SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

CMPT 431 Distributed Systems

Fall 2019

Model of Distributed Computations

https://www.cs.sfu.ca/~keval/teaching/cmpt431/fall19/

Instructor: Keval Vora

Model of Distributed Computations

- Recall assumptions in distributed computing (processes are autonomous, no global clock, no shared memory, no direct synchornization)
- Analyzing and designing distributed systems is challenging!

- We will study how to reason about a distributed system
- Theoretical foundations of distributed computing
- Similar to "reasoning correctness" portion in this course

Reading

- [DC] Chapter 2

Distributed Program

- A distributed program consists of n asynchronous processes: $p_1, p_2, \ldots, p_i, \ldots, p_n$
- Processes do not share a global memory
  - Communicate solely by passing messages
- Processes do not share a global clock
- Assume each process is running on a different processor

Distributed Program

- Process execution and message transfer are asynchronous
- C_{ij} : channel from process pi to process pj
- m_{ij} : a message sent by p_i to p_j
- Message transmission delay is finite and unpredictable

Distributed Execution Model

- Execution of a process consists of a sequential execution of its events (or actions)
  - Internal event
  - Message send
  - Message receive
- Events are atomic (indivisible and instantaneous)

Distributed Execution Model

- Events change states of respective processes and channels
  - Internal event changes the state of the process at which it occurs
  - Send event changes the state of the process that sends the message and the state of the channel on which the message is sent
  - Receive event changes the state of the process that receives the message and the state of the channel on which the message is received

Distributed Execution Model

- Linear ordering among events at a process
- Process p_i produces a sequence of events e_i^1, e_i^2, ...

- $H_i = (h_i, \rightarrow_i)$
  - h_i is the set of events produced by p_i
  - Binary relation $\rightarrow_i$ defines the linear order on events in h_i
- Relation $\rightarrow_i$ expresses causal dependencies

- Note: subscripts/superscripts dropped when context is clear

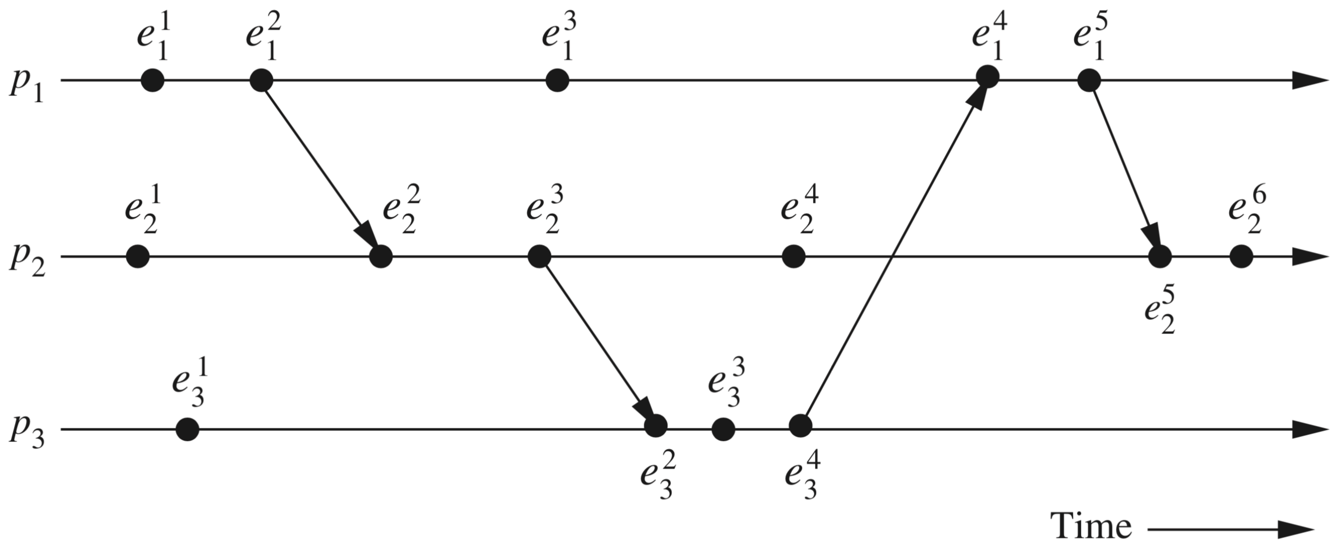
Distributed Execution Model

- For every message m

$$\text{send}(m) \rightarrow_{msg} \text{rec}(m)$$

- Relation $\rightarrow_{msg}$ captures causal dependency due to message exchange

Distributed Execution Model



- Our goal is to reason about how these events are related

Causal Precedence Relation

- Consider all process histories together

$$H = \cup_i h_i$$

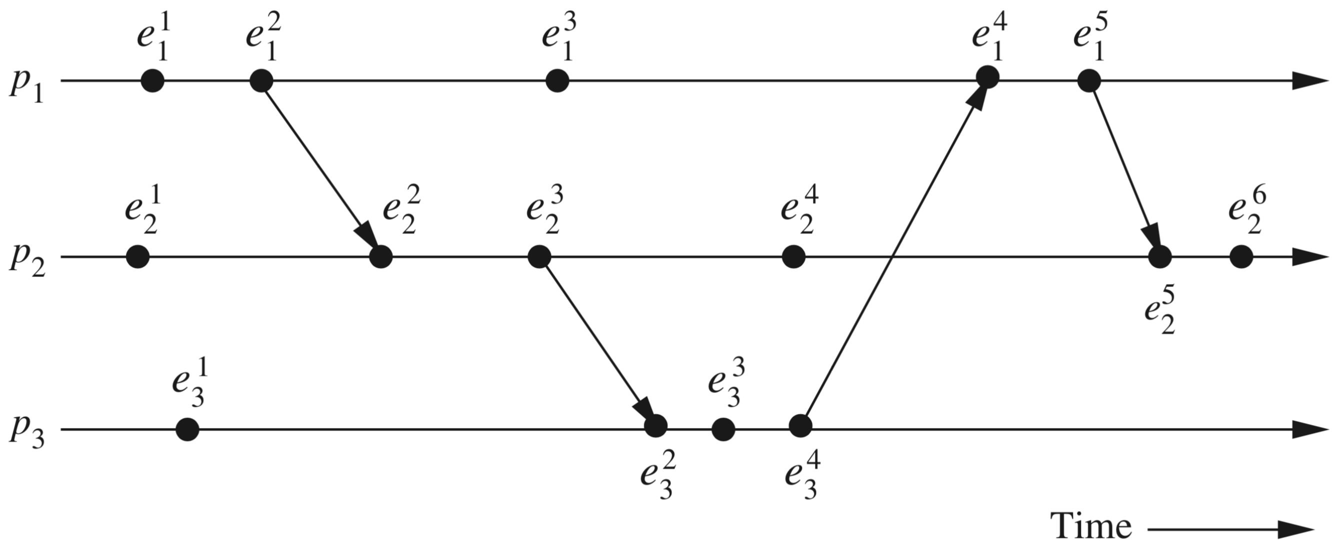
- Causal precedence relation: $\mathcal{H} = (H, \rightarrow)$

$$\forall e_i^x,\ \forall e_j^y \in H,\quad e_i^x \ \rightarrow\ e_j^y \quad \Leftrightarrow \quad \begin{cases} e_i^x \rightarrow_i e_j^y \text{ i.e., } (i = j) \wedge (x < y) \\ \text{or} \\ e_i^x \rightarrow_{msg} e_j^y \\ \text{or} \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \ \wedge\ e_k^z \rightarrow e_j^y \end{cases}$$

- Irreflexive partial order on the events of H

Causal Precedence Relation

- Irreflexive partial order on the events of H?



Causal Precedence Relation

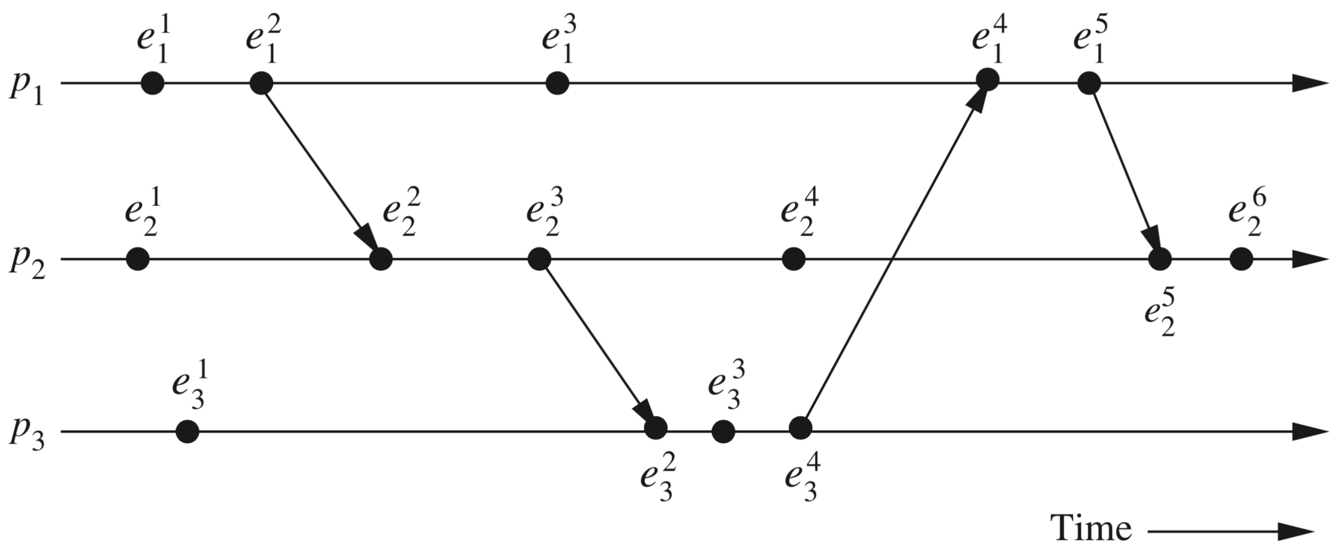
- Relation → is Lamport's "happens before" relation
- For any two events e_i and e_j, if $e_i \rightarrow e_j$, then event e_j is directly or transitively dependent on event e_i

- Transitive dependency: there exists e' such that e_i happens before e' and e' happens before e_j

Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System: https://lamport.azurewebsites.net/pubs/time-clocks.pdf

Causal Precedence Relation

- How to know if two events are related by → below?
  - Check if there is a directed path between two events



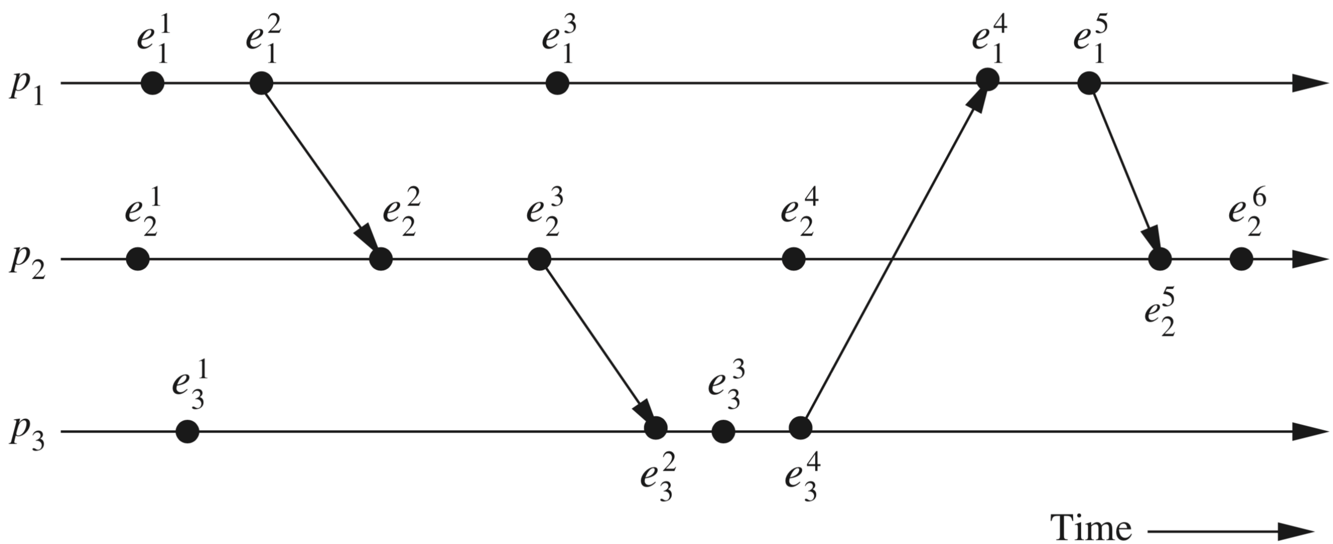
Causal Precedence Relation

- Relation $\rightarrow$ denotes flow of information in a distributed computation
- $e_i \rightarrow e_j$ means that all the information available at e_i is potentially accessible at e_j

- Powerful because now we can reason about behavior in terms of (global) information, progress, etc.

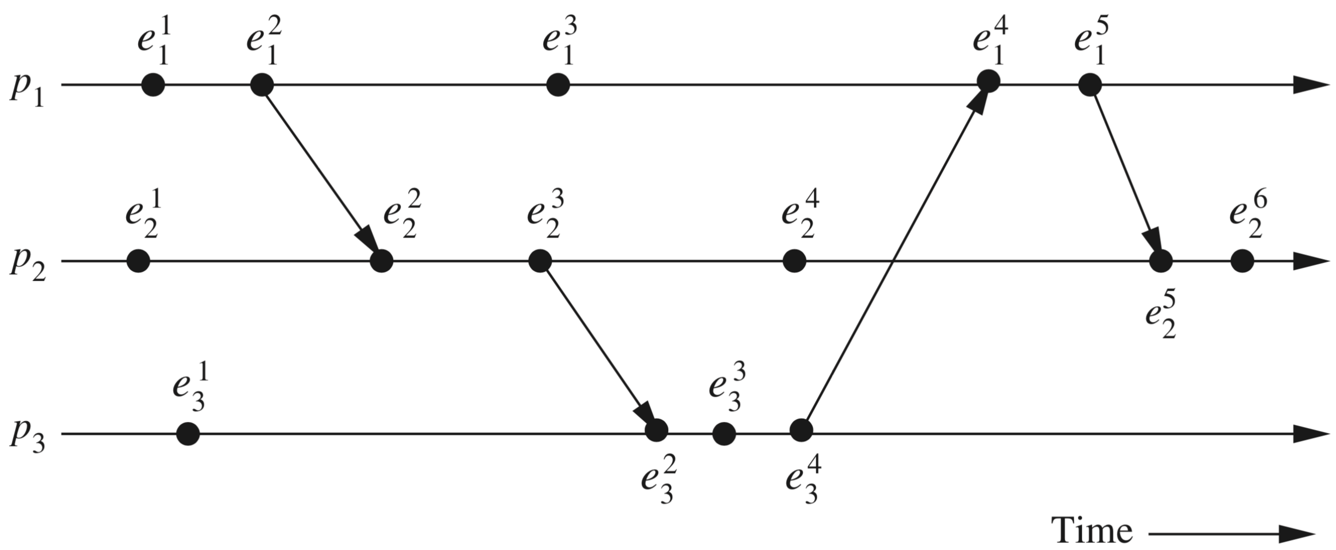
Causal Precedence Relation

- What information does e_2^6 have?
- Knowledge about all other events!



Causal Precedence Relation

- $e_i \not\rightarrow e_j$: e_j does not directly or transitively depend on e_i
  - e_i does not causally affect e_j
- e_j is not aware of execution of e_i or any event executed after e_i on the same process
- Example: $e_1^3 \not\rightarrow e_3^3$ and $e_2^4 \not\rightarrow e_3^1$



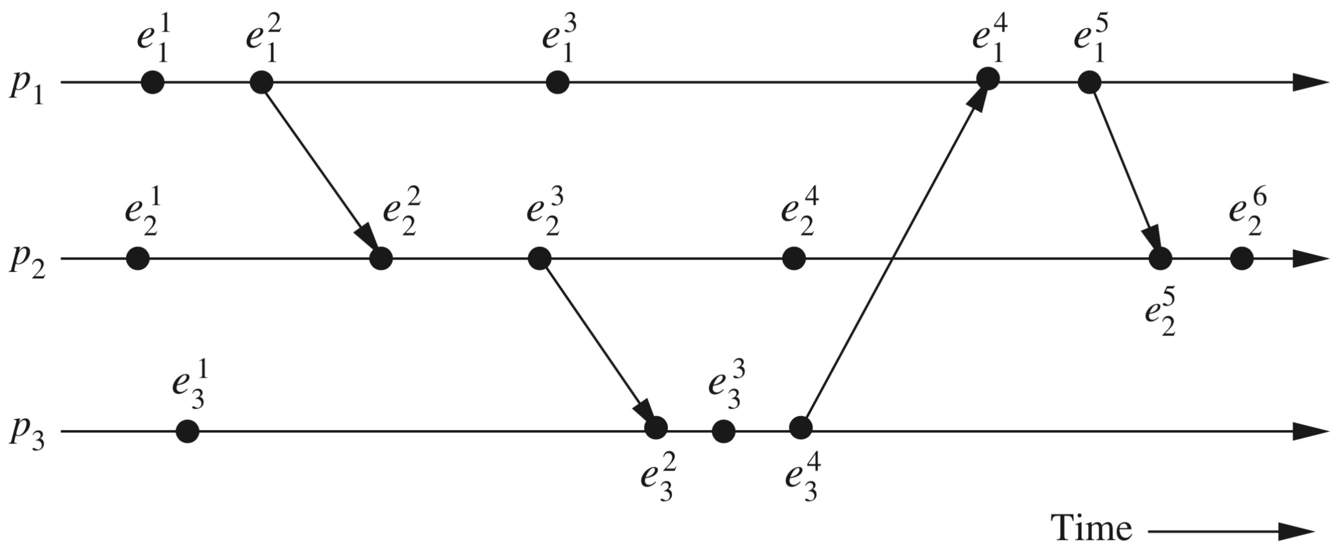
Causal Precedence Relation

- $e_i \not\rightarrow e_j$: e_j does not directly or transitively depend on e_i
  - e_i does not causally affect e_j
- e_j is not aware of execution of e_i or any event executed after e_i on the same process

- Rules:
  - $e_i \not\rightarrow e_j \not\Rightarrow e_j \not\rightarrow e_i$
  - $e_i \rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$

Concurrent Events

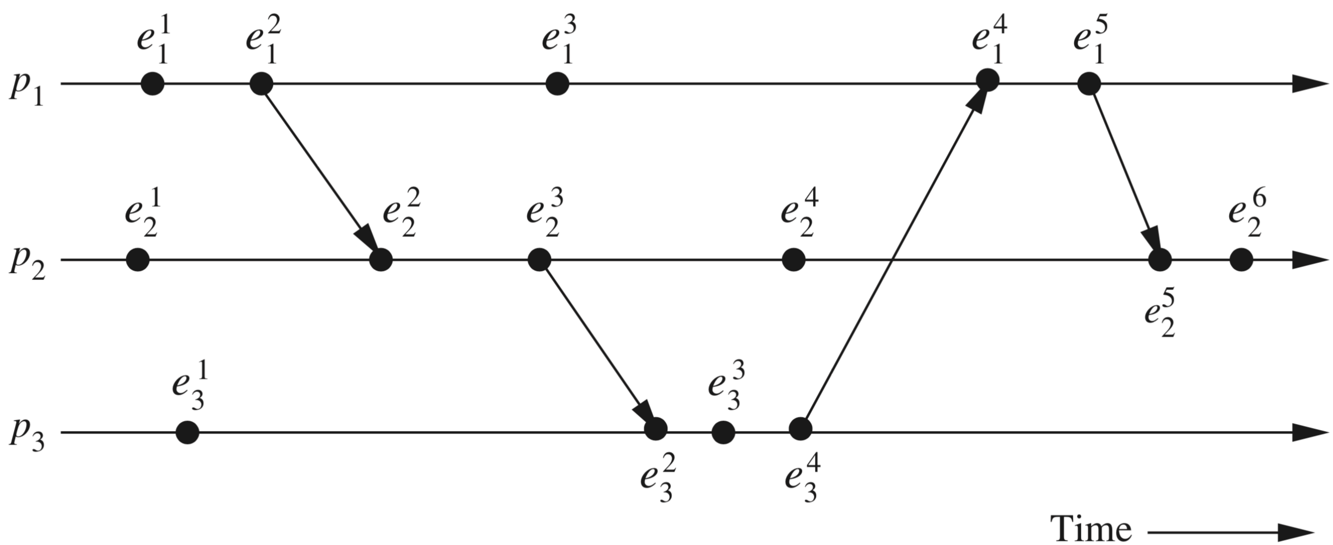
- e_i and e_j are concurrent if $e_i \nrightarrow e_j$ and $e_j \nrightarrow e_i$
  - Denoted as $e_i \parallel e_j$
- Example: $e_1^3 \parallel e_3^3$ and $e_2^4 \parallel e_3^1$



Concurrent Events

- Is ∥ transitive?
- $(e_i \parallel e_j) \wedge (e_j \parallel e_k) \nRightarrow e_i \parallel e_k$

- Example: $e_3^3 \parallel e_2^4$ and $e_2^4 \parallel e_1^5$, however e3 ∦ e15



Logical v/s Physical Concurrency

- Analogous to concurrent v/s parallel
- Logically concurrent: $e_i \parallel e_j$
- Physically concurrent: e_i and e_j occur at the same instant in physical (real) time
- Logically concurrent events may not be physically concurrent
- Does it matter if events are physically concurrent?
- Being physically concurrent doesn't change the outcome
- Hence, we can assume logically concurrent events occurred at the same instant in physical time

Communication Models

- Options: FIFO, Non-FIFO, and Causal Ordering

- FIFO model
  - Each channel acts as a first-in first-out message queue
  - Message ordering is preserved by a channel.

- Non-FIFO model
  - Each channel acts like a set
  - No ordering guarantee

Communication Models

- Based on Lamport's "happens before" relation

For any two messages m_{ij} and m_{kj},
if $send(m_{ij}) \rightarrow send(m_{kj})$, then $rec(m_{ij}) \rightarrow rec(m_{kj})$

- Causally related messages destined to the same destination are delivered in an order that is consistent with their causality relation

Communication Models

- Causal ordering model simplifies design of distributed systems because it inherently provides synchronization
- Example: in a replicated data-store, every process responsible for updating a replica receives updates in the same order to maintain consistency

- How is causal ordering related to FIFO?
- CO $\subset$ FIFO $\subset$ Non-FIFO
  - Causally ordered delivery implies FIFO delivery

Global State of a Distributed System

- Collection of local states of its components
  - processes and communication channels
- State of a process
  - Data (contents of processor registers, stacks, local memory, etc.)
  - Depends on the context of distributed application
- State of a channel
  - Set of messages in transit in the channel

Global State of a Distributed System

- Events change states of respective processes and channels
  - Internal event changes the state of the process at which it occurs.
  - A send event changes the state of the process that sends the message and the state of the channel on which the message is sent
  - A receive event changes the state of the process that receives the message and the state of the channel on which the message is received

Notations

- LS_i^x : state of process p_i after the occurrence of event e_i^x and before the event e_i^{x+1}
- LS_i^0 : initial state of process p_i
- LS_i^x : result of execution of all the events till e_i^x by p_i

- $send(m) \leq LS_i^x$: $\exists y: 1 \leq y \leq x$ s.t. $e_i^y = send(m)$
- $rec(m) \not\leq LS_i^x$: $\forall y: 1 \leq y \leq x$ s.t. $e_i^y \neq rec(m)$

Channel State

- State of a channel depends on the states of the processes it connects
- $SC_{ij}^{x,y}$: state of channel C_{ij}

$$SC_{ij}^{x,y} = \{m_{ij} \mid send(m_{ij}) \leq e_i^x \land rec(m_{ij}) \nleq e_j^y\}$$

- $SC_{ij}^{x,y}$ denotes all messages that p_i sent up to event e_i^x and which p_j had not received until event e_j^y

Global State

$$GS = \{\bigcup_i LS_i^{xi}, \bigcup_{j,k} SC_{jk}^{yj,zk}\}$$

- For a global state to be meaningful, the states of all the components must be recorded at the same instant
  - Not possible!

Consistent Global State

- Basic idea: global state should not violate causality:
  An effect should not be present without its cause
  - A message cannot be received if it was not sent

- Even if the state of all components is not recorded at the same instant, it is meaningful provided every message that is recorded as received is also recorded as sent

Consistent Global State

- A global state GS = $\{\bigcup_i LS_i^{xi}, \bigcup_{j,k} SC_{jk}^{yj,zk}\}$ is a consistent global state iff

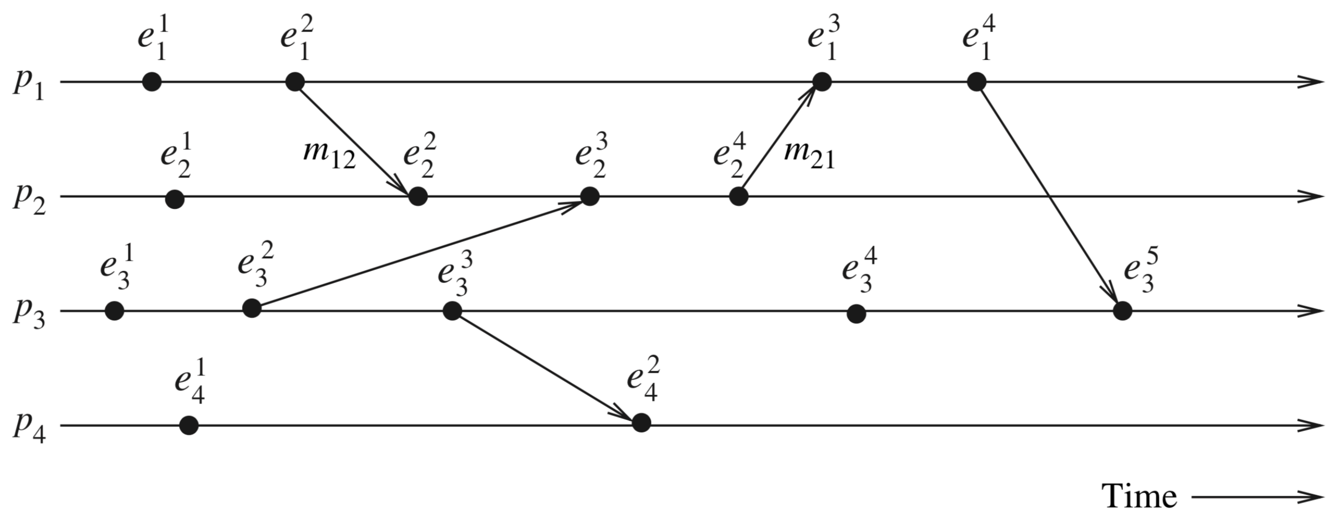
  $\forall m_{ij} : send(m_{ij}) \not\leq LS_i^{xi} \Leftrightarrow m_{ij} \notin SC_{ij}^{xi,yj} \wedge rec(m_{ij}) \not\leq LS_j^{yj}$

- Channel state $SC_{jk}^{yj,zk}$ and process state LS_i^{xi} must not include any message that p_i sent after executing e_i^{xi}

- Inconsistent global states are not meaningful
- A distributed system can never be in an inconsistent state

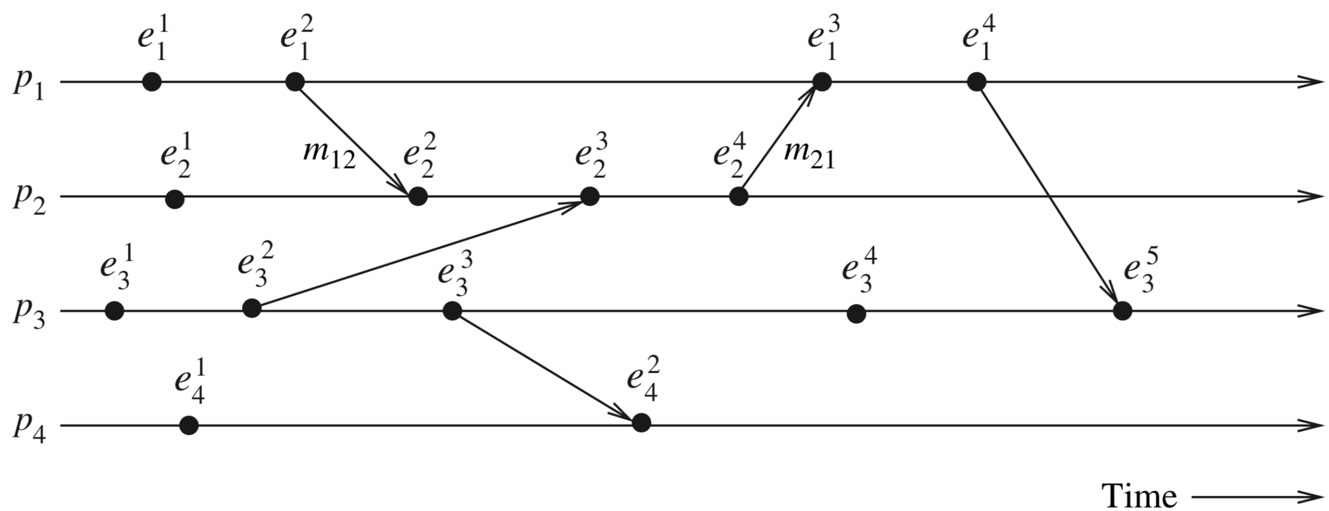
Global State of a Distributed System

- Is GS = $\{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$ consistent?
- No: p_2 has recorded receipt of message m_{12}, but p_1 has not recorded its send



Global State of a Distributed System

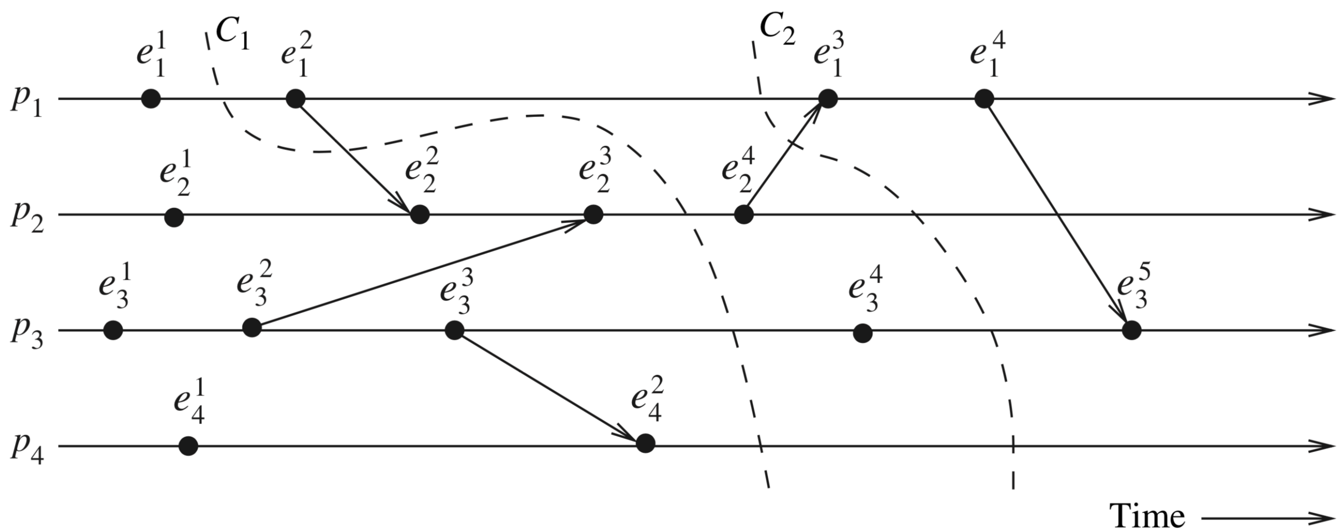
- Is GS = $\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$ consistent?
- Yes: All channels are empty, except c_{21} which contains message m_{21} whose send is recorded by p_2



Cut of a Distributed Computation

- Cut is a global state of distributed computation
- Slices the space-time diagram into two parts: past & future
    - Past: all events to the left of the cut
    - Future: all events to the right of the cut
- Powerful graphical aid in representing and reasoning about global states of a computation

Cut of a Distributed Computation



Consistent Cut

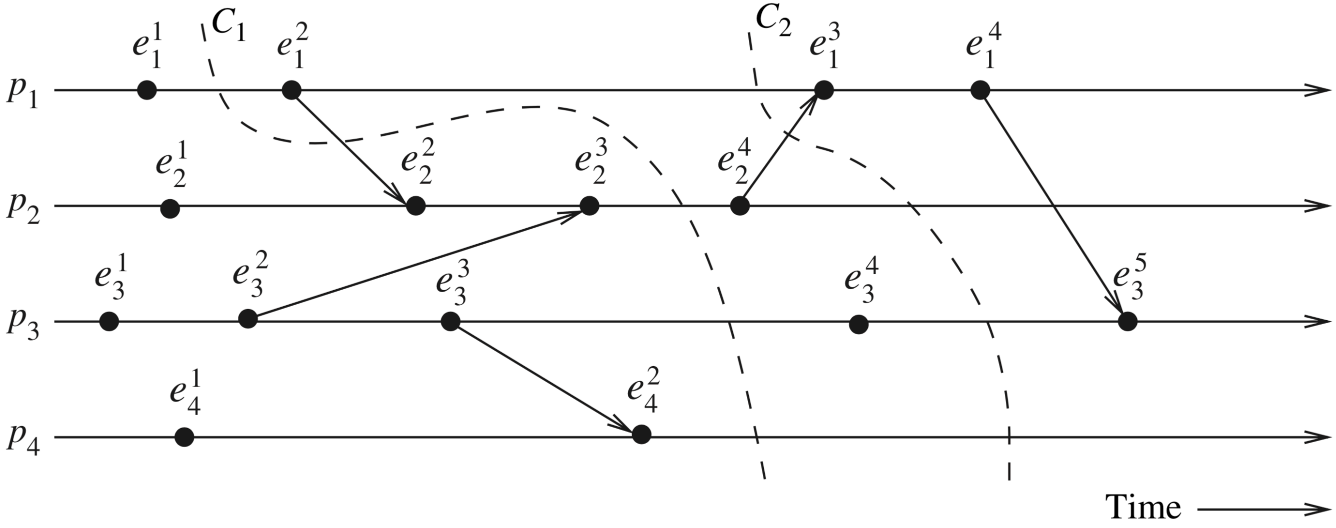
- Every message received in past of the cut must be sent in the past of that cut
- Messages can cross the cut from past to future
    - Messages in transit

- Inconsistent cut: if a message crosses the cut from the future to past

Consistent Cut

Is C_1 consistent?

Is C_2 consistent?



Past and Future Cones of an Event

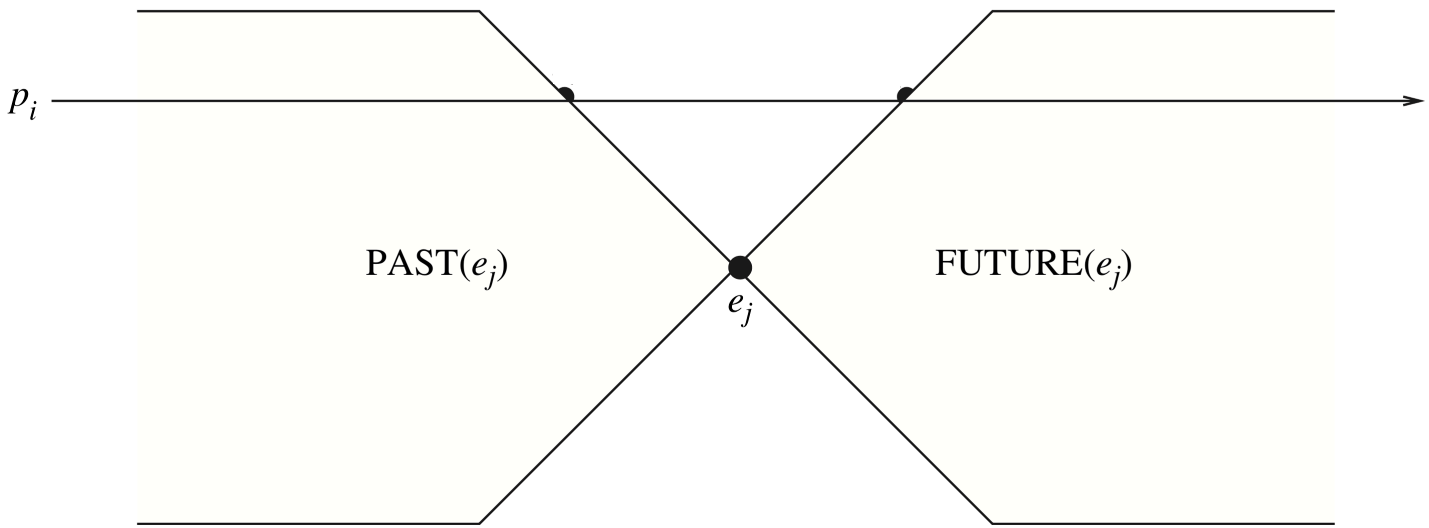
$$\text{Past}(e_j)=\{e_i \mid \forall e_i \in H, e_i \rightarrow e_j\}$$

- $\text{Past}(e_j)$ contains all events e_i such that could affect e_j
- All the information available at e_i could be accessible at e_j

$$\text{Future}(e_j)=\{e_i \mid \forall e_i \in H, e_j \rightarrow e_i\}$$

- $\text{Future}(e_j)$ contains all events e_i that e_j could affect
- All the information available at e_j could be propagated to e_i

Past and Future Cones of an Event



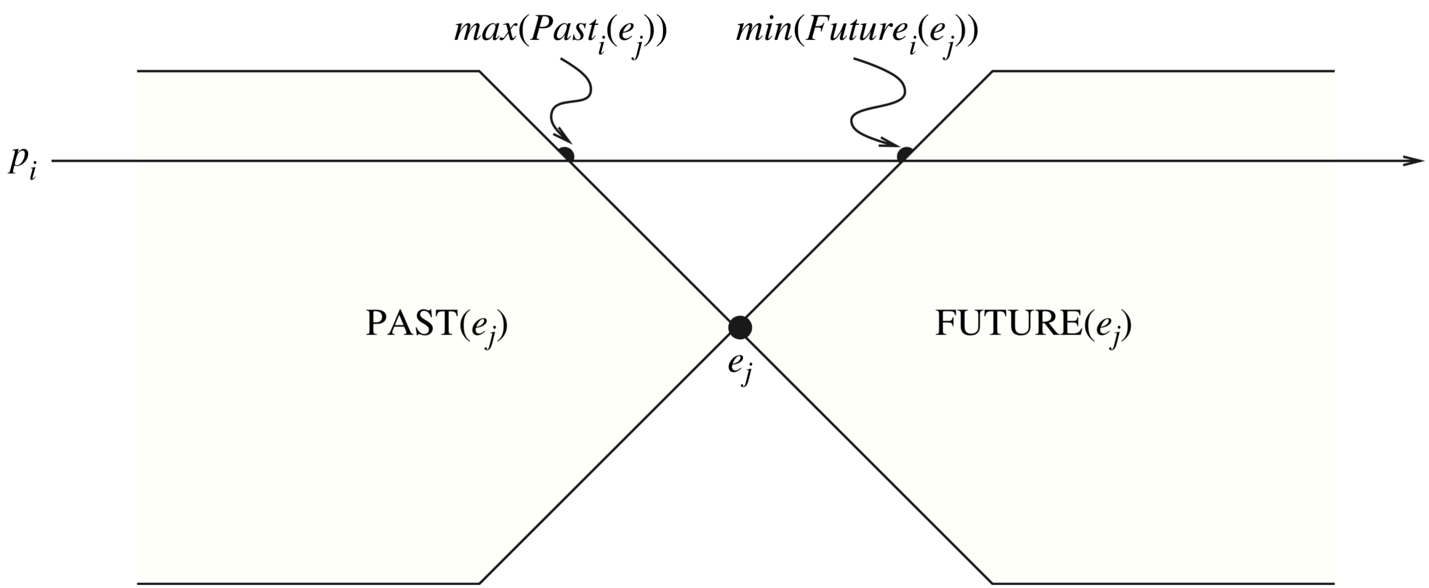
Past and Future Cones of an Event

- $Past_i(e_j)$: set of all those events of p_i in $Past(e_j)$
- $Past_i(e_j)$ is a totally ordered set – Why?
  - Ordered by the relation $\rightarrow_i$

- $max(Past_i(e_j))$: Maximal element of $Past_i(e_j)$
- $max(Past_i(e_j))$ is the latest event at process p_i that affected event e_j

- $min(Future_i(e_j))$ is the earliest event at process p_i that p_j could affect

Past and Future Cones of an Event



Past and Future Cones of an Event

$$\text{Max\_Past}(e_j) = \bigcup_i \max(\text{Pasti}(ej))$$

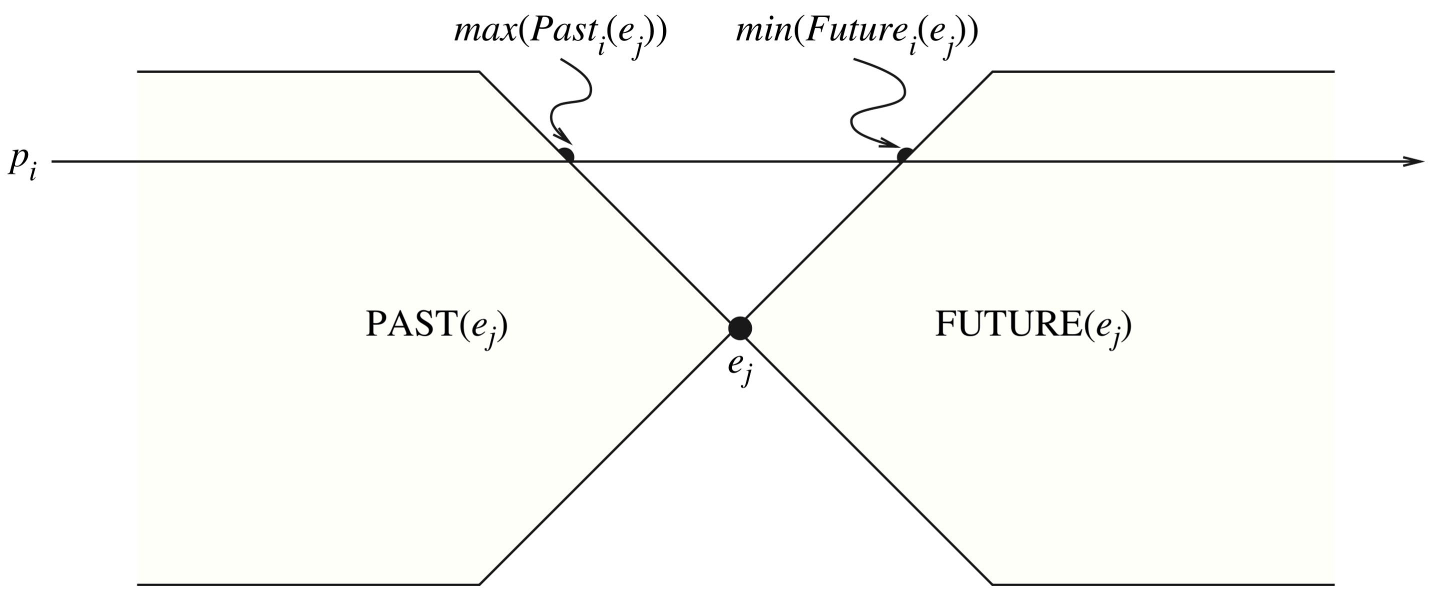
- Max_Past(e_j) contains all the latest events at all process that affected e_j
- Max_Past(e_j) is the surface of the past cone of e_j

$$\text{Min\_Future}(e_j) = \bigcup_i \min(\text{Future}_i(ej))$$

- Min_Future(e_j) contains all the earliest events at all process that are affected by e_j
- Min_Future(e_j) is the surface of the future cone of e_j

Past and Future Cones of an Event

- What can be say about events after max($Past_i(e_j)$) but before min($Future_i(e_j)$)?
  - They are concurrent with e_j



Model of Distributed Computations

- Useful to analyze, design and debug distributed systems
- Reason about operations/events in distributed computing
  - Construct a consistent global state of the distributed system
  - Events that happen concurrently
  - Events that happened before a certain event
  - Events that affect a certain event
  - Events that are affected by a certain event

- Abstraction of events depends on application
  - E.g., with threads within a process, shared memory reads and writes must be captured