Chapter 5

Synchronization

- Part I Clock Synchronization & Logical clocks
- Part II Global State, Election, & Critical Sections
- Part III Transactions

Chapter 5

Synchronization

Part I

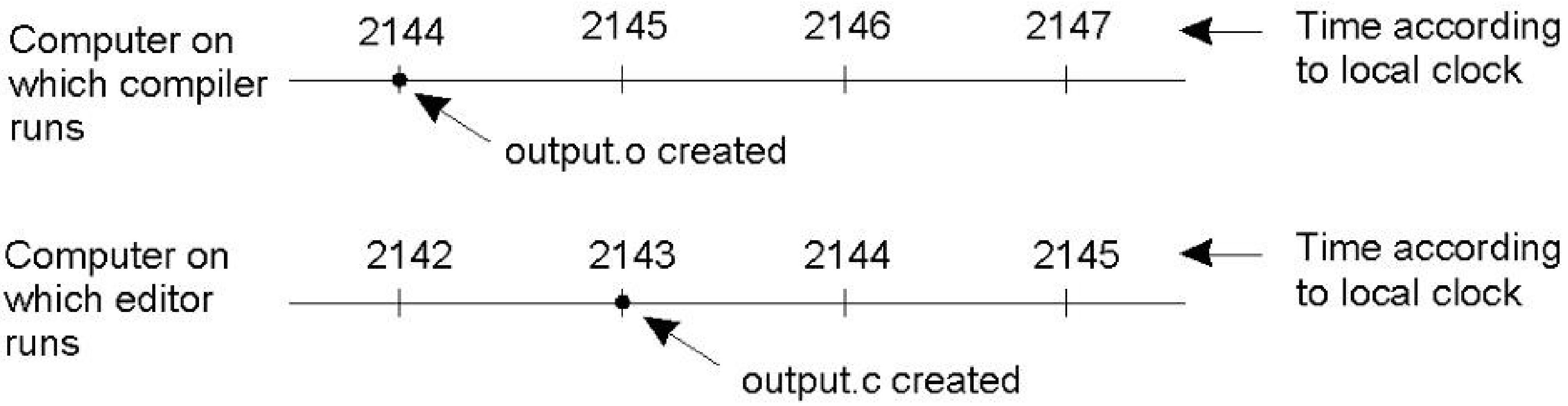
Clock Synchronization

& Logical clocks

Lack of Global Time in DS

- It is impossible to guarantee that physical clocks run at the same frequency
- Lack of global time, can cause problems
- Example: UNIX make
  - Compile output.c at a client
  - output.o is at a server
  - Client machine clock can be lagging behind the server machine clock

Lack of Global Time – Example



When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

A Note on Real Time

- International Atomic Time (TAI) = avg of 133 cesium clocks
- Universal Coordinated Time (UTC) = TAI ± leap seconds (to adjust with solar time)
- Radio stations can broadcast UTC for receivers to collect it
  - WWV SW station in Colorado

Physical Clock Synchronization (1)

- External: synchronize with an external resource, UTC source

$$|S(t) - C_i(t)| < D$$

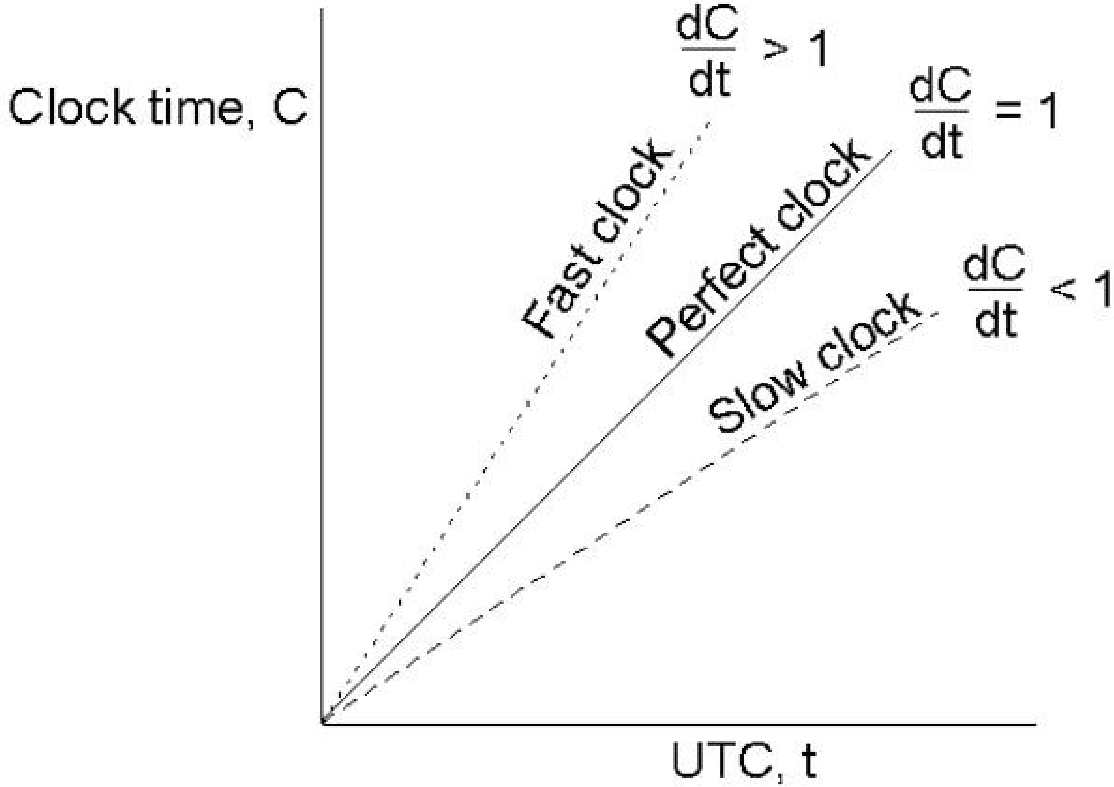
- Internal: synchronize without access to an external resource

$$|C_i(t) - C_j(t)| < D$$

- Monotonicity: time never goes back

$$t' > t \Rightarrow C(t') > C(t)$$

Physical Clock Skew



The relation between clock time and UTC when clocks tick at different rates.

Cristian's Algorithm – External Synch

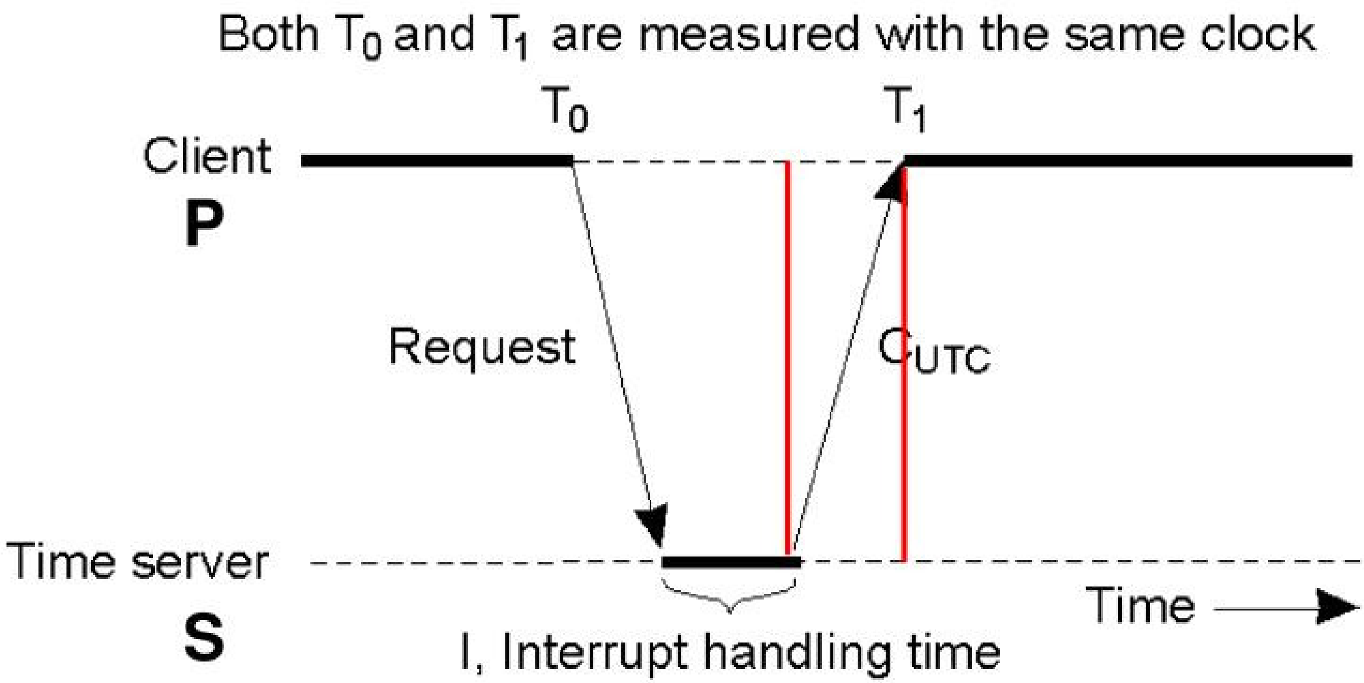
- External source S
- Denote clock value at process X by C(X)

Periodically, a process P:

1. send message to S, requesting time
2. Receive message from S, containing time C(S)
3. **Adjust C at P, C(P) = C(S)**

- Reply takes time
- Time for different replies varies
- When P adjusts C(P) to C(S), C(S) > C(P)

Cristian's Algorithm



Getting the current time from a time server.

Adjusting Client's Clock

- T_{round} = time to send a request and receive reply
- T_0 time request is sent
- T_1 time reply received
- $T_{round} = (T_1 - T_0)/2$ (estimate)
- C at P = time(S) + T_{round}
- Works if both request and reply are sent on the same network
- What if P's clock ticks faster than S's clock?

Improvements to Cristian's Algorithm

- I = interrupt handling time at S
- $T_{round} = T_1 - T_0 - I$

- Take several measurements for T_{round}
- Discard best and worst cases
- Average measurements over time

Berkeley Algorithm – Internal Synch

Periodically,

S: send C(S) to each client P

P: calculate $\delta_P = C(P) - C(S)$

send δ_P to S

S: receive all δ_P's

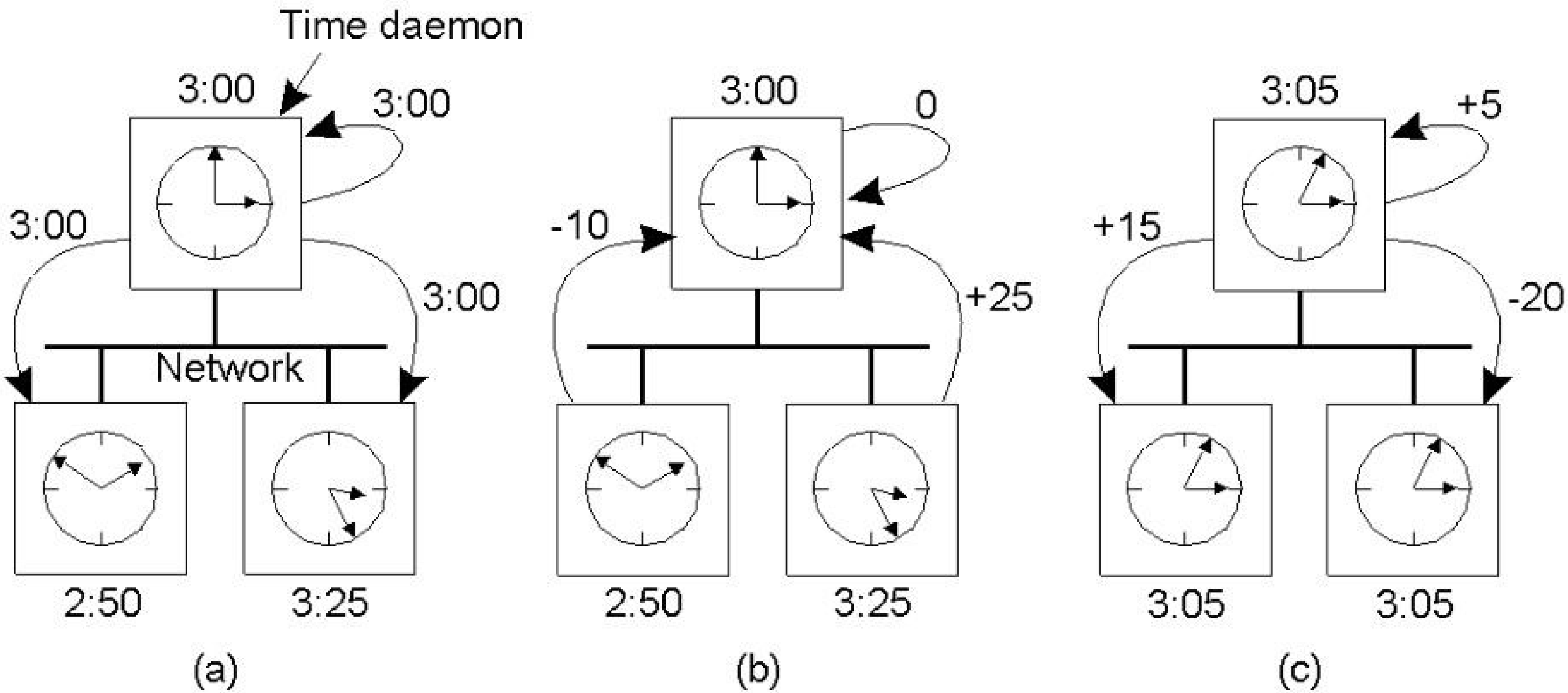
compute an average Δ of δ_P, including δ_S

send Δ to all clients P

P: apply Δ to C(P)

- Propagation time?
- Extreme cases? Faulty clocks?
- Time server fails?

The Berkeley Algorithm



a) The time daemon asks all the other machines for their clock values
b) The machines answer
c) The time daemon tells everyone how to adjust their clock

Importance of Synchronized Clocks

- New H/W and S/W for synchronizing clocks is easily available
- Now a days, it is possible to keep millions of clocks synchronized to within few milliseconds of UTC
- **New algorithms can benefit**

Logical Clocks

- For many DS algorithms, it suffices for machines to agree on the same time, NOT necessarily the real time

- Lamport's timestamps
- Vector timestamps
- Matrix timestamps

Lamport's Timestamps

- Events:
  - send message m, send(m)
  - Receive message m, receive(m)
  - Other internal (to the process) events
    - Read, write, etc …

- $e <_i e'$ denotes: event e happens before event e' at process P_i

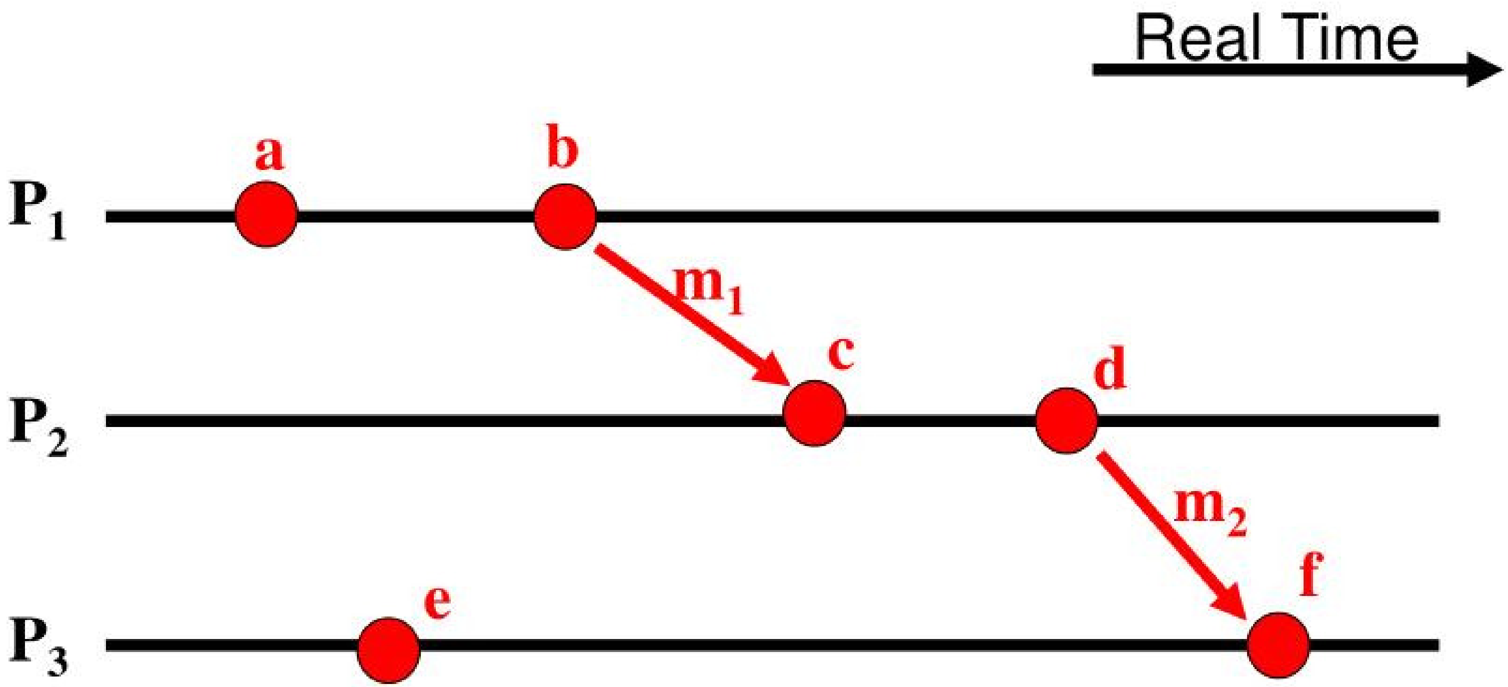
Lamport's "Happens-Before" Partial Order

- Given two events e & e', e < e' if:

1. **Same process**: e $<_i$ e', for some process P_i
2. **Same message**: e = send(m) and e'=receive(m) for some message m
3. **Transitivity**: there is an event e'' such that e < e'' and e'' < e'

Concurrent Evets

- Given two events e & e':
- If not e < e' and not e'< e, then e || e'



Lamport’s Logical Clocks

- Processes keep S/W counters
  - **TS_i** denotes counter of process P_i
  - TS_i always increases
- P_i timestamps events with TS_i
- **e.TS** denotes timestamp of event e
- **m.TS** denotes time stamp attached to message m

Lamport's Timestamp Algorithm

P_i: (initially $TS_i = 0$)

On event e:

Case e is **send(m)**, where m is a message

$TS_i = TS_i + 1$

$m.TS = TS_i$

Case e is **receive(m)**, where m is a message

$TS_i = \max(TS_i, m.TS)$

$TS_i = TS_i + 1$

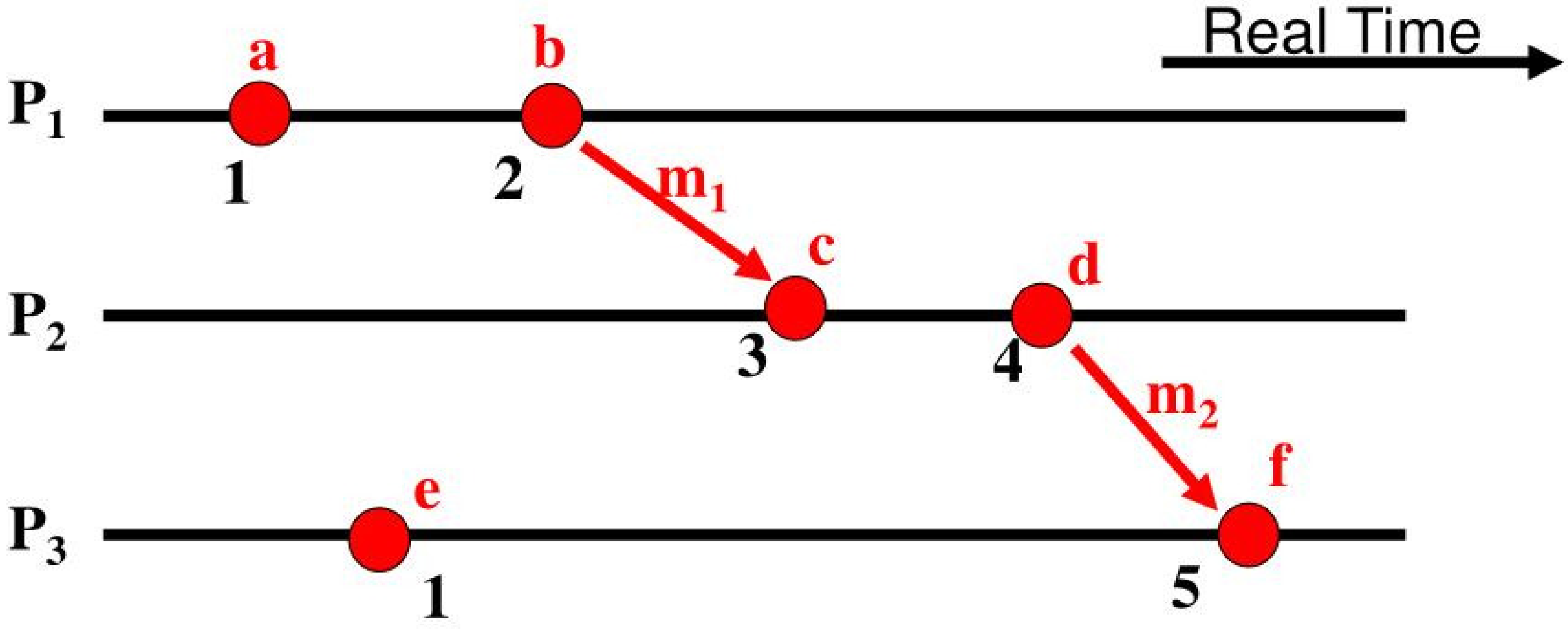
Case e is any other event

$TS_i = TS_i + 1$

$e.TS = Ts_i$ /* timestamp e */

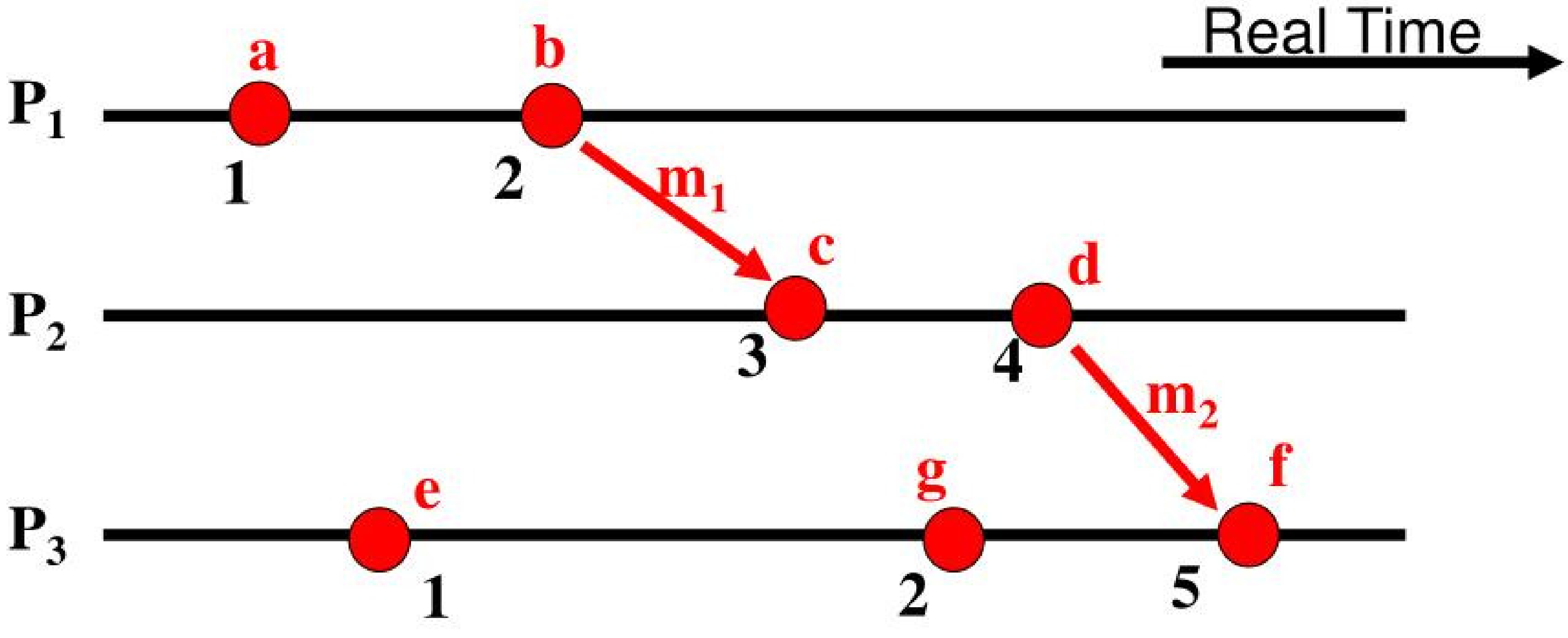
Lamport's Algorithm Analysis (1)

- Claim: if e < e', then e.TS < e'.TS
- Proof: by induction on the length of the sequence of events relating to e and e'



Lamport's Algorithm Analysis (2)

- Claim: if e.TS < e'.TS, then it is **not** necessarily true that e < e'



Total Ordering of Events

- Happens before is only a partial order

- Make the timestamp of an event e of process P_i be:

  $$(e.TS, i)$$

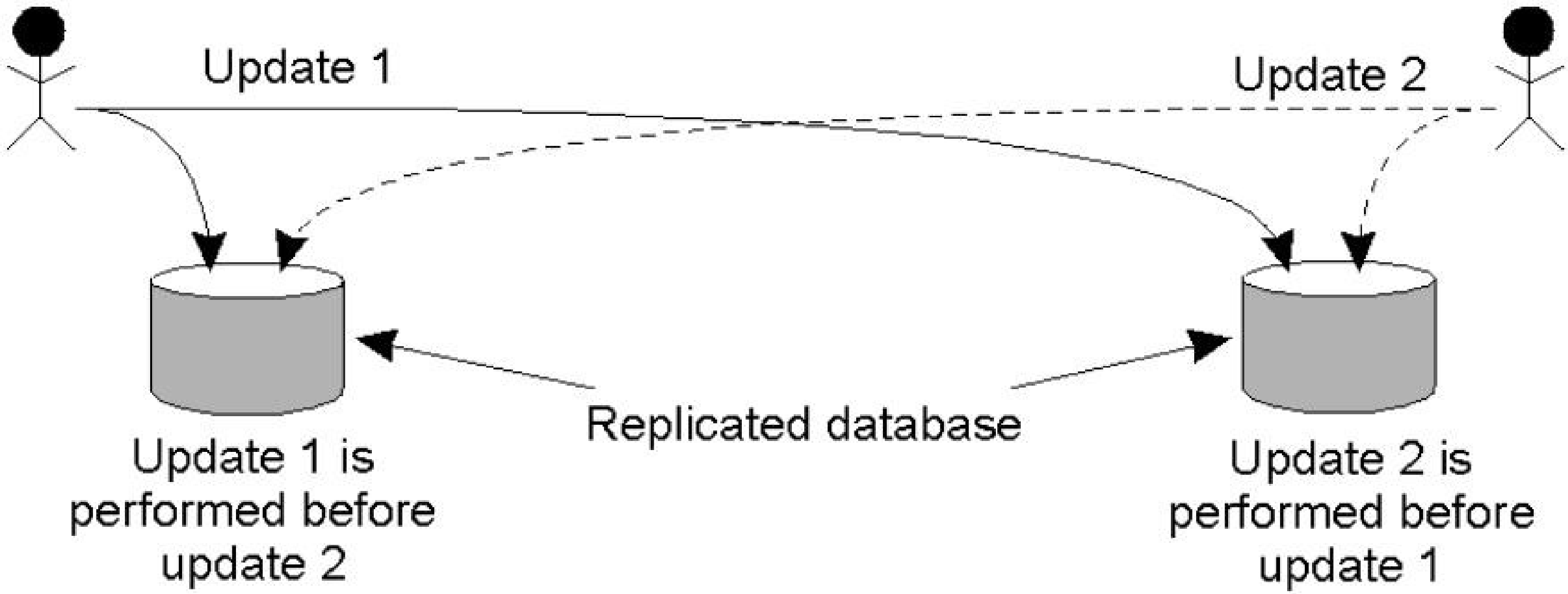
- $(a,b) < (c,d)$ iff $a < c$, or $a = c$ and $b < d$

Application 1: Critical Sections

- Lamport used his timestamps to order the entry of processes to enter a critical section

- **Exercise**

Application 2: Totally-Ordered Multicasting



Updating a replicated database and leaving it in an inconsistent state.

Extending Timestamps

- e.TS < e'.TS does not imply e < e'
- P_i's clock is a vector $VT_i[]$
- $VT_i[i]$ = number of events P_i has stamped
- $VT_i[j]$ = what P_i thinks number of events P_j has stamped ($i \neq j$)

Vector Timestamp Algorithm

P_i: (initially $VT_i = [0, \ldots, 0]$)

On event e:

Case e is **send(m)**, where m is a message

$VT_i[i] = VT_i[i] + 1$

$m.VT = VT_i$

Case e is **receive(m)**, where m is a message

for j = 1 to N /* vector length */

$VT_i[j] = \max(VT_i[j], m.VT[j])$

$VT_i[i] = VT_i[i] + 1$

Case e is any other event

$VT_i[i] = VT_i[i] + 1$

$e.VT = VT_i$ /* timestamp e */

Comparing Vectors

- VT = VT’ iff VT[i] = VT’[i] for all i
- VT ≤ VT’ iff VT[i] ≤ VT’[i]
- VT < VT’ iff VT[i] ≤ VT’[i] & VT ≠ VT’

Vector Timestamp Analysis

- Claim: e < e’ **iff** e.VT < e’.VT