Time and Global States

CMPS 4760/6760: Distributed Systems

Acknowledgement: slides adapted from Indranil Gupta's lecture notes:
https://courses.engr.illinois.edu/cs425/fa2019/index.html

Overview

- Physical clocks
- States and events
- Logical clocks and vector clocks
- Global states and snapshot algorithm

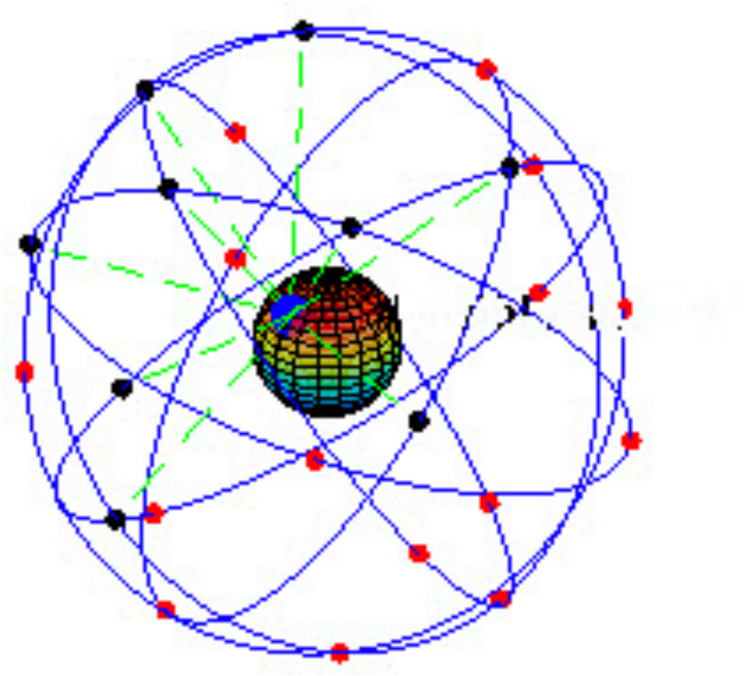
Time and Clock

- Primary standard of time: rotation of earth
  - 1 solar second = 1/86,400th of a solar day that the Earth takes to complete one revolution around its axis
- De facto primary standard of time: atomic clocks
  - 1 atomic second = **9,192,631,770** orbital transitions of **Cesium-133** atom.
  - 86400 atomic sec = 1 solar day – approx. 3 ms (*leap second* correction each year)
- Coordinated Universal Time **(UTC)** does the adjustment for leap seconds ± number of hours in your time zone

Global Positioning System: GPS

A system of 30+ satellites broadcasting accurate spatial coordinates and atomic times

- Location and precise time computed by triangulation
- no leap sec. correction => 18 seconds ahead of UTC (as of 2017)
- Per the theory of relativity, an additional correction is needed. Locally compensated by the receivers



Terminology

Drift rate $\rho = \left|\frac{d(C(t)-t)}{dt}\right|$

Clock skew δ

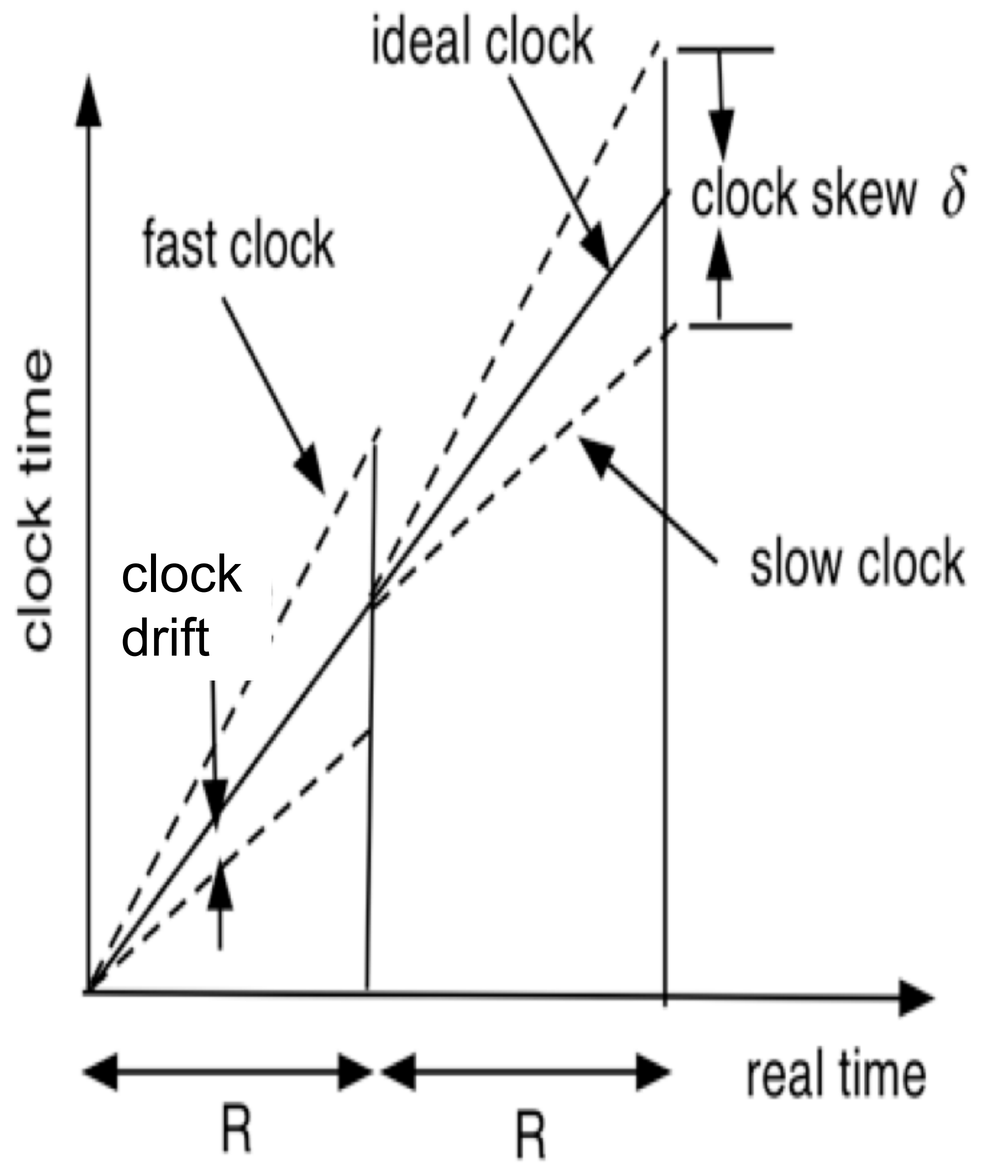
Resynchronization interval R

Max drift rate ρ_{max} implies:

$(1-\rho_{max}) \leq \frac{dC(t)}{dt} \leq (1+\rho_{max})$

Drift is unavoidable:

- Ordinary quartz-oscillators clocks: 10^{-6}
- “High precision” quartz clocks: 10^{-8}
- Atomic clocks: 10^{-13}



Physical clock synchronization

- Why accurate physical time is important?
  - Accurate time keeping: air-traffic control systems
  - Accurate timestamps: multi-version objects
  - Some security mechanisms depend on the physical times of events, e.g., Kerberos

Physical clock synchronization

- External Synchronization: $|C_i(t) - S(t)| < \delta/2$
  - S: a source of UTC time
- Internal Synchronization: $\left|C_i(t) - C_j(t)\right| < \delta$
- Challenges: account for propagation delay, processing delay, and faulty clocks

Physical clock synchronization

- Bounded drift rate
- Monotonicity: $t' > t \Rightarrow C(t') > C(t)$
  - Y2K bug
- Hardware clock vs. Software clock
  - $C_i(t) = \alpha H_i(t) + \beta$

External Synchronization: Cristian's method

- Developed by Cristian in 1989
- Client sends a request to a time server at T_1
- Server receives the request at T_s and sends it back
- Client receives the response at T_2, estimates the round trip time RTT = $T_2 - T_1$, and sets $C_i = T_s + RTT/2$
  - Q: Assume that the minimum transmission min is known, what is the accuracy of client's clock right after synchronization ?
  - A: $\pm(RTT/2 - min)$

External Synchronization: Cristian's method

- $\delta/2$ – desired accuracy bound
- ρ – clock drift rate
- Client pulls data from a time server every R unit of time, where $R < \delta/2\rho$ (why?)
  - *RTT* should be sufficiently short compared with the required accuracy
- Improve accuracy and fault tolerance
  - Query multiple times & take minimum *RTT*
  - Query multiple time servers

Internal Synchronization: The Berkeley Algorithm

- Initially designed by Gusella and Zatti in 1989 for internal synchronization of a collection of computers running Berkeley UNIX

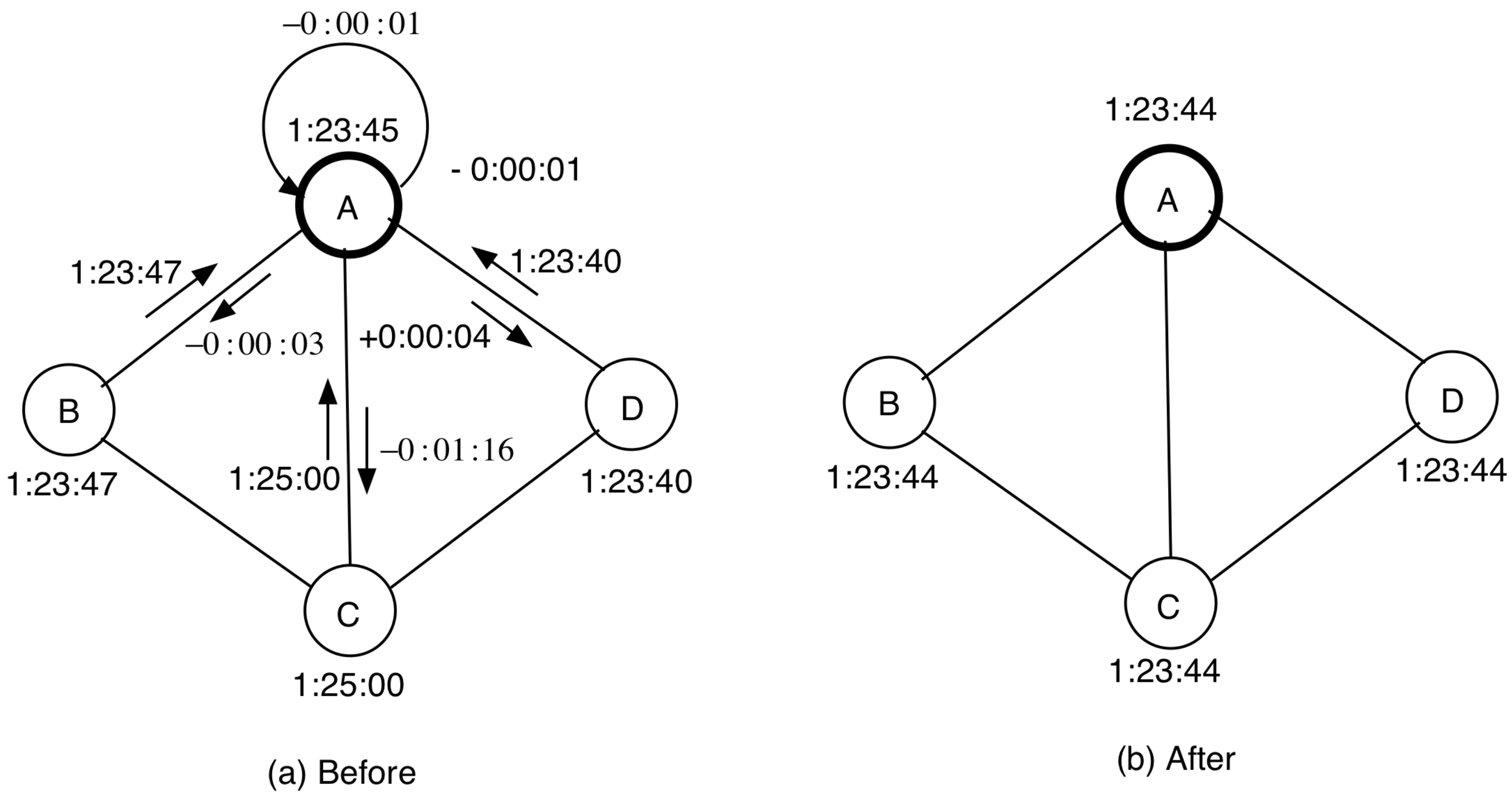
- The participants elect a master (leader)
- The master coordinates the synchronization

Step 1. Salves send their clock values to the master

Step 2. Master discards outliers and computes the average

Step 3. Master sends the needed adjustment to the slaves

The Berkeley Algorithm



To maintain Monotonicity

- Negative correction => slowdown
- Positive correction => speedup

RTT adjustment as in Cristian's method (not shown in the figure)

Internal synchronization with Byzantine clocks

- Lamport and Melliar-Smith's algorithm

Assume N clocks, at most f are faulty

Clock i runs the following algorithm:

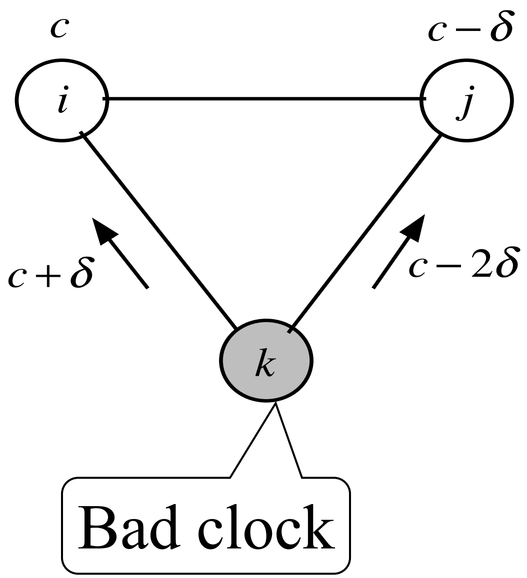
Step 1. Read every clock in the system

- $c_i[j]$: clock i's reading of clock j's value

Step 2. if $|c_i[j] - c_i[i]| > \delta, c_i[j] = c_i[i]$

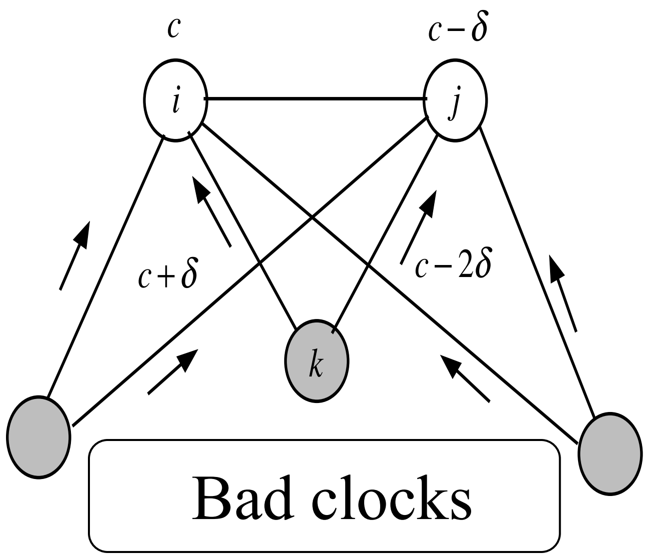
Step 3. Update the clock using the average of these values

Synchronization is maintained if $N > 3f$



A faulty clock exhibits 2-faced or byzantine behavior

Lamport and Melliar-Smith's algorithm



The maximum difference between the averages computed by two non-faulty nodes is $(3f\delta/N)$

To keep the clocks synchronized,

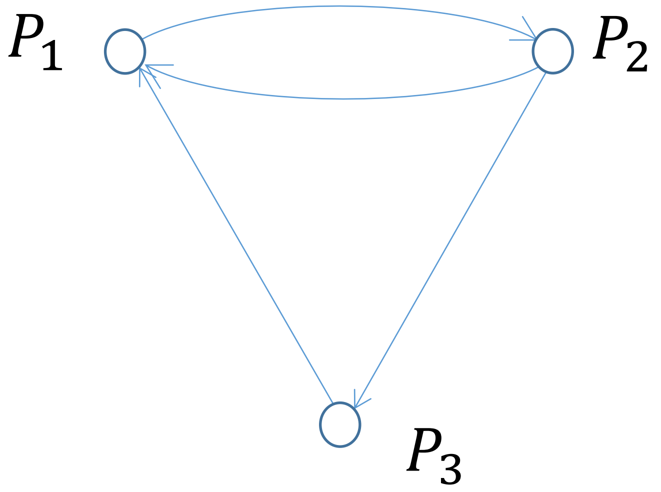
$$\frac{3f\delta}{N} < \delta \Leftrightarrow N > 3f$$

Overview

- Physical clocks
- States and events
- Logical clocks and vector clocks
- Global states and snapshot algorithm

System Model

- A distributed program consists of a set of N processes, $\{P_1, P_2, \ldots P_N\}$, and a set of unidirectional channels.
  - Message passing only, no shared memory, no global clock
  - Channel model: error free, arbitrary but finite delay, no assumptions on ordering



States and Actions

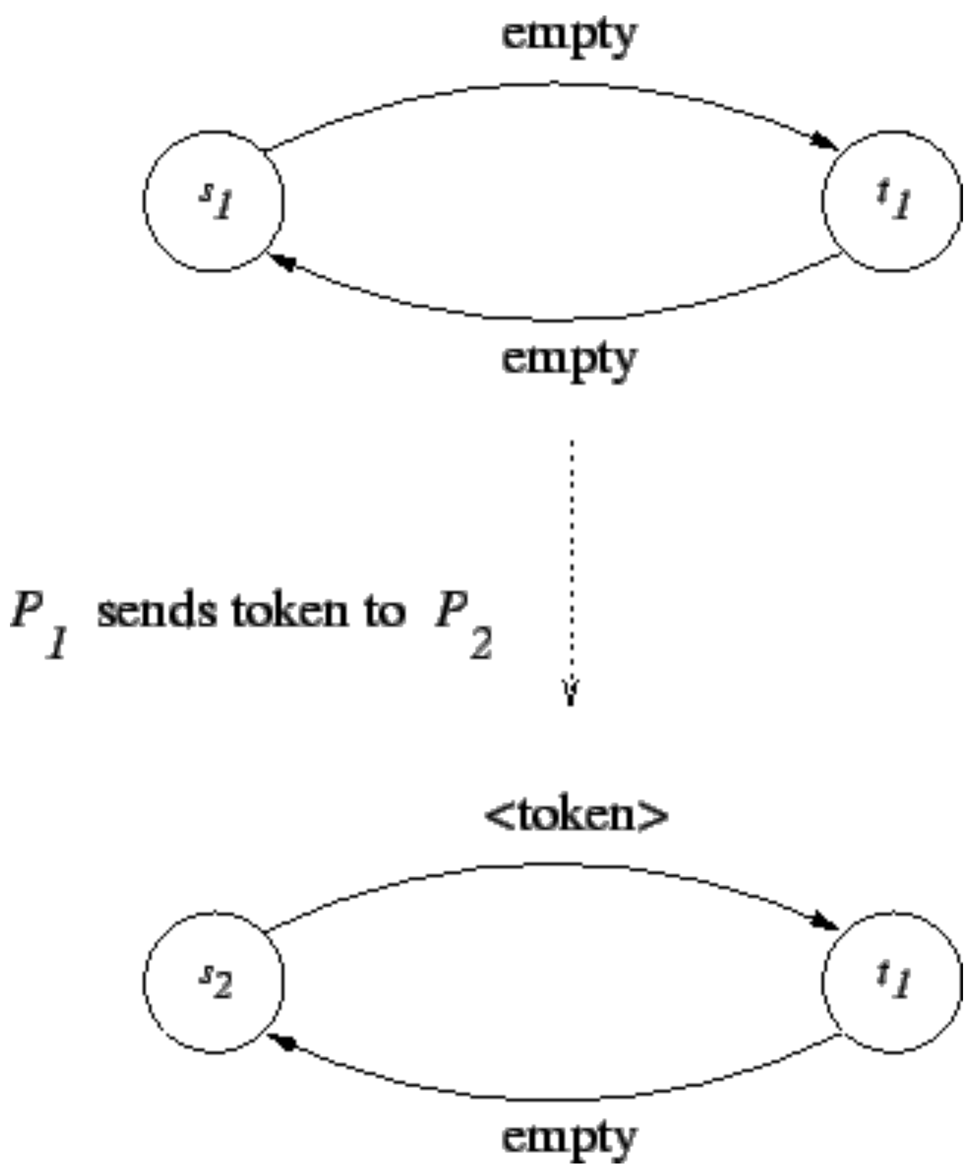
- Each process P_i has a state s_i that, in general, it transforms as it executes
  - a state includes the values of all the variables within it (including the program counter)
  - may also include the values of any objects in its local operating system environment that it affects, such as files
- As each process P_i executes it takes a series of actions, each of which is either
  - message *send* or *receive* operation
  - or an operation that transforms P_i's state – one that changes one or more of the values in s_i
- State of a channel: sequence of messages set along the channel but not received
  - A process may record messages sent and received as part of its local state

States and Events

- An event $e \in E$ corresponds to an action and may change the state of a process and the state of at most one channel incident on that process
  - Internal events only change the state of a process
  - External events: sends to/receives from other process
- A set of events $(e_i^0, e_i^1, e_i^2, \dots)$ in a single process is called sequential, and their occurrences can be totally ordered in time using the clock at that process
- A run or a computation of a process P_i is defined as a sequence of local states and events: $s_i^0 e_i^0 s_i^1 \dots e_i^k s_i^k$

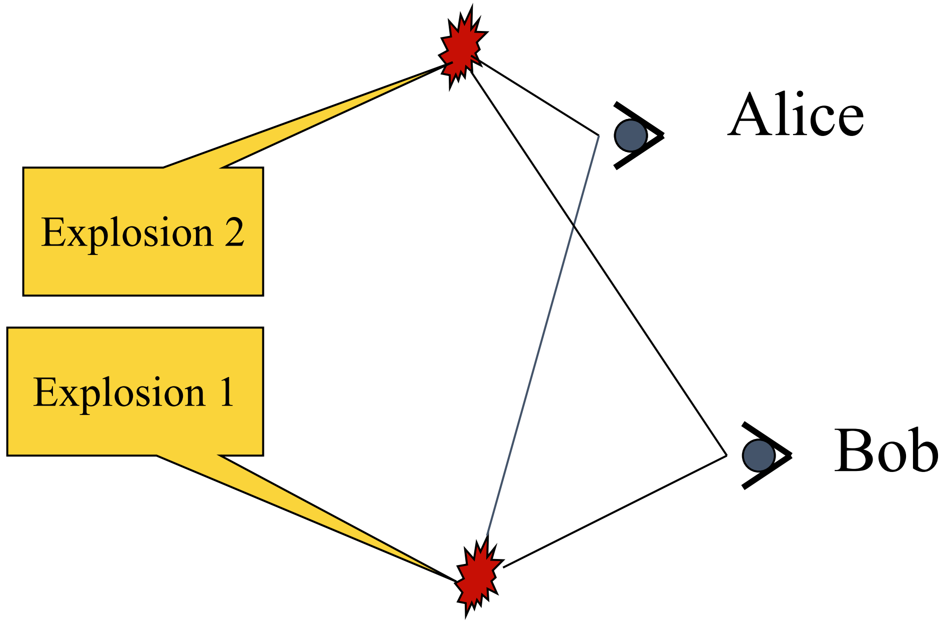
Global States

- The set of global states = the cross product of local states and the states of channels.
- An initial global state is one in which all local states are initial, and all the channels are empty



Global States

- "time-based model": a global state is a set of local states that occur simultaneously
- "happened-before model": a global state is a set of local states that are all concurrent with each other



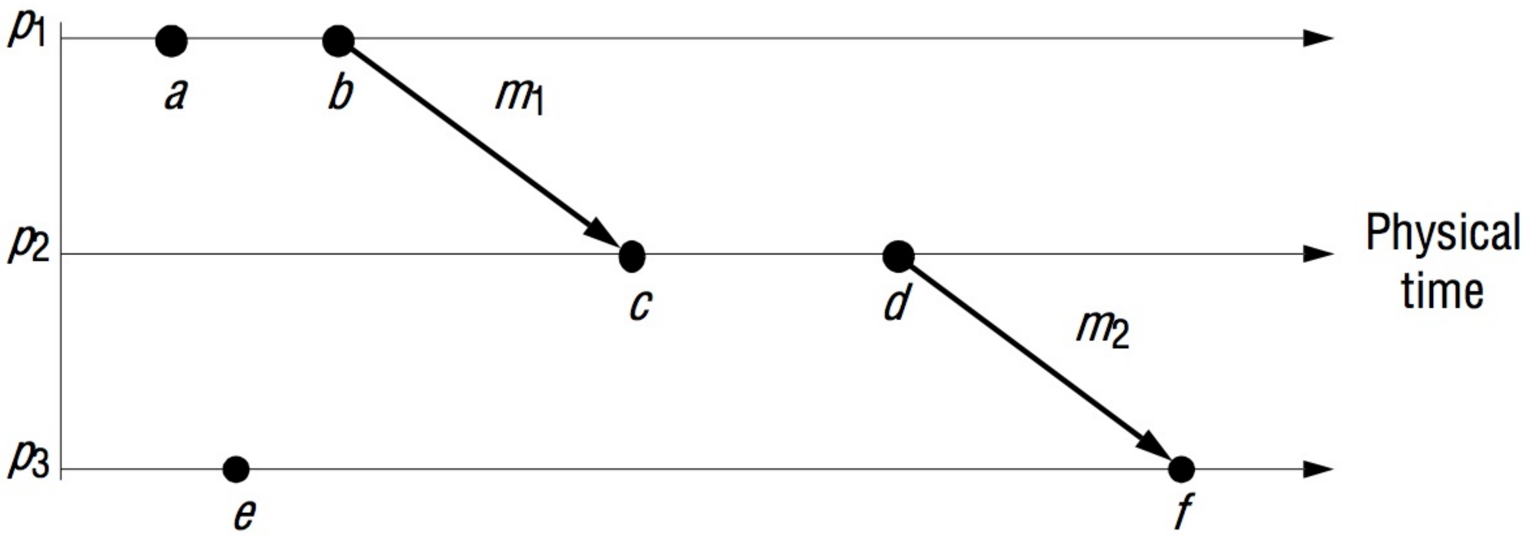
There is nothing called simultaneous in the physical world.

Global States

- Questions:
  - How do we decide if an event happened before another event when the two events are on different processes?
  - How do we decide if two events are concurrent?
  - Can we do these without using physical clocks, since physical clocks are not be perfectly synchronized?

Causality

- The event of sending message must have happened before the event of receiving that message



Happened Before Model

Notations:

$e \prec f$ iff e, f are two events in a single process P, and e proceeds f

$e \preccurlyeq f$ iff $e \prec f \vee e = f$

$e \rightsquigarrow f$ iff e = sending a message, and f = receipt of that message

These definitions also apply to states

Happened Before Model

The ***happened-before*** relation ($\rightarrow$) is the smallest relation that satisfies

Rule 1. if $(e \prec f) \vee (e \rightsquigarrow f)$ then $e \rightarrow f$.

Rule 2. $(e \rightarrow f) \wedge (f \rightarrow g) \Rightarrow e \rightarrow g$

$\rightarrow$ defines a partial order on E (the set of all events) – why?

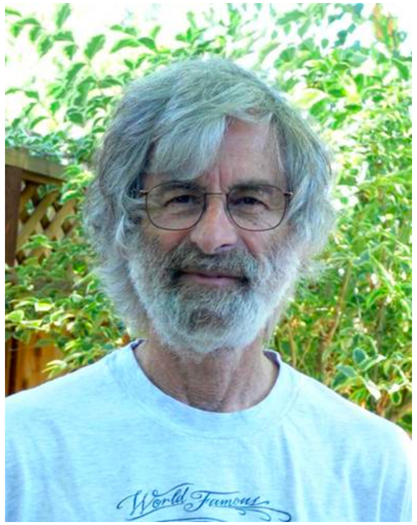
e and f are concurrent (denoted by $e||f$) if $\neg(e \rightarrow f) \wedge \neg(f \rightarrow e)$

Again, we can similarly define happened-before relation for states

Overview

- Physical clocks
- States and events
- Logical clocks and vector clocks
- Global states and snapshot algorithm

Logical Clocks



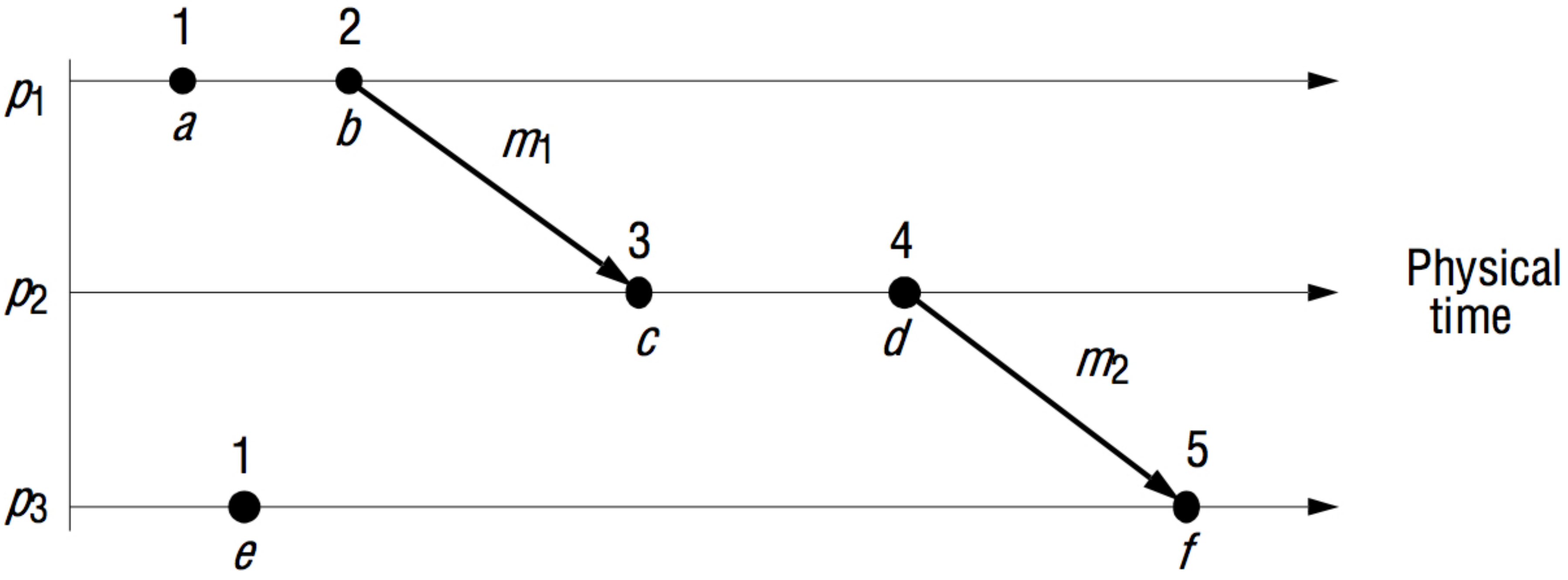
Leslie B. Lamport

- A logical clock LC is a map from E to $\mathbb{N}$ with the constraint:

$$\forall\, e, f \in E, e \rightarrow f \Rightarrow LC(e) < LC(f)$$

- The constraint models:
  - sequential nature of execution at each process
  - Physical requirement that any message transmission requires a nonzero amount of time

Lamport timestamps



Implementation

P_i::

var

LC: integer initially 0;

internal event ():

$LC = LC + 1;$

send event (m):

$LC = LC + 1;$
piggybacks LC on m;

receive event (m, t):

$LC = \max(LC, t) + 1;$

A total ordering on events

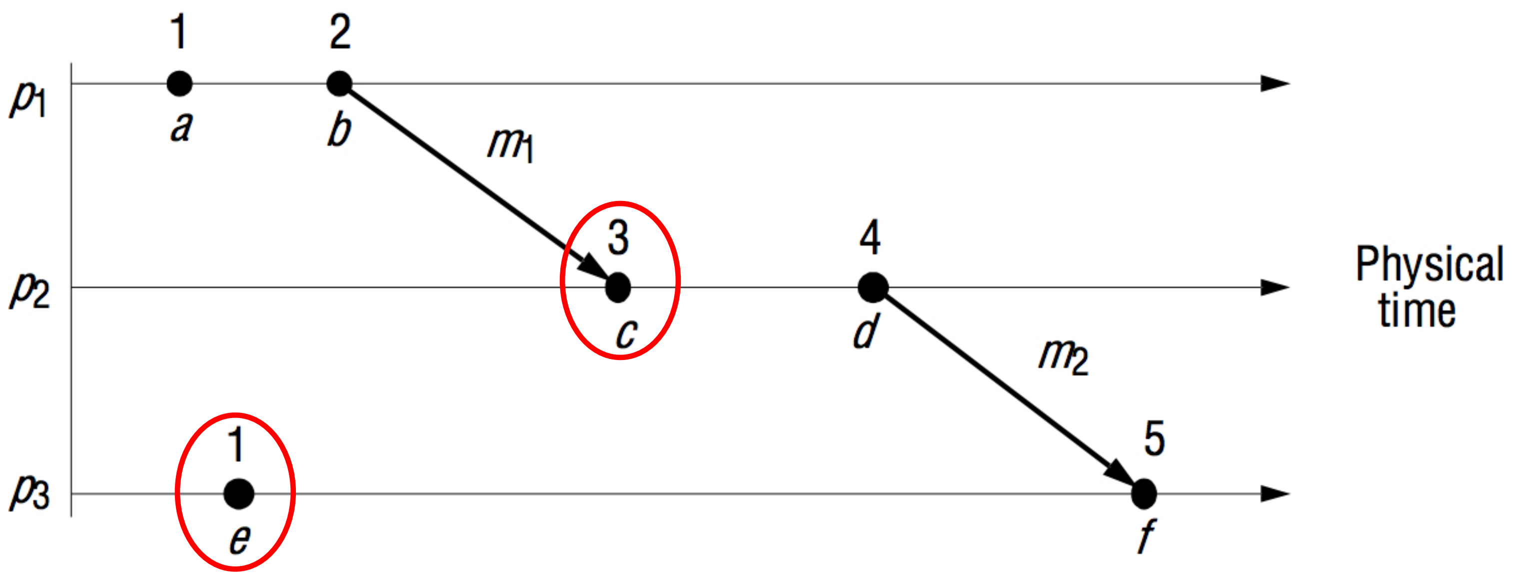
- Let e and f be two events at processes i and j, respectively. We can define a total order $\ll$ of events as:

$$e \ll f \text{ iff either } LC(e) < LC(f)$$

$$\text{or } LC(e) = LC(f) \text{ and } i < j$$

Logical Clocks and Causality

- Logical clocks cannot detect causality (why?)
  - $\exists\, e, f \in E,\ \big(LC(e) < LC(f)\big) \wedge (e \nrightarrow f)$



Vector Clocks

- A vector clock VC is a map from E to $\mathbb{N}^N$ with the constraint:

$$\forall\ e, f\ \in E, e \rightarrow f\ \Leftrightarrow VC(e) < VC(f)$$

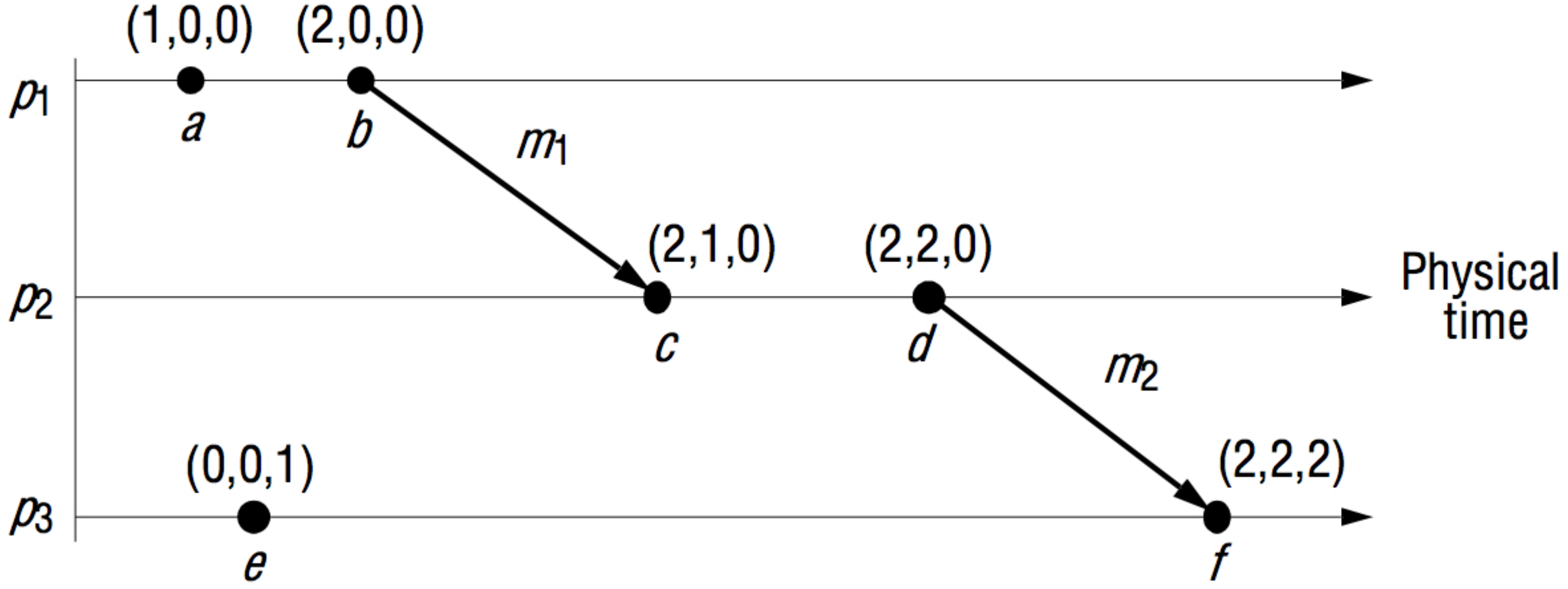
- Given two vectors x and y of dimension N, we compare them as follows:

$$x < y := (\forall k: 0 \leq k \leq N-1: x[k] \leq y[k]) \wedge (\exists j: 0 \leq j \leq N-1: x[j] < y[j])$$

E.g., $(2,1,0) < (2,1,1)$

$$x \leq y := (x < y) \vee (x = y)$$

Vector timestamps



Implementation

P_i::

var

VC: array$[1..N]$ of integer; initially $VC[j] = 0\ \forall j$;

internal event ():

$VC[i] = VC[i] + 1$;

send event (m):

$VC[i] = VC[i] + 1$;

piggybacks VC on m;

receive event (m, t):

$VC[i] = VC[i] + 1$;

$\forall j$: $VC[j] = \max(VC[j], t[j])$;

Properties

$\forall\, e, f \in E, e \rightarrow f \Leftrightarrow VC(e) < VC(f)$

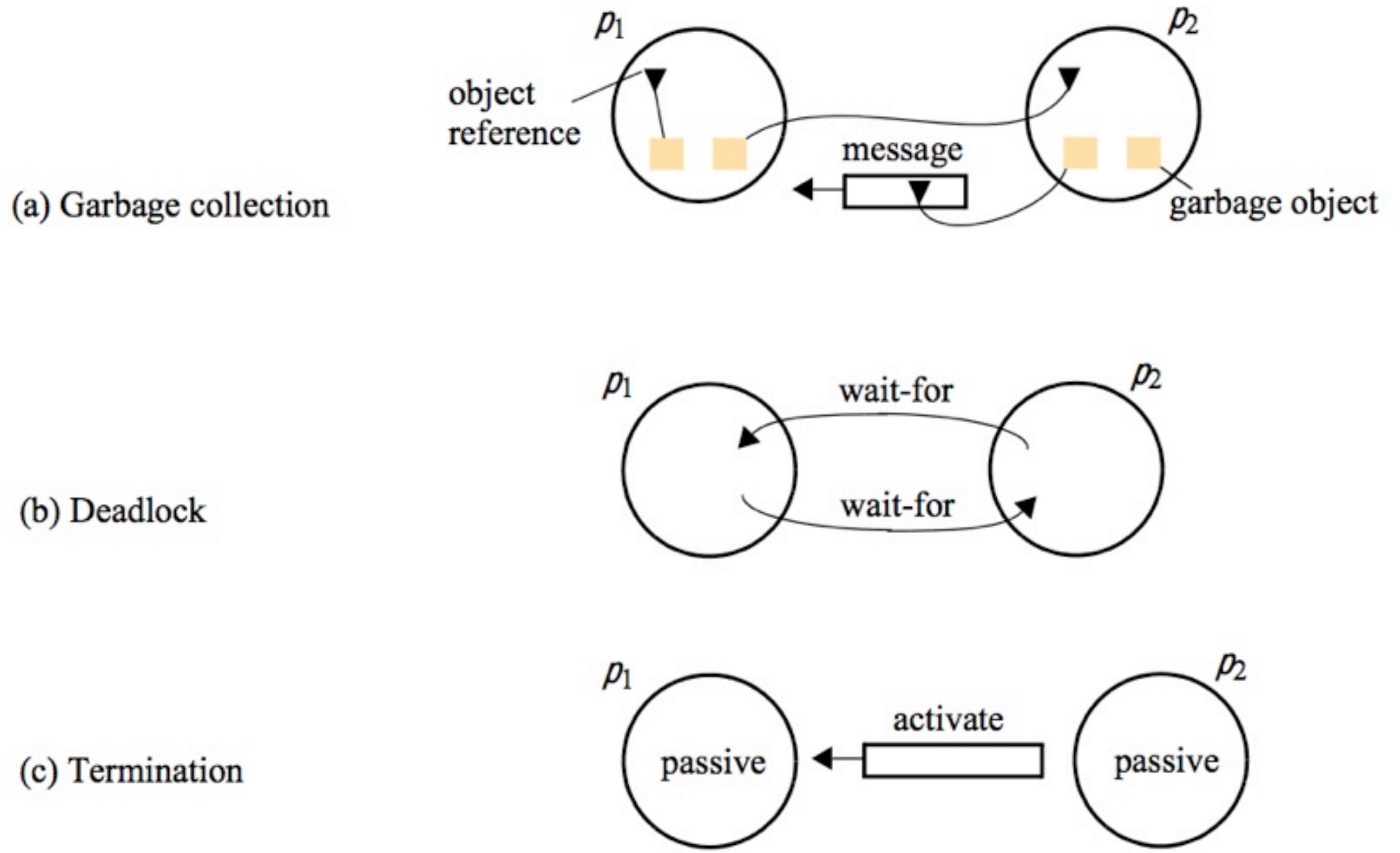
Theorem: For any two events e and f occurred at two processes i and j respectively, we have

$e \rightarrow f \Leftrightarrow (\forall k \neq j\text{: } VC(e)[k] \leq VC(f)[k]) \wedge (VC(e)[j] < VC(f)[j])$

Overview

- Physical clocks
- States and events
- Logical clocks and vector clocks
- Global states and snapshot algorithm

Detecting Global Properties



Global State

- Current global state is hard to get
  - No process has global knowledge
- Past global state is often sufficient
  - Failure recovery
  - Monitoring stable properties

Global State Predicates

- A global state predicate that maps from the set of global states of processes in the system to {True, False}
- A property is called stable if once it is true it stays true forever
    - E.g., Is the object garbage? Is the system deadlocked? Or Is the system terminated?

Global Snapshot

Global State

- "time-based model": a global state is a set of local states that occur simultaneously.

- "happened-before model": a global state is a set of local states that are all concurrent with each other.

Local States

- A distributed system $\wp$ of N processes, $\{P_1, P_2, \ldots P_N\}$, connected by unidirectional channels.
- Event ordering on a single process
  - $e \rightarrow_i e'$: event e occurs before e' at P_i
  - $history(P_i) = h_i =< e_i^0, e_i^1, e_i^2, \ldots >$
  - $h_i^k =< e_i^0, e_i^1, e_i^2, \ldots, e_i^k >$
  - s_i^k - the state of process P_i immediately before the kth event occurs, so that s_i^0 is the initial state of P_i
  - processes record the sending or receipt of all messages as part of their state

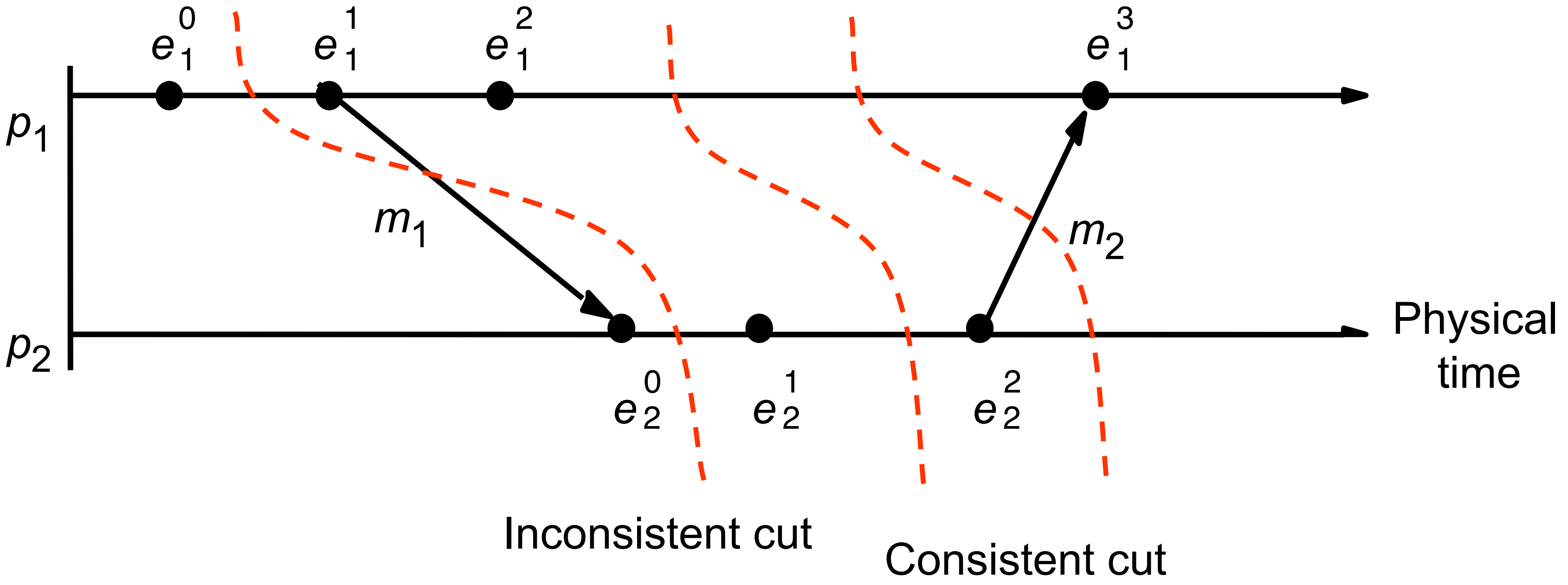
Cuts

- A *cut* of the system's execution is a subset of its global history that is a union of prefixes of process histories:

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \cdots \cup h_N^{c_N}$$

- $e_i^{c_i}$ - the last event processed by P_i in the cut C
- The set of events $\{e_i^{c_i}: i = 1,2, \dots, N\}$ is called the frontier of the cut
- A cut defines a global state $S = (s_1, s_2, \dots, s_N)$ where s_i is the state of P_i immediately after event $e_i^{c_i}$

Cuts



A cut C is consistent, if for each event it contains, it also contains all the events that happened-before that event: $\forall e \in C, f \rightarrow e \Rightarrow f \in C$

A consistent global state (or a snapshot) is one that corresponds to a consistent cut

Runs and Linearizations

- A run is a total ordering of all the events in a global history that is consistent with each local history's ordering
- A linearization (or a consistent run) is a total ordering of all the events in a global history that is consistent with the happened-before relation
- A state S' is reachable from a state S if there is a linearization that passes through S and then S'

Safety and Liveness

- Safety – ``something bad does no occur''
  - A system is safe with respect to an undesirable property α if the α evaluates to False for all states S reachable from the original state S_0
- Liveness – ``something good will eventually occur''
  - For any linearization L starting in the state S_0, a desirable property β evaluates to True for some state S_L reachable from S_0

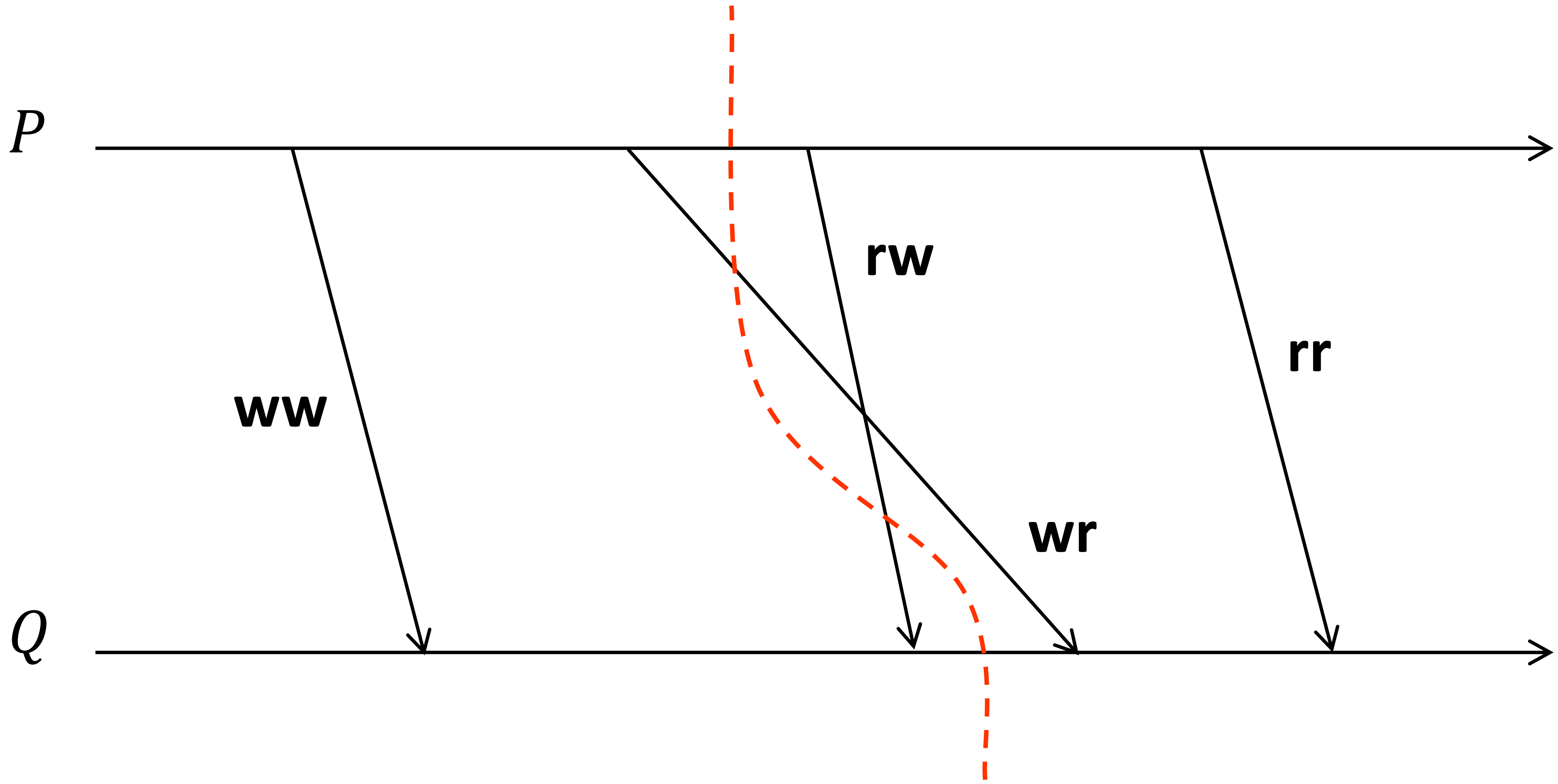
Chandy and Lamport’s snapshot algorithm

- Assumptions
  - Neither channels nor processes fail
  - Channels are unidirectional and FIFO-ordered (First in First out)
  - The graph of processes and channels is strongly connected (there is a path between any two processes)
  - Any process may initiate a global snapshot at any time
  - The processes may continue their execution and send and receive normal messages while the snapshot takes place

Chandy and Lamport's snapshot algorithm

- Informal description
  - Each process is either white or red. All processes are initially white
  - After recording the local state, a process turns red
- Two difficulties
  - Need to ensure that the recorded local states are mutually concurrent
  - Need to capture the state of channels

Classification of Messages



Chandy and Lamport's snapshot algorithm

- Informal description
  - Each process is either white or red. All processes are initially white
  - After recording the local state, a process turns red
  - Once a process turns red, it is required to send a special message called a *marker* along all its outgoing channels before it sends out any normal message, and start recording messages from all incoming channels
  - Once P_i receives a marker from P_j
    - it is required to turn red if it has not already done so
    - it stops recording messages from P_j

Chandy and Lamport's snapshot algorithm

P_i::

var

$color$: $\{white, red\}$ initially $white$;

// assume k incoming channels

$chan$: array$[1..k]$ queues of messages initially $null$;

$closed$: array$[1..k]$ of boolean initially $false$;

$turn\_red()$ **enabled if** $(color == white)$:

save_local_state;

$color = red$;

send $(marker)$ to all neighbors;

Upon $receive(marker)$ on channel j

if$(color == white)$ $turn\_red()$;

$closed[j] = true$;

Upon $receive(prog\_message)$ on channel j

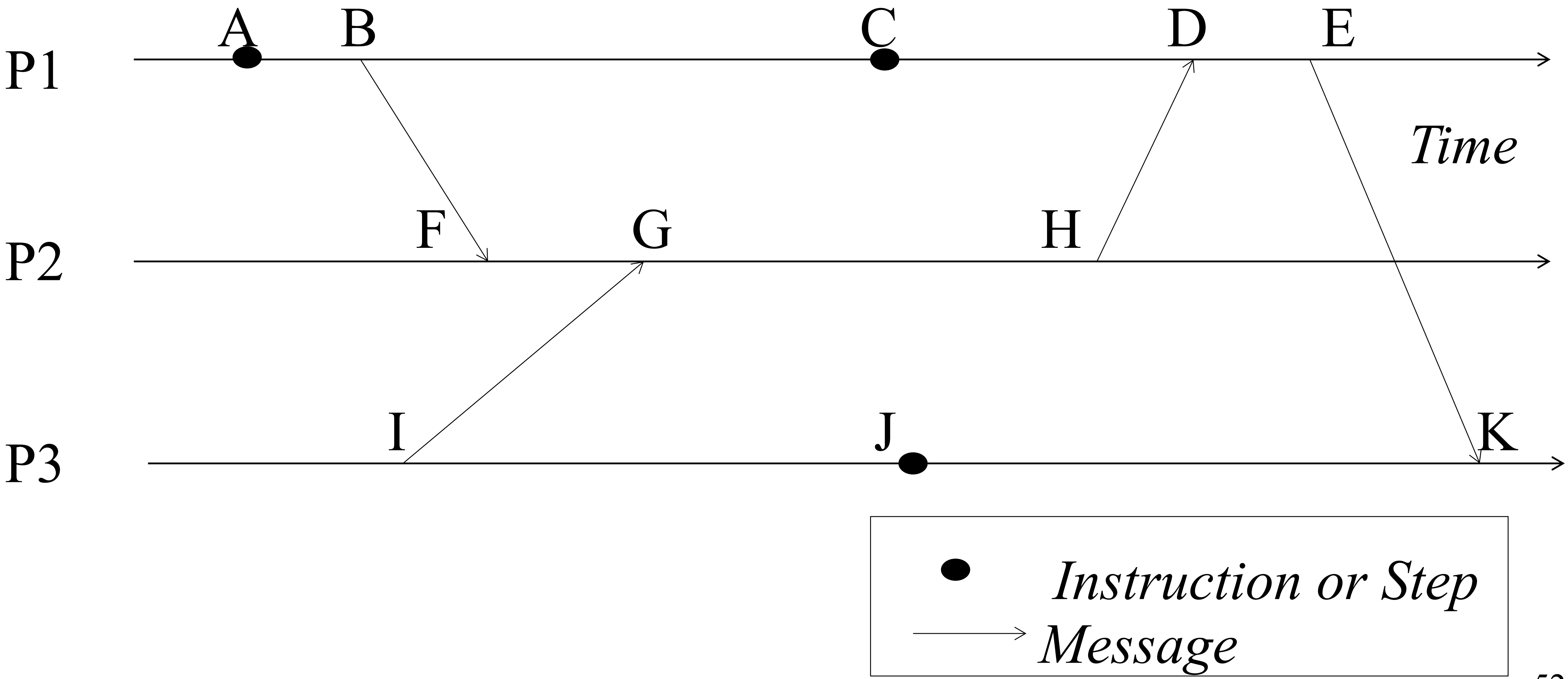
if$(color == red \wedge \neg closed[j])$

$chan[j] = append(chan[j], prog\_message)$

Chandy and Lamport’s snapshot algorithm

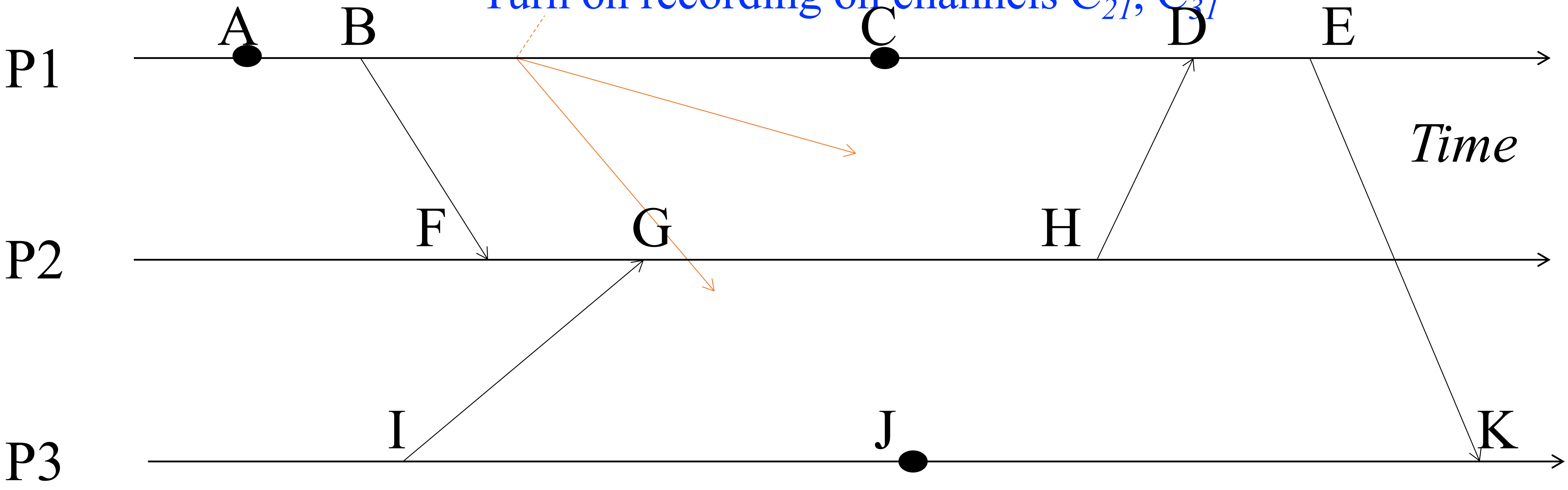
- The algorithm terminates when
  - All processes have received a Marker
    - To record their own state
  - All processes have received a Marker on all the (N-1) incoming channels at each
    - To record the state of all channels
- Then, (if needed), a central server collects all these partial state pieces to obtain the full global snapshot

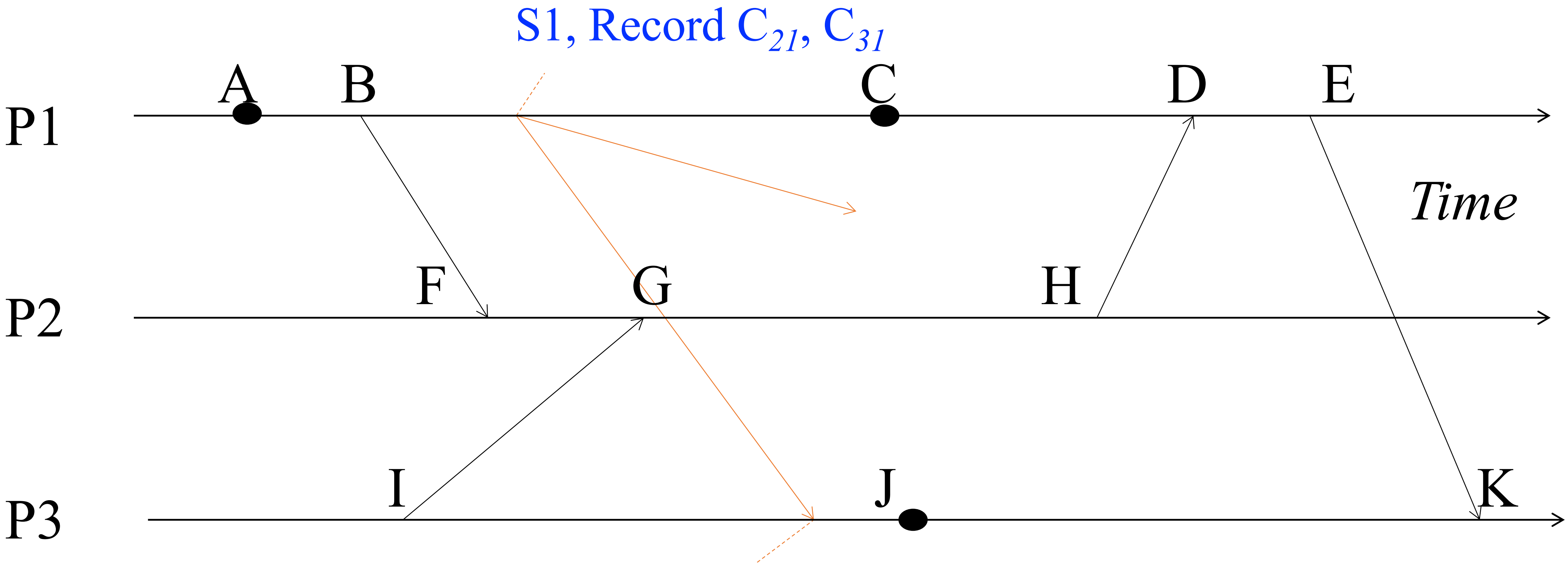
Example 1



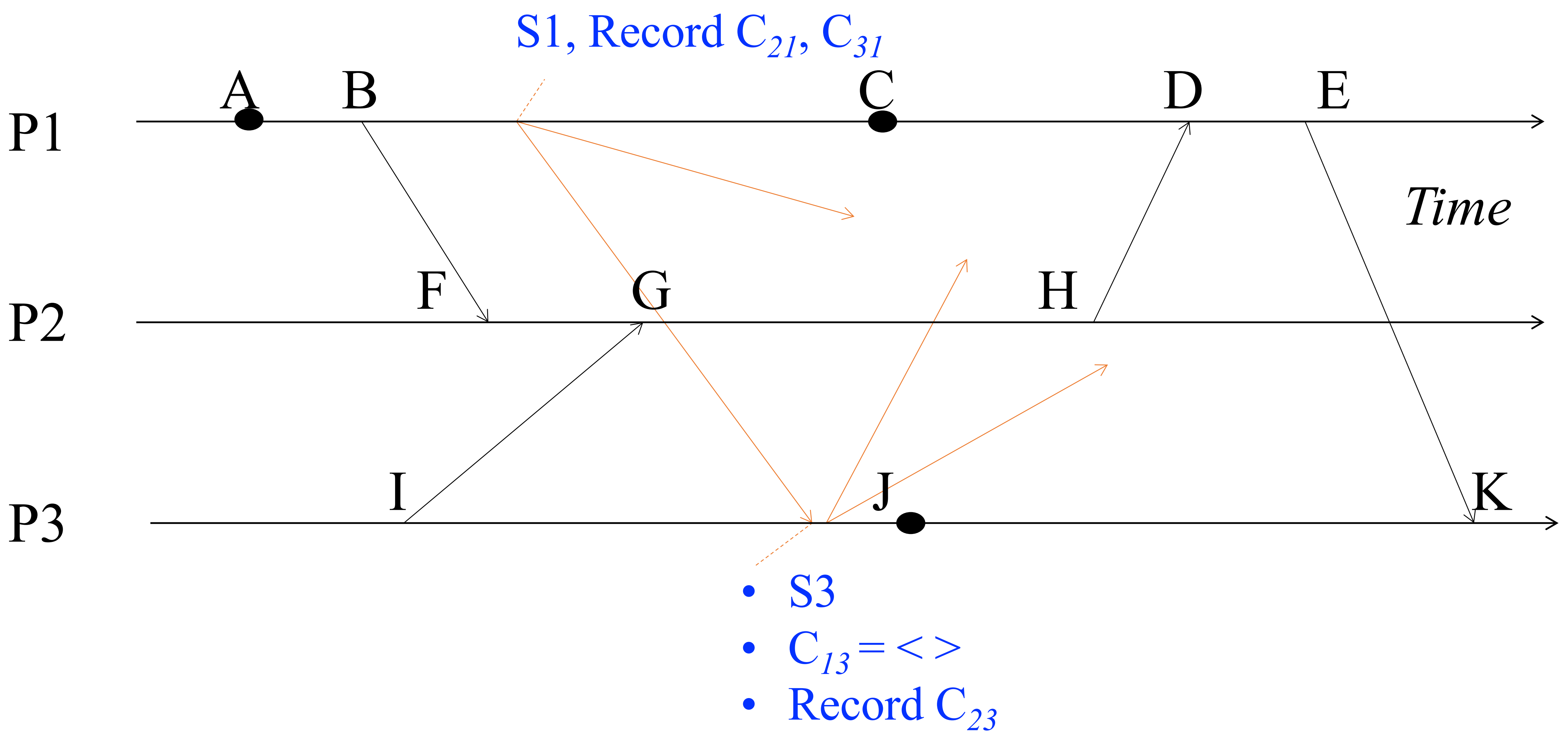
P1 is Initiator:

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

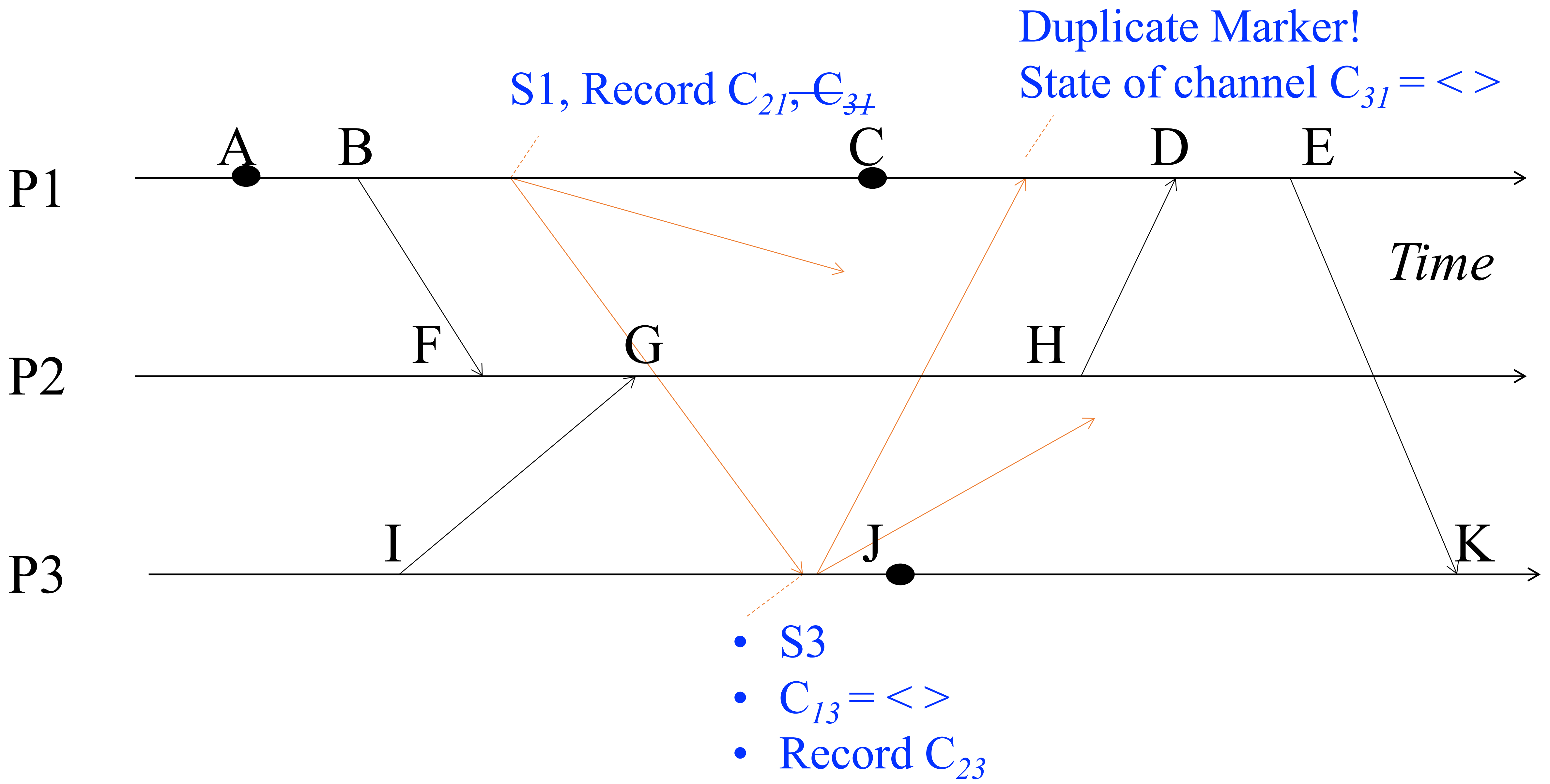


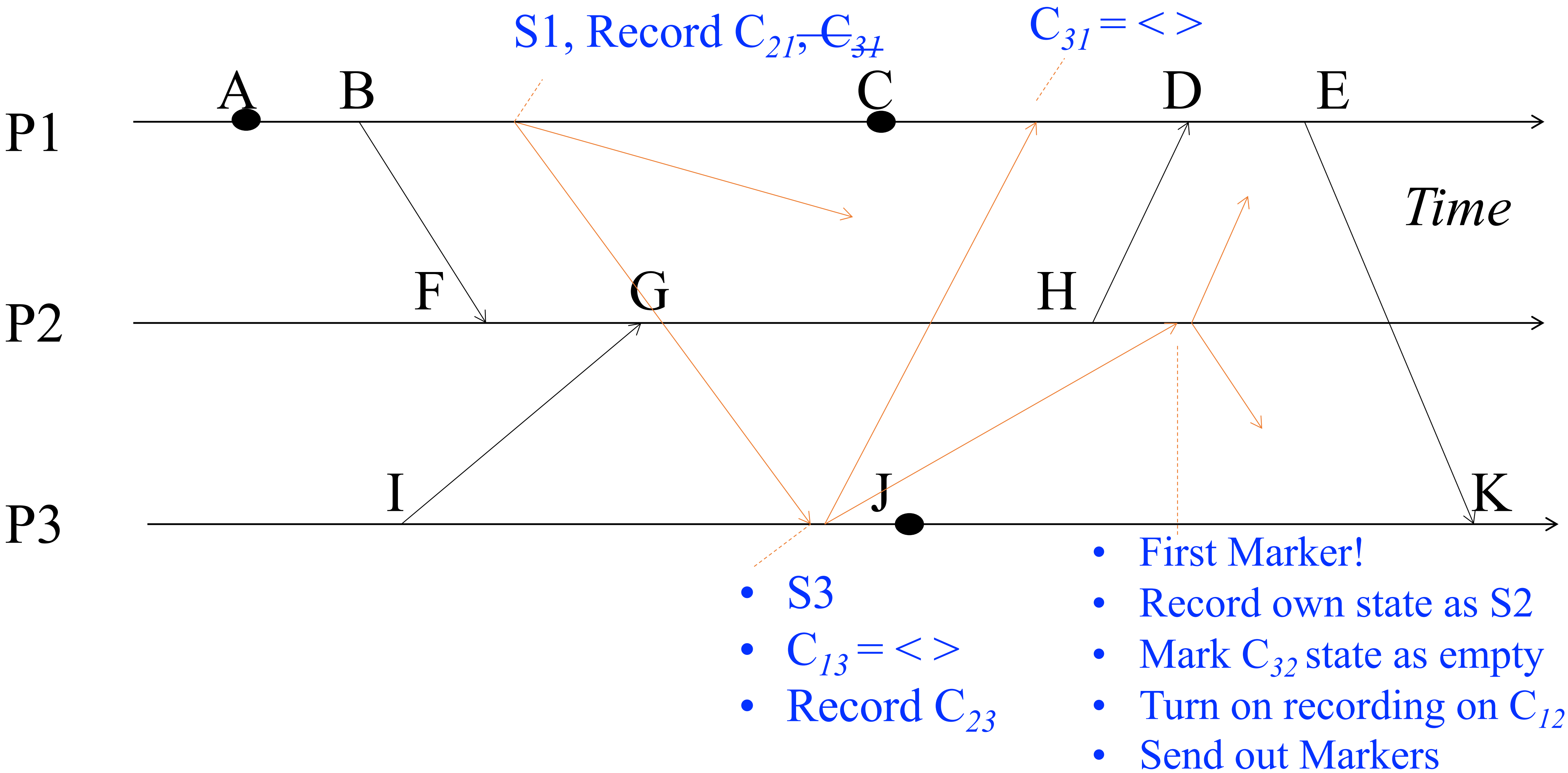


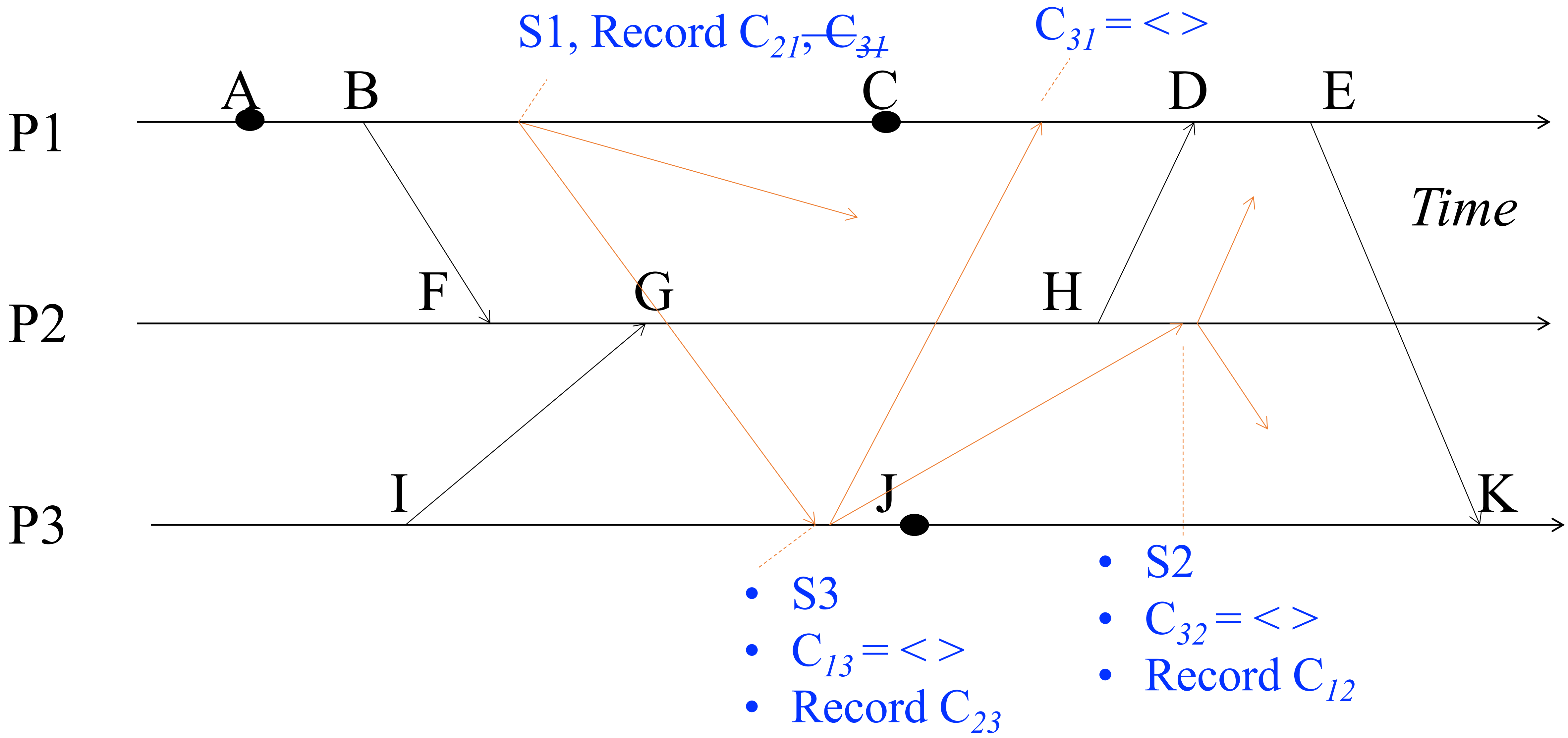
- First Marker!
- Record own state as S3
- Mark C_{13} state as empty
- Turn on recording on other incoming C_{23}
- Send out Markers

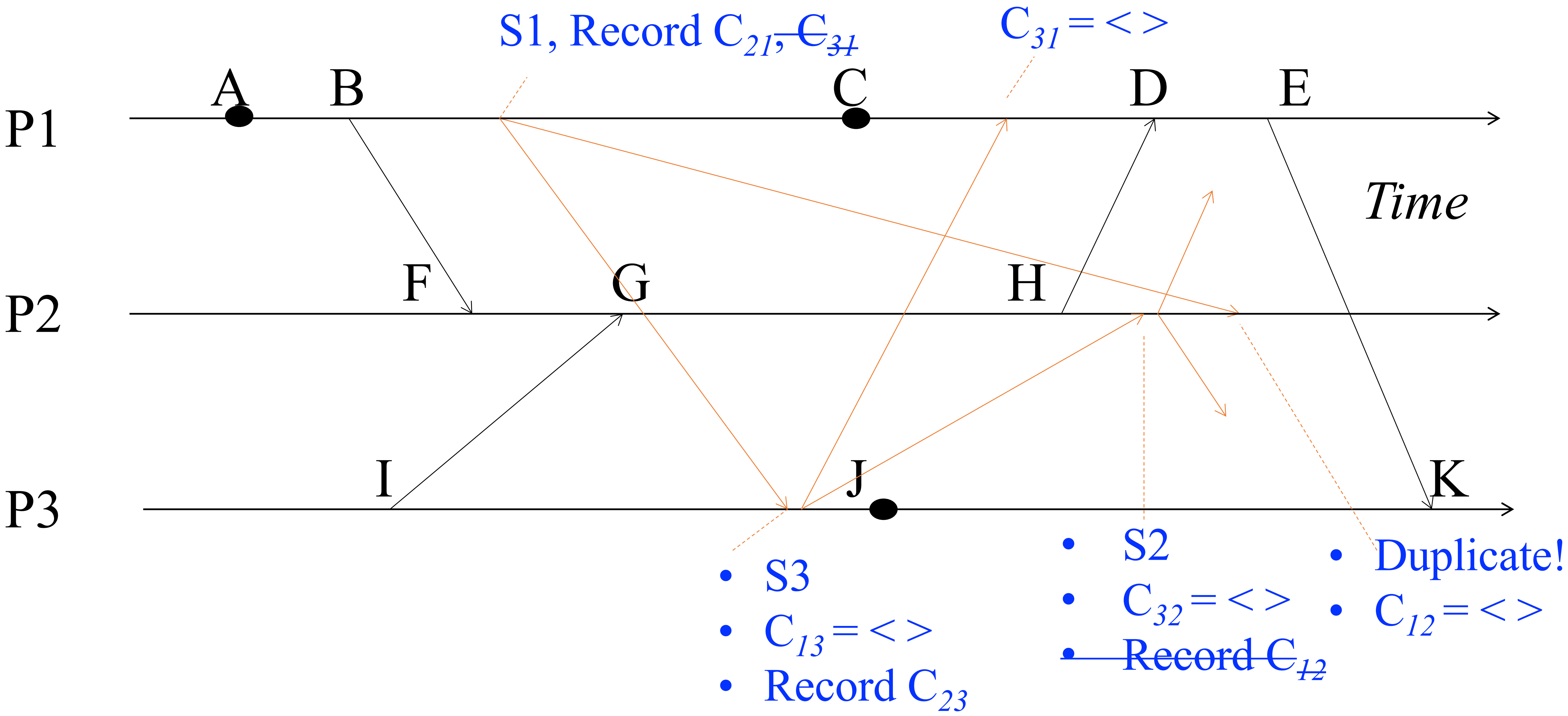


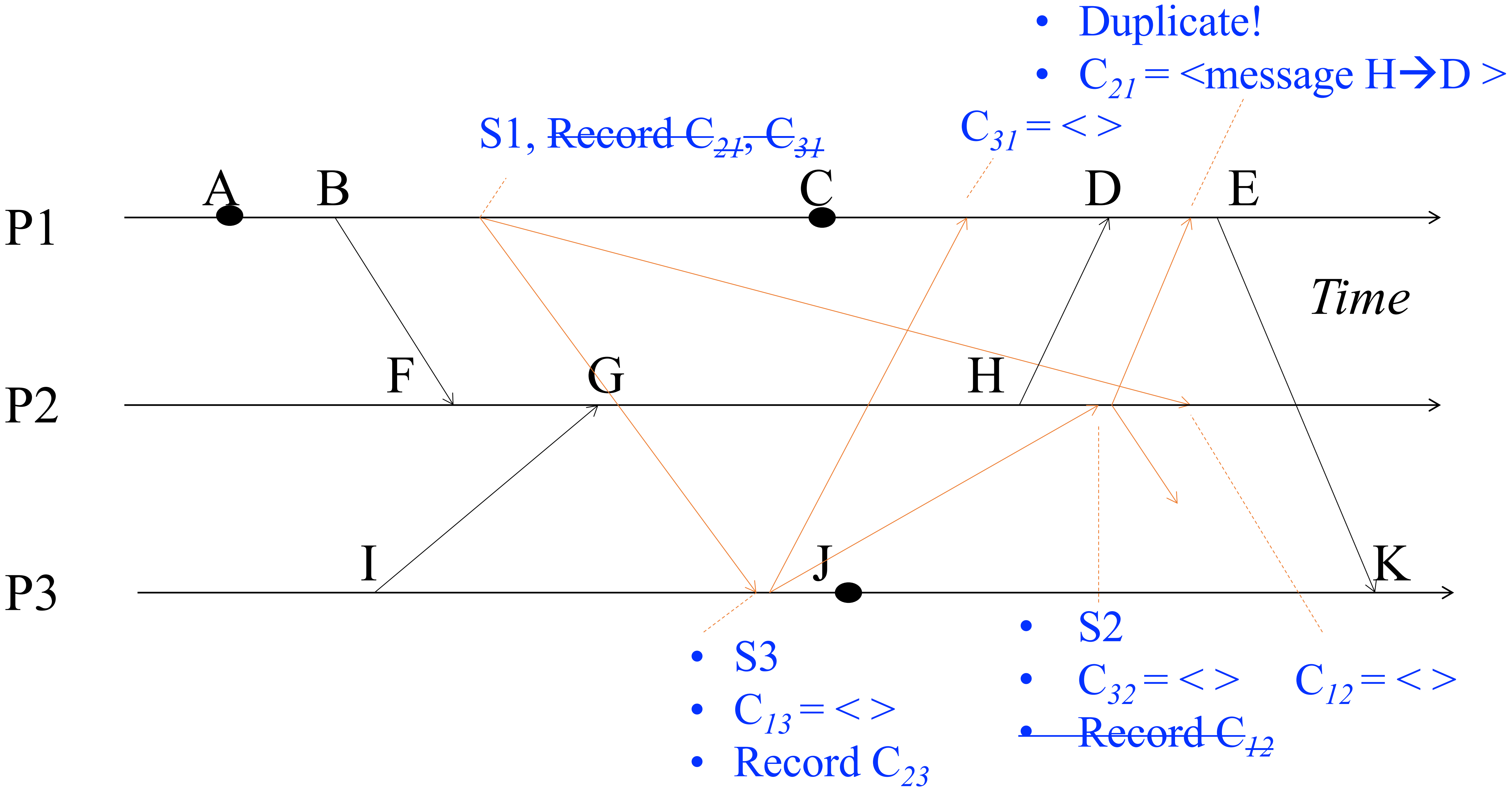
S1, Record C_{21}, C_{31}
A
B
C
D
E
P1
Time
F
G
H
P2
I
J
K
P3
• S3
• C_{13} = < >
• Record C_{23}

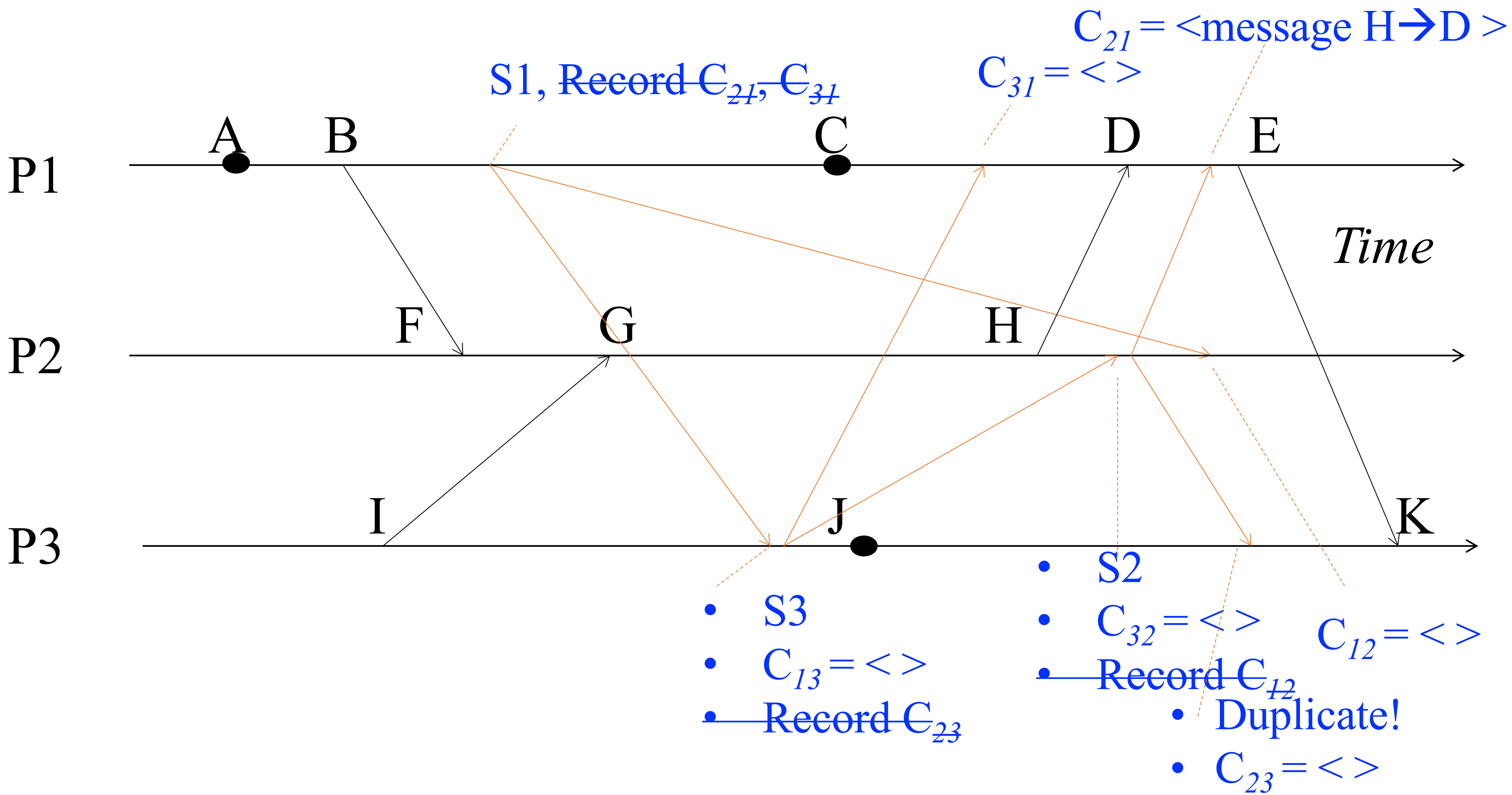

Duplicate Marker!
State of channel C31 = < >
S1, Record C21, C31
A
B
C
D
E
P1
Time
F
G
H
P2
I
J
K
P3
• S3
• C13 = < >
• Record C23


S1, Record C_{21}, ~~C_{31}~~
C_{31} = < >
A
B
C
D
E
P1
Time
F
G
H
P2
I
J
K
P3
• S3
• C_{13} = < >
• Record C_{23}
• First Marker!
• Record own state as S2
• Mark C_{32} state as empty
• Turn on recording on C_{12}
• Send out Markers


S1, Record C21, C31
C31 = < >
A
B
C
D
E
P1
Time
F
G
H
P2
I
J
K
P3
S3
C13 = < >
Record C23
S2
C32 = < >
Record C12

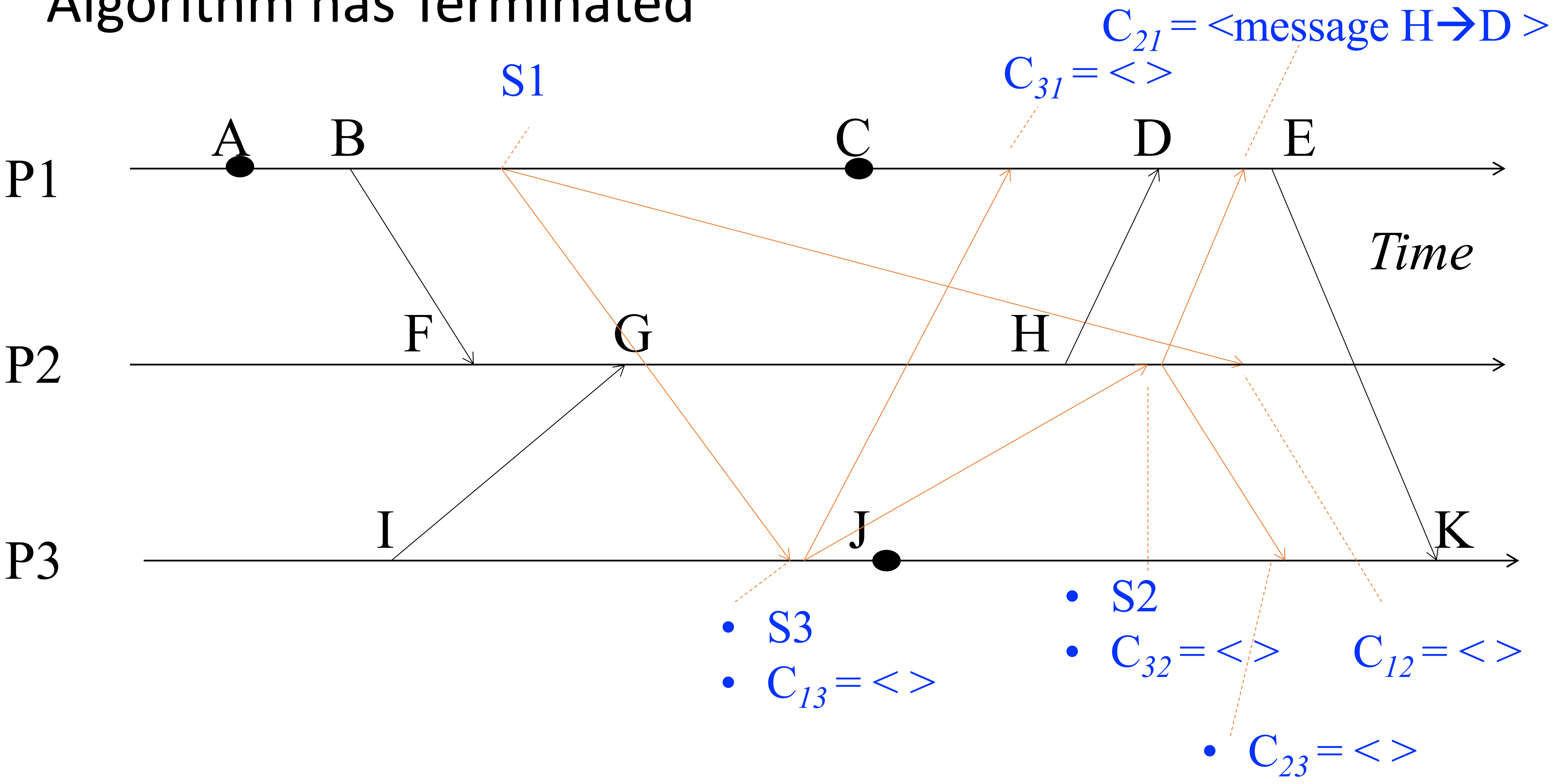

S1, Record C_{21}, ~~C_{31}~~
$C_{31} = < >$
A
B
C
D
E
P1
Time
F
G
H
P2
I
J
K
P3
• S3
• $C_{13} = < >$
• Record C_{23}
• S2
• $C_{32} = < >$
~~• Record C_{12}~~
• Duplicate!
• $C_{12} = < >$


• Duplicate!
• C21 = <message H→D >
S1, ~~Record C21, C31~~
C31 = < >
A
B
C
D
E
P1
Time
F
G
H
P2
I
J
K
P3
• S3
• C13 = < >
• Record C23
• S2
• C32 = < >
~~• Record C12~~
C12 = < >

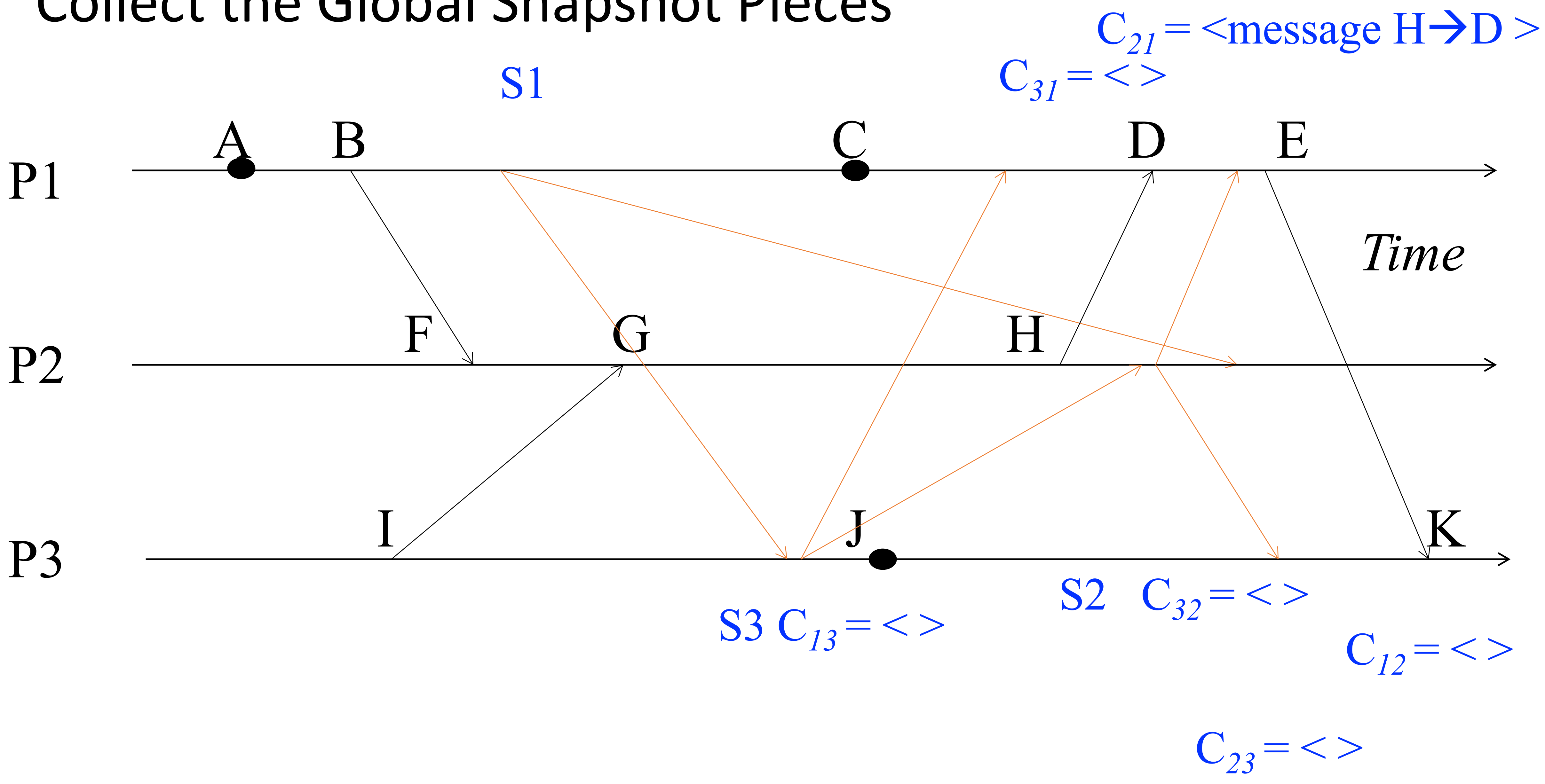


C_{21} = <message H→D >
C_{31} = < >
S1, ~~Record C_{21}, C_{31}~~
A
B
C
D
E
P1
Time
F
G
H
P2
I
J
K
P3
S3
C_{13} = < >
~~Record C_{23}~~
S2
C_{32} = < >
~~Record C_{12}~~
Duplicate!
C_{23} = < >
C_{12} = < >

Algorithm has Terminated

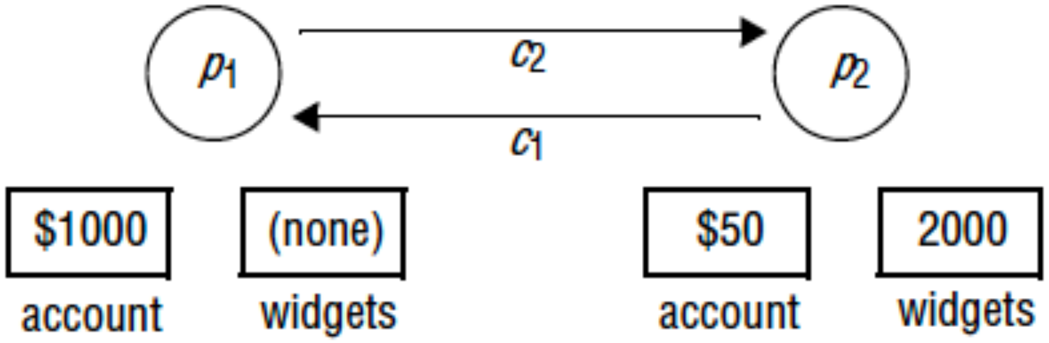


Collect the Global Snapshot Pieces

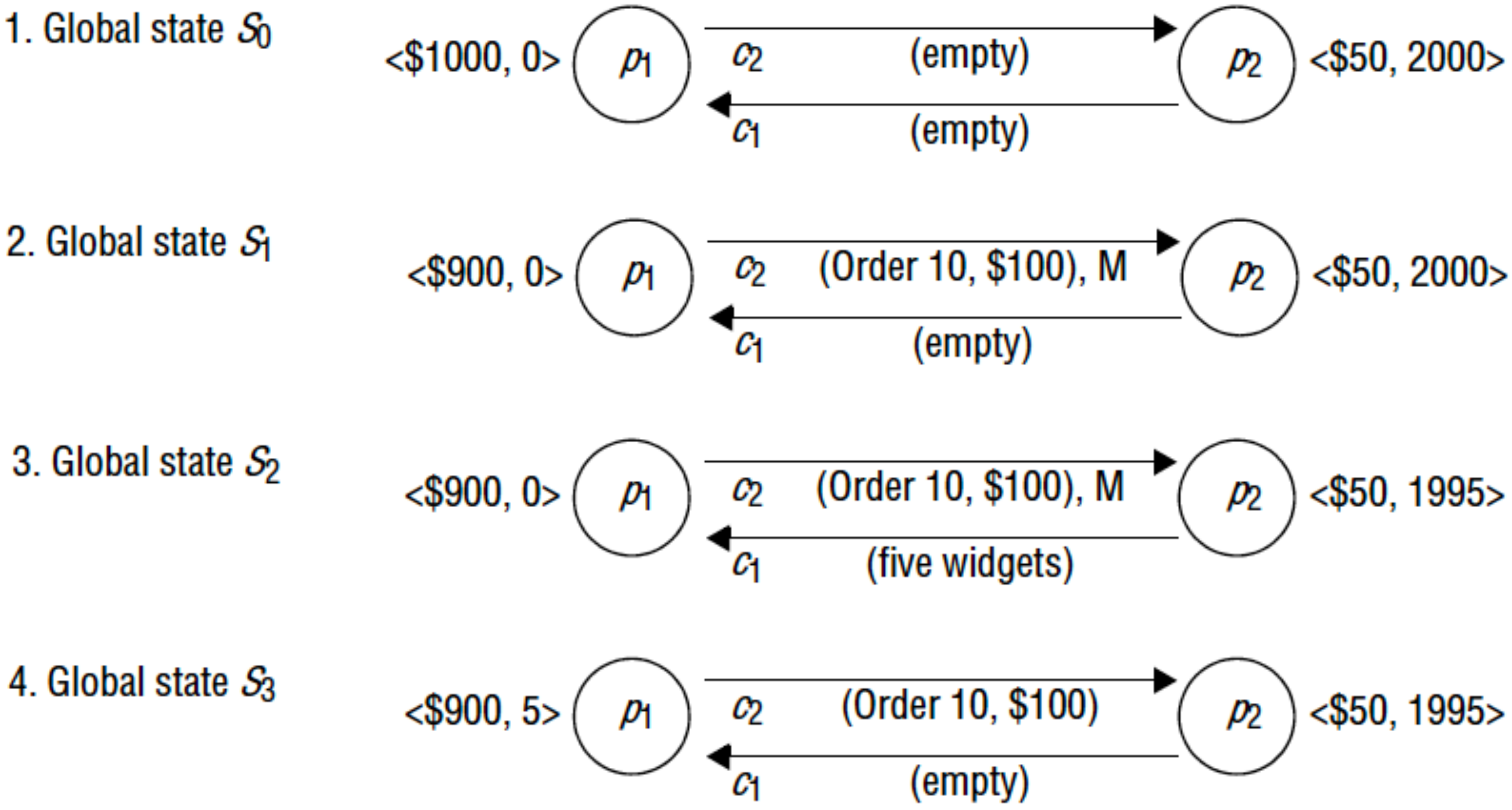


Example 2

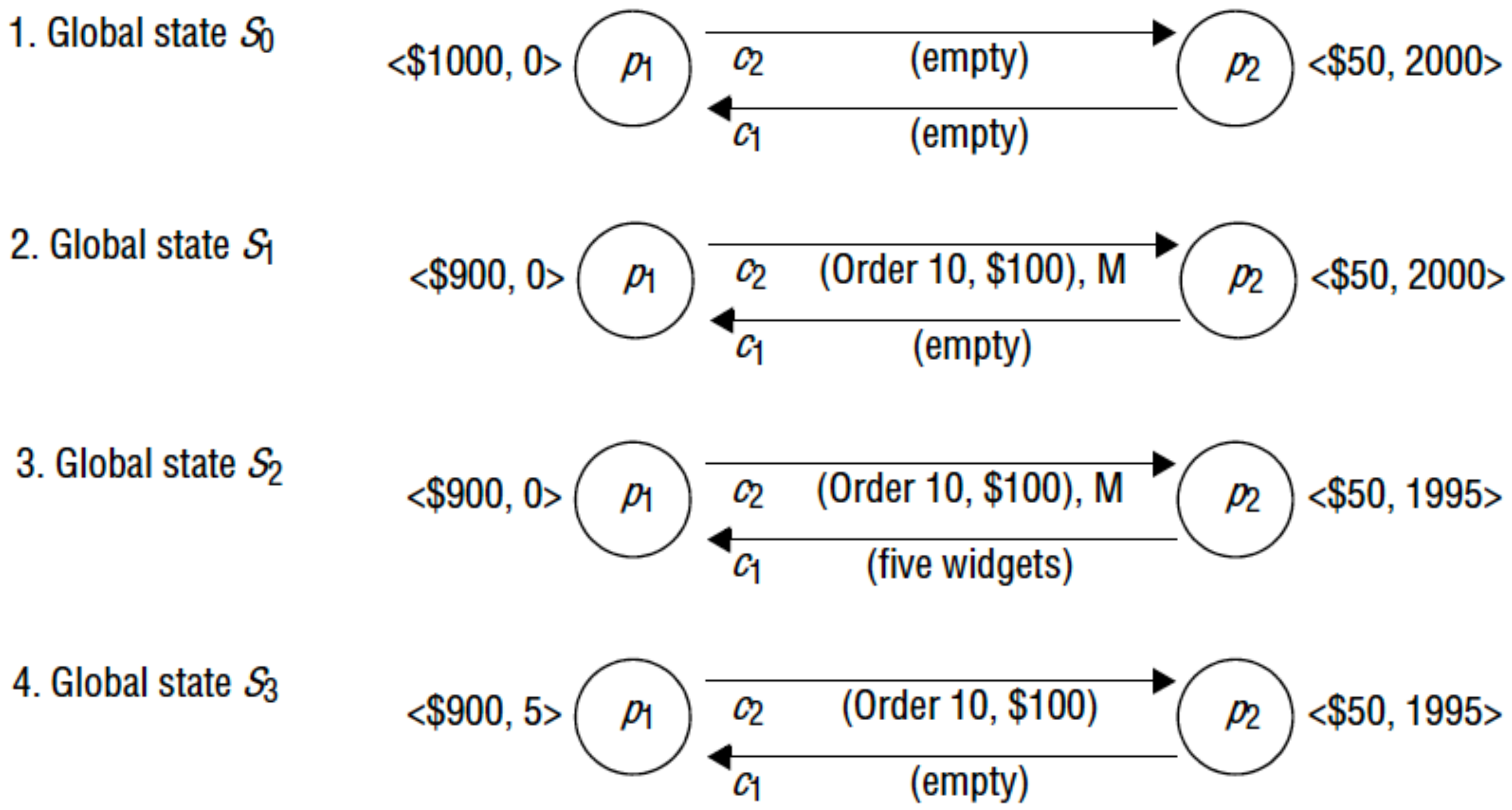
- Two processes trade in 'widgets'
- Process p_1 sends orders for widgets over c_2 to p_2, enclosing payment at the rate of $10 per widget
- Some time later, process p_2 sends widgets along channel c_1 to p_1



An Execution of the System



An Execution of the System



The snapshot state is $p_1: \langle \$1000{,}0 \rangle,\ p_2: \langle \$50{,}1995 \rangle, c1: \langle \text{five widgets} \rangle, c_2: \langle \quad \rangle$

- this state differs from all the global states through which the system actually passed

Termination

- Theorem: The Chandy–Lamport algorithm ensures that eventually all processes turn red and all channels are closed
- Proof sketch:
  - We assume that a process that has received a marker message records its state within a finite time and sends marker messages over each outgoing channel within a finite time.
  - If there is a path of communication channels and processes from a process p_i to a process p_j, then p_j will record its state a finite time after p_i recorded its state and close its incoming channel from p_i. It will then send a marker to p_i (if p_i is an outgoing neighbor of p_j) so that p_i will close its incoming channel from p_j within a finite time.
  - Since the graph of processes and channels to be strongly connected, all processes will have recorded their states and the states of incoming channels a finite time after some process initially records its state.

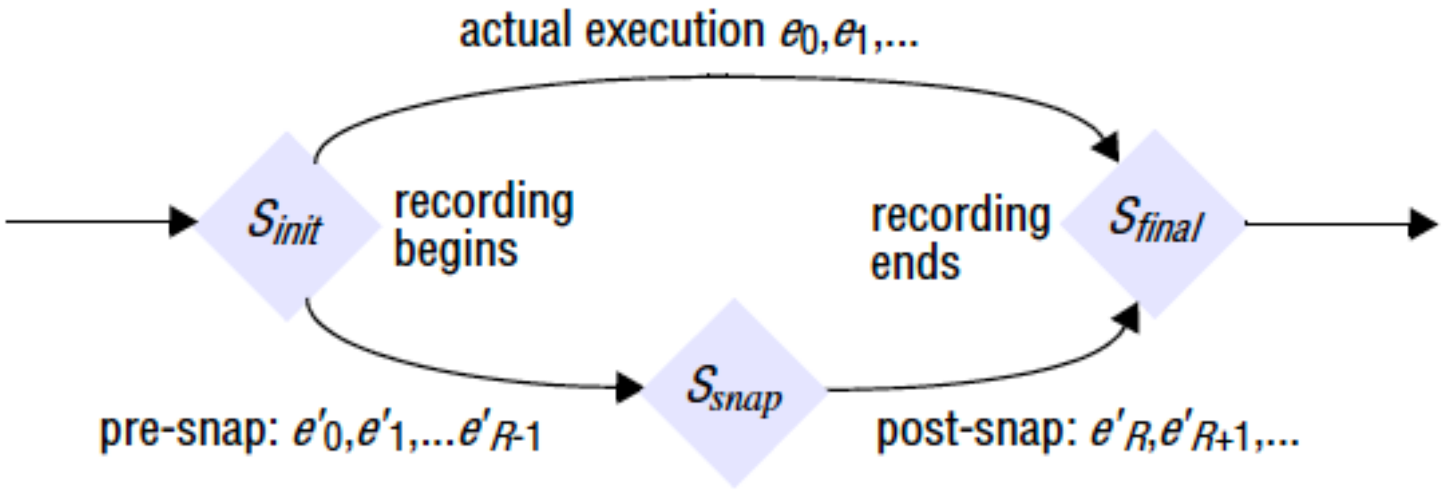
Correctness

- Theorem: The Chandy–Lamport algorithm records a consistent cut/global state and all the WR messages
- Proof sketch (of the first part):
  - Let e_i and e_j be events occurring at p_i and p_j, respectively, such that $e_i \rightarrow e_j$. We assert that if e_j is in the cut, then e_i is in the cut
  - That is, if e_j occurred before p_j recorded its state, then e_i must have occurred before p_i recorded its state. This is obvious if $i = j$. Assume $i \neq j$.
  - Assume p_i recorded its state before e_i occurred (proof by contradiction)
  - Consider the sequence of H messages $m_1, m_2, \dots, m_H$, giving rise to $e_i \rightarrow e_j$.
  - By FIFO ordering of channels and the marker sending and receiving rules, a marker message would have reached p_j ahead of each of $m_1, m_2, \dots, m_H$, so that p_j would have recorded its state before e_j, a contradiction.

Understanding snapshot

- The snapshot state is a consistent global state that is reachable from the initial state. It may not actually be visited during a specific execution
- The final state of the original computation is always reachable from the snapshot state
- Proof in the textbook

Reachability



- If a stable predicate is True in the state S_{snap} then we may conclude that the predicate is True in the state S_{final}
- If the predicate evaluates to False for S_{snap} , then it must also be False for S_{init}