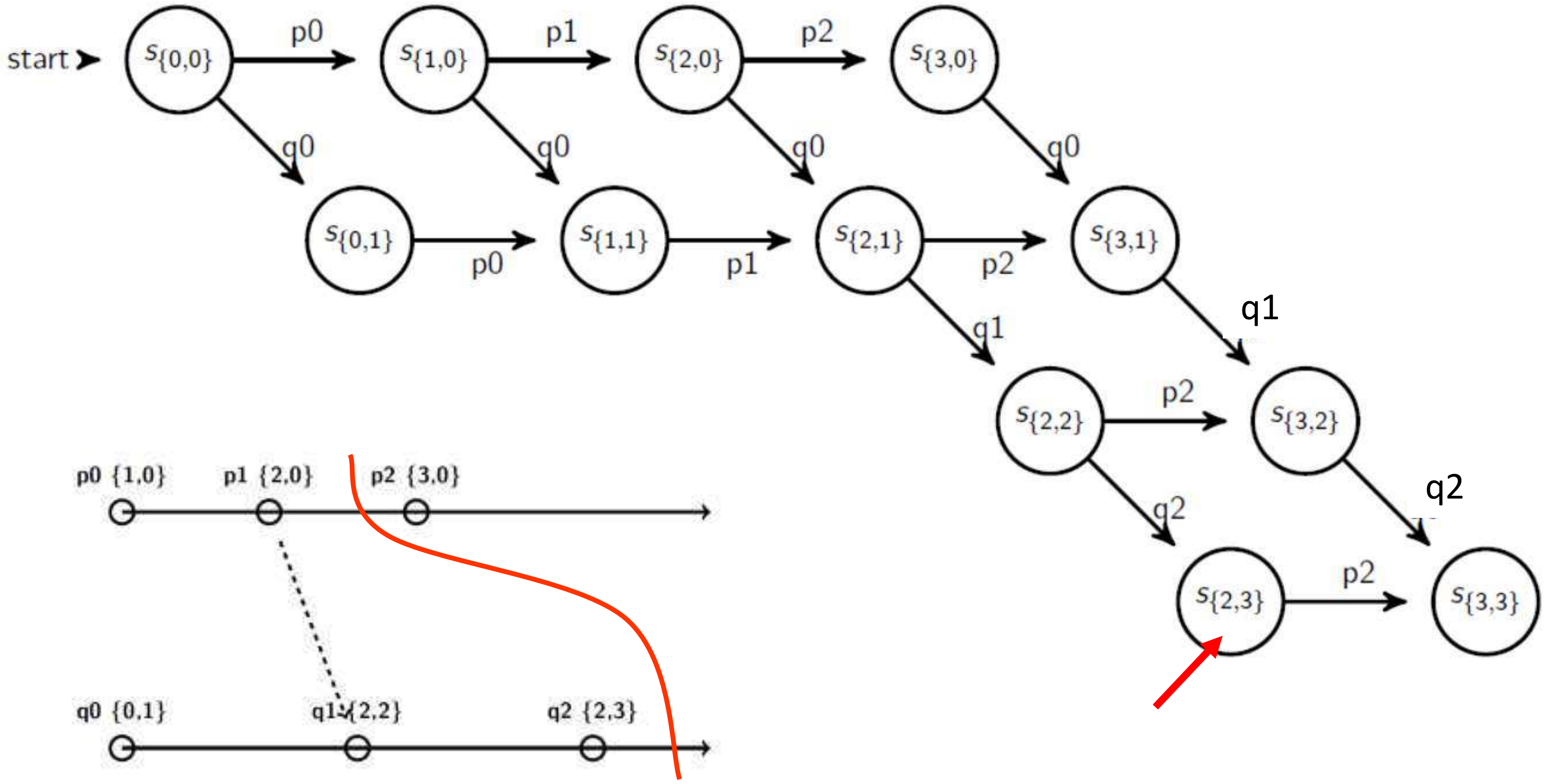
State Transitions: Example



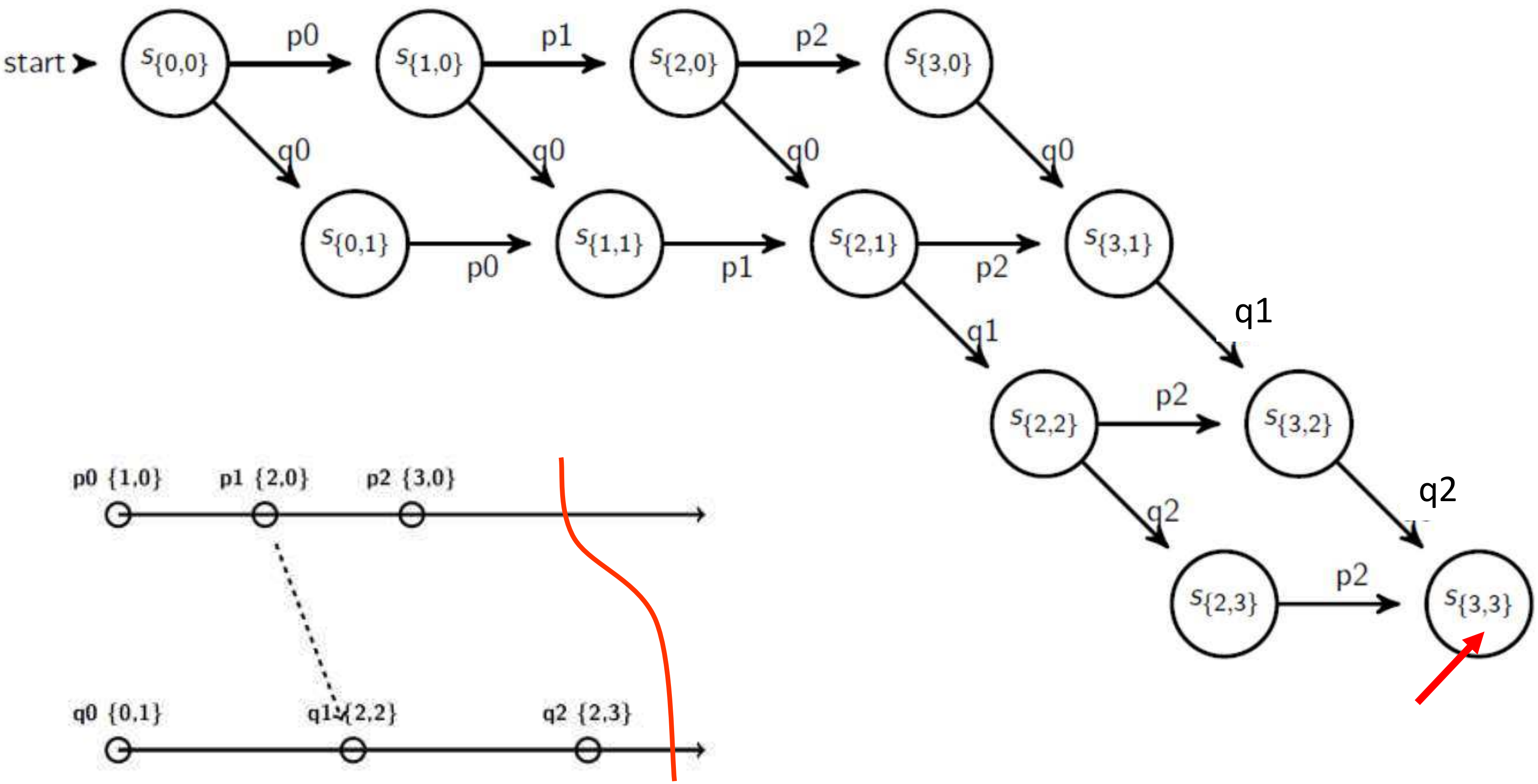
State Transitions: Example



State Transitions: Example



State Transitions: Example



More notations and definitions

- A run is a total ordering of events in H that is consistent with each 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 S_k is reachable from global state S_i, if there is a linearization that passes through S_i and then through S_k.
- The distributed system evolves as a series of transitions between global states S_0 , S_1 , ….

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

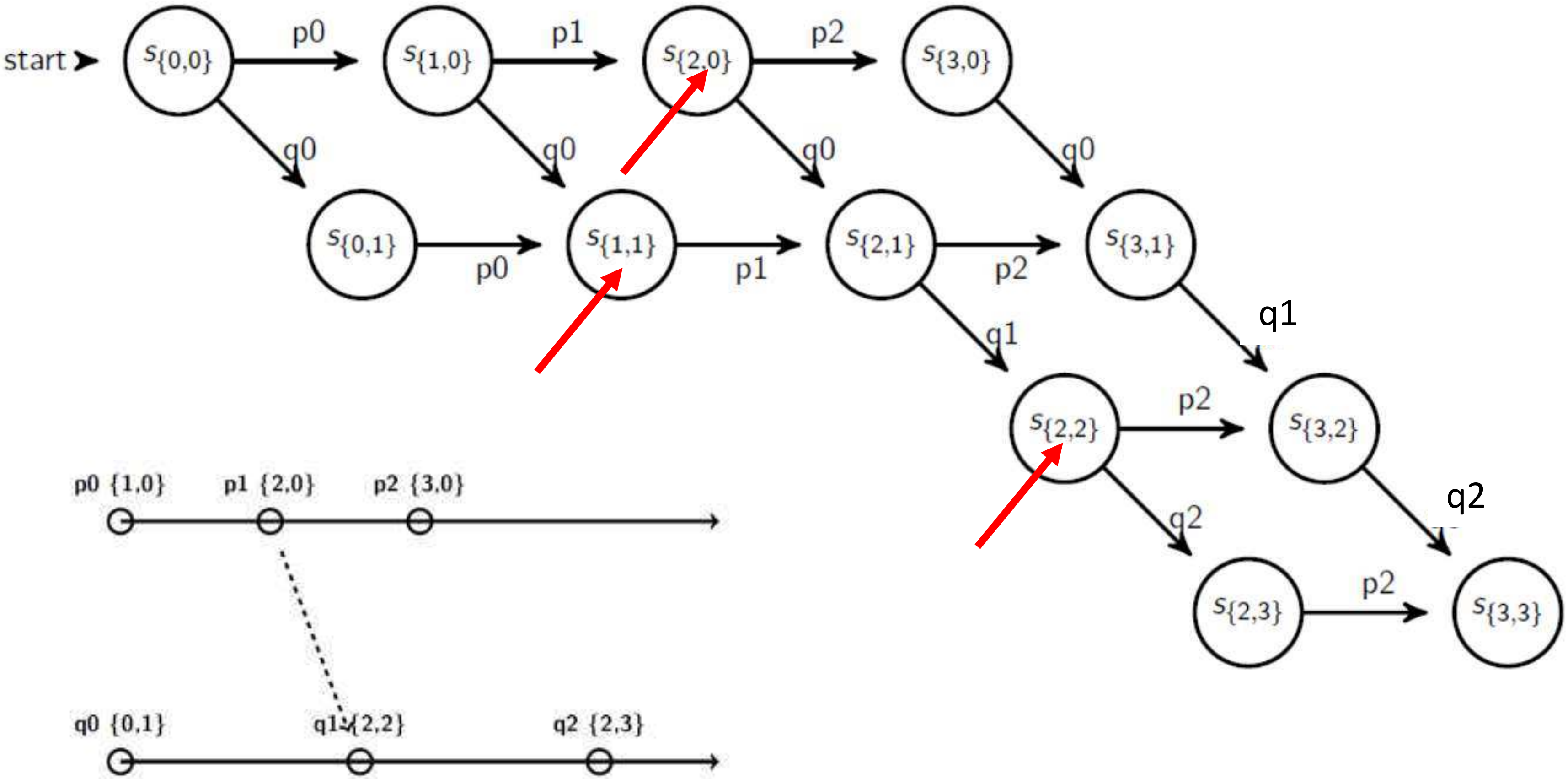
- Liveness = guarantee that something good will happen, eventually

- 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:
  - liveness(P(S_0)) $\equiv$ $\forall L \in$ linearizations from S_0, L passes through a S_L & P(S_L) = true
  - For all linearizations starting from S_0, P is true for some state S_L reachable from S_0.

Liveness Example

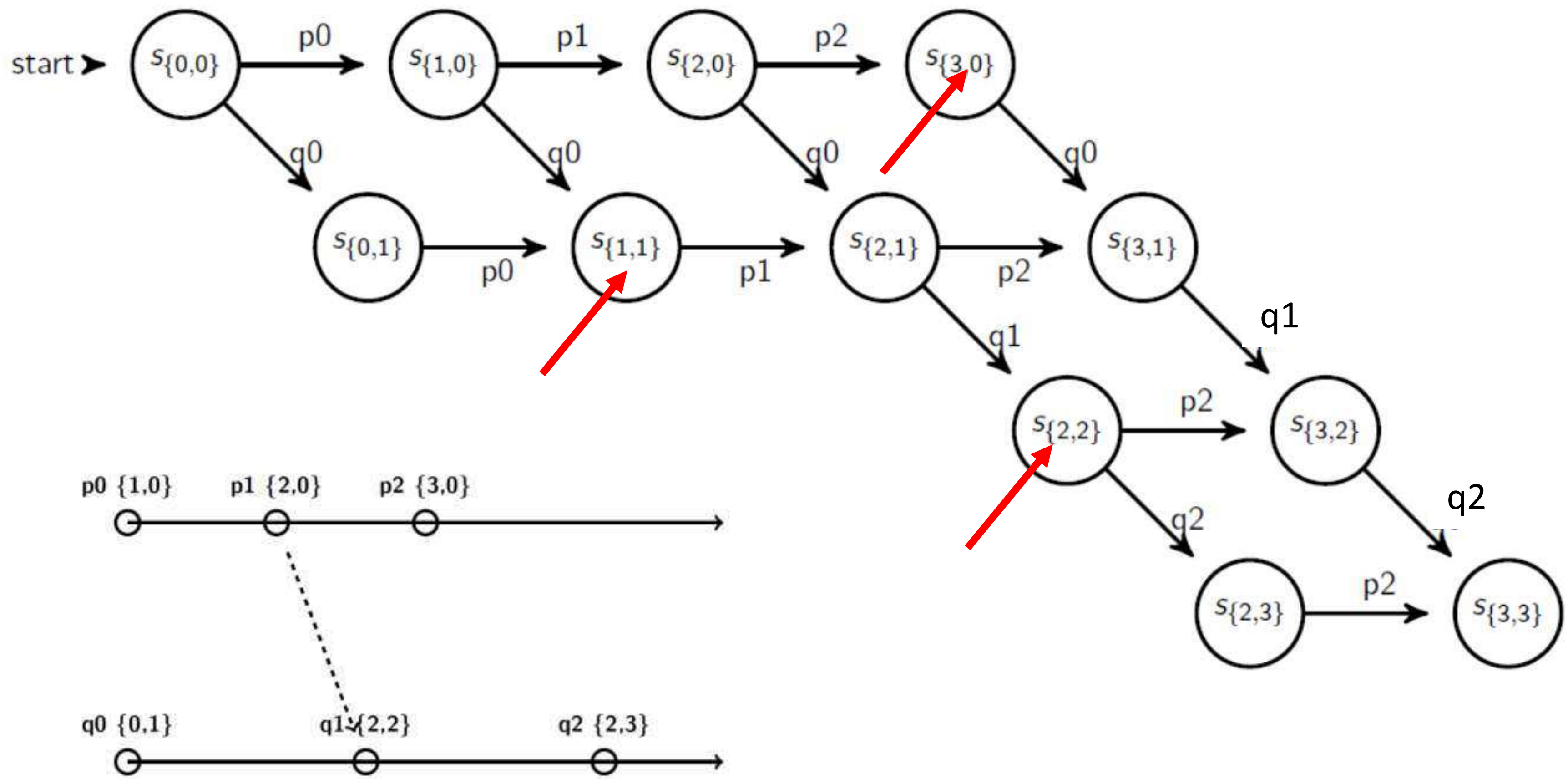
If predicate is true only in the marked states, does it satisfy liveness?
Yes



Liveness Example

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

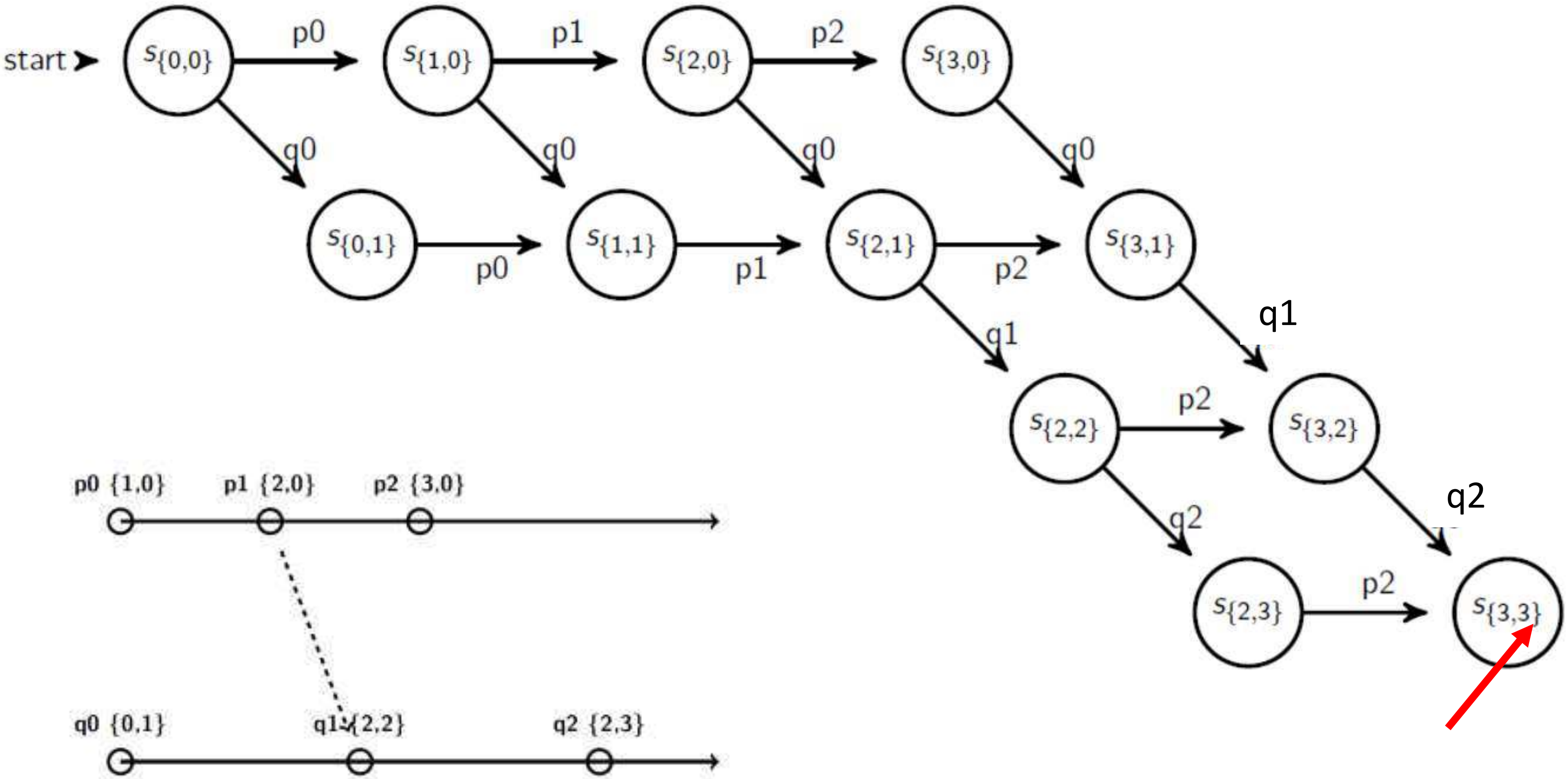
No



Liveness Example

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

Yes



Liveness

- 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)) $\equiv$ $\forall L \in$ 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.

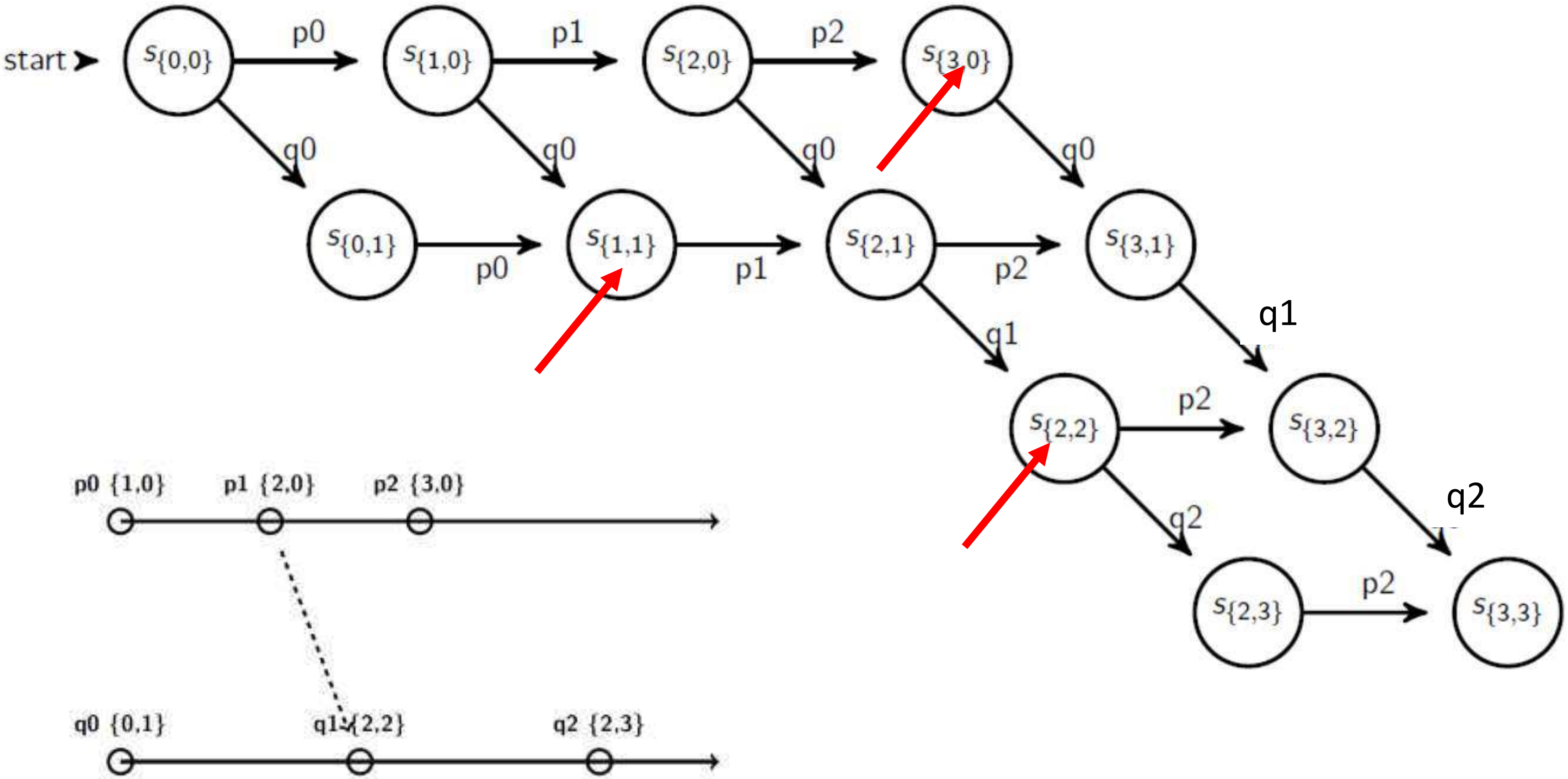
Safety

- 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:
  - $safety(P(S_0)) \equiv \forall S$ reachable from S_0, $P(S) = true$.
  - For all states S reachable from S_0, P(S) is true.

Safety Example

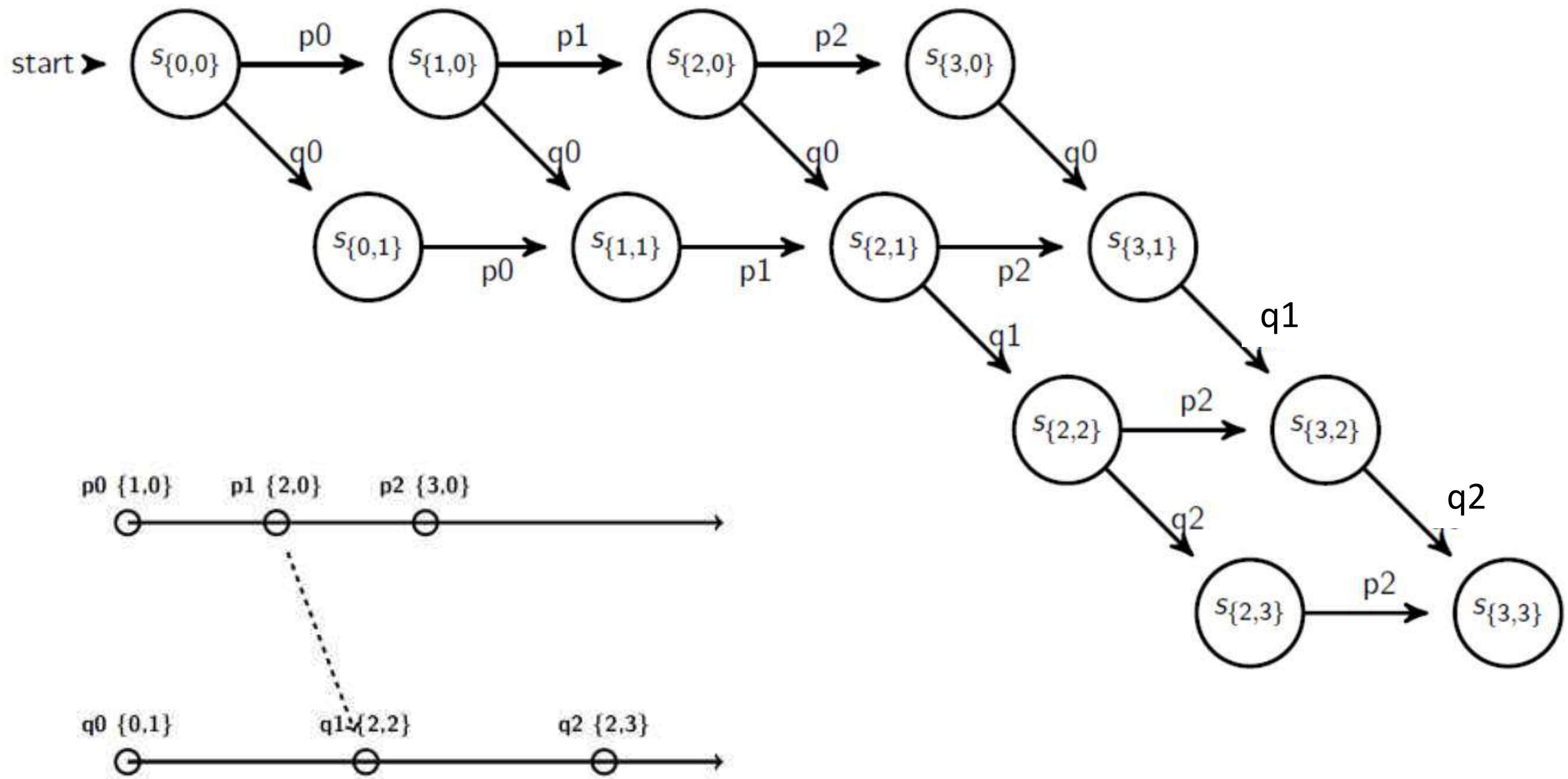
If predicate is true only in the marked states, does it satisfy safety?

No



Safety Example

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



Safety

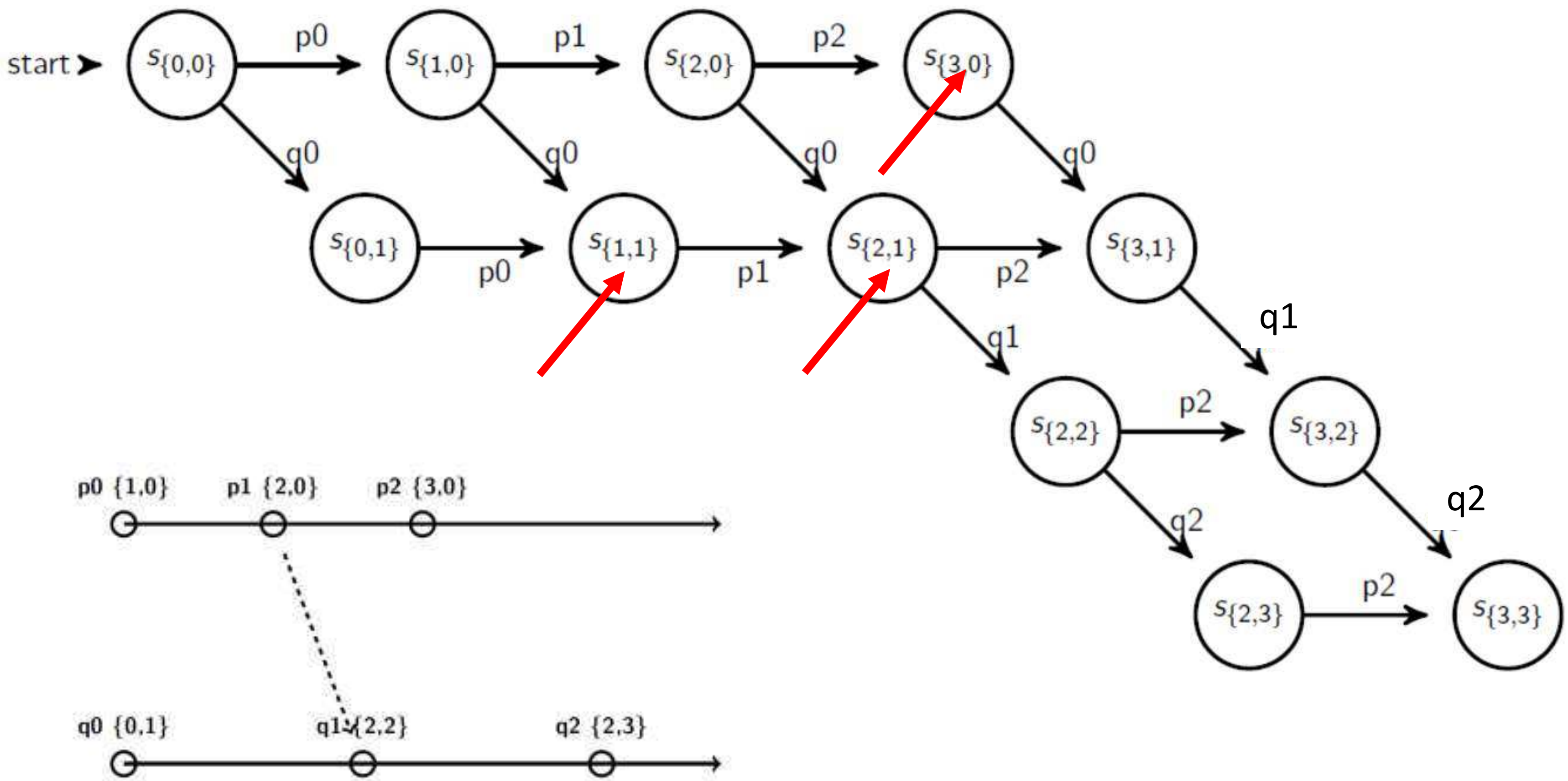
- Safety = guarantee that something bad will never happen.

- 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:
  - $\text{safety}(P(S_0)) \equiv \forall S$ reachable from S_0, $P(S) = \text{true}$.
  - For all states S reachable from S_0, P(S) is true.

Liveness Example

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



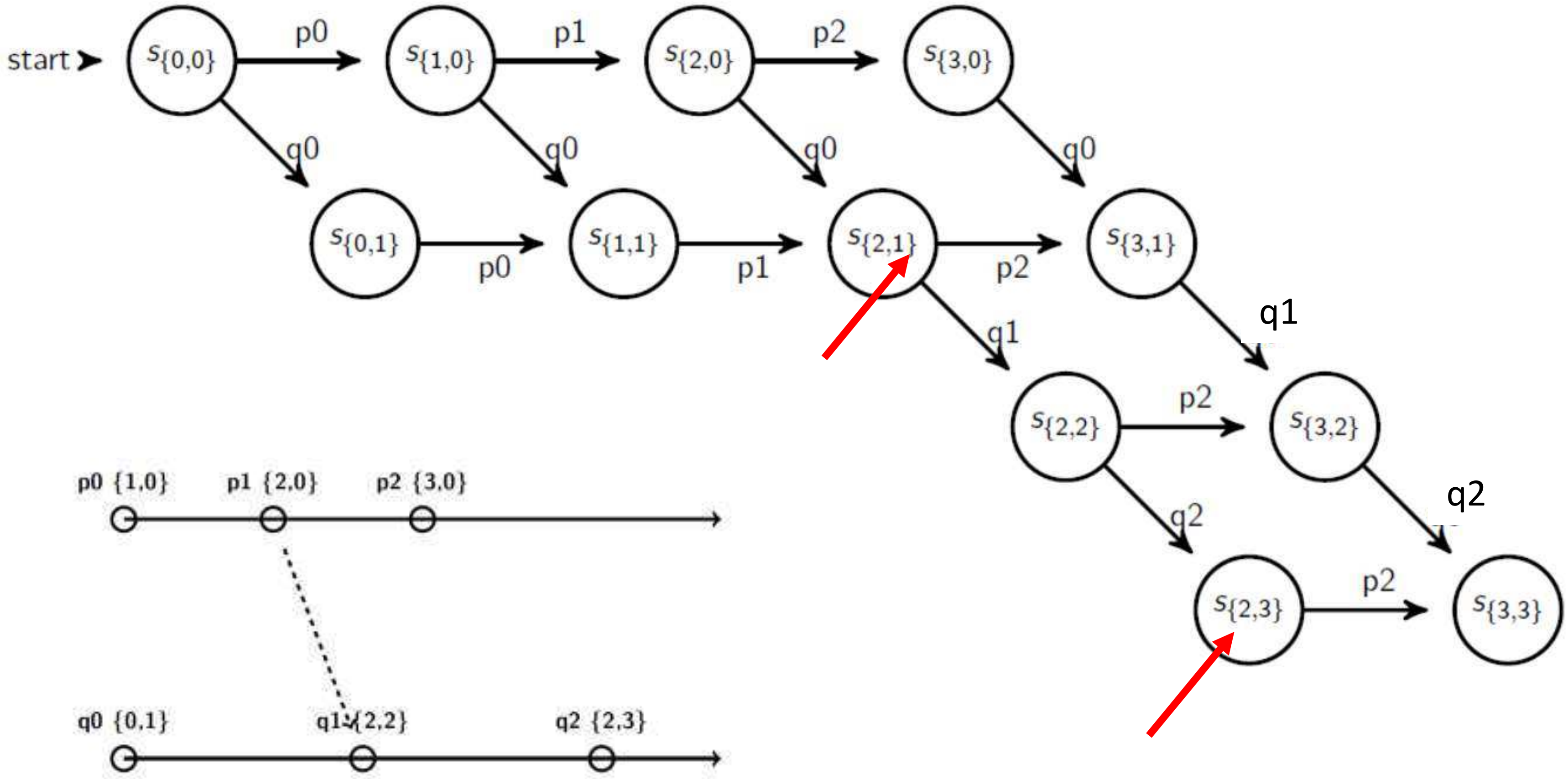
Stable Global Predicates

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

Stable Global Predicates

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

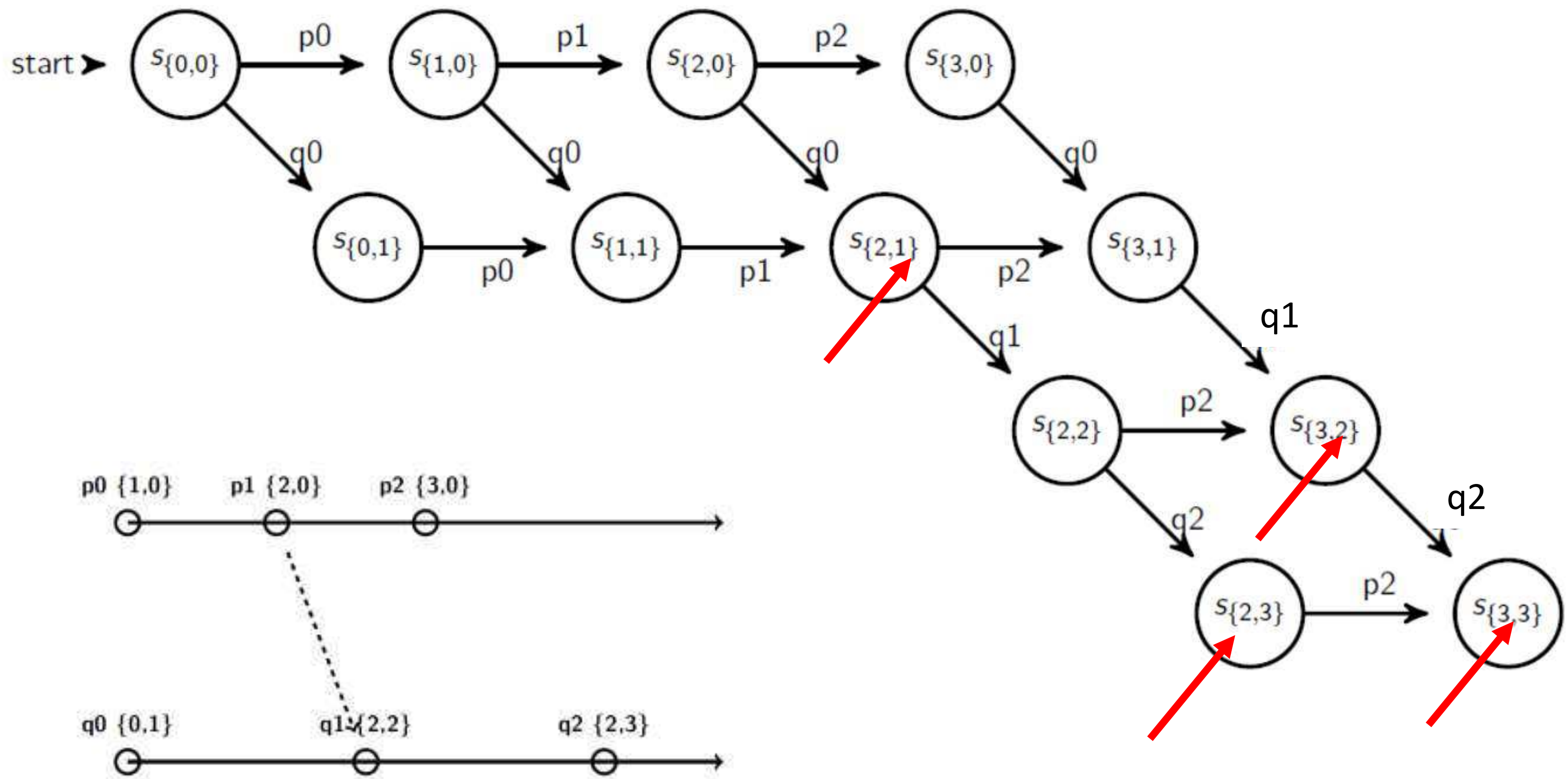
No



Stable Global Predicates

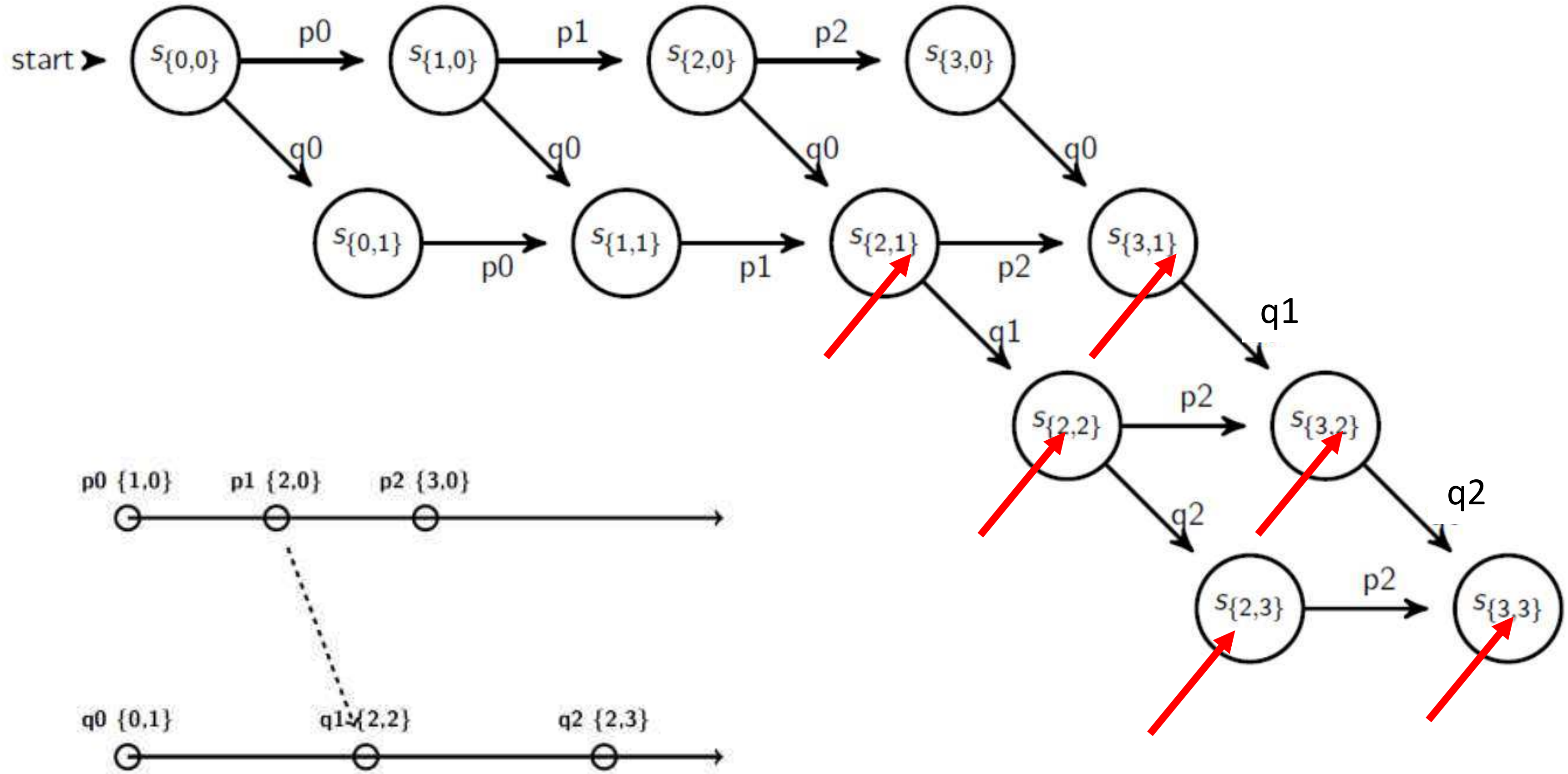
If predicate is true only in the marked states, is it stable?

No



Stable Global Predicates

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



Stable Global Predicates

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