Distributed Systems

The second half of *Concurrent and Distributed Systems*

https://www.cl.cam.ac.uk/teaching/current/ConcDisSys

Dr. Martin Kleppmann (mk428@cam)

University of Cambridge

Computer Science Tripos, Part IB

Lecture 3

Time, clocks, and ordering of events

A detective story

In the night from 30 June to 1 July 2012 (UK time), many online services and systems around the world crashed simultaneously.

Servers locked up and stopped responding.

Some airlines could not process any reservations or check-ins for several hours.

What happened?

Clocks and time in distributed systems

Distributed systems often need to measure time, e.g.:

- Schedulers, timeouts, failure detectors, retry timers

Clocks and time in distributed systems

Distributed systems often need to measure time, e.g.:

- Schedulers, timeouts, failure detectors, retry timers
- Performance measurements, statistics, profiling

Clocks and time in distributed systems

Distributed systems often need to measure time, e.g.:

- Schedulers, timeouts, failure detectors, retry timers
- Performance measurements, statistics, profiling
- Log files & databases: record when an event occurred

Clocks and time in distributed systems

Distributed systems often need to measure time, e.g.:

- Schedulers, timeouts, failure detectors, retry timers
- Performance measurements, statistics, profiling
- Log files & databases: record when an event occurred
- Data with time-limited validity (e.g. cache entries)

Clocks and time in distributed systems

Distributed systems often need to measure time, e.g.:

- Schedulers, timeouts, failure detectors, retry timers
- Performance measurements, statistics, profiling
- Log files & databases: record when an event occurred
- Data with time-limited validity (e.g. cache entries)
- Determining order of events across several nodes

Clocks and time in distributed systems

Distributed systems often need to measure time, e.g.:

- Schedulers, timeouts, failure detectors, retry timers
- Performance measurements, statistics, profiling
- Log files & databases: record when an event occurred
- Data with time-limited validity (e.g. cache entries)
- Determining order of events across several nodes

We distinguish two types of clock:

- **physical clocks**: count number of seconds elapsed
- **logical clocks**: count events, e.g. messages sent

Clocks and time in distributed systems

Distributed systems often need to measure time, e.g.:

- Schedulers, timeouts, failure detectors, retry timers
- Performance measurements, statistics, profiling
- Log files & databases: record when an event occurred
- Data with time-limited validity (e.g. cache entries)
- Determining order of events across several nodes

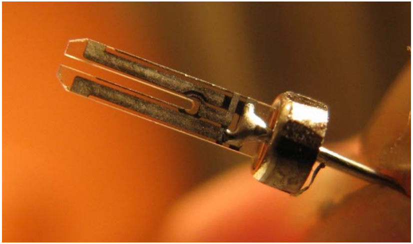
We distinguish two types of clock:

- **physical clocks**: count number of seconds elapsed
- **logical clocks**: count events, e.g. messages sent

NB. Clock in digital electronics (oscillator)
$\neq$ clock in distributed systems (source of **timestamps**)

Quartz clocks

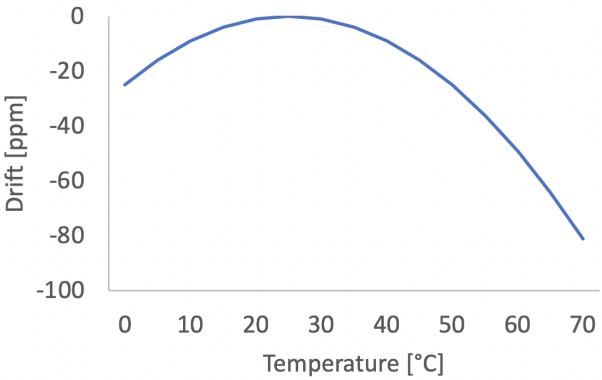
- Quartz crystal laser-trimmed to mechanically resonate at a specific frequency
- Piezoelectric effect: mechanical force $\Leftrightarrow$ electric field
- Oscillator circuit produces signal at resonant frequency
- Count number of cycles to measure elapsed time



Quartz clock error: drift

- One clock runs slightly fast, another slightly slow
- Drift measured in **parts per million** (ppm)
- 1 ppm = 1 microsecond/second = 86 ms/day = 32 s/year
- Most computer clocks correct within ≈ 50 ppm

Temperature significantly affects drift



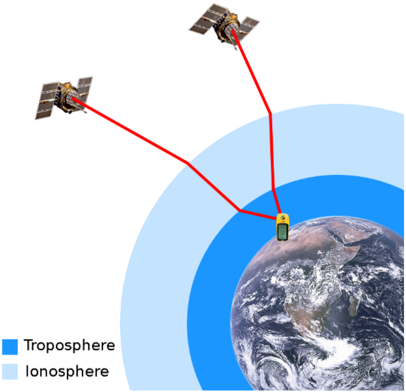
Atomic clocks

- Caesium-133 has a resonance ("hyperfine transition") at ≈ 9 GHz
- Tune an electronic oscillator to that resonant frequency
- 1 second $=$ 9,192,631,770 periods of that signal
- Accuracy ≈ 1 in 10^{-14} (1 second in 3 million years)
- Price $\approx$ £20,000 (?) (can get cheaper rubidium clocks for $\approx$ £1,000)

https://www.microsemi.com/product-directory/cesium-frequency-references/4115-5071a-cesium-primary-frequency-standard

GPS as time source

- 31 satellites, each carrying an atomic clock
- satellite broadcasts current time and location
- calculate position from speed-of-light delay between satellite and receiver
- corrections for atmospheric effects, relativity, etc.
- in datacenters, need antenna on the roof



https://commons.wikimedia.org/wiki/File:Gps-atmospheric-efects.png

Coordinated Universal Time (UTC)

Greenwich Mean Time (GMT, solar time): it's noon when the sun is in the south, as seen from the Greenwich meridian

Coordinated Universal Time (UTC)

Greenwich Mean Time (GMT, solar time): it's noon when the sun is in the south, as seen from the Greenwich meridian

International Atomic Time (TAI): 1 day is $24 \times 60 \times 60 \times$ 9,192,631,770 periods of caesium-133's resonant frequency

Coordinated Universal Time (UTC)

Greenwich Mean Time (GMT, solar time): it's noon when the sun is in the south, as seen from the Greenwich meridian

International Atomic Time (TAI): 1 day is $24 \times 60 \times 60 \times$ 9,192,631,770 periods of caesium-133's resonant frequency

Problem: speed of Earth's rotation is not constant

Coordinated Universal Time (UTC)

Greenwich Mean Time (GMT, solar time): it's noon when the sun is in the south, as seen from the Greenwich meridian

International Atomic Time (TAI): 1 day is $24 \times 60 \times 60 \times$ 9,192,631,770 periods of caesium-133's resonant frequency

Problem: speed of Earth's rotation is not constant

Compromise: UTC is TAI with corrections to account for Earth rotation

Coordinated Universal Time (UTC)

Greenwich Mean Time (GMT, solar time): it's noon when the sun is in the south, as seen from the Greenwich meridian

International Atomic Time (TAI): 1 day is $24 \times 60 \times 60 \times$ 9,192,631,770 periods of caesium-133's resonant frequency

Problem: speed of Earth's rotation is not constant

Compromise: UTC is TAI with corrections to account for Earth rotation

Time zones and **daylight savings time** are offsets to UTC

Leap seconds

Every year, on 30 June and 31 December at 23:59:59 UTC, one of three things happens:

- The clock immediately jumps forward to 00:00:00, skipping one second (**negative leap second**)
- The clock moves to 00:00:00 after one second, as usual
- The clock moves to 23:59:60 after one second, and then moves to 00:00:00 after one further second (**positive leap second**)

This is announced several months beforehand.

http://leapsecond.com/notes/leap-watch.htm

How computers represent timestamps

Two most common representations:

- **Unix time**: number of seconds since 1 January 1970 00:00:00 UTC (the "epoch"), *not counting leap seconds*
- **ISO 8601**: year, month, day, hour, minute, second, and timezone offset relative to UTC
  example: `2020-11-09T09:50:17+00:00`

How computers represent timestamps

Two most common representations:

- **Unix time**: number of seconds since 1 January 1970 00:00:00 UTC (the "epoch"), *not counting leap seconds*
- **ISO 8601**: year, month, day, hour, minute, second, and timezone offset relative to UTC
  example: `2020-11-09T09:50:17+00:00`

Conversion between the two requires:

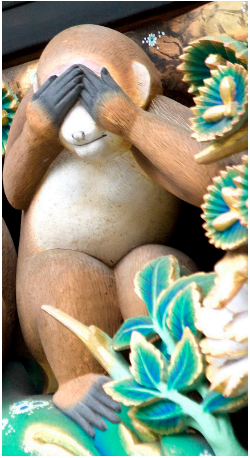
- Gregorian calendar: 365 days in a year, except leap years

  ```
  (year % 4 == 0 && (year % 100 != 0 ||
                     year % 400 == 0))
  ```

- Knowledge of past and future leap seconds...?!

How most software deals with leap seconds

By ignoring them!

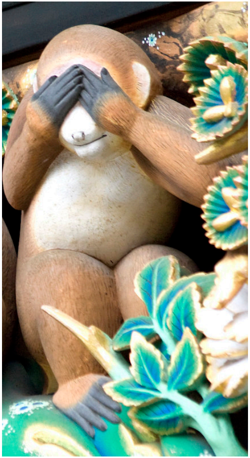


https://www.flickr.com/photos/ru_boff/37915499055/

How most software deals with leap seconds

By ignoring them!

However, OS and DistSys often need timings with sub-second accuracy.



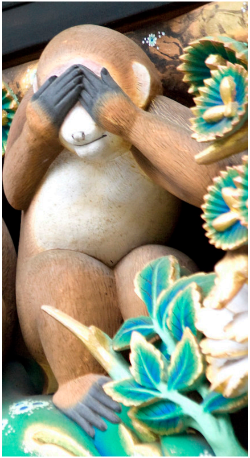
https://www.flickr.com/photos/ru_boff/37915499055/

How most software deals with leap seconds

By ignoring them!

However, OS and DistSys often need timings with sub-second accuracy.

30 June 2012: bug in Linux kernel caused livelock on leap second, causing many Internet services to go down



https://www.flickr.com/photos/ru_boff/37915499055/

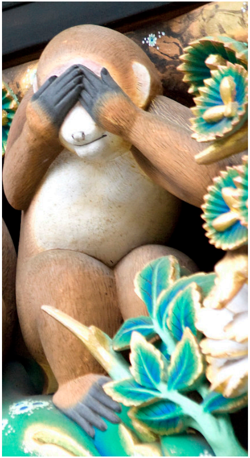
How most software deals with leap seconds

By ignoring them!

However, OS and DistSys often need timings with sub-second accuracy.

30 June 2012: bug in Linux kernel caused livelock on leap second, causing many Internet services to go down

Pragmatic solution: “**smear**” (spread out) the leap second over the course of a day



https://www.flickr.com/photos/ru_boff/37915499055/

Clock synchronisation

Computers track physical time/UTC with a quartz clock (with battery, continues running when power is off)

Due to **clock drift**, clock error gradually increases

Clock synchronisation

Computers track physical time/UTC with a quartz clock
(with battery, continues running when power is off)

Due to **clock drift**, clock error gradually increases

Clock skew: difference between two clocks at a point in time

Clock synchronisation

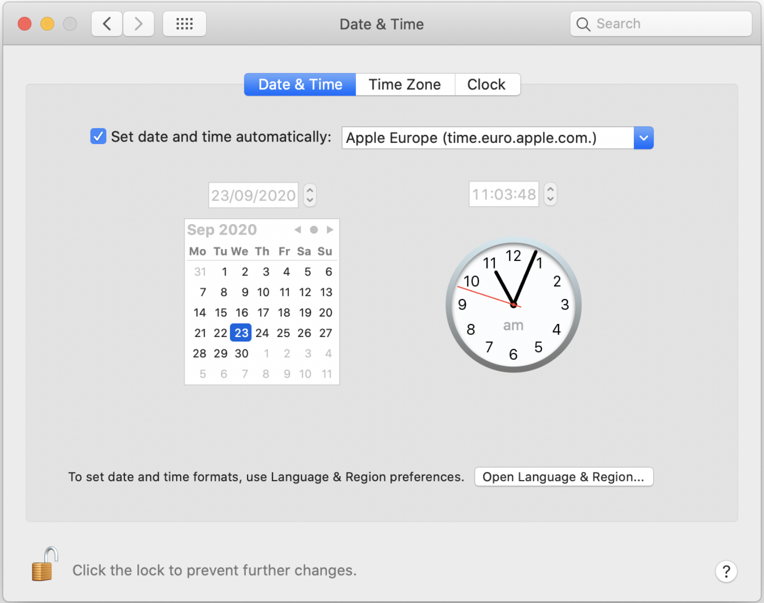
Computers track physical time/UTC with a quartz clock (with battery, continues running when power is off)

Due to **clock drift**, clock error gradually increases

Clock skew: difference between two clocks at a point in time

Solution: Periodically get the current time from a server that has a more accurate time source (atomic clock or GPS receiver)

Protocols: Network Time Protocol (**NTP**),
Precision Time Protocol (**PTP**)


Date & Time
Search
Date & Time
Time Zone
Clock
Set date and time automatically:
Apple Europe (time.euro.apple.com.)
23/09/2020
11:03:48
Sep 2020
Mo Tu We Th Fr Sa Su
31 1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 1 2 3 4
5 6 7 8 9 10 11
12 1 2 3 4 5 6 7 8 9 10 11
am
To set date and time formats, use Language & Region preferences.
Open Language & Region...
Click the lock to prevent further changes.
?

Network Time Protocol (NTP)

Many operating system vendors run NTP servers, configure OS to use them by default

Network Time Protocol (NTP)

Many operating system vendors run NTP servers, configure OS to use them by default

Hierarchy of clock servers arranged into **strata**:

- Stratum 0: atomic clock or GPS receiver
- Stratum 1: synced directly with stratum 0 device
- Stratum 2: servers that sync with stratum 1, etc.

Network Time Protocol (NTP)

Many operating system vendors run NTP servers, configure OS to use them by default

Hierarchy of clock servers arranged into **strata**:

- Stratum 0: atomic clock or GPS receiver
- Stratum 1: synced directly with stratum 0 device
- Stratum 2: servers that sync with stratum 1, etc.

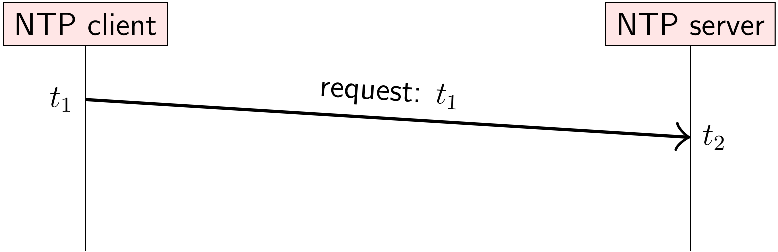
May contact multiple servers, discard outliers, average rest

Makes multiple requests to the same server, use statistics to reduce random error due to variations in network latency

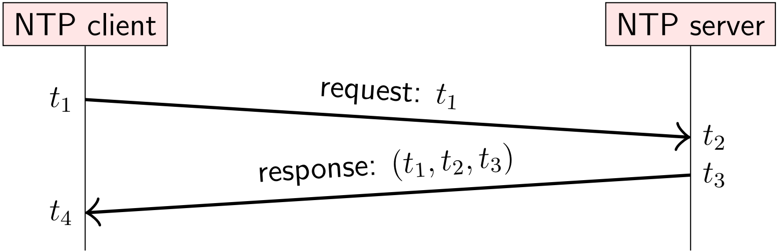
Reduces clock skew to a few milliseconds in good network conditions, but can be much worse!

Estimating time over a network

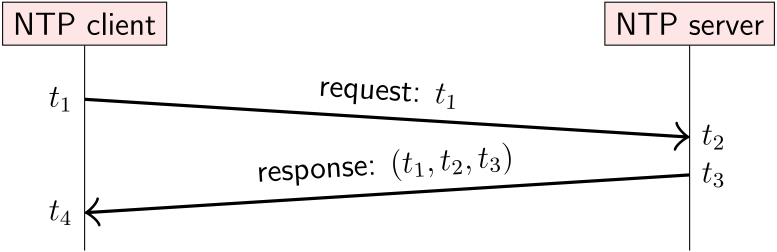
Estimating time over a network



Estimating time over a network

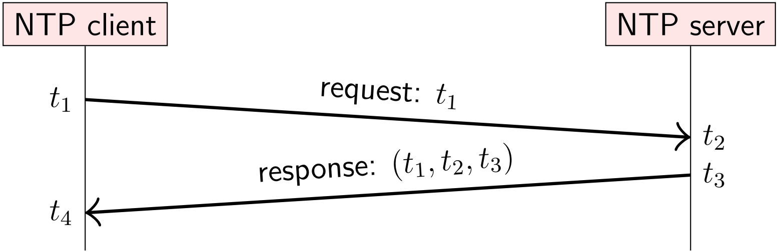


Estimating time over a network



Round-trip network delay: $\delta = (t_4 - t_1) - (t_3 - t_2)$

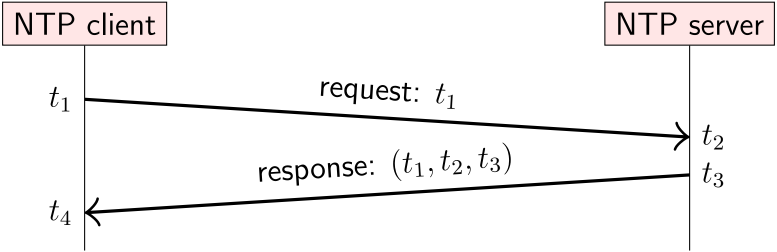
Estimating time over a network



Round-trip network delay: $\delta = (t_4 - t_1) - (t_3 - t_2)$

Estimated server time when client receives response: $t_3 + \dfrac{\delta}{2}$

Estimating time over a network



Round-trip network delay: $\delta = (t_4 - t_1) - (t_3 - t_2)$

Estimated server time when client receives response: $t_3 + \frac{\delta}{2}$

Estimated clock skew: $\theta = t_3 + \frac{\delta}{2} - t_4 = \frac{t_2 - t_1 + t_3 - t_4}{2}$

Correcting clock skew

Once the client has estimated the clock skew θ, it needs to apply that correction to its clock.

- If $|\theta| < 125$ ms, **slew** the clock:
  slightly speed it up or slow it down by up to 500 ppm
  (brings clocks in sync within $\approx$ 5 minutes)

Correcting clock skew

Once the client has estimated the clock skew θ, it needs to apply that correction to its clock.

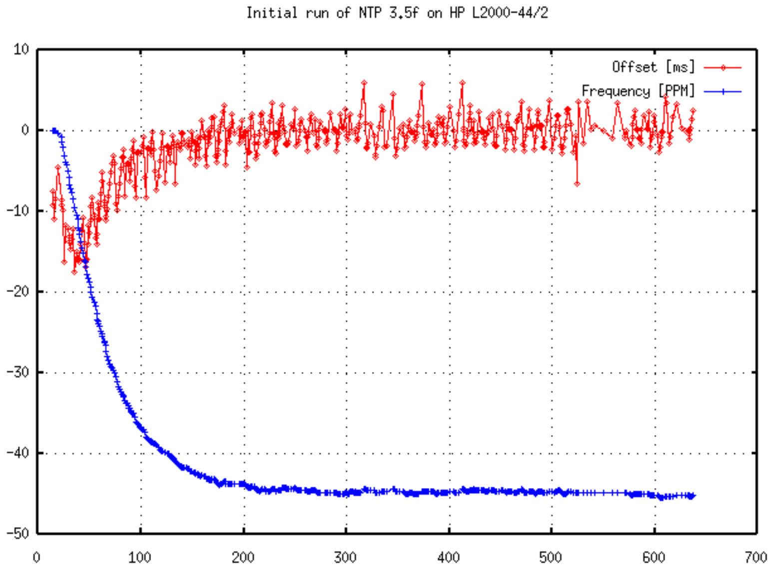
- If $|\theta| < 125$ ms, **slew** the clock:
  slightly speed it up or slow it down by up to 500 ppm
  (brings clocks in sync within $\approx$ 5 minutes)
- If 125 ms $\leq |\theta| <$ 1,000 s, **step** the clock:
  suddenly reset client clock to estimated server timestamp

Correcting clock skew

Once the client has estimated the clock skew θ, it needs to apply that correction to its clock.

- If $|\theta| < 125$ ms, **slew** the clock:
  slightly speed it up or slow it down by up to 500 ppm
  (brings clocks in sync within $\approx$ 5 minutes)
- If 125 ms $\leq |\theta| <$ 1,000 s, **step** the clock:
  suddenly reset client clock to estimated server timestamp
- If $|\theta| \geq$ 1,000 s, **panic** and do nothing
  (leave the problem for a human operator to resolve)

Systems that rely on clock sync need to monitor clock skew!



http://www.ntp.org/ntpfaq/NTP-s-algo.htm

Monotonic and time-of-day clocks

```
// BAD:
long startTime = System.currentTimeMillis();
doSomething();
long endTime = System.currentTimeMillis();
long elapsedMillis = endTime - startTime;
// elapsedMillis may be negative!
```

Monotonic and time-of-day clocks

```
// BAD:
long startTime = System.currentTimeMillis();
doSomething();
long endTime = System.currentTimeMillis();
long elapsedMillis = endTime - startTime;
// elapsedMillis may be negative!
```

NTP client steps the clock during this

Monotonic and time-of-day clocks

```
// BAD:
long startTime = System.currentTimeMillis();
doSomething();
long endTime = System.currentTimeMillis();
long elapsedMillis = endTime - startTime;
// elapsedMillis may be negative!
```

NTP client steps the clock during this

```
// GOOD:
long startTime = System.nanoTime();
doSomething();
long endTime = System.nanoTime();
long elapsedNanos = endTime - startTime;
// elapsedNanos is always >= 0
```

Monotonic and time-of-day clocks

Time-of-day clock:

- Time since a fixed date (e.g. 1 January 1970 epoch)

Monotonic clock:

- Time since arbitrary point (e.g. when machine booted up)

Monotonic and time-of-day clocks

Time-of-day clock:

- Time since a fixed date (e.g. 1 January 1970 epoch)
- May suddenly move forwards or backwards (NTP stepping), subject to leap second adjustments

Monotonic clock:

- Time since arbitrary point (e.g. when machine booted up)
- Always moves forwards at near-constant rate

Monotonic and time-of-day clocks

Time-of-day clock:

- Time since a fixed date (e.g. 1 January 1970 epoch)
- May suddenly move forwards or backwards (NTP stepping), subject to leap second adjustments
- Timestamps can be compared across nodes (if synced)

Monotonic clock:

- Time since arbitrary point (e.g. when machine booted up)
- Always moves forwards at near-constant rate
- Good for measuring elapsed time on a single node

Monotonic and time-of-day clocks

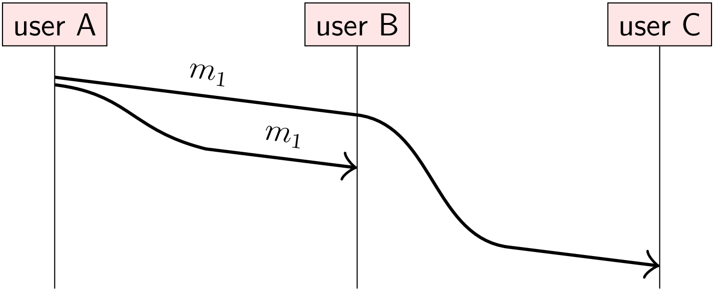
Time-of-day clock:

- Time since a fixed date (e.g. 1 January 1970 epoch)
- May suddenly move forwards or backwards (NTP stepping), subject to leap second adjustments
- Timestamps can be compared across nodes (if synced)
- Java: `System.currentTimeMillis()`
- Linux: `clock_gettime(CLOCK_REALTIME)`

Monotonic clock:

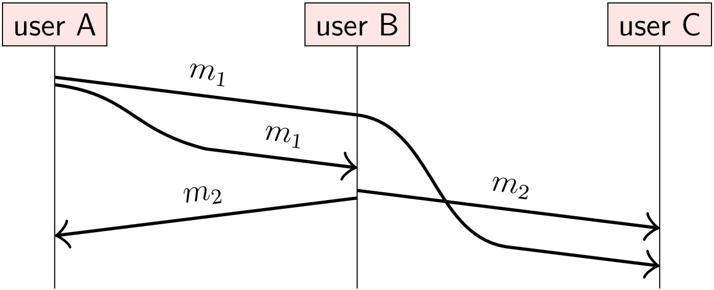
- Time since arbitrary point (e.g. when machine booted up)
- Always moves forwards at near-constant rate
- Good for measuring elapsed time on a single node
- Java: `System.nanoTime()`
- Linux: `clock_gettime(CLOCK_MONOTONIC)`

Ordering of messages



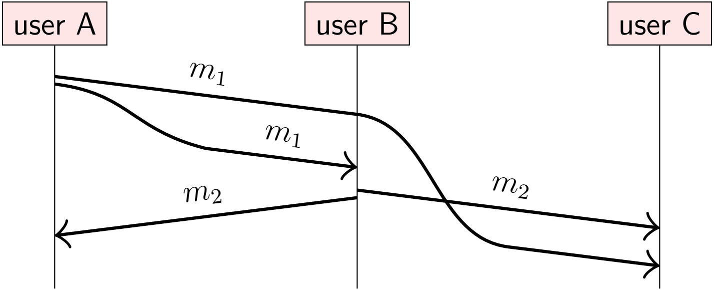
$m_1 =$ "A says: The moon is made of cheese!"

Ordering of messages



$m_1 =$ "A says: The moon is made of cheese!"
$m_2 =$ "B says: Oh no it isn't!"

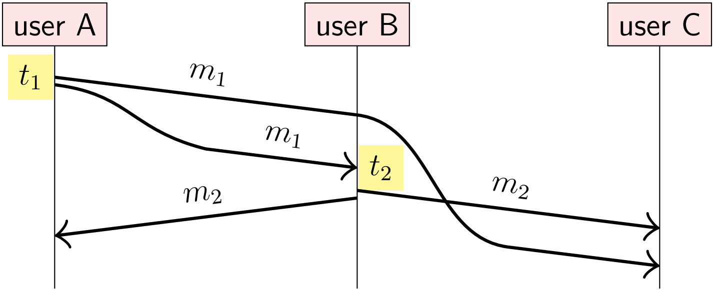
Ordering of messages



m_1 = "A says: The moon is made of cheese!"
m_2 = "B says: Oh no it isn't!"

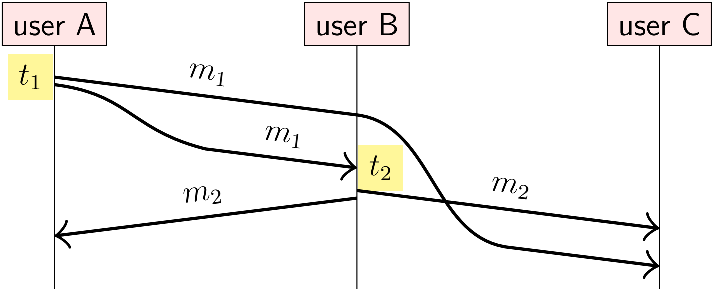
C sees m_2 first, m_1 second,
even though logically m_1 **happened before** m_2.

Ordering of messages using timestamps?



$m_1 = (t_1,$ "A says: The moon is made of cheese!"$)$
$m_2 = (t_2,$ "B says: Oh no it isn't!"$)$

Ordering of messages using timestamps?



$m_1 = (t_1,$ "A says: The moon is made of cheese!"$)$
$m_2 = (t_2,$ "B says: Oh no it isn't!"$)$

Problem: even with synced clocks, $t_2 < t_1$ is possible.
Timestamp order is inconsistent with expected order!

The happens-before relation

An **event** is something happening at one node (sending or receiving a message, or a local execution step).

We say event a **happens before** event b (written $a \rightarrow b$) iff:

The happens-before relation

An **event** is something happening at one node (sending or receiving a message, or a local execution step).

We say event a **happens before** event b (written $a \rightarrow b$) iff:

- a and b occurred at the same node, and a occurred before b in that node's local execution order; or

The happens-before relation

An **event** is something happening at one node (sending or receiving a message, or a local execution step).

We say event a **happens before** event b (written $a \to b$) iff:

- a and b occurred at the same node, and a occurred before b in that node's local execution order; or
- event a is the sending of some message m, and event b is the receipt of that same message m (assuming sent messages are unique); or

The happens-before relation

An **event** is something happening at one node (sending or receiving a message, or a local execution step).

We say event a **happens before** event b (written $a \rightarrow b$) iff:

- a and b occurred at the same node, and a occurred before b in that node's local execution order; or
- event a is the sending of some message m, and event b is the receipt of that same message m (assuming sent messages are unique); or
- there exists an event c such that $a \rightarrow c$ and $c \rightarrow b$.

The happens-before relation

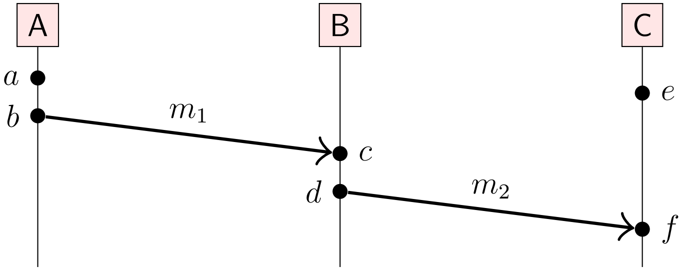
An **event** is something happening at one node (sending or receiving a message, or a local execution step).

We say event a **happens before** event b (written $a \rightarrow b$) iff:

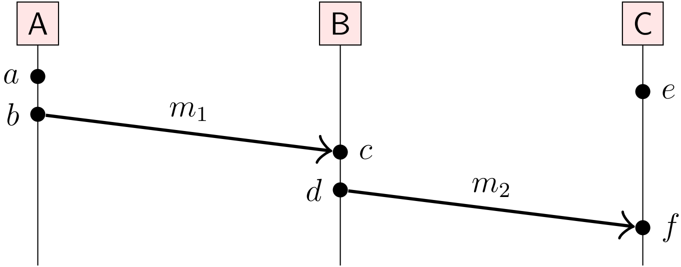
- a and b occurred at the same node, and a occurred before b in that node's local execution order; or
- event a is the sending of some message m, and event b is the receipt of that same message m (assuming sent messages are unique); or
- there exists an event c such that $a \rightarrow c$ and $c \rightarrow b$.

The happens-before relation is a partial order: it is possible that neither $a \rightarrow b$ nor $b \rightarrow a$. In that case, a and b are **concurrent** (written $a \parallel b$).

Happens-before relation example

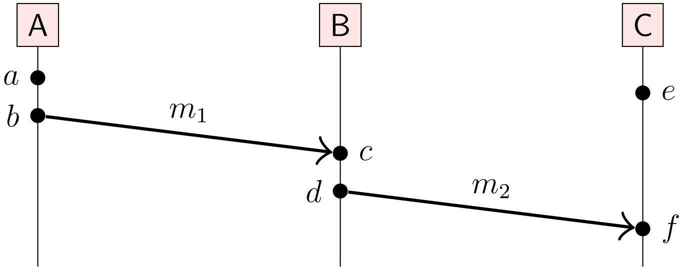


Happens-before relation example



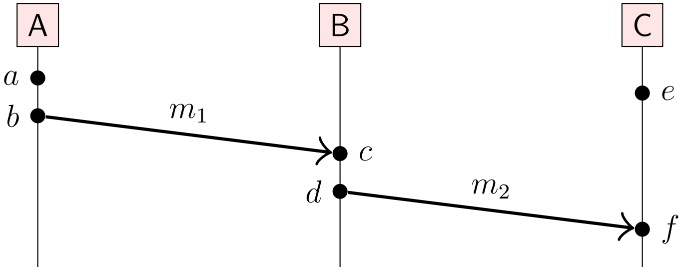
- $a \to b$, $c \to d$, and $e \to f$ due to node execution order

Happens-before relation example



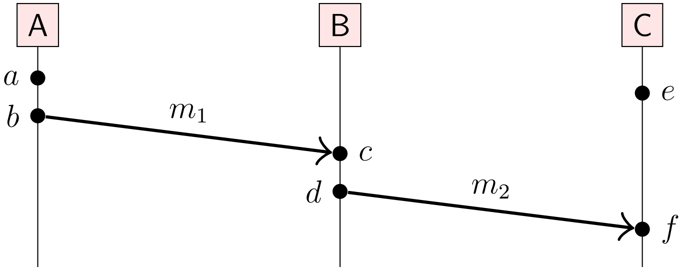
- $a \to b$, $c \to d$, and $e \to f$ due to node execution order
- $b \to c$ and $d \to f$ due to messages m_1 and m_2

Happens-before relation example



- $a \to b$, $c \to d$, and $e \to f$ due to node execution order
- $b \to c$ and $d \to f$ due to messages m_1 and m_2
- $a \to c$, $a \to d$, $a \to f$, $b \to d$, $b \to f$, and $c \to f$ due to transitivity

Happens-before relation example



- $a \to b$, $c \to d$, and $e \to f$ due to node execution order
- $b \to c$ and $d \to f$ due to messages m_1 and m_2
- $a \to c$, $a \to d$, $a \to f$, $b \to d$, $b \to f$, and $c \to f$ due to transitivity
- $a \parallel e$, $b \parallel e$, $c \parallel e$, and $d \parallel e$

Causality

Taken from physics (relativity).

- When $a \rightarrow b$, then a **might have caused** b.
- When $a \parallel b$, we know that a **cannot have caused** b.

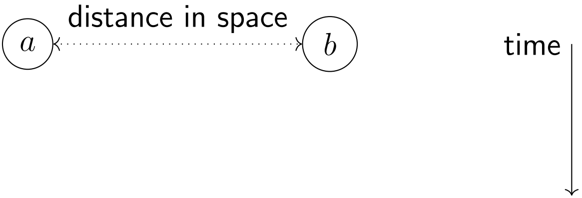
Happens-before relation encodes **potential causality**.

Causality

Taken from physics (relativity).

- When $a \rightarrow b$, then a **might have caused** b.
- When $a \parallel b$, we know that a **cannot have caused** b.

Happens-before relation encodes **potential causality**.

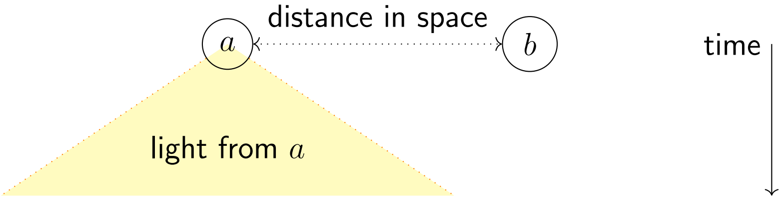


Causality

Taken from physics (relativity).

- When $a \to b$, then a **might have caused** b.
- When $a \parallel b$, we know that a **cannot have caused** b.

Happens-before relation encodes **potential causality**.

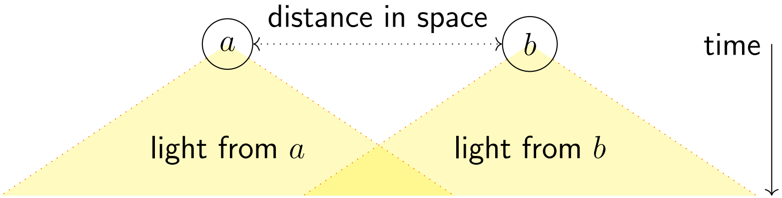


Causality

Taken from physics (relativity).

- When $a \rightarrow b$, then a **might have caused** b.
- When $a \parallel b$, we know that a **cannot have caused** b.

Happens-before relation encodes **potential causality**.

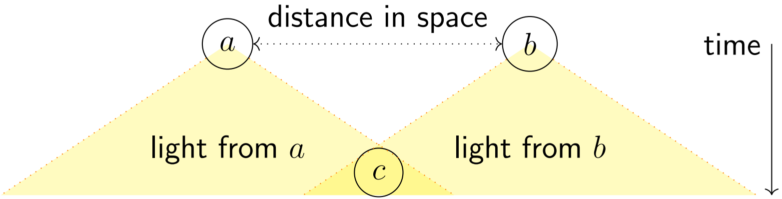


Causality

Taken from physics (relativity).

- When $a \to b$, then a **might have caused** b.
- When $a \parallel b$, we know that a **cannot have caused** b.

Happens-before relation encodes **potential causality**.

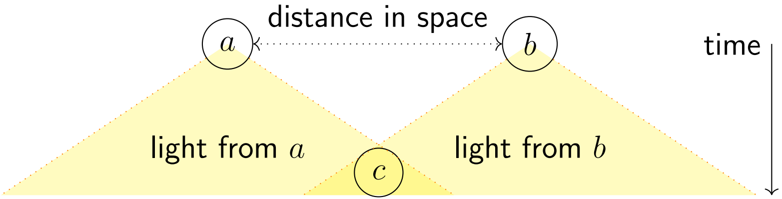


Causality

Taken from physics (relativity).

- When $a \to b$, then a **might have caused** b.
- When $a \parallel b$, we know that a **cannot have caused** b.

Happens-before relation encodes **potential causality**.



Let $\prec$ be a strict total order on events.
If $(a \to b) \implies (a \prec b)$ then $\prec$ is a **causal order**
(or: $\prec$ is "consistent with causality").
NB. "causal" $\neq$ "casual"!