Programming of Distributed Systems

Topic IV – Time and Order

Dr.-Ing. Dipl.-Inf. Erik Schaffernicht

Reading Remarks

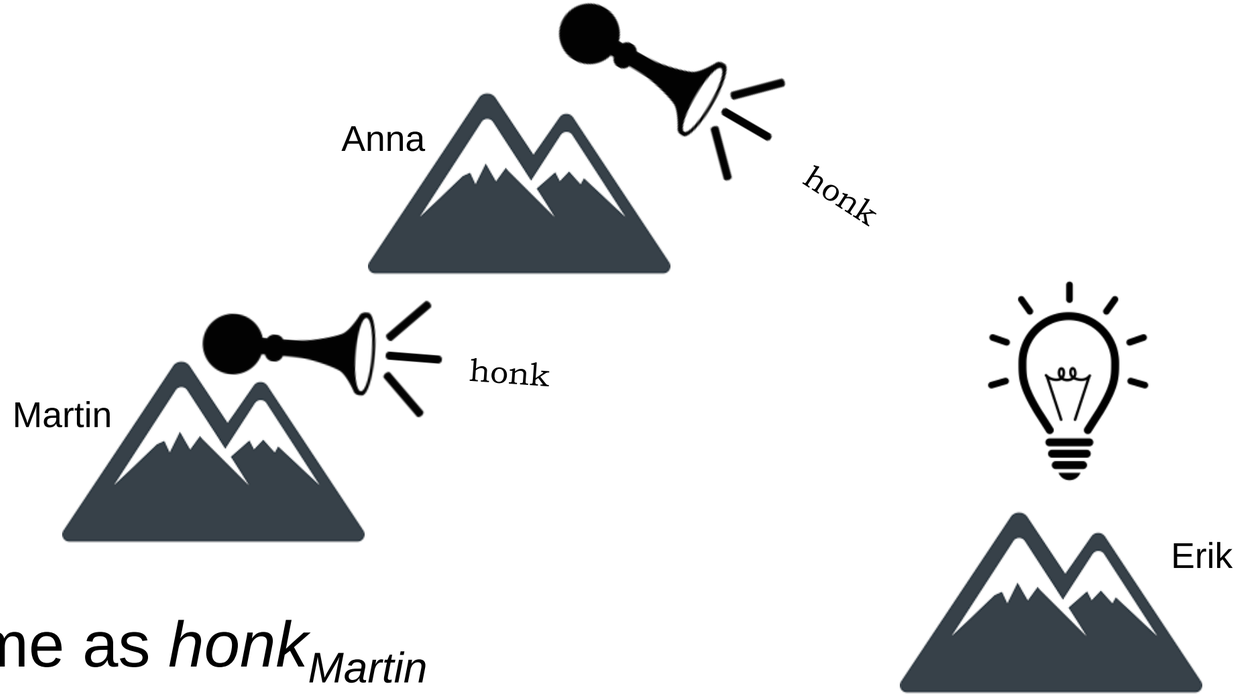
Reading Task:
Chapter 6.1 & 6.2.

Sequences

Central role for coordination, security and failure tolerance algorithms

- Order of distributed events
  - Fairness, Selection via *First Come First Served*
  - Insuring deadlock free transactions
  - Software management; e.g. `make`
- Shared time
  - Real-time systems
  - time stamps in cryptographic protocols
  - authentication & authorisation tickets (time of validity)

The problem

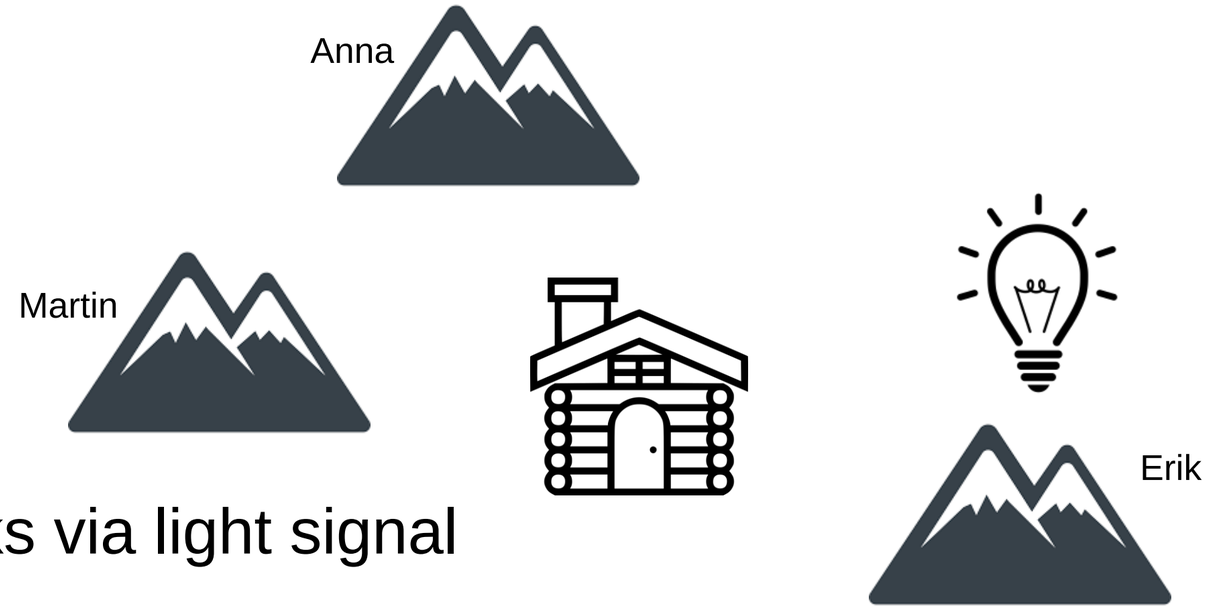


Three different views:

1. **Erik's view**
   $honk_{Anna}$ at the same time as $honk_{Martin}$
2. **Anna's view**
   $honk_{Anna}$ before $honk_{Martin}$
3. **Martin's view**
   $honk_{Anna}$ after $honk_{Martin}$

➔ Consequence of latency: 3 observer, 3 different sequences of events

Another problem



Meeting in the hut at 5pm

Sychronization of local clocks via light signal

Result

- All three arrive at different times,
  and everyone believes they are exactly on time

→ Local clocks lead to different notions of previously sychronized times

Roadmap

Temporal sequences

- physical clocks and their properties
- algorithms to synchronize physical clocks

Causal sequences

- logical clocks and their properties
- algorithms to synchronize logical clocks

Make example

make all → compiles *foo.c* iff

- *foo* does not exist or
- *foo* is older than *foo.c*

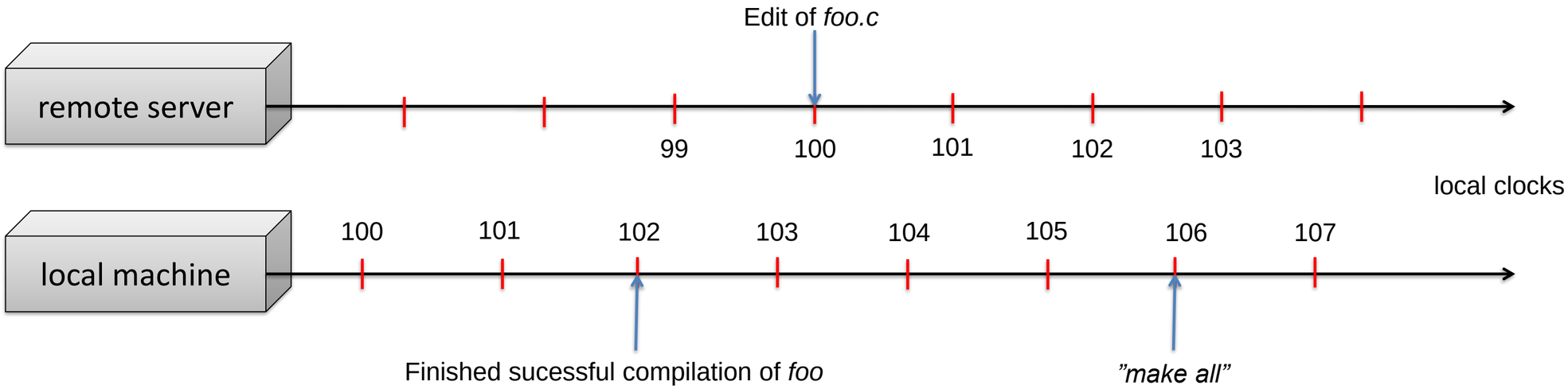
```
Makefile:
all: gcc -o foo foo.c
```

Distributed scenario

- *foo.c* on remote fileserver
- *foo* and compiling on local machine
- editor and *make all* executed on local machine

```
Makefile:
all: gcc -o foo /src/area51/foo.c
```

Make example contd



- Reality: *foo* is older than *foo.c* ➔ needs to be compiled
- Local perspective: *foo* is younger than *foo.c* ➔ no compilation

➔ How to create a shared perspective of time?

Coordinated Universal Time (UTC)

Paris, 05 July 2022

Bulletin C 64

To authorities responsible
for the measurement and
distribution of time

INFORMATION ON UTC - TAI

NO leap second will be introduced at the end of December 2022.
The difference between Coordinated Universal Time UTC and the
International Atomic Time TAI is :

from 2017 January 1, 0h UTC, until further notice : UTC-TAI = -37 s

Leap seconds can be introduced in UTC at the end of the months of December or June, depending on the evolution of UT1-TAI. Bulletin C is mailed every six months, either to announce a time step in UTC, or to confirm that there will be no time step at the next possible date.

Christian BIZOUARD
Director
Earth Orientation Center of IERS
Observatoire de Paris, France

Derived from the International Atomic Time (TAI)

- around 50 institutes with 400 atomic clocks
- mostly caesium clocks (drift approx. 1 sec in 100 000 000 years)

➔ weighted average time via GPS satellites

Published by shortwave radio stations and satellites

Computer Clocks

- electrical component that counts the oscillation of quartz crystals
- counter is available via register/memory cells
- operating systems makes this information available via OS-API and provides translations into common time and date formats

Resolution

- about 1ns (clock frequency of 1 GHz)

Drift

- about 1 sec in 11 days (drift rate 10^{-6}) in todays clocks

Drift effects

Worst-case difference of two clocks with a drift rate of 10^{-6}

After 10 seconds: 10 secs * 2 * 10^{-6} = 20 000 ns

After 1 day: 86400 sec * 2 * 10^{-6} = 0.17 sec

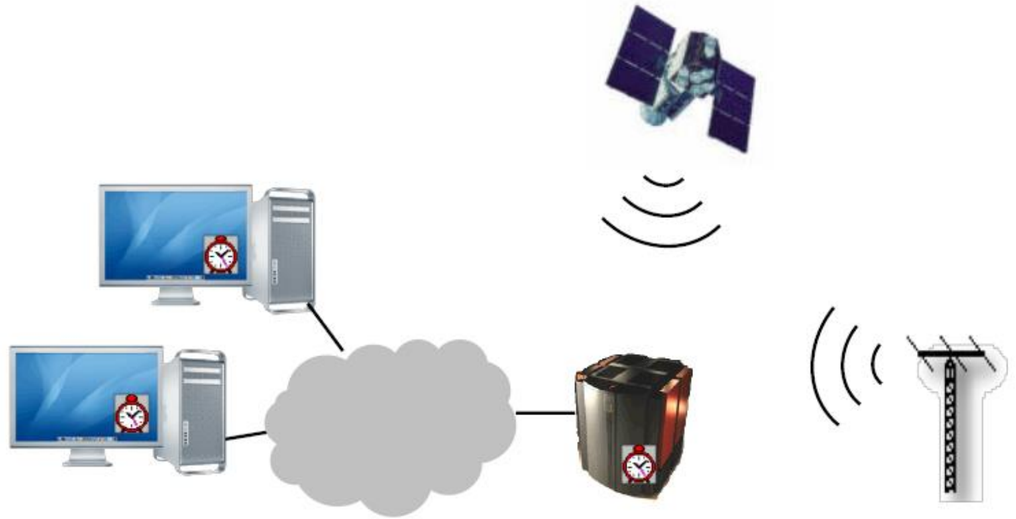
→ (Synchronized) clocks diverge fast

Synchronziation required! → When? How often? How?

Synchronization of physical clocks

Level of correspondence between a computer clock and UTC depends on

1. Precision of the reference clock
2. Jitter of the signals: distance from the reference, atmospheric conditions
3. Drift of local clocks
4. Sychronization algorithm



Quantifying drift

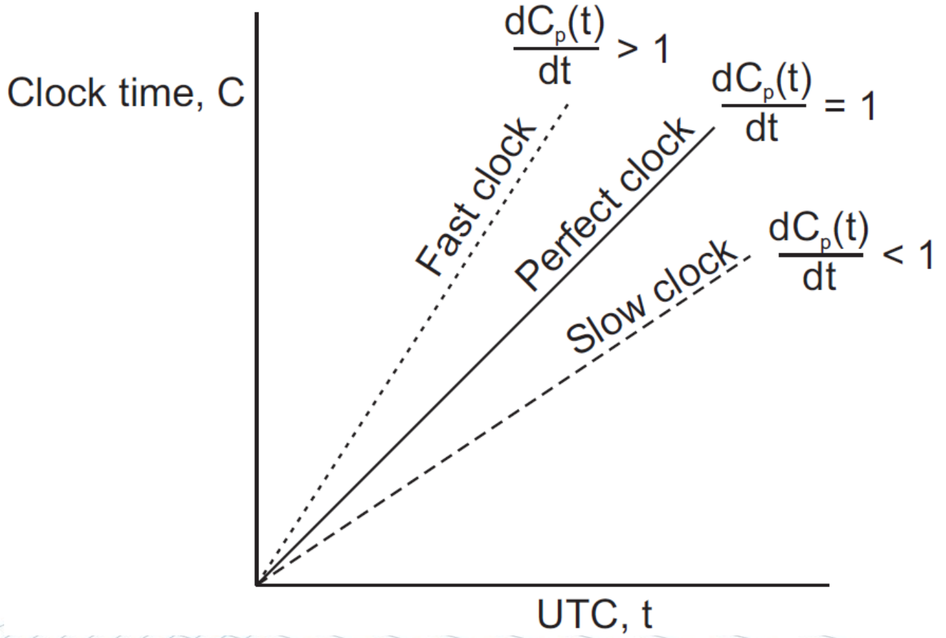
Goal

To synchronize in a way that the clock deviation is less then α

t reference time (UTC)

$C(t)$ value of the computer clock at time t

Ideally: $C(t) = t$; $\frac{C(t_2)-C(t_1)}{t_2-t_1} = \frac{dc}{dt} = 1$



Maximum clock drift rate

ρ maximum drift rate, part of clock specification

- Clock working within the specification iff $1 - \rho \leq \frac{dc}{dt} \leq 1 + \rho$
- Quartz based clocks $10^{-6} \leq \rho \leq 10^{-5}$

Maximum deviation δ of two sychronized clocks after Δt:

$$\delta \leq (\rho_1 + \rho_2)\Delta t$$

➔ Resychronize clocks with a maximum deviation α every

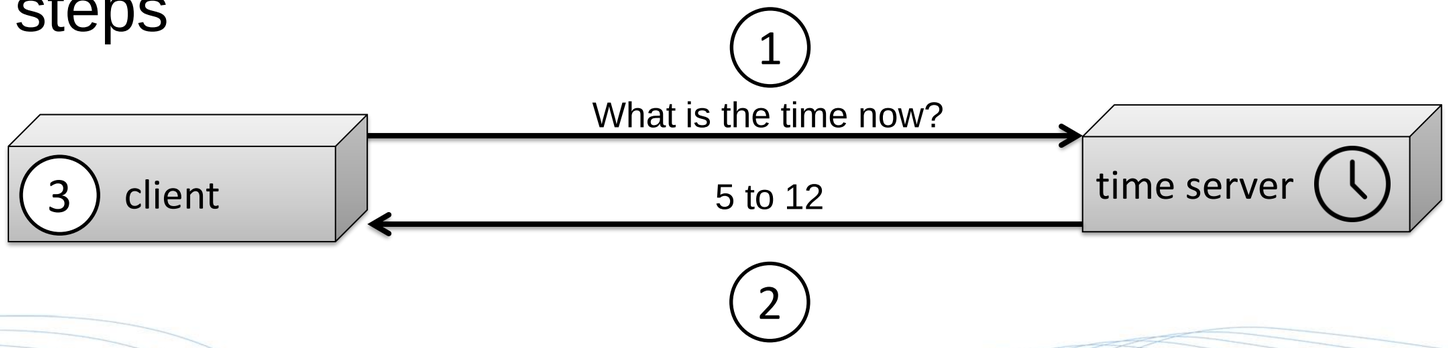
$$\Delta t = \frac{\alpha}{\rho_1 + \rho_2} \text{ seconds}$$

Passive time server: Christian's algorithm

Scenario

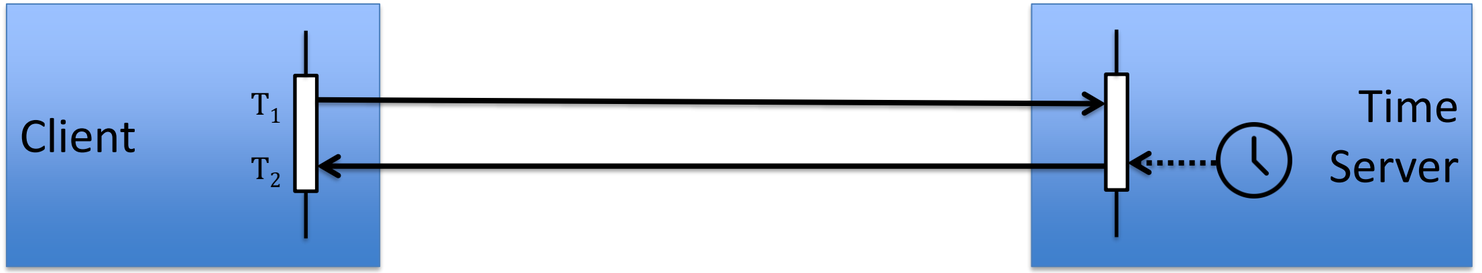
- Time server with reference time
- Clients with local clocks
- Maximum drift deviation α → individual sychronization interval

3 simple steps



3 problems – 3 corrections

1. Message roundtrip time



→ Correction of half the roudtrip time for the time request

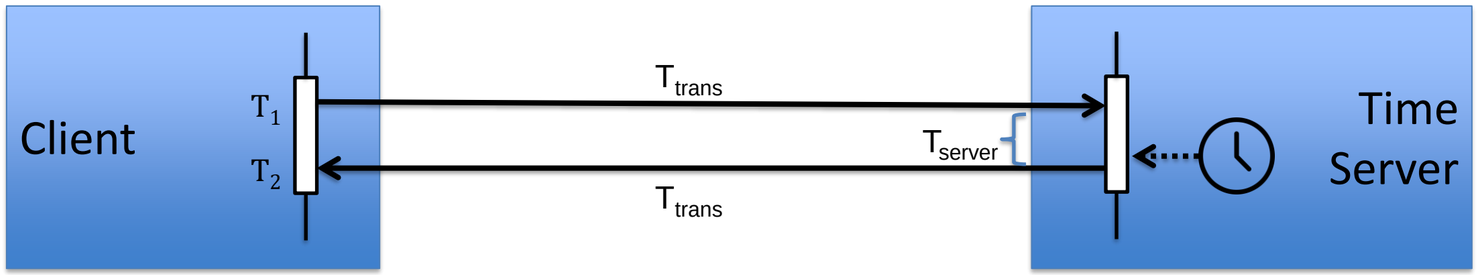
$$K = \frac{T_2 - T_1}{2}$$

2. Different latencies (jitter) for the messages

→ Statistical average of the round trip times $K = \frac{\mathrm{E}(T_2 - T_1)}{2}$

3 problems – 3 corrections

3. Unknown server load



→ Correction for the server work time T_{server}

Assumptions

- Both directions are equally fast
- Server checks clock directly before sending reply, $K \approx T_{trans}$

$$T_2 - T_1 = 2T_{trans} + T_{server};\ K = T_{trans} = \frac{T_2 - T_1 - T_{server}}{2}$$

Limitations of Christian's algorithm

Scalability

- time server is a bottleneck

Precision

- jitter of message transport times in WANs

Failure tolerance

- time server is a single-point-of-failure

Time monotony

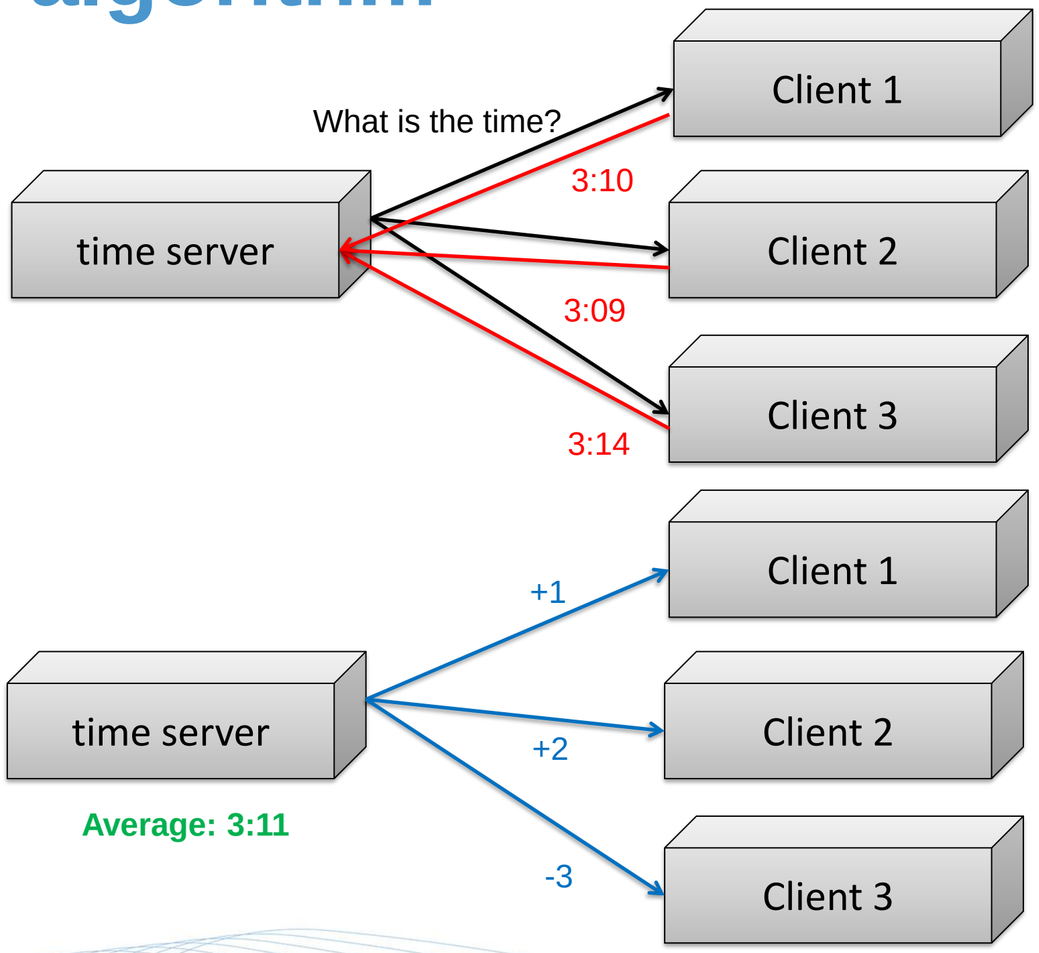
- (backward) jumps when correcting local time

- time server group
- different sychronization algorithm

- soft speed-up or slow-down of the clock

Active time server: Berkeley algorithm

1. Server sends time request to clients
2. Clients respond with their local time
3. Server calculates the average time
4. Sends the time difference to each client



Active time server: Berkeley algorithm

Pros

- No reference clock required
- No single point of failure, as every client can take the server role

Cons

- No reference clock required
- Scalability
  - Server has to handle multiple request/reply messages
  - Interval of synchronization dependent on worst clock drift

Network Time Protocol

Goal

Reliable, scalable UTC time sychronization under changing message travel times ➔ Internet time

Features

- Precision (deviation from UTC 10 msec in WANs, 1msec in LANs, 1µsec in LANs with local UTC clock)
- Reliability (fail-stop & byzantian failures, attack resilient)
- Scalability (optimizing number and structure of time servers)

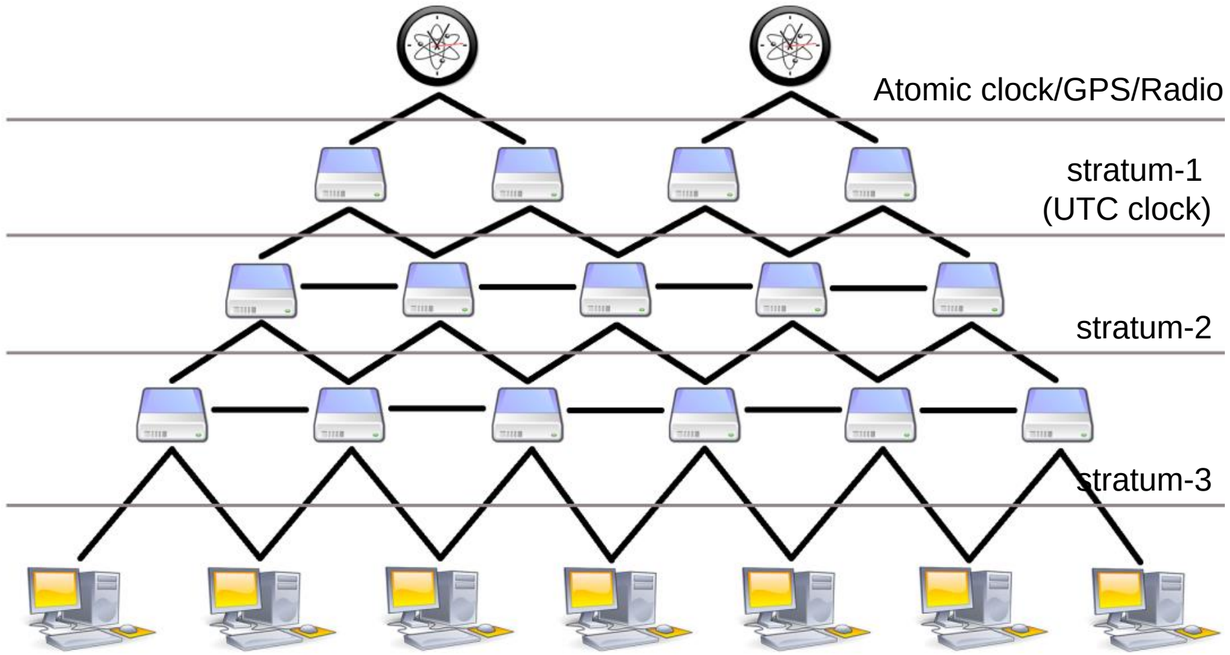
Network Time Protocol

Standard service in the internet

(currently > 1 million servers)

Client/Server model

- Servers in different communication roles
- Hierarchical, tree-like sychronization topology



edges show communcation path for sychronization

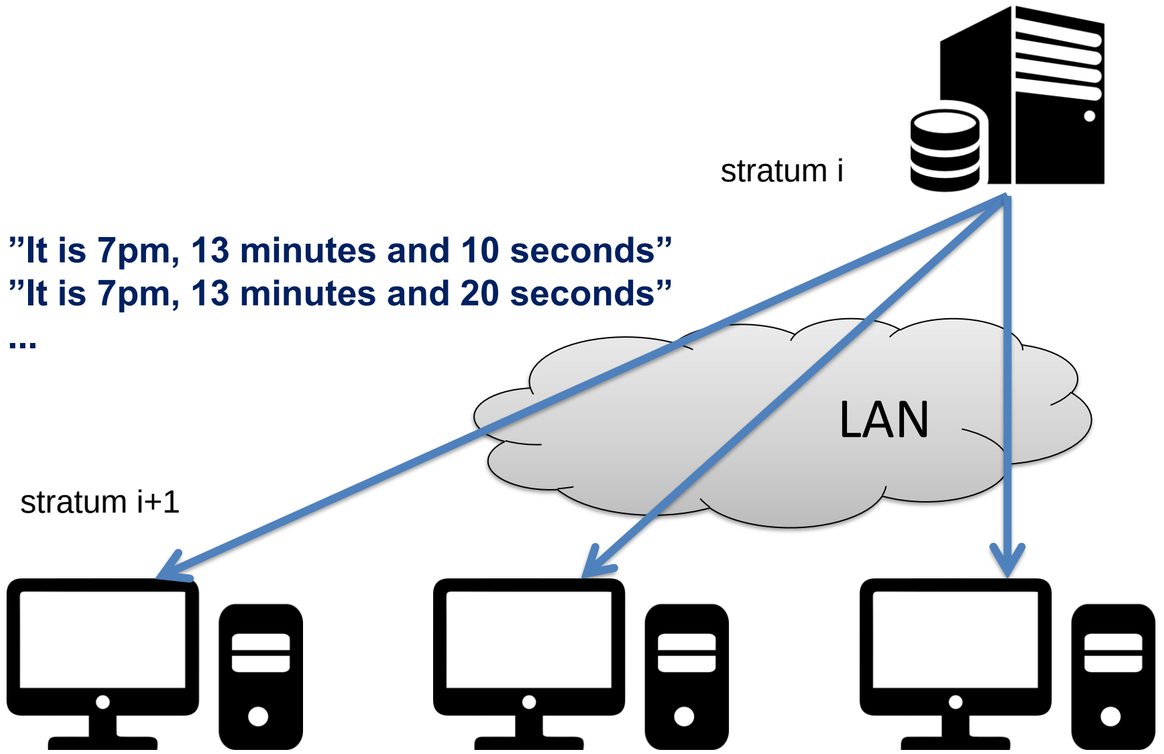
1. Time Announcements using Multicast

Idea

Time server sends periodically time messages to receivers

➔ Feasible only in LANs

Single time message (commonly UDP)

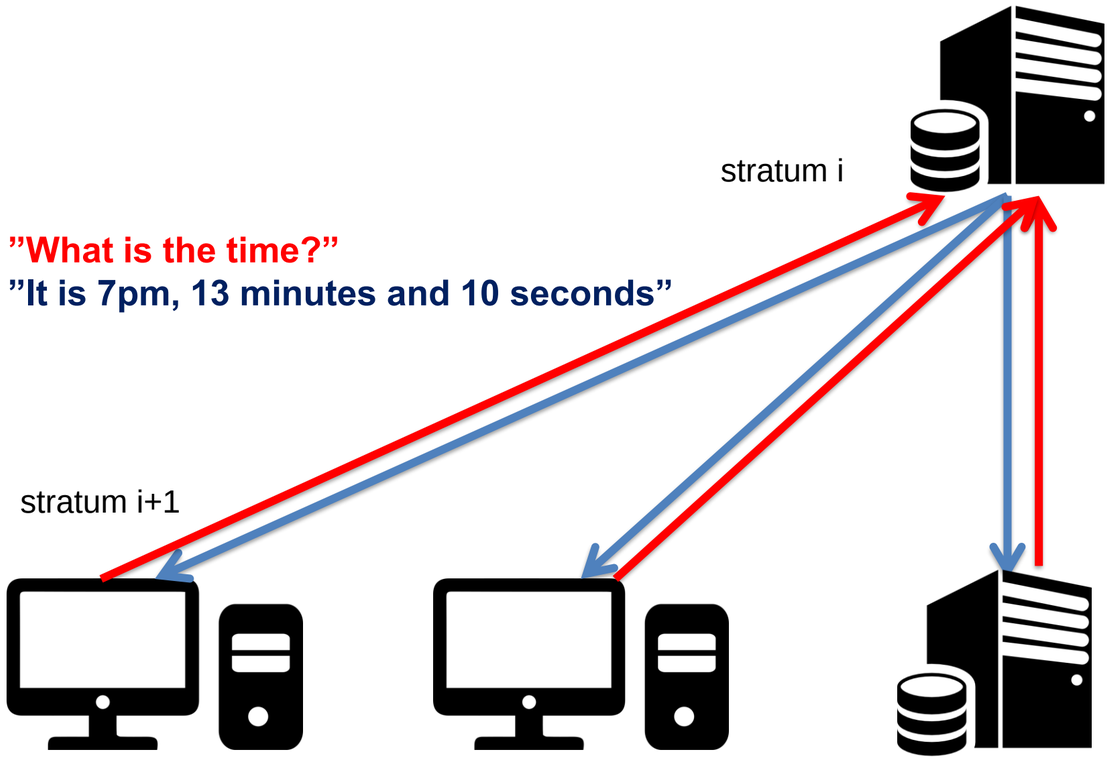


2. Time Requests with Christian's algorithm

Used in networks without multicast, and for higher precision

- typical for file servers, name services, authentication and authorization servers work stations ...

Two messages (request/reply per UDP)



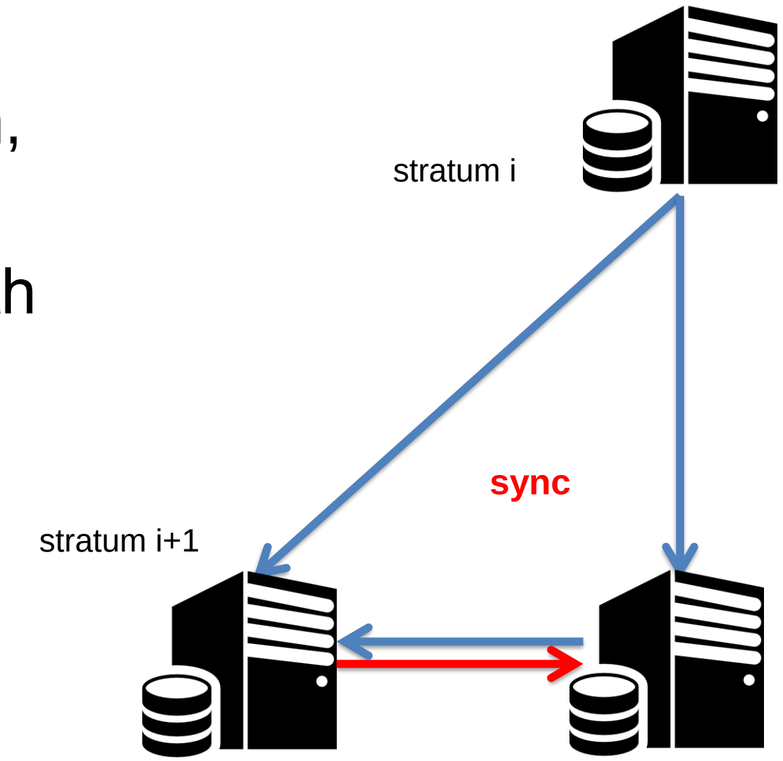
3. Symmetric time calibration

Idea

Bi-directional exchange of time information, most often used on low strata numbers

Adjustment of a server's stratum based with whom it sychronizes

Two messages pairs
(again UDP)



Sychronization algorithm

Goal

Sychronizing time server A with time server B

Method

A estimates offset o of the local clock compared to time server B

1. estimating o by exchanging a message pair
2. calculation of the error
3. restart if error is exceeding a limit

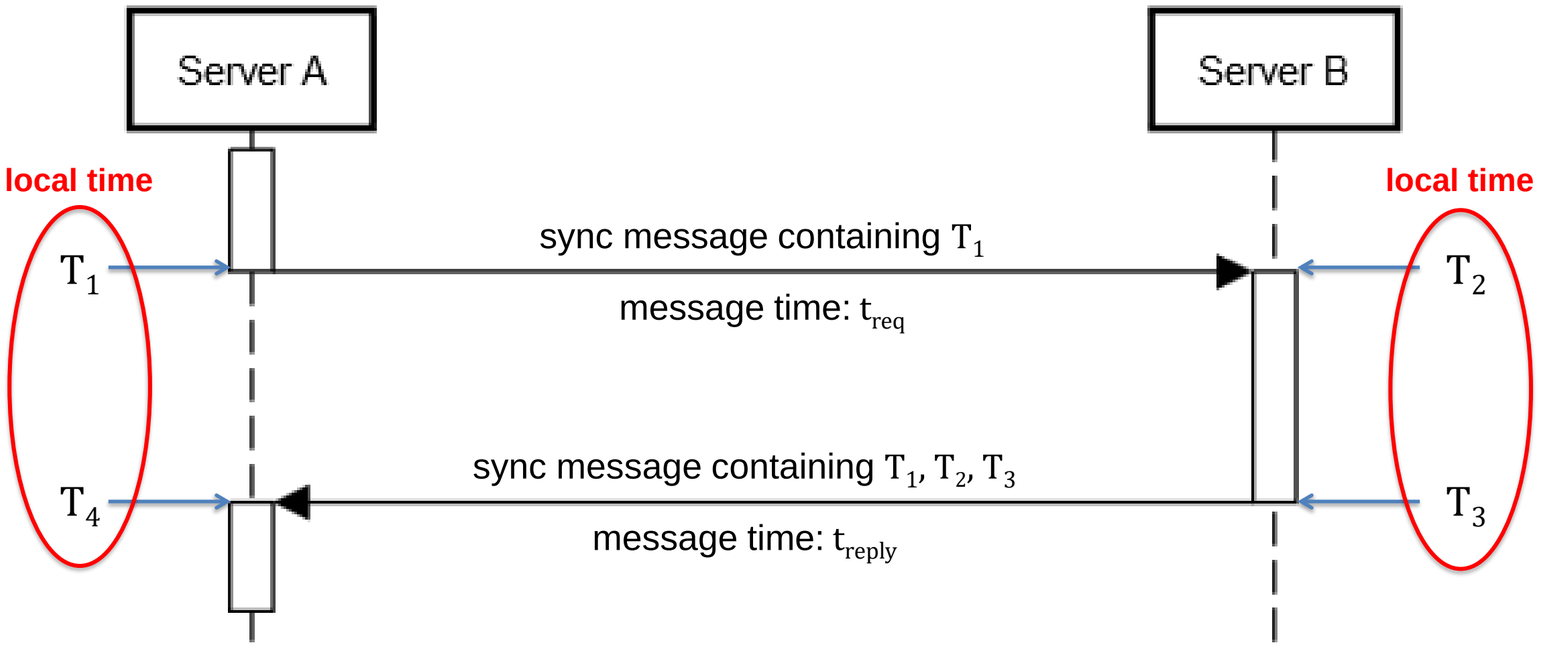
Offset estimation (1)

Three unknowns

Clock offset o

Request message time t_{req}

Reply message time t_{reply}



$$T_4 = T_3 + t_{reply} - o$$

$$T_2 = T_1 + t_{req} + o$$

Offset estimation (2)

Solving for o

$$o = T_3 - T_4 + t_{reply}$$
$$o = T_2 - T_1 - t_{req}$$

Adding the two equation

$$2o = T_3 - T_4 + t_{reply} + T_2 - T_1 - t_{req}$$

Rearraning the equation

$$o = \frac{T_3 - T_4 + T_2 - T_1}{2} + \frac{t_{reply} - t_{req}}{2}$$

The only unknowns on the right side of the equation.

Term equals zero iff $t_{reply} = t_{req}$

Offset estimation (3)

Reasonable assumption

$$\tilde{o} \approx \frac{T_3 - T_4 + T_2 - T_1}{2}$$

Worst case scenario

$$o = \frac{T_3 - T_4 + T_2 - T_1}{2} + \frac{t_{reply}}{2} \text{ or } o = \frac{T_3 - T_4 + T_2 - T_1}{2} - \frac{t_{req}}{2}$$

An upper bound on the error

$$o - \frac{t_{reply} + t_{req}}{2} \leq \tilde{o} \leq o + \frac{t_{reply} + t_{req}}{2}$$

with $t_{reply} + t_{req} = T_4 - T_3 + T_2 - T_1$

If that term becomes too big, do not synchronize and try again instead.

Network Time Protocol properties

Precision

Divergence from UTC in WANs up to 50 msec
(<30 msec for 99% of cases) and up to 1 msec in LANs (NTPv4)

Reliability / Security

Fail-stop of a *stratum-i* server → dependent *stratum-i+1* servers need to connect to a new *stratum-i* server

Digitally signed time information → authenticity & integrity

Scalability

load distribution via distributed server system

Use case:
RPC failure semantics

Implementation of *at-most-once*
with client/server synchronized time

Failure semantics

4. At-most-once

- in case of sucessful reply: exactly one execution
- in case of a failure: no multiple executions (no execution, partial execution or one execution)

→ More complex failure reaction: resend with identical ID

Implementation

- Client: manage RPC IDs

Goals

- avoid management costs for RPC call IDs
- define a time for which results are stored on the server

Use case: RPC failure semantics

Approach

RPC Client

Each RPC call contains

- unique client ID (not for each individual call)
- original call time stamp

RPC Server

- stores the last time stamp per client non-persistently
- storage of results of RPC calls
- duplicate filtering and resending of results

Server failure behavior

Fail-stop-return scenario

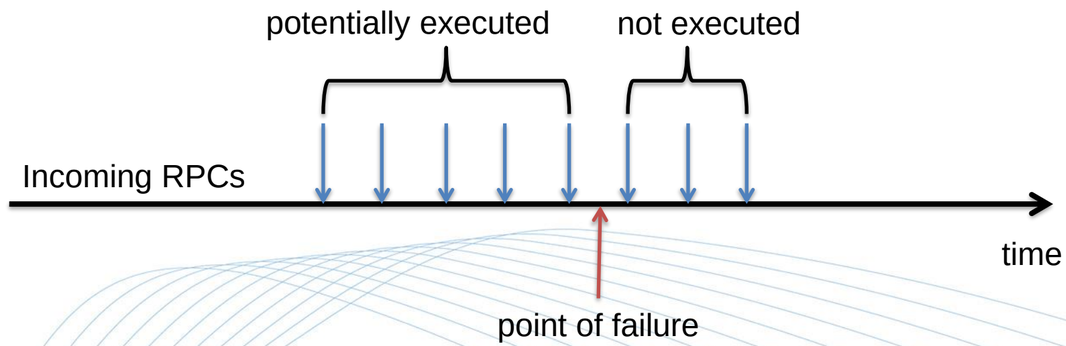
→ server does not recognize duplicates anymore after restart

Idea

- server knows a rough time estimate of its failure persitently
- recognizes all RPCs that are potential duplicate calls

Requried

- timestamp of the RPC
- failure time

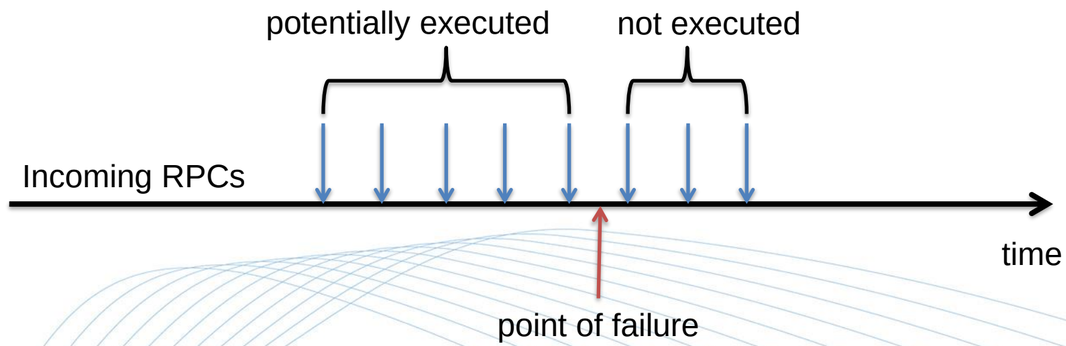


Server failure behavior

Consequence

- a not executed RPC will be considered a duplicate
- no harm done: *at-most-once* compliant, handling of ”lost” RPCs is then task of the client

→ reduction of expensive operations on persistent memory



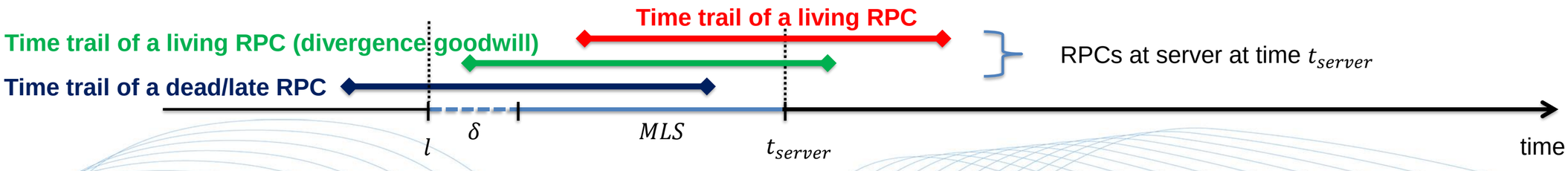
Use case: RPC failure semantics

Prerequisites

- client/server share a sychronized time with a maximum divergence δ
- client attaches to each RPC its birthdate
- global constant for the maximum life span MLS of RPCs

→ Server computes a cut-off time l (lates)

$$l = t_{server} - MLS - \delta$$



Use case: RPC failure semantics

Algorithm

- every Δt time units the server writes t_{server} into persistent memory
- at restart: $l \leftarrow stored.t_{server} + \Delta t$
  - RPCs arriving with a younger time stamp → alive
  - RPCs arriving with a older time stamp → dead (either already executed or bad luck)

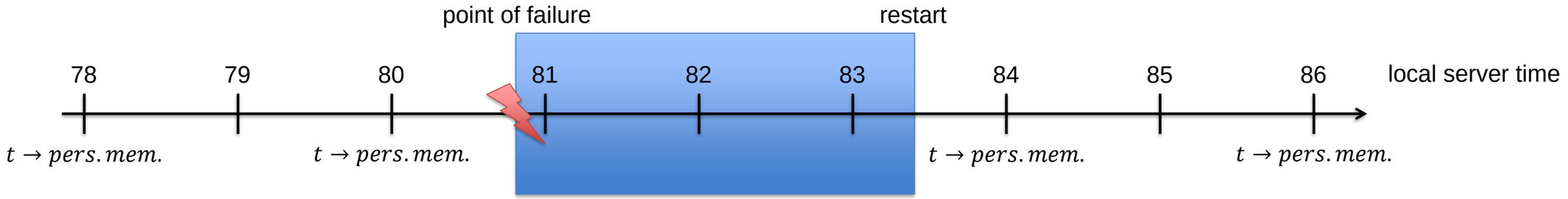
Comment

l solves the problem regarding storage time of results

→ clean-up after RPC life time is up

Example

$\Delta t = 2;\ \delta = 1; MLS = 3$



t=79: l=75; $l \leftarrow t - MLS - \delta$

t=80: l=77; $l \leftarrow t - MLS - \delta$; persistent memory $\leftarrow 80$

t=rst: l=82; $l \leftarrow$ persistant memory + Δt

t=84: l=82; persistent memory $\leftarrow 84$; trail length 2

t=85: l=82; trail length 3

t=86: l=82; persistent memory $\leftarrow 86$; trail length 4

t=87: l=83; $l \leftarrow t - MLS - \delta$

Application examples for synchronized clocks

Communication infrastructure

- efficient implementation of failure semantics
- data stream sychronization

Real-time systems

- Quality-of-service guarantees

Distributed resource management

- Failure tolerance: *leases* (*locks* with a life span)

Application examples for synchronized clocks (2)

IT security

- authorisation ticket validity (e.g. Kerberos)
- ”freshness” guarantee of cryptographic authentification protocols

Distributed file systems

- compiler management (e.g. `make`)
- local caches with time limited consistency guarantees

Logical clocks

Back to the *make all* example:

compile iff $t(last\_compile) < t(last\_filechange)$

➔ comparison of time stamps

Problems

- costs of synchronizing clocks
- fuzziness of synchonized clocks due to drift

Causal order
A ↪ B
A ”happens before” B

Better: compile iff $last\_compile \hookrightarrow last\_filechange$

Relative causal order

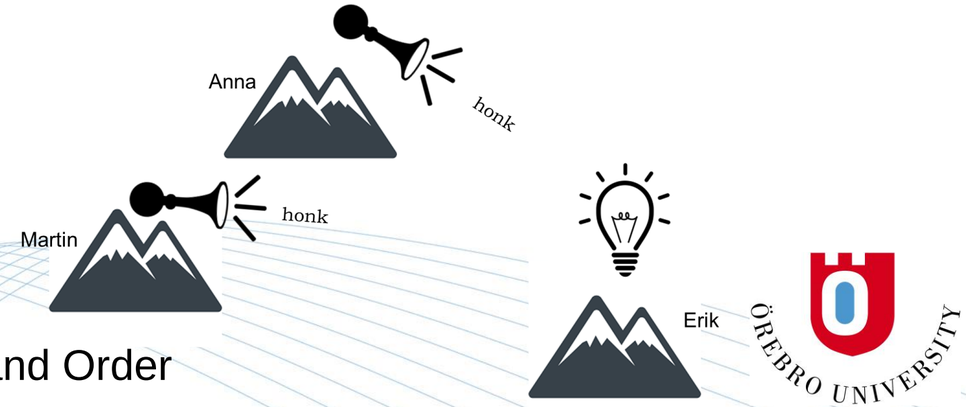
Motivation

- Consensus regarding the order/sequence of distributed events is fundamental for many algorithms
- Often it is enough to establish a relative order of observable events without referencing time

Consensus regarding the order/sequence of distributed events

For two events a and b it has to be possible to decide **system wide**, if a↦b or b↦a

→ Erik's view = Anna's view = Martin's view



⮑ *happens-before* relation

Definition

1. Local order

   If events A and B happen on the same node and A happens before B, then A ⮑B

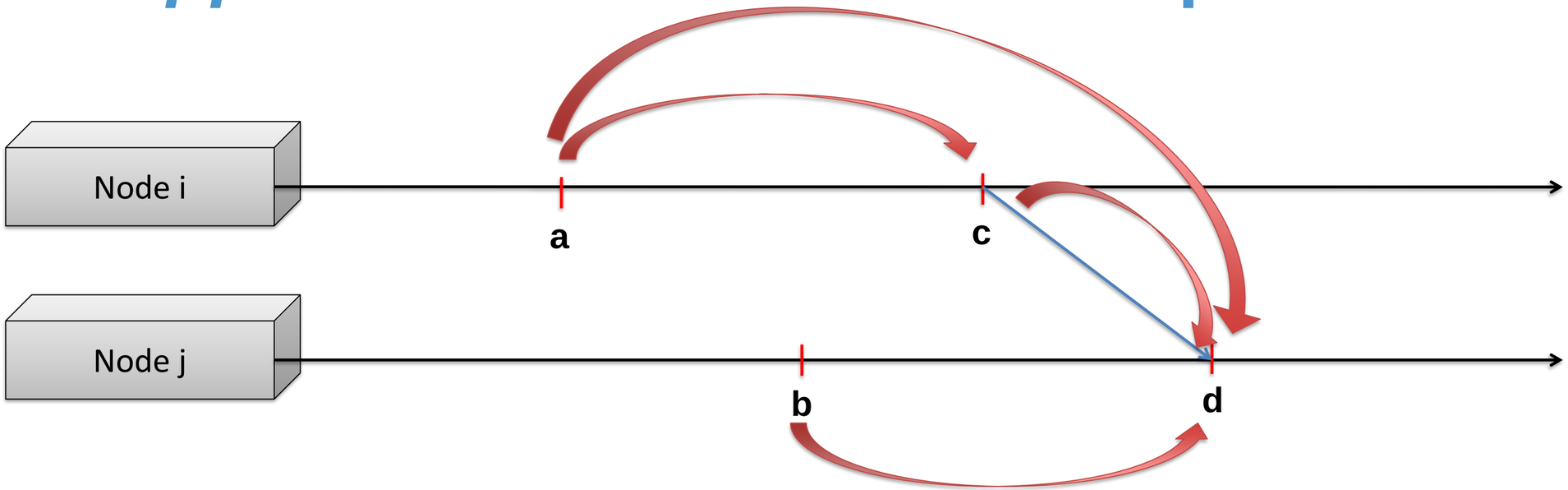
2. Distributed order

   If event A is sending a message and event B is receiving that message on a different node, then A ⮑B

3. Transitive closure

   $A \hookrightarrow B \wedge B \hookrightarrow C \Longrightarrow A \hookrightarrow C$

⮑ *happens-before* relation - Example



a ⮑ c

c ⮑ d

a ⮑ d

b ⮑ d

a ? b

b ? c

⮑ *happens-before* relation

Properties

1. Partial (non-total) order

   For two events A and B, it is possible
   that neither A ⮑B nor B ⮑A is true (i.e. concurrent events exist)

2. Strict order

   ~~A ⮑A~~

3. Potential causal order

   A ⮑B does not guarantee any causal dependency,
   ⮑ implies the possibility of causality only

Four variants of causal order

	partial order	**total order**
potential causal order	partial potential causal order (ppco)	total potential causal order (tpco)
causal order	partial causal order (pco)	total causal order (tco)

Formal ➥ *happens-before* relation

${e_i}^x$ denotes an event x in node i

$$ {e_i}^x \rightarrow {e_j}^y \Leftrightarrow \begin{cases} (i = j \wedge x < y) \vee \\ (i \neq j \wedge \exists m\colon\ {e_i}^x = send(m) \wedge\ {e_j}^y = receive(m)) \vee \\ (\exists\ {e_k}^z\colon\ {e_i}^x \rightarrow {e_k}^z \wedge\ {e_k}^z \rightarrow {e_j}^y) \end{cases} $$

Comments

- $e \rightarrow f$ $\Longrightarrow$ e is potentially the cause of f
- $\neg(e \rightarrow f)$ $\Longrightarrow$ e is definitely NOT the cause of f

The set of all events H together with the ➥ relation of those events $(H, \rightarrow)$ constitute the event-based model of a distributed algorithm

Example: Analysis of distributed systems

Cause and effect

- What events are the cause for a given event?
- What events are the effect of a given event?

1. Causal past

   Set of all events that are the potential cause for e {f | f ⮑ e}

2. Causal future

   Set of all events that the potential effect of e {f | e ⮑ f}

3. Causal independence

   $\neg(e \hookrightarrow f) \wedge \neg(f \hookrightarrow e) \implies$ e and f are causal independent

Implementing a causal order

Goal

Implementation of $\rightarrowtail$ by associating discrete values with each event

Required

- All nodes can calculate an identical sequence function o, so that for events e and f holds: $e \rightarrowtail f \implies o(e) < o(f)$
- Ideally, all nodes can calculate an identical sequence function o, so that for events e and f holds: $e \rightarrowtail f \iff o(e) < o(f)$

Logical clocks

A **logical clock** is a strictly monotonic increasing counting function $lc: \rightarrow \mathbb{N}$, with each sequence …, lc_i,…, lc_j,…subject to

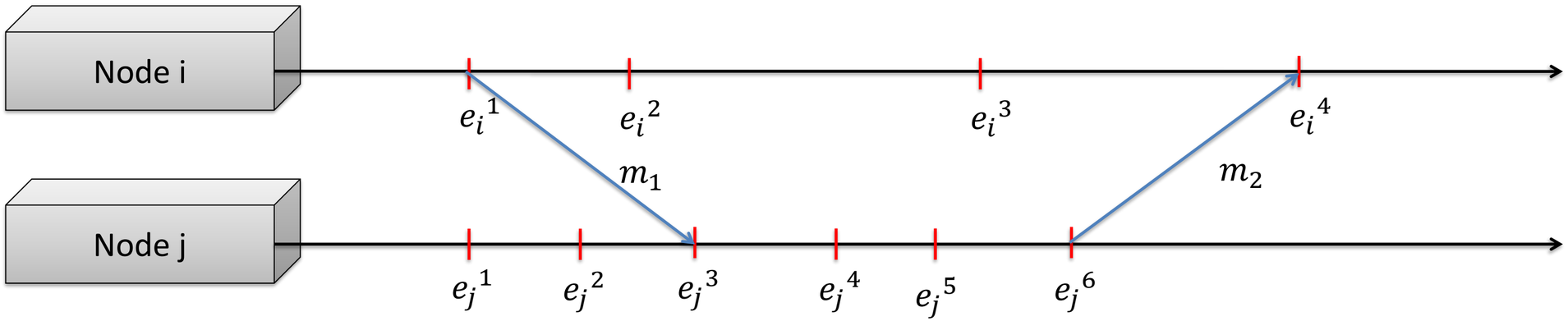
$$\forall i, j, i < j : lc_i < lc_j$$

Comments

- Each call of lc returns a higher value then the previous call
- There is no relationship to any real time (UTC or other)

Logical clocks

- easy to implement for a single node (simply a counter)
- not enough to implement the ➥ relation



send(m_1)➥receive(m_1), with $o(e_i^1)=1 < o(e_j^3)=3$

send(m_2)➥receive(m_2), with $o(e_j^6)=6 < o(e_i^4)=4$

Lamport's Algorithm (1)

Goal

Implementation of ↪ using logical clocks

Approach

- local logical clocks (local order) plus
- synchronization of all logical clocks (distributed order)

Lamport's Algorithm (2)

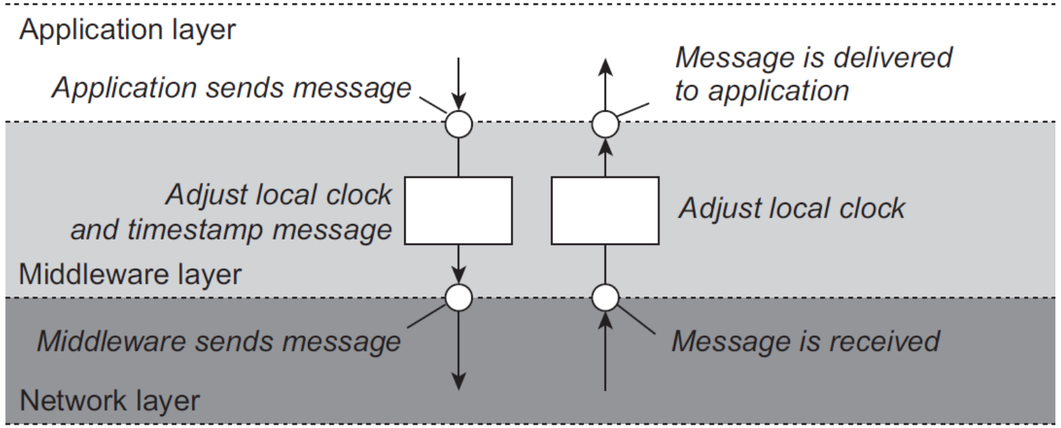
Idea

- two non-interacting nodes cannot recognize logical clock differences
- when nodes exchange messages, logical clocks can sychronize as a side effect → cheap (”piggy-backing”)

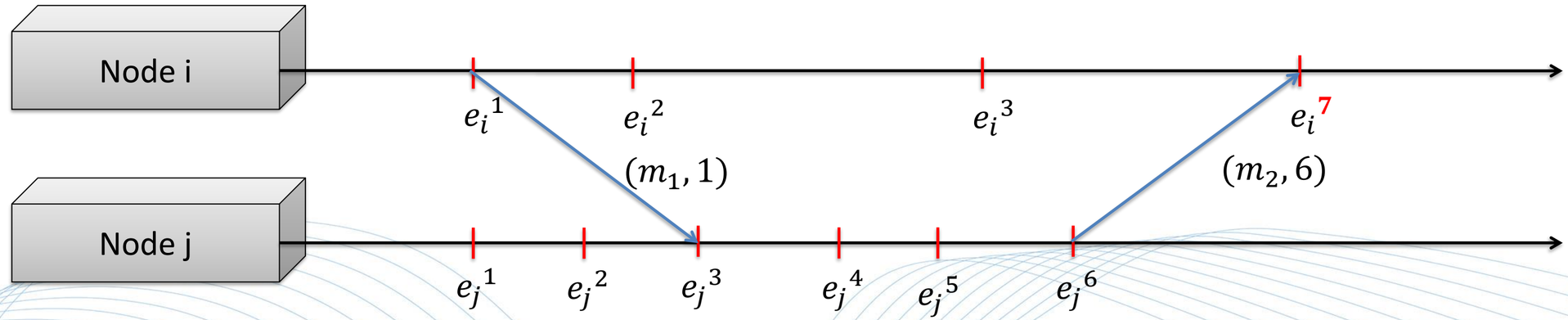
→ Implementing a partial potential causal order (ppco) with

$$e \rightarrowtail f \implies ppco(e) < ppco(f)$$

Lamport's Algorithm (3)



- each node n_i uses a local clock lc_i
- $ppco(e_i) = lc_i$, each event in n_i is timestamped by lc_i: ${e_i}^{lc_i}$
- lc_i is added to each message sent by n_i
- on arrival of a message in node n_j
  1. n_j sychronizes its local logical clock $lc_j \leftarrow \max(lc_i, lc_j)$
  2. n_j tags the receive event with $lc_j + 1$

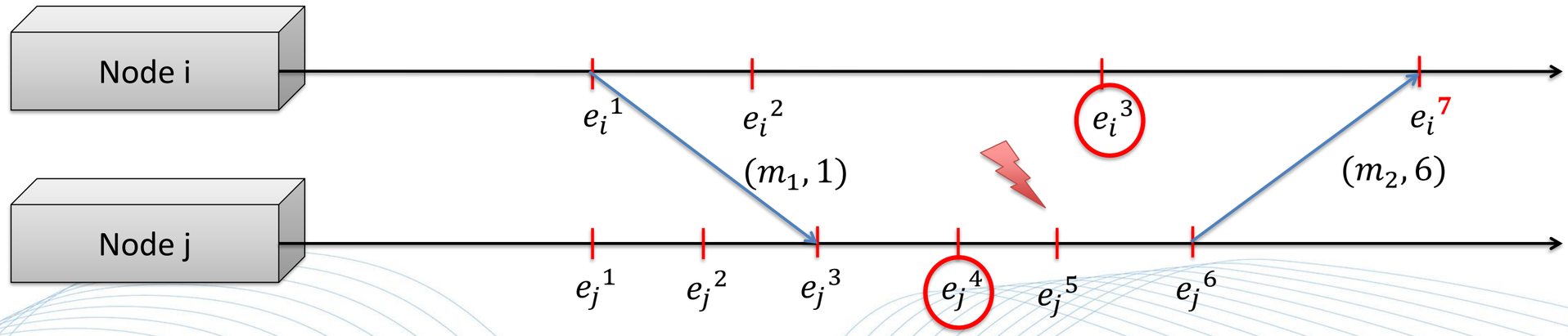


Lamport's Algorithm (3)

Caveat

$$e \rightarrow f \Rightarrow ppco(e) < ppco(f)$$

- allows concurrent events
- $ppco(e) < ppco(f) \Rightarrow e \rightarrow f$ is not guaranteed!



Partial causal order

Goal

Implementing a partial causal order (pco) with

$$e \rightarrowtail f \Leftrightarrow pco(e) < pco(f)$$

Idea

Local bookkeeping of <u>all</u> logical clocks

→ each node n_i stores instead lc_i a vector vc_i (vector clock of n_i) with the last known local times

$$vc_i = \begin{bmatrix} lc_1 \\ \vdots \\ lc_i \\ \vdots \\ lc_n \end{bmatrix}$$

Partial causal order (2)

- each node n_i uses a local vector clock vc_i
- $pco(e_i) = vc_i$, each event in n_i is timestamped by vc_i: ${e_i}^{vc_i}$
- vc_i is added to each message sent by n_i
- on arrival of a message in node n_j

1. n_j sychronizes its local vector clock

$$\forall k \neq j: vc_j[k] \leftarrow \max(vc_i[k], vc_j[k])$$

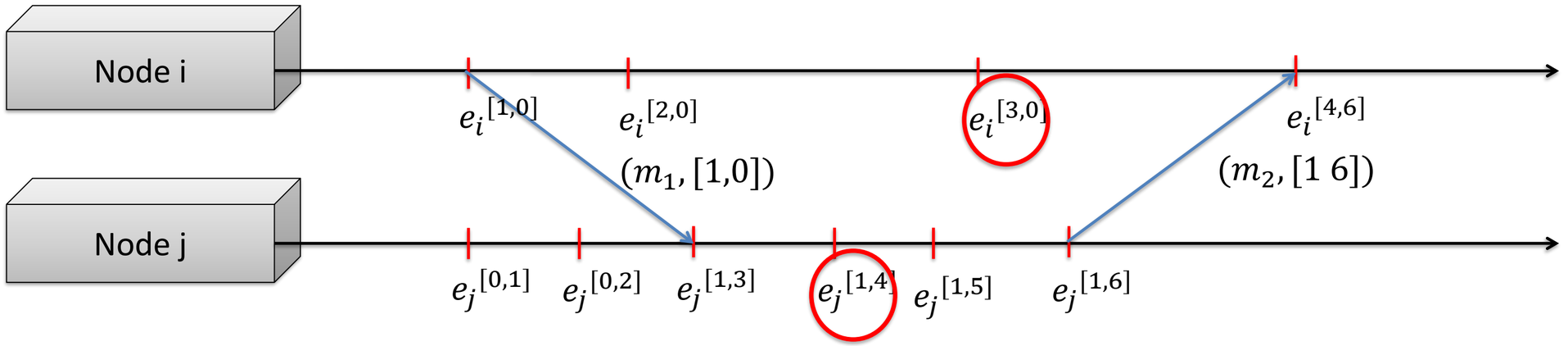
2. n_j tags the receive event with vc_j

Consequence

$$\forall\, e_i \neq e_j:\ e_i \rightarrow e_j \Leftrightarrow pco(e_i) < pco(e_j) \Leftrightarrow vc_i < vc_j$$

$$vc_i < vc_j \Leftrightarrow \forall \boldsymbol{k = 1..n}: \boldsymbol{vc_i[k] \leq vc_j[k]} \wedge \exists \boldsymbol{vc_i[k] < vc_j[k]}$$

Vector clock example



Uncomparable events

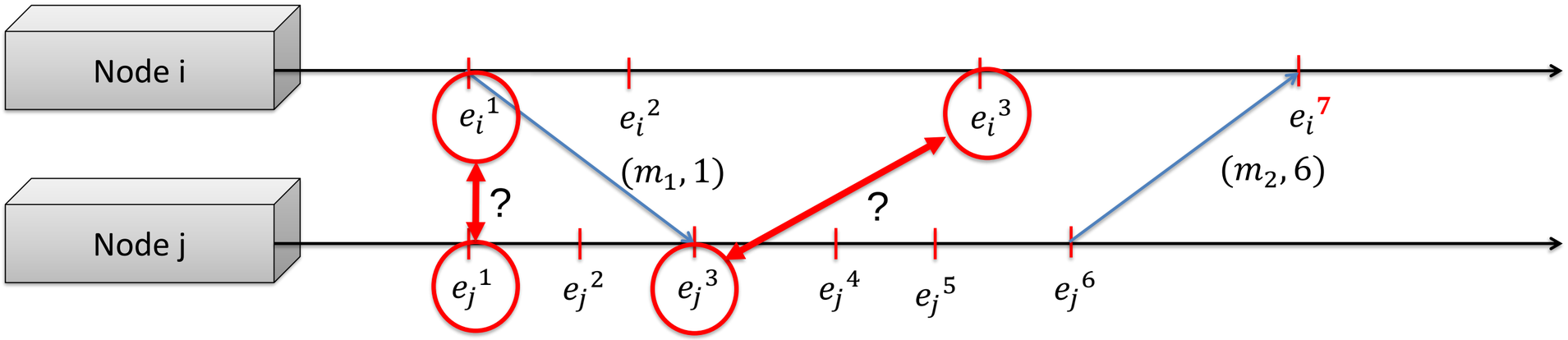
neither $pco(\ e_i^{[3,0]}) < pco(\ e_j^{[1,4]})$ nor $pco(\ e_j^{[1,4]}) < pco(\ e_i^{[3,0]})$ is true

marked events are concurrent → partial order

Total order

Problem

Partial order is not always sufficient



$ppco(e_i^1) = ppco(e_j^1)$ and $ppco(e_i^3) = ppco(e_j^3)$, similar for pco

Total potential causal order

Definition

- I is a finite index set with a total strict order over ”<”
- all nodes n_i have a unique identifier $i \in I$ → globally unique ID
- lc_i is the synchronized local logical clock of n_i

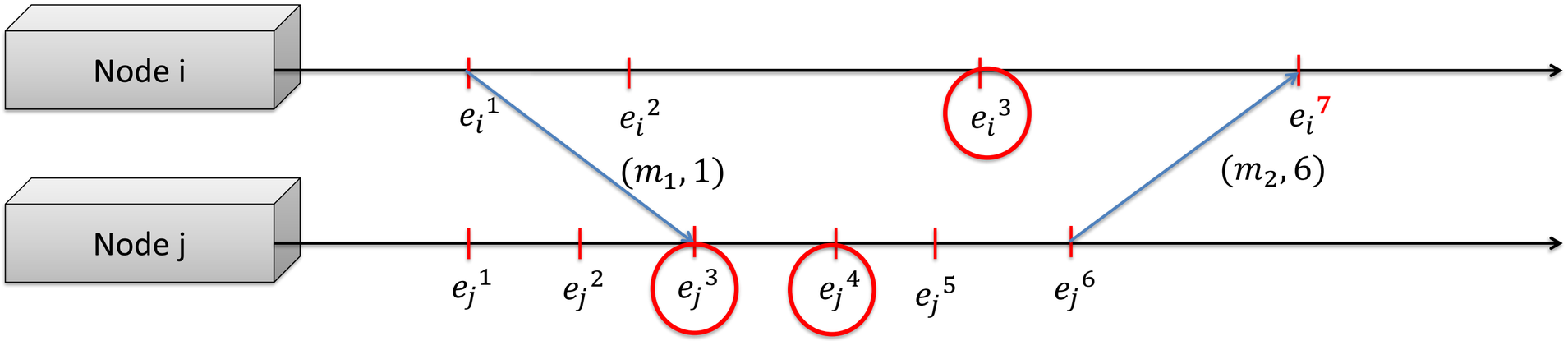
The total potential causal order for event e_i in node n_i is

$$tpco(e_i) = \left(i, lc_i(e_i)\right)$$

and the relation $tpco(e_i) < tpco\left(e_j\right)$ is defined as

$$tpco(e_i) < tpco\left(e_j\right) \Leftrightarrow \begin{cases} lc_i(e_i) < lc_i\left(e_j\right) & \text{| if } lc_i(e_i) \neq lc_i\left(e_j\right) \\ i < j & \text{| otherwise} \end{cases}$$

Total potential causal order - example



$tpco(e_i^3) < tpco(e_j^3)$ and $tpco(e_i^3) < tpco(e_j^4)$

➔ The total order is **not** connected to the actual physical timing of events

Total causal order (*tco*)
Same as *tpco* only with vector clocks instead of logical clocks!

Still has its merit, for all n_i the computation of $tpco$ is equal

Summary

Temporal order

- Fuzzy: a, b at the same time $\Leftrightarrow t(a) - \rho \leq t(b) \leq t(a) + \rho$
- Total
- Approximate real-time
- Expensive depending on sychronization regime

Algorithms

- Active and passive time server
- Infrastructure: NTP

Summary

Causal order

- precise
- relative order with no relation to the actual time
- 4 specific variants:
  partial vs. total + potential causal vs. causal
- with different costs (ppco → cheap; tco → expensive

Based on

- Sychronized local logical (vector) clocks
- task IDs