

Distributed systems

Lecture 12: Clock synchronization and logical time

Michaelmas 2018

Dr Richard Mortier and
Dr Anil Madhavapeddy

(With thanks to Dr Robert N. M. Watson
and Dr Steven Hand)

Last time

- Object-Oriented Middleware (OOM)
- Started to look at time in distributed systems
 - Coordinating actions between processes
- Physical clocks '**tick**' based on physical processes (e.g. oscillations in quartz crystals, atomic transitions)
 - Imperfect, so gain/lose time over time
 - (wrt nominal perfect 'reference' clock (e.g., UTC))
- What clocks in computers are for...

The clock synchronization problem

- In distributed systems, we'd like all the different nodes to have the same notion of time, but
 - quartz oscillators oscillate at slightly different frequencies (time, temperature, manufacture)
- Hence clocks tick at different rates:
 - create ever-widening gap in perceived time
 - this is called **clock drift**
- The difference between two clocks at a given point in time is called **clock skew**
- Clock synchronization aims to minimize clock skew between two (or a set of) different clocks

Clock skew and clock drift



08:00:00



08:00:00

February 18, 2012
08:00:00

NB: Steve Hand's watches, not mine.

Clock skew and clock drift



08:01:24

Skew = 84 seconds

Drift = $84s / 34 \text{ days}$
= +2.47s per day

March 23, 2012
08:00:00



08:01:48

Skew = 108 seconds

Drift = $108s / 34 \text{ days}$
= +3.18s per day

Dealing with drift

- A clock can have positive or negative drift with respect to a reference clock (e.g. UTC)
 - Need to [re]synchronize periodically
- Can't just set clock to 'correct' time
 - Jumps (particularly backward!) can confuse apps
- Instead aim for gradual compensation
 - If clock fast, make it run slower until correct
 - If clock slow, make it run faster until correct

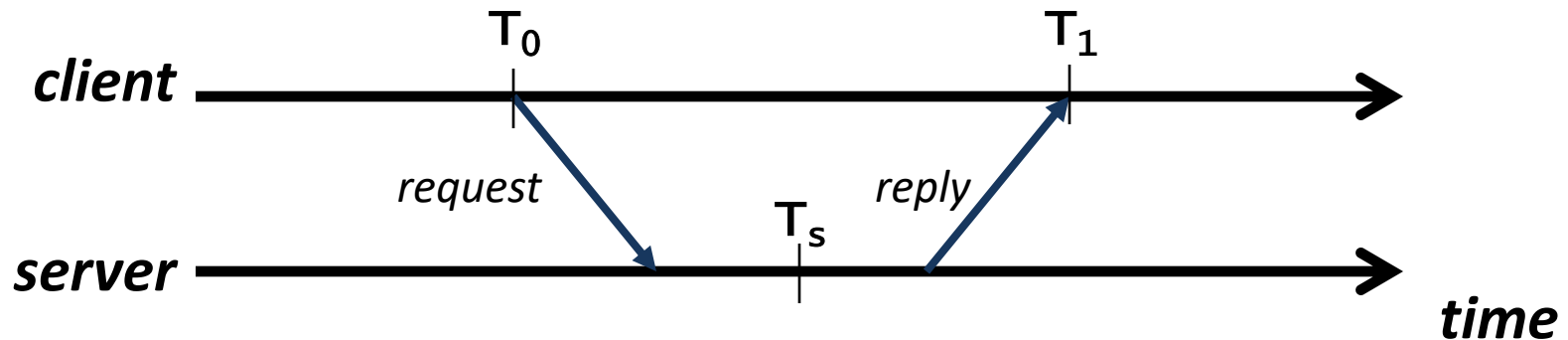
Compensation

- Most systems relate real-time to cycle counters or periodic interrupt sources
 - E.g. calibrate CPU **Time-Stamp Counter (TSC)** against **CMOS Real-Time Clock (RTC)** at boot, and compute scaling factor (e.g. cycles per ms)
 - Can now convert TSC differences to real-time
 - Similarly can determine how much real-time passes between periodic interrupts: call this **delta**
 - On interrupt, add delta to software real-time clock
- Making small changes to delta gradually adjusts time
 - Once synchronized, change delta back to original value
 - (Or try to estimate drift & continually adjust delta)
 - Minimise time discontinuities from **stepping**

Obtaining accurate time

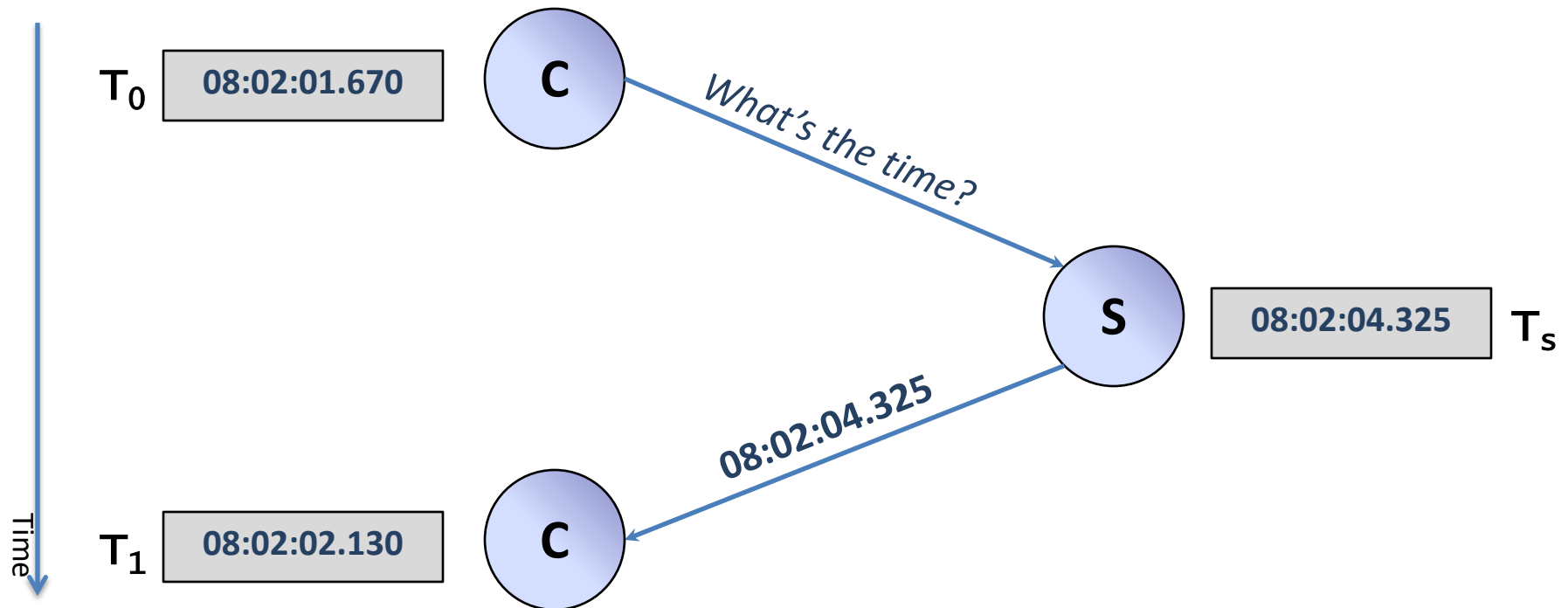
- Of course, need some way to know correct time (e.g. UTC) in order to adjust clock!
 - could attach a GPS receiver (or GOES receiver) to computer, and get $\pm 1\text{ms}$ (or $\pm 0.1\text{ms}$) accuracy...
 - ...but too expensive/clunky for general use
 - (RF in server rooms and data centres non-ideal)
- Instead can ask some machine with a more accurate clock over the network: a **time server**
 - e.g. send RPC `getTime()` to server
 - What's the problem here?

Cristian's Algorithm (1989)



- Attempt to compensate for network delays
 - Remember local time just before sending: T_0
 - Server gets request, and puts T_s into response
 - When client receives reply, notes local time: T_1
 - Correct time is then approximately $(T_s + (T_1 - T_0) / 2)$ (assumes symmetric behaviour...)

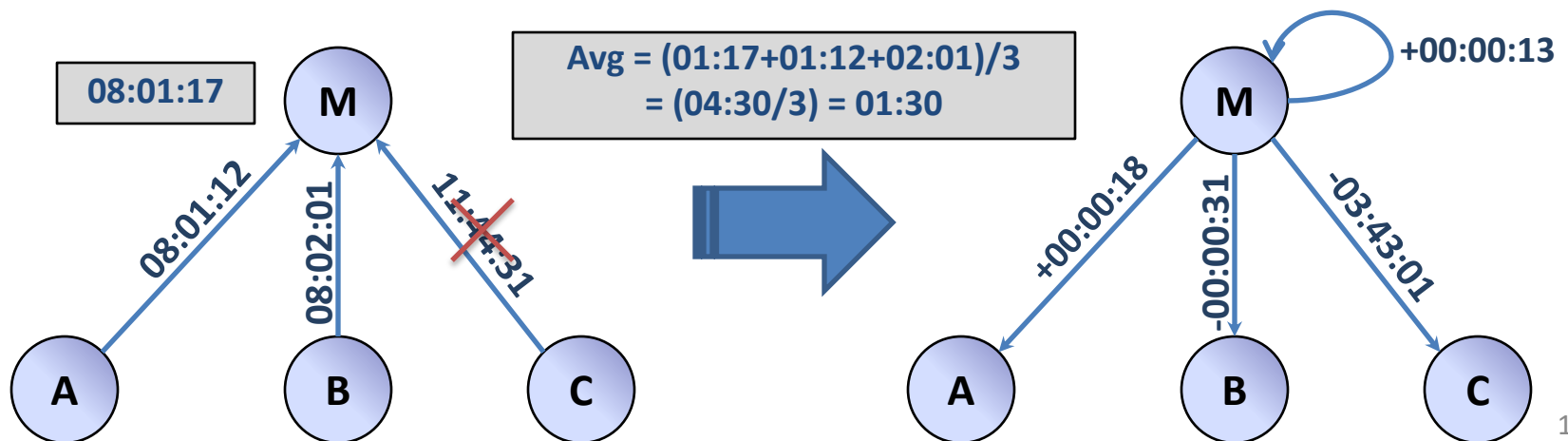
Cristian's Algorithm: Example



- RTT = 460ms, so one way delay is [approx] 230ms.
- Estimate correct time as $(08:02:04.325 + 230\text{ms}) = 08:02:04.555$
- Client gradually adjusts local clock to gain 2.425 seconds

Berkeley Algorithm (1989)

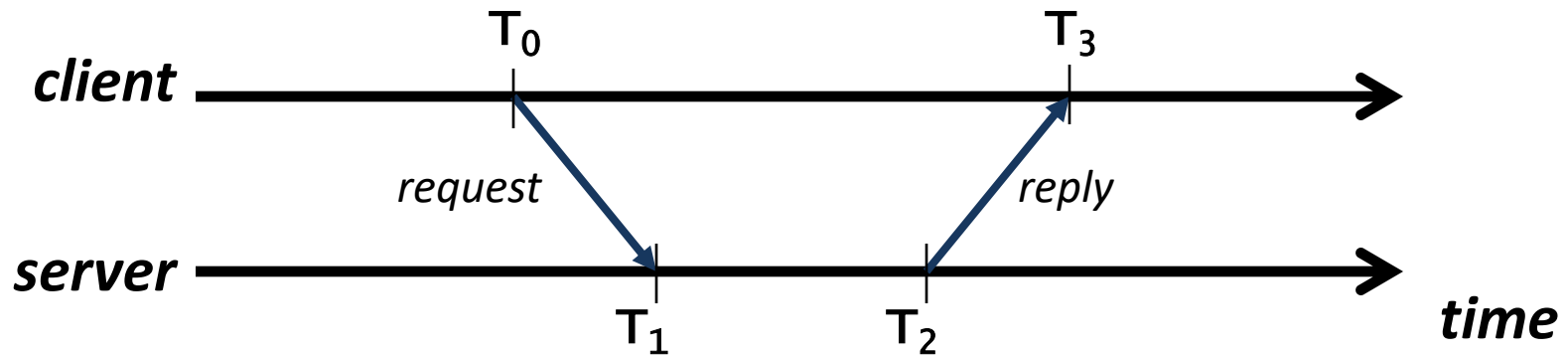
- Don't assume have an accurate time server
- Try to synchronize a set of clocks to the average
 - One machine, **M**, is designated the master
 - **M** periodically polls all other machines for their time
 - (can use Cristian's technique to account for delays)
 - Master computes average (including itself, but ignoring outliers), and sends an adjustment to each machine



Network Time Protocol (NTP)

- Previous schemes designed for LANs; in practice today's systems use NTP:
 - Global service designed to enable clients to stay within (hopefully) a few ms of UTC
- Hierarchy of clocks arranged into **strata**
 - Stratum0 = atomic clocks (or maybe GPS, GEOS)
 - Stratum1 = servers directly attached to stratum0 clock
 - Stratum2 = servers that synchronize with stratum1
 - ... and so on
- Timestamps made up of seconds and 'fraction'
 - e.g. 32 bit seconds-since-epoch; 32 bit 'picoseconds'

NTP algorithm

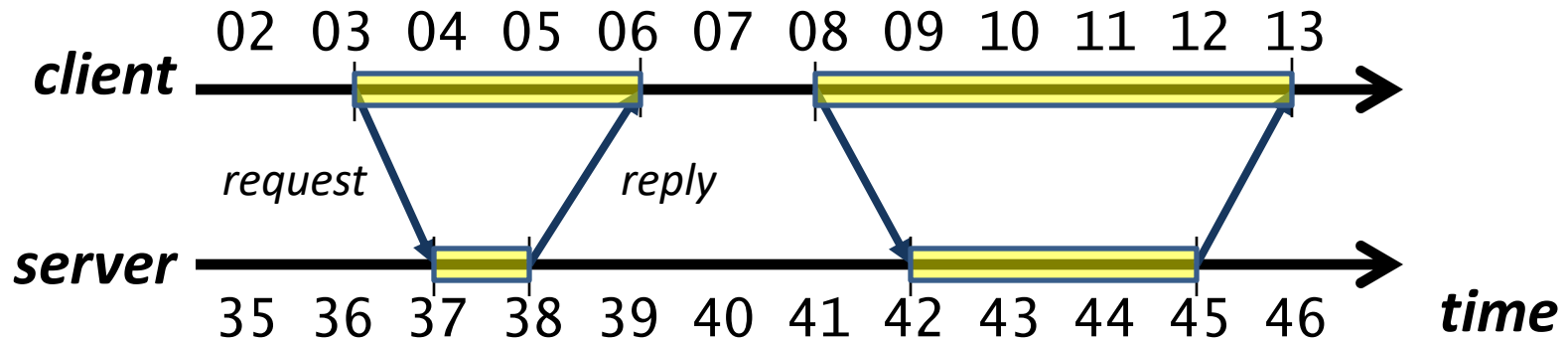


- UDP/IP messages with slots for four timestamps
 - systems insert timestamps at earliest/latest opportunity
- Client computes:
 - Offset $\mathbf{O} = ((T_1 - T_0) + (T_2 - T_3)) / 2$
 - Delay $\mathbf{D} = (T_3 - T_0) - (T_2 - T_1)$
- Relies on symmetric messaging delays to be correct (but now excludes variable processing delay at server)

Measured difference in average timestamps: $(T_1 + T_2) / 2 - (T_0 + T_3) / 2$

Estimated two-way communication delay minus processing time

NTP example



- First request/reply pair:
 - Total message delay is $((6-3) - (38-37)) = 2$
 - Offset is $((37-3) + (38-6)) / 2 = 33$
- Second request/reply pair:
 - Total message delay is $((13-8) - (45-42)) = 2$
 - Offset is $((42-8) + (45-13)) / 2 = 33$

NTP: additional details (1)

- NTP uses multiple requests per server
 - Remember <offset, delay> in each case
 - Calculate the **filter dispersion** of the offsets & discard outliers
 - Chooses remaining candidate with the smallest delay
- NTP can also use multiple servers
 - Servers report **synchronization dispersion** = estimate of their quality relative to the root (stratum 0)
 - Combined procedure to select best samples from best servers (see RFC 5905 for the gory details)

NTP: additional details (2)

- Various operating modes:
 - **Broadcast** (“multicast”): server advertises current time
 - **Client-server** (“procedure call”): as described on previous slides
 - **Symmetric**: between a set of NTP servers
- Security is supported
 - Authenticate server, prevent replays
 - Cryptographic cost compensated for

Physical clocks: summary

- Physical devices exhibit **clock drift**
 - Even if initially correct, they tick too fast or too slow, and hence time ends up being wrong
 - Drift rates depend on the specific device, and can vary with time, temperature, acceleration, ...
- Instantaneous difference between clocks is **clock skew**
- **Clock synchronization algorithms** attempt to minimize the skew between a set of clocks
 - Decide upon a target correct time (atomic, or average)
 - Communicate to agree, compensating for delays
 - In reality, will still have 1-10ms skew after sync ;-)

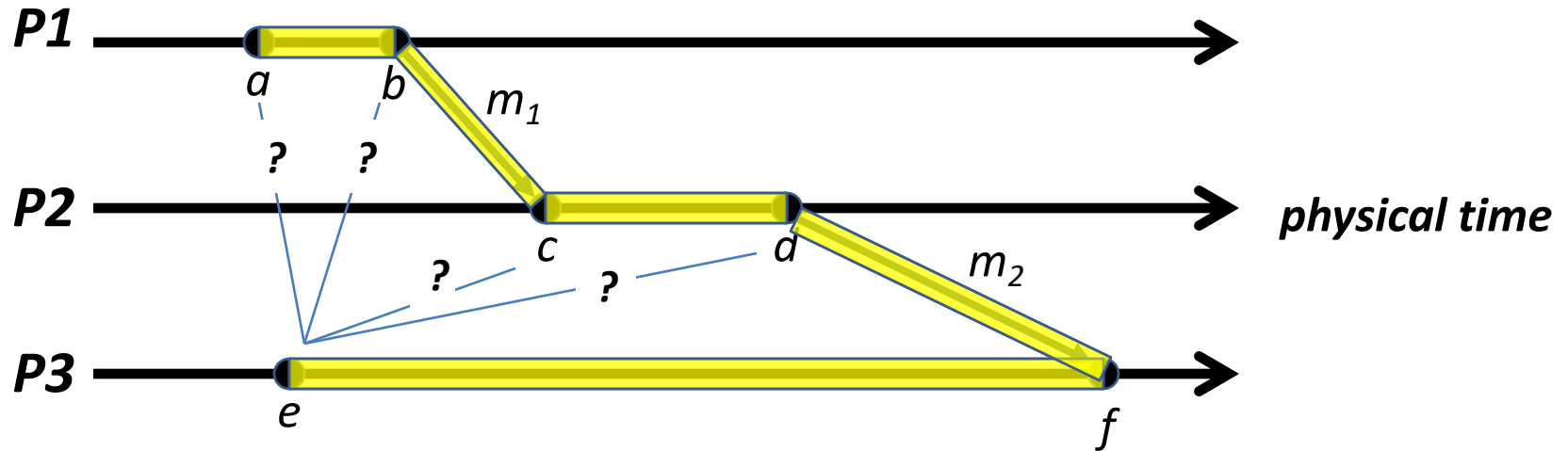
Ordering

- One use of time is to provide ordering
 - If I withdrew £100 cash at 23:59.44...
 - And the bank computes interest at 00:00.00...
 - Then interest calculation shouldn't include the £100
- But in distributed systems we can't perfectly synchronize time => cannot use this for ordering
 - Clock skew can be large, and may not be trusted
 - And over large distances, relativistic events mean that ordering depends on the observer
 - (similar effect due to finite 'speed of Internet' ;-)

The “happens-before” relation

- Often don't need to know when event ***a*** occurred
 - Just need to know if ***a*** occurred before or after ***b***
- Define the **happens-before** relation, ***a* → *b***
 - If events ***a*** and ***b*** are within the same process, then ***a* → *b*** if ***a*** occurs with an earlier local timestamp
 - Messages between processes are ordered **causally**, i.e. the event ***send(m)*** → the event ***receive(m)***
 - Transitivity: i.e. if ***a* → *b*** and ***b* → *c***, then ***a* → *c***
- Note that this only provides a partial order:
 - Possible for neither ***a* → *b*** nor ***b* → *a*** to hold
 - We say that ***a*** and ***b*** are **concurrent** and write ***a* ~ *b***

Example

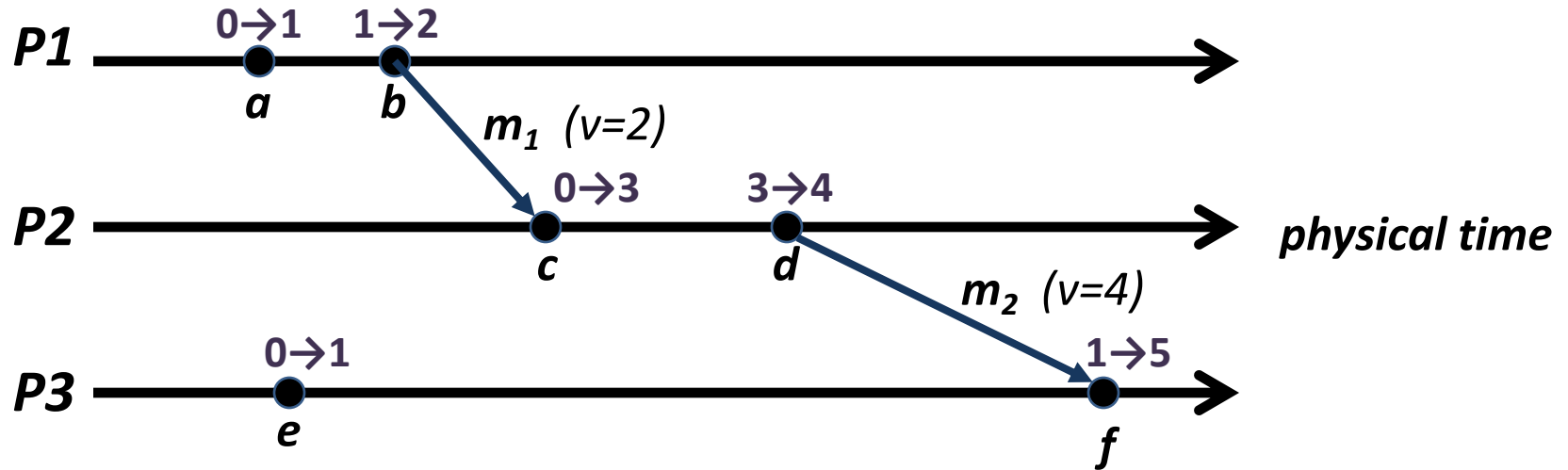


- Three processes (each with 2 events), and 2 messages
 - Due to process order, we know $a \rightarrow b$, $c \rightarrow d$ and $e \rightarrow f$
 - Causal order tells us $b \rightarrow c$ and $d \rightarrow f$
 - And by transitivity $a \rightarrow c$, $a \rightarrow d$, $a \rightarrow f$, $b \rightarrow d$, $b \rightarrow f$, $c \rightarrow f$
- However, event e is **concurrent** with a , b , c and d

Implementing Happens-Before

- One early scheme due to Lamport [1978]
 - Each process P_i has a logical clock L_i
 - L_i can simply be an integer, initialized to 0
 - L_i is incremented on every local event e
 - We write $L_i(e)$ or $L(e)$ as the timestamp of e
- **Distributed time** is implemented by propagating timestamps via messages on the network:
 - When P_i sends a message, it increments L_i and copies the value into the packet
 - When P_i receives a message from P_j , it extracts L_j and sets $L_i := \max(L_i, L_j)$, and then increments L_i
- Guarantees that if $a \rightarrow b$, then $L(a) < L(b)$
- However if $L(x) < L(y)$, this doesn't imply $x \rightarrow y$!

Lamport Clocks: Example



- When P_2 receives m_1 , it extracts timestamp 2 and sets its clock to **$\max(0, 2)$** before increment
- Possible for events to have duplicate timestamps
 - E.g., event e has the same timestamp as event a
- If desired can break ties by looking at pids, IP addresses, ...
 - This gives a **total order**, but doesn't imply happens-before!
- Why might total order without happens-before be useful?

Summary + next time (ironically)

- Clock skew and drift
- The clock synchronization problem
- Cristian's Algorithm, Berkeley Algorithm, NTP
- Logical time via the happens-before relation
- Vector clocks
- Consistent cuts
- Group communication
- Enforcing ordering vs. asynchrony
- Distributed mutual exclusion