Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Logistics Related

- HW1 has been released.
  - You can solve first 4 questions right away
  - You can solve last two questions hopefully by end of today's class.

- MP0 due on Wednesday.

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.

How to capture global state?

- Ideally: 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: find a consistent global state.
  - 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.

How to find a consistent global state that corresponds to a consistent cut ?

Chandy-Lamport Algorithm

- Goal:
  - Record a global snapshot
    - 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.

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).
    - if p_i sends **m** before **m'** to p_j , then p_j receives **m** before **m'**.
  - All messages arrive intact, and are not duplicated.
  - No failures: neither channel nor processes fail.

Chandy-Lamport Algorithm

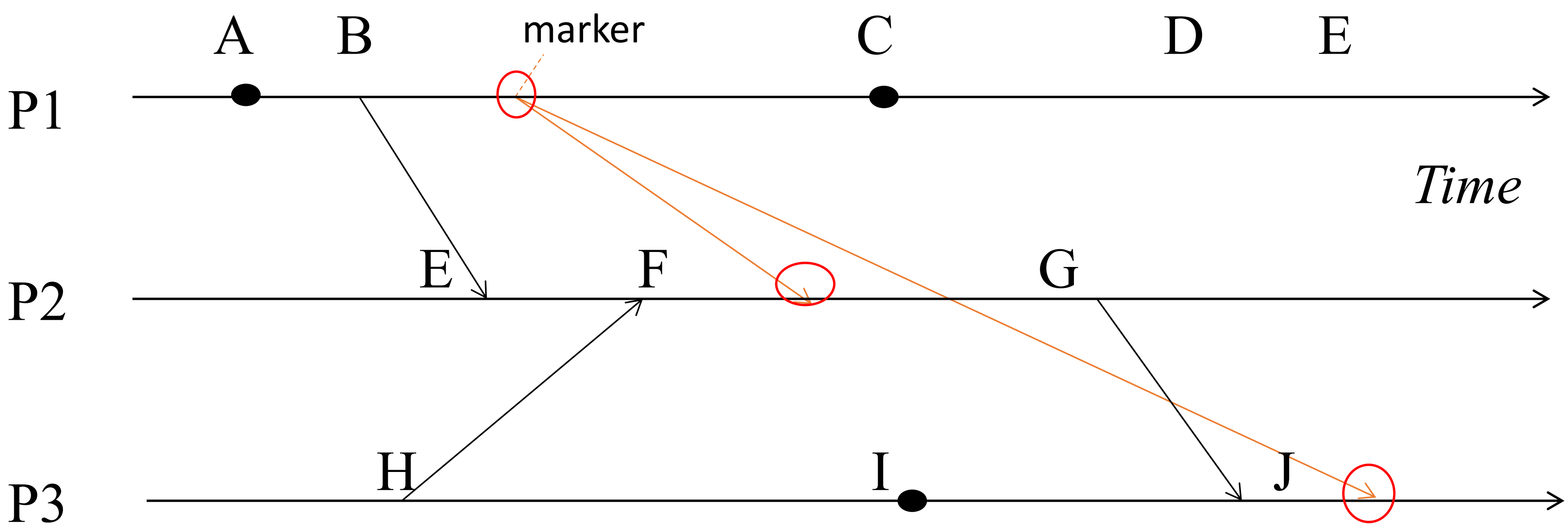
- *Requirements:*
  - Snapshot should not interfere with normal application actions, and it should not require application to stop sending messages.
  - Any process may initiate algorithm.

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.

Example



Inconsistent cut: {B, F, J}
(J is in cut but G isn't)

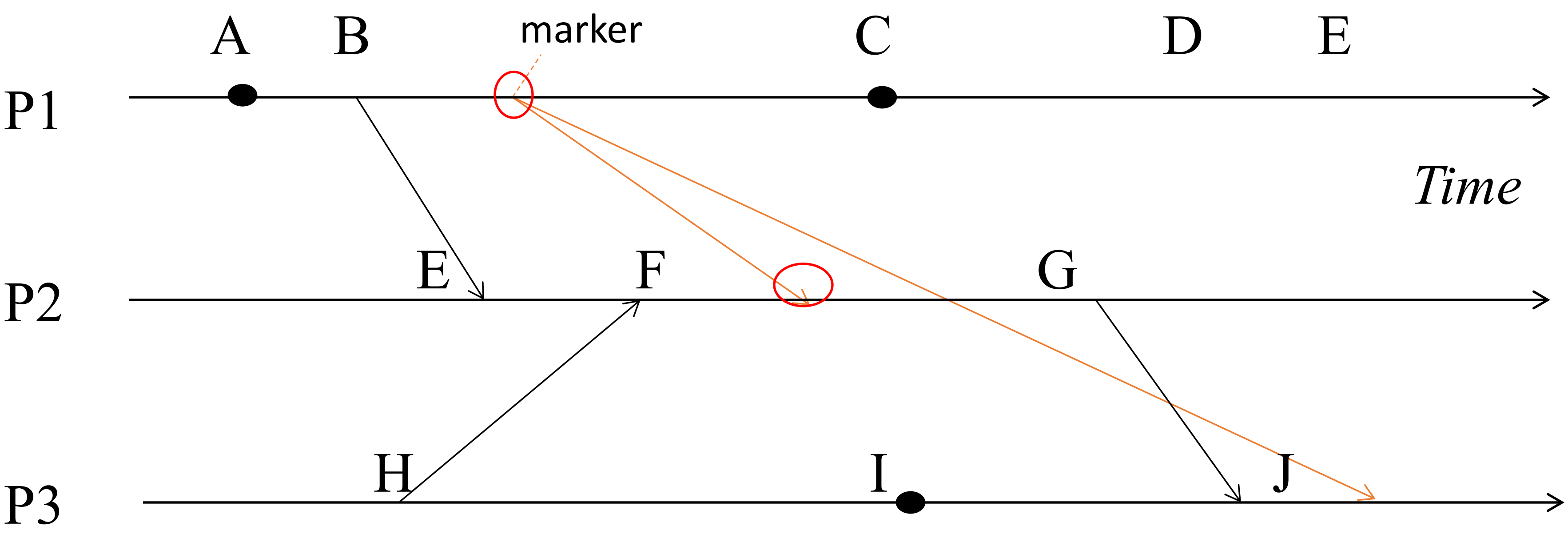
Chandy-Lamport Algorithm Intuition

- 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**.
  - If marker is received for the first time.
    - records its own state.
    - sends marker on all other channels.

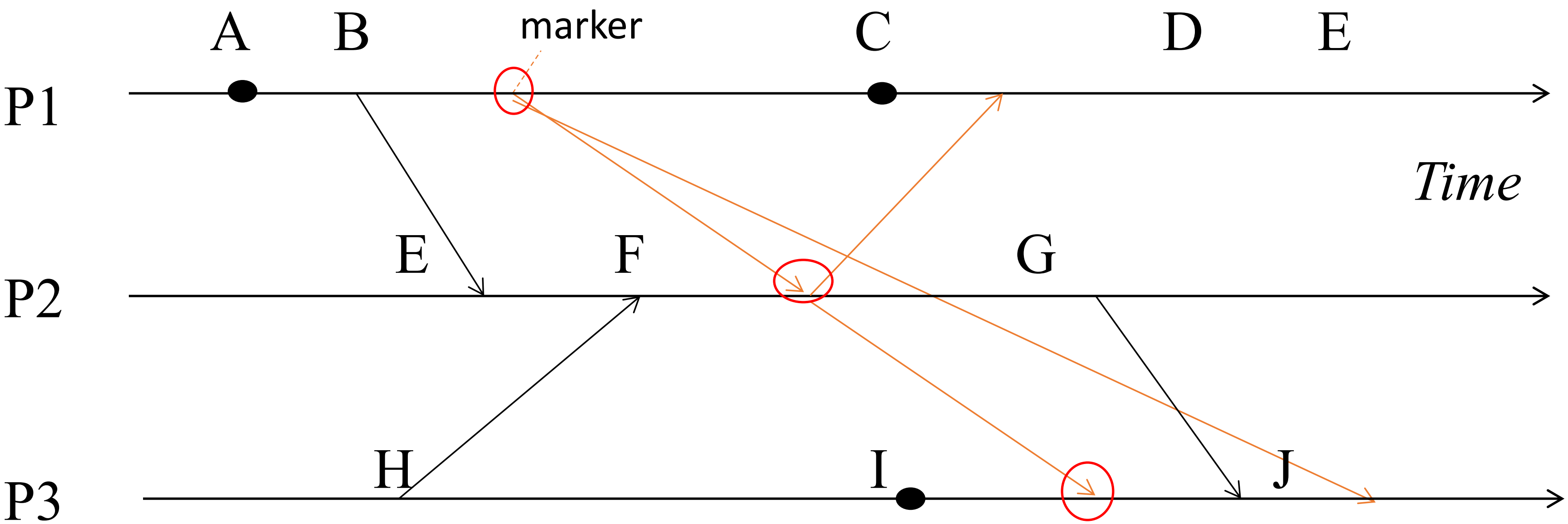
Leads to a consistent cut

Example



What can we say about marker from P2 to P3?

Example



Marker from P2 must reach P3 before J

Consistent cut: {B, F, I}

Chandy-Lamport Algorithm

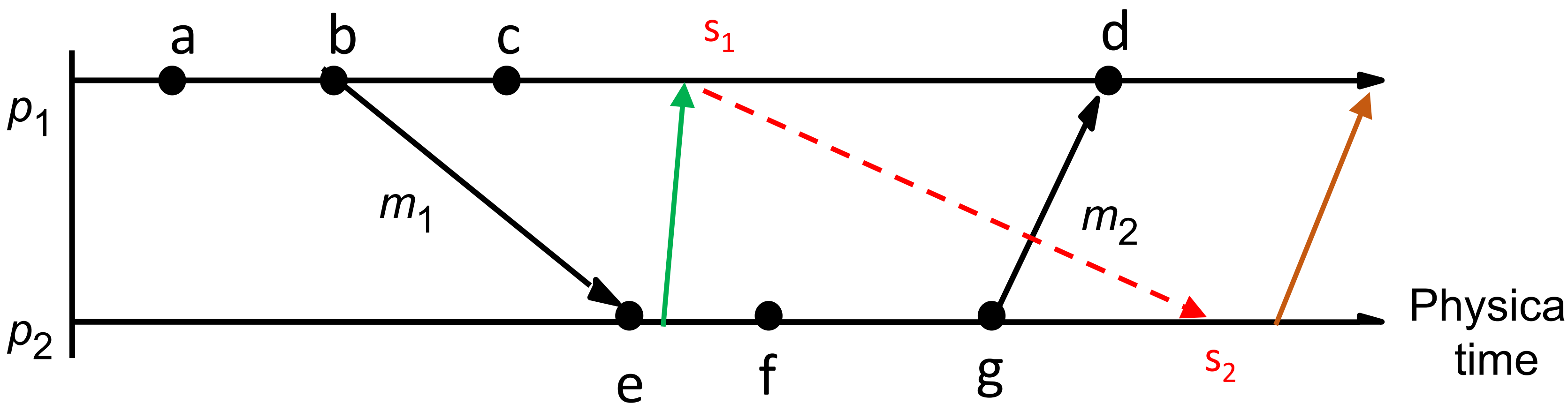
- First, initiator p_i:
  - records its own state.
  - creates a special **marker** message.
  - sends the **marker** to all other process.

- When a process receives a **marker**.
  - If marker is received for the first time.
    - records its own state.
    - sends marker on all other channels.

Leads to a consistent cut.

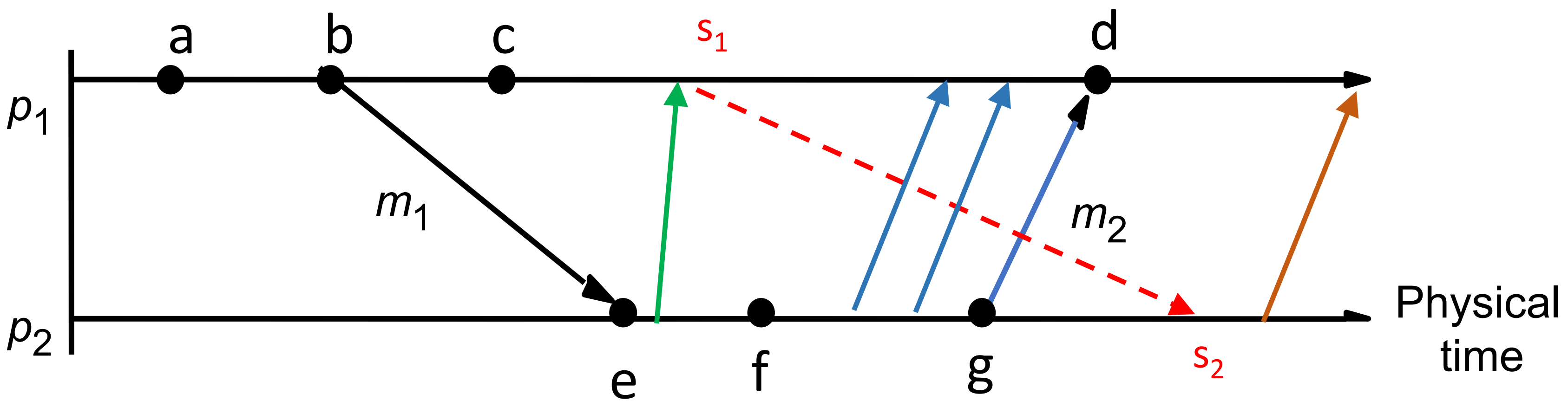
What about the channel state?

Chandy-Lamport Algorithm Intuition



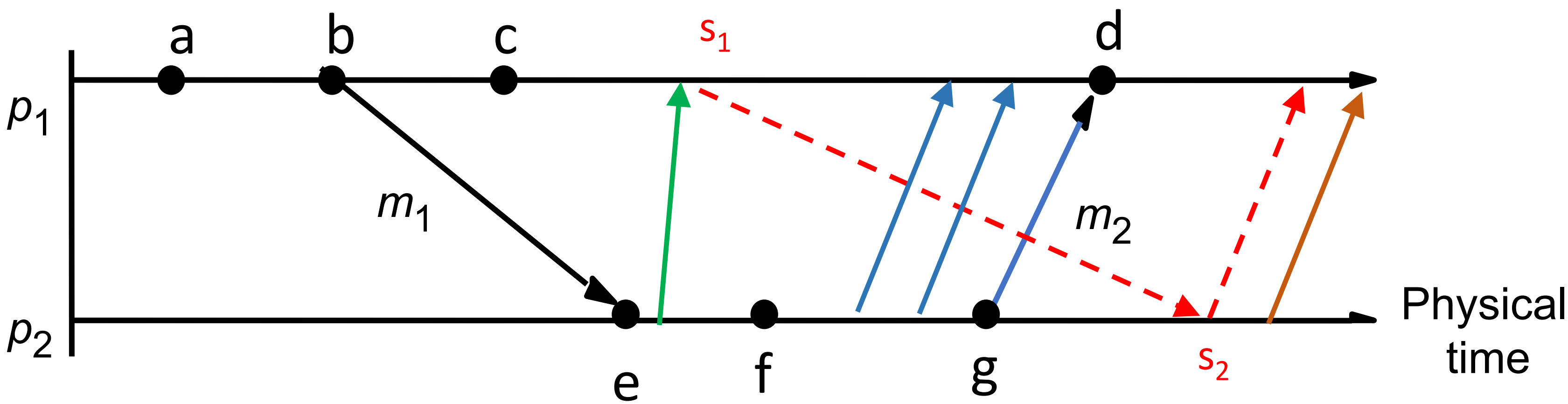
Cut frontier: {c, g}

Chandy-Lamport Algorithm Intuition



Cut frontier: {c, g}

Chandy-Lamport Algorithm Intuition



Cut frontier: {c, g}

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

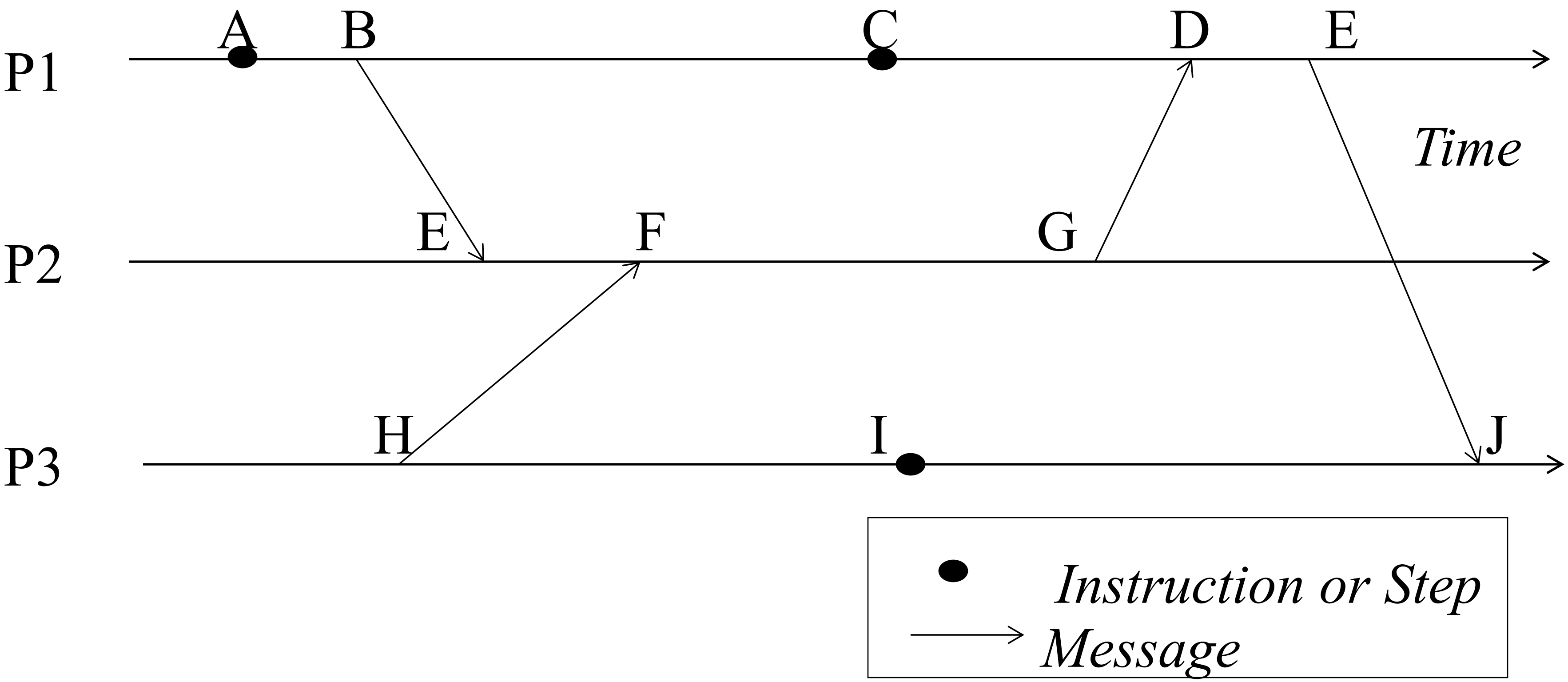
- if (this is the first **marker** p_i is seeing)
  - p_i records its own state first
  - marks the state of channel c_{ki} as "empty"
  - for j=1 to n except i
    - p_i sends out a **marker** message on outgoing channel c_{ij}
  - starts recording the incoming messages on each of the incoming channels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

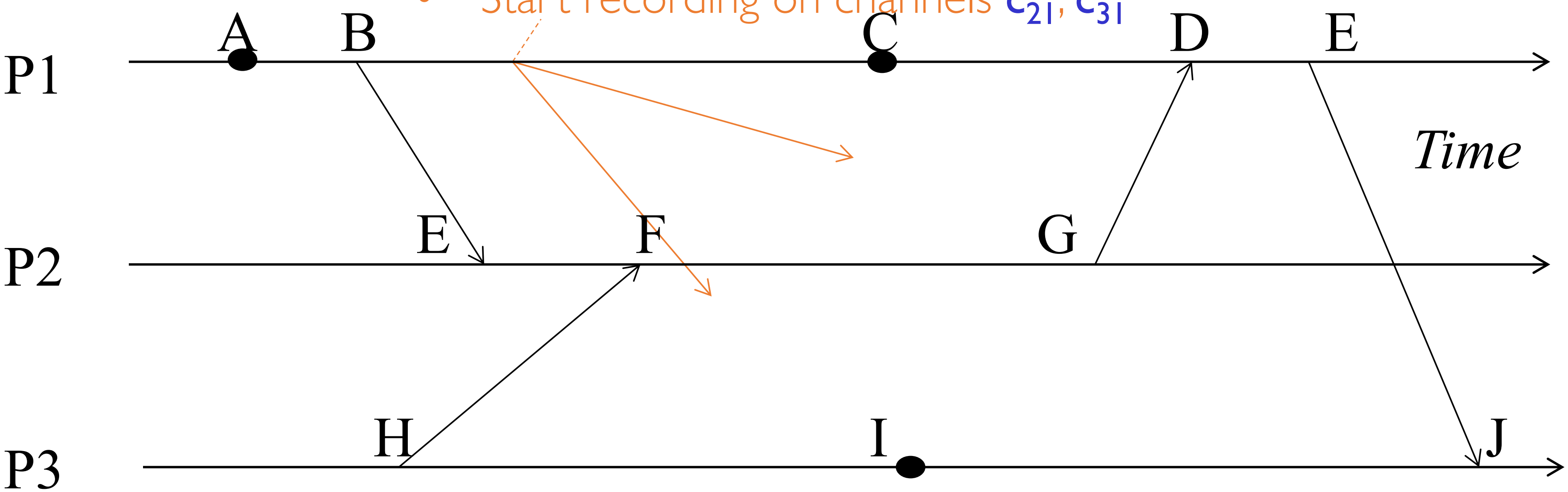
Example



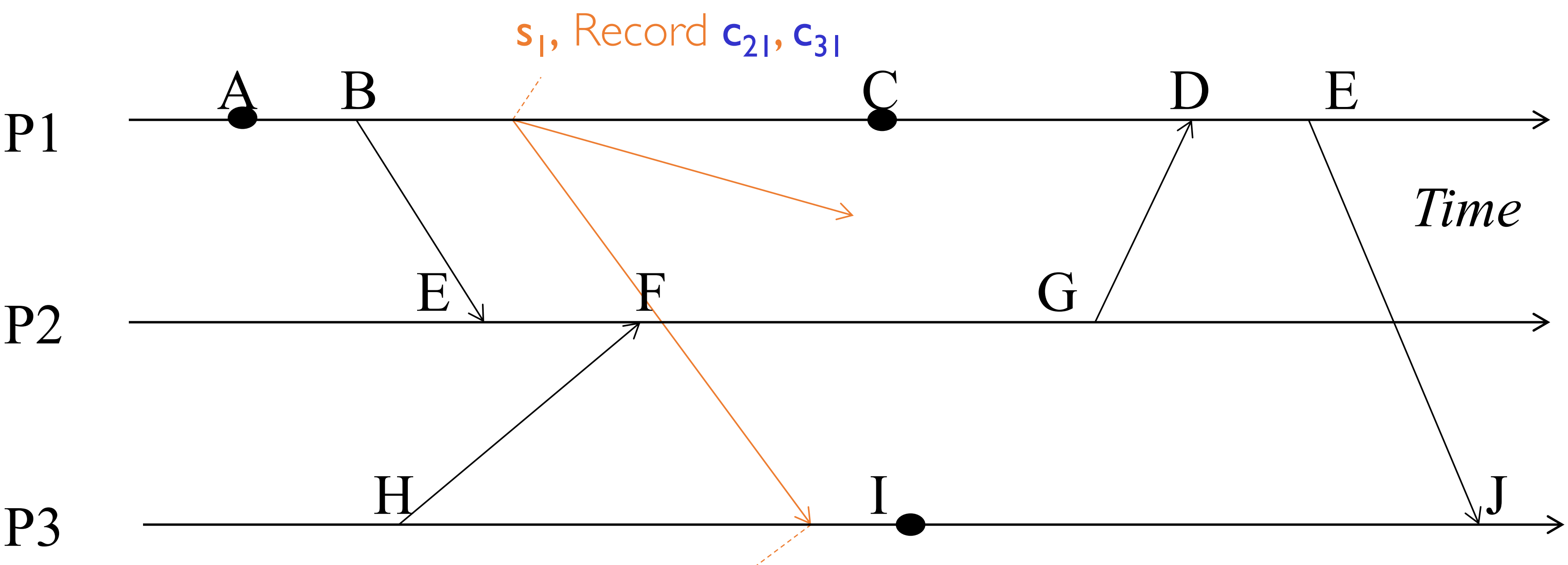
Example

p_1 is initiator:

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

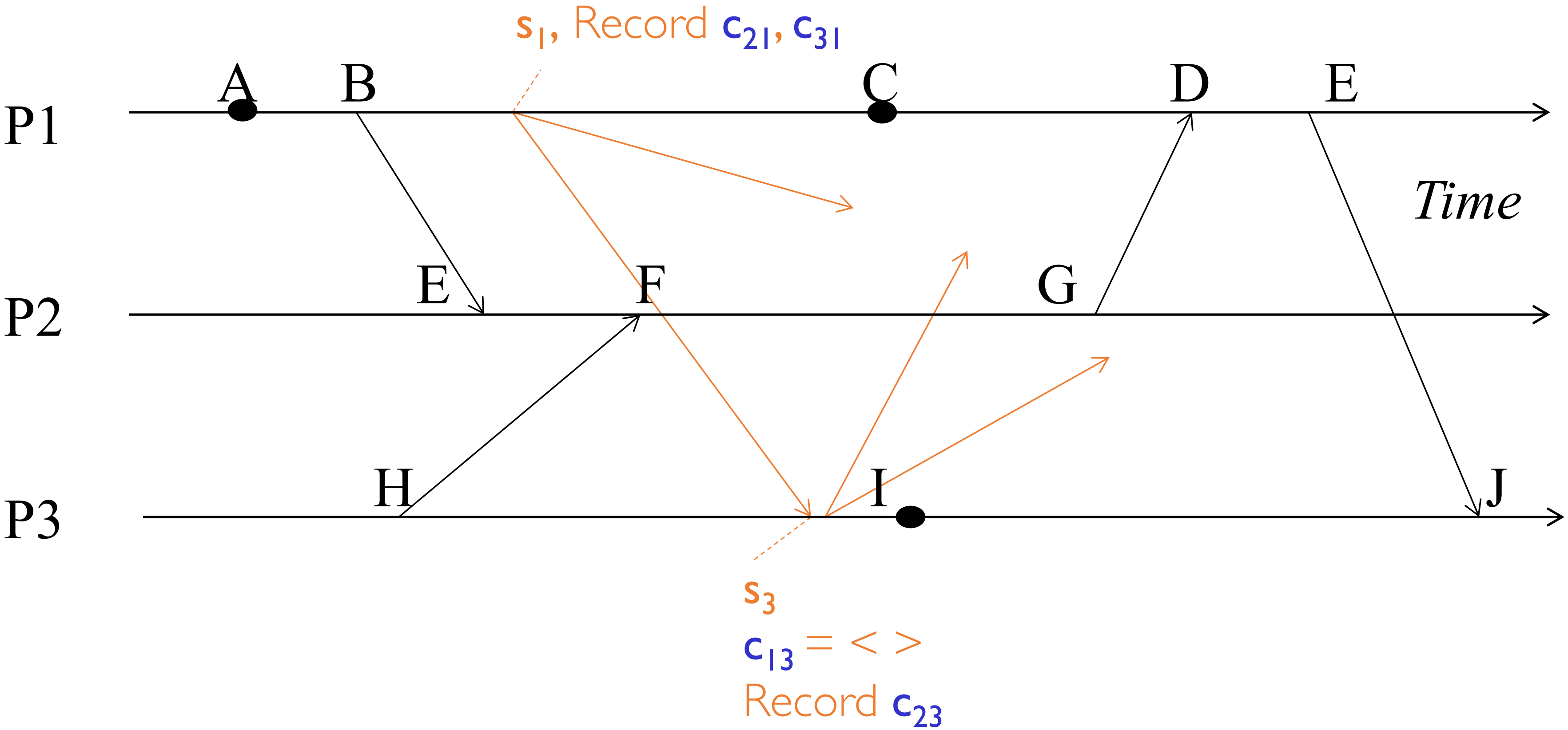


Example

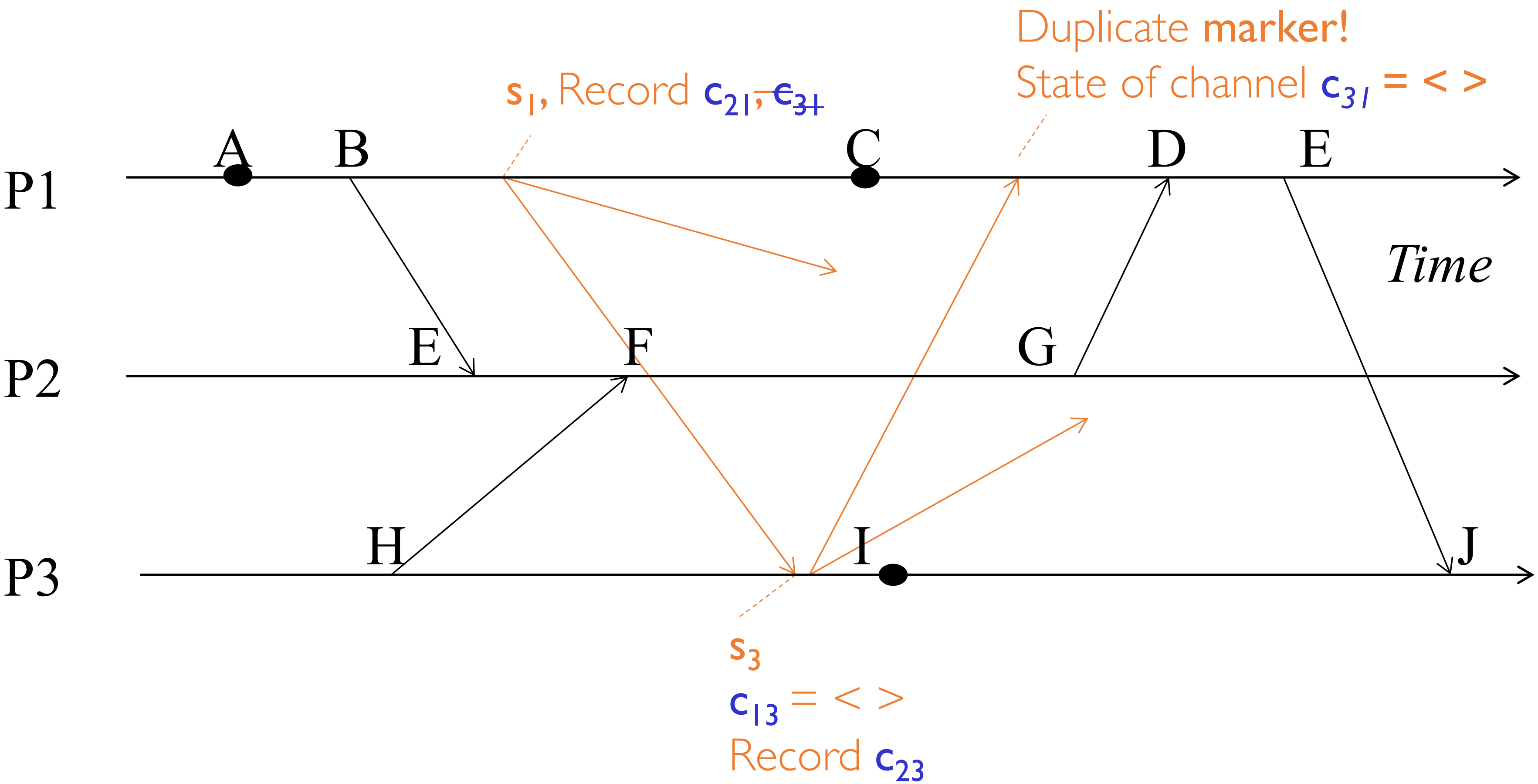


- First **marker**!
- Record own state as s_3
- Mark c_{13} state as empty
- Start recording on other incoming c_{23}
- Send out markers

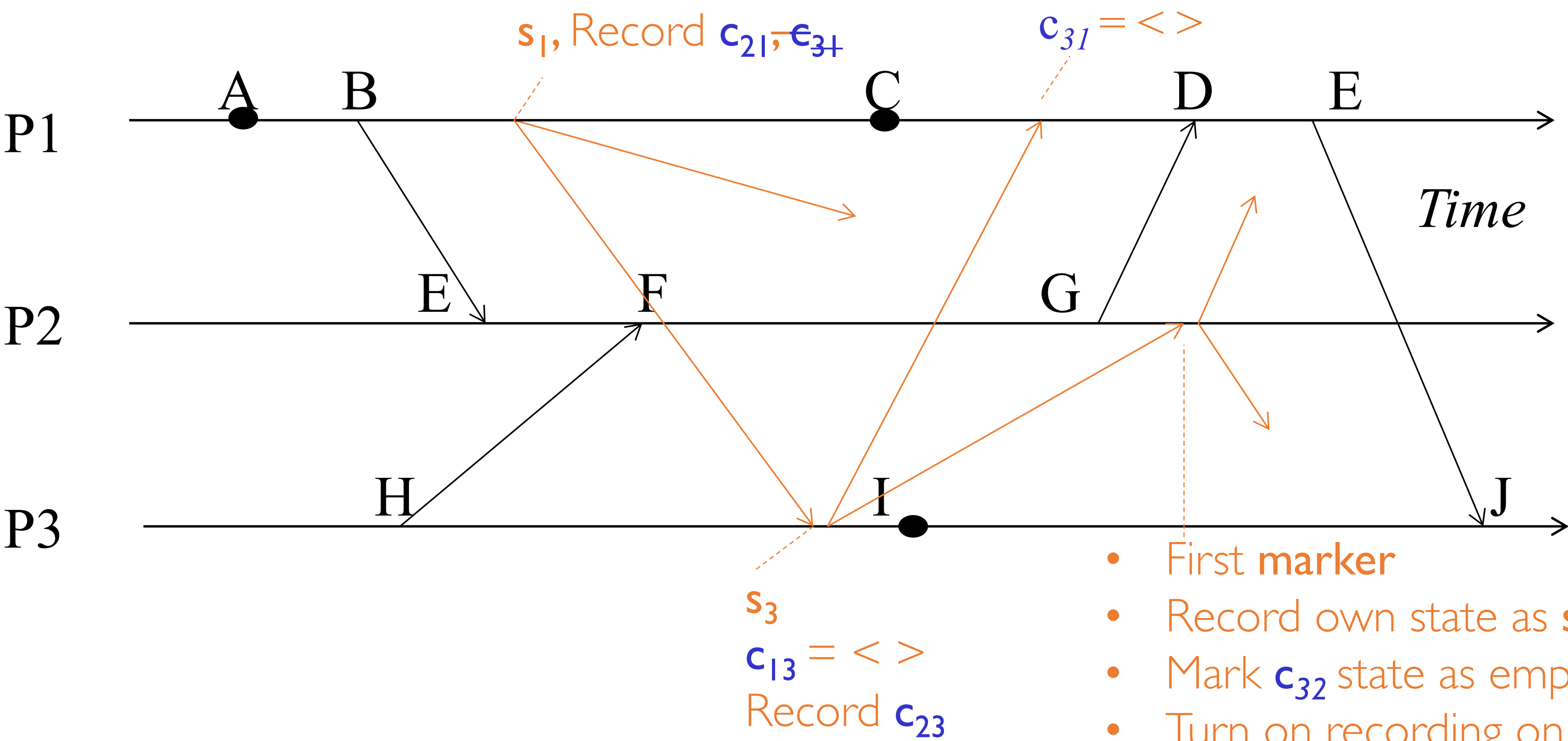
Example



Example

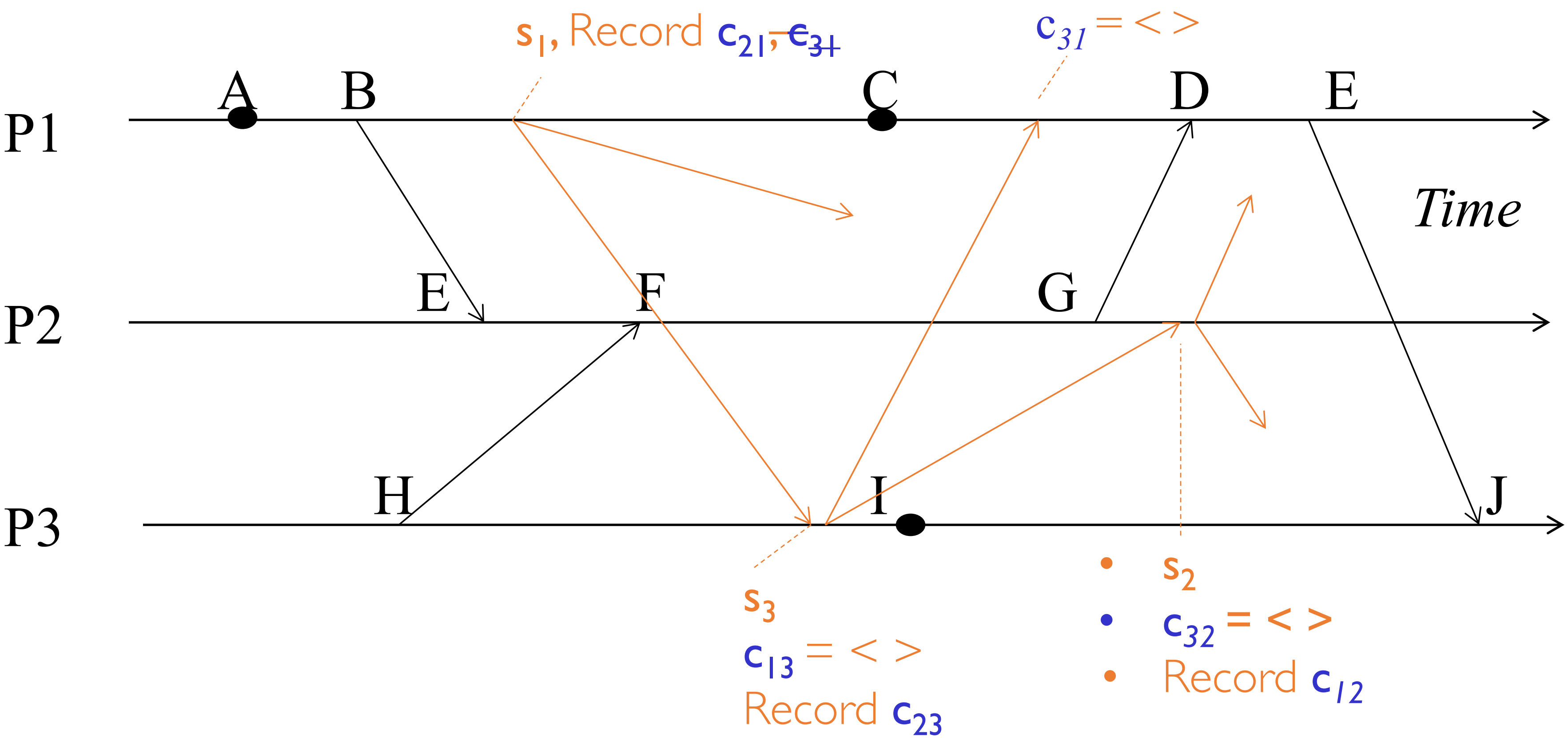

Duplicate **marker!**
State of channel c_{31} = < >
s_1, Record c_{21}, c_{31}
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s_3
c_{13} = < >
Record c_{23}

Example



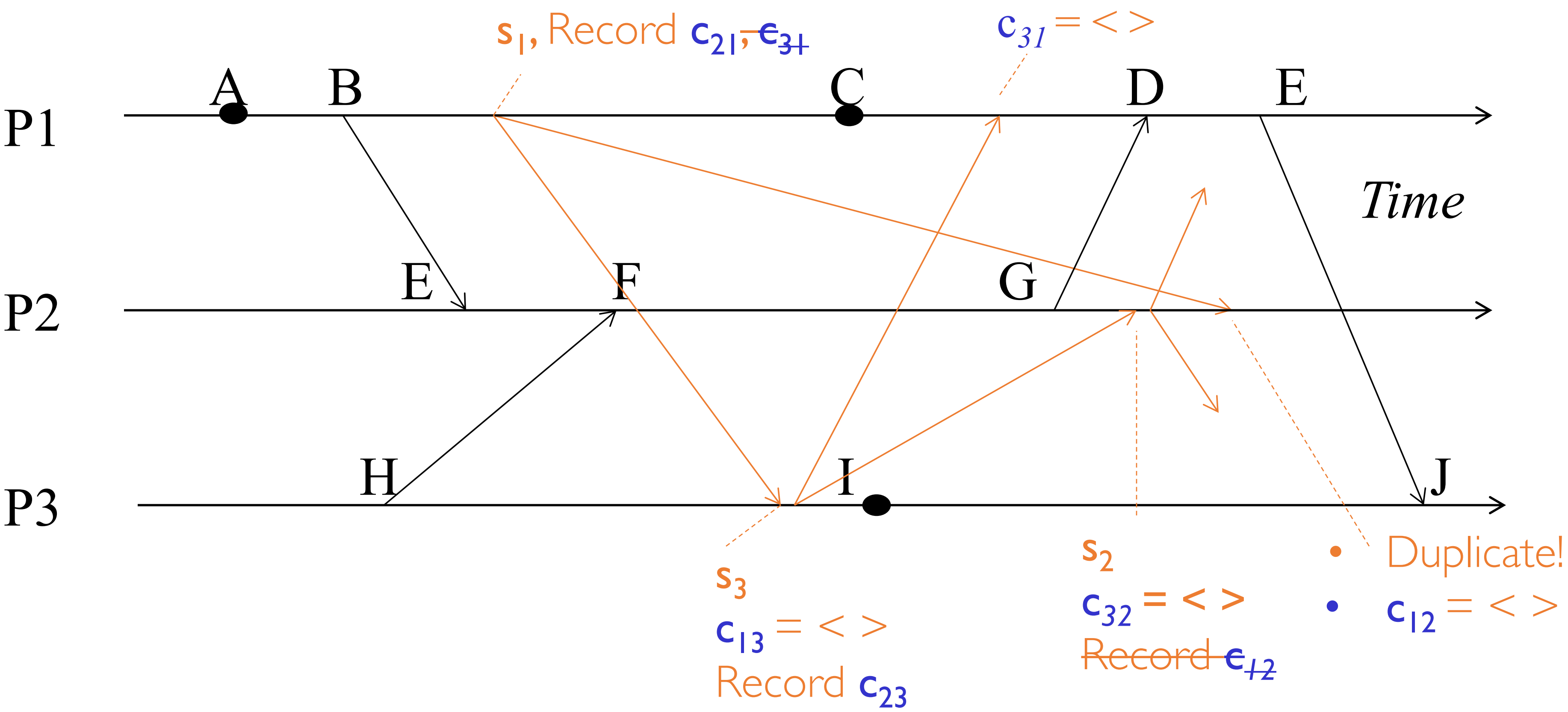
- First **marker**
- Record own state as s_2
- Mark c_{32} state as empty
- Turn on recording on c_{12}
- Send out **markers**

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

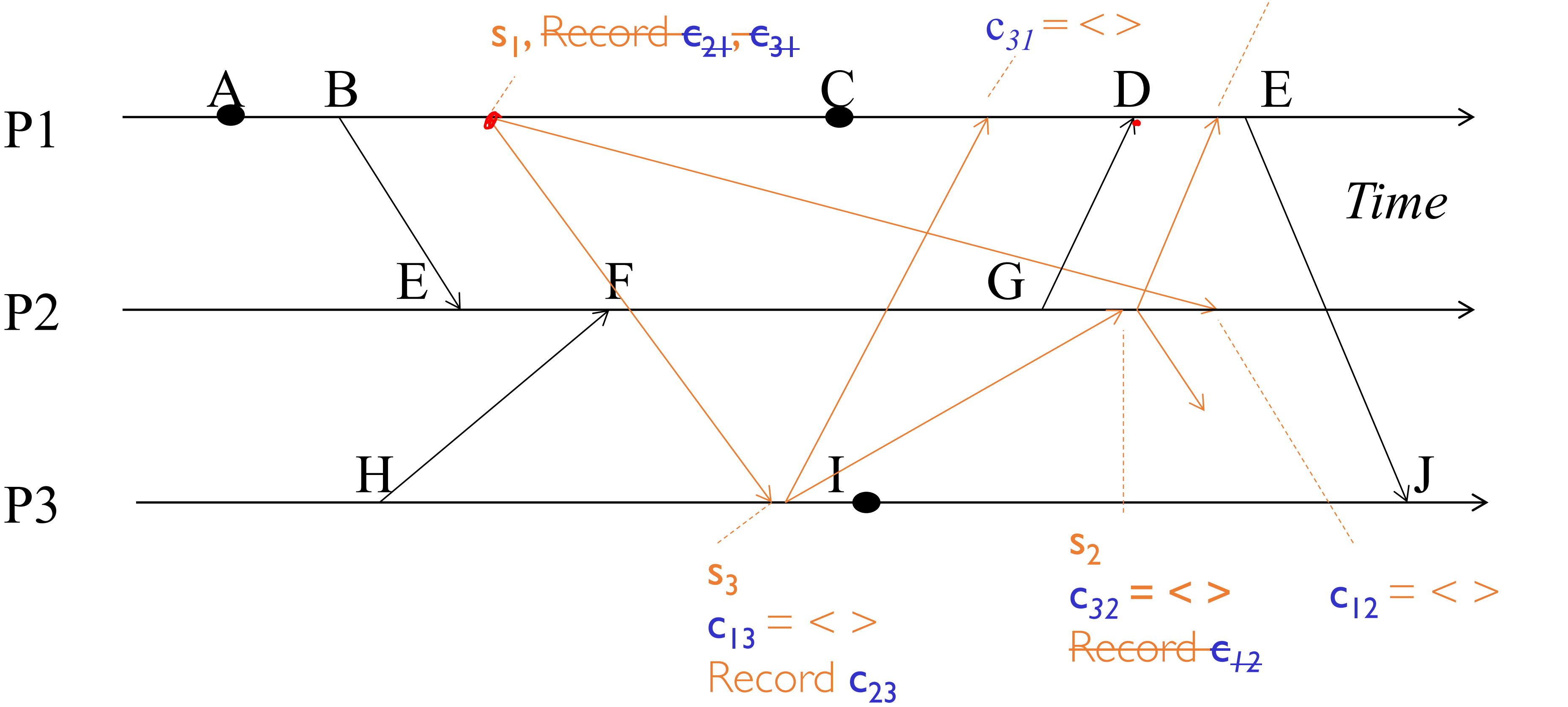
Example



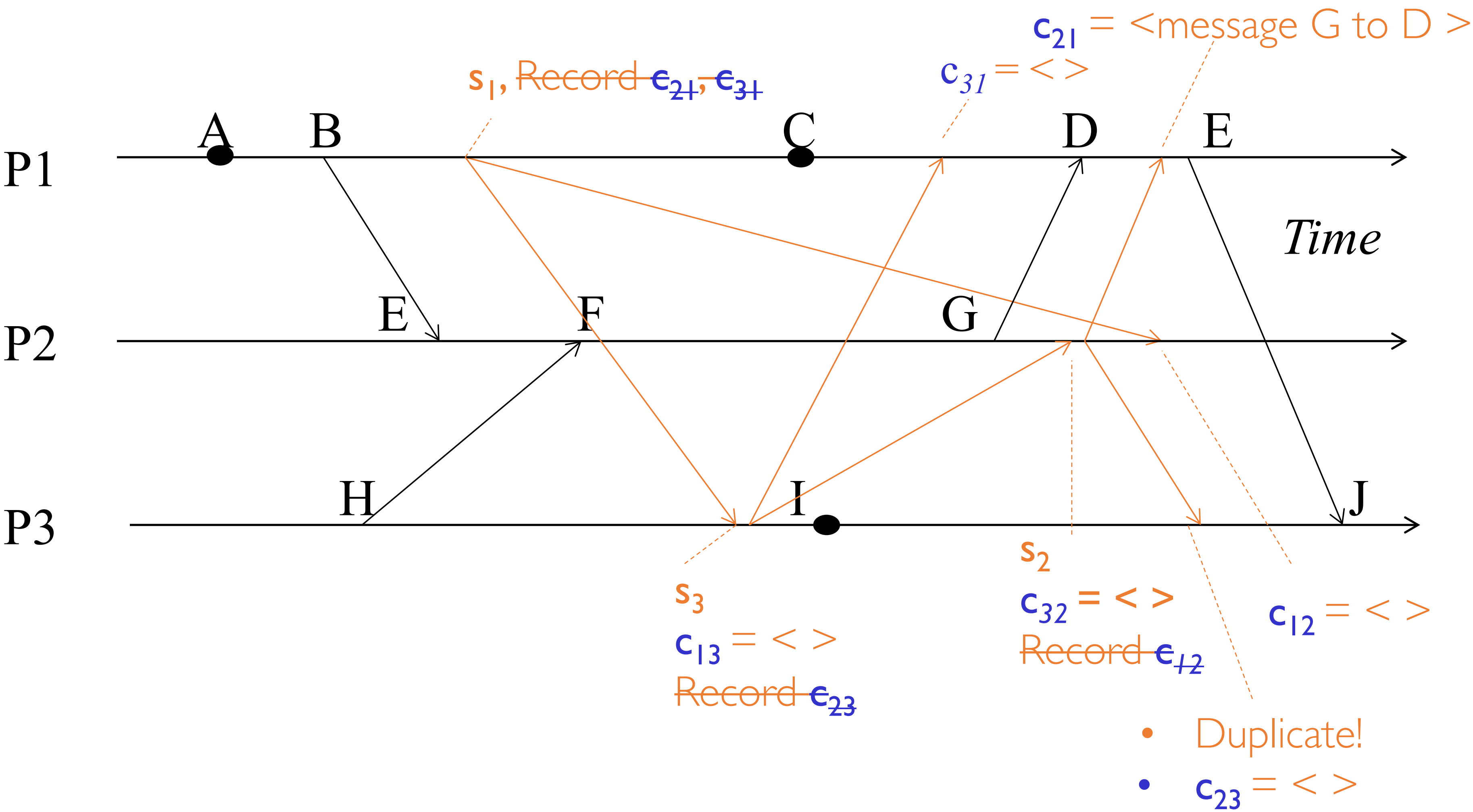
s1, Record c21, c31
c31 = < >
P1
A
B
C
D
E
Time
P2
E
F
G
P3
H
I
J
s3
c13 = < >
Record c23
s2
c32 = < >
Record c12
Duplicate!
c12 = < >

Example

- Duplicate!
- c_{21} = <message G to D >

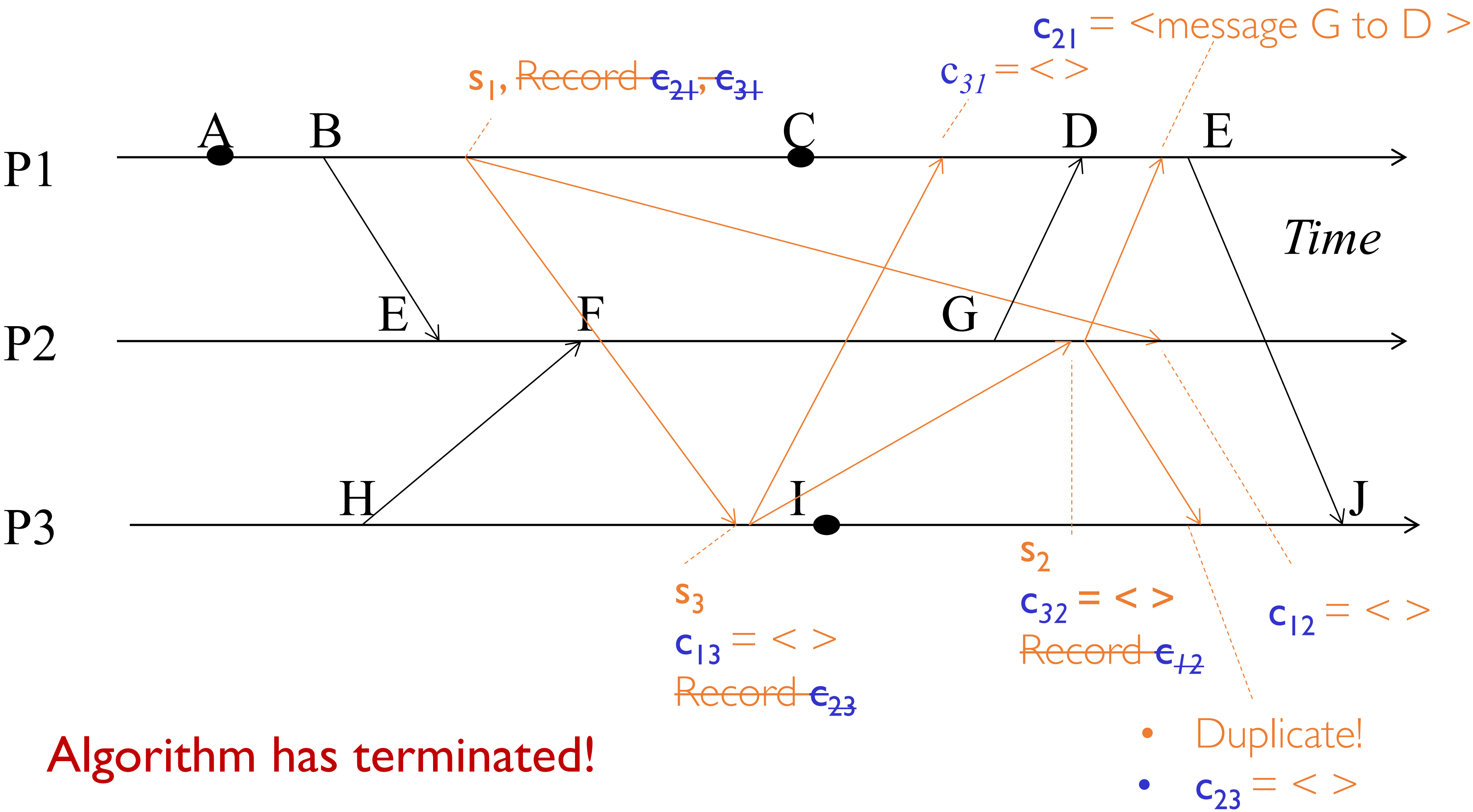


Example



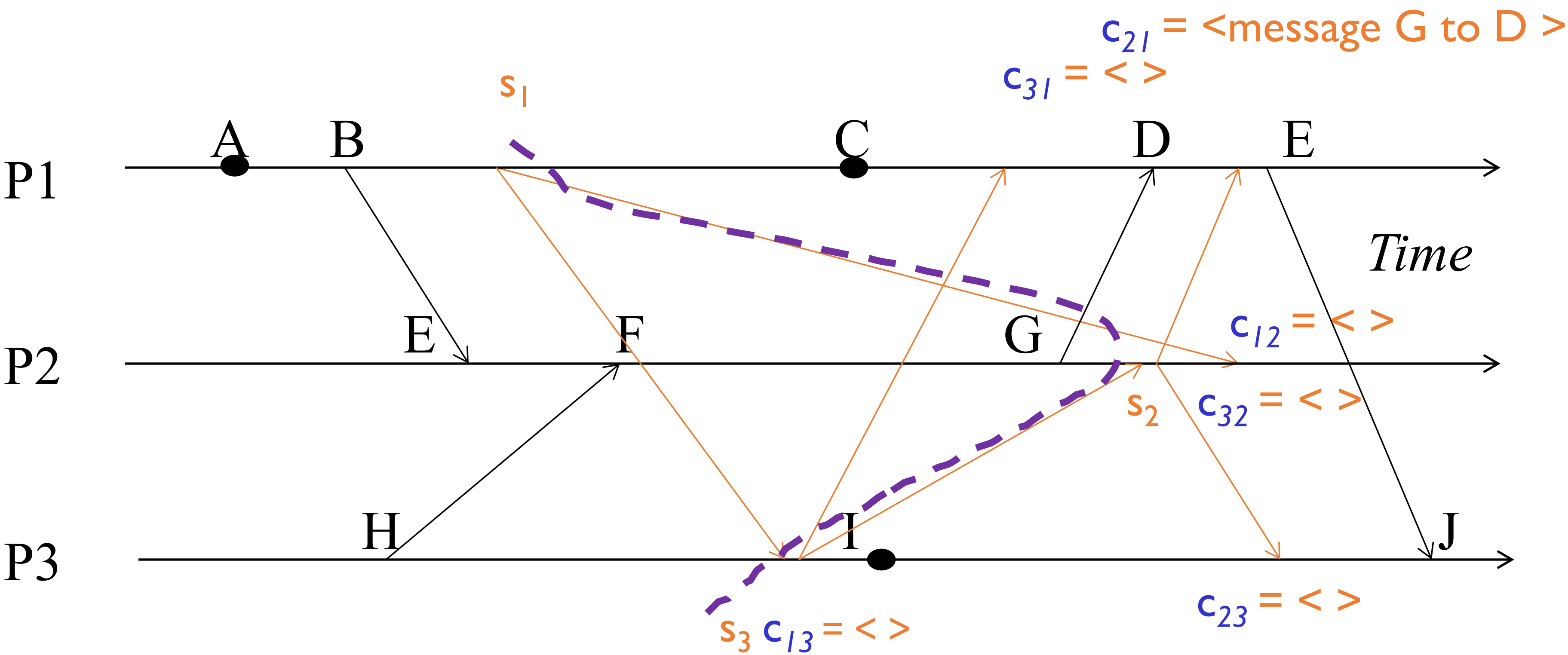
c21 = <message G to D >
c31 = < >
s1, Record c21, c31
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
s3
c13 = < >
Record c23
s2
c32 = < >
Record c12
c12 = < >
Duplicate!
c23 = < >

Example



Algorithm has terminated!

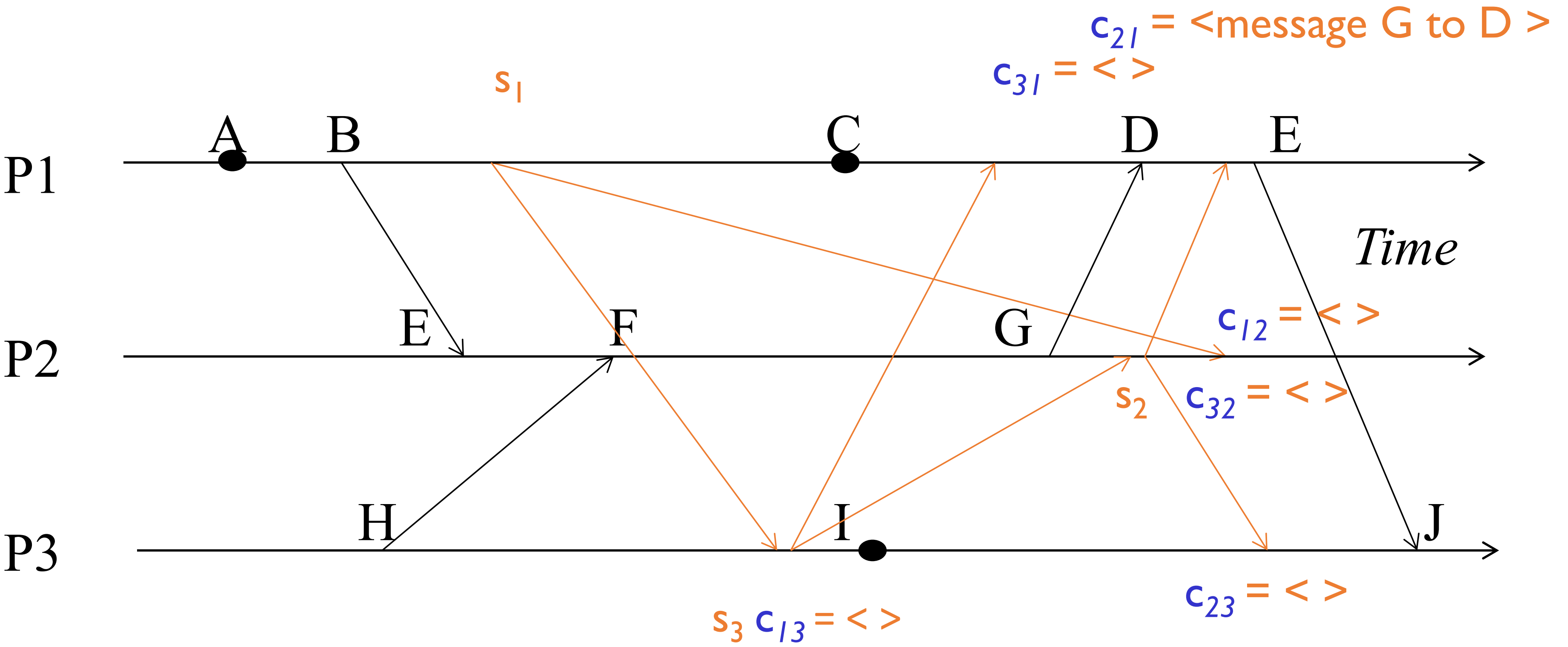
Example



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

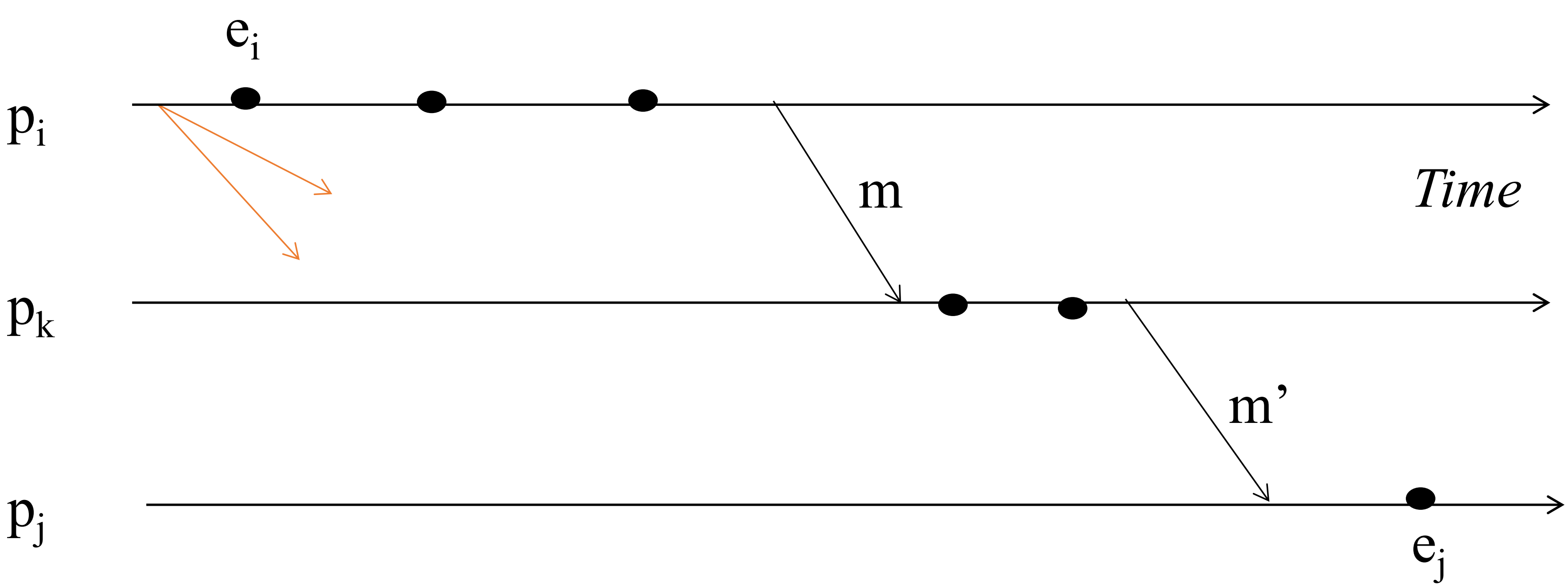
  if $\mathbf{e}_j$ is in the cut then $\mathbf{e}_i$ is also in the cut.

- 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>.

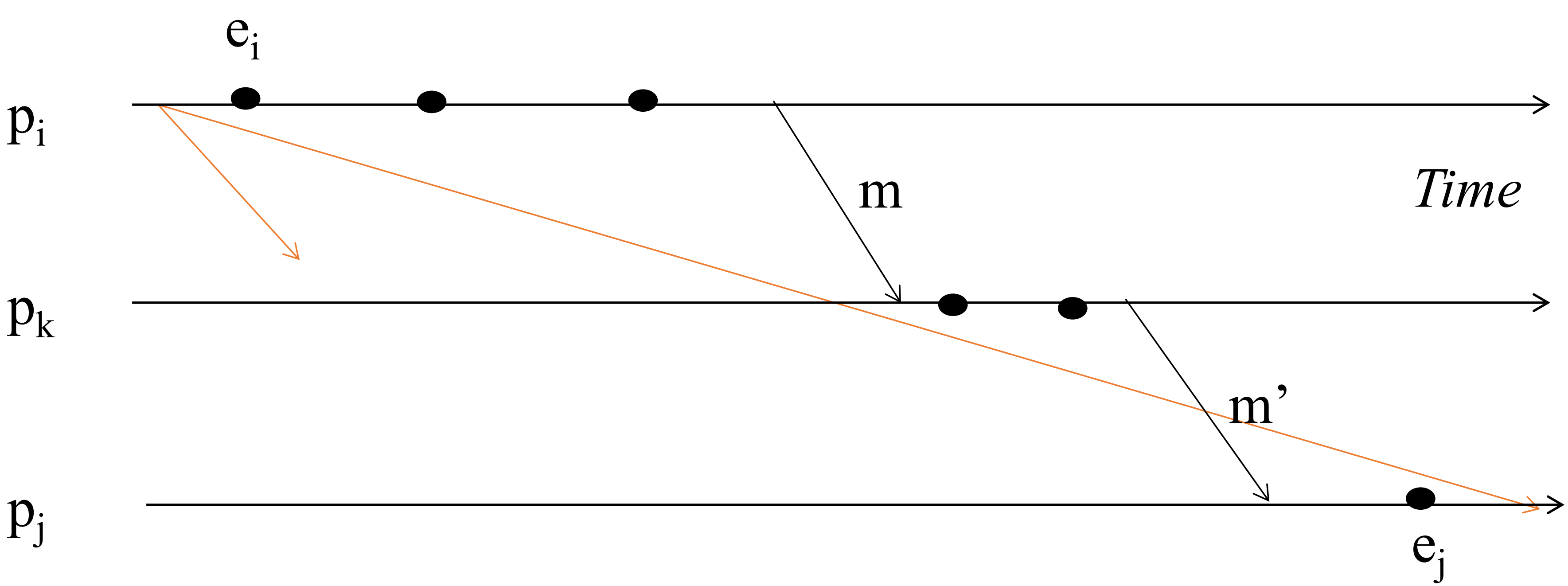
Chandy-Lamport Algorithm: Properties

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



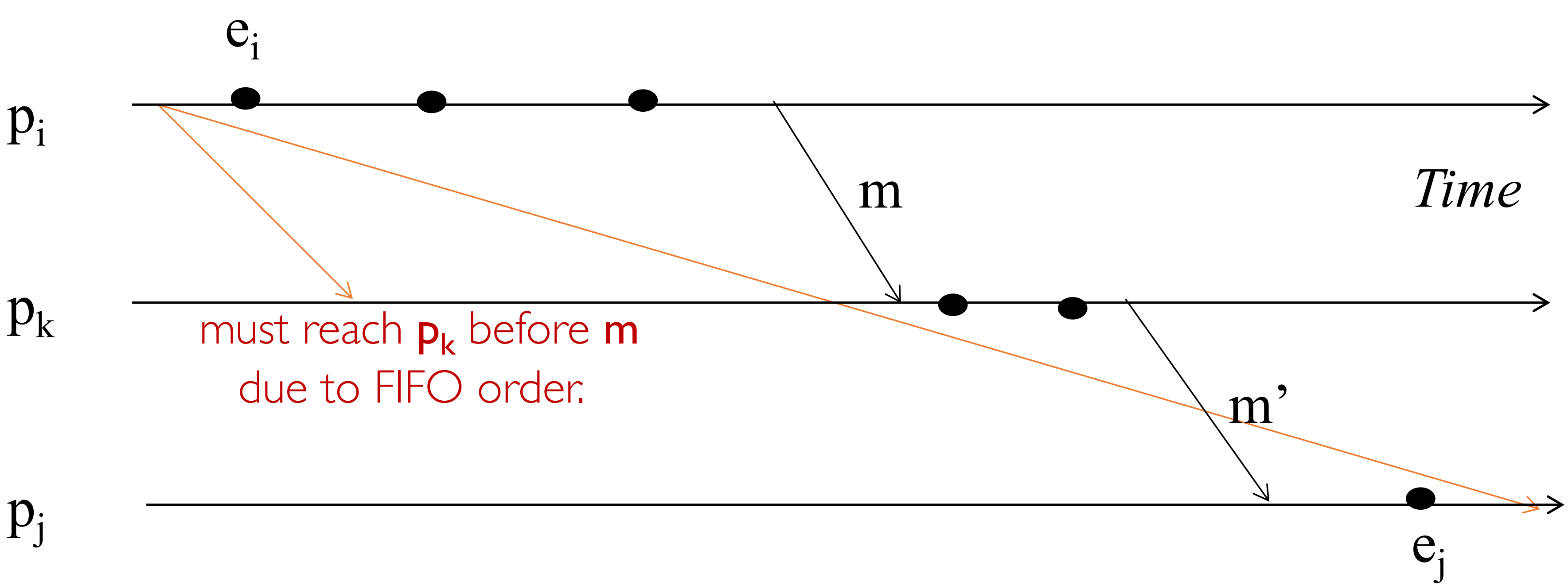
Chandy-Lamport Algorithm: Properties

- 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* $\mathbf{e_j}$ *→ <* $\mathbf{p_j}$ *records its state>, and <*$\mathbf{p_i}$ *records its state> →* $\mathbf{e_i}$.



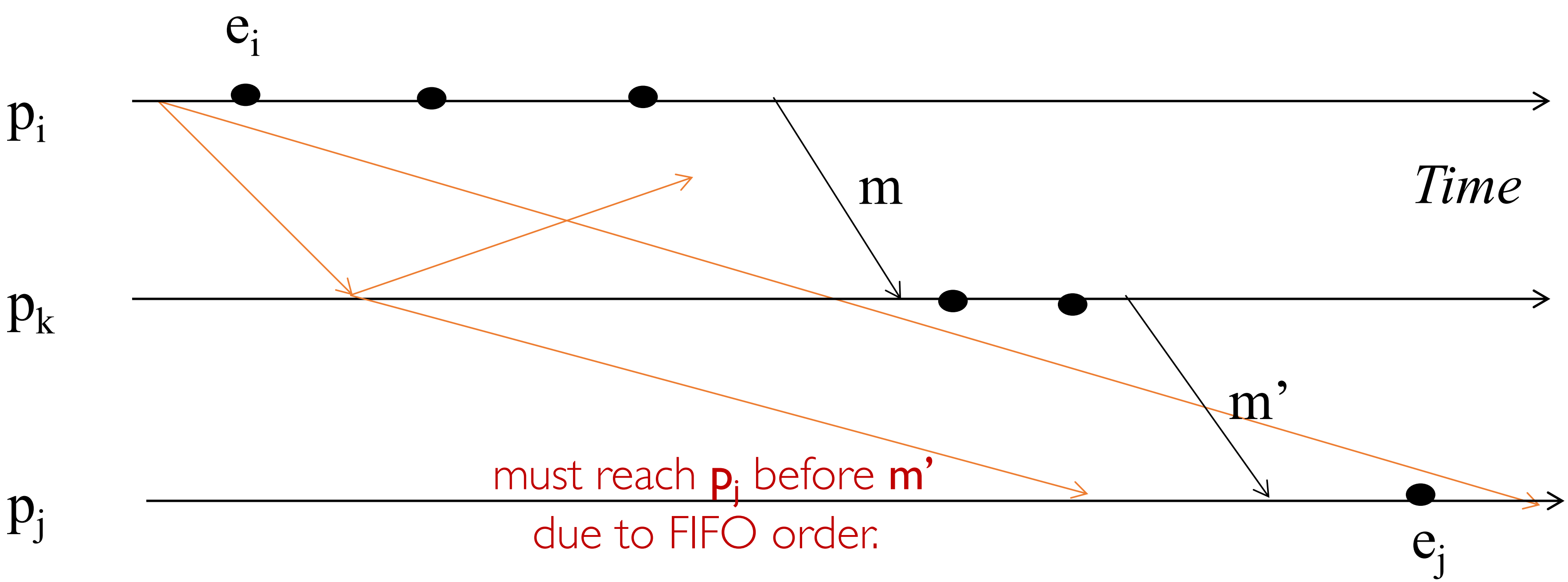
Chandy-Lamport Algorithm: Properties

- 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>* → $\mathbf{e_i}$*.*



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>* $\rightarrow \mathbf{e_i}$.
- 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> $\rightarrow \mathbf{e_i}$, 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.

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.

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

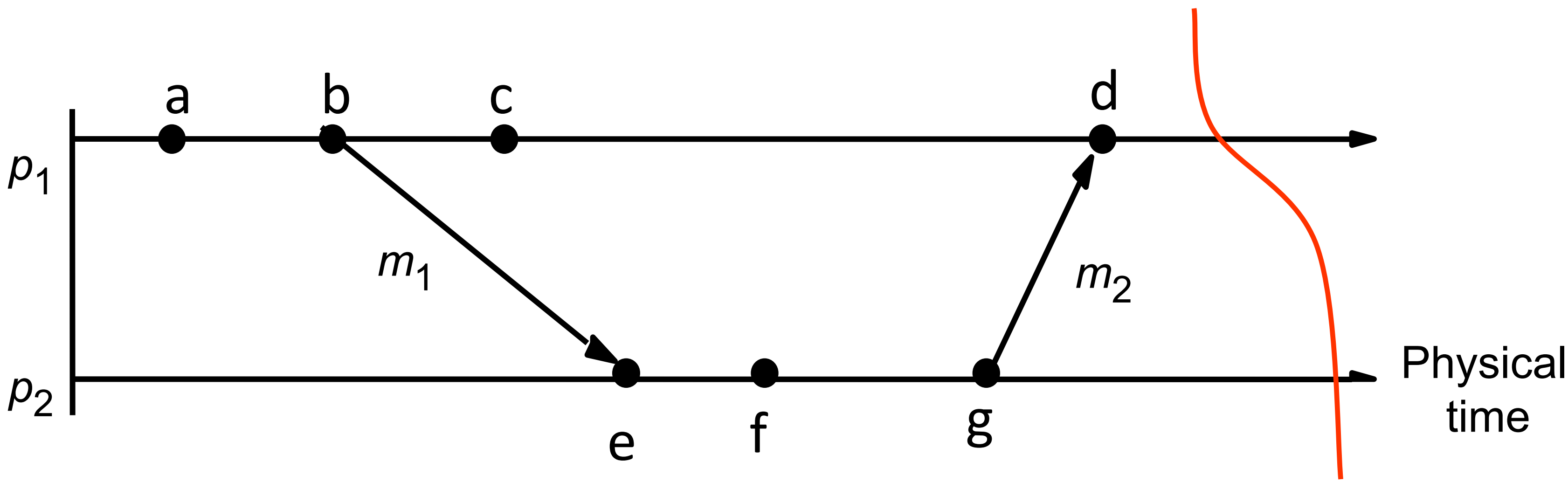
  the frontier of $C = \{e_i^{c_i}, i = 1,2, \ldots n\}$

  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



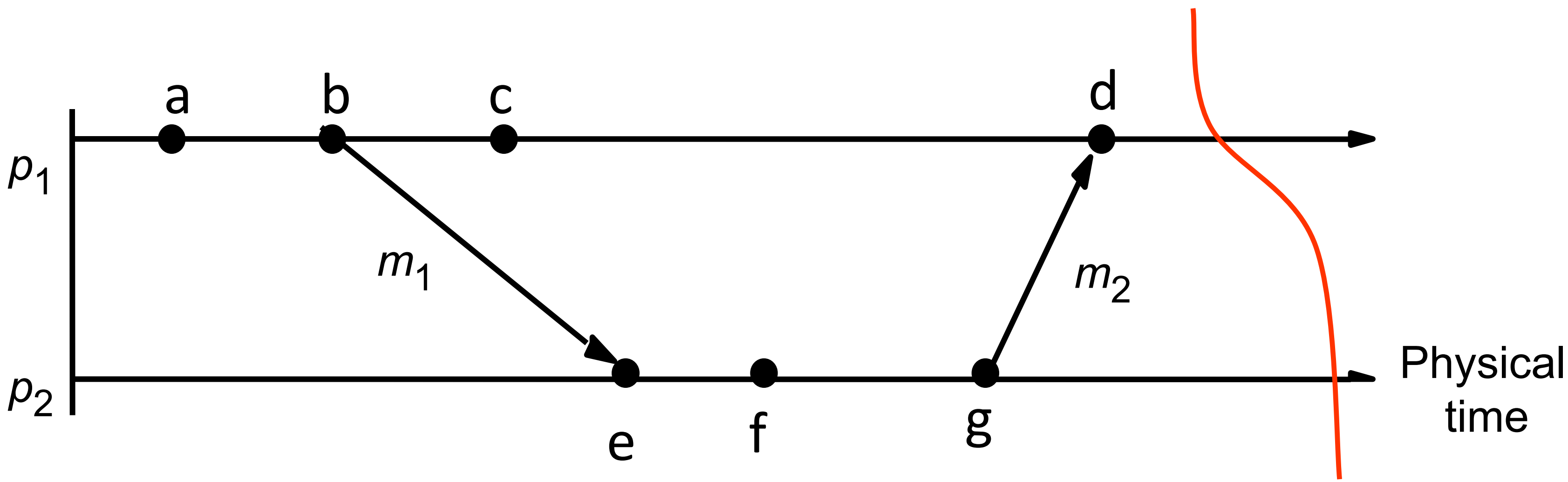
Order at p_1: < a, b, c, d >   Order at p_2: < e, f,g>

Causal order across p_1 and p_2: $b \rightarrow e$, $g \rightarrow d$

Run: < a, b, c, d , e, f, g>

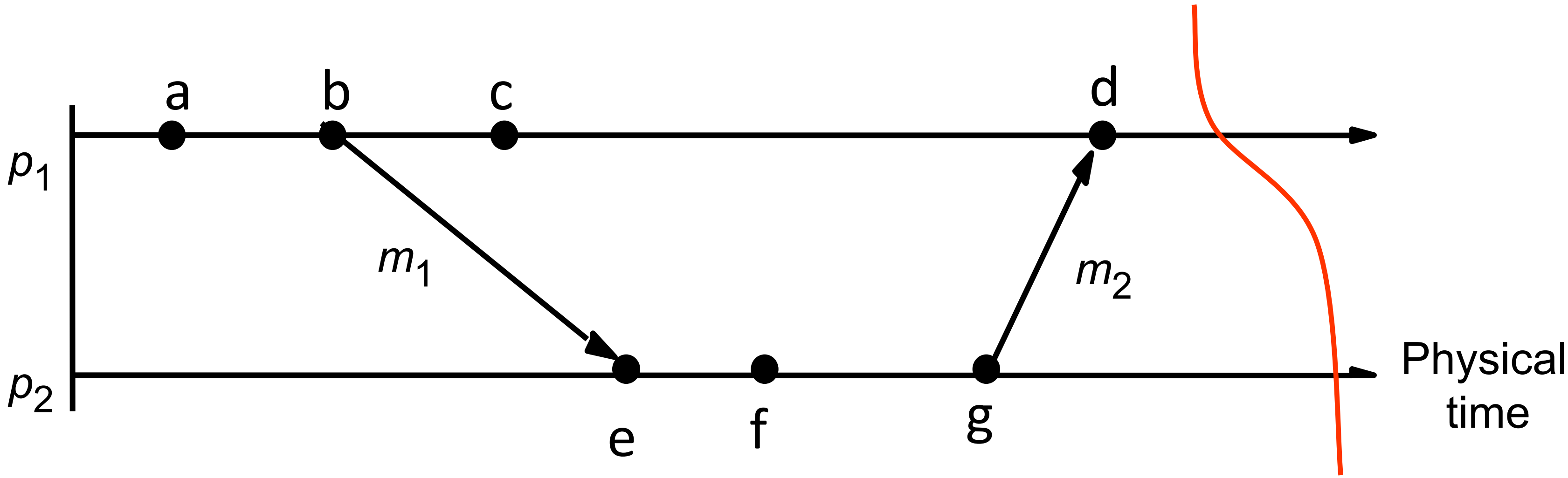
Linearization: <a, b, c, e, f, g , d >

Example



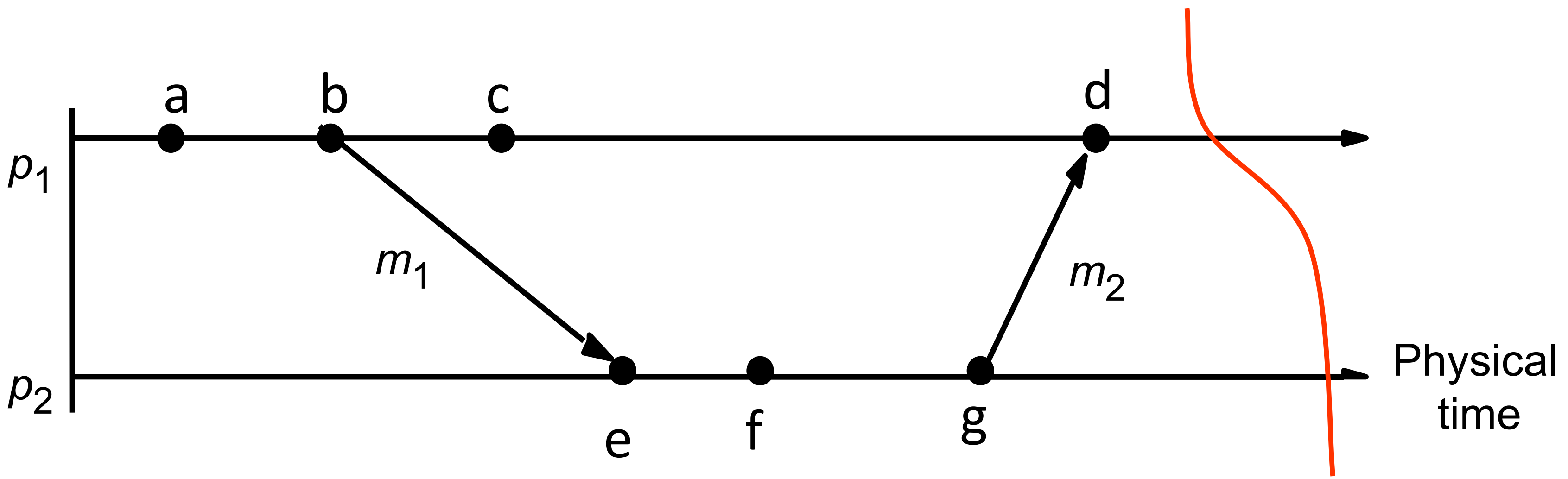
< a, b, e, f, c, g , d >:

Example



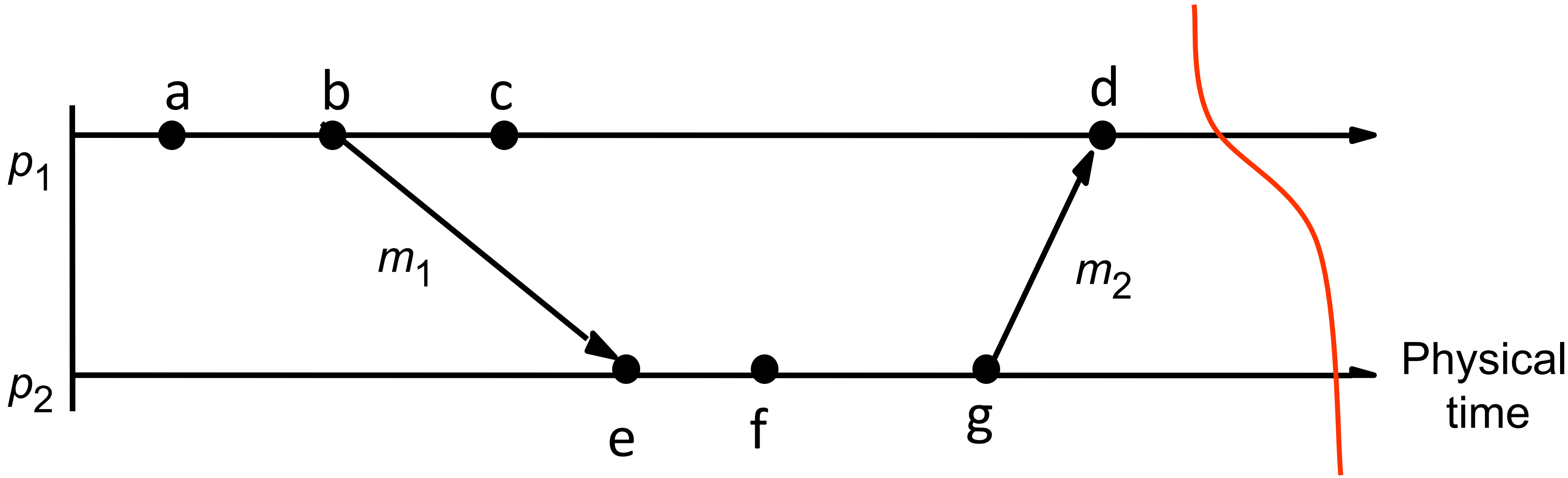
< a, b, e, f, c, g , d >: Linearization

Example



< a, f, e , b, c, g , d >:

Example

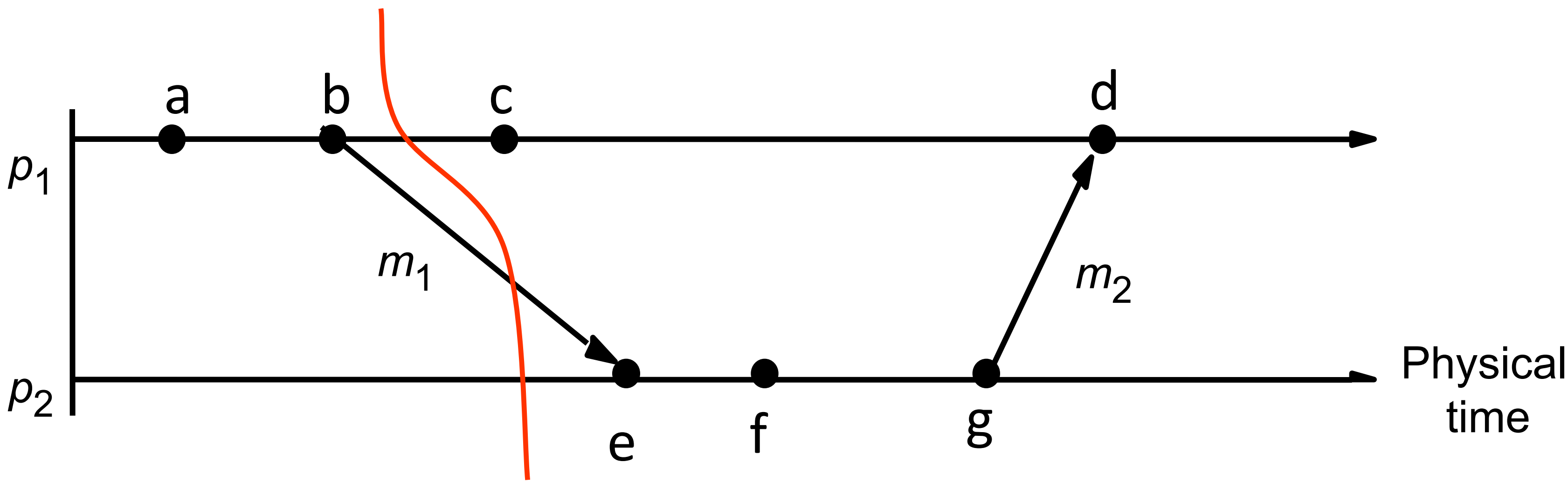


< a, f, e , b, c, g , d >: Not even a run

More notations and definitions

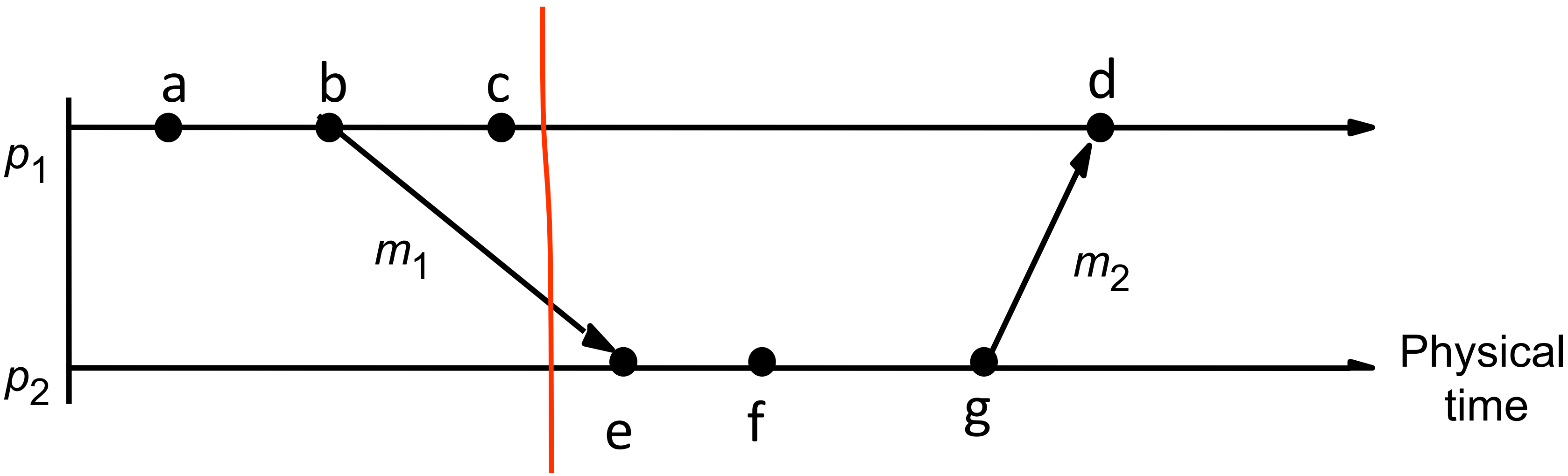
- A **run** is a total ordering of events in H that is consistent with each $\mathbf{h_i}$'s ordering.
- A **linearization** is a run consistent with happens-before ($\rightarrow$) relation in H.
- Linearizations pass through consistent global states.

Example



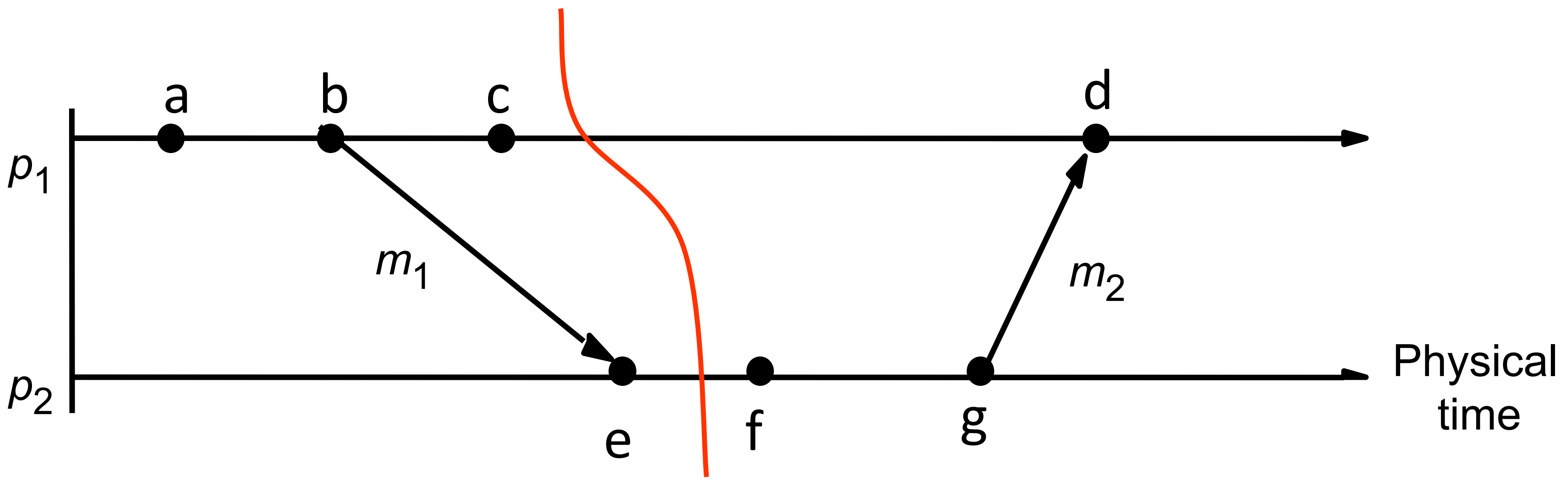
Linearization: < a, b, c, e, f, g , d >

Example



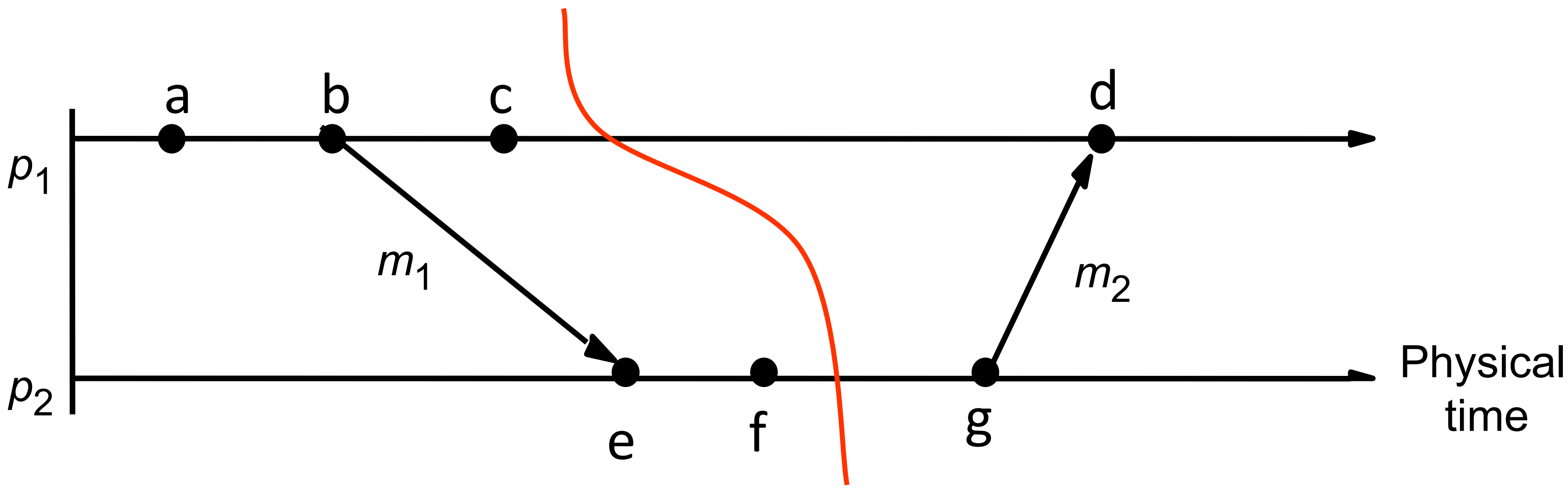
Linearization: < a, b, c, e, f, g , d >

Example



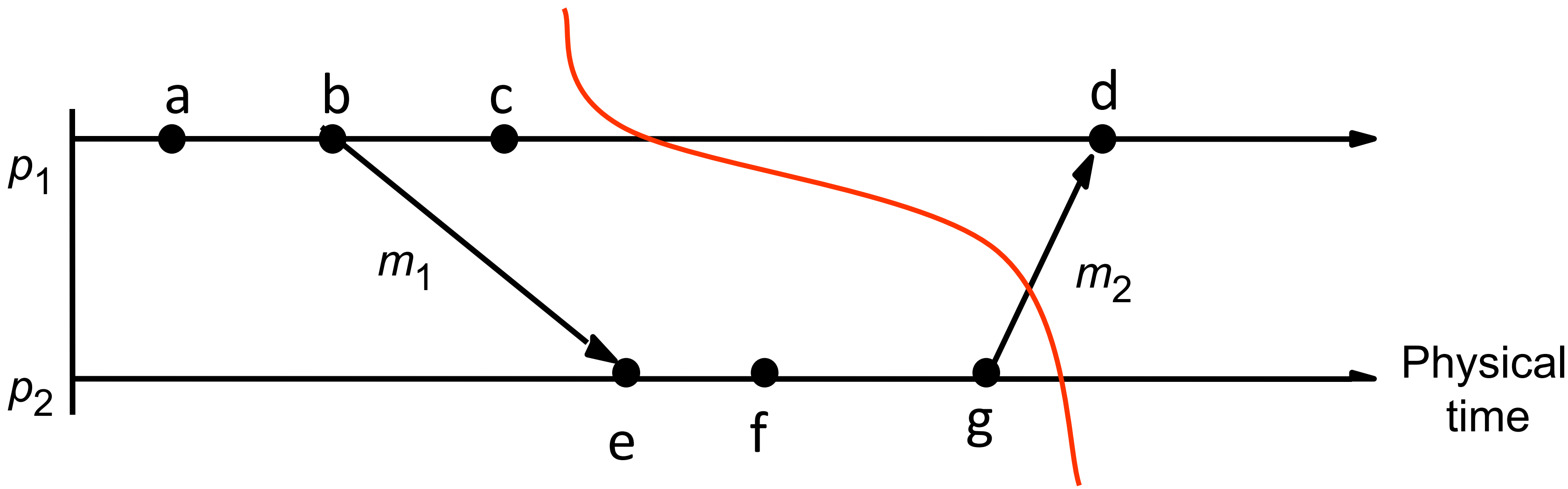
Linearization: < a, b, c, e, f, g , d >

Example



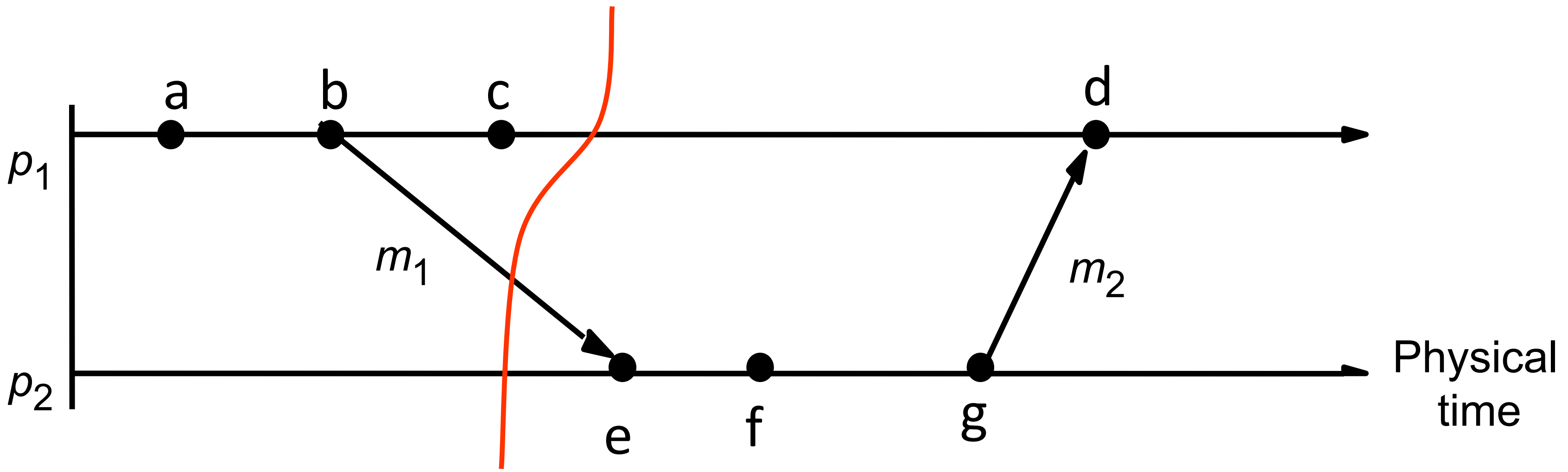
Linearization: < a, b, c, e, f, g , d >

Example



Linearization: < a, b, c, e, f, g , d >

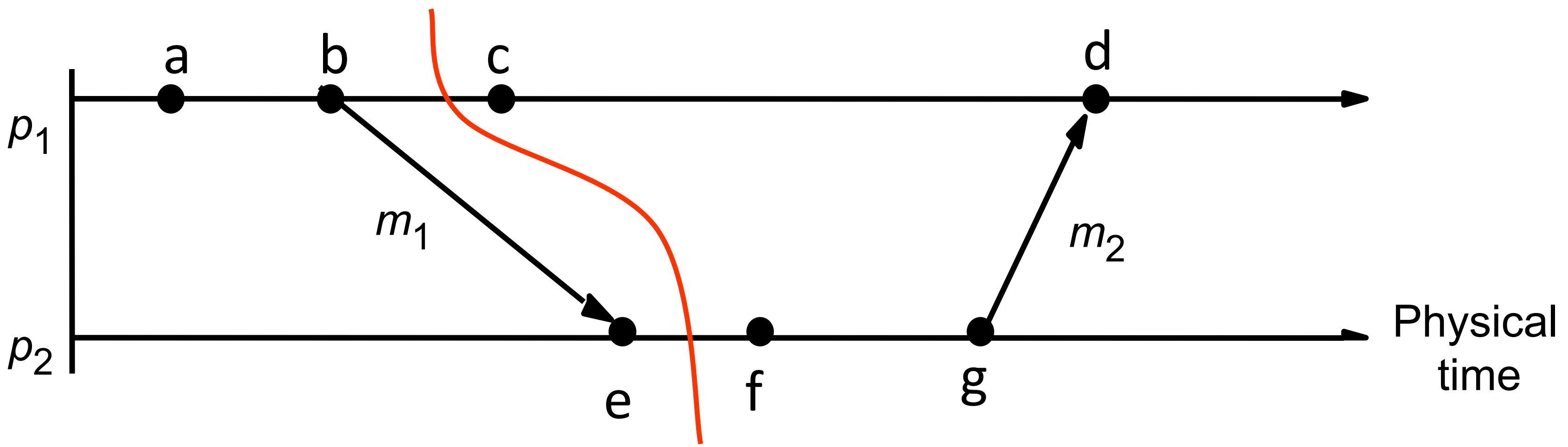
Example



Linearization: < a, b, c, e, f, g , d >

Linearization: < a, b, e, c, f, g , d >

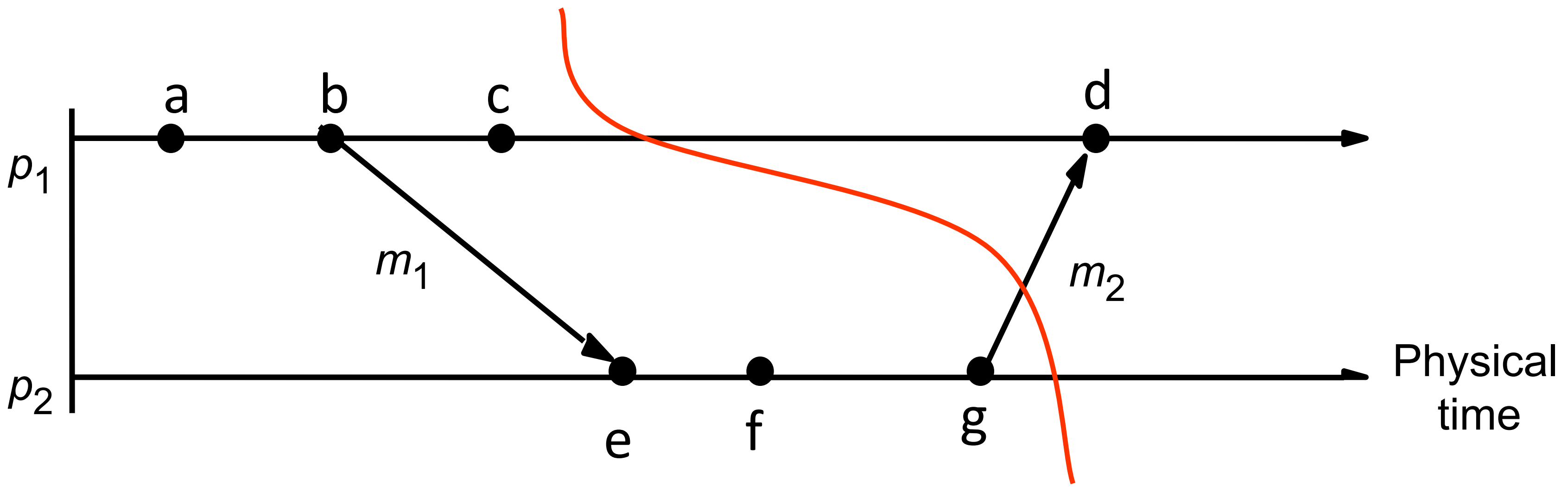
Example



Linearization: < a, b, c, e, f, g , d >

Linearization: < a, b, e, c, f, g , d >

Example



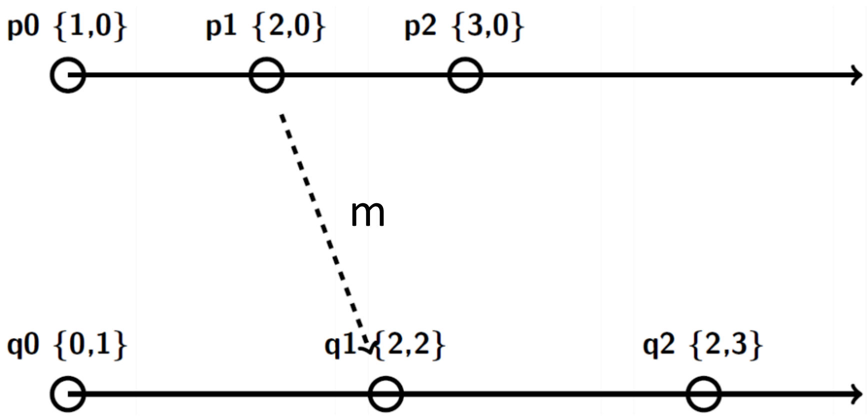
Linearization: < a, b, c, e, f, g | d >

Linearization: < a, b, e, c, f, g | d >

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



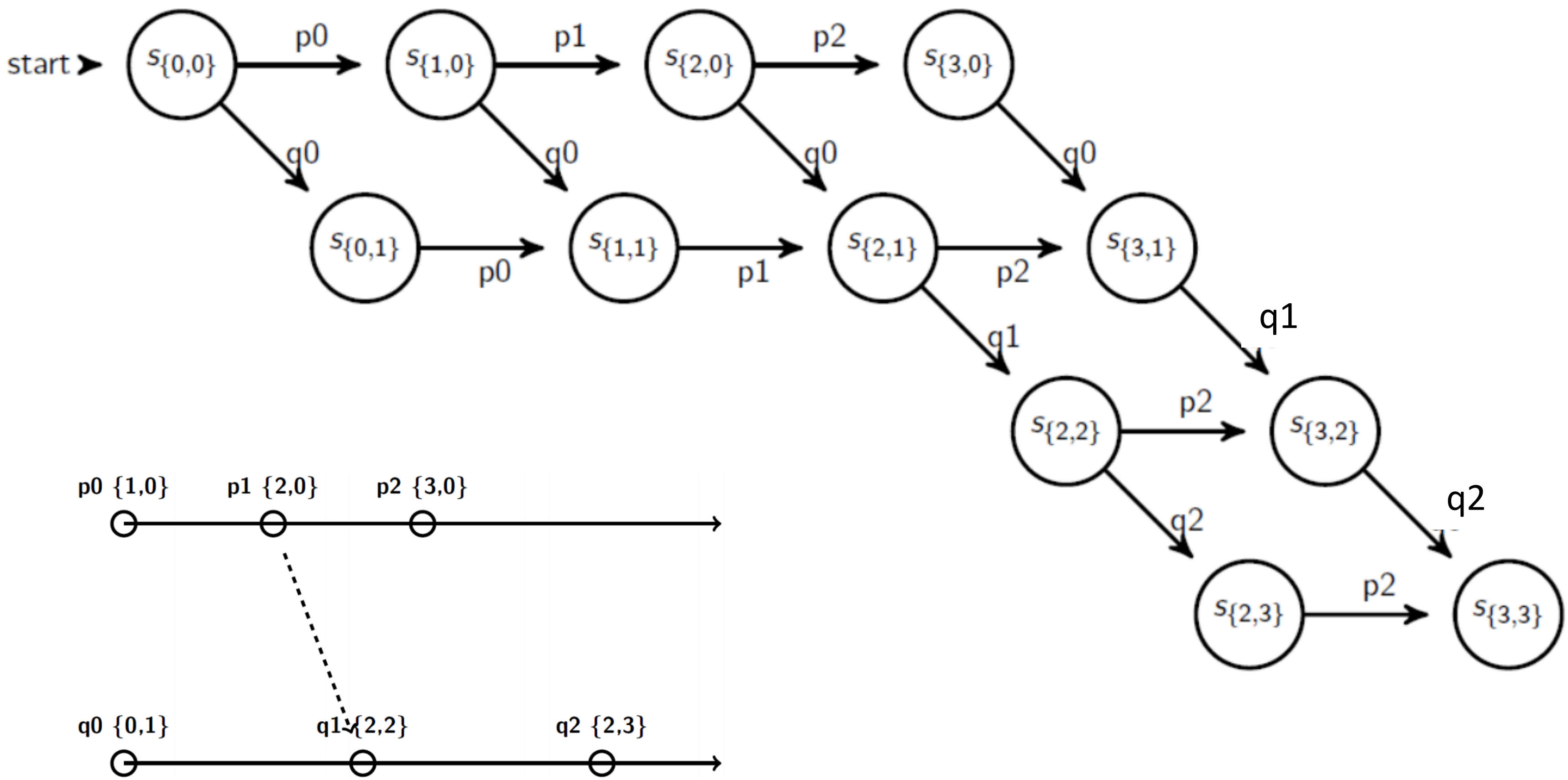
- Causal order:
  - p0 → p1 → p2
  - q0 → q1 → q2
  - p0 → p1 → q1 → q2
- Concurrent:
  - p0 || q0
  - p1 || q0
  - p2 || q0, p2 || q1, p2 || q2

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 >
- ……

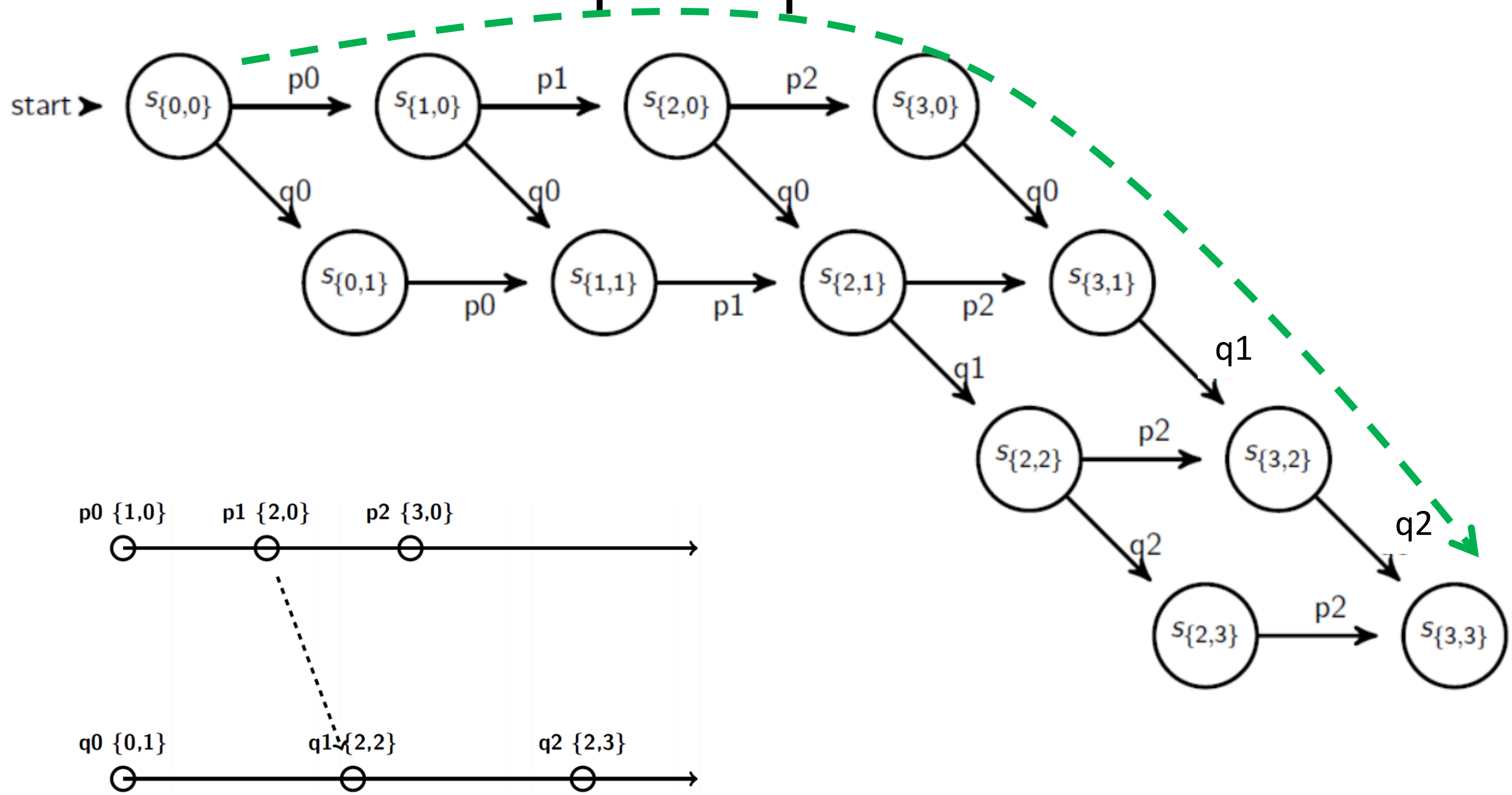
State Transitions: Example

Execution Lattice. Each path represents a linearization.



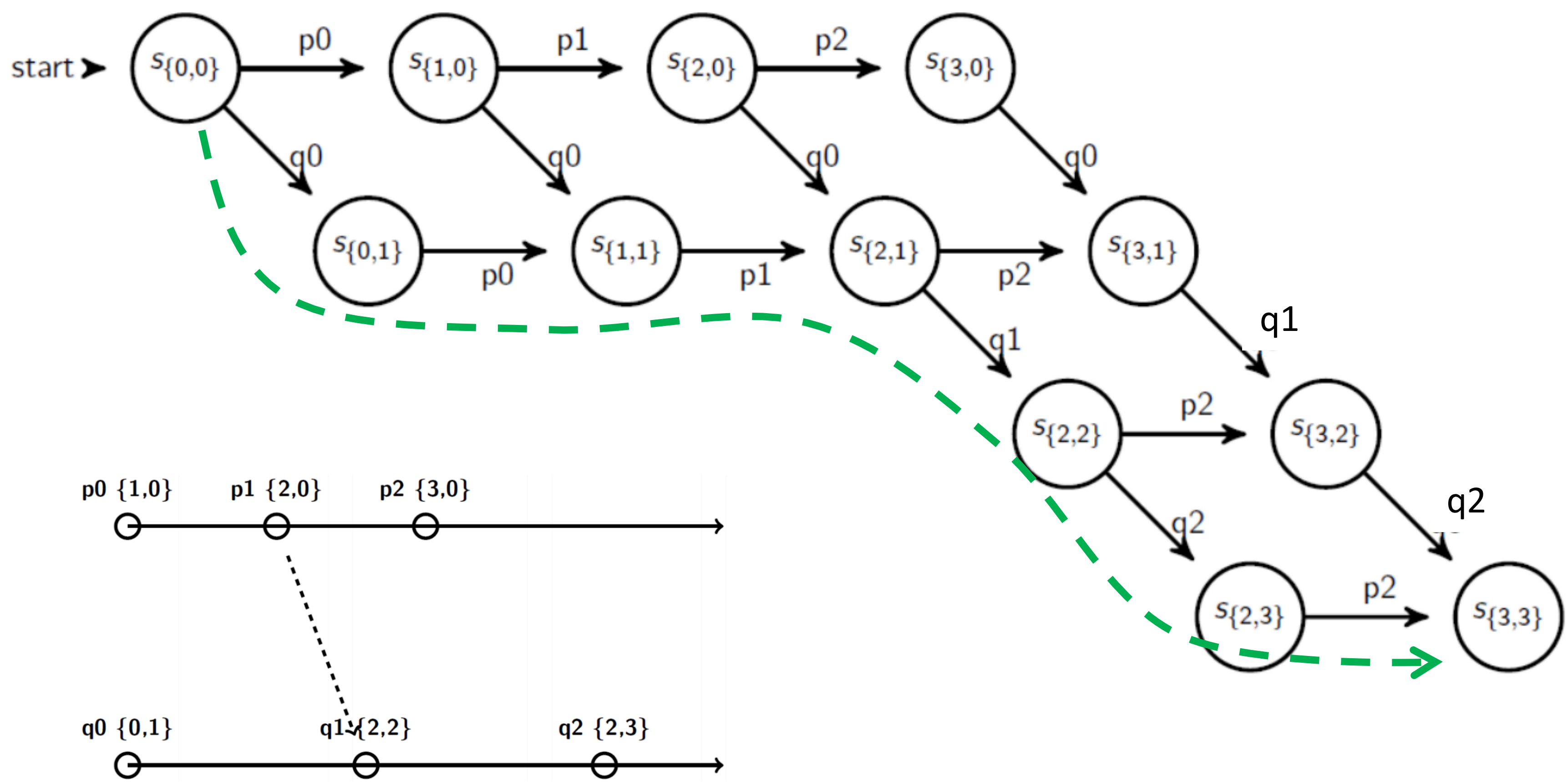
State Transitions: Example

Execution Lattice. Each path represents a linearization.



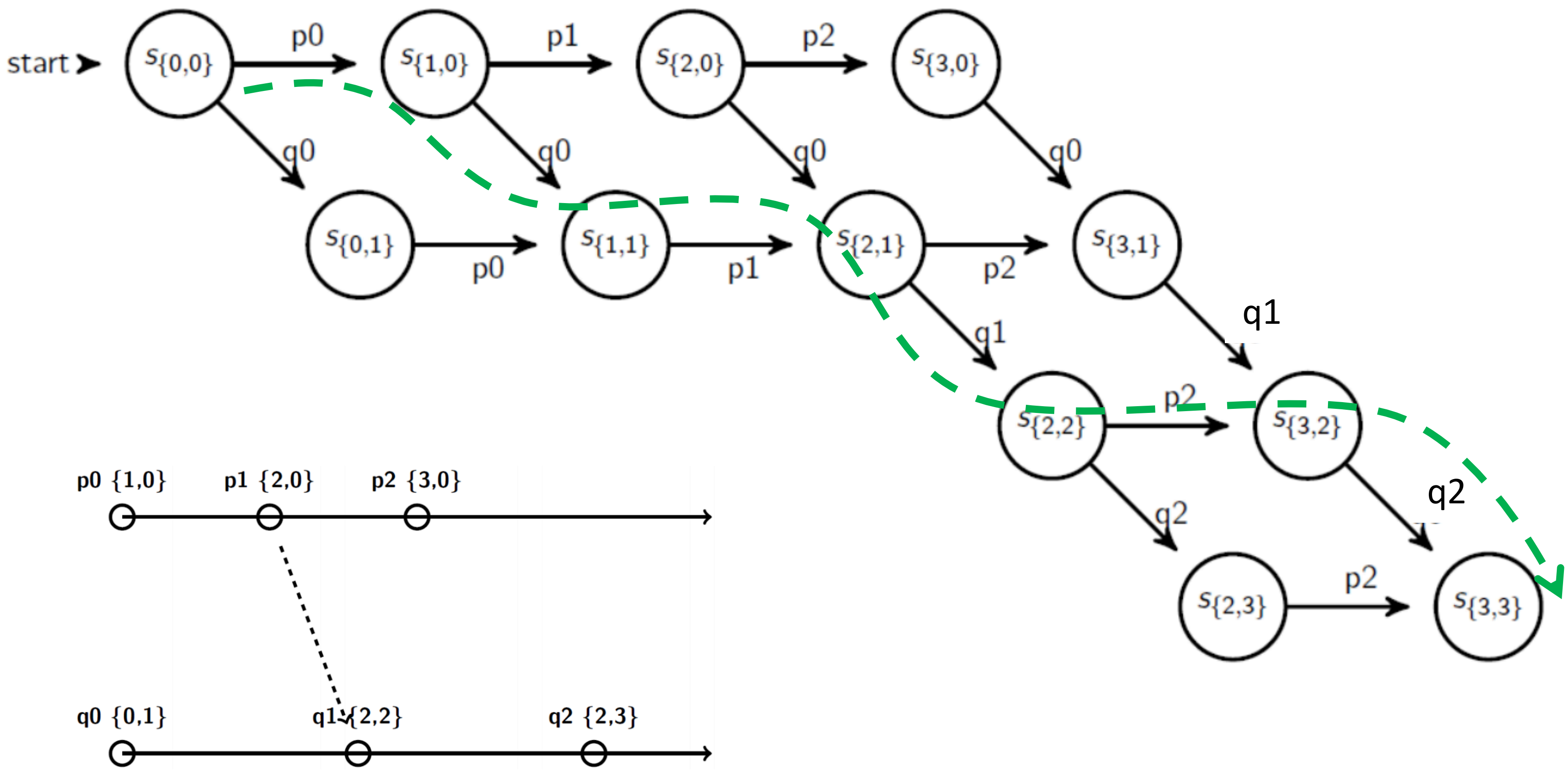
State Transitions: Example

Execution Lattice. Each path represents a linearization.



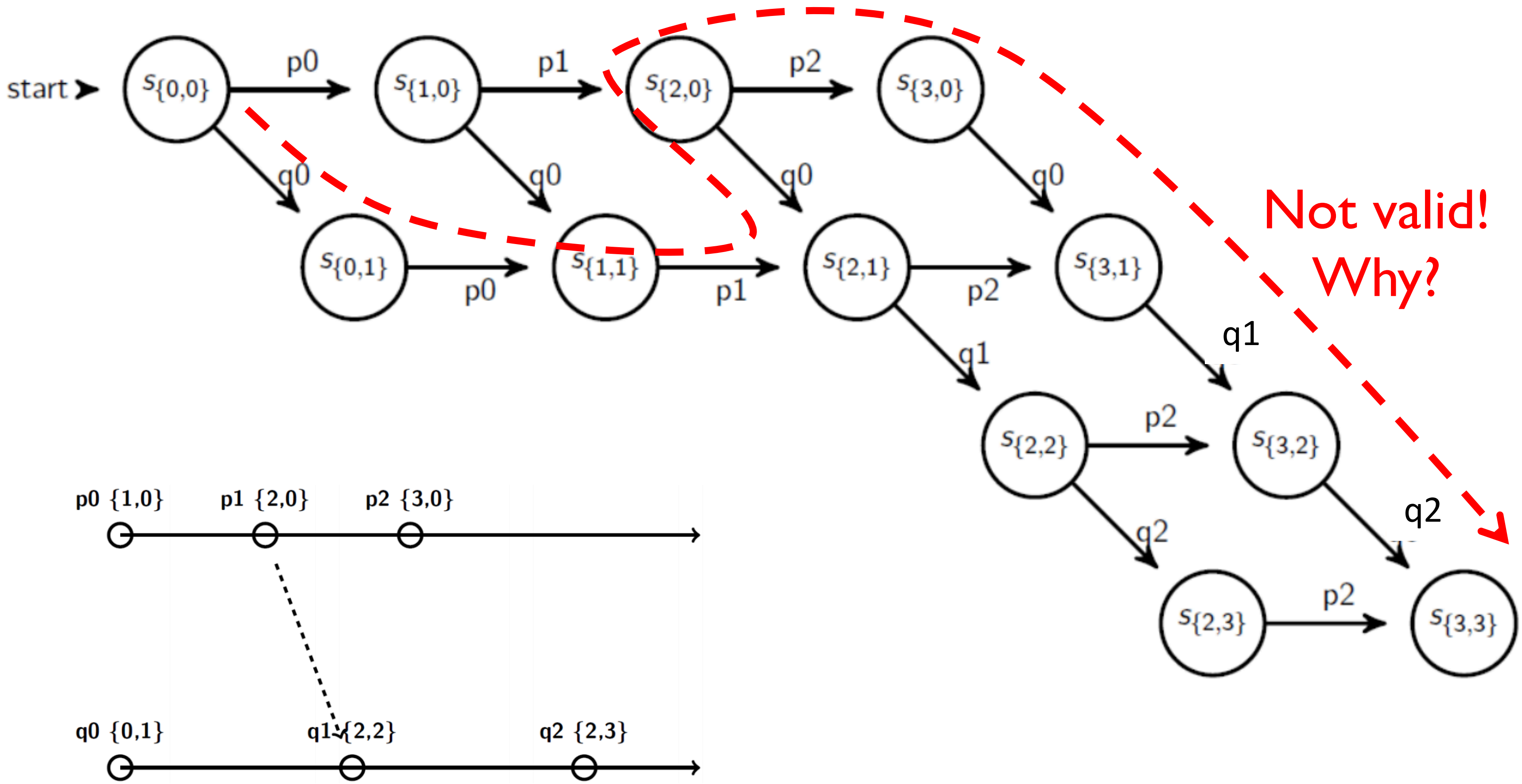
State Transitions: Example

Execution Lattice. Each path represents a linearization.

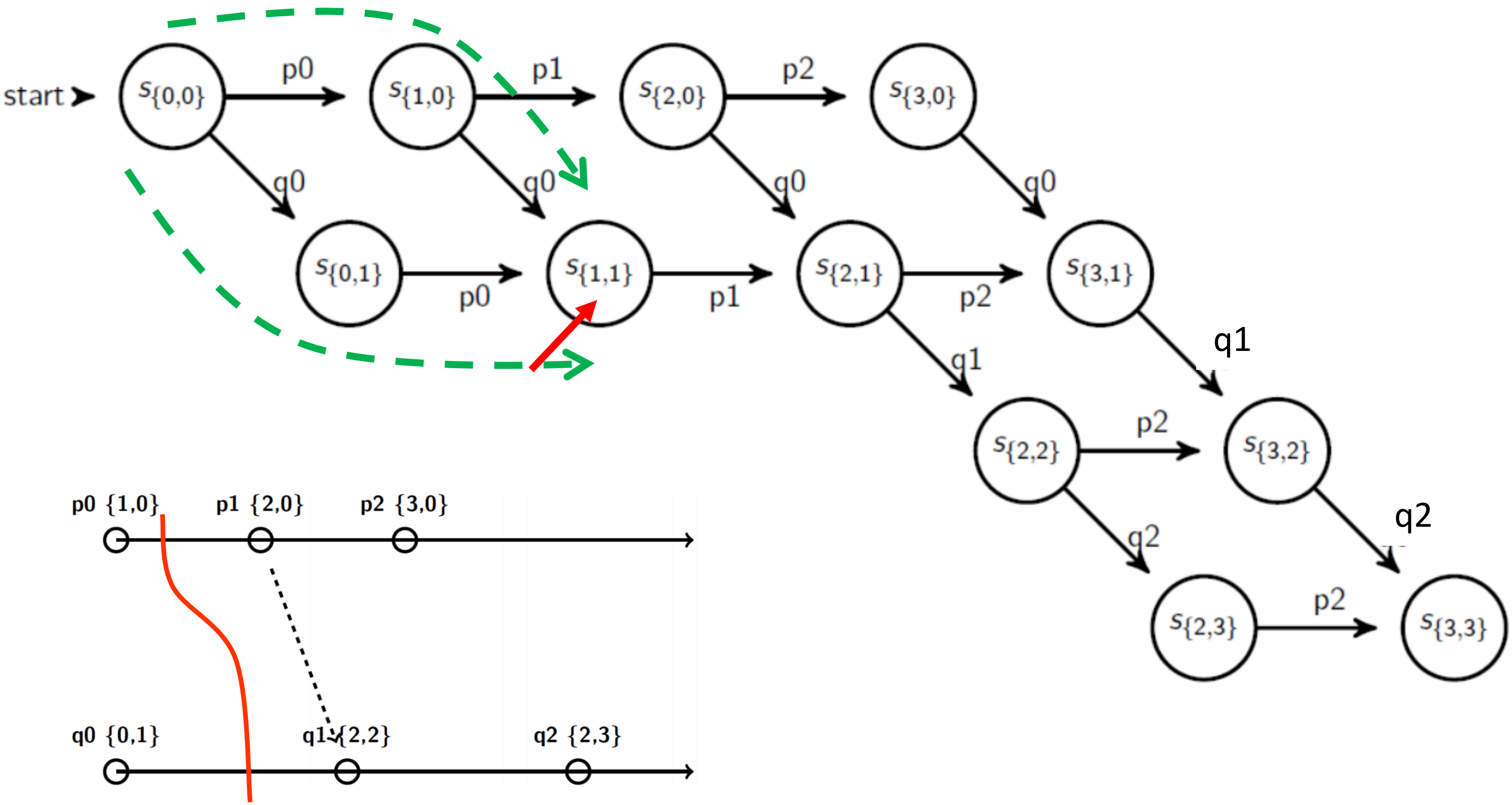


State Transitions: Example

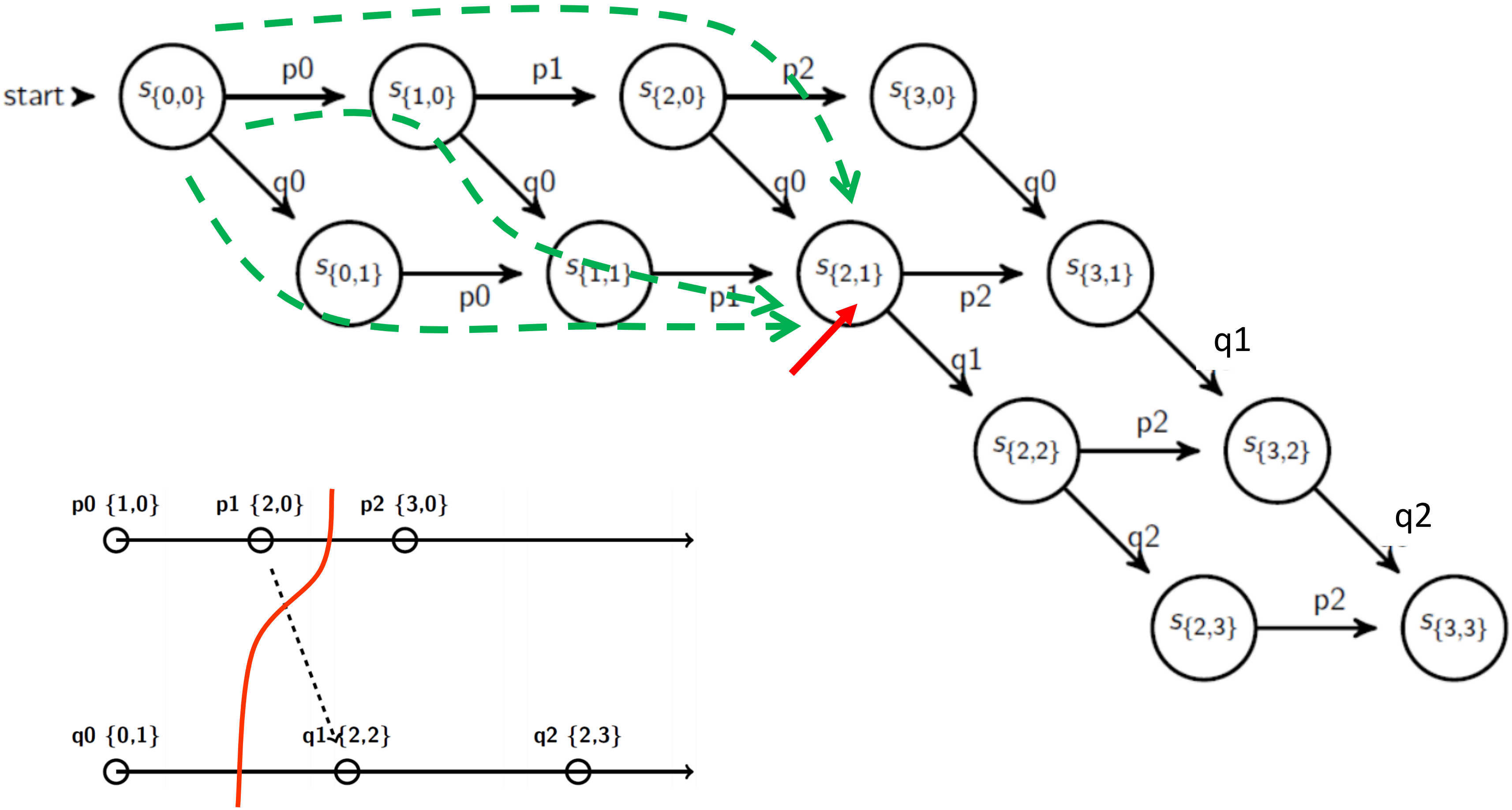
Execution Lattice. Each path represents a linearization.



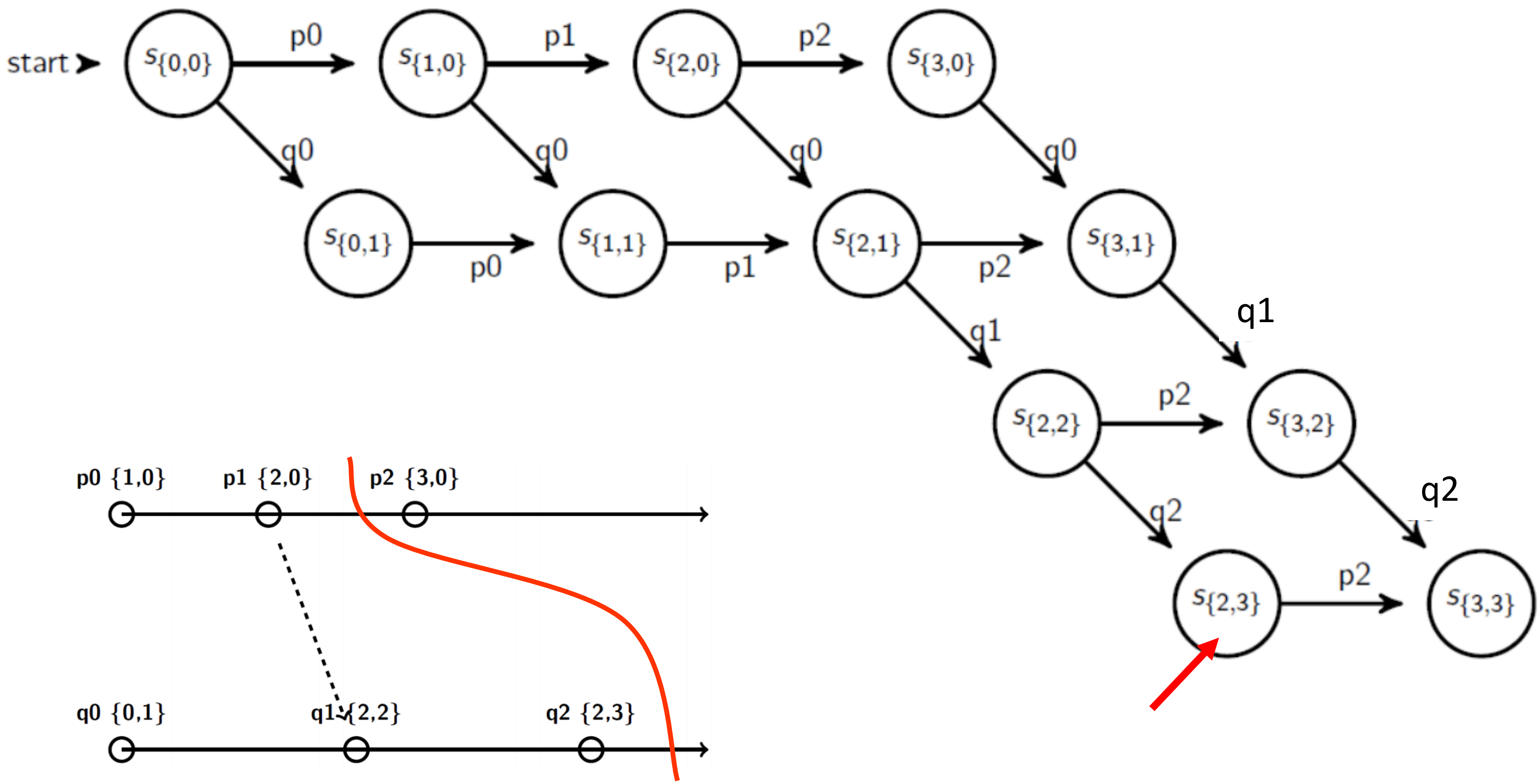
State Transitions: Example



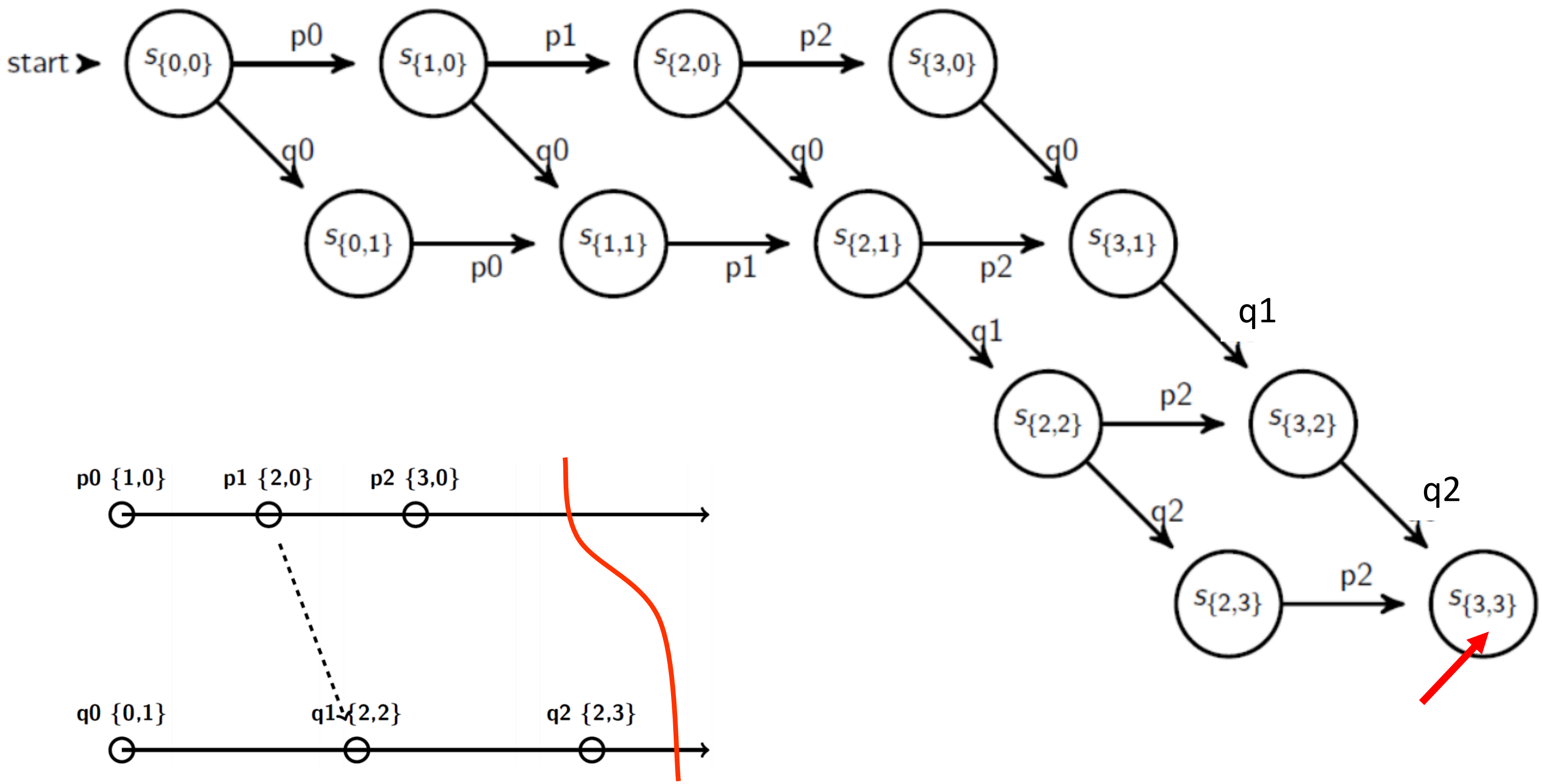
State Transitions: Example



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 $\mathbf{h_i}$'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 , ….

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
- To be continued in next class…..