Time

Hardware and Software Clocks

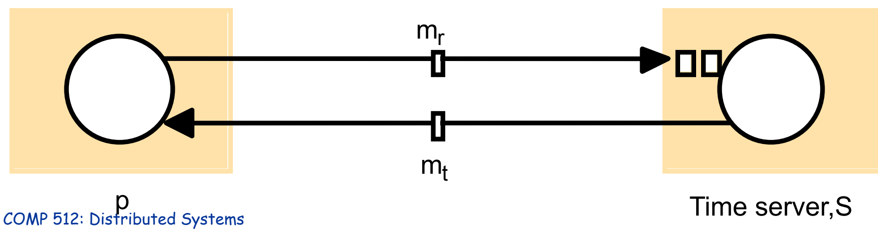
- At real time, t, the OS reads the time on the computer's hardware clock $H_i(t)$
- It calculates the time on its software clock
  $C_i(t) = \alpha H_i(t) + \beta$
  - ☆ E.g. a 64 bit number giving nanoseconds since some base time
- Clock resolution
  - ☆ How accurate does it need to be?
- Clock Drift: 1 sec in 11 days
- Monotonicity
  - ☆ $t' > t \Rightarrow C(t') > C(t)$
  - ☆ can achieve monotonicity with a hardware clock that runs fast by adjusting the values of α and β in $C_i(t) = \alpha H_i(t) + \beta$

Synchronization

- Internal synchronization within bound D
  - ☆ given a set of computer clocks $C_j(t)$, $i = 1, 2, \ldots N$
  - ☆ $| C_i(t) - C_j(t)| < D$ for $i = 1, 2, \ldots N$
- Coordinated Universal Time (UTC)
  - ☆ based on very accurate physical clocks
  - ☆ Computer need receivers;
    - 0.1-10 milliseconds accurate for land-based reception
    - 1 microsecond through GPS
- External synchronization within bound D
  - ☆ Given a source S of UTC time and computer clock C
  - ☆ $|S(t) - C(t)| < D$

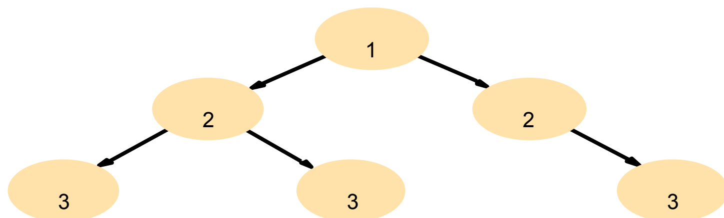
Synchronizing clocks in an asynchronous system

- Ideas:
  - ☆ use a time server S that receives signals from a UTC source
- Synchronizing process p with time server
  - ☆ Process p requests time in m_r and receives t in m_t from S
  - ☆ p sets its clock to
    - $t + T_{round}/2$
- Build average over several requests



Network Time ProtocolNTP

- Primary Servers are connected to UTC sources
- Secondary servers are synchronized to primary servers
- lowest level servers are user's computers
- accuracy of
  - 10s of milliseconds for Internet
  - 200 microseconds in Intranet

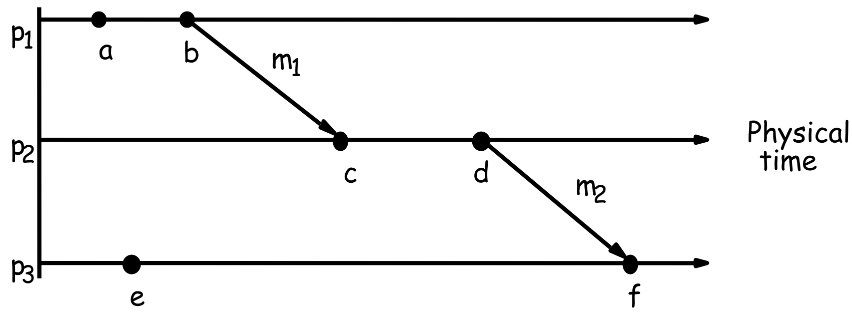


Events and Logical time

- A distributed system consists of a collection of N processes pi, i=1, 2, 3, …, N
- Each process executes on a single processor.
- Information exchange only occurs through message exchange
- Each process p_i has a state s_i. The state changes according to the operations the process performs. (the state i.e., is the content of all application variables).
- Each process executes a sequence of events/actions,
  - send, receive or change own state
- Ordering of events in a distributed system
  - The sequence of events within a single process can be placed in a total order:
    - $e \rightarrow_i e'$ if and only if event e occurs before e' at p_i.
    - Whenever a message is sent, the event of sending a message occurs before the event of receiving the message
- A history of process p_i: is a series of events ordered by $\rightarrow_i$

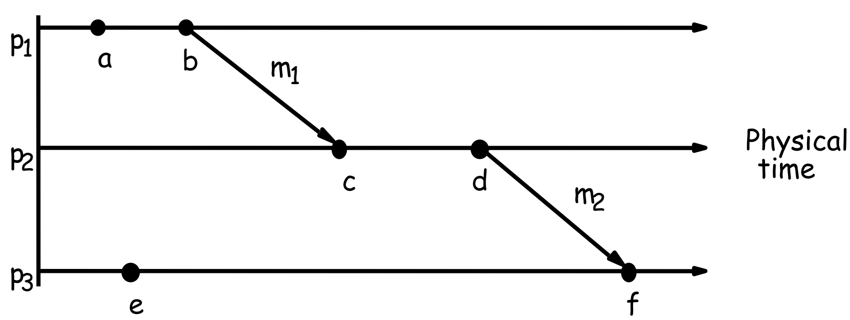
  history(p_i)= $h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$

Events in a distributed system



Logical time

- We define the happened-before relation, denoted by ->, as follows
  - ☆ If there is a process pi: e $->_i$ e', then e -> e
  - ☆ For any message m, send(m) -> receive(m) (where send(m) is the event of sending the message and receive(m) the receiving event)
  - ☆ If e, e', and e''' are events such that e -> e' and e' -> e'', then e -> e''
  - ☆ If neither e -> e' nor e' -> e, then e and e' are concurrent e || e'



Logical clocks (Lamport clocks)

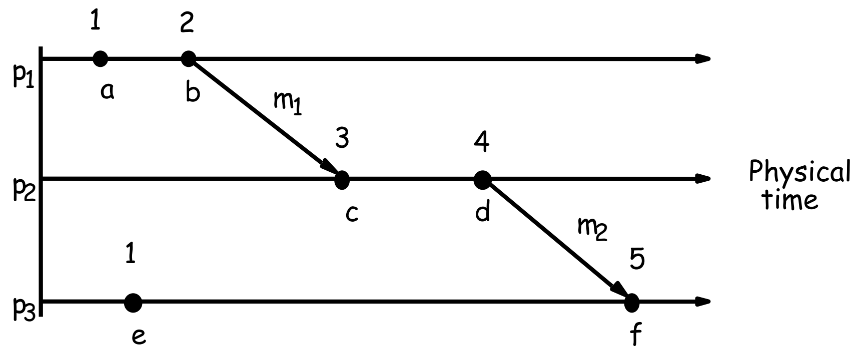
- Each process p_i maintains a logical clock L_i.
- L_i is a monotonically increasing counter (no relationship to real time)
- Each site timestamps events with the value of its logical clock.
- We denote the timestamp of event e at p_i by $L_i(e)$ (and omit the index if we speak about any event in the system)
- Goal: whenever e -> e', then L(e) < L(e'), i.e., when e happens before e', then e should have a smaller timestamp than e'.

Logical clocks (Lamport clocks)

- Algorithm at process pi when event e takes place:

```
Li := Li+1
If Local event e:
    Li(e) := Li
Else if Send(m) event e:
    Li(e) := Li;
     Piggyback Li(e) on m (m,Li(e))
Else if Receive(m,Lj(e')) event e:
    Li := max(Li, Lj(e')+1);
    Li(e) := Li
```

Events in a distributed system



- It is easy to see that
  - If e -> e' then L(e) < L(e')
- The opposite does not hold
  - If L(e) < L(e'), then this does not mean that e -> e', instead it could also be that e || e'

Vector Clocks

- Goal: find a timestamp V such that
  - e -> e' <=> V(e) < V(e')
  - In particular, this means, that if neither e -> e' nor e' -> e, then neither V(e) < V(e') nor V(e') < V(e) (the timestamps are incomparable)
- Each processor p_i has a vector clock Vi which is an array of N integers. (one for each processor)
- We compare vector timestamps as follows
  - V = V' iff V[j] = V'[j] for j = 1, 2, … N
  - V <= V' iff V[j] <= V'[j] fo j = 1, 2, …. N
  - V < V' iff V <= V' and not V = V'

Vector Clocks

- Algorithm at process p_i

```
Upon start:  Vi[j] := 0, for j = 1, 2, ….N
Upon event e:
   Vi[i] := Vi[i] +1;
   If e is local:
      Vi(e) := Vi
   Else If e is send request m:
      Vi(e) := Vi
      Send (m, Vi)
   Else If e is receive(m,t) event:
      Vi[j] := max (Vi[j], t[j]), for j = 1, 2, …. N.
        (componentwise maximum)
```


Vector timestamps