Global State and Gossip

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: Computing Systems and Concurrency
Lecture 6

Marco Canini

Credits: Indranil Gupta developed much of the original material.

Today

1. **Global snapshot of a distributed system**

2. Chandy-Lamport’s algorithm

3. Gossip

Distributed snapshot

- Let's think of this as a picture of all servers and their states comprising a distributed system
- How do you calculate a "global snapshot" in a distributed system?
- What does a "global snapshot" even mean?
- Why is the ability to obtain a "global snapshot" important?

Some uses of global system snapshot

- **Checkpointing**
  – can restart distributed system on failure
- **Gargabe collection** of objects
  – objects at servers that don't have any other objects (at any servers) with references to them
- **Deadlock detection**
  – useful in database transaction systems
- **Termination of computation**
  – useful in batch computing systems
- **Debugging**
  – useful to inspect the global state of the system

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 <u>each process</u>

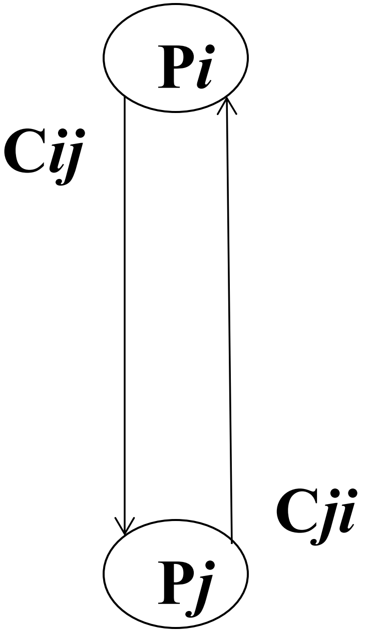
- And the instantaneous *state* of <u>each communication channel</u>, i.e., *messages* in transit on the channels

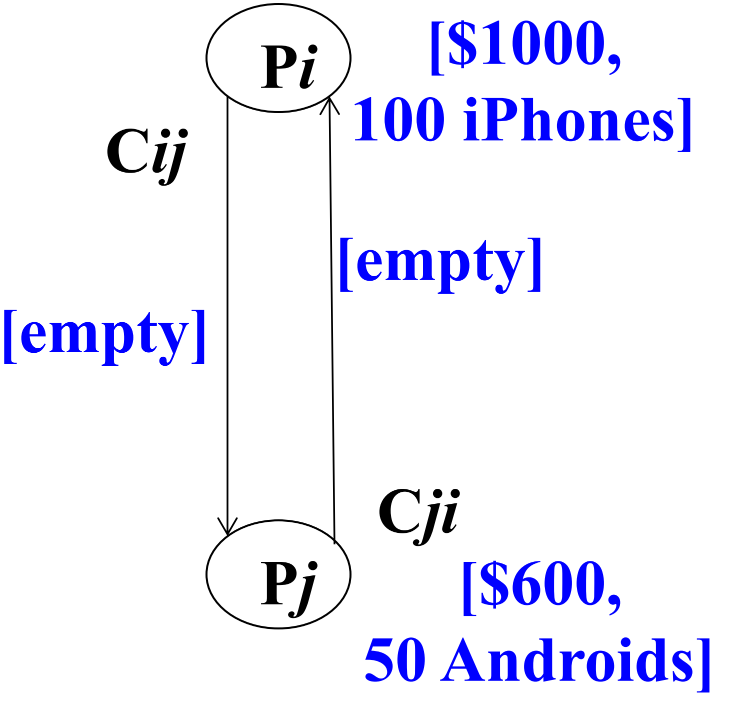
A strawman solution

- Synchronize clocks of all processes
- Ask all processes to record their states at known time *t*
- Problems?
  - Time synchronization always has error
    - Your bank might inform you, “We lost the state of our distributed cluster due to a 1 ms clock skew in our snapshot algorithm.”
  - Also, does not record the state of messages in the channels

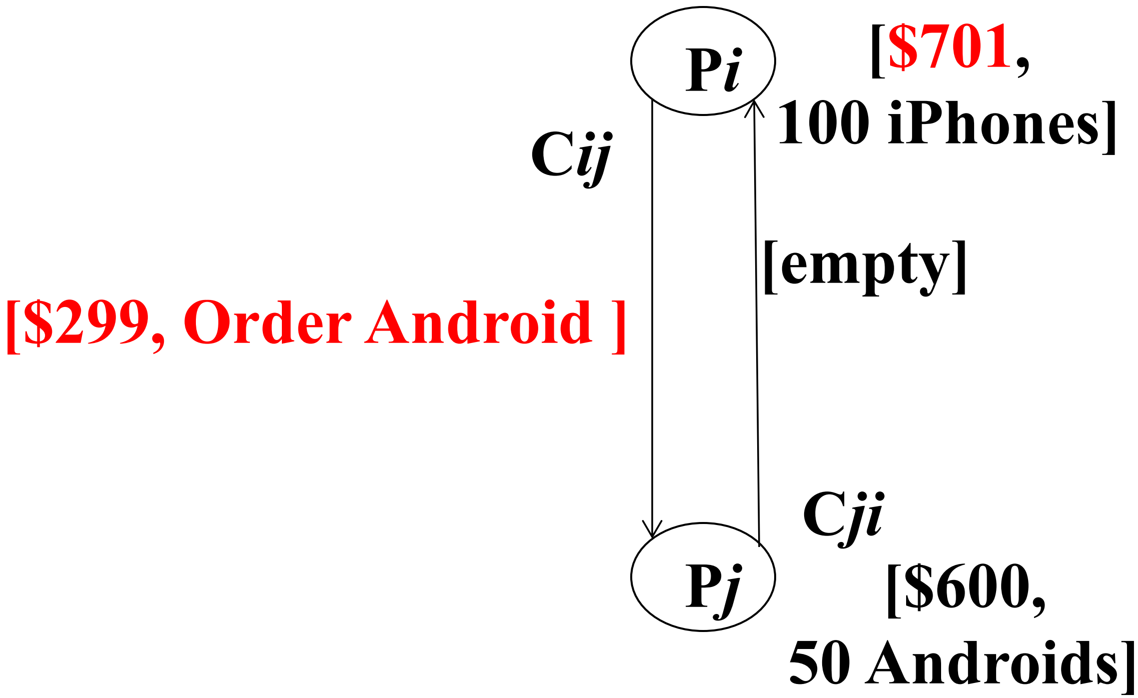
- Again: synchronization not required – causality is enough!

Example

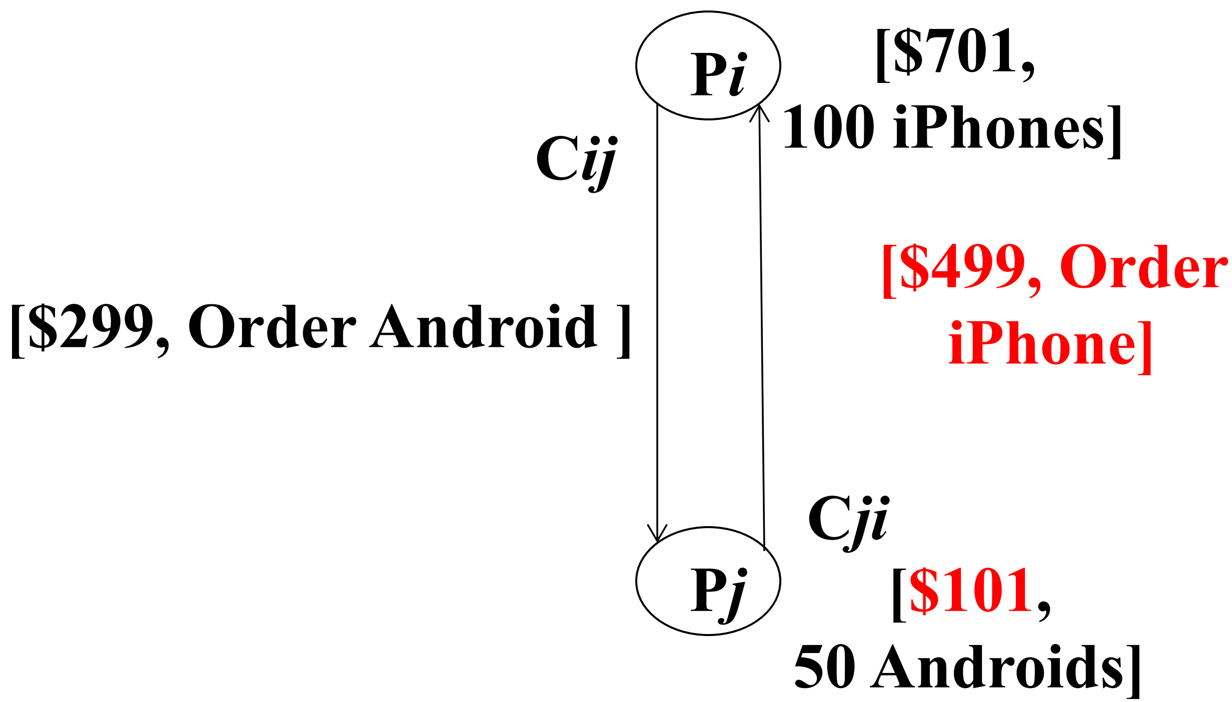




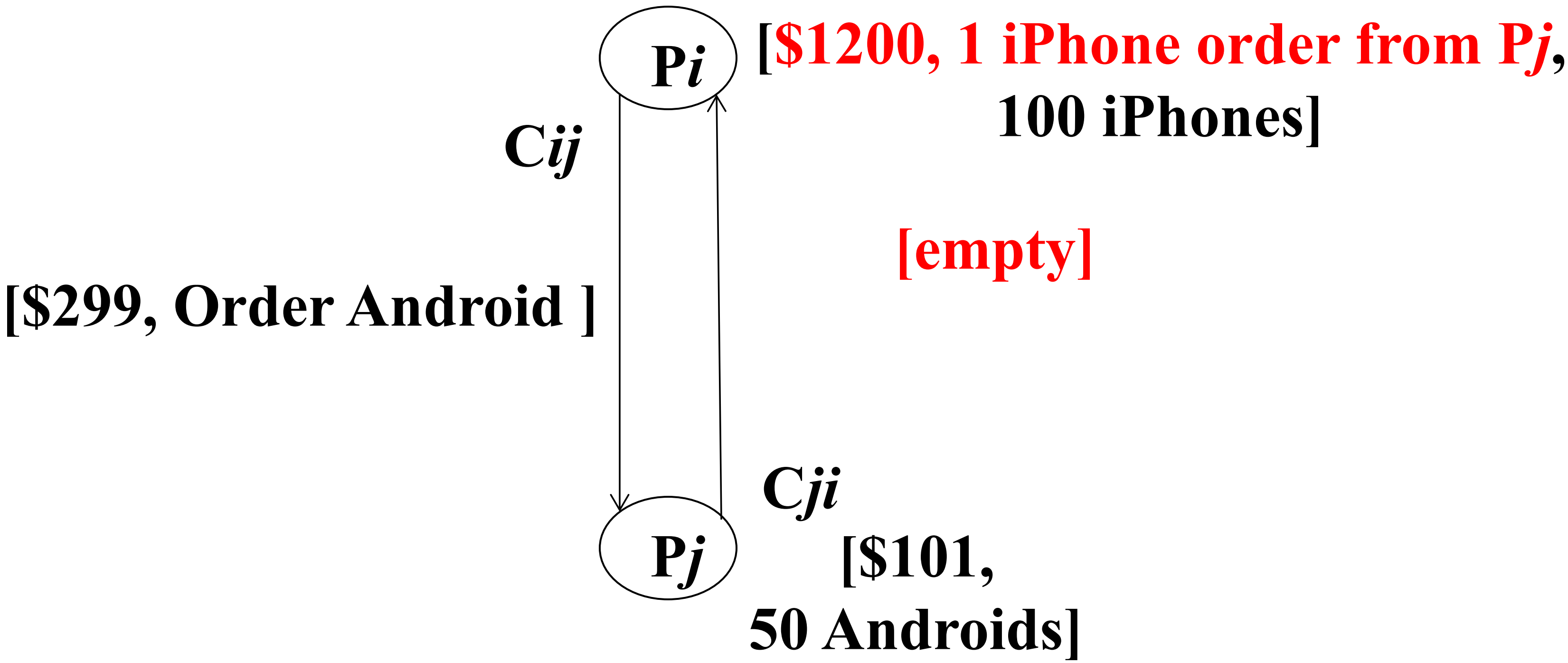
[Global Snapshot 0]



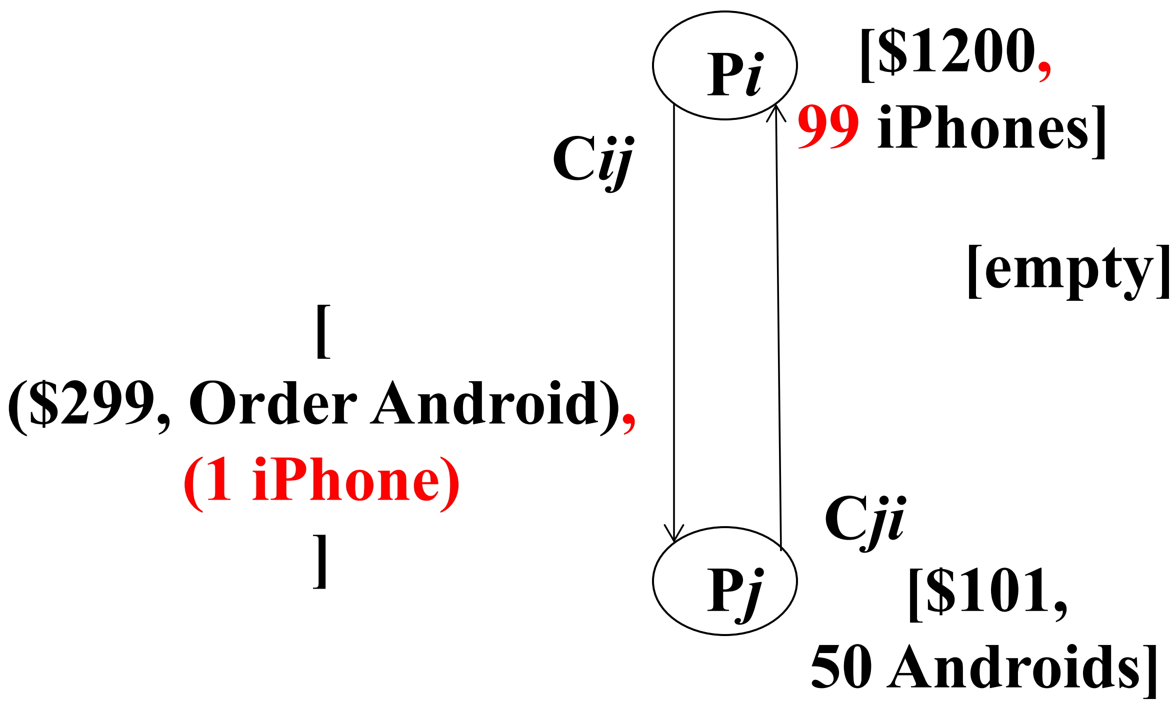
[Global Snapshot 1]



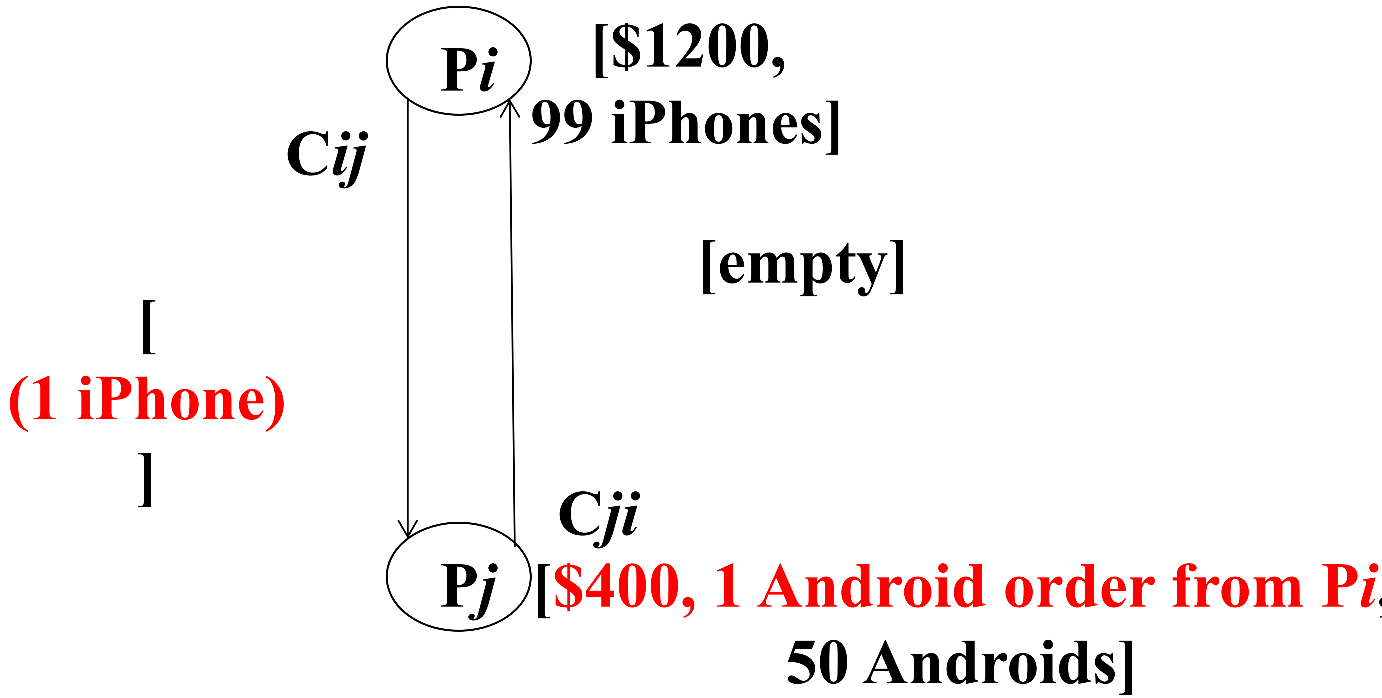
[Global Snapshot 2]



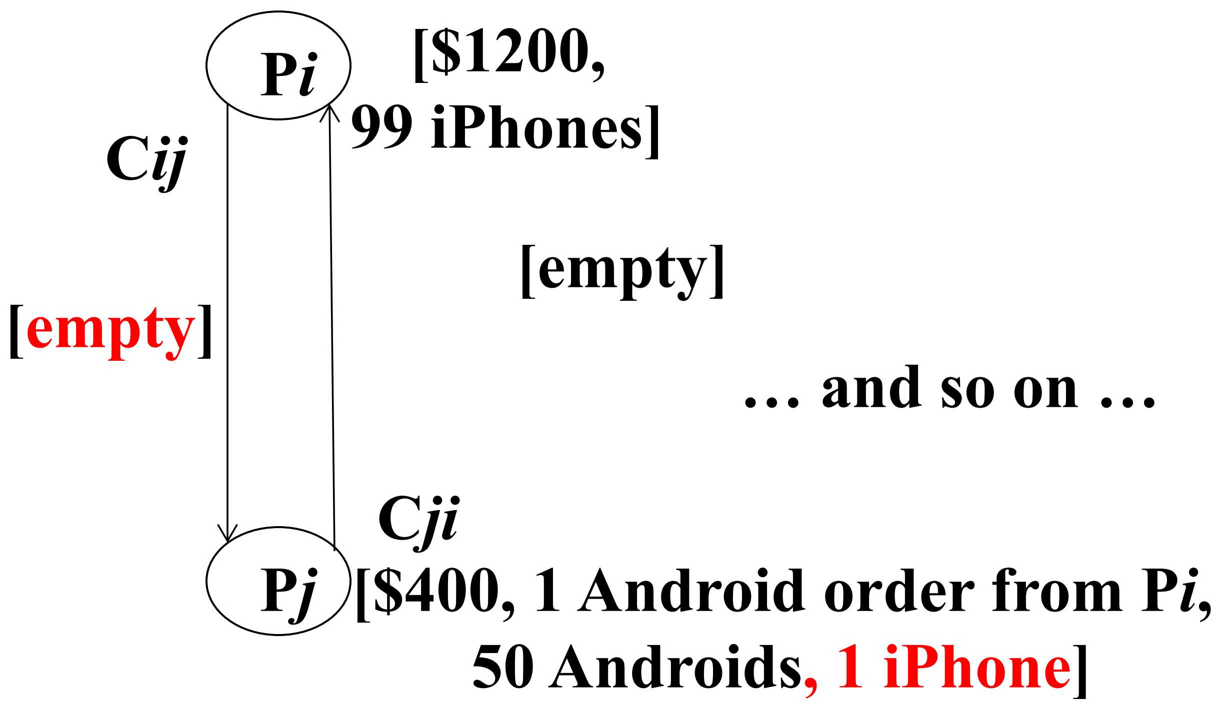
[Global Snapshot 3]



[Global Snapshot 4]



[Global Snapshot 5]



… and so on …

[Global Snapshot 6]

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 <u>obeys causality</u>**
  - Next: Causal algorithm for Global Snapshot calculation

Today

1. Global snapshot of a distributed system

2. **Chandy-Lamport's algorithm**

3. Gossip

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 ordered process pair P*j* → P*i* and P*i* → P*j*
  - Communication channels are FIFO-ordered
    - First in First out
  - No failure
  - All messages arrive intact, and are not duplicated
    - Other papers later relaxed some of these assumptions

Requirements

- **Snapshot should not interfere with normal application actions, and it should not require application to stop sending 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)
- **Global state is collected in a distributed manner**
- **Any process may initiate the snapshot**
  - We'll assume just one snapshot run for now

Chandy-Lamport Global Snapshot Algorithm

- **First: Initiator P*i* records its own state**
- **Initiator process creates special messages called "Marker" messages**
  - – 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}
  - (*N-1*) channels
- 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 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
  - 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}

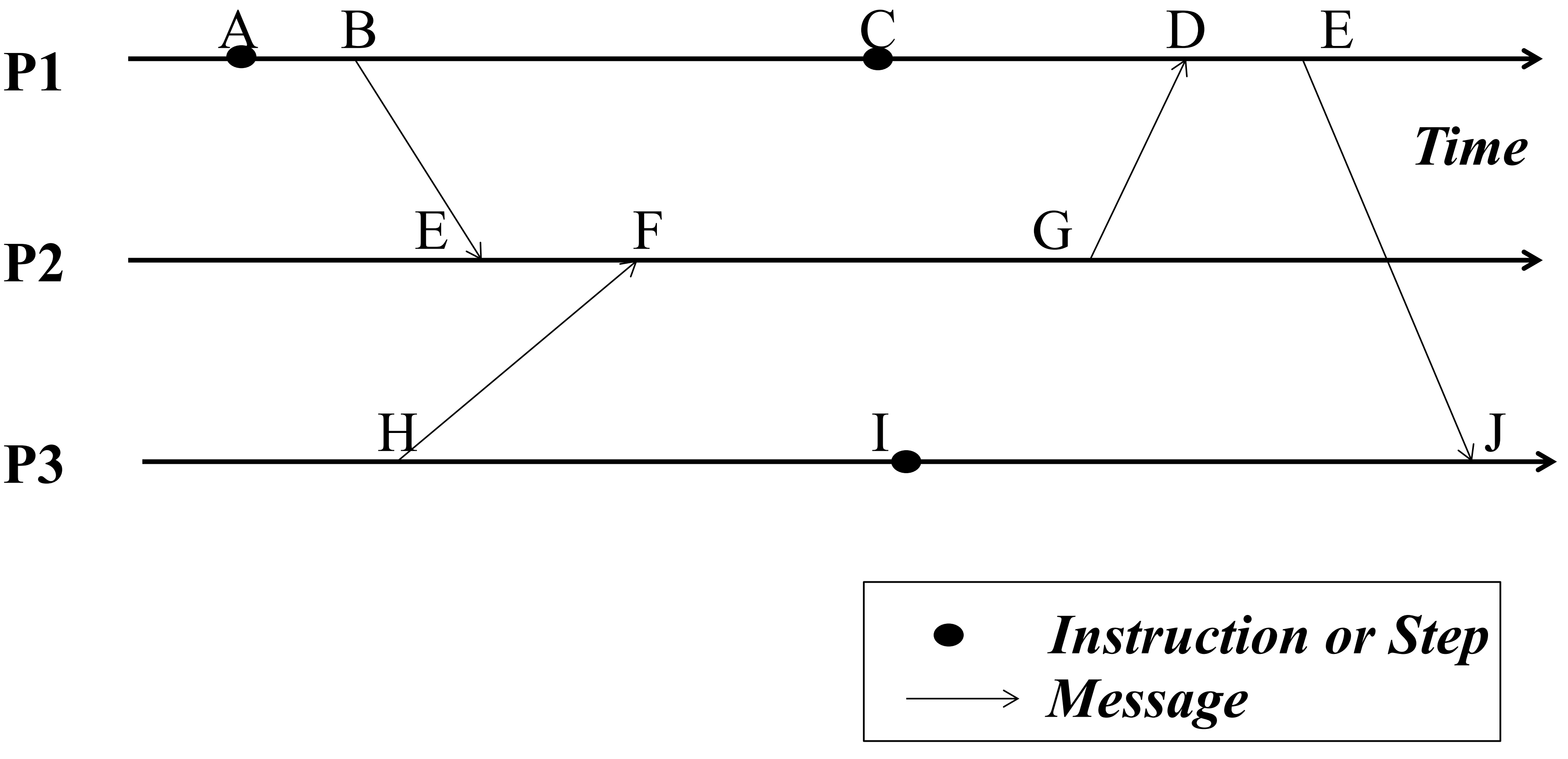
Chandy-Lamport Global Snapshot Algorithm (3)

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

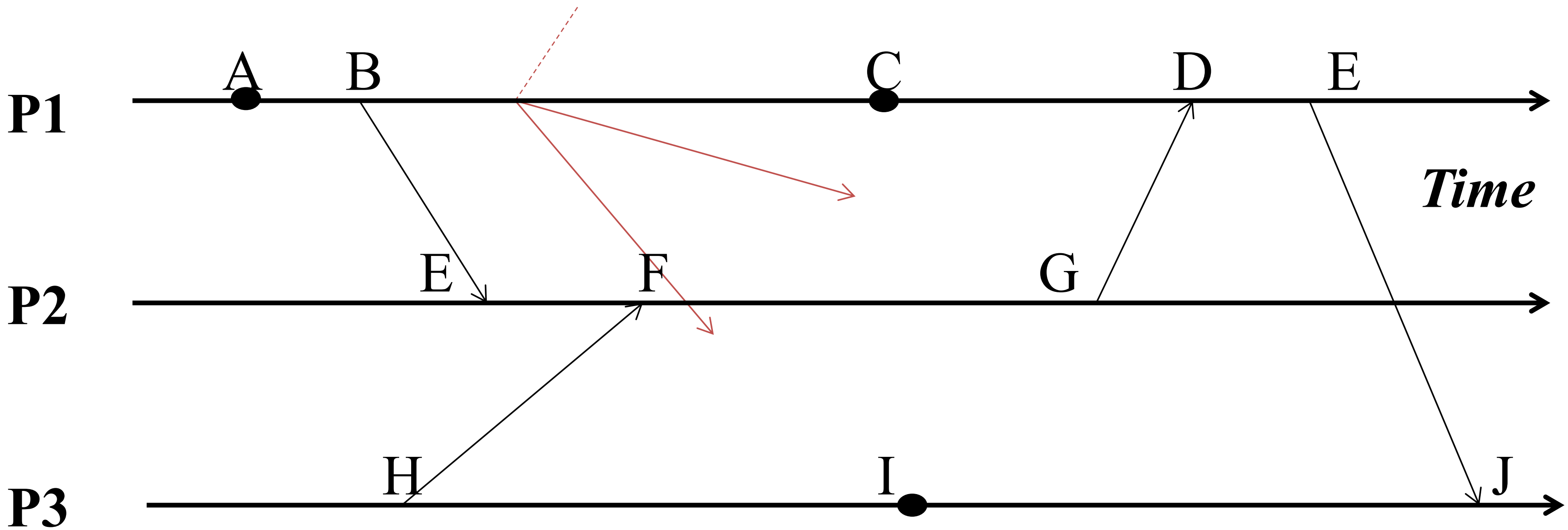
Then, (if needed), a central server collects all these partial state pieces to obtain the full global snapshot

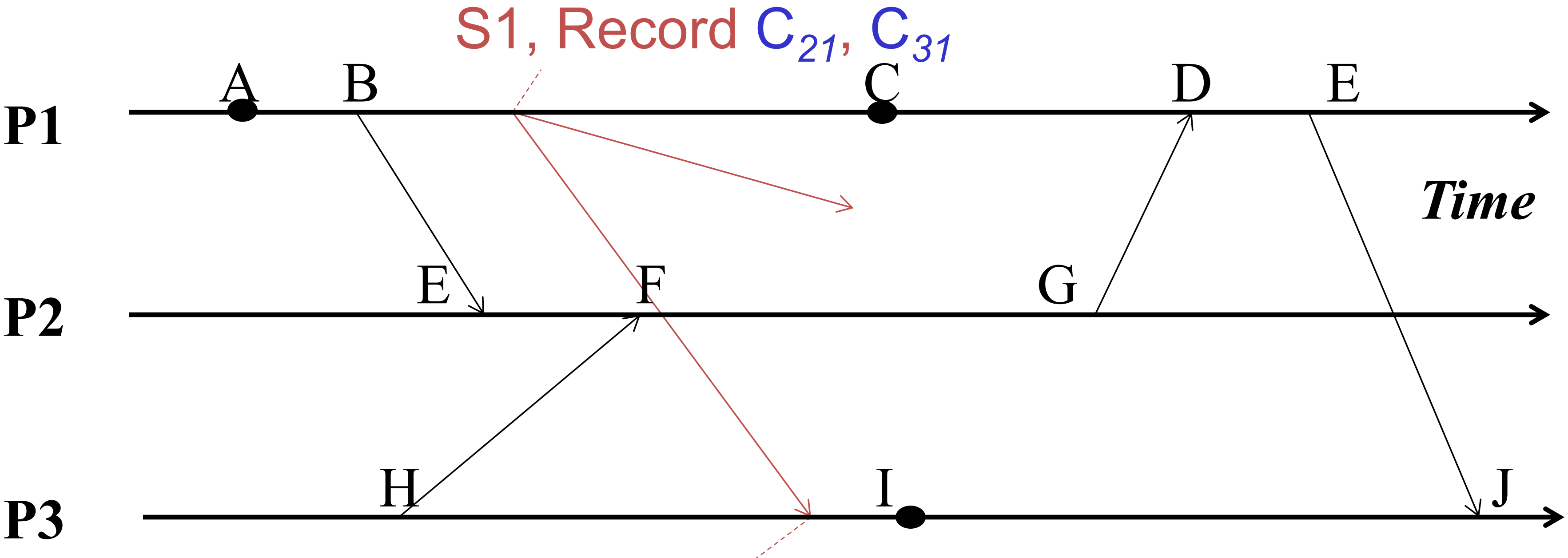
Example



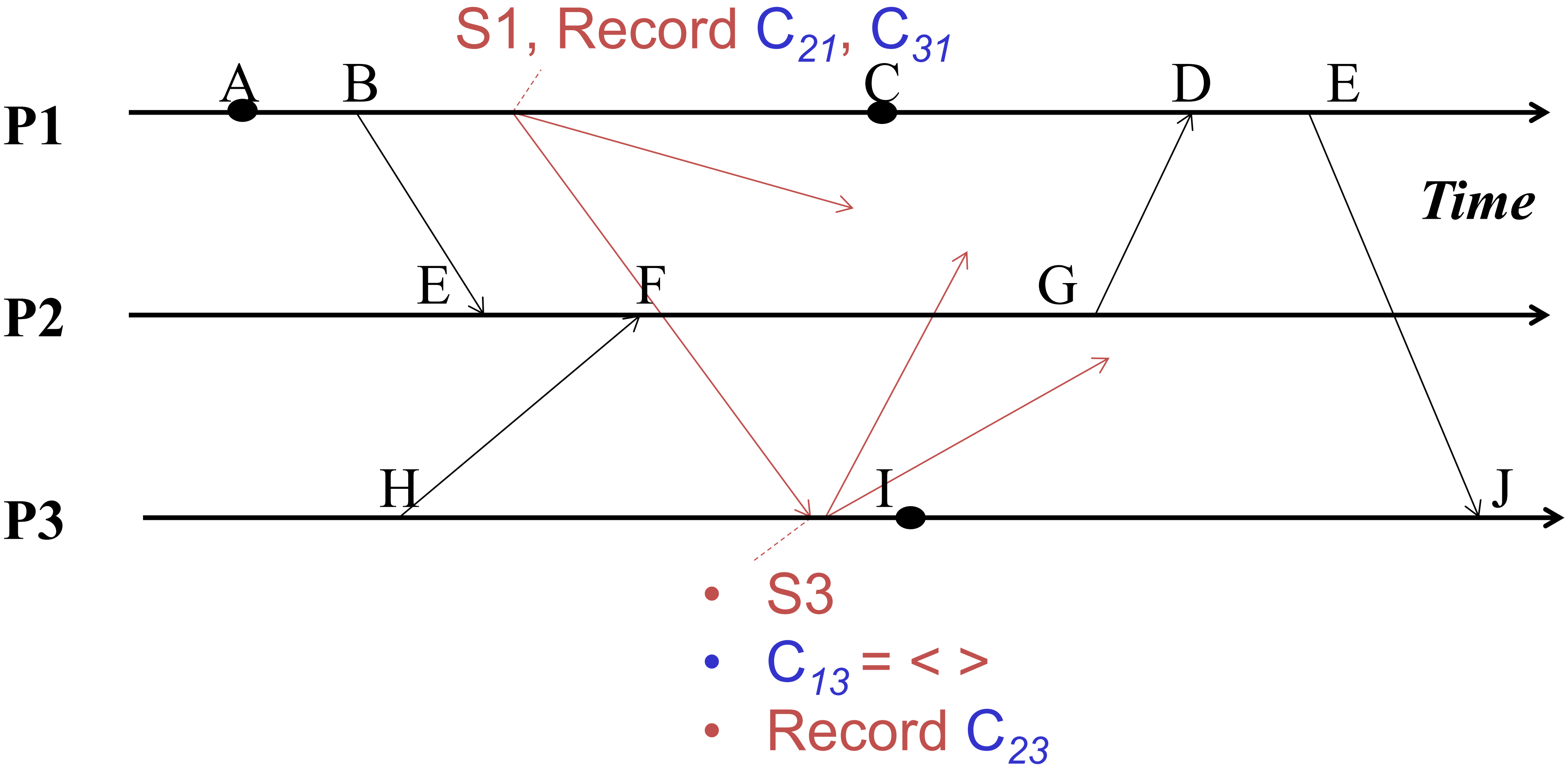
P1 is Initiator:

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

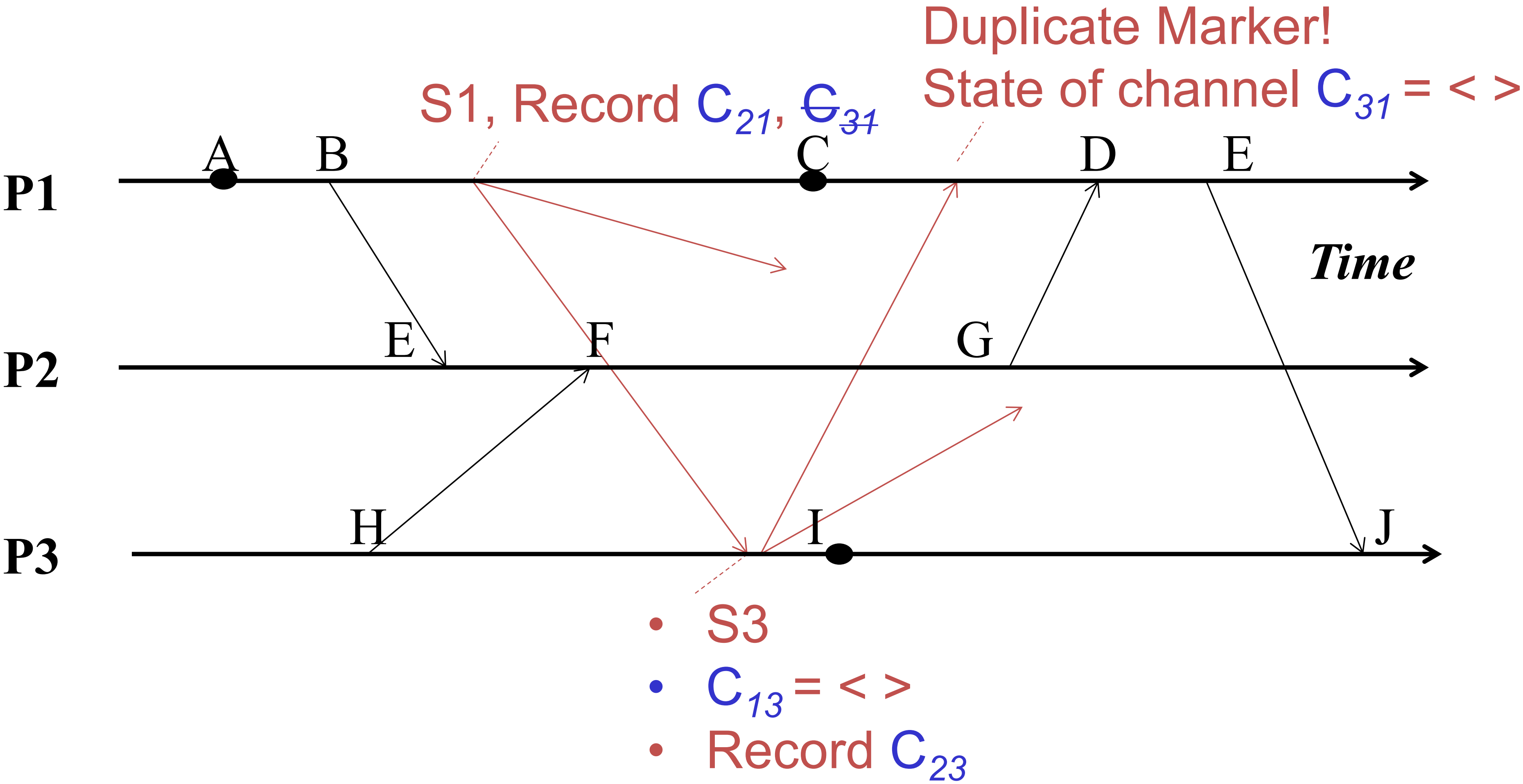


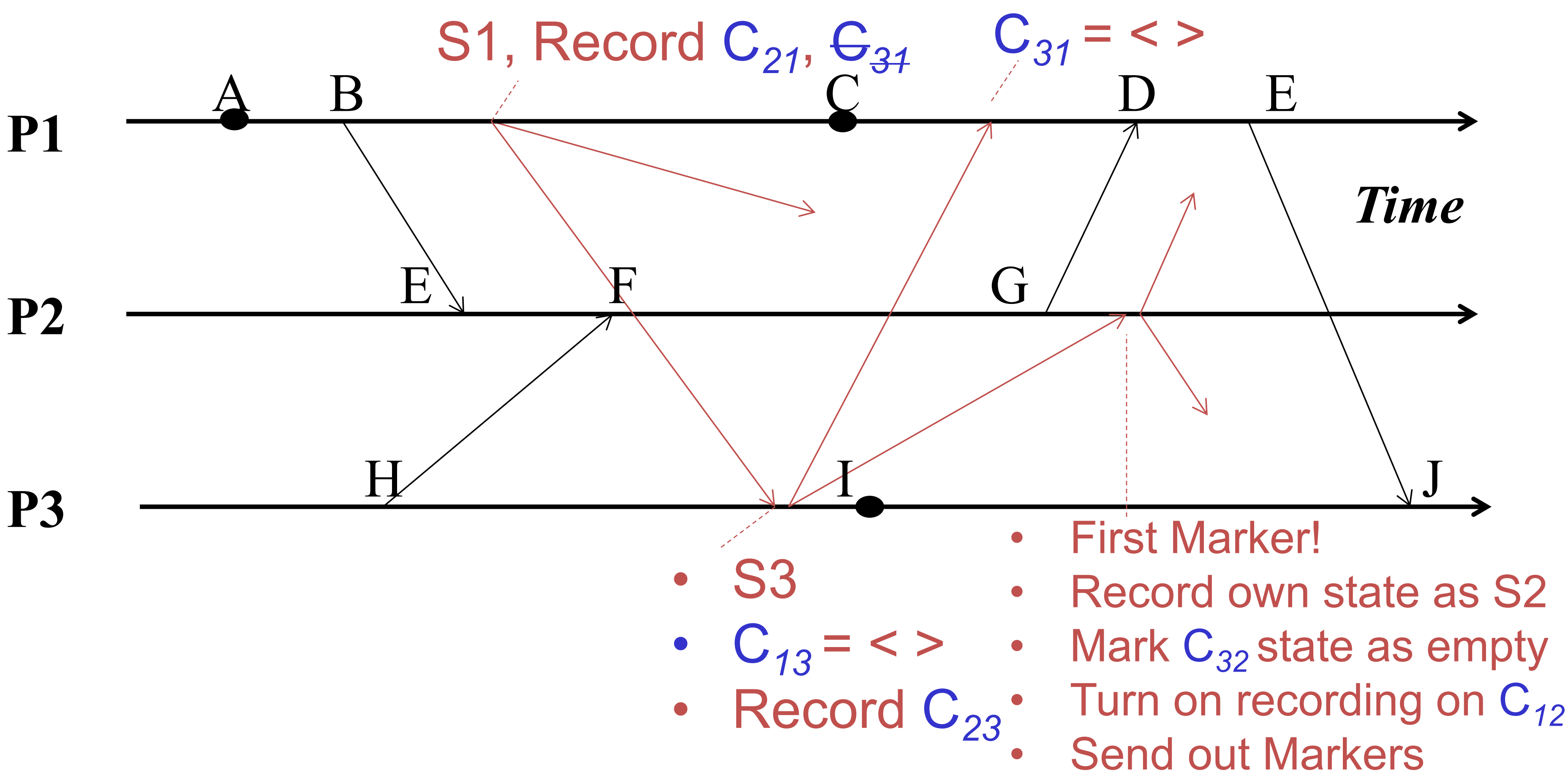


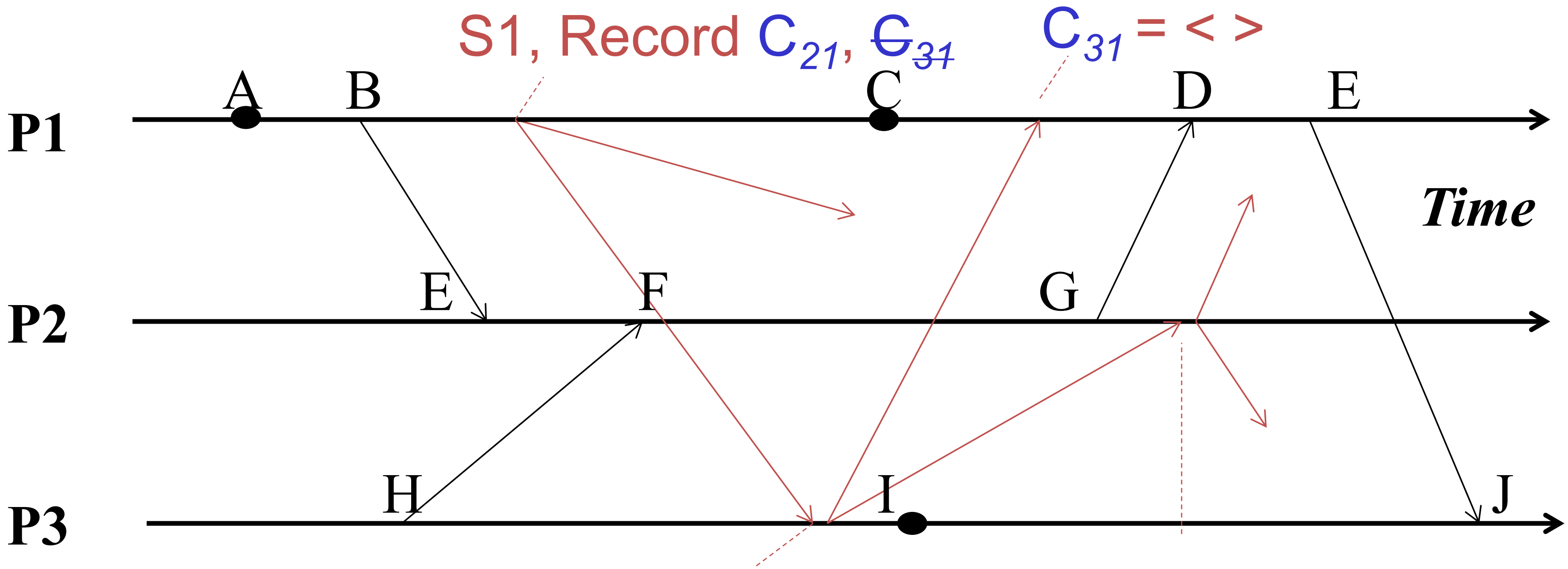
- First Marker!
- Record own state as S3
- Mark C_{13} state as empty
- Turn on recording on other incoming C_{23}
- Send out Markers



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

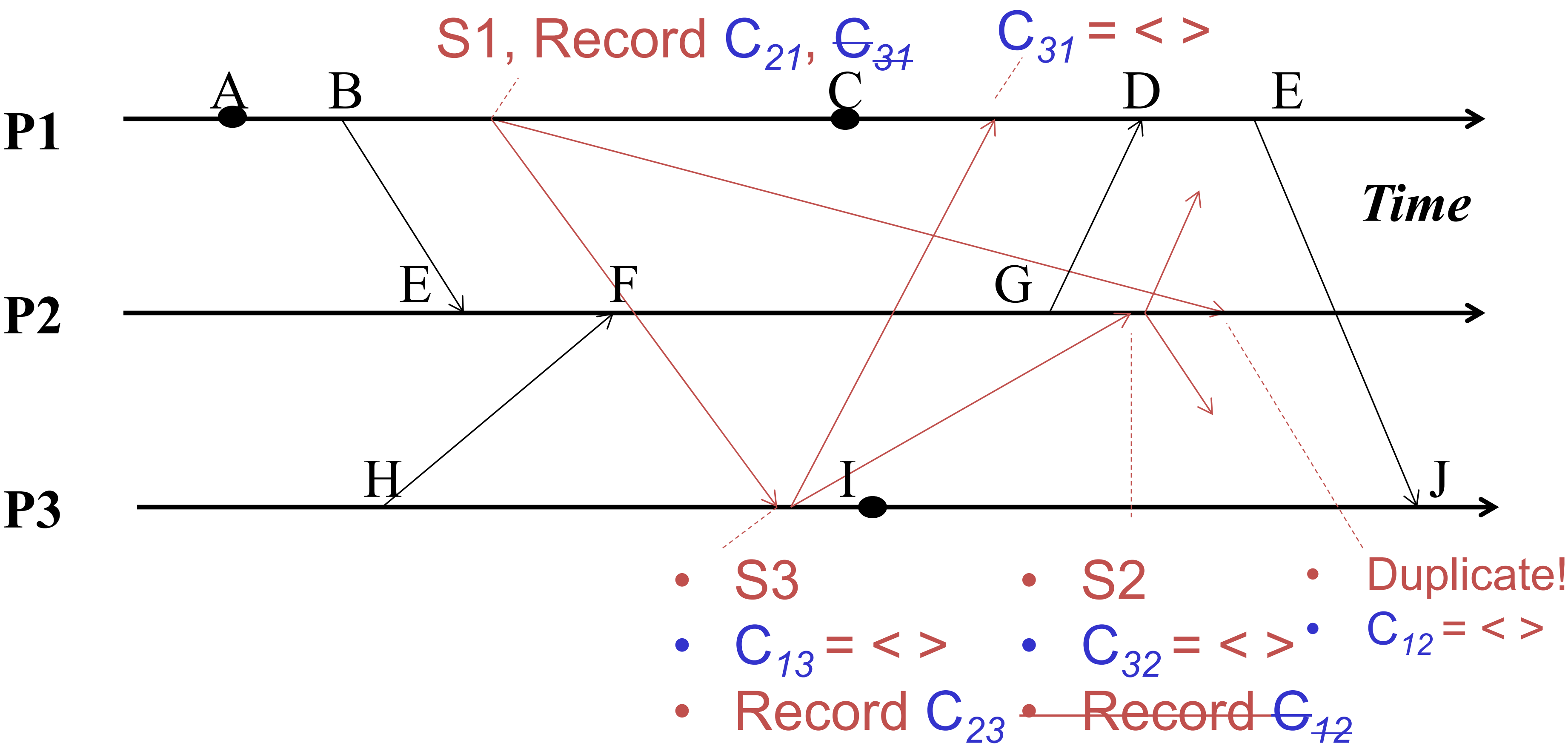

Duplicate Marker!
State of channel C_{31} = < >
S1, Record C_{21}, ~~C_{31}~~
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
S3
C_{13} = < >
Record C_{23}

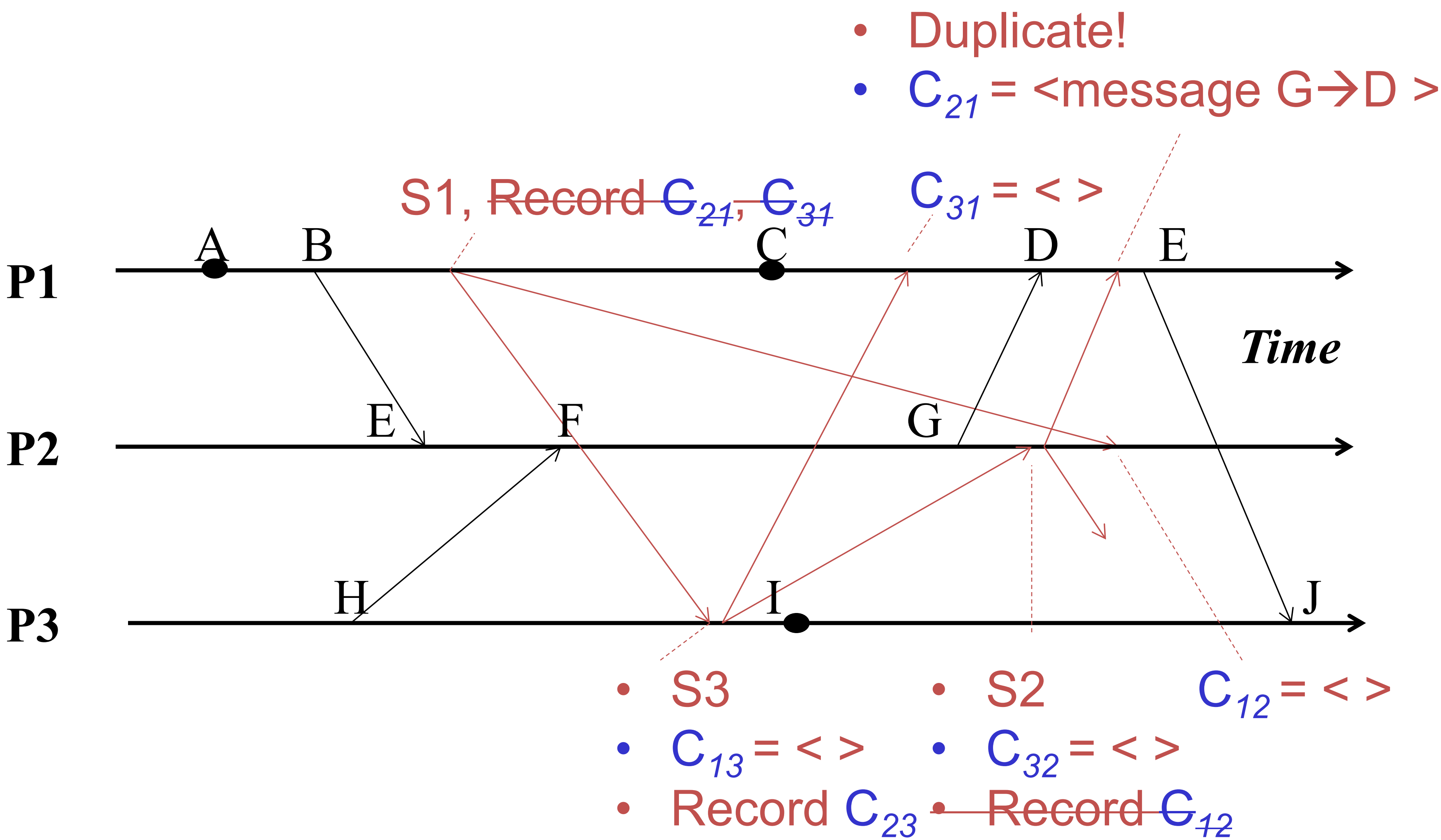

S1, Record C21, ~~C31~~
C31 = < >
P1
A
B
C
D
E
Time
P2
E
F
G
P3
H
I
J
S3
C13 = < >
Record C23
First Marker!
Record own state as S2
Mark C32 state as empty
Turn on recording on C12
Send out Markers



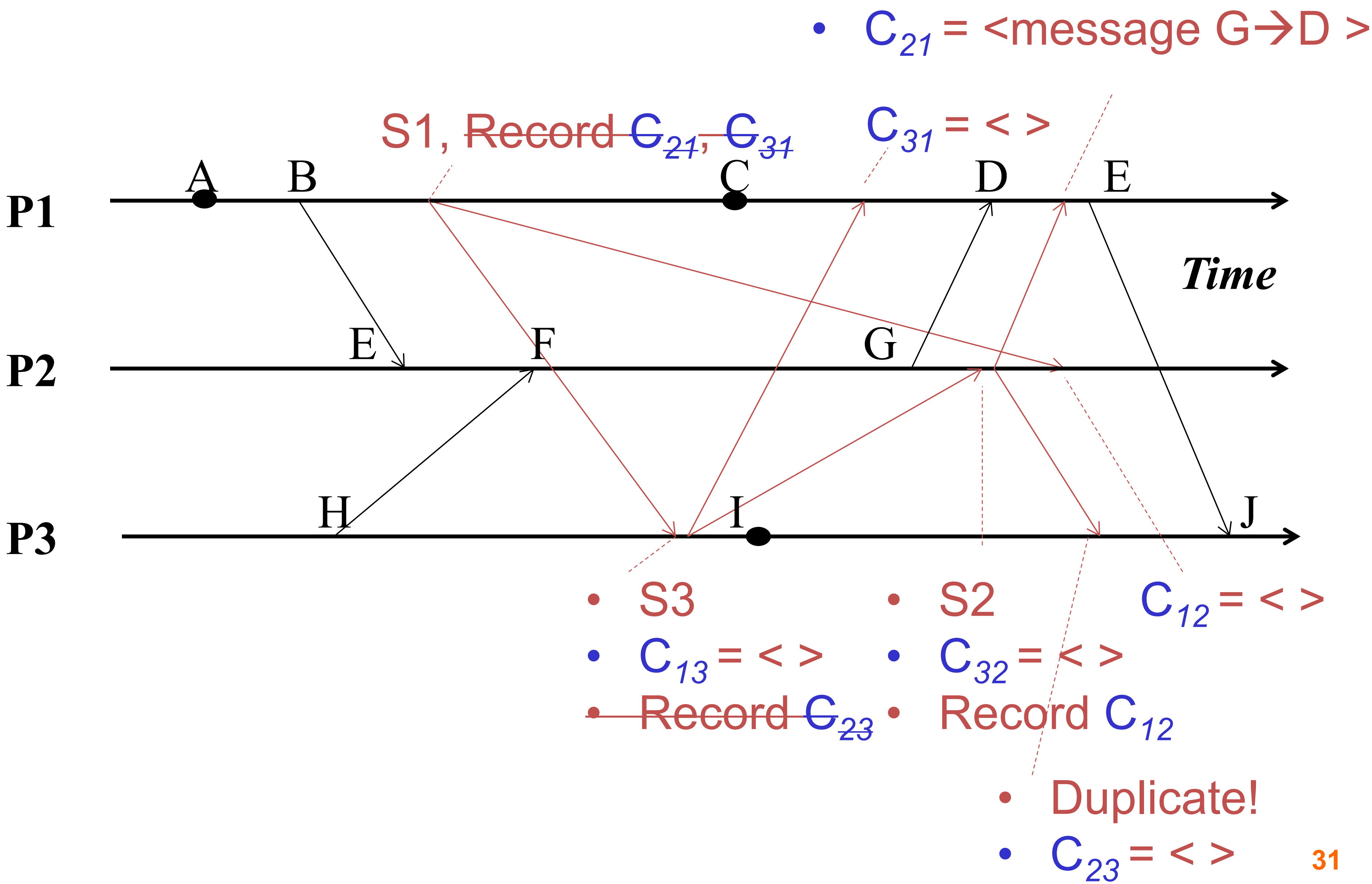
- S3
- C_{13} = < >
- Record C_{23}

- S2
- C_{32} = < >
- Record C_{12}


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

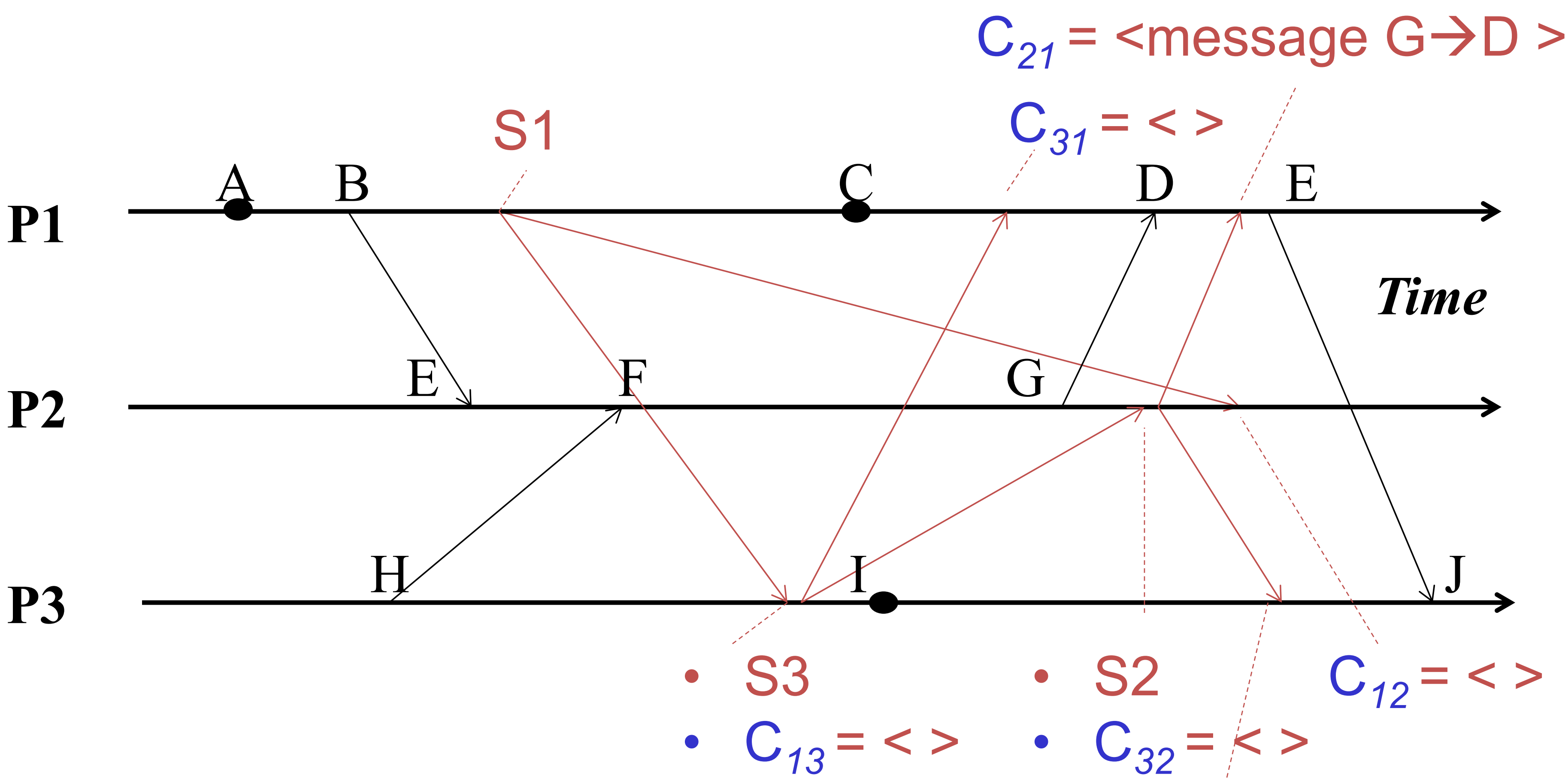


• Duplicate!
• C21 = <message G→D >
C31 = < >
S1, ~~Record C21, C31~~
A
B
C
D
E
P1
Time
E
F
G
P2
H
I
J
P3
• S3
• C13 = < >
• Record C23
• S2
• C32 = < >
• ~~Record C12~~
C12 = < >



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

Algorithm has terminated

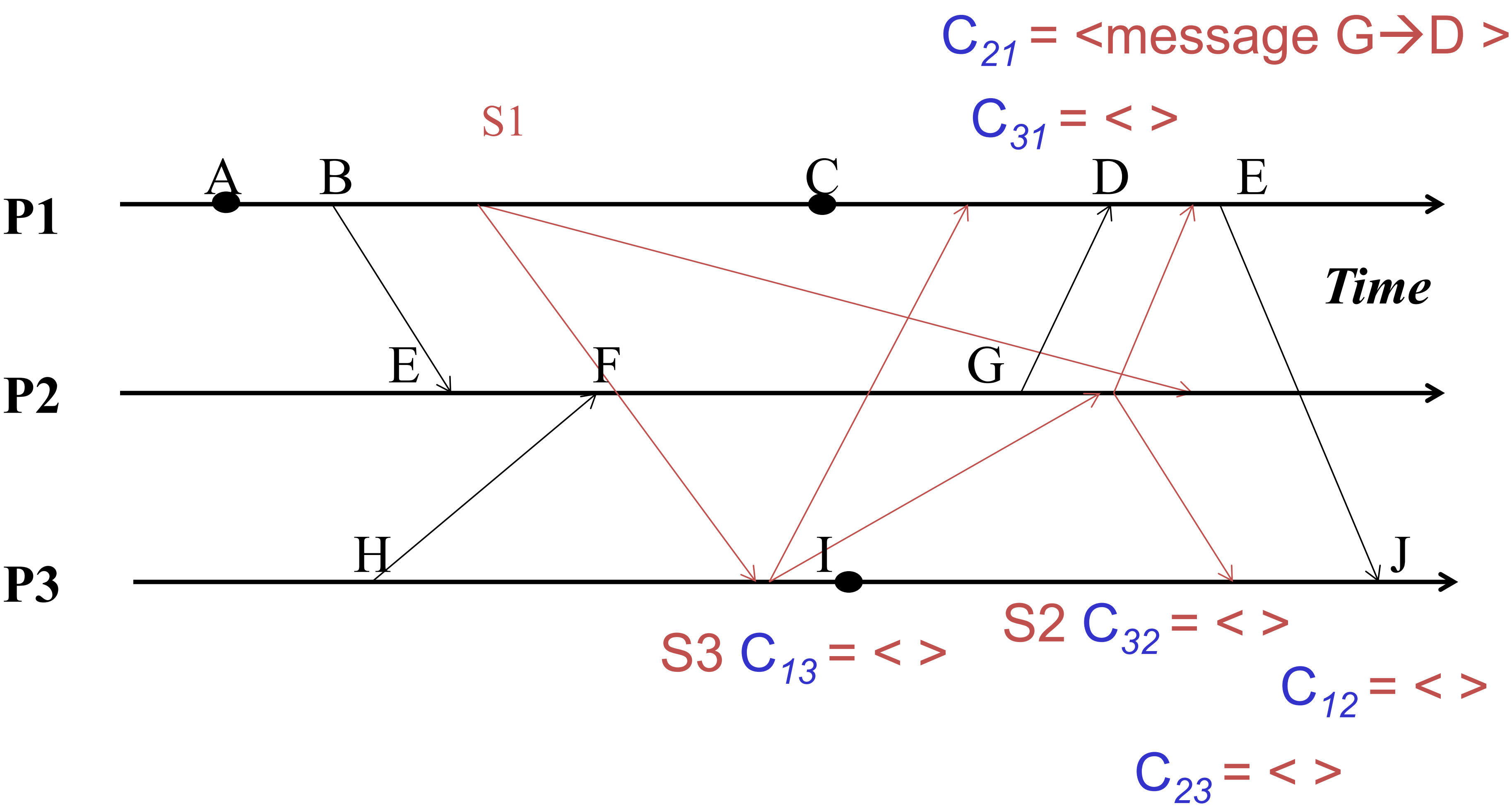


- S3
- C_{13} = < >

- S2
- C_{32} = < >

- C_{23} = < >

Collect the global snapshot pieces



Next

- **Global Snapshot calculated by Chandy-Lamport algorithm is <u>causally correct</u>**
  - What?

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

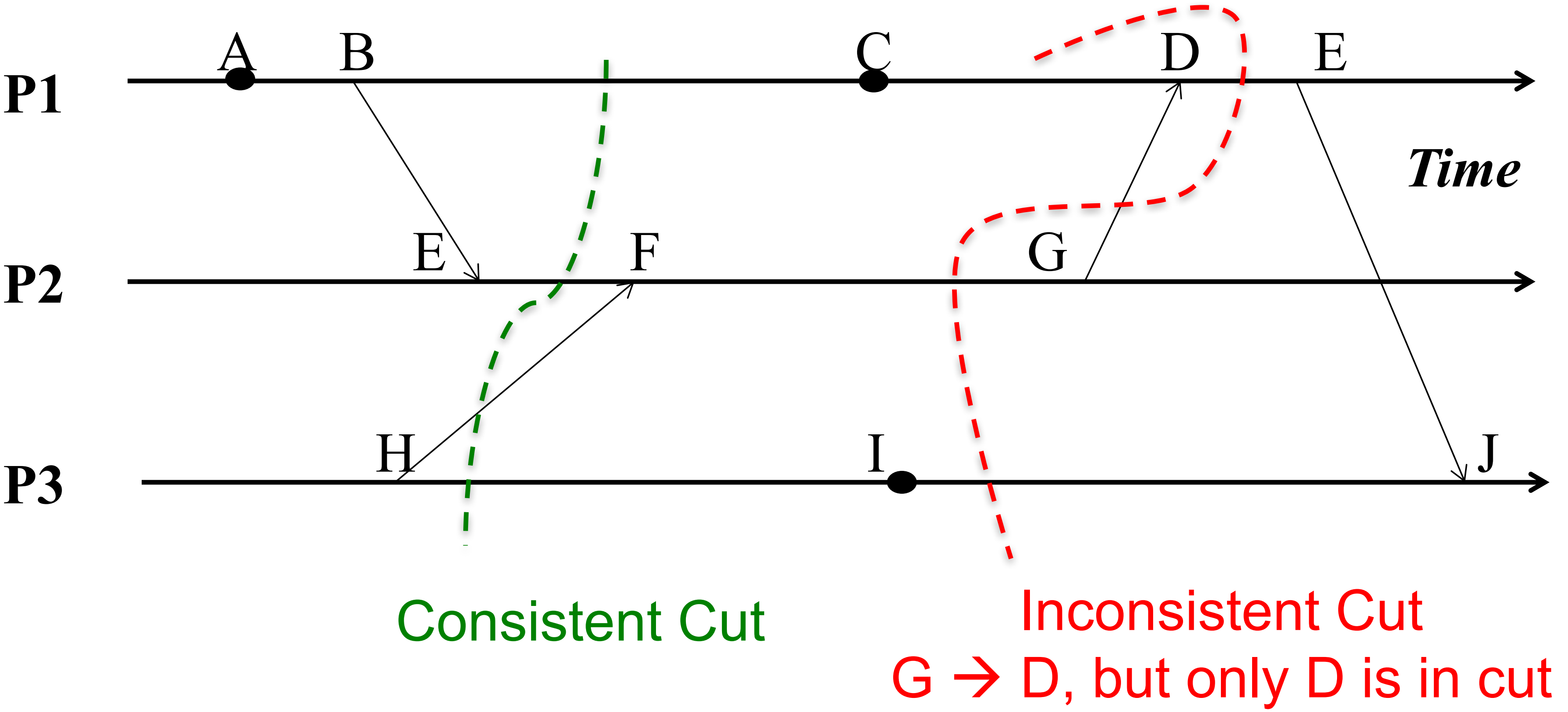
- Cut C is a consistent cut if and only if:

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

  – Such that event e is in the cut C, and if f $\rightarrow$ e (f happens-before e)

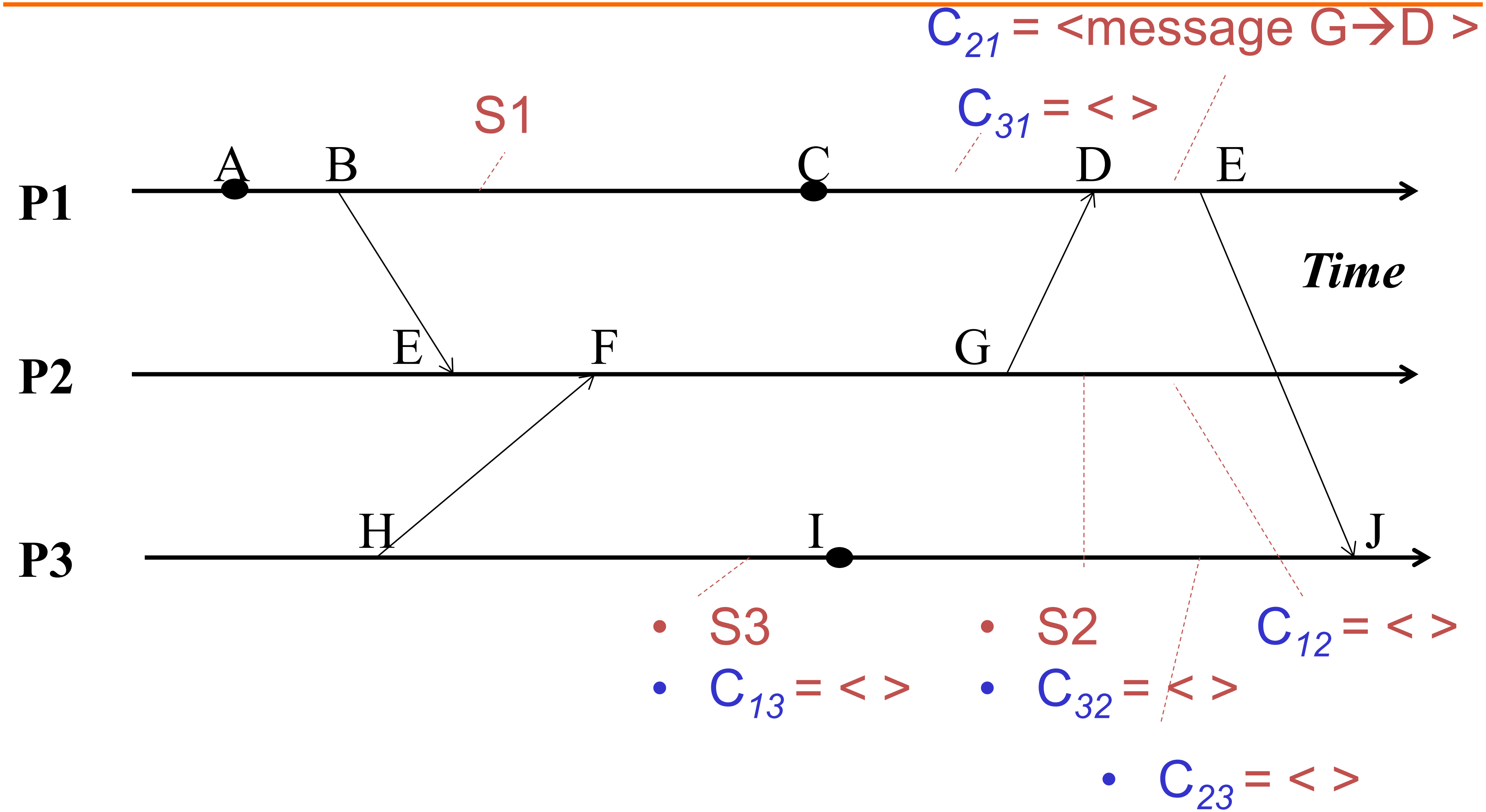
    - Then: Event f is also in the cut C

Example

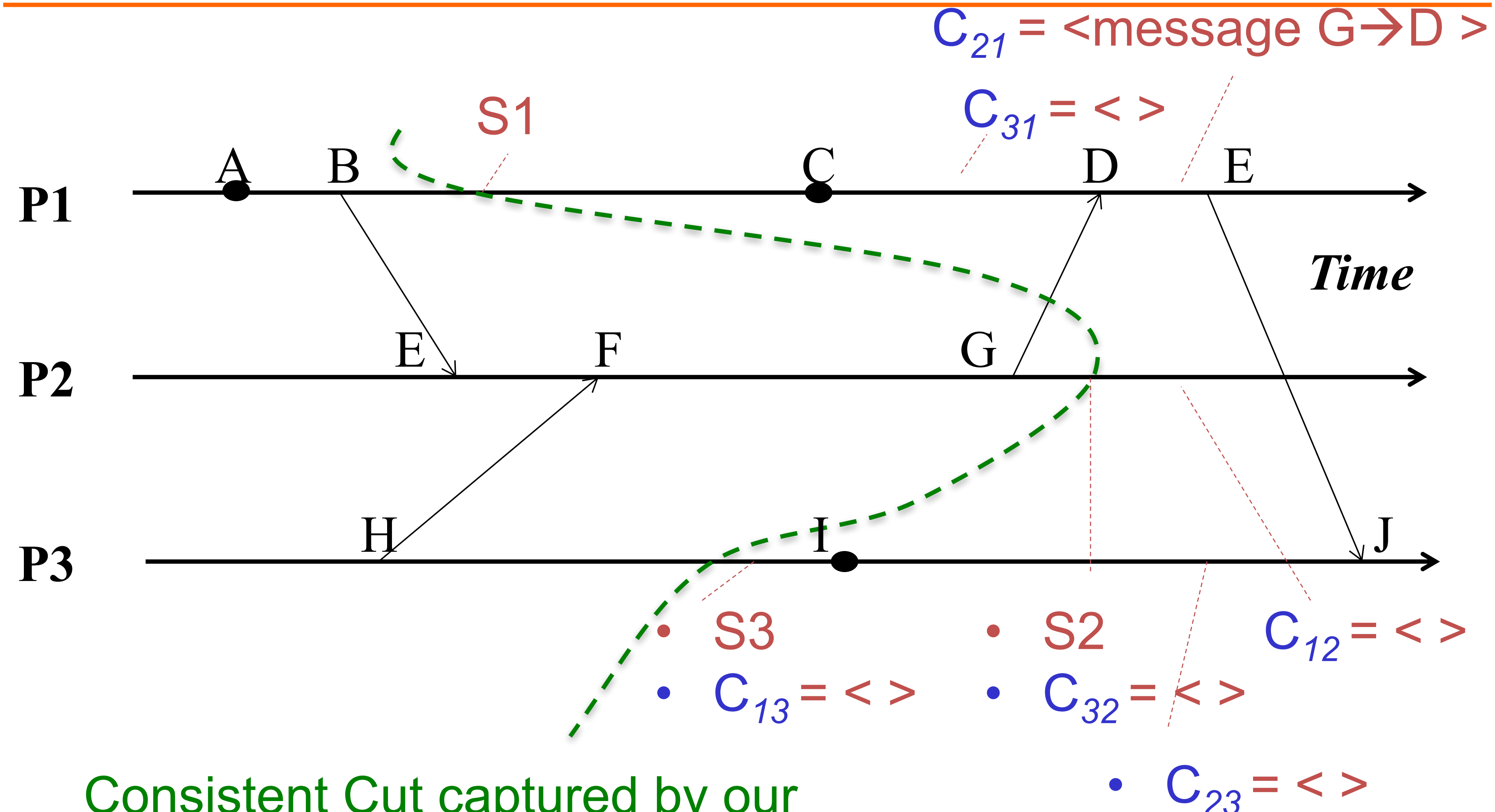


P1
P2
P3
A
B
C
D
E
E
F
G
H
I
J
Time
Consistent Cut
Inconsistent Cut
G → D, but only D is in cut

Our Global Snapshot Example …



… is causally correct



Consistent Cut captured by our Global Snapshot Example

In fact…

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

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 $e_j \rightarrow$ <P*j* records its state>, then

- it must be true that $e_i \rightarrow$ <P*i* records its state>

Chandy-Lamport Global Snapshot algorithm creates a consistent cut

- **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 $e_i \rightarrow e_j$
  - 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)

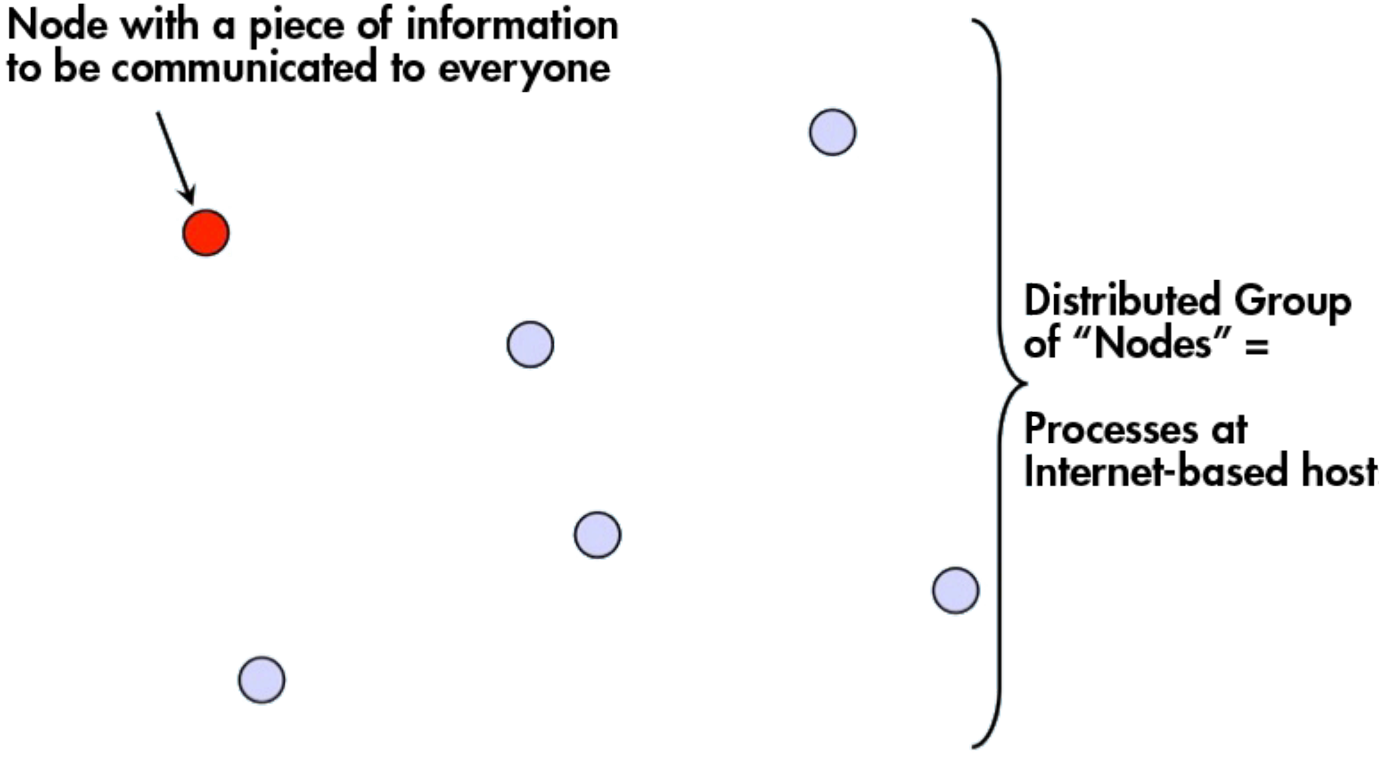
Distributed snapshot algorithm summary

- Chandy & Lamport, 1985
  - algorithm to select a consistent cut
  - any process may initiate a snapshot at any time
  - processes can continue normal execution
    - send and receive messages
  - assumes:
    - no failures of processes & channels
    - strong connectivity
      - at least one path between each process pair
    - unidirectional, FIFO channels
    - reliable delivery of messages

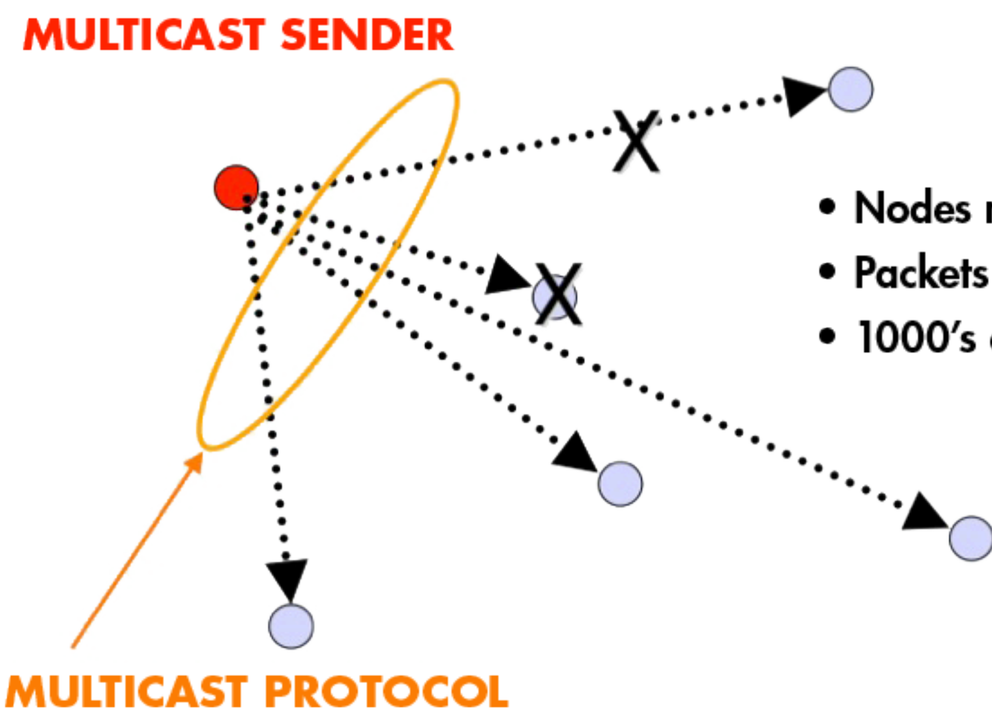
Today

1. Global snapshot of a distributed system
2. Chandy-Lamport's algorithm
3. **Gossip**

Multicast problem



Fault-tolerance and Scalability

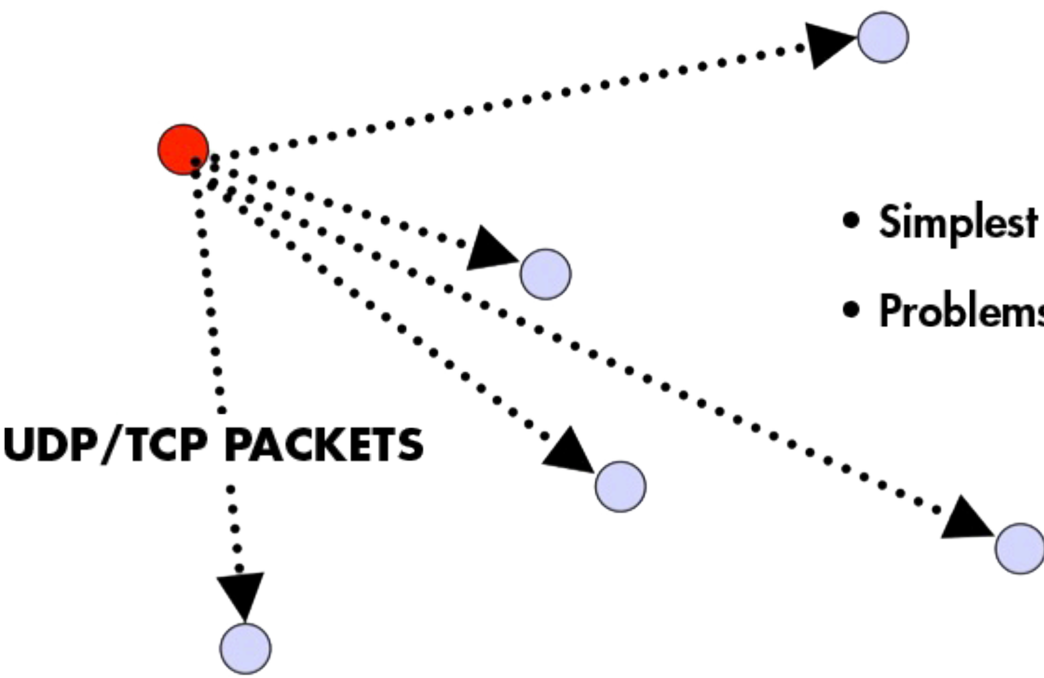


- Nodes may crash
- Packets may be dropped
- 1000's of nodes

Needs:

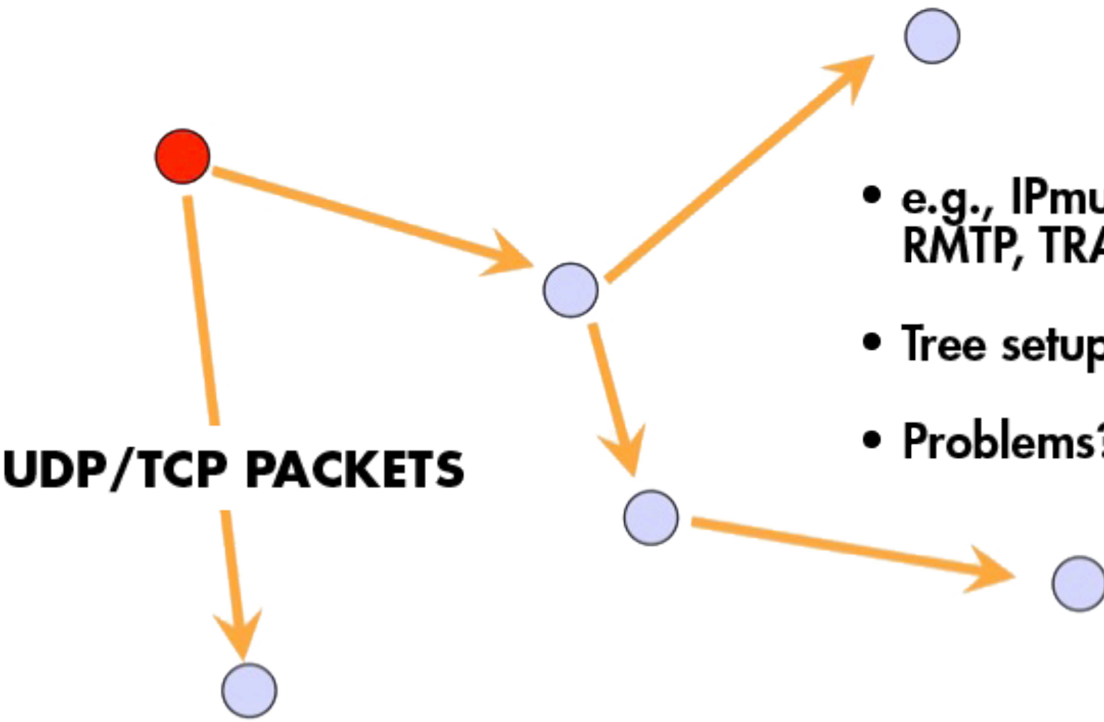
1. Reliability (Atomicity)
   - 100% receipt
2. Speed

Centralized



- Simplest implementation
- Problems?

Tree-Based

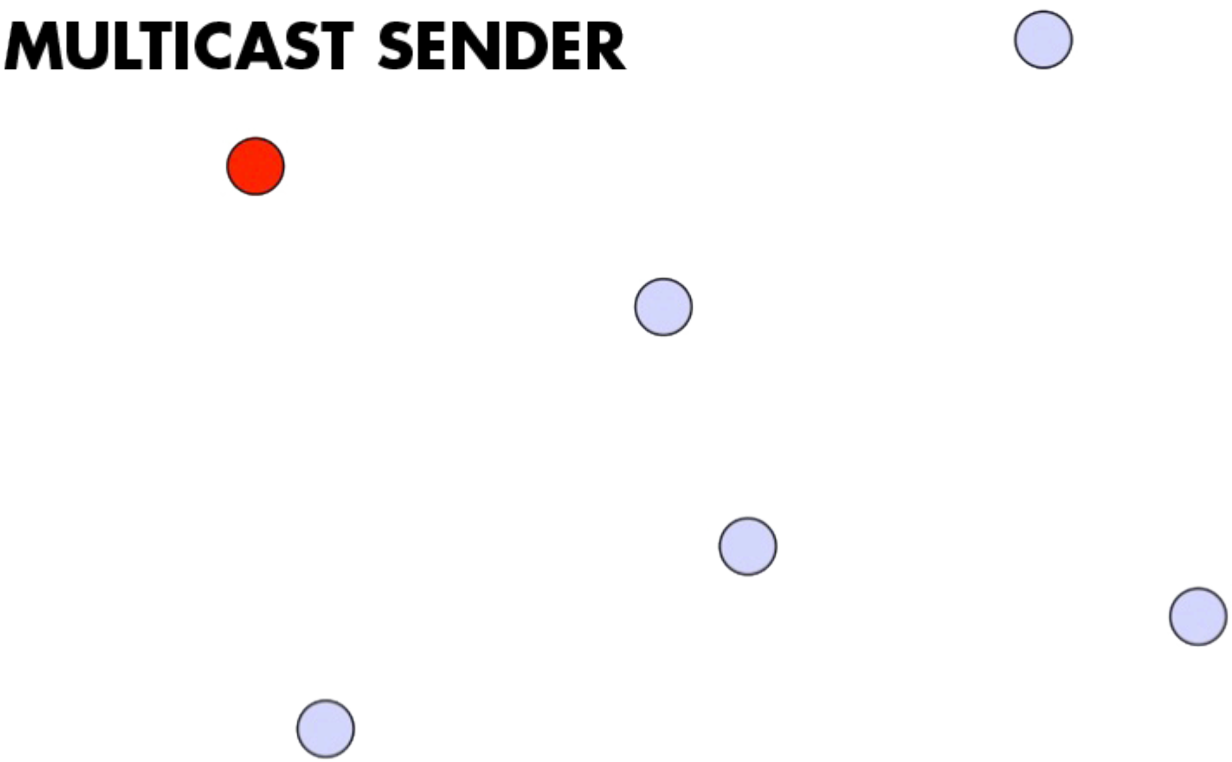


- e.g., IPmulticast, SRM RMTP, TRAM, TMTP
- Tree setup and maintenance
- Problems?

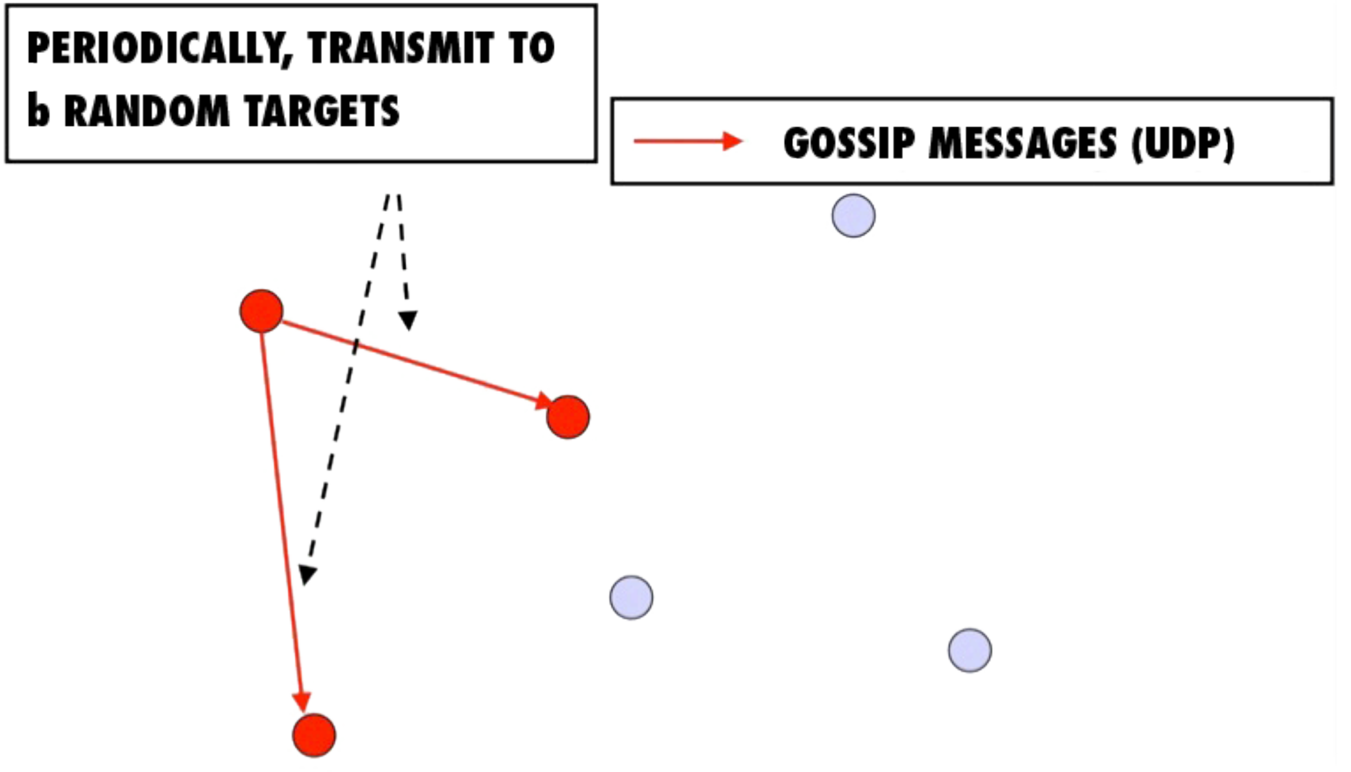
Tree-based Multicast Protocols

- Build a spanning tree among the processes of the multicast group
- Use spanning tree to disseminate multicasts
- Use either acknowledgments (ACKs) or negative acknowledgements (NAKs) to repair multicasts not received
- SRM (Scalable Reliable Multicast)
  - Uses NAKs
  - But adds random delays, and uses exponential backoff to avoid NAK storms
- RMTP (Reliable Multicast Transport Protocol)
  - Uses ACKs
  - But ACKs only sent to designated receivers, which then re-transmit missing multicasts
- These protocols still cause an O(N) ACK/NAK overhead [Birman99]

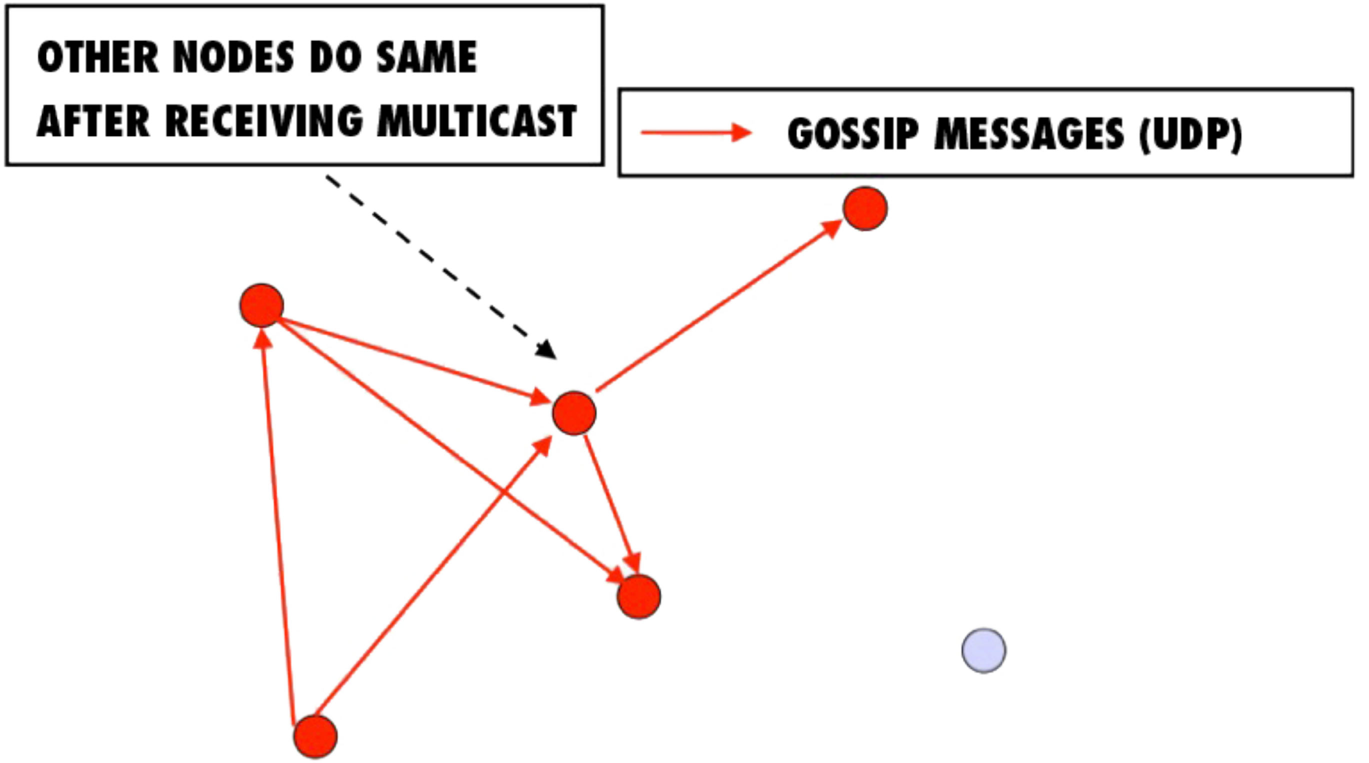
A Third Approach



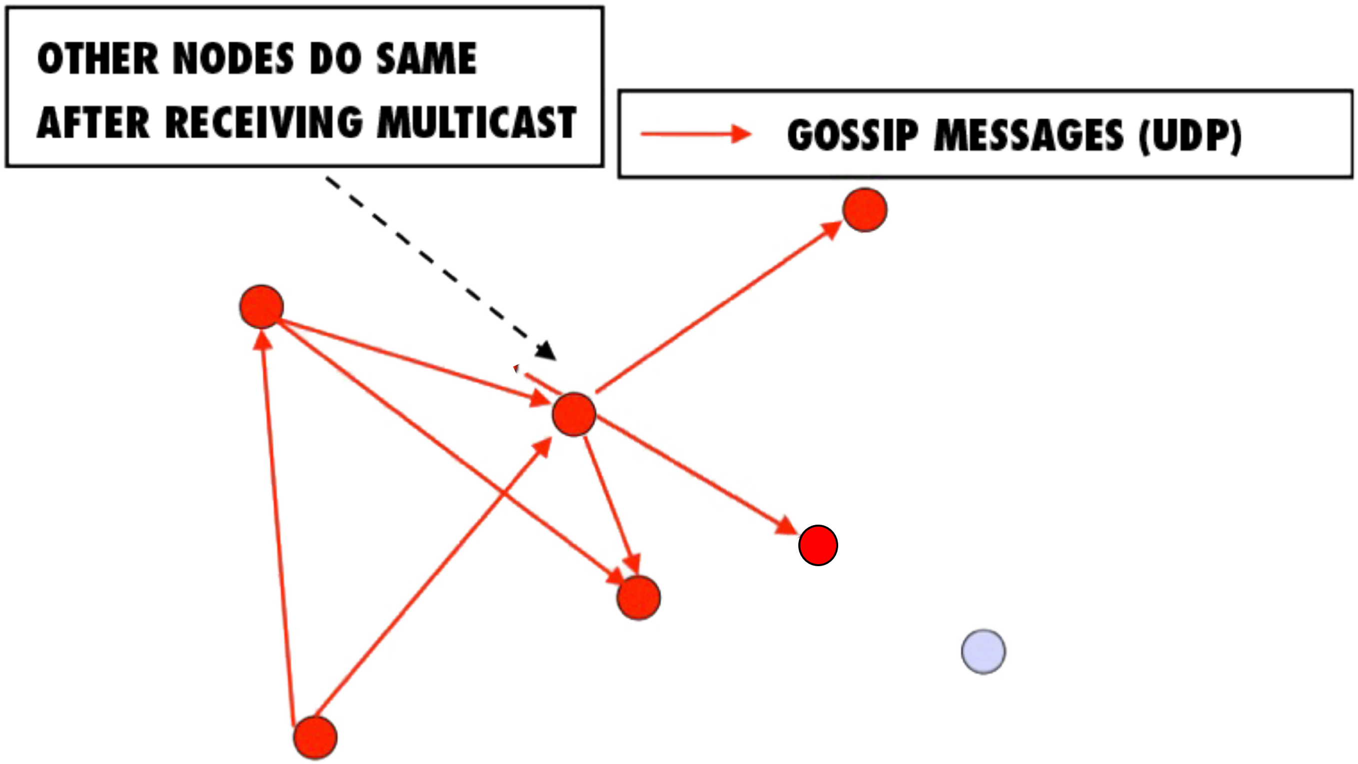
A Third Approach



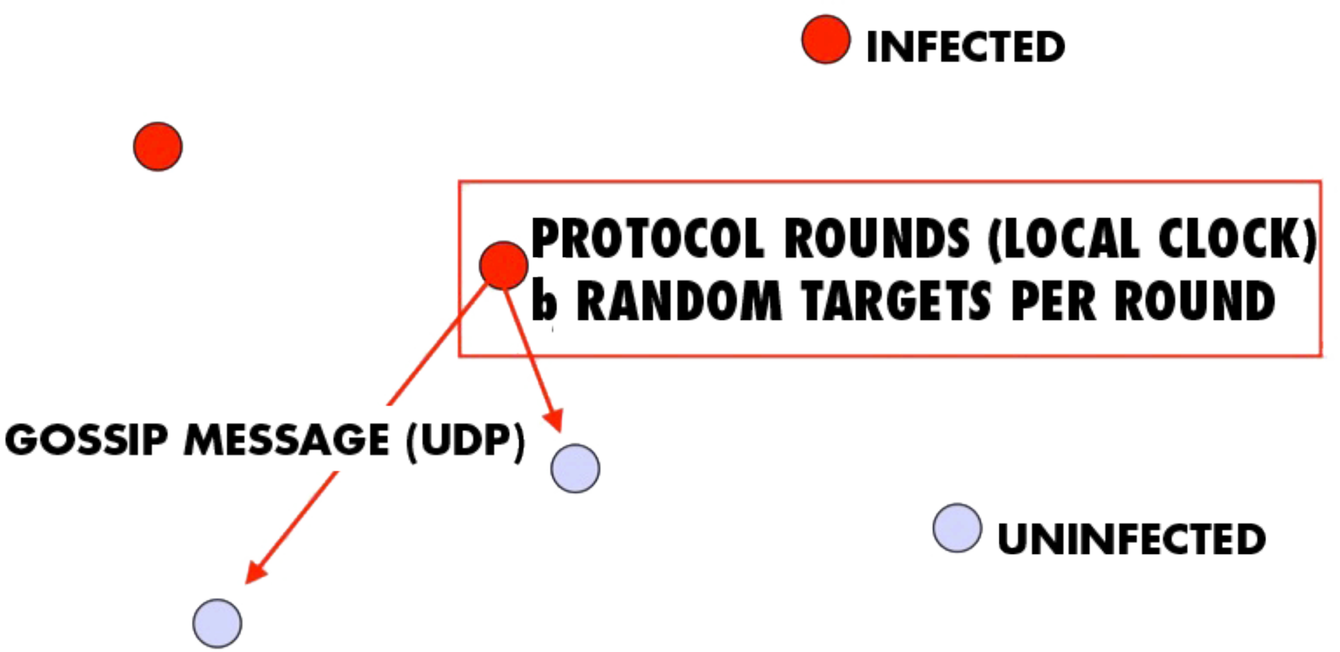
A Third Approach



A Third Approach



“Epidemic” Multicast (or “Gossip”)



Push vs. Pull

- So that was “Push” gossip
  - Once you have a multicast message, you start gossiping about it
  - Multiple messages? Gossip a random subset of them, or recently-received ones, or higher priority ones
- There’s also “Pull” gossip
  - Periodically poll a few randomly selected processes for new multicast messages that you haven’t received
  - Get those messages
- Hybrid variant: Push-Pull
  - As the name suggests

Properties

Claim that the simple Push protocol

- Is lightweight in large groups
- Spreads a multicast quickly
- Is highly fault-tolerant

Analysis

From old mathematical branch of *Epidemiology* [Bailey75]

- Population of ($n+1$) individuals mixing homogeneously
- Contact rate between any individual pair is β
- At any time, each individual is either uninfected (numbering x) or infected (numbering y)
- Then, $x_0 = n, y_0 = 1$

  and at all times $x + y = n + 1$
- Infected–uninfected contact turns latter infected, and it stays infected

Analysis (contd.)

- Continuous time process
- Then

$$\frac{dx}{dt} = -\beta xy$$

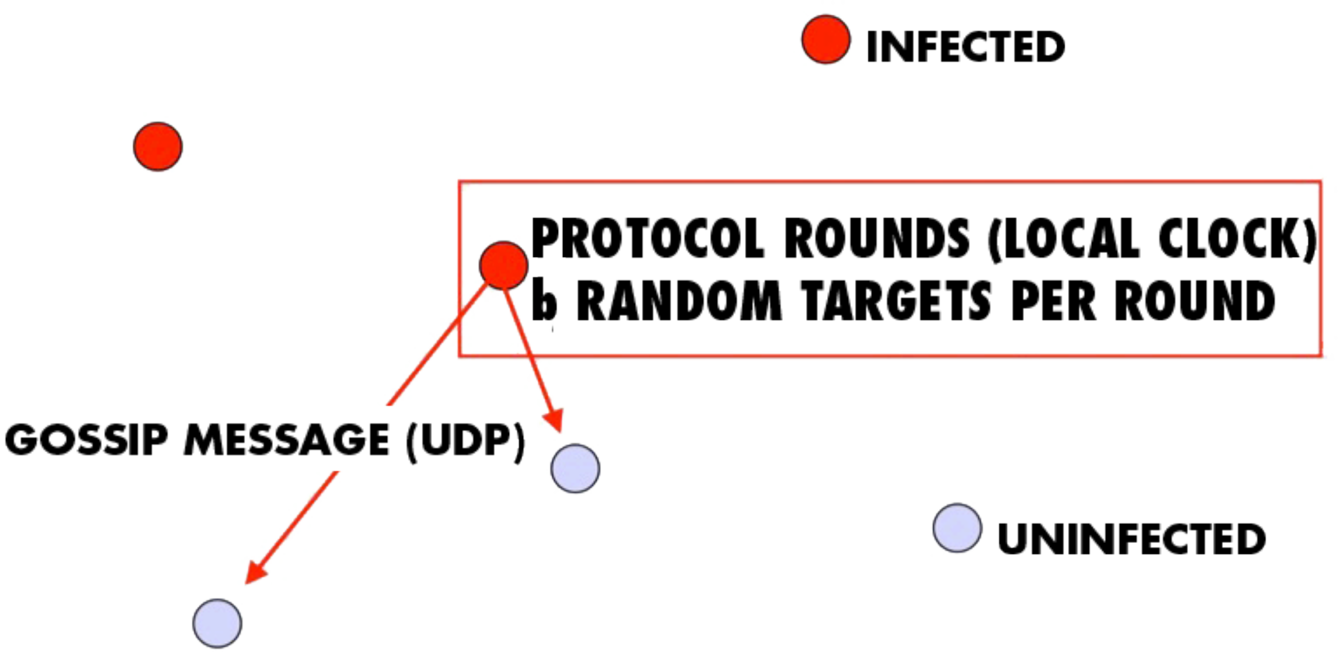
(why?)

with solution:

$$x = \frac{n(n+1)}{n + e^{\beta(n+1)t}}, y = \frac{(n+1)}{1 + ne^{-\beta(n+1)t}}$$

(can you derive it?)

Epidemic Multicast



Epidemic Multicast Analysis

$$\beta = \frac{b}{n}$$ (why?)

Substituting, at time *t=clog(n)*, the number of infected is

$$y \approx (n+1) - \frac{1}{n^{cb-2}}$$

(correct? can you derive it?)

Analysis (contd.)

- Set *c, b* to be small numbers independent of *n*
- Within *clog(n)* rounds, [low latency]
  - all but $\frac{1}{n^{cb-2}}$ number of nodes receive the multicast

    [reliability]
  - each node has transmitted no more than *cblog(n)* gossip messages [lightweight]

Why is log(N) low?

- log(N) is not constant in theory
- But pragmatically, it is a very slowly growing number
- Base 2
  - log(1000) ~ 10
  - log(1M) ~ 20
  - log (1B) ~ 30
  - log(all IPv4 address) = 32

Fault-tolerance

- Packet loss
  - 50% packet loss: analyze with *b* replaced with *b*/2
  - To achieve same reliability as 0% packet loss, takes twice as many rounds
- Node failure
  - 50% of nodes fail: analyze with *n* replaced with *n*/2 and *b* replaced with *b*/2
  - Same as above

Fault-tolerance

- With failures, is it possible that the epidemic might die out quickly?
- Possible, but improbable:
  - Once a few nodes are infected, with high probability, the epidemic will not die out
  - So the analysis we saw in the previous slides is actually behavior *with high probability*

  [Galey and Dani 98]
- Think: why do rumors spread so fast? why do infectious diseases cascade quickly into epidemics? why does a virus or worm spread rapidly?

Pull Gossip: Analysis

- In all forms of gossip, it takes O(log(N)) rounds before about N/2 processes get the gossip
  - Why? Because that's the fastest you can spread a message – a spanning tree with fanout (degree) of constant degree has O(log(N)) total nodes
- Thereafter, pull gossip is faster than push gossip
- After the *i*th, round let p_i be the fraction of non-infected processes. Let each round have *k* pulls. Then

$$p_{i+1} = \left(p_i\right)^{k+1}$$

- This is super-exponential
- Second half of pull gossip finishes in time O(log(log(N))

Summary

- Multicast is an important problem
- Tree-based multicast protocols
- When concerned about scale and fault-tolerance, gossip is an attractive solution
- Also known as epidemics
- Fast, reliable, fault-tolerant, scalable, topology-aware

Next Topic:

Primary-backup replication
(pre-reading: VM replication)