# Real Time Systems – SS2016

**Prof. Dr. Karsten Weronek**
**Faculty 2**
**Computer Science and Engineering**

## Synchronization

# NTP

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

**Network Time Protocol** (NTP) is a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks. In operation since before 1985, NTP is one of the oldest Internet protocols in current use.

NTP is intended to synchronize all participating computers to within a few milliseconds of Coordinated Universal Time (UTC). It uses a modified version of Marzullo's algorithm to select accurate time servers and is designed to mitigate the effects of variable network latency. NTP can usually maintain time to within tens of milliseconds over the public Internet, and **can achieve better than one millisecond accuracy in local area networks under ideal conditions**. Asymmetric routes and network congestion can cause **errors of 100 ms or more**.[

**The Precision Time Protocol** (PTP) is a protocol used to synchronize clocks throughout a computer network. On a local area network, it achieves clock accuracy in the sub-microsecond range, making it suitable for measurement and control systems.

PTP was originally defined in the IEEE 1588-2002 standard, officially entitled *"Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems"* and published in 2002. In 2008, IEEE 1588-2008 was released as a revised standard; also known as PTP Version 2, it improves accuracy, precision and robustness but is not backward compatible with the original 2002 version

# Clocks in Computers are not perfect

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

- Synchronization-error (Skew(Versatz))
- The values of two clocks differ Different pace (Drift-Rate)
- The values of two clocks diverge
- simple quartz clock approx. one second in 11-12 days ($10^{-6}$ s/s) (s: second)
- high-precision quartz clock approx. $10^{-8}$ s/s

Result of drift:

The clocks have to be synchronized periodically!

# Relationship

- internal synchronization of clocks does not necessarily result in external sychnronization as
  all clocks can deviate from the reference clock

- External synchronization doubles the error compared to internal synchronization

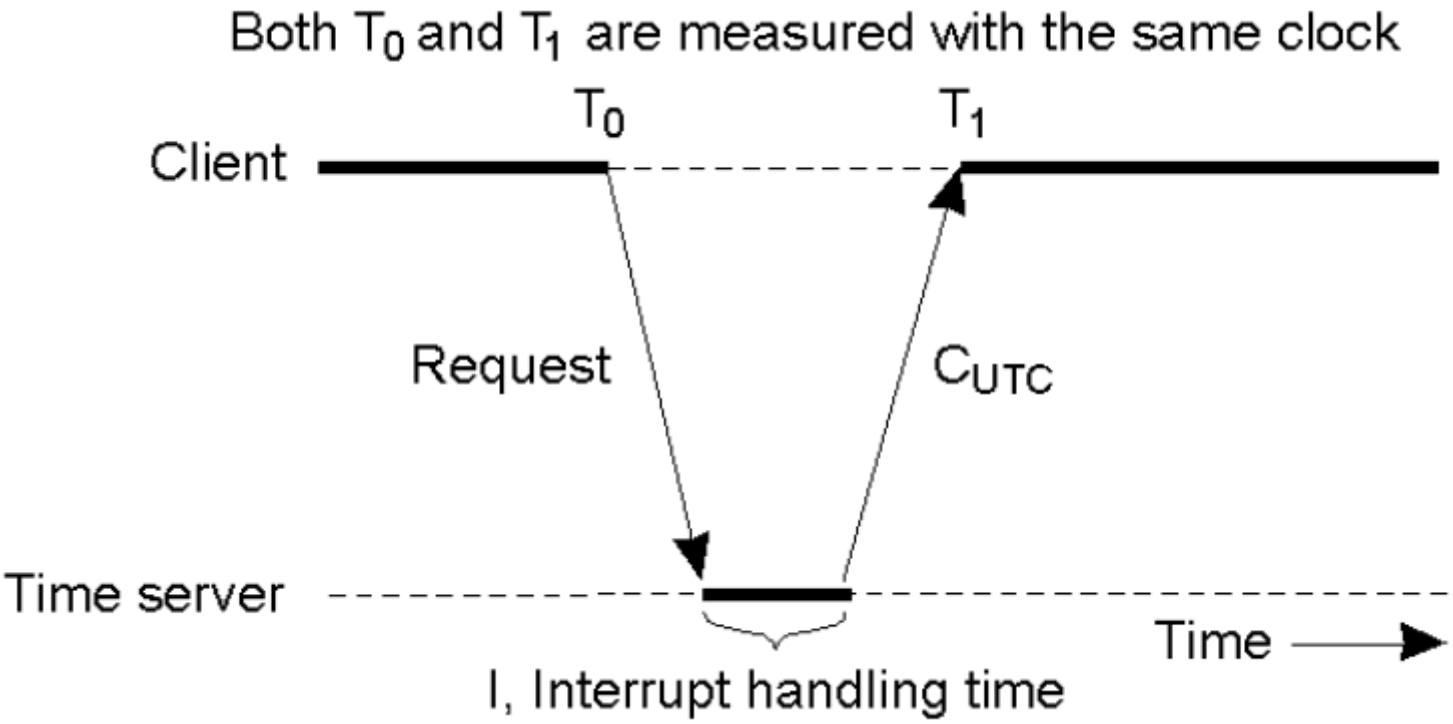
# Algorithm of F. Christian

Assumption: there is a UTC-receiver within the overall system

- this UTC-receiver operates as a Time-Server
- external synchronization

All other machines periodically send
a request for the current time to the server.

The server replies as soon as possible with the current value of UTC



# Algorithm of F. Christian

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

## Problem

- The transmission of messages consumes a non-zero amount of time.
  At the time the answer to the request is received,
  it is already outdated

## Approach of Christian:

Try to measure the time to transmit a message

## Best Estimate if no other information is available:

$(T_1 - T_0)/2$

## If the time I used by the server to process the request is known:

$(T_1 - T_0 - I)/2$

## Variant: Frequently calculate this value,

omit outliers and

use the average of the remaining values

# Adjustment of Clocks

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Naive approach:

Use the received UTC-value,
correct it according to the estimated delay and
set the local clock to the resulting value

This results into problems if the local clock was to fast:

- time can never proceed backwards.
  No point in time can ever occur twice!
- monotonicity is required: $t' > t \Rightarrow C(t') > C(t)$; C: time-stamp
- We must not adjust our clock to a time value that
  represents a point in time previous to the one perceived up the adjustment

Solution: The clocks are gradually adjusted until
the desired time value is reached (the clock is slowed down)

- For example during the adjustment phase
  each clock-tick represents only 9 ms instead of 10 ms

# Berkeley-Algorithm

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Important if no UTC-receiver is available

- internal synchronization

One machine is the coordinator.

The coordinator asks all machines for their local times

After taking into account the message transmission times,

the average of the received clock values is calculated.

- outliers (with respect to clock value or message transmission time) are eliminated

Each machine is informed about the difference of its local clock to the average time and adjusts its clock properly

- Using the difference of clock values is less error-prone (fehleranfällig) than using absolute clock values

# Logical Time (Lamport)



It is impossible to synchronize physical clocks in a distributed system absolutely.

As a result it is not possible to order events based on physical clocks.

Many applications need information about order but
do have to be aware of the physical time

Solution: Logical time

- If two events take place within the same process, they can be ordered
- If two processes exchange a message,
the sending event is preceding the receiving event

# System-Model for Logical Time

The distributed system consists of N processes $p_i$

Processes communicate only via messages.

Possible events:

- sending a message
- receiving a message
- local event of a process

On the events e of a single process $p_i$
we can define a total order $\rightarrow_i$ with

- $e \rightarrow_i e'$ if e in $p_i$ occurred before e'

# Realization

Logical clocks are realized by monotonically increasing counters.

Each Process $p_i$ has its own logical clock that is used when events a occurs and produces a value $C_i(a)$.

This value has to be adjusted to form a globally unique timestamp C(a)

The algorithm of logical clocks has to ensure that:
if $a \rightarrow b$, then $C(a) < C(b)$

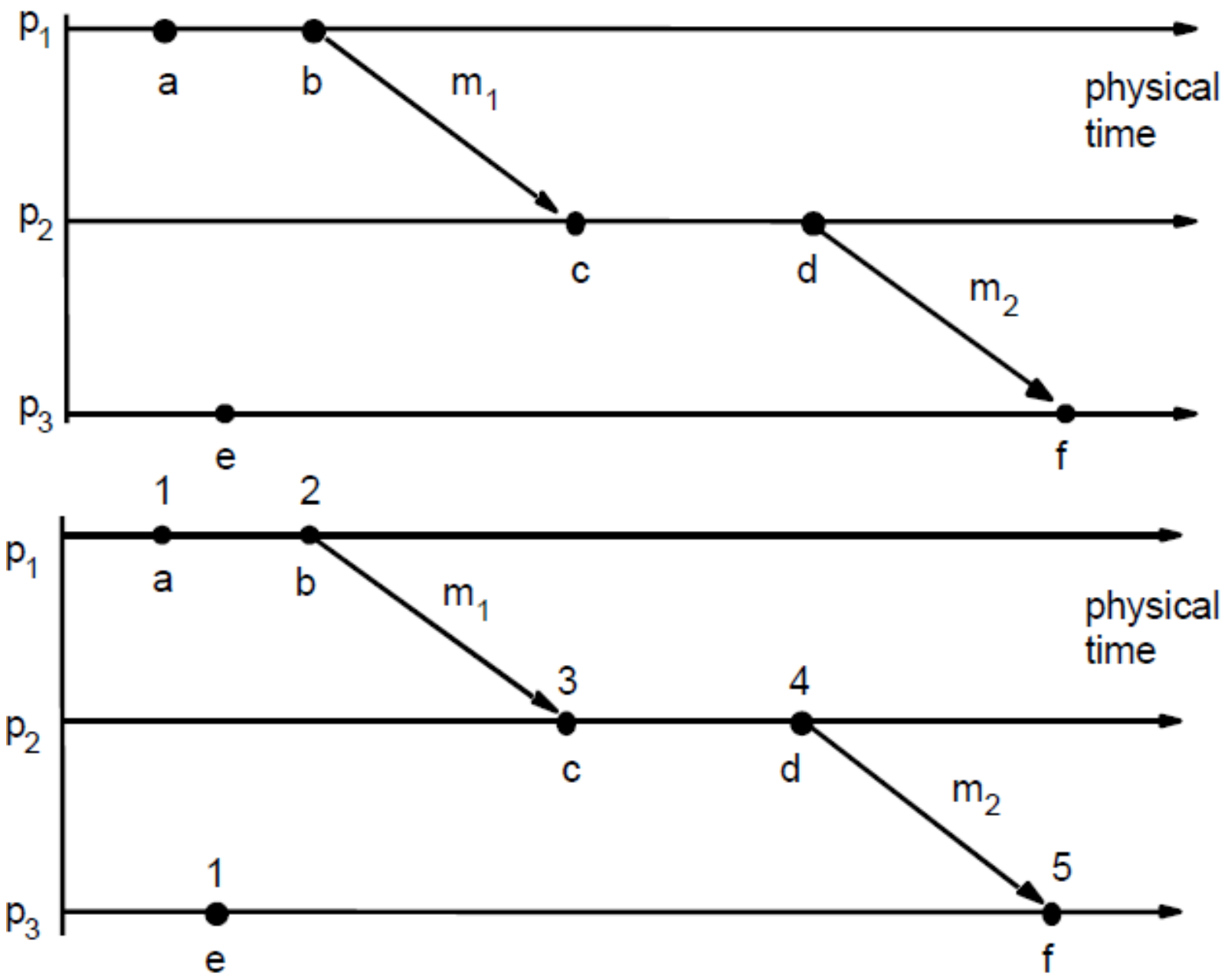
# Algirithm for Lamport Clock



Use the following rules:

- $C_i$ is incremented before each new event in $p_i$ by one:
  $C_i := C_i + 1$
  This value is the timestamp of the event
- If process $p_i$ sends a **message m**,
  the current value of $t = C_i$ is included
- When receiving (**m**, t) $p_k$ calculates the new value of its clock as
  $C_k := \max(C_k, t)$
  and applies the first rule to assign the timestamp for receive(m)

# Example Lamport Time



## Vectortime is an extension of Lamport-Time

- Using Vectortime we can see from the timestamps whether two events are ordered by the happened-before-relation or concurrent

## Approach

- Each process $p_i$ has a vectorclock $V_i$ in form of N counters
  one counter for each process in the system
- Initialization:
  $V_i = (0, ..., 0)$ for all $i = 1, ..., N$
- Assignment of timestamps:
  before each event in pi : $V_i[i] = V_i[i] + 1$
  the resulting value of $V_i$ is the timestamp of the event
- Send:
  If a process $p_i$ sends a message m, it includes the current value $t = V_i$
- Receive:
  When receiving (m, t) $p_k$ calculates the new value of its vectorclock as
  $V_k[j] = \max(V_k[j], t[j])$ for all $j = 1, ..., N$
  and applies the above rule to assign the timestamp for receive(m)

# Order with Vectortime

Order of timestamps by comparing their components.

For two **timestamps V** and **V'** holds:

- $V \leq V'$ if $V[j] \leq V'[j]$ for all $j = 1, ..., N$
- $V = V'$ if $V[j] = V'[j]$ for all $j = 1, ..., N$
- $V < V'$ if $V \leq V'$ and $V \neq V'$

With V(a) we denote the timestamp of event a

Properties:

- Vectortime maintains causality:
  $a \rightarrow b$ implies $V(a) < V(b)$
- Looking at the timestamp we can infer the causal relationship:
  $V(a) < V(b)$ implies $a \rightarrow b$

# Example Vector Time