Distributed Systems

CS425/ECE428

02/05/2020

Logistics

- Campus cluster access for MP0:
  - All requests sent by last Friday should have received access.
  - Other requests are getting processed.
  - If you have any specific concerns that have not yet been addressed, please see Prof. Borisov after class.

- Please reach out if you require any DRES related accommodations.

Today's agenda

- Wrap up logical clocks
  - Chapter 14.4
- Global states and snapshots
  - Chapter 14.5

Recap from last class: clock synchronization

- Cristian Algorithm
  - Synchronization between a client and a server.
  - Synchronization bound = $(T_{round} / 2) - min \leq T_{round} / 2$
- Berkeley Algorithm
  - Internal synchronization between clocks.
  - A central server picks the average time and disseminates offsets.
- Network Time Protocol
  - Hierarchical time synchronization over the Internet.
  - Symmetric mode synchronization for better accuracy.

Event Ordering

- A usecase of synchronized clocks:
  - Reasoning about order of events.
- Can we reason about order of events without synchronized clocks?

Event Ordering

- Easy to order events within a single process $\mathbf{p_i}$, based on their time of occurrence.

- How do we reason about events across processes?
  - A message must be *sent* before it gets *received* at another process.

- These two notions help define *happened-before* (HB) relationship denoted by $\rightarrow$.
  - $\mathbf{e} \rightarrow \mathbf{e'}$ means $\mathbf{e}$ *happened before* $\mathbf{e'}$.

Happened-Before Relationship

- *Happened-before* (HB) relationship denoted by →.
  - **e** → **e'** means **e** *happened before* **e'**.
  - **e** $\rightarrow_i$ **e'** means **e** *happened before* **e'**, as observed by p_i.

- HB rules:
  - If ∃ p_i , **e** $\rightarrow_i$ **e'** then **e** → **e'**.
  - For any message m, **send(m)** → **receive(m)**
  - If **e** → **e'** and **e'** → **e''** then **e** → **e''**

- Also called *"potentially causal"* ordering.

Lamport's Logical Clock

- Logical timestamp for each event that captures the *happened-before* relationship.

- *Algorithm:* Each process $\mathbf{p_i}$
  1. initializes local clock $\mathbf{L_i = 0}$.
  2. increments $\mathbf{L_i}$ before timestamping each event.
  3. piggybacks $\mathbf{L_i}$ when sending a message.
  4. upon receiving a message with clock value $\mathbf{t}$
     - sets $\mathbf{L_i = max(t, L_i)}$
     - increments $\mathbf{L_i}$ (as per point 2).

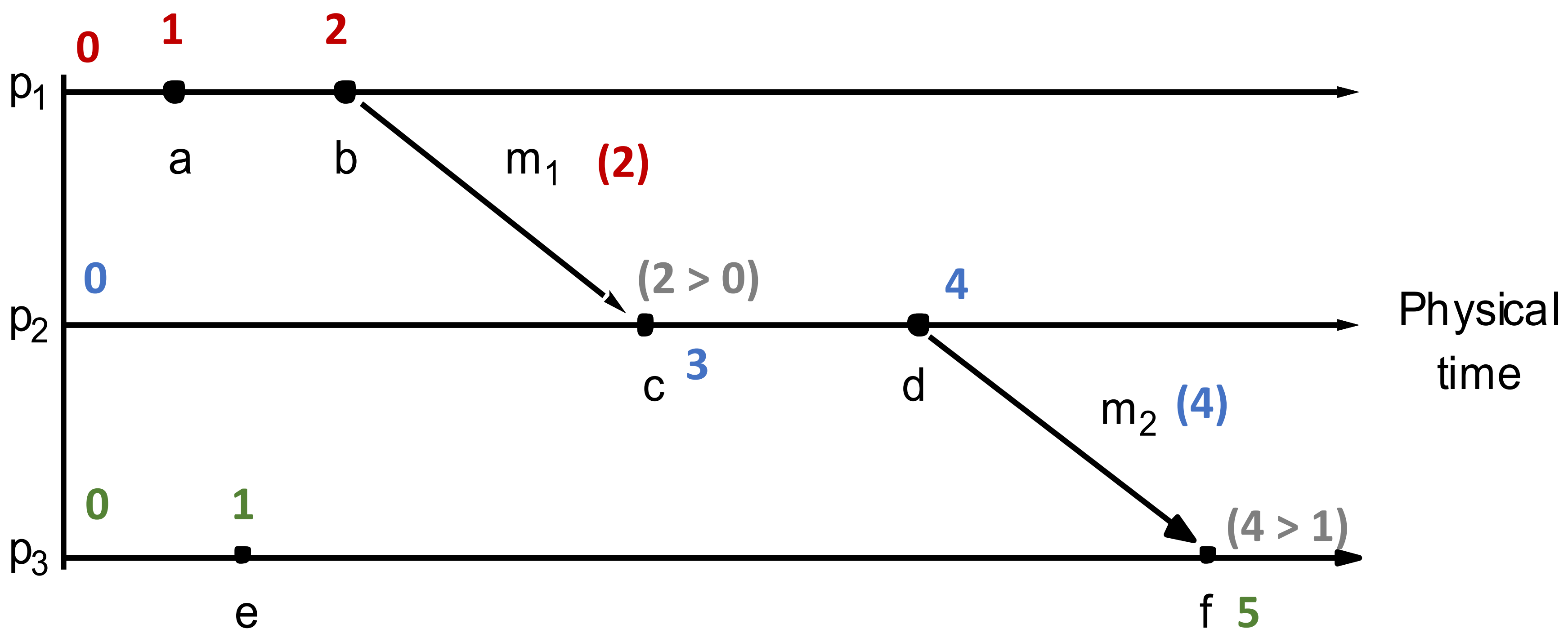
Lamport's Logical Clock

- Logical timestamp for each event that captures the *happened-before* relationship.

- If **e** $\rightarrow$ **e'** then **L(e) < L(e')**

- What if **L(e) < L(e')**?
  - We cannot say that **e** $\rightarrow$ **e'**
  - We can say: **e'** $\nrightarrow$ **e**
  - Either **e** $\rightarrow$ **e'** or **e || e'**

Logical Timestamps: Example



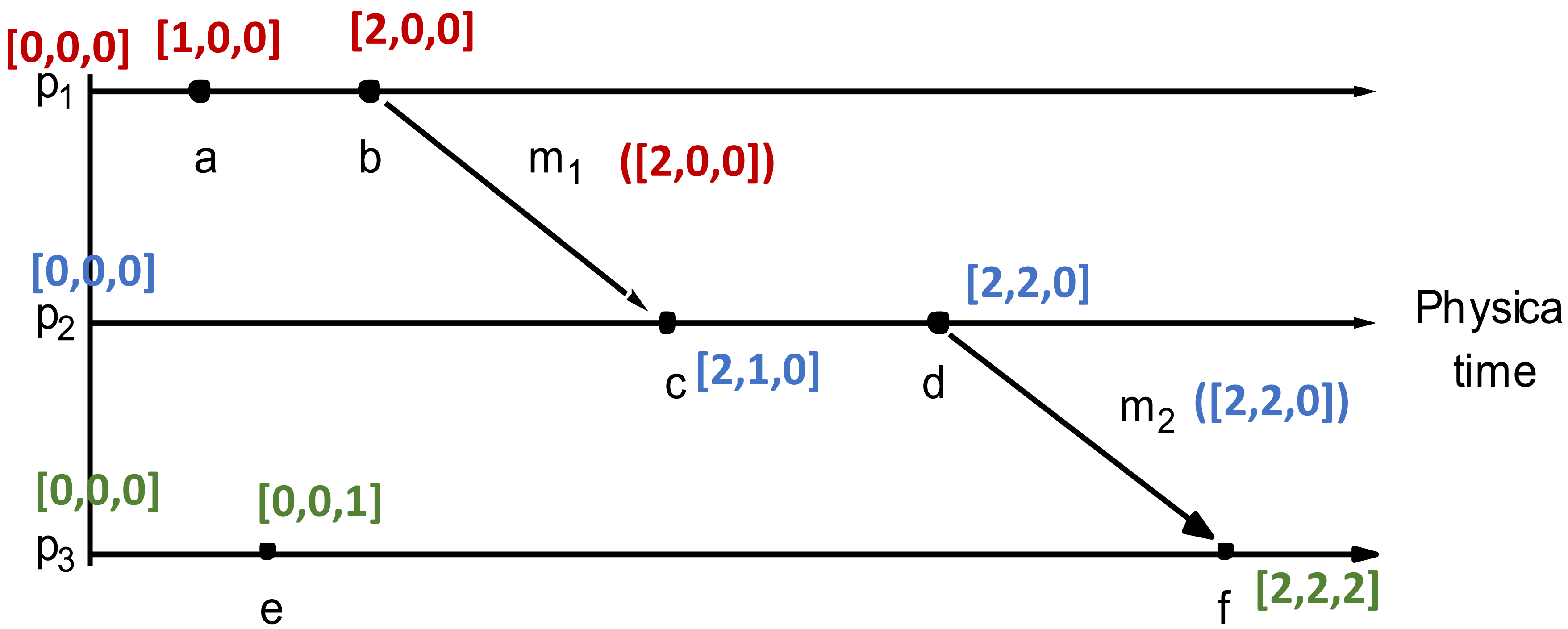
$L(e) < L(d)$, $e \parallel d$ $L(e) < L(f)$, $e \rightarrow f$

Vector Clocks

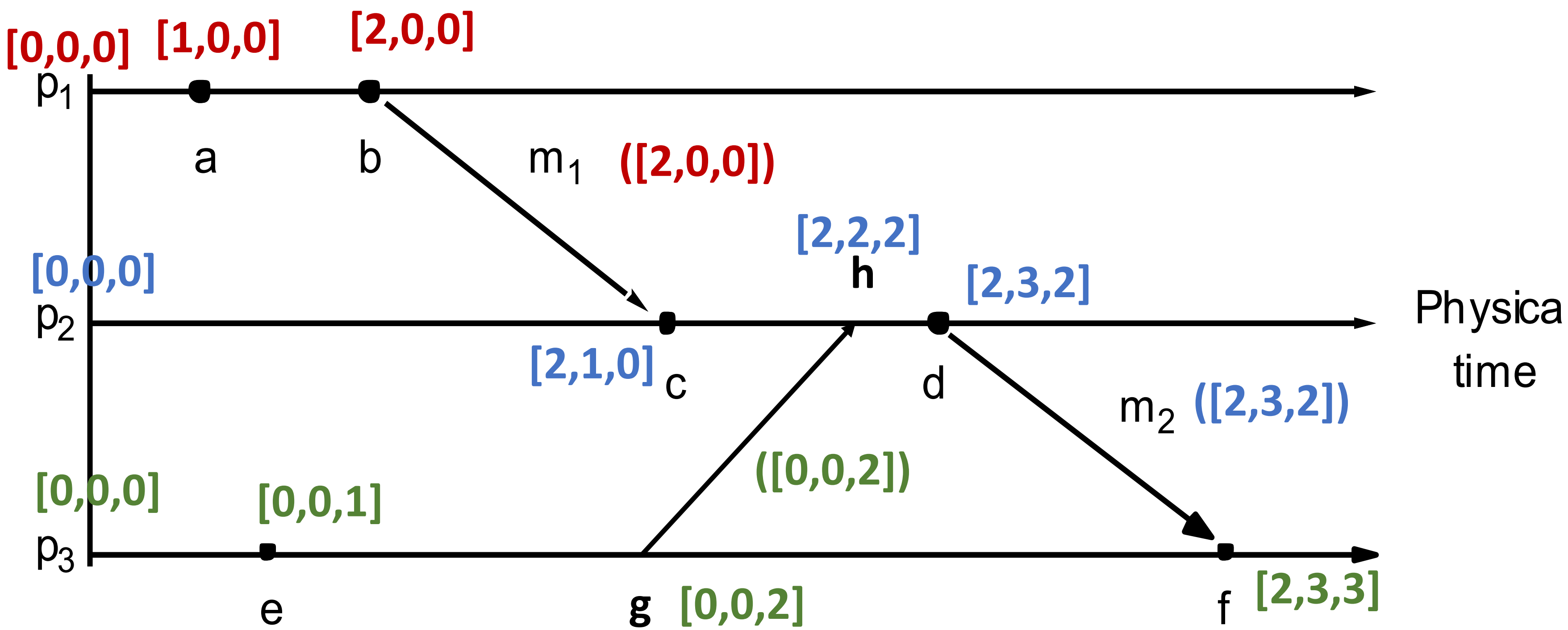
- Each event associated with a vector timestamp.
- Each process maintains vector of clocks $\mathbf{V_i}$
  - $\mathbf{V_i[j]}$ is the clock for process $\mathbf{p_j}$

- Algorithm: each process $\mathbf{p_i}$:
  1. initializes local clock $\mathbf{V_i[j] = 0}$
  2. increments $\mathbf{V_i[i]}$ before timestamping each event.
  3. piggybacks $\mathbf{V_i}$ when sending a message.
  4. upon receiving a message with clock value $\mathbf{t}$
     - sets $\mathbf{V_i[j] = max(V_i[j], t[j])}$ for all j=1…n.
     - increments $\mathbf{V_i[i]}$ (as per point 2).

Vector Timestamps: Example



Vector Timestamps: Example



Comparing Vector Timestamps

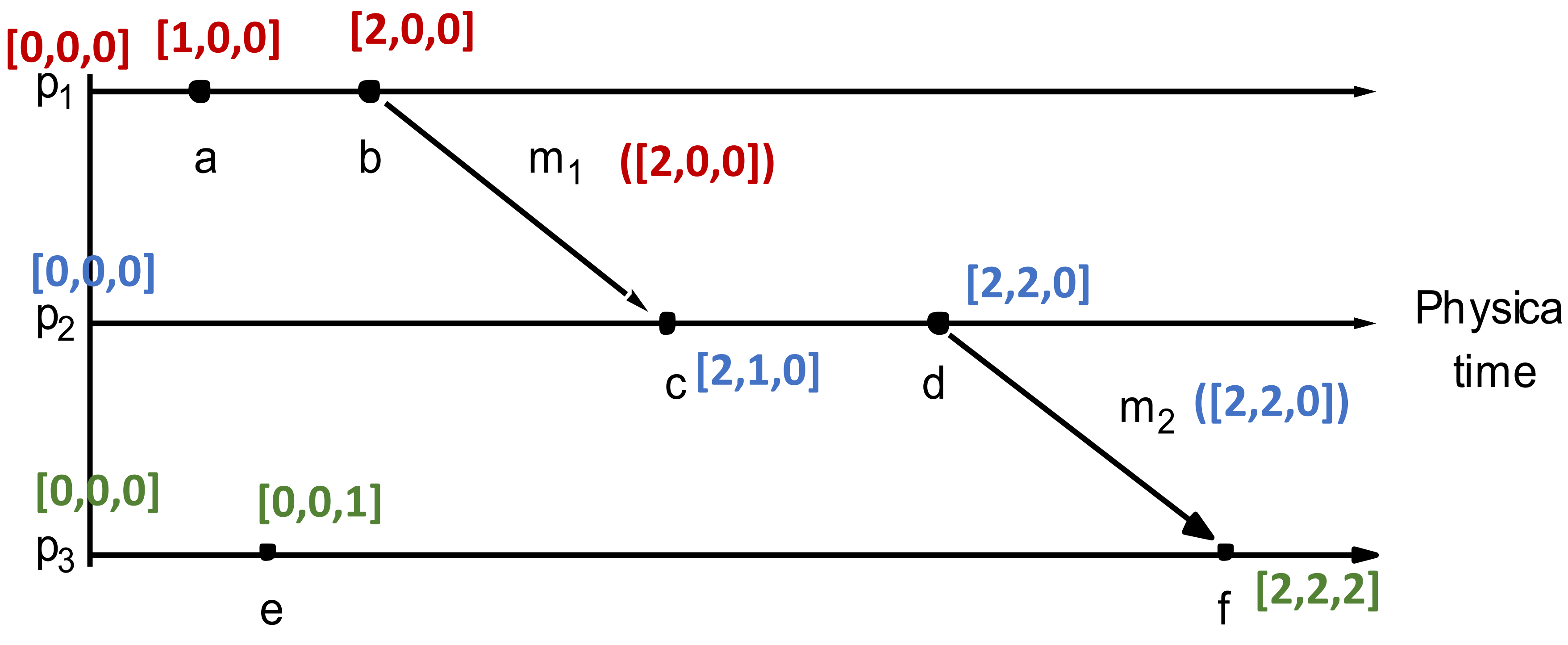
- Let $V(e) = V$ and $V(e') = V'$

- $V = V'$, iff $V[i] = V'[i]$, for all $i = 1, \ldots, n$
- $V \leq V'$, iff $V[i] \leq V'[i]$, for all $i = 1, \ldots, n$
- $V < V'$, iff $V \leq V'$ & $V \neq V'$

  iff $V \leq V'$ & $\exists$ j such that $(V[j] < V'[j])$

- $e \rightarrow e'$ iff $V < V'$
  - ($V < V'$ implies $e \rightarrow e'$) and ($e \rightarrow e'$ implies $V < V'$)
- $e \,||\, e'$ iff ($V \not< V'$ and $V' \not< V$)

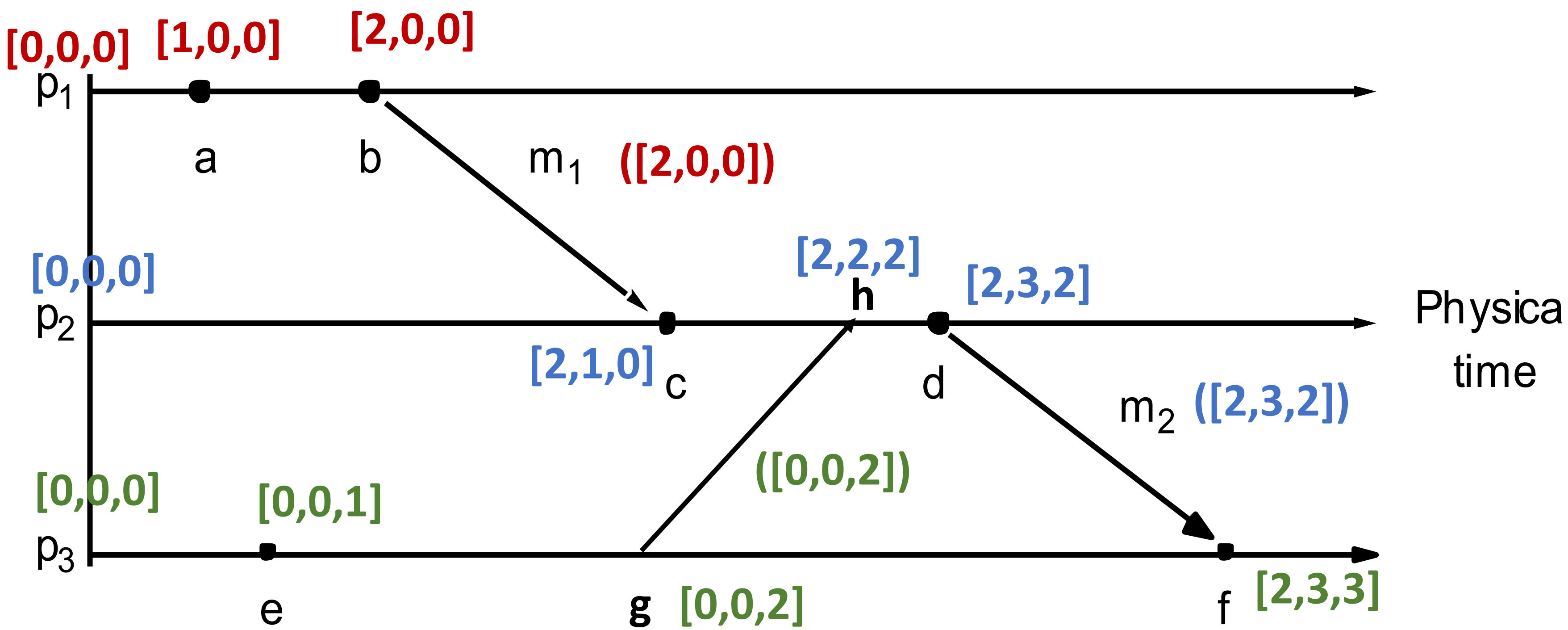
Vector Timestamps: Example



$V(e) < V(f)$, $e \rightarrow f$

$V(e) \not< V(d)$ and $V(d) \not< V(e)$, $e \parallel d$

Vector Timestamps: Example



$V(e) < V(f)$, $e \rightarrow f$

$V(e) < V(d)$, $e \rightarrow d$

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.

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.

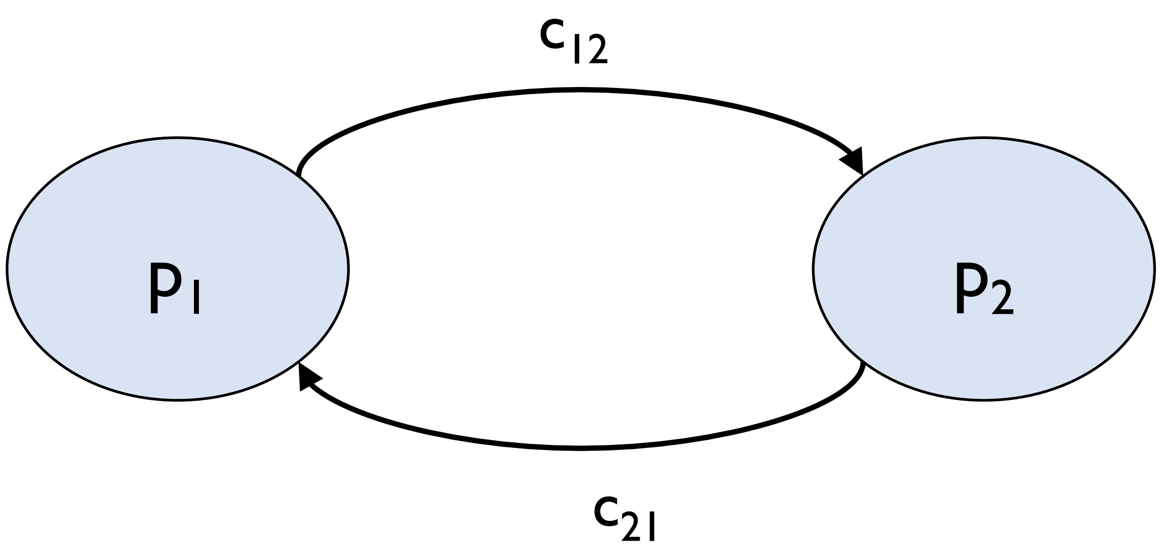
Process, state, events

- Consider a system with **n** processes: $<p_1, p_2, p_3, \ldots., p_n>$.
- Each process p_i is associated with *state* $\mathbf{s_i}$.
  - 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:



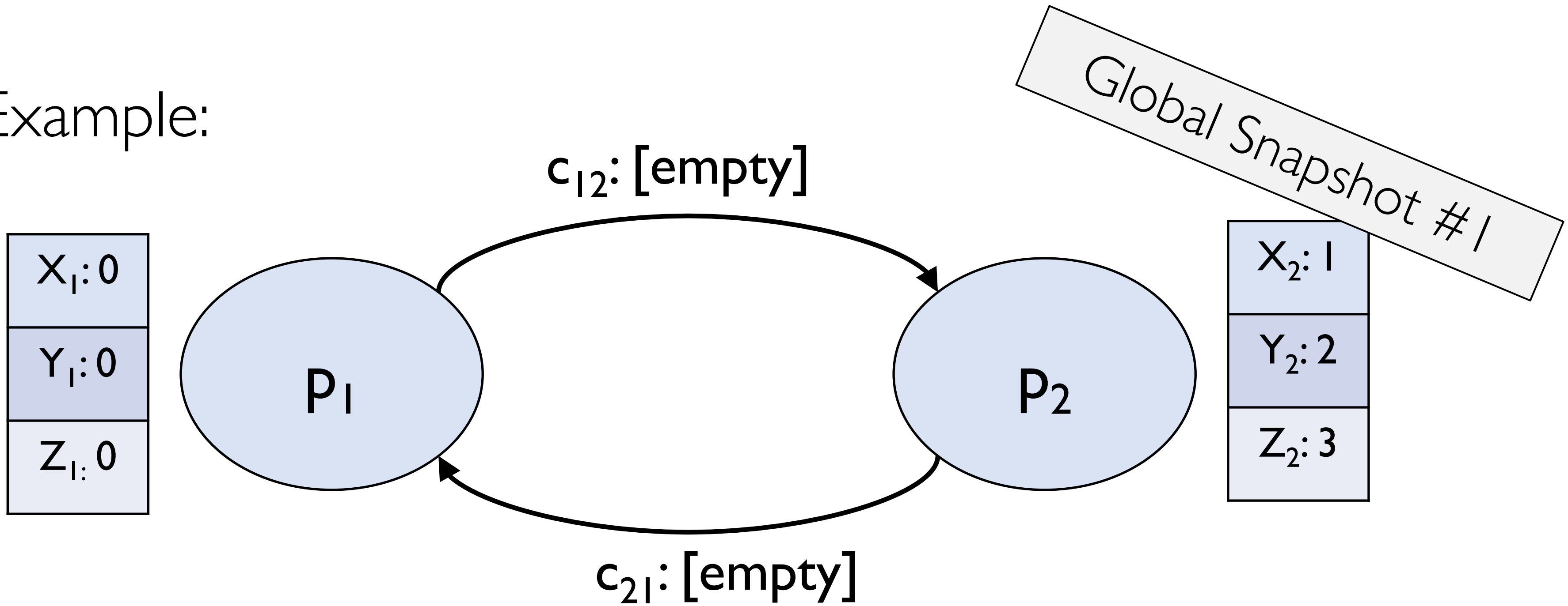
Two processes: p_1 and p_2.

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.

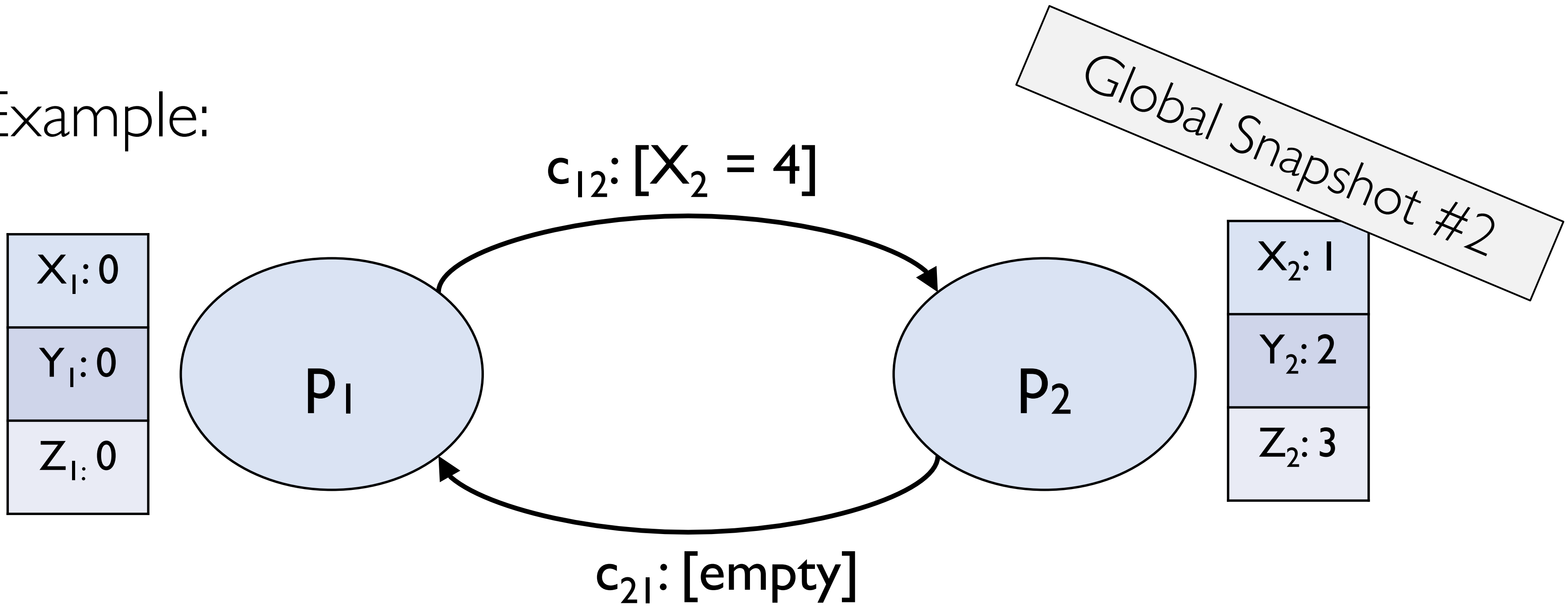
- Example:



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:

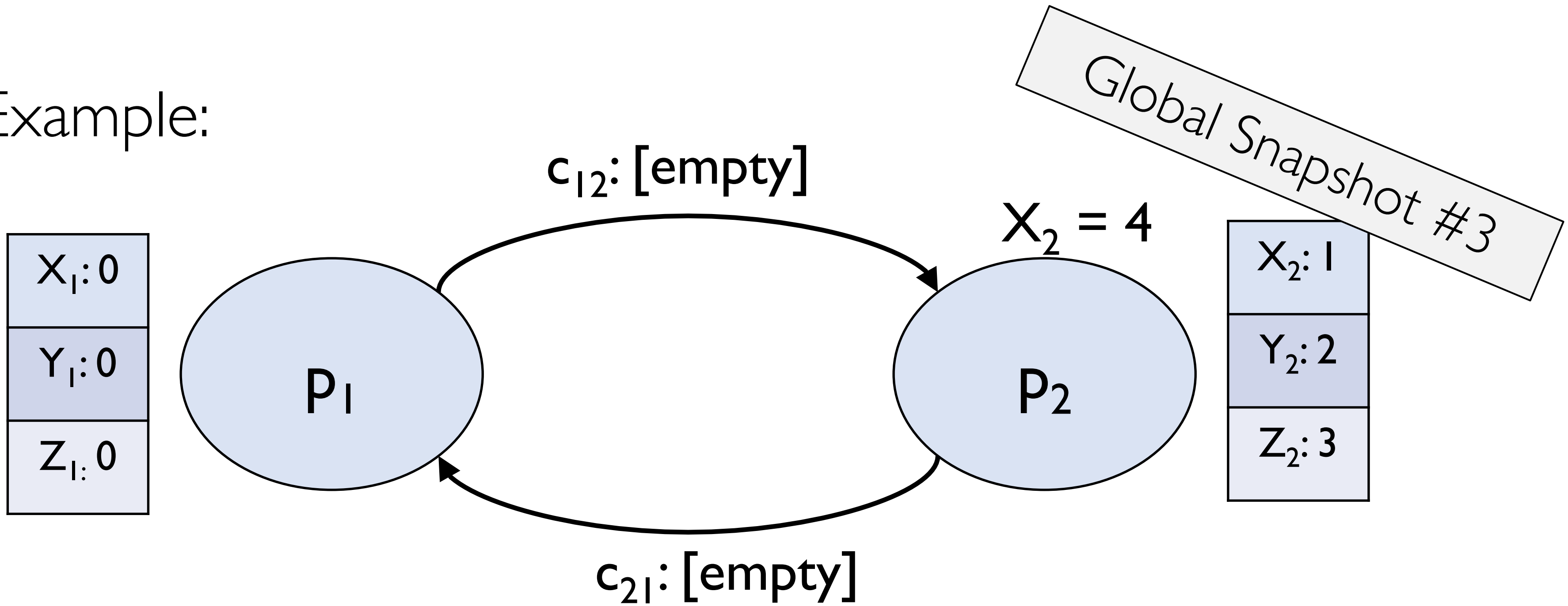


p_1 send a message to p_2 asking it to set $X_2 = 4$

Global State (or Global Snapshot)

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

- Example:

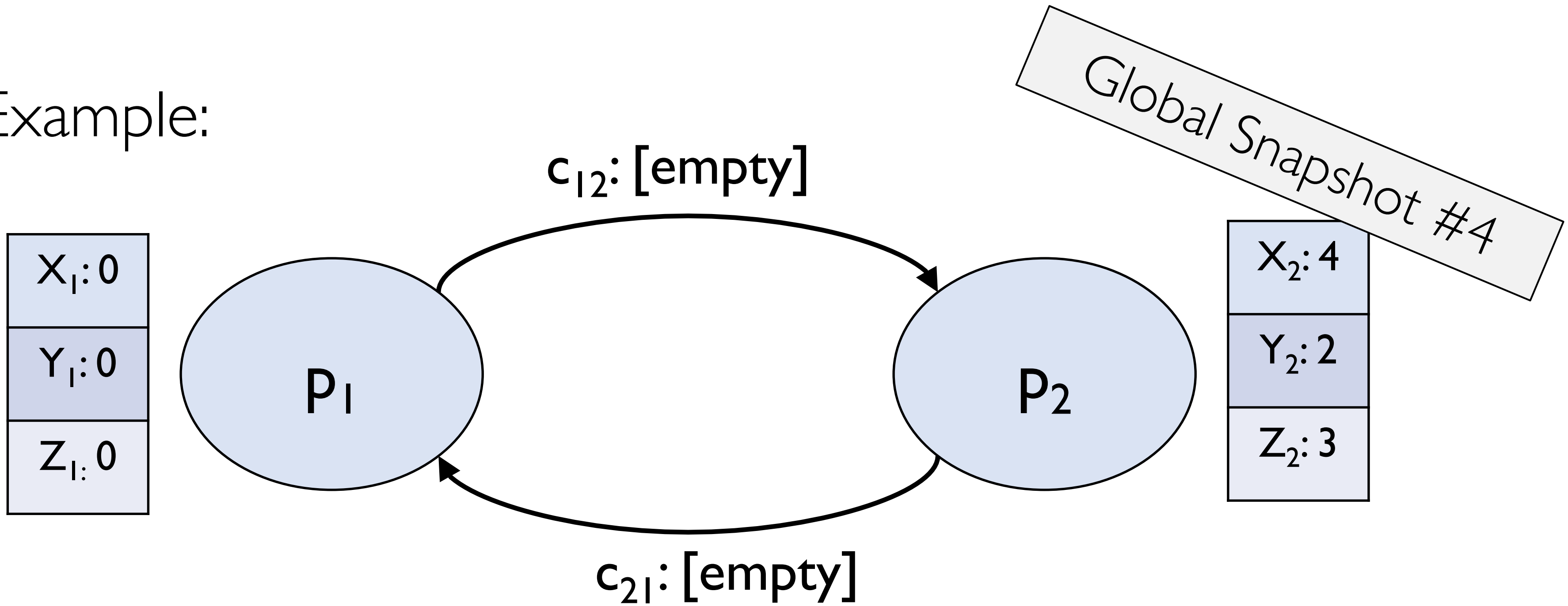


p_2 receives the message.

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?**

Some more notations and definitions

- Consider a system with **n** processes: $<p_1, p_2, p_3, \ldots, p_n>$.
- Each process p_i is associated with *state* $\mathbf{s_i}$.
  - 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.

Some more notations and definitions

- Consider a system with **n** processes: $<p_1, p_2, p_3, \ldots, p_n>$.
- Each process p_i is associated with *state* $\mathbf{s_i}$.
  - 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. e_i^j is the j^{th} event at p_i. 3 types of events:
  - local computation, sending a message, receiving a message.

Some 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: $S = \cup_i (s_i)$

Some 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?

Some 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: $H = \cup_i (h_i)$

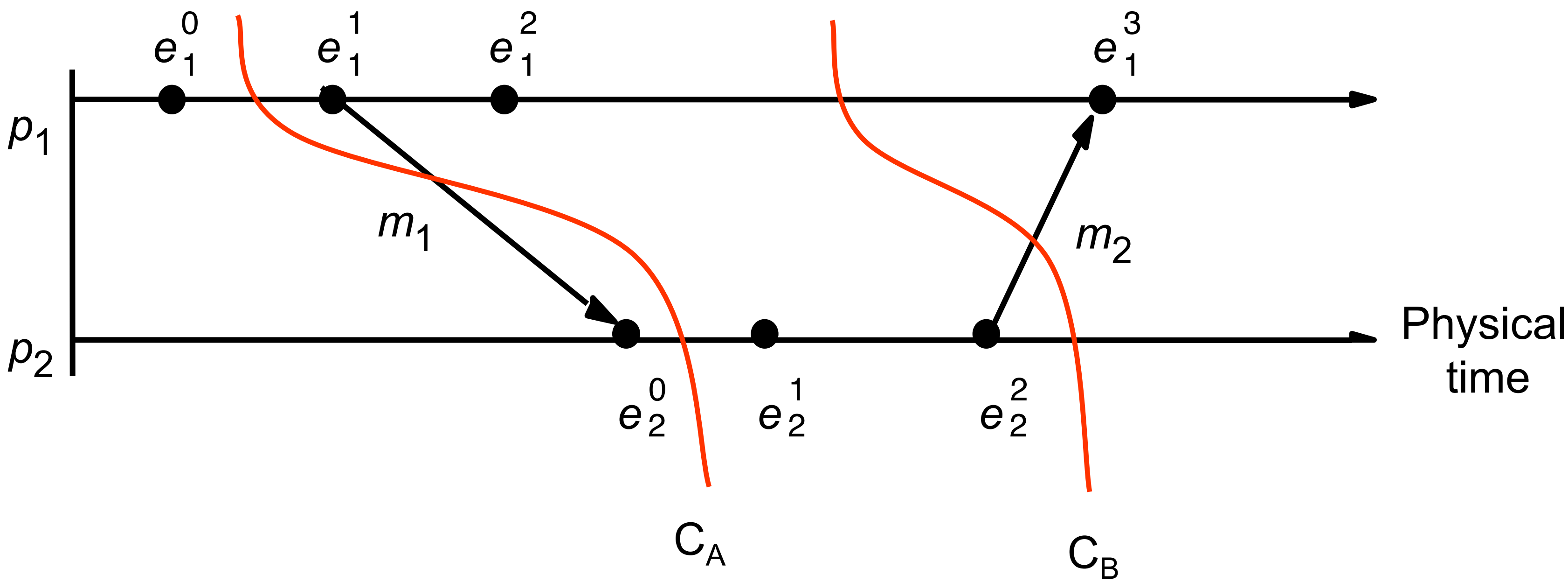
  global state: $S = \cup_i (s_i^{k_i})$

  a cut $C \subseteq H = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_3}$

  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: {e_1^0, e_2^0}

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}

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

  a cut $C \subseteq H = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_3}$

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

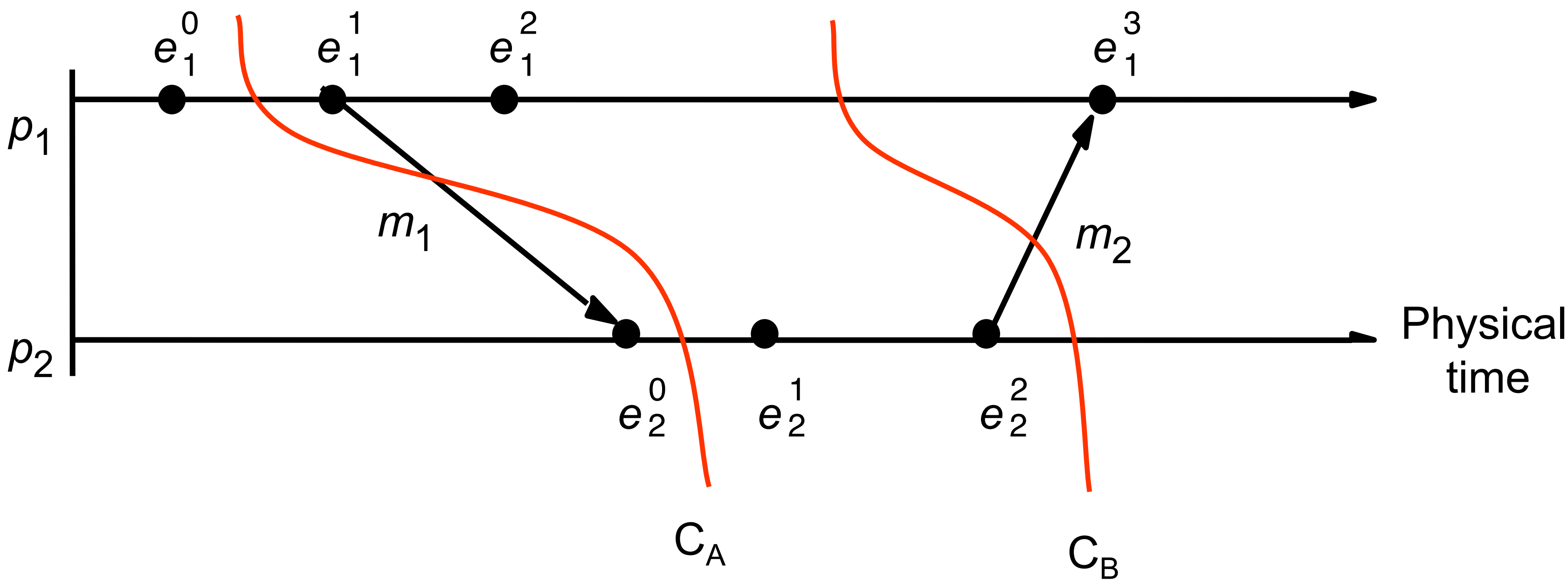
  global state S that corresponds to cut $C = \cup_i (s_i^{c_i})$

Consistent cuts and snapshots

- A cut **C** is **consistent** if and only if

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

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.

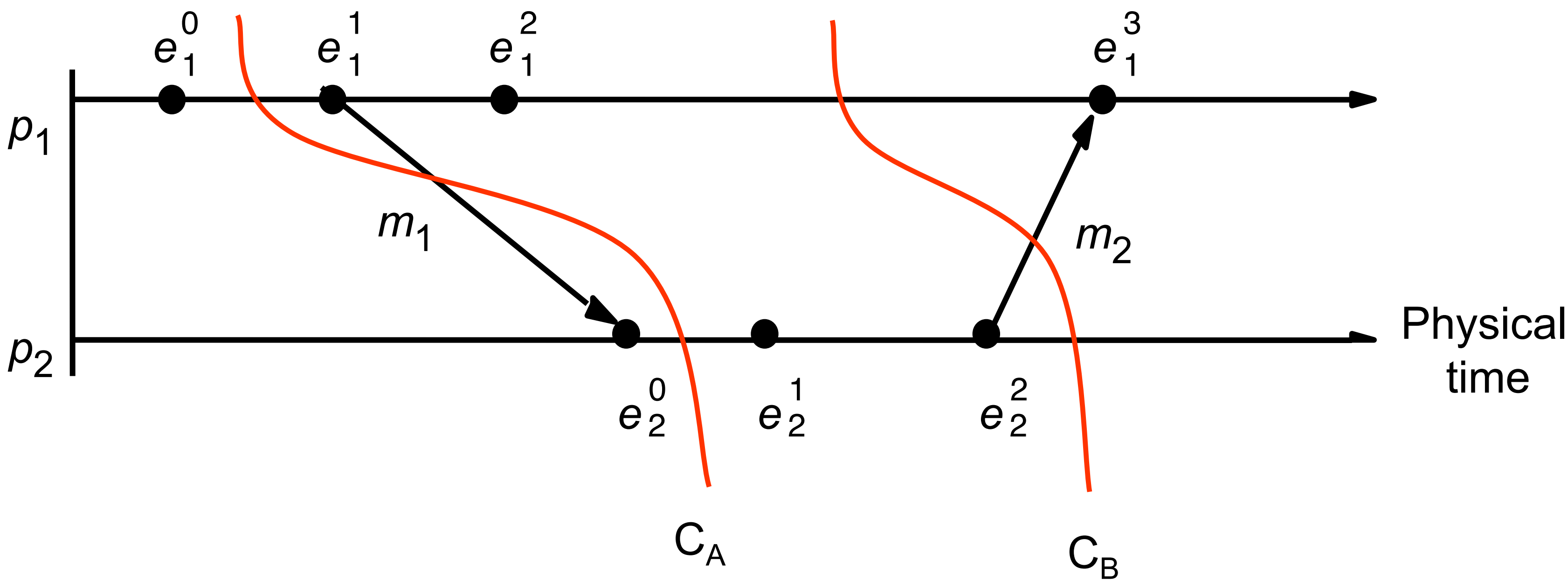
Consistent cuts and snapshots

- A cut **C** is **consistent** if and only if

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

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

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

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

- 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
    - Set of 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.

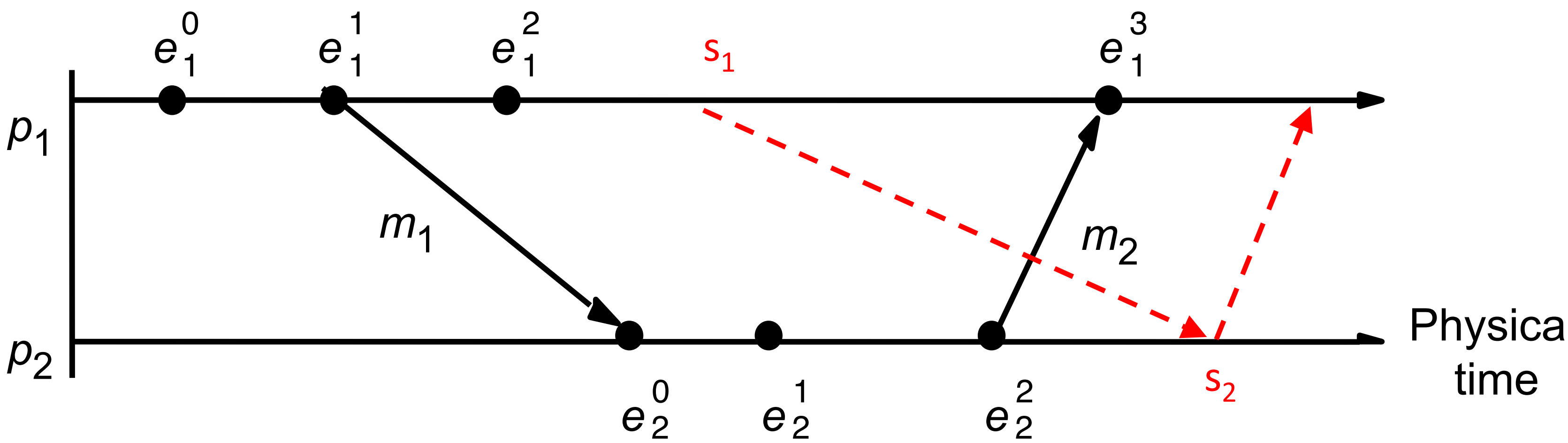
- When a process receives a **marker**.
  - records its own state.

Chandy-Lamport Algorithm Intuition

- 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.

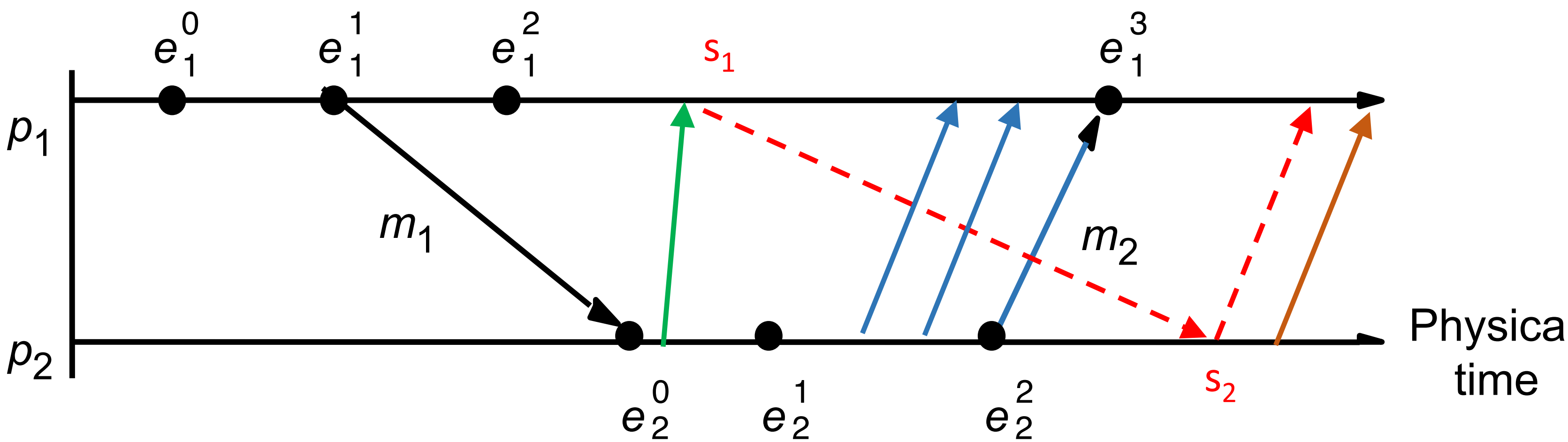
- When a process receives a **marker**.
  - records its own state.

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.

Chandy-Lamport Algorithm

- First, initiator $\mathbf{p_i}$:
  - records its own state.
  - creates a special **marker** message.
  - for $j=1$ *to* n except i
    - $\mathbf{p_i}$ sends a **marker** message on outgoing channel $\mathbf{c}_{ij}$
    - starts recording the incoming messages on each of the incoming channels at $\mathbf{p_i}$: $\mathbf{c}_{ji}$ (for $j=1$ *to* n except i).

Chandy-Lamport Algorithm

Whenever a process $\mathbf{p_i}$ receives a **marker** message on an incoming channel $\mathbf{c_{ki}}$

- if (this is the first **marker** $\mathbf{p_i}$ is seeing)
  - $\mathbf{p_i}$ records its own state first
  - marks the state of channel $\mathbf{c_{ki}}$ as "empty"
  - for j=1 to n except i
    - $\mathbf{p_i}$ sends out a **marker** message on outgoing channel $\mathbf{c_{ij}}$
  - starts recording the incoming messages on each of the incoming channels at $\mathbf{p_i}$: $\mathbf{c_{ji}}$ (for *j=1 to n* except *i and k*).
- else // already seen a **marker** message
  - mark the state of channel $\mathbf{c_{ki}}$ as all the messages that have arrived on it since recording was turned on for $\mathbf{c_{ki}}$

Chandy-Lamport 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
  - To record the state of all channels

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).