# Time and Global States

From **Coulouris, Dollimore, and Kindberg**
Updated and revised by **Kulik & Tanin**

# Introduction

**Time is important**

- Auditing
- Authentication
- Consistency

**Yet, there is no global clock in a distributed system**

# Introduction

**Each computer has its own internal clock**

- used by local processes to obtain the value of the current time
- processes on different computers can timestamp their events
- but clocks on different computers may give different times
- computer clocks drift from perfect time and their drifting rates may differ from one another

**Even if clocks on all computers are set to the same time, their clocks will eventually vary quite significantly unless corrections are applied**

# Assumptions

**A Distributed System (DS) of *N* processes**

**Each on a single processor with its own physical clock**

**No shared memory**

**Each process *p* has a state *s* at a given time**

- State depends on internal variable values, files it works on, etc

**Processes can only communicate via messages**

**Events in a process can be ordered**

- i.e., $e \rightarrow_i e'$

**Multi-threading**

- Cannot change this: we are on a single processor for each process

**History of a process**

- $h_i = \langle e^0_i, e^1_i, \ldots \rangle$

# Clocks

**Hardware clock of a system is**

- $H_i(t)$

**Software clock is a scaled and offset added version**

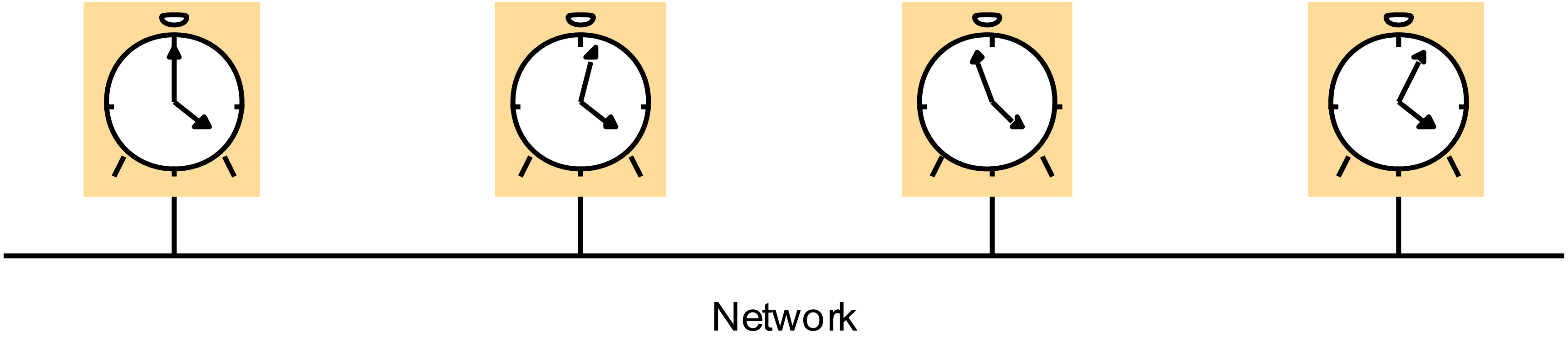
- $C_i(t) = \alpha H_i(t) + \beta$

**This is an approximation of the real physical time that a process on this machine can see**

***t* is the absolute frame of reference for time and is commonly different (hopefully slightly) then *C***

**Two processes are happening at two different times if there is at least a period of time that can be measured with the resolution of the clocks available**

# Skew for Clocks in a Distributed System




- Clocks can also drift even if they were set perfectly to be the same at the beginning
- The speed of an individual clock can also change, e.g., with temperature

# Coordinated Universal Time

**Computer clocks can be synchronized to external sources**

**The most known accurate ones use atomic oscillators with drift rate 1 in $10^{13}$**

- 1 sec is 9,192,631,770 periods of transition between the two hyperfine levels of the ground state of $Cs^{133}$

**Humans historically used astronomical time but it varies as Earth varies its state (e.g., tides)**

**Coordinated Universal Time is atomic time that is adjusted (rarely) to astronomical time (UTC)**

**UTC is broadcasted all over the world to synchronize their time (with some minor error in this sync process)**

# External & Internal Synchronization

## External synchronization

- Use an external time server to synchronize process times
- For a synchronization bound $D > 0$ and for a source $S$ of UTC time: $|S(t) - C_i(t)| < D$ for $i = 1, 2, ..., N$ and for all real times $t$ in interval $I$
- The clocks Ci are accurate to within the bound D

## Internal synchronization

- For a synchronization bound $D > 0$: $|C_i(t) - Cj(t)| < D$ for $i = 1, 2, ..., N$ and for all real times $t$ in interval $I$
- The clocks $C_i$ agree within the bound $D$
- Once synchronized, processes can communicate
- If all clocks drift then all can become different than the initial external time service

# Faulty Clocks

***Monotonicity* condition**

- A clock always advances: $t' > t \Rightarrow C(t') > C(t)$

**We can also bound/correct the drift of a clock**

***Faulty clock***

- A clock that do not obey the monotonicity condition
- and/or the bounds on its drift

**A clock is said to had a *crash failure* if it totally stops running, else it is said to have an *arbitrary failure***

**A correct clock is not necessarily an accurate clock!**

# Synchronization in a Synchronous System

**A synchronous distributed system is one in which the following bounds are defined**

- Time to execute each step of a process has known lower and upper bounds
- Each message transmitted over a channel is received within a known bounded time
- Each process has a local clock whose drift rate from real time has a known bound

**In theory if a clock has time t and send a message m to another then the other clock should set its time to *t* + Transmission_time**

- This does not work as the transmission time varies and is unknown
- Only the minimum transition time (min) can be known
- In some cases the maximum transition time (max) can also be found

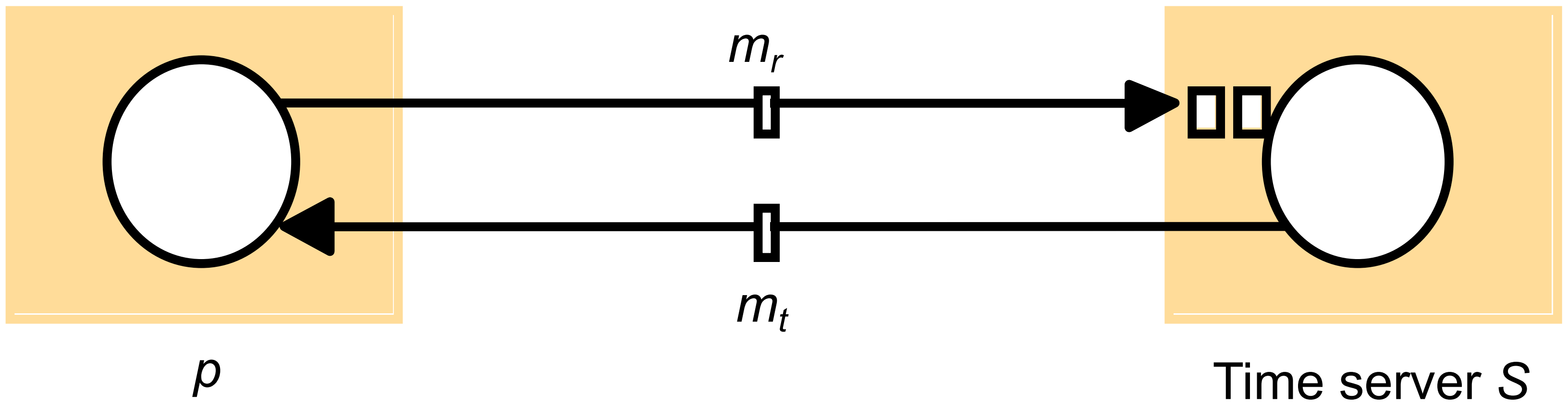
**Optimal point to set clocks on a network**

- 2 clocks: $t$ + (max + min)/2
- $N$ clocks optimal bound on clock skew is $u(1-1/N)$ where $u$ = max – min
- This cannot work for the Internet!

# Cristian's Method: Asynchronous System

## A time server *S* receives signals from a UTC source

- Process *p* requests time in $m_r$ and receives *t* in $m_t$ from *S*
- *p* sets its clock to $t + T_{round}/2$
- Accuracy $\pm$ ($T_{round}/2 - min$):
  - The earliest time *S* puts *t* in message $m_t$ is *min* after *p* sent $m_r$
  - The latest time was *min* before $m_t$ arrived at *p*
  - The time by *S*'s clock when $m_t$ arrives is in the range [$t+min$, $t + T_{round} - min$]



# Berkeley Algorithm

**Cristian's algorithm**

- A single time server might fail, need to use of a group of synchronized servers
- It does not deal with faulty servers

**Berkeley algorithm**

- An algorithm for internal synchronization of a group of computers
- A master polls to collect clock values from the others (slaves)
- The master uses round trip times to estimate the slaves' clock values
- It takes an average (eliminating any above some average round trip time or with faulty clocks)
- It sends the required adjustment to the slaves (why not the updated times?)

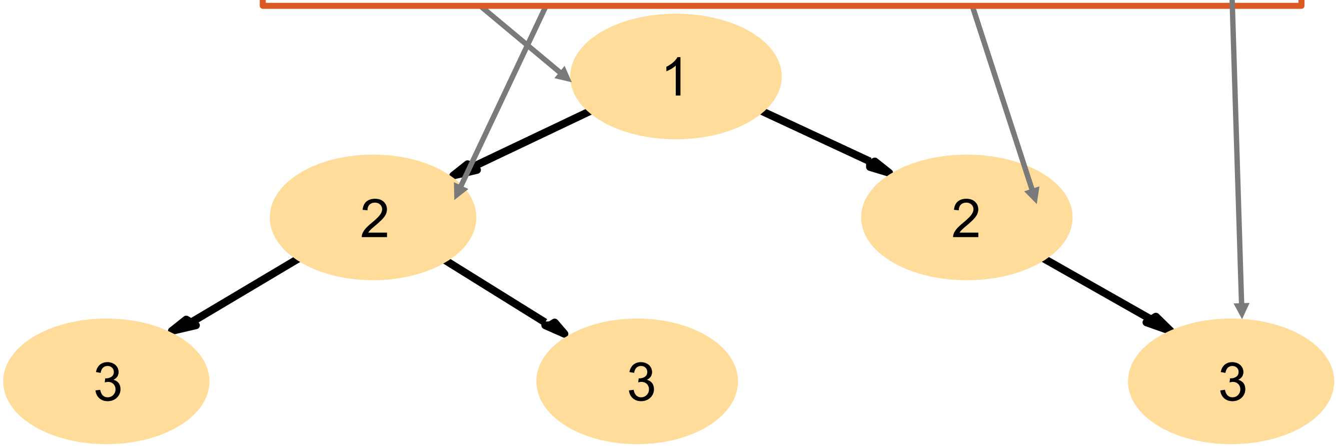
# Network Time Protocol (NTP)

**A time service for the Internet - synchronizes clients to UTC**

Primary servers are connected to UTC sources

Secondary servers are synchronized to primary servers

Synchronization subnet - lowest level servers in users' computers




Reliability from redundant paths, scalable, authenticates time sources

# NTP: Synchronisation of Servers

**The synchronization subnet can reconfigure if failures occur, e.g.:**

- A primary clock that loses its UTC source can become a secondary
- A secondary that loses its primary can use another primary

**Modes of synchronization:**

**Multicast**

- A server within a high speed LAN multicasts time to others which set clocks assuming some delay (not very accurate)

**Procedure call**

- A server accepts requests from other computers (like Cristian's algorithm). Higher accuracy. Useful if no hardware multicast.

**Symmetric**

- Pairs of servers exchange messages containing time information
- Used where very high accuracies are needed (e.g. for higher levels)

# Message Exchange for a Pair of NTP peers

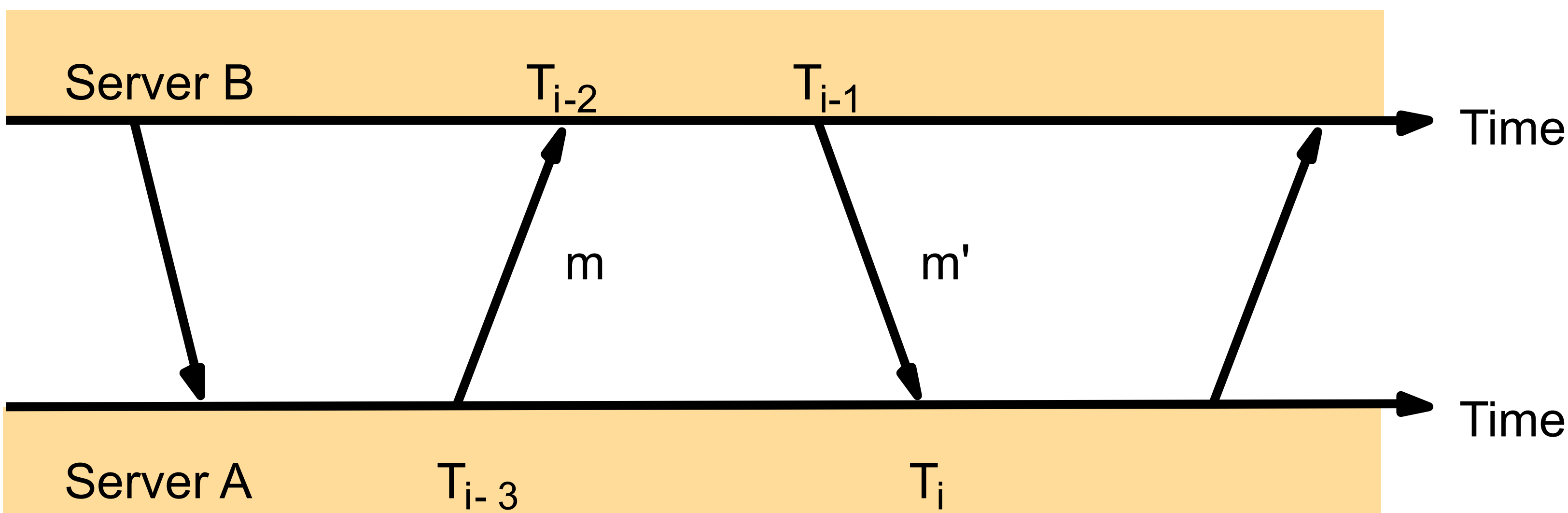
**All modes use UDP**

**Each message bears timestamps of recent events:**

- Local times of Send and Receive of previous message
- Local times of Send of current message

**Recipient notes the time of receipt $T_i$ (we have $T_{i-3}$, $T_{i-2}$, $T_{i-1}$, $T_i$)**

**In symmetric mode there can be a non-negligible delay between messages**



# Accuracy of NTP

**For each pair of messages between two servers, NTP estimates an offset *o*, between the two clocks and a delay $d_i$ (total time for the two messages, which take *t* and *t'* )**

$T_{i-2} = T_{i-3} + t + o$ and $T_i = T_{i-1} + t' - o$

**This gives us (by adding the equations) :**

$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$

**Also (by subtracting the equations)**

$o = o_i + (t' - t)/2$ where $o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$

**Using *t*, *t'* ≥ 0 it can be shown that**

$o_i - d_i/2 \leq o \leq o_i + d_i/2$

$o_i$ is an estimate of the offset and $d_i$ is a measure of the accuracy

**NTP servers filter pairs <$o_i$, $d_i$>, estimating reliability from variation, allowing them to select peers**

**In general, higher level peers are preferred as they are closer to the UTC**

# Logical Time & Clocks (Lamport 1978)

## Idea of a logical time

- Absolute order in physical time is not necessary
- But the causality relationships between events has to be preserved
- Use logical times to express causal order

## Local events

- Ordered in time for each process
- Logical times of all events have to respect all dependencies between events

## Order of two events in a distributed system

- Global relation, called *happened-before*, denoted by $\rightarrow$
- *happened-before* $\rightarrow$ is based on a local (and thus easily observable) local *happened-before* relation $\rightarrow_p$ within a process *p*

# *Happened-Before* Relation

**Definition (*happened-before*, $\rightarrow$)**

- Let $a$, $b$, $c$ be three events. Then, the following global happened-before orders hold:

  HB1: If $\exists$ process $p$: $a \rightarrow_p b$, then $a \rightarrow b$

  HB2: For any message $m$: $a$ = send($m$) $\rightarrow$ b = receive($m$)

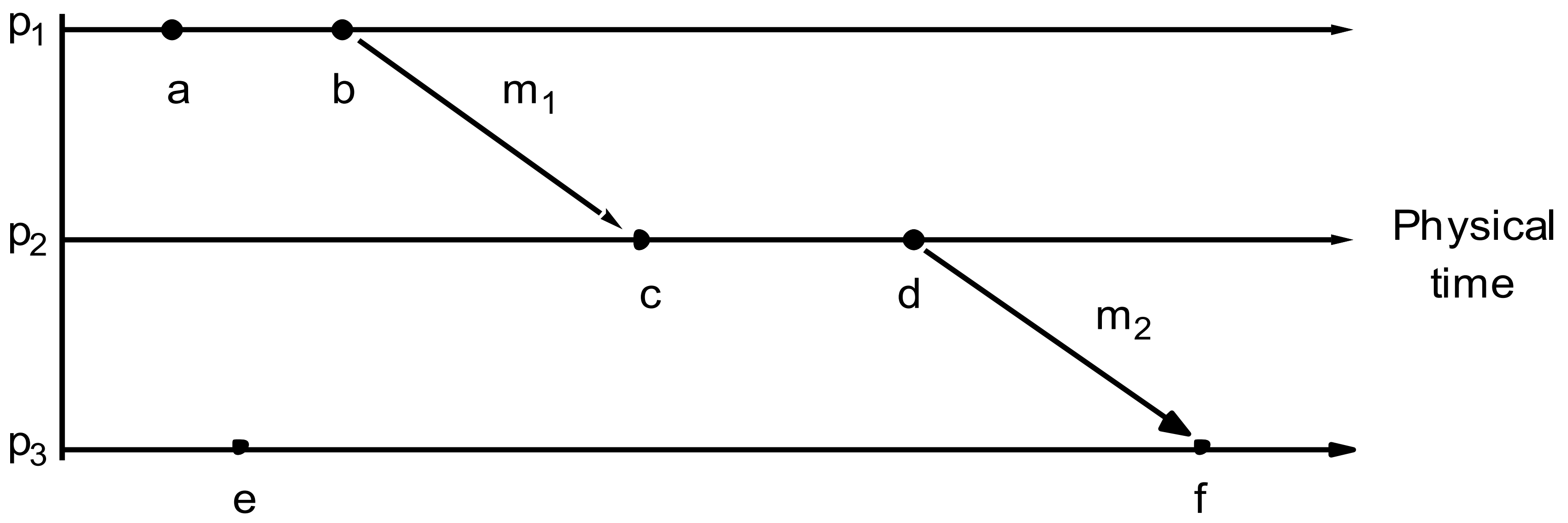
  HB3: If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

**$\rightarrow$ is a partial order**

- If two events $a$ and $b$ happen in different processes which do not exchange messages, then a and b are not ordered with respect $\rightarrow$ that is neither $a \rightarrow b$ nor $b \rightarrow a$ holds

# A Simple Example

- $a \rightarrow b$ (at $p_1$), $c \rightarrow d$ (at $p_2$)
- $b \rightarrow c$ because of $m_1$, $d \rightarrow f$ because of $m_2$
- $a$ and $e$ *are not related by* $\rightarrow$ (different processes and no chain of messages to relate them)
- $a$ and $e$ are concurrent; write as $a \parallel e$



# Lamport's Logical Clocks

**Apply logical timestamps $L_i$ to events for each process $p_i$**

- LC1

  $L_i$ is incremented by 1 before each event at process $p_i$
- LC2:

  When process $p_i$ sends message *m*, it piggybacks $t = L_i$

  When $p_j$ receives *(m,t)* it sets $L_j := max(L_j, t)$ and applies LC1 before time stamping the event *receive*(*m*)
- Each process has its logical clock initialized to zero
- $e \rightarrow e'$ implies $L(e) < L(e')$
- *However,* $L(e) < L(e')$ does not imply $e \rightarrow e'$

# Lamport Clock In-Class Example

**Given processes $p_0$, $p_1$, $p_2$**

**$s_i$ and $r_i$ are corresponding send and receive events**

**Consider the following sequences of events**

- $p_0$: $a\ s_1\ r_3\ b$
- $p_1$: $c\ r_2\ s_3$
- $p_2$: $r_1\ d\ s_2\ e$

**Provide all events with Lamport's clock values**

# Important Note

**Note that the happened-before relation does not state anything about who caused what**

- An event occuring earlier does not mean that it’s the cause of a later event

# Vector Clocks

## Generating vector clock time stamps

- Vector clock $V_i$ at process $p_i$ is an array of $N$ integers
- VC1: set $V_i[j] = 0$ for $i, j = 1, 2, \ldots, N$
- VC2: before $p_i$ timestamps an event it sets $V_i[i] := V_i[i] + 1$
- VC3: $p_i$ piggybacks $t = V_i$ on every message it sends
- VC4: when $p_i$ receives $(m, t)$ it sets $V_i[j] := \max(V_i[j], t[j])$, $j = 1, 2, \ldots, N$ (and adds 1 before the next event to its own element using VC2)

## Properties

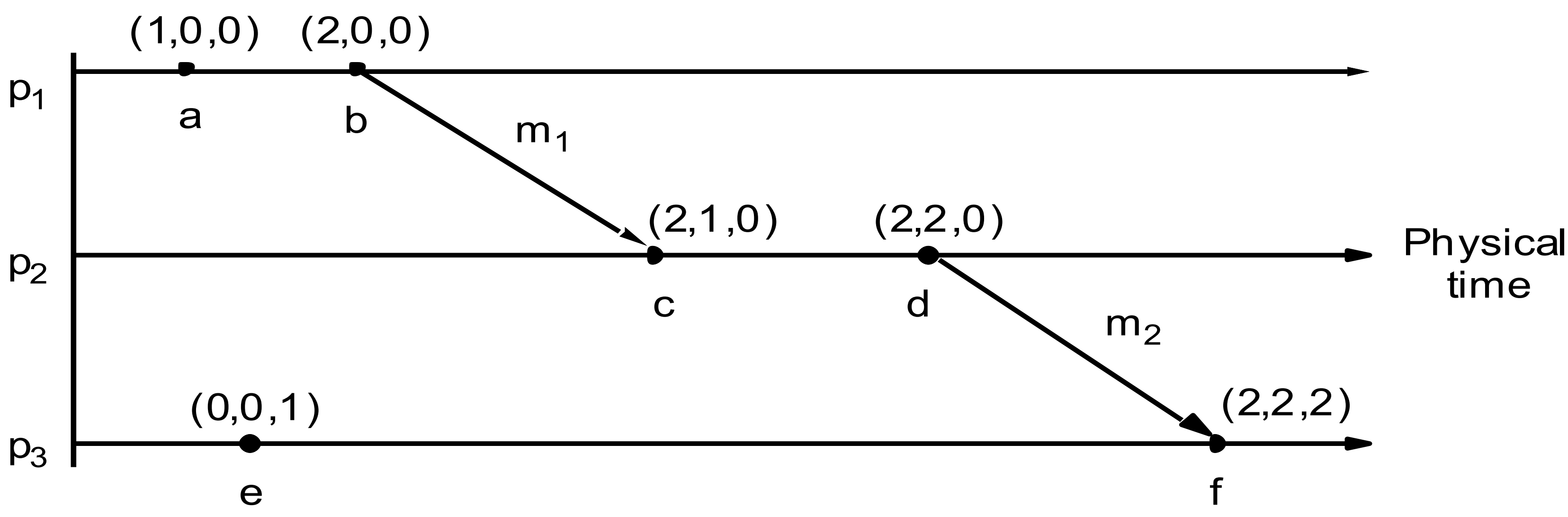
- Vector timestamps are used to timestamp local events

## Vector time stamp comparison

- $V = V'$ iff $V[j] = V'[j]$ for $j = 1, 2, \ldots, N$
- $V <= V'$ iff $V[j] <= V'[j]$ for $j = 1, 2, \ldots, N$
- $V < V'$ iff $V <= V'$ and $V != V'$
- $e \rightarrow e'$ iff $V(e) < V(e')$

# Vector Clocks: An Example

- At $p_1$: a (1,0,0), b (2,0,0), then piggyback (2,0,0) on $m_1$
- At $p_2$: on receipt of $m_1$ get max((0,0,0), (2,0,0)) = (2,0,0) add 1 to own element = (2,1,0)
- c || e (parallel) because neither V(c) <= V(e) nor V(e) <= V(c)



# Vector Clock In-Class Example

**Given processes $p_0$, $p_1$, $p_2$**

**$s_i$ and $r_i$ are corresponding send and receive events**

**Consider the following sequences of events**

- $p_0$: $a\ s_1\ r_3\ b$
- $p_1$: $c\ r_2\ s_3$
- $p_2$: $r_1\ d\ s_2\ e$

**Provide the Vector clock values for all events**

# Time & Clocks in Distributed Systems

## Summary

- Accurate timekeeping is important for distributed systems
- Synchronization of clocks is possible in spite of their drift and the variability of message delays
- Clock synchronization is not always practical for ordering of an arbitrary pair of events at different computers
- happened-before relation is a partial order on events that reflects a flow of information between them
- Lamport clocks are counters that are updated according to the happened-before relationship between events
- Vector clocks are an improvement on Lamport clocks: two events are ordered by happened-before or are concurrent by comparing their vector timestamps

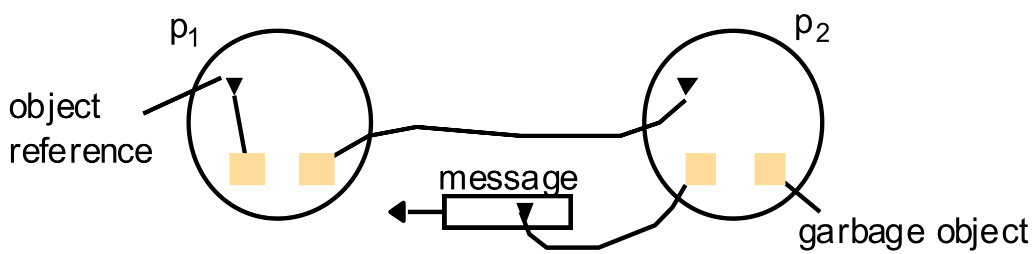
# Global States I

## Aim

- Determine whether a particular property is true of a distributed system as it executes
- Use logical time to construct a global view of the system state

## Distributed garbage collection

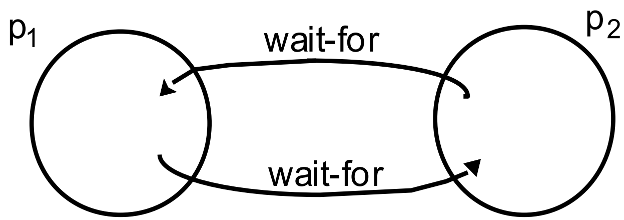
- Are there references to an object anywhere in the system? References may exist at the local process, at another process, or in the communication channel.



# Global States II

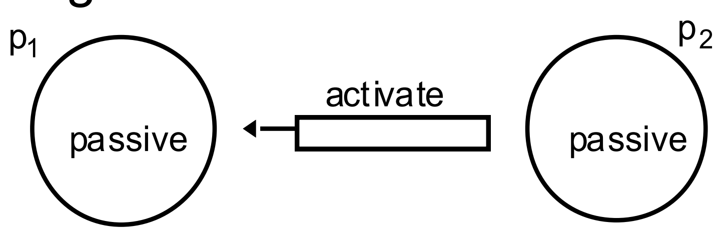
## Distributed deadlock detection

- Is there a cycle in the graph of the "waits for" relationship between processes?




## Distributed termination detection

- Has a distributed algorithm terminated?




## Distributed debugging

- Given two processes $p_1$ and $p_2$ with variables $x_1$ and $x_2$ respectively, can we determine whether the condition $|x_1 – x_2| > \delta$ is ever true?

# Distributed Debugging

**A particularly difficult problem**

**Demonstrates clearly the need to observe a global state (i.e.: to debug)**

**Can we assemble a global state from local states recorded at different times?**

**Remember:**

- Physical time cannot be perfectly synchronized in a distributed system
- It is not possible to gather the global state of the system at a particular time.

# Histories

**History of process $p_i$ was defined as**

- $h_i = \langle e^0_i, e^1_i, \ldots \rangle$

**A prefix of a history is defined as**

- $h^k_i = \langle e^0_i, e^1_i, \ldots, e^k_i \rangle$

**A global history of a system of $N$ processes $p_0$ to $p_{N-1}$**

- Union of the individual process histories:

  $H = h_0 \cup h_1 \cup \ldots \cup h_{N-1}$

**Global state**

- Take the set of states of the individual processes: $S = (s_0, s_1, \ldots, s_{N-1})$

**Cuts**

- "Assemble a meaningful global state from local states recorded at different times"

# Cuts

## Record the state

- All processes record sending and receiving of messages
- To capture messages in the communication channel: each process records sending or receipt of messages as part of their state

## A cut of the system's execution

- Subset of its global history that is a union of prefixes of process histories (see next slide)

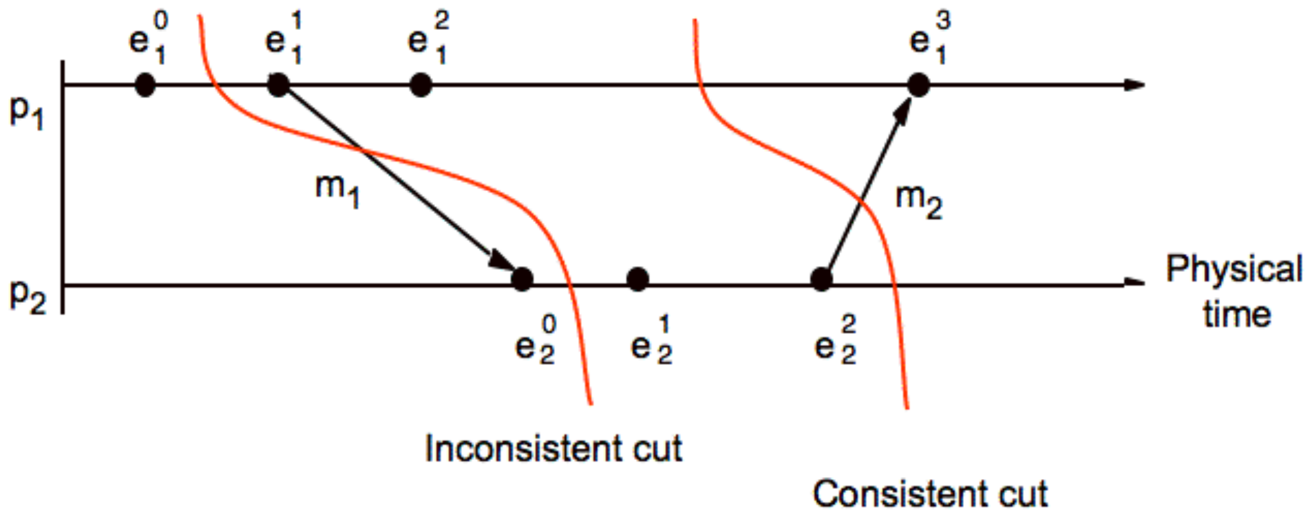
## Frontier of a cut

- The last state in each process.

## Consistent cut

- For all events e and e': $e \in C$ and $e' \rightarrow e \Rightarrow e' \in C$

# Cuts: An Example




**The frontier of a cut (red lines)**

**The left cut is inconsistent**

- We have the effect but not the cause of an event in the cut. The DS can never be in this state!

# Consistent Global States, Runs, Linearizations

## Consistent global state

- A global state that corresponds to a consistent cut
- A DS evolves through consistent global states

## Run

- Total ordering of all the events in a global history that is consistent with each process' local history ordering

## Linearization (consistent run)

- Ordering of events in a global history that is consistent with the happened-before relation

## Reachable state

- A state *S'* is reachable from another state S if there is a linearization through these states

# Snapshots: Motivation

**A snapshot of an execution of a distributed algorithm**

- Should return a configuration of an execution in the same computation.

**Use of Snapshots**

- Restarting after a failure
- Off-line determination of stable properties, which remain true as soon as they have become true such as deadlocks, garbage objects
- Debugging

**Challenge**

- Taking a snapshot without freezing the execution

# Snapshots

**Finding the global states of a DS**

- Algorithms for this are called snapshot algorithms.
- Note that this requires keeping track of the channel states too.

**Snapshots can be used to evaluate stable global predicates**

**Purpose**

- Storing information locally for taking a snaphot
- Collection of local snapshots is a different problem

**Basic and control messages**

- Distinguish basic messages of the underlying distributed algorithm and control messages of the snapshot algorithm

**A snapshot of a (basic) execution consists of:**

- A local snapshot of the (basic) state of each process, and
- The channel state of (basic) messages in transit for each channel

# A Snapshot Algorithm: Assumptions

**Neither channels nor processes fail**

- Communication is reliable

**Channels are unidirectional and messages arrive in order (FIFO)**

**There is a path between any two processes**

- Strongly connected network

**Any process can initiate the snapshot algorithm**

**The processes can continue to work while the snapshot takes place**

# The Main Idea

**Each process should record, for each channel,**

- Any messages that arrived after it recorded its state
- And before the sender recorded its own state

**The algorithm uses special marker messages to enforce this**

**Process initiation**

- Acting as if it received a marker (from a non-existing process)
- Following the marker receiving rule (next slide)

# Chandy and Lamport's Snapshot Algorithm

**Marker receiving rule for process $p_i$**

```
On pᵢ's receipt of a marker message over channel c:
  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 incoming channels;
  else
      pᵢ records the state of c as the set of
          messages it has received over c since it
          saved its state;
  end if
```

**Marker sending rule for process $p_i$**

```
After pᵢ recorded its state, for each outgoing channel c:
   pᵢ sends one marker message over c
    (before it sends any other message over c).
```

# Termination of the Snapshot Algorithm

## Assumptions

- A process that has received a marker message records its state within a finite time
- A process sends marker messages over each outgoing channel within a finite time (even if no application messages are sent over these channels)

## Termination

- If there is a path from a process $p_i$ to a process $p_j$ ($j \neq i$), then $p_j$ will record its state a finite time after $p_i$ recorded its state
- The graph is strongly connected, thus all processes will record their states and those of incoming channels after a finite time
- The algorithm terminates after each process has received a marker on all of its incoming channels

# Correctness and Complexity

## Complexity

- Let $e$ be the number of edges and $d$ the diameter of the network
- Recording a single instance of the algorithm requires O($e$) messages and O($d$) time

## Correctness

- FIFO: no message sent after the marker on that channel is recorded in the channel state
- When a process $p_j$ receives message $m_{ij}$ that precedes the marker on channel $C_{ij}$:
    - If process $p_j$ has not taken its snapshot, then it includes $m_{ij}$ in its recorded snapshot;
    - Otherwise, it records $m_{ij}$ in the state of the channel $C_{ij}$

# Consistency of the Snapshot Algorithm

**Theorem: Snapshots taken by the Chandy-Lamport Algorithm correspond to consistent global states**

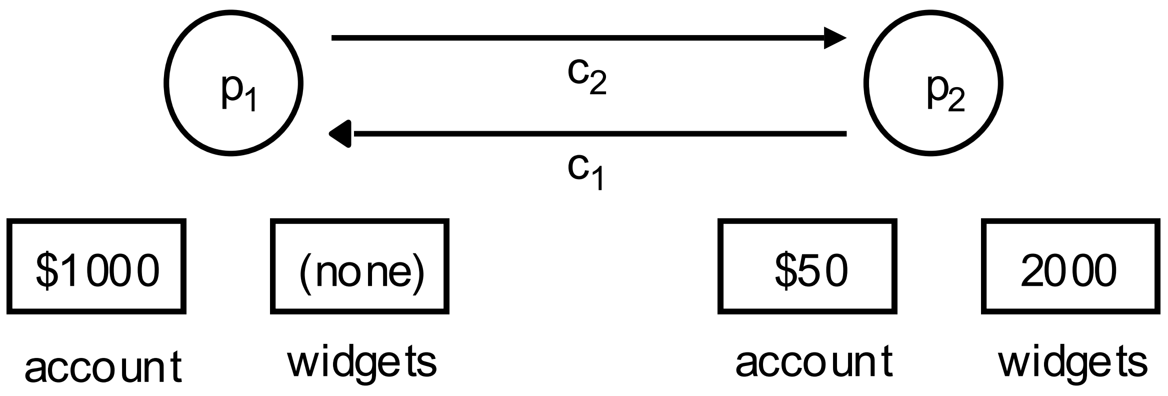
**Proof**

- Let $e_i$ and $e_k$ be events at $P_i$ and $P_k$, and let $e_i \rightarrow e_k$
- We need to show: if $e_k$ is in the cut, so is $e_i$.
- That means, if $e_k$ occurred before $P_k$ recorded its state, then $e_i$ must have occurred before $P_i$ recorded its state
- $P_i = P_k$: obvious
- $P_i \neq P_k$: assume $P_i$ recorded its state before $e_i$ occurred
    - There must be a finite sequence of messages $m_1$, ..., $m_n$ that induced $e_i \rightarrow e_k$
    - Before any of the $m_1$, ..., $m_n$ had arrived, a marker must have arrived at $P_k$, and $P_k$ must have recorded its state before $e_k$ occurred, i.e., $e_k$ is not in the cut (contradiction!)

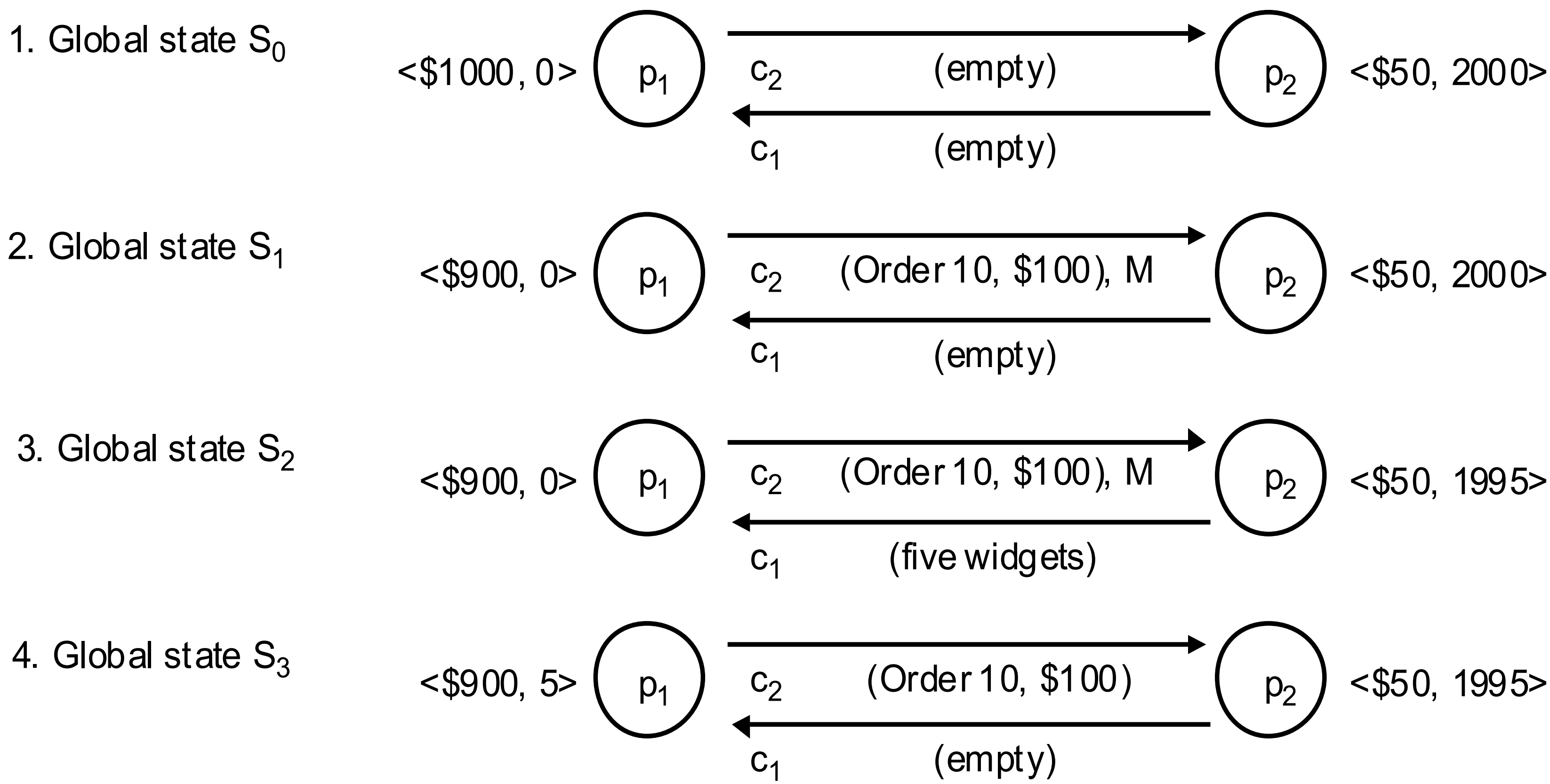
# Example: Two processes

**Two processes trade in "widgets"**

- Process $p_1$ sends orders for widgets over $c_2$ to $p_2$, $10 per widget
- Later, process $p_2$ sends widgets along channel $c_1$ to $p_1$
- The processes have the initial states shown in the figure
- Process $p_2$ has already received an order for five widgets



# The execution of the processes




(M = marker message)

Snapshot is: P1 <1000, 0>, P2 <50, 1995>, c1 <five widgets>, c2 < >

# Important Note

Note that the snapshot hear does not need to be an actual state but a consistent state recorded by the algorithm.

# Consistent Snapshots

## Presnapshot event

- Occurs at a process before the local snapshot at this process is taken

## Postsnapshot event

- Occurs at a process after the local snapshot at this process is taken

## Consistent snapshot

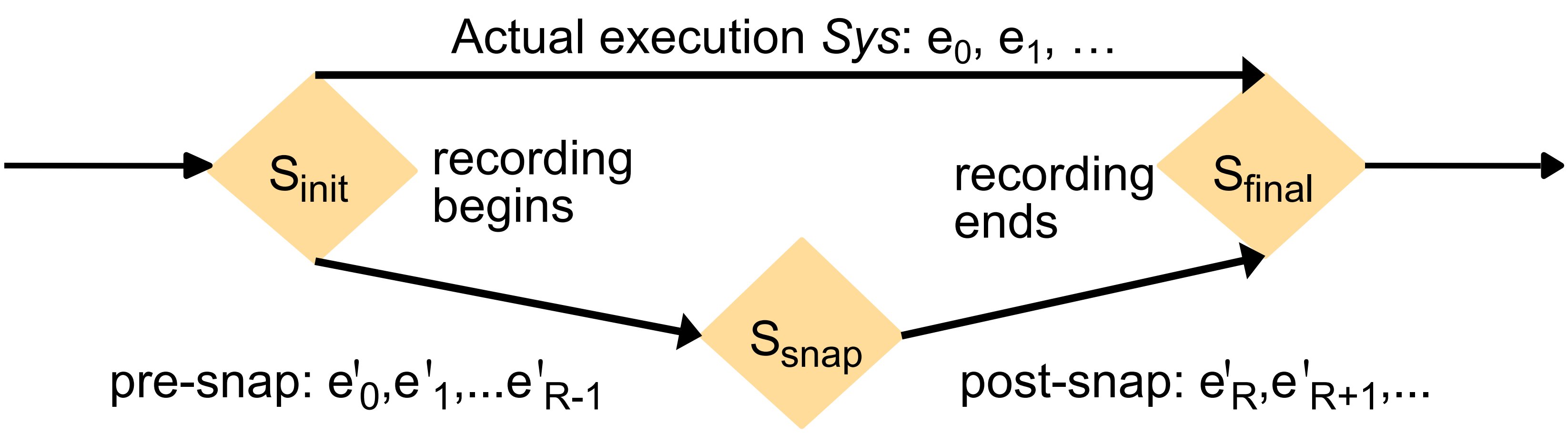
- For each presnapshot event *e*, all events that are causally before *e* are also presnapshot
- A basic message is included in a channel state if and only if the corresponding send event is presnapshot while the corresponding receive event is postsnapshot

# Reachability in the Snapshot Algorithm

- $S_{init}$: the global state immediately before the first process recorded its state
- $S_{final}$: the global state when the snapshot algorithm terminates, (immediately after the last state-recording action)
- $S_{snap}$: the recorded global state

## Reachability Theorem

- Let Sys = $e_0$, $e_1$, … be the linearization of a system execution. Then there is a permutation Sys'= $e'_0$, $e'_1$, ... of Sys such that
- $S_{init}$, $S_{snap}$ and $S_{final}$ occur in Sys'
- $S_{snap}$ is reachable from $S_{init}$ in Sys', and $S_{final}$ is reachable from $S_{snap}$ in Sys'



# Proof

**Split events in Sys in**

- Pre-snap events: occurred before the respective process in which this event occurred recorded its state
- Post-snap events: all other events

**Order events to obtain Sys'**

- Assumption $e_j$ is post-snap event at one process, and $e_{j+1}$ pre-snap in a different process
- $e_j \rightarrow e_{j+1}$ is not possible (otherwise a marker message would have preceded the message, making the reception of the message a post-snap event, but we assumed that $e_{j+1}$ is a pre-snap event)
- Thus, $e_j$ and $e_{j+1}$ may be swapped in Sys'
- Swap adjacent events, if necessary and possible, until in Sys' all pre-snap events precede all post-snap events
- Since we have disturbed neither $S_{\text{init}}$ nor $S_{final}$ we have established the reachability relationship amongst these states

# Important Note

**Reachability property of the snapshot algorithm**

- Useful for detecting stable predicates
- If a stable predicate is TRUE in the state $S_{snap}$ then the predicate is TRUE in the state $S_{final}$
- Reason: if a stable predicate is TRUE for a state then it will remain TRUE for any state that is reachable from it
- Similarly if a stable predicate is FALSE for our snapshot we can say that it was FALSE from the beginning

# Distributed Debugging

- A distributed system records its states
- Sent the states to an external server later
- The external server then assembles global consistent states

## Aim

- Determine where a given predicate was definitely true at some point in the execution and cases where it was possibly true

## Examples

- The difference between two variables *x* and *y* is always non-zero
- The valves $v_1$ and $v_2$ may never be open at the same time

## Chandy-Lamport snapshot algorithm

- Best case: prove violation of these properties

# Possibly and Definitely

**Possibly $\phi$**

- There is a consistent global state through which a linearization passes such that this predicate is true

**Definitely $\phi$**

- For all linearizations $L$, there is a consistent global state through which $L$ passes such that this predicate is true

**Chandy-Lamport**

- $\varphi(S_{snap}) \Rightarrow$ pos $\varphi$

**Inference**

- $\neg$pos $\varphi \Rightarrow$ def $\neg\varphi$
- The converse is not true ($\neg\varphi$ holds at some state on every linearization: $\varphi$ may hold for other states

**Hence, we need to**

- Collect process states and then evaluate possibly and definitely

# Monitoring Algorithm (Marzullo-Neiger)

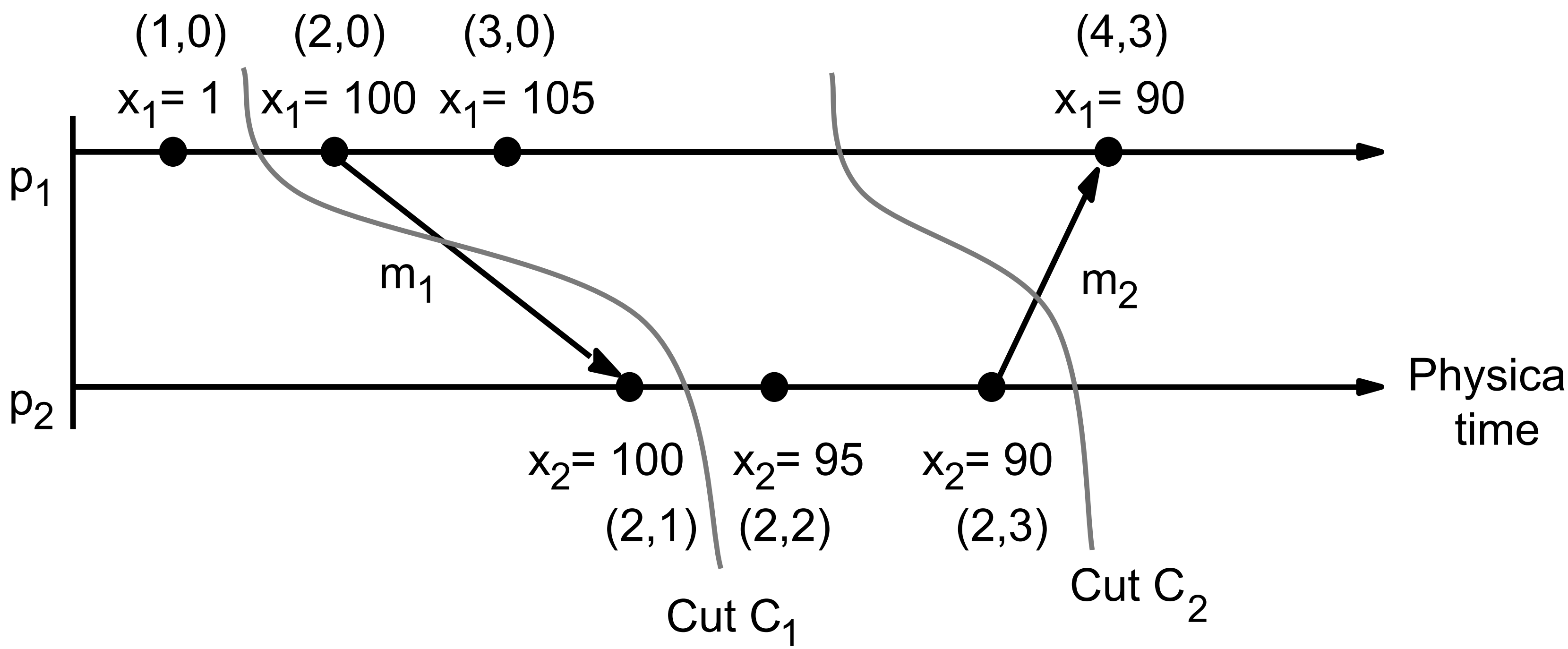
## Centralized algorithm

- One external observer connected to all processes by (perfect) message passing channels, receives from the processes in the system periodic messages containing their local state
- Monitor does not interfere with the system's computation
- Monitor assembles consistent global states from the messages it receives
- State collection
  - processes $p_i$ send initial state to monitor $M$ which records state messages in separate FIFO queue $Q_i$ for each $i$
  - $p_i$ send their local state when necessary, namely
    - When the local state changes a portion of the global state that affects the evaluation of φ
    - When the local state change causes φ to change its value

# Monitoring Algorithm (Marzullo-Neiger)

- In order for the monitor to infer consistency of the constructed state information the processes maintain vector clocks
- Processes piggyback their vector clock value with every message to *M*
- Let *S* a global state that M has constructed from the state messages received, and $V(s_i)$ the vector time stamp received from process *i*. S is consistent iff
  - $V(s_i)[i] \geq V(s_k)[i]$ $\forall i,k$ (condition CGS)
  - The number of *i*'s events known at *k* when it sent $s_k$ is no more than the number of events at *i* when it sent $s_i$

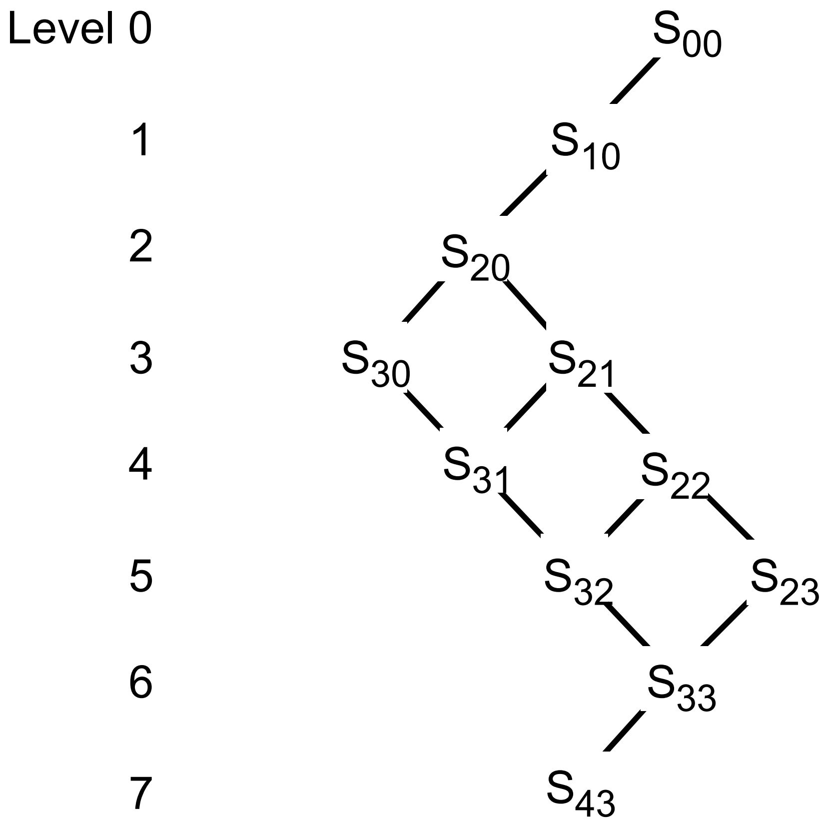
# Vector Timestamps & Variable Values During Execution




**Example**

- The consistency condition is clearly violated for $V(s_i) = (1,0)$ and $V(s_k) = (2,1)$. Hence C1 is inconsistent and does not constitute a violation of φ.

# The Lattice of Global States for the Execution




*Sij* = global state after *i* events at process 1 and *j* events at process 2

# Algorithms to Evaluate *Possibly* $\phi$ and *Definitely* $\phi$

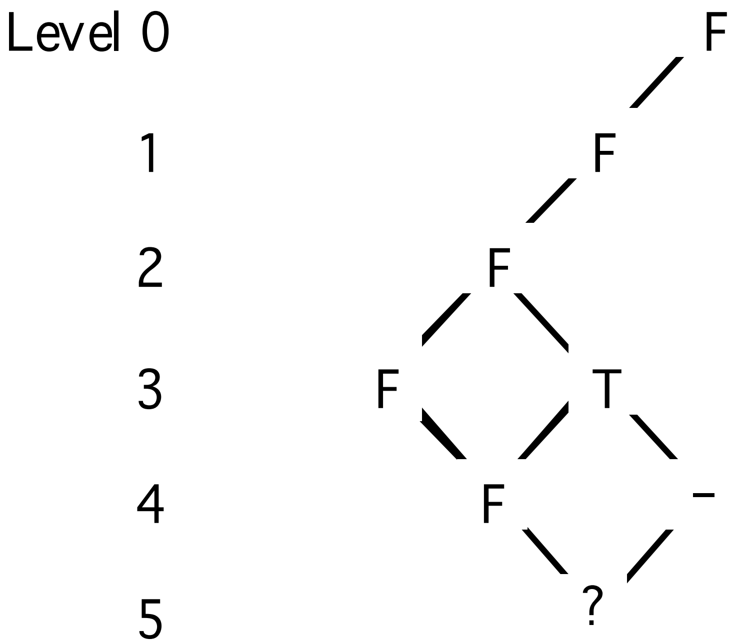
1. *Evaluating possibly $\phi$ for global history H of N processes*

$L := 0$;
*States* $:= \{ (s_1^0, s_2^0, \ldots, s_N^0) \}$;
*while* ($\phi(S) = False$ for all $S \in$ States)
  $L := L + 1$;
  *Reachable* $:= \{ S' : S'$ reachable in $H$ from some $S \in$ States $\wedge$ level$(S') = L \}$;
  *States* := *Reachable*
*end while*
output "*possibly* $\phi$";

2. *Evaluating definitely $\phi$ for global history H of N processes*

$L := 0$;
*if* $(\phi(s_1^0, s_2^0, \ldots, s_N^0))$ *then States* $:= \{\}$ *else States* $:= \{ (s_1^0, s_2^0, \ldots, s_N^0) \}$;
*while* (*States* $\neq \{\}$)
  $L := L + 1$;
  *Reachable* $:= \{ S' : S'$ reachable in $H$ from some $S \in$ States $\wedge$ level$(S') = L \}$;
  *States* $:= \{ S \in Reachable : \phi(S) = False \}$
*end while*
output "*definitely* $\phi$";

# Evaluating *definitely* ϕ



# Costs

**Time complexity**

- $N$ processes
- $k$ is the maximum number of messages per process
- Monitor M compares states of each of the N processes with each other: $O(k^N)$ (exponential in the number of processes)

Space complexity

- *O(kN)* space requirement
- We can do a bit better: state information can be deleted from $Q_i$ if that state message from $i$ can under no circumstances become part of a consistent global state

# Monitoring in Synchronous Systems

## Monitoring in asynchronous networks

- Observation of global states that the system may not have traversed
- Any two process states in a global state may have occurred an arbitrary period of time apart from each other

## Idea for synchronous systems

- Use physical clocks in synchronous networks in addition to logical network clocks in order to limit the number of states to be considered
- Monitor only considers those local state sets that could possibly have occurred simultaneously, given the known bounds on the clock synchonization