Module 5: Coordination

VIT®

Vellore Institute of Technology

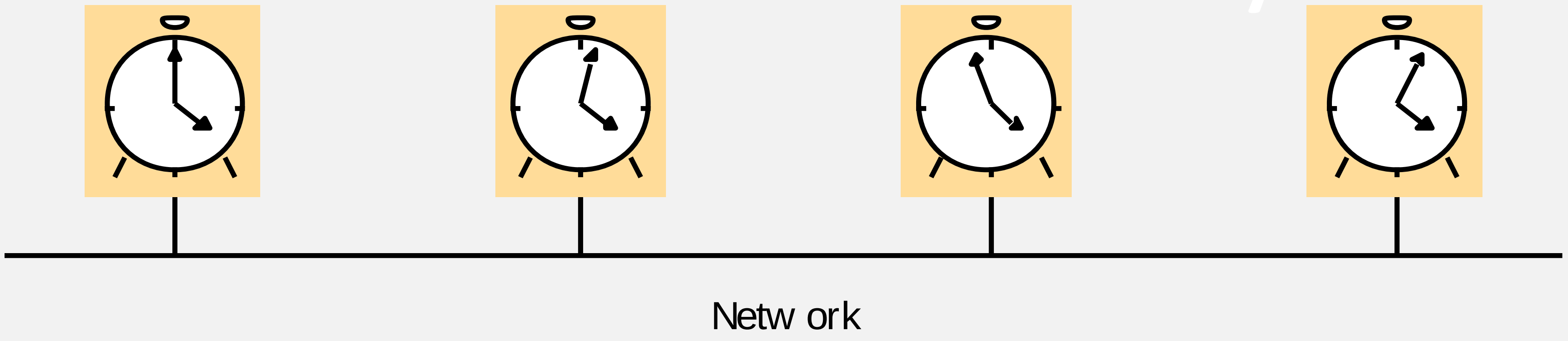
(Deemed to be University under section 3 of UGC Act, 1956)

Contents

Time

- Time is an important and interesting issue.
  - Time is a quantity often want to measure the happening of a certain event accurately. E.g. e-commerce transaction time at merchant and bank's computers
  - Algorithms depend upon clock synchronization. E.g. use of timestamps to serialize transactions to maintain data consistency. Order of events required
  - Synchronize local clock with an authoritative, external source of time. Atomic oscillator clock is the most accurate physical clock. International Atomic Time and Coordinated Universal Time.

Skew between computer clocks in a distributed system



- Each node maintain a physical clock. However, they tend to drift even after an accurate initial setting.
- Skew: the difference between the readings of any two clocks.
- Clock drift: the crystal-based clock count time at different rates. Oscillator has different frequency.
- **Drift rate** is usually used to measure the change in the offset per unit of time. Ordinary quartz crystal clock, 1second per 11.6 days.

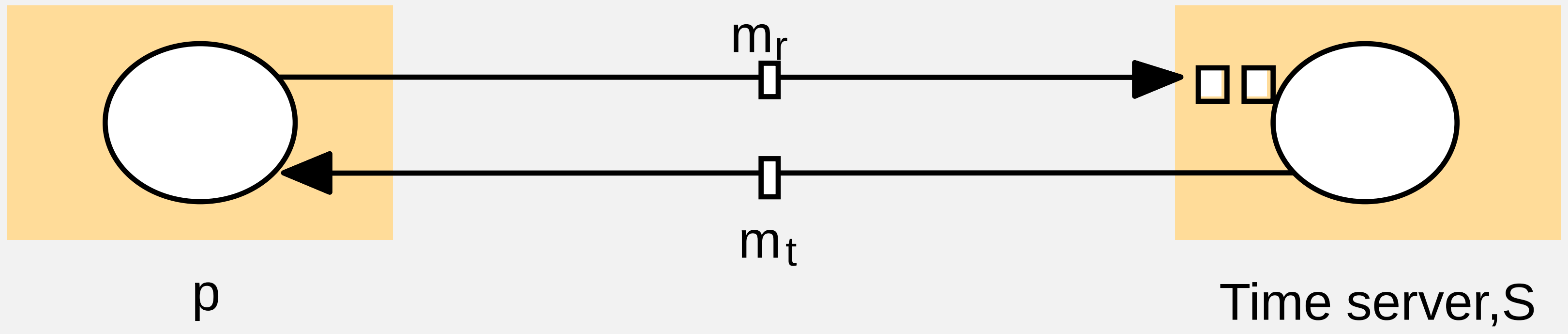
Synchronizing Physical Clocks

- External synchronization: Ci is synchronized to a common standard.
  - $|S(t) - Ci(t)| < D$, for i = 1,2,…N and for all real time t, namely clock Ci are accurate to within the bound D. S is standard time.
- Internal synchronization: Ci is synchronized with one another to a known degree of accuracy.
  - $|Ci(t) - Cj(t)| < D$ for i,j=1,2,…N, and for all real time t, namely, clocks Ci agree with each other within the bound D.

Simplest Case of Internal Synchronization

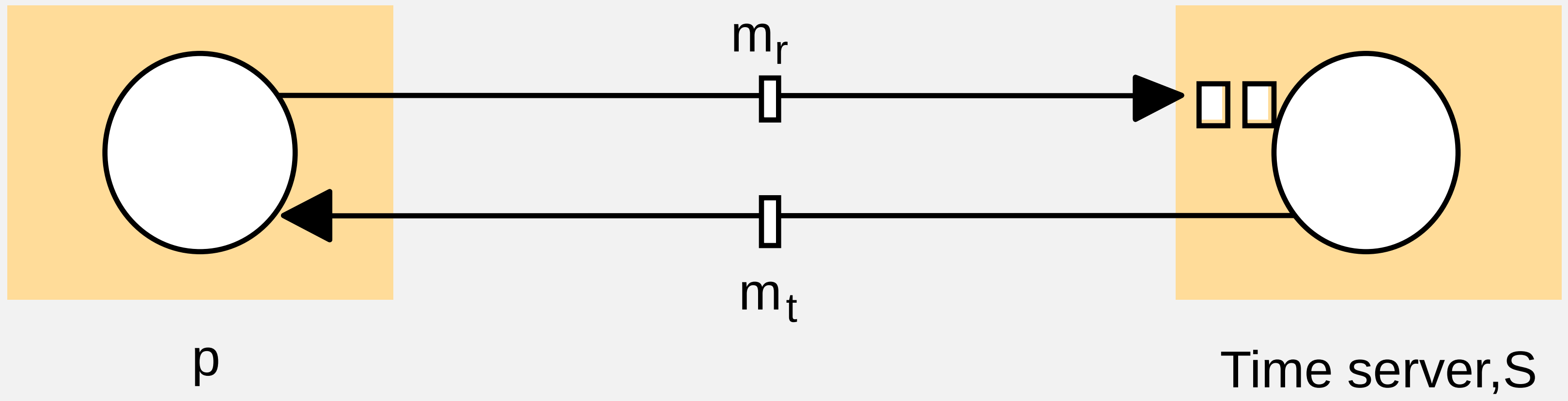
- In a synchronous system, bounds exist for clock drift rate, transmission delay and time for computing of each step.
- One process sends the time t on it local clock to the other in a message m. The receiver should set its clock to $t+T_{trans}$. It doesn't matter whether t is accurate or not
  - Synchronous system: T_{trans} could range from min to max. The uncertainty is u=(max-min). If receiver set clock to be t+min or t+max, the skew is as much as u. If receiver set the clock to be t+(min+max)/2, the skew is at most u/2.
  - Asynchronous system: no upper bound max. only lower bound.

Clock synchronization using a time server



- Cristian's method: Time server, connected to a device receiving signals from UTC. Upon request, the server S supplies the time t according to its clock.
- The algorithm is probabilistic and can achieve synchronization only if the observed round trip time are short compared with required accuracy.
- From p's point of view, the earliest time S could place the time in mt was min after p dispatch mr. The latest was min before mt arrived at p.

Cristian's method



- The time of S by the time p receives the message mt is in the range of [t+min, t+Tround –min].
- P can measure the roundtrip time then p should set its time as (t + Tround/2) as a good estimation.
- The width of this range is (Tround -2min). So the accuracy is +-(Tround /2-min)

Cristian’s algorithm

- Suffers from the problem associated with single server that single time server may fail.
- Cristian suggested to use a group of synchronized time servers. A client multicast is request to all servers and use only the first reply.
- A faulty time server that replies with spurious time values or an imposter time server with incorrect times.

Berkeley Algorithm

- Internal synchronization when developed for collections of computers running Berkeley UNIX.
- A coordinator is chosen to act as the master.
- It periodically polls the other computers whose clocks are to be synchronized, called slave.
- The slaves send back their clock values to it.
- The master estimate their local clock times by observing the round-trip time similar to Cristian’s method.
- It averages the values obtained including its own.

Berkeley Algorithm

- Instead of sending the updated current time back to other computers, which further introduce uncertainty of message transmission, the master sends the amount by which each individual slave’s clock should adjust.

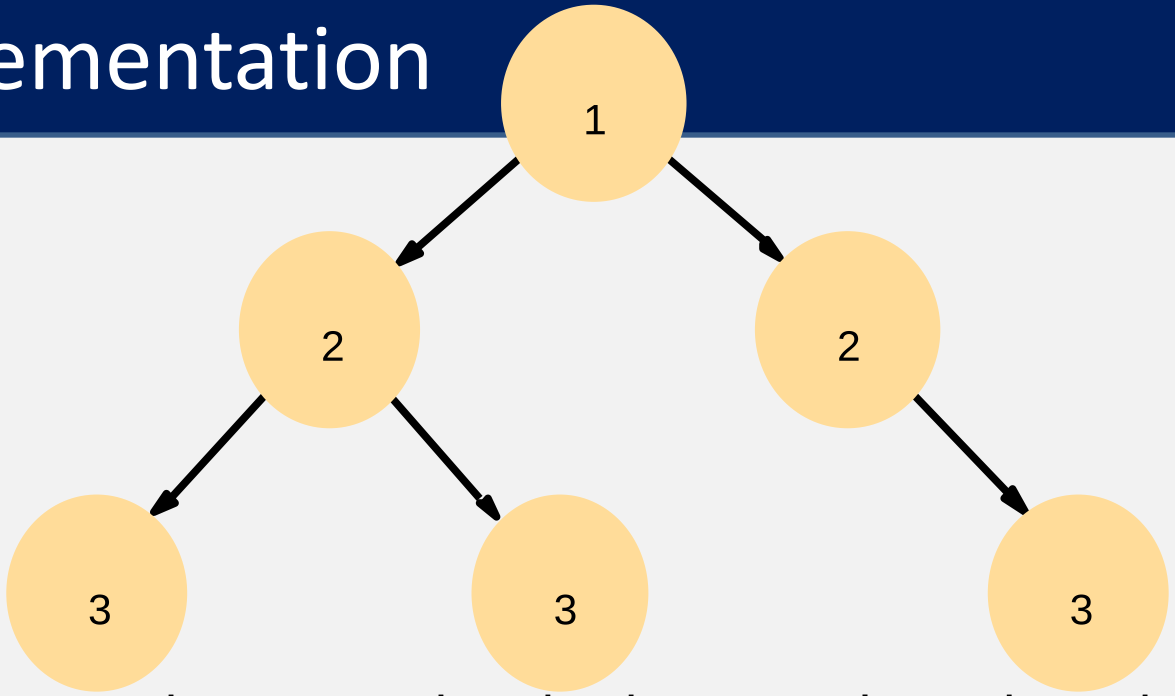
Berkeley Algorithm

- The master takes a fault-tolerant average, namely a subset of clocks is chosen that do not differ from one another by more than a specified bound.
- The algorithm eliminates readings from faulty clocks. Such clocks could have a adverse effect if an ordinary average was taken.

The Network Time Protocol

- Cristian's method and Berkeley algorithm are primarily for Intranets.
- The Network Time Protocol(NTP) defines a time service to distribute time information over the Internet.
  - Clients across the Internet to be synchronized accurately to UTC. Statistical techniques
  - Reliable service that can survive lengthy losses of connectivity. Redundant servers and redundant paths between servers.
  - Clients resynchronized sufficiently frequently to offset the rates of drift.
  - Protection against interference with time services. Authentication technique from claimed trusted sources

An example synchronization subnet in an NTP implementation



Note: Arrows denote synchronization control, numbers denote strata.

Hierarchical structure called synchronization subnet

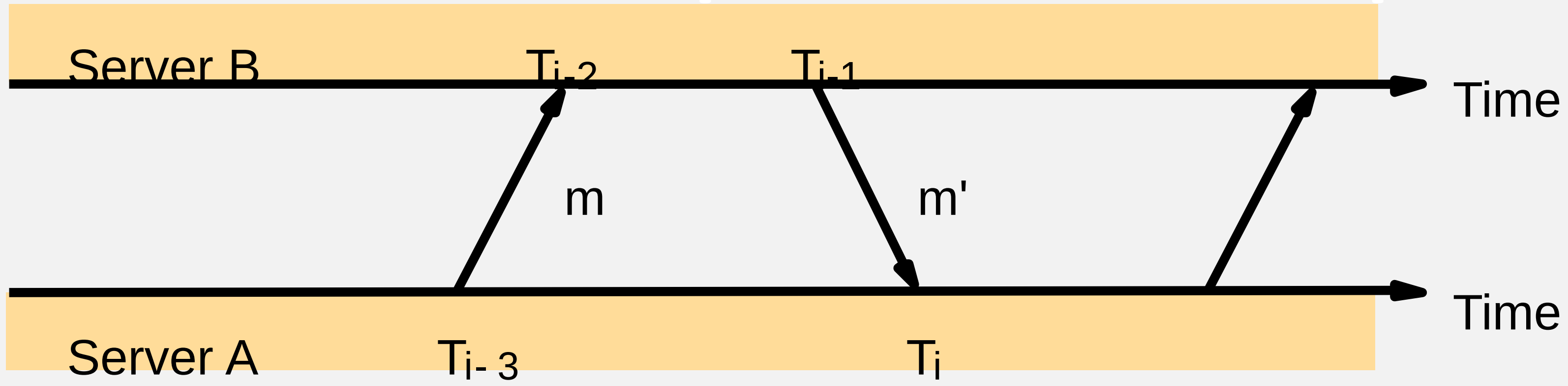
- Primary server: connected directly to a time source.
- Secondary servers are synchronized with primary server.
- Third servers are synchronized with secondary servers.

Such subnet can reconfigure as servers become unreachable or failures occur.

The Network Time Protocol Server

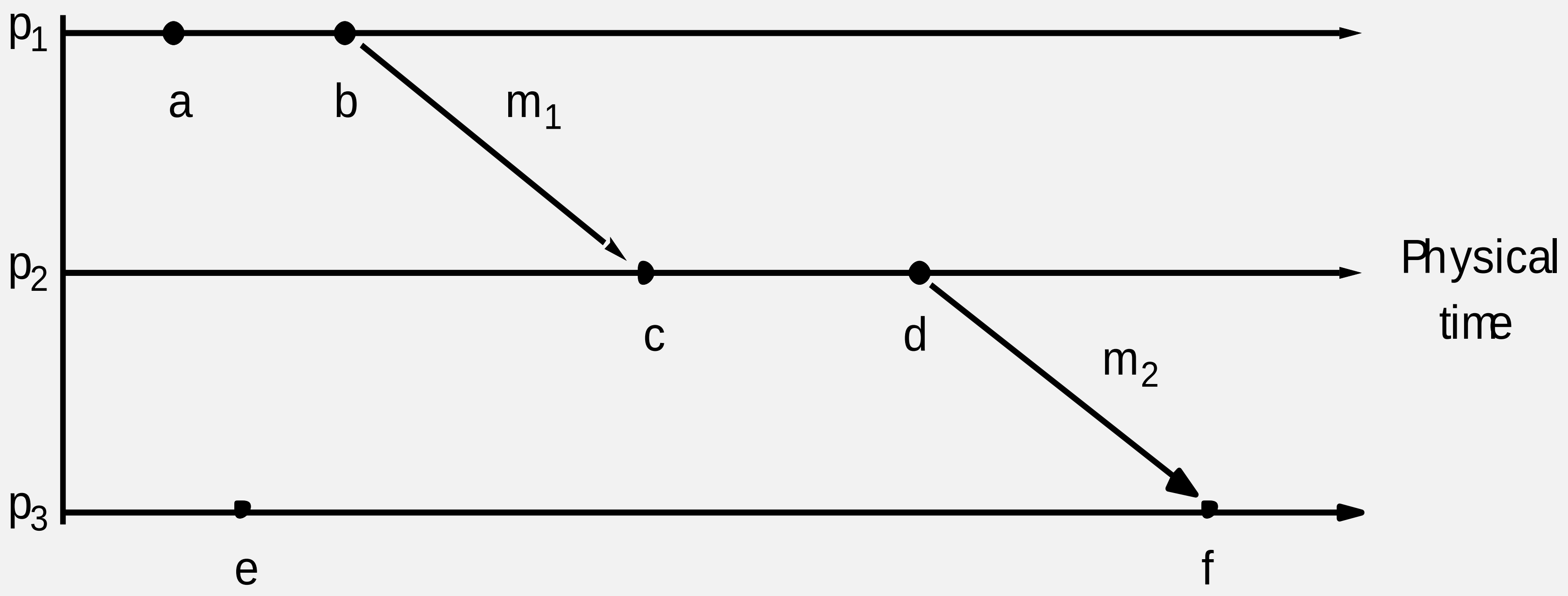
- NTP servers synchronize in one of three modes:
  - 1. Multicast mode: for high-speed LAN. One or more servers periodically multicasts the time to servers connected by LAN, which set their times assuming small delay. Achieve low accuracy.
  - 2. Procedure call: similar to Cristian's algorithm. One server receives request, replying with its timestamp. Higher accuracy than multicast or multicast is not supported.
  - 3. Symmetric mode: used by servers that supply time in LAN and by higher level of synchronization subnet. Highest accuracy. A pair of servers operating in symmetric mode exchange messages bearing timing information.

Messages exchanged between a pair of NTP peers



In all modes, messages are delivered unreliably, using UDP Internet transport protocol.
In procedure-call and symmetric mode, processes exchange pairs of messages.
Each message bears timestamps of recent message events: the local times when the previous NTP message between the pair was sent and received, and the local time when the current message was transmitted.
The recipient of the NTP message notes the local time when it receives the message.

Events occurring at three processes



Logical Time and Logical Clocks

- In single process, events are ordered by local physical time. Since we cannot synchronize physical clocks perfectly across a distributed system, we cannot use physical time to find out the order of any arbitrary pair of events.

Logical Time and Logical Clocks

- We will use logical time to order events happened at different nodes. Two simple points:
  - If two events occurred at the same process, then they occurred in the order in which pi observes them
  - Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message

Happen-before Relation/ Causal Ordering

Lamport (1978) called the partial ordering by generalizing these two relationships the happened-before relation.

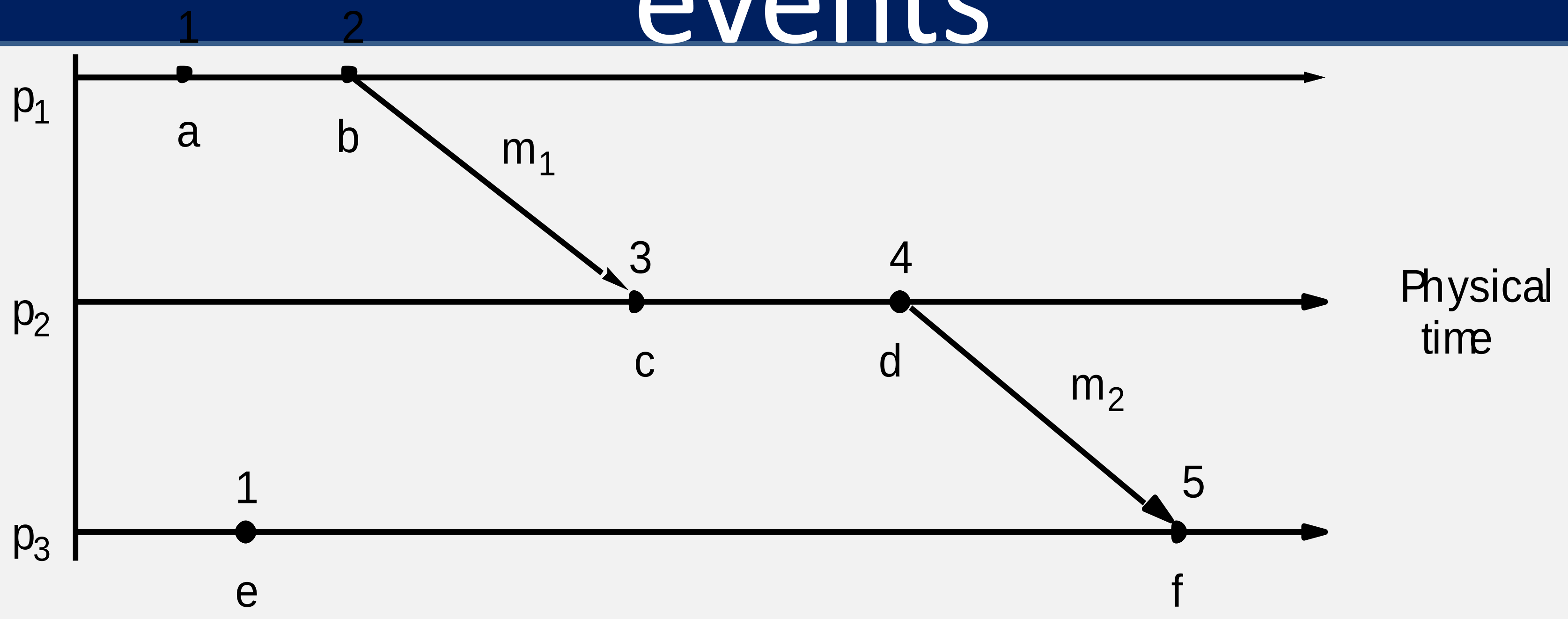
$$HB1: If\ \exists\ process\ p_i : e \rightarrow_i e', then\ e \rightarrow e'$$

$$HB2: For\ any\ message\ m,\ send(m) \rightarrow receive(m)$$

$$HB3: If\ e, e', and\ e''\ are\ events,\ such\ that\ e \rightarrow e'\ and$$

$$e' \rightarrow e'', then\ e \rightarrow e''$$

Lamport timestamps for the events



$a \rightarrow_1 b$ *and* $c \rightarrow_2 d$

$b \rightarrow c$ *and* $d \rightarrow f$

combing them, $a \rightarrow f$

$a \not\rightarrow e$ *and* $e \not\rightarrow a$

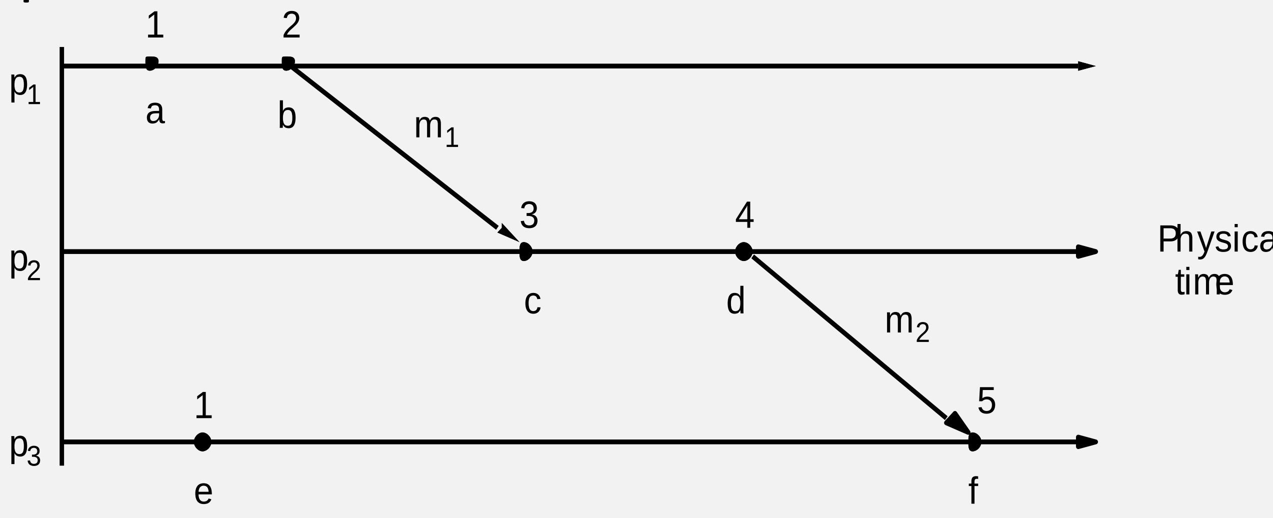
concurrent $a \parallel e$

Logical Clocks

- Lamport invented a logical clock Li, which is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock. Each process pi keeps its own logical clock.
- LC1: Li is incremented before each event is issued at process pi: Li = Li +1
- LC2:
  - a. Pi sends a message m, it piggybacks on m the value t = Li
  - b. On receiving (m,t), a process pj computes Lj=max(Lj,t) and then applies LC1 before timestamping the event receive(m).

Logical Clock

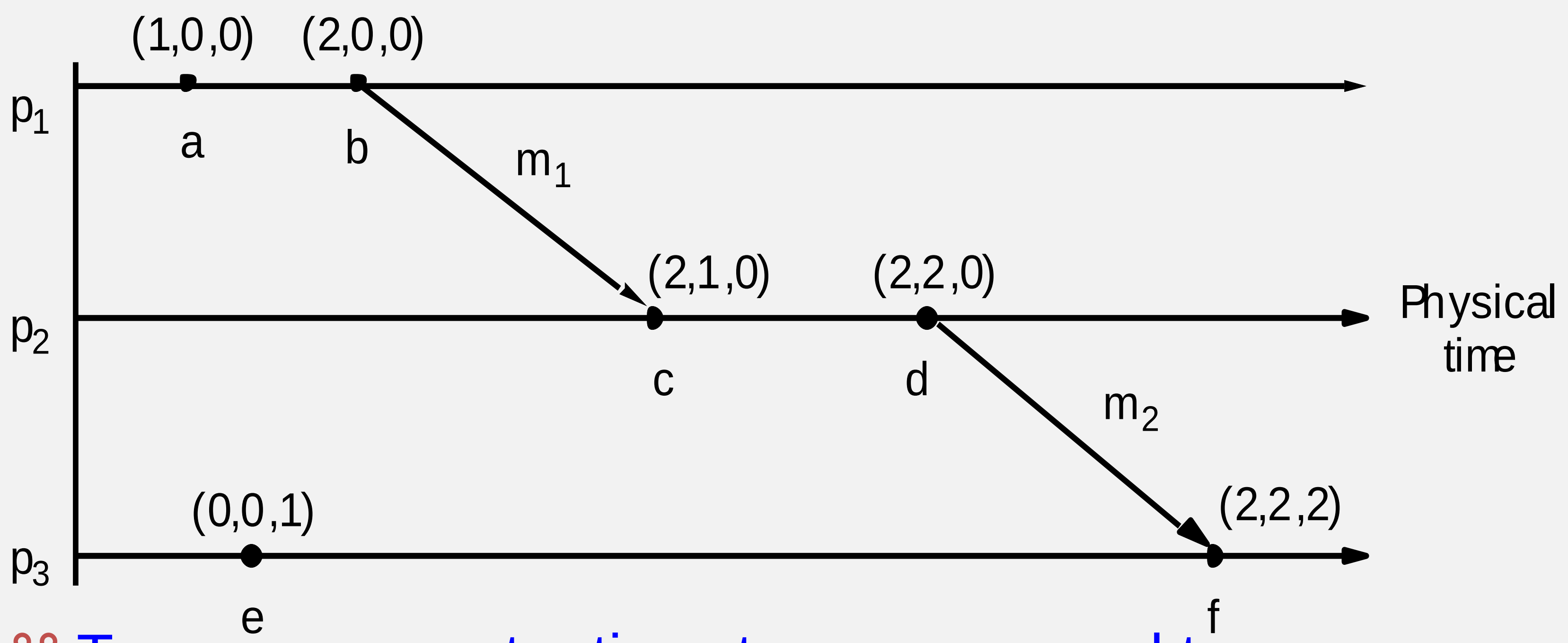
- It can be easily shown that:
- If e->e’ then L(e) < L(e’).
- However, the converse is not true. If L(e) < L(e’), then we cannot infer that e->e’. E.g b and e
- L(b)>L(e) but b||e
- How to solve this problem?



Vector Clock

- Lamport's clock: L(e)<L(e') we cannot conclude that e->e'.
- Vector clock to overcome the above problem.
- N processes is an array of N integers. Each process keeps its own vector clock Vi, which it uses to timestamp local events.
- VC1: initially, Vi[j] = 0, for i,j = 1,2…N
- VC2: just before pi timestamps an event, it sets Vi[i] = vi[i]+1
- VC3: pi includes the value t= Vi in every message it sends
- VC4: when pi receives a timestamp t in a message, it sets Vi[j]=max(Vi[j], t[j])for j =1,2…,N. Merge operation.

Vector timestamps for the events



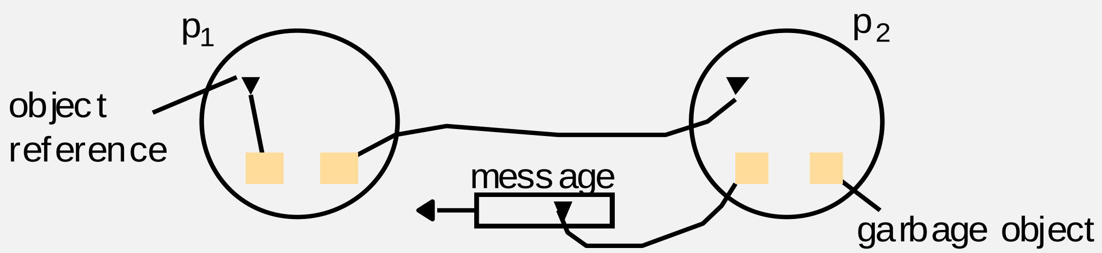
- To compare vector timestamps, we need to compare each bit. Concurrent events cannot find a relationship.
- Drawback compared with Lamport time, taking up an amount of storage and message payload proportional to N.

Detecting global properties

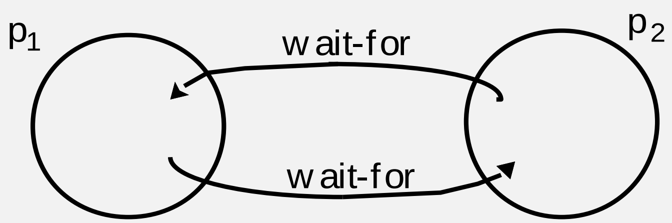
- We want to find out whether a particular property is true of a distributed system as it executes.
- We will see three examples:
  - Distributed garbage collection: if there are no longer any reference to objects anywhere in the distributed system, the memory taken up by the objects should be reclaimed.
  - Distributed deadlock detection: when each of a collection of processes waits for another process to send it a message, and where there is a cycle in the graph of this "wait-for" relationship.
  - Distributed termination detection: detect if a distributed algorithm has terminated. It seems that we only need to test whether each process has halted. However, it is not true. E.g. two processes and each of which may request values from the other. It can be either in passive or active state. Passive means it is not engaged in any activity but is prepared to respond. Two processes may both in passive states. At the same time, there is a message in on the way from P2 to P1, after P1 receives it, it will become active again. So the algorithm has not terminated.

Detecting global properties

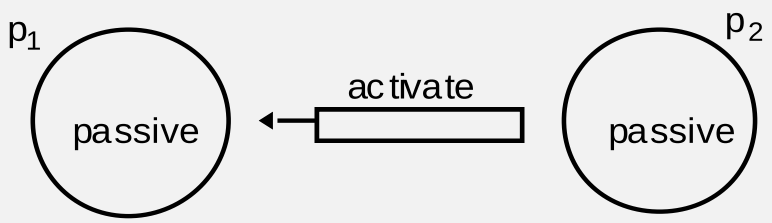
a. Garbage collection



b. Deadlock



c. Termination



Global States

- 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”.

Definitions

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

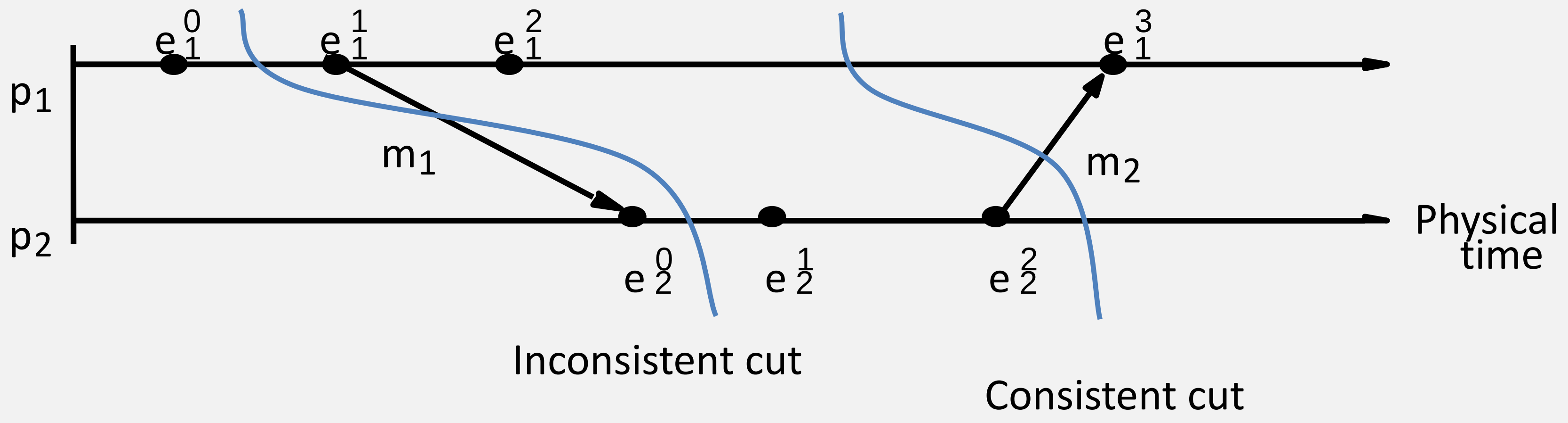
$$finite\ prefix\ of\ history : h_i^k = < e_i^0, e_i^1, e_i^2, \ldots e_i^k >$$

- ⌘ A series of events occurs at each process. Each event is either an internal action of the process (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 kth event occurs, so S_i^0 is the initial state of Pi.
- ⌘ Thus the global state corresponds to initial prefixes of the individual process histories.

Global States

- How to detect a set of consistent states among distributed processes?
- This set of local states for a global set called the “cut”. This is made complex by a message passed around among the processes.
- A cut C is consistent if for each event it contains all the events that happened-before(HB) that event.
- A consistent global state is one that corresponds to a consistent cut.

Global States



⌘ 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.

Global States

- Inconsistent cut: since P2 contains receiving of m1, but at P1 it does not include sending of that message. This cut shows the an effect without a cause. We will never reach a global state that corresponds to process state at the frontier by actual execution under this cut.
- Consistent cut: it includes both the sending and receipt of m1. It includes the sending but not the receipt of m2. It is still consistent with actual execution.

Global States

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

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

- 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 this happened-before relation.

Global state predicate

- Global state predicate is a function that maps from the set of global states of processes n the system to true or false.
- Stable characteristics associated with object being garbage, deadlocked or terminated: once the system enters a state in which the predicate is True. It remains True in all future states reachable from that state.
- Safety (evaluates to deadlocked false for all states reachable from S0)
- Liveness (evaluate to reaching termination true for some of the states reachable from S0)

Chandy and Lamport’s ‘snapshot’ algorithm

- Chandy and Lamport(1985) describe a “snapshot” algorithm for determining global states of distributed system.
- Record a set of process and channel states for a set of processes Pi such that even though the combination of recorded states may never have occurred at the same time, the recorded global state is consistent.
- The algorithm records state locally at processes without giving a method for gathering the global state.

Assumption of 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.

Snapshots Ideas

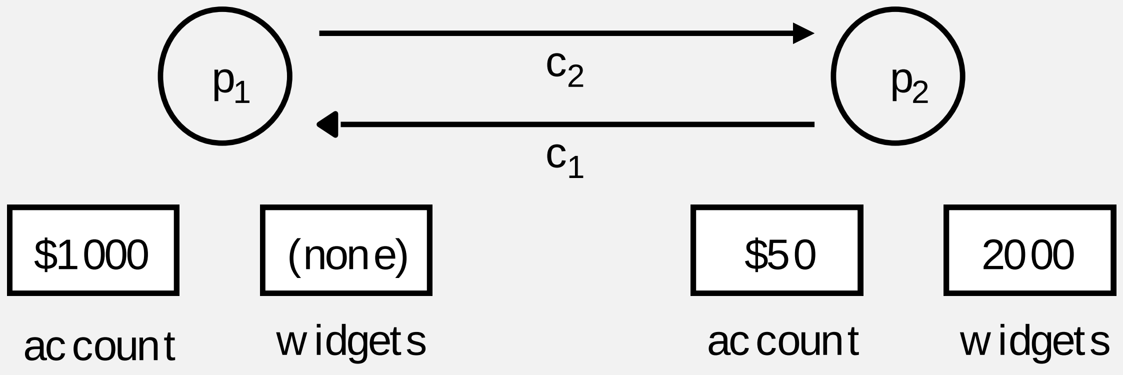
- Each process records its own state and also for each incoming channel a set of messages sent to it.
- Allow us to record process states at different times but to account for the differential between process states in terms of message transmitted but not yet received.
- If process pi has sent a message m to process pj, but pj has not received it, then we account for m as belong to the state of the channel between them.

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 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).

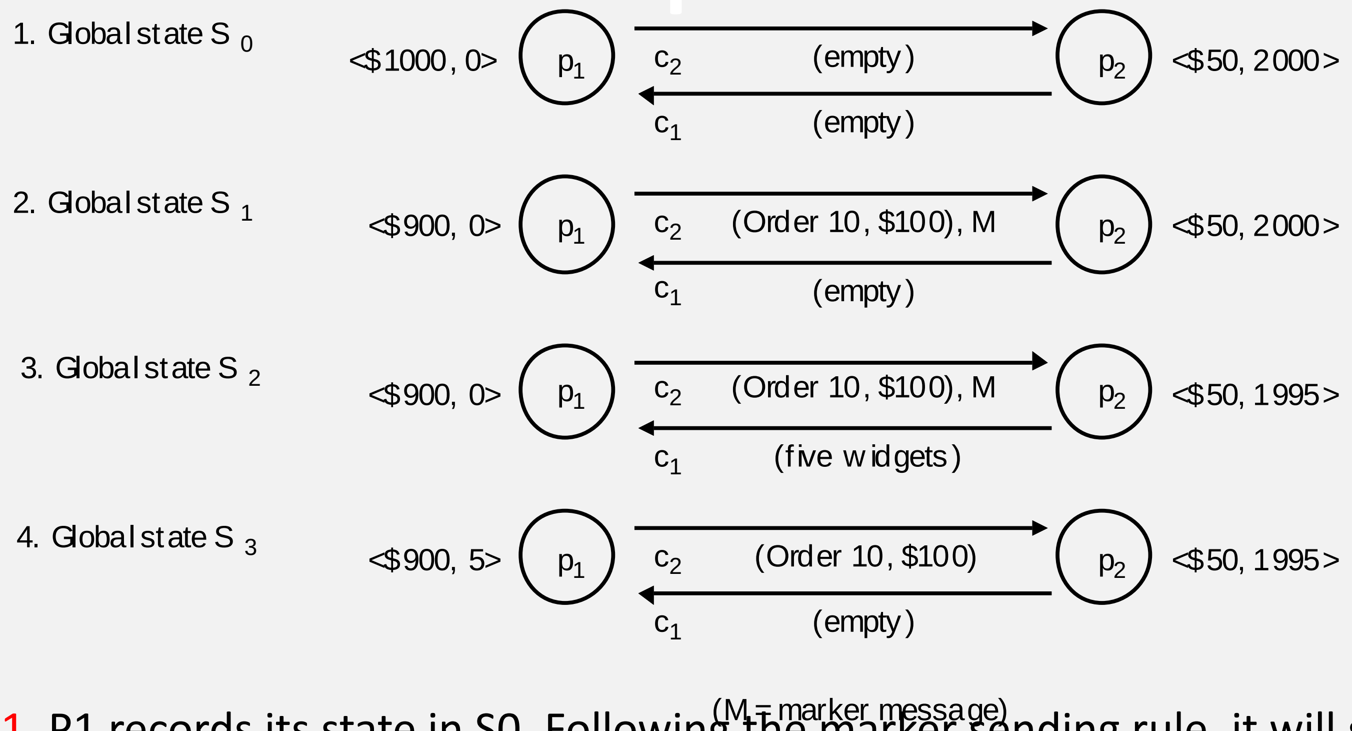
Use of special marker message. It has a dual role, as a prompt for the receiver to save its own state if it has not done so; and as a means of determining which messages to include in the channel state.

Two processes and their initial states



Two processes connected by two unidirectional channels, c1 and c2. The two 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.

The execution of the processes



Final recorded state is:
P1<$1000,0>
P2<$50,1995>
C1<five widgets>
C2<>

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 (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.

- Instructor’s Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 4 ,© Pearson Education 2005