Time, Clocks, and Ordering of events

A detective story

In the night between 30 June and 1 July 2012,
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

Distributed systems often need to *measure time*, to implement for instance:

- timeouts, failure detectors, retry timers
- performance measurements, statistics, profiling
- log files, event recording, temporal databases
- data with time-limited validity (e.g. cache entries, security protocols)
- determining order of events across different nodes
- geolocalization (GPS)

geolocalization (GPS)

- 31 satellites
- each carrying an high-precision clock
- each broadcasting time and location
- signals reach the same device on earth at different moments
- based on these differences, device can “triangulate” its own position.

Clocks and time

Distributed systems often need to *measure time*, to implement for instance:

- timeouts, failure detectors, retry timers
- performance measurements, statistics, profiling
- log files, event recording, temporal databases
- data with time-limited validity (e.g. cache entries, security protocols)
- determining order of events across different nodes
- geolocalization (GPS)
    - 31 satellites
    - each carrying an high-precision clock
    - each broadcasting time and location
    - signals reach the same device on earth at different moments
    - based on these differences, device can "triangulate" its own position.

event = something happening at a node (e.g. state change, send, receive)

timestamp = a value in a *partially ordered domain*

clock = mapping from *events* to *timestamps*

Time, Clocks, and Ordering of events

A detective story

In the night between 30 June and 1 July 2012,
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

Distributed systems often need to *measure time*, to implement for instance:

- timeouts, failure detectors, retry timers
- performance measurements, statistics, profiling
- log files, event recording, temporal databases
- data with time-limited validity (e.g. cache entries, security protocols)
- determining order of events across different nodes
- geolocalization (GPS)

event = something happening at a node (e.g. state change, send, receive)

timestamp = a value in a *partially ordered domain*

clock = mapping from *events* to *timestamps*

Two types of clocks

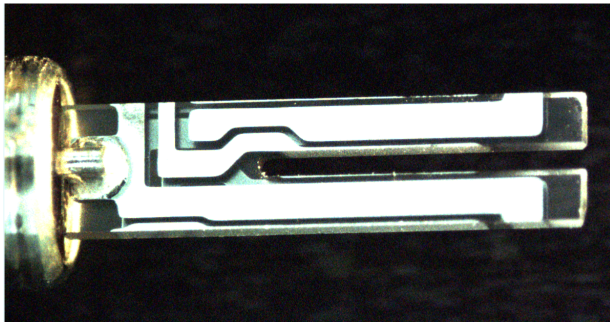
physical clocks: count *seconds elapsed*

logical clocks: count *events happened*

physical clocks: count *seconds elapsed*

Quartz clock

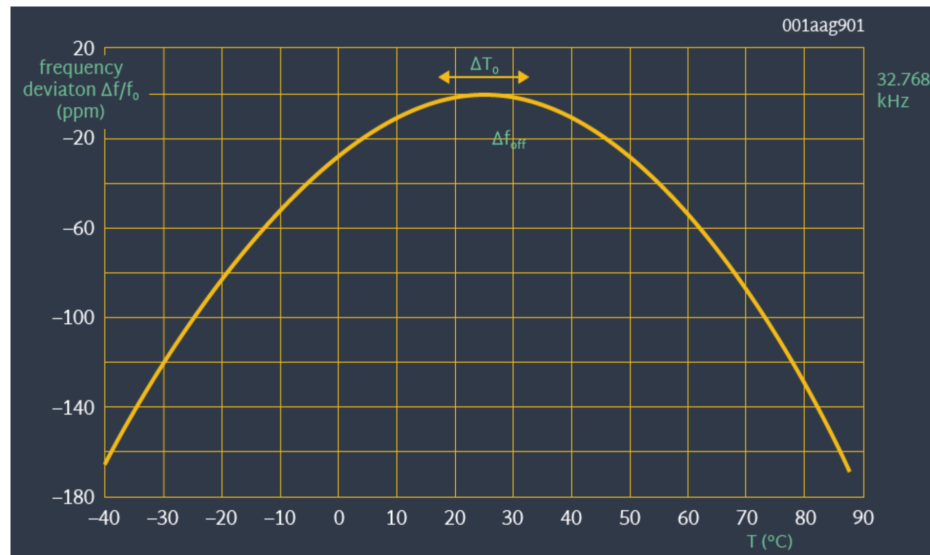
- a piezoelectric material tuned to resonate at fixed frequency (e.g. 32768 Hz)
- a feedback circuit that maintains oscillations at resonant frequency
- a flip-flop circuit that updates a state based on oscillations.



Frequency in fact may vary

Frequency in fact may vary

E.g. may depend on temperature:



clock **drift** = deviation from designed frequency
(e.g. in modern computers < 50 ppm ≈ 5ms/day)

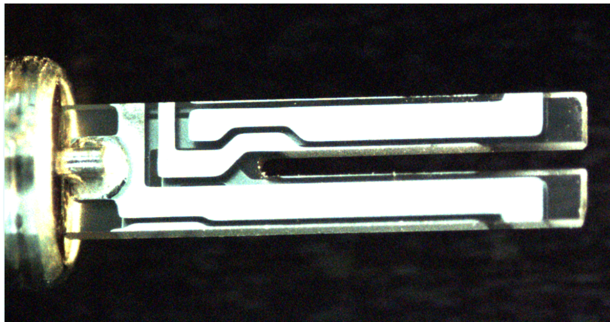
clock **skew** = difference between two clocks at a point in time *

* A man with a watch knows what time it is. A man with two watches is never sure
[Segal's law]

physical clocks: count *seconds elapsed*

Quartz clock

- a piezoelectric material tuned to resonate at fixed frequency (e.g. 32768 Hz)
- a feedback circuit that maintains oscillations at resonant frequency
- a flip-flop circuit that updates a state based on oscillations.



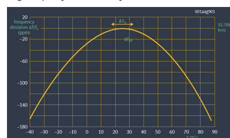
Atomic clock

- exploits transition frequencies of electrons in atoms to fine-tune (again) a quartz clock.

E.g. Caesium-133 clock:
frequency ≈9 GHz
drift ≈1s / 1 million years
price ≈ 10K$

Frequency in fact may vary

E.g. may depend on temperature:



clock **drift** = deviation from designed frequency
(e.g. in modern computers < 50 ppm ≈ 5ms/day)

clock **skew** = difference between two clocks at a point in time *

Time standards

* A man with a watch knows what time it is. A man with two watches is never sure
[Segal's law]

Time standards

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

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

...and there are many other time standards...

Problem: speed of Earth rotation in not constant

- **Coordinated Universal Time** (**UTC**)
  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:

1. the clock moves to 00:00:00 after one second, as usual
2. the clock immediately jumps forward to 00:00:00, skipping one second (**negative leap second**)
3. 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: https://hpiers.obspm.fr/iers/bul/bulc/bulletinc.dat

How do software and O.S. deal with leap seconds?

...in most cases they ignore them!

e.g. on 30 June 2012, on leap second,
a bug in Linux kernel generated a livelock,
causing many Internet services to go down.

...or they "smear" (spread out) the leap second over the course of a day

Clock synchronization

Each node tracks physical time with an internal quartz clock.

Due to clock *drift*, clock *skew* may gradually increase.

Solution: periodically get the current time (e.g. UTC) from a server with a more accurate clock, and correct the error accordingly.

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

Hierarchy of clock servers arranged into levels:

- level 0: atomic clock or GPS receiver
- level 1: synced directly with level-0 nodes
- level 2: servers that sync with level-1 nodes, etc.

Each node may:

- contact multiple servers, discard outliers, average rest
- make multiple request to the same server, using statistics to reduce random errors

Network Time Protocol (NTP)

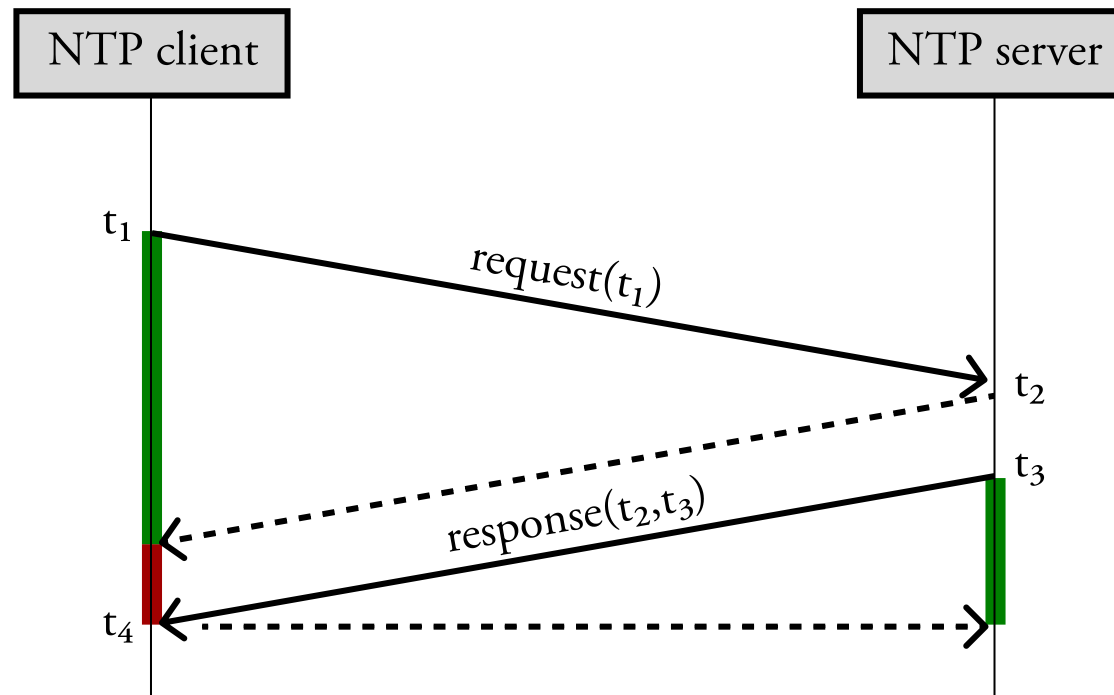
Hierarchy of clock servers arranged into levels:

- level 0: atomic clock or GPS receiver
- level 1: synced directly with level-0 nodes
- level 2: servers that sync with level-1 nodes, etc.

Each node may:

- contact multiple servers, discard outliers, average rest
- make multiple request to the same server, using statistics to reduce random errors

Estimating and correcting clock skew



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

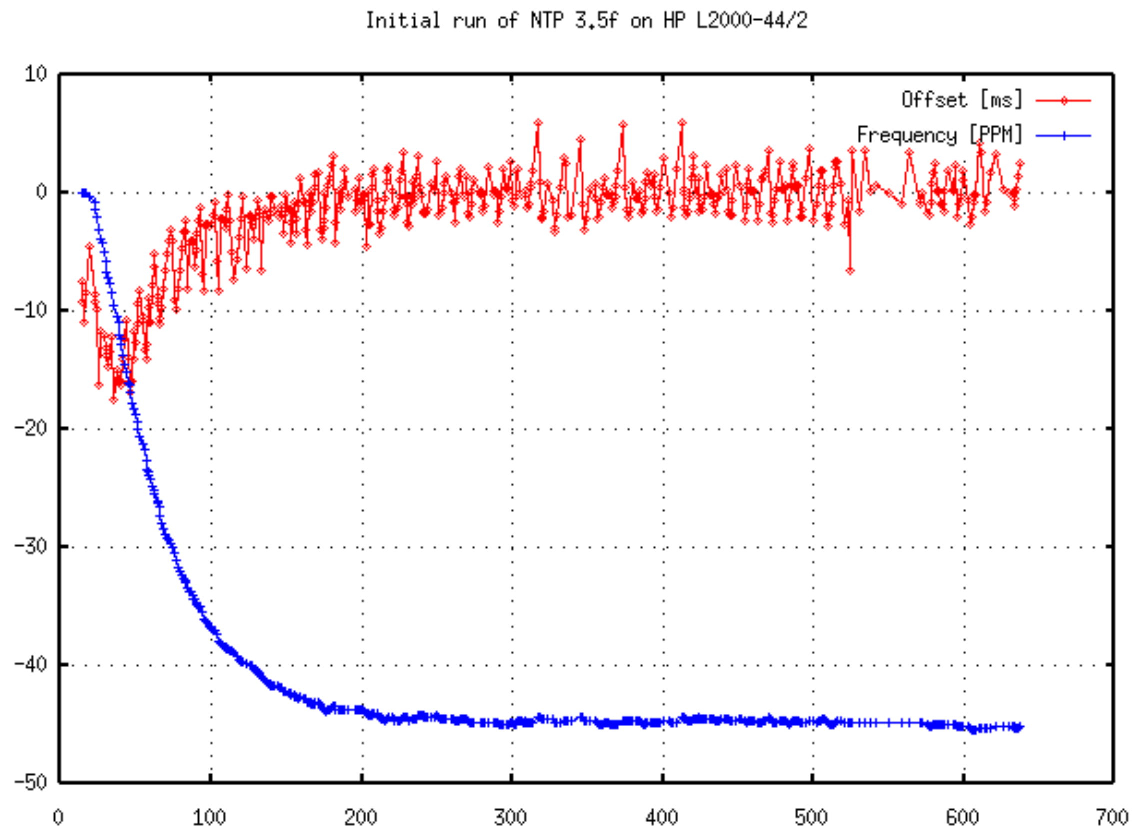
estimated server time
on receive response: $t_3 + \frac{\delta}{2}$

estimated clock skew
on receive response: $\theta = t_3 + \frac{\delta}{2} - t_4$

Once clock skew is estimated:

- if $|\theta| < 125\text{ms}$, then slightly speed up the clock or slow it down by up to 500ppm (brings clocks in sync within ≈5 minutes)
- if $125\text{ms} \leq |\theta| < 1\text{s}$, then set the clock to the estimated server time
- if $|\theta| \geq 1\text{s}$, then panic! (leave the problem for the human operator)

Estimating and correcting clock skew



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

estimated server time
on receive response: $t_3 + \frac{\delta}{2}$

estimated clock skew
on receive response: $\theta = t_3 + \frac{\delta}{2} - t_4$

Once clock skew is estimated:

- if $|\theta| < 125\text{ms}$, then slightly speed up the clock or slow it down by up to 500ppm (brings clocks in sync within ≈5 minutes)
- if $125\text{ms} \le |\theta| < 1\text{s}$, then set the clock to the estimated server time
- if $|\theta| \ge 1\text{s}$, then panic! (leave the problem for the human operator)

Time, Clocks, and Ordering of events

A detective story

In the night between 30 June and 1 July 2012,
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

Distributed systems often need to *measure time*, to implement for instance:

- timeouts, failure detectors, retry timers
- performance measurements, statistics, profiling
- log files, event recording, temporal databases
- data with time-limited validity (e.g. cache entries, security protocols)
- determining order of events across different nodes
- geolocalization (GPS)

event = something happening at a node
(e.g. state change, send, receive)

timestamp = a value in a *partially ordered domain*

clock = mapping from *events* to *timestamps*

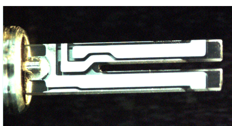
Two types of clocks

physical clocks: count *seconds elapsed*

logical clocks: count *events happened*

Quartz clock

- a piezoelectric material tuned to resonate at fixed frequency (e.g. 32768 Hz)
- a feedback circuit that maintains oscillations at resonant frequency
- a flip-flop circuit that updates a state based on oscillations.



Atomic clock

- exploits transition frequencies of electrons in atoms to fine-tune (again) a quartz clock.

E.g. Caesium-133 clock:
frequency ≈9 GHz
drift ≈1s / 1 million years
price ≈ 10K$

Frequency in fact may vary

E.g. may depend on temperature:

clock **drift** = deviation from designed frequency
(e.g. in modern computers < 50 ppm ≈ 5ms/day)

clock **skew** = difference between two clocks at a point in time *

* A man with a watch knows what time it is. A man with two watches is never sure
[Segal's law]

Time standards

- **Greenwich Mean Time (GMT)**
it's noon when the sun is in the south,
as seen from the Greenwich meridian
- **International Atomic Time (TAI)**
1 day is 24 × 60 × 60 × 9,192,631,770
periods of caesium-133's resonant frequency

...and there are many other time standards...

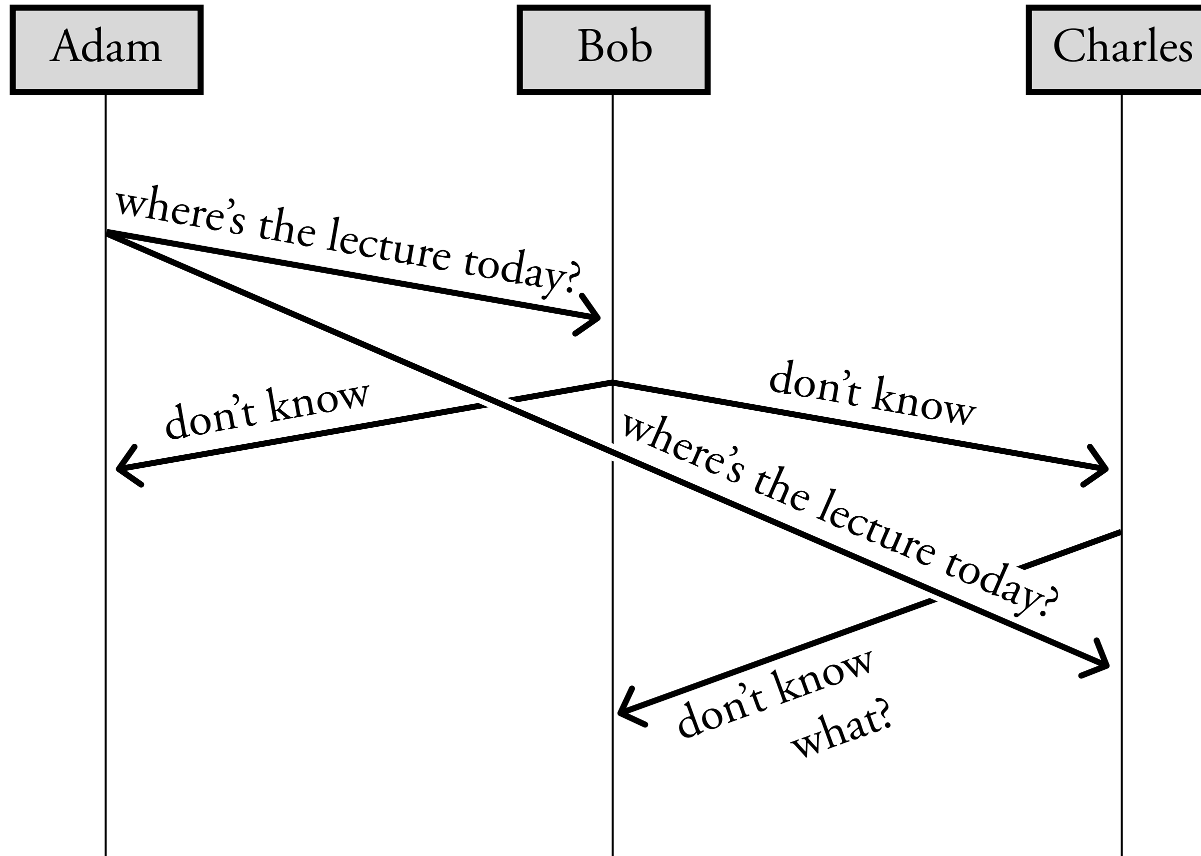
Problem: speed of Earth rotation in not constant

- **Coordinated Universal Time (UTC)**
TAI with corrections to account
for Earth rotation.
Time zones and daylight savings
time are offsets to UTC.

Leap seconds

Clock synchronization

Ordering of messages



Problem: even with synched physical clocks, message ordering may not be as expected!

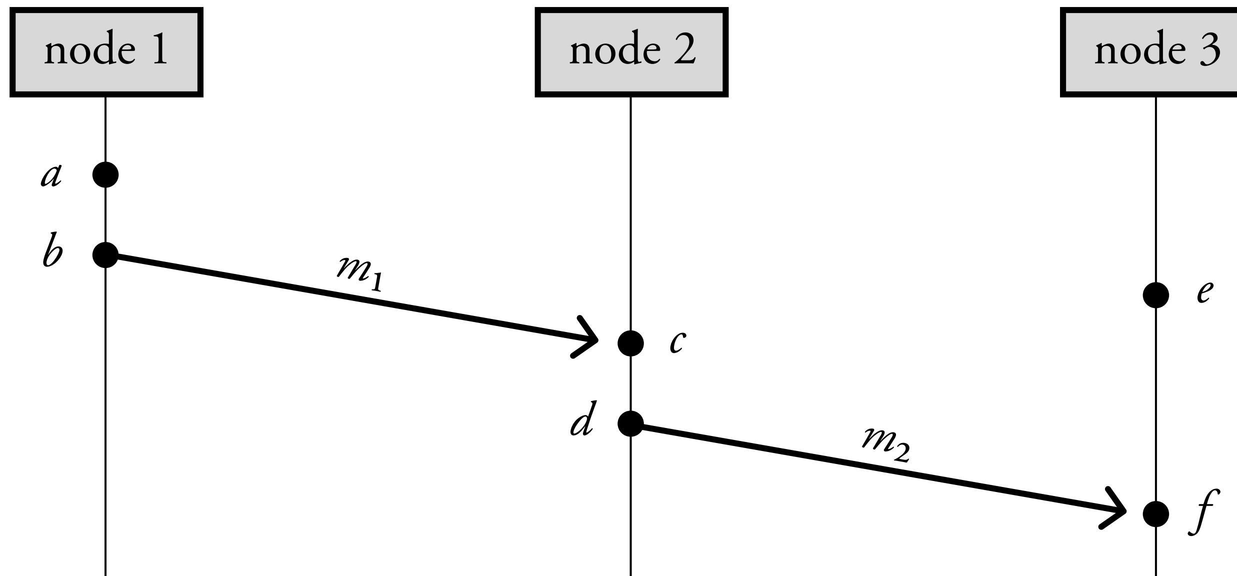
Happens-before relation

event *a* **happens before** event *b* = when one of the following conditions hold:

$a \prec b$

1) events *a*, *b* occured at the same node, and *a* occurred before *b* according to the node's local execution order
2) *a* is the sending of a message *m*, and *b* is the corresponding event that receives *m*
3) there is an even *c* such that $a \prec c \prec b$

$\prec$ is a *partial order* : it is possible that neither $a \prec b$ nor $b \prec a$ holds.
In this case we say that *a*, *b* are **concurrent** events, and write $a \parallel b$

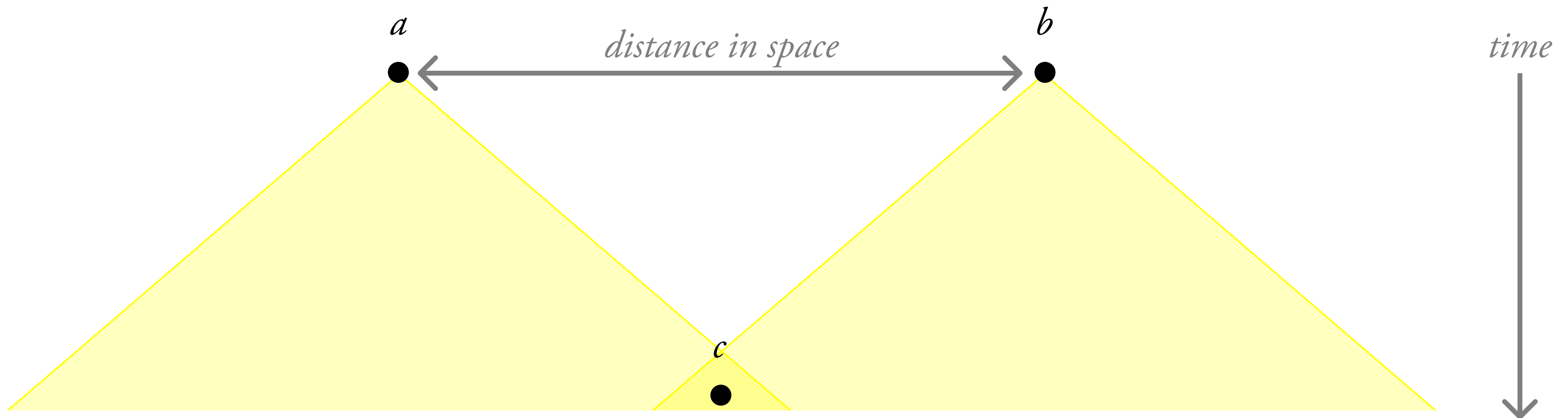


- $a \prec b$, $c \prec d$, and $e \prec f$, due to node execution order
- $b \prec c$ and $d \prec f$, due to messages m_1 and m_2
- $a \prec b \prec c \prec d \prec f$, due to transitivity
- $a, b, c, d \parallel e$

Causality

Happens-before relation encodes *potential causality*:

- if $a \prec b$, then *a* **may have caused** *b*
- if $a \parallel b$, then *a* **cannot have caused** *b*



We'd like to have a clock whose induced ordering coincides with / refines the happens-before relation.

Physical clocks are useful for many things, but not here: they may be even inconsistent with causality!
(e.g. could happen that *a* has caused *b*, but *timestamp*(*a*) > *timestamp*(*b*))

Lamport clocks

```
on initialisation do    // each node has a local counter t
    t := 0

on local event (no send or receive) do
    t := t + 1

on request to send msg over ch do
    t := t + 1
    send <msg, t> over ch
```

piggybacking

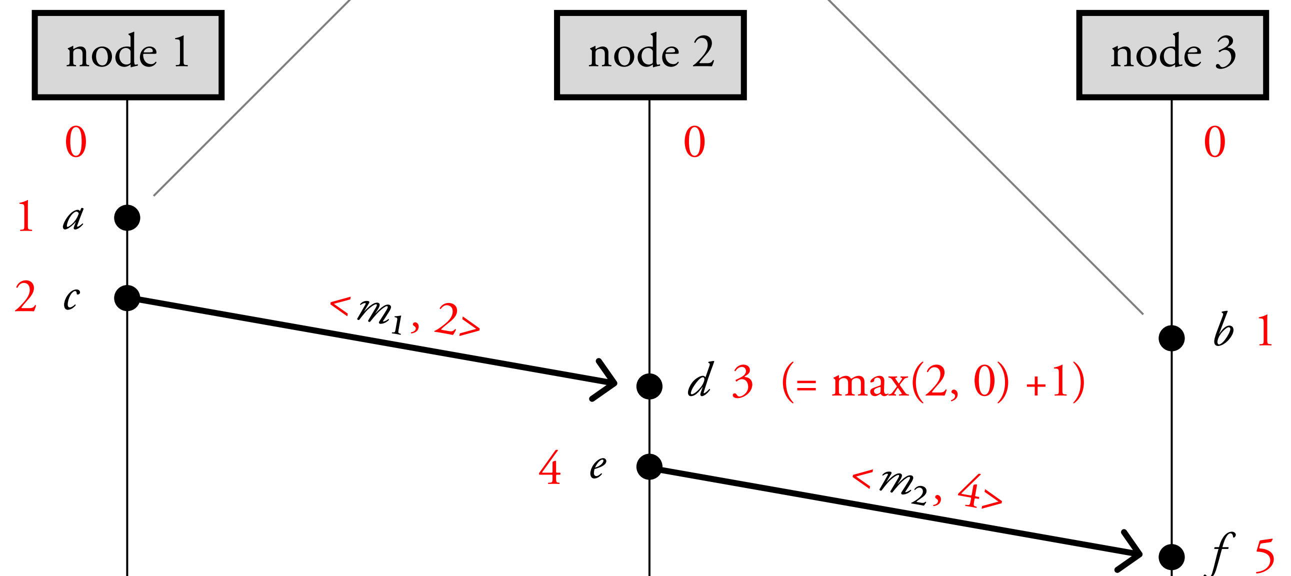
```
on receiving <msg, t'> do
    t := max(t, t') + 1
    deliver msg
```

Lamport clocks

```
on initialisation do      // each node has a local counter t
    t := 0

on local event (no send or receive) do
    t := t + 1

on request to send msg over ch do
    t := t + 1
    send <msg, t> over ch
                  piggybacking

on receiving <msg, t'> do
    t := max(t, t') + 1
    deliver msg
```

This gives a *total ordering* $<$ refining $\prec$
e.g. $a < b < c < d < e < f$

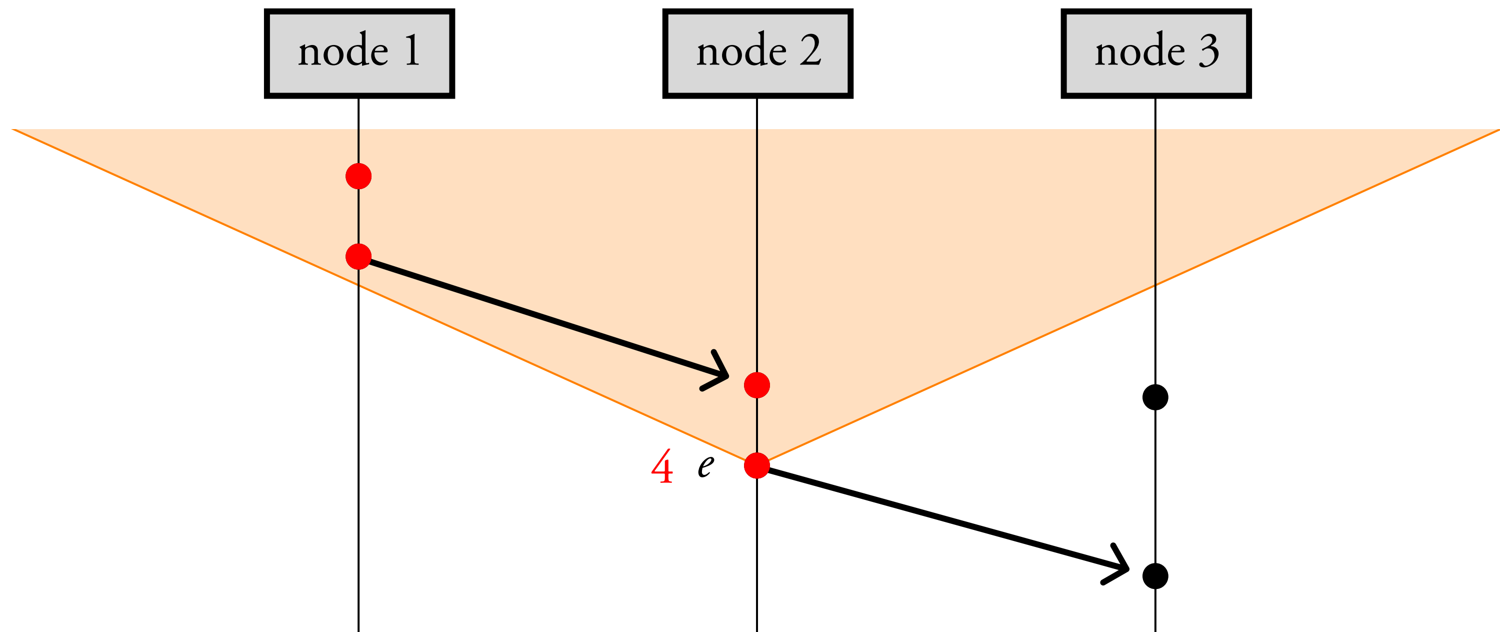
break ties using node IDs
(e.g. a@node1 $<$ b@node3)



Semantics of Lamport clocks

∀ event e at node n

$t_n(e)$ = max length of a chain of events totally ordered by happened before and ending in e (events of the chain may occur at arbitrary nodes)



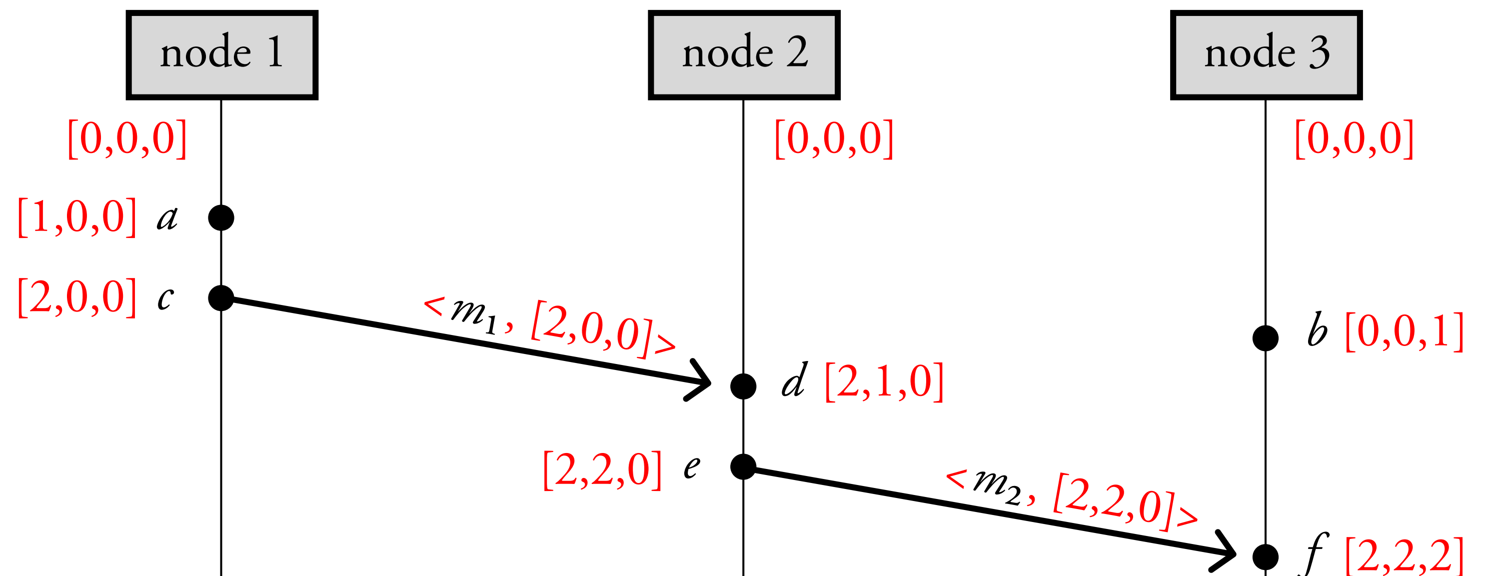
Exercise: prove the claim, and then use it to derive $d \prec e \;\Rightarrow\; t(d) < t(e)$

Vector clocks

```
on initialisation do      // each node has a local array t of counters
    t := [0,0,...,0]      // (one entry for each node in the network)


on local event (no send or receive) do
    t[my_id] := t[my_id] + 1


on request to send msg over ch do
    t[my_id] := t[my_id] + 1
    send <msg, t> over ch


on receiving <msg, t'> do
    for all nodes i
        t[i] := max(t[i], t'[i])
    t[my_id] := t[my_id] + 1
    deliver msg
```

Vector clocks

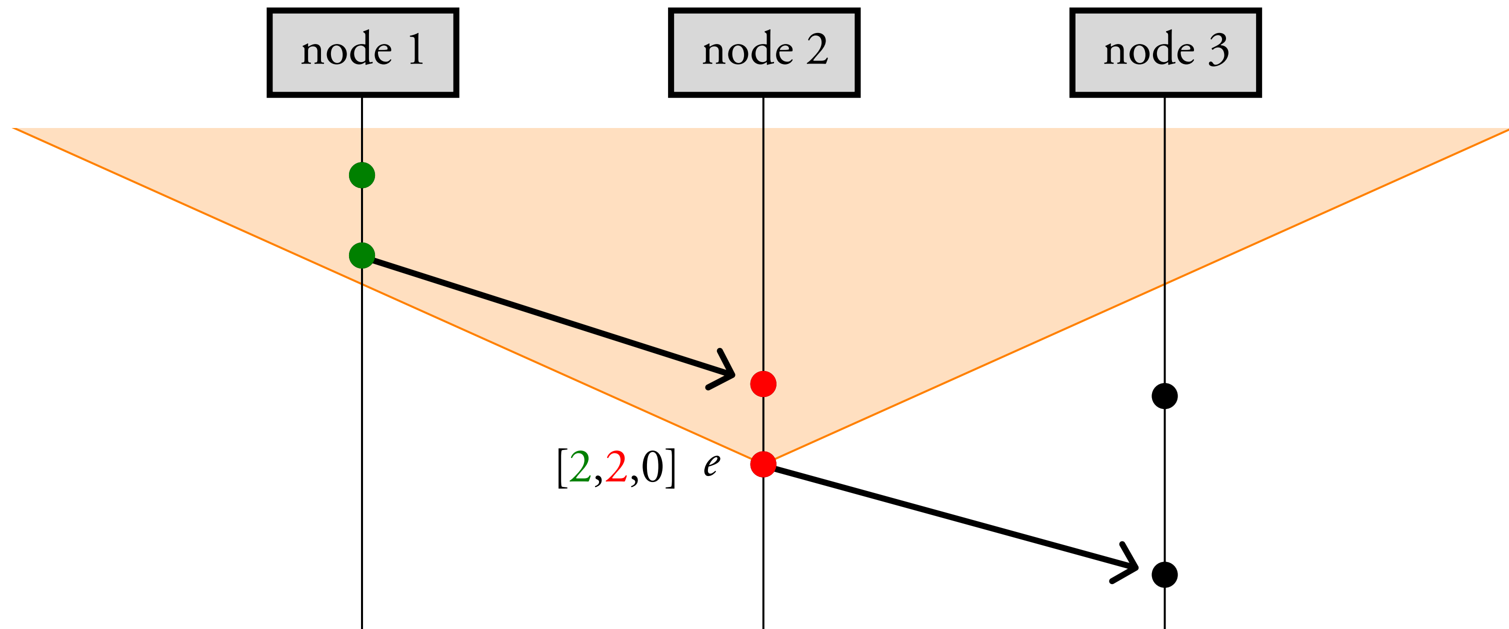
```
on initialisation do        // each node has a local array t of counters
    t := [0,0,...,0]        // (one entry for each node in the network)


on local event (no send or receive) do
    t[my_id] := t[my_id] + 1


on request to send msg over ch do
    t[my_id] := t[my_id] + 1
    send <msg, t> over ch


on receiving <msg, t'> do
    for all nodes i
        t[i] := max(t[i], t'[i])
    t[my_id] := t[my_id] + 1
    deliver msg
```



Semantics of vector clocks

∀ event e at node n ∀ i

$t_n(e)[i]$ = no. of events happened at node i and before event e (+1 if $i = n$)



Write $t(d) \leq t(e)$ if $\forall i \;\; t(d)[i] \leq t(e)[i]$ (pointwise partial ordering)

$t(d) = t(e)$ if $\forall i \;\; t(d)[i] = t(e)[i]$

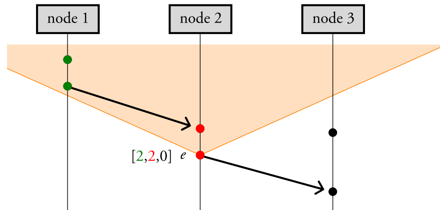
$t(d) < t(e)$ if $t(d) \leq t(e) \;\wedge\; t(d) \neq t(e)$

Prove $t(d) < t(e) \;\Leftrightarrow\; d \prec e$ (vector clocks ordering = happen-before)

Semantics of vector clocks

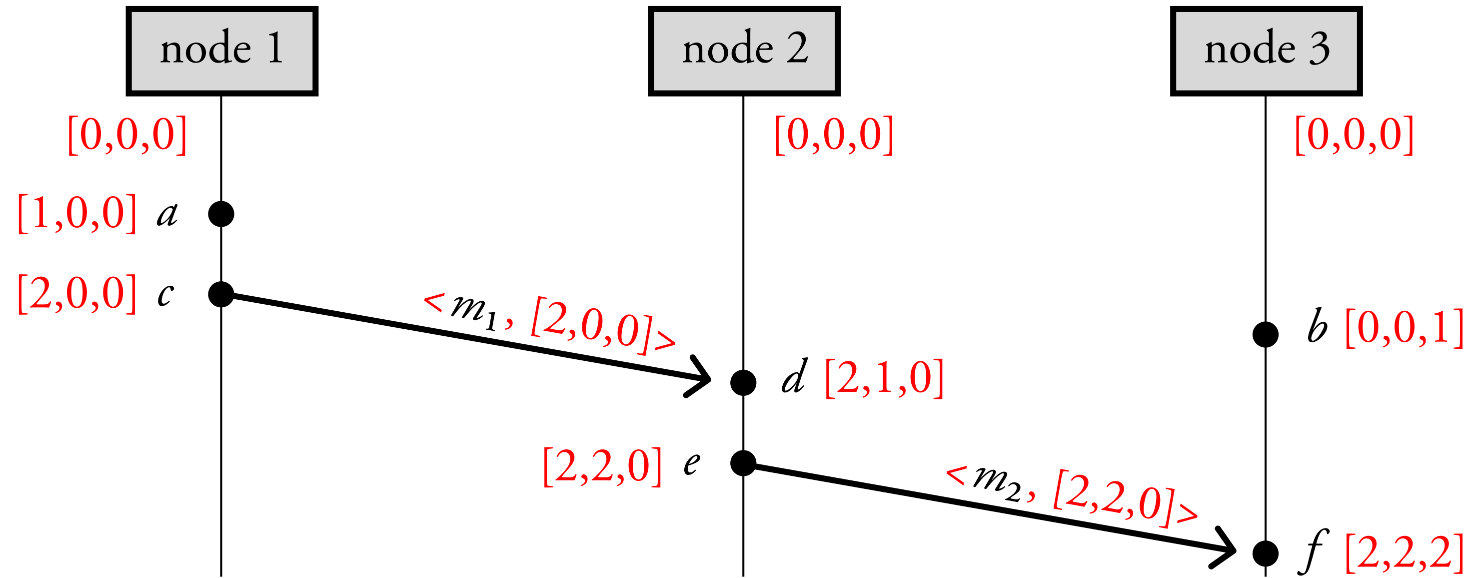
$\forall$ event e at node n $\forall$ i

$t_n(e)[i]$ = no. of events happened <u>at node i</u> and <u>before event e</u>
(+1 if $i = n$)



Write $t(d) \leq t(e)$ if $\forall i \;\; t(d)[i] \leq t(e)[i]$ (pointwise partial ordering)
$t(d) = t(e)$ if $\forall i \;\; t(d)[i] = t(e)[i]$
$t(d) < t(e)$ if $t(d) \leq t(e) \wedge t(d) \neq t(e)$

Prove $t(d) < t(e) \Leftrightarrow d \prec e$ (vector clocks ordering = happen-before)



Potential implementation problems:

- need to know number of nodes (= size of vector clock)
- need to identify each node with a coordinate
- annotated messages may become very large

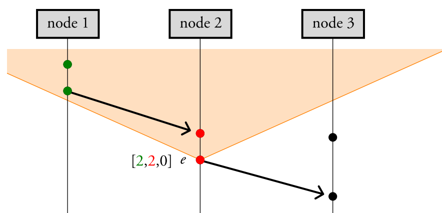
1) use *associative arrays* and hide entries with value 0

2) besides actual value $t[i]$, each node maintains:

- $u[i]$ = value of t[`my_id`] when <u>$t[i]$ was last updated</u>
- $s[i]$ = value of t[`my_id`] when <u>a message to i was last sent</u>

Note:

- $u[i]$ only changes if $t[i]$ changes, and $s[i]$ only changes if a message is sent to i
- for every i and j, $t[j]$ has changed since a message to i was last sent only if $s[i] < u[j]$



Write $t(d) \leq t(e)$ if $\forall i \;\; t(d)[i] \leq t(e)[i]$ (pointwise partial ordering)

$t(d) = t(e)$ if $\forall i \;\; t(d)[i] = t(e)[i]$

$t(d) < t(e)$ if $t(d) \leq t(e) \;\wedge\; t(d) \neq t(e)$

Prove $t(d) < t(e) \;\Leftrightarrow\; d \prec e$ (vector clocks ordering = happen-before)

[1,0,0] a

[2,0,0] c

$<m_1$, [2,0,0]$>$

b [0,0,1]

d [2,1,0]

[2,2,0] e

$<m_2$, [2,2,0]$>$

f [2,2,2]

Potential implementation problems:

- need to know number of nodes (= size of vector clock)
- need to identify each node with a coordinate
- annotated messages may become very large

1) use *associative arrays* and hide entries with value 0

2) besides actual value $t[i]$, each node maintains:

- $u[i]$ = value of t[`my_id`] when $t[i]$ was last updated
- $s[i]$ = value of t[`my_id`] when a message to i was last sent

Note:

- $u[i]$ only changes if $t[i]$ changes, and $s[i]$ only changes if a message is sent to i
- for every i and j, $t[j]$ has changed since a message to i was last sent only if $s[i] < u[j]$

tinyurl.com/vectorclock

tinyurl.com/vectorclockbenchmark

Time, Clocks, and Ordering of events

[back]

A detective story

In the night between 30 June and 1 July 2012,
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

Distributed systems often need to *measure time*, to implement for instance:
- timeouts, failure detectors, retry timers
- performance measurements, statistics, profiling
- log files, event recording, temporal databases
- data with time-limited validity (e.g. cache entries, security protocols)
- determining order of events across different nodes
- geolocalization (GPS)

event = something happening at a node (e.g. state change, send, receive)

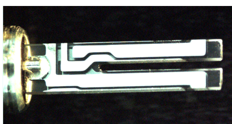
timestamp = a value in a *partially ordered domain*

clock = mapping from *events* to *timestamps*

Two types of clocks

physical clocks: count *seconds elapsed*

Quartz clock
- a piezoelectric material tuned to resonate at fixed frequency (e.g. 32768 Hz)
- a feedback circuit that maintains oscillations at resonant frequency
- a flip-flop circuit that updates a state based on oscillations.



Atomic clock
- exploits transition frequencies of electrons in atoms to fine-tune (again) a quartz clock.

E.g. Caesium-133 clock:
frequency ≈9 GHz
drift ≈1s / 1 million years
price ≈ 10K$

Frequency in fact may vary

E.g. may depend on temperature:

clock **drift** = deviation from designed frequency (e.g. in modern computers < 50 ppm ≈ 5ms/day)

clock **skew** = difference between two clocks at a point in time *

* A man with a watch knows what time it is. A man with two watches is never sure
[Segal's law]

Time standards

- **Greenwich Mean Time** (GMT)
  it's noon when the sun is in the south, as seen from the Greenwich meridian
- **International Atomic Time** (TAI)
  1 day is 24 × 60 × 60 × 9,192,631,770 periods of caesium-133's resonant frequency

...and there are many other time standards...

Problem: speed of Earth rotation in not constant

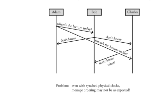
- **Coordinated Universal Time** (UTC)
  TAI with corrections to account for Earth rotation.
  Time zones and daylight savings time are offsets to UTC.

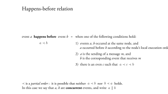
Leap seconds

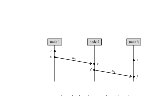
Clock synchronization

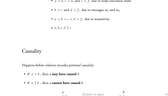
logical clocks: count *events happened*

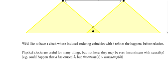
Ordering of messages



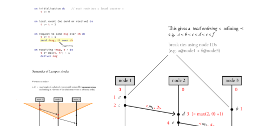








Lamport clocks



Vector clocks