Distributed Computing Concepts - Global Time and State in Distributed Systems

Prof. Nalini Venkatasubramanian

Distributed Systems Middleware - Lecture 2

Global Time & Global States 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
  - Rate of event occurrence is very high
  - Event execution times are very small
- We can only *approximate* the global view
  - *Simulate synchronous* distributed system on a given asynchronous system
  - *Simulate* a *global time* – Clocks (Physical and Logical)
  - *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

The Concept of Time in Distributed Systems

- A standard time is a set of instants with a temporal precedence order $<$ satisfying certain conditions [Van Benthem 83]:
    - Irreflexivity
    - Transitivity
    - 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
- A linearly ordered structure of time is not always adequate for distributed systems
  - Captures dependence, not independence of distributed activities
- Time as a partial order
  - A ***partially ordered system of vectors*** forming a *lattice* structure is a natural representation of time in a distributed system.

Global time in distributed systems

- An accurate notion of global time is difficult to achieve in distributed systems.
  - Uniform notion of time is necessary for correct operation of many applications (mission critical distributed control, online games/entertainment, financial apps, smart environments etc.)
- 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 synchronization is needed to simulate global time
  - Physical Clocks vs. Logical clocks
    - Physical clocks are logical clocks that must not deviate from the real-time by more than a certain amount.

We often derive causality of events from loosely synchronized clocks

Physical Clock Synchronization

Physical Clocks

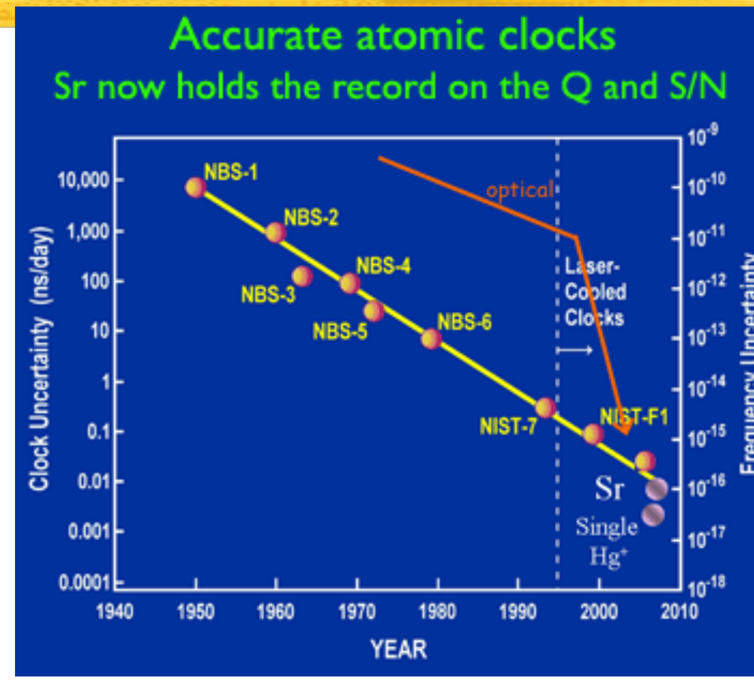
- **How do we measure real time?**
  - 17th century - Mechanical clocks based on astronomical measurements
    - Solar Day - Transit of the sun
    - Solar Seconds - Solar Day/(3600*24)
  - Problem (1940) - Rotation of the earth varies (gets slower)
  - Mean solar second - average over many days
  - UT1 -- astronomical observation at 0 deg. longitude (GMT)

Date	Duration in mean solar time
February 11	24 hours
March 26	24 hours – 18.1 seconds
May 14	24 hours
June 19	24 hours + 13.1 seconds
July 26	24 hours
September 16	24 hours – 21.3 seconds
November 3	24 hours
December 22	24 hours + 29.9 seconds

Length of apparent solar day (1998) – (*cf: wikipedia*)

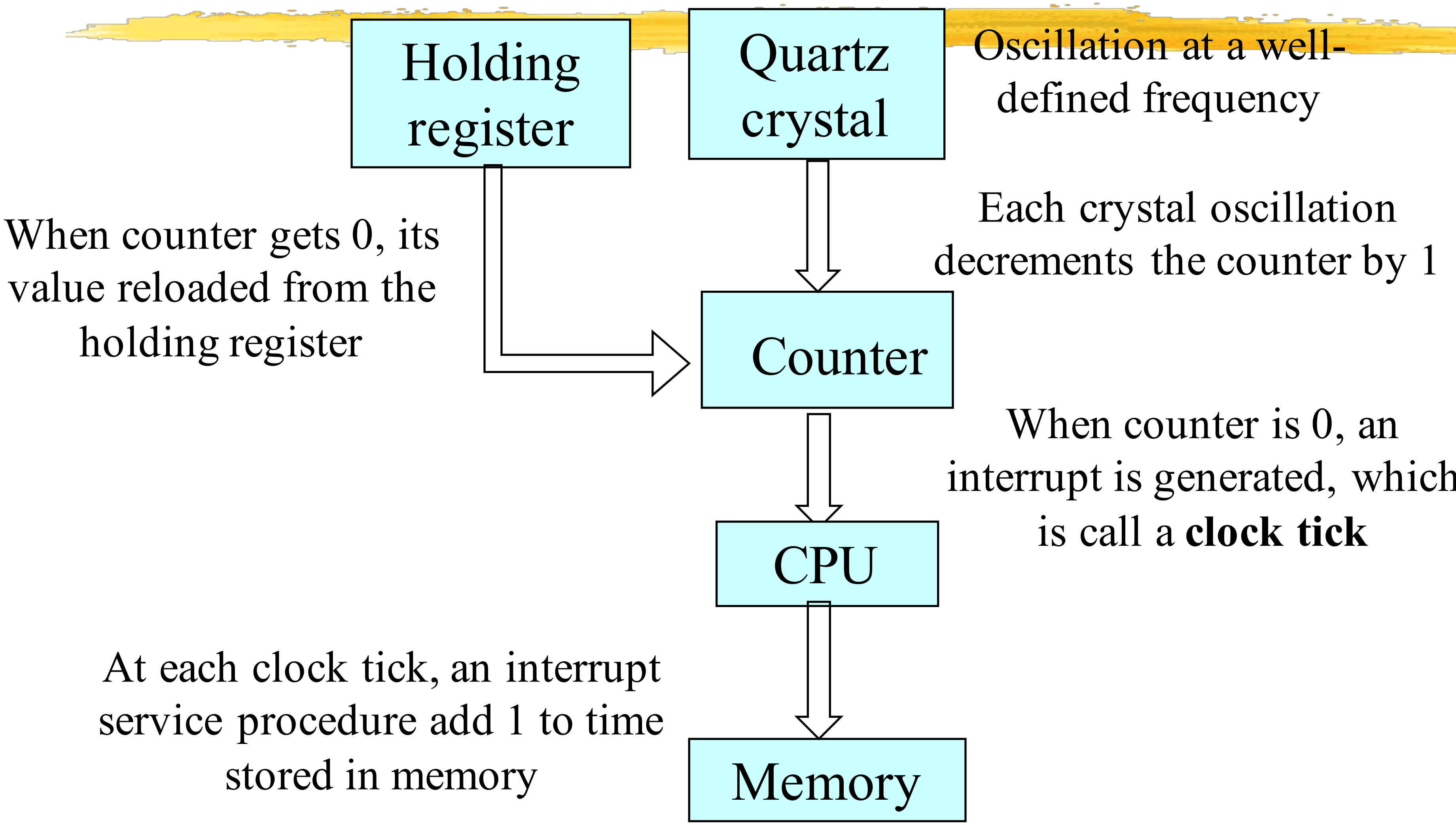
Atomic Clocks

- 1948 - Counting transitions of a crystal (Cesium 133, quartz) used as atomic clock
  - crystal oscillates at a well known frequency
- 2014 – NIST-F2 Atomic clock
  - Accuracy: ± 1 sec in 300 mil years
  - NIST-F2 measures particular transitions in Cesium atom (9,192,631,770 vibrations per second), in much colder environment, minus 316F, than NIST-F1
- TAI - International Atomic Time
  - 9,192,631,779 transitions = 1 mean solar second in 1958



UTC (Universal Coordinated Time)
From time to time, UTC skips a solar second to stay in phase with the sun (30+ times since 1958)
UTC is broadcast by several sources (satellites…)

How Clocks Work in Computers



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 RPC)
  - 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 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)

Cristian's Algorithm - Summary (cf. Princeton course)

1. Client sends a ***request*** packet, timestamped with its local clock T_1

2. Server timestamps its receipt of the request T_2 with its local clock

3. Server sends a ***response*** packet with its local clock T_3 and T_2

4. Client locally timestamps its receipt of the server's response T_4

How the client can use these timestamps to synchronize its local clock to the server's local clock?

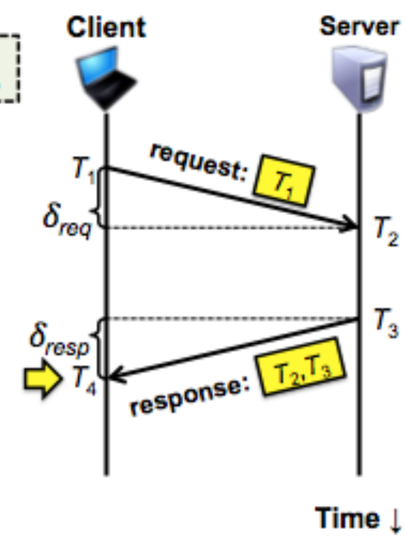
Cristian's algorithm: Offset sample calculation

Goal: Client sets clock ← $T_3 + \delta_{resp}$

- Client samples *round trip time* δ = $\delta_{req} + \delta_{resp} = (T_4 - T_1) - (T_3 - T_2)$
- **But client knows δ, not δ_{resp}**

Assume: $\delta_{req} \approx \delta_{resp}$

Client sets clock ← $T_3 + ½\delta$



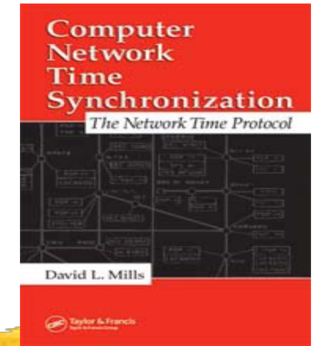
Berkeley UNIX algorithm

- One Version
  - 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.
- Another Version
  - Master/daemon uses Cristian's algorithm to calculate time from multiple sources, removes outliers, computes average and broadcasts

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.

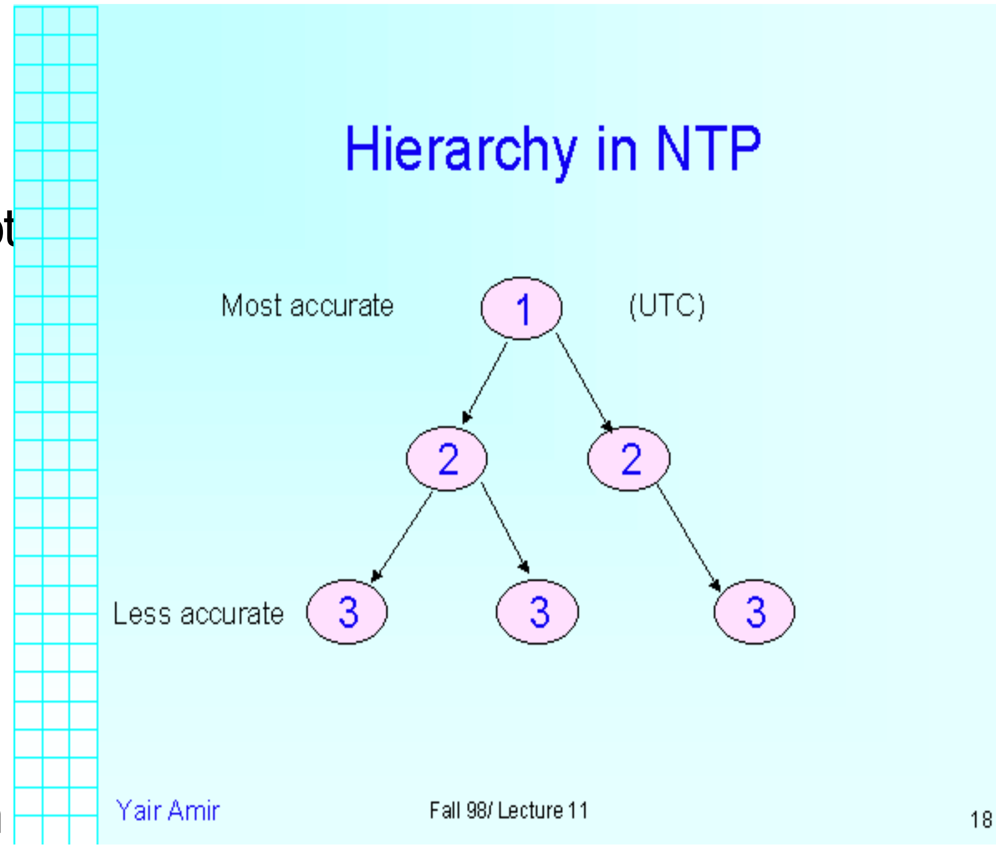
Network Time Protocol (NTP)



- Most widely used physical clock synchronization protocol on the Internet (http://www.ntp.org)
  - Currently used: NTP V3 and V4
- 10-20 million NTP servers and clients in the Internet
- Claimed Accuracy (Varies)
  - milliseconds on WANs, submilliseconds on LANs, submicroseconds using a precision timesource
  - Nanosecond NTP in progress

NTP Design

- Hierarchical tree of time servers.
  - The primary server at the root synchronizes with the UTC.
  - The next level contains secondary servers, which act as a backup to the primary server.
  - At the lowest level is the synchronization subnet which has the clients.
  - Variant of Cristian's algorithm that does not use RTT's, but multiple 1-way messages



DCE Distributed Time Service

- Software service that provides precise, fault-tolerant clock synchronization for systems in local area networks (LANs) and wide area networks (WANs).
- determine duration, perform event sequencing and scheduling.
- Each machine is either a time server or a clerk
- software components on a group of cooperating systems;
- client obtains time from **DTS entity**
- DTS entities
  - **DTS server**
  - **DTS clerk** that obtain time from DTS servers on other hosts

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
  - Periodically a clerk obtains time-intervals from several servers ,e.g. all the time servers on its LAN
  - Based on their answers, it computes a new time and gradually converges to it.
    - Compute the intersection where the intervals overlap. Clerks then adjust the system clocks of their client systems to the midpoint of the computed intersection.
    - When clerks receive a time interval that does not intersect with the majority, the clerks declare the non-intersecting value to be faulty.
    - Clerks ignore faulty values when computing new times, thereby ensuring that defective server clocks do not affect clients.

Spanner: Google's Globally Distributed Database and the TrueTime Architecture

https://youtu.be/NthK17nbpYs

Logical Clock Synchronization

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

Logical Clocks

- Used to determine causality in distributed systems
- Time is represented by non-negative integers
- Event structures represent distributed computation (in an abstract way)
  - 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 Message
    - Receive Message
    - Internal (change of state)

Logical Clocks

- 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
  - $C: E \to T$ :: T is a partially ordered set : $e < e' \to C(e) < C(e')$ holds
- Consequences of the clock condition [Morgan 85]:
  - Events occurring at a particular process are totally ordered by their local sequence of occurrence
    - 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
    - Each receive event has a corresponding send event
    - Future can not influence the past (causality relation)

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)

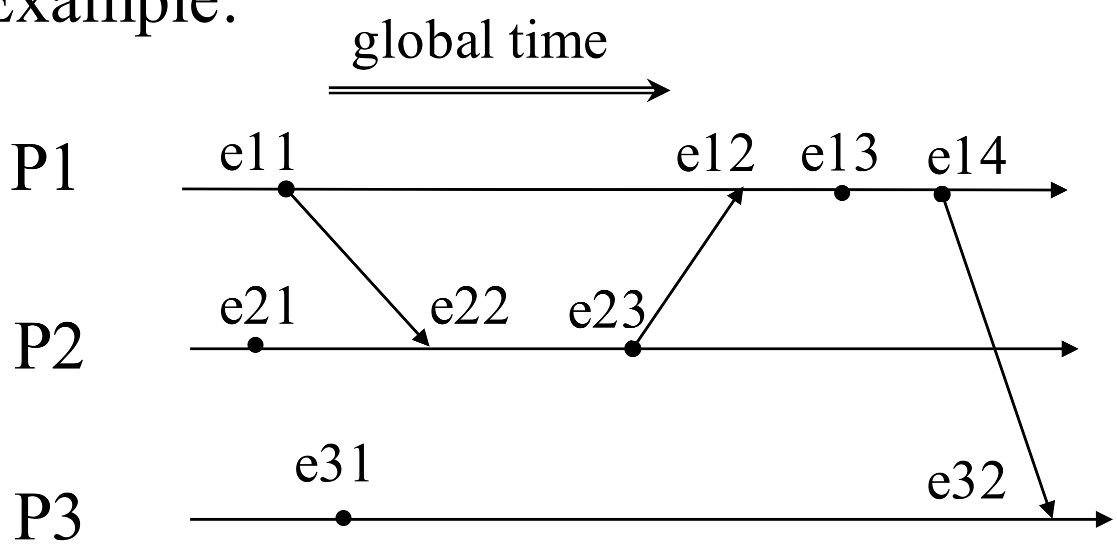
Event Ordering- *the example*

Processor Order: e precedes e′ in the same process

Send-Receive: e is a send and e′ is the corresponding receive

Transitivity: exists e″ s.t. e < e″ and e″< e′

Example:



Program order: e13 < e14
Send-Receive: e23 < e12
Transitivity: e21 < e32

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

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 e_i and e_j, $C(e_i) < C(e_j)$ does not imply $e_i \rightarrow e_j$.
- 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 ~(e<e′)$\wedge$~(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 ~(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)

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 – loses information
- Mapping partial ordered events onto a linearly ordered set of integers 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 Clocks

- Independently developed by Fidge, Mattern and Schmuck.
- Aim: To construct a mechanism by which each process gets an optimal approximation of global time
- Time representation
  - 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 Pj and 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

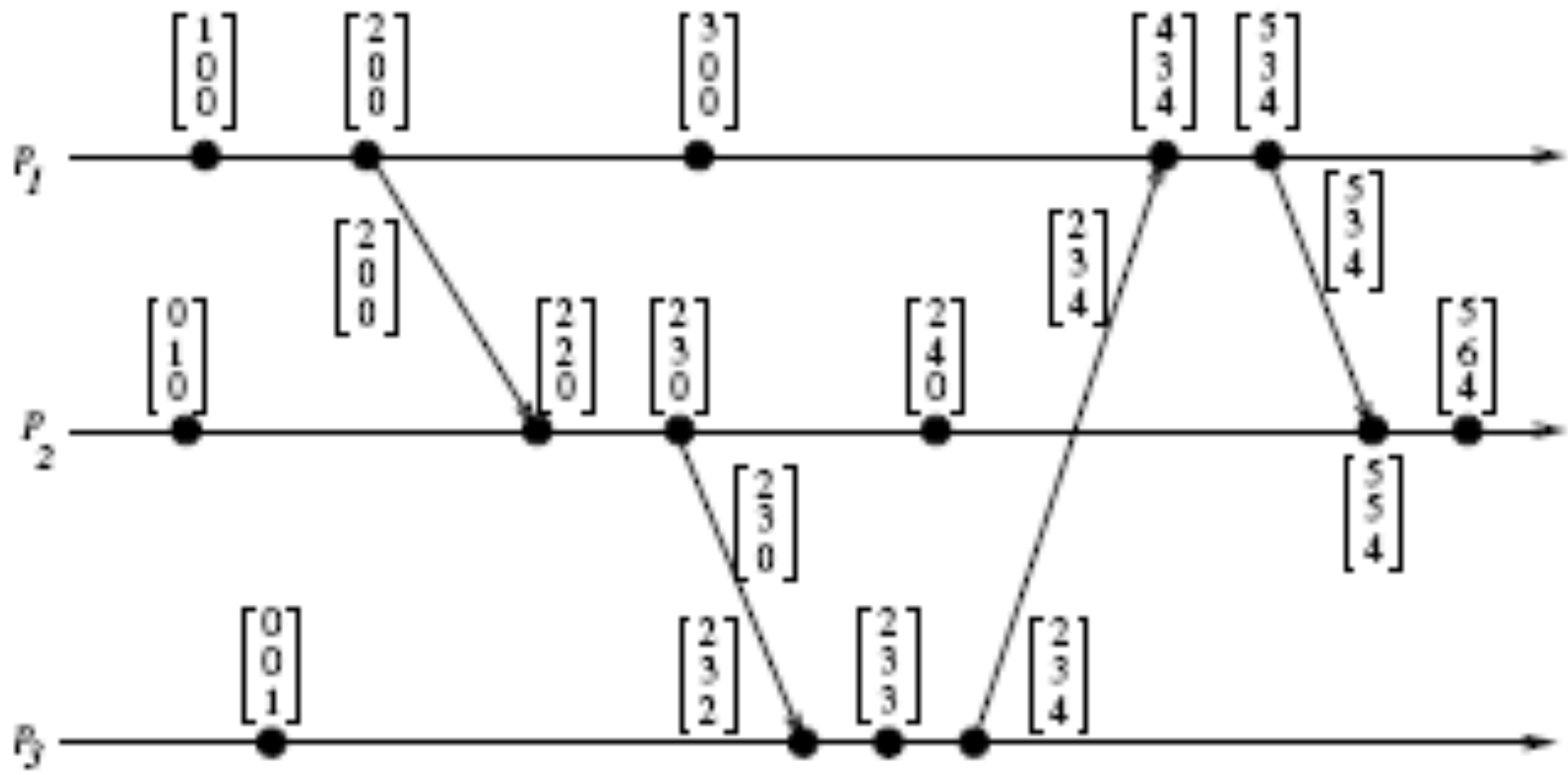


Figure 3.2: Evolution of vector time.

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 $\forall i,j$: $C_i[i] \geq C_j[i]$

Structure of the Vector Time

- For two time vectors u,v
  - $u \le v$ iff $\forall i: u[i] \le v[i]$
  - $u < v$ iff $u \le v \wedge u \ne v$
  - $u || v$ iff $\sim(u<v) \wedge \sim(v<u)$ :: || is not transitive
- For an event set E,
  - $\forall e,e' \in E: e<e'$ 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

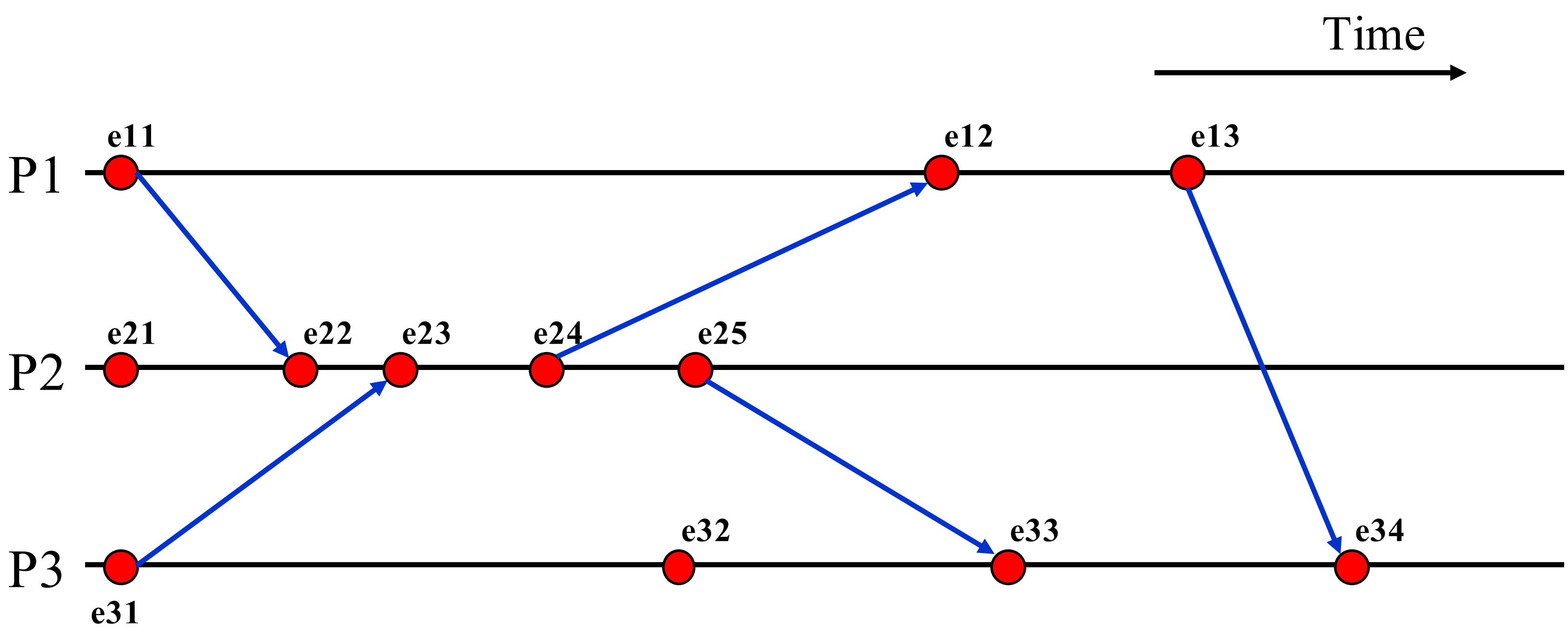
Time Manager Operations

- 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)

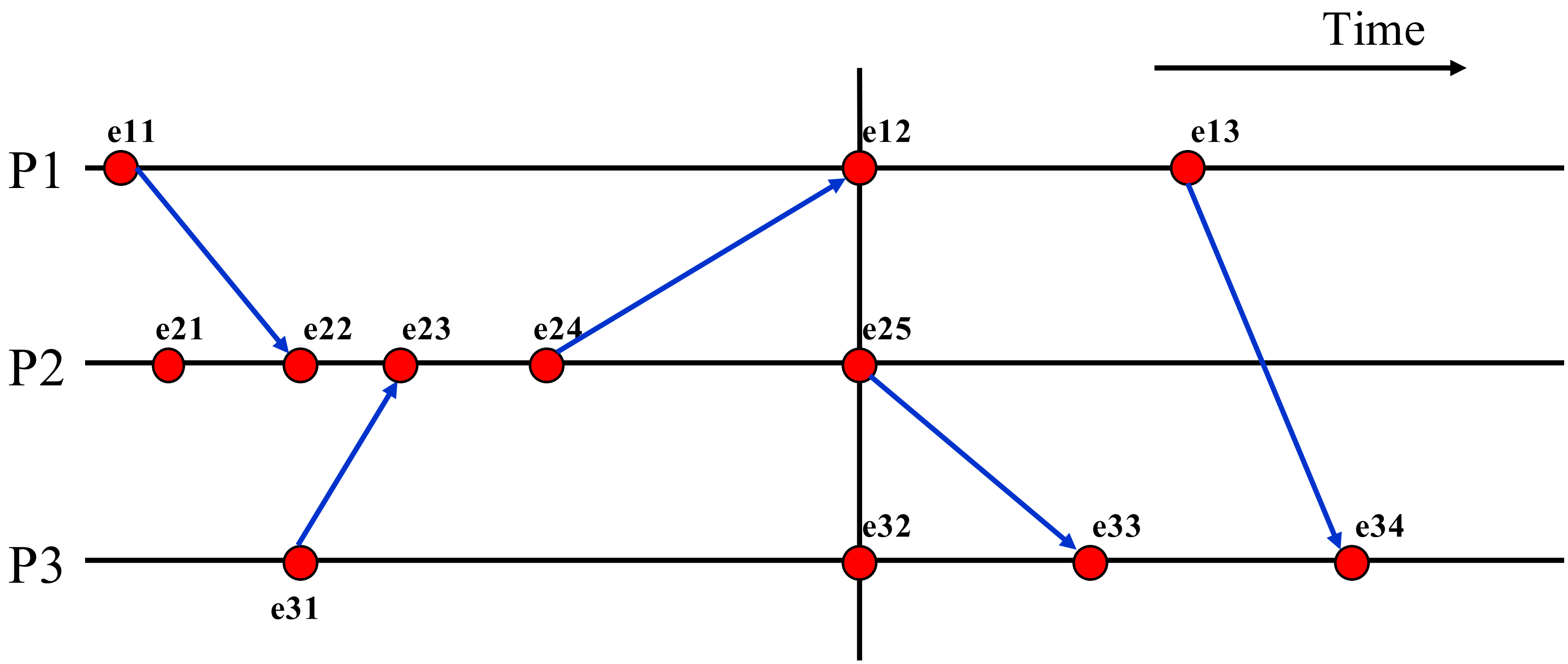
Simulate A Global State

- 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)

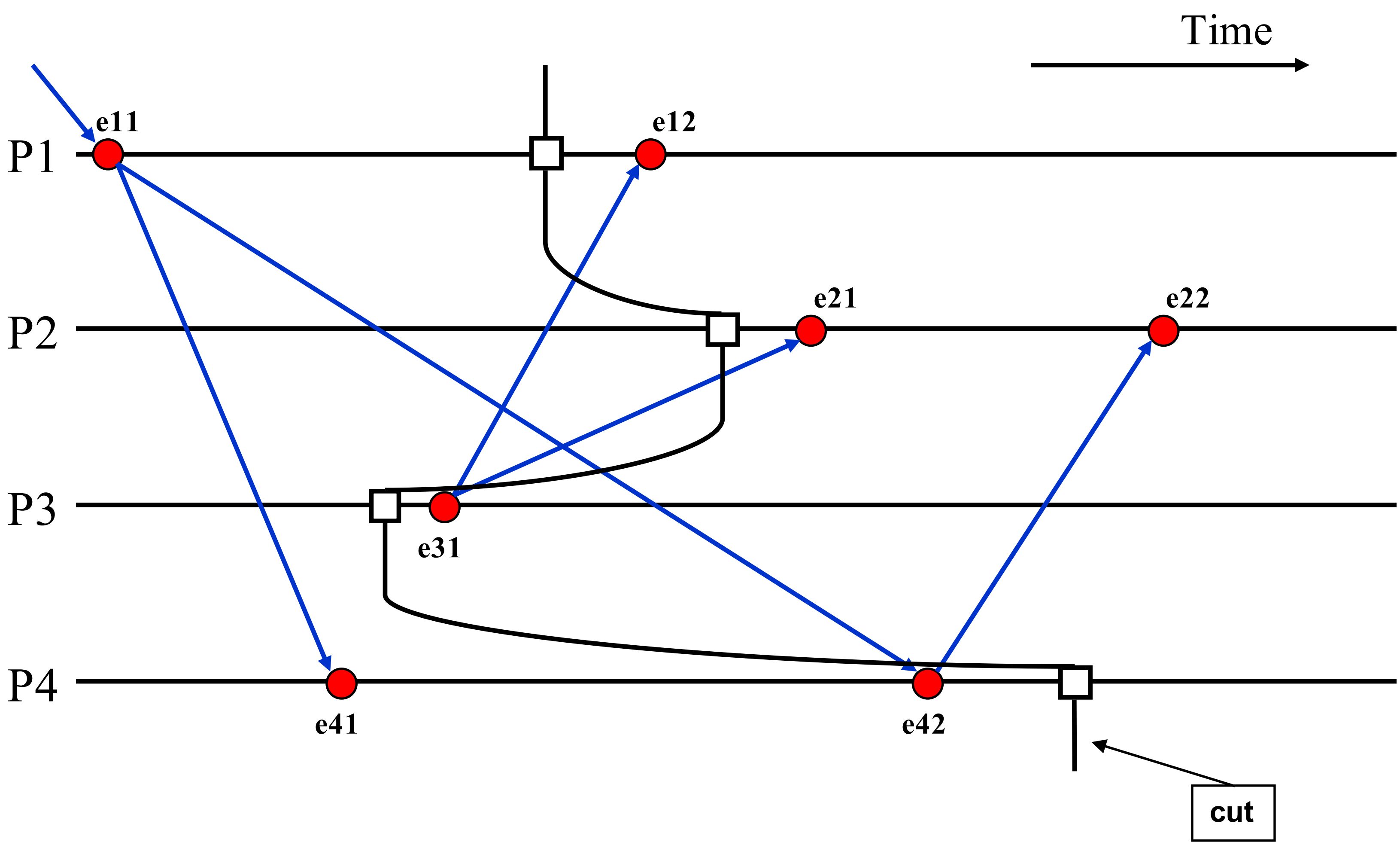
Event Diagram



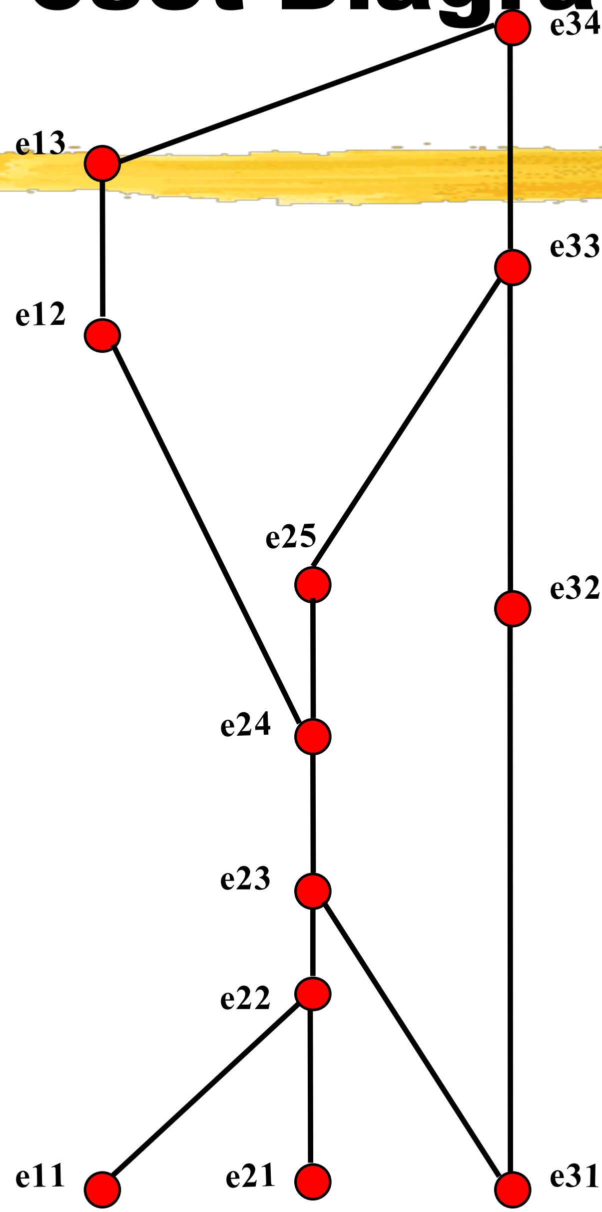
Equivalent Event Diagram



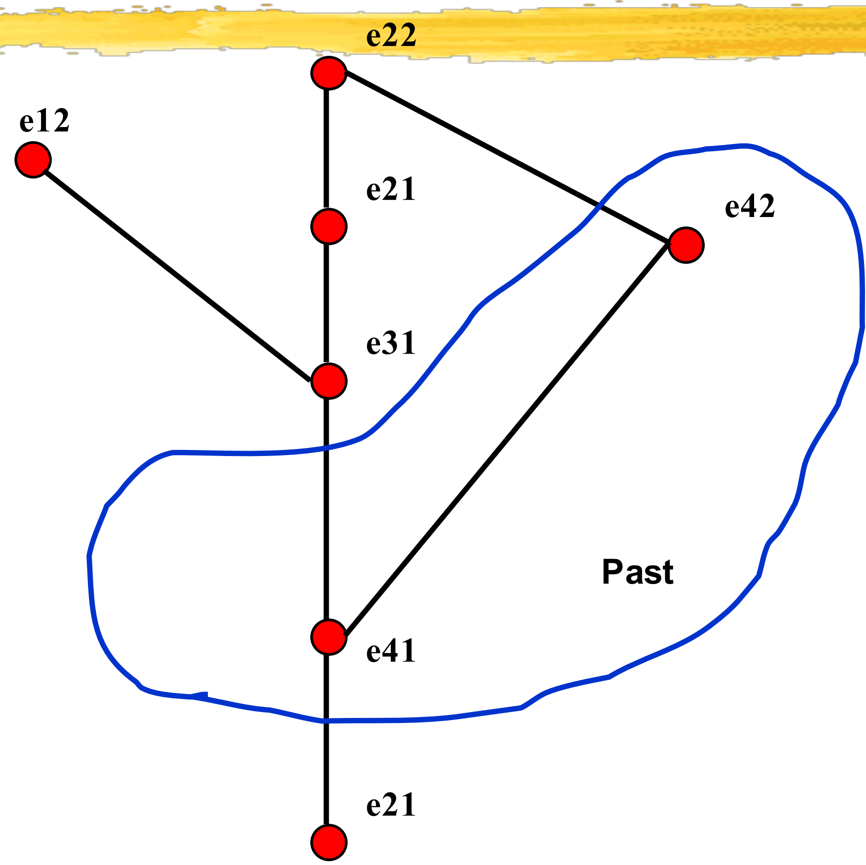
Rubber Band Transformation



Poset Diagram



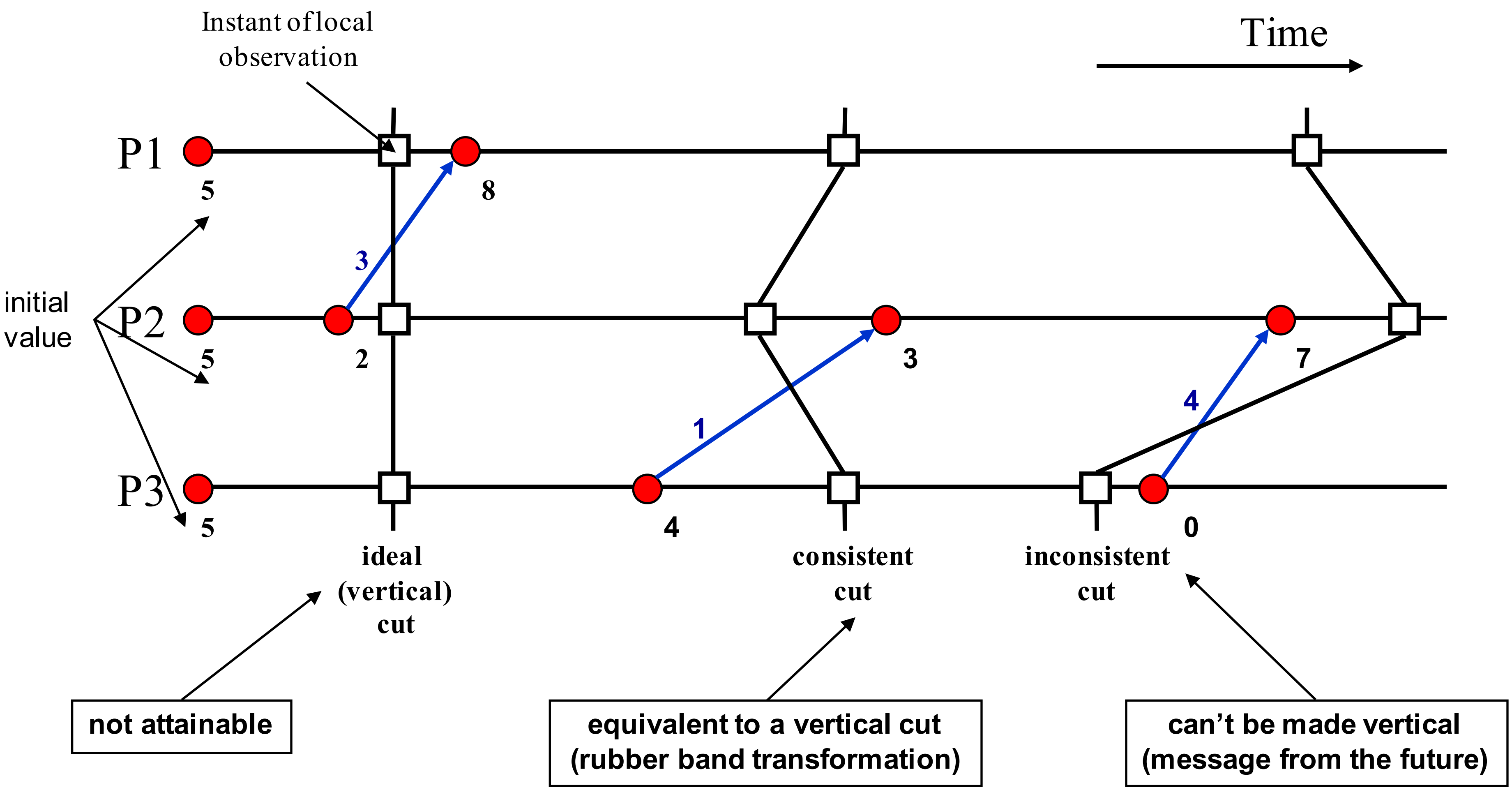
Poset Diagram



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:$E' = E \cup \{c_i,\ldots,c_n\}$ $\therefore$
    - 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)



Consistent Cuts

- Some Theorems
  - 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***

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

System Model for Global Snapshots

- The system consists of a collection of n processes p1, p2, ..., pn that are connected by channels.
- There is no globally shared memory and physical global clock and processes communicate by passing messages through communication channels.
- No failures in the system.
- Messages are not lost, duplicated or altered.
- 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 pi , denoted by LSi , is a result of the sequence of all the events executed by pi till that instant.
- For an event e and a process state LSi , e∈LSi iff e belongs to the sequence of events that have taken process pi to state LSi .
- For an event e and a process state LSi , e (not in) LSi iff e does not belong to the sequence of events that have taken process pi to state LSi .
- For a channel Cij , the following set of messages can be defined based on the local states of the processes pi and pj

  Transit: transit(LSi , LSj) = {mij |send(mij) ∈ LSi V
  rec(mij) (not in) LSj }

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

Initiator Process

Records its process state

Creates a special "marker" message

Sends a "marker" message to all outgoing processes

Turns on recording of messages arriving over all incoming channels

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 - Lai-Yang Algorithm

- Uses a ***coloring*** scheme that works as follows
  - White (before snapshot); Red (after snapshot)
  - Every process is initially white and turns red while taking a snapshot. The equivalent of the "Marker Sending Rule" (virtual broadcast) is executed when a process turns red.
  - Every message sent by a white (red) process is colored white (red).
  - Thus, a white (red) message is a message that was sent before (after) the sender of that message recorded its local snapshot.
  - Every white process takes its snapshot at its convenience, but no later than the instant it receives a red message.

Computing Global States without FIFO Assumption - Lai-Yang Algorithm (cont.)

- Every white process records a history of all white messages sent or received by it along each channel.
- When a process turns red, it sends these histories along with its snapshot to the initiator process that collects the global snapshot.
- Determining Messages in transit (i.e. White messages received by red process)
  - The initiator process evaluates transit(LSi, LSj) to compute the state of a channel Cij as given below:
  - SCij = {white messages sent by pi on Cij − white messages received by pj on Cij}
  - $= \{ send(M_{ij}) | send(m_{ij}) \in LS_i \} - \{ rec(m_{ij}) | rec(m_{ij}) \in LS_j \}$.

Computing Global States without FIFO Assumption: Termination

- First method
  - Each process I keeps a counter cntri that indicates the difference between the number of white messages it has sent and received before recording its snapshot, i.e number of messages still in transit.
  - It reports this value to the initiator along with its snapshot and forwards all white messages, it receives henceforth, to the initiator.
  - Snapshot collection terminates when the initiator has received Σi cntri number of forwarded white messages.
- Second method
  - Each red message sent by a process piggybacks the value of the number of white messages sent on that channel before the local state recording. Each process keeps a counter for the number of white messages received on each channel.
  - Termination – Process receives as many white messages on each channel as the value piggybacked on red messages received on that channel.

Computing Global States without FIFO Assumption: Mattern's Algorithm

- Uses Vector Clocks
  - 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 (Mattern 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 components of the vector can be omitted
  - The first broadcast phase is unnecessary
  - Counter modulo 2
- Termination
  - Distributed termination detection algorithm [Mattern 87]