University of Freiburg, Germany
Department of Computer Science

# Distributed Systems

## Chapter 3 Time and Global States

Christian Schindelhauer

05. May 2014

# 2: Time and Global States

How can distributed processes be coordinated and synchronized, e.g.

- when accessing shared resources,
- when determining the order of triggered events?

## The importance of time

- Distributed systems do not have only one clock.
- Clocks on different machines are likely to differ.
- Physical versus logical time.

# 2.1: Physical Time

## Example; distributed software development using UNIX `make`

- Computer sets its clock back after compiling a source file
- User edits the source file
- `make` assumes the source file has not been changed since compilation
- `make` will not recompile

## TAI and UTC

- International Atomic Time TAI: mean number of ticks of caesium 133 clocks since midnight Jan. 1, 1958 divided by number of ticks per second 9,192,631,770.
- Problem: 86,400 TAI seconds (corresponding to a day) are today 3 msec less than a mean solar day (because solar days are getting longer because of tidal forces).
- Solution: whenever discrepancy between TAI and solar time grows to 800 msec a leap second is added to solar time.
- The corresponding time is called Universal Coordinated Time UTC.
- UTC is broadcast every second as a short pulse by the National Institute of Standard Time NIST. It is broadcast by GPS as well.

## Time in distributed systems

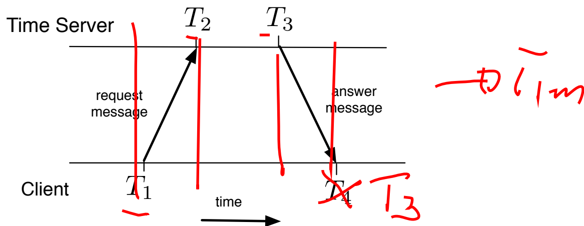
- Each computer $p$ is equipped with a local clock $C_p$, which causes $H$ interrupts per second. Given UTC time $t$, the clock value of $p$ is given by $C_p(t)$.
- Let $C'_p(t) = \frac{dC_p}{dt}$
- Ideally, $C'_p(t) = 1$, real clocks have an error of about $\pm 10^{-5}$ (10 ppm)
- If there exists some constant $\rho$ such that

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho,$$

  $\rho$ is called the maximum drift rate.
- If synchronized $\Delta t$ ago, two clocks may differ at most by $2\rho\Delta t$.
- To ensure synchronization within precision $\delta$, then they need to be synchronized at least every $\frac{\delta}{2\rho}$ seconds.

## Network Time Protocol NTP

- Assumption, one system $C$ is connected to a UTC server. This system is called time-server.
- Each machine $C$ , every $\frac{\delta}{2\rho}$ seconds, sends a time request to the time-server, which immediately responds with the current UTC.
- machine $C$ sets its time to be $T_3$,
  - where $T$ is the received time
  - RTT is the round trip time

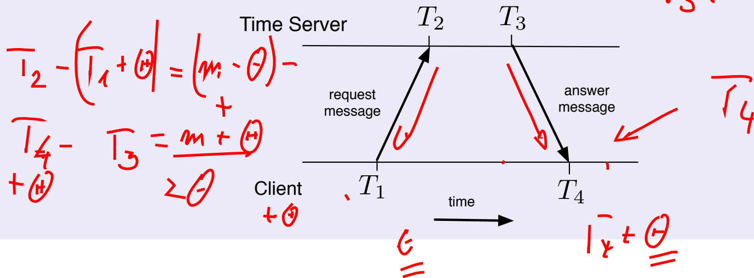


## Problems and solutions

- Problem: time may run backwards!
- Solution: clocks converge to the correct time.
- Problem: Because of message delays, reported time will be outdated when received by a client.
- Solution: Try to find a good estimation for the delay.
- . . . (next slide)

## Problems and solutions

- Problem: Because of message delays, reported time will be outdated when received by a client.
- Solution: Try to find a good estimation for the delay.
  - **Algorithm of Flaviu Cristian**
    Use $\frac{(T_4 - T_1)}{2}$ if no other information is available.
  - If interrupt handling time $I$ is known, use $\frac{(T_4 - T_1 - I)}{2}$.
  - . . . else . . .

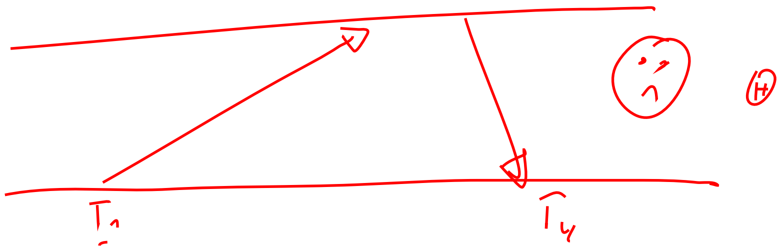


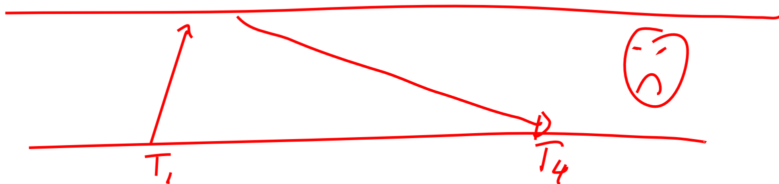
## Problems and solutions

- Problem: Because of message delays, reported time will be outdated when received by a client.
- Solution: Try to find a good estimation for the delay
  **NTP: Network Time Protocol**
  - . . . else . . .
  - To adjust $A$ to $B$, use piggybacking:
  - $A$ sends a request to $B$ timestamped with $T_1$.
  - $B$ records the time of receipt $T_2$ (taken from its local clock) and returns a response timestamped with $T_3$ and piggybacking $T_2$.
  - $A$ records the time of arrival $T_4$. The propagation time from $A$ to $B$ is assumed to be the same as from $B$ to $A$, $T_2 - T_1 \approx T_4 - T_3$.
  - The offset $\theta$ of $A$ relative to $B$ can be estimated by $A$:

$$\theta = T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_4 = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

  - If $\theta < 0$, in principle, $A$ has to set its clock backwards.
- Take the measures several times and compute the mean while ignoring outliers.


T4


T1
T4

## Examples: *A* has to be adjusted to *B*.

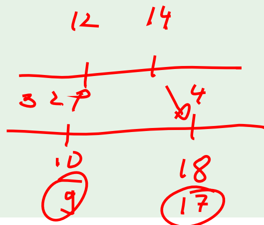
$A$ sends a request to $B$ timestamped with $T_1$. $B$ records the time of receipt $T_2$ (taken from its local clock) and returns a response timestamped with $T_3$ and piggybacking $T_2$. $A$ records the time of arrival $T_4$.
The offset $\theta$ of $A$ relative to $B$ can be estimated by $A$:

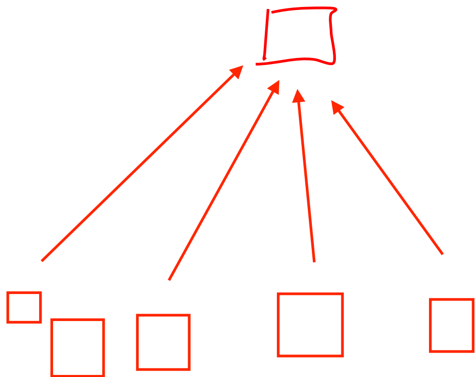
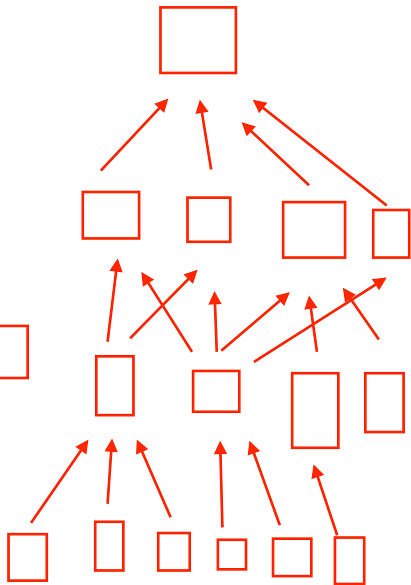
$$\theta = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

(a) No need for adaption detected.
$T_1 = 10,\ T_2 = 12,\ T_3 = 14,\ T_4 = 16 \Longrightarrow \theta = 0.$

(b) $A$ has to slow down.
$T_1 = 10,\ T_2 = 12,\ T_3 = 14,\ T_4 = 18 \Longrightarrow \theta = -1.$

(c) $A$ has to hurry up.
$T_1 = 10,\ T_2 = 12,\ T_3 = 14,\ T_4 = 14 \Longrightarrow \theta = 1.$

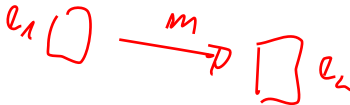


## On scalability of NTP (roughly)

- NTP is an Internet standard (RFC 5905).
- NTP service is provided by a network of servers.
- Primary servers are directly connected to a UTC-source.
- Secondary servers synchronize themselves with primary servers.
- This approach is applied recursively leading to several layers.
- Server $A$ adjusts itself to server $B$ if $B$ is assigned a lower layer than $A$.
- The whole network is reconfigurable and thus is able to react on errors.

# 2.2: Logical Time



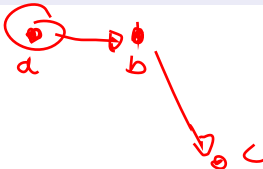
## Why?

- Getting physical clocks absolutely synchronized is not possible.
- Thus it is not always possible to determine the order of two events.
- For such cases logical time can be used as a solution.
  - If two events happen in the same process they are ordered as observed.
  - If two processes interchange messages, then the sending event is always considered to be before the receiving event.

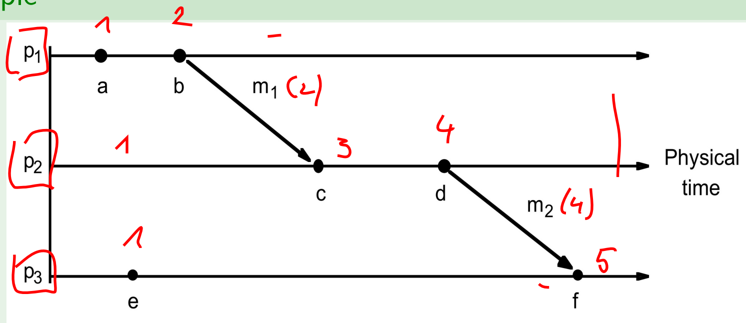
## Lamport's happened-before relation (causal ordering)

- If two events $a, b$ happen in the same process $p_i$ they are ordered as observed and we write $a \rightarrow_i b$.
  Moreover, this implies $a \rightarrow b$ systemwide.
- If two processes interchange messages, then the sending event $a$ is always considered to be before the receiving event $b$, thus $a \rightarrow b$.
- Whenever $a \rightarrow b$ and $b \rightarrow c$, then also $a \rightarrow c$.

Events not being ordered by $\rightarrow$ are called concurrent.

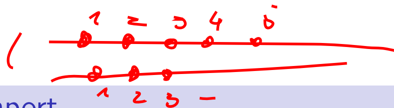


## Example




from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

We conclude $a \to b, b \to c, c \to d, d \to f, a \to f$, however not $a \to e$; $a, e$ are concurrent.



## Algorithm of Leslie Lamport

- Let $L_i(e)$ denote the time stamp of event $e$ at process $P_i$.
- When a new event $a$ occurs in process $P_i$:

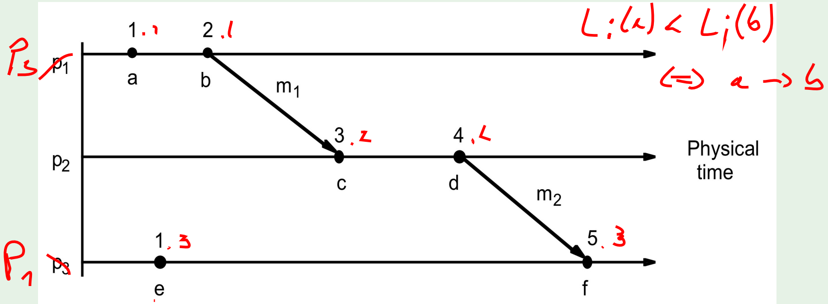
$$L_i := L_i + 1$$

- Each message $m$ sent from $P_i$ to $P_j$ is piggybacked by the timestamp $L_i(a)$ of the send-event $a$.
- When $(m, t_a)$ is received by $P_j$, $P_j$ adjusts its logical clock $L_j$ to the logical clock of $P_j$.

$$L_j := \max\{L_j, t_a\}$$

and increments $L_j$ for the received message event.

## Three clocks with application of Lamport's algorithm.




from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

$L(a) = L(b)$

$\Rightarrow a \parallel b$

$\Rightarrow$ a and b

lie on different

Process

$a \to b \Rightarrow L(a) < L(b)$

$b \to a \Rightarrow L(b) < L(a)$

Yes

NO

?

## Totally ordered logical clocks

- Extend the Lamport clock for each process $P_i$:
- Clock values must be systemwide unique
  - for this the clock value $L_i$ is referred to with the process id $i$, i.e. $(L_i, i)$
  - all distinct clocks $L_i$ can be unified into a system clock $L$.
- Define the total ordering

$$(T_i, i) < (T_j, j) \quad :\Longleftrightarrow \quad \begin{cases} i < j & \text{if } T_i = T_j \\ T_i < T_j & \text{else} \end{cases}$$

- So, we translate a partial ordering into a total ordering
- However from the total ordering $L(a) < L(b)$ one cannot conclude $a \to b$.

## Mattern's Vector Clocks

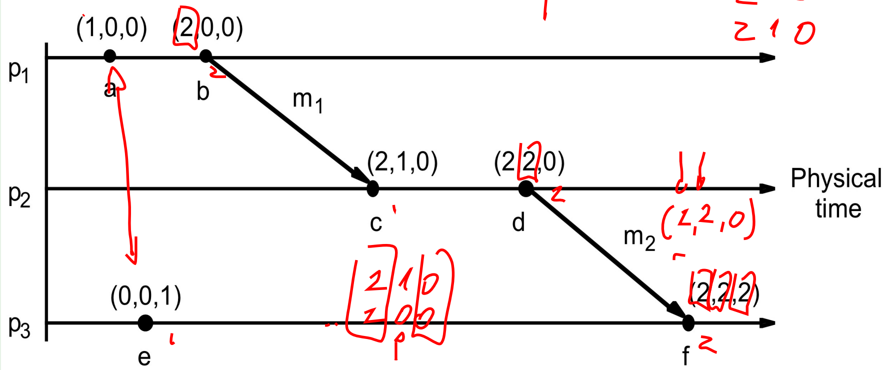
- Vector clock for a system of $n$ processes: array of $n$ integers.
- Each process $P_i$ keeps its own vector clock $V_i$ which is used to timestamp local events.
- Processes piggyback their own vector clock on messages they send.
- Update rules for vector clocks:

VC1: Initially, $V_i[j] := 0$ for $i, j \in \{1, \ldots, n\}$
VC2: $P_i$ timestamps prior to each event: $V_i[i] := V_i[i] + 1$.
VC3: $P_i$ sends the value $t = V_i$ with each message.
VC4: When $P_i$ receives some message piggybacked with timestamp $t$, it sets

$$V_i[j] := max\{V_i[j], t[j]\} \quad \text{for } i = 1, 2, \ldots, n$$

- $V_i[i]$ is the number of events that $P_i$ has timestamped.
- $V_i[j]$ for $i \neq j$ is the number of events that have occured at $P_j$ to the knowledge of $P_i$.

## Vector Clock Example




from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg