CSD511 – Distributed Systems

分散式系統

Chapter 11
Time and Global States

吳俊興

國立高雄大學 資訊工程學系

Chapter 11 Time and Global States

11.1 Introduction

- Importance of time in distributed systems
  - A quantity to timestamp events accurately
    - To know what time a particular event occurs
    - i.e. Recording when an e-commerce transaction occurs
  - A synchronization source for several distributed algorithms
    - To maintain consistency of distributed data
    - i.e. Eliminating duplicate updates
  - A timing source for multiple events
    - To provide relative order of two events
    - i.e. Ensuring the order of cause and effect
- Clocks in computers to establish
  - *Time* at which an event occurred
  - *Duration* of an event or interval between two events
  - *Sequence* of a series of events or the order in which events occurred

11.2 Clocks, Events and Process States

- A distributed system consists of a collection *P* of ***N*** processes p_i, $i = 1,2,\ldots N$
  - Each process p_i has a state s_i consisting of its variables (which it transforms as it executes)
  - Processes communicate only by messages (via a network)
    - **Actions** of processes: *Send, Receive,* change own state
- ***Event:*** the occurrence of a single action that a process carries out as it executes
- Events at a single process p_i, can be placed in a total **ordering** denoted by the relation $\rightarrow_i$ between the events. i.e.
  - $e \rightarrow_i e'$ if and only if event *e* occurs before event *e'* at process p_i
- A history of process p_i is a series of events ordered by $\rightarrow_i$
  - ***history***$(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$

Clocks

To timestamp events, use the computer's clock

- At **real time, *t***, the OS reads the time on the computer's **hardware clock $H_i(t)$**
- It calculates the time on its **software clock $C_i(t) = \alpha H_i(t) + \beta$**
  - e.g. a 64 bit value giving nanoseconds since some base time
  - **Clock resolution**: period between updates of the clock value
- In general, the clock is not completely accurate
  - but if C_i behaves well enough, it can be used to timestamp events at p_i

Skew between computer clocks in a distributed system

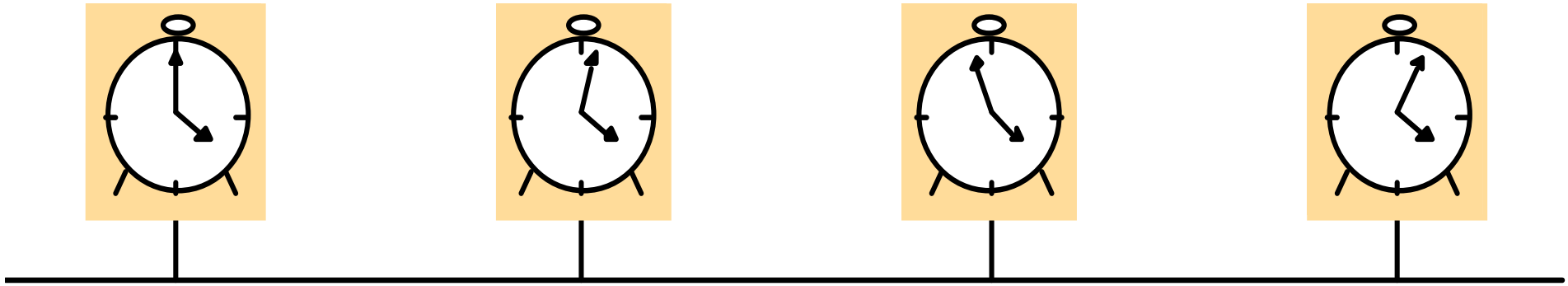


Figure 11.1

Computer clocks are not generally in perfect agreement

- ***Clock skew***: the difference between the times on two clocks (at any instant)
- Computer clocks use crystal-based clocks that are subject to physical variations
  - ***Clock drift***: they count time at different rates and so diverge (frequencies of oscillation differ)
  - ***Clock drift rate***: the difference per unit of time from some ideal reference clock
  - Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10^{-6} secs/sec).
  - High precision quartz clocks drift rate is about 10^{-7} or 10^{-8} secs/sec

Coordinated Universal Time (UTC)

- UTC is an international standard for time keeping
  - It is based on atomic time, but occasionally adjusted to astronomical time
  - International Atomic Time is based on very accurate physical clocks (drift rate 10^{-13})
- It is broadcast from radio stations on land and satellite (e.g. GPS)
- Computers with receivers can synchronize their clocks with these timing signals (by requesting time from GPS/UTC source)
  - Signals from land-based stations are accurate to about 0.1-10 millisecond
  - Signals from GPS are accurate to about 1 microsecond

11.3 Synchronizing physical clocks

Two models of synchronization

- External synchronization: a computer's clock C_i is synchronized with an external authoritative time source *S*, so that:
  - $|S(t) - C_i(t)| < D$ for $i = 1, 2, \ldots N$ over an interval, *I* of real time
  - The clocks C_i are **accurate** to within the bound *D*.
- Internal synchronization: the clocks of a pair of computers are synchronized with one another so that:
  - $| C_i(t) - C_j(t)| < D$ for $i = 1, 2, \ldots N$ over an interval, *I* of real time
  - The clocks C_i and C_j **agree** within the bound *D*.

Internally synchronized clocks are not necessarily externally synchronized, as they may drift collectively

- if the set of processes *P* is synchronized externally within a bound *D*, it is also internally synchronized within bound 2*D (worst case polarity)*

Clock correctness

- ***Correct clock***: a hardware clock H is said to be correct if its drift rate is within a bound $\rho > 0$ (e.g. 10^{-6} secs/ sec)

  This means that the error in measuring the interval between real times t and t' is bounded:

  – $(1 - \rho)\ (t' - t) \leq H(t') - H(t) \leq (1 + \rho)\ (t' - t)$ (where $t'>t$)
  – Which forbids jumps in time readings of hardware clocks
- ***Clock monotonicity***: weaker condition of correctness
  – $t' > t \Rightarrow C(t') > C(t)$
  – e.g. required by Unix *make*
  – A hardware clock that runs fast can achieve monotonicity by adjusting the values of α and β such that $C_i(t) = \alpha H_i(t) + \beta$
- ***Faulty clock***: a clock not keeping its correctness condition
  – *crash failure* - a clock stops ticking
  – *arbitrary* failure - any other failure
    - e.g. jumps in time; Y2K bug

11.3.1 Synchronization in a synchronous system

A synchronous distributed system is one in which the following bounds are defined (Section 2.3.1; p.48):

- the 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 (min and max)
- each process has a local clock whose drift rate from real time has a known bound

- Internal synchronization in a synchronous system
  - One process p_1 sends its local time t to process p_2 in a message m
  - p_2 could set its clock to $t + T_{trans}$ where T_{trans} is the time to transmit m
  - T_{trans} is unknown but $min \leq T_{trans} \leq max$
  - uncertainty $u = max - min$. Set clock to $t + (max - min)/2$ then skew $\leq u/2$

11.3.2 Cristian's method for an 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 $\pm (T_{round}/2 - min)$:
    - because 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 $[t+min, t + T_{round} - min]$
    - the width of the range is $T_{round} + 2min$

T_{round} is the round trip time recorded by *p*
min is an estimated minimum round trip time

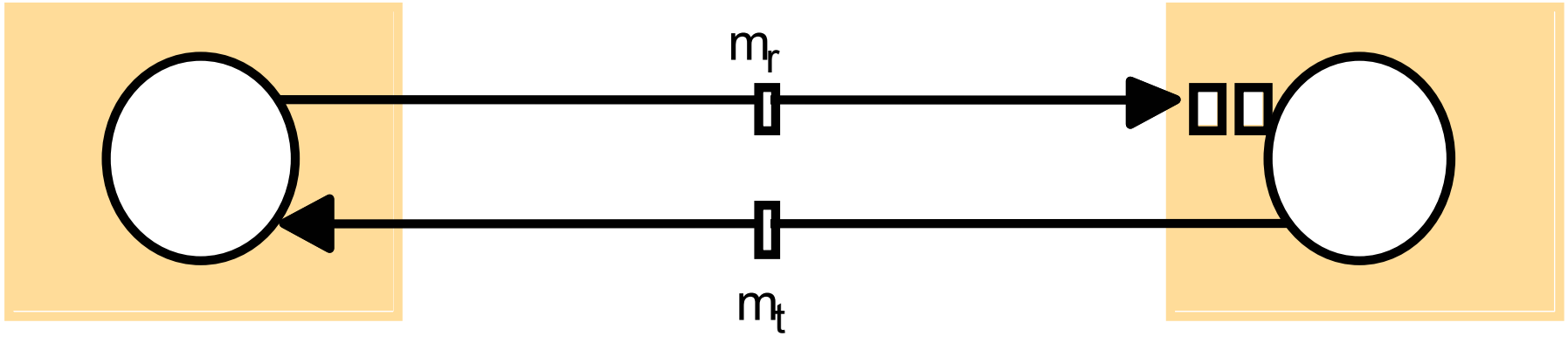


Figure 11.2

11.3.3 The Berkeley algorithm

- Problem with Cristian's algorithm
  - a single time server might fail, so they suggest the use of a group of synchronized servers
  - it does not deal with faulty servers
- Berkeley algorithm (also 1989)
  - 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 (better than sending the time which depends on the round trip time)
  - Measurements
    - 15 computers, clock synchronization 20-25 millisecs drift rate < 2×10^{-5}
    - If master fails, can elect a new master to take over (not in bounded time)

11.3.4 Network Time Protocol (NTP)

- A time service for the Internet - synchronizes clients to UTC
  - Reliability from redundant paths, scalable, authenticates time sources
- Architecture
  - Primary servers are connected to UTC sources
  - Secondary servers are synchronized to primary servers
  - Synchronization subnet - lowest level servers in users' computers
    - strata: the hierarchy level

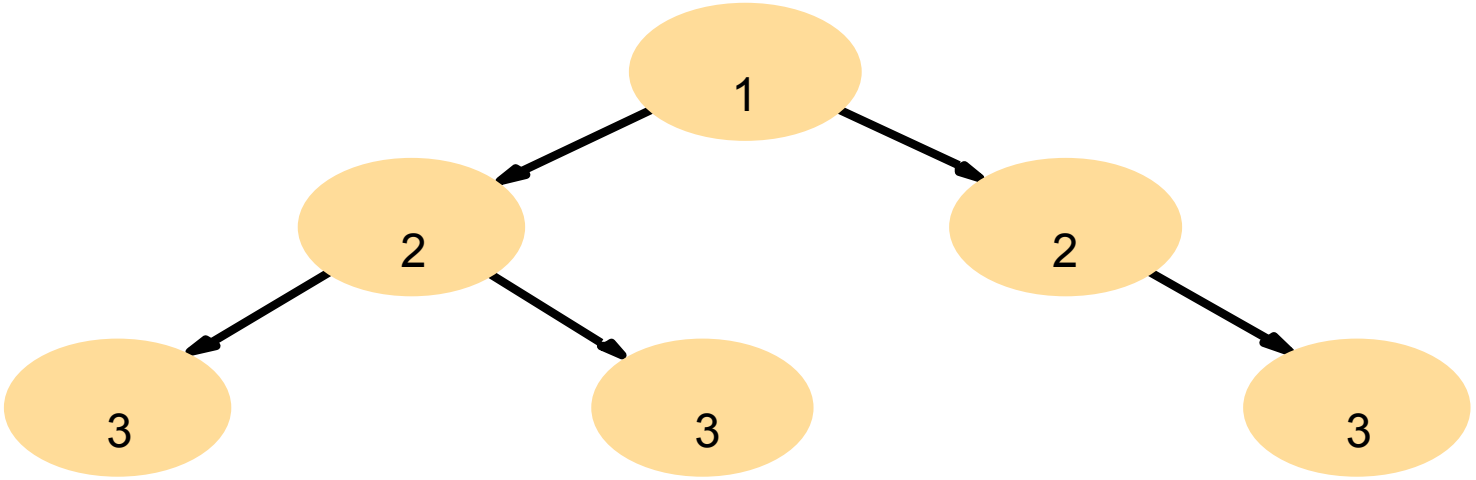


Figure 11.3 An example synchronization subnet in an NTP implementation

NTP - synchronization of servers

- The synchronization subnet can reconfigure if failures occur
  - a primary that loses its UTC source can become a secondary
  - a secondary that loses its primary can use another primary
- Modes of synchronization for NTP servers:
  - 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)

Messages exchanged between 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

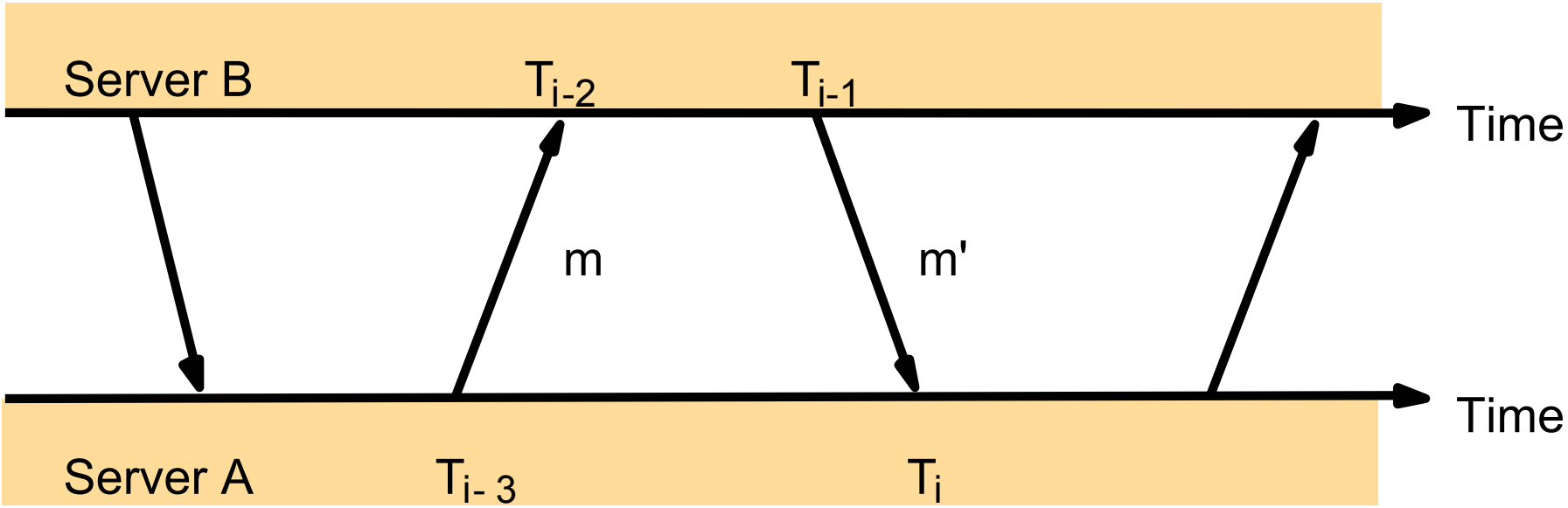


Figure 11.4

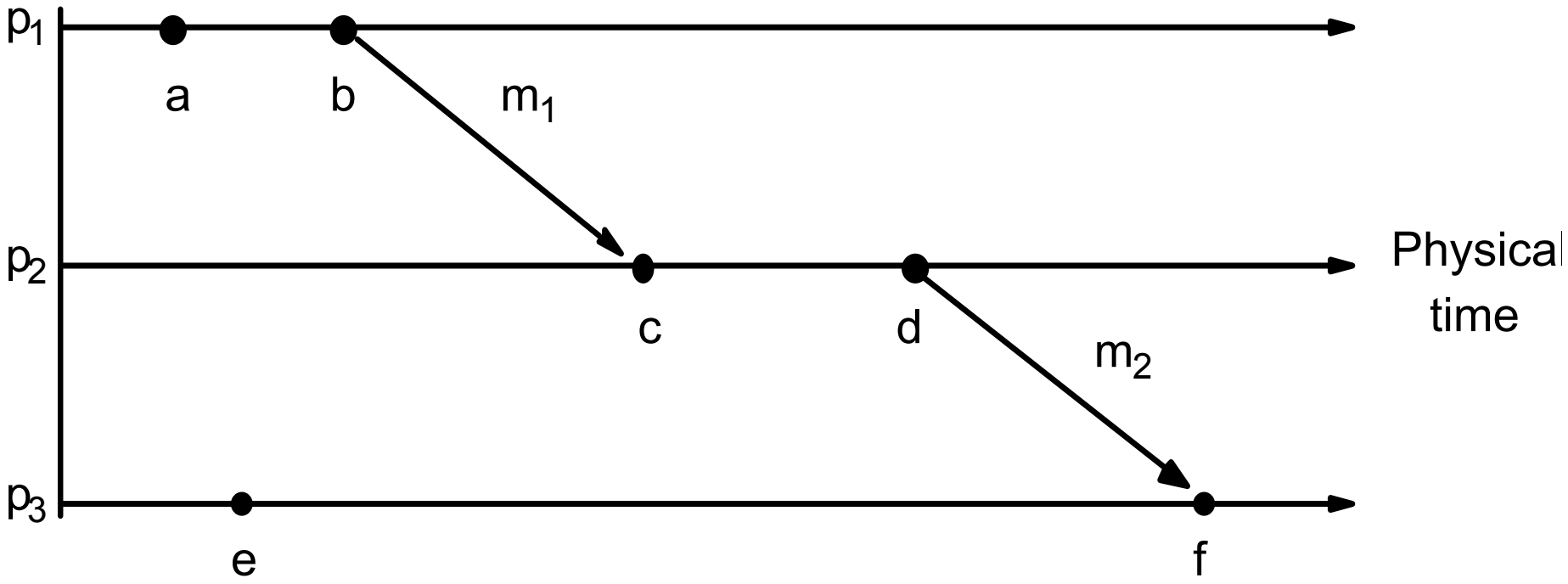
Accuracy of NTP

- Estimations of clock offset and message delay
  - For each pair of messages between two servers, NTP estimates an offset o_i (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 the fact that $t, t' > 0$ it can be shown that
    $o_i - d_i/2 \leq o \leq o_i + d_i/2$.
    - Thus o_i is an estimate of the offset and d_i is a measure of the accuracy
- Data filtering
  - NTP servers filter pairs $<o_i, d_i>$, estimating reliability from variation (dispersions), allowing them to select peers; and synchronization based on the lowest dispersion or min d_i ok
    - A relatively high filter dispersion represents relatively unreliable data
  - Accuracy of tens of milliseconds over Internet paths (1 ms on LANs)

11.4 Logical time and logical clocks

- Instead of synchronizing clocks, event ordering can be used
  1. If two events occurred at the same process *pi* (i = 1, 2, … N) then they occurred in the order observed by *pi*, that is order $\rightarrow_i$
  2. when a message, *m* is sent between two processes, *send(m)* happened before *receive(m)*
- Lamport[1978] generalized these two relationships into the **happened-before relation: $e \rightarrow_i e'$**
  - HB1: if $e \rightarrow_i e'$ in process p_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 $e \rightarrow e''$

Figure 11.5 Events occurring at three processes



- HB1: a→b, c→d, e→f
- HB2: b→c, d→f
- HB3: a→b→c→d→f
- **a||e**: a and e are concurrent (neither a→e nor e→a)

Lamport's logical clocks

- Each process p_i has a logical clock L_i
  - a monotonically increasing software counter
  - not related to a physical clock
- Apply Lamport timestamps to events with happened-before relation
  - LC1: L_i is incremented by 1 before each event at process p_i
  - LC2:

    (a) when process p_i sends message m, it piggybacks $t = L_i$

    (b) when p_j receives (m,t), it sets $L_j := max(L_j, t)$ and applies LC1 before timestamping the event *receive* (m)
- $e \rightarrow e'$ implies $L(e)<L(e')$, but $L(e)<L(e')$ does not imply $e \rightarrow e'$

$L(b)>L(e)$ but $b||e$

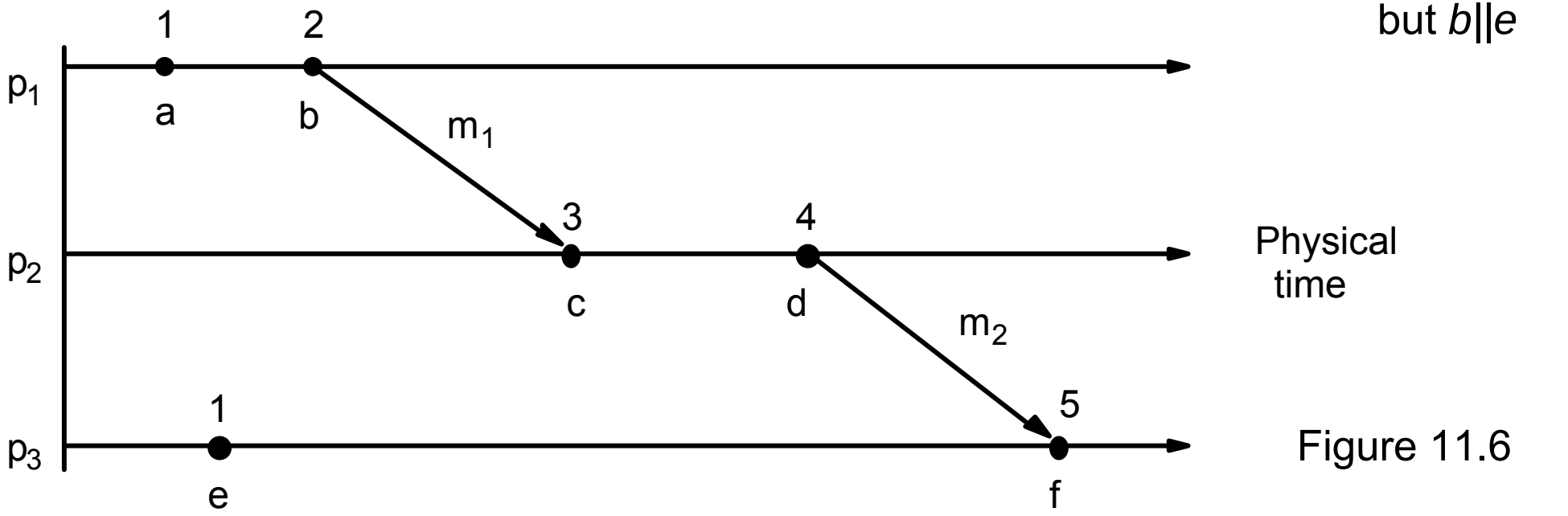


Figure 11.6

Totally ordered logical clocks

- Some pairs of distinct events, generated by different processes, may have numerically identical Lamport timestamps
  - Different processes may have same Lamport time
- Totally ordered logical clocks
  - If e is an event occurring at p_i with local timestamp T_i, and if e' is an event occurring at p_j with local timestamp T_j
  - Define global logical timestamps for the events to be (T_i, i) and (T_j, j)
  - Define $(T_i, i) < (T_j, j)$ iff
    - $T_i < T_j$ or
    - $T_i = T_j$ and $i < j$
  - No general physical significance since process identifiers are arbitrary

Vector clocks

- Shortcoming of Lamport clocks:

  $L(e) < L(e')$ doesn't imply $e \rightarrow e'$
- Vector clock: an array of N integers for a system of N processes
  - Each process keeps its own vector clock V_i to timestamp local events
  - Piggyback vector timestamps on messages
- Rules for updating vector clocks:
  - *Vi*[*i*]] is the number of events that *pi* has timestamped
  - *Vi*j*i*] ($j \neq i$) is the number of events at p_j that p_i has been affected by

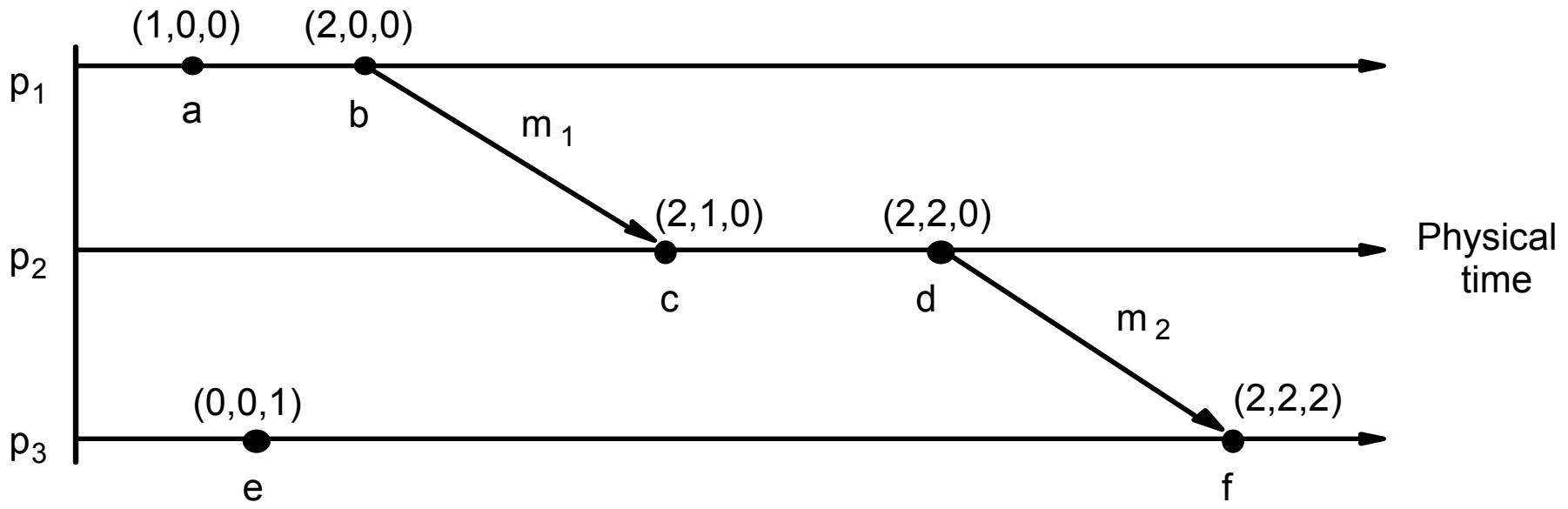
  VC1: Initially, $V_i[\ j\ ] := 0$ for p_i, j=1.. N (N processes)

  VC2: before p_i timestamps an event, $V_i[\ i\ ] := V_i[\ i\ ]+1$

  VC3: p_i piggybacks $t = V_i$ on every message it sends

  VC4: when p_i receives a timestamp t, it sets $V_i[\ j\ ] := \max(V_i[\ j\ ] , t[\ j\ ])$ for j=1..N (merge operation)

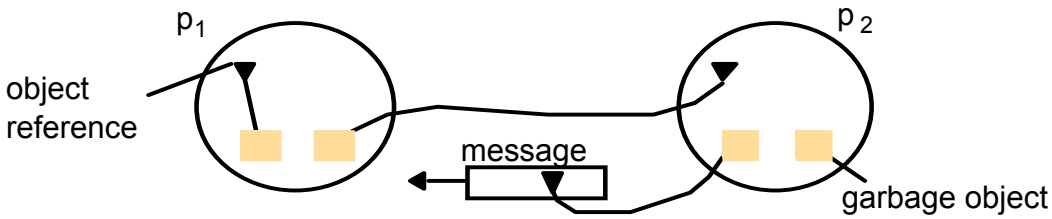
Figure 11.7 Vector timestamps for events shown in Figure 11.5



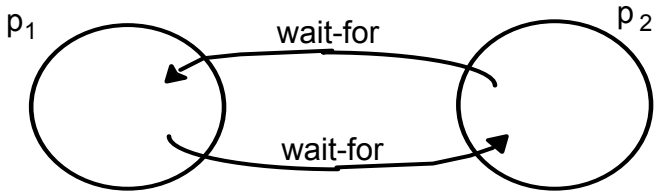
- Compare vector timestamps
  - V=V' iff V[j] = V'[j] for j=1..N
  - V>=V' iff V[j] <= V'[j] for j=1..N
  - V<V' iff V<= V' ^ V!=V'
- Figure 11.7 shows
  - *a*→*f* since V(a) < V(f)
  - c || e since neither V(c) <= V(e) nor V(e) <= V(c)

11.5 Global states

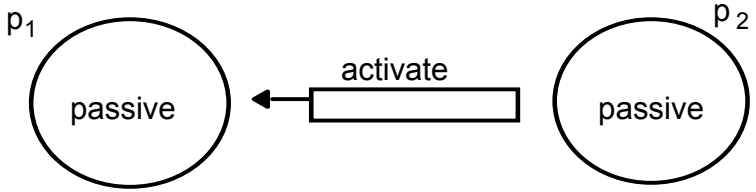
- How do we find out if a particular property is true in a distributed system? For examples, we will look at:
  - Distributed Garbage Collection
  - Deadlock Detection
  - Termination Detection
  - Debugging



a. Garbage collection



b. Deadlock

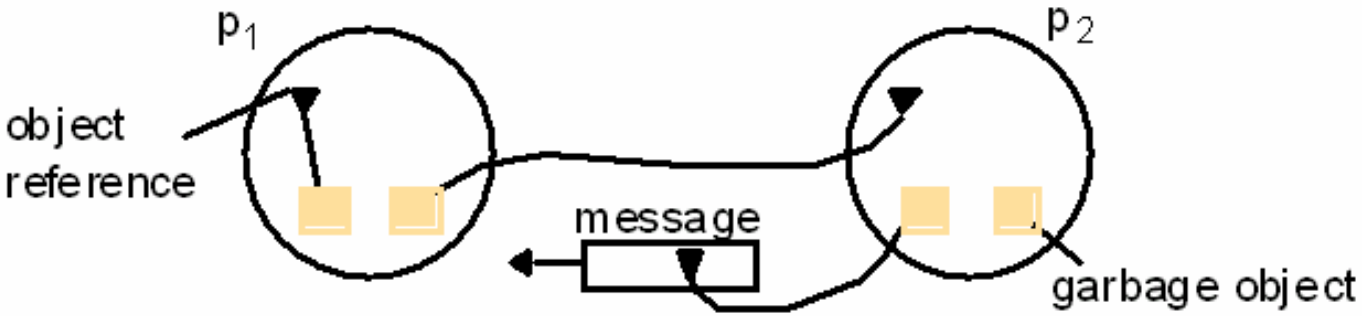


c. Termination

Figure 11.8
Detecting
global
properties

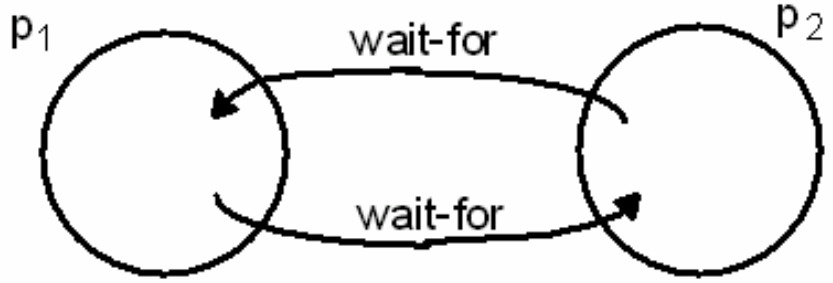
Distributed Garbage Collection

- Objects are identified as ***garbage*** when there are no longer any references to them in the system
- Garbage collection reclaims memory used by those objects
- In figure 11.8a, process p_2 has two objects that do not have any references to other objects, but one object does have a reference to a message in transit. It is not garbage, but the other p_2 object is
- Thus we must consider communication channels as well as object references to determine unreferenced objects



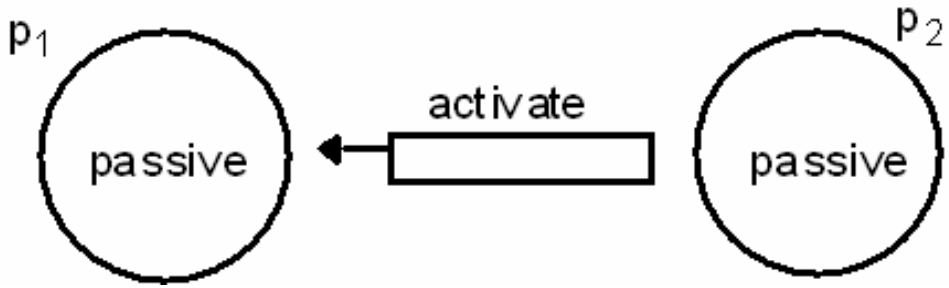
Deadlock Detection

- A distributed deadlock occurs when each of a collection of processes waits for another process to send it a message, and there is a cycle in the graph of the ***waits-for*** relationship
- In figure 11.8b, both p_1 and p_2 wait for a message from the other, so both are blocked and the system cannot continue



Termination Detection

- It is difficult to tell whether a distributed algorithm has terminated. It is not enough to detect whether each process has halted
- In figure 11.8c, both processes are in passive mode, but there is an activation request in the network
- Termination detection examines multiple states like deadlock detection, except that a deadlock may affect only a portion of the processes involved, while *termination detection must ensure that all of the processes have completed*



Distributed Debugging

- Distributed processes are complex to debug. One of many possible problems is that consistency restraints must be evaluated for simultaneous attribute values in multiple processes at different instants of time (Section 11.6)

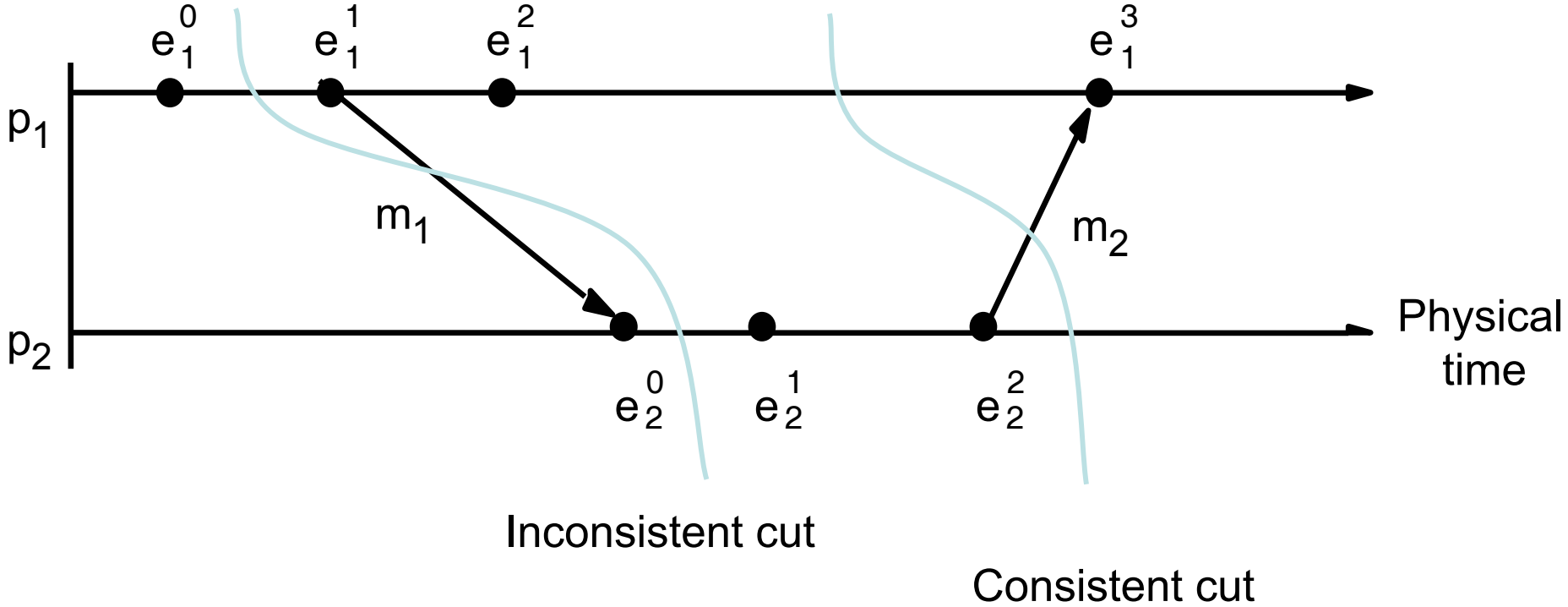
- All four of the distributed problems discussed in this section have particular solutions, but all of them also illustrate the need to observe global states. We will now look at a general approach to observing global states

11.5.1 Global states and consistent cuts

- Without global time identified by perfectly synchronized clocks, the ability to identify successive states in an individual process does not translate into the ability to identify successive states in distributed processes
- We can assemble meaningful global states from local states recorded at different local times in many circumstances, but must do so carefully and recognize limits to our capabilities
- A general system P of N processes p_i (i=1..N)
  - p_i's history: history(p_i)=h_i=<e_i^0, e_i^1, e_i^2, …>
  - finite prefix of p_i's history: h_i^k= <e_i^0, e_i^1, e_i^2, …, e_i^k>
  - state of p_i immediately before the kth event occurs: s_i^k
  - global history H=h_1 U h_2 U…U h_N
  - A cut of the system's execution is a subset of its global history that is a union of prefix of process histories C=h_1^{c1} U h_2^{c2} U…U h_N^{cN}

Figure 11.9 Cuts

- Figure 11.9 gives an example of an inconsistent cut_{ic} and a consistent cut_{cc}. The distinguishing characteristic is that
  - cut_{ic} includes the receipt of message m_1 but *not* the sending of it, while
  - cut_{cc} includes the sending *and* receiving of m_1 *and* cuts between the sending and receipt of the message m_2.
- A consistent cut cannot violate temporal causality by implying that a result occurred before its cause, as in message m_1 being received before the cut and being sent after the cut.



11.5.2 Global state predicates

- A Global State Predicate is a function that maps from the set of global process states to True or False.
- Detecting a condition like deadlock or termination requires evaluating a Global State Predicate.
- A Global State Predicate is stable: once a system enters a state where it is true, such as deadlock or termination, it remains true in all future states reachable from that state. However, when we monitor or debug an application, we are interested in non stable predicates.

11.5.3 The Snapshot Algorithm

- Chandy and Lamport defined a snapshot algorithm to determine global states of distributed systems
- The goal of a snapshot is to record a set of process and channel states (a snapshot) for a set of processes so that, even if the combination of recorded states may not have occurred at the same time, the recorded global state is consistent
  - The algorithm records states locally; it does not gather global states at one site.
- The snapshot algorithm has some assumptions
  - Neither channels nor processes fail
  - Reliable communications ensure every message sent is received exactly once
  - Channels are unidirectional
  - Messages are received in FIFO order
  - There is a path between any two processes
  - Any process may initiate a global snapshot at any time
  - Processes may continue to function normally during a snapshot

Snapshot Algorithm

- For each process, incoming channels are those which other processes can use to send it messages. Outgoing channels are those it uses to send messages. Each process records its state and for each incoming channel a set of messages sent to it. The process records for each channel, any messages sent after it recorded its state and before the sender recorded its own state. This approach can differentiate between states in terms of messages transmitted but not yet received
- The algorithm uses special marker messages, separate from other messages, which prompt the receiver to save its own state if it has not done so and which can be used to determine which messages to include in the channel state.
- The algorithm is determined by two rules

Figure 11.10 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 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:

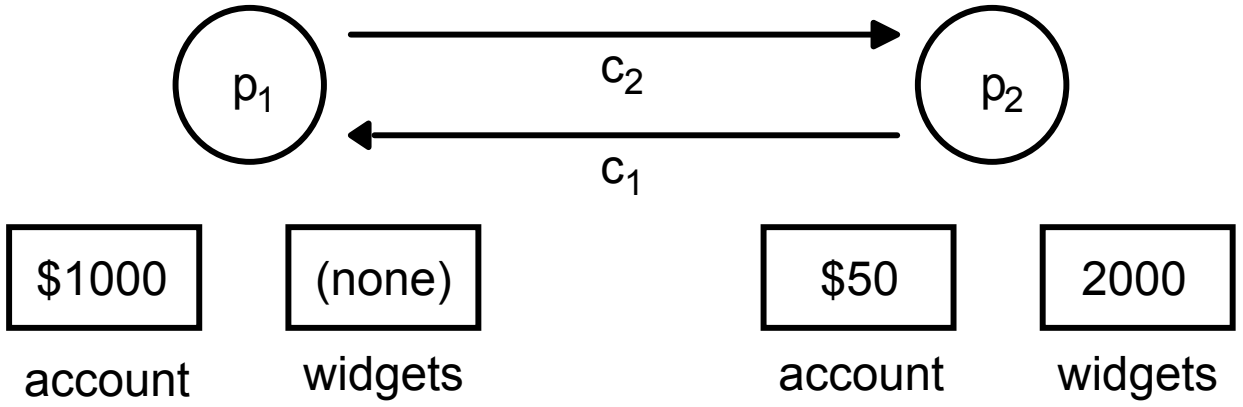
p_i sends one marker message over c

(before it sends any other message over c).

Example

- Figure 11.11 shows an initial state for two processes.
- Figure 11.12 shows four successive states reached and identified after state transitions by the two processes.
- Termination: it is assumed that all processes will have recorded their states and channel states a finite time after some process initially records its state.

Figure 11.11 Two processes and their initial states



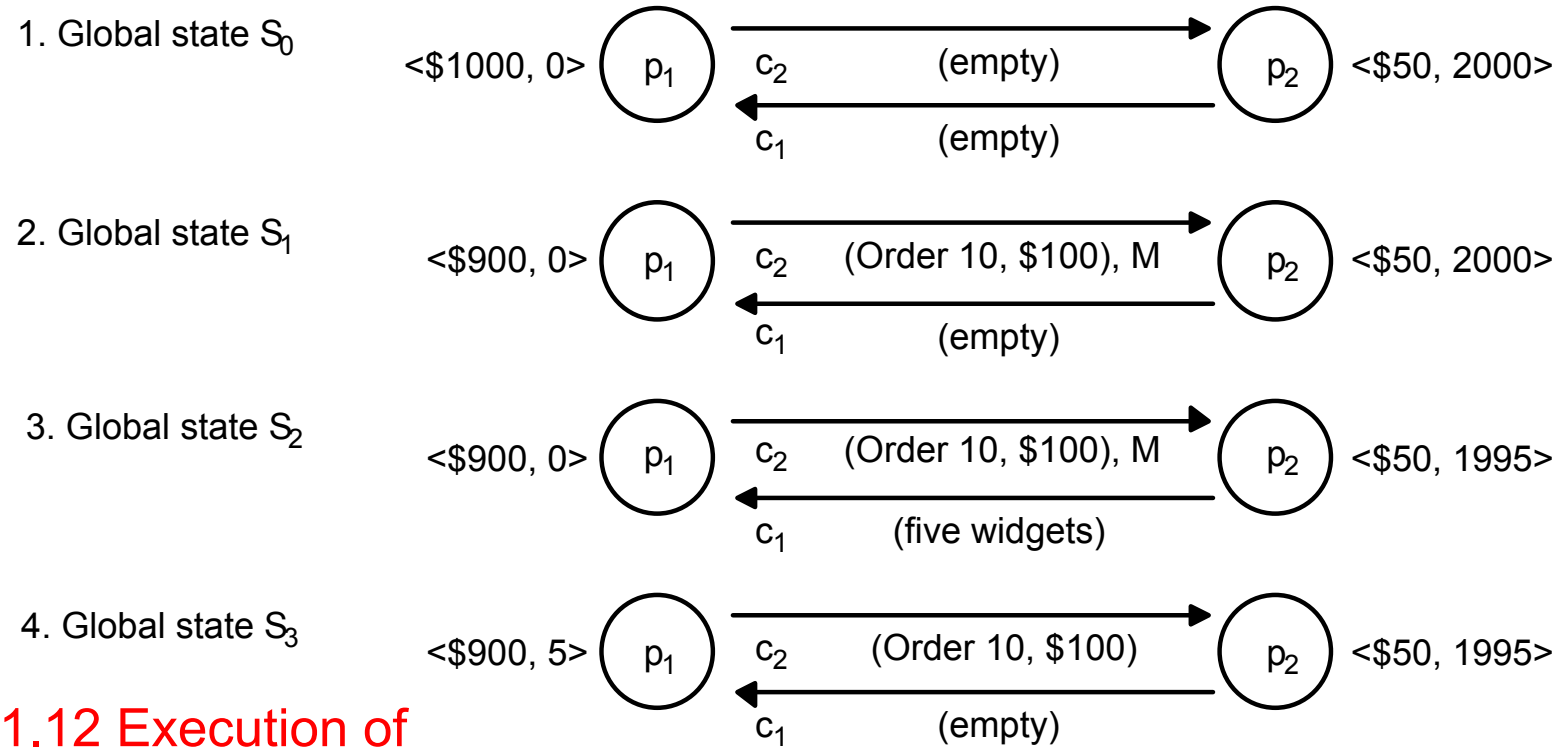


Figure 11.12 Execution of processes in Figure 11.11

(M = marker message)

Characterizing a state

- A snapshot selects a consistent cut from the history of the execution. Therefore the state recorded is consistent. This can be used in an ordering to include or exclude states that have or have not recorded their state before the cut. This allows us to distinguish events as pre-snap or post-snap events.
- The reachability of a state (figure 11.13) can be used to determine stable predicates.

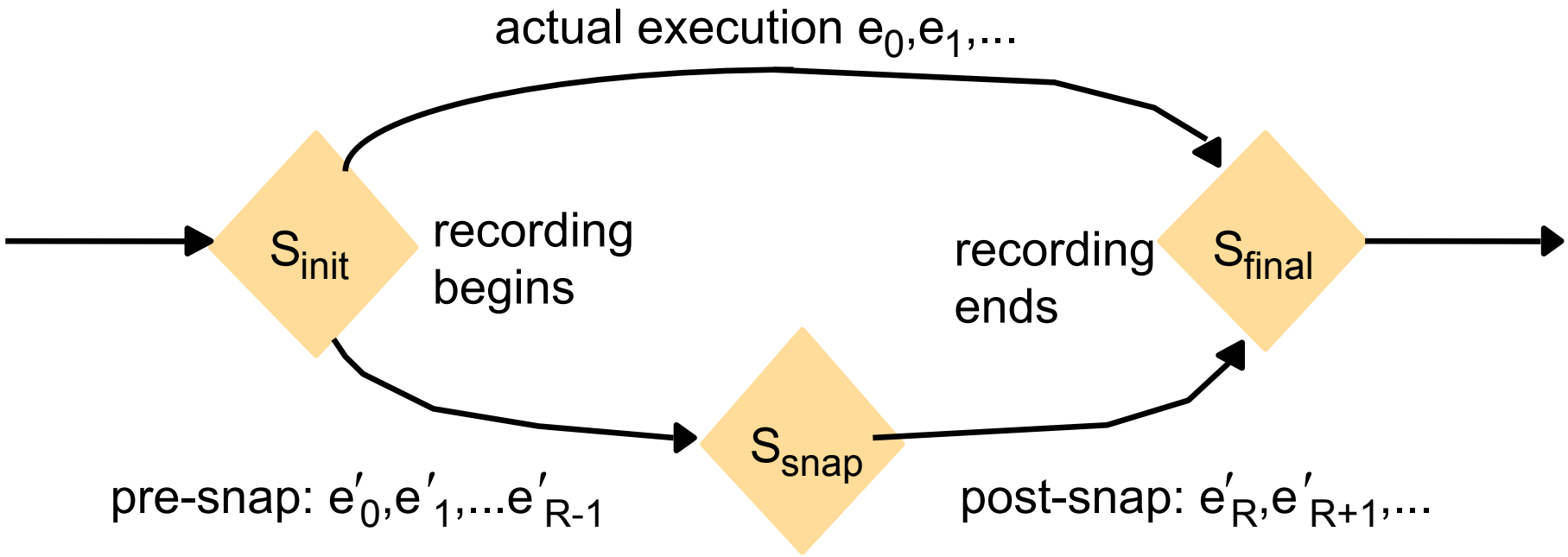


Figure 11.13 Reachability between states in the snapshot algorithm