Distributed systems

Lecture 4: Clock synchronization; logical clocks

Dr Robert N. M. Watson

1

Last time

- 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 (such as UTC))
- The process of gaining/losing time is **clock drift**
- The difference between two clocks is called **clock skew**
- **Clock synchronization** aims to minimize clock skew between two (or a set of) different clocks

2

From last lecture

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

3

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

4

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**

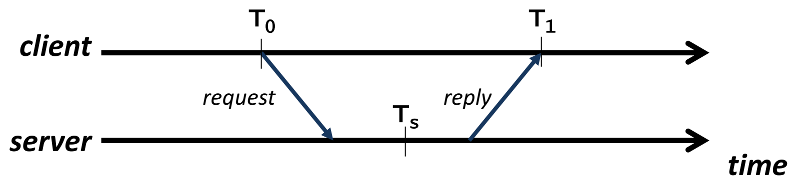
5

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 ±1ms (or ±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?

6

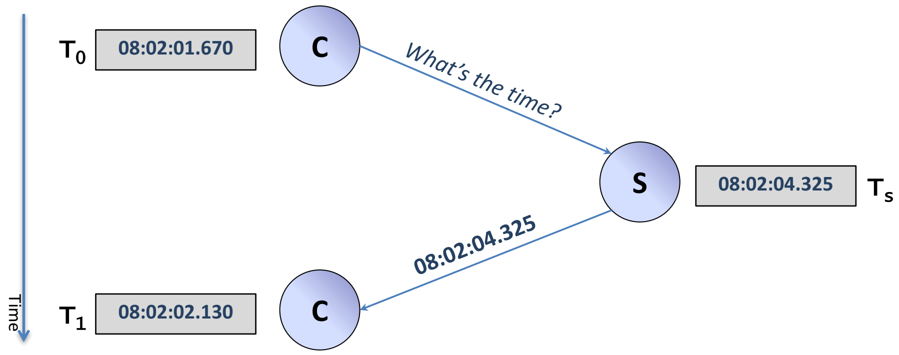
Cristian's Algorithm (1989)



- 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...)

7

Cristian's Algorithm: Example

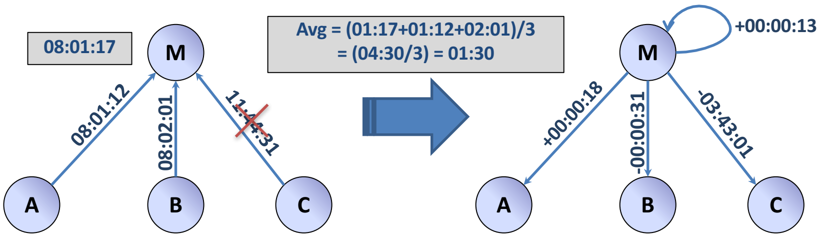


- RTT = 460ms, so one way delay is [approx] 230ms.
- Estimate correct time as (08:02:04.325 + 230ms) = 08:02:04.555
- Client gradually adjusts local clock to gain 2.425 seconds

8

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



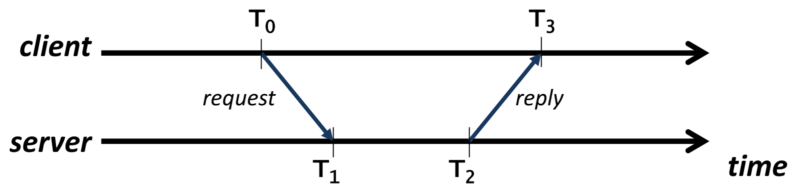
9

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'

10

NTP algorithm

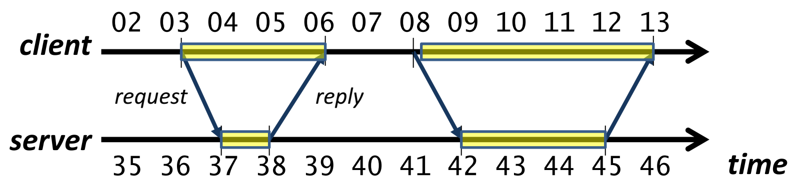


- UDP/IP messages with slots for four timestamps
  - systems insert timestamps at earliest/latest opportunity
- Client computes:
  - Offset **O** = $((T_1 - T_0) + (T_2 - T_3)) / 2$
  - Delay **D** = $(T_3 - T_0) - (T_2 - T_1)$
- Relies on symmetric messaging delays to be correct (but now excludes variable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

15

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' ;-)

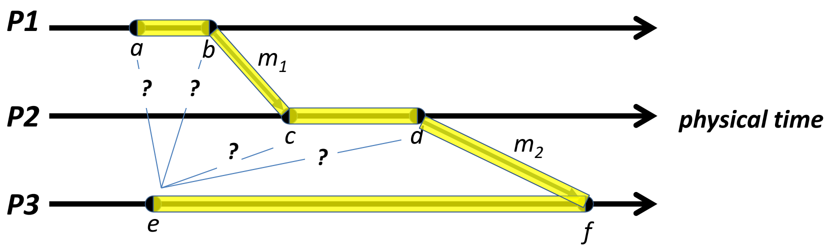
16

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* → *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***

17

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***

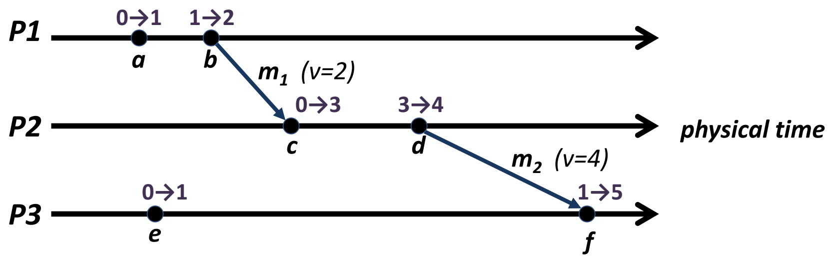
18

Implementing Happens-Before

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

19

Lamport Clocks: Example



- When **P2** receives $\mathbf{m_1}$, 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!

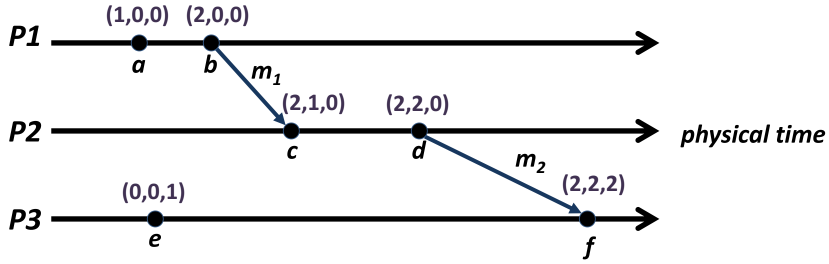
20

Vector clocks

- With Lamport clocks, given **L(*a*)** and **L(*b*)**, we can't tell if ***a→b*** or ***b→a*** or ***a ~ b***
- One solution is **vector clocks**:
  - An ordered list of logical clocks, one per-process
  - Each process $\mathbf{P_i}$ maintains $\mathbf{V_i[]}$, initially all zeroes
  - On a local event ***e***, $\mathbf{P_i}$ increments $\mathbf{V_i[i]}$
    - If the event is message send, new $\mathbf{V_i[]}$ copied into packet
  - If $\mathbf{P_i}$ receives a message from $\mathbf{P_j}$ then, for all **k** = 0, 1, …, it sets $\mathbf{V_i[k] := max(V_j[k], V_i[k])}$, and increments $\mathbf{V_i[i]}$
- Intuitively $\mathbf{V_i[k]}$ captures the number of events at process $\mathbf{P_k}$ that have been observed by $\mathbf{P_i}$

21

Vector clocks: example



- When **P2** receives $\mathbf{m_1}$, it **merges** the entries from **P1**'s clock
  - choose the maximum value in each position
- Similarly when **P3** receives $\mathbf{m_2}$, it merges in **P2**'s clock
  - this incorporates the changes from **P1** that **P2** already saw
- Vector clocks ***explicitly track the transitive causal order***: ***f***'s timestamp captures the history of ***a***, ***b***, ***c*** & ***d***

22

Using vector clocks for ordering

- Can compare vector clocks piecewise:
  - $V_i = V_j$ iff $V_i[k] = V_j[k]$ for $k = 0, 1, 2, \ldots$
  - $V_i \leq V_j$ iff $V_i[k] \leq V_j[k]$ for $k = 0, 1, 2, \ldots$
  - $V_i < V_j$ iff $V_i \leq V_j$ and $V_i \neq V_j$
  - $V_i \sim V_j$ otherwise

e.g. [2,0,0] versus [0,0,1]

- For any two event timestamps **T(a)** and **T(b)**
  - if $a \rightarrow b$ then $T(a) < T(b)$; **and**
  - if $T(a) < T(b)$ then $a \rightarrow b$
- Hence can use timestamps to determine if there is a causal ordering between any two events
  - i.e. determine whether $a \rightarrow b$, $b \rightarrow a$, or $a \sim b$

Does this seem familiar? Recall **Time-Stamp Ordering** and **Optimistic Concurrency Control** for transactions last term.

23

Summary + next time (ironically)

- The clock synchronization problem
- Cristian's Algorithm, Berkeley Algorithm, NTP
- Logical time via the happens-before relation
- Vector clocks

- More on vector clocks
- Consistent cuts
- Group communication
- Enforcing ordering vs. asynchrony
- Distributed mutual exclusion

24