

CS 425 / ECE 428

Distributed Systems

Fall 202

Aishwarya Ganesan & Ram Kesavan

Lecture 13: Snapshots

Ack: Indranil Gupta

All slides © IG

Announcements

- HW2 solutions will be released today
- MP2 grades have been released
- In-class midterm Thursday 10/8
 - Bring pencil+eraser for MCQ bubble sheet
 - Course webpage specifies the two rooms
 - **Make sure you're in correct classroom for the midterm**
 - Piazza post with more details
 - Practice mid-term released

A Local Snapshot

- EoY/EoQ factory inventory audit
- Once work stops on the last Friday of the Y/Q
 - Assembly lines frozen
 - Employees walk around to count all components and completed products sitting on shelves/containers/etc.
 - Stitch together all counts
- Result: Accurate frozen picture of the inventory

Distributed Snapshot

- Inventory audit across many factories across the globe
- But, global supply chain never stops moving & factories are in different time zones
- Each location $\langle X \rangle$:
 - Does its inventory at local midnight
 - Labels all outgoing boxes with “captured in Q3 audit at location $\langle X \rangle$ ”
 - If it receives a box with no label, counts it locally
- Global logically consistent inventory picture
 - Without needing to temporarily shut down factories

In a Distributed System

- **In the cloud: application/service is running on multiple servers**
- **Servers handle concurrent events and interact with each other**
- **The ability to obtain a “global picture” of the system is important**
- **Some uses of having a global picture of the system**
 - *Checkpointing*: can restart distributed application on failure
 - *Garbage collection* of objects: objects at servers that don't have any other objects (at any servers) with pointers to them
 - Deadlock detection: Useful in database transaction systems
 - Termination of computation: Useful in batch computing systems like Folding@Home, SETI@Home

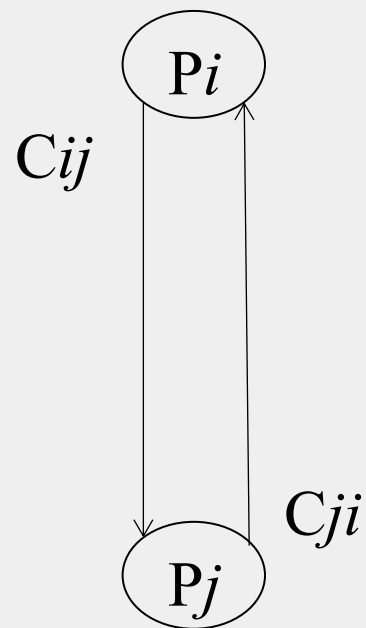
What's a Global Snapshot?

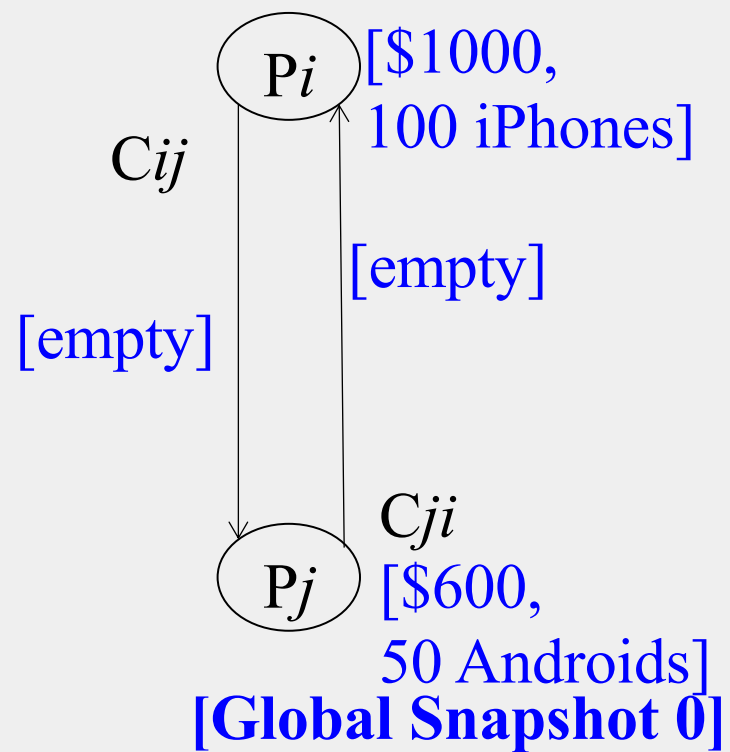
- **Global Snapshot = Global State =**
Individual state of each process in the distributed system
+
Individual state of each communication channel in the distributed system
- Capture the *instantaneous state* of each process
- And the instantaneous *state* of each communication channel, i.e., *messages* in transit on the channels

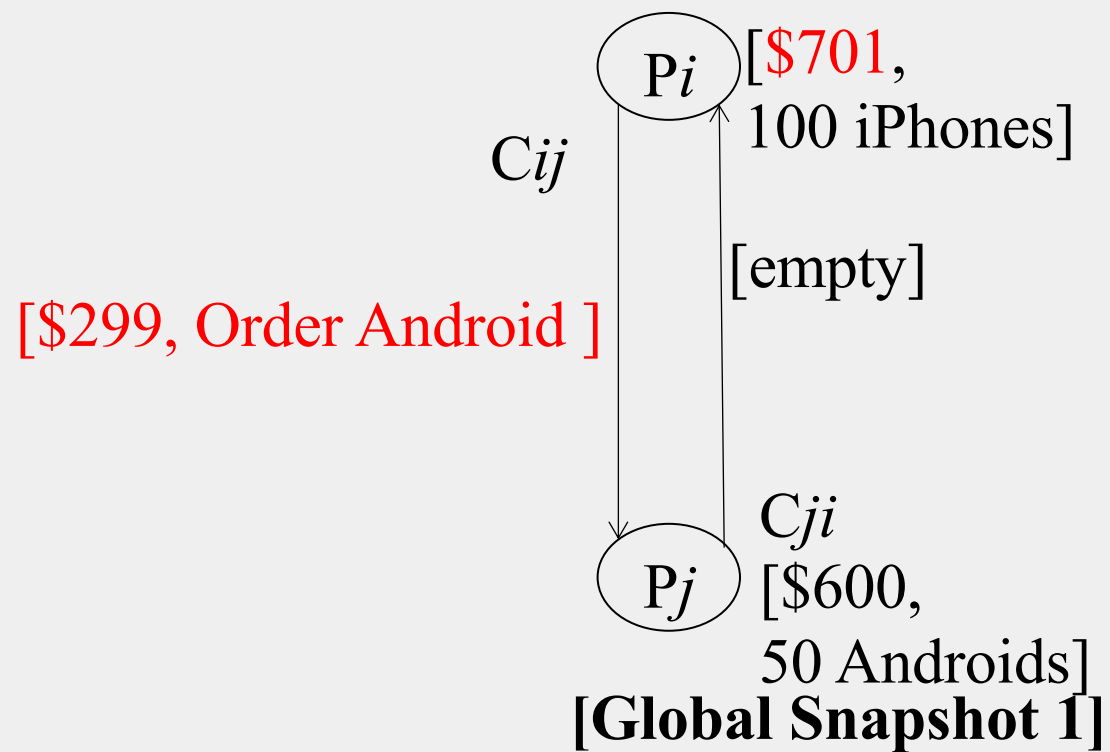
Obvious First Solution

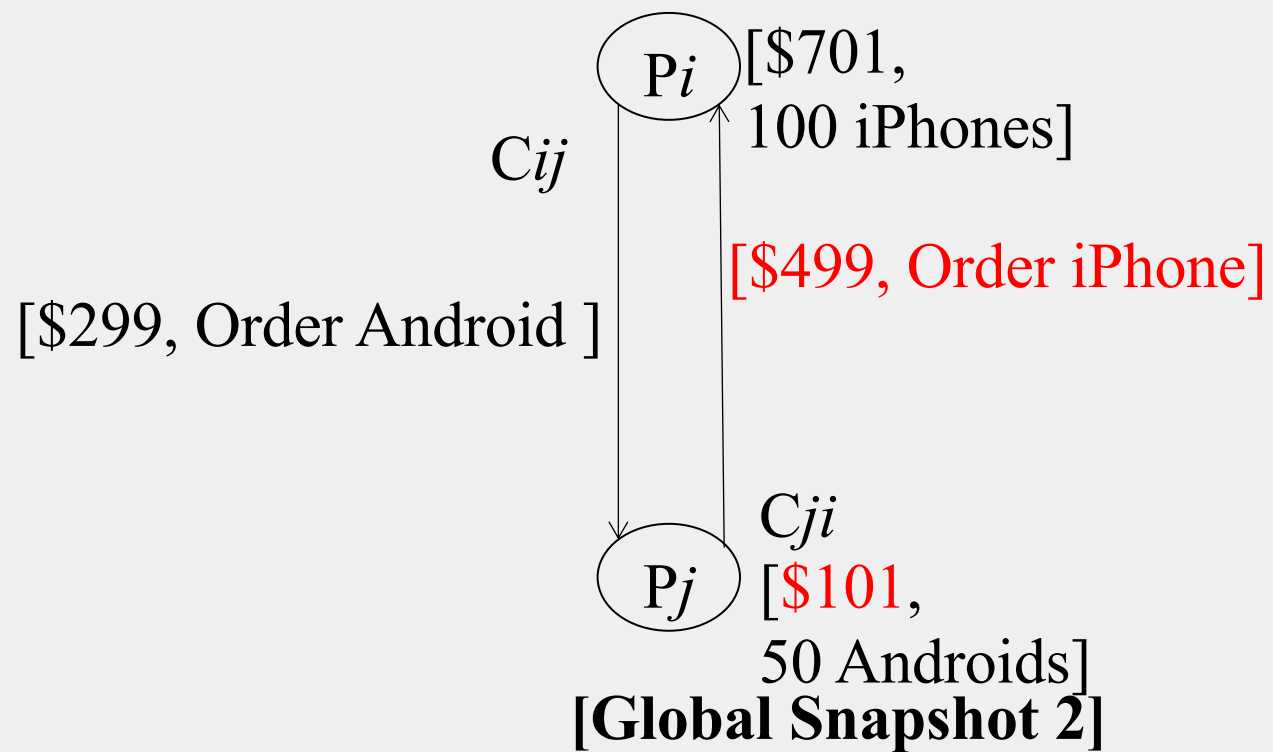
- Synchronize clocks of all processes
- Ask all processes to record their states at known time t
- Problems?
 - Time synchronization is not perfect
 - Also, does not record the state of messages in the channels
- Synchronization not required – causality is enough!

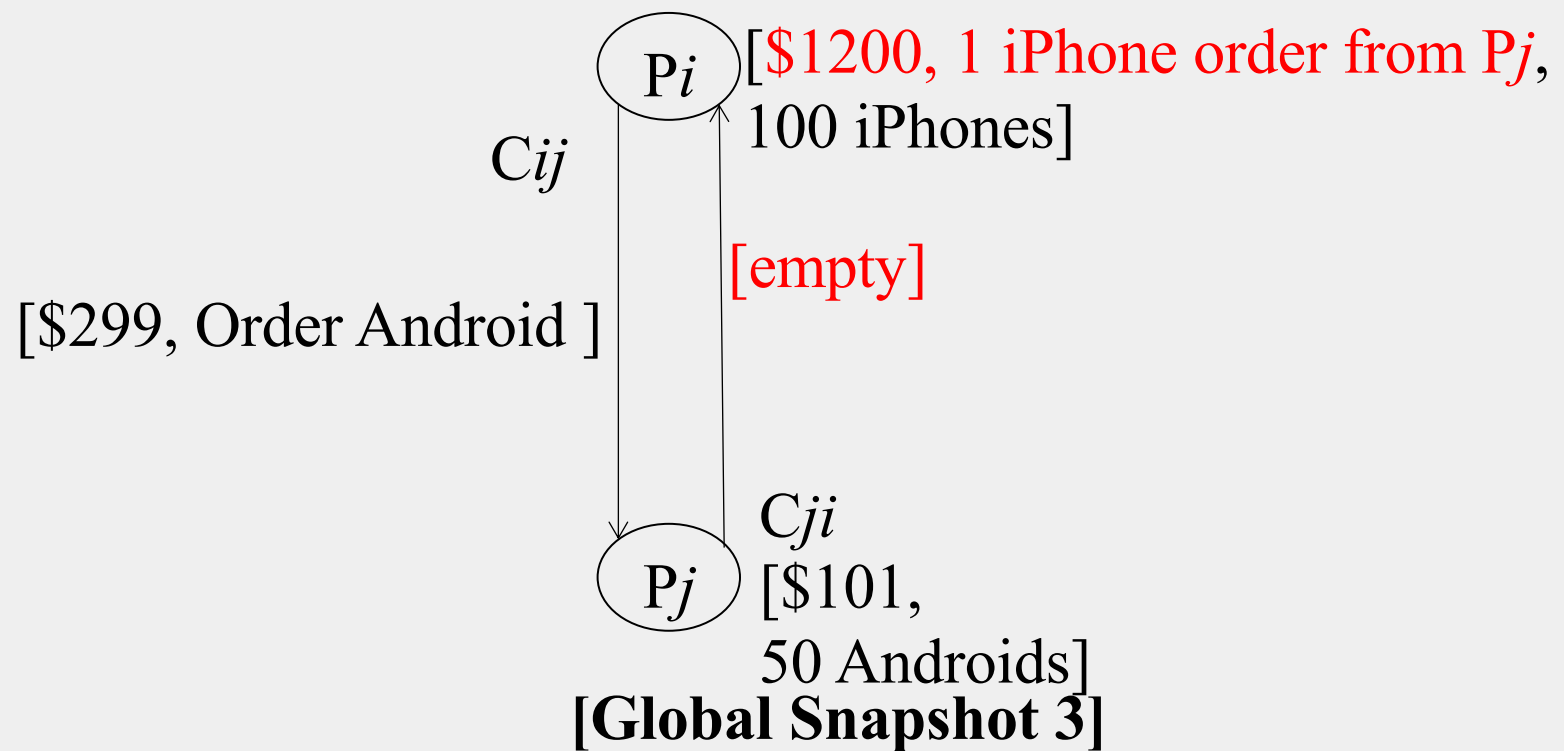
Example



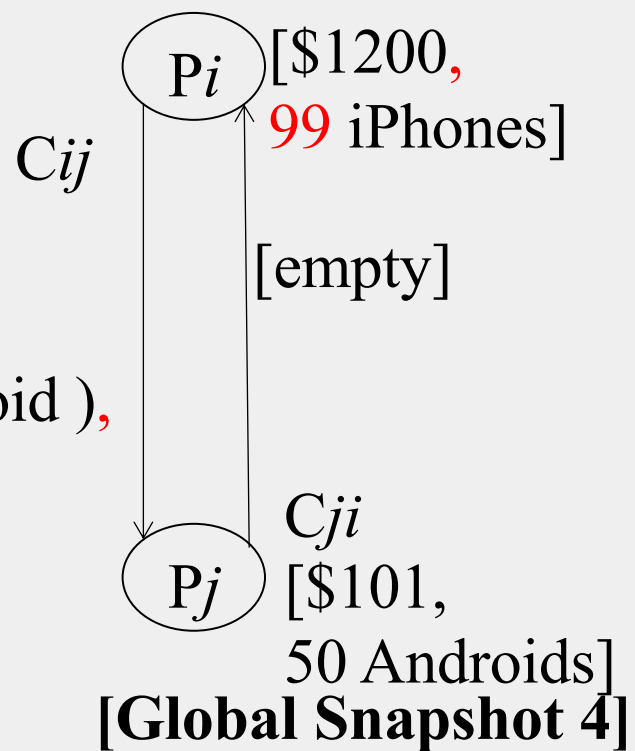




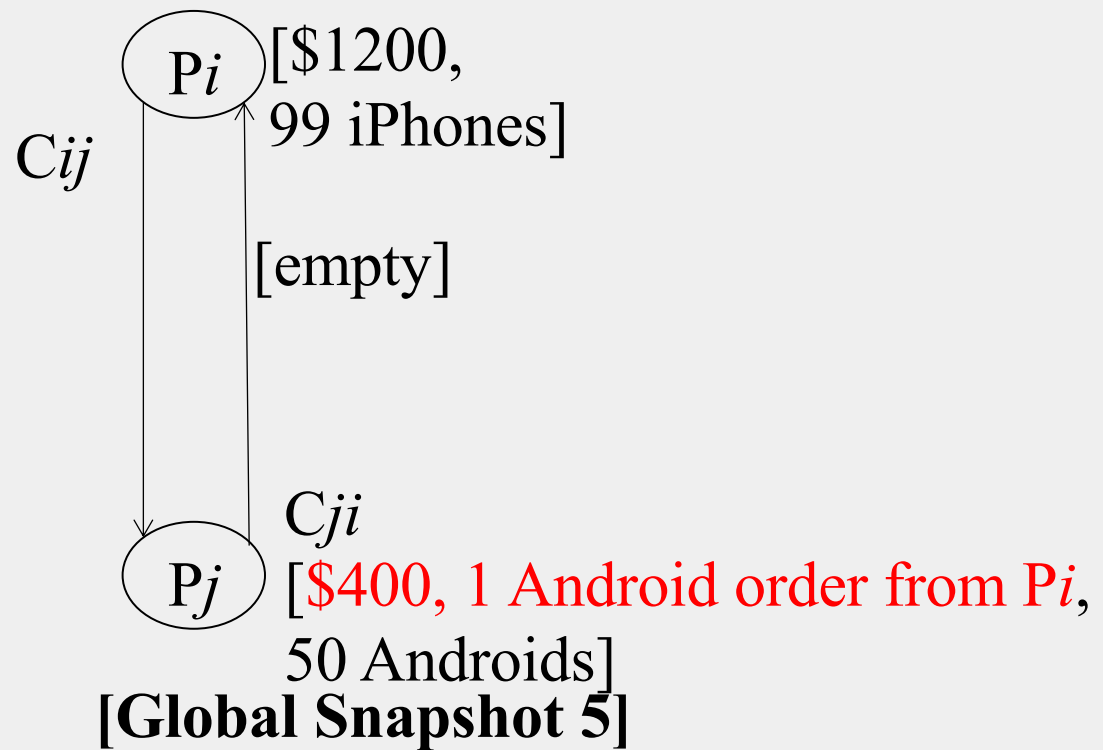


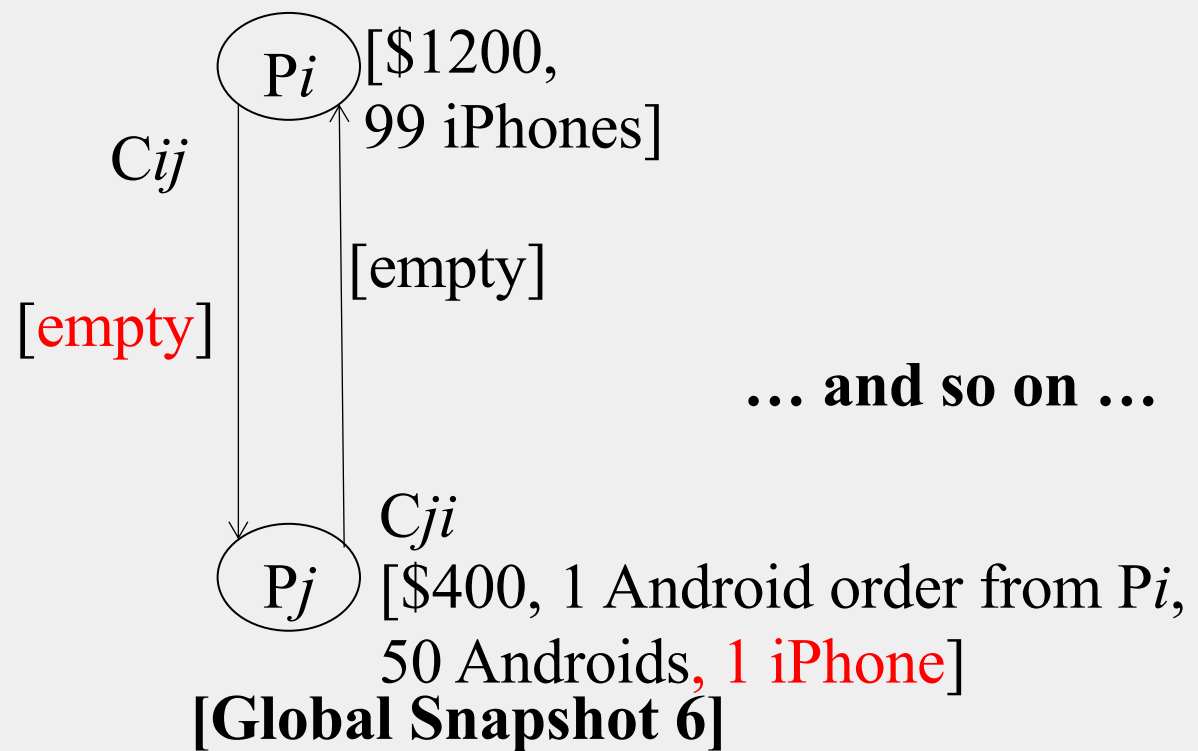


[
(\$299, Order Android),
(1 iPhone)
]



[
(1 iPhone)
]





Moving from State to State

- **Whenever an event happens anywhere in the system, the global state changes**
 - Process receives message
 - Process sends message
 - Process takes a step
- **State to state movement obeys causality**
 - Next: Causal algorithm for Global Snapshot calculation

System Model

- **Problem:** Record a global snapshot (state for each process, and state for each channel)
- **System Model:**
 - N processes in the system
 - There are two uni-directional communication channels between each process pair : $P_j \rightarrow P_i$ and $P_i \rightarrow P_j$
 - Communication channels are FIFO-ordered
 - First in First out in each direction
 - Does not apply across channels
 - No failure
 - All messages arrive intact, and are not duplicated and dropped
 - Other papers later relaxed some of these assumptions

Requirements

- **Snapshot should not interfere with normal application actions.**
- **Snapshot should not require application to pause the sending or receiving messages**
- **Each process is able to record its own state**
 - Process state: Application-defined state or, in the worst case:
 - its heap, registers, program counter, code, etc. (essentially the coredump)
- **Any process may initiate the snapshot**
 - We'll assume just one snapshot run for now
- **Global state can be collected asynchronously in a distributed fashion**

Chandy-Lamport Global Snapshot Algorithm

Initiator P_i **records** its own state

Initiator P_i creates special messages called “**Markers**”

- Not an application message, does not interfere with application messages

for $j=1$ **to** N **except** i

P_i **sends** out a Marker message on outgoing channel C_{ij}

for $j=1$ **to** N **except** i

Starts recording all incoming messages on each of the incoming channels at P_i : C_{ji}

Chandy-Lamport Global Snapshot Algorithm (2)

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
 - **Sets 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**
 - **Stops recording** and **records** the state of channel C_{ki} = all messages that arrived on it since recording was turned on.

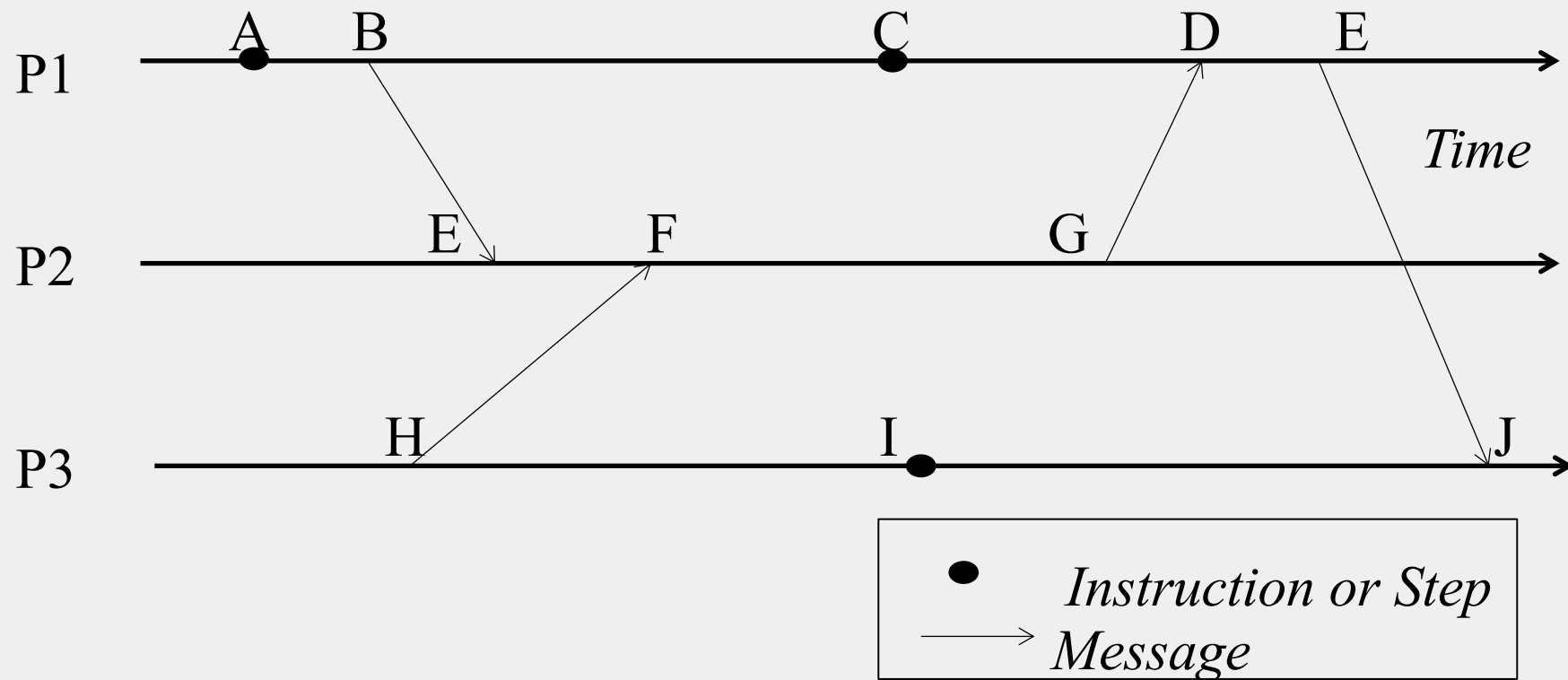
Chandy-Lamport Global Snapshot Algorithm (3)

The algorithm terminates when

- Each process has received (or sent!) a first Marker
 - To record its own state
- Each process has received a Marker on the remaining $(N-2)$ **incoming** channels
 - To record the state of all those channels

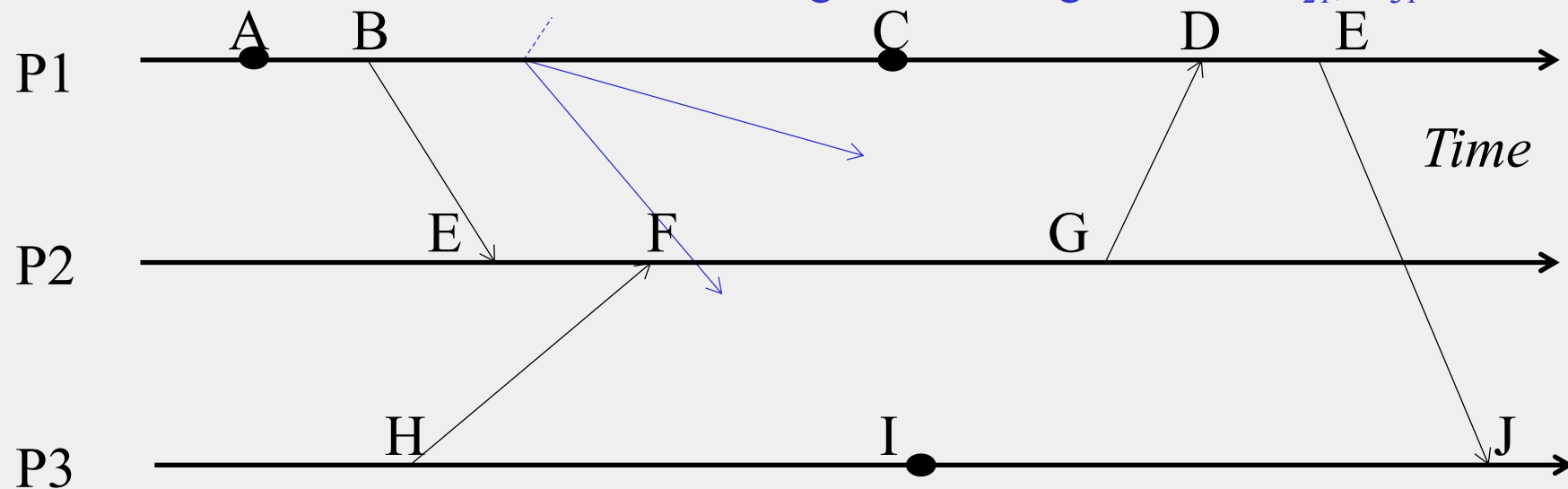
Then, all of this distributed state can be collected to obtain the complete global snapshot

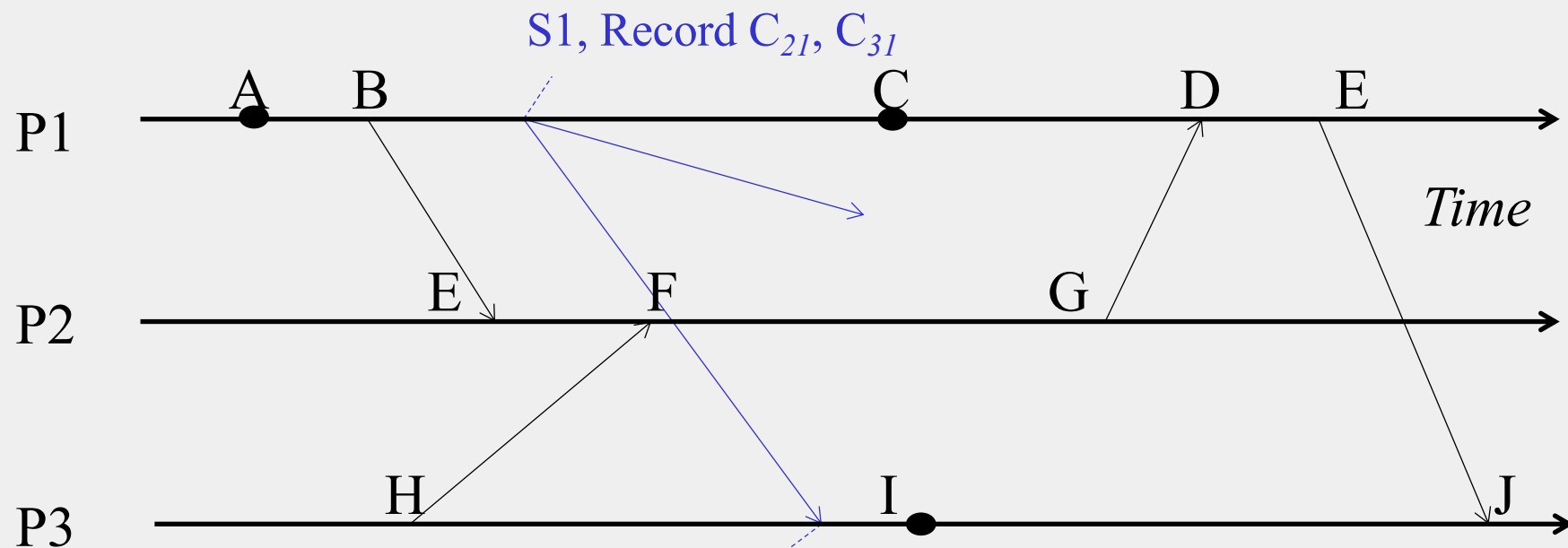
Example

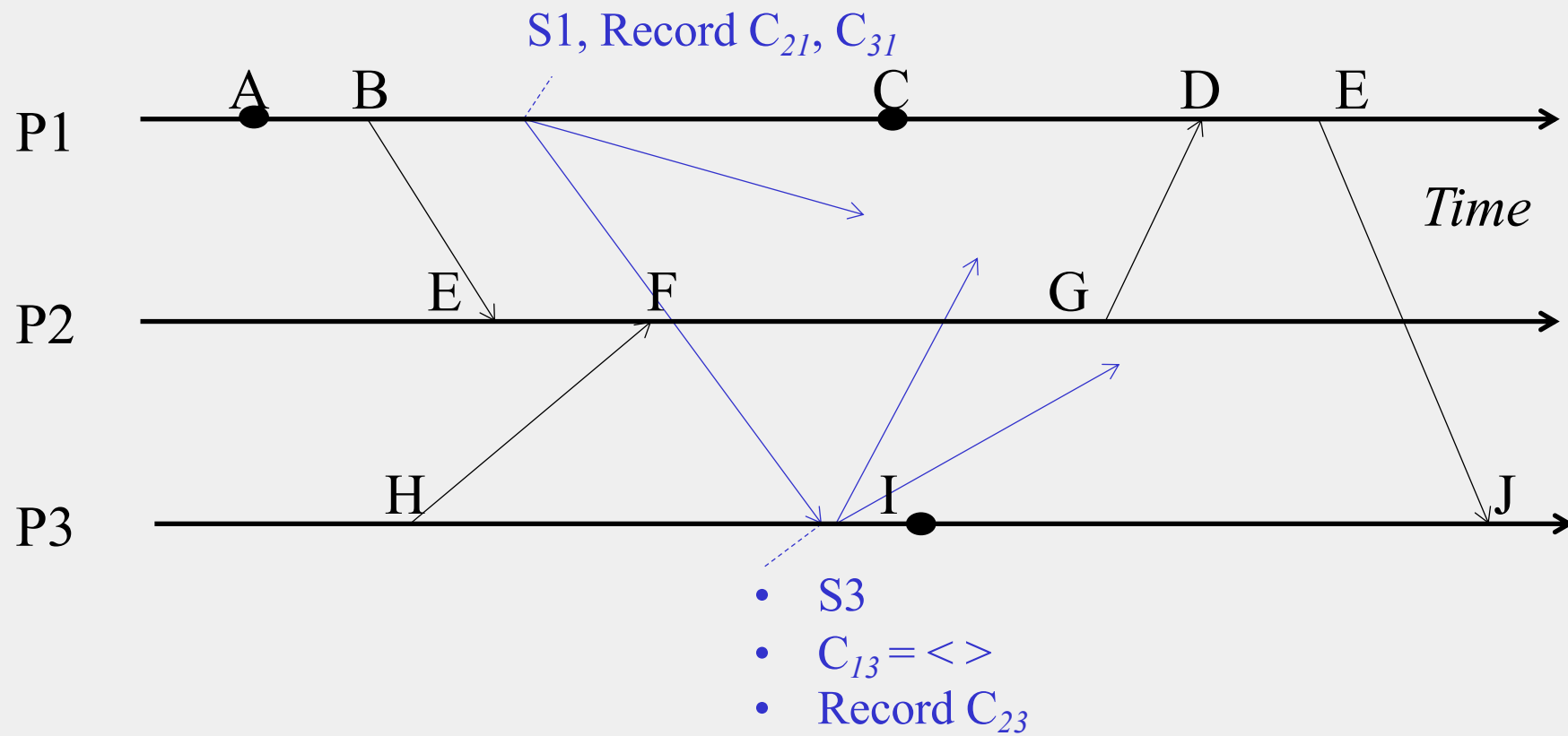


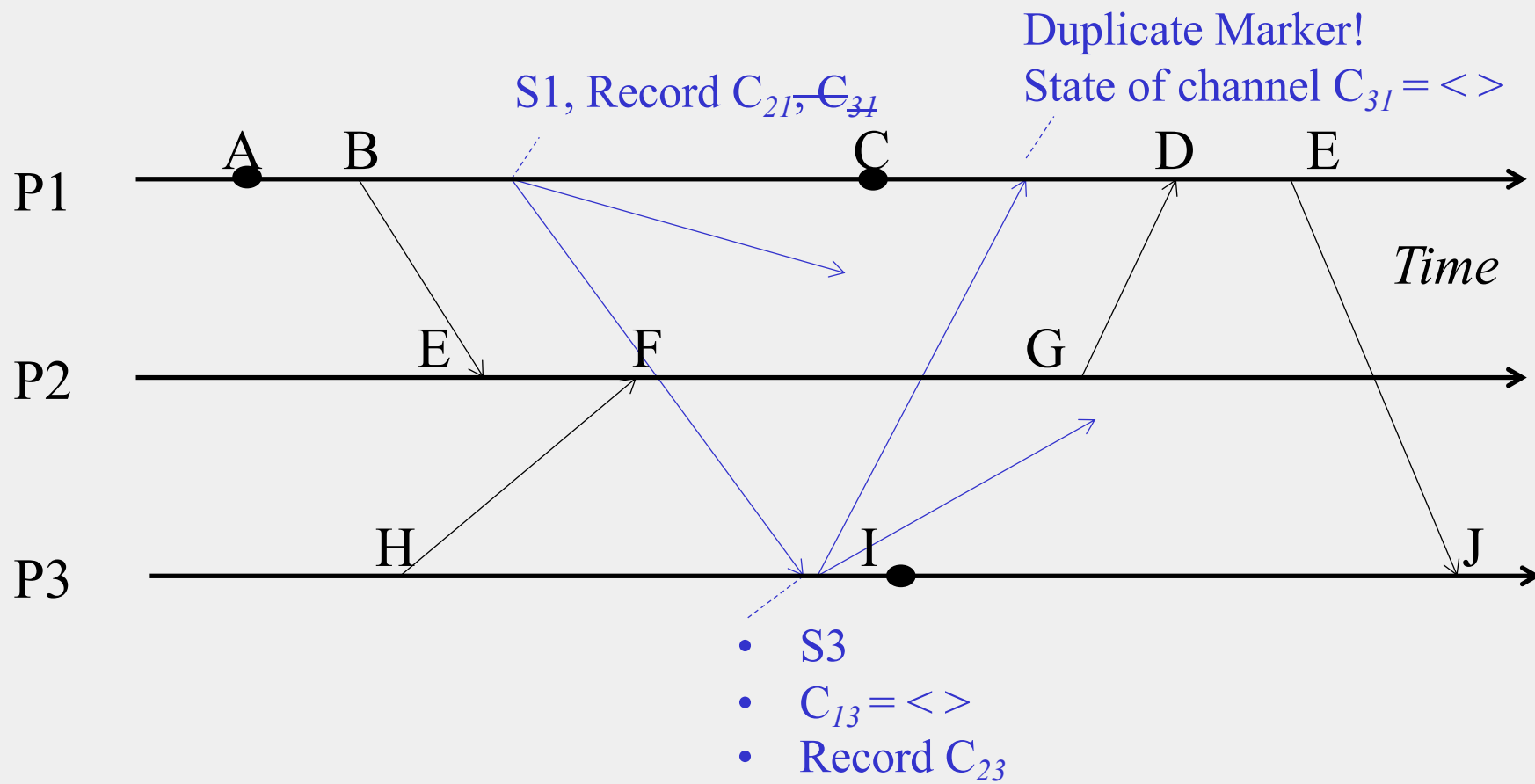
P1 is Initiator:

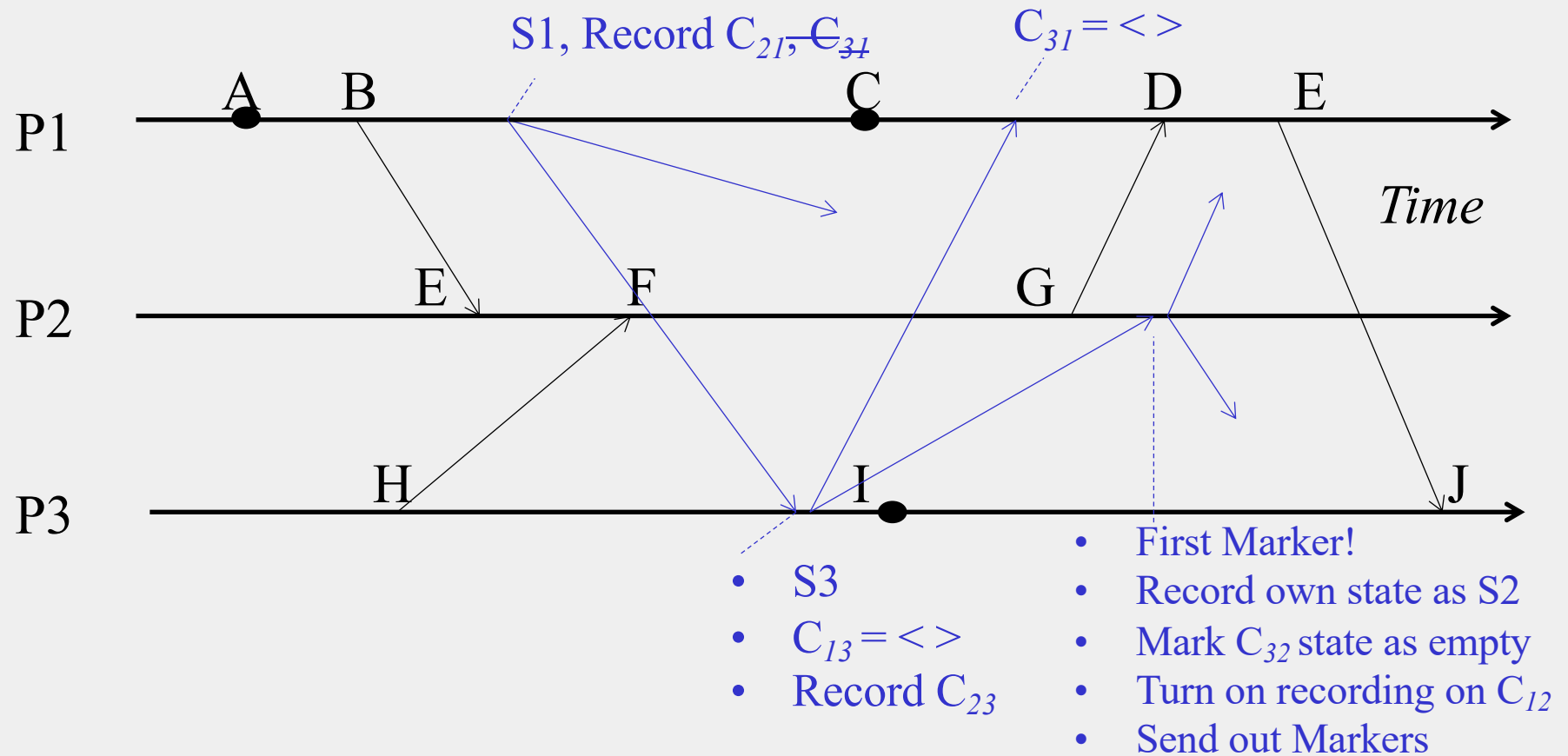
- Record local state S1,
- Send out markers
- Turn on recording on incoming channels C_{21}, C_{31}

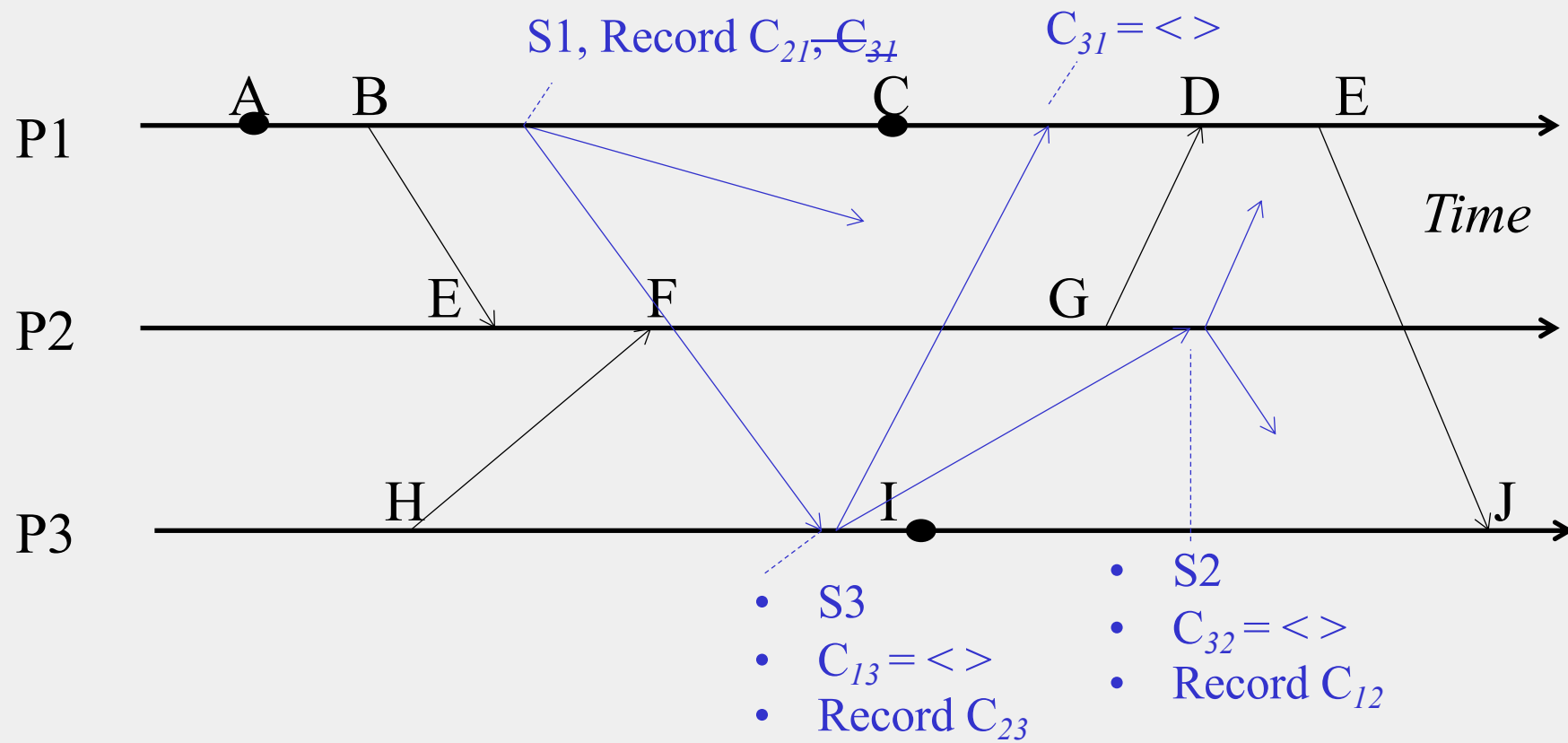


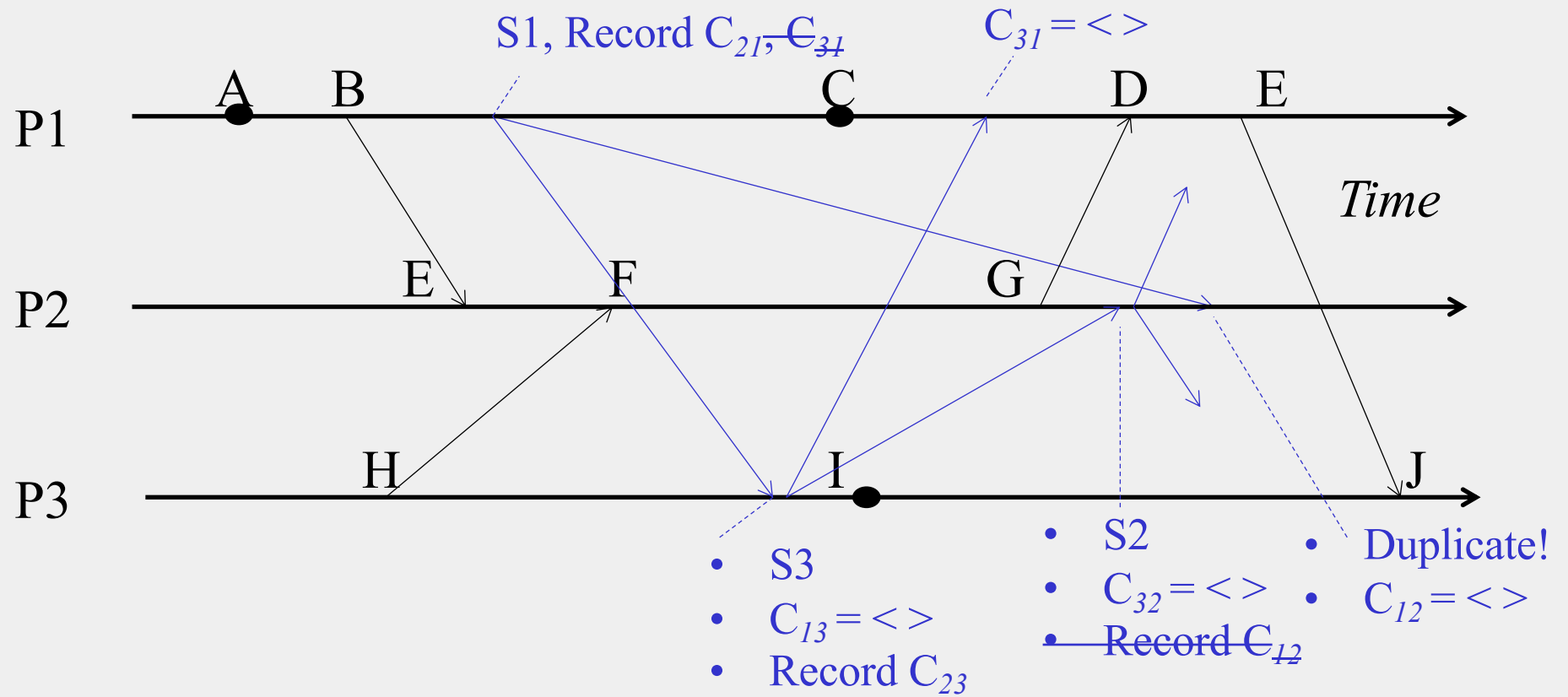


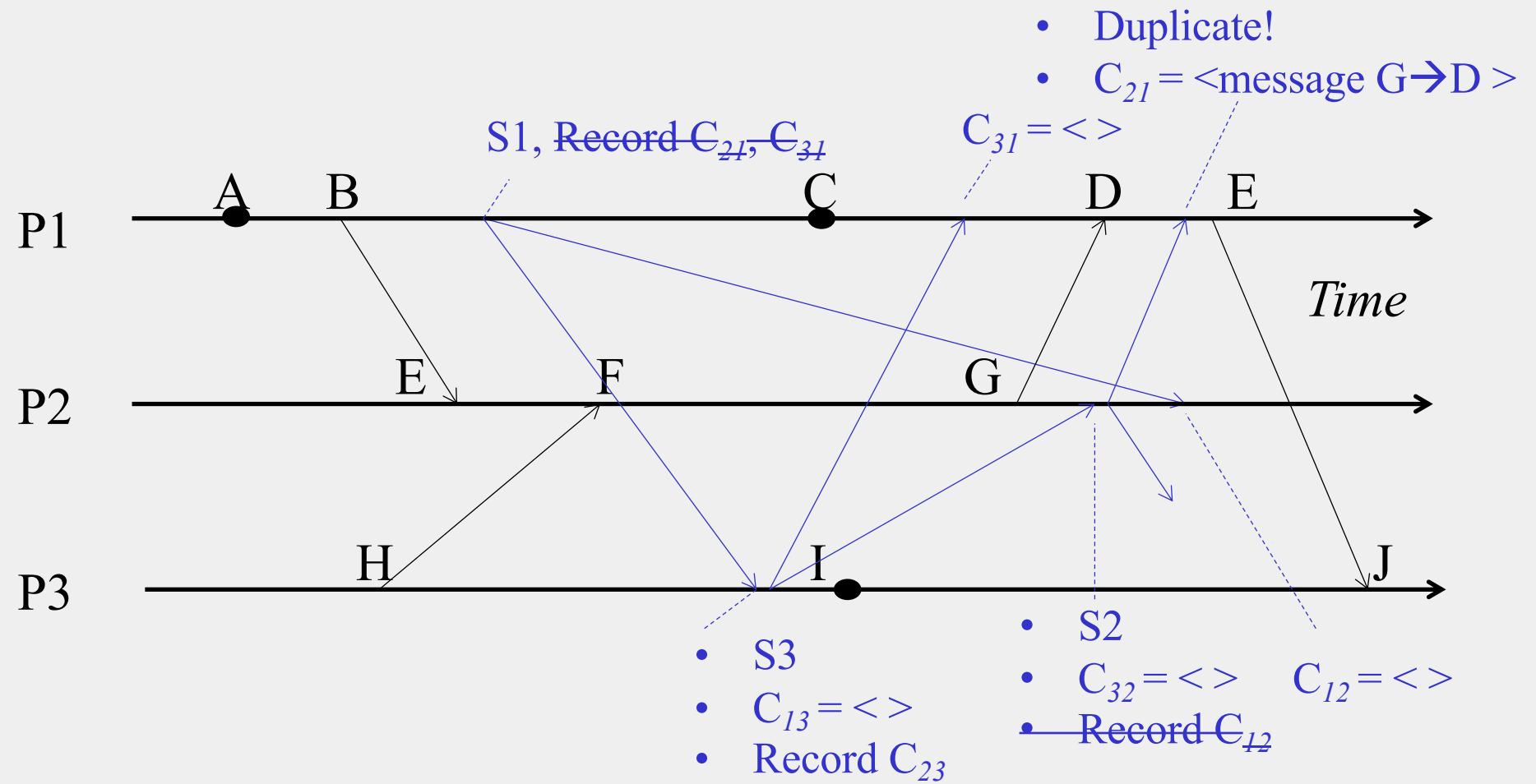


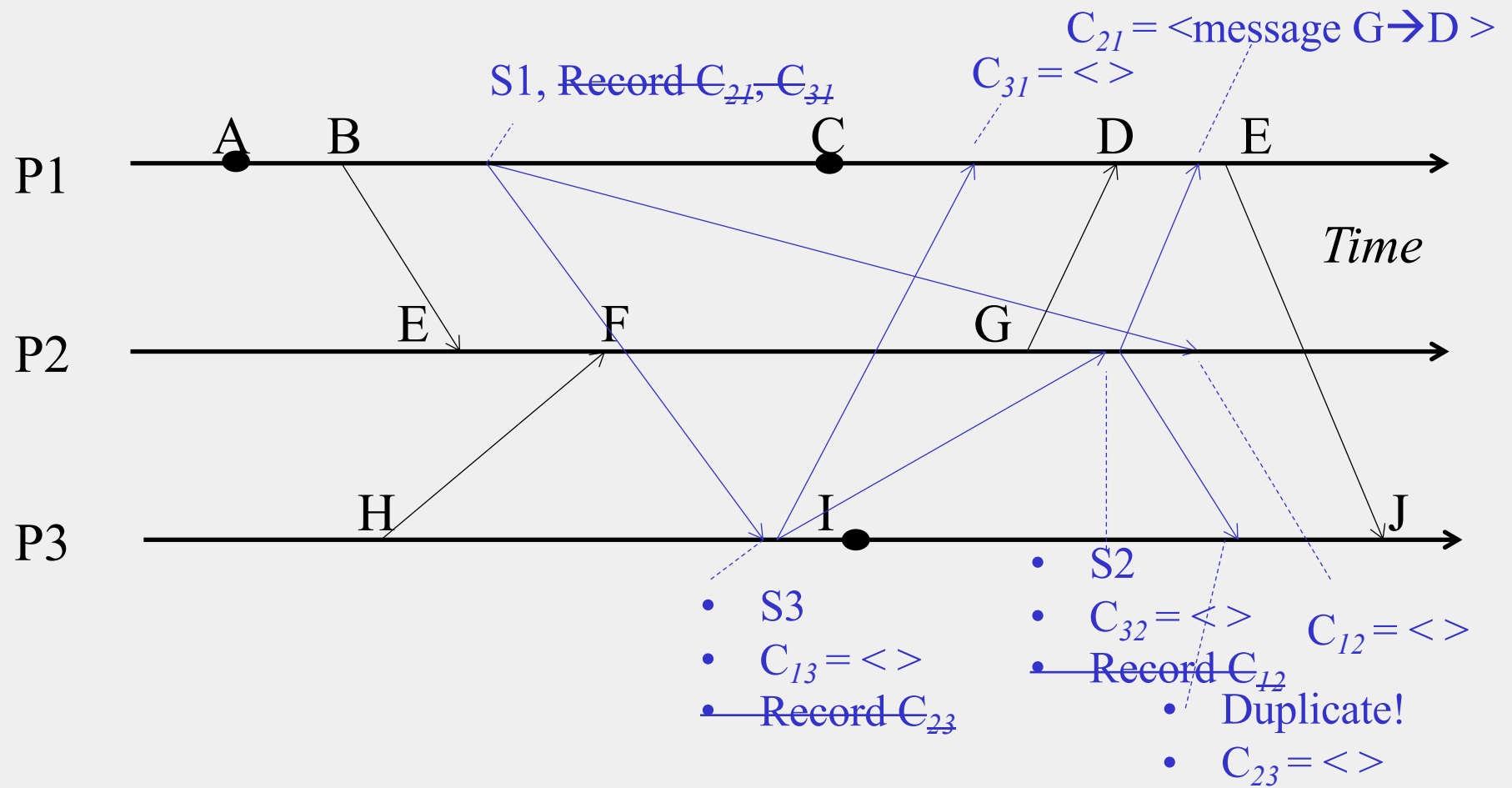




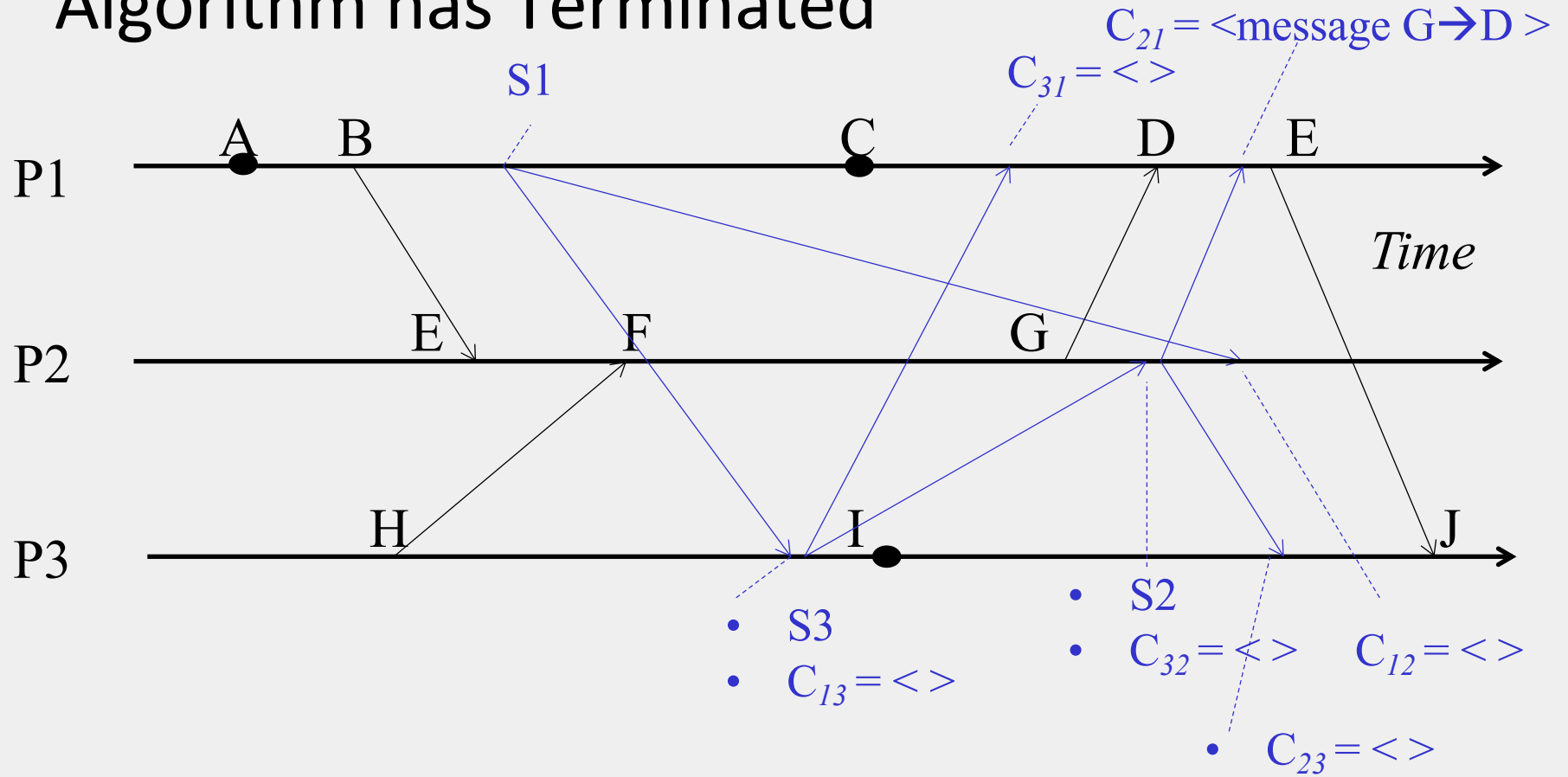




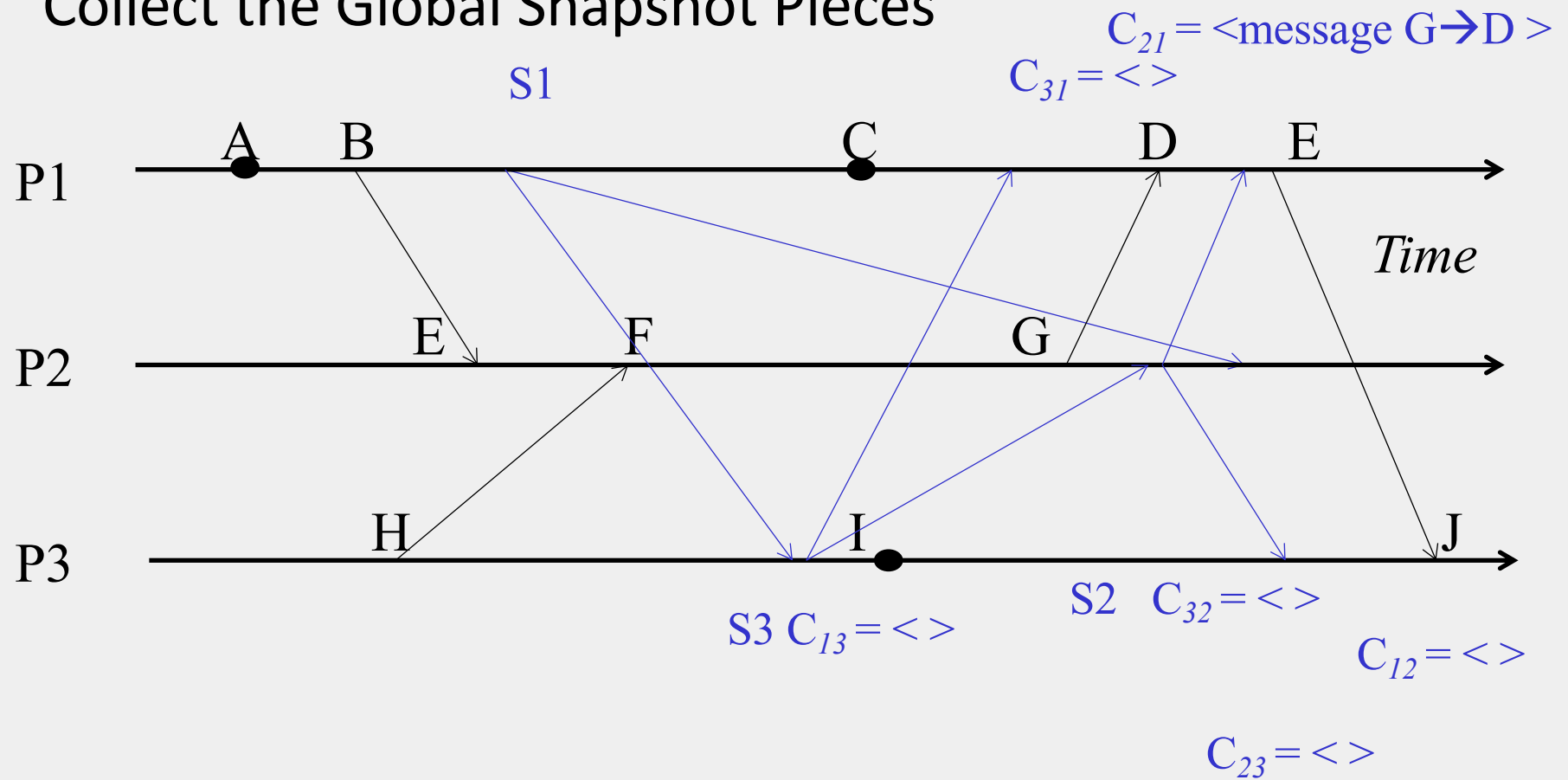




Algorithm has Terminated



Collect the Global Snapshot Pieces



Next

- **Global Snapshot calculated by Chandy-Lamport algorithm is causally correct**
 - How do we know that?



Cuts

- **Cut** = time frontier at each process and at each channel
- Events at the process/channel that happen before the cut are “in the cut”
 - And happening after the cut are “out of the cut”

Consistent Cuts

Consistent Cut: a cut that obeys causality

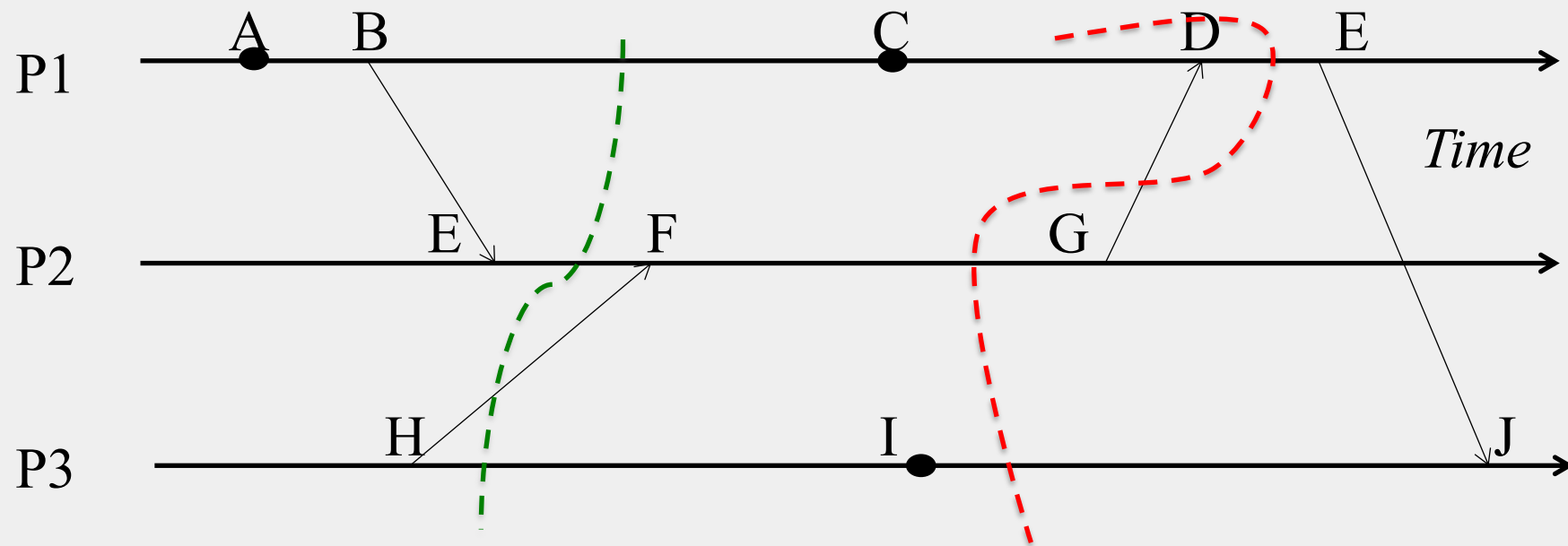
A cut C is a consistent cut if and only if:

for (each pair of events e, f in the system)

if (f is in cut C) && ($e \rightarrow f$) // e happens-before f

then e is also in cut C

Example

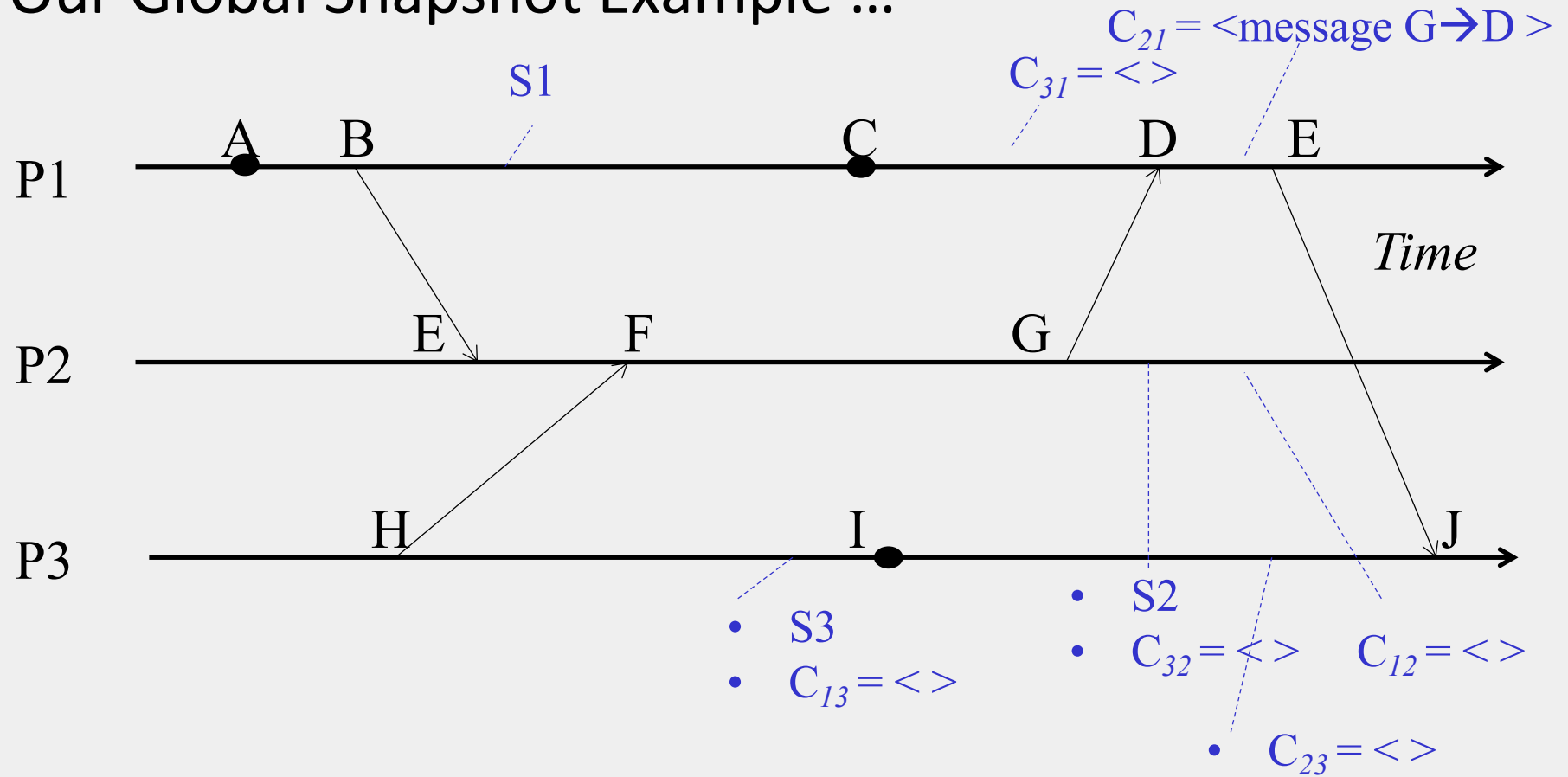


Consistent Cut

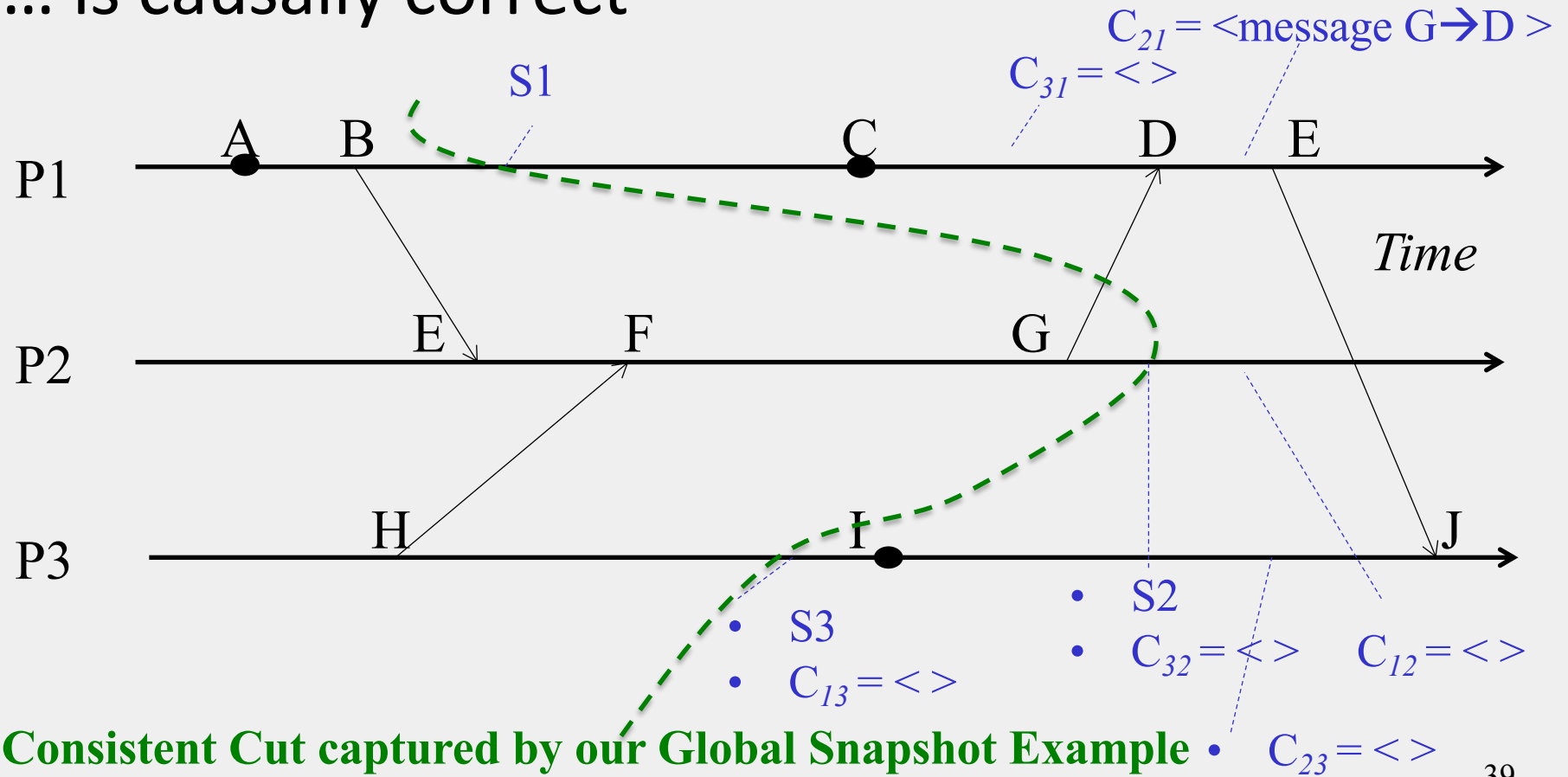
Inconsistent Cut

$G \rightarrow D$, but only D is in cut

Our Global Snapshot Example ...



... is causally correct



In fact...

- Any run of the Chandy-Lamport Global Snapshot algorithm creates a consistent cut

AND

- Every message straddling the consistent cut is captured in the channel states of the snapshot

Chandy-Lamport Global Snapshot algorithm creates a consistent cut

Let's quickly look at the proof

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 $e_j \rightarrow \langle P_j \text{ records its state} \rangle$, then
 - it must be true that $e_i \rightarrow \langle P_i \text{ records its state} \rangle$.

Chandy-Lamport Global Snapshot algorithm creates a consistent cut: Proof

- if $e_j \rightarrow \langle P_j \text{ records its state} \rangle$, then it must be true that $e_i \rightarrow \langle P_i \text{ records its state} \rangle$.
 - By contradiction, suppose $e_j \rightarrow \langle P_j \text{ records its state} \rangle$ and $\langle P_i \text{ records its state} \rangle \rightarrow e_i$
 - Consider the path of app messages (through other processes) that go from $e_i \rightarrow e_j$
 - Due to FIFO ordering, markers on each link in above path will precede regular app messages
 - Thus, since $\langle P_i \text{ records its state} \rangle \rightarrow e_i$, it must be true that P_j received a marker before e_j
 - Thus, e_j is not in the cut $\Rightarrow$ contradiction

“Correctness” in Distributed Systems

- Can be seen in two ways
- Liveness and Safety
- Often confused – it’s important to distinguish from each other

Liveness

- **Liveness** = guarantee that something **good** will happen, **eventually**
 - Eventually $\Rightarrow$ does not imply a time bound, but if you let the system run long enough, then ...

Liveness: Examples

- **Liveness** = guarantee that something **good** will happen, **eventually**
 - Eventually == does not imply a time bound, but if you let the system run long enough, then ...
- **Examples in Real World**
 - Guarantee: at least one of the athletes in the 100m final will win gold
 - Aspirational: a criminal will eventually be jailed
- **Examples in a Distributed System**
 - Distributed computation: will eventually terminate
 - “Completeness” in failure detection: each failure is eventually detected by some non-faulty process
 - In Consensus: All processes eventually decide on a value

Safety

- Safety = guarantee that something **bad** will **never** happen

Safety: Examples

- **Safety** = guarantee that something **bad** will **never** happen
- **Examples in Real World**
 - Aspirational: an innocent person will never be jailed
- **Examples in a Distributed System**
 - No deadlock in a distributed transaction system
 - No object is orphaned in a distributed object system
 - “Accuracy” in failure detection
 - In Consensus: no 2 processes decide on different values

Can We Guarantee Both?

- **Can be difficult to satisfy both liveness and safety in an asynchronous distributed system!**
 - Failure Detection: Completeness (Liveness) and Accuracy (Safety) cannot both be guaranteed in an asynchronous distributed system
 - Consensus: Decisions (Liveness) and correct decisions (Safety) cannot both be guaranteed by any consensus protocol in an asynchronous distributed system
 - Very difficult for legal systems (anywhere in the world) to guarantee that all criminals are jailed (Liveness) and that no innocents are jailed (Safety)

In the language of Global States

- **Recall that a distributed system moves from one global state to another global state, via causal steps**
- **Liveness w.r.t. a property Pr in a given state S means**
 - S satisfies Pr , or there is **some** causal path of global states from S to S' where S' satisfies Pr
- **Safety w.r.t. a property Pr in a given state S means**
 - S satisfies Pr , and **all** global states S' reachable from S also satisfy Pr

Using Global Snapshot Algorithm

- **Chandy-Lamport algorithm can be used to detect global properties that are *stable***
 - Stable = once true, stays true forever afterwards
- **Stable Liveness examples**
 - Computation has terminated
- **Stable Non-Safety examples**
 - There is a deadlock
 - An object is orphaned (no pointers point to it)
- **All stable global properties can be detected using the Chandy-Lamport algorithm**
 - Due to its causal correctness

Summary

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