CS 60002: Distributed Systems

T4: Global States

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

Study Material

We'll broadly talk about this paper and some associated results:

- Chandy, K. Mani, and Leslie Lamport. "Distributed snapshots: Determining global states of distributed systems." *ACM Transactions on Computer Systems (TOCS)* 3.1 (1985): 63-75.

- Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 7 (July 1978), 558–565.

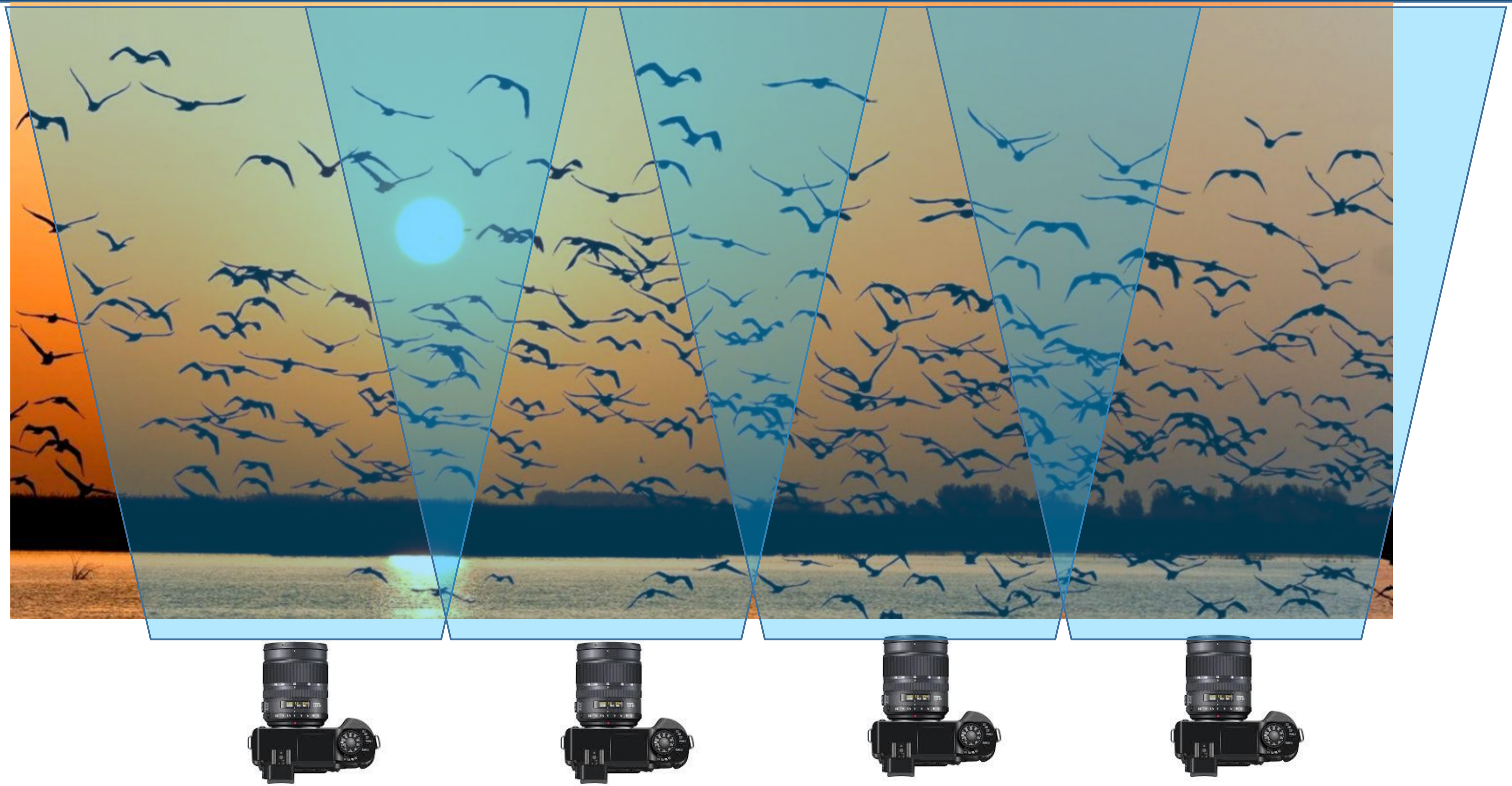
Global Snapshot

Image source: https://www.wallpaperkiss.com/wken/wohxii/

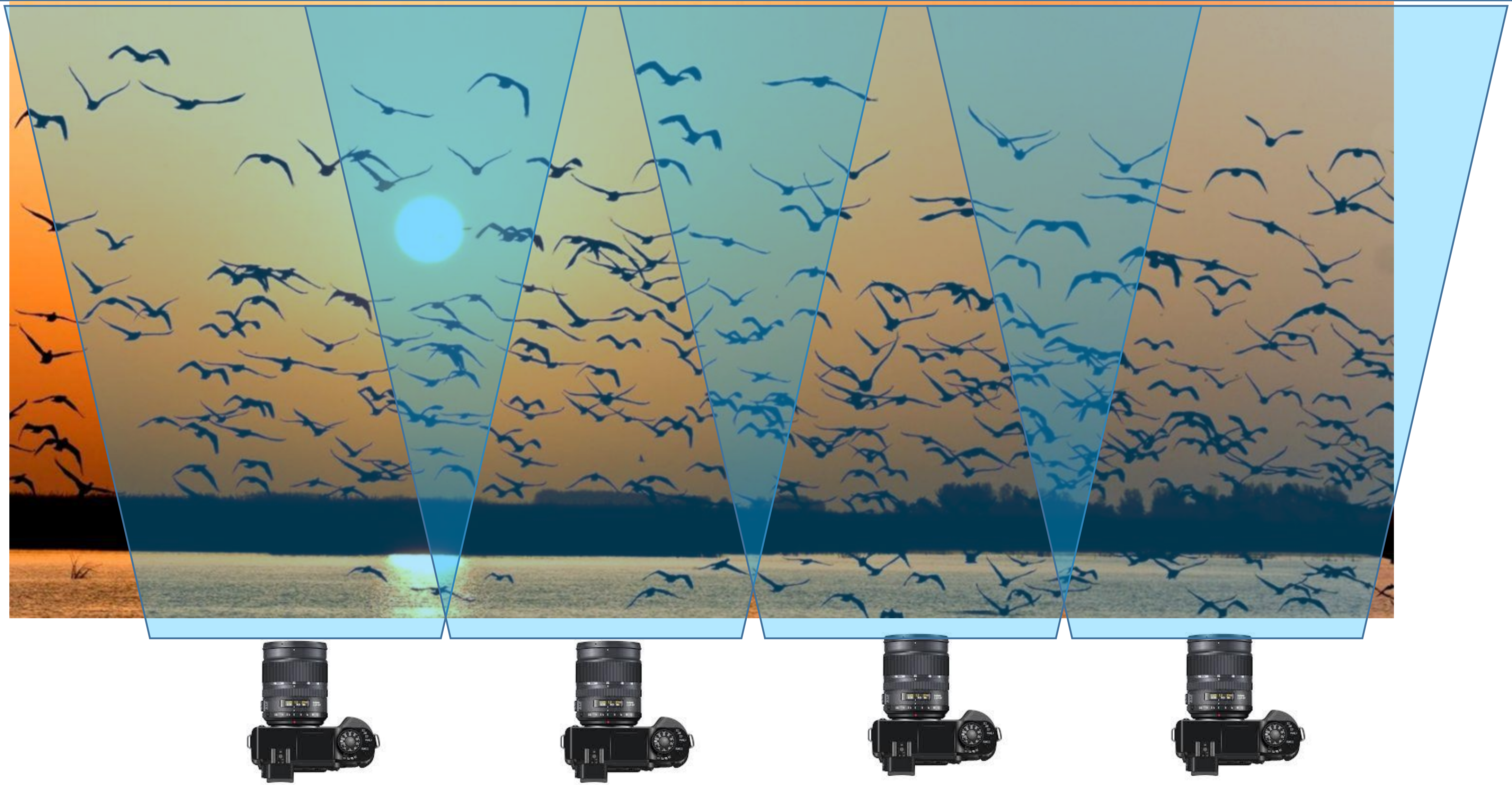
Global Snapshot

Image source: https://www.wallpaperkiss.com/wken/wohxii/

Global Snapshot

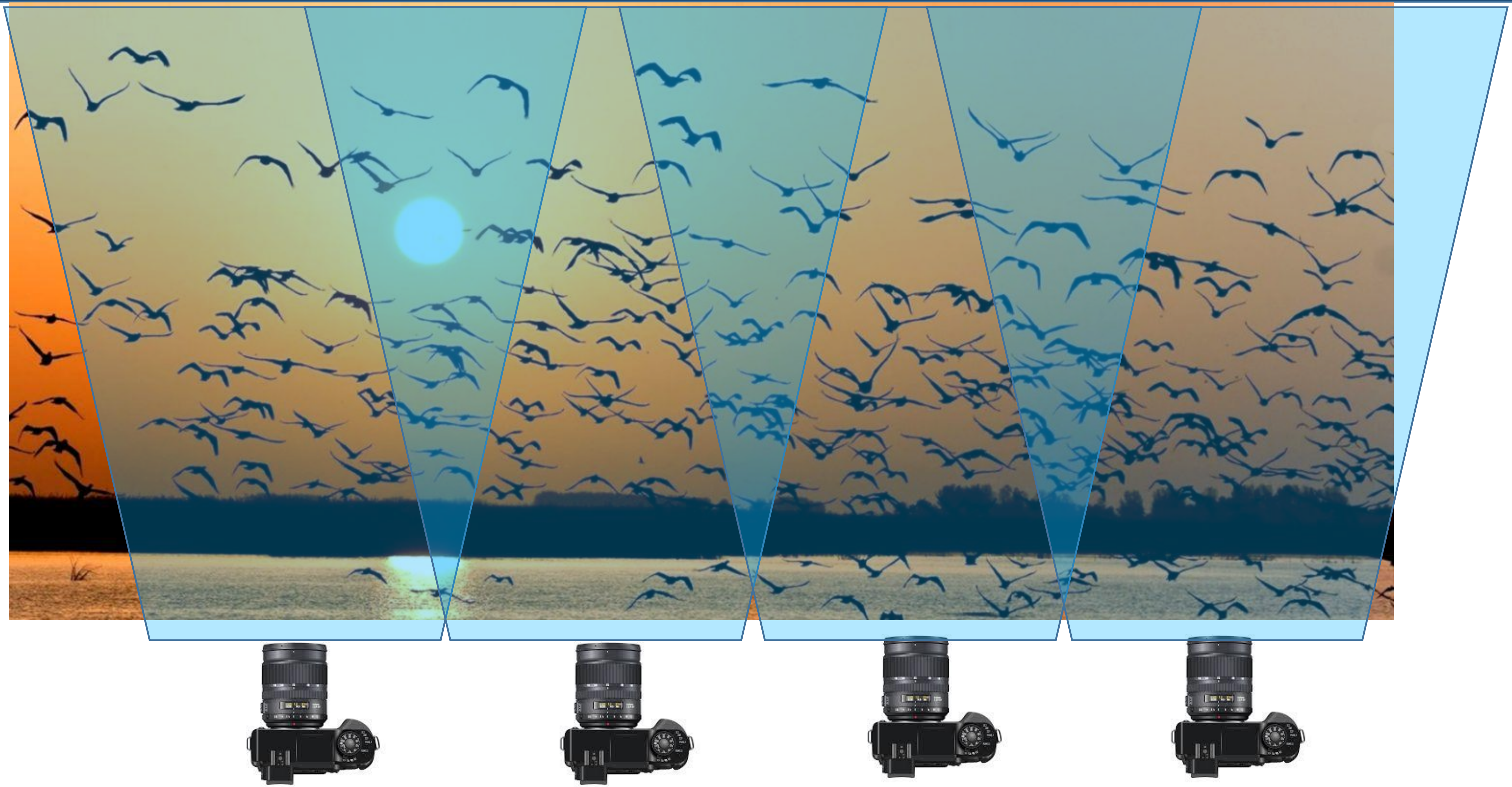


Global Snapshot



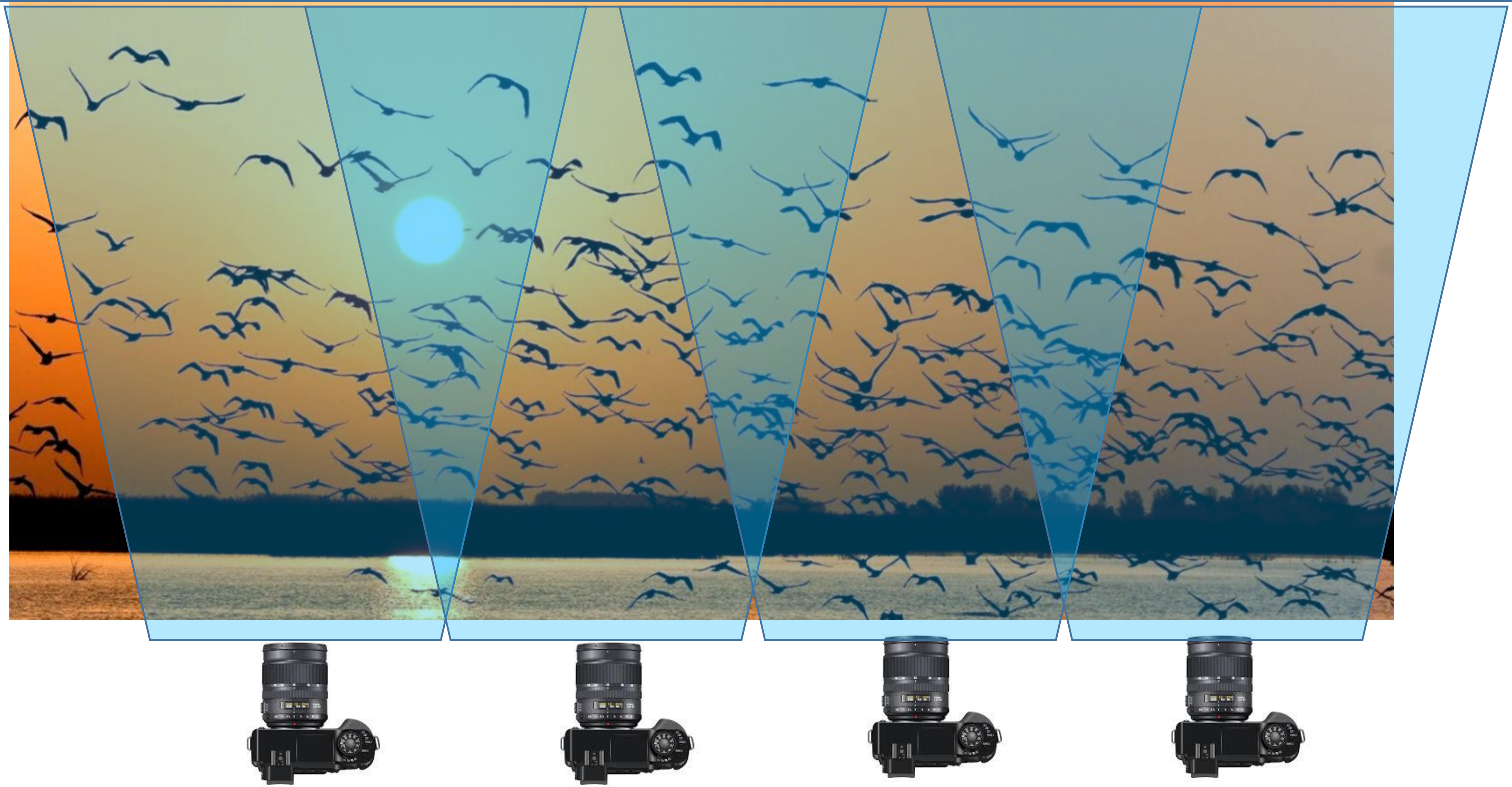
Photographers should not disturb the bird

Global Snapshot



What will be a "meaningful" photo here?

Global Snapshot



How do we take a "meaningful" photo?

Global States

- Collection of local states for all the (correct) agents
- <u>We do not have a global clock</u> – we cannot take instantaneous snapshot for all the agents
- <u>We do not have a central agent</u> that can collect and combine the local snapshots from all other agents
  - Every agent must do it independently
  - However, all (correct) agent should have the same view of the system (the snapshot should be the **common knowledge**)
- In a typical distributed system
  - One agent may collect the local states from others through message passing
  - Ideal world (Synchronous, reliable, no failure): Get the states of others at <u>nearby</u> time instances and combine them

Global States

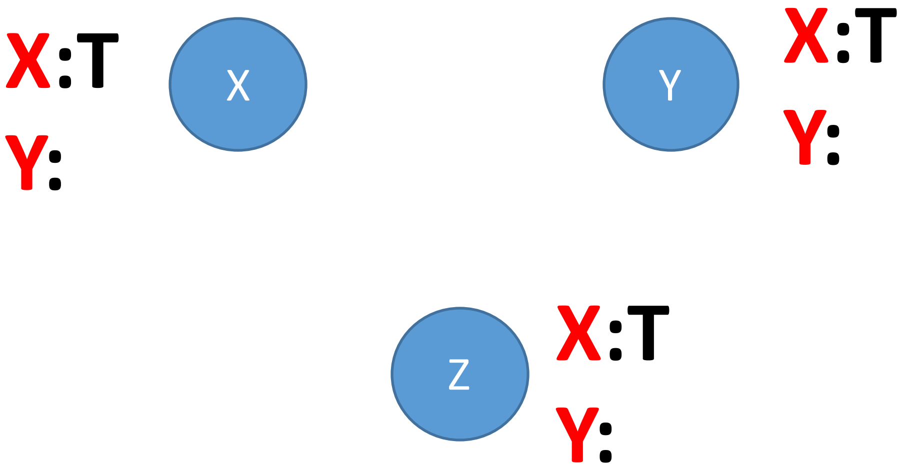
- Collection of local states for all the (correct) agents
- We do not have a global clock – we cannot take instantaneous snapshot for all the agents
- We do not have a central agent that can collect and combine the local snapshots from all other agents
  - Every agent must do it independently
  - However, all (correct) agent should have the same view of the system (the snapshot should be the **common knowledge**)
- In a t
  - Or
  - Id me ins

How can we collect a consistent, global snapshot of the system in a real-world?

Consistent Global State

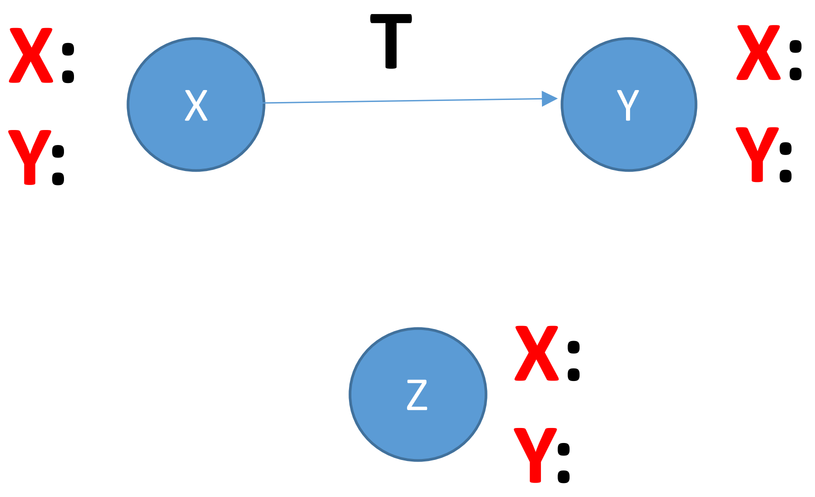
- Every (correct) agent should have the same view of the global state
  - Henceforth, we'll use "agent" to mean a correct agent
- Say, a token has been transferred from agent X to agent Y
  - **Consistent Global State:** The token is removed from X, and is included in Y just after the message is received at Y

Before a token transfer



Consistent Global State

- Every (correct) agent should have the same view of the global state
  - Henceforth, we'll use "agent" to mean a <u>correct agent</u>
- Say, a token has been transferred from agent X to agent Y
  - **Consistent Global State:** The token is removed from X, and is included in Y just after the message is received at Y



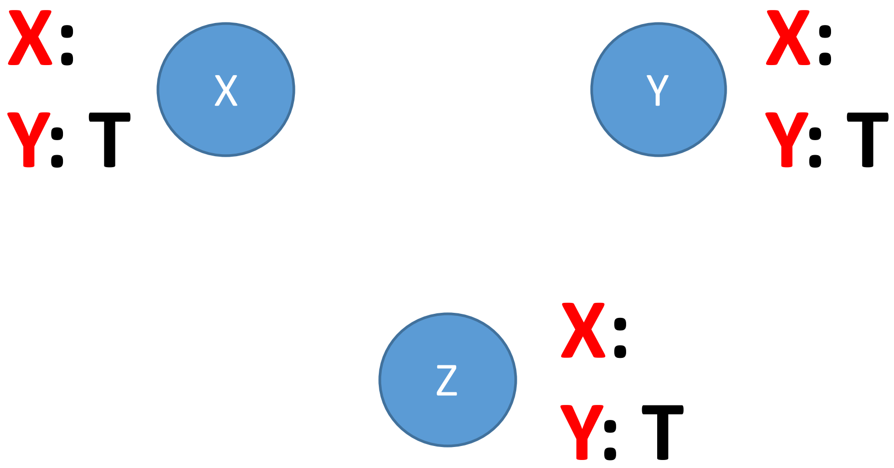
<u>During a token transfer</u>

Consistent Global State

- Every (correct) agent should have the same view of the global state
  - Henceforth, we'll use "agent" to mean a correct agent

- Say, a token has been transferred from agent X to agent Y
  - **Consistent Global State:** The token is removed from X, and is included in Y just after the message is received at Y

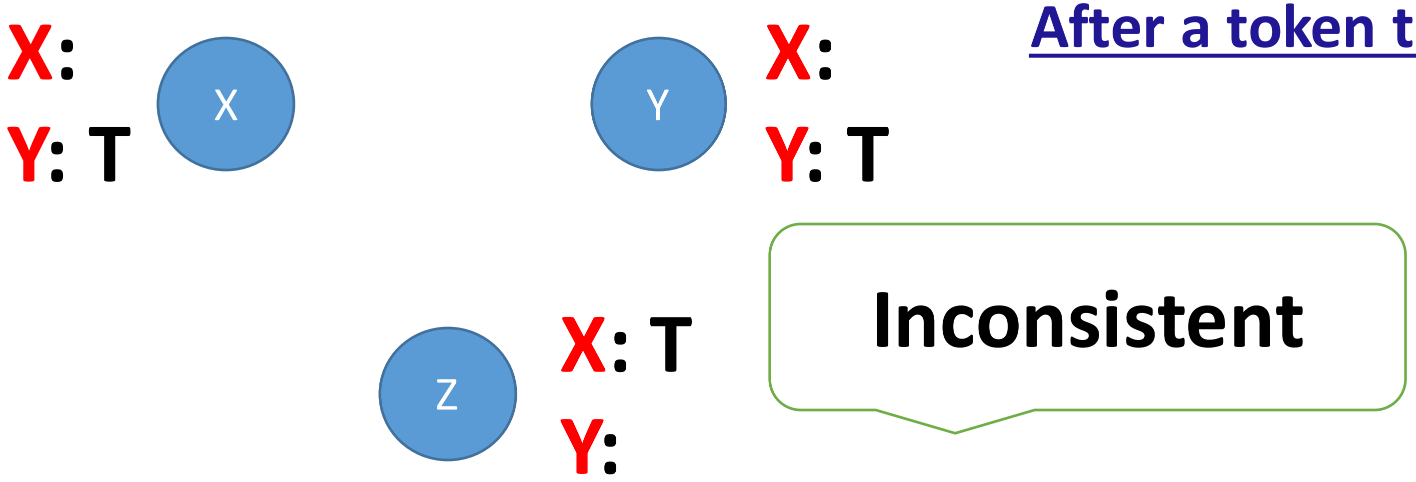
After a token transfer



Consistent Global State

- Every (correct) agent should have the same view of the global state
  - Henceforth, we'll use "agent" to mean a correct agent
- Say, a token has been transferred from agent X to agent Y
  - **Consistent Global State:** The token is removed from X, and is included in Y just after the message is received at Y

After a token transfer



Consistent Global State

- Every (correct) agent should have the same view of the global state
  - Henceforth, we'll use "agent" to mean a correct agent

- Say, a token has been transferred from agent X to agent Y
  - **Consistent Global State:** The token is removed from X, and is included in Y just after the message is received at Y
  - **Inconsistent:** The token is shown to both agents X and Y, or the token is shown to X

- Global states can also be incomplete
  - The states of some agents are missing – message loss? Unbounded delay?

- States might have changed since the last checkpoint is taken

How Do We Compute Consistent Global States

- Say, every agent keeps on sending messages about their local states; eventually every agent will receive the state messages
  - Ensures that the global state is complete (have information from all the agents)
  - Ensures that the global state is current (not stale)
  - **Does not ensure that the global state is consistent – why?**

How Do We Compute Consistent Global States

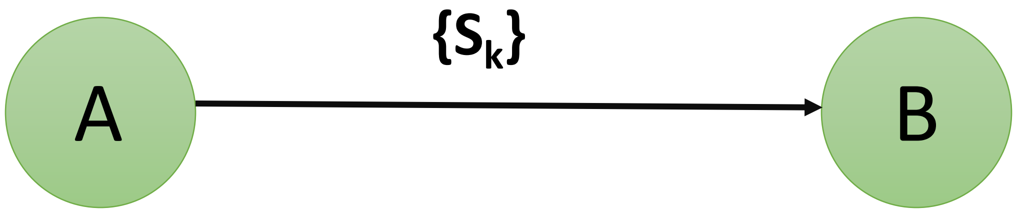
- Say, every agent keeps on sending messages about their local states; eventually every agent will receive the state messages
  - Ensures that the global state is complete (have information from all the agents)
  - Ensures that the global state is current (not stale)
  - **Does not ensure that the global state is consistent – why?**

- Does not ensure that all agents have the same view
  - **Does not ensure the states as a common knowledge –** Agent A does not know whether Agent B knows Agent A's state
  - **Even an ACK does not ensure the common knowledge – why?**

Achieving Consistent Global States

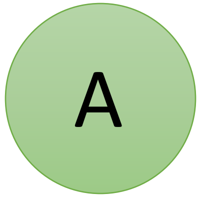
State of A = S_k

Achieving Consistent Global States

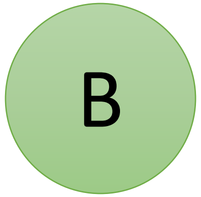


State of A = S_k

Achieving Consistent Global States



State of A = S_k



State of A = S_k

Achieving Consistent Global States

State of A = S_k

State of A = S_k

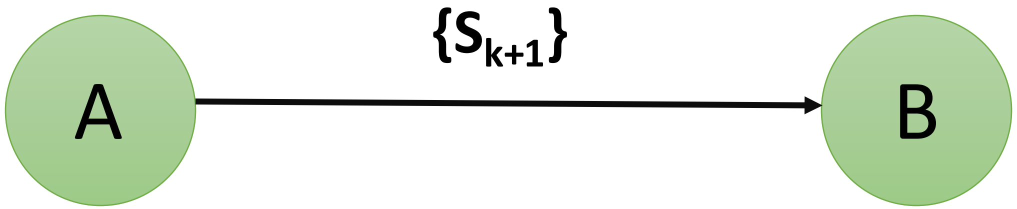
Achieving Consistent Global States

State of A = S_k

State of A = S_{k+1}

State of A = S_k

Achieving Consistent Global States



State of A = S_k

State of A = S_{k+1}

State of A = S_k

Achieving Consistent Global States

A

State of A = S_k

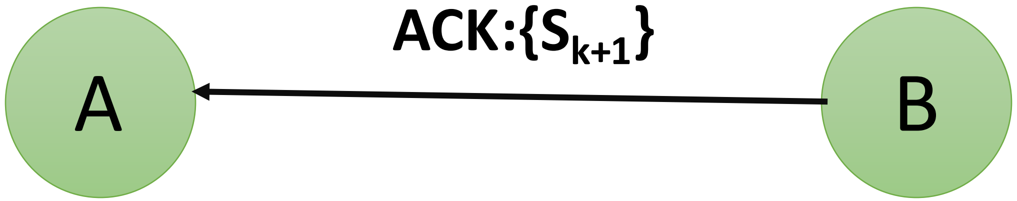
State of A = S_{k+1}

B

State of A = S_k

State of A = S_{k+1}

Achieving Consistent Global States



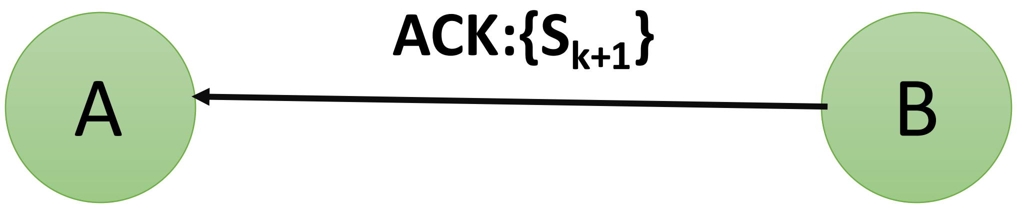
State of A = S_k

State of A = S_{k+1}

State of A = S_k

State of A = S_{k+1}

Achieving Consistent Global States



State of A = S_k

State of A = S_{k+1}

State of A = S_{k+2}

State of A = S_k

State of A = S_{k+1}

Achieving Consistent Global States

A

State of A = S_k

State of A = S_{k+1}

State of A = S_{k+2}

State of A = S_{k+3}

...

...

B

State of A = S_k

State of A = S_{k+1}

State of A = S_{k+2}

...

...

Why Do We Need Consistent Global States

- Important for many distributed problems
  - Deadlock detection
  - Termination detection
  - System checkpoint, backup and recovery
  - Analysis of system logs
  - …

How Do We Compute Consistent Global States?

- Determine the global state *S* of the system; compute *y(S)* to see if the stable property *y* holds

- **Core idea:**
  - Distributed algorithms typically works in phases: transient (when the computation is done), stable (system cycles endlessly without changing its states)
  - Typical life cycle of a distributed system: T(v1), S(v1), T(v2), S(v2), T(v3), S(v3), … ; v1, v2, v3, … are the views of the system
  - Detect the stability to shift the system from the current view to the next view
  - Take the checkpoint during this stable phase: no agent will change its state during this phase -> no ping-pong among checkpointed states

How Do We Compute Consistent Global States?

- We have three information
  - The local state of agent i
  - The local state of agent j
  - Agent i sends a message to agent j, which changes the local states of agents i and j

- Say, there are two events E(e1) and E(e2). These events can be **S(i,M)** -- Send message M to agent i, **R(i, M) --** Receive message M from agent i, etc. To decide a consistent global state, we need to first determine that E(e1) has happened before E(e2)

- We do not have any global clock; the local clocks might not be synchronized – **How do we do such ordering of events**?

Modeling the System

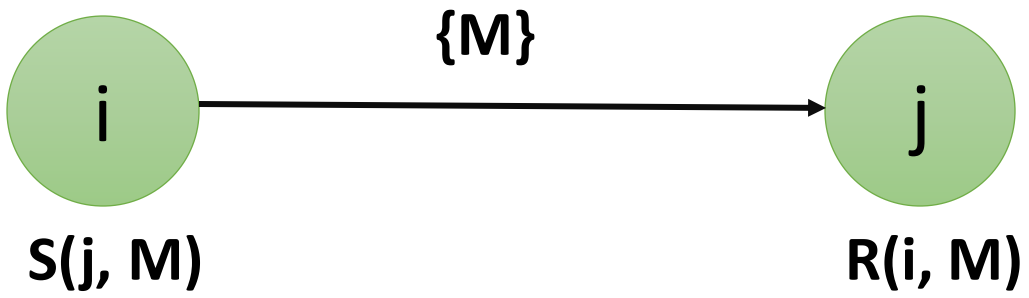
- Finite set of processes p1, p2, p3, …, pn
- Finite set of channels – each pair of processes has a channel
  - Channels are reliable but may deliver messages out of order
- Activities are distributed among the n number of processes; activities are event-triggered
- Each process can observe three kinds of events
  - Events local to that process
  - Send message M to process Pi
  - Receive message M from process Pj
- Events are recorded at the local history of the processes; union of these local histories from all the participating processes is the global history (**do not confuse this with the global state**)

Global History

- Global history is just the union of the local histories from the participating processes
  - Does not **order** the events

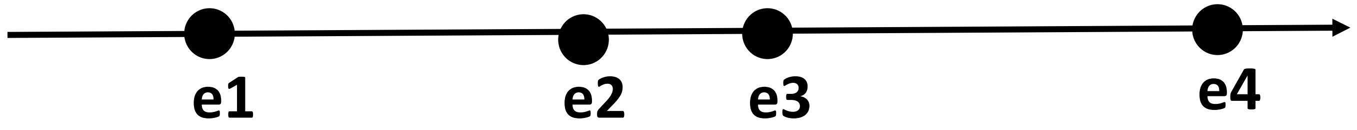
- An event E(e1) is ordered with respect to another event E(e2) if E(e1) happening affects to E(e2) in some way
  - Agent i triggers S(j, M); this can trigger R(i, M) at agent j

- We order the events using **"happened before"** relationship
  - S(j, M) happened before R(i, M)



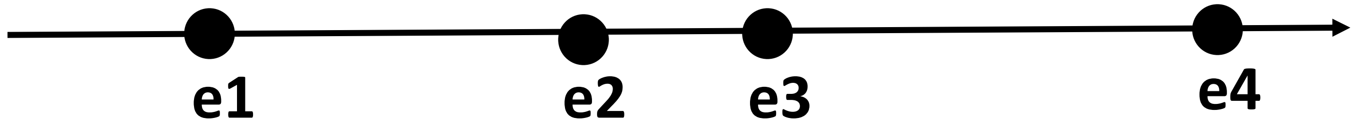
Partial Order

- Events of a process form a sequence



Partial Order of Events

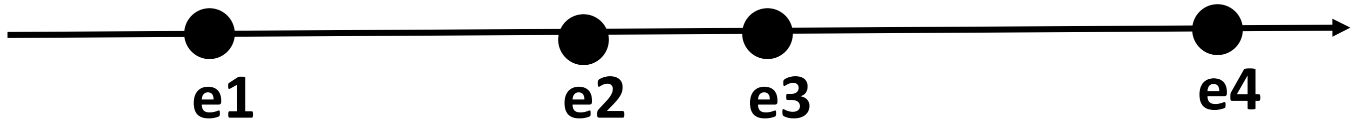
- Events of a process form a sequence



- ei happens before ej (ei -> ej) iff one the following three conditions hold:
  - ei and ej are the events of same process and ei occurs before ej locally, then ei -> ej
  - If ei = S(y, M) at process x and ej = R(x, M) at process y, then ei -> ej
  - If ei -> ej and ej -> ek, then ei -> ek

Partial Order of Events

- Events of a process form a sequence

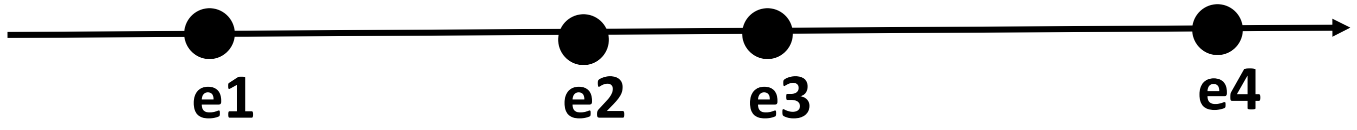


- ei happens before ej (ei -> ej) iff one the following three conditions hold:
  - ei and ej are the events of same process and ei occurs before ej locally, then ei -> ej
  - If ei = S(y, M) at process x and ej = R(x, M) at process y, then ei -> ej
  - If ei -> ej and ej -> ek, then ei -> ek

- All other events are considered concurrent
  - If ei and ej are concurrent, then both ei -> ej and ej -> ei are false

Partial Order of Events

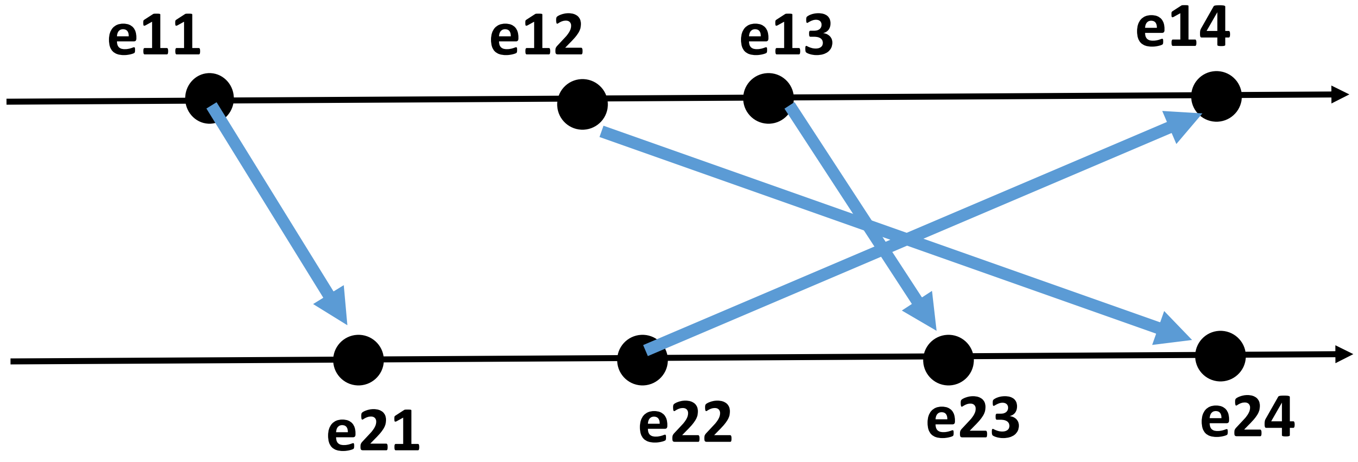
- Events of a process form a sequence



- ei happens before ej (ei -> ej) iff one the following three conditions hold:
  - ei and ej are the events of same process and ei occurs before ej locally, then ei -> ej
  - If ei = S(y, M) at process x and ej = R(x, M) at process y, then ei -> ej
  - If ei -> ej and ej -> ek, then ei -> ek
- All other events are considered concurrent
  - If ei and ej are concurrent, then both ei -> ej and ej -> ei are false
- **Formally, a distributed computation is poset (E, ->)**

Partial Order of Events

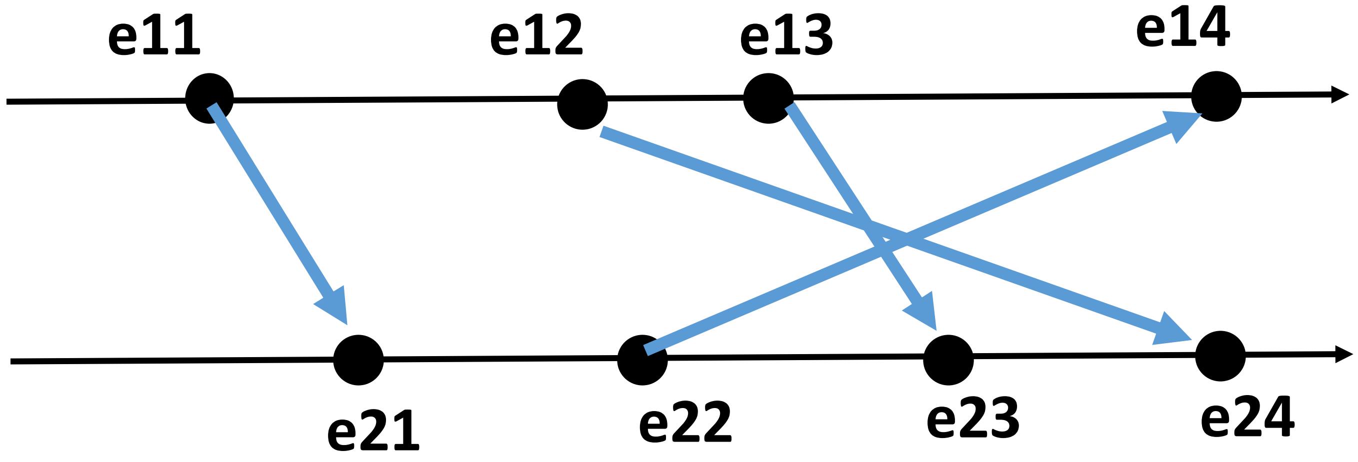
- Not all the events are ordered in a distributed system
- Events within each process are totally ordered
- Events across processes are partially ordered



e11 -> e12 -> e13 -> e14
e21 -> e22 -> e23 -> e24
e11 -> e21
e12 -> e24
e13 -> e23
e22 -> e14

Partial Order of Events

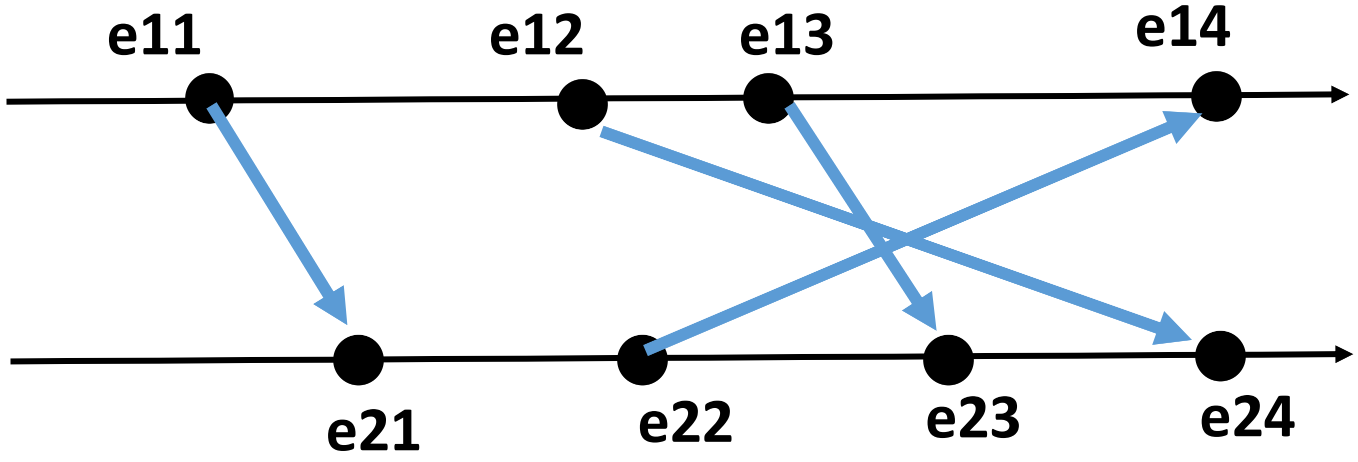
- Not all the events are ordered in a distributed system
- Events within each process are totally ordered
- Events across processes are partially ordered



e11 -> e12 -> e13 -> e14
e21 -> e22 -> e23 -> e24
e11 -> e21
e12 -> e24
e13 -> e23
e22 -> e14
e21 -> e14 ?

Partial Order of Events

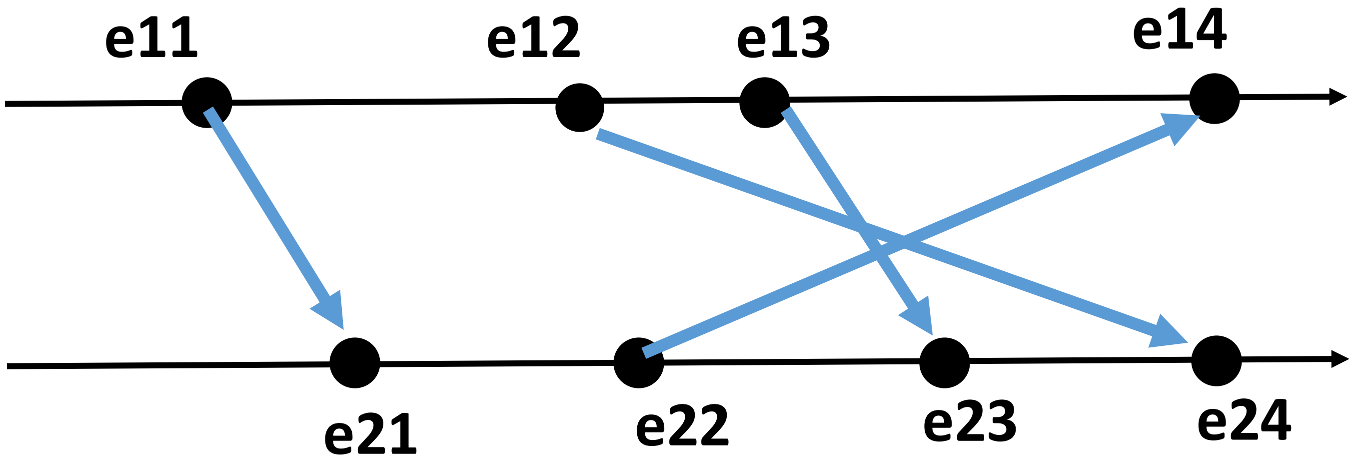
- Not all the events are ordered in a distributed system
- Events within each process are totally ordered
- Events across processes are partially ordered



e11 -> e12 -> e13 -> e14
e21 -> e22 -> e23 -> e24
e11 -> e21
e12 -> e24
e13 -> e23
e22 -> e14
e21 -> e14
e23 -> e14 ?

Lamport's Clock

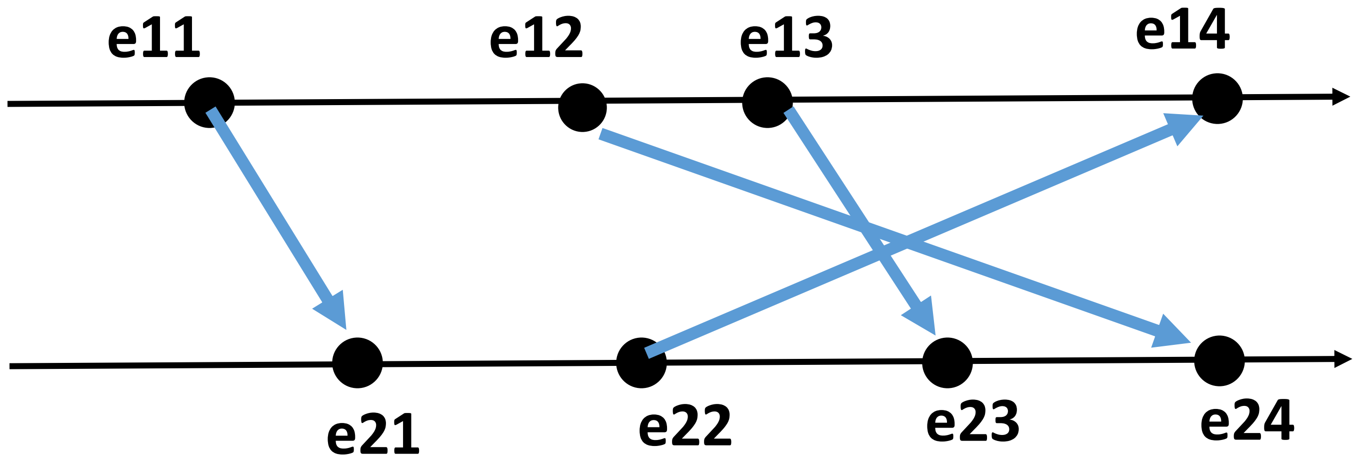
- Let C(e) be the logical clock for the event e; the logical time when e happens. Then, for any two events e1 and e2: e1 -> e2 => C(e1) < C(e2)



C(e11) < C(e23)
C(e21) < C(e14)
C(e12) < C(e23)
…

Lamport's Clock

- Let C(e) be the logical clock for the event e; the logical time when e happens. Then, for any two events e1 and e2: e1 -> e2 => C(e1) < C(e2)

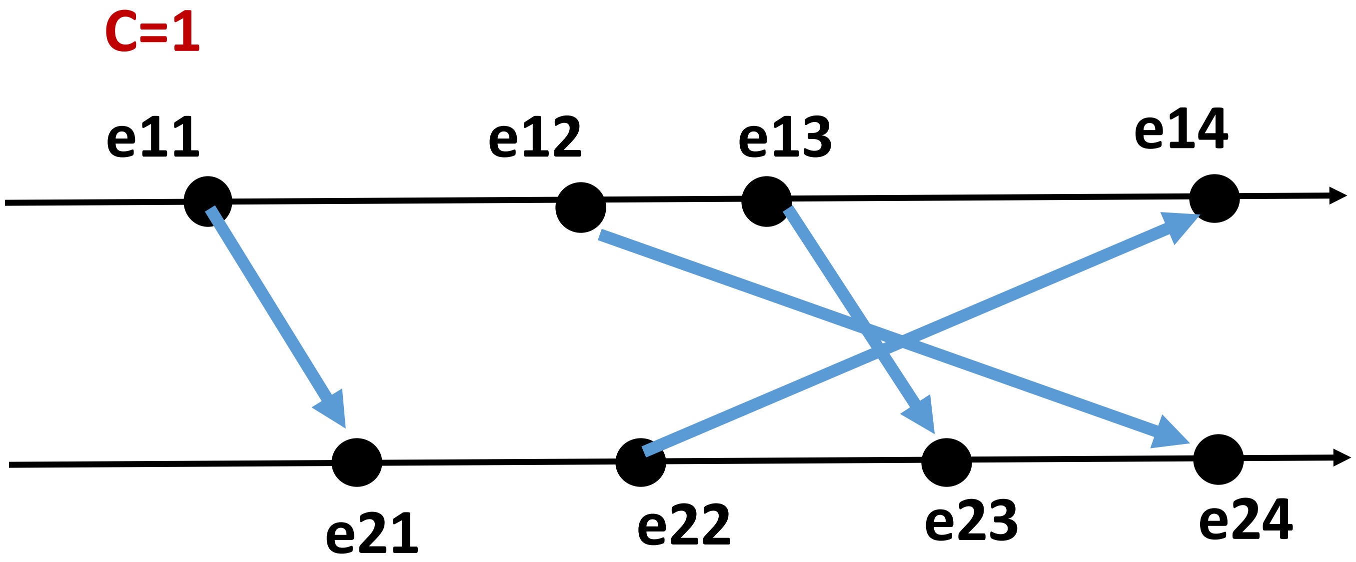


C(e11) < C(e23)
C(e21) < C(e14)
C(e12) < C(e23)
…

Lamport's Clock provides a relative ordering of the events in a Distributed System

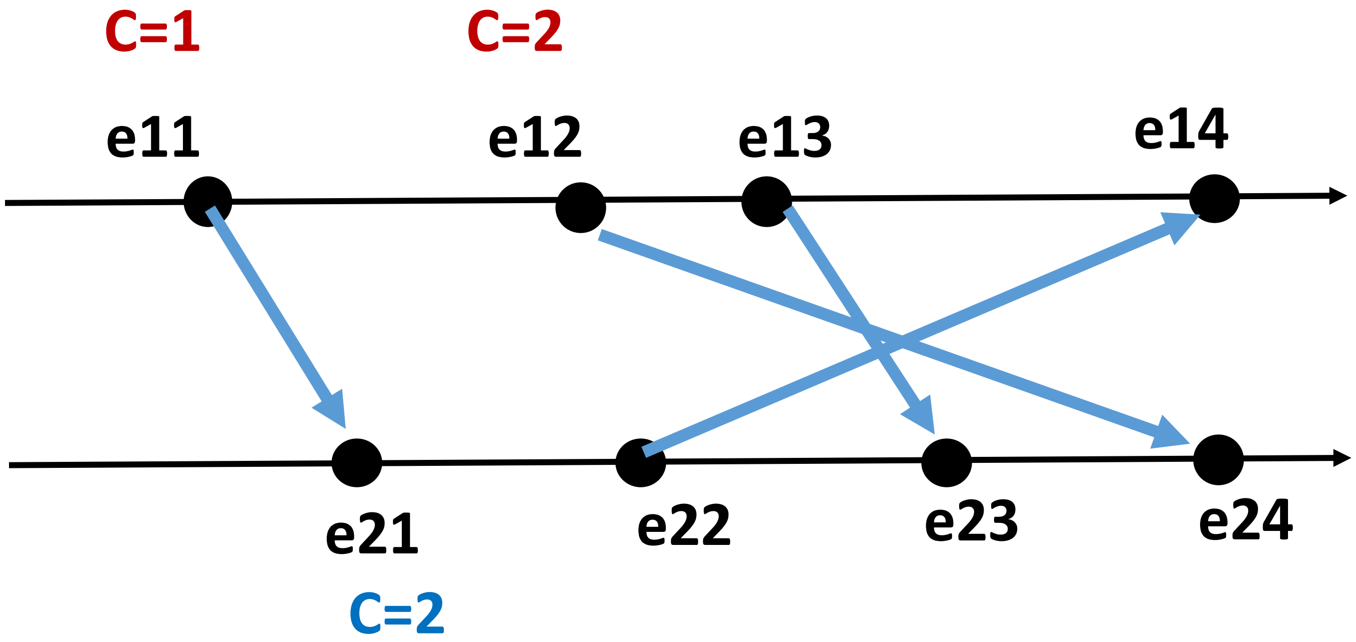
Lamport's Clock

- The clock ticks with each event. Each message contains the send timestamp.



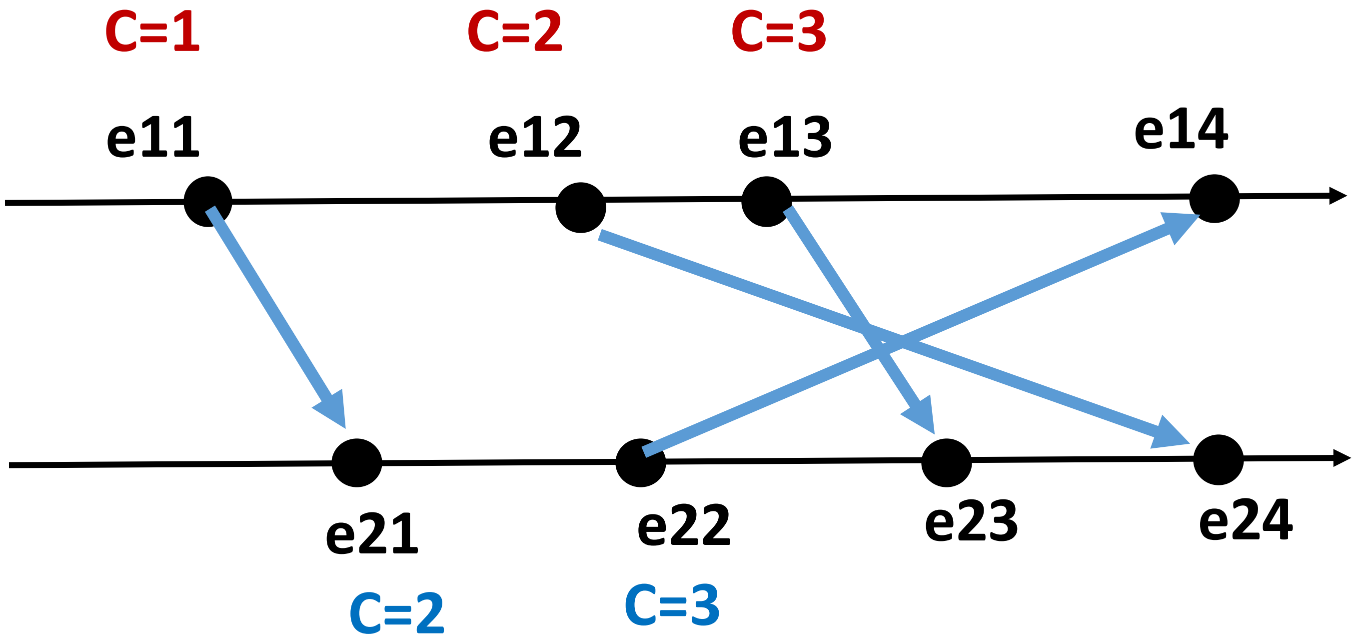
Lamport's Clock

- The clock ticks with each event. Each message contains the send timestamp.



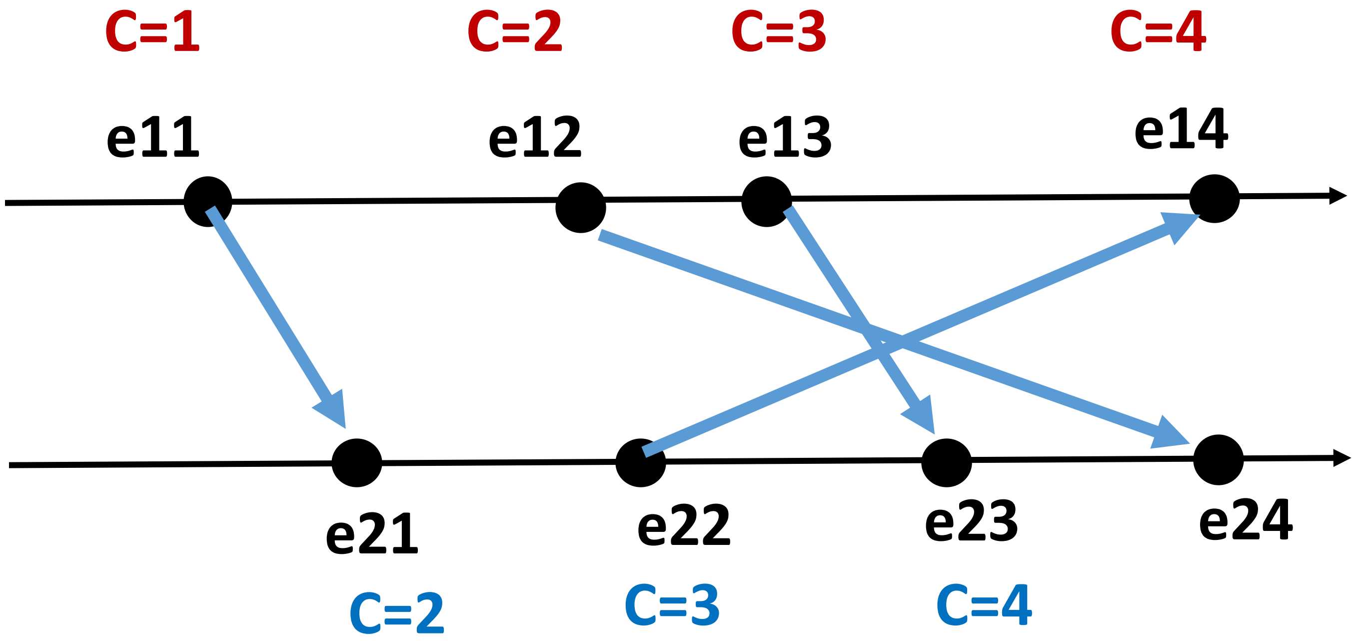
Lamport's Clock

- The clock ticks with each event. Each message contains the send timestamp.



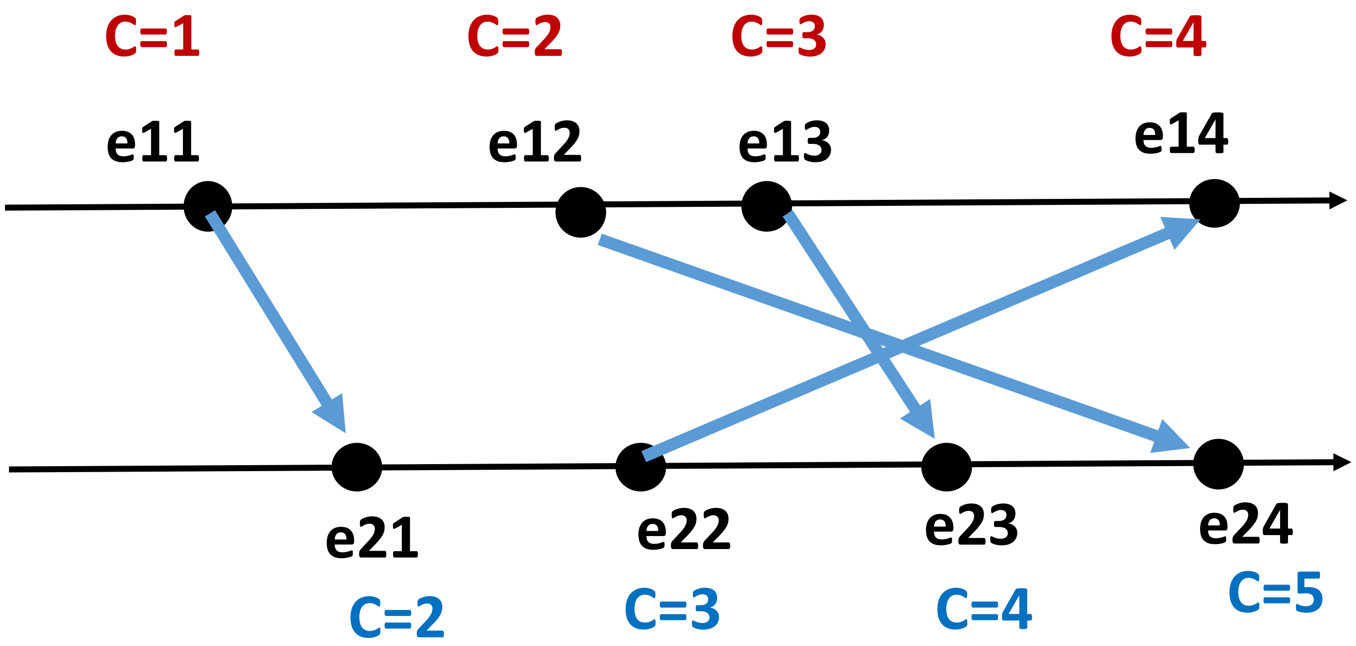
Lamport's Clock

- The clock ticks with each event. Each message contains the send timestamp.



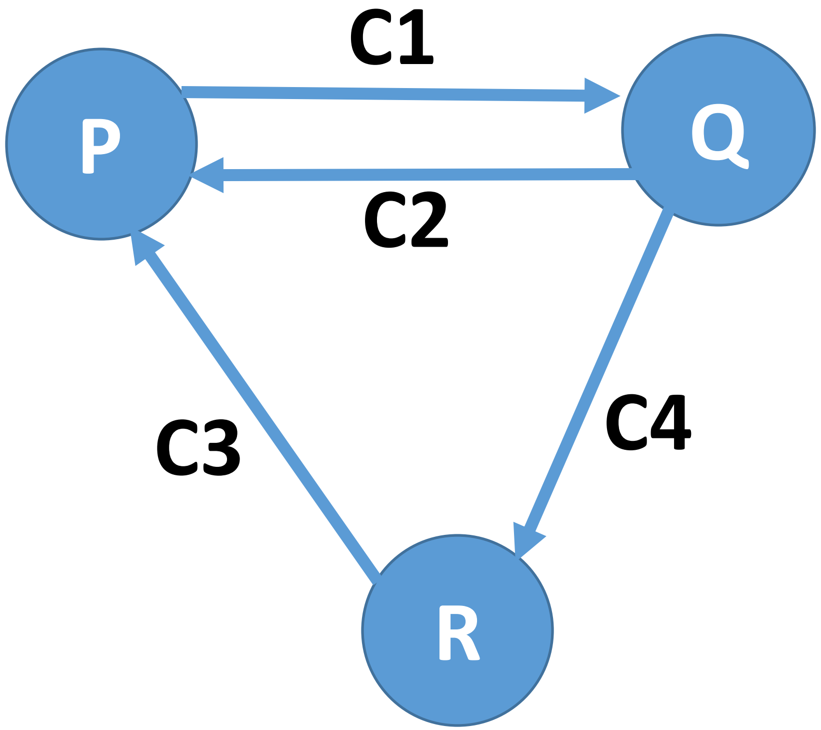
Lamport's Clock

- The clock ticks with each event. Each message contains the send timestamp.



Chandy Lamport Algorithm for Distributed Snapshot

- Model of a distributed system
  - A set of process -> nodes of a graph
  - A set of communication channels -> edges of the graph



Chandy Lamport Algorithm for Distributed Snapshot

- **Assumptions**:
  - There is a channel between any two processes in the system
  - There is no failure, each message arrives exactly once
  - Communication channels are unidirectional and FIFO ordered
  - Channels are assumed to have infinite buffer
  - Delay experienced by a message in the channel is arbitrary but finite
  - There is no global clock, events are ordered based on their logical clock

- The state of a channel is the sequence of messages sent along the channel (excluding the sequence of messages received through the channel)

- Any process can start the snapshot algorithm

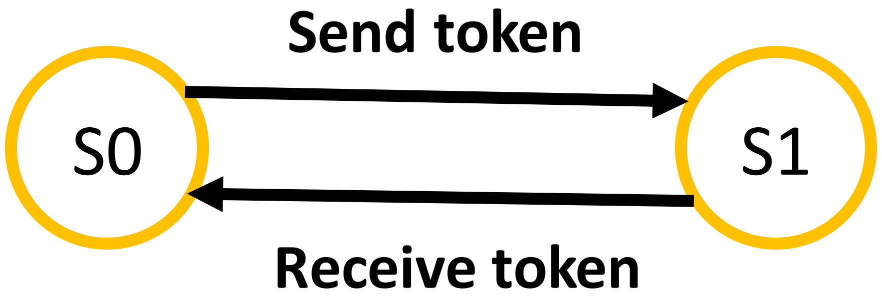
Chandy Lamport Algorithm for Distributed Snapshot

- Process is defined by a set of states, an initial state and a set of events associated with that process
- An event e in a process p is an atomic action
  - May change the state of p
  - May change the state of **at most one channel** c incident on p
- **The global state of distributed system is a set of component process and channel states**
  - **Initial state**: Each process are in their initial state, each channel is in their empty state
  - The occurrence of an event changes the global state of the distributed system
  - *next* (S, e) : The global state immediately after the occurrence of an event e in the global state S

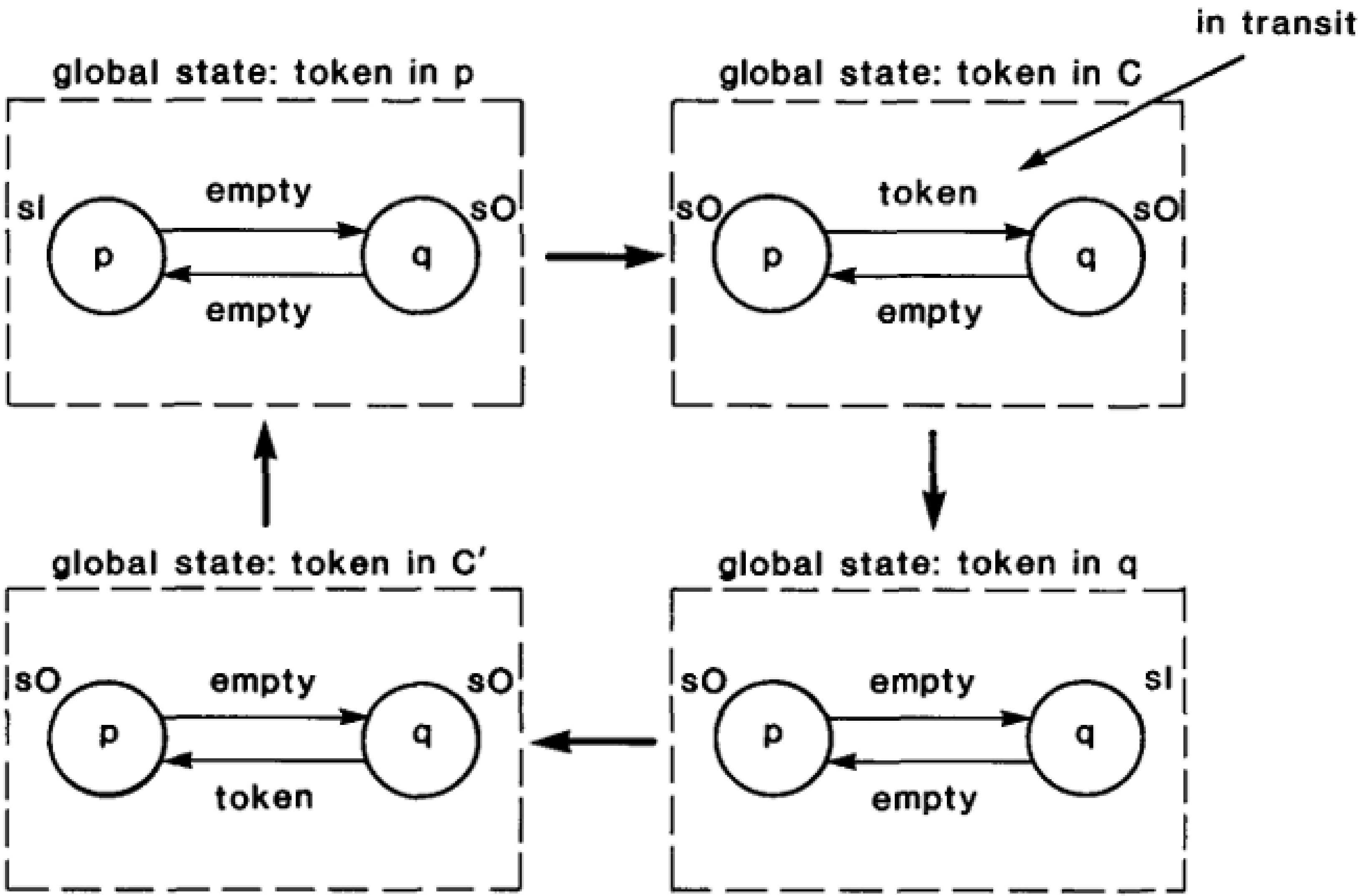
Chandy Lamport Algorithm for Distributed Snapshot

- A distributed system moves through a sequence of global states starting from the initial state S_0 with $S_{i+1} = next\ (S_i, e_i)$ for $0 \leq i \leq n$.
  - **Note that as per Lamport's clock, distributed computation is a partially ordered set of events**

- Let us consider a single token conservation system



Single Token Conservation System

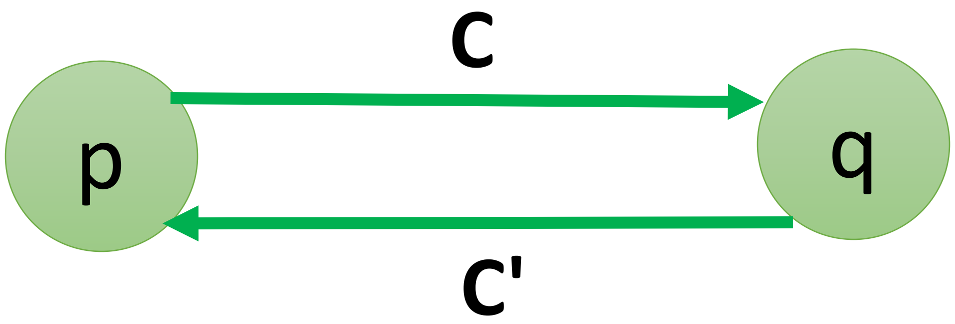


Chandy-Lamport Algorithm: Broad Idea

- The algorithm runs by sending token in forms of messages
  - It must run concurrently with the underlying computation, but should not alter or interfere the computation

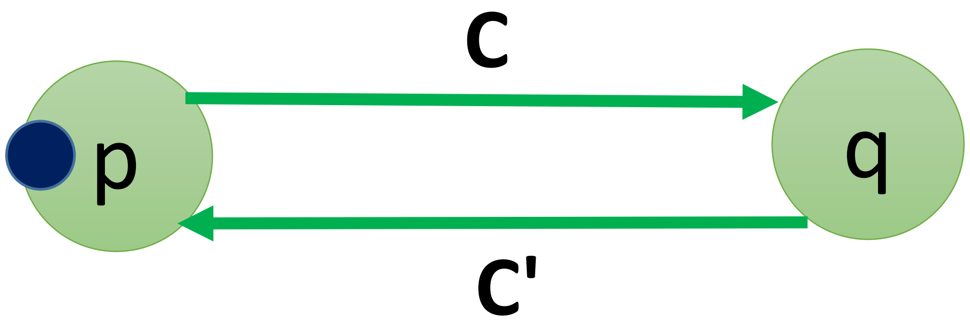
Chandy-Lamport Algorithm: Broad Idea

- The algorithm runs by sending token in forms of messages
  - It must run concurrently with the underlying computation, but should not alter or interfere the computation



Chandy-Lamport Algorithm: Broad Idea

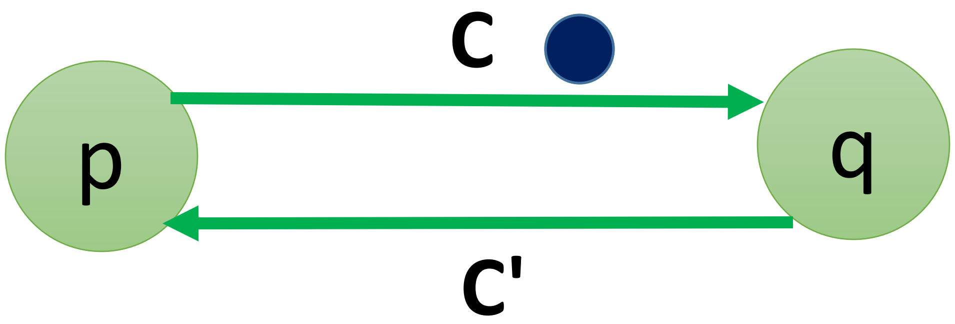
- The algorithm runs by sending token in forms of messages
  - It must run concurrently with the underlying computation, but should not alter or interfere the computation



- p records its state as in-p

Chandy-Lamport Algorithm: Broad Idea

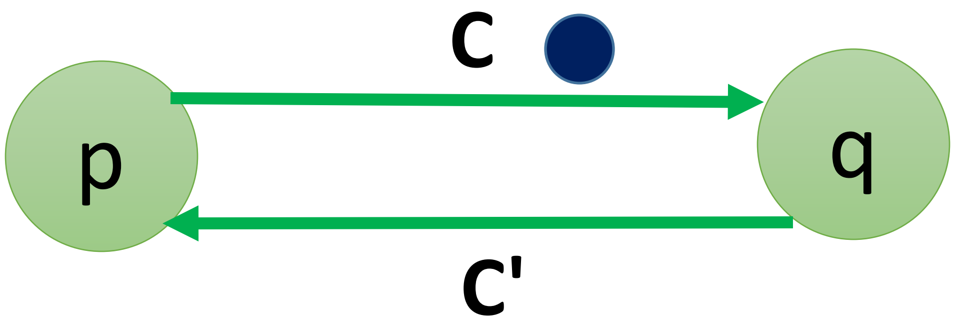
- The algorithm runs by sending token in forms of messages
  - It must run concurrently with the underlying computation, but should not alter or interfere the computation



- p records its state as in-p

Chandy-Lamport Algorithm: Broad Idea

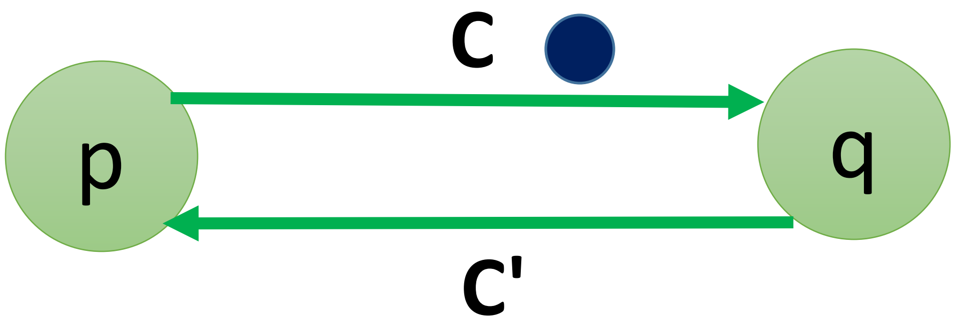
- The algorithm runs by sending token in forms of messages
  - It must run concurrently with the underlying computation, but should not alter or interfere the computation



- p records its state as in-p
- q records
  - state of C as in-C, state of C' as null,
  - its own state as null

Chandy-Lamport Algorithm: Broad Idea

- The algorithm runs by sending token in forms of messages
  - It must run concurrently with the underlying computation, but should not alter or interfere the computation

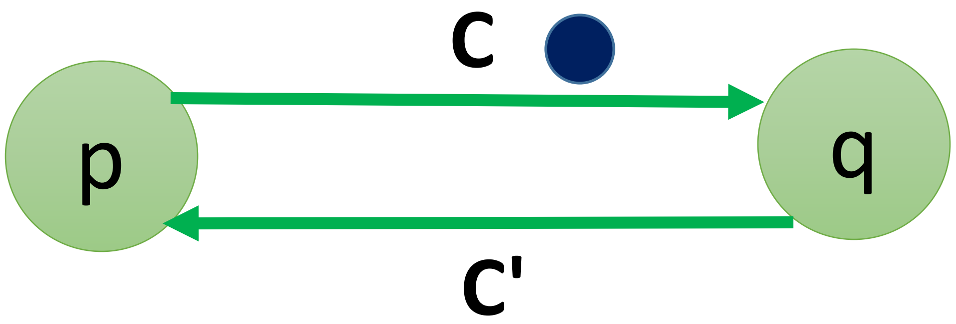


- p records its state as in-p
- q records
  - state of C as in-C, state of C' as null,
  - its own state as null

Inconsistent global state, two tokens in the system

Chandy-Lamport Algorithm: Broad Idea

- The algorithm runs by sending token in forms of messages
  - It must run concurrently with the underlying computation, but should not alter or interfere the computation



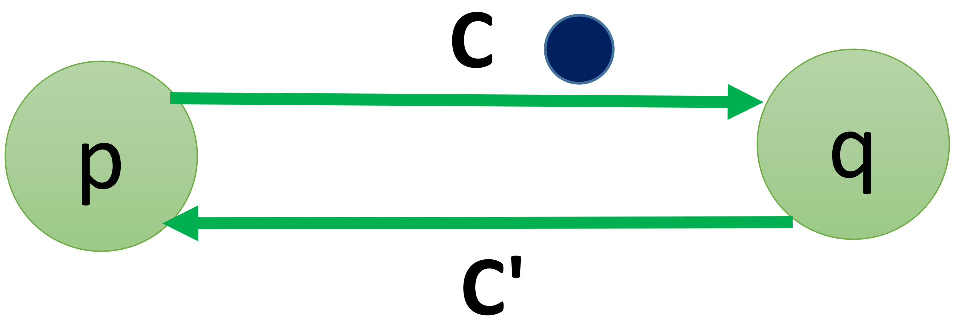
- p records its state as in-p
- q records
  - state of C as in-C, state of C' as null,
  - its own state as null

Inconsistent global state, two tokens in the system

The state of p is recorded before sending token, the state of C is recorded after sending token

Chandy-Lamport Algorithm: Broad Idea

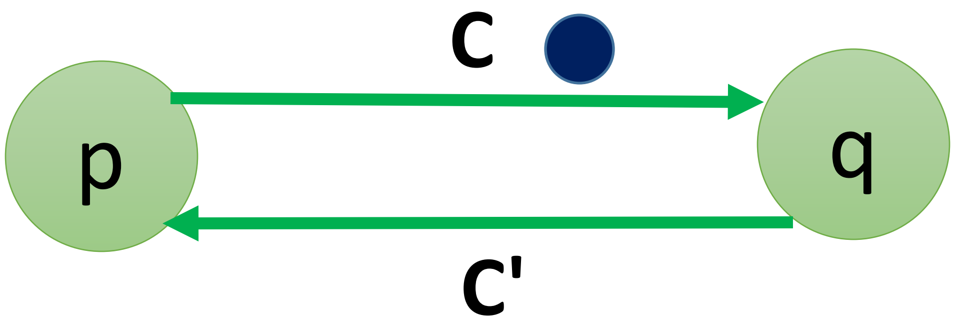
- The algorithm runs by sending token in forms of messages
  - It must run concurrently with the underlying computation, but should not alter or interfere the computation



- Let n be the number of messages sent from p before p's state is recorded, let n' be the number of messages sent through C before its state is recorded
  - The recorded global state may be inconsistent if n < n'

Chandy-Lamport Algorithm: Broad Idea

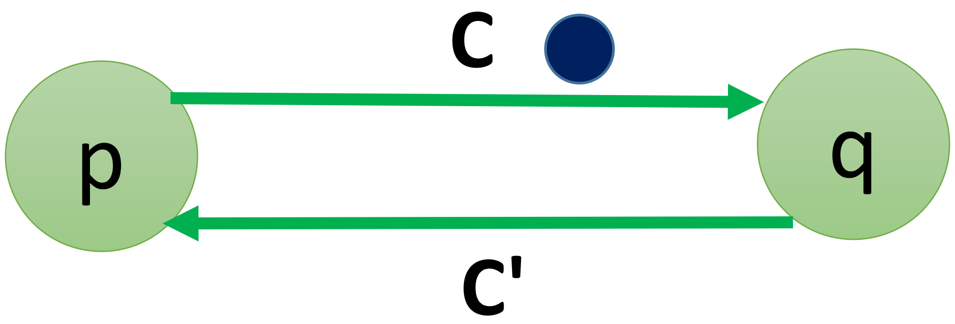
- The algorithm runs by sending token in forms of messages
  - It must run concurrently with the underlying computation, but should not alter or interfere the computation



- The recorded global state may also be inconsistent if n > n' -- **try to show this!**

Chandy-Lamport Algorithm: Broad Idea

- The algorithm runs by sending token in forms of messages
  - It must run concurrently with the underlying computation, but should not alter or interfere the computation



- A consistent global state requires n = n'.

The Algorithm

- **Marker sending rule for a Process p:** For each channel C, incident on, and directed away from p
  - p sends one marker along C after p records its state and before p sends further messages along C

The Algorithm

- **Marker sending rule for a Process p:** For each channel C, incident on, and directed away from p
  - p sends one marker along C after p records its state and before p sends further messages along C
- **Marker receiving rule for a Process q:** On receiving a marker along a channel C
  - if q has not recorded its state, then q records its state; q records the state C as empty
  - Else q records the state of C as the sequence of messages received along C after q's state was recorded and q received the marker along C

The Algorithm

- **Marker sending rule for a Process p:** For each channel C, incident on, and directed away from p
  - p sends one marker along C after p records its state and before p sends further messages along C
- **Marker receiving rule for a Process q:** On receiving a marker along a channel C
  - if q has not recorded its state, then q records its state; q records the state C as empty
  - Else q records the state of C as the sequence of messages received along C after q's state was recorded and q received the marker along C

Can you argue that the global state collected through Chandy-Lamport's Algorithm will be Consistent?

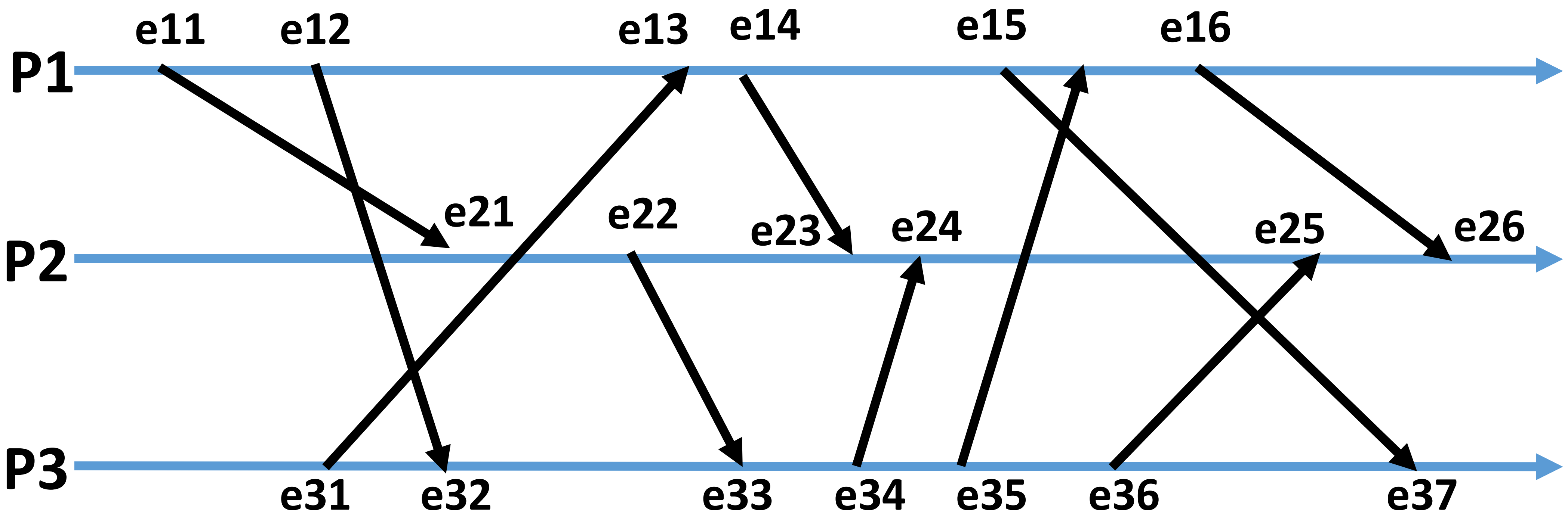
Cuts of a Distributed Computation

- Is a subset of its global history H and contains an initial prefix of each of the agent's local history
  - Defined by the tuple (C1, C2, C3, …., CN) where each process Pi's last event in the cut is Ci

Study Material: Özalp Babaoğlu and Keith Marzullo. 1993. Consistent global states of distributed systems: fundamental concepts and mechanisms. Distributed systems (2nd Ed.)

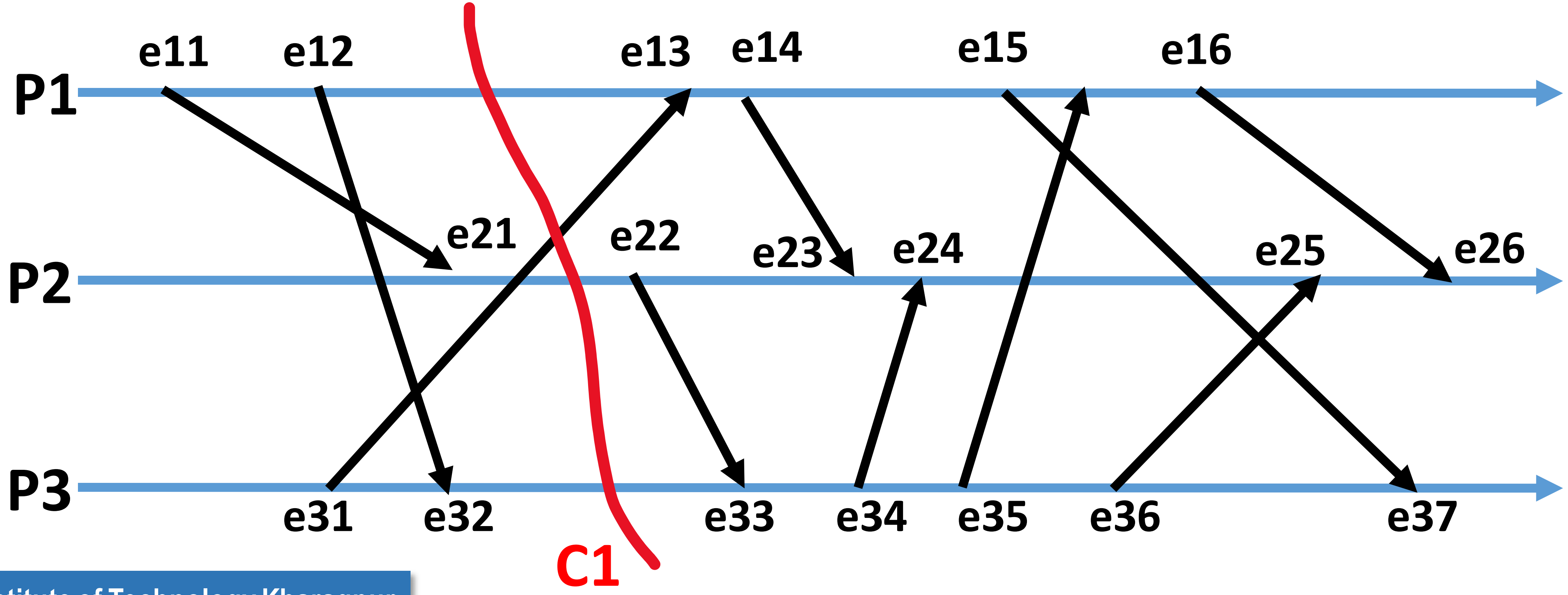
Cuts of a Distributed Computation

- Is a subset of its global history H and contains an initial prefix of each of the agent's local history
  - Defined by the tuple (C1, C2, C3, …., CN) where each process Pi's last event in the cut is Ci



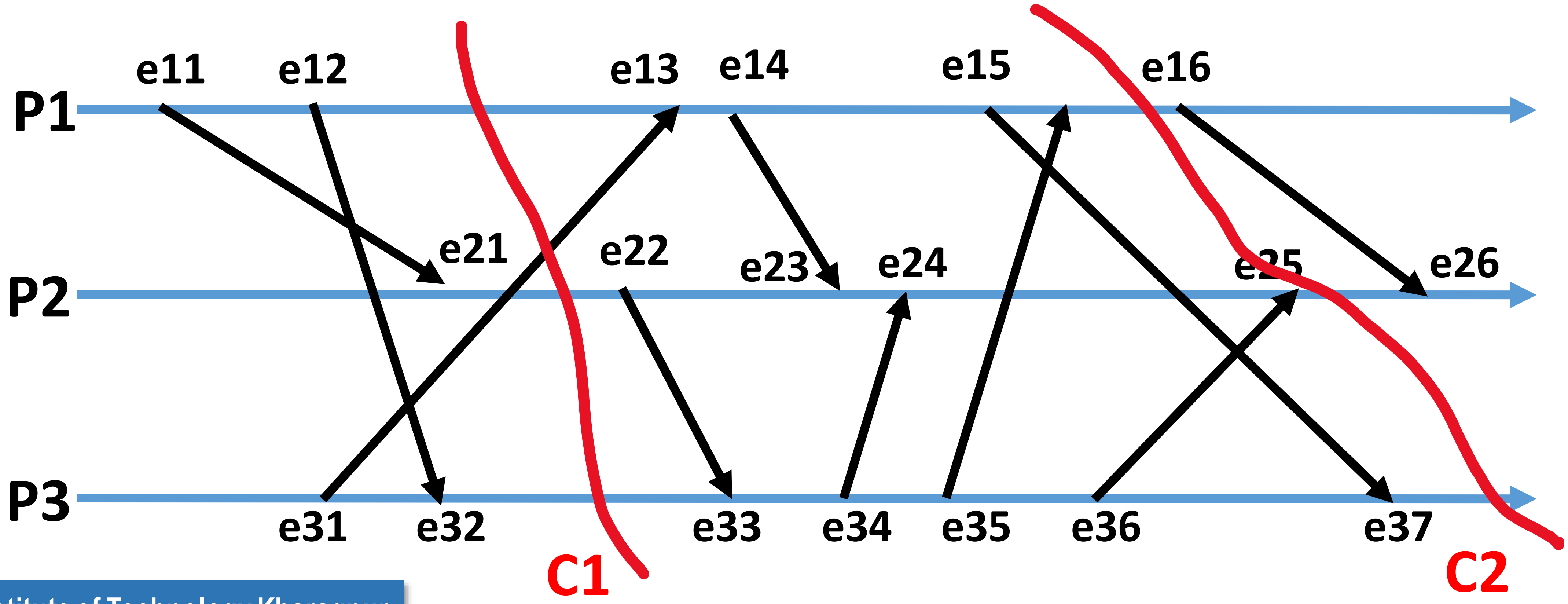
Cuts of a Distributed Computation

- Is a subset of its global history H and contains an initial prefix of each of the agent's local history
  - Defined by the tuple (C1, C2, C3, …., CN) where each process Pi's last event in the cut is Ci



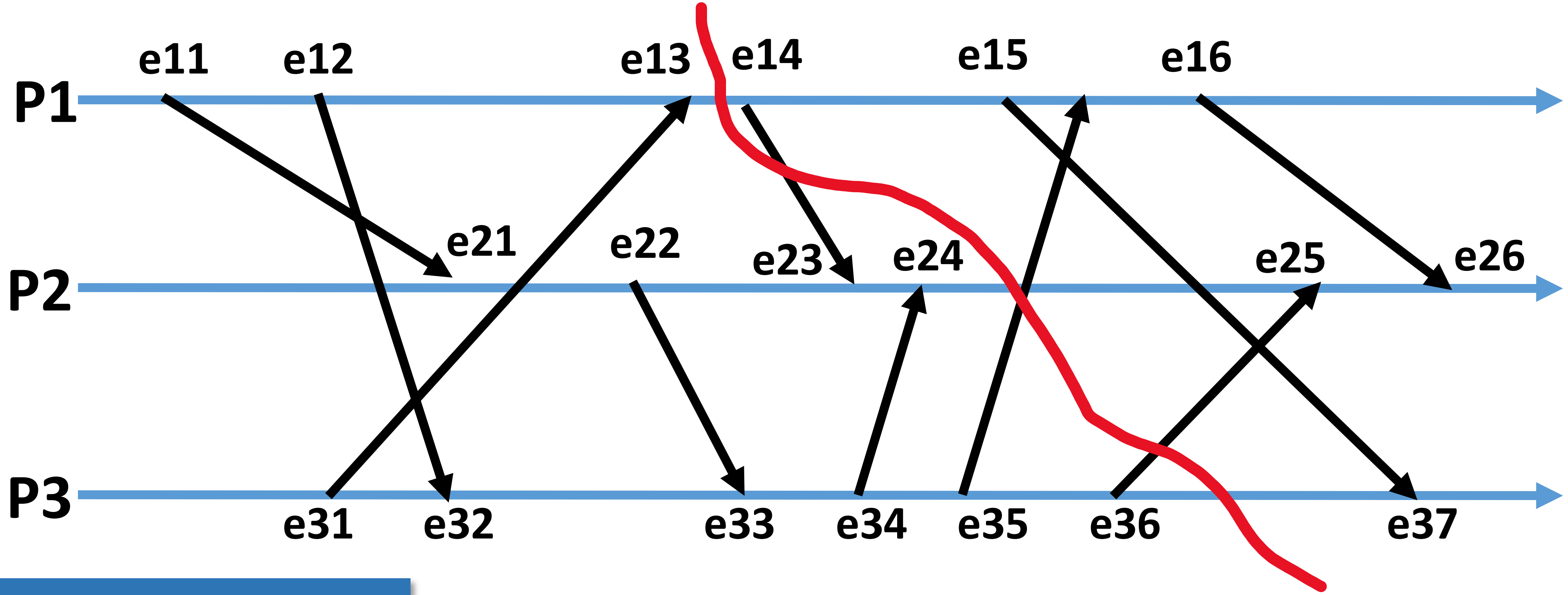
Cuts of a Distributed Computation

- Is a subset of its global history H and contains an initial prefix of each of the agent's local history
  - Defined by the tuple (C1, C2, C3, …., CN) where each process Pi's last event in the cut is Ci



Inconsistent Cuts

- If a cut includes receipt of a message but not the sending of a message, then it is inconsistent
  - If e(i, j) < e(k, l) and e(k, l) is in the cut, then e(i, j) must be there in the cut



Inconsistent Cuts

- If a cut includes receipt of a message but not the sending of a message, then it is inconsistent
  - If $e(i, j) < e(k, l)$ and $e(k, l)$ is in the cut, then $e(i, j)$ must be there in the cut

- Formally, a cut C is consistent if for all events e and e',
  $(e \in C) \wedge (e' \rightarrow e) \Rightarrow (e' \in C)$

- **Consistent global states correspond to consistent cuts in a distributed system**
  - Check whether Chandy-Lamport's algorithm yield a consistent cut!

Inconsistent Cuts

- If a cut includes receipt of a message but not the sending of a message, then it is inconsistent
  - If e(i, j) < e(k, l) and e(k, l) is in the cut, then e(i, j) must be there in the cut

- Formally, a cut C is consistent if for all events e and e',
  $(e \in C) \wedge (e' \rightarrow e) \Rightarrow (e' \in C)$

- **Consistent global states correspond to consistent cuts in a distributed system**
  - Check whether Chandy-Lamport's algorithm yield a consistent cut!

- Indeed, global properties of a distributed system must be checked using consistent cuts

Runs

- A run of a distributed computation is a total ordering R
  - Includes all of the events from the global history
  - The events are consistent with each of the processes' local histories
- For process Pi, the events of Pi occurs in the same order in R as in the history of Pi
- However, there can be many possible runs of a single distributed computation with a history H
  - Remember the partial ordering notion as we discussed in Lamport's clock
  - Concurrent events can be placed in different orders in different possible runs

Reachable States

- A run R is consistent if for all e1 < e2 implies that e1 appears before e2 in R

- For any two global states S_i and S_j in a distributed computation, S_j is reachable from S_i if there is a sequence of consistent states S_k In between S_i and S_j, such that S_i -> S_k -> S_j in at least one run R.
  - S_i -> S_k indicates S_k = next (S_i, e) for an event e

Reachable States

- A run R is consistent if for all e1 < e2 implies that e1 appears before e2 in R

- For any two global states S_i and S_j in a distributed computation, S_j is reachable from S_i if there is a sequence of consistent states S_k In between S_i and S_j, such that S_i -> S_k -> S_j in at least one run R.
  - S_i -> S_k indicates S_k = *next* (S_i, e) for an event e

- The set of all consistent global states of the computation along with the *next* relation defines a lattice

Distributed Computation and Lattice

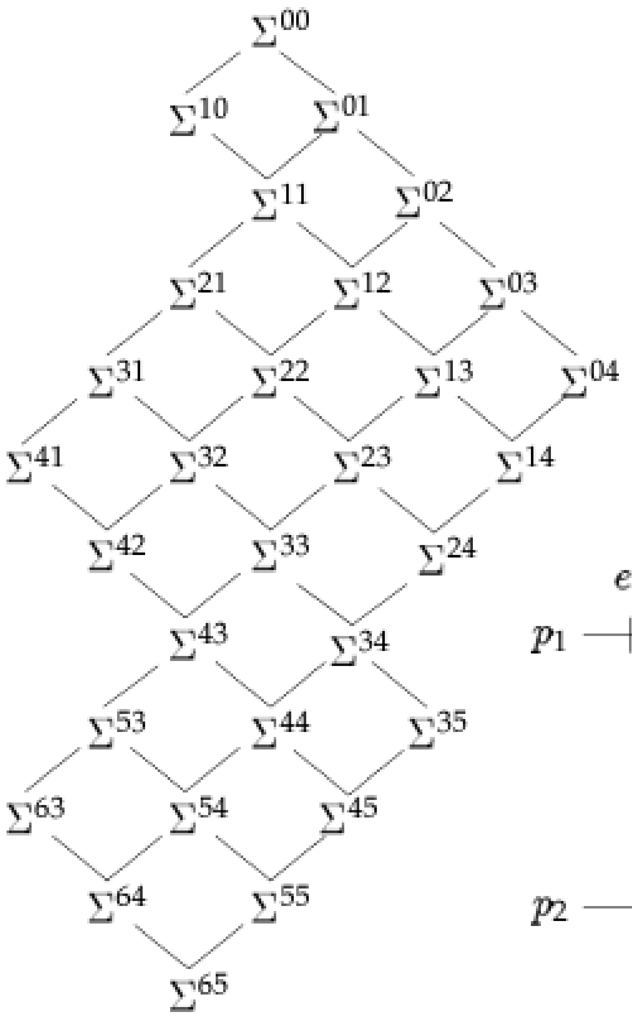
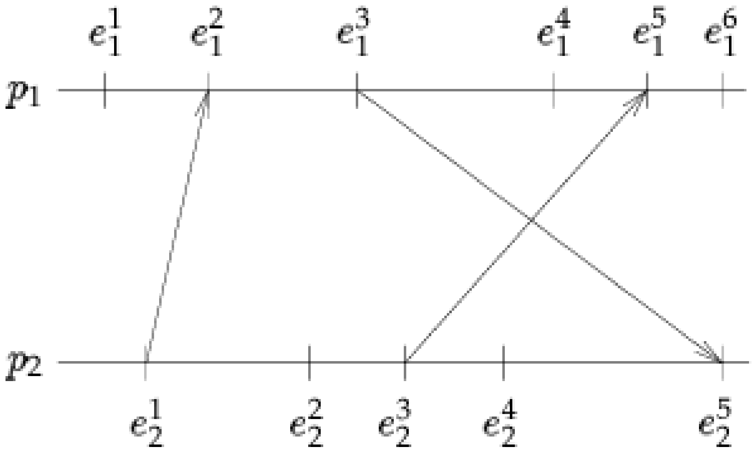


Image Source: Özalp Babaoğlu and Keith Marzullo. 1993. Consistent global states of distributed systems: fundamental concepts and mechanisms. Distributed systems (2nd Ed.)



Happy Learning