Cloud computing – Distributed Systems

Heonchang Yu

Distributed and Cloud Computing Lab.

Clocks, events and process states

– Distributed system
  - 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
    - ✓ a message Send or Receive operation
    - ✓ an operation that transforms process' state
  - Event - the occurrence of a single action that a process carries out as it executes <Ex.> Send, Receive, state-transforming action
  - The sequence of events within a single process p_i can be placed in a single, total ordering, denoted by the relation $\rightarrow_i$ between the events. i.e. $e \rightarrow_i e'$ if and only if the event e occurs before e' at p_i
  - A history of process p_i: is a series of events ordered by $\rightarrow_i$
    $history(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$

Global states

- Distributed garbage collection
  - To be garbage if there are no longer any references to it anywhere in the distributed system
  - To check that an object is garbage, we must verify that there are no references to it anywhere in the system
  - Need to check a reference to it in a message that is in transit between the processes
- Distributed deadlock detection
  - There is a cycle in the graph of this 'waits-for' relationship.
- Distributed termination detection
  - To test whether each process has halted

Global states

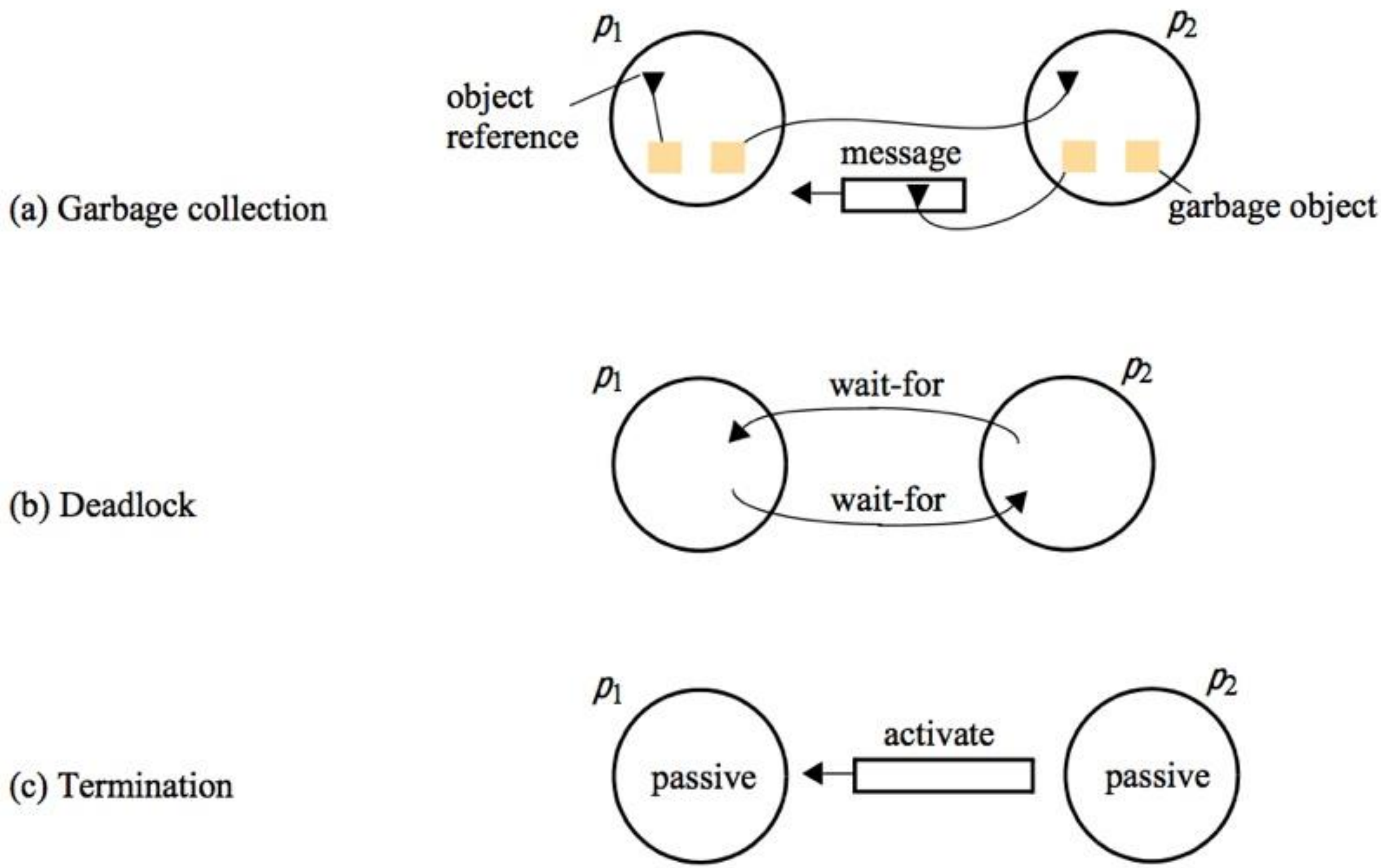


Figure 14.8 Detecting global properties

Global states

– Global states and consistent cuts
  - Problem: absence of global time
  - Event : an internal action of the process or the sending or receipt of a message over the communication channels
  - Cut - A subset of its global history
    - ✓ Take any set of states of the individual processes to form a global state S=(s_1, s_2, … , s_N)
    - ✓ At p_2 it includes the receipt of the message m_1, but at p_1 it does not include the sending of that message – showing an 'effect' without a 'cause'
  - Consistent cut - if, for each event it contains, it also contains all the events that happened-before that event:

    for all events $e \in C$, $f \rightarrow e \Rightarrow f \in C$
  - Consistent global state is one that corresponds to a consistent cut.

Global states

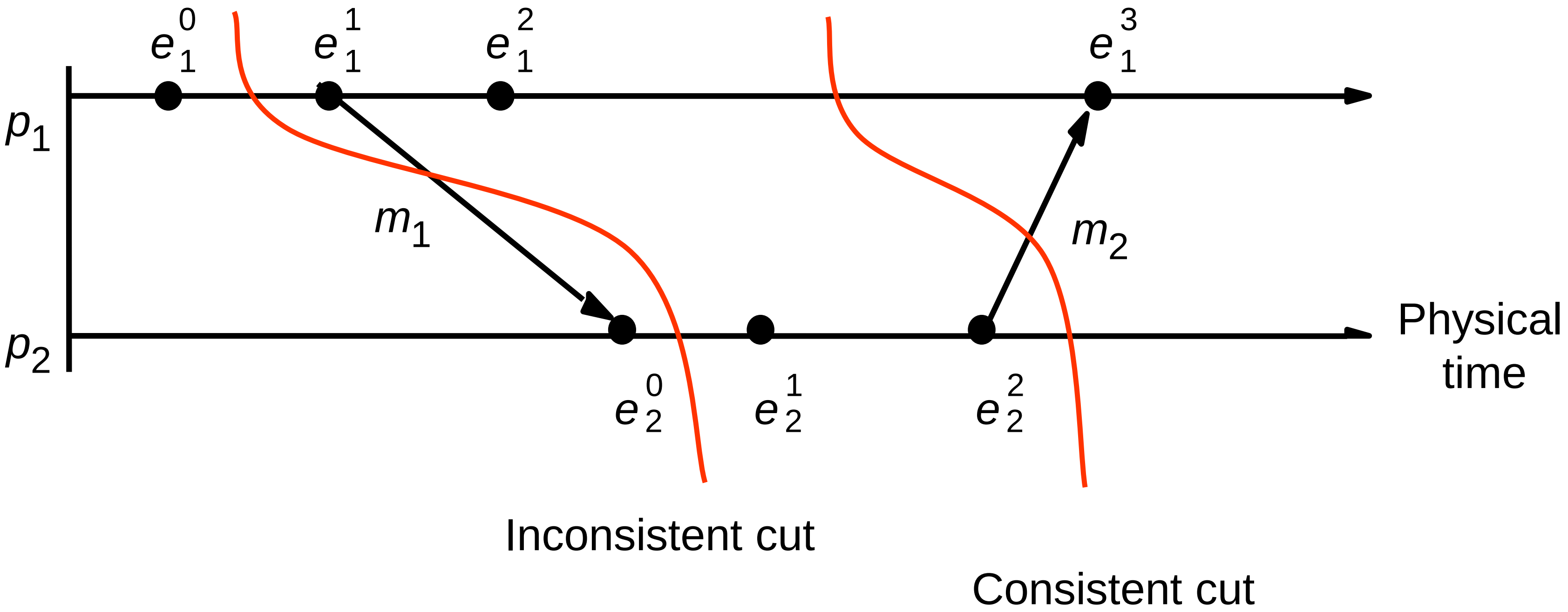


Figure 14.9 Cuts

Global states

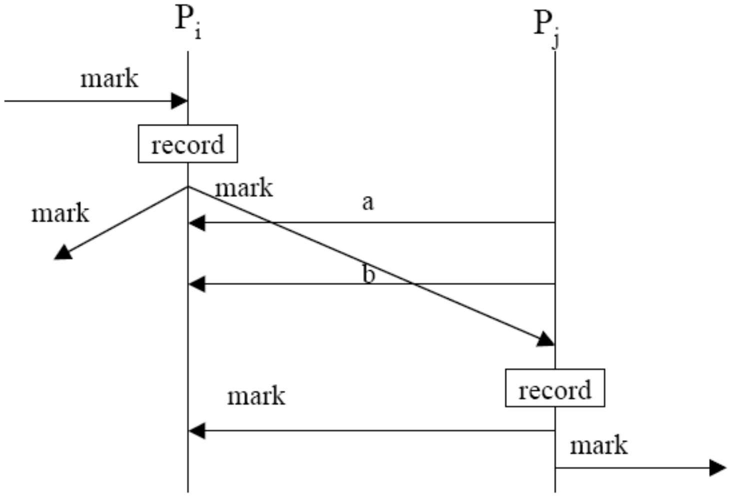
- Global state predicates, stability, safety and liveness
  - Stable: once the system enters a state in which the predicates associated is *True*, it remains *True* in all future states reachable from that state.
  - Safety: the assertion that α (being deadlocked) evaluates to *False* for all states S reachable from S_0
  - Liveness: the property that, for any linearization L starting in the state S_0, β evaluates to *True* for some state S_L reachable from S_0

Global states

– The 'snapshot' algorithm of Chandy and Lamport
  - Determining global states of distributed systems
  - Assumptions
    - ✓ Neither channels nor processes fail; communication is reliable so that every message sent is eventually received intact, exactly once;
    - ✓ Channels are unidirectional and provide FIFO-ordered message delivery;
    - ✓ The graph of processes and channels is strongly connected;
    - ✓ Any process may initiate a global snapshot at any time;
    - ✓ The processes may continue their execution and send and receive normal messages while the snapshot takes place.
  - System model
    - ✓ Incoming and Outgoing channels
    - ✓ Special marker messages
      - ❖ A prompt for a receiver to save its own state, if it has not already done so
      - ❖ A means of determining which messages to include in the channel state

Global states

– The ‘snapshot’ algorithm of Chandy and Lamport
  - Two rules of algorithm
    - ✓ Marker receiving rule
      - ❖ Obligates a process that has not recorded its state to do so
      - ❖ When a process that has already saved its state receives a marker, it records the state of that channel as the set of messages it received on it since it saved on it.
    - ✓ Marker sending rule
      - ❖ Obligates processes to send a marker after they have recorded their state, but before they send any other messages
    - ✓ Example



Global states

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).

Figure 14.10 Chandy and Lamport's 'snapshot' algorithm

Global states

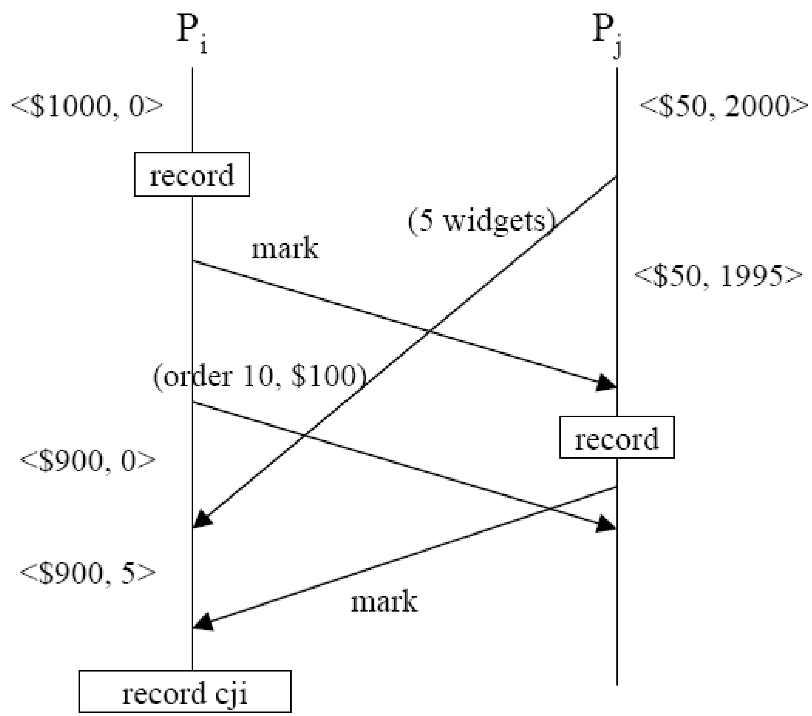
- The 'snapshot' algorithm of Chandy and Lamport
  - Example: a system of two processes, p_1 and p_2 connected by two unidirectional channels, c_1 and c_2
    - ✓ Two processes trade in 'widgets'.
    - ✓ Process p_1 sends orders for widgets over c_2 to p_2, enclosing payment at the rate of \$10 per widget.
    - ✓ Process p_2 sends widgets along channel c_1 to p_1.
    - ✓ Process p_2 has already received an order for five widgets
      - ❖ S_0 : Process p_1 records its state when p_1's state is <\$1000, 0>.
      - ❖ S_1 : Following the marker sending rule, process p_1 emits a marker message over its outgoing channel c_2 before it sends the next application-level message: (order 10, \$100) over channel c_2.
      - ❖ S_2 : Before p_2 receives the marker, it emits an application message (five widgets) over c_1 in response to p_1's previous order. Process p_1 receives p_2's message (five widgets), and p_2 receives the marker. Following the marker receiving rule, p_2 records its state as <\$50, 1995> and that of channel c_2 as the empty sequence. Following the marker sending rule, it sends a marker message over c_1.

Global states

– The ‘snapshot’ algorithm of Chandy and Lamport

❖ S_3 : When process p_1 receives p_2's marker message, it records the state of channel c_1 as the single message (five widgets) that it received after it first recorded its state.

✓ p_1 :<\$1000, 0>; p_2 :<\$50, 1995>; c_1 :<(five widgets)>; c_2 :<>



Logical time and logical clocks

– As Lamport pointed out, since we cannot synchronize clocks perfectly across a distributed system, we cannot use physical time to find out the order of any arbitrary pair of events.

- For ordering
  - ✓ If two events occurred at the same process p_i (i=1,...,N), then they occurred in the order in which p_i observes them.
  - ✓ Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message.
- Happened-before relation (→)
  - ✓ HB1: If ∃ process p_i : $e \rightarrow_i e'$, then $e \rightarrow e'$.
  - ✓ HB2: For any message m, $send(m) \rightarrow receive(m)$
    - where $send(m)$ is the event of sending the message, and $receive(m)$ is the event of receiving it.
  - ✓ HB3: If e, e' and e'' are events such that $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

Logical time and logical clocks

– In figure 14.5, not all events are related by the relation →.
  - a $\nrightarrow$ e and e $\nrightarrow$ a
    - ✓ Since they occur at different processes, and there is no chain of messages intervening between them.
    - ✓ Concurrent : a ॥ e

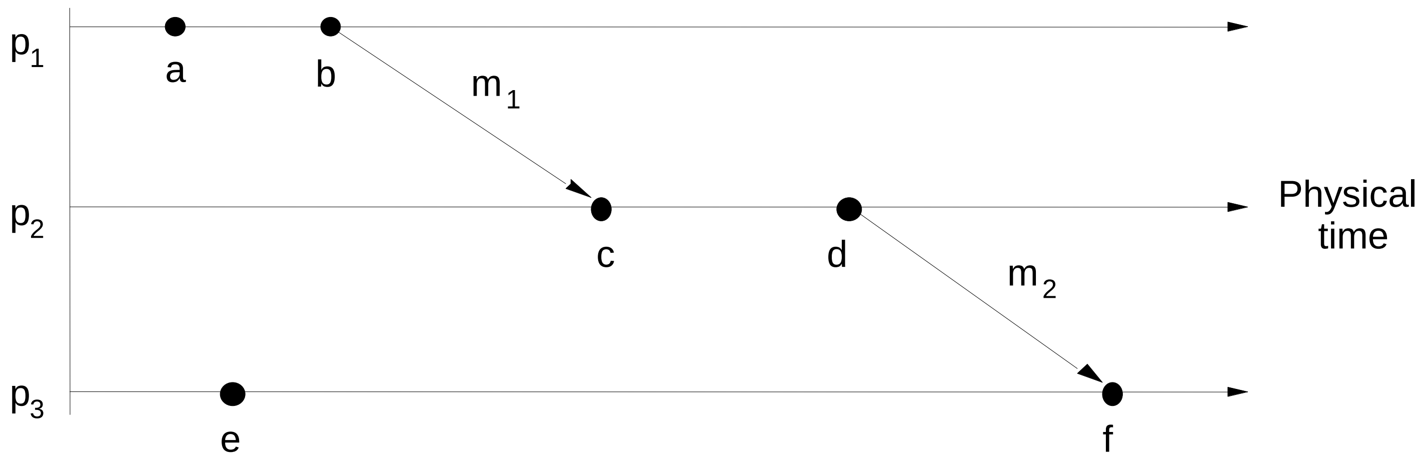


Figure 14.5 Events occurring at three processes

Logical time and logical clocks

– Logical clocks: a monotonically increasing software counter
  - LC1: L_i is incremented before each event is issued at process p_i :
    $L_i := L_i + 1$
  - LC2: (a) When a process p_i sends a message m, it piggybacks on m the value $t = L_i$.
    (b) On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$ and then applies LC1 before timestamping the event $receive(m)$.

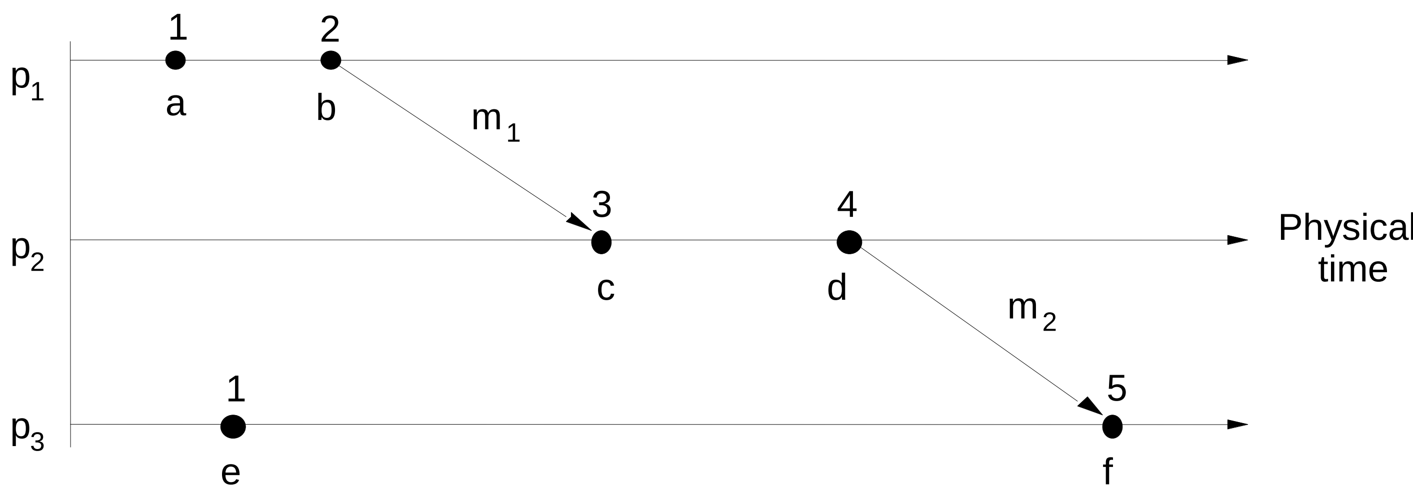


Figure 14.6 Lamport timestamps for the events shown in Figure 14.5

Logical time and logical clocks

– Logical clocks
  - For two events e and e', $e \rightarrow e' \Rightarrow L(e) < L(e')$.
  - The converse is not true: $L(e) < L(e') \not\Rightarrow e \rightarrow e'$.
    - ✓ <ex> $L(b) > L(e)$ but $b \parallel e$.

– Totally ordered logical clocks
  - If e is an event occurring at p_i with local timestamp T_i, and e' is an event occurring at p_j with local timestamp T_j, we define the global logical timestamps for these events to be (T_i, i) and (T_j, j).
    - ✓ $(T_i, i) < (T_j, j)$: $T_i < T_j$, or $T_i = T_j$ and $i < j$

– Vector clocks
  - To overcome the shortcoming of Lamport's clocks: the fact that from $L(e) < L(e')$ conclude that $e \rightarrow e'$
  - A vector clock for a system of N processes is an array of N integers.
  - Processes piggyback vector timestamps on the messages they send to one another.

Logical time and logical clocks

- Rules of vector clocks
  - VC1: Initially, $V_i[j] = 0$, for $i, j = 1, 2, \ldots, N$
  - VC2: Just before p_i timestamps an event, it sets $V_i[i] := V_i[i] + 1$.
  - VC3: p_i includes the value $t = V_i$ in every message it sends.
  - VC4: When p_i receives a timestamp t in a message, it sets $V_i[j] := \max(V_i[j], t[j])$, for $j = 1, 2, \ldots, N$. Taking the component-wise maximum of two vector timestamps in this way is known as a merge operation.

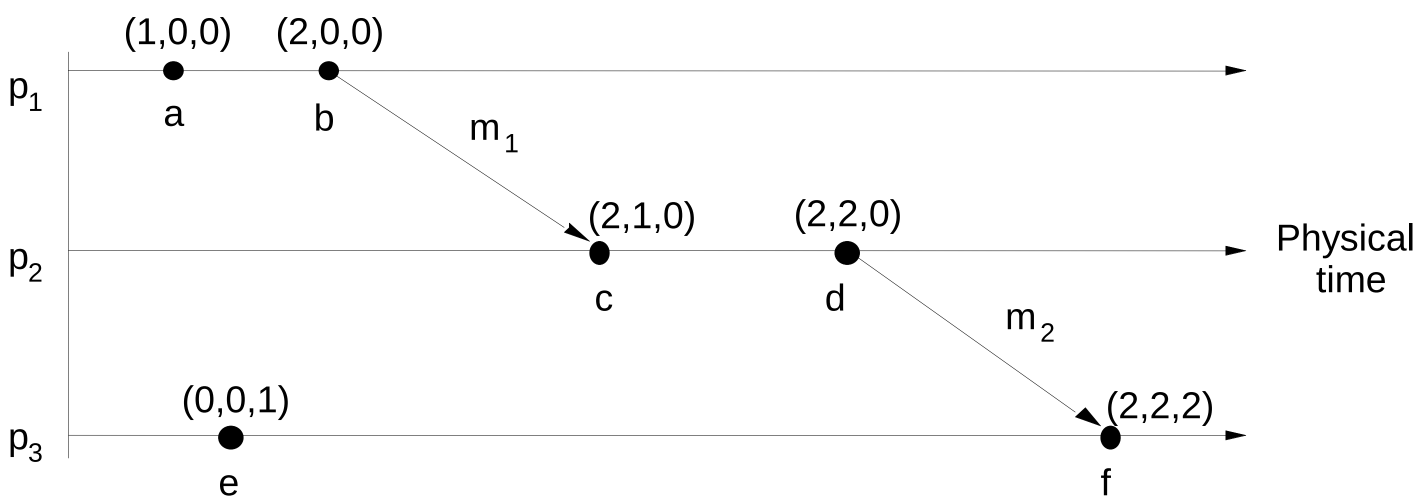


Figure 14.7 Vector timestamps for the events shown in Figure 14.5