TDDD25
Distributed Systems

Distributed Real-Time Systems

Christoph Kessler

IDA
Linköping University
Sweden

LINKÖPING UNIVERSITY

Agenda

DISTRIBUTED REAL-TIME SYSTEMS

1. What is a Real-Time System?
2. Distributed Real Time Systems
3. Predictability of Real-Time Systems
4. Process Scheduling
5. Static and Dynamic Scheduling
6. Clock Synchronization
7. Universal Time
8. Clock Synchronization Algorithms
9. Real-Time Communication
10. Protocols for Real-Time Communication

Real-Time Systems

A **real-time system** is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations but also on the time when the results are produced.

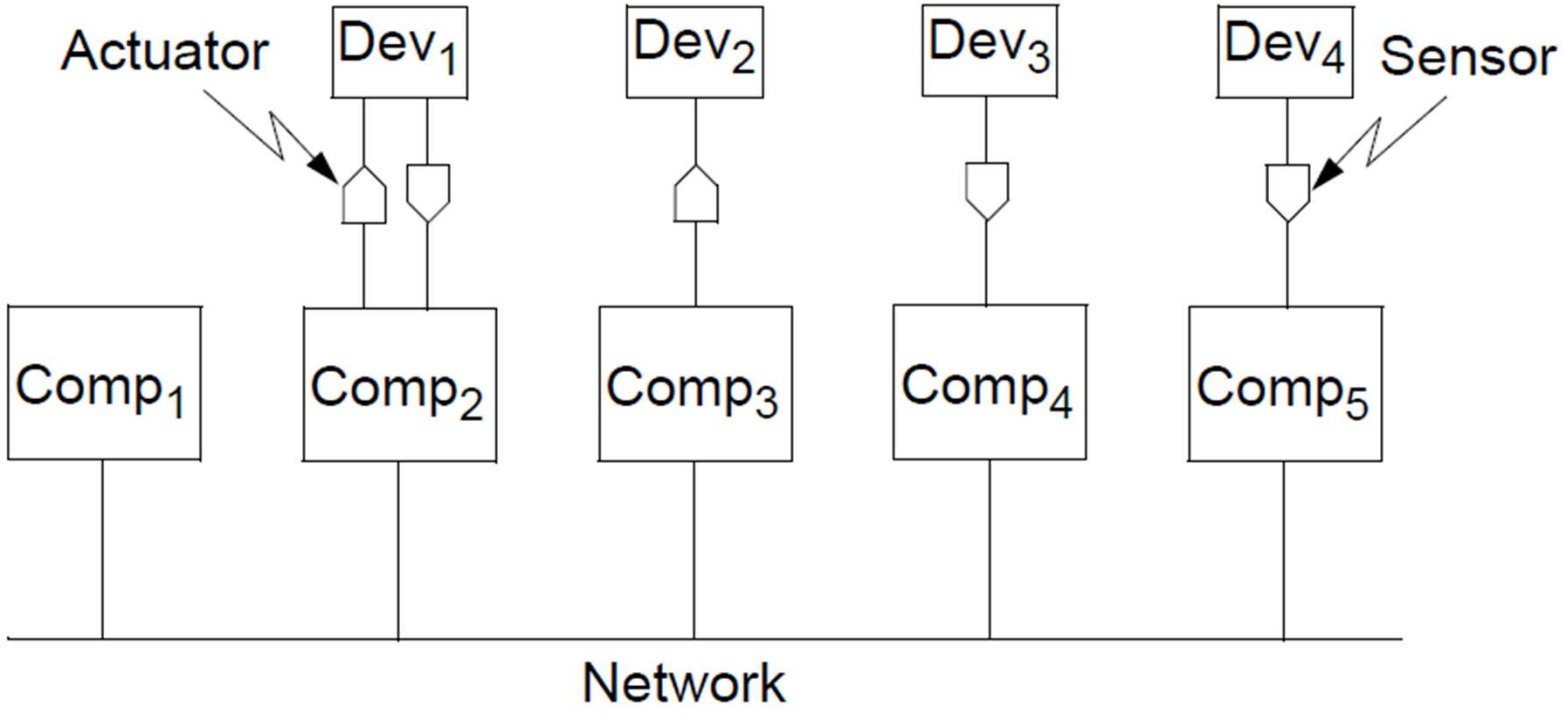
- Real-time systems usually are in strong interaction with their physical environment.
- They receive data, process it, and return results **in right time**.

- Examples:
  - Process control systems
  - Computer-integrated manufacturing systems
  - Aerospace and avionics systems
  - Automotive electronics
  - Medical equipment
  - Nuclear power plant control
  - Defense systems
  - Consumer electronics
  - Multimedia
  - Telecommunications

Distributed Real-Time Systems

Real-time systems often are implemented as **distributed** systems.

- Some reasons:
  - Fault tolerance
  - Certain collection/processing of data has to be performed at the location of the sensors and actuators
  - Performance issues



Real-Time Systems

Some Typical Features:

- They are **time-critical**.
  - The failure to meet time constraints can lead to degradation of the service or to a catastrophe.
- They are made up of **concurrent processes** (tasks).
- Processes share resources (e.g. processor) and communicate to each other.
  - This makes **scheduling** of processes a central problem.
- **Reliability** and **fault tolerance** are essential.
  - Many real-time applications are **safety critical**.
- Such systems are very often embedded in a larger system, like a car, CD-player, phone, camera, etc.

Soft and Hard Deadlines

Time constraints are often expressed as **deadlines** at which processes have to complete their execution.

A deadline imposed on a process can be:

- **Hard deadline**: has to be met strictly; if not → "catastrophe"
  - should be guaranteed a-priori, off-line.
- **Soft deadline**: processes can be finished after their deadline, although the value provided by the completion may *degrade* with time.
- **Firm deadline**: similar to hard deadlines, but if the deadline is missed, there is no catastrophe, only the result produced is of *no* use any more.

Predictability

Predictability is one of the most important properties of any real-time system.

- **Predictability** means that it is possible to **guarantee** that deadlines are met as imposed by requirements:
  - hard deadlines are always fulfilled.
  - soft deadlines are fulfilled to a degree which is sufficient for the imposed quality of service.

Some **problems** to solve towards predictability:

- Determine **worst case execution times** for each process.
- Determine **worst case communication delays** on the network.
- Determine **bound on clock drift and skew** (see later).
- Determine **time overheads due to operating system** (interrupt handling, process management, context switch, etc.).

After all the problems above have been solved, comes the "big question":

- **Schedulability**:
  Can the given processes and their related communications be **scheduled** on the available **resources** (processors, buses),
  so that deadlines are fulfilled?

Scheduling

The scheduling problem:

Which process and communication has to be executed at a certain moment on a given processor or bus respectively, so that time constraints are fulfilled?

- A set of processes is **schedulable** if, given a certain **scheduling policy**, all constraints will be completed (if a solution to the scheduling problem can be found).

Scheduling Policies

When are the decisions taken?

- **static scheduling**
- **dynamic scheduling**

Are processes **preempted**?

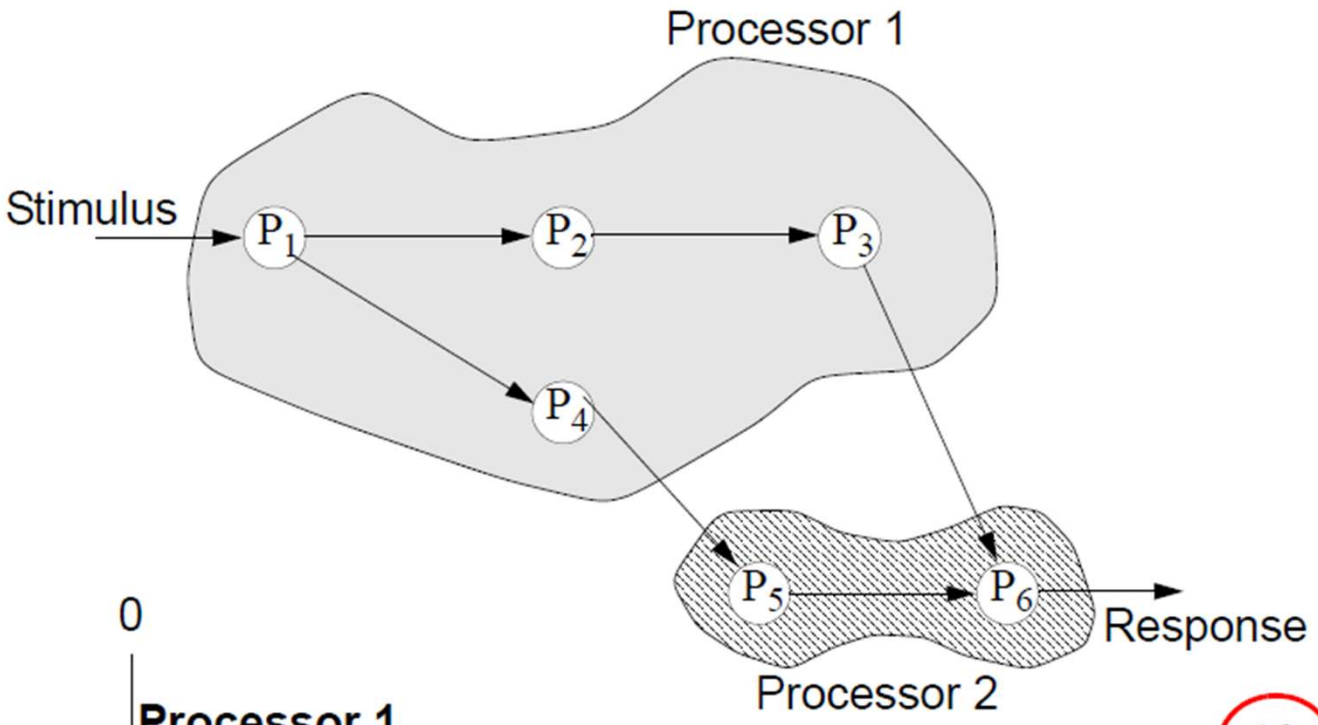
- **preemptive scheduling**
- **non-preemptive scheduling**

Static Scheduling

Static scheduling: decisions are taken **off-line**.

- A **table** containing activation times of processes and communications is generated off-line.
  - This table is used at run-time by a very simple kernel (**table-driven execution**).

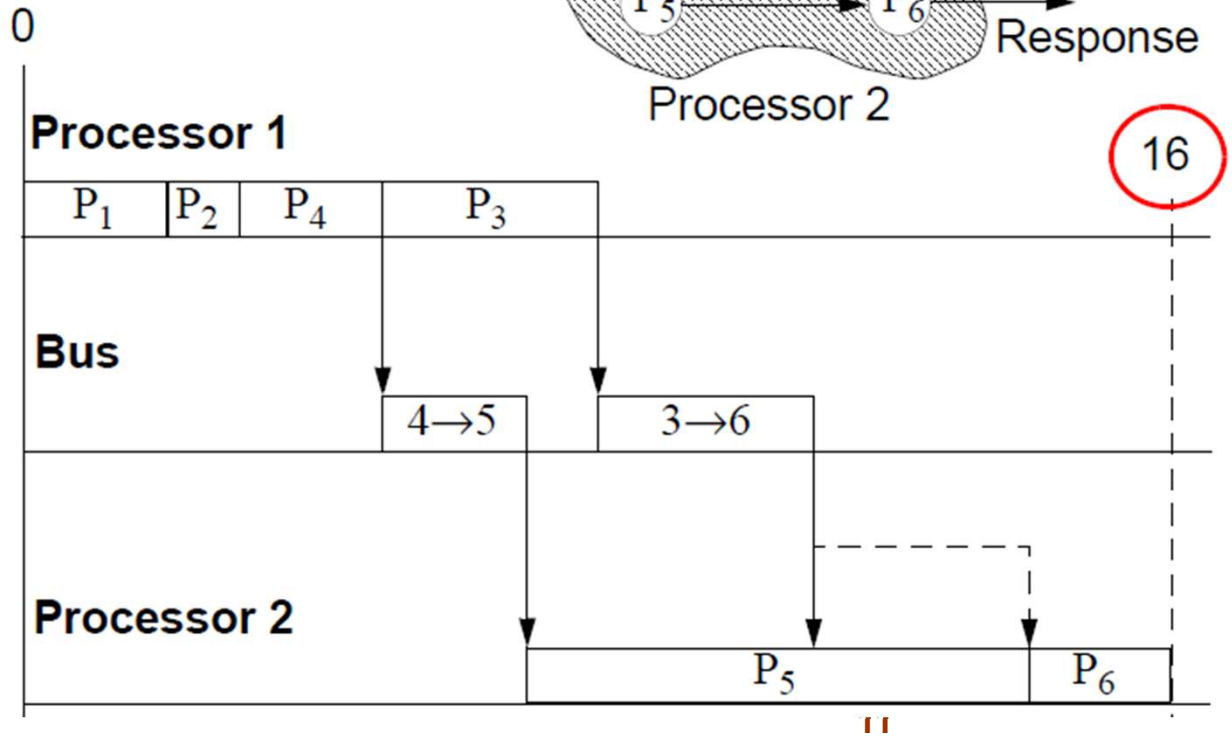
Static Scheduling



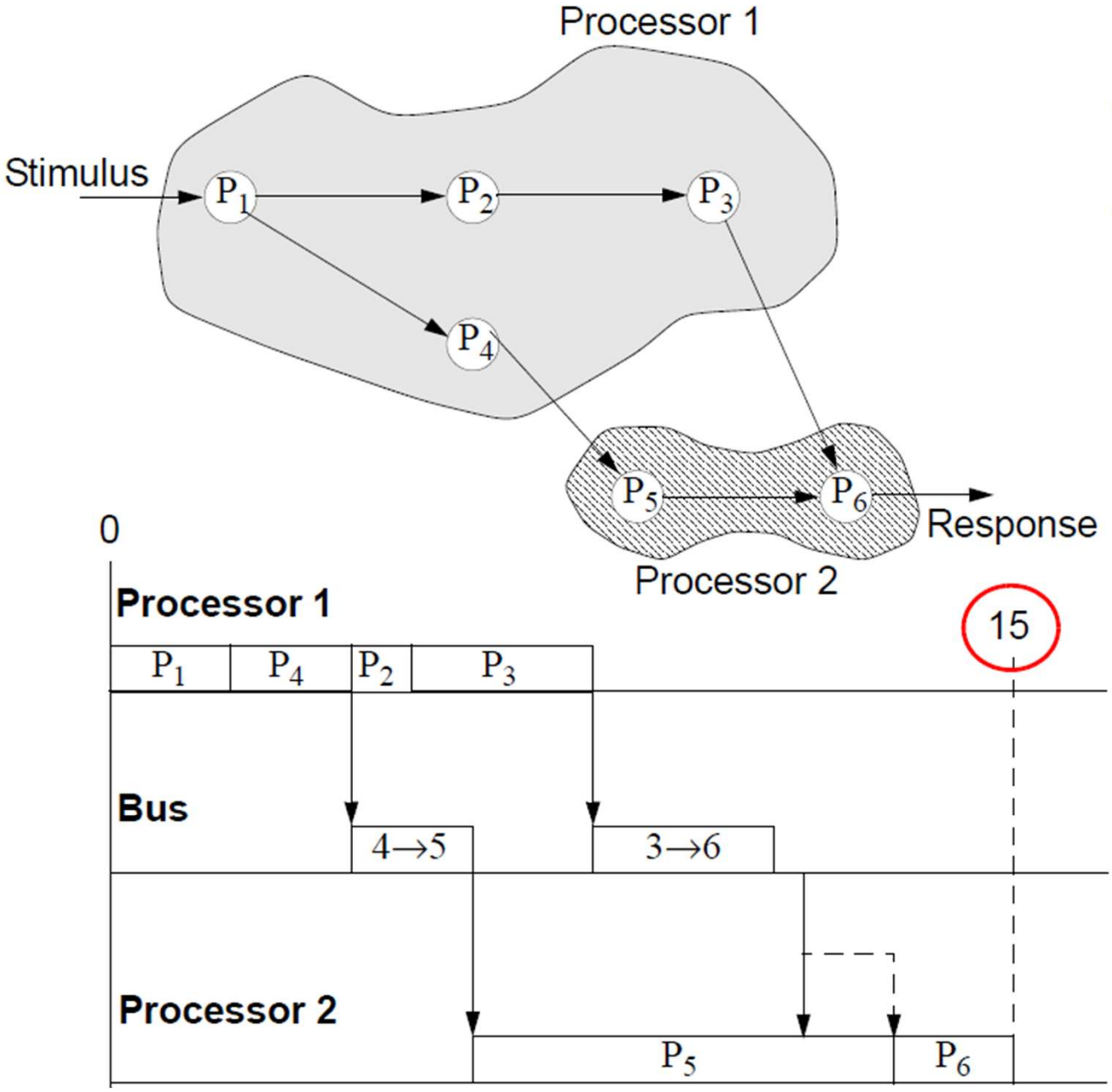
Deadline on *response*: 15

Worst case execution times:

P_1	2
P_2	1
P_3	3
P_4	2
P_5	7
P_6	2
$4 \rightarrow 5$	2
$3 \rightarrow 6$	3



Static Scheduling

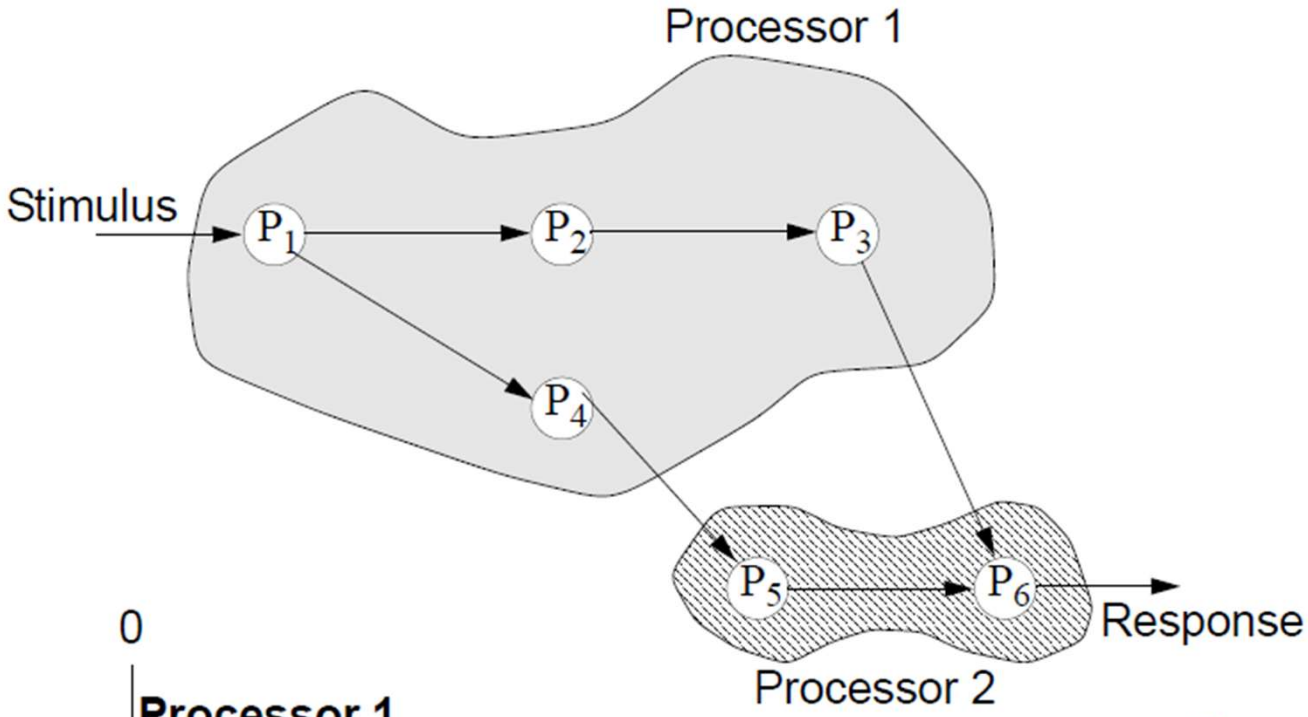


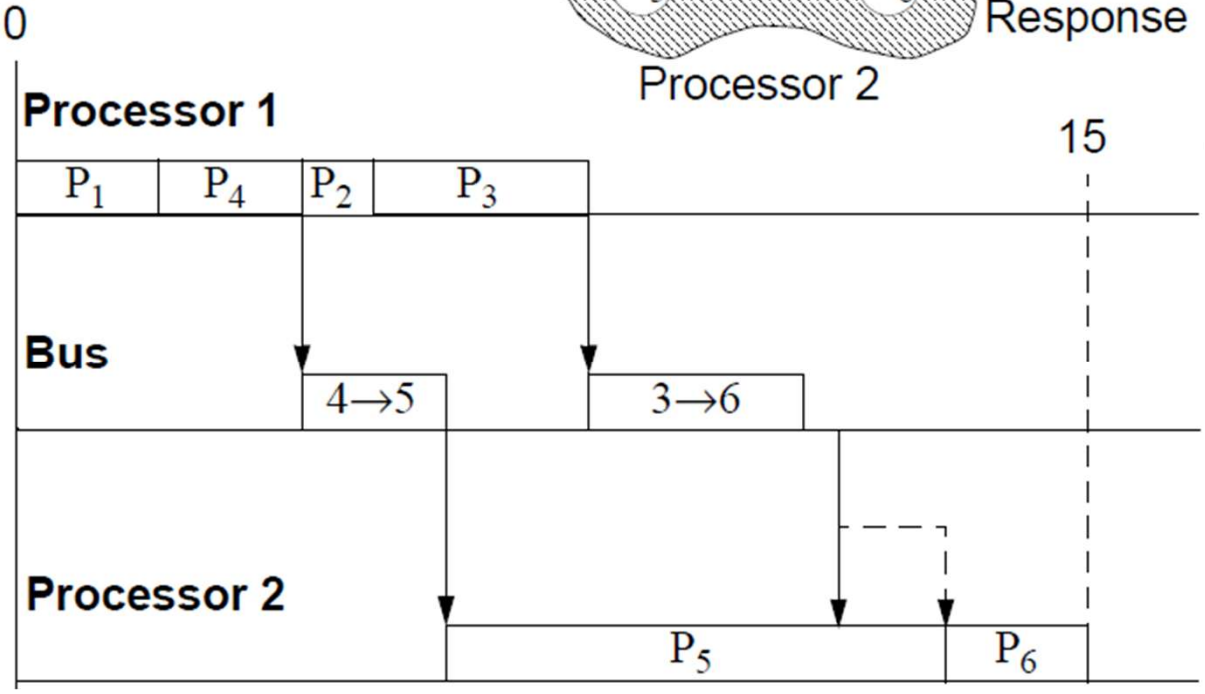
Deadline on *response*: 15

Worst case execution times:

P_1	2
P_2	1
P_3	3
P_4	2
P_5	7
P_6	2
4→5	2
3→6	3

Static Scheduling





SCHEDULE TABLE:

Processor 1

0	P_1
2	P_4
4	P_2
4	write message 4→5
5	P_3
8	write message 3→6

Processor 2

6	read message 4→5
6	P_5
11	read message 3→6
13	P_6
15	write output

Static Scheduling

What is **good**?

- High predictability.
  - Deadlines can be guaranteed - if the scheduling algorithm succeeded in building the schedule table.
- Easier to debug.
- Low execution time overhead.

What is **bad**?

- Assumes prior knowledge of processes (communications) and their characteristics.
- Not flexible
  - it works well only as long as processes/communications strictly behave as predicted

Dynamic Scheduling

- No schedule (predetermined activation times) is generated off-line.

- *Will the processes meet their deadlines?*
  - This question can be answered only in very particular situations of dynamic scheduling!
  - **Schedulability analysis** tries to answer it.

Dynamic Scheduling

- Processes are activated in response to **events** (arrival of a signal, message, etc.)

- Processes have associated **priorities**
  - If several processes are ready to be activated on a processor, the highest priority process will be executed.

- **Priority based preemptive scheduling**:
  - At any given time, the highest priority ready process is running.
  - If a process becomes ready to be executed (the respective event has occurred), and it has a higher priority than the running process, the running process will be **preempted** and the new one will execute.

Dynamic Scheduling

With **restrictions** in the process model,
schedulability analysis is possible:

For example:

- One single processor.
- All the *n* processes are **periodic** and have a **fixed** (worst case) computation **time** c_i, and **period** T_i.
- All processes have a deadline equal to their period.
- Priorities are assigned to processes according to their period → the process with **shorter period** gets the **higher priority**.

Under the circumstances above,
known as **rate-monotonic scheduling**,
all processes will **meet** their deadline if $\sum_{i=1}^{n} \frac{c_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$

→ See IDA course on real-time systems

Specific Issues in *Distributed* Real-Time Systems

1. Clock synchronization

2. Real-Time Communication

Clock Synchronization

The need for synchronized distributed clocks:

- **Time-driven systems**:
  In statically scheduled systems, activities are started at "precise" times in different points of the distributed system.

- **Time stamps**:
  Certain events or messages are associated with a time stamp showing the actual time when they have been produced.
  - Certain decisions in the system are based on the "exact" time of the event.

- **Calculating the duration of activities**:
  If such an activity starts on one processor/computer and finishes on another (e.g. transmitting a message), calculating the duration needs clocks to be synchronized.

Computer Clocks

- A **quartz crystal** oscillates at well defined frequency and oscillations are counted (by hardware) in a **register**.
- After a number of oscillations, an interrupt is generated; this is the **clock tick**.
- At each clock tick, the **computer clock** is incremented by (system) software.

Computer Clocks

The problems:

1. **Crystals cannot be tuned perfectly**.
   - *Temperature* and other external factors can also influence their frequency.

   → **Clock drift**: the computer clock differs from the real time.

2. **Two crystals are never identical**.

   → **Clock skew**: the computer clocks on different computers of the distributed system show different time.

"Universal" Time

- The standard for measurement of time:
  International Atomic Time (**TAI**).
  - It defines the *standard second*
    and is based on atomic oscillators.

- **Coordinated Universal Time** (**UTC**):
  is based on TAI, but is kept in step with astronomical time
  (by occasionally inserting or deleting a "leap second").
  - **UTC signals** are broadcast from satellites and land-based radio stations.

External and Internal Synchronization

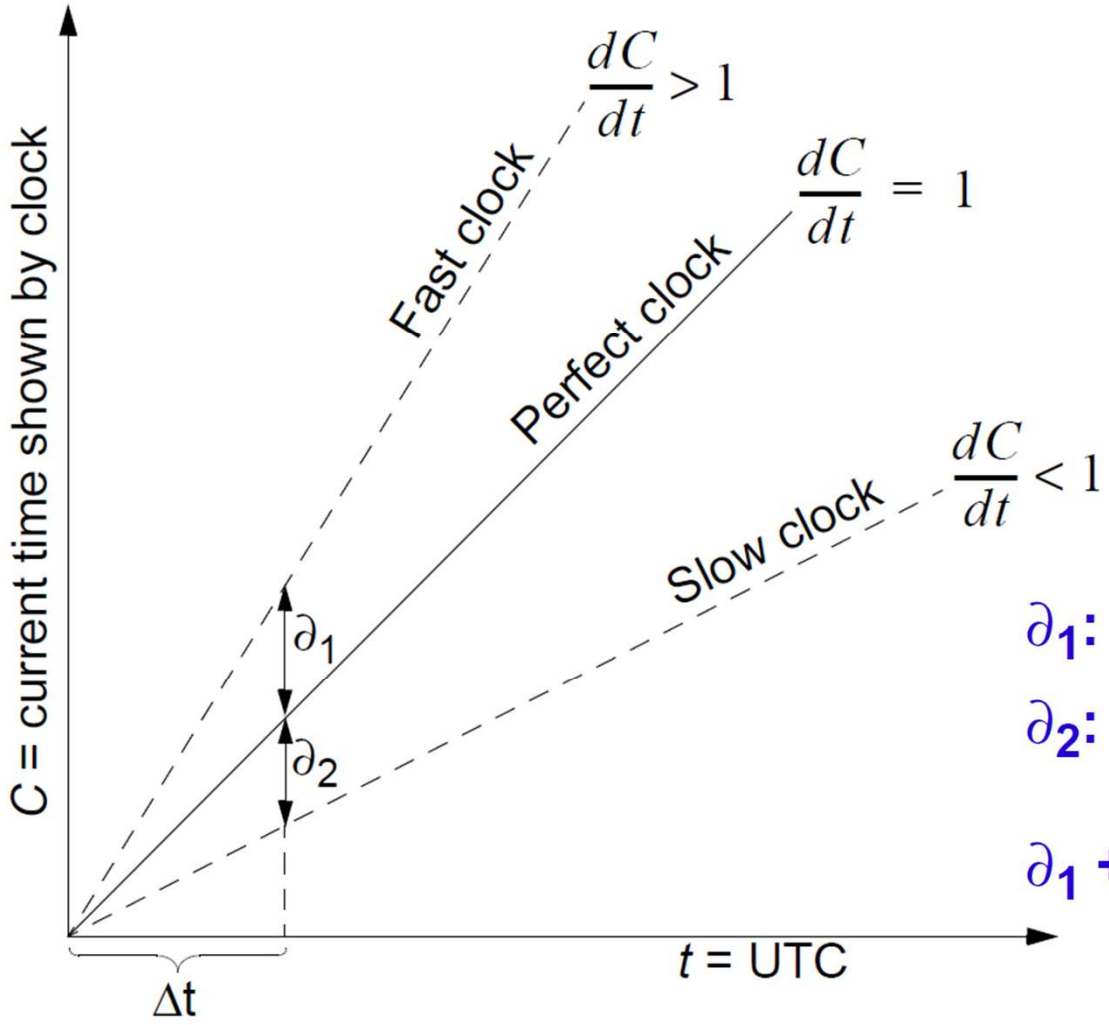
External Synchronization

- Synchronization with a time source external to the distributed systems, such as UTC broadcasting system.
  - One computer in the system (possibly several) is equipped with UTC receiver (time providers).
  - By external synchronization, the system is kept synchronous with the "real time".
    This allows to exchange consistently timing information with other systems and with users.

Internal Synchronization

- Synchronization among computers of the distributed system.
  - Needed to keep a consistent view of time over the system.
  - A few computers synchronize externally, and the whole system is kept consistent by internal synchronization.
  - Sometimes only internal synchronization is performed (if we do not care for the drift from external/real time).

Drifting of Clocks

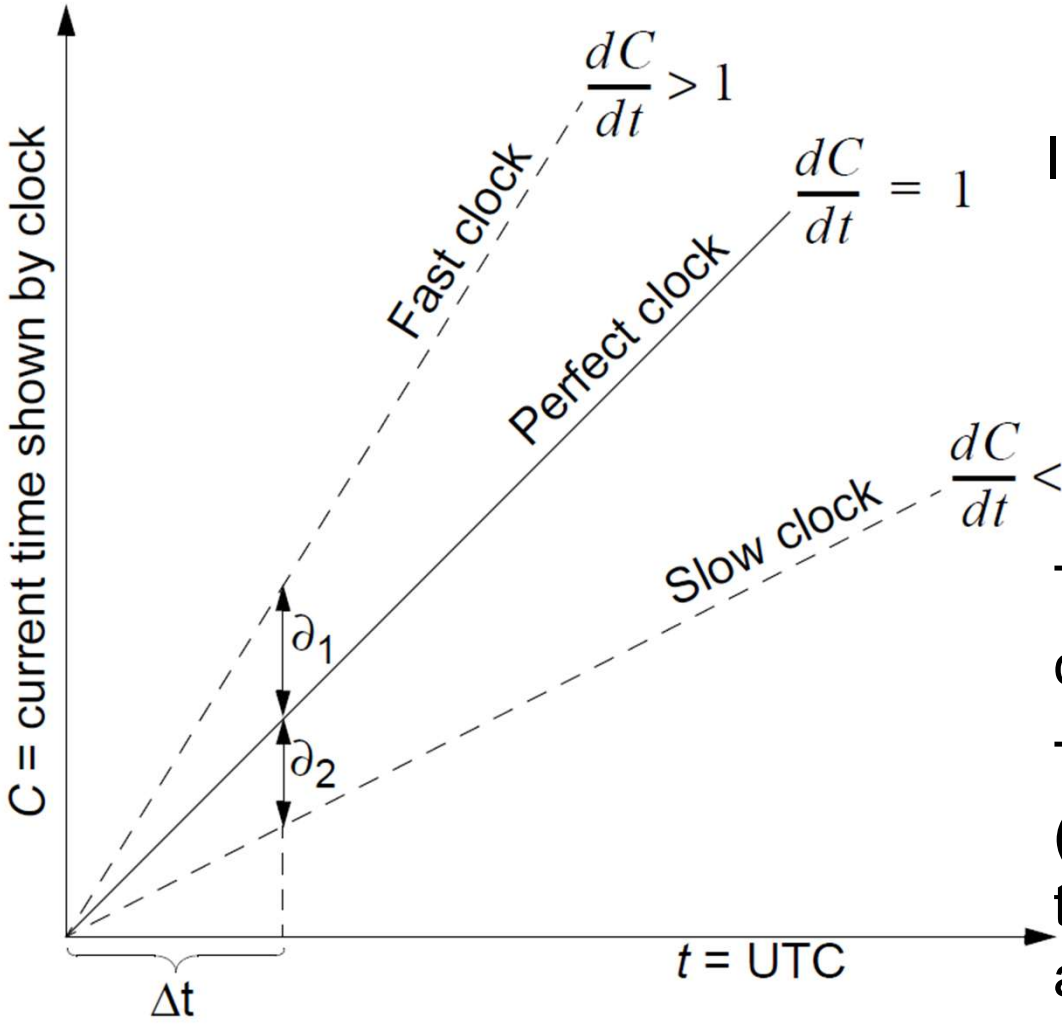


∂_1: drift of first clock after Δt.

∂_2: drift of second clock after Δt.

$\partial_1 + \partial_2$: skew between clocks after Δt.

Drifting of Clocks



In **ideal case**, the clock shows UTC:

$$C = t$$

$$\rightarrow \quad \frac{dC}{dt} = 1$$

In **reality**,

$$\frac{dC}{dt} \geq 1 - \rho$$

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

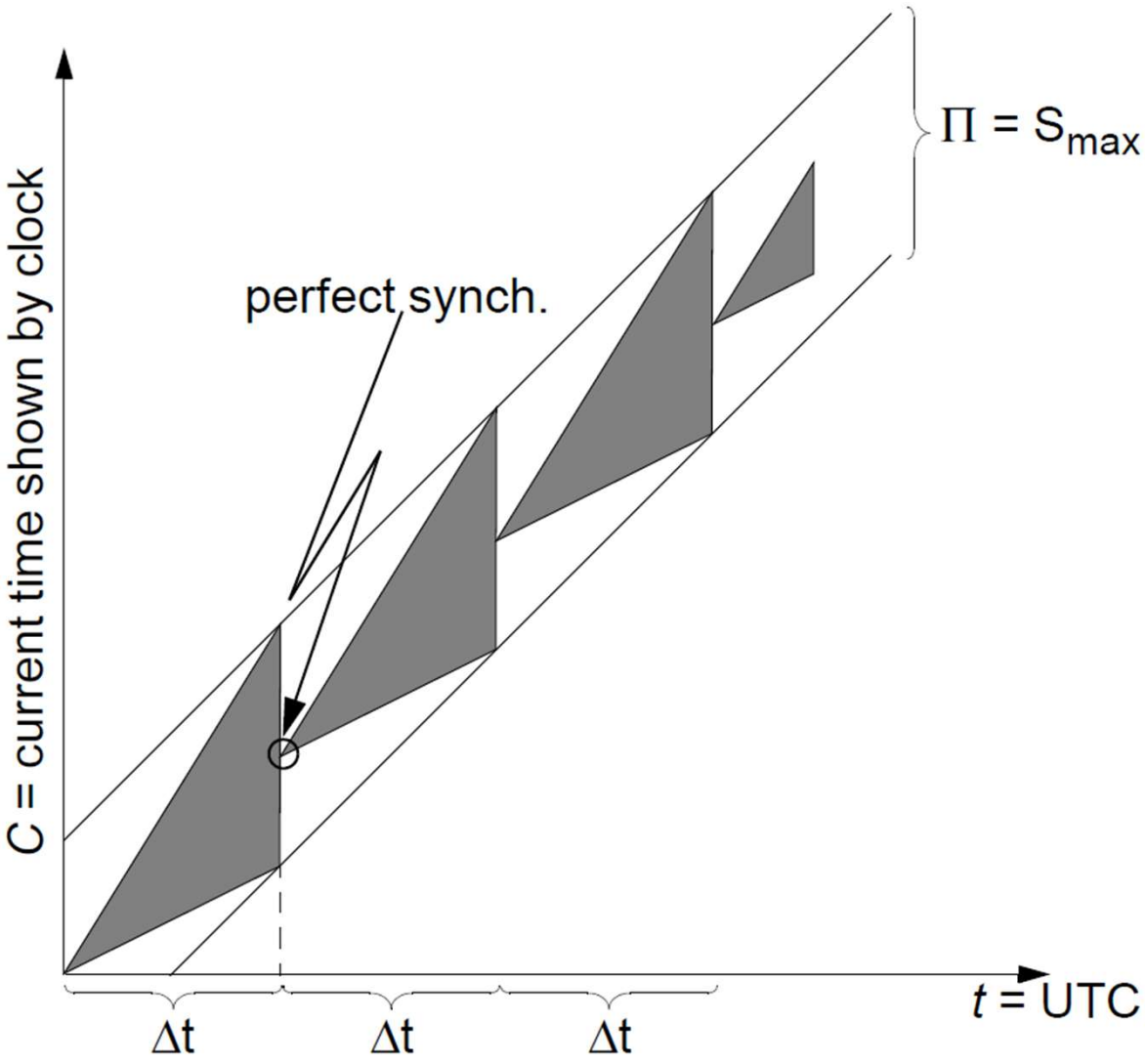
ρ is the **maximum drift rate**; specified by the manufacturer.

Two processors with similar clocks could be apart by $S \leq 2\rho\,\Delta t$

To guarantee a precision $S_{max} = \Pi$ (max. skew between the two clocks), the clocks have to be **synchronized** at an interval:

$$\Delta t = S_{max} / 2\rho = \Pi / 2\rho$$

Drifting of Clocks



Only if we assume that, after synchronization, the clocks are **perfectly** aligned, then:

$S_{max} = 2\rho\,\Delta t$ and thus,

$\Delta t = S_{max} / 2\rho$

In **reality**, the alignment after synchronisation is **not perfect.**

- The **convergence function** Φ denotes the offset of the time values immediately after resynchronisation.

In order to achieve a certain precision S_{max}:

$$S_{max} = \Phi + 2\rho\,\Delta t$$

$$\Delta t = (S_{max} - \Phi) / 2\rho$$

Clock Synchronization Algorithms

Centralized Algorithms

- There exists one particular node, the so-called **time server node**.
  - **Passive time server**: the other machines ask periodically for the time. Goal is to keep clocks of all nodes synchronized with the time server. Often the case when the time server has an UTC receiver.
  - **Active time server**: the time server is active, polling the other machines periodically. Based on time values received, it computes an update value of the clock, which is sent back to the machines.

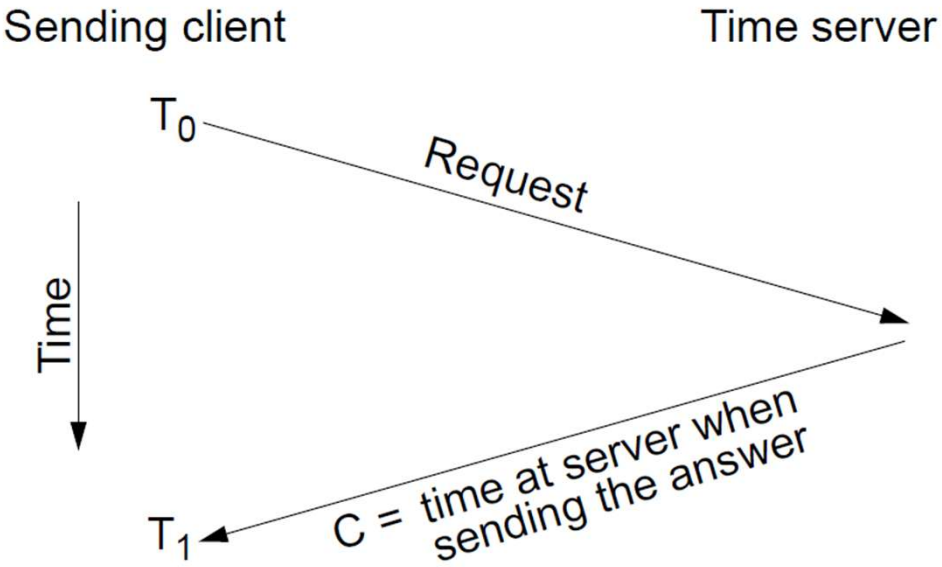
Distributed Algorithms

- There is no particular time server.
  The processors **periodically** reach an **agreement** on the clock value.
  - This can be used if no UTC receiver exists (no external synchronization). Only **internal synchronization** is performed.
  - Even if several computers (possibly all) have an UTC receiver, this does not avoid clock skews.
    Internal synchronization is performed using a **distributed clock synchronization** strategy.

Cristian's Algorithm

Cristian's algorithm is a centralized algorithm with passive time server. The server is supposed to deliver the correct time (has a UTC receiver).

- Periodically (with period less than $(S_{max} - \Phi) / 2\rho$), each client sends a message to the time server asking for the current time:



T_0 and T_1 are the time shown by the clock of the client when sending the request and receiving the answer, respectively.

Simplest: set receiving client clock

$T_{rec} = C$

However, it takes a certain time, T_{trans}, for the reply to get back to the client:

$T_{rec} = C + T_{trans}$

How large is T_{trans}? Estimation:

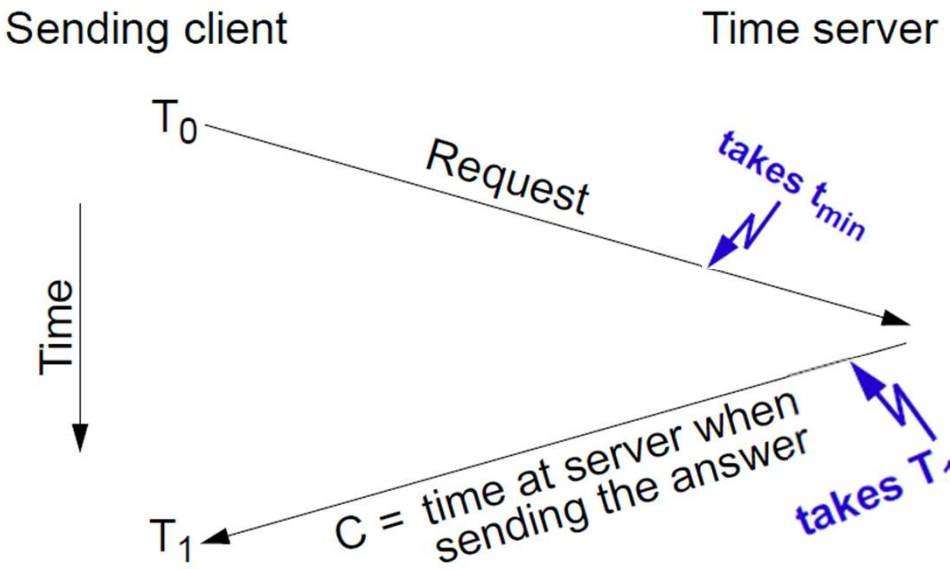
$T_{rec} = C + (T_1 - T_0) / 2$?

Time to receive the answer can be different from that to transmit request! Can we, at least, determine the *accuracy* of the estimation?

Cristian's Algorithm

Cristian's algorithm is a centralized algorithm with passive time server. The server is supposed to deliver the correct time (has a UTC receiver).

- Periodically (with period less than $(S_{max} - \Phi) / 2\rho$), each client sends a message to the time server asking for the current time:



T_0 and T_1 are the time shown by the clock of the client when sending the request and receiving the answer, respectively.

Suppose the *minimum* time t_{min} for a communication between the machine and the time server is known:

$$T_{rec}^{min} = C + t_{min}$$

$$T_{rec}^{max} = C + (T1 - T0) - t_{min}$$

The range:

$$T_{rec}^{max} - T_{rec}^{min} = (T1 - T0) - 2t_{min}$$

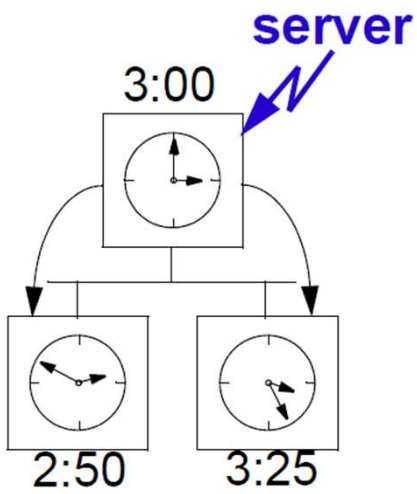
Time set with **absolute accuracy** $\pm ((T_1 - T_0)/2 - t_{min})$

Improve accuracy: issue *several* requests; the answer with the smallest $(T_1 - T_0)$ is used to update the clock.

The Berkeley Algorithm

The Berkeley algorithm is a **centralized** algorithm with **active time server**.

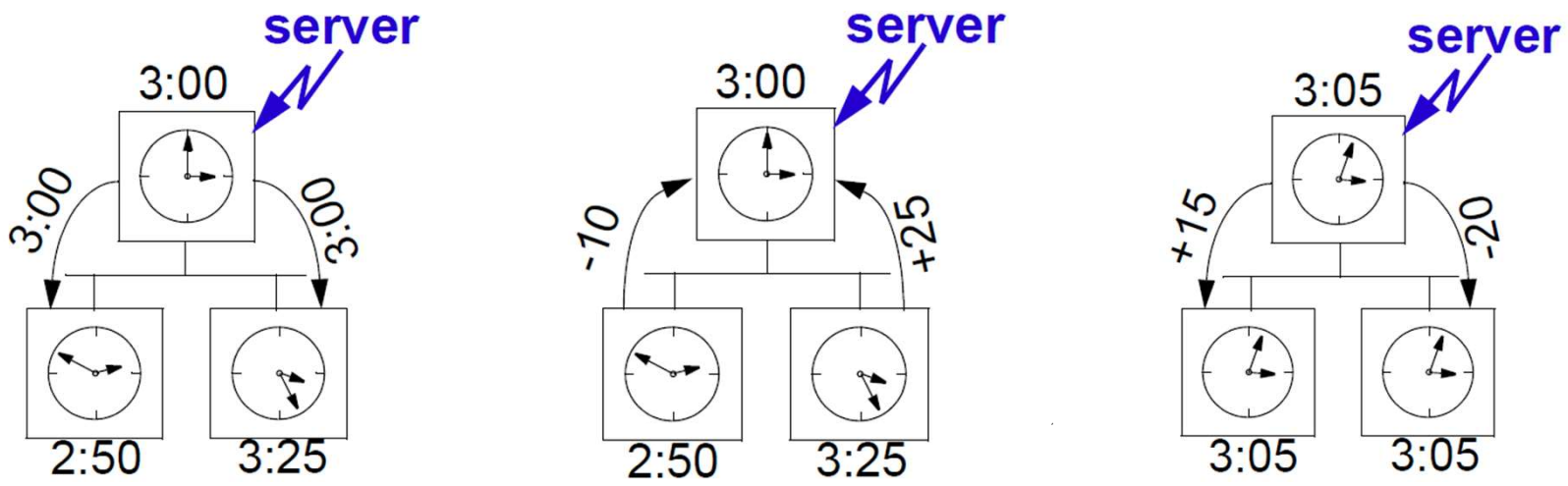
- It tries also to address the problem of possible faulty clocks.



The Berkeley Algorithm

The Berkeley algorithm is a **centralized** algorithm with **active time server**.

- It tries also to address the problem of possible faulty clocks.



The **server** polls periodically every machine and gets their time offset.

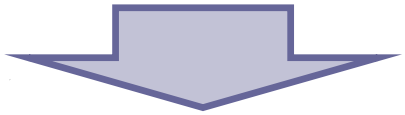
Based on received values, the server computes an average.

The server, finally, tells each machine by which amount to advance or slow down its clock.

The Berkeley Algorithm

The situation is more complicated:

- The server performs corrections, taking into consideration estimated propagation times for messages, before computing averages.
- If on a certain processor the clock has to be set back, this has to be performed in a special way, in order to avoid problems.
- The server tries to avoid taking into consideration values from clocks which are drifted badly or that have failed.



Only clock values are considered that do not differ from one another by more than a certain amount.

Distributed Clock Synchronization Algorithms

With **distributed** clock synchronization there is no particular time server

Distributed clock synchronization proceeds in **three phases**, which are repeated periodically:

1. Each node sends out information concerning its own time, and receives information from the other nodes concerning their local time.
2. Every node analyzes the collected information received from the other nodes and calculates a correction value for its **own** clock.
3. The local clocks of the nodes are updated according to the values computed at step 2.

The typical algorithm used at point 2 performs the following:

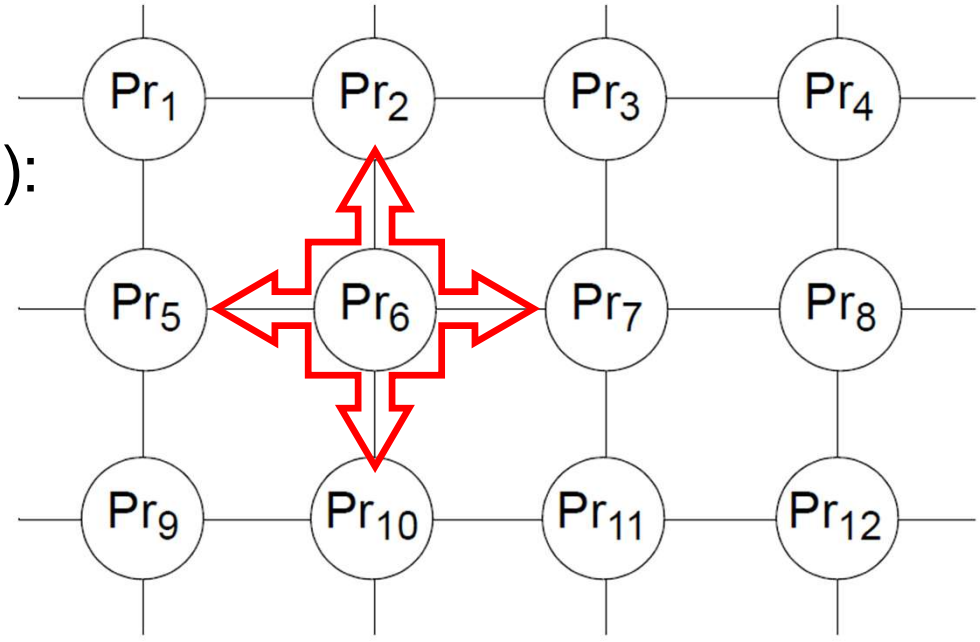
- The **correction value** for the local clock is based on an **average** of the received clock values.
- Corrections are performed taking into consideration estimated delays due to message passing (as above for Cristian's algorithm).
- Only clock values are considered that do not differ from one another by more than a certain amount.

Distributed Clock Synchronization

Localized Averaging Algorithm

- With a large network it is impractical to perform synchronization among all nodes of the system. Broadcasting synchronization messages from each node to all other nodes generates huge traffic.
- In large networks, nodes are **logically grouped** into structures, like *k*-D grid or ring (→ overlay network).
  Each node synchronizes with its **neighbours** in the structure.

  - Example: a **2D grid** (2D mesh):
    - Pr_6 synchronizes with Pr_2, Pr_7, Pr_{10}, and Pr_5
    - Pr_7 synchronizes with Pr_3, Pr_8, Pr_{11}, and Pr_6



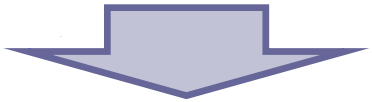
Adjusting Drifted Clocks

The problem:

- Suppose that the current time on the processor is T_{curr} and, as result of clock synchronization, it has to be updated to T_{new}
  - If $T_{new} > T_{curr}$: advance the local clock to the new value T_{new}
  - If $T_{new} < T_{curr}$, **we are not allowed to just set back the local clock** to T_{new}

Why?

- Setting back the clock can produce severe errors, like faulty time stamps to files (copies with identical time stamp, or later copies with smaller time stamp) and events.



It is not allowed to turn time back!

Instead of turning the clock back, it is "**slowed down**" until it, progressively, reaches a desired value.

Adjusting Drifted Clocks

At each **clock tick**, an increment of the **internal clock value** θ is performed:

$\theta = \theta + \nu$ (ν is the **step** by which the internal clock is incremented).

- In order to be able to perform time adjustment, the software time (T_{curr}), which is visible to the programs, is not directly θ, but a **software clock** which is updated at each tick with a certain correction relative to θ:

  $T_{curr} := \theta*(1 + a) + b$

  - if no adjustment is needed, then $a = b = 0$.
  - the parameters a and b are set when a certain adjustment is needed, and used for the period the adjustment is going on.

Adjusting Drifted Clocks

Suppose at a certain moment:

- The internal clock shows θ
- The software clock shows T_{curr}
- The clock has to be adjusted to T_{new}
- The adjustment performed "smoothly" over a period of N clock ticks.

We have to fix a and b that are to be used during the adjustment period:

- For the starting point we have to have:

$$T_{curr} = \theta(1 + a) + b \qquad (1)$$

- After N ticks,
  - the "real" time will be: $T_{new} + N\nu$
  - the software clock will show: $(\theta + N\nu)(1 + a) + b$
- If after N ticks the time adjustment is to be finished:

$$(\theta + N\nu)(1 + a) + b = T_{new} + N\nu \qquad (2)$$

From (1) and (2) we get:

- $a = (T_{new} - T_{curr}) / N\nu$
- $b = T_{curr} - (1 + a)\theta$

Adjusting Drifted Clocks

- The strategy works regardless if the adjustment has to be performed forward ($T_{new} > T_{curr}$) or backward ($T_{new} < T_{curr}$).
- If the adjustment is forward, it can be performed by just updating the clock.
- If the adjustment is backward, the clock has to be changed smoothly.

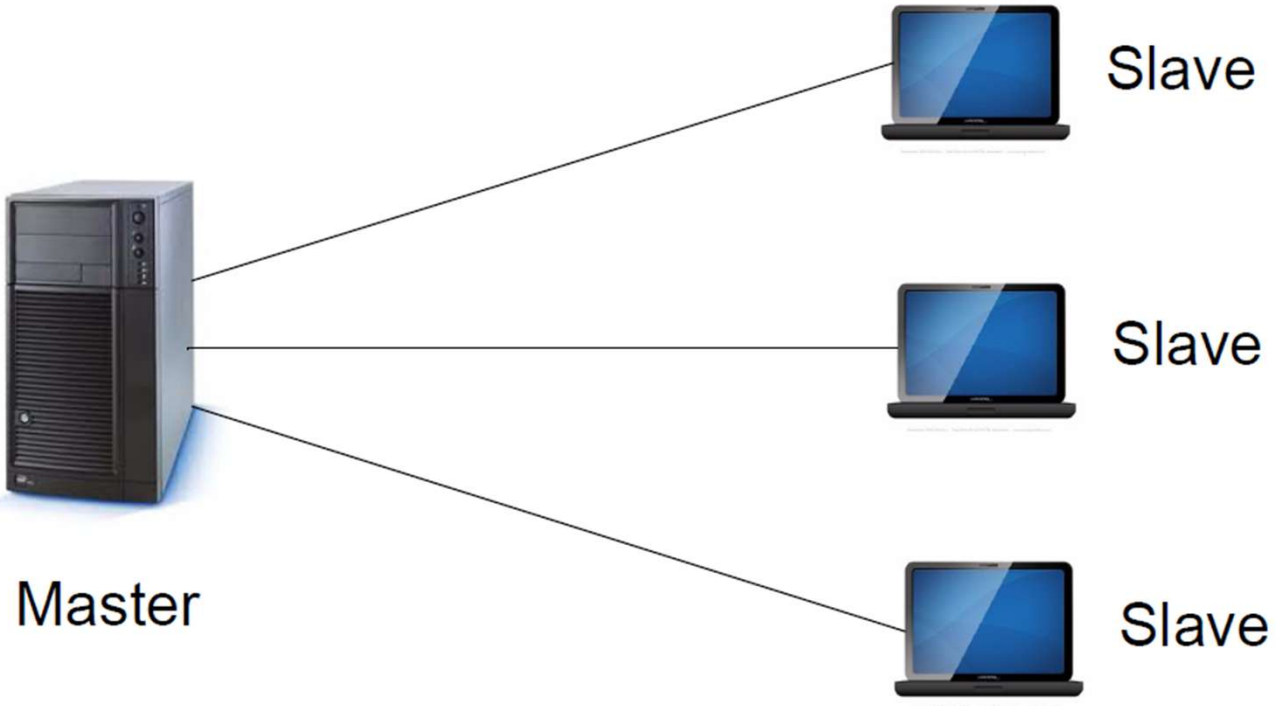
The Precision Time Protocol (PTP)

- The **Precision Time Protocol – PTP** (IEEE-1588 standard) provides a method to precisely synchronise distributed computer clocks over a Local Area Network with an accuracy of less than 1 microsecond.

- PTP is primarily intended for use in special-purpose networks for industrial automation, measurement systems, robotics, automotive technology, etc.

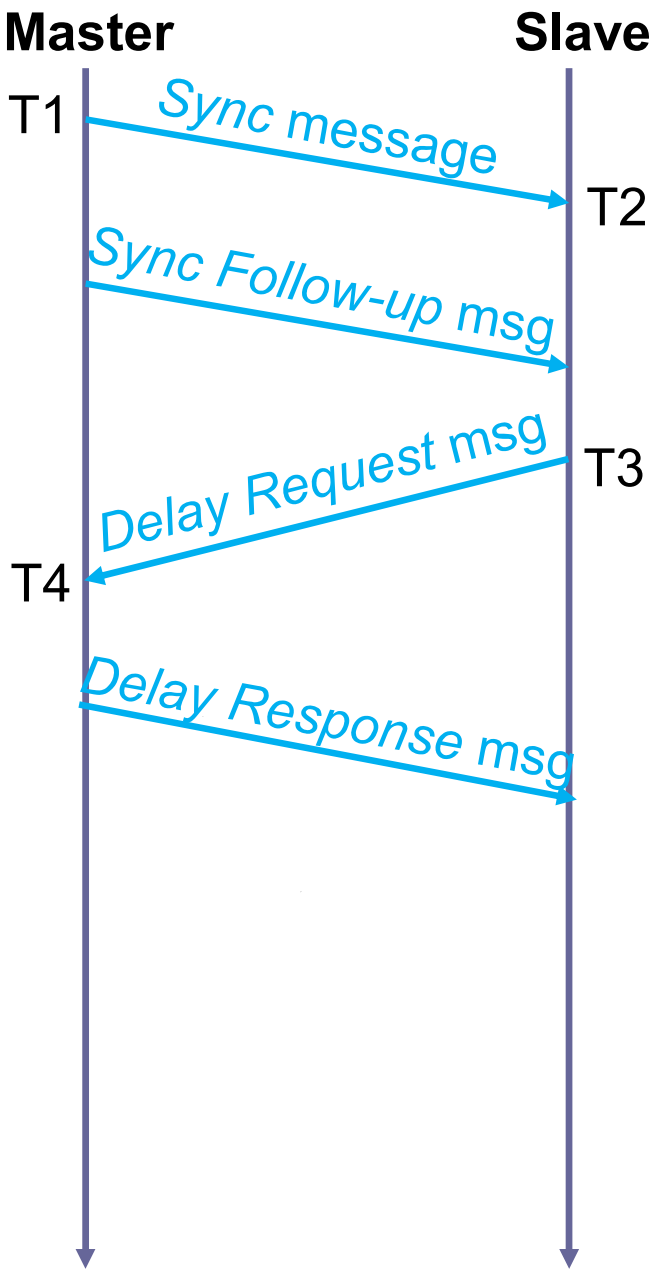
- The synchronisation approach in PTP is based on a **centralised** technique: a **master node** synchronises one or several **slave nodes** connected to it (remember Cristian's algorithm).

The Precision Time Protocol



- The Master node is provider of time;
  the Slave node synchronises to the Master.
- The time of the Master is reported to the Slave as accurately as possible.
- The goal of the employed algorithm is to compensate for the processing times on the nodes and for communication delays.

The Precision Time Protocol

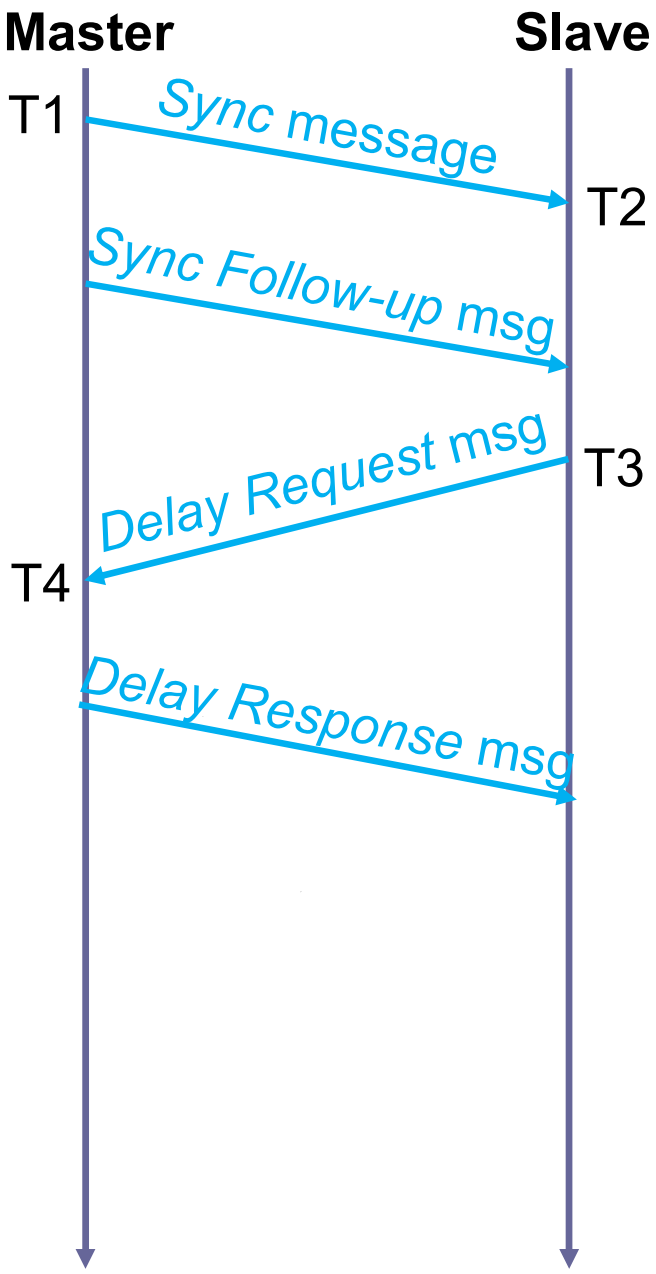


- ***Sync***: issued at T1, arrives at T2
- ***Sync Follow-up***: carries the value of T1;
- ***Delay Request***: issued at T3 arrives at T4;
- ***Delay Response***: carries the value of T4.

Question: Why does not the *Sync* message carry the value of T1?

Answer: The value of T1 is the exact moment when the *Sync* message has left the master. This is later than the moment when the message is assembled and issued for transmission; thus, the value of T1 cannot be written into the message.
By this policy, the interval T1-T2 does not contain any delay due to running the protocol stack or due to the queuing time of the message in the case of congestion.

The Precision Time Protocol

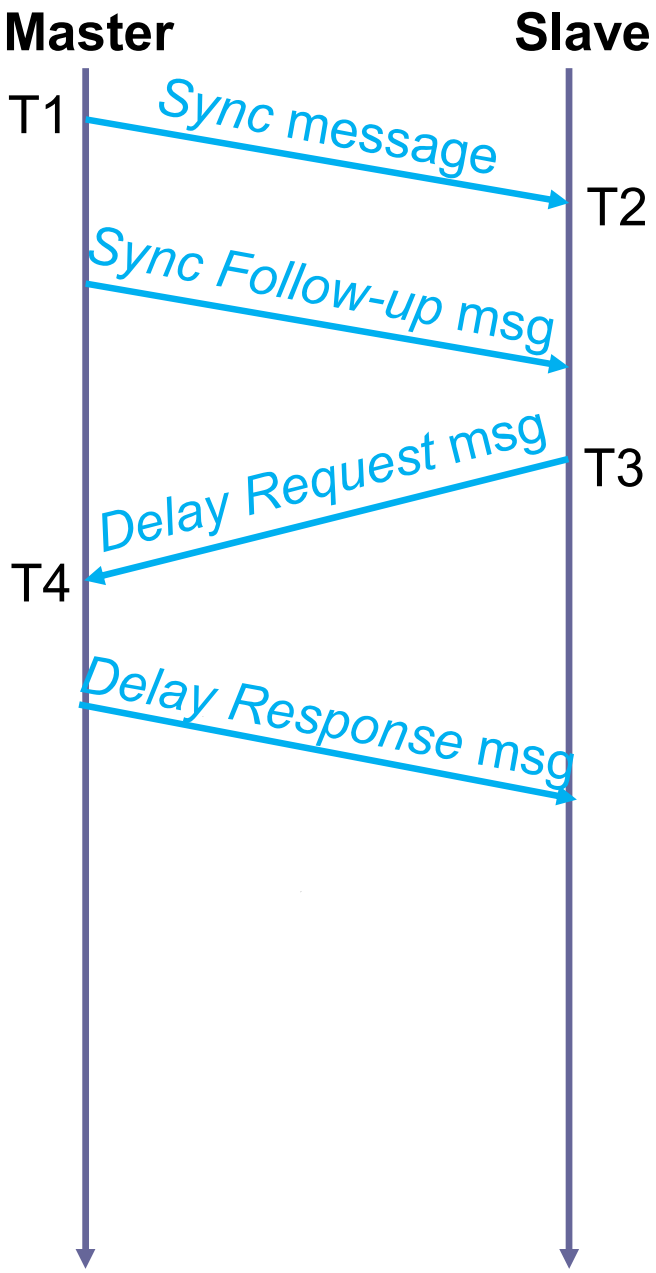


Using timestamps T1, T2, T3, T4 the slave is able to accurately synchronize its clock.

Performed in two phases:

- **Phase 1**: offset calculation
  - Messages *Sync* and *Sync Follow-up*;
- **Phase 2**: delay measurement
  - Messages *Delay Request* and *Delay Response*;

The Precision Time Protocol



A naive approach:

Calculate the master→slave offset Θ_{MS}:

$$\Theta_{MS} = T2 - T1$$

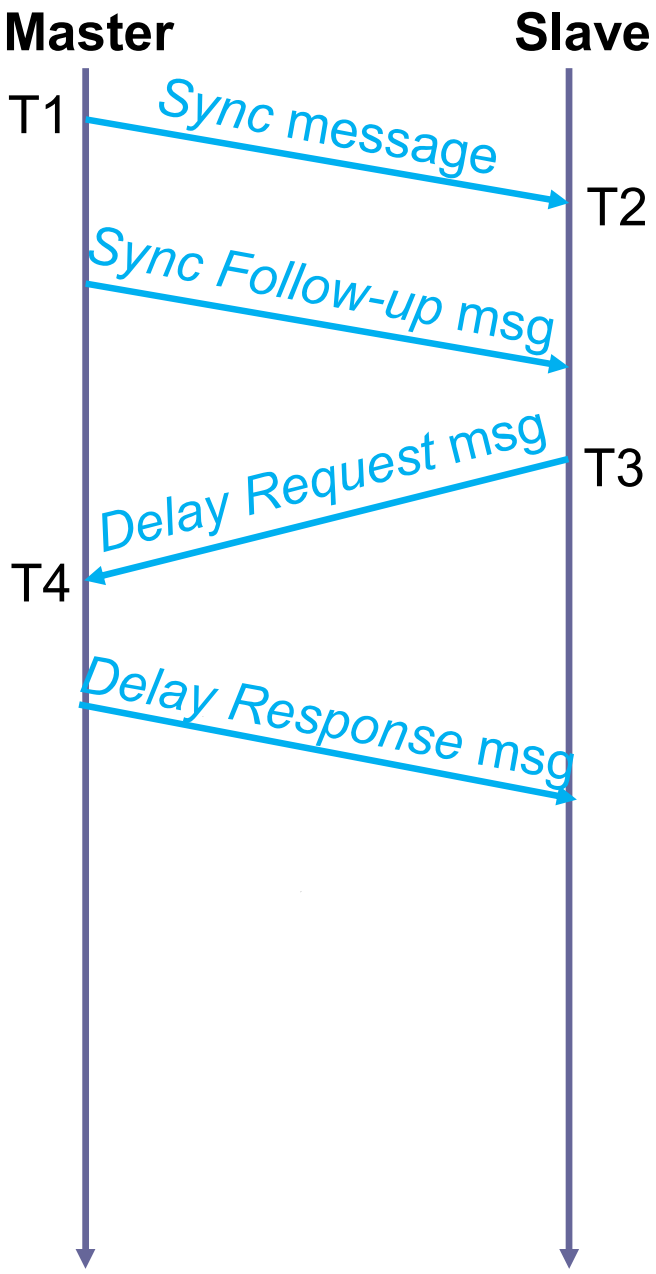
Update the slave clock T_S:

$$T_S = T_S - \Theta_{MS} = T_S - (T2 - T1)$$

The offset Θ_{MS}, calculated above, includes the communication delay of the *Sync* message

→ The above synchronization is accurate only if this delay is zero (which, of course, it is not)!

The Precision Time Protocol



An accurate approach:

Estimate the communication delay:

$$T2 - T1 = \Theta_{MS} + \delta$$

$$T4 - T3 = \Theta_{MS} + \delta$$

$$\delta = (T2 - T1 + T4 - T3) / 2$$

Update the slave clock:

$$\Theta_{MS} = T2 - T1 - \delta$$

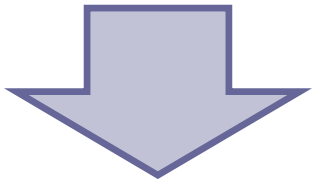
$$T_S = T_S - \Theta_{MS} = T_S - (T2 - T1) + \delta$$

- Phase 1 is performed typically every 2 seconds.
- Phase 2 is performed at greater intervals (between 4 and 60 seconds).
- Between two runs of Phase 2, the last update of delay δ is used for the server clock update after each run of Phase 1.

The Precision Time Protocol

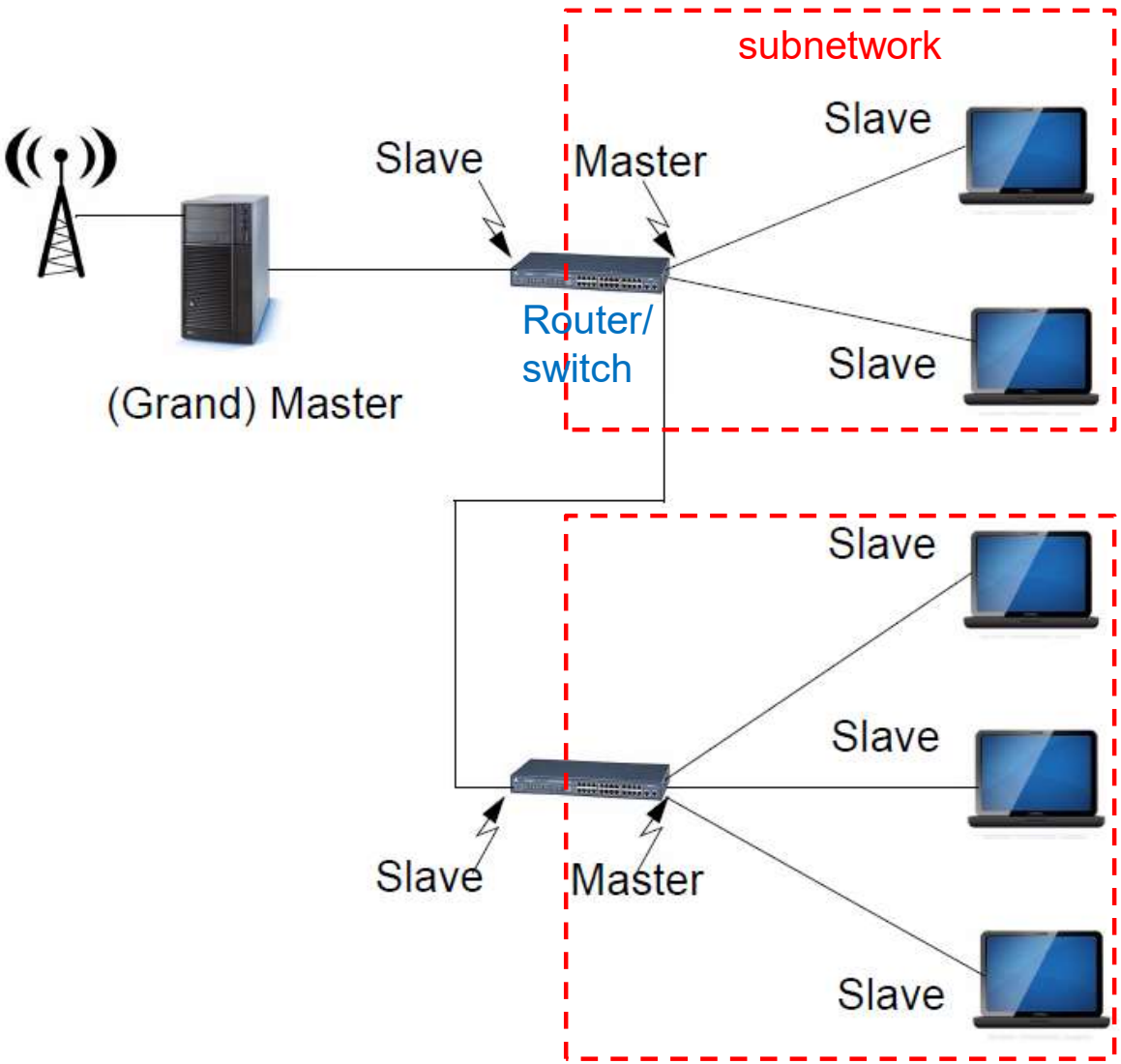
The calculation of δ assumes that the communication delay was identical on the way master → slave and slave → master!

- This is true if we have a **direct connection** between master and slave.
- If there are **switches/routers** on the way, this is not true any more
  - queuing delay, congestion etc. can produce significant fluctuation



Boundary clock switches solve this!

The Precision Time Protocol



Boundary clock (BC) switches contain a clock that is synchronized to a connected master.

- Switch blocks all PTP messages to/from subnetwork
- They themselves behave as masters on all other ports and are used for synchronization by connected slaves.
- Using BC switches, **all synchronizations are over point-to-point connections**.

A **Best-Master-Clock Algorithm (BMC)** determines master-slave relations depending on accuracies of clocks.

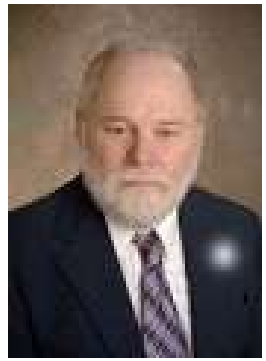
- As result, a **tree structure** is determined **automatically**, with the node containing the best available clock – the **grand master** – as root.

The Precision Time Protocol

In order to achieve high precision, the **time-stamping** has to be implemented with hardware support.

- **Software implementation**
  - A PTP software daemon running on non-standard hardware:
    Synchronization in the range 10 -100 microseconds is achievable.
- **Hardware implementation**
  - Hardware timestamping at master and slave plus PTP-enabled network switches (with boundary clock):
    Synchronization in the range 10 - 100 nanoseconds is achievable.

The Network Time Protocol (NTP)



David L. Mills,
1938-2024
Inventor of NTP

The **NTP** has been adopted as a **de-facto standard for clock synchronization** in **general-purpose** UNIX, Windows, etc. workstations and servers.

- Connection over global Internet by standard routers and gateways is assumed (no specialized hardware).
- There are no particular hardware requirements and customized components.
- Accuracy at the level of milliseconds can be achieved.
- Network overhead is an issue with NTP,
  since the network is shared with demanding Internet applications (such as speech, video);
  clock update intervals can be in the range of minutes (even hours).
- The actual master-slave synchronization algorithm is, in broad terms, similar to that used in PTP.

Acknowledgments

- Most of the slide contents is based on a previous version by Petru Eles, IDA, Linköping University.