Distributed Systems

CS425/ECE428

01/31/2020

Today's agenda

- Clock synchronization
  - Chapter 14.1-14.3
- Logical clocks
  - Chapter 14.4

Recap from last class: Failures

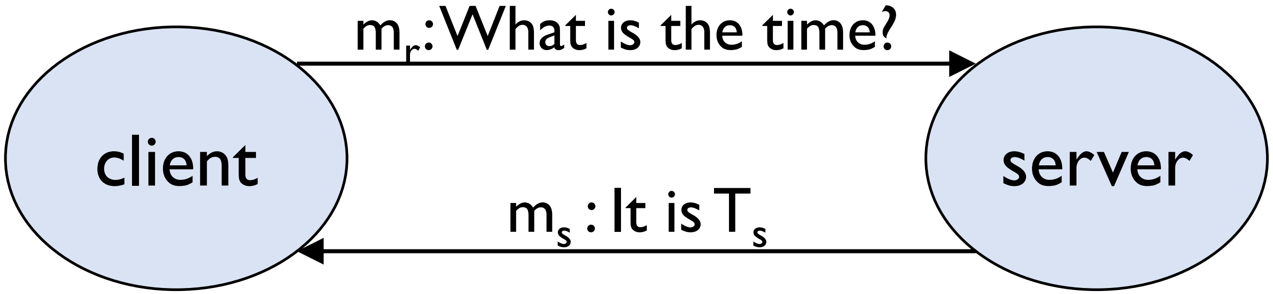
- Three types: *omission, arbitrary, timing*.

- Failure detection (detecting a crashed process):
  - Send periodic ping-acks or heartbeats.
  - Report crash if no response until a timeout.
  - Timeout can be precisely computed for synchronous systems and estimated for asynchronous.
  - Metrics: *completeness, accuracy, failure detection time, bandwidth.*
  - Failure detection for a system with multiple processes:
    - Centralized, ring, all-to-all
    - Trade-off between completeness and bandwidth usage.

Recap from last class: Clocks

- Useful to compare timestamps across processes (or know *accurate* time).
- Clocks in different computers show different times.
  - Clock skew: relative difference between two clock values.
- Clocks in different computers drift at different rates.
  - Clock drift rate: change in skew from a perfect reference clock per unit time (measured by the reference clock).
- Need for *synchronization*:
  - *External:* with an authoritative clock, for achieving accuracy
  - *Internal:* among the processes within a distributed system.

Synchronization of clocks



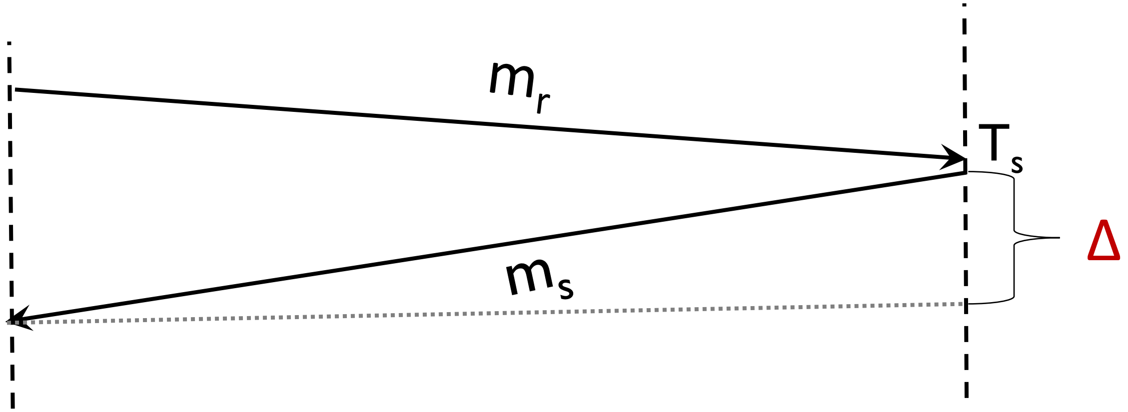
m_r: What is the time?

client

server

m_s : It is T_s

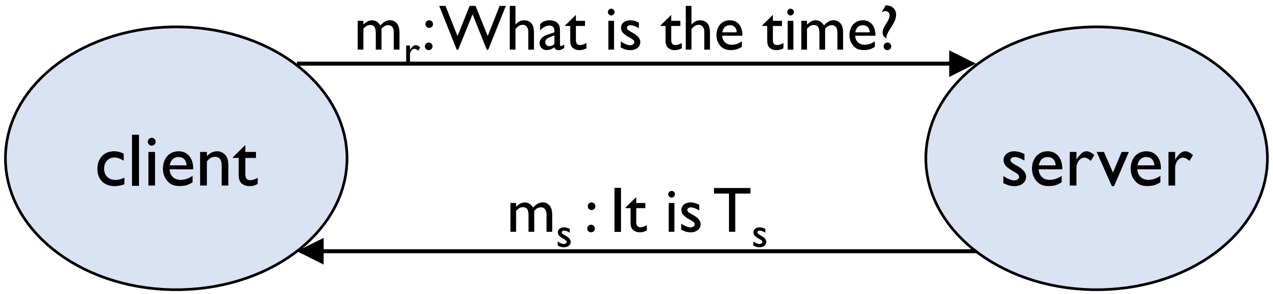
What time T_c should client adjust its local clock to after receiving m_s ?



$T_c = T_s + \Delta$

But the value of Δ is unknown.

Synchronization in synchronous systems



m_r: What is the time?

client

server

m_s : It is T_s

What time T_c should client adjust its local clock to after receiving m_s ?

Let *max* and *min* be maximum and minimum network delay.

If $T_c = T_s$, skew(client, server) $\leq$ *max.*

If T_c = **(T_s + max)**, skew(client, server) $\leq$ *(max – min)*

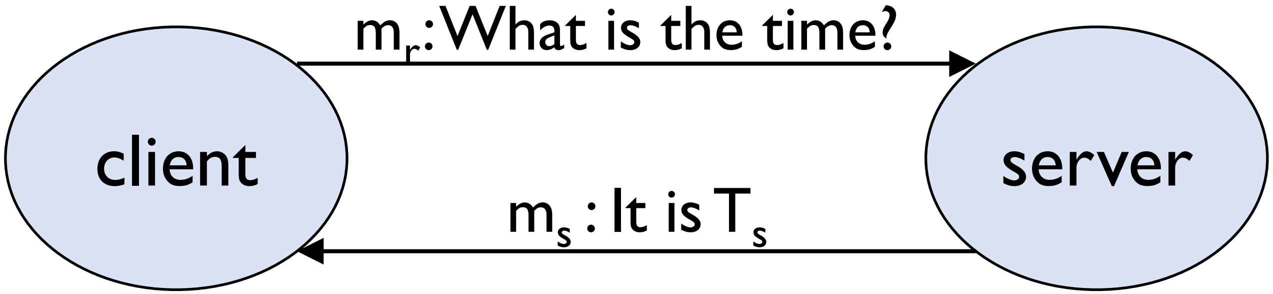
If T_c = **(T_s + min)**, skew(client, server) $\leq$ *(max – min)*

T_c = **(T_s + (min + max)/2)**, skew(client,server) $\leq$ *(max – min)/2*

Synchronization in asynchronous systems

- Cristian Algorithm
- Berkeley Algorithm
- Network Time Protocol

Cristian Algorithm



What time T_c should client adjust its local clock to after receiving m_s ?

Client measures the round trip time (**T_{round}**).

$T_c = T_s + (T_{round} / 2)$

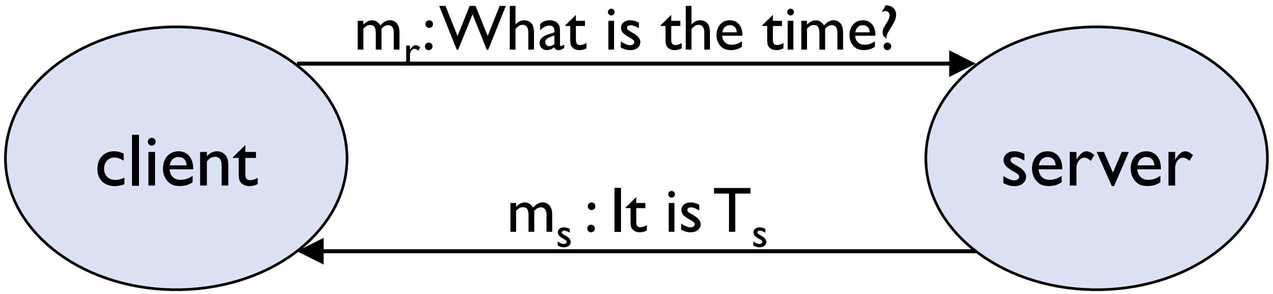
skew $\leq (T_{round} / 2) - min$

(*min* is minimum one way network delay).

Try deriving the worst case skew!

Hint: *client is assuming its one-way delay from server* (**Δ**) *is $T_{round}/2$. How off can it be?*

Cristian Algorithm



m_r: What is the time?

client

server

m_s : It is T_s

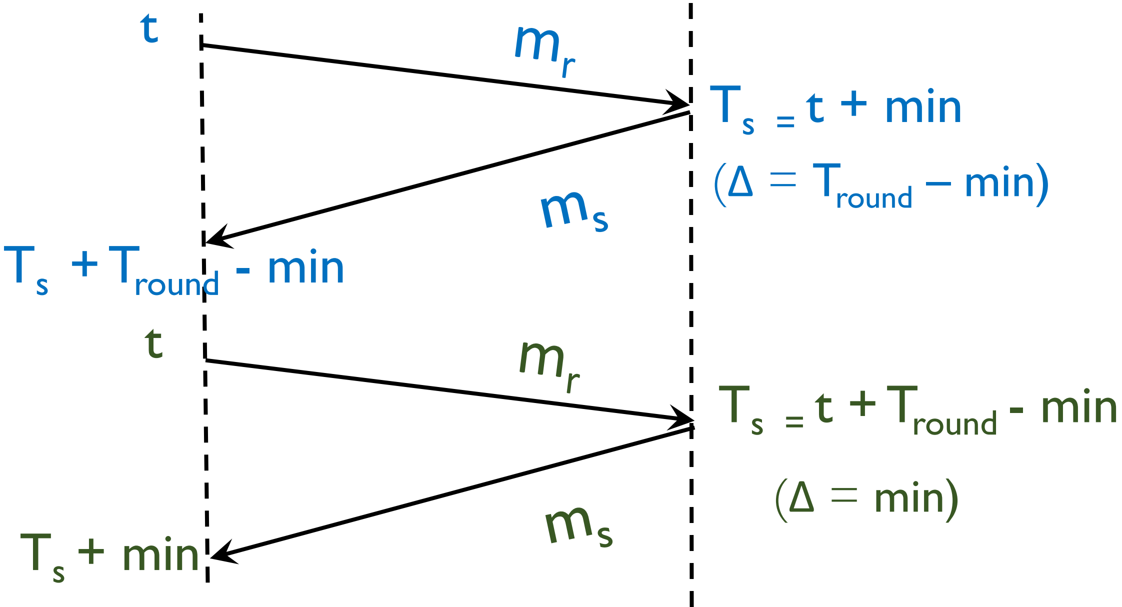
What time T_c should client adjust its local clock to after receiving m_s ?

Client measures the round trip time (T_{round}).

$T_c = T_s + (T_{round} / 2)$

skew $\leq (T_{round} / 2) - min$

(*min* is minimum one way network delay).



t

m_r

$T_s = t + min$

$(\Delta = T_{round} - min)$

m_s

$T_s + T_{round} - min$

t

m_r

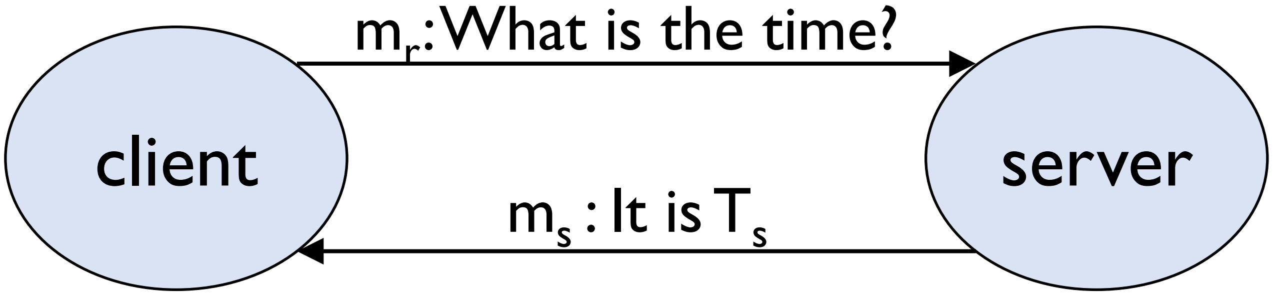
$T_s = t + T_{round} - min$

$(\Delta = min)$

m_s

$T_s + min$

Cristian Algorithm



m_r: What is the time?

client

server

m_s : It is T_s

What time T_c should client adjust its local clock to after receiving m_s ?

Client measures the round trip time (**T_{round}**).

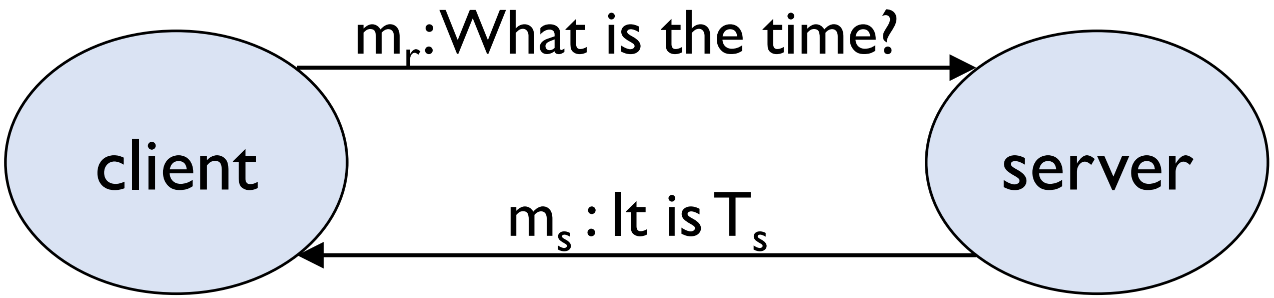
$T_c = T_s + (T_{round} / 2)$

skew ≤ $(T_{round} / 2) - min$
(*min* is minimum one way network delay).

Improve accuracy by sending multiple spaced requests and using response with smallest T_{round}.

Server failure: Use multiple synchronized time servers.

Cristian Algorithm



What time T_c should client adjust its local clock to after receiving m_s ?

Client measures the round trip time (T_{round}).

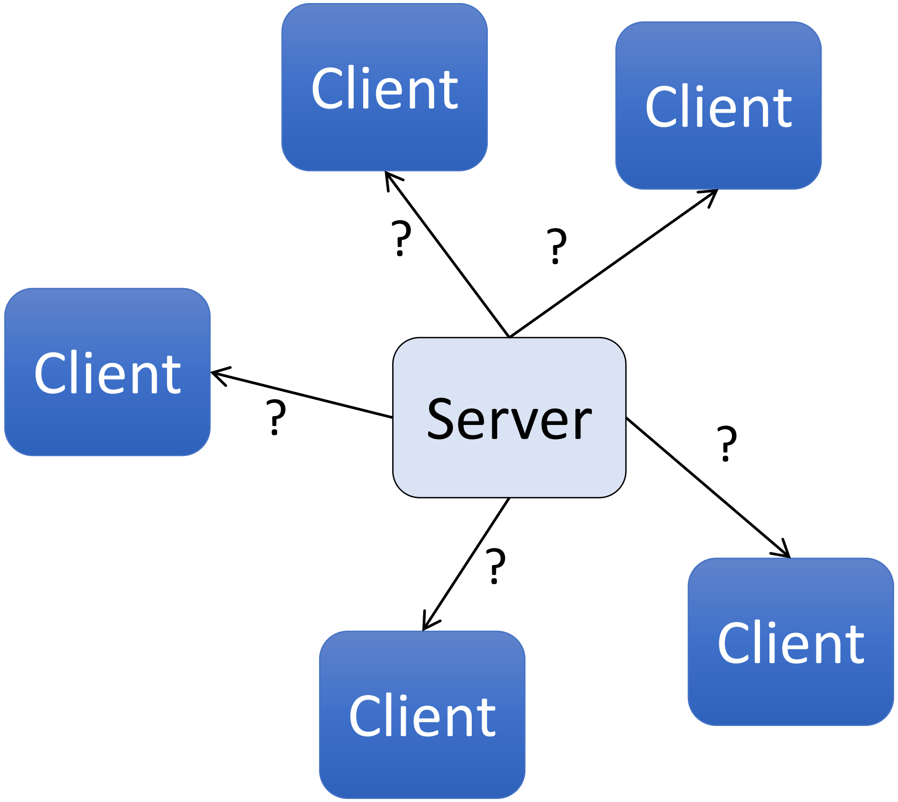
$T_c = T_s + (T_{round} / 2)$

skew $\leq (T_{round} / 2) - min$
(*min* is minimum one way network delay).

Cannot handle faulty time servers.

Berkeley Algorithm

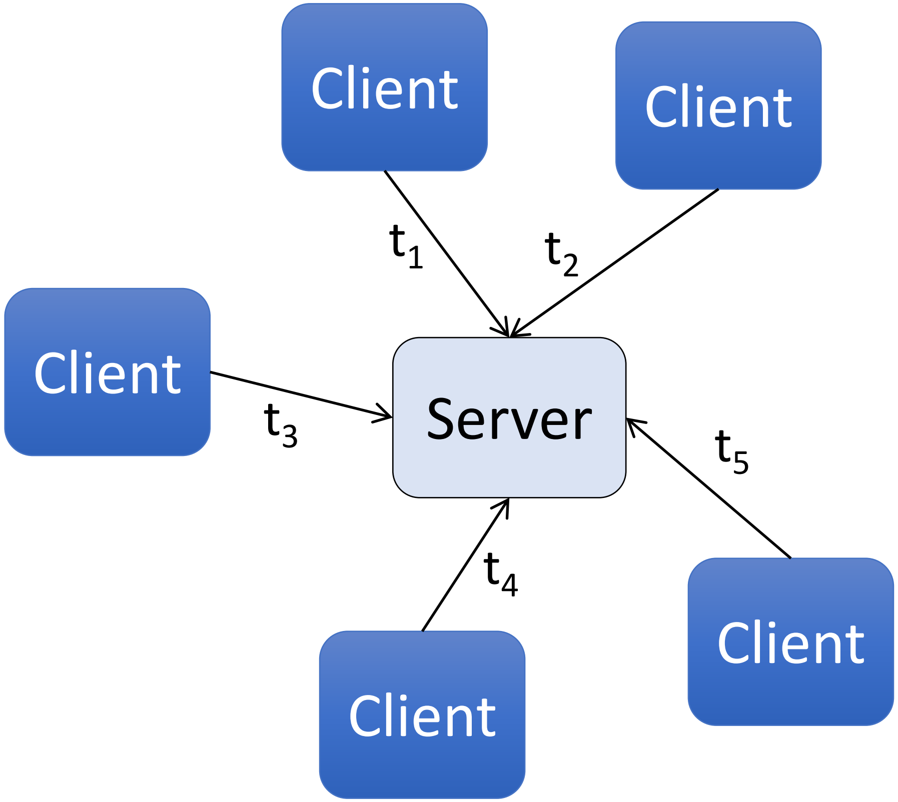
Only supports internal synchronization.



1. Server periodically polls clients:
   "what time do you think it is?"

Berkeley Algorithm

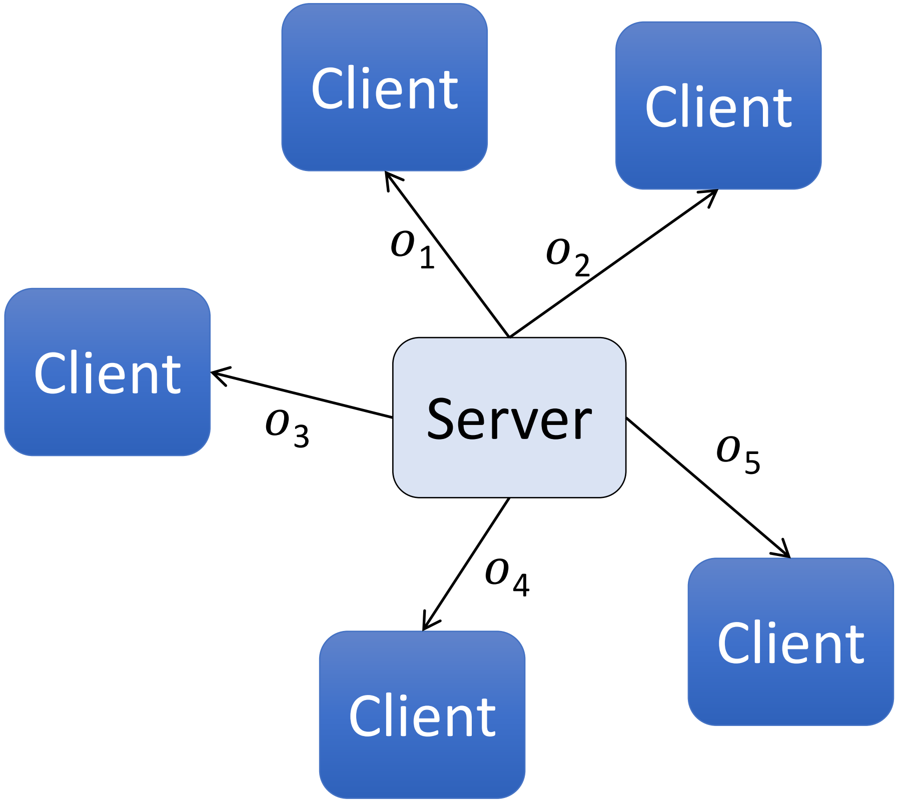
Only supports internal synchronization.



1. Server periodically polls clients: *"what time do you think it is?"*
2. Each client responds with its local time.
3. Server uses Cristian algorithm to estimate local time at each client.
4. Average all local times (including its own) – use as updated time.

Berkeley Algorithm

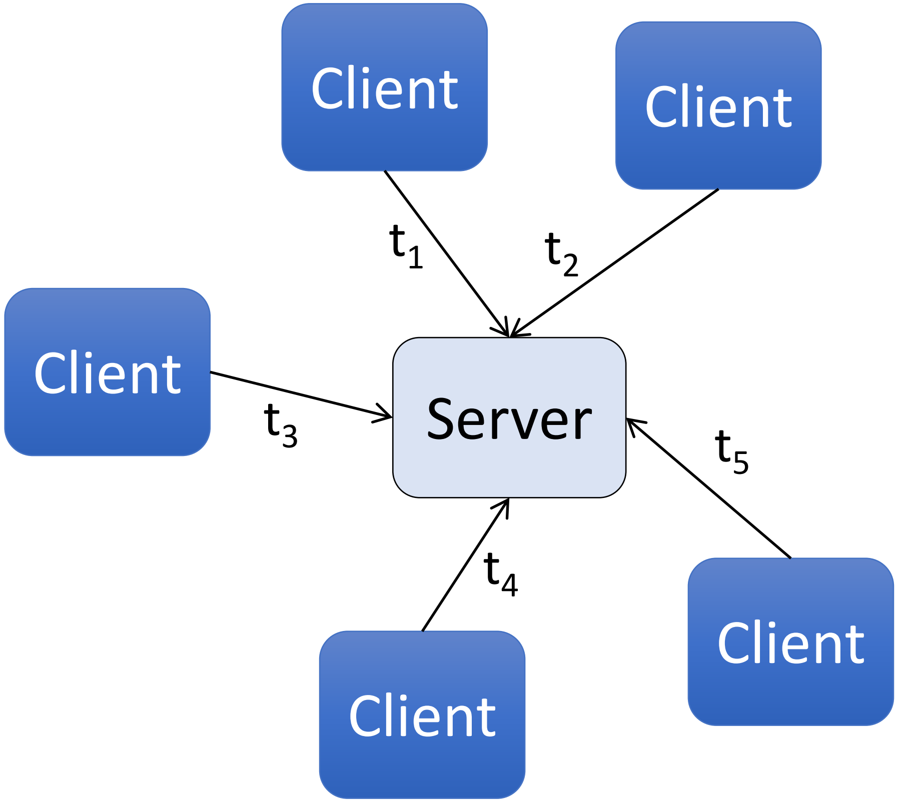
Only supports internal synchronization.



1. Server periodically polls clients: *"what time do you think it is?"*
2. Each client responds with its local time.
3. Server uses Cristian algorithm to estimate local time at each client.
4. Average all local times (including its own) – use as updated time.
5. Send the offset (amount by which each clock needs adjustment).

Berkeley Algorithm

Only supports internal synchronization.

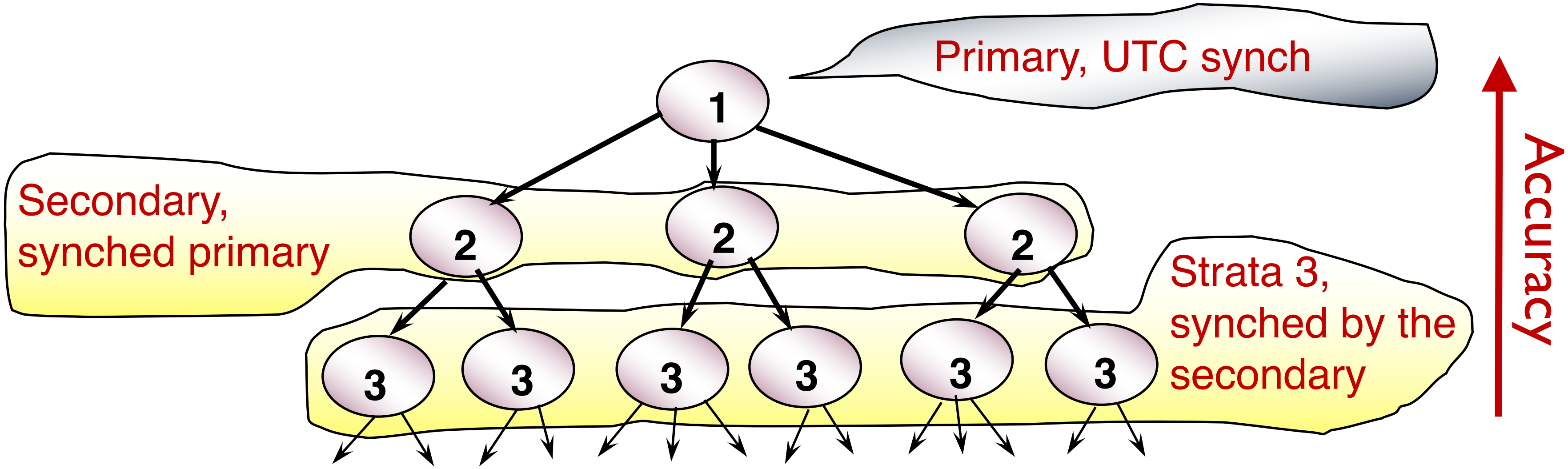


Handling faulty processes:
Only use timestamps within some threshold of each other.

Handling server failure:
Detect the failure and elect a new leader.

Network Time Protocol

Time service over the Internet for synchronizing to UTC.



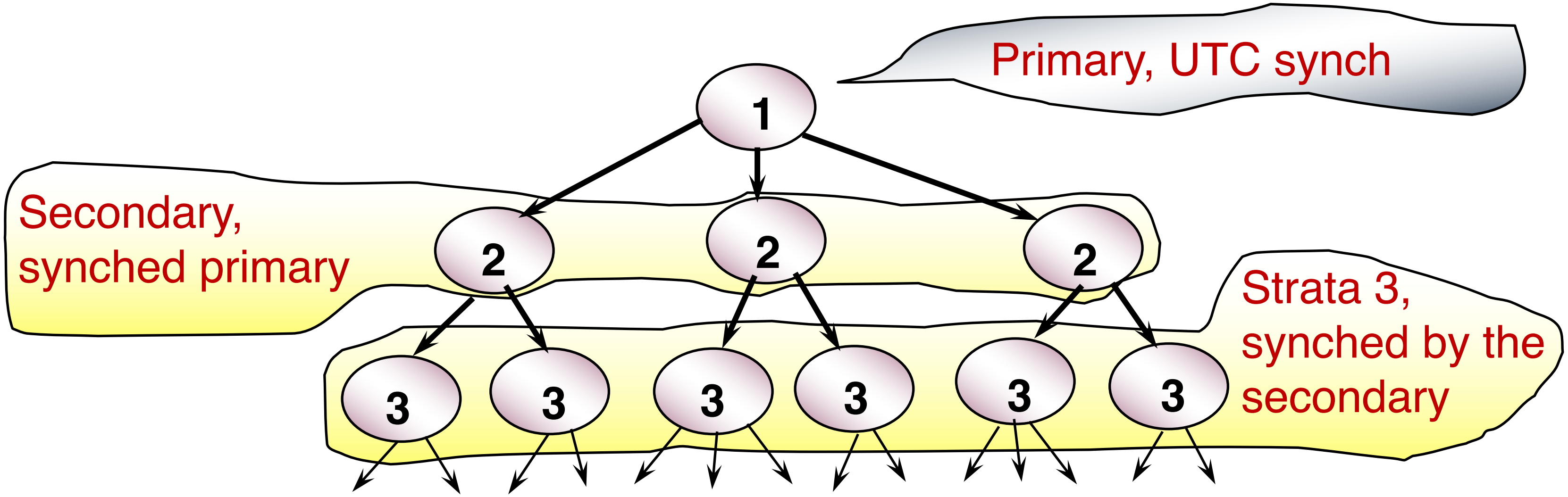
Hierarchical structure for *scalability*.

Multiple lower strata servers for *robustness*.

Authentication mechanisms for *security*.

Statistical techniques for better *accuracy*.

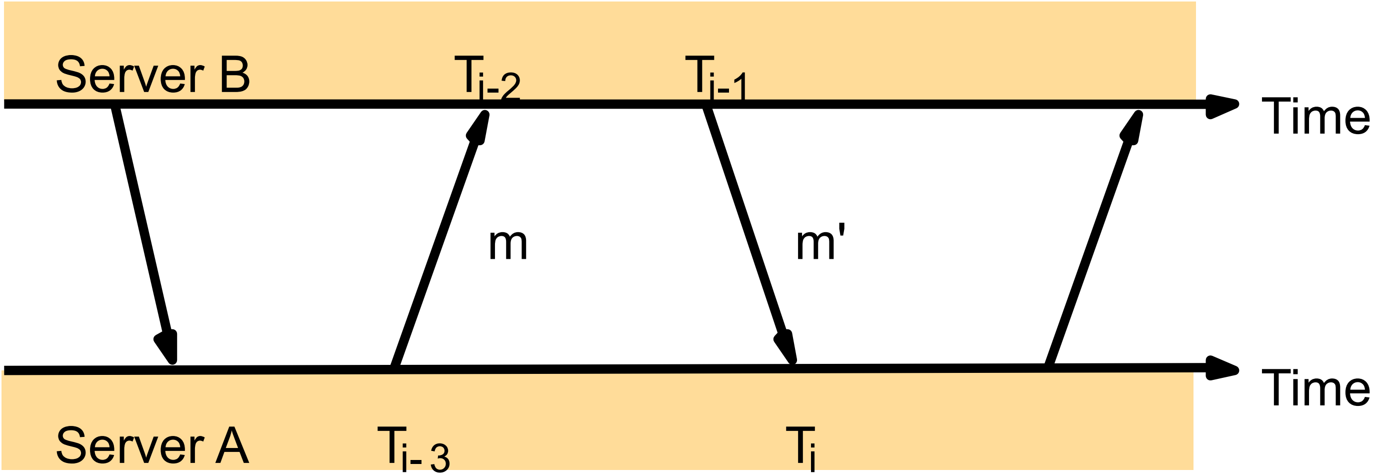
Network Time Protocol



How clocks get synchronized:

- Servers may *multicast* timestamps within a LAN. Clients adjust time assuming a small delay. *Low accuracy*.
- *Procedure-call* (Cristian algorithm). *Higher accuracy.*
- *Symmetric mode* used to synchronize lower strata servers. *Highest accuracy*.

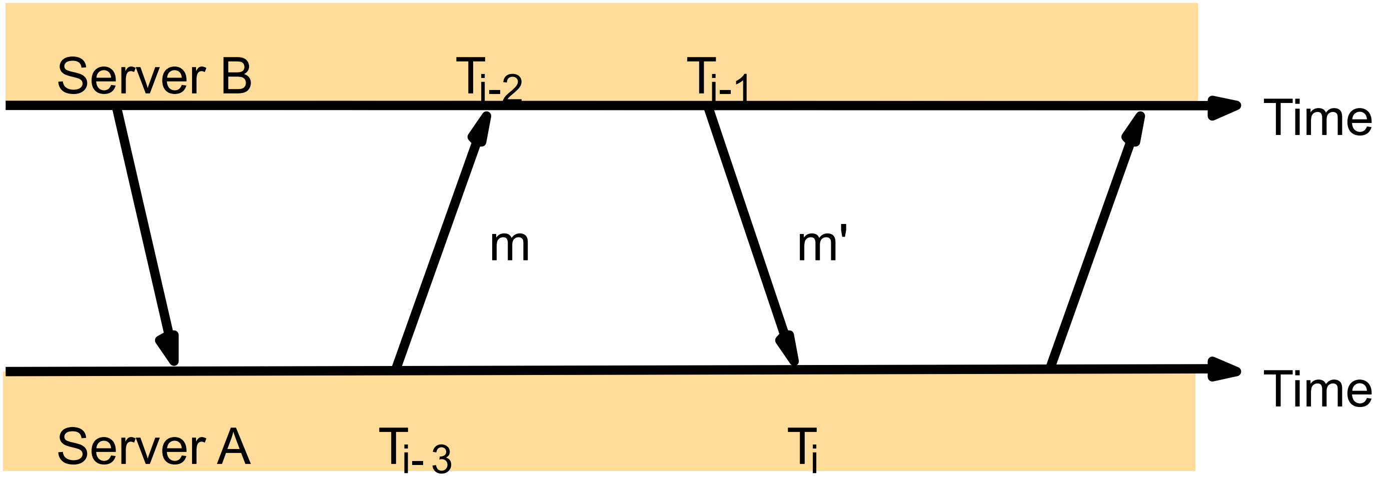
NTP Symmetric Mode



A and B exchange messages and record the send and receive timestamps.

Use these timestamps to compute offset with respect to one another (**o_i**).

NTP Symmetric Mode

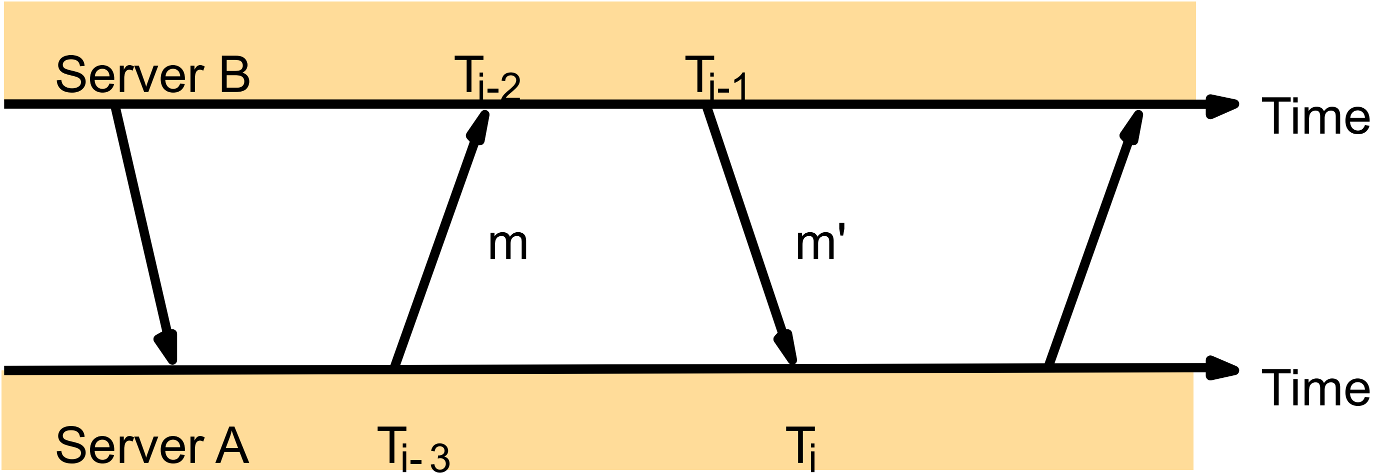


- t and t': actual transmission times for m and m'(unknown)
- o: <u>true</u> offset of clock at B relative to clock at A (unknown)
- o_i: <u>estimate</u> of actual offset between the two clocks
- d_i: estimate of <u>accuracy</u> of o_i ; total transmission times for m and m'; $d_i=t+t'$

$$T_{i-2} = T_{i-3} + t + o$$
$$T_i = T_{i-1} + t' - o$$

$$d_i = t + t' = (T_{i-2} - T_{i-3}) + (T_i - T_{i-1})$$
$$o_i = ((T_{i-2} - T_{i-3}) - (T_i - T_{i-1}))/2$$
$$o = o_i + (t' - t)/2$$

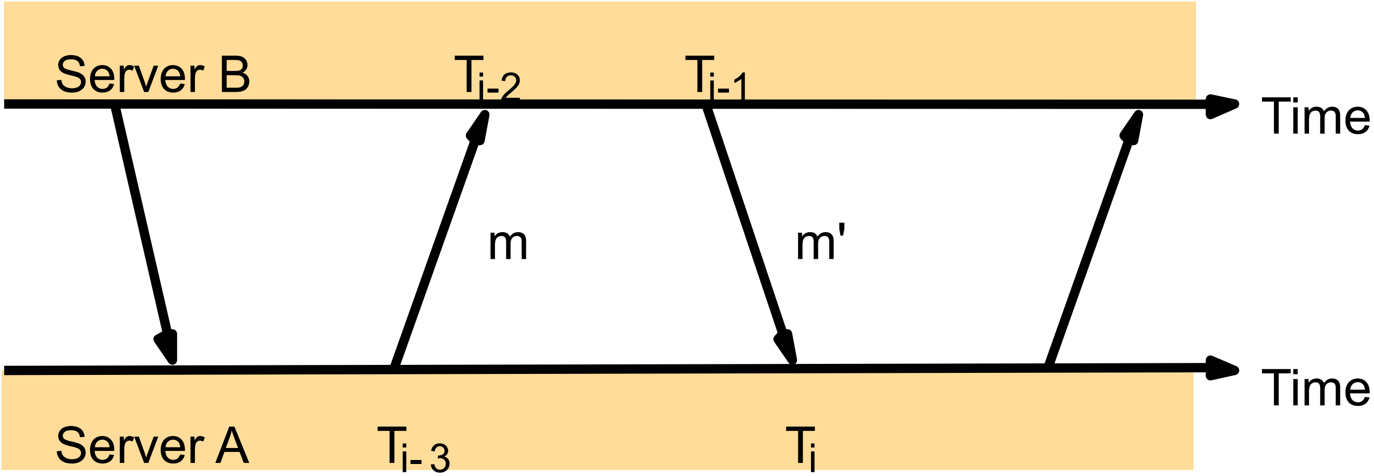
NTP Symmetric Mode



- t and t': actual transmission times for m and m'(unknown)
- o: true offset of clock at B relative to clock at A (unknown)
- o_i: estimate of actual offset between the two clocks
- d_i: estimate of accuracy of o_i ; total transmission times for m and m'; $d_i=t+t'$

$$T_{i-2} = T_{i-3} + t + o$$
$$T_i = T_{i-1} + t' - o$$
$$d_i = t + t' = (T_{i-2} - T_{i-3}) + (T_i - T_{i-1})$$
$$o_i = ((T_{i-2} - T_{i-3}) - (T_i - T_{i-1}))/2$$
$$o = o_i + (t' - t)/2$$
$$t, t' \geq 0$$
$$(o_i - d_i / 2) \leq o \leq (o_i + d_i / 2)$$

NTP Symmetric Mode



A and B exchange messages and record the send and receive timestamps.

Use these timestamps to compute offset with respect to one another (o_i).

A server computes its offset from multiple different sources and adjust its local time accordingly.

Synchronization in asynchronous systems

- Cristian Algorithm
  - Synchronization between a client and a server.
  - Synchronization bound = $(T_{round} / 2) - min \leq T_{round} / 2$
- Berkeley Algorithm
  - Internal synchronization between clocks.
  - A central server picks the average time and disseminates offsets.
- Network Time Protocol
  - Hierarchical time synchronization over the Internet.

Event Ordering

- A usecase of synchronized clocks:
  - Reasoning about order of events.

- Can we reason about order of events without synchronized clocks?

Process, state, events

- Consider a system with **n** processes: $<p_1, p_2, p_3, \ldots, p_n>$
- Each process $\mathbf{p_i}$ is described by its *state* $\mathbf{s_i}$ that gets transformed over time.
    - State includes values of all local variables, affected files, etc.
- $\mathbf{s_i}$ gets transformed when an *event* occurs.
- Three types of events:
    - Local computation.
    - Sending a message.
    - Receiving a message.

Event ordering

- Easy to order events within a single process, based on timestamps.
    - **e_i^j** is the **jth** event of the **ith** process.
    - **history(p_i) = h_i = < e_i^0, e_i^1, e_i^1, …. e_i^m >**
    - Initial state

Event Ordering

- Easy to order events within a single process $\mathbf{p_i}$, based on their time of occurrence.

- How do we reason about events across processes?
  - A message must be *sent* before it gets *received* at another process.

- These two notions help define *happened-before* (HB) relationship denoted by →.
  - **e** → **e'** means **e** *happened before* **e'**.

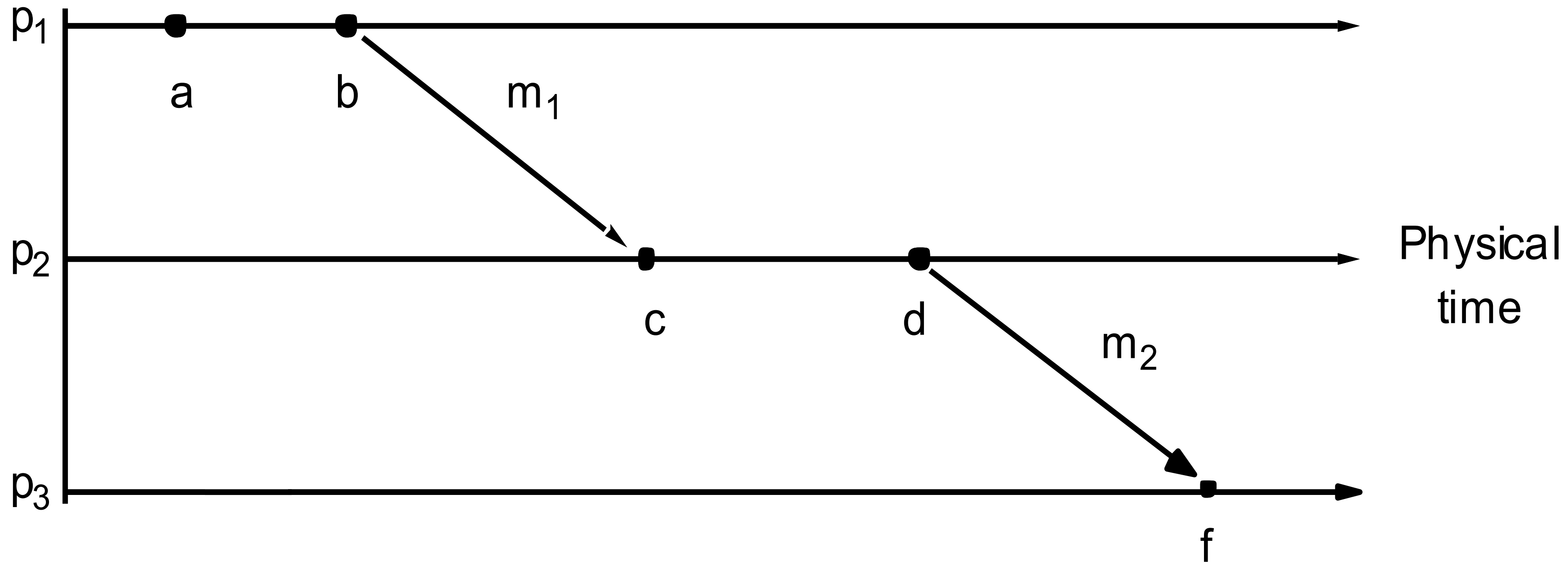
Happened-Before Relationship

- *Happened-before* (HB) relationship denoted by →.
    - **e** → **e'** means **e** *happened before* **e'**.
    - **e** $\rightarrow_i$ **e'** means **e** *happened before* **e'**, as observed by $\mathbf{p_i}$.

- HB rules:
    - If ∃ $\mathbf{p_i}$, **e** $\rightarrow_i$ **e'** then **e** → **e'**.
    - For any message m, **send(m)** → **receive(m)**
    - If **e** → **e'** and **e'** → **e''** then **e** → **e''**

- Also called *"potentially causal"* ordering.

Event Ordering: Example

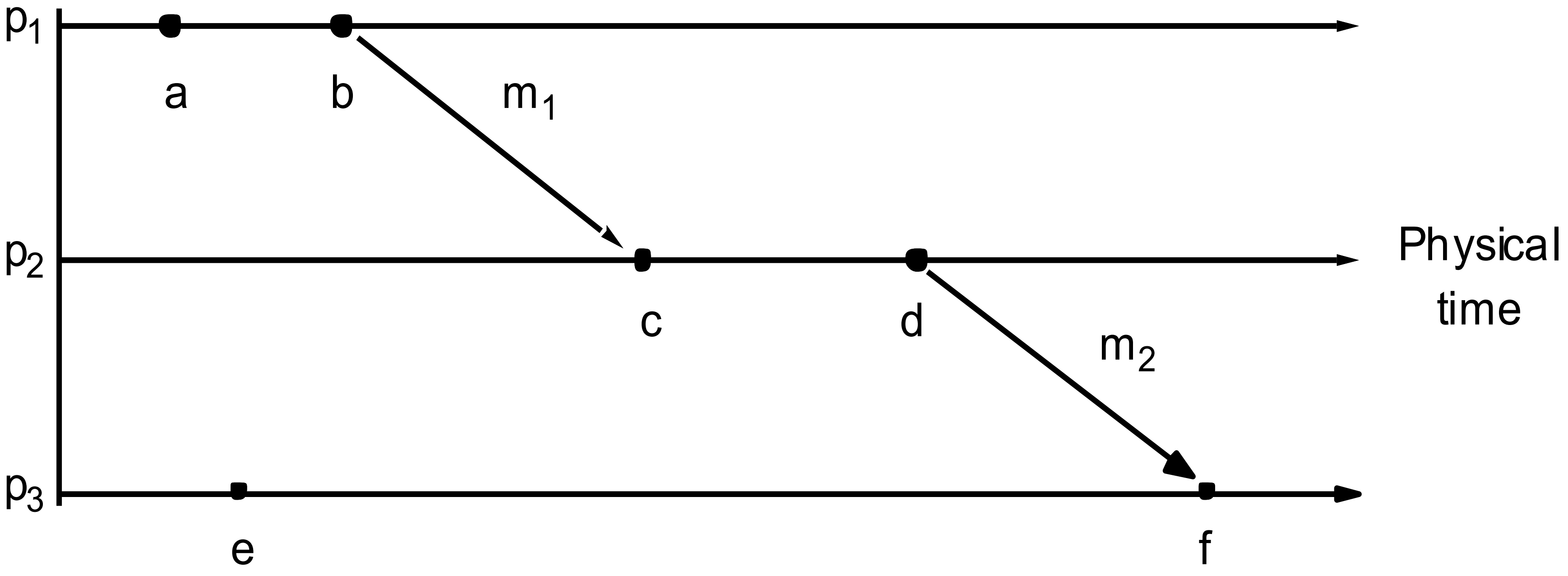


Which event happened first?

a → b and b → c and c → d and d → f

a → b and a → c and a → d and a → f

Event Ordering: Example



What can we say about **e**?

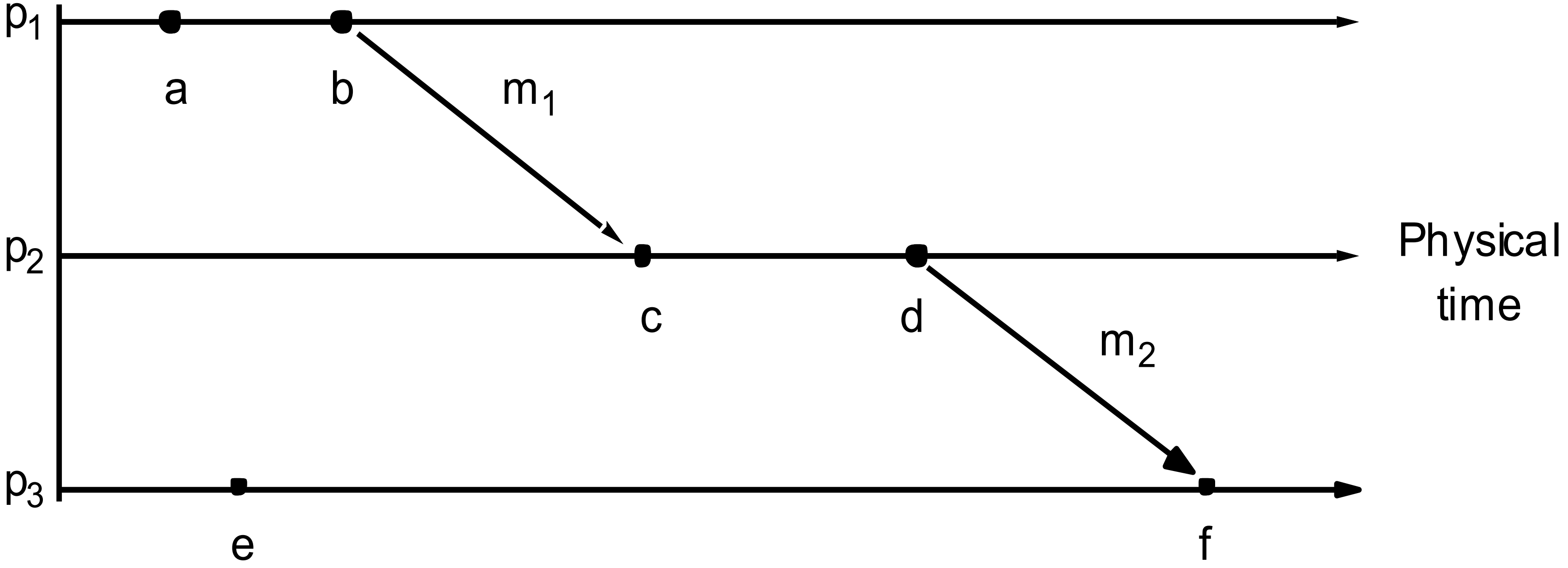
e → **f**

a ↛ **e** and **e** ↛ **a**

a || **e**

a and **e** are *concurrent*.

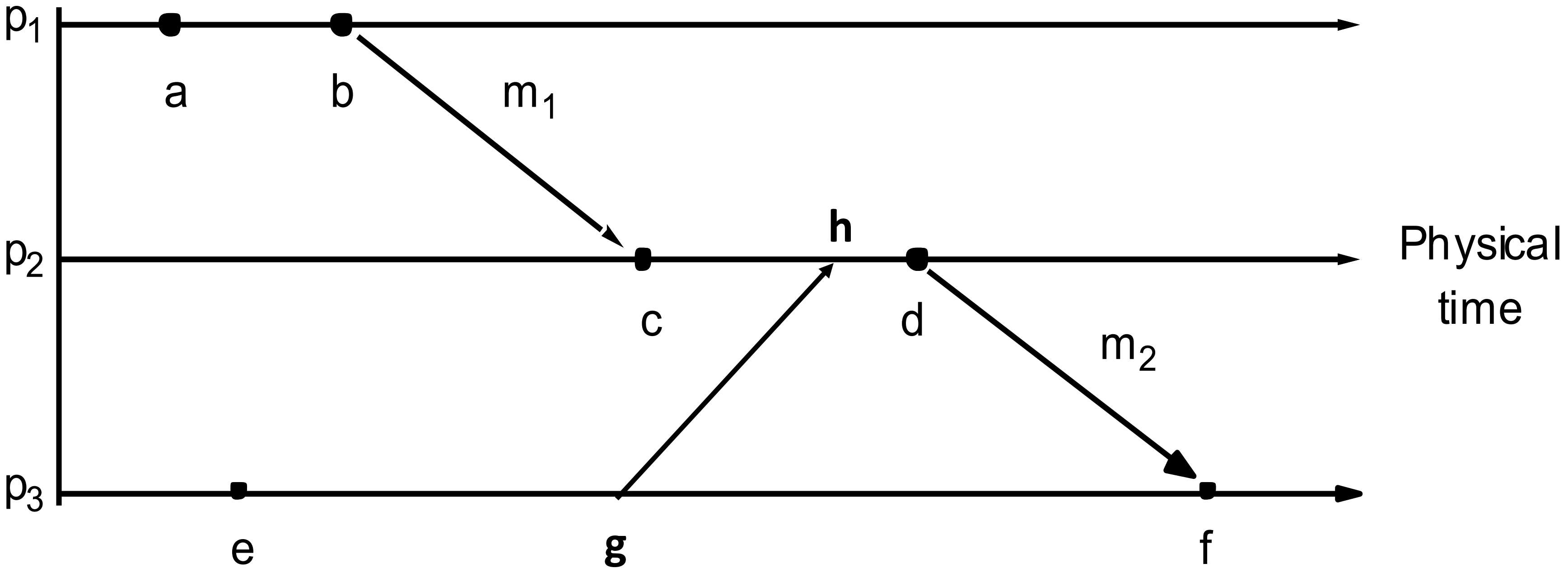
Event Ordering: Example



What can we say about **e** and **d**?

e || d

Logical Timestamps: Example



What can we say about **e** and **d**?

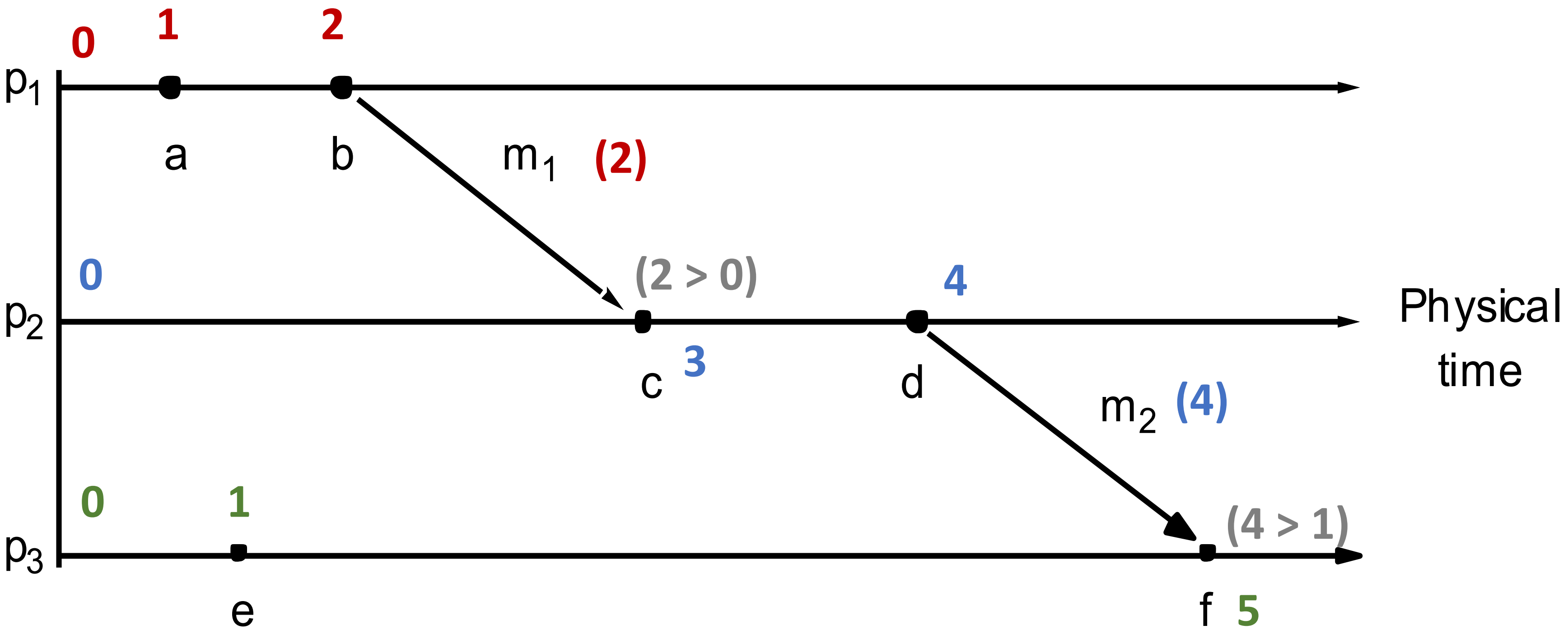
e → d

Lamport's Logical Clock

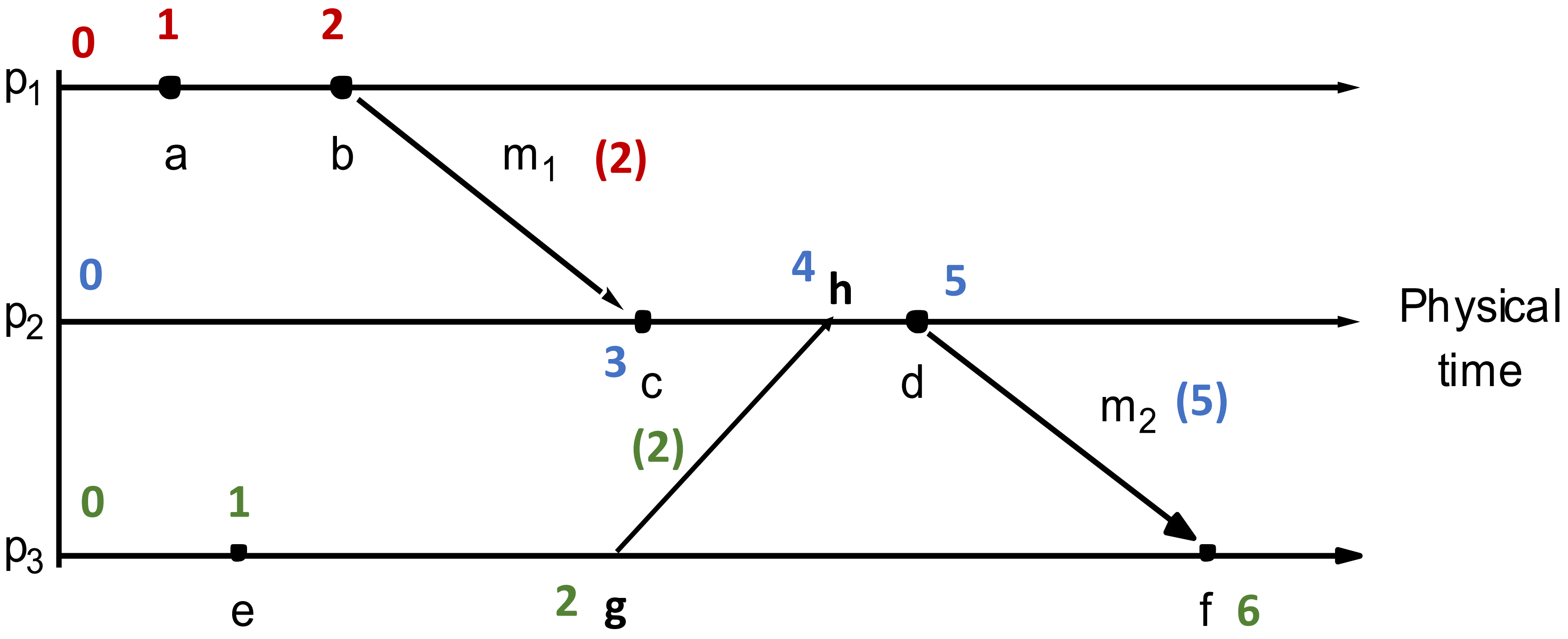
- Logical timestamp for each event that captures the *happened-before* relationship.

- *Algorithm:* Each process $\mathbf{p_i}$
  1. initializes local clock $\mathbf{L_i = 0}$.
  2. increments $\mathbf{L_i}$ before timestamping each event.
  3. piggybacks $\mathbf{L_i}$ when sending a message.
  4. upon receiving a message with clock value $\mathbf{t}$
     - sets $\mathbf{L_i = max(t, L_i)}$
     - increments $\mathbf{L_i}$ (as per point 2).

Logical Timestamps: Example



Logical Timestamps: Example



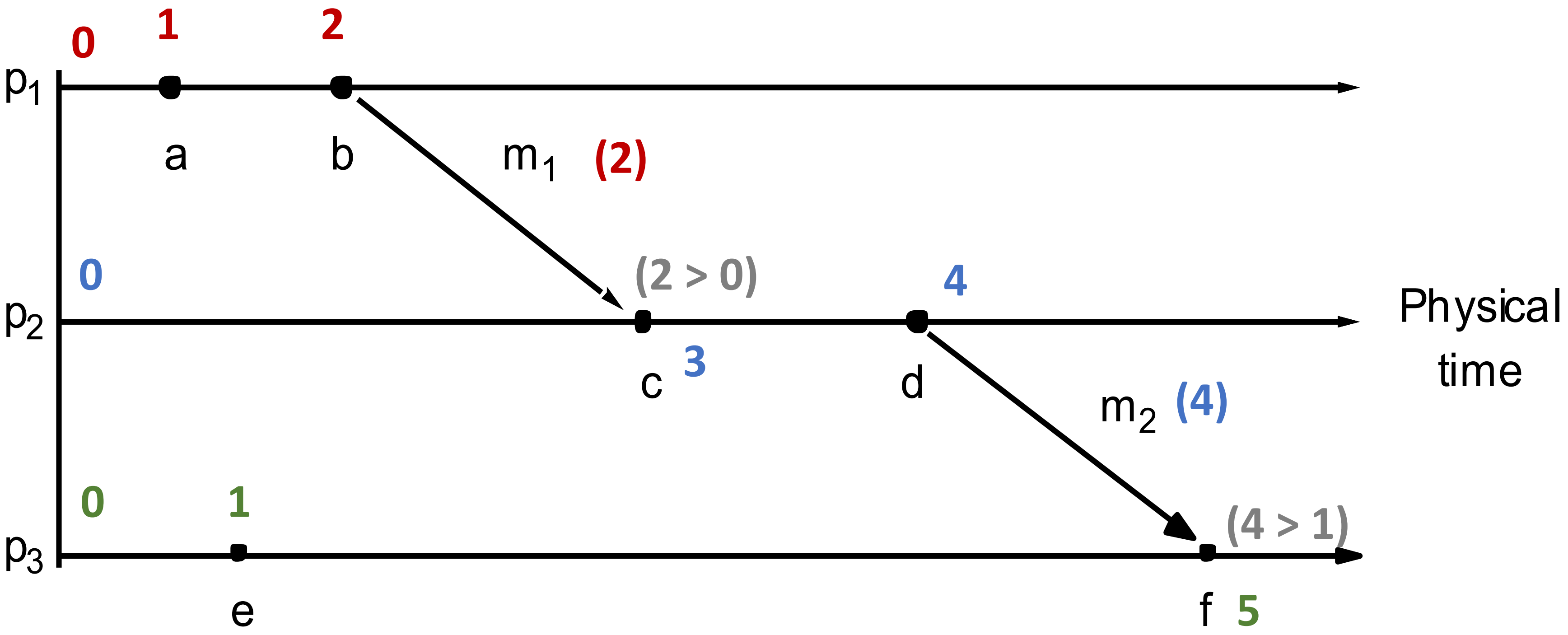
Lamport's Logical Clock

- Logical timestamp for each event that captures the *happened-before* relationship.

- If **e** → **e'** then **L(e) < L(e')**

- What if **L(e) < L(e')**?
  - We cannot say that **e** → **e'**
  - We can say: **e'** ↛ **e**
  - Either **e** → **e'** or **e** || **e'**

Logical Timestamps: Example



$L(e) < L(d)$, $e \parallel d$ $L(e) < L(f)$, $e \rightarrow f$

Vector Clocks

- Each event associated with a vector timestamp.
- Each process maintains vector of clocks $\mathbf{V_i}$
  - $\mathbf{V_i[j]}$ is the clock for process $\mathbf{p_j}$

- Algorithm: each process $\mathbf{p_i}$:
  1. initializes local clock $\mathbf{L_i = 0}$.
  2. increments $\mathbf{L_i}$ before timestamping each event.
  3. piggybacks $\mathbf{L_i}$ when sending a message.
  4. upon receiving a message with clock value $\mathbf{t}$
     - sets $\mathbf{L_i = max(t, L_i)}$
     - increments $\mathbf{L_i}$ (as per point 2).

Vector Clocks

- Each event associated with a vector timestamp.
- Each process maintains vector of clocks V_i
  - $V_i[j]$ is the clock for process p_j

- Algorithm: each process p_i:
  1. initializes local clock $V_i[j] = 0$
  2. increments L_i before timestamping each event.
  3. piggybacks L_i when sending a message.
  4. upon receiving a message with clock value t
     - sets $L_i = \max(t, L_i)$
     - increments L_i (as per point 2).

Vector Clocks

- Each event associated with a vector timestamp.
- Each process maintains vector of clocks V_i
  - $V_i[j]$ is the clock for process p_j

- Algorithm: each process p_i:
  1. initializes local clock $V_i[j] = 0$
  2. increments $V_i[i]$ before timestamping each event.
  3. piggybacks L_i when sending a message.
  4. upon receiving a message with clock value **t**
     - sets $L_i = \max(t, L_i)$
     - increments L_i (as per point 2).

Vector Clocks

- Each event associated with a vector timestamp.
- Each process maintains vector of clocks V_i
  - $V_i[j]$ is the clock for process p_j

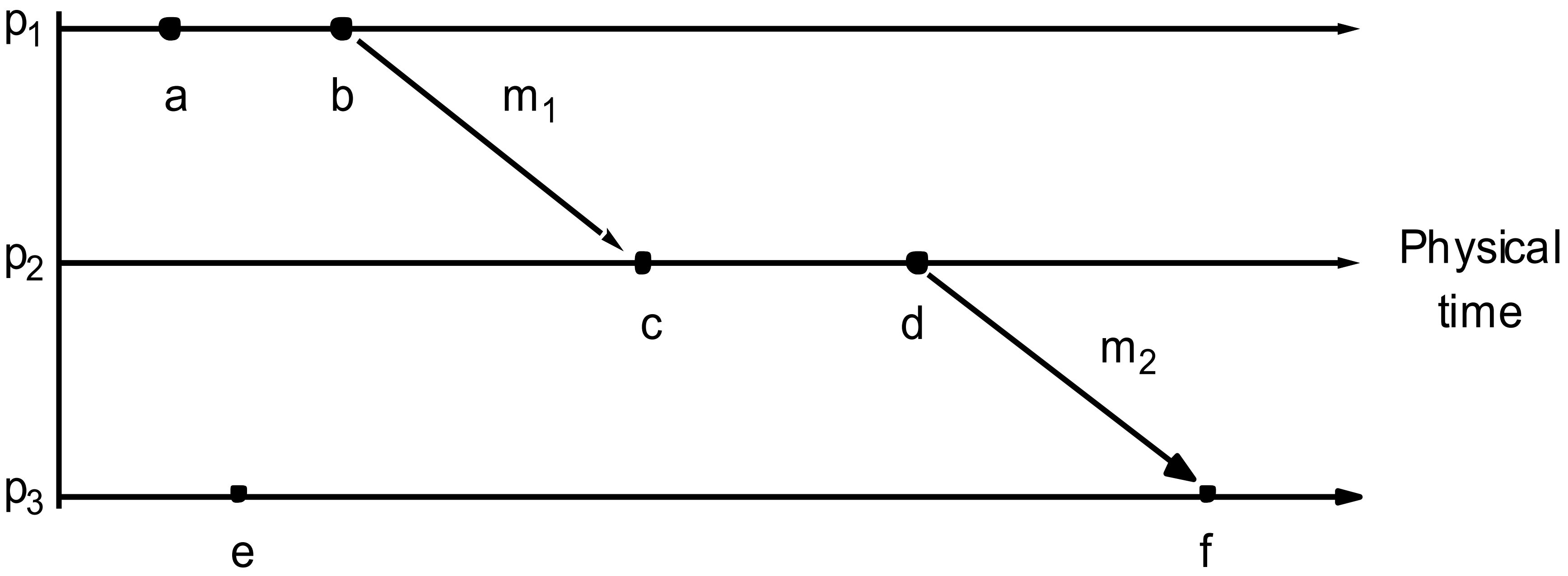
- Algorithm: each process p_i:
  1. initializes local clock $V_i[j] = 0$
  2. increments $V_i[i]$ before timestamping each event.
  3. piggybacks V_i when sending a message.
  4. upon receiving a message with clock value **t**
     - sets $L_i = \max(t, L_i)$
     - increments L_i (as per point 2).

Vector Clocks

- Each event associated with a vector timestamp.
- Each process maintains vector of clocks $\mathbf{V_i}$
  - $\mathbf{V_i[j]}$ is the clock for process $\mathbf{p_j}$

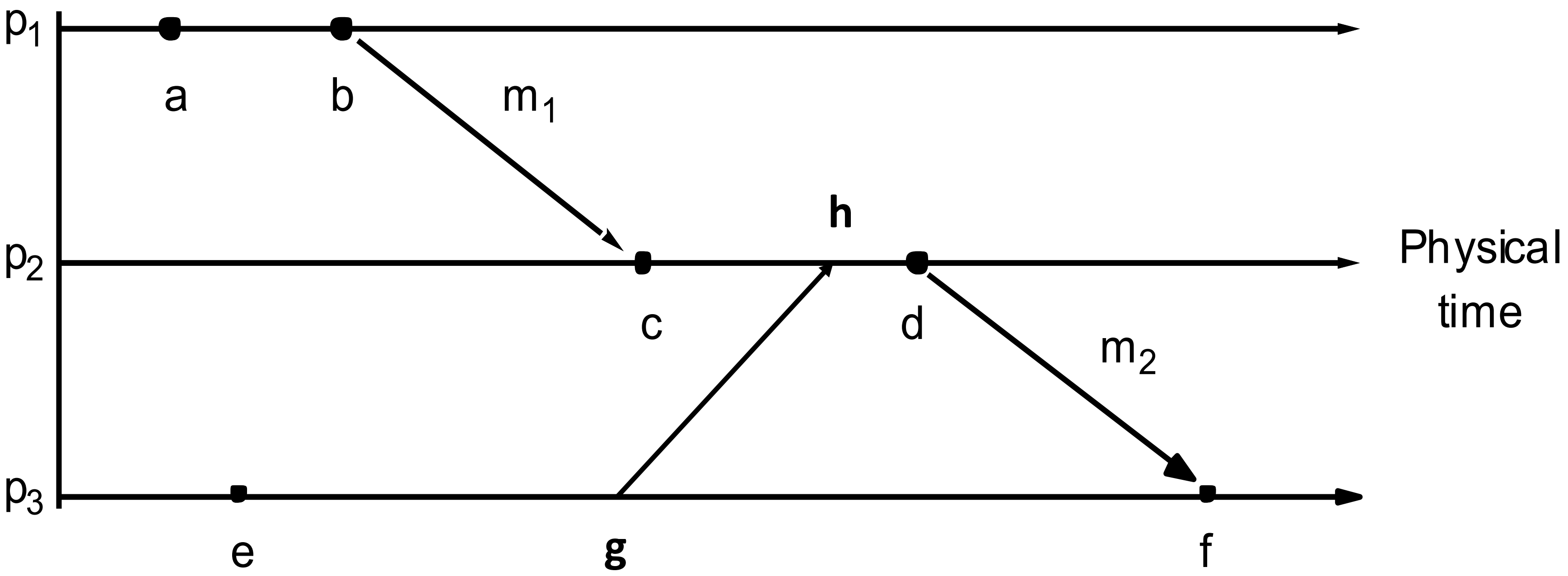
- Algorithm: each process $\mathbf{p_i}$:
  1. initializes local clock $\mathbf{V_i[j] = 0}$
  2. increments $\mathbf{V_i[i]}$ before timestamping each event.
  3. piggybacks $\mathbf{V_i}$ when sending a message.
  4. upon receiving a message with clock value $\mathbf{t}$
     - sets $\mathbf{V_i[j] = max(V_i[j], t[j])}$ for all j=1…n.
     - increments $\mathbf{V_i[i]}$ (as per point 2).

Vector Timestamps: Example



Assign a vector timestamp to each event!

Vector Timestamps: Example



Assign a vector timestamp to each event!

Comparing Vector Timestamps

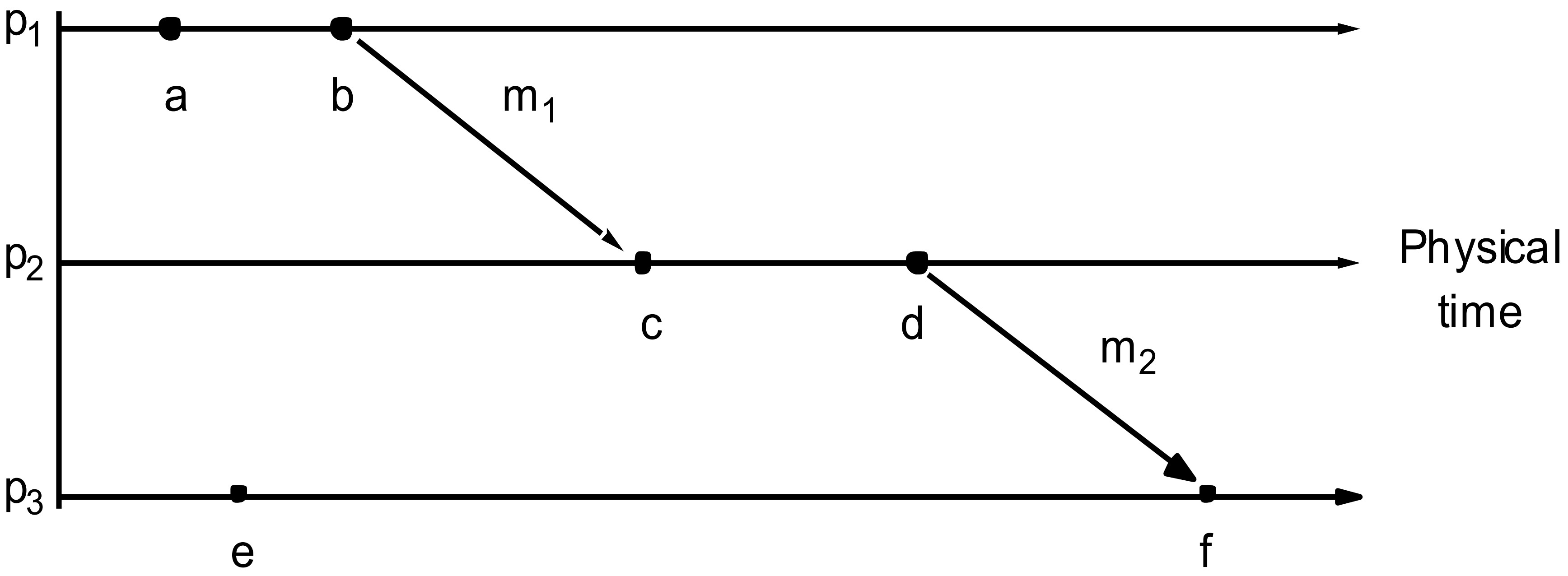
- Let $V(e) = V$ and $V(e') = V'$

- $V = V'$, iff $V[i] = V'[i]$, for all $i = 1, \ldots, n$
- $V \leq V'$, iff $V[i] \leq V'[i]$, for all $i = 1, \ldots, n$
- $V < V'$, iff $V \leq V'$ & $V \neq V'$

  iff $V \leq V'$ & $\exists\, j$ such that $(V[j] < V'[j])$

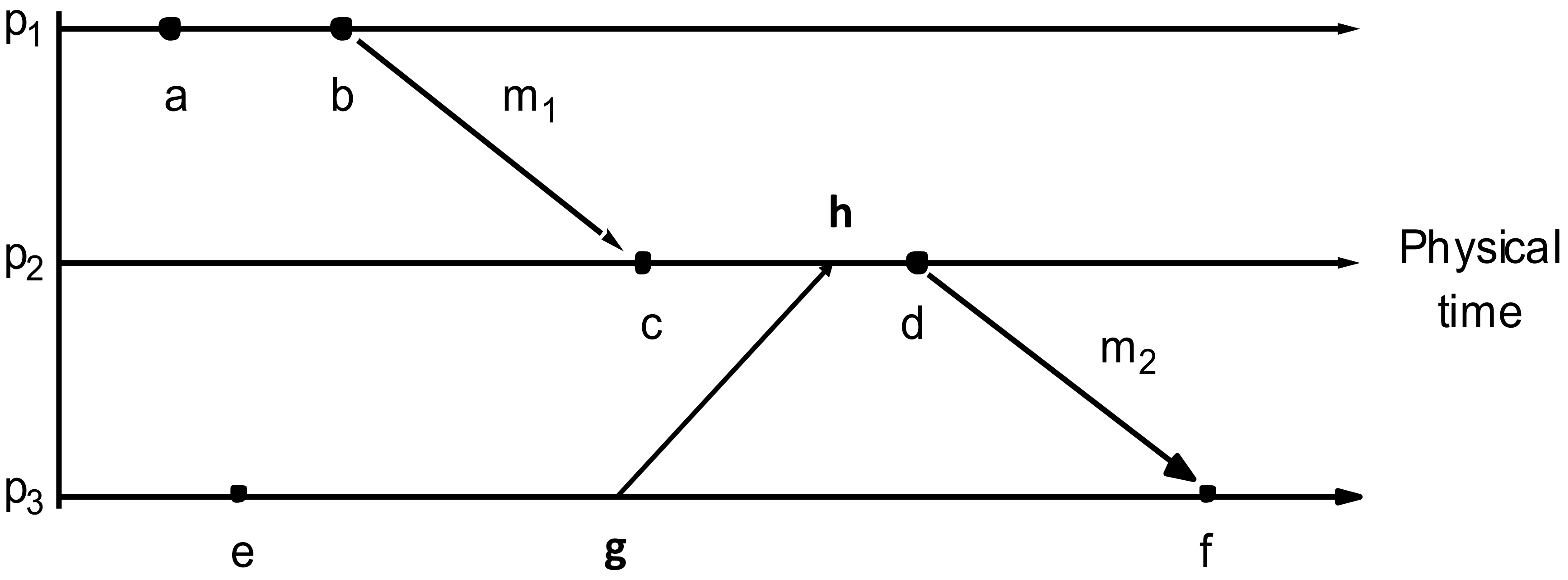
- $e \rightarrow e'$ iff $V < V'$
  - ($V < V'$ implies $e \rightarrow e'$) and ($e \rightarrow e'$ implies $V < V'$)
- $e \,||\, e'$ iff ($V \not< V'$ and $V' \not< V$)

Vector Timestamps: Example



Compare vector timestamps between e & f and e & d.

Vector Timestamps: Example



Compare vector timestamps between e & f and e & d.

Summary

- Time synchronization important for distributed systems
    - Cristian's algorithm
    - Berkeley algorithm
    - NTP

- Relative order of events enough for practical purposes
    - Lamport's logical clocks
    - Vector clocks

- **Next class:** Global State and Snapshots

HW1 will be released tonight!

- We will release HW1 by tonight.
- Announcement and submission instructions will be made available on Campuswire.
- Due on Feb 13, 11:59pm.
- Relevant material for the last1-2 questions will get covered by next week.