Distributed Systems

CS425/ECE428

02/07/2020

Today’s agenda

- Wrap-up global states and snapshots
  - Chapter 14.5

- ~~Multicast~~
  - ~~Chapter 15.4~~

Recap: Timestamping events

- **Comparing timestamps across events is useful.**
  - e.g. *reconciling updates made to an object in a distributed datastore.*
  - e.g. *rollback recovery during failures.*

- **How to compare timestamps across different processes?**
  - **Physical timestamp:** requires clock synchronization.
    - e.g. Google's Spanner Distributed Database uses "TrueTime".
  - **Lamport's timestamps:** cannot fully differentiate between causal and concurrent ordering of events.
    - e.g. Oracle uses "System Change Numbers" based on Lamport's clock.
  - **Vector timestamps:** larger message sizes.
    - e.g. Amazon's DynamoDB uses vector clocks.

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* the system state.
  - Reasoning about unreferenced objects (for garbage collection).
  - Deadlock detection.
  - Distributed debugging.

Recap: Global snapshot

- State of each process (and each channel) in the system at a given instant of time.

- Difficult to capture a global snapshot of the system.
  - Requires precise clock synchronization across processes.

- *How do we capture global snapshots without precise time synchronization across processes?*
  - Relax the requirement for capturing the state of different processes and channels at the same real time instant.
  - As long as the global state is *consistent*, it is still useful.

Recap: more notations and definitions

- For a process $\mathbf{p_i}$, where events $\mathbf{e_i^0, e_i^1}$, … occur:

  history(p_i) = h_i = <$e_i^0, e_i^1, \ldots$ >

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

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

- For a set of processes <$p_1, p_2, p_3, \ldots, p_n$>:

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

  a **cut** $C \subseteq H = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_3}$

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

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

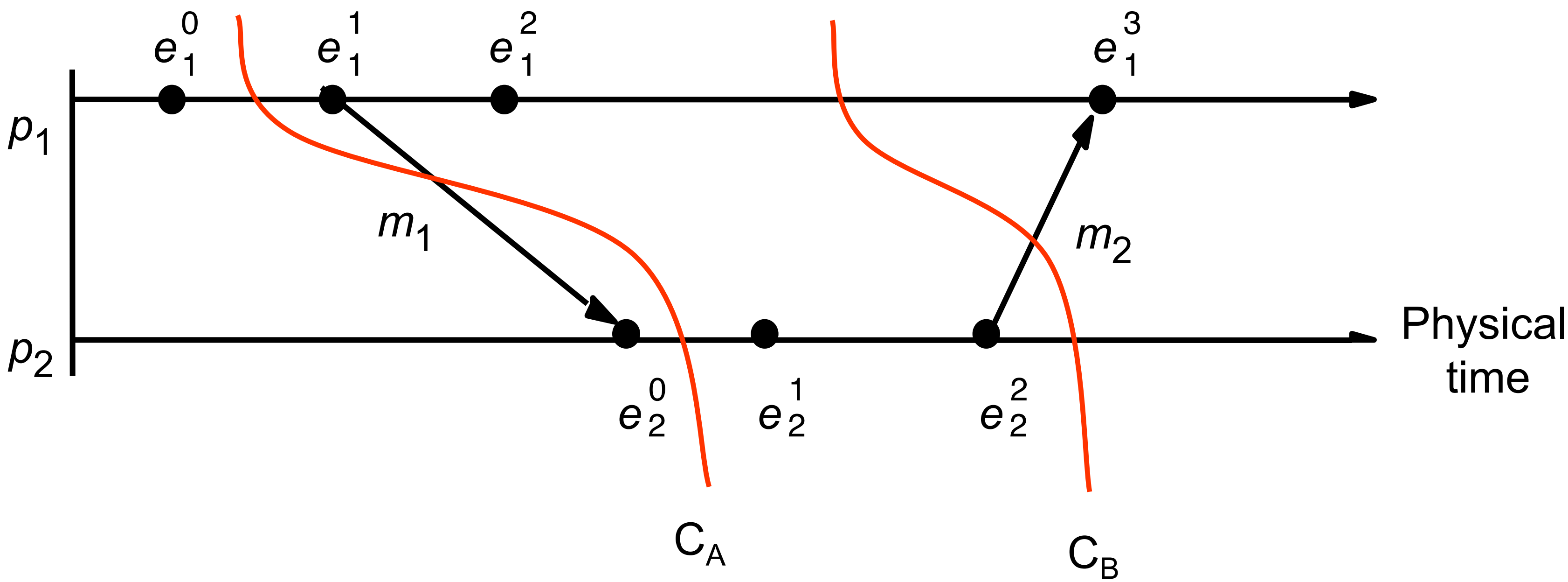
Recap: consistent cuts and snapshots

- A cut **C** is **consistent** if and only if

$$\forall \mathbf{e} \in \mathbf{C} \; (\text{if } \mathbf{f} \rightarrow \mathbf{e} \text{ then } \mathbf{f} \in \mathbf{C})$$

- A global state **S** is consistent if and only if it corresponds to a consistent cut.

Recap: Example: Cut



C_A: < e_1^0, e_2^0>
Frontier of C_A: {e_1^0, e_2^0}
Inconsistent cut.

C_B: < e_1^0, e_1^1, e_1^2, e_2^0, e_2^1 e_2^2 >
Frontier of C_B: {e_1^2, e_2^2}
Consistent cut.

Recap: Consistent cuts and snapshots

- A cut **C** is **consistent** if and only if

$$\forall \mathbf{e} \in \mathbf{C} \text{ (if } \mathbf{f} \rightarrow \mathbf{e} \text{ then } \mathbf{f} \in \mathbf{C}\text{)}$$

- A global state **S** is consistent if and only if it corresponds to a consistent cut.

- *How do we find consistent global states?*

Recap: Chandy-Lamport Algorithm

- Goal:
  - Record a global snapshot
    - Set of process state (and channel state) for a set of processes.
  - The recorded global state is consistent.
- Identifies a consistent cut.
- Records corresponding state locally at each process.

Recap: Chandy-Lamport Algorithm

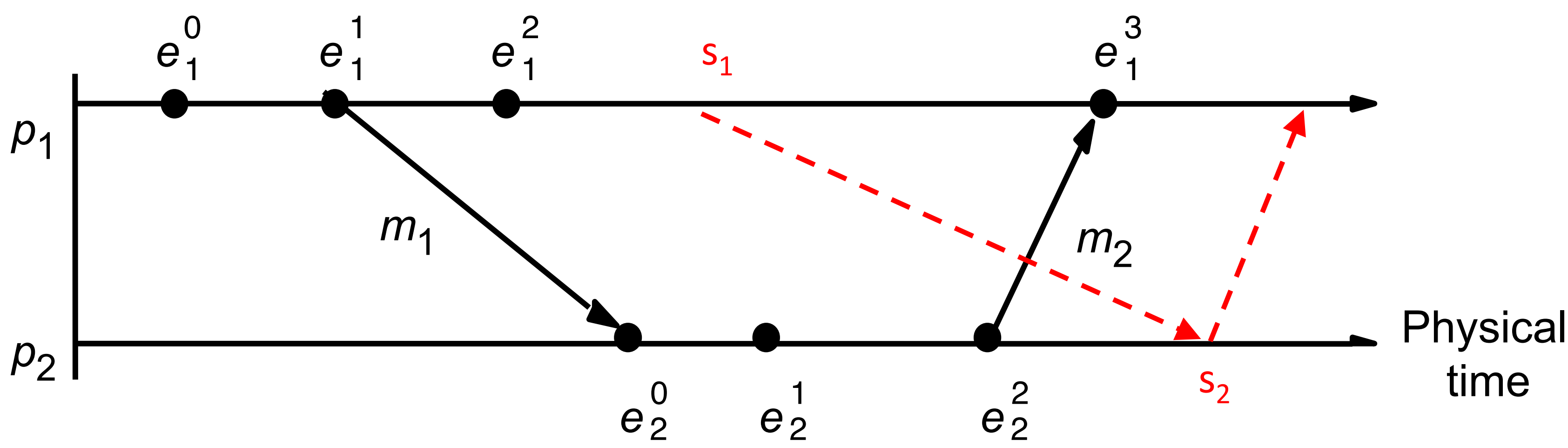
- *System model and assumptions:*
  - System of **n** processes: $<p_1, p_2, p_3, \ldots., p_n>$.
  - There are two uni-directional communication channels between each ordered process pair : p_j to p_i and p_i to p_j.
  - Communication channels are FIFO-ordered (first in first out).
  - All messages arrive intact, and are not duplicated.
  - No failures: neither channel nor processes fail.
- *Requirements:*
  - Snapshot should not interfere with normal application actions, and it should not require application to stop sending messages.
  - Any process may initiate 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.

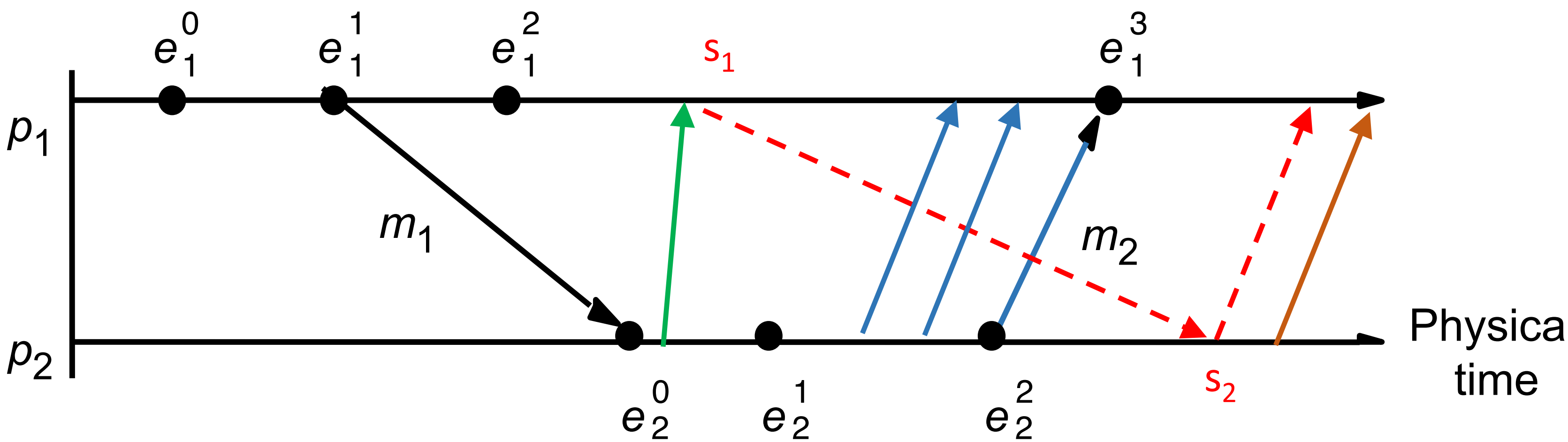
- When a process receives a **marker**.
  - records its own state.

Chandy-Lamport Algorithm Intuition



Cut frontier: $\{e_1{}^2, e_2{}^2\}$

Chandy-Lamport Algorithm Intuition



Cut frontier: $\{e_1^2, e_2^2\}$

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 on an incoming channel c_{ki}

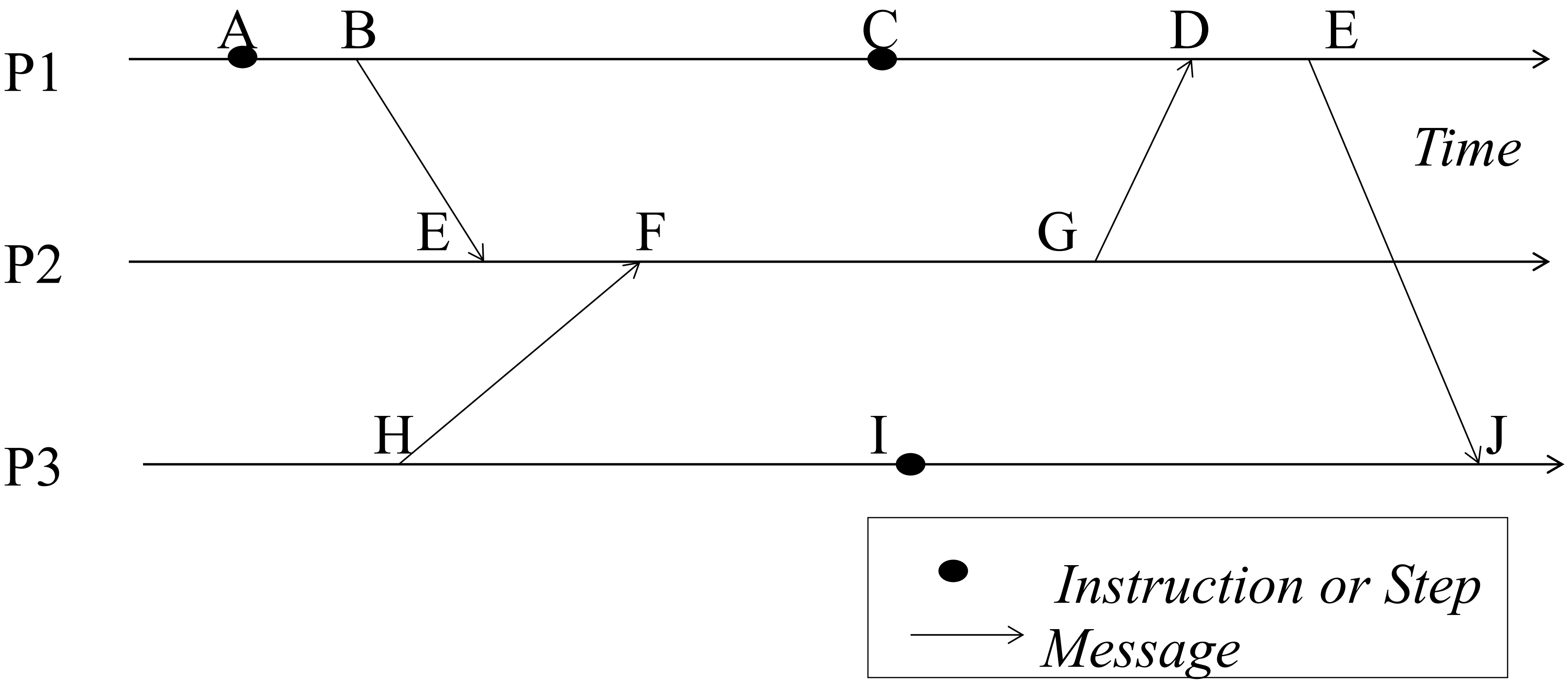
- if (this is the first **marker** p_i is seeing)
  - p_i records its own state first
  - marks the state of channel c_{ki} as "empty"
  - for j=1 to n except i
    - p_i sends out a **marker** message on outgoing channel c_{ij}
  - starts recording the incoming messages on each of the incoming channels 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 the (n-1) incoming channels
  - To record the state of all channels

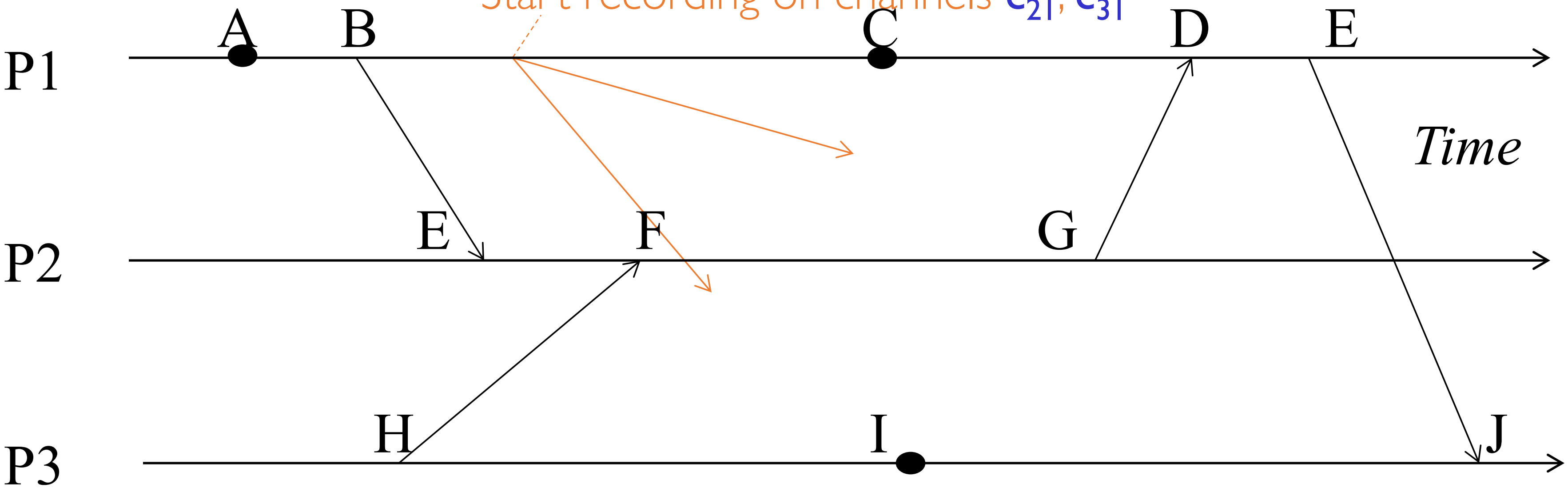
Example



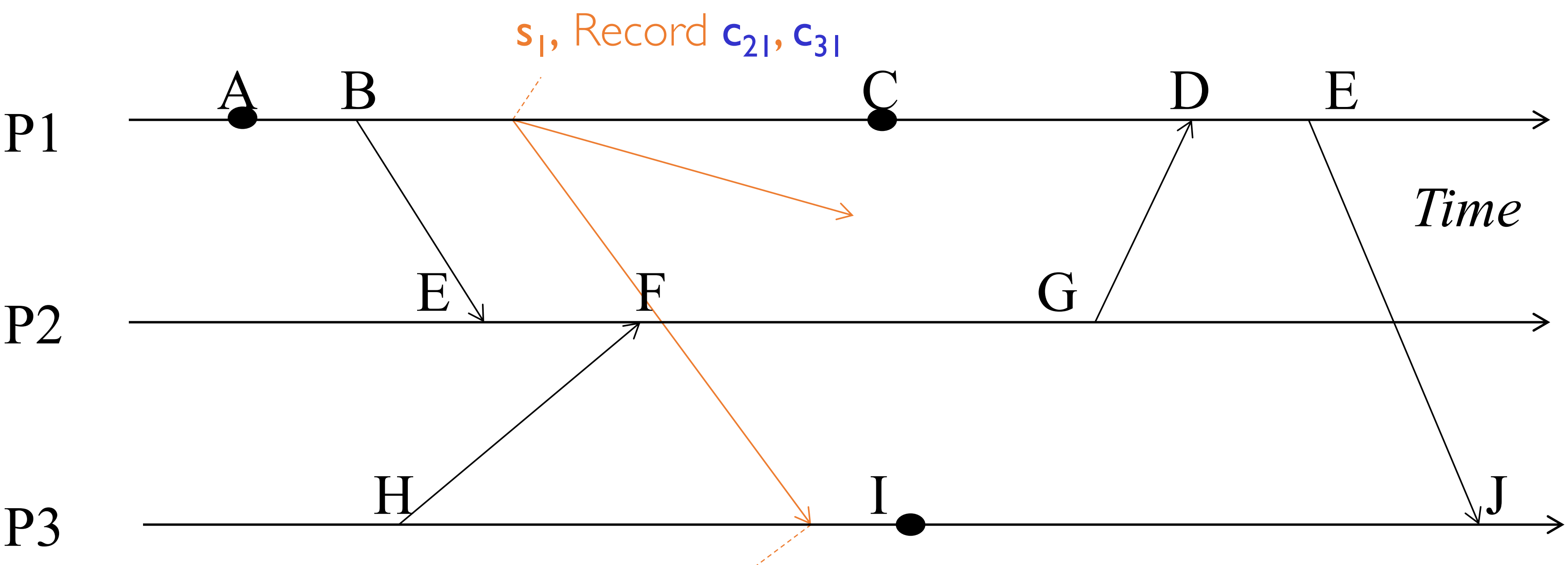
Example

p_1 is initiator:

- Record local state s_1,
- Send out markers
- Start recording on channels c_{21}, c_{31}

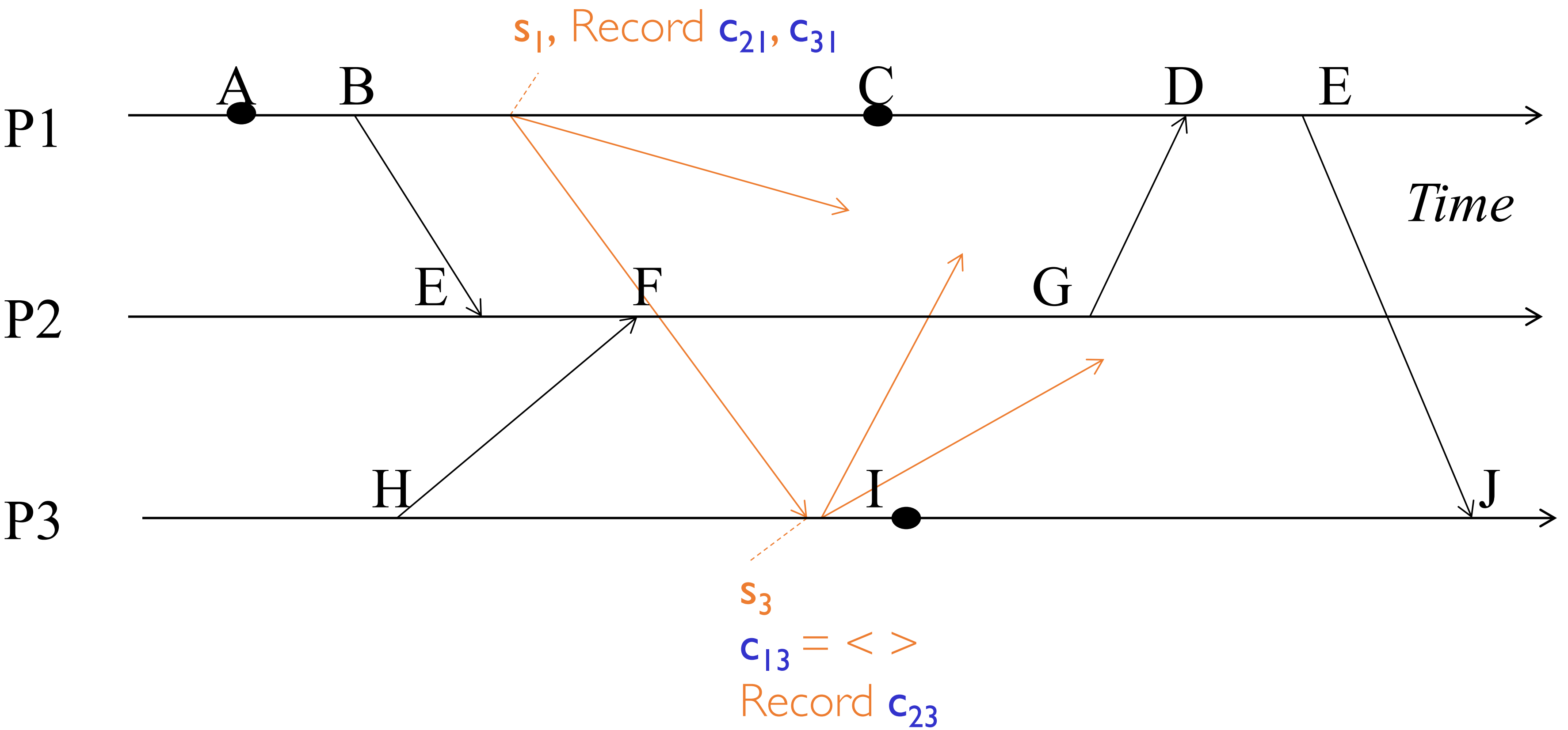


Example

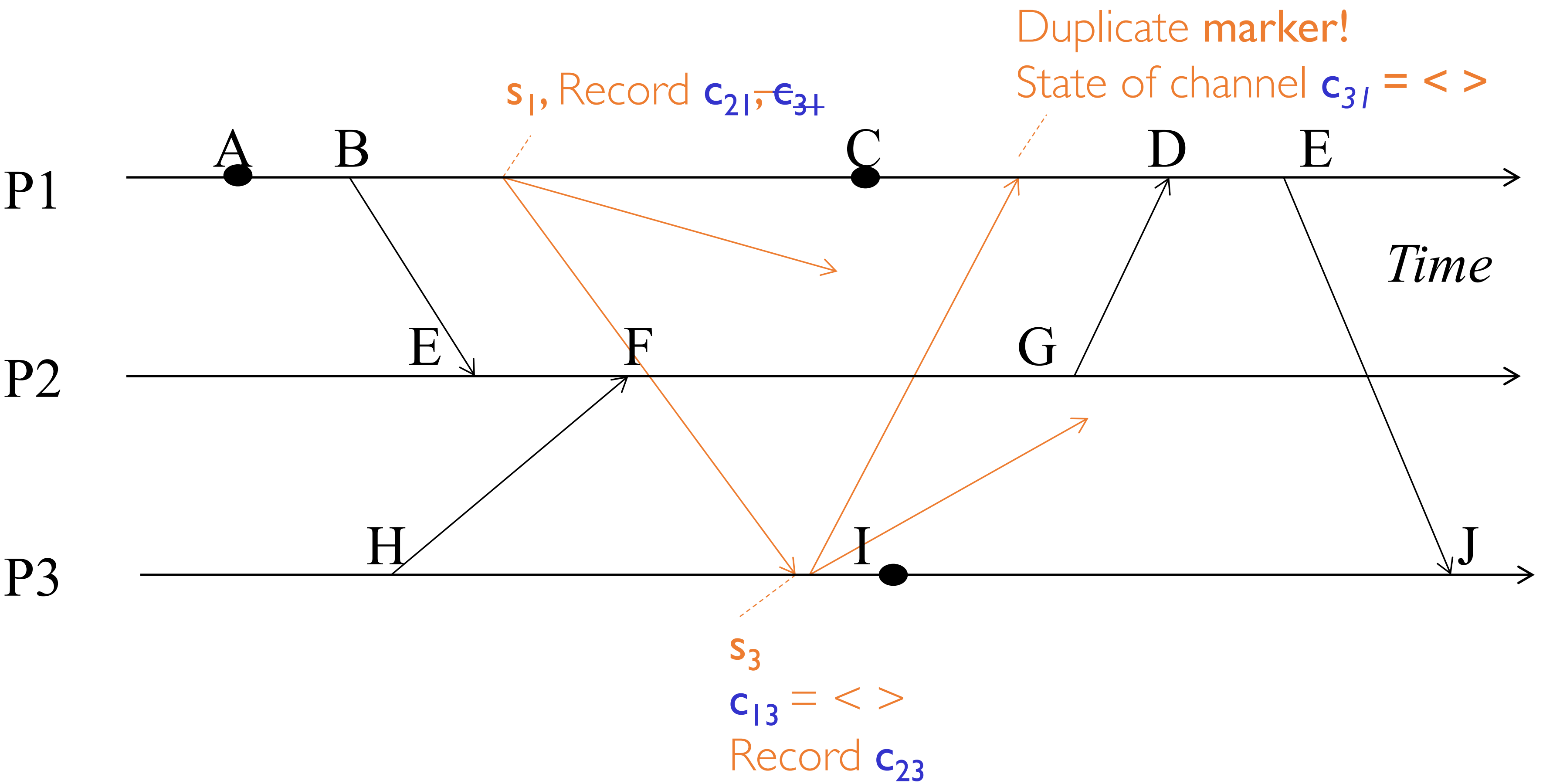


- First **marker**!
- Record own state as s_3
- Mark c_{13} state as empty
- Start recording on other incoming c_{23}
- Send out markers

Example

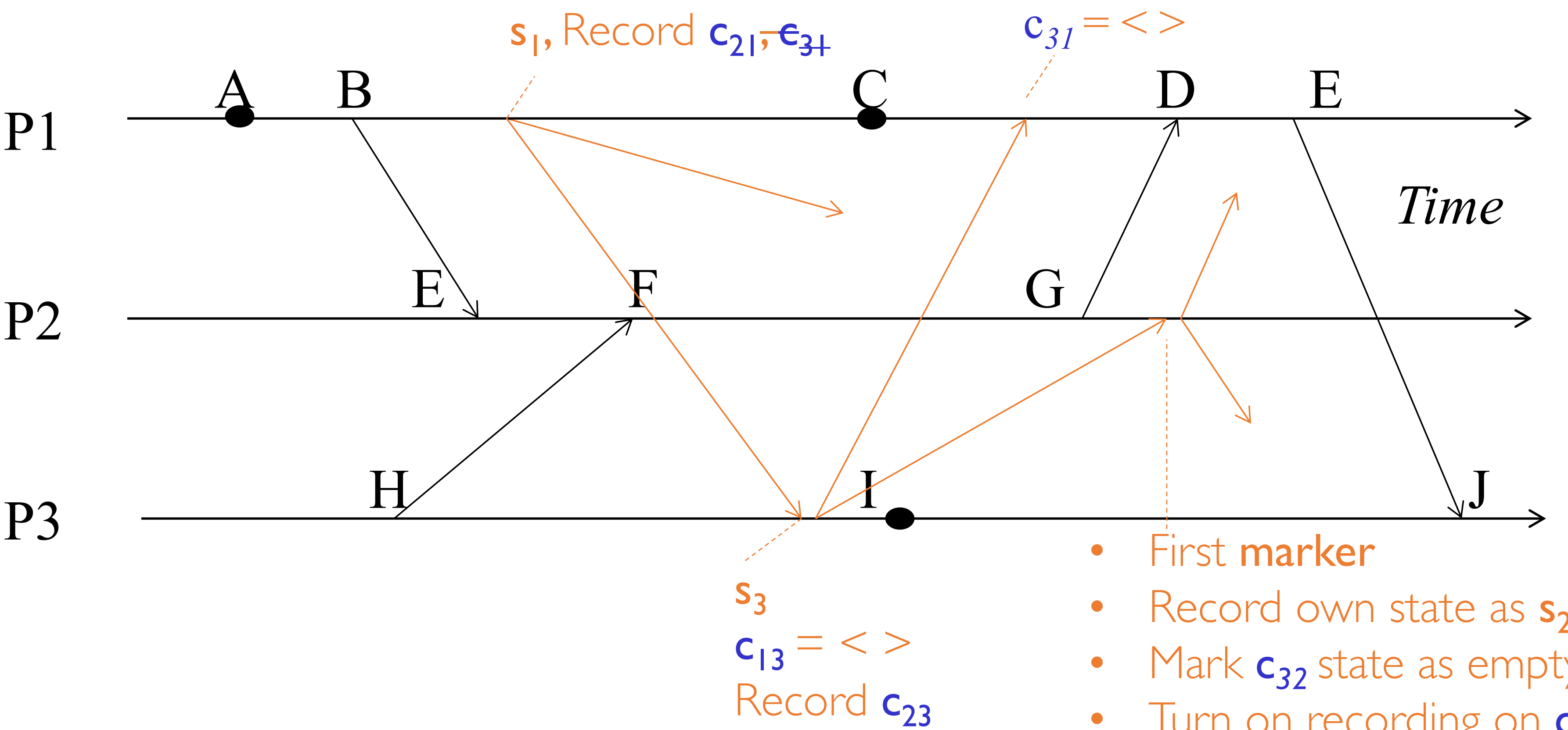


Example



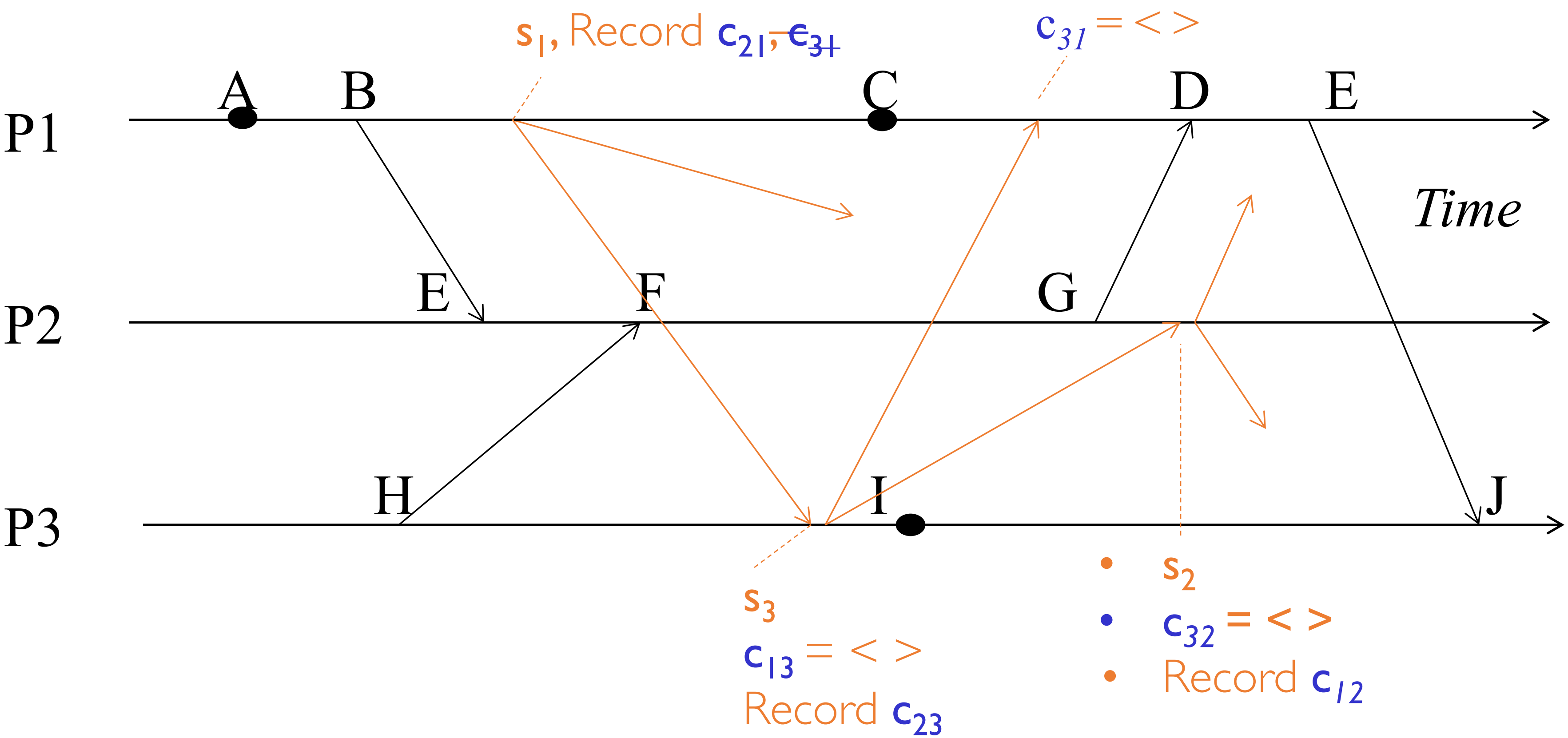
Duplicate **marker!**
State of channel c_{31} = < >
s_1, Record c_{21}, ~~c_{31}~~
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s_3
c_{13} = < >
Record c_{23}

Example



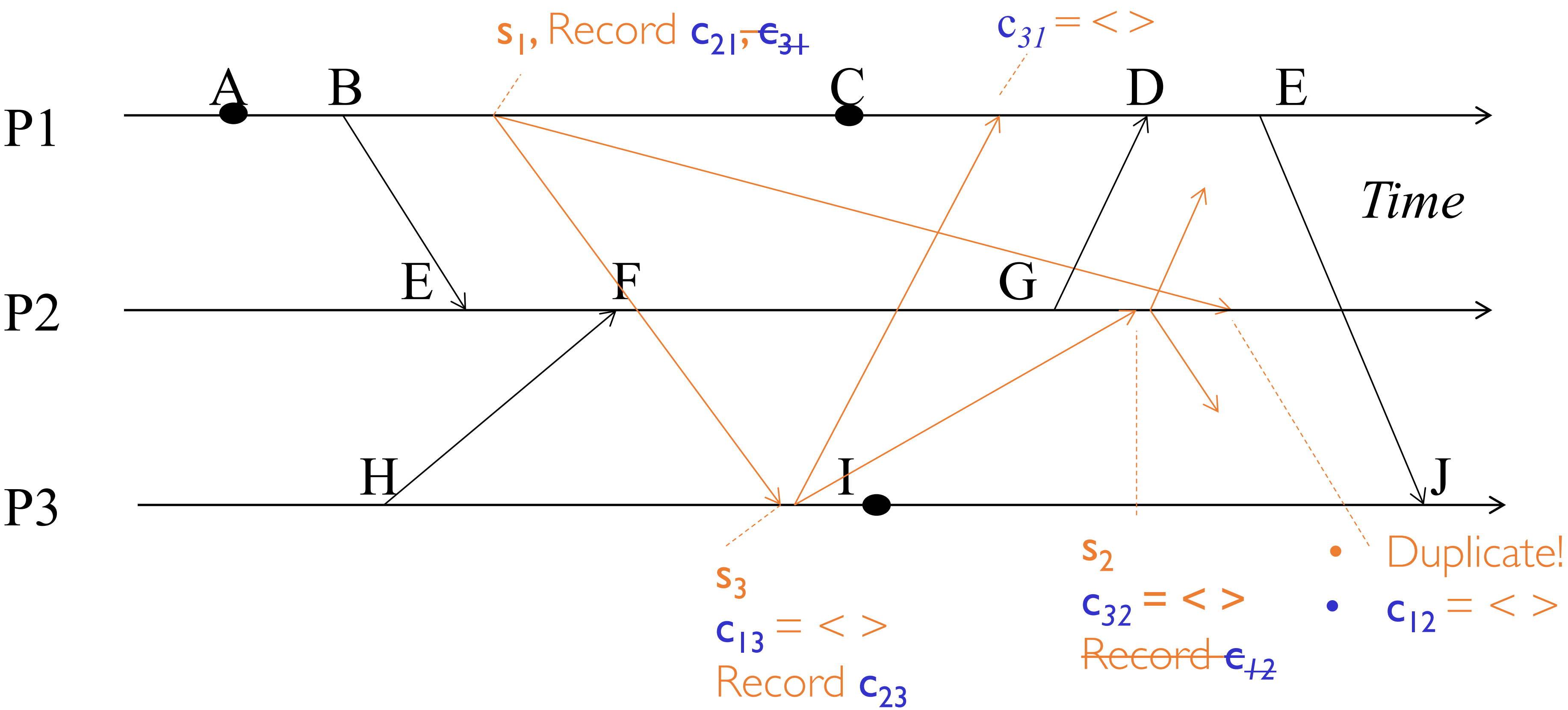
- First **marker**
- Record own state as $\mathbf{s_2}$
- Mark $\mathbf{c_{32}}$ state as empty
- Turn on recording on $\mathbf{c_{12}}$
- Send out **markers**

Example



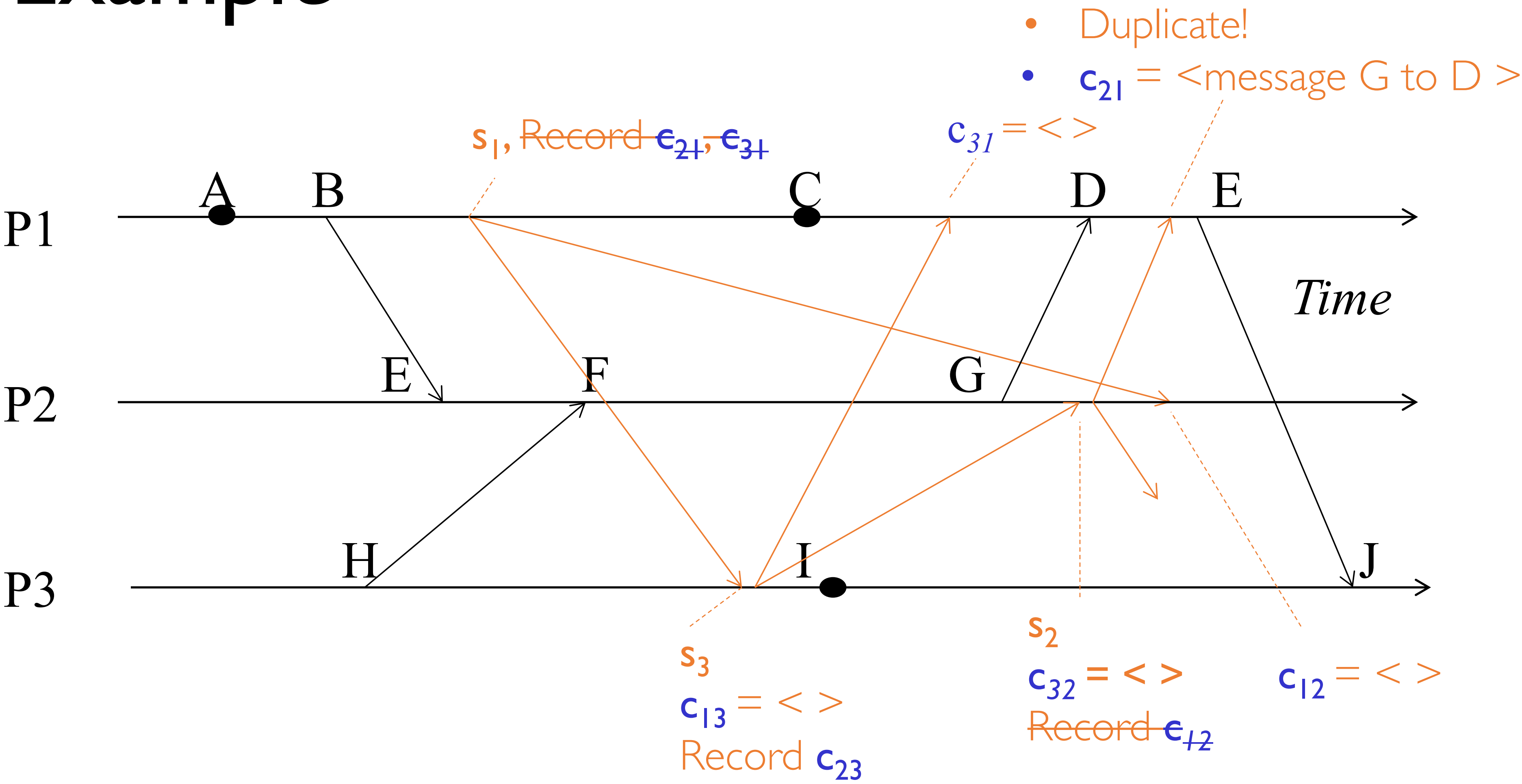
s1, Record c21, c31
c31 = < >
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s3
c13 = < >
Record c23
s2
c32 = < >
Record c12

Example



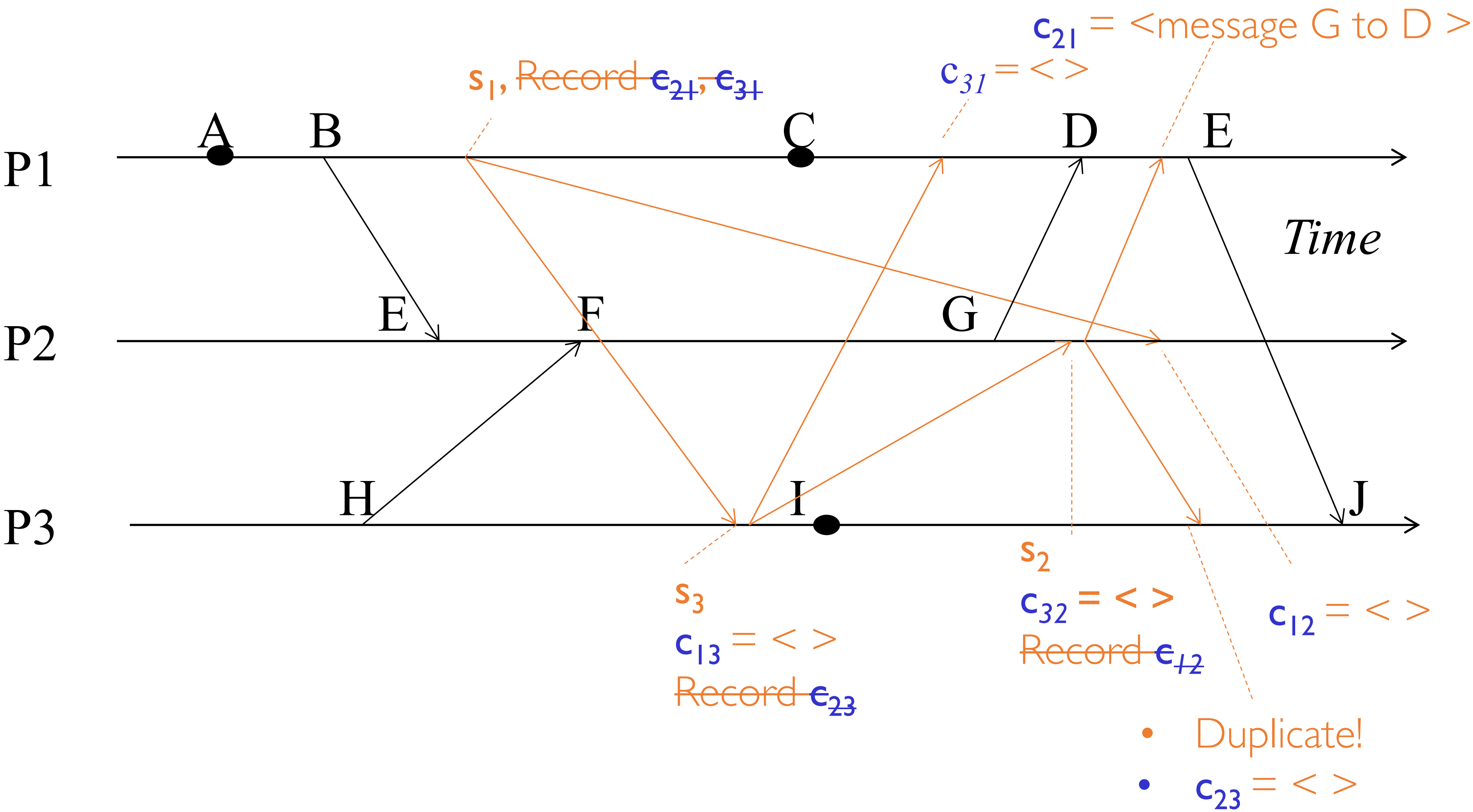
s1, Record c21, c31
c31 = < >
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s3
c13 = < >
Record c23
s2
c32 = < >
Record c12
Duplicate!
c12 = < >

Example



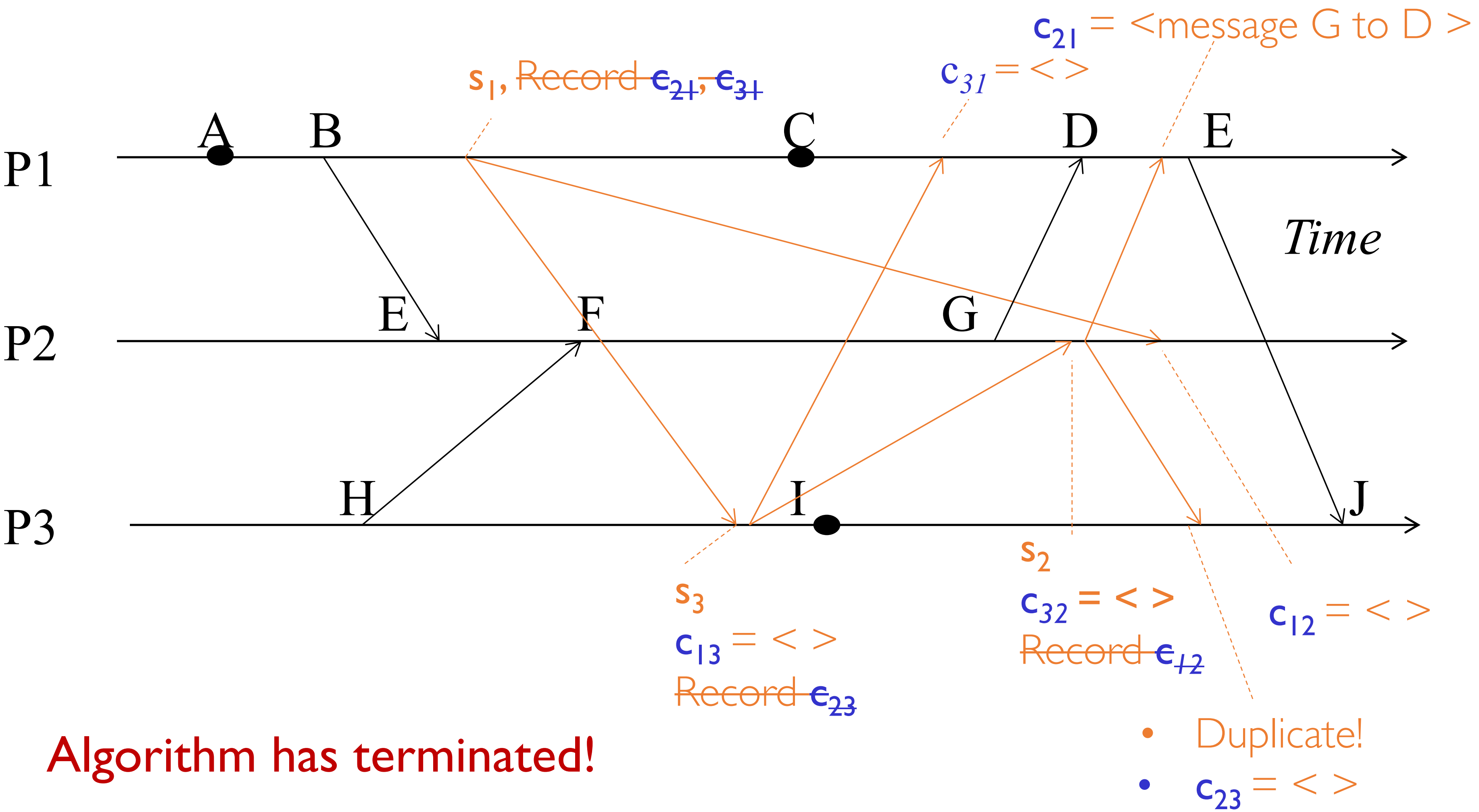
Duplicate!
c_{21} = <message G to D >
s_1, ~~Record c_{21}, c_{31}~~
c_{31} = < >
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s_3
c_{13} = < >
Record c_{23}
s_2
c_{32} = < >
~~Record c_{12}~~
c_{12} = < >

Example



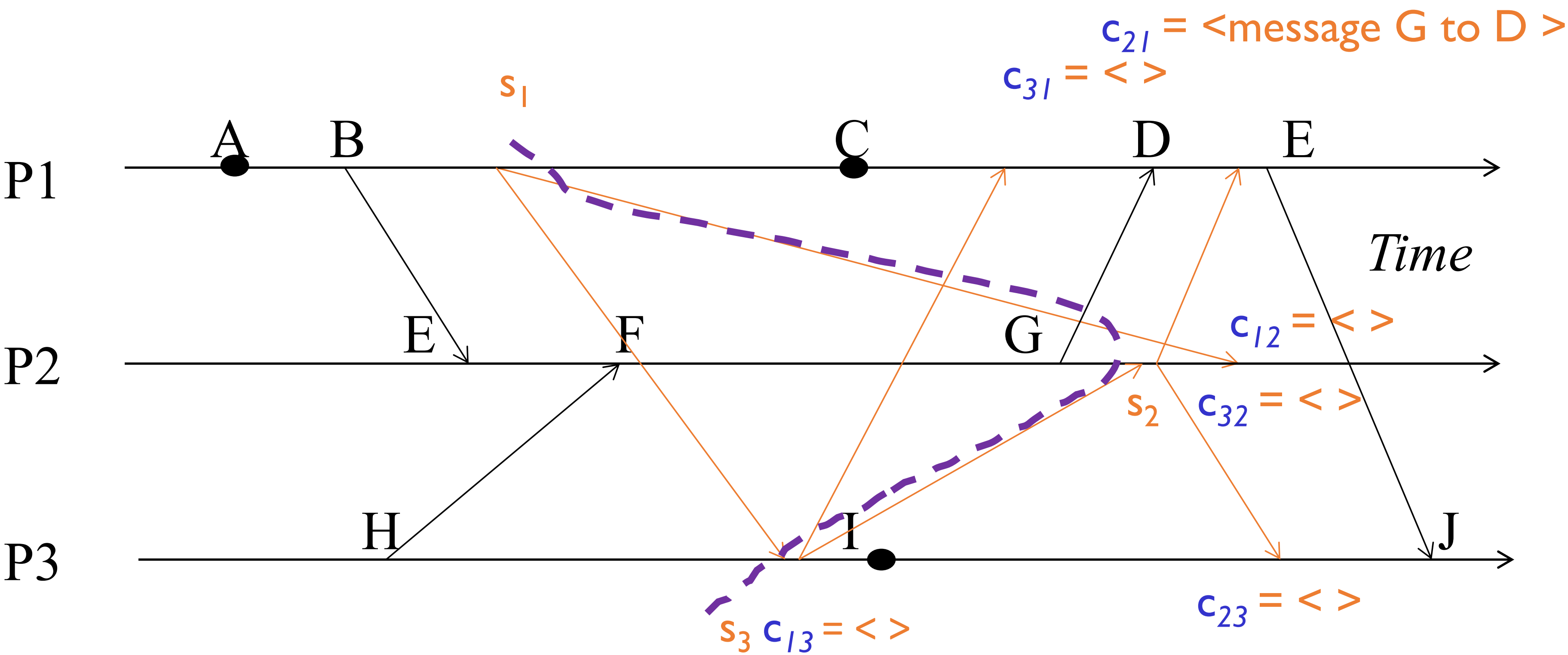
c_{21} = <message G to D >
s_1, Record c_{21}, c_{31}
c_{31} = < >
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s_3
c_{13} = < >
Record c_{23}
s_2
c_{32} = < >
Record c_{12}
c_{12} = < >
• Duplicate!
• c_{23} = < >

Example



Algorithm has terminated!

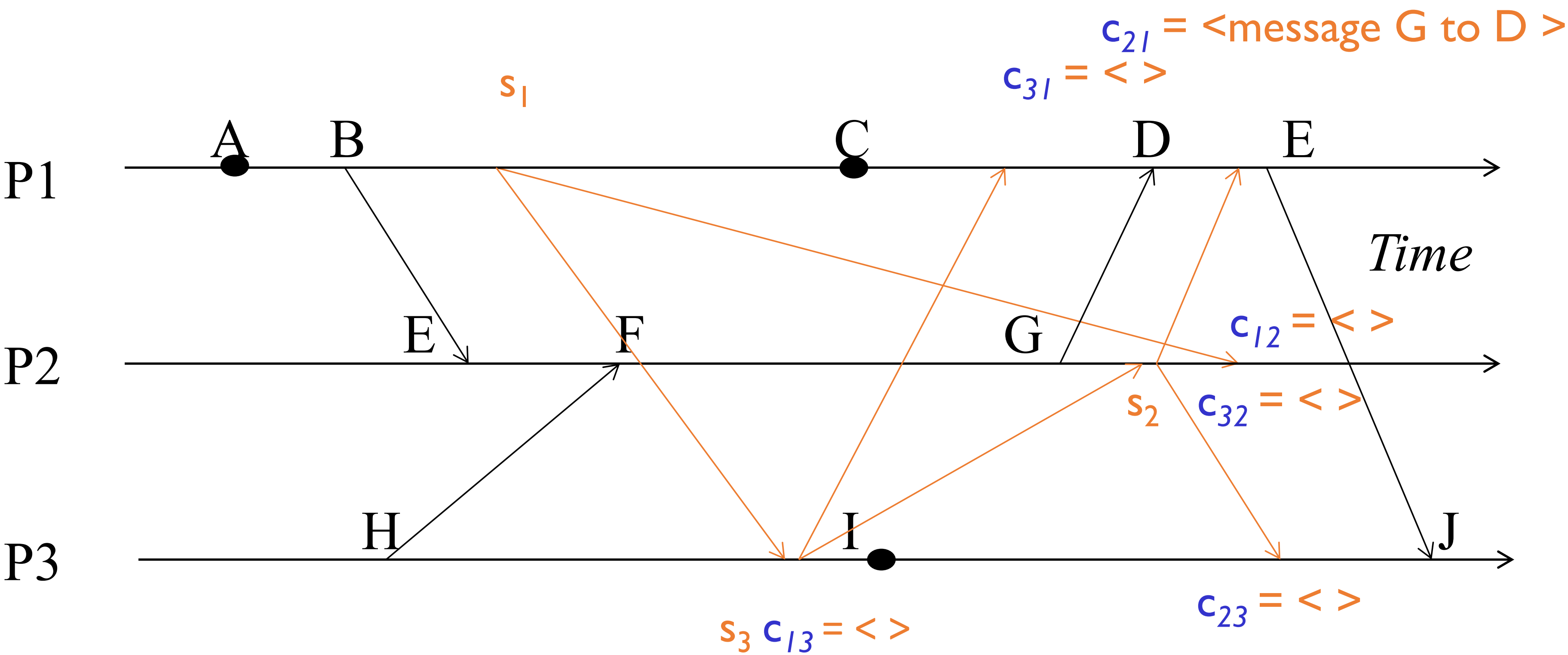
Example



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

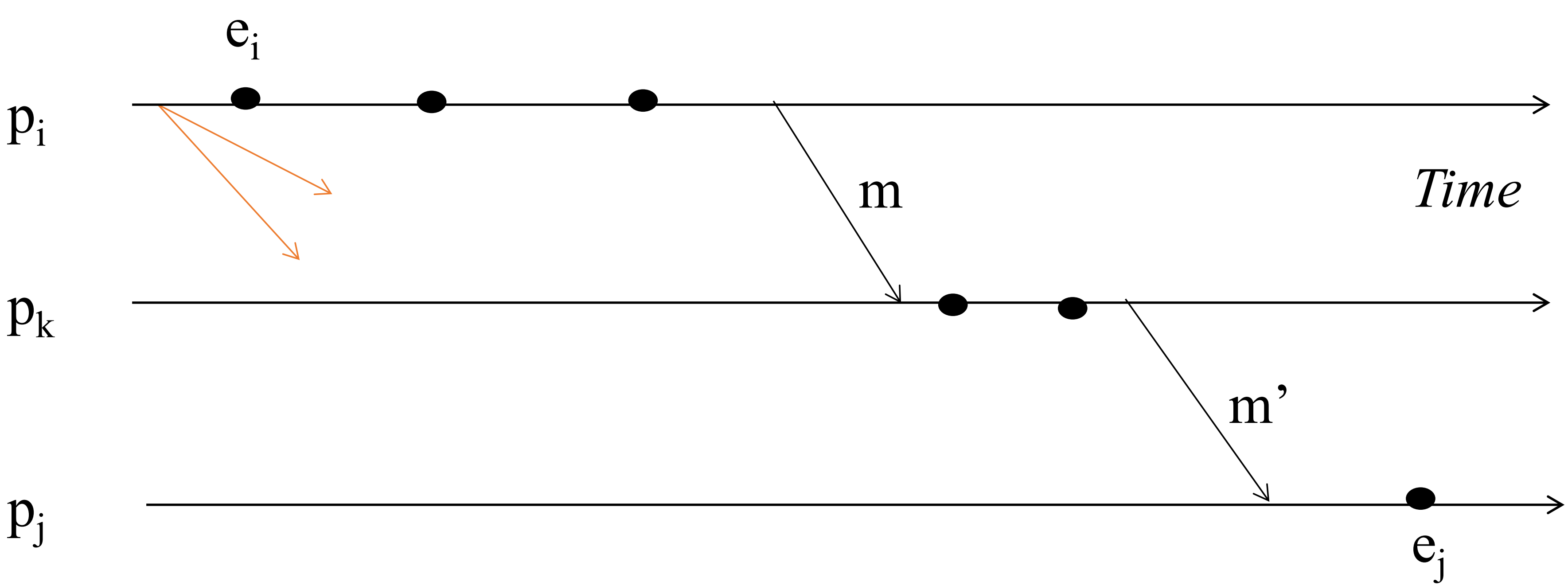
- Any run of the Chandy-Lamport Global Snapshot algorithm creates a consistent cut.
- Let $\mathbf{e}_i$ and $\mathbf{e}_j$ be events occurring at $\mathbf{p}_i$ and $\mathbf{p}_j$, respectively such that
  - $\mathbf{e}_i \rightarrow \mathbf{e}_j$ ($\mathbf{e}_i$ happens before $\mathbf{e}_j$)
- The snapshot algorithm ensures that

  if $\mathbf{e}_j$ is in the cut then $\mathbf{e}_i$ is also in the cut.

- That is: if $\mathbf{e}_j \rightarrow$ < $\mathbf{p}_j$ records its state>, then it must be true that $\mathbf{e}_i \rightarrow$ <$\mathbf{p}_i$ records its state>.

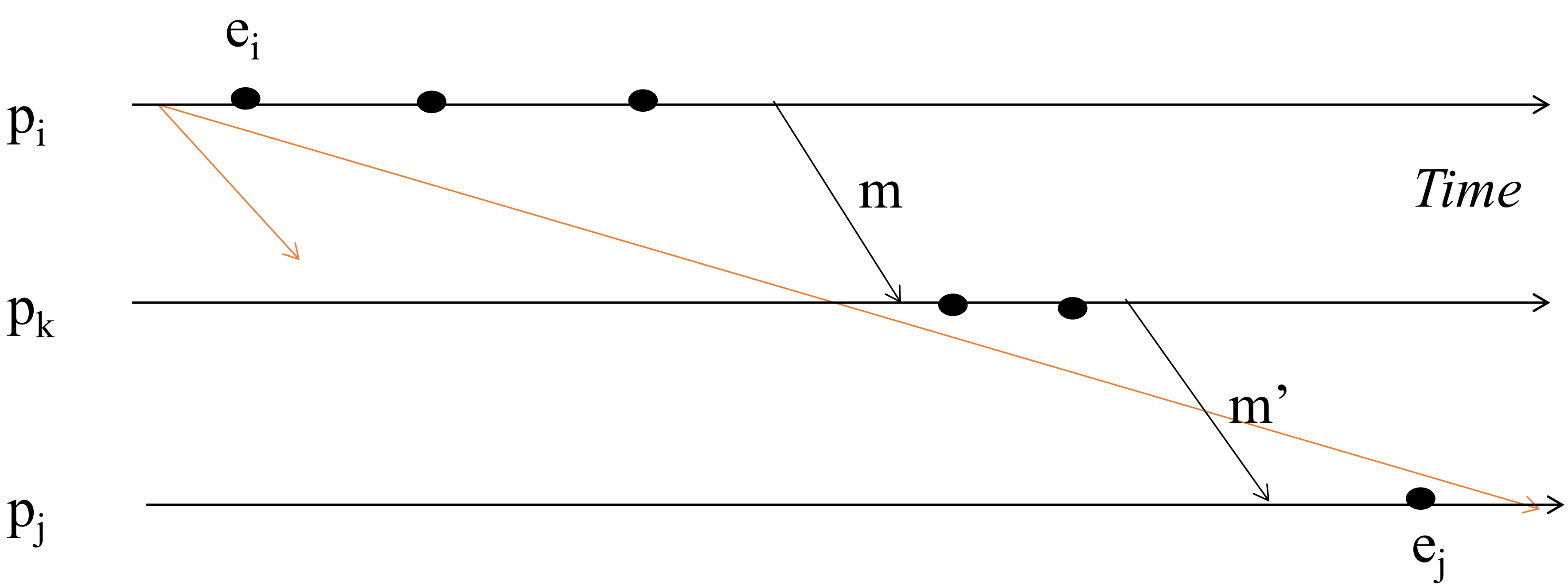
Chandy-Lamport Algorithm: Properties

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



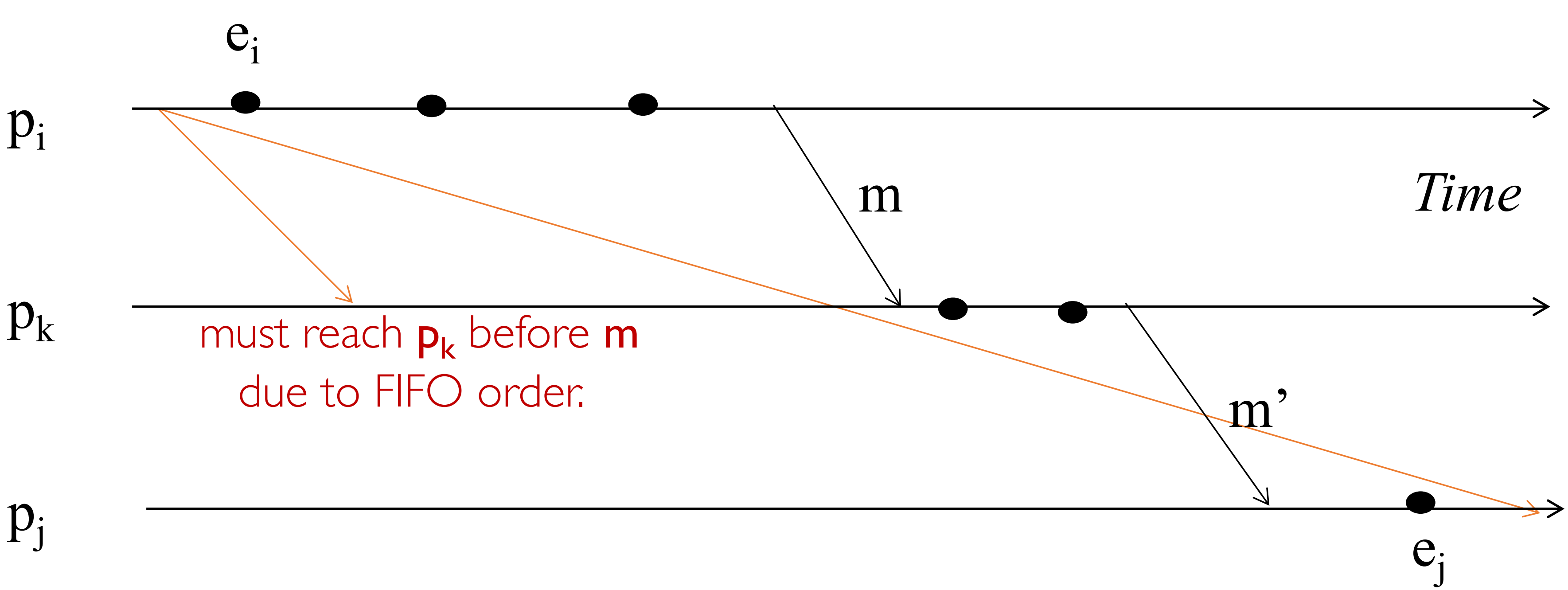
Chandy-Lamport Algorithm: Properties

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



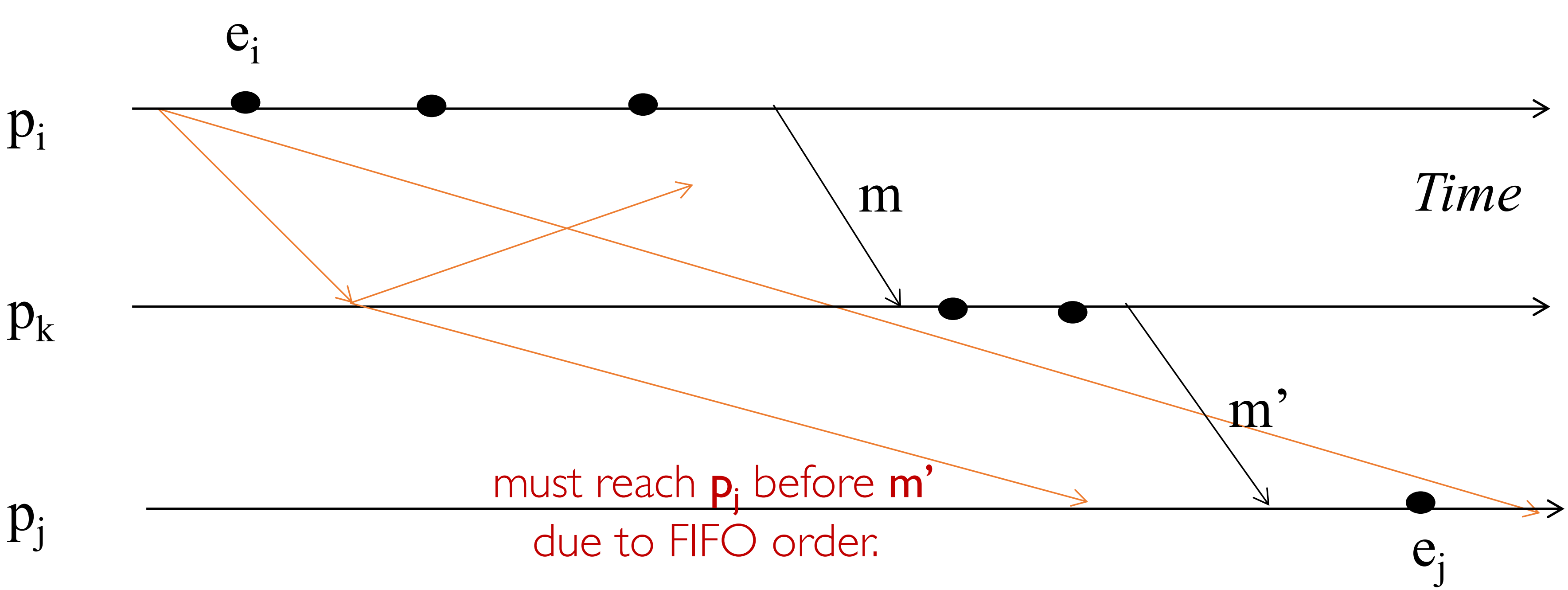
Chandy-Lamport Algorithm: Properties

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



Chandy-Lamport Algorithm: Properties

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



Chandy-Lamport Algorithm: Properties

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

Chandy-Lamport Algorithm: Usefulness

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

Revisions: notations and definitions

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

  history(p_i) = $h_i = <e_i^0, e_i^1, \ldots >$

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

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

- For a set of processes $<p_1, p_2, p_3, \ldots, p_n>$:

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

  a cut $C \subseteq H = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_3}$

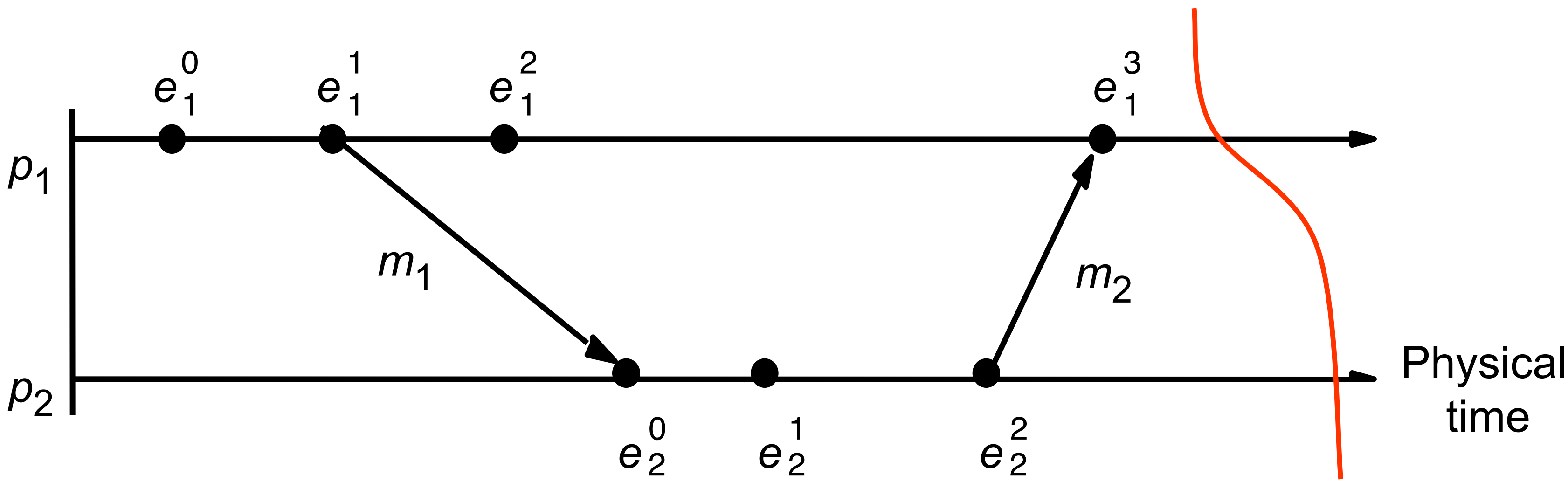
  the frontier of $C = \{e_i^{c_i}, i = 1,2, \ldots n\}$

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

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.

Example



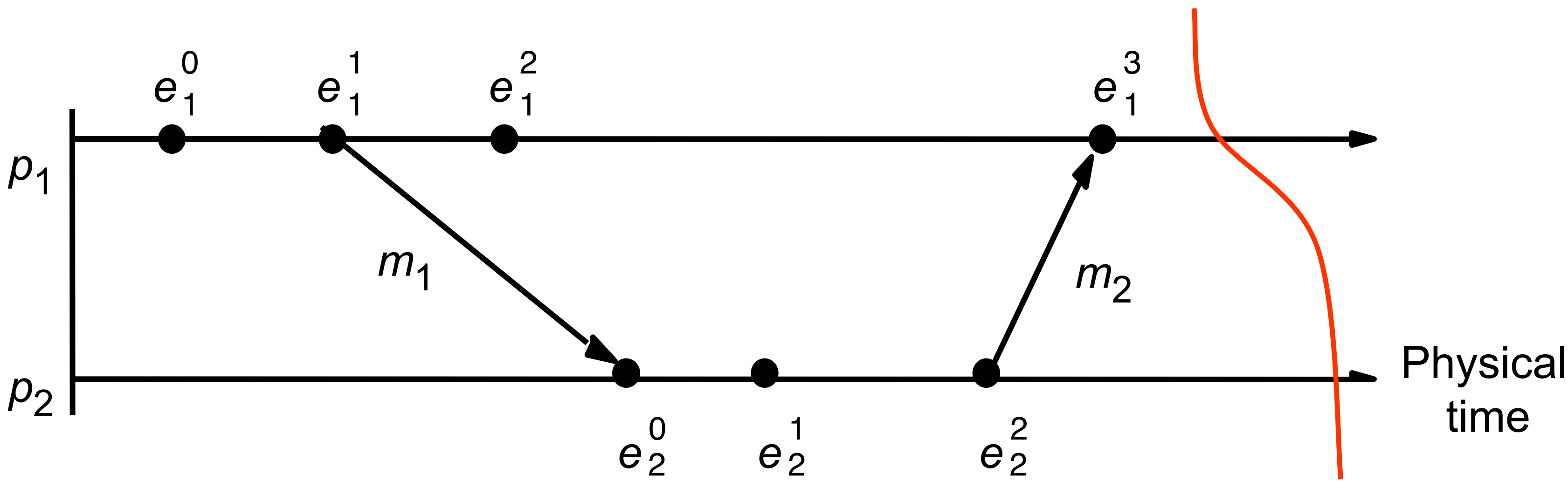
Order at p_1: $< e_1^0, e_1^1, e_1^2, e_1^3 >$ Order at p_2: $< e_2^0, e_2^1, e_2^2 >$

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



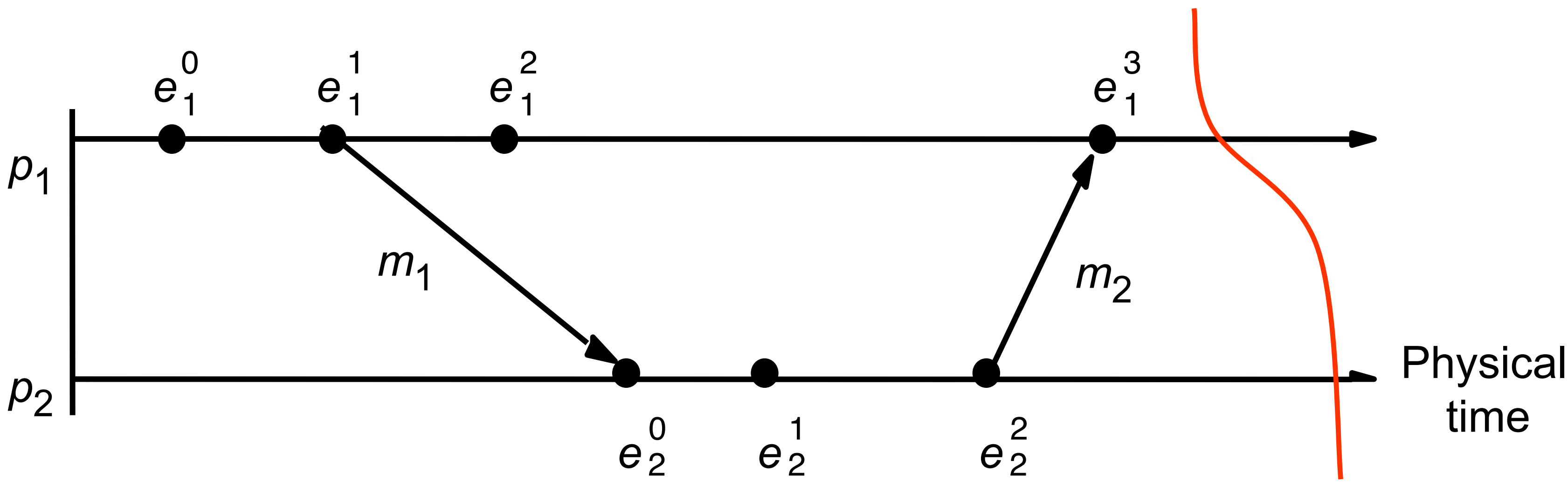
Order at p_1: < e_1^0, e_1^1, e_1^2, e_1^3 >   Order at p_2: < e_2^0, e_2^1,e_2^2>

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

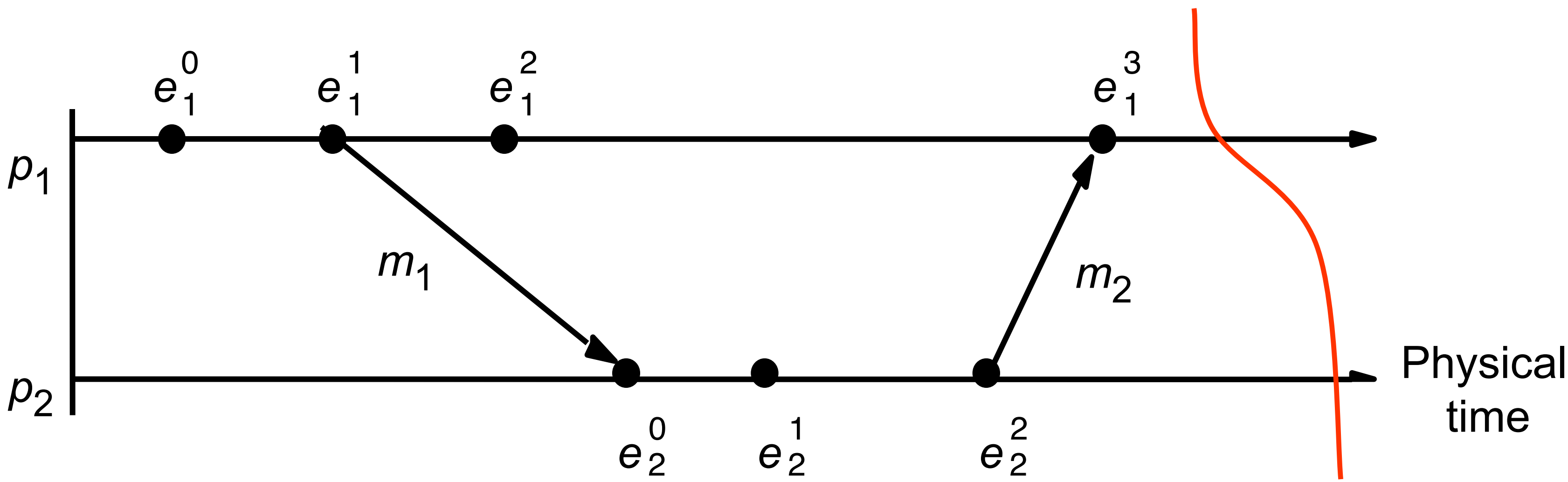
$< e_1^0, e_1^1, e_2^0, e_2^1, e_1^2, e_2^2, e_1^3 >$: Linearization

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

More notations and definitions

- 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.

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

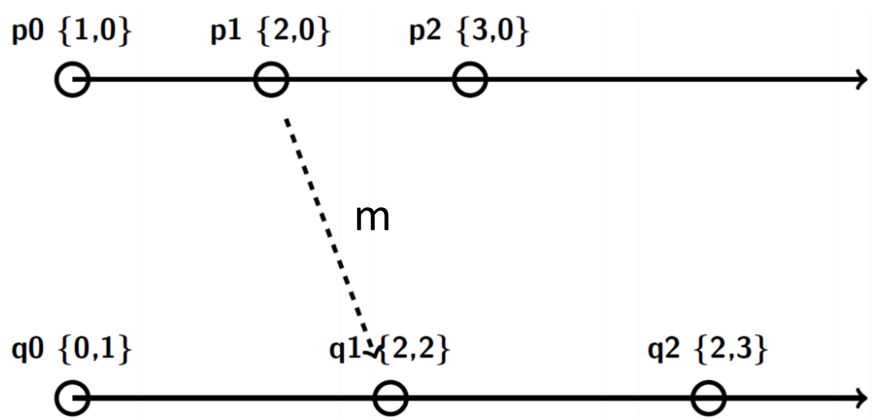
Linearization: $< e_1^0, e_1^1, e_1^2, e_2^0, e_2^1\ e_2^2, e_1^3 >$

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

- 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



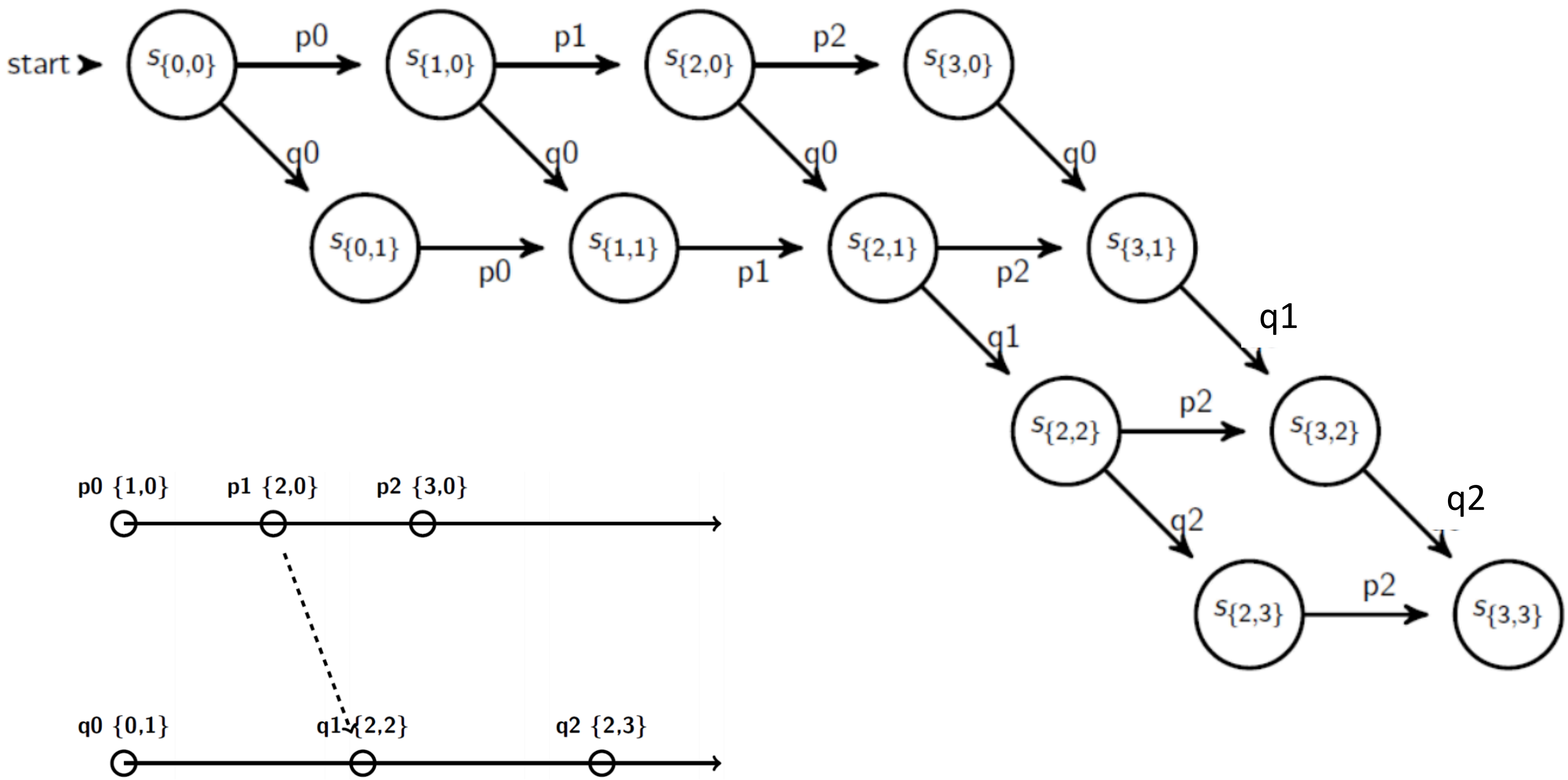
- Causal order:
  - p0 → p1 → p2
  - q0 → q1 → q2
  - p0 → p1 → q1 → q2
- Concurrent:
  - p0 || q0
  - p1 || q0
  - p2 || q0, p2 || q1, p2 || q2

Many linearizations:

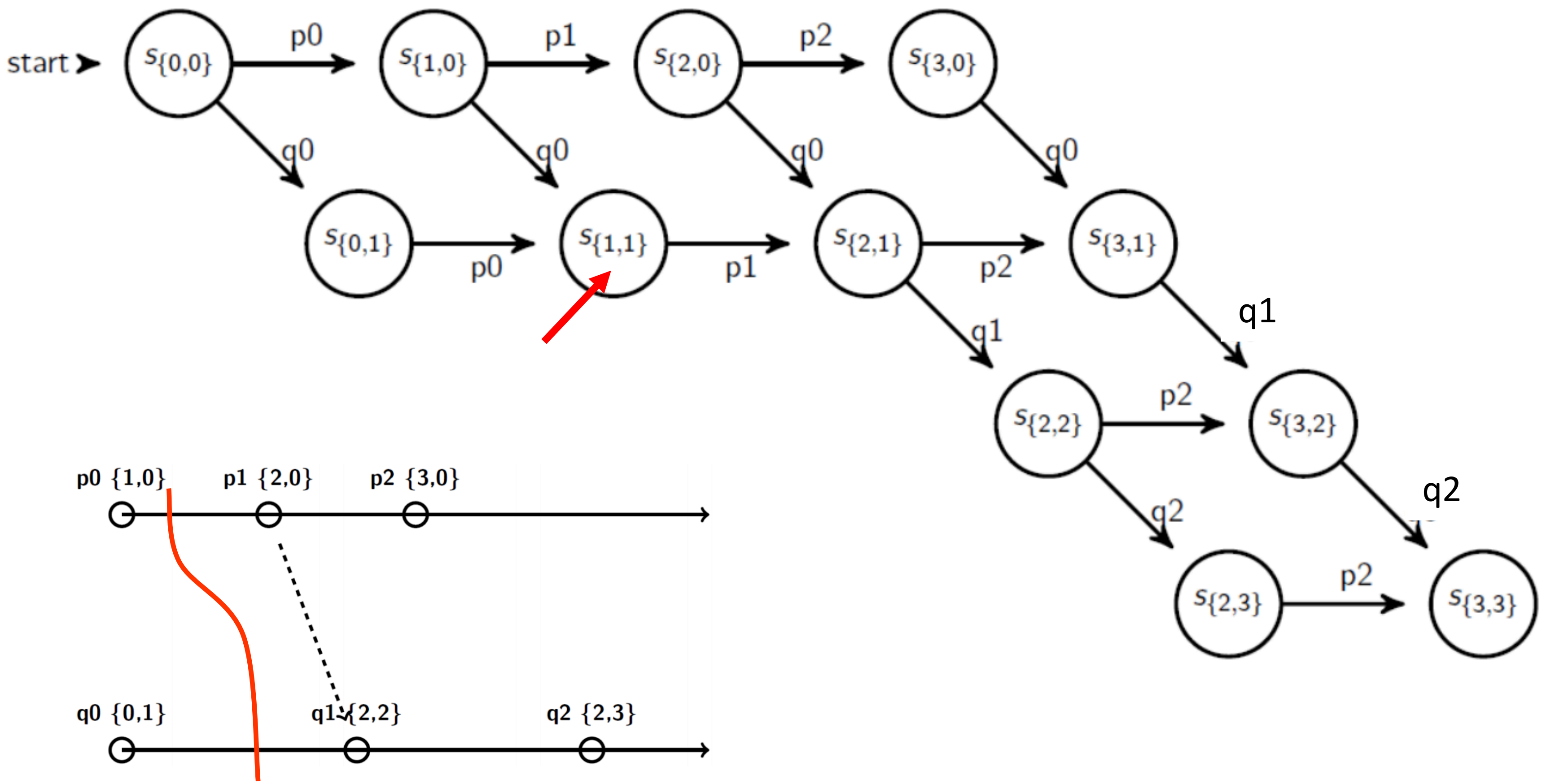
- < p0, p1, p2, q0, q1, q2>
- < p0, q0, p1, q1, p2, q2>
- <q0, p0, p1, q1, p2, q2 >
- <q0, p0, p1, p2, q1,q2 >
- ……

State Transitions: Example

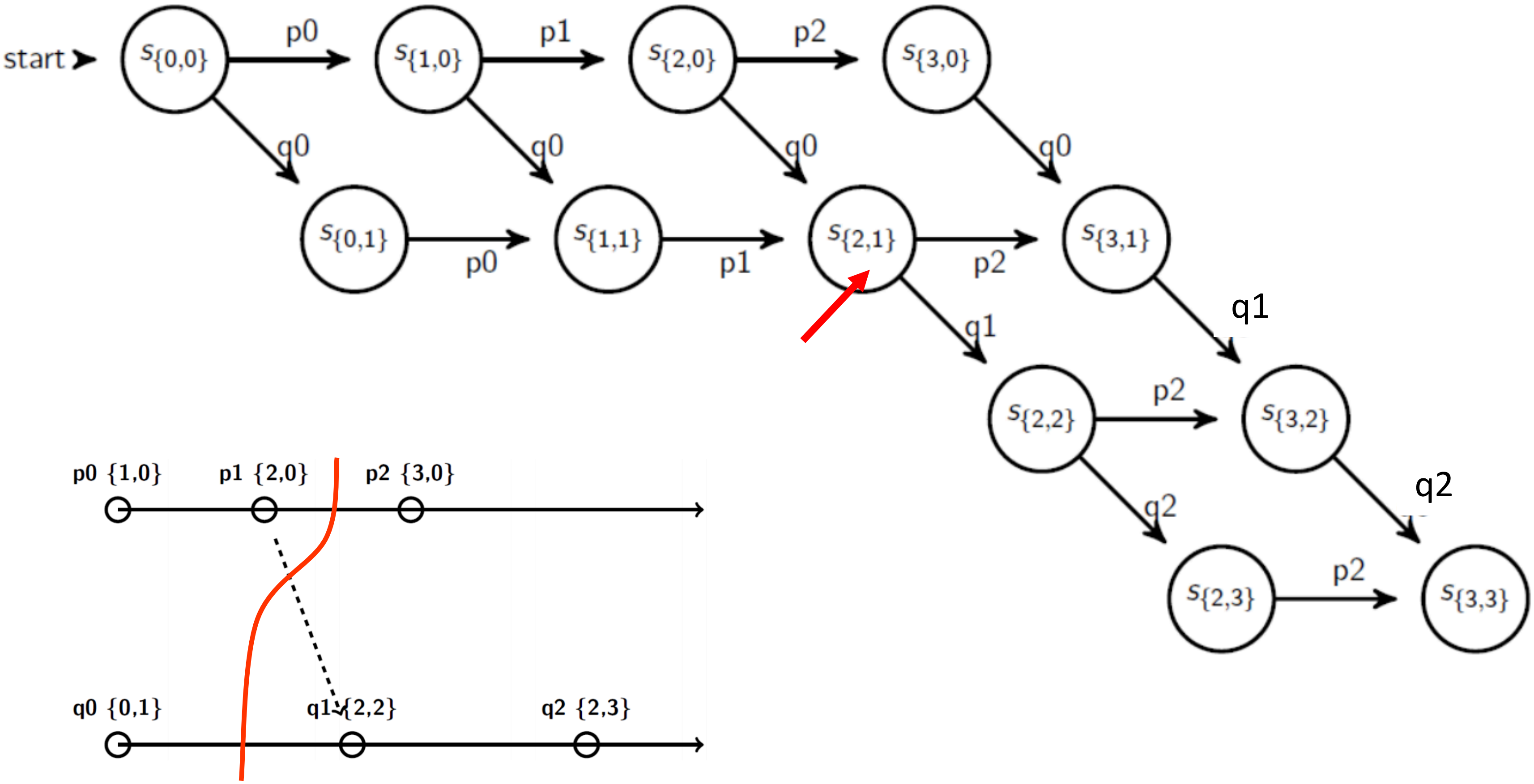
Execution Lattice. Each path is a linear execution of events.



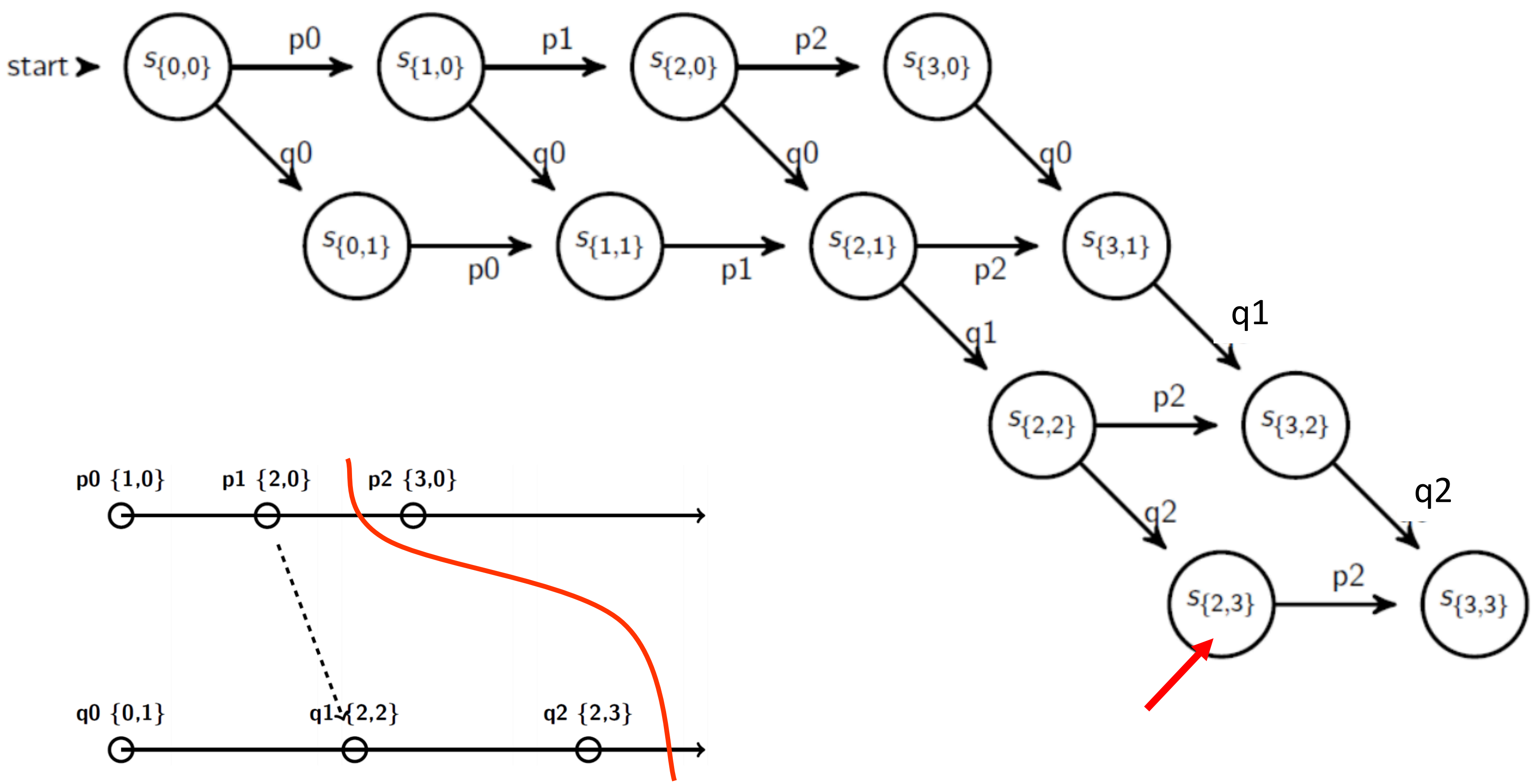
State Transitions: Example



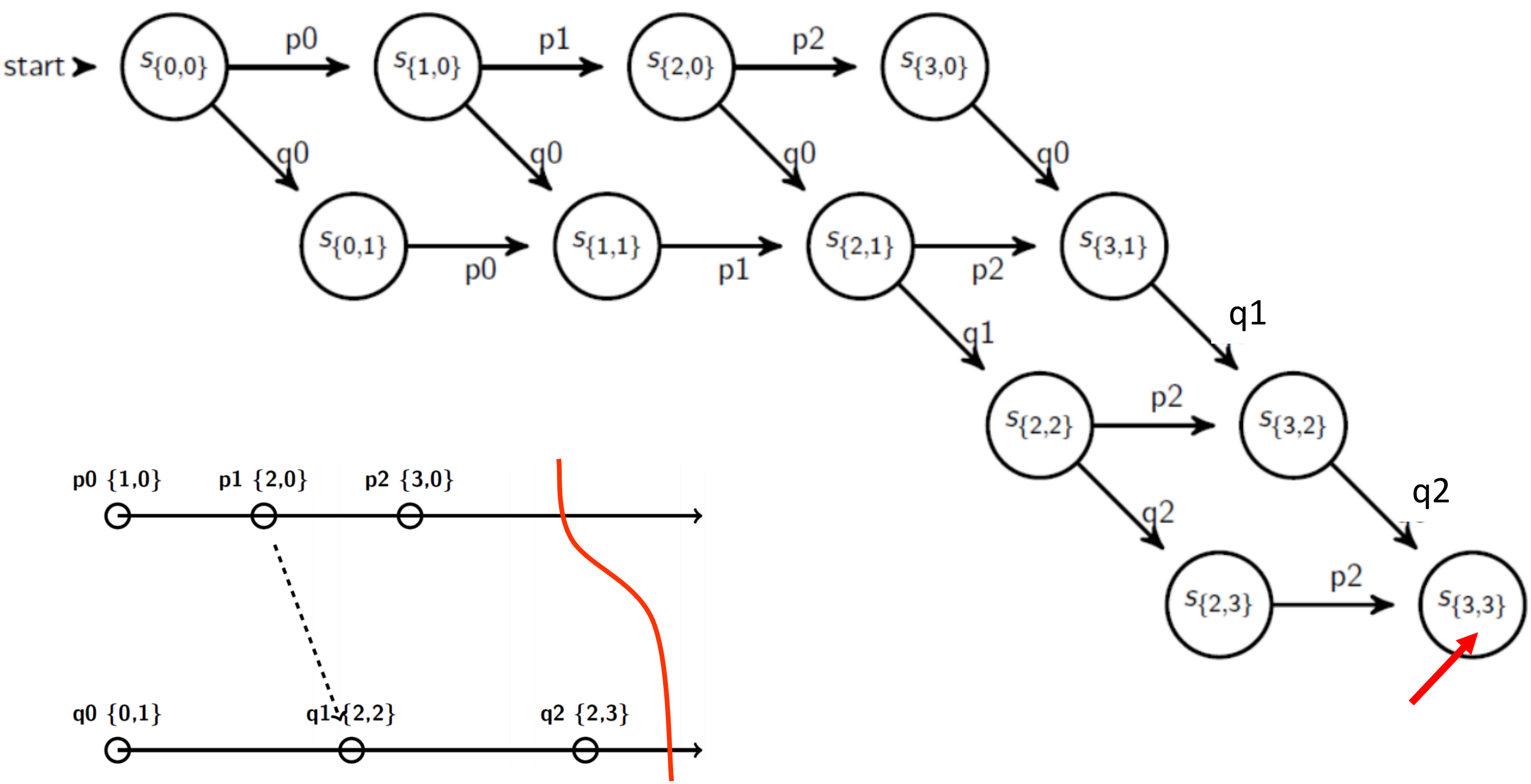
State Transitions: Example



State Transitions: Example



State Transitions: Example



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**

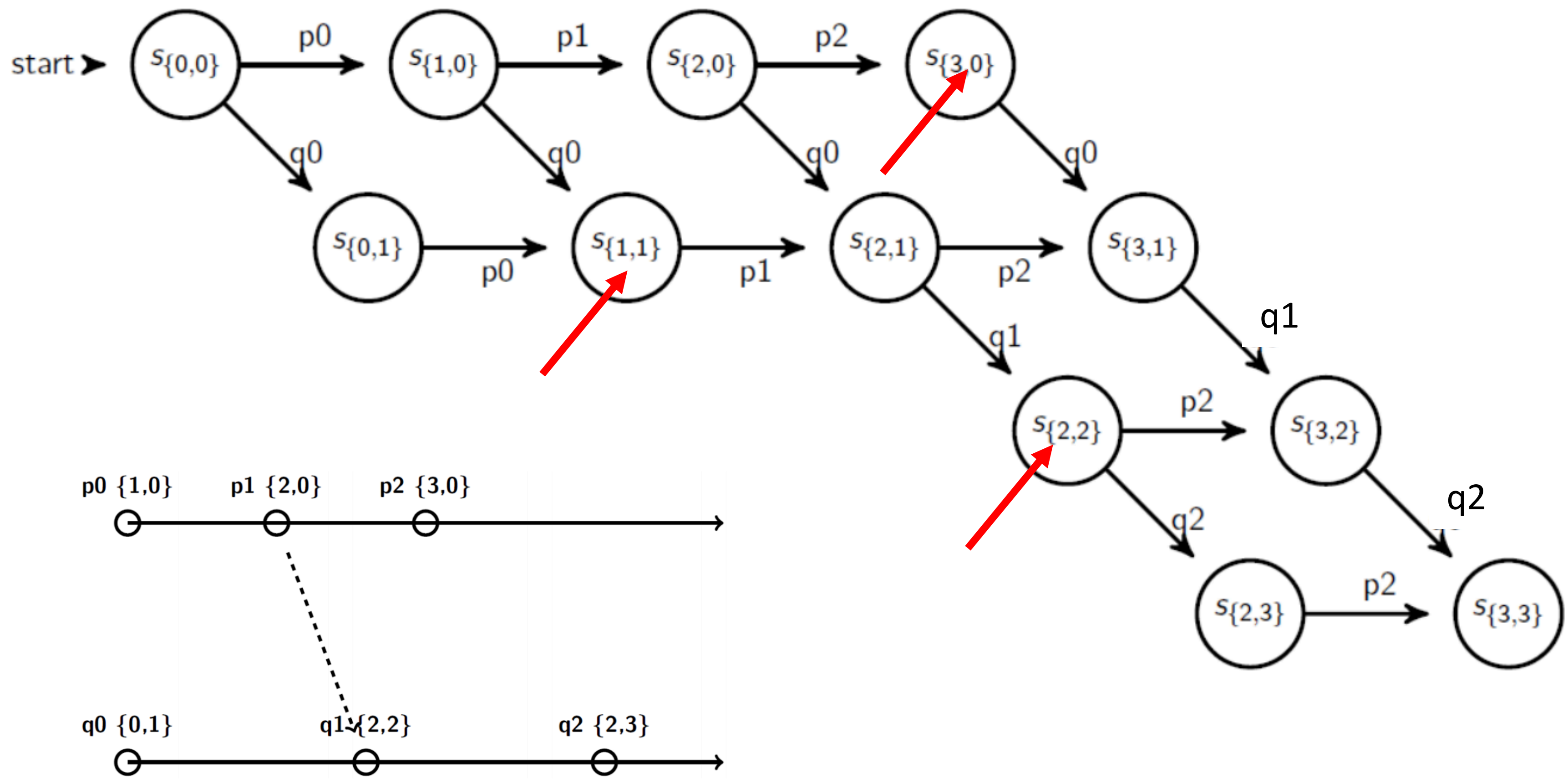
- **Examples:**
  - Guarantee that a distributed computation will terminate.
  - "Completeness" in failure detectors.
  - All processes eventually decide on a value.

- 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?

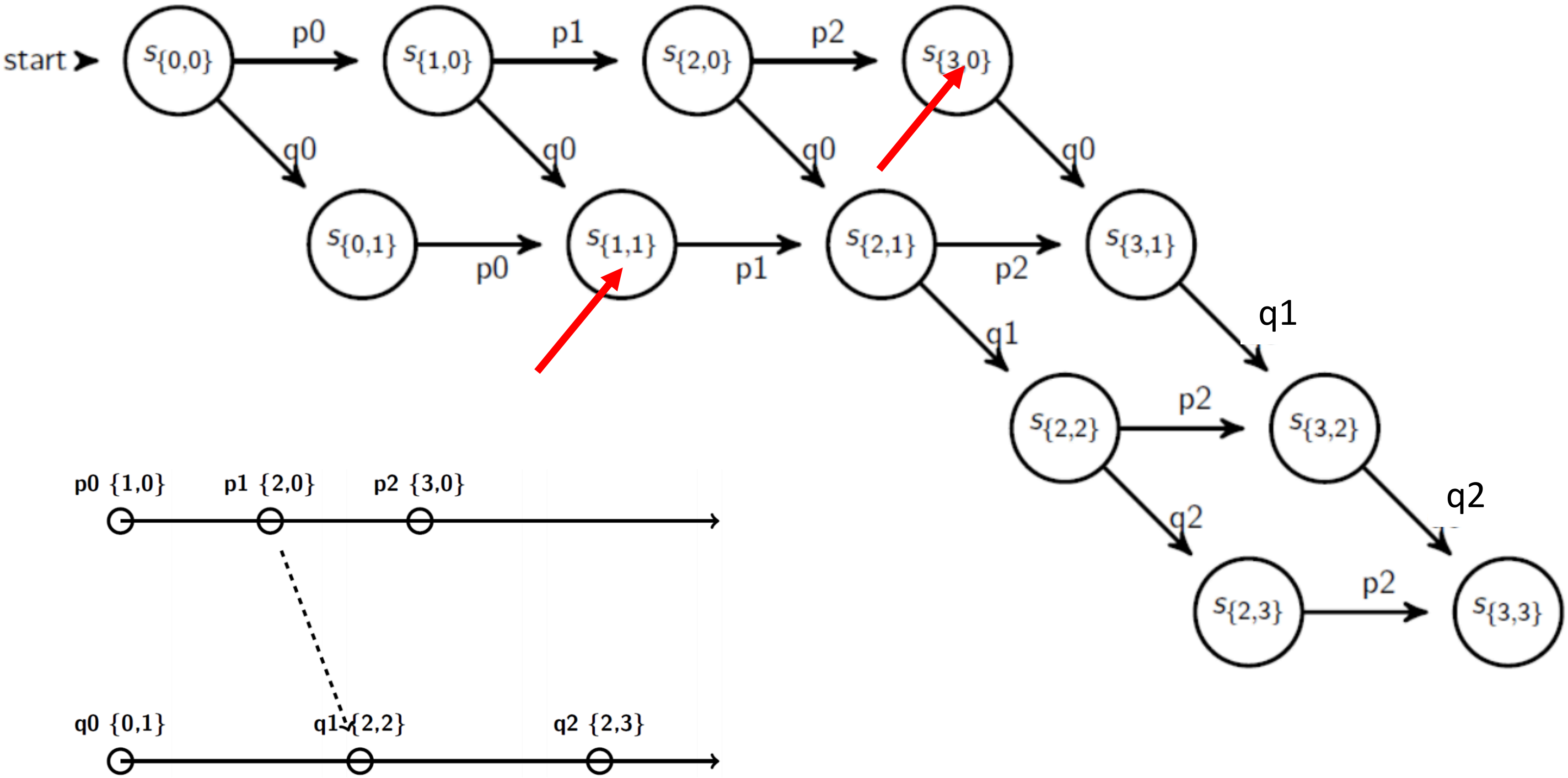
No



Liveness Example

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

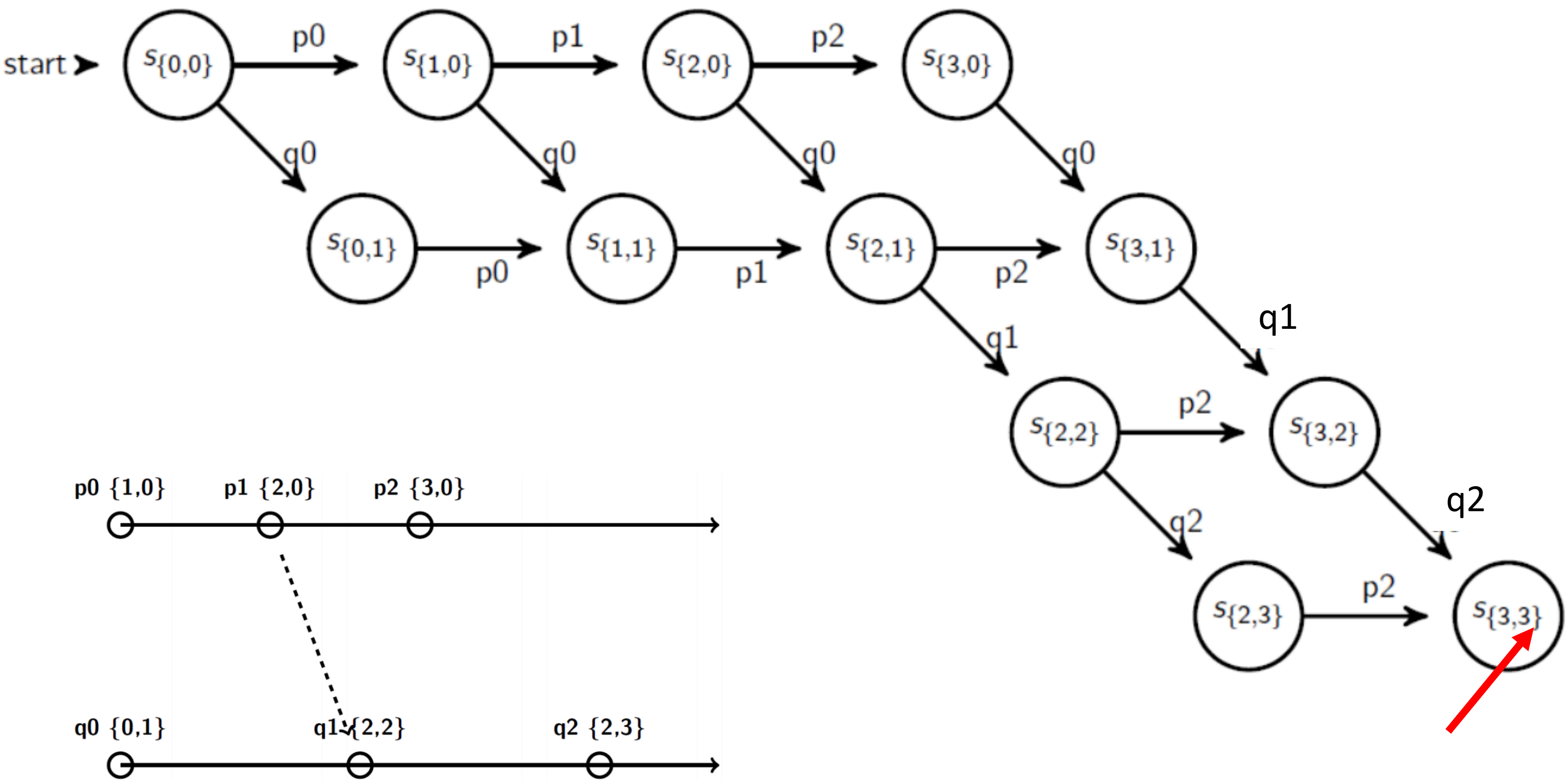
No



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:**
  - 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)) ≡ $\forall L \in$ 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.

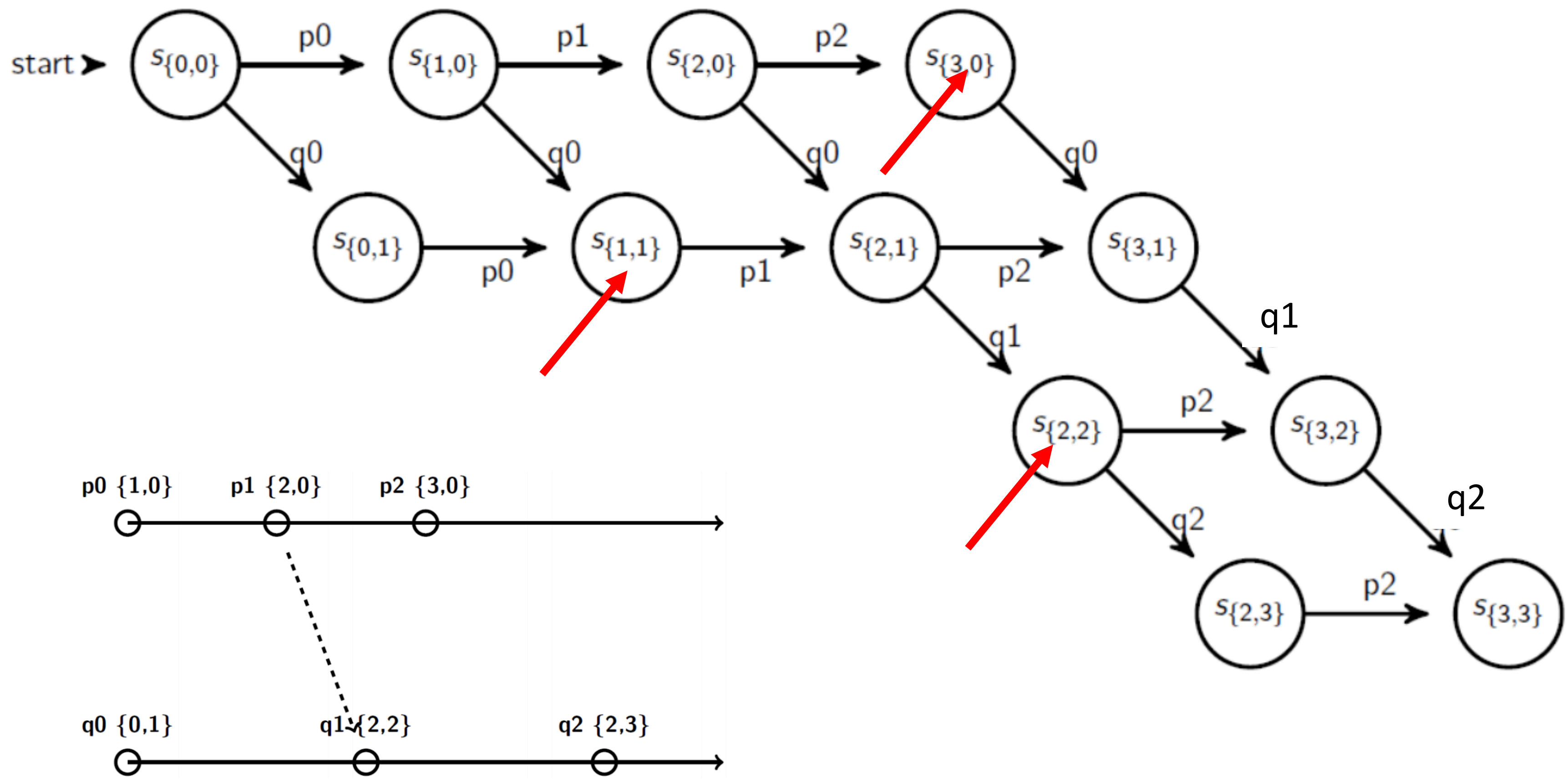
Safety

- Safety = guarantee that something bad will never happen.
- Examples:
  - There is no deadlock in a distributed transaction system.
  - "Accuracy" in failure detectors.
  - 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?

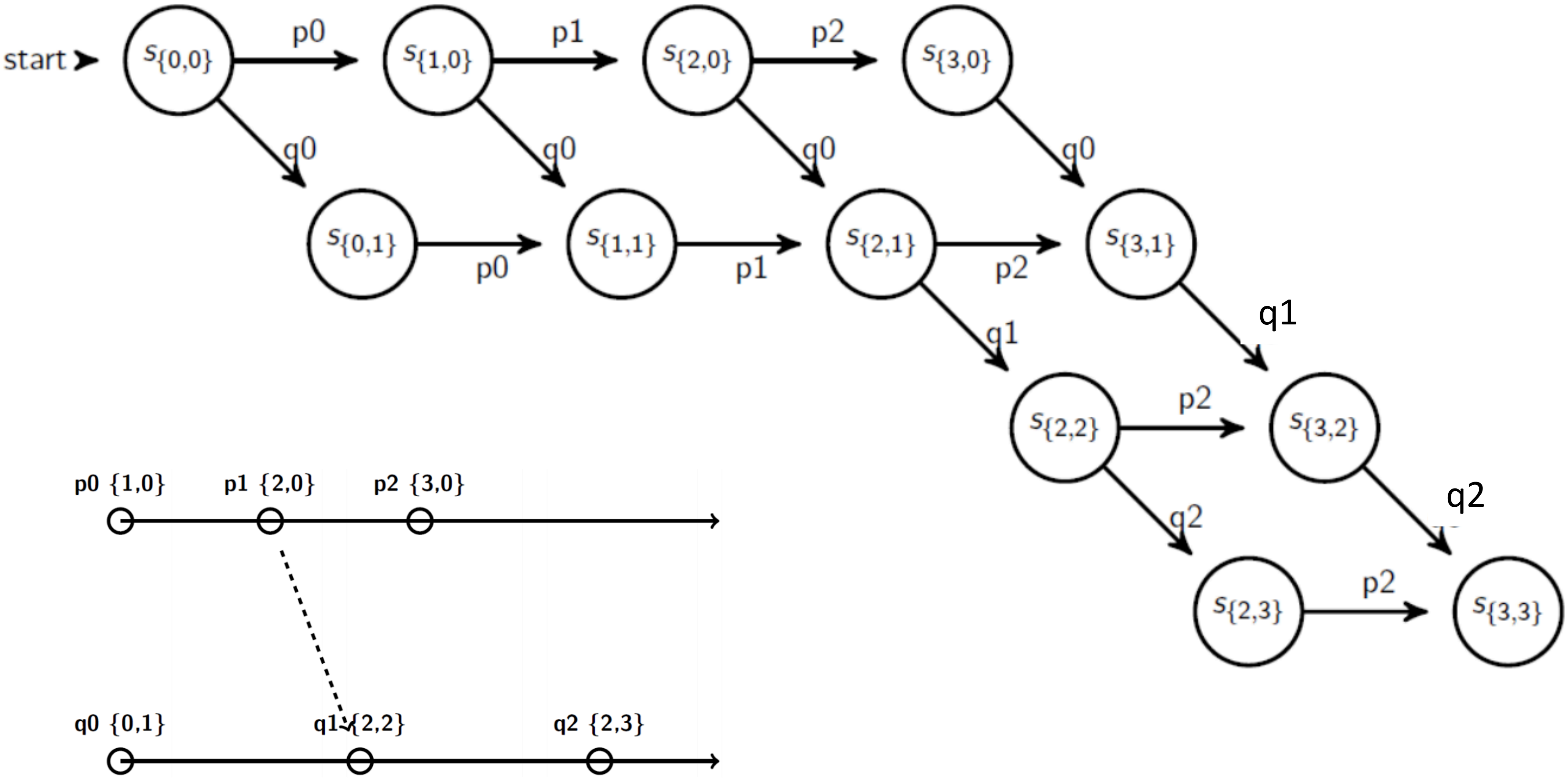
No



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.
  - 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.

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.