Distributed Systems

ECE428

Lecture 5

Adopted from Spring 2021

Recap: Timestamps Summary

- Comparing timestamps across events is useful.
  - Reconciling updates made to an object in a distributed datastore.
  - Rollback recovery during failures:

    1. Checkpoint state of the system; 2. Log events (with timestamps); 3. Rollback to checkpoint and replay events in order if system crashes.

- How to compare timestamps across different processes?
  - Physical timestamp: requires clock synchronization.
    - Google's Spanner Distributed Database uses "TrueTime".
  - Lamport's timestamps: cannot fully differentiate between causal and concurrent ordering of events.
    - Oracle uses "System Change Numbers" based on Lamport's clock.
  - Vector timestamps: larger message sizes.
    - Amazon's DynamoDB uses vector clocks.

Recap: Timestamps Summary

- Comparing timestamps across events is useful.
  - Reconciling updates made to an object in a distributed datastore.
  - Rollback recovery during failures:

    1. Checkpoint state of the system; 2. Log events (with timestamps); 3. Rollback to checkpoint and replay events in order if system crashes.
- How to compare timestamps across different processes?
  - Physical timestamp: requires clock synchronization.
    - Google's Spanner Distributed Database uses "TrueTime".
  - Lamport's timestamps: cannot fully differentiate between causal and concurrent ordering of events.
    - Oracle uses "System Change Numbers" based on Lamport's clock.
  - Vector timestamps: larger message sizes.
    - Amazon's DynamoDB uses vector clocks.

Today’s agenda

- Global State
  - Chapter 14.5
  - Goal: reason about how to capture the state across all processes of a distributed system without requiring time synchronization.

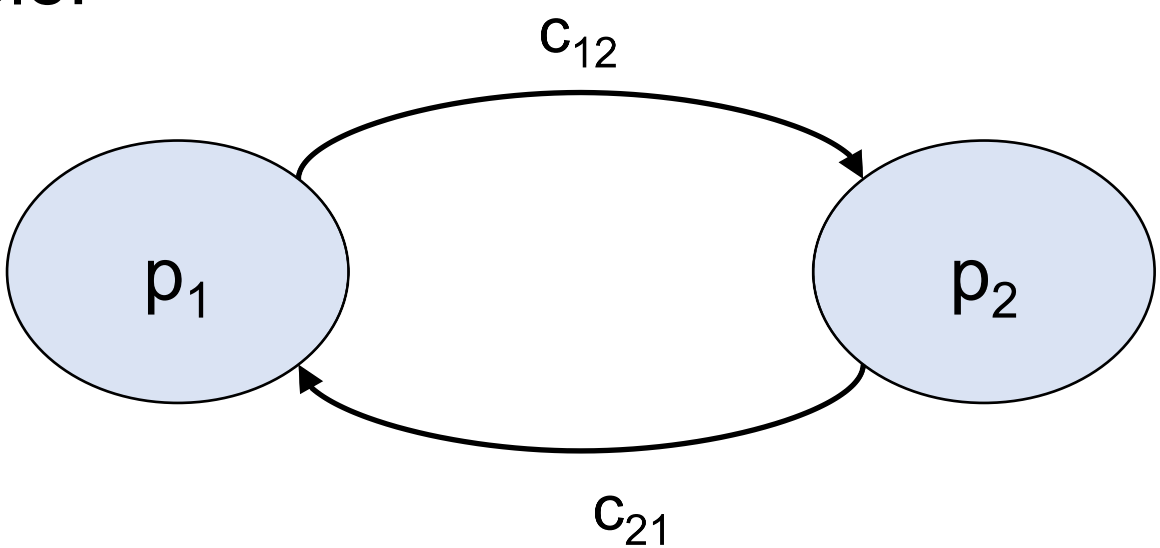
Process, state, events

- Consider a system with n processes: $<p_1, p_2, p_3, \ldots, p_n>$.
- Each process p_i is associated with *state* s_i.
  - State includes values of all local variables, affected files, etc.
- Each channel can also be associated with a state.
  - Which messages are currently *pending* on the channel.
  - Can be computed from process' state:
    - Record when a process sends and receives messages.
    - if p_i sends a message that p_j has not yet received, it is pending on the channel.
- State of a process (or a channel) gets transformed when an *event* occurs. 3 types of events:
  - local computation, sending a message, receiving a message.

Global State (or Global Snapshot)

- State of each process (and each channel) in the system at a given instant of time.

- Example:



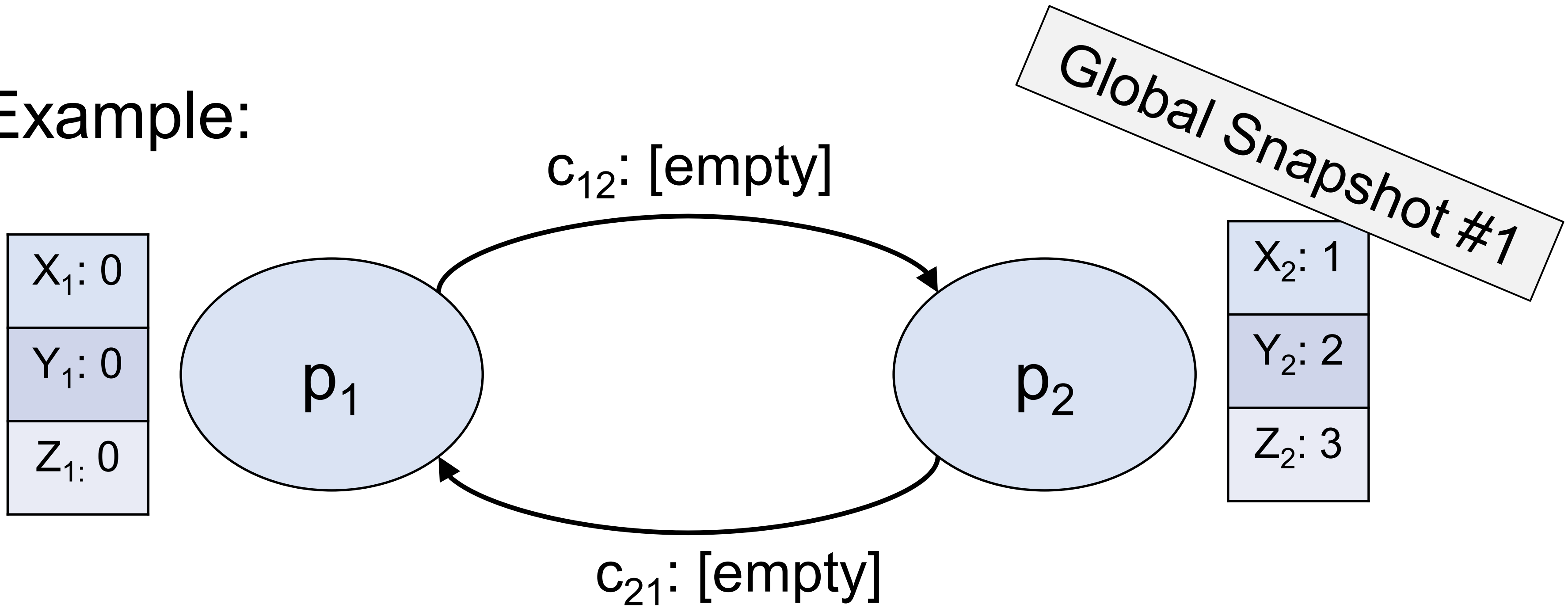
Two processes: p_1 and $p_{2.}$

c_{12}: channel from p_1 to $p_{2.}$ c_{21}: channel from p_2 to $p_{1.}$

Global State (or Global Snapshot)

- State of each process (and each channel) in the system at a given instant of time.

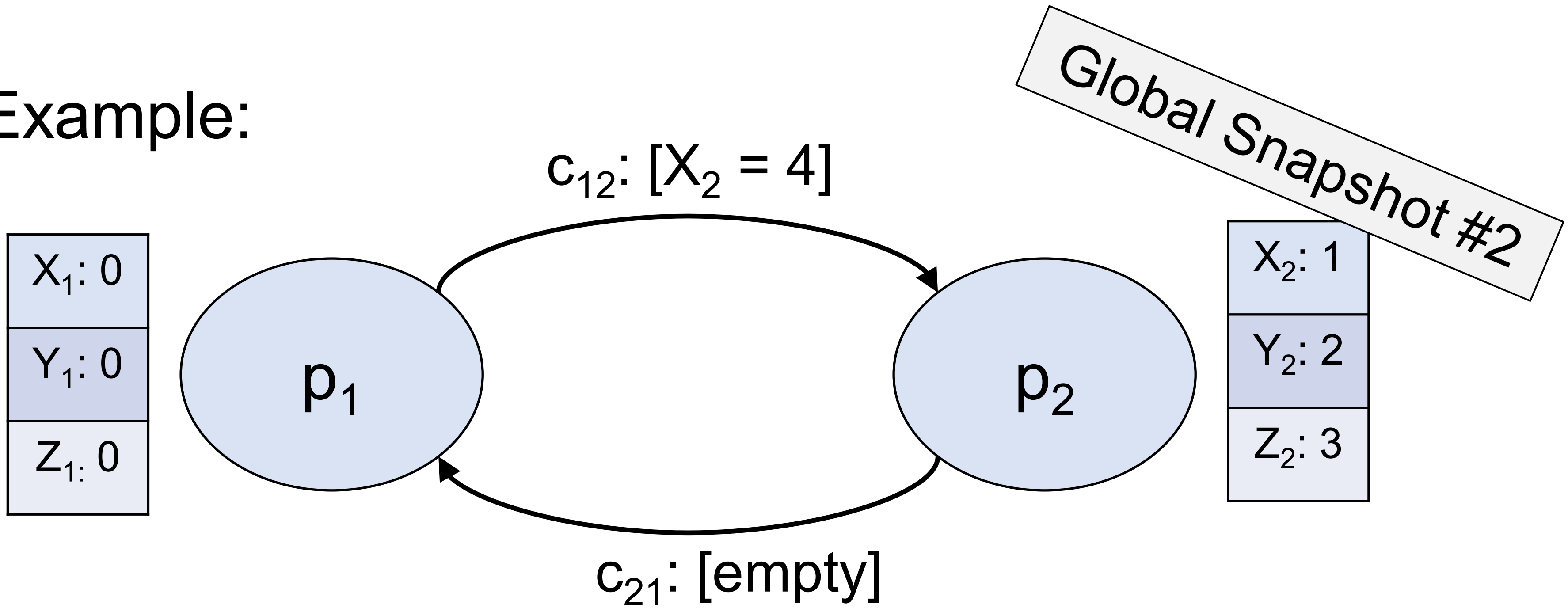
- Example:



Process state for p_1 and p_2.
No pending messages on the channels.

Global State (or Global Snapshot)

- State of each process (and each channel) in the system at a given instant of time.
- Example:

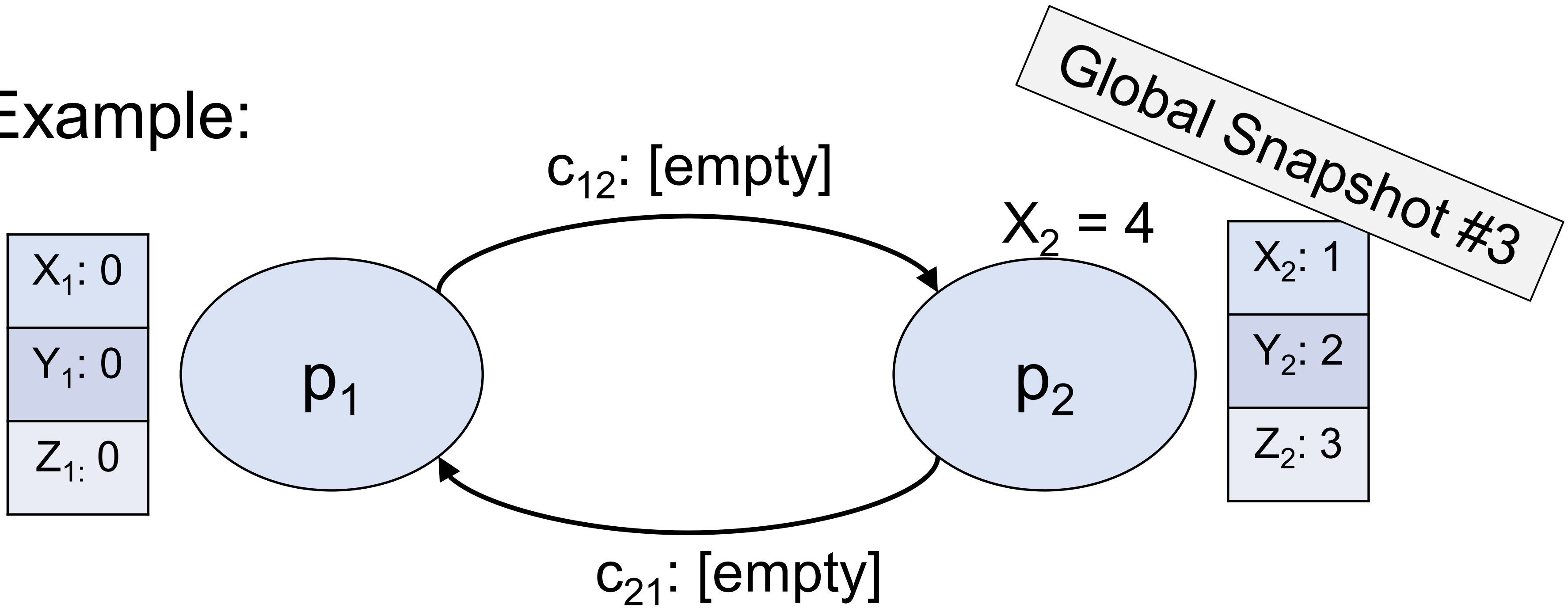


p_1 send a message to p_2 asking it to set $X_2 = 4$

Global State (or Global Snapshot)

- State of each process (and each channel) in the system at a given instant of time.

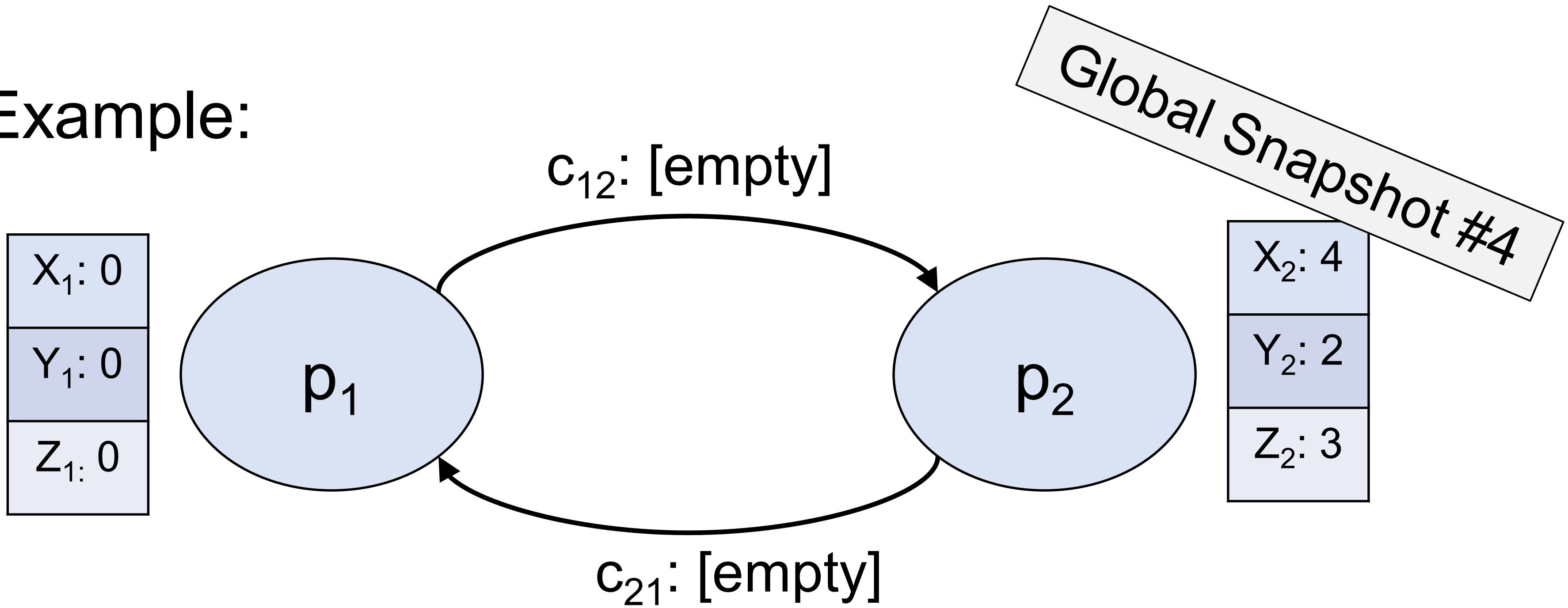
- Example:



p_2 receives the message.

Global State (or Global Snapshot)

- State of each process (and each channel) in the system at a given instant of time.
- Example:



p_2 changes the value of X_2

Capturing a global snapshot

- Useful to capture a global snapshot of the system:
    - *Checkpointing* the system state.
    - Reasoning about unreferenced objects (for garbage collection).
    - Deadlock detection.
    - Distributed debugging.

Capturing a global snapshot

- Difficult to capture a global snapshot of the system.
- Global state or global snapshot is state of each process (and each channel) in the system at a given *instant of time*.
- Strawman:
  - Each process records its state at 3:15pm.
  - We get the global state of the system at 3:15pm.
  - *But precise clock synchronization is difficult to achieve.*
- How do we capture global snapshots without precise time synchronization across processes?

More notations and definitions

- State of a process (or a channel) gets transformed when an *event* occurs.

- 3 types of events:
  - local computation, sending a message, receiving a message.

- e_i^n is the n^{th} event at p_i.

More notations and definitions

- For a process p_i , where events e_i^0, e_i^1, … occur:

  history(p_i) = h_i = $<e_i^0, e_i^1, \ldots >$

  prefix history(p_i^k) = h_i^k = $<e_i^0, e_i^1, \ldots, e_i^k >$

  s_i^k : p_i's state immediately after k^{th} event.
- For a set of processes $<p_1, p_2, p_3, \ldots, p_n>$:

  global history: $H = \cup_i (h_i)$

  global state: $S = \cup_i (s_i)$

More notations and definitions

- For a process p_i , where events $e_i^0, e_i^1, \ldots$ occur:

  history(p_i) = $h_i = \langle e_i^0, e_i^1, \ldots \rangle$

  prefix history(p_i^k) = $h_i^k = \langle e_i^0, e_i^1, \ldots, e_i^k \rangle$

  s_i^k : p_i's state immediately after k^{th} event.

- For a set of processes $\langle p_1, p_2, p_3, \ldots, p_n \rangle$:

  global history: $H = \cup_i (h_i)$

  global state: $S = \cup_i (s_i)$

  But state at what time instant?

More notations and definitions

- For a process p_i , where events e_i^0, e_i^1, … occur:

  history(p_i) = $h_i = <e_i^0, e_i^1, \ldots >$

  prefix history(p_i^k) = $h_i^k = <e_i^0, e_i^1, \ldots, e_i^k >$

  s_i^k : p_i's state immediately after k^{th} event.

- For a set of processes $<p_1, p_2, p_3, \ldots, p_n>$:

  global history: $H = \cup_i (h_i)$

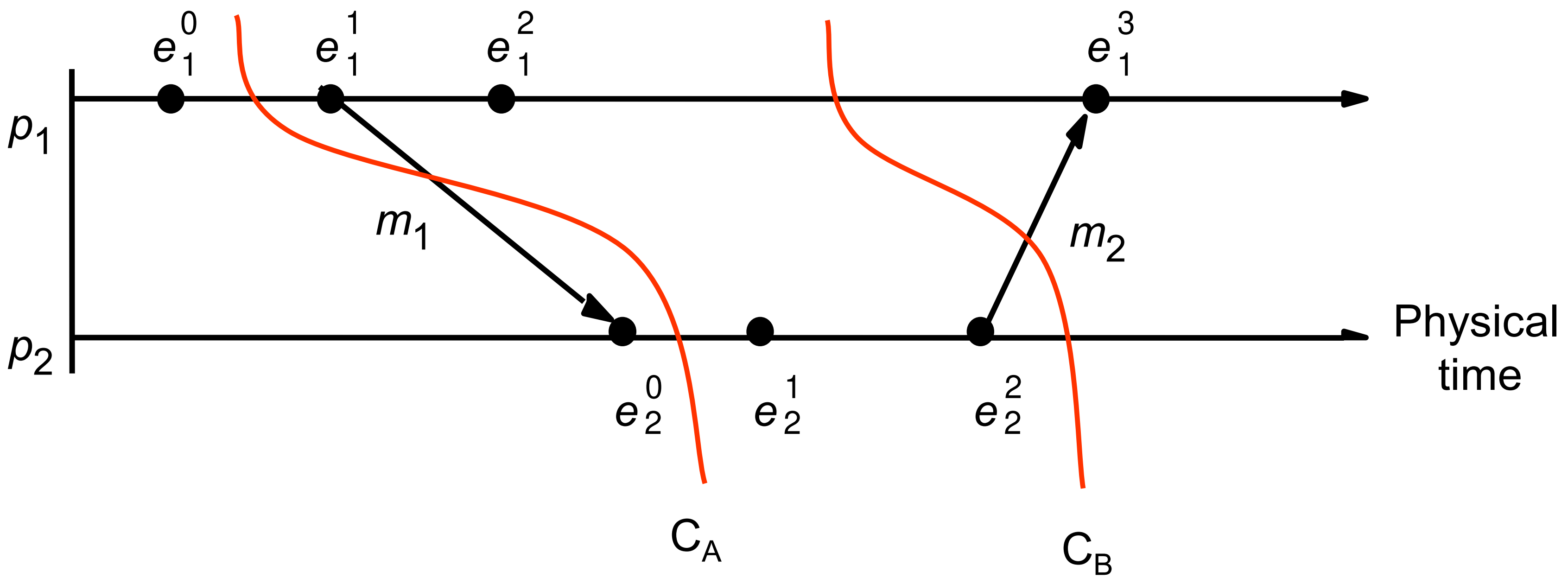
  global state: $S = \cup_i (s_i^{k_i})$

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

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

  global state S that corresponds to cut $C = \cup_i (s_i^{c_i})$

Example: Cut



C_A: < e_1^0, e_2^0>

Frontier of C_A:

C_B: < e_1^0, e_1^1, e_1^2, e_2^0, e_2^1 e_2^2 >

Frontier of C_B:

More notations and definitions

- For a process p_i , where events e_i^0, e_i^1, … occur:

  history(p_i) = h_i = $<e_i^0, e_i^1, \ldots >$

  prefix history(p_i^k) = h_i^k = $<e_i^0, e_i^1, \ldots, e_i^k >$

  s_i^k : p_i's state immediately after k^{th} event.

- For a set of processes $<p_1, p_2, p_3, \ldots., p_n>$:

  global history: $H = \cup_i (h_i)$

  global state: $S = \cup_i (s_i^{k_i})$

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

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

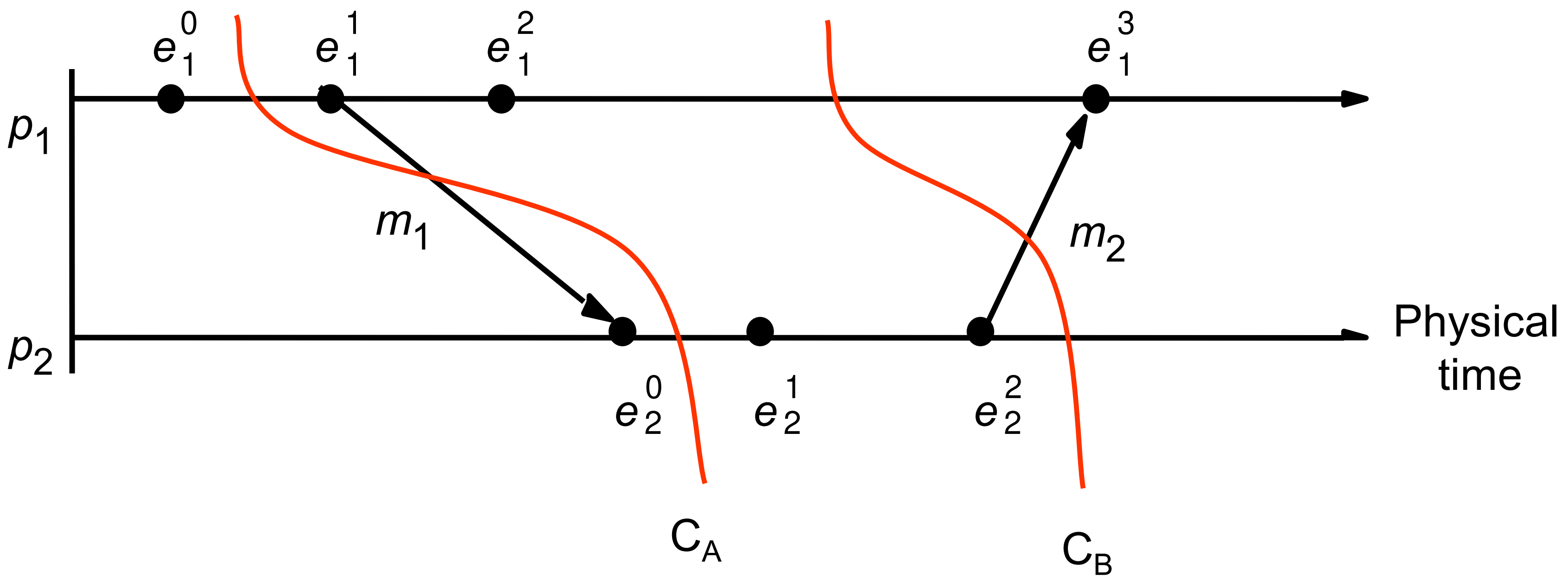
  global state S that corresponds to cut C = $\cup_i (s_i^{c_i})$

Consistent cuts and snapshots

- A cut C is consistent if and only if

$$\forall e \in C \text{ (if } f \rightarrow e \text{ then } f \in C)$$

Example: Cut



C_A: < e_1^0, e_2^0>
Frontier of C_A: {e_1^0, e_2^0}
Inconsistent cut.

C_B: < e_1^0, e_1^1, e_1^2, e_2^0, e_2^1 e_2^2 >
Frontier of C_B: {e_1^2, e_2^2}
Consistent cut.

Consistent cuts and snapshots

- A cut C is consistent if and only if

$$\forall e \in C \text{ (if } f \rightarrow e \text{ then } f \in C)$$

- A global state S is consistent if and only if it corresponds to a consistent cut.

Consistent cuts and snapshots

- A cut C is consistent if and only if

$$\forall e \in C \text{ (if } f \rightarrow e \text{ then } f \in C\text{)}$$

- A global state S is consistent if and only if it corresponds to a consistent cut.

- *How do we find consistent global states?*

Chandy-Lamport Algorithm

- Goal:
  - Record a global snapshot
    - Process state (and channel state) for a set of processes.
  - The recorded global state is consistent.
- Identifies a consistent cut.
- Records corresponding state locally at each process.

Chandy-Lamport Algorithm

- *System model and assumptions:*
  - System of n processes: <p_1, p_2, p_3, …., p_n>.
  - There are two uni-directional communication channels between each ordered process pair : p_j to p_i and p_i to p_j.
  - Communication channels are FIFO-ordered (first in first out).
  - All messages arrive intact, and are not duplicated.
  - No failures: neither channel nor processes fail.

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

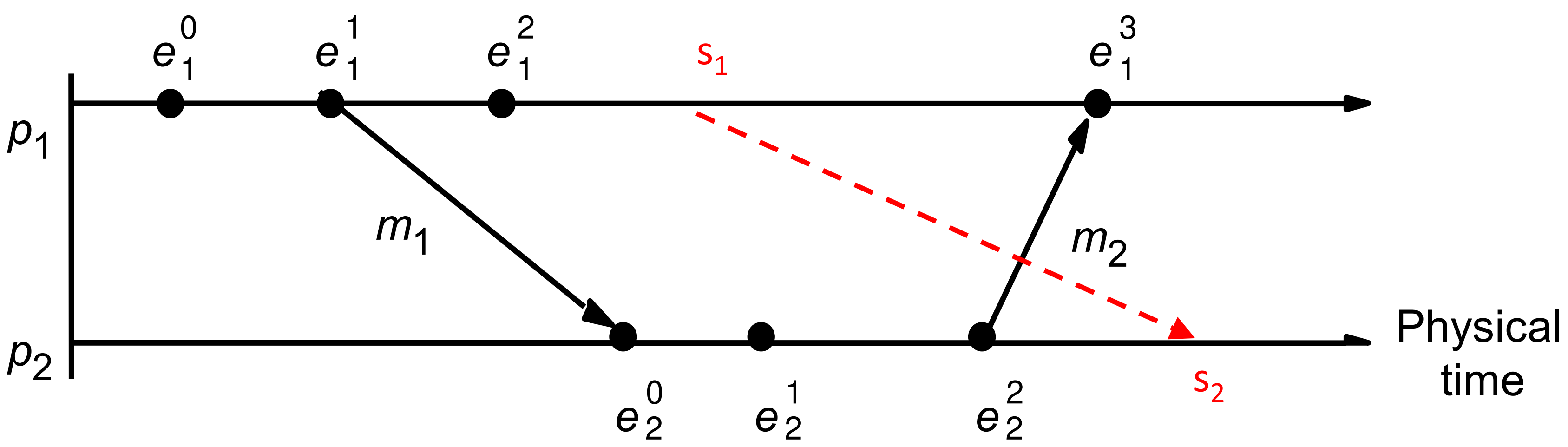
Chandy-Lamport Algorithm Intuition

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

.

- When a process receives a marker.
  - records its own state.

Chandy-Lamport Algorithm Intuition

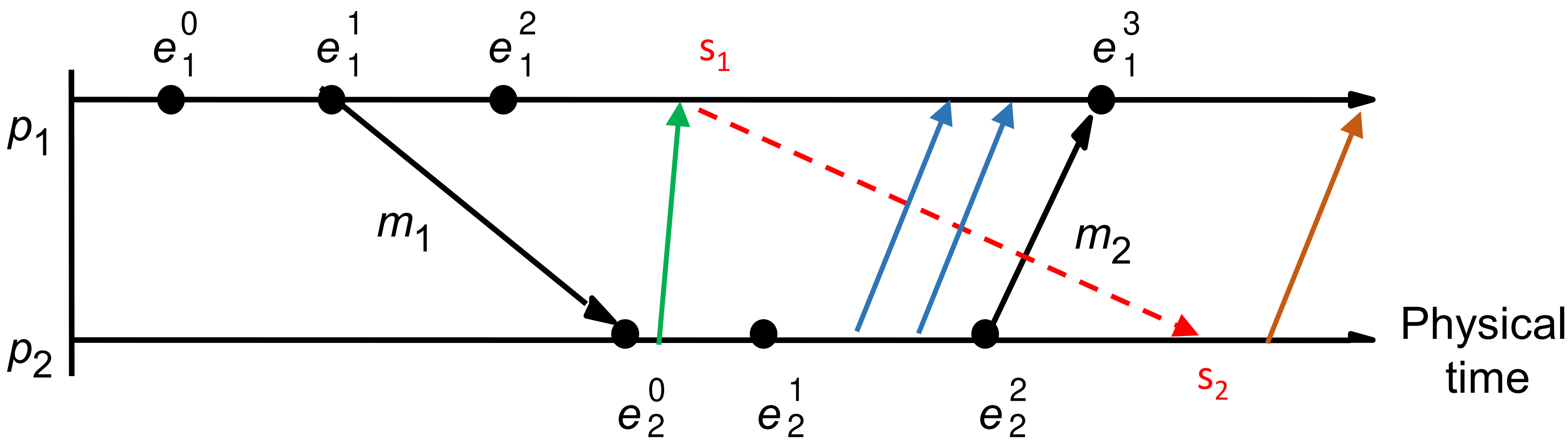


Cut frontier: $\{e_1{}^2, e_2{}^2\}$

Process, state, events

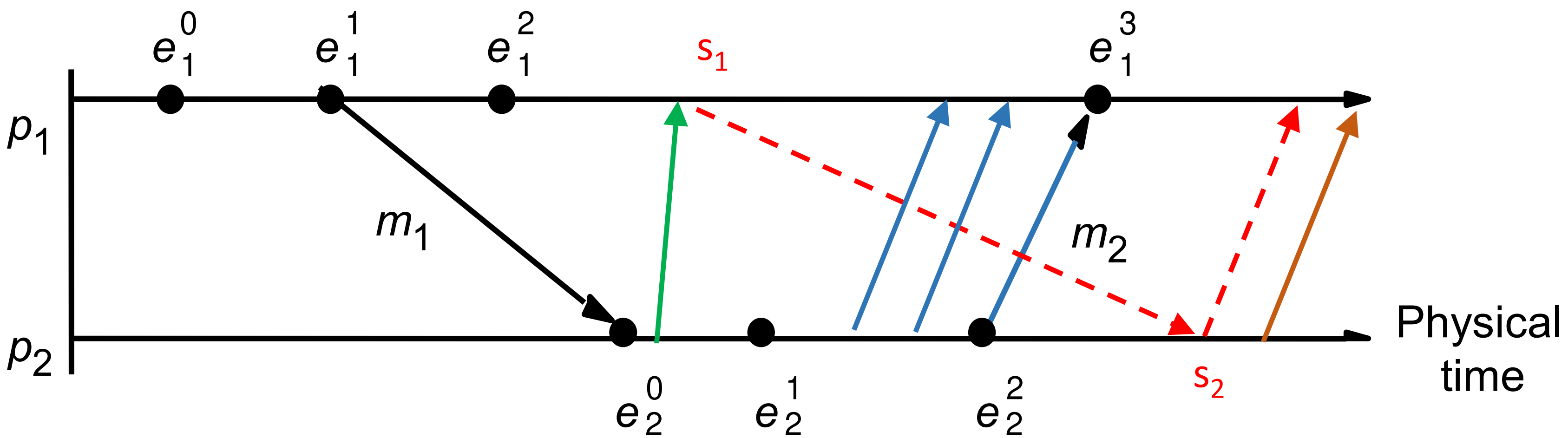
- Consider a system with n processes: <p_1, p_2, p_3, …., p_n>.
- Each process p_i is associated with *state* s_i.
  - State includes values of all local variables, affected files, etc.
- Each channel can also be associated with a state.
  - Which messages are currently *pending* on the channel.
  - Can be computed from process' state:
    - Record when a process sends and receives messages.
    - if p_i sends a message that p_j has not yet received, it is pending on the channel.
- State of a process (or a channel) gets transformed when an *event* occurs. 3 types of events:
  - local computation, sending a message, receiving a message.

Chandy-Lamport Algorithm Intuition



Cut frontier: $\{e_1^2, e_2^2\}$

Chandy-Lamport Algorithm Intuition



Cut frontier: $\{e_1^2, e_2^2\}$

Chandy-Lamport Algorithm Intuition

- First, initiator p_i:
  - records its own state.
  - creates a special marker message.
  - sends the marker to all other process.
  - start recording messages received on other channels.
    - until a marker is received on a channel.
- When a process receives a marker.
  - If marker is received for the first time.
    - records its own state.
    - sends marker on all other channels.
    - start recording messages received on other channels.
      - until a marker is received on a channel.