Virtual Time and Global States in Distributed Systems

Prof. Nalini Venkatasubramanian

Distributed Systems Middleware - Lecture 2

The Concept of Time

- The Concept of Time
  - A standard time is a set of instants with a temporal precedence order < satisfying certain conditions [Van Benthem 83]:
    - Transitivity
    - Irreflexivity
    - Linearity
    - Eternity ($\forall x \exists y: x<y$)
    - Density ($\forall x,y: x<y \rightarrow \exists z: x<z<y$)

  - Transitivity and Irreflexivity imply asymmetry

Time as a Partial Order

- A linearly ordered structure of time is not always adequate for distributed systems
  - Captures dependence, not independence of distributed activities

- A partially ordered system of *vectors* forming a *lattice* structure is a natural representation of time in a distributed system

- Resembles Einstein-Minkowski's relativistic space-time

Global Time & Global State of Distributed Systems

- Asynchronous distributed systems consist of several *processes* without common memory which communicate (solely) via *messages* with unpredictable transmission delays
- Global time & global state are hard to realize in distributed systems
  - Processes are distributed geographically
  - Rate of event occurrence can be high (unpredictable)
  - Event execution times can be small
- We can only *approximate* the global view
  - *Simulate synchronous* distributed system on given asynchronous systems
  - *Simulate* a *global time* – Logical Clocks
  - *Simulate* a *global state* – Global Snapshots

Simulate Synchronous Distributed Systems

- *Synchronizers* [Awerbuch 85]
  - Simulate clock pulses in such a way that a message is only generated at a clock pulse and will be received before the next pulse
  - Drawback
    - Very high message overhead

Simulating global time

- An accurate notion of global time is difficult to achieve in distributed systems.
  - We often derive "causality" from loosely synchronized clocks
- Clocks in a distributed system drift
  - Relative to each other
  - Relative to a real world clock
    - Determination of this real world clock itself may be an issue
  - Clock Skew versus Drift
    - Clock Skew = Relative Difference in clock values of two processes
    - Clock Drift = Relative Difference in clock frequencies (rates) of two processes
- .

Clock Synchronization

- A non-zero clock drift will cause skew to continuously increase
- Maximum Drift Rate (MDR) of a clock
  - Absolute MDR is defined relative to a Coordinated Universal Time (UTC)
  - MDR of a process depends on the environment.
  - Max drift rate between two clocks with similar MDR is 2 * MDR

Max-Synch-Interval = (MaxAcceptableSkew — CurrentSkew) / (MDR * 2)

- Clock synchronization is needed to simulate global time
  - Correctness – consistency, fairness
- Physical Clocks vs. Logical clocks
  - Physical clocks - must not deviate from the real-time by more than a certain amount.

Physical Clocks

- How do we measure real time?
  - 17th century - Mechanical clocks based on astronomical measurements
    - Solar Day - Transit of the sun
    - Solar Seconds - Solar Day/(3600*24)
  - Problem (1940) - Rotation of the earth varies (gets slower)
  - Mean solar second - average over many days

Atomic Clocks

- 1948
  - counting transitions of a crystal (Cesium 133) used as atomic clock
  - TAI - International Atomic Time
    - 9192631779 transitions = 1 mean solar second in 1948
  - UTC (Universal Coordinated Time)
    - From time to time, we skip a solar second to stay in phase with the sun (30+ times since 1958)
    - UTC is broadcast by several sources (satellites...)

Accuracy of Computer Clocks

- Modern timer chips have a relative error of 1/100,000 - 0.86 seconds a day
- To maintain synchronized clocks
  - Can use UTC source (time server) to obtain current notion of time
  - Use solutions without UTC.

Cristian's (Time Server) Algorithm

- Uses a *time server* to synchronize clocks
  - Time server keeps the reference time (say UTC)
  - A client asks the time server for time, the server responds with its current time, and the client uses the received value *T* to set its clock
- But network round-trip time introduces errors…
  - Let **RTT = response-received-time – request-sent-time** (measurable at client),
  - If we know (a) min = minimum client-server one-way transmission time and (b) that the server timestamped the message at the last possible instant before sending it back
  - Then, the actual time could be between **[T+min,T+RTT— min]**

Cristian's Algorithm

- Client sets its clock to halfway between T+min and T+RTT— min i.e., at T+RTT/2
  - ☹ Expected (i.e., average) skew in client clock time = (RTT/2 – *min*)
- Can increase clock value, should never decrease it.
- Can adjust speed of clock too (either up or down is ok)
- Multiple requests to increase accuracy
  - For unusually long RTTs, repeat the time request
  - For non-uniform RTTs
    - Drop values beyond threshold; Use averages (or weighted average)

Berkeley UNIX algorithm

- One daemon without UTC
- Periodically, this daemon polls and asks all the machines for their time
- The machines respond.
- The daemon computes an average time and then broadcasts this average time.

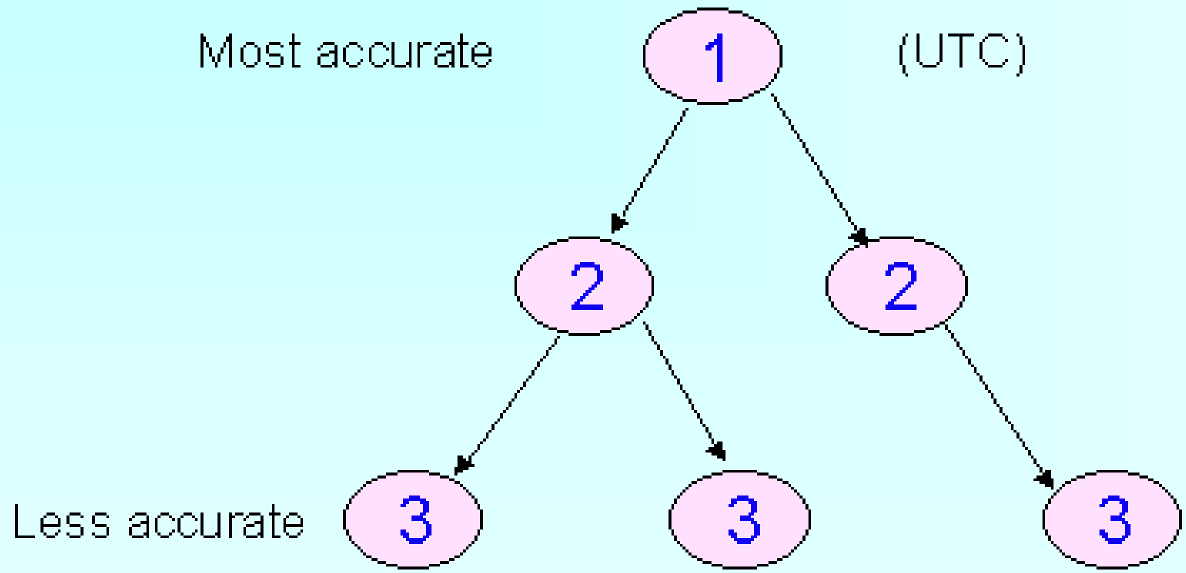
Decentralized Averaging Algorithm

- Each machine has a daemon without UTC
- Periodically, at fixed agreed-upon times, each machine broadcasts its local time.
- Each of them calculates the average time by averaging all the received local times.

Clock Synchronization in DCE

- DCE's time model is actually in an interval
  - I.e. time in DCE is actually an interval
  - Comparing 2 times may yield 3 answers
    - t1 < t2
    - t2 < t1
    - not determined
  - Each machine is either a time server or a clerk
  - Periodically a clerk contacts all the time servers on its LAN
  - Based on their answers, it computes a new time and gradually converges to it.

Hierarchy in NTP



Event Structures

- A process can be viewed as consisting of a sequence of events, where an event is an atomic transition of the local state which happens in no time
- Process Actions can be modeled using the 3 types of events
  - Send
  - Receive
  - Internal (change of state)

Causal Relations

- Distributed application results in a set of distributed events
  - Induces a partial order → causal precedence relation
- Knowledge of this causal precedence relation is useful in reasoning about and analyzing the properties of distributed computations
  - Liveness and fairness in mutual exclusion
  - Consistency in replicated databases
  - Distributed debugging, checkpointing

An Event Framework for Logical Clocks

- Events are related
  - Events occurring at a particular process are totally ordered by their local sequence of occurrence
  - Each receive event has a corresponding send event
  - Future can not influence the past (causality relation)
  - Event structures represent distributed computation (in an abstract way)
    - An event structure is a pair $(E,<)$, where E is a set of events and $<$ is a irreflexive partial order on E, called the causality relation
  - For a given computation, $e<e'$ holds if one of the following conditions holds
    - e,e' are events in the same process and e precedes e'
    - e is the sending event of a message and e' the corresponding receive event
    - $\exists e'': e<e'' \wedge e''<e'$

Event Ordering

- Lamport defined the “happens before” (=>) relation
  - If a and b are events in the same process, and a occurs before b, then a => b.
  - If a is the event of a message being sent by one process and b is the event of the message being received by another process, then a => b.
  - If X =>Y and Y=>Z then X => Z.

  If a => b then time (a) => time (b)

Causal Ordering

- "Happens Before" also called causal ordering
- Possible to draw a causality relation between 2 events if
  - They happen in the same process
  - There is a chain of messages between them
- "Happens Before" notion is not straightforward in distributed systems
  - No guarantees of synchronized clocks
  - Communication latency

Logical Clocks

- Used to determine causality in distributed systems
- Time is represented by non-negative integers
- A logical Clock C is some abstract mechanism which assigns to any event $e \in E$ the value $C(e)$ of some time domain T such that certain conditions are met
    - $C: E \rightarrow T$ :: T is a partially ordered set : $e < e' \rightarrow C(e) < C(e')$ holds
- Consequences of the clock condition [Morgan 85]:
    - If an event e occurs before event e' at some single process, then event e is assigned a logical time earlier than the logical time assigned to event e'
    - For any message sent from one process to another, the logical time of the send event is always earlier than the logical time of the receive event

Implementation of Logical Clocks

- Requires
  - Data structures local to every process to represent logical time and
  - a protocol to update the data structures to ensure the consistency condition.
- Each process Pi maintains data structures that allow it the following two capabilities:
  - A local logical clock, denoted by LC_i , that helps process Pi measure its own progress.
  - A logical global clock, denoted by GCi , that is a representation of process Pi ’s local view of the logical global time. Typically, lci is a part of gci
- The protocol ensures that a process’s logical clock, and thus its view of the global time, is managed consistently.
  - The protocol consists of the following two rules:
    - R1: This rule governs how the local logical clock is updated by a process when it executes an event.
    - R2: This rule governs how a process updates its global logical clock to update its view of the global time and global progress.

Types of Logical Clocks

- Systems of logical clocks differ in their representation of logical time and also in the protocol to update the logical clocks.
- 3 kinds of logical clocks
  - Scalar
  - Vector
  - Matrix

Scalar Logical Clocks - Lamport

- Proposed by Lamport in 1978 as an attempt to totally order events in a distributed system.
- Time domain is the set of non-negative integers.
- The logical local clock of a process pi and its local view of the global time are squashed into one integer variable Ci .
- Monotonically increasing counter
  - No relation with real clock
- Each process keeps its own logical clock used to timestamp events

Consistency with Scalar Clocks

- To guarantee the clock condition, local clocks must obey a simple protocol:
  - When executing an internal event or a send event at process P_i the clock C_i ticks
    - C_i += d (d>0)
  - When P_i sends a message m, it piggybacks a logical timestamp t which equals the time of the send event
  - When executing a receive event at P_i where a message with timestamp *t* is received, the clock is advanced
    - $C_i = \max(C_i, t) + d$ (d>0)
- Results in a partial ordering of events.

Lamport Logical Clock

P1	P2	P3
0	0	0
4	5	3
8	10	6
12	15	9
16	20	12
20	25	15
24	30	18
28	35	21
32	40	24

P1	P2	P3
0	0	0
4	5	3
8	10	6
12	15	**11**
16	20	14
20	25	17
24	30	20
28	35	23
36	40	26

Total Ordering

- Extending partial order to total order

time	Proc_id

- Global timestamps:
  - (Ta, Pa) where Ta is the local timestamp and Pa is the process id.
  - (Ta,Pa) < (Tb,Pb) iff
    - (Ta < Tb) or ((Ta = Tb) and (Pa < Pb))
  - Total order is consistent with partial order.

Properties of Scalar Clocks

- Event counting
  - If the increment value d is always 1, the scalar time has the following interesting property: if event e has a timestamp h, then h-1 represents the minimum logical duration, counted in units of events, required before producing the event e;
  - We call it the height of the event e.
  - In other words, h-1 events have been produced sequentially before the event e regardless of the processes that produced these events.

Properties of Scalar Clocks

- No Strong Consistency
- The system of scalar clocks is not strongly consistent; that is, for two events ei and ej , C(ei) < C(ej) 6=⇒ ei → ej .
- Reason: In scalar clocks, logical local clock and logical global clock of a process are squashed into one, resulting in the loss of causal dependency information among events at different processes.

Independence

- Two events e,e' are mutually independent (i.e. $e||e'$) if $\sim(e<e')\wedge\sim(e'<e)$
  - Two events are independent if they have the same timestamp
  - Events which are causally independent may get the same or different timestamps
- By looking at the timestamps of events it is not possible to assert that some event *could not* influence some other event
  - If $C(e)<C(e')$ then $\sim(e<e')$ *however*, it *is not possible* to decide whether $e<e'$ or $e||e'$
  - C is an order *homomorphism* which preserves < but it does not preserves negations (i.e. obliterates a lot of structure by mapping E into a linear order)
  - An *isomorphism* mapping E onto T is requiered

Problems with Total Ordering

- A linearly ordered structure of time is not always adequate for distributed systems
  - captures dependence of events
  - loses independence of events - artificially enforces an ordering for events that need not be ordered.
  - Mapping partial ordered events onto a linearly ordered set of integers it is *losing information*
    - Events which may happen simultaneously may get different timestamps as if they happen in some definite order.
- A partially ordered system of *vectors* forming a *lattice* structure is a natural representation of time in a distributed system

Vector Times

- The system of vector clocks was developed independently by Fidge, Mattern and Schmuck.
- To construct a mechanism by which each process gets an optimal approximation of global time
- In the system of vector clocks, the time domain is represented by a set of n-dimensional non-negative integer vectors.
- Each process has a clock C_i consisting of a vector of length *n*, where *n* is the total number of processes vt[1..n], where vt[j] is the local logical clock of Pjand describes the logical time progress at process Pj .
  - A process P_i ticks by incrementing its own component of its clock
    - $C_i[i] += 1$
  - The timestamp C(e) of an event e is the clock value after ticking
  - Each message gets a piggybacked timestamp consisting of the vector of the local clock
    - The process gets some knowledge about the other process' time approximation
    - $C_i=\sup(C_i,t)::\ \sup(u,v)=w : w[i]=\max(u[i],v[i]), \forall i$

Vector Clocks example

- An Example of vector clocks

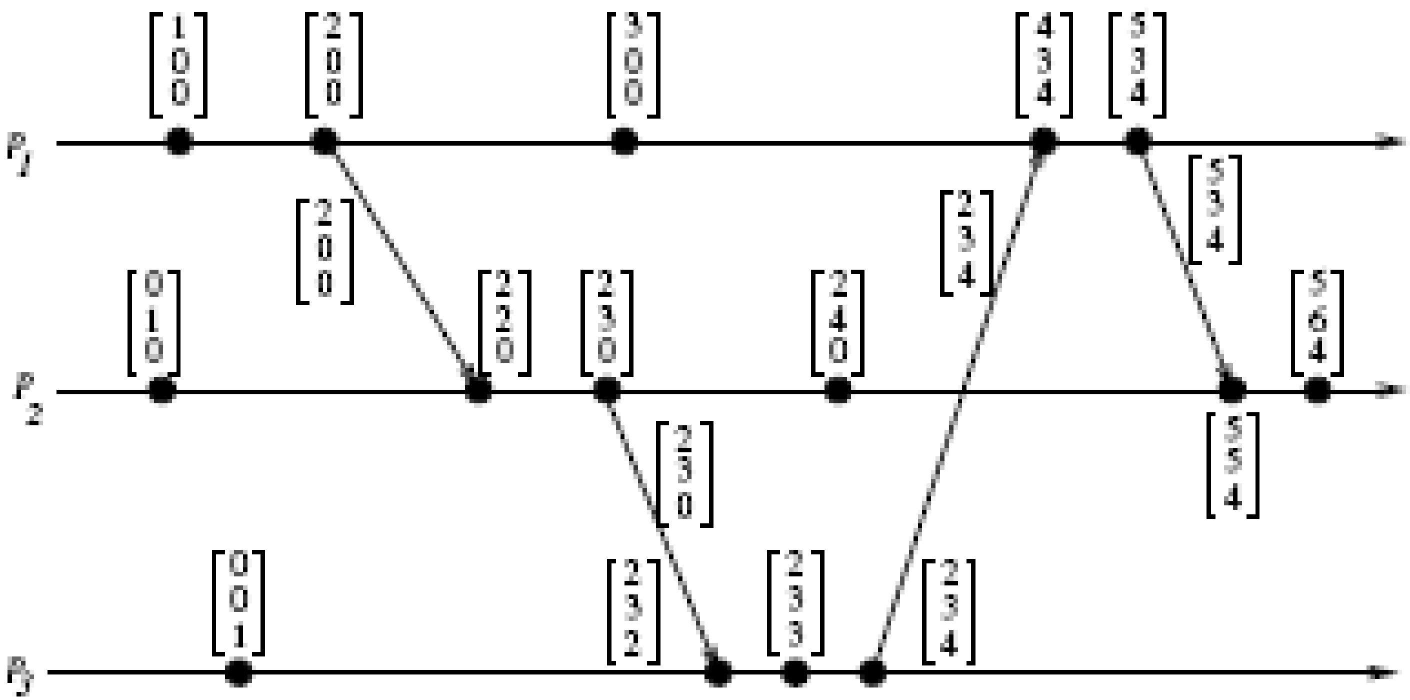


Figure 3.2: Evolution of vector time.

From A. Kshemkalyani and M. Singhal (Distributed Computing)

Vector Times (cont)

- Because of the transitive nature of the scheme, a process *may receive* time updates about clocks in non-neighboring process
- Since process P_i can advance the i^{th} component of global time, it always has the most accurate knowledge of its local time
    - At any instant of real time $\forall i,j$: $C_i[i] \geq C_j[i]$
  - For two time vectors u,v
    - $u \leq v$ iff $\forall i$: $u[i] \leq v[i]$
    - $u < v$ iff $u \leq v \wedge u \neq v$
    - $u || v$ iff $\sim(u<v) \wedge \sim(v<u)$ :: $||$ is not transitive

Structure of the Vector Time

- For any $n>0$, $(N^n,\leq)$ is a lattice
- The set of possible time vectors of an event set E is a sublattice of $(N^n,\leq)$
- For an event set E, the lattice of consistent cuts and the lattice of possible time vectors *are isomorphic*
- $\forall e,e' \in E: e<e'$ iff $C(e)<C(e') \wedge e||e'$ iff iff $C(e)||C(e')$

- In order to determine if two events e,e' are causally related or not, just take their timestamps C(e) and C(e')
  - if $C(e)<C(e') \vee C(e')<C(e)$, then the events *are causally related*
  - Otherwise, they *are causally independent*

Matrix Time

- Vector time contains information about latest direct dependencies
  - What does Pi know about Pk
- Also contains info about latest direct dependencies of those dependencies
  - What does Pi know about what Pk knows about Pj
- Message and computation overheads are high
- Powerful and useful for applications like distributed garbage collection

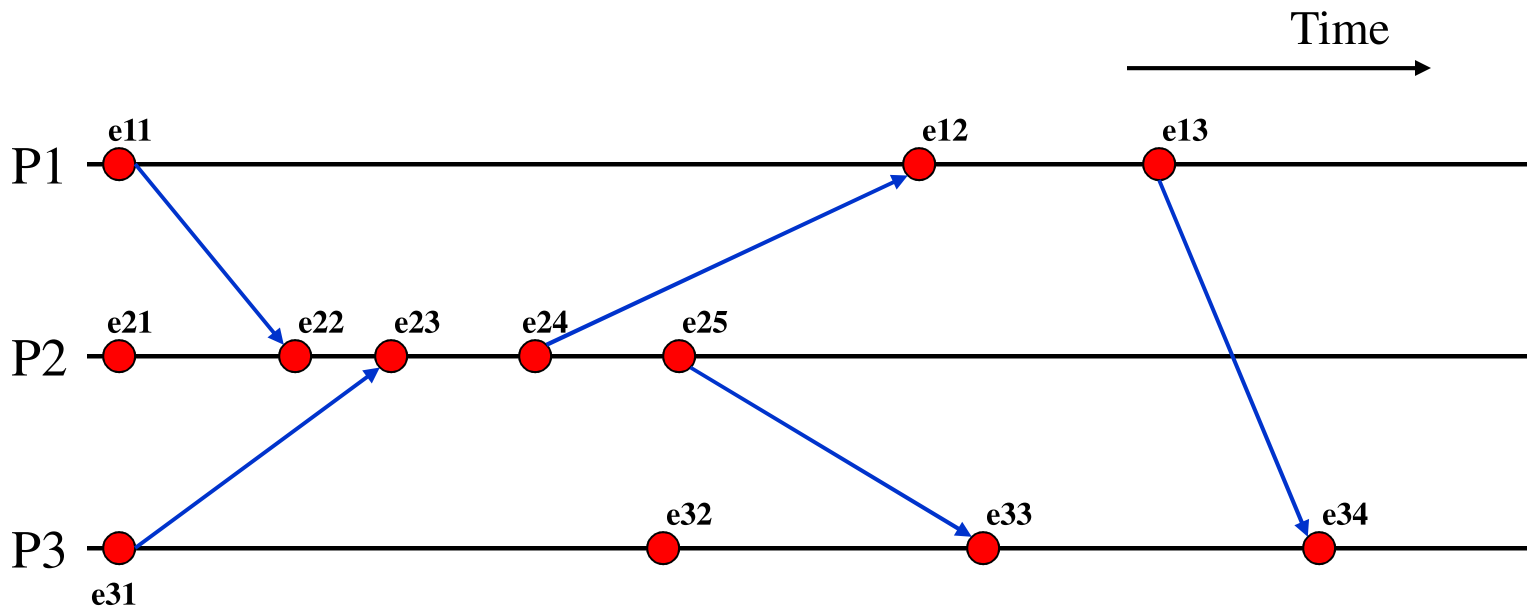
Time Manager Operations

- Logical Clocks
  - C.adjust(L,T)
    - adjust the local time displayed by clock C to T (can be gradually, immediate, per clock sync period)
  - C.read
    - returns the current value of clock C
- Timers
  - TP.set(T) - reset the timer to timeout in T units
- Messages
  - receive(m,l); broadcast(m); forward(m,l)

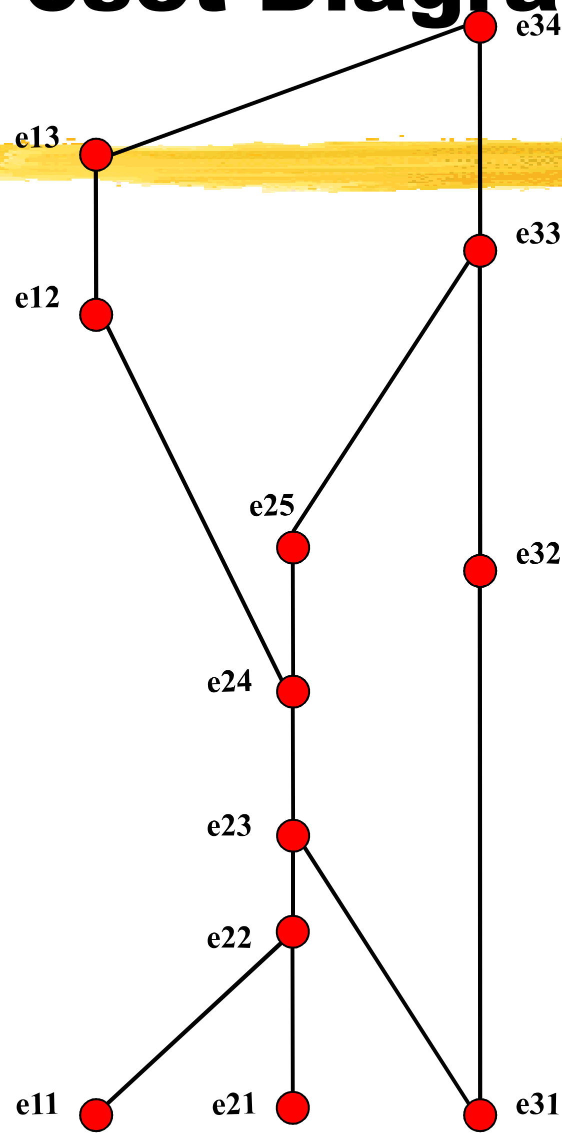
Simulate A Global State

- The notions of global time and global state are closely related
- A process can (without *freezing* the whole computation) compute the *best possible approximation* of a global state [Chandy & Lamport 85]
- A global state that *could* have occurred
  - No process in the system can decide whether the state did really occur
  - Guarantee stable properties (i.e. once they become true, they remain true)

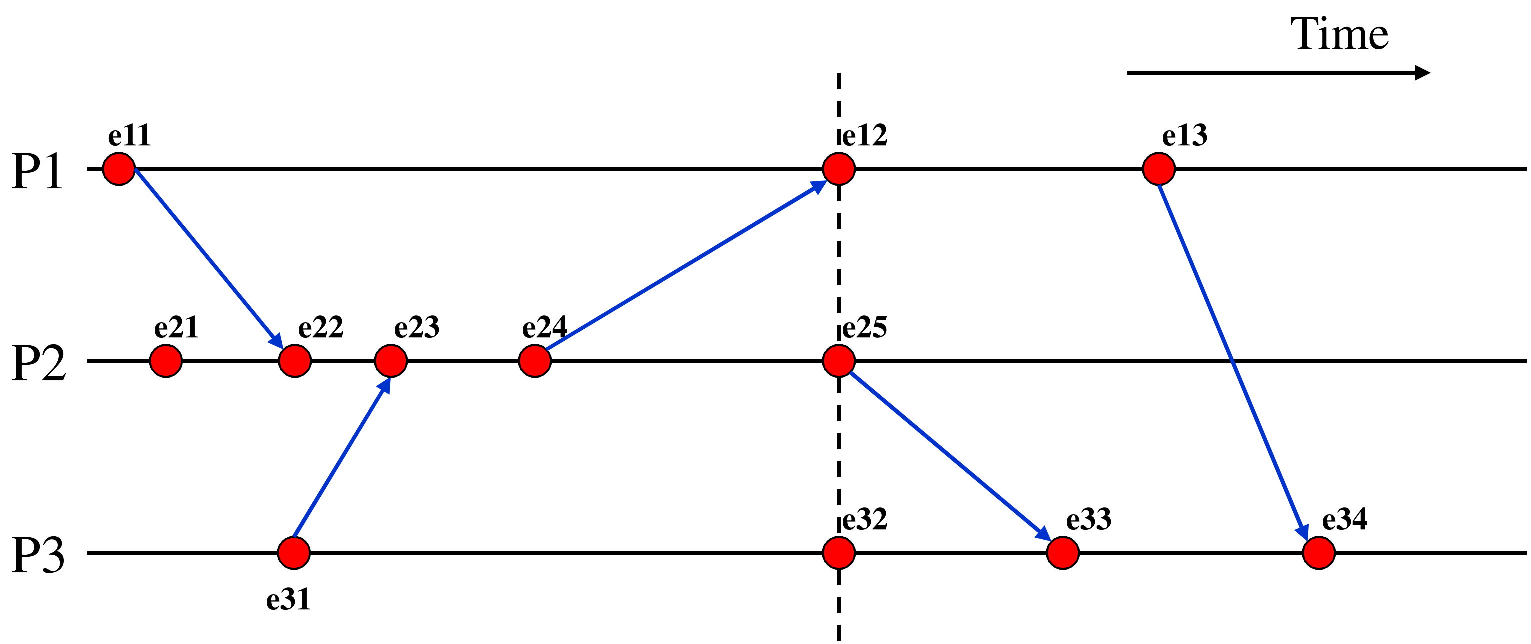
Event Diagram



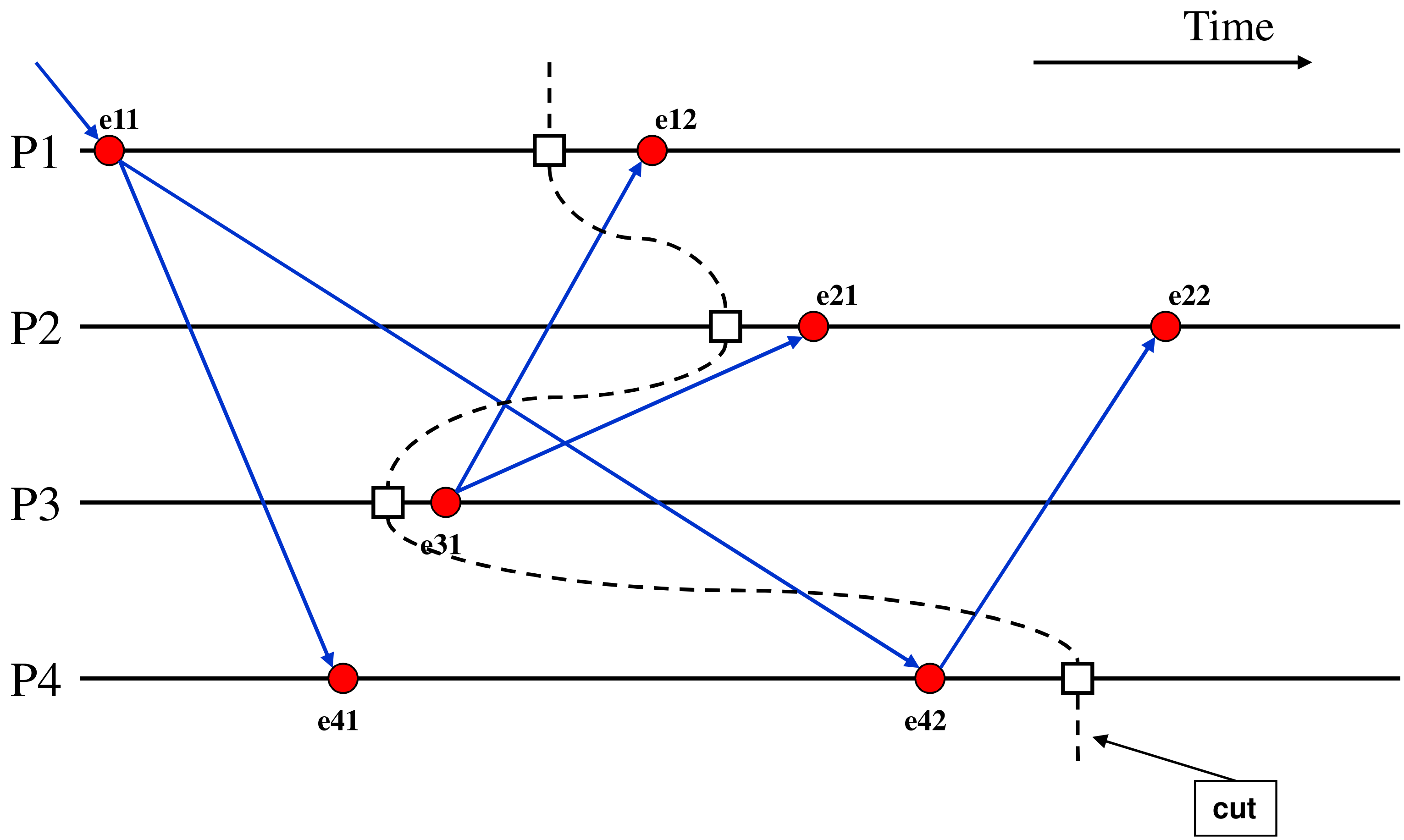
Poset Diagram



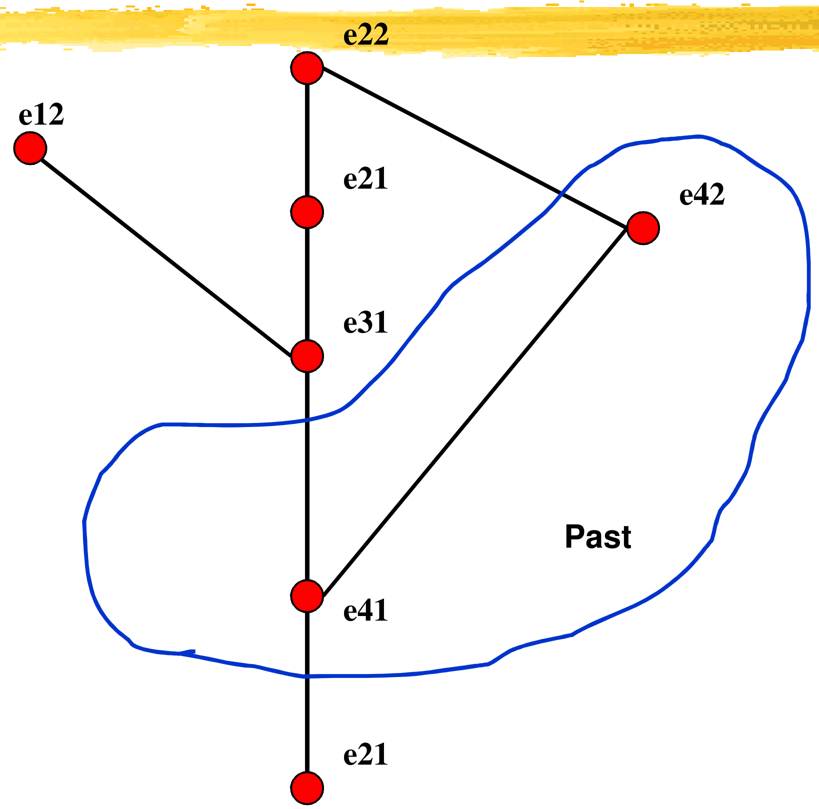
Equivalent Event Diagram



Rubber Band Transformation



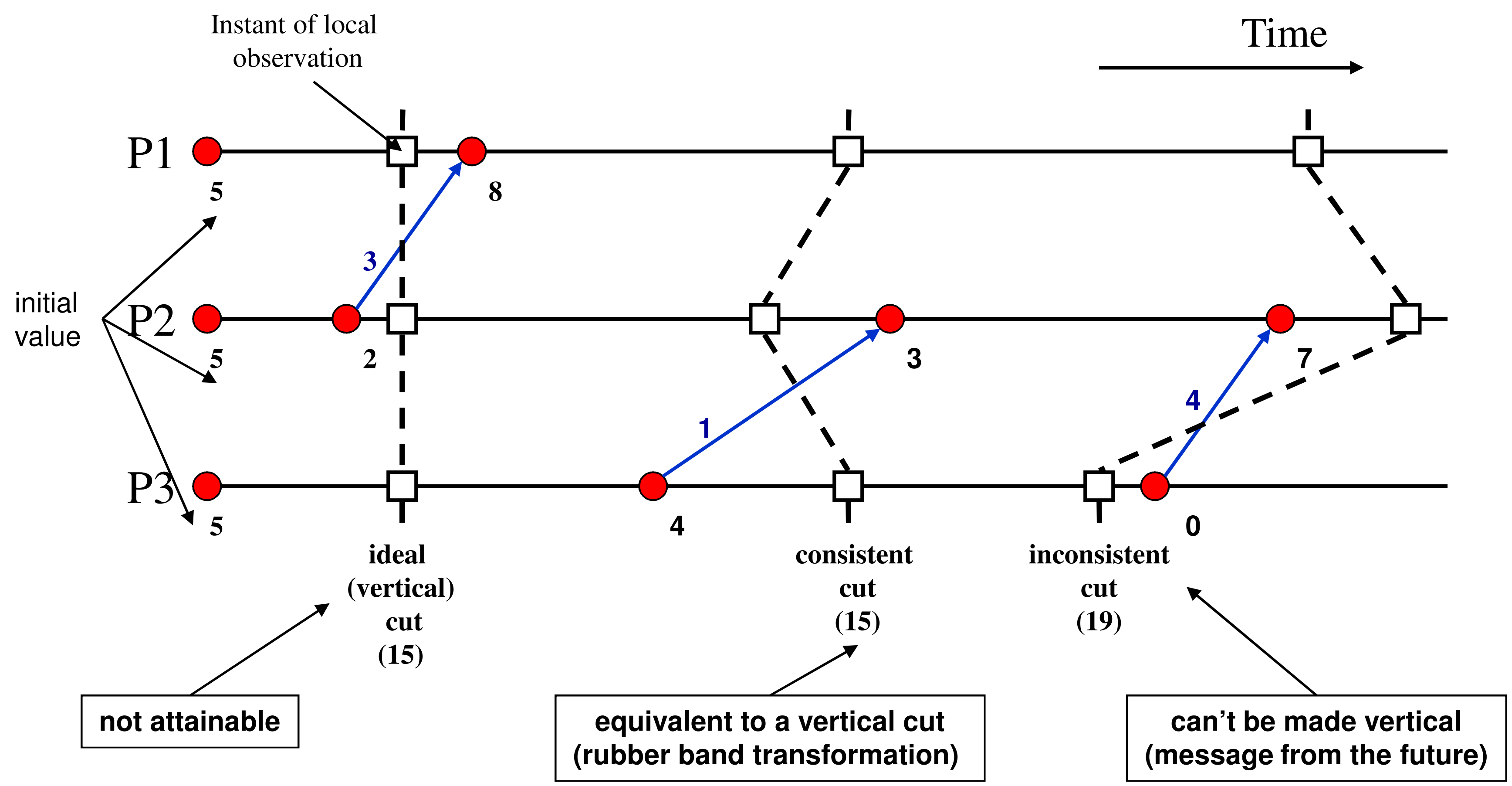
Poset Diagram



Consistent Cuts

- A cut (or time slice) is a zigzag line cutting a time diagram into 2 parts (past and future)
  - E is augmented with a cut event c_i for each process P_i: $E' = E \cup \{c_i, \ldots, c_n\}$ ∴
    - A cut C of an event set E is a finite subset $C \subseteq E$: $e \in C \wedge e' <_l e \rightarrow e' \in C$
    - A cut C_1 is later than C_2 if $C_1 \supseteq C_2$
    - A consistent cut C of an event set E is a finite subset $C \subseteq E$: $e \in C \wedge e' < e \rightarrow e' \in C$
      - i.e. a cut is consistent if every message received was previously sent (but not necessarily vice versa!)

Cuts (Summary)



"Rubber band transformation" changes metric, but keeps topology

Consistent Cuts

- Theorems
  - With operations $\cup$ and $\cap$ the set of cuts of a partially ordered event set E form a lattice
  - The set of consistent cuts is a sublattice of the set of all cuts
  - For a consistent cut consisting of cut events $c_i,\ldots,c_n$, no pair of cut events is causally related. i.e $\forall c_i,c_j \sim(c_i < c_j) \wedge \sim(c_j < c_i)$
  - For any time diagram with a consistent cut consisting of cut events $c_i,\ldots,c_n$, there is an equivalent time diagram where $c_i,\ldots,c_n$ occur simultaneously. i.e. where the cut line forms a straight vertical line
    - All cut events of a consistent cut *can occur simultaneously*

System Model for Global Snapshots

- The system consists of a collection of n processes p1, p2, ..., pn that are connected by channels.
- There are no globally shared memory and physical global clock and processes communicate by passing messages through communication channels.
- C_{ij} denotes the channel from process pi to process pj and its state is denoted by SC_{ij} .
- The actions performed by a process are modeled as three types of events:
  - Internal events,the message send event and the message receive event.
  - For a message mij that is sent by process pi to process pj , let send(m_{ij}) and rec(m_{ij}) denote its send and receive events.

Process States and Messages in transit

- At any instant, the state of process p_i , denoted by LS_i , is a result of the sequence of all the events executed by p_i till that instant.
- For an event e and a process state LS_i , $e \in LS_i$ iff e belongs to the sequence of events that have taken process p_i to state LS_i .
- For an event e and a process state LS_i , e (not in) LS_i iff e does not belong to the sequence of events that have taken process p_i to state LS_i .
- For a channel C_{ij} , the following set of messages can be defined based on the local states of the processes p_i and p_j

  Transit: $transit(LS_i , LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \ V$
  $rec(m_{ij})$ (not in) $LS_j \}$

Global States of Consistent Cuts

- The global state of a distributed system is a collection of the local states of the processes and the channels.
- A *global state* computed along a consistent cut is correct
- The *global state* of a consistent cut comprises the local state of each process at the time the cut event happens and the set of all messages sent but not yet received
- The *snapshot problem* consists in designing an efficient protocol which yields only consistent cuts and to collect the local state information
  - Messages crossing the cut must be captured
  - Chandy & Lamport presented an algorithm assuming that message transmission is FIFO

Chandy-Lamport Distributed Snapshot Algorithm

- Assumes FIFO communication in channels
- Uses a control message, called a marker to separate messages in the channels.
  - After a site has recorded its snapshot, it sends a marker, along all of its outgoing channels before sending out any more messages.
  - The marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded in the snapshot.
- A process must record its snapshot no later than when it receives a marker on any of its incoming channels.
- The algorithm terminates after each process has received a marker on all of its incoming channels.
- All the local snapshots get disseminated to all other processes and all the processes can determine the global state.

Chandy-Lamport Distributed Snapshot Algorithm

Marker receiving rule for Process Pi

*If (*Pi has not yet recorded its state*) it*

records its process state now

records the state of c as the empty set

turns on recording of messages arriving over other channels

else

Pi records the state of c as the set of messages received over c since it saved its state

Marker sending rule for Process Pi

After Pi has recorded its state,for each outgoing channel c:

Pi sends one marker message over c

(before it sends any other message over c)

Computing Global States without FIFO Assumption

- Algorithm
  - All process agree on some future virtual time s or a set of virtual time instants $s_1, \ldots s_n$ which are mutually concurrent and did not yet occur
  - A process takes its local snapshot at virtual time s
  - After time s the local snapshots are collected to construct a global snapshot
    - P_i ticks and then fixes its next time $s = C_i + (0,\ldots,0,1,0,\ldots,0)$ to be the common snapshot time
    - P_i broadcasts s
    - P_i blocks waiting for all the acknowledgements
    - P_i ticks again (setting $C_i = s$), takes its snapshot and broadcast a dummy message (i.e. force everybody else to advance their clocks to a value $\geq s$)
    - Each process takes its snapshot and sends it to P_i when its local clock becomes $\geq s$

Computing Global States without FIFO Assumption (cont)

- Inventing a n+1 virtual process whose clock is managed by P_i
- P_i can use its clock and because the virtual clock C_{n+1} ticks only when P_i initiates a new run of snapshot :
  - The first n component of the vector can be omitted
  - The first broadcast phase is unnecessary
  - Counter modulo 2
- 2 states
  - White (before snapshot)
  - Red (after snapshot)
  - Every message is red or white, indicating if it was send before or after the snapshot
  - Each process (which is initially white) becomes red as soon as it receives a red message for the first time and starts a virtual broadcast algorithm to ensure that all processes will eventually become red

Computing Global States without FIFO Assumption (cont)

- Virtual broadcast
  - Dummy red messages to all processes
  - Flood the network by using a protocol where a process sends dummy red messages to all its neighbors
- Messages in transit
  - White messages received by red process
  - Target process receives the white message and sends a copy to the initiator
- Termination
  - Distributed termination detection algorithm [Mattern 87]
  - Deficiency counting method
    - Each process has a counter which counts messages send – messages received. Thus, it is possible to determine the number of messages still in transit