Distributed systems

Lecture 11: Clocks, physical and logical time

Michaelmas 2019

Dr Martin Kleppmann

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

Client-server interaction: summary

- Server handles requests from client
  - Simple request/response protocols (like HTTP) useful, but lack language integration
  - RPC schemes (SunRPC, DCE RPC) address this
  - OOM schemes (CORBA, DCOM, RMI) extend RPC to understand objects, types, interfaces, exns, …
- Recent WWW developments move away from traditional RPC/RMI:
  - Avoid explicit IDLs since can slow evolution
  - Enable asynchrony, or return to request/response

Representational State Transfer (REST)

- AJAX still does RPC (just asynchronously)
- Is a procedure call / method invocation really the best way to build distributed systems?
- **Representational State Transfer** (REST) is an alternative ‘paradigm’ (or a throwback?)
  - Resources have a name: URL or URI
  - Manipulate them via POST (create), GET (select), PUT (create/overwrite), and DELETE (delete)
  - More recently added: PATCH (partial update in place)
  - Send state along with operations
- Very widely used today (Amazon, Flickr, Twitter)

Clocks and distributed time

- Distributed systems need to be able to:
  - **order events** produced by concurrent processes;
  - **synchronize** senders and receivers of messages;
  - **serialize** concurrent accesses to shared objects; and
  - generally **coordinate** joint activity
- This can be provided by some sort of **clock**:
  - **physical clocks** keep time of day
    - (must be kept consistent across multiple nodes – why?)
  - **logical clocks** keep track of event ordering
- NB. Clock in digital electronics (**oscillator**) ≠ clock in distributed systems (**source of timestamps**)

Physical clock technology

- Quartz Crystal Clocks (1929)
  - resonator shaped like a tuning fork
  - laser-trimmed to vibrate at 32,768 Hz
  - standard resonators accurate to 6ppm at 31°C... so will gain/lose around 0.5 seconds per day
  - stability better than accuracy (about 2s/month)
  - best resonators get accuracy of ~1s in 10 years
- Atomic clocks (1948)
  - count transitions of the cesium 133 atom
  - 9,192,631,770 periods defined to be 1 second
  - accuracy is better than 1 second in 6 million years...
  - relativity can’t be ignored: think satellites

Coordinated Universal Time (UTC)

- Physical clocks provide **ticks** but we want to know the actual time of day
  - determined by astronomical phenomena
- Several variants of universal time
  - **UT0**: mean solar time on Greenwich meridian
  - **UT1**: UT0 corrected for polar motion; measured via observations of quasars, laser ranging, & satellites
  - **UT2**: UT1 corrected for seasonal variations
  - **UTC**: civil time, tracked using atomic clocks, but kept within 0.9s of UT1 by occasional leap seconds

Computer clocks

- Typically have a **Real-Time Clock (RTC)**
  - CMOS clock driven by a quartz oscillator
  - battery-backed so continues when power is off
- Also have range of other clocks (PIT, ACPI, HPET, TSC, ...), mostly **higher frequency**
  - free running clocks driven by quartz oscillator
  - mapped to real time by OS at boot time
  - programmable to generate interrupts after some number of ticks (~= some amount of real time)

Operating-system use of clocks

- OSes use time for many things
  - Periodic events – e.g., time sharing, statistics, at, cron
  - Local I/O functions – e.g., peripheral timeouts; entropy
  - Network protocols – e.g., TCP DELACK, retries, keep-alive
  - Cryptographic certificate/ticket generation, expiration
  - Performance profiling and sampling features
- **Ticks** trigger interrupts
  - Historically, timers at fixed intervals (e.g., 100Hz)
  - Now, **tickless**: timer reprogrammed for next event
  - Saves energy, CPU resources – especially as cores scale up

Which of these require **physical time** vs **logical time**? What will happen to each if the real-time clock **drifts** or **steps** due to synchronization?

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 days*
= +2.47s per day
= 28.6 ppm

March 23, 2012
08:00:00

08:01:48

Skew = *108 seconds*
Drift = *108s / 34 days*
= +3.18s per day
= 36.8 ppm

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

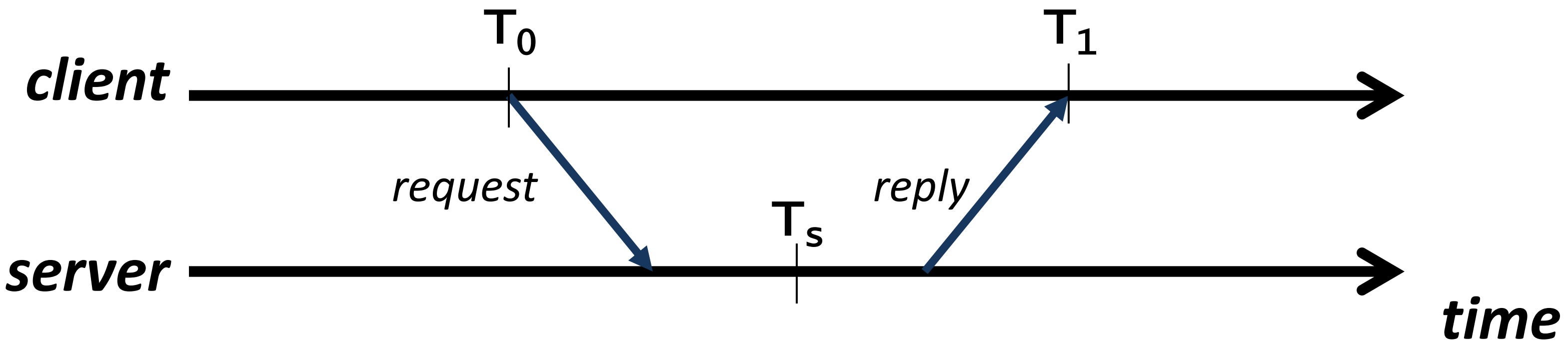
Compensation

- 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

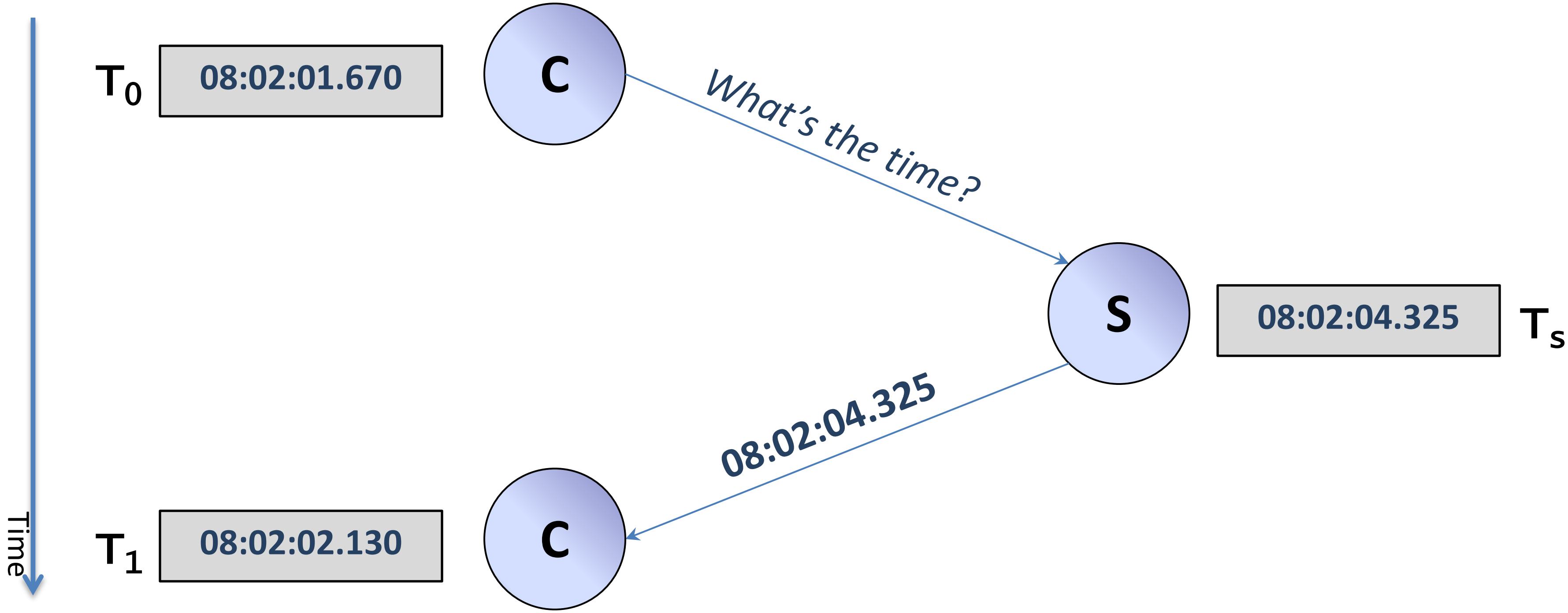
- Of course, need some way to know correct time (e.g. UTC) in order to adjust clock!
  - could attach a GPS receiver (or atomic clock) to computer, and get ±0.1ms accuracy…
  - …but too expensive/clunky for general use
  - (RF in server rooms and data centres non-ideal)
- Instead can ask some machine 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...)

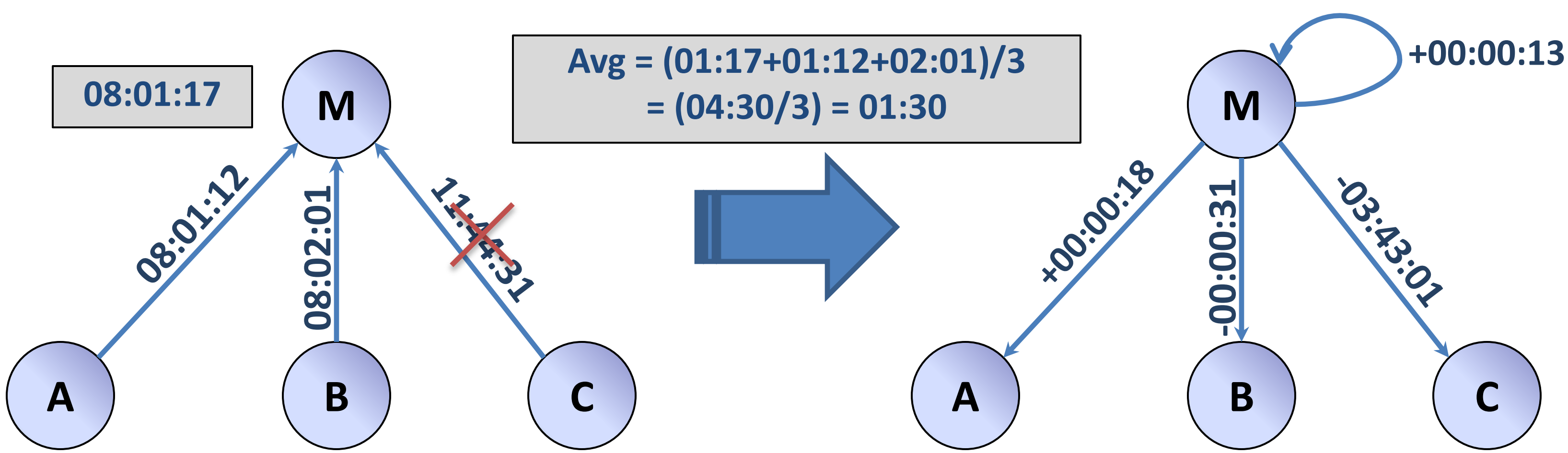
Cristian's Algorithm: Example



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

Berkeley Algorithm (1989)

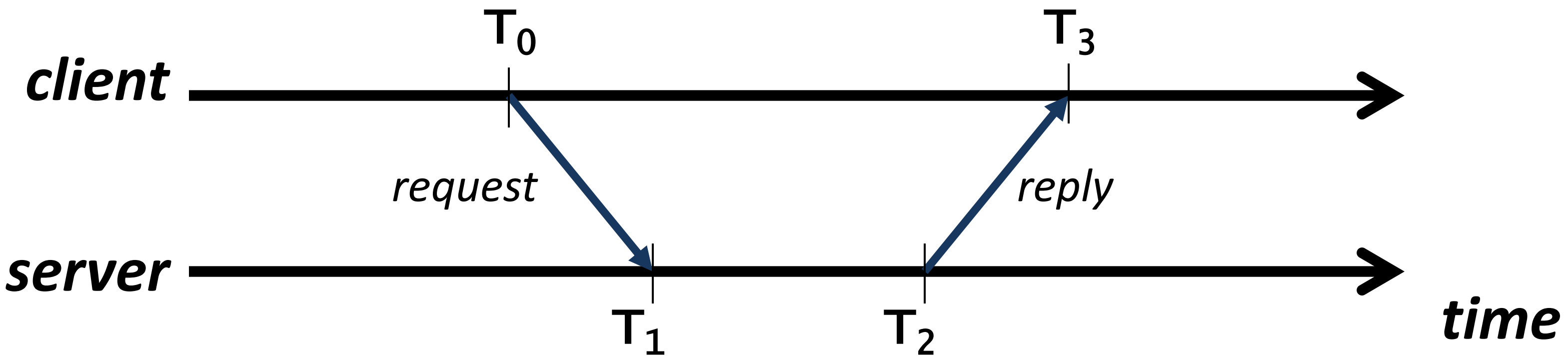
- Don't assume have an accurate time server
- Try to synchronize a set of clocks to the average
  - One machine, **M**, is designated the master
  - **M** periodically polls all other machines for their time
  - (can use Cristian's technique to account for delays)
  - Master computes average (including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

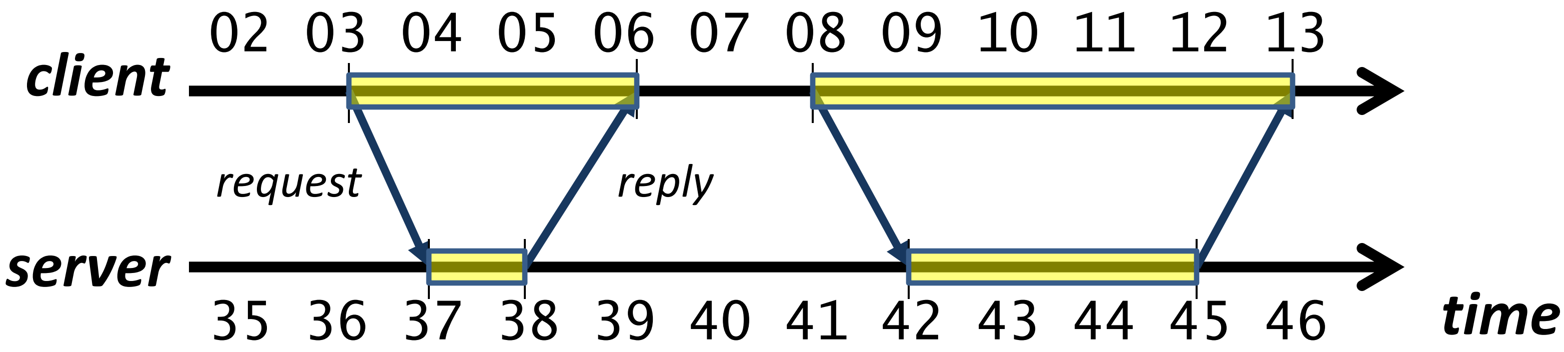


- UDP/IP messages with slots for four timestamps
  - systems insert timestamps at earliest/latest opportunity
- Client computes:
  - Offset **O** = $((T_1 - T_0) + (T_2 - T_3)) / 2$
  - Delay **D** = $(T_3 - T_0) - (T_2 - T_1)$
- Relies on symmetric messaging delays to be correct (but now excludes variable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

Measured difference in average timestamps: (T1+T2)/2 – (T0+T3)/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 ;-(

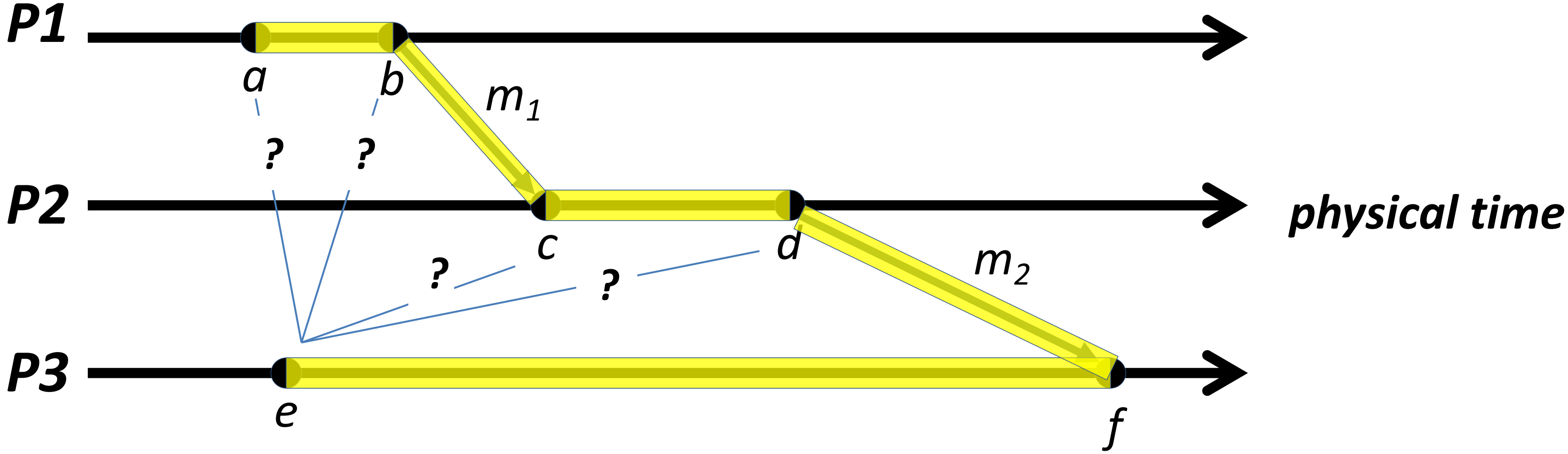
Ordering

- 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
  - Message sent at T = 2.0 s (according to sender clock) may be received at T = 1.9 s (according to recipient)

The "happens-before" relation

- Often don't need to know <u>when</u> event ***a*** occurred
  - Just need to know if ***a*** occurred before or after ***b***
- Define the **happens-before** relation, ***a*** $\rightarrow$ ***b***
  - If events ***a*** and ***b*** are within the same process, then ***a*** $\rightarrow$ ***b*** if ***a*** occurs with an earlier local timestamp
  - Messages between processes are ordered ***causally***, i.e. the event ***send(m)*** $\rightarrow$ the event ***receive(m)***
  - Transitivity: i.e. if ***a*** $\rightarrow$ ***b*** and ***b*** $\rightarrow$ ***c***, then ***a*** $\rightarrow$ ***c***
- Note that this only provides a partial order:
  - Possible for neither ***a*** $\rightarrow$ ***b*** nor ***b*** $\rightarrow$ ***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*→*b*, *c*→*d*** and ***e*→*f***
  - Causal order tells us ***b*→*c*** and ***d*→*f***
  - And by transitivity ***a*→*c*, *a*→*d*, *a*→*f*, *b*→*d*, *b*→*f*, *c*→*f***
- However, event *e* is **concurrent** with ***a***, ***b***, ***c*** and ***d***

Summary + next time

- 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