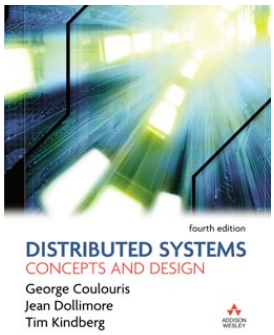
Slides for Chapter 14: Time and Global State

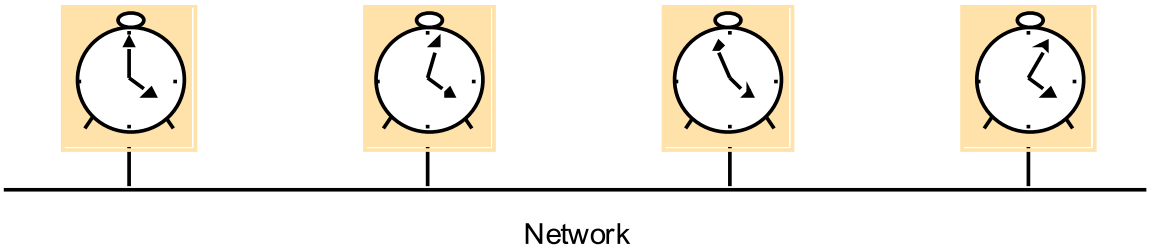


From **Coulouris, Dollimore and Kindberg**
Distributed Systems:
Concepts and Design

Time and global states

- Time is important also in distributed systems
  - Measuring delays, executions
  - in transactions, we need authoritative external time
  - timestamps in authentication protocols
- Algorithms for clock synchronization
- Measuring global time is difficult: multiple frames of reference (like in Special Theory of Relativity)
  - event order may be different for different observers
  - there is no universal physical clock
- There is no absolute, global time in a distributed system
- But often it is not really needed

Figure 11.1
Skew between computer clocks in a distributed system



- In distributed systems, there is no global clock or common memory. Each processor has its own internal clock and its own notion of time.
- These clocks can easily drift seconds per day, accumulating significant errors over time.
- Also, because different clocks tick at different rates, they may not remain always synchronized although they might be synchronized when they start.
- This clearly poses serious problems to applications that depend on a synchronized notion of time.

Physical Clock Synchronization

- For most applications and algorithms that run in a distributed system, we need to know time in one or more of the following contexts:
  - The time of the day at which an event happened on a specific machine in the network.
  - The time interval between two events that happened on different machines in the network.
  - The relative ordering of events that happened on different machines in the network.
- Unless the clocks in each machine have a common notion of time, time-based queries cannot be answered.
- Clock synchronization has a significant effect on many problems like secure systems, fault diagnosis and recovery, scheduled operations, database systems, and real-world clock values.

Physical Clock Synchronization

- **Clock synchronization** = the process of ensuring that physically distributed processors have a common notion of time.
- Due to different clocks rates, the clocks at various sites may diverge with time => periodically a clock synchronization must be performed to correct this clock skew in distributed systems.
- Clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time).
- Clocks that must not only be synchronized with each other but also have to adhere to physical time are termed *physical clocks*.

Physical Clock Synchronization: definitions and terminology

Let C_a and C_b be any two clocks.

- **Time:** The time of a clock in a machine p is given by the function $C_p(t)$, where $C_p(t) = t$ for a perfect clock.
- **Frequency:** Frequency is the rate at which a clock progresses. The frequency at time t of clock C_a is $C_a^{'}(t)$.
- **Offset:** Clock offset is the difference between the time reported by a clock and the *real time*. The offset of the clock C_a is given by $C_a(t) - t$. The offset of clock C_a relative to C_b at time $t \geq 0$ is given by $C_a(t) - C_b(t)$.
- **Skew:** The skew of a clock is the difference in the frequencies of the clock and the perfect clock. The skew of a clock C_a relative to clock C_b at time t is $(C_a'(t) - C_b'(t))$. If the skew is bounded by ρ, then as per Equation (1), clock values are allowed to diverge at a rate in the range of $1 - \rho$ to $1 + \rho$.
- **Drift (rate):** The drift of clock C_a is the second derivative of the clock value with respect to time, namely, $C_a''(t)$. The drift of clock C_a relative to clock C_b at time t is $C_a''(t) - C_b''(t)$.

Physical Clock Synchronization: clock inaccuracies

- Physical clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time).
- However, due to the clock inaccuracy discussed above, a timer (clock) is said to be working within its specification if (where constant ρ is the maximum skew rate specified by the manufacturer.)

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho \qquad (1)$$

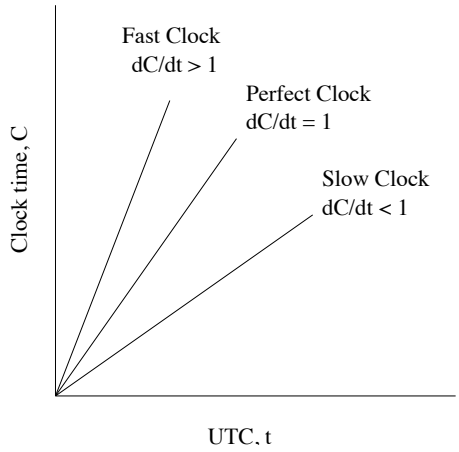
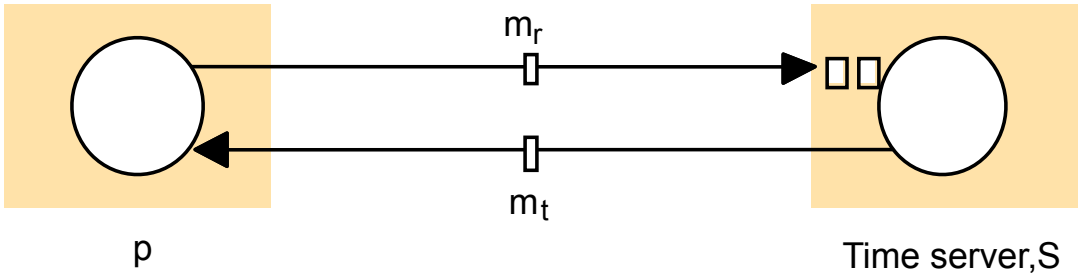


Figure 14.2 - Cristian Algorithm
Clock synchronization using a time server

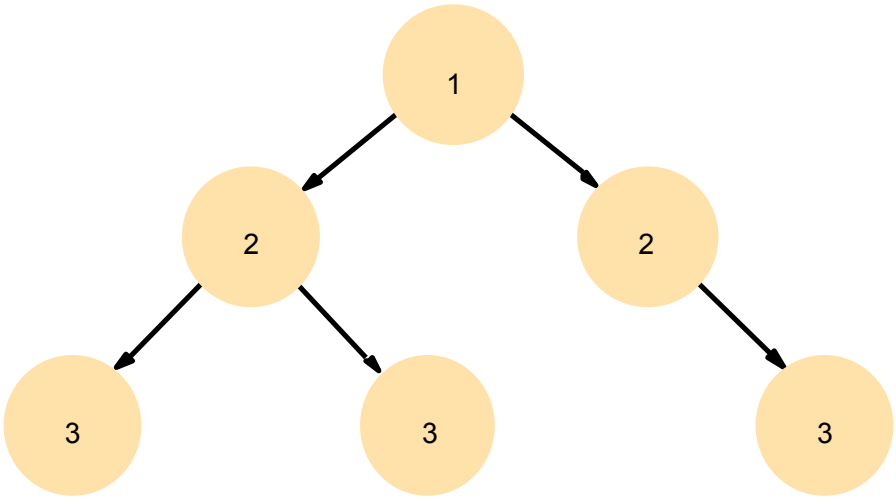


- Simplest form:
  - Client requests time to a time server, by sending message m_r
  - Time server answers with m_t, containing its time
- This protocol does not take into account network delay/jitter, so it can be used only on networks with low and stable latency (e.g. LANs)

Physical Clock Synchronization: NTP

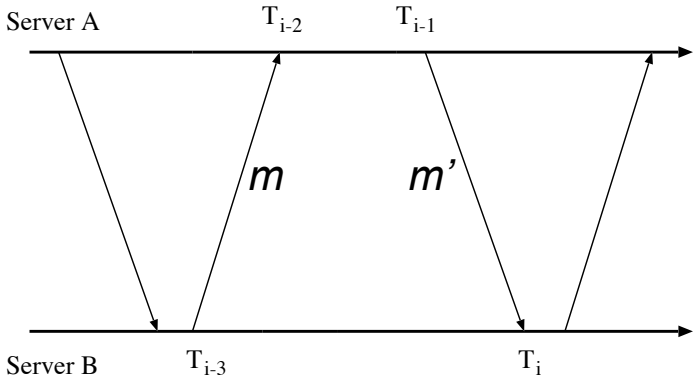
- The Network Time Protocol (NTP) uses the Offset Delay Estimation method
- Widely used for clock synchronization on the Internet - implemented by the `ntpd` daemon
- Precision: tens of ms on Internet, 1 ms in LAN
- The design of NTP involves a hierarchical tree of time servers.
  - The primary server at the root synchronizes with the UTC (stratum 0).
  - The next level (stratum 1) contains secondary servers, acting as backups to the primary server.
  - At the lowest level is the synchronization subnet which has the clients.

Figure 11.3
An example synchronization subnet in an NTP implementation



Note: Arrows denote synchronization control, numbers denote strata.

Messages exchanged between a pair of NTP peers



- A pair of servers in symmetric mode exchange pairs of timing messages.
- A store of data is then built up about the relationship between the two servers (pairs of offset and delay).
  Specifically, assume that each peer maintains pairs (O_i, D_i), where
  O_i - measure of offset (θ)
  D_i - transmission delay of two messages (δ).
- The offset corresponding to the minimum delay is chosen.
  Specifically, the delay and offset are calculated as follows. Assume that message m takes time t to transfer and m' takes t' to transfer.

- The offset between A's clock and B's clock is O. If A's local clock time is $A(t)$ and B's local clock time is $B(t)$, we have

$$A(t) = B(t) + O \tag{3}$$

  Then,

$$T_{i-2} = T_{i-3} + t + O \tag{4}$$

$$T_i = T_{i-1} - O + t' \tag{5}$$

  Assuming $t = t'$, the offset O_i can be estimated as:

$$O_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2 \tag{6}$$

  The round-trip delay is estimated as:

$$D_i = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2}) \tag{7}$$

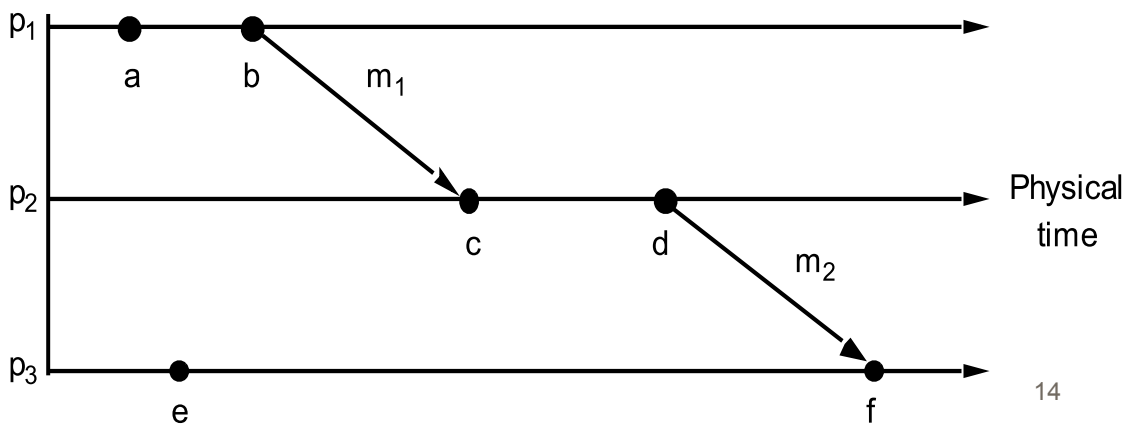
- The eight most recent pairs of (O_i, D_i) are retained.
- The value of O_i that corresponds to minimum D_i is chosen to estimate O.

Logical time and logical clocks

- can't synchronize physical clocks perfectly
- but often absolute time might not be necessary, just need ordering of events
  - If e, e' occur in process i, then they occur in the order in which p_i observes them
  - if e=send(m) and e'=receive(m), then e occurs before e'
- *happened-before* relationship among events (called also *potential causal ordering*) $\rightarrow$
  - HB1: if e $\rightarrow$ e' in process i, then e $\rightarrow$ e'
  - HB2: for any message m, send(m) $\rightarrow$ receive(m)
  - HB3: if e $\rightarrow$ e' and e' $\rightarrow$ e'', then e $\rightarrow$ e''

Causal ordering

- If e, e' are events such that e $\rightarrow$ e', then there is a (non necessarily unique) series $e_1,\ldots,e_n$ s.t.
  - e_1=e, e_n=e'
  - $e_i \rightarrow e_{i+1}$ for i=1,…,n-1, which means either e_i, e_{i+1} are events inside the same process, or e_i is a send and e_{i+1} a receive



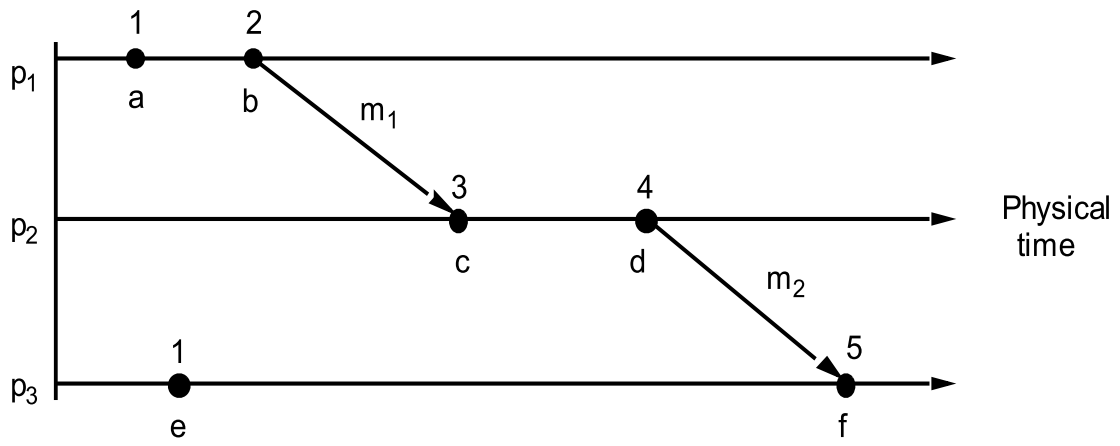
Causal ordering

- (E,→) is a partial order
- there may be events which are not related
  - e.g. a and e, c and e, ecc.
- We say that these events are *concurrent* (denoted as “a || e”)
- Causal order may not capture all events which caused an event (e.g. those “external”)
- Causality may be not effective (e.g., e happened before e’, but these two events are unrelated).
- So a proper name is *potential causal ordering*
- (Adding a relation of conflict between events we get the notion of *event structure*)

Logical clocks [Lamport 1978]

- each process p_i keeps a local *clock* L_i (an integer)
- no connection with any physical time/clock
- event e at p_i is timestamped with the value of L_i at that moment; denoted as $L_i(e)$ or L(e)
- updating logical clocks:
  - LC1: $L_i := L_i + 1$ for each event in process i
  - LC2: When a process i sends a message m:
    - it piggybacks on m the value of $t=L_i$
    - Upon receiving (m,t), process j computes $L_j := \max(L_j, t)$ and then applies LC1 before timestamping the event receive(m)
- **Prop**: if e → e', then L(e) < L(e')
  but L(e) < L(e') doesn't imply e → e' (Why?)

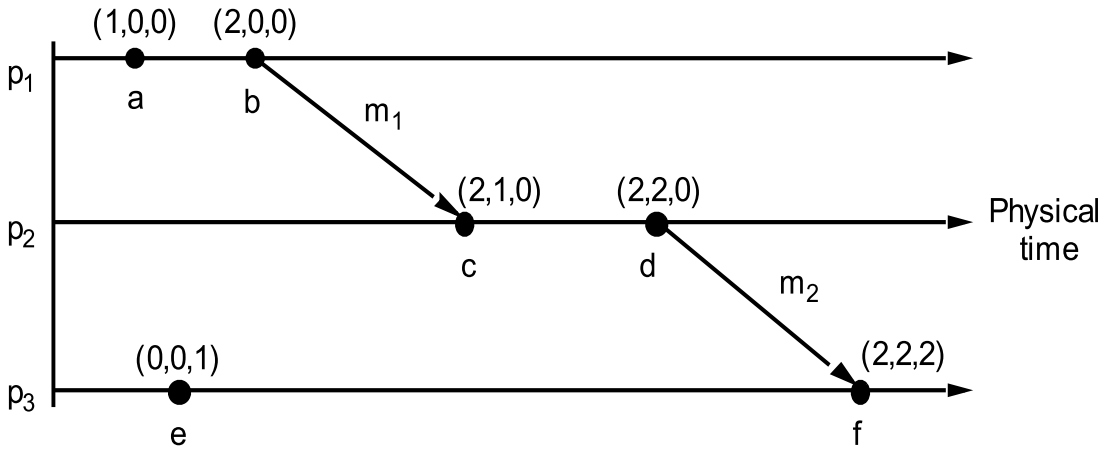
Figure 14.6
Lamport timestamps for the events shown in Figure 14.5



Vector clocks (Mattern 1989, Fidge 1991)

- Lamport clocks: L(e) < L(e') doesn't imply e → e'
- each process keeps its own vector clock V_i
- $V_i[j]$ = "the time of p_j according to p_i"
- piggyback timestamps (vectors) on messages
- updating vector clocks:
  - VC1: Initially, $V_i[j] := 0$ for p_i, j=1.. N (N processes)
  - VC2: before p_i timestamps an event, $V_i[i] := Vi[i]+1$
  - VC3: p_i piggybacks $t = V_i$ on every message it sends
  - VC4: when p_j receives a timestamp t, it sets
    $Vj[k] := \max(Vj[k] , t[k])$ for k=1..N
    (merge operation) *before* timestamping the receive event

Figure 14.7
Vector timestamps for the events shown in Figure 14.5



Vector clocks

- At p_i:
  - $V_i[i]$ is the number of events p_i timestamped locally
  - $V_i[j]$ is the number of events that have occurred at p_j (that has potentially affected pi)
  - Could more events than $V_i[j]$ have occurred at p_j?
- Comparing vector timestamps
  - V = V' iff V[j] = V'[j], j=1..N
  - V ≤ V' iff V[j] ≤ V'[j], j=1..N
  - V < V' iff V ≤ V' and V != V'
    - Which is different from requiring < in all elements

Vector clocks capture precisely causal order

- **Prop**: if e → e', then V(e) < V(e')
  - Proof: by induction on the length of the causal sequence $e=e_1,...,e_n=e'$
- **Prop**: if e || e' then neither V(e) ≤ V(e') nor V(e') ≤ V(e)
  - Proof: there is no causal sequence from e to e', hence vectors cannot have been merged
- **Prop**: if V(e) < V(e'), then e → e'
  - Proof: if V(e) < V(e') then V(e) ≤ V(e') (and not V(e') ≤ V(e)), hence by previous proposition it is not e || e', which means that there is a causal sequence from e to e'. (It cannot be vice versa otherwise it would be V(e') ≤ V(e), which is absurd). Hence e → e'

Vector clocks vs Lamport logical clocks

- Compared to Lamport logical clocks, vectors take N cells for a network with N nodes
- [Charron-Bost 1991] proved that if we want to be able to tell whether two events are concurrent in a network of N nodes, the minimal size of timestamps is N
  - Some optimizations are possible, though
- Variant: *matrix clocks* [Raynal-Singhal 1996]

Example application: causally ordered chat

- Let P_1, …, P_N be the participants of a group chat (e.g. Whatsapp)
- When a user sends a message, this is received by all clients (via the server) but not necessarily in the same order
- Problem is how to show messages to the users in a *causally correct* order: if user P_i has sent message *m* after having read message *m'*, then *m* must be shown to all the other users only after *m'*
- Two user can send messages "simultaneously", i.e., without reading each other's - in this case, their messages can be shown in any order

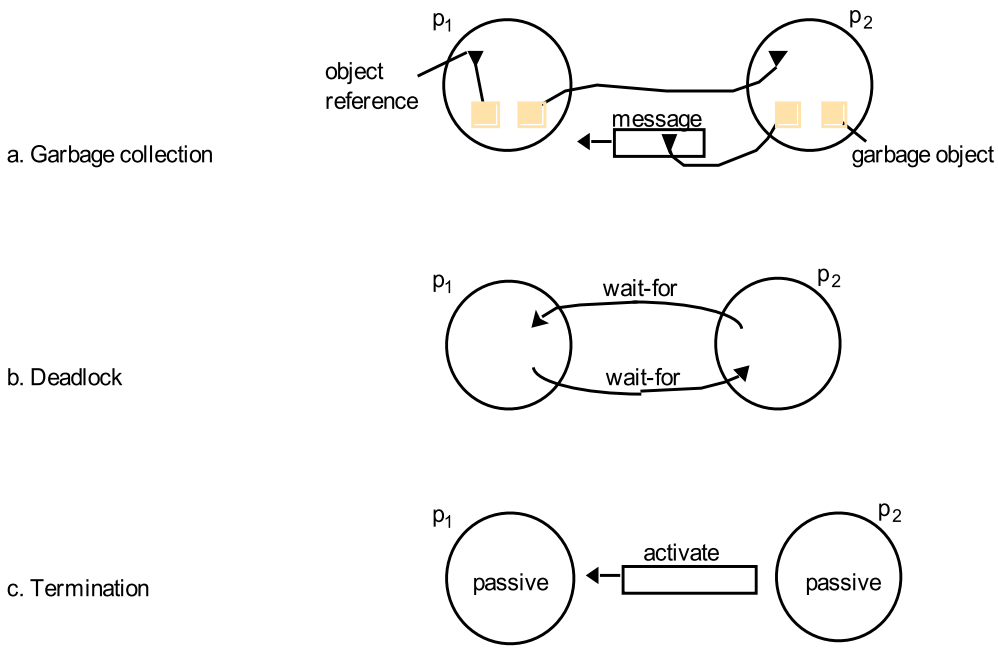
Example application: causally ordered chat

- Solution: each client keeps a vector clock with N components; we count only "send" events
- Each client does as follows:
  - When the user at the client P_i sends message m, the client P_i increments the i-th component of its vector V_i, and timestamps the message with its vector
  - When P_j (which has the vector clock V_j) receives the message (i,m,V):
    - if $V[i] = V_j[i]+1$ and for all $k \neq i$: $V[k] \leq V_j[k]$
      then show m to the user and increment $V_j[i]$
      else put (i,m,V) in a pool of "waiting messages"
    - When the clock is incremented, check if in the pool there are messages whose timestamp satisfies the condition with the actual clock, and deliver them (in any order)

Global state

- Sometimes we need to decide whether a particular property holds in a given state of the distributed system. Some examples:
  - **Distributed garbage collection**: understanding if an object is not referenced, not even remotely
  - **Distributed deadlock detection**: a cycle of wait-for-message relations among distributed processes, so that the system never make progress
  - **Distributed termination detection**: when a distributed algorithm has terminated? Testing whether all processes have halted is not enough (messages may be delayed...)
  - **Distributed debugging**: invariant properties may involve variables and states of distributed processes

Figure 14.8
Detecting global properties



Global States and consistent cuts

- It is possible to observe the succession of states of an individual process, but the question of how to ascertain a global state of the system – the state of the collection of processes is much harder.
- The essential problem is the absence of global time. If we had perfectly synchronized clocks at which processes would record its state, we can assemble the global state of the system from local states of all processes at the same time.
- The question is: can we assemble the global state of the system from local states recorded at different real times?
- The answer is “YES”.

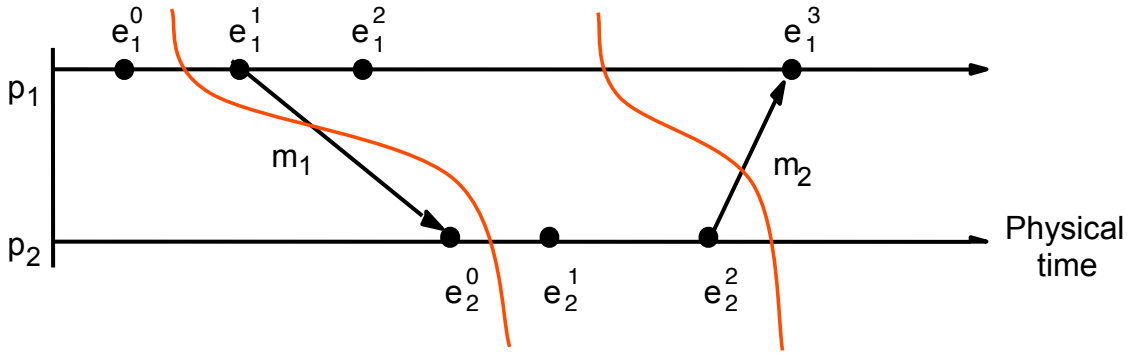
Some definitions

$$history(p_i) = h_i =< e_i^0, e_i^1, e_i^2, ... >$$

$$finite\ prefix\ of\ history: h_i^k =< e_i^0, e_i^1, e_i^2, ... e_i^k >$$

- A series of events occurs at each process. Each event is either an internal action of the process (e.g. variables updates) or it is the sending or receipt of a message over the channel.
- S_i^k is the state of process Pi before k-th event occurs, so S_i^0 is the initial state of Pi.
- Thus the global state S corresponds to initial prefixes of the individual process histories.

Cuts

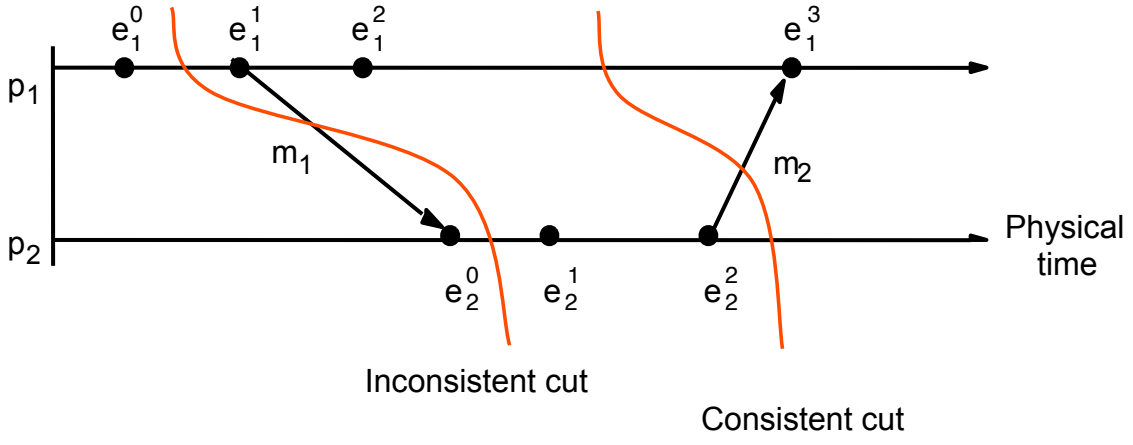


- A **cut** of the system's execution is a subset of its global history that is a union of prefixes of process histories.

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_N^{c_N}$$

- The state of each process is in the state after the last event occurs in its own cut. The set of last events from all processes are called **frontier** of the cut.

Consistent and inconsistent cuts



- **Inconsistent cut**: since P_2 contains receiving of m_1, but at P_1 it does not include sending of that message. This cut shows an effect without a cause. Actual execution cannot reach a global state that corresponds to process states at the frontier under this cut.
- **Consistent cut**: it includes both the sending and receipt of m1, or it includes the sending but not the receipt of m2. It is still consistent with actual execution.

Consistent cuts

- A cut C is consistent if, for each event it contains, it also contains all the events that happened-before that event.

$$for\ all\ events\ e \in C,\ f \rightarrow e \Rightarrow f \in C$$

- A *consistent global state* is one that corresponds to a consistent cut.
- A *run* is a total ordering of all the events in a global history that is consistent with each local history's ordering.
- A *linearization* or *consistent run* is an ordering of the events in a global history that is consistent with the happened-before relation.
- A state S' is *reachable* from S if there is a linearization that passes through S and then S'

Global state predicates

- A *global state predicate* is a function that maps from the set of global states of processes to True or False.
- A predicate P is *stable* if, once the system enters a state in which the predicate is True, it remains True in all future states reachable from that state
  - e.g. object being garbage, system deadlocked, algorithm terminated...
- *Safety* with respect to a predicate P means that P evaluates to False for all states reachable from S0
- *Liveness* with respect to a predicate P means that for all linearization starting from S0, P evaluates to True for some state reachable from S0

Chandy-Lamport's "snapshot" algorithm (1985)

- An algorithm for determining global states of distributed system.
- Aim is to record a set of process and channel states for a set of processes Pi (a "snapshot") such that, even though this combination of recorded states may never have occurred at the same time, the recorded global state is consistent.
- The algorithm records states locally at processes; it does not give a method for gathering the global state at one site.
  - (All processes could send the state they have recorded to some collector process, but this can be done later)

Assumptions of the snapshot algorithm

1. Neither channels nor processes fail; communication is reliable so that every message sent is eventually received intact, exactly once;
2. Channel are unidirectional: either incoming or outgoing and provide FIFO order message delivery;
3. The graph of processes and channels is strongly connected (there is a path between any two processes).
4. Any process may initiate a global snapshot at any time.
5. The processes may continue their normal execution and send and receive normal massages while the snapshot takes place.

The snapshot algorithm idea

- Each process records its own state and also for each incoming channel a set of messages sent to it.
- This allows to record process states at different times but to account for the differential between process states in terms of message transmitted but not yet received.
- If process p_i has sent a message m to process p_j, but p_j has not received it, then p_j accounts for m as belong to the state of the incoming channel.
- Use of special **marker** message, used as a prompt for the receiver to save its own state if it has not done so; and as a means of determining which messages to include in the channel state.

Figure 14.10
Chandy and Lamport's 'snapshot' algorithm

Marker receiving rule for process p_i

On p_i's receipt of a *marker* message over channel c:

if (p_i 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

p_i records the state of channel c as the set of messages it has received over c since it saved its state.

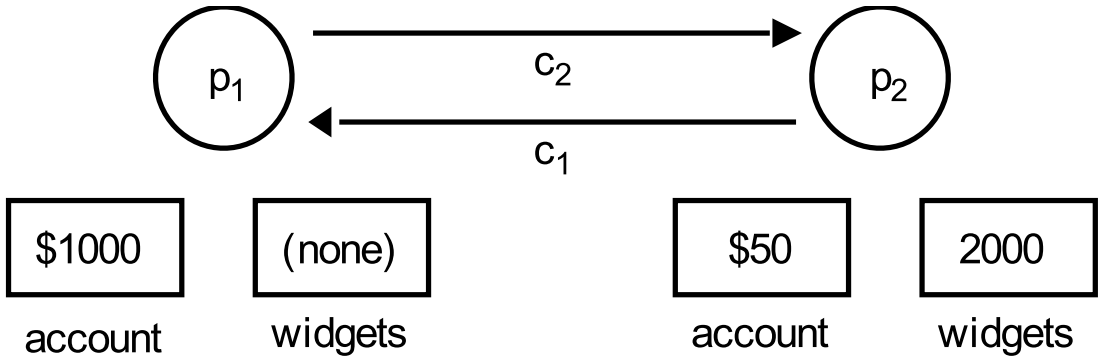
end if

Marker sending rule for process p_i

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

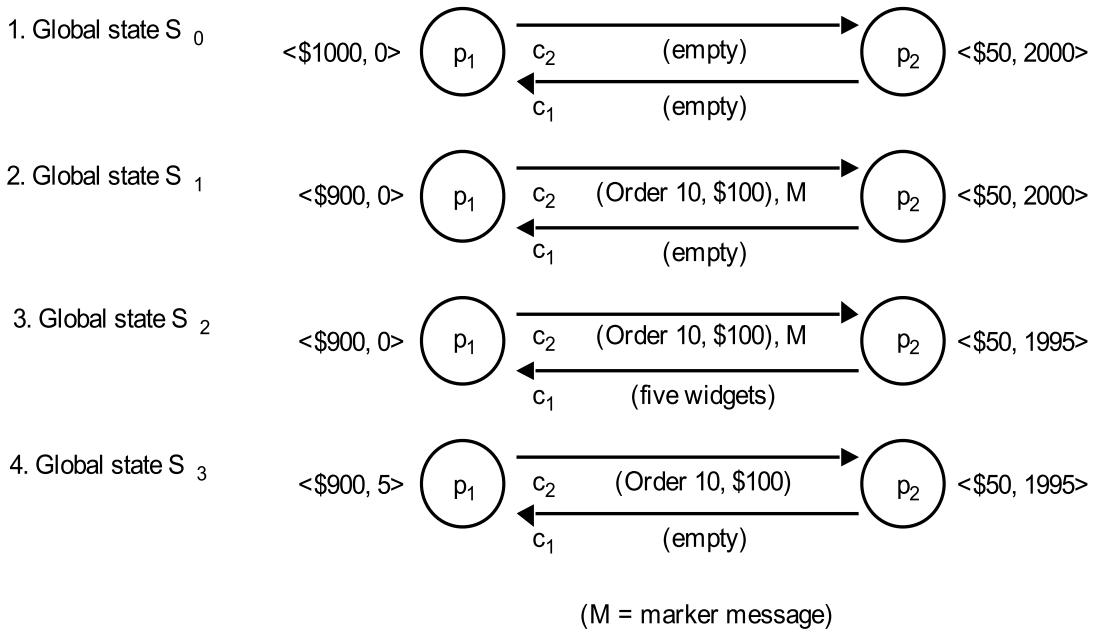
p_i sends one marker message over c
(before it sends any other message over c).

Figure 14.11
Two processes and their initial states



- Two processes connected by two unidirectional channels, c1 and c2. Processes trade in 'widgets'.
  - Process p1 sends orders for widgets over c2 to p2, enclosing payment at the rate of $10 per widget.
  - Some time later, process p2 sends widgets along channel c1 to p1.
- Process p2 already received an order for five widgets, which it will shortly dispatch to p1.

Figure 14.12
The execution of the processes in Figure 14.11



1. P1 records its state in S0. Following the marker sending rule, it will send a marker over c2 to p2 before it sends the next order (Order 10, $100).
2. Before p2 receives the marker, it sends five widgets to p1 over c1.
3. Now P1 receives five widgets and P2 receives marker. P2 will record it state S2 and record c2 as empty. Following the sending rule, p2 sends a marker to p1.
4. P1 receives the marker, P1 records the state of c1 as five widget that it received after it first recorded its state.
5. Final recorded state is:
   - P1 = <$1000,0>, P2 = <$50,1995>
   - C1 = <five widgets>, C2 = <>

Termination of the Chandy-Lamport Algorithm

Theorem: The Chandy-Lamport Algorithm terminates.

Proof: Let us assume that a process receiving a marker message will record its state and send marker messages via each outgoing channel in finite time.

If there is a communication path from P_i to P_k, then P_k will record its state a finite period of time after P_i.

Since the communication graph is strongly connected, all process in the graph will have terminated recording their state and the state of incoming channels a finite time after some process initiated snapshot taking.

Characterizing the observed state

The snapshot algorithm selects a cut from the history of the system, corresponding to a state of the system.

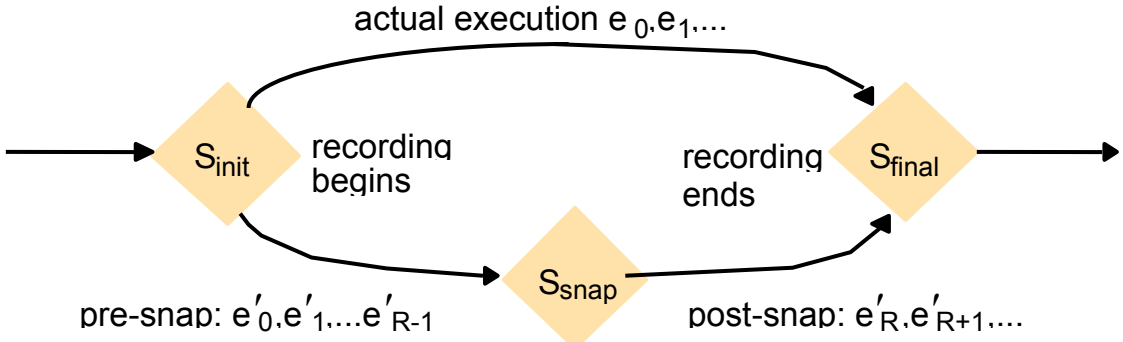
Theorem: The cut (and hence the state) is consistent.

Proof: Let e_i and e_j be events at P_i and P_j, and let $e_i \rightarrow e_j$. Claim: if e_j is in the cut, so is e_i. That means, if e_j occurred before P_j recorded its state, then e_i must have occurred before P_i recorded its state

- j=i: obvious.
- j≠i: assume P_i recorded its state before e_i occurred. As j≠i there must be a finite sequence of messages $m_1,\ldots, m_n$ that induced $e_i \rightarrow e_j$. Then, before any of the $m_1,\ldots, m_n$ had arrived, a marker must have arrived at P_j, and P_j must have recorded its state before e_j occurred. But then e_j would be not in the cut – contradiction.

Figure 14.13
Reachability between states in the snapshot algorithm

- We can establish a reachability relation between the observed state, and the initial and final states



- The observed state is obtained by a permutation of the actual execution into pre-snap and post-snap events
  - if e_j is a post-snap event at p_i, and e_{j+1} is a pre-snap event at p_k, then it cannot be $e_j \rightarrow e_{j+1}$ because these events would be send/receive of a message m, and a marker would have preceded the message m, thus e_{j+1} would be post-snap
  - then we can swap e_j and e_{j+1}

Applications

- The state S_{snap} can be used for deciding some properties
  - **Safety**: if P holds in S_{snap}, then the safety is violated (there exists at least one executions in which P holds)
  - **Stable** predicates:
    - if P holds in S_{snap}, then it holds in S_{final}.
    - If P does not hold in S_{snap}, then it does not hold in $S_{initial}$.
- Other algorithms have been proposed

Distributed debugging

- Typically interested in invariant safety properties
  - the system does not reach a deadlock state
  - the difference between variables x and y is always non-zero
  - the valves v1 and v2 may never be open at the same time
- Chandy-Lamport snapshot algorithm can at best prove violation of these properties
- Interested in a monitoring algorithm that records system traces in order to decide whether safety properties were, or may have been, violated in a given system run

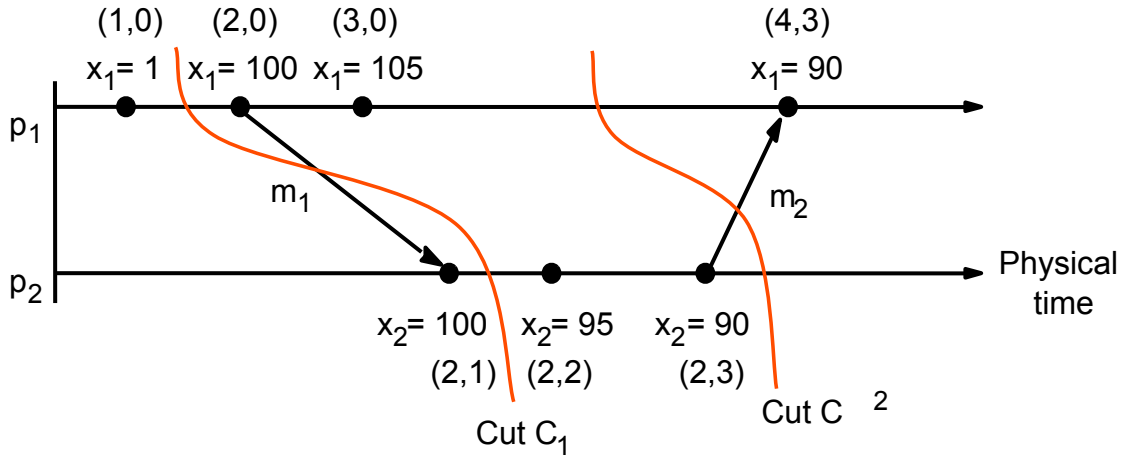
Distributed debugging

- Let H the execution history of a system and φ a state predicate
  - **pos** φ: there is a consistent global state S through which a linearization of H passes such that φ(S) = true (in CTL: EFφ)
  - **def** φ: for all linearizations L of H there is a consistent global state S through which L passes such that φ(S) = true. (in CTL: AFφ)
  - For Chandy-Lamport: $\varphi(S_{snap}) \Rightarrow$ pos φ
- Inference:
  - ¬pos φ ⇒ def¬φ
  - the converse is not true!
- Monitoring algorithms have been proposed

Centralized monitoring algorithm [Marzullo-Neiger 1991]

- A *monitor process* receives states from a set of *observed processes* p_i (i=1,…N)
- Each observed process sends to the monitor its initial state and from time to time its current state
  - Only the relevant part of the state needs to be sent, and only when these parts actually change
- The monitor assembles a consistent global state
  $S = (s_1,\ldots,s_N)$
  against which φ is evaluated
- However, the state may be not consistent: the monitor cannot deduce the ordering of states from their arrival order, due to variable network latencies
- This may lead to wrong conclusions

Wrong global states caused by delays

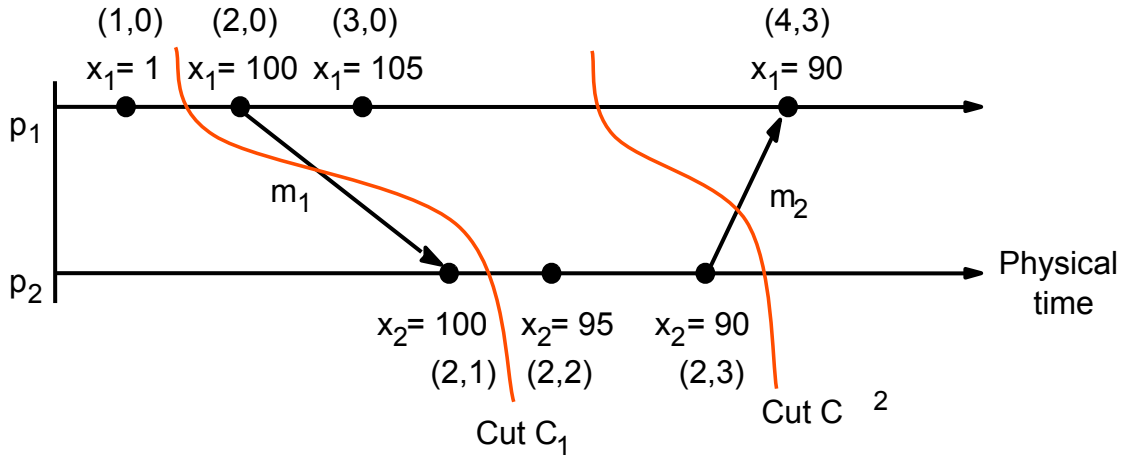


- Let the invariant be φ = "$|x_1-x_2| \leq 50$"
- If the update from p_2 arrives before that of p_1, the state appears to be $S = (x_1=1, x_2=100)$ (cut C1)
- Here φ does not hold - but actually this is not the case!

Correct ordering of state updates

- The monitor has to distinguish updates yielding consistent global states to those yielding inconsistent ones
- To this end, each process keeps a vector clock which is used to causally order the state updates
- Each state message is tagged with the vector clock
- For each process p_i, the monitor keeps a queue Q_i of incoming messages, in sending order
- Let $S = (s_1,\ldots,s_N)$ a state obtained by assembling the various states, and $V(s_i)$ the vector clock of s_i.
  Then, S is a consistent global state if and only if
  for all i,j =1,…,N: $V(s_i)[i] \geq V(s_j)[i]$
- "the number of p_i's events known at p_j when it sent s_j is no more than the number of events the had occurred at p_i when it sent s_i"

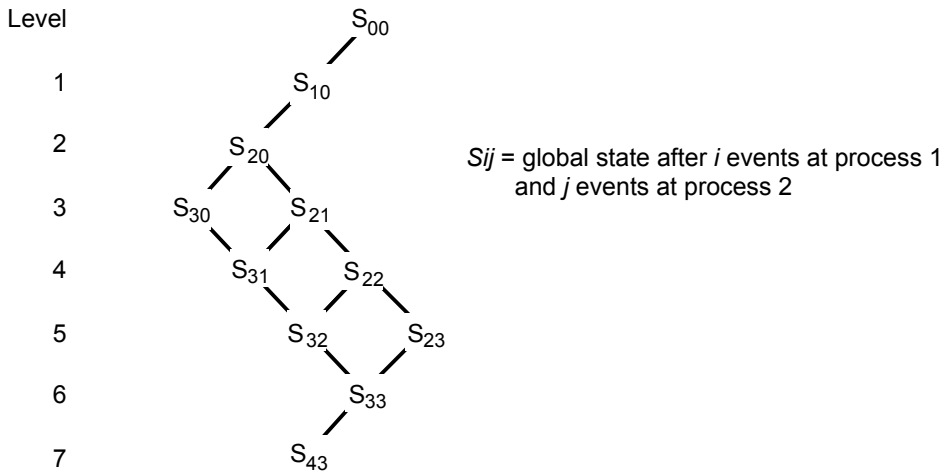
Vector timestamps and variable values



- Accepting the update "x_2=100" from p_2 would yield an inconsistent state—indeed, $V(s_1)=(1,0)$, $V(s_2)=(2,1)$, and it is not true that $V(s_1)[1] \geq V(s_2)[1]$.
- So the update is kept in the queue, until the update "x_1=100" from p_1 arrives

The lattice of global states

- Each time an update is accepted, the global state changes
- This leads to a *lattice* of possible states.
- All linearization can be obtained by traversing this lattice
- A state $S'=(s_1,\ldots s_i',\ldots,s_N)$ is *reachable* from $S=(s_1,\ldots s_i,\ldots,s_N)$ if and only if for $j = 1,\ldots N$, $j \neq i$: $V(s_j)[j] \geq V(s_i)[j]$



Algorithms to evaluate *possibly* ϕ and *definitely* ϕ

1. *Evaluating possibly* ϕ *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* ϕ";

2. *Evaluating definitely* ϕ *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* ϕ";