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

What is a Real-Time System

- 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*.

<u>Examples</u>:

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

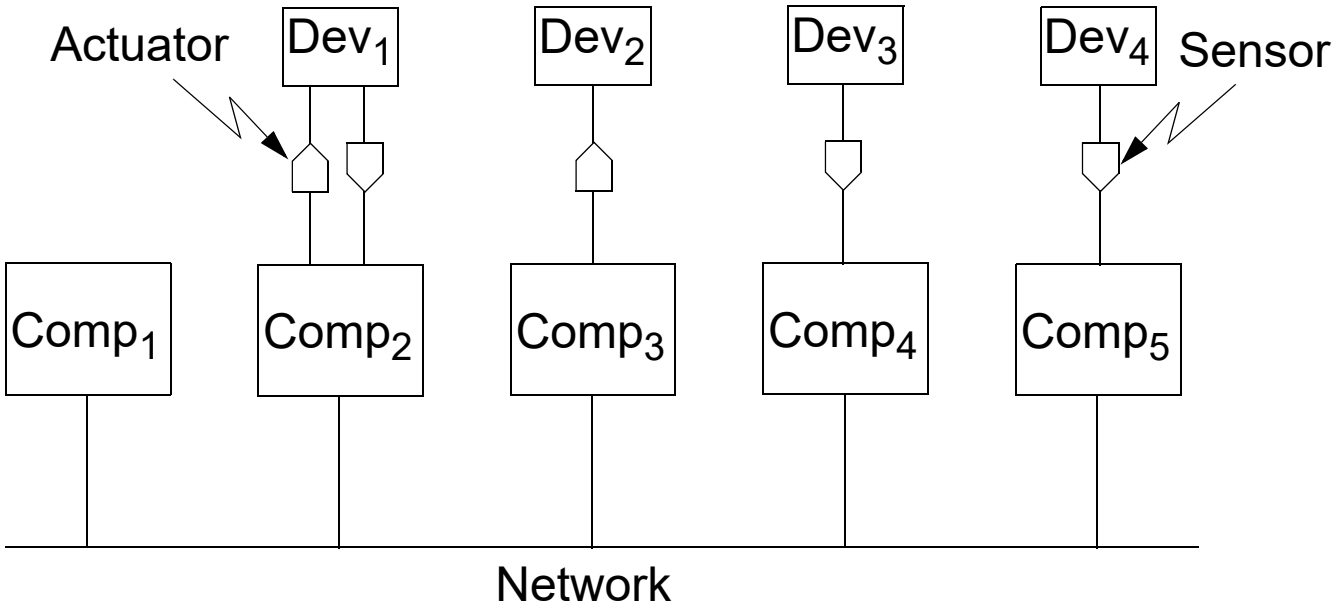
Distributed Real-Time Systems

- **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.**

Distributed Real-Time Systems



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 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 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:**
  - **<u>Hard deadline</u>: has to be met strictly, if not ⇒ "catastrophe".**
    - **should be guaranteed a-priori, off-line.**
  - **<u>Soft deadline</u>: processes can be finished after their deadline, although the value provided by completion may degrade with time.**
  - **<u>Firm deadline</u>: 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.

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.
- <u>Some problems concerning predictability</u>:
  - 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.).

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.**
- **<u>Some problems concerning predictability</u>:**
  - **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":**

  ***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?

Scheduling

<u>The scheduling problem</u>:

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

Scheduling Policies

When are the decisions taken?

- static scheduling
- dynamic scheduling

Are processes preempted?

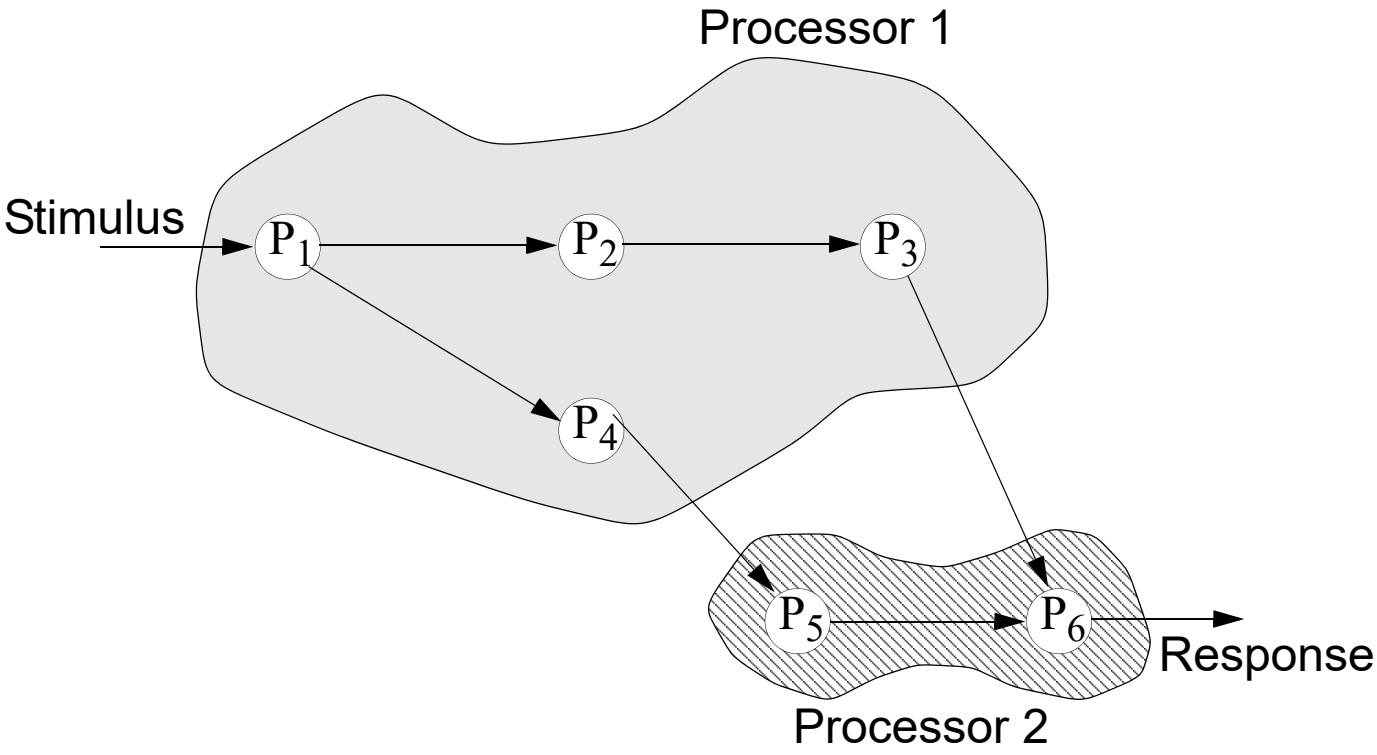
- preemptive scheduling
- non-preemptive scheduling

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.**

Static Scheduling

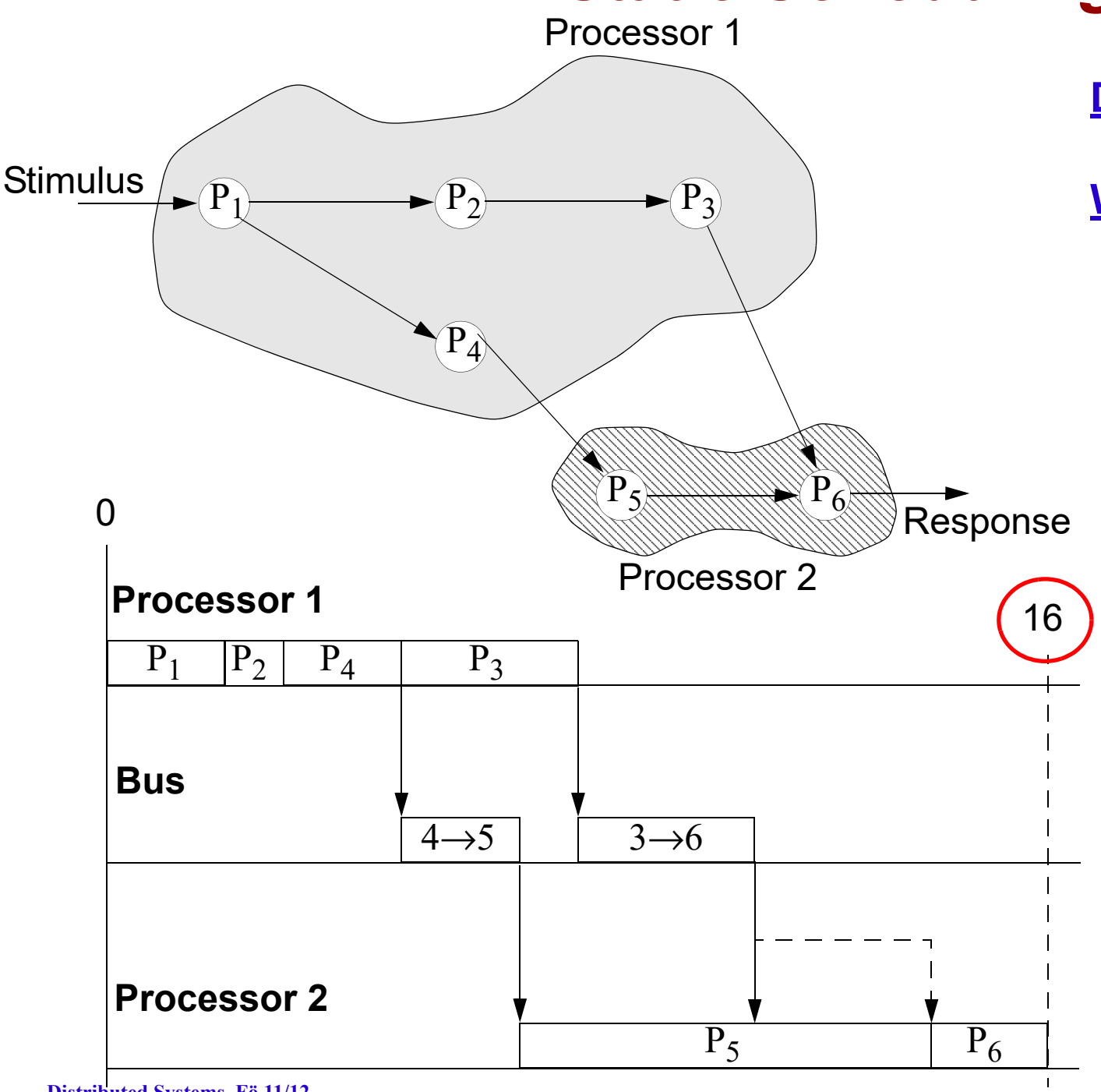


Deadline on *response*: 15

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

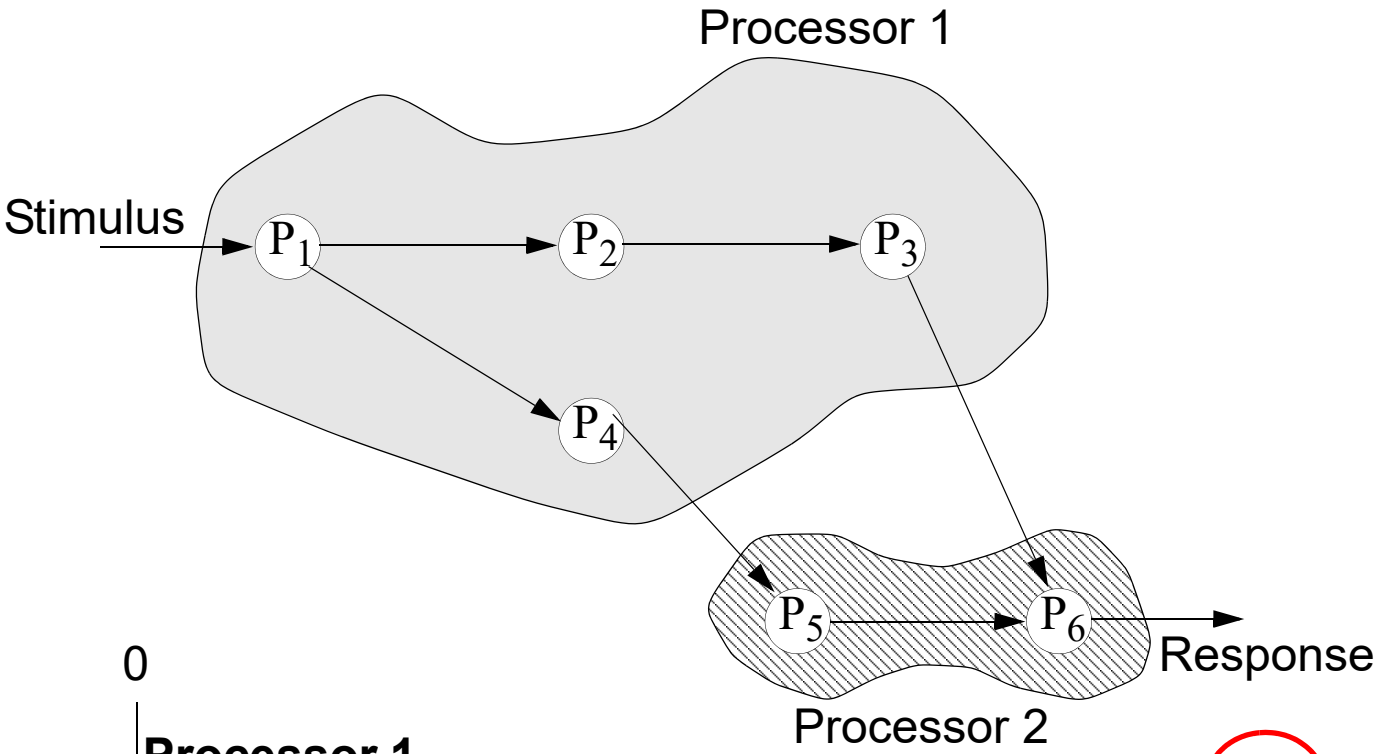


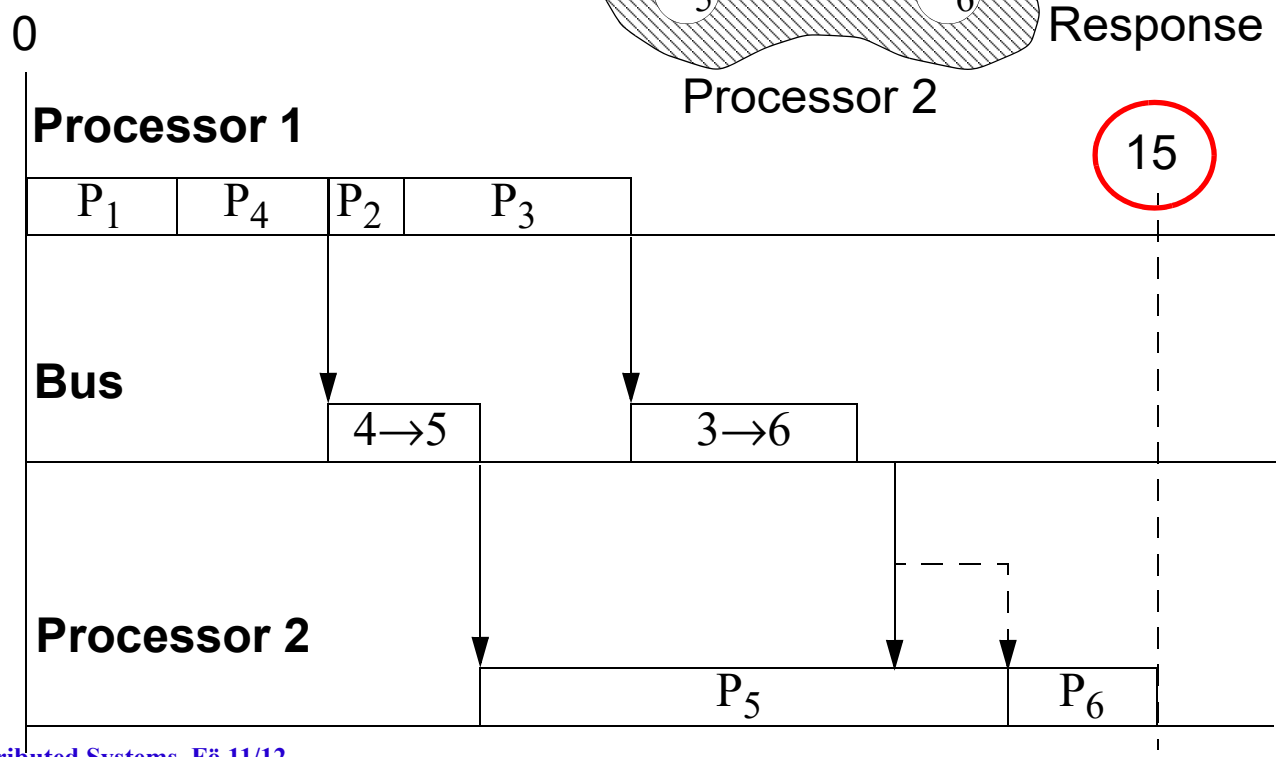
Deadline on *response*: 15

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



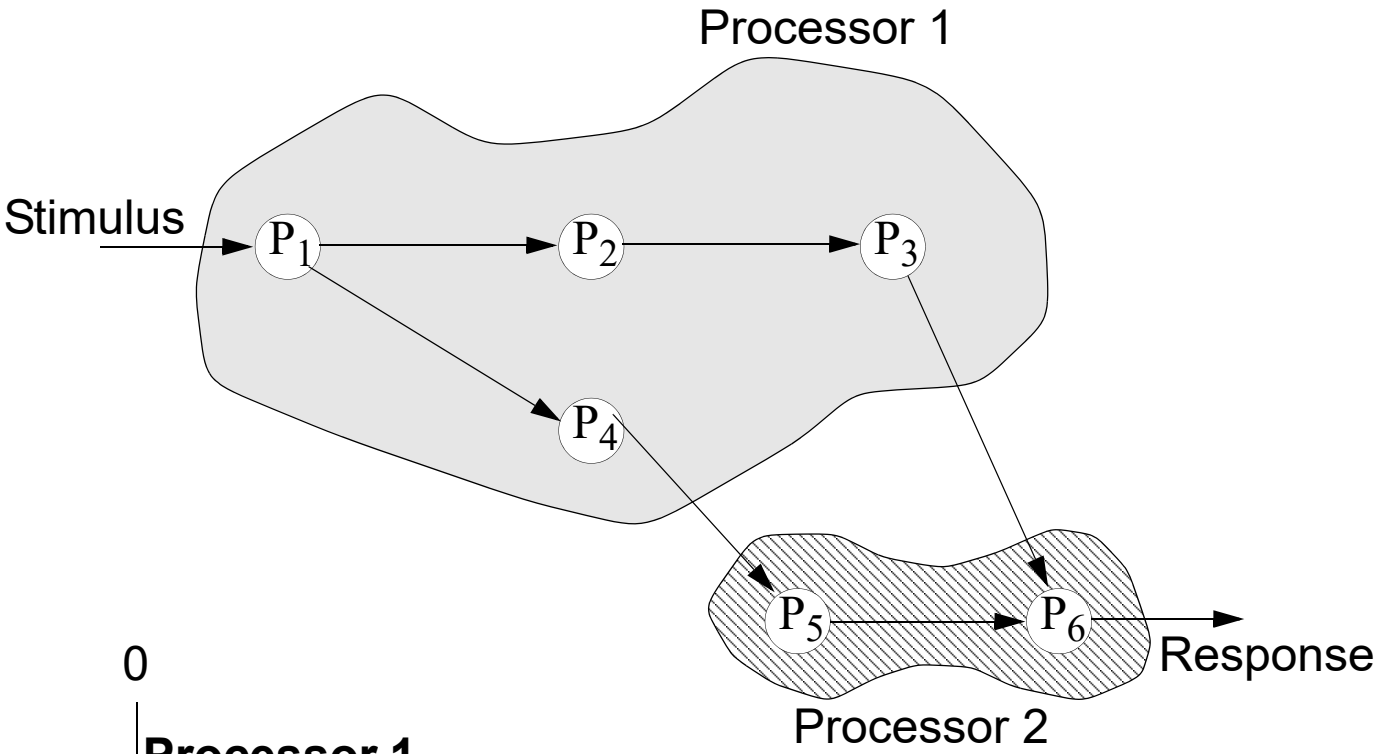


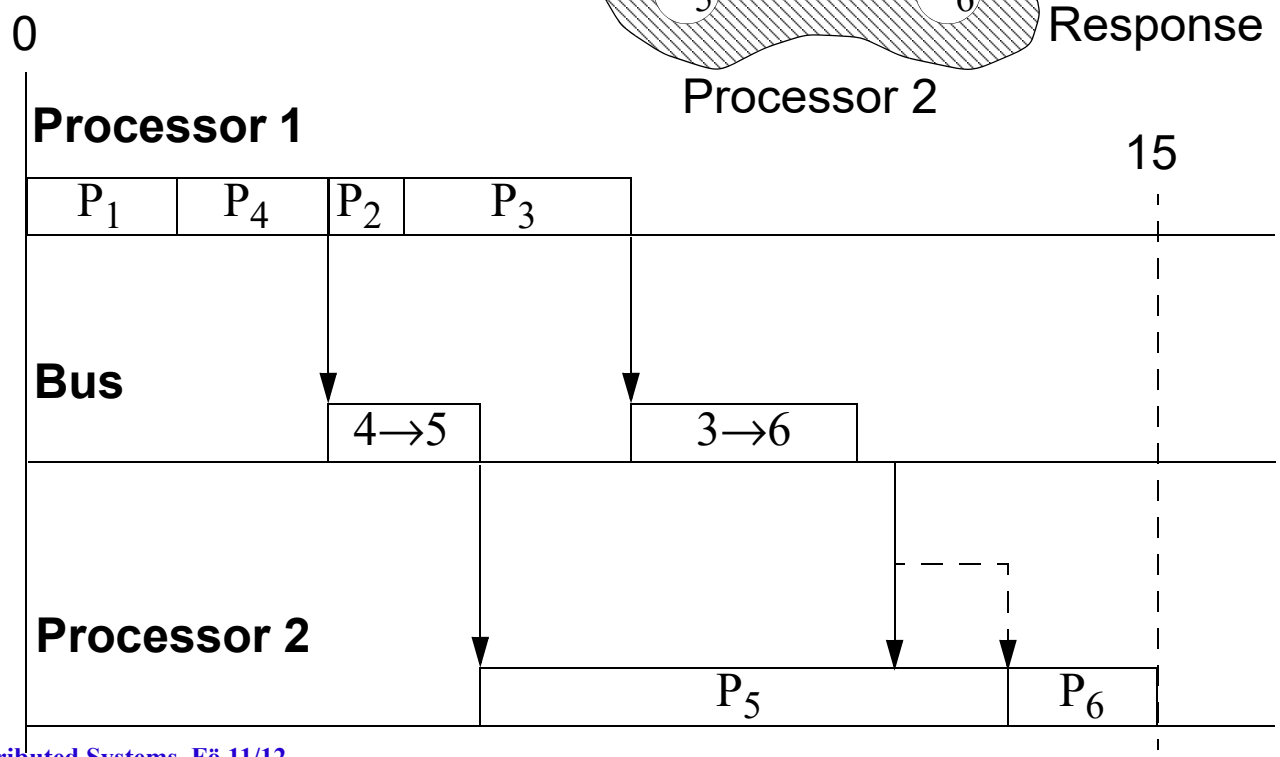
Deadline on *response*: 15

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





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

- <u>What is good</u>?
  - High predictability. Deadlines can be guaranteed - if the scheduling algorithm succeeded in building the schedule table.
  - Easier to debug.
  - Low execution time overhead.

Static Scheduling

- <u>What is good</u>?
  - High predictability. Deadlines can be guaranteed - if the scheduling algorithm succeeded in building the schedule table.
  - Easier to debug.
  - Low execution time overhead.

- <u>What is bad</u>?
  - 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 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.

Dynamic Scheduling

- **Processes 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 analyses is possible:**

<u>For example</u>:

- **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 $\Rightarrow$ 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 the following is true:

$$\sum_{i=1}^{n} \frac{c_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

Specific Issues Concerning *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 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 software.

Computer Clocks

<u>The problems</u>:

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 processors 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 processor in the system (possibly several) is equipped with UTC receivers (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.

External and Internal Synchronization

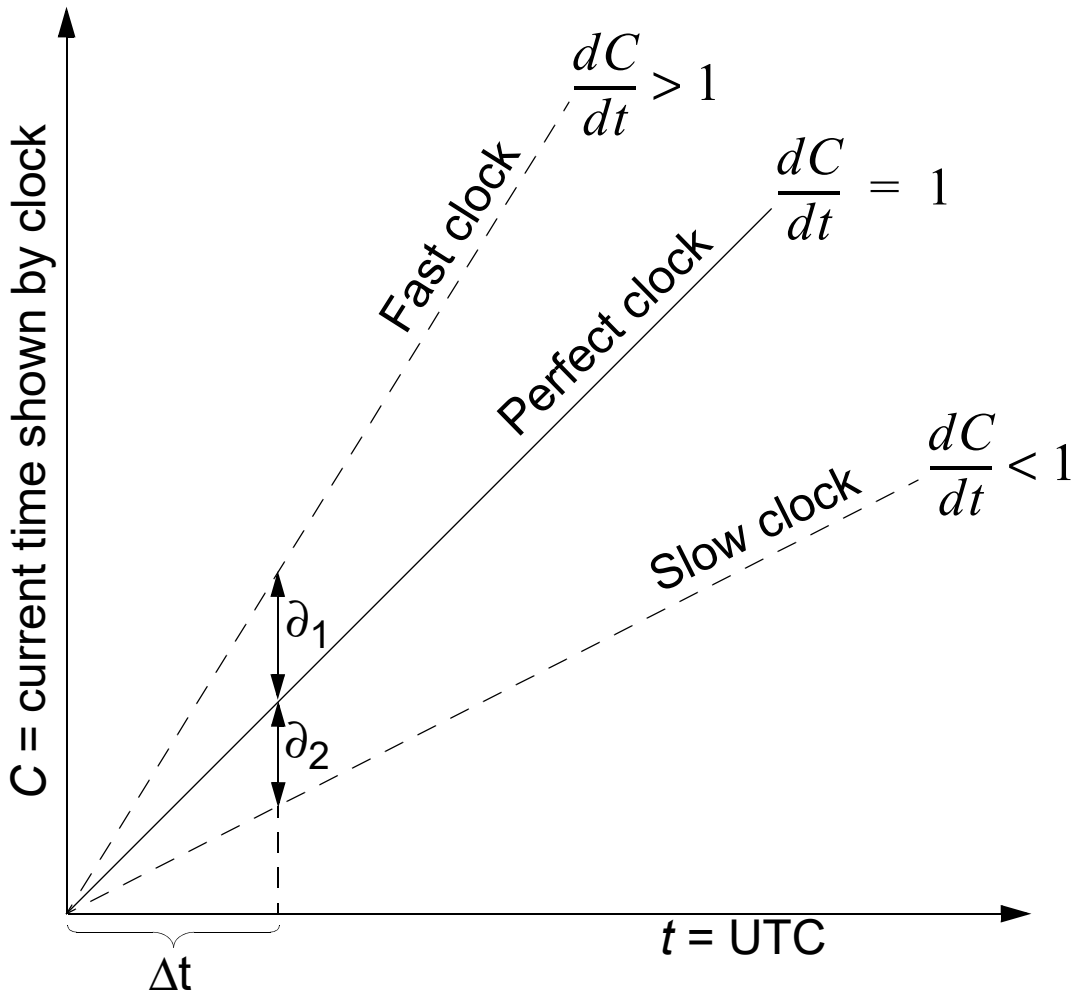
External Synchronization

- Synchronization with a time source external to the distributed systems, such as UTC broadcasting system.
  - One processor in the system (possibly several) is equipped with UTC receivers (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 processors of the system.
  - Needed in order to keep a consistent view of time over the system.
  - A few processors synchronize externally and the whole system is kept consistent by internal synchronization.
  - Sometimes only internal synchronization is performed (we don’t care for the drift from external/real time).

Drifting of Clocks

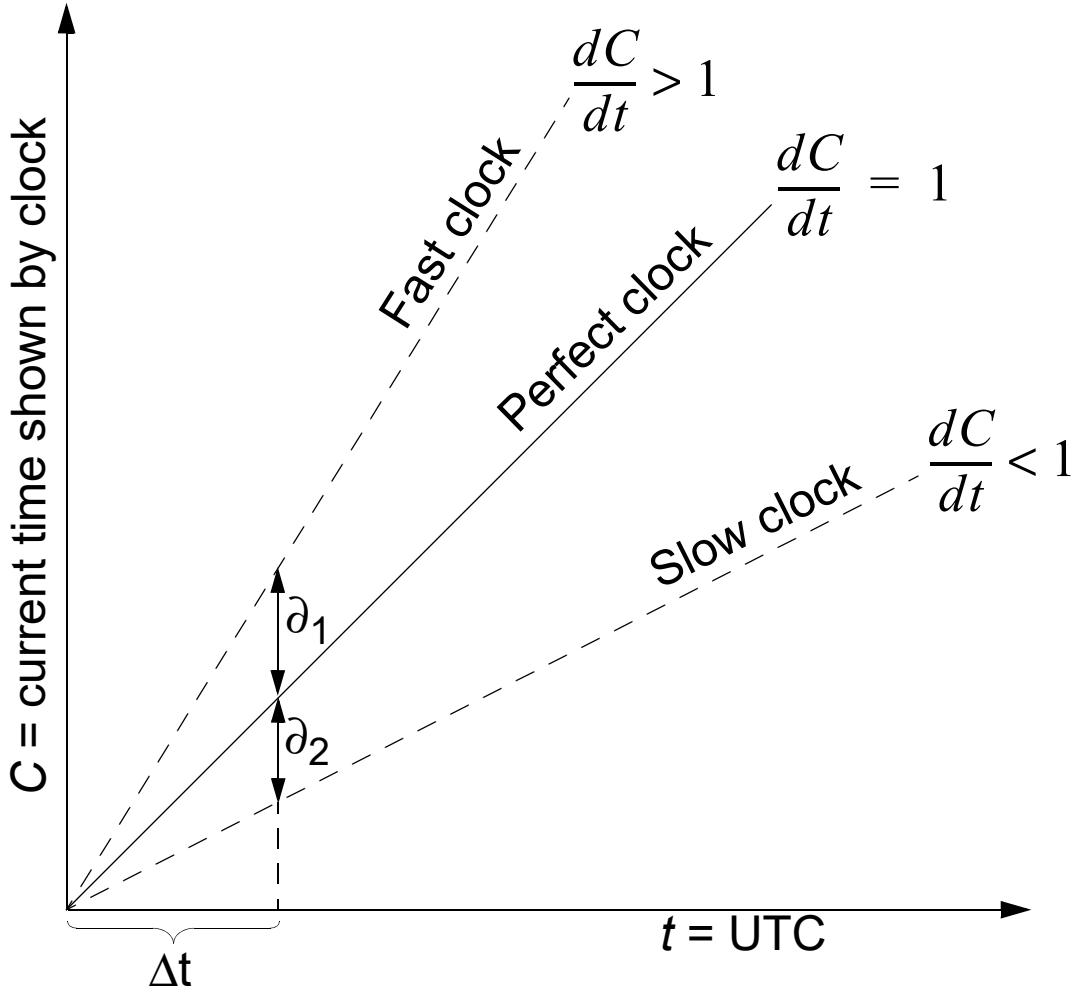


∂_1: drift of first clock after Δt.

∂_2: drift of second clock after Δt.

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

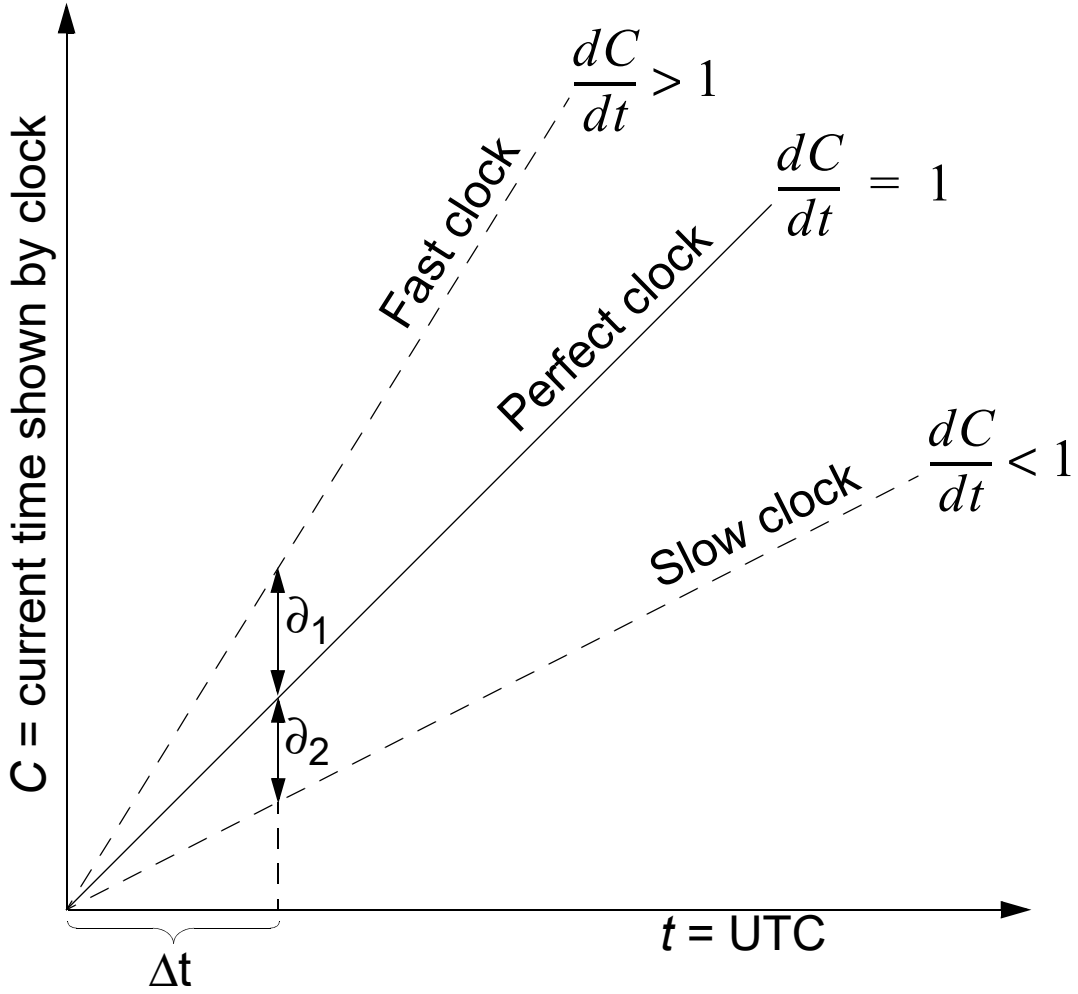
Drifting of Clocks



<u>In ideal case</u>, the clock shows UTC: *C* = *t*.

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

Drifting of Clocks



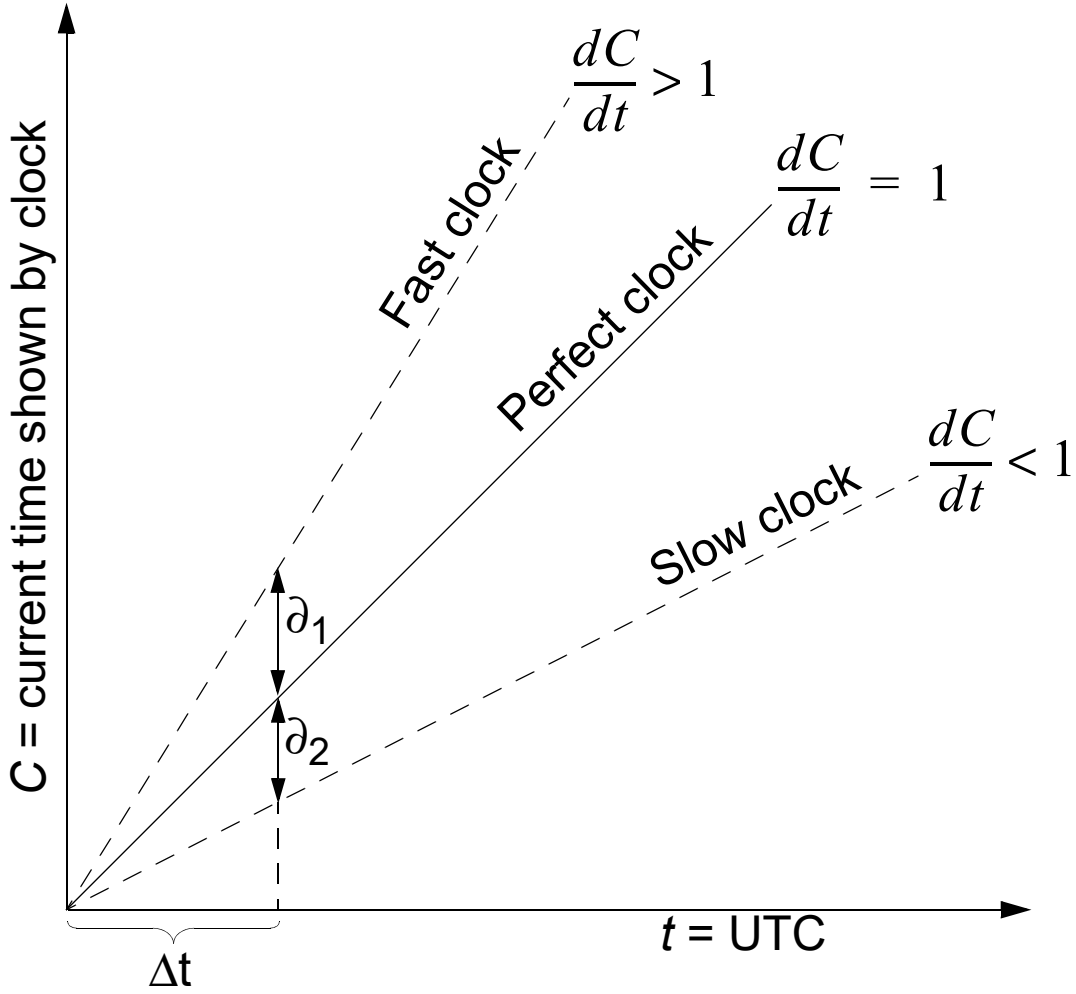
<u>In ideal case</u>, the clock shows UTC: *C* = *t*.

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

<u>In reality</u>: $\frac{dC}{dt} = 1 \pm \rho$

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

Drifting of Clocks



<u>In ideal case</u>, the clock shows UTC: *C* = *t*.

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

<u>In reality</u>: $\frac{dC}{dt} = 1 \pm \rho$

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

Two processors with similar clocks could be apart with:

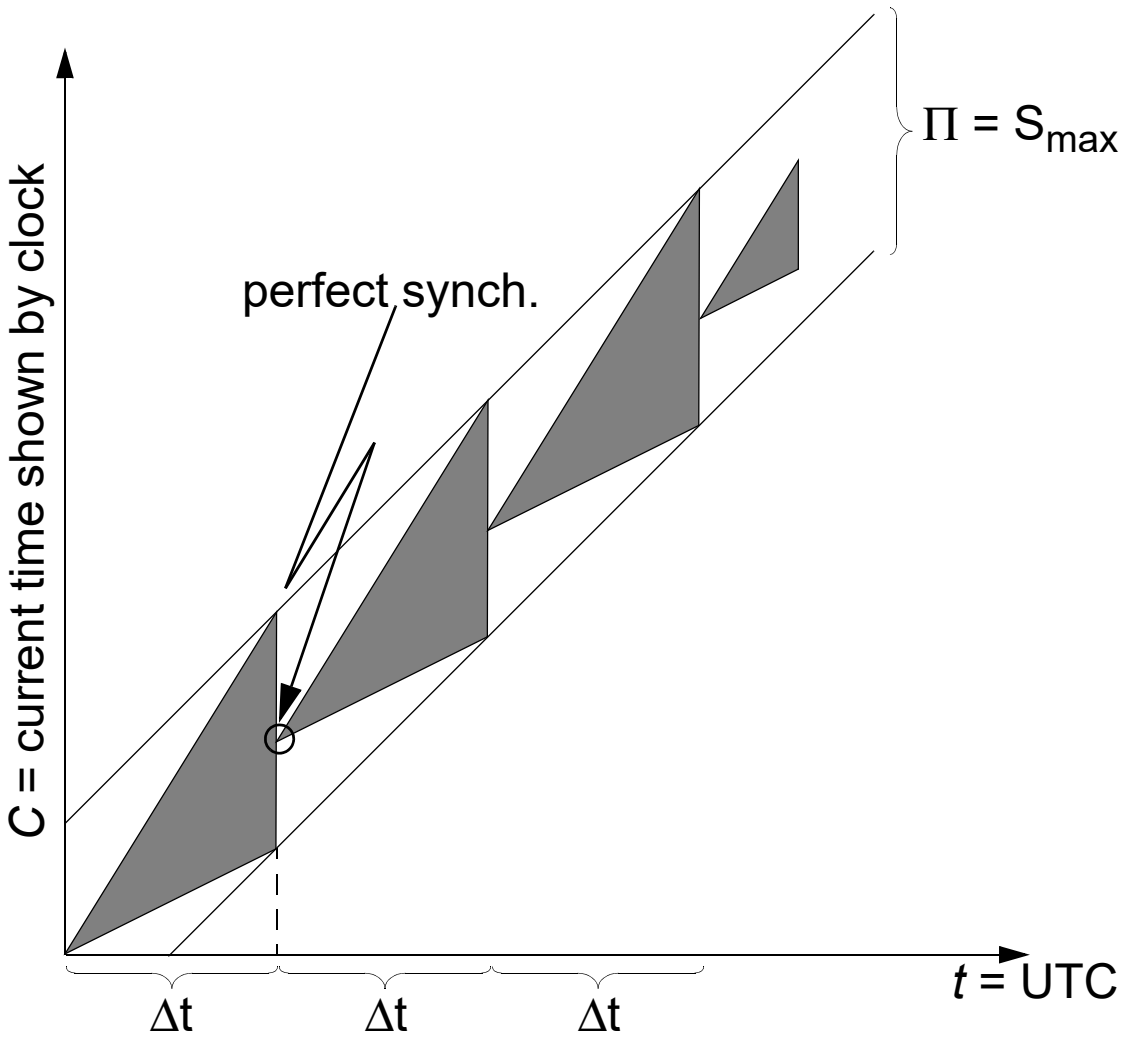
$$S = 2\rho\Delta t$$

To guarantee a *precision* Π (max. skew between the two clocks), the clocks have to be synchronized at an interval:

$$S_{max} = \Pi$$

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

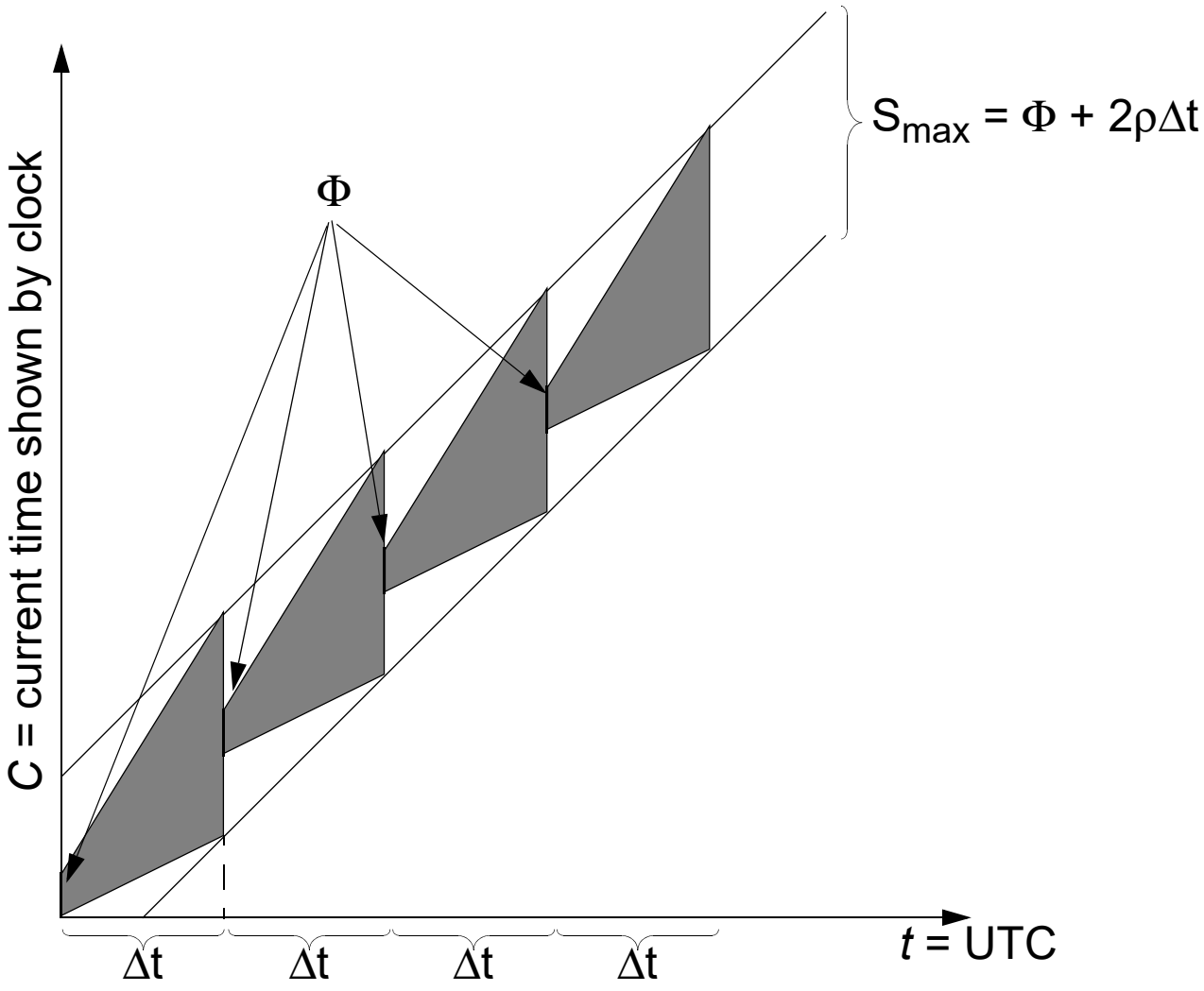
Drifting of Clocks



Only if we assume that after synchronisation the clocks are perfectly aligned, then it is the case that:

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

Drifting of Clocks



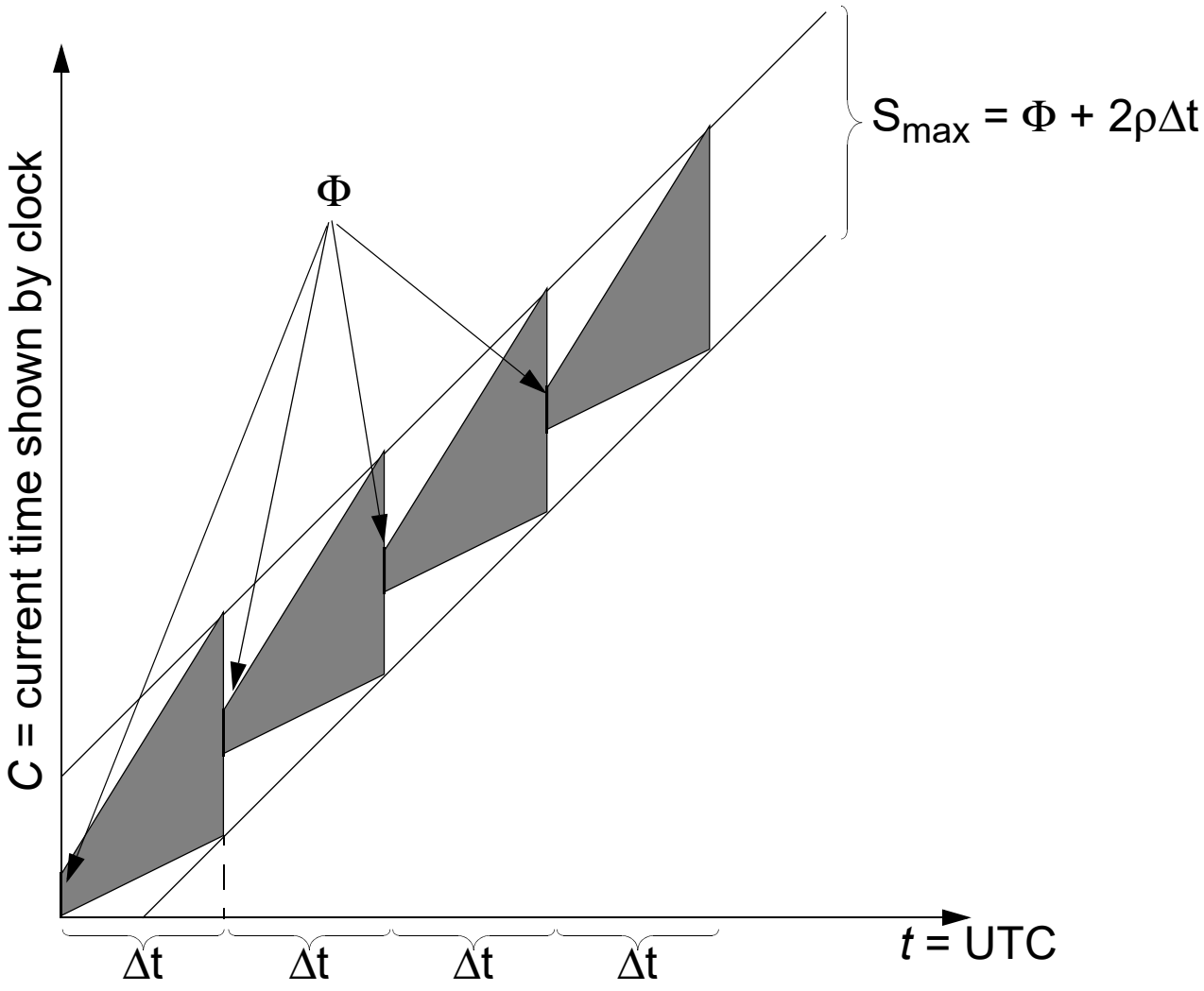
Only if we assume that after synchronisation the clocks are perfectly aligned, then it is the case that:

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

In reality, the alignment after synchronisation is not perfect:

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

Drifting of Clocks



Only if we assume that after synchronisation the clocks are perfectly aligned, then it is the case that:

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

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:

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

Clock Synchronization Algorithms

- **Centralized Algorithms**
  - **There exists one particular node, the so called *time server node*.**
    - **<u>Passive time server</u>: 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.**
    - **<u>Active time server</u>: 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.**

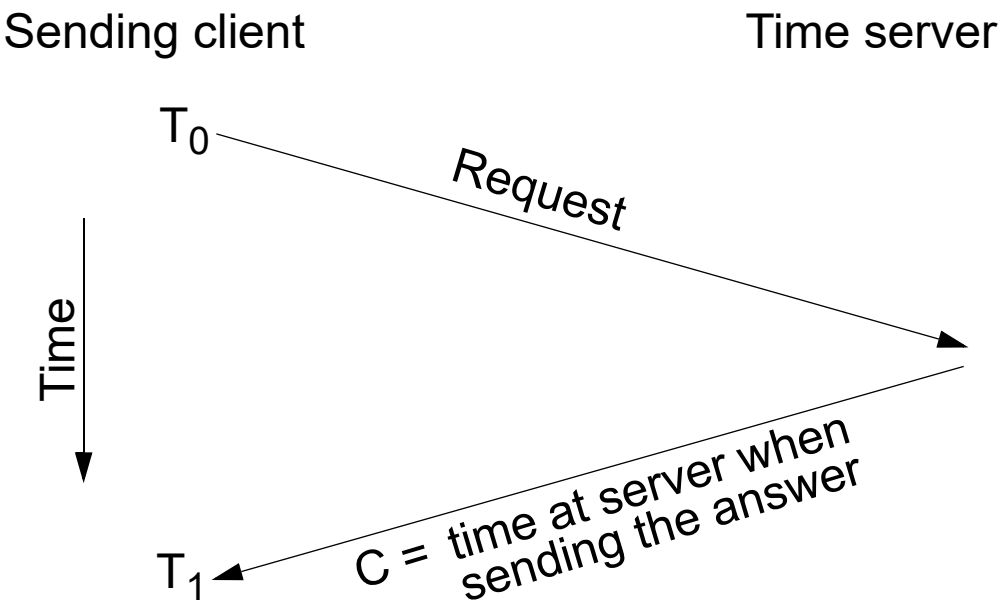
Clock Synchronization Algorithms

- **Centralized Algorithms**
  - **There exists one particular node, the so called *time server node*.**
    - **<u>Passive time server</u>: 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.**
    - **<u>Active time server</u>: 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 processors (possibly all) have an UTC receiver, this doesn't 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 an 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:**



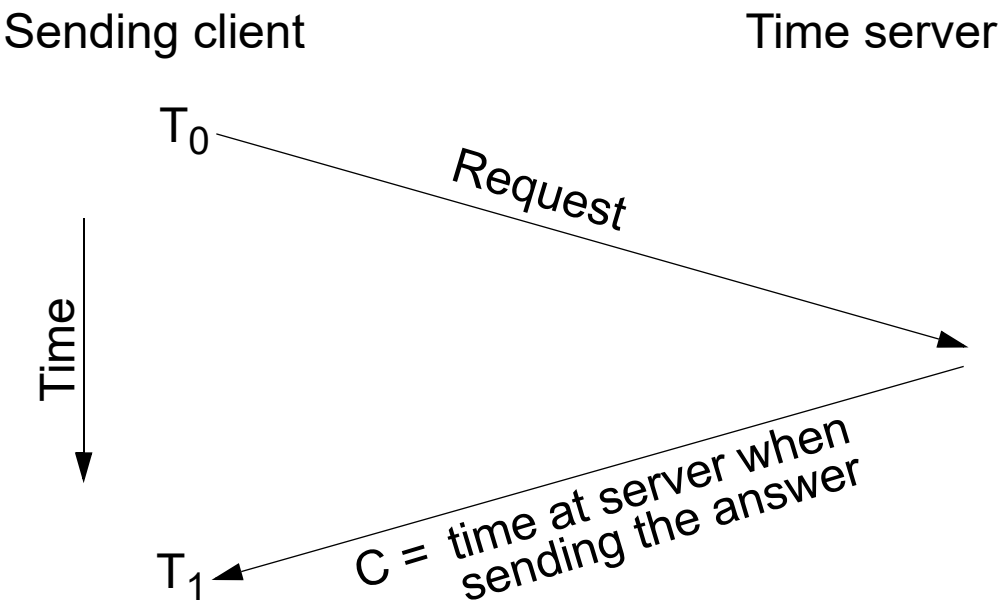
Simplest: set receiving client clock

$$T_{rec} = C$$

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

Cristian's Algorithm

- **Cristian's algorithm is a centralized algorithm with passive time server. The server is supposed to deliver the correct time (has an 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:**



Simplest: set receiving client clock

$$T_{rec} = C$$

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

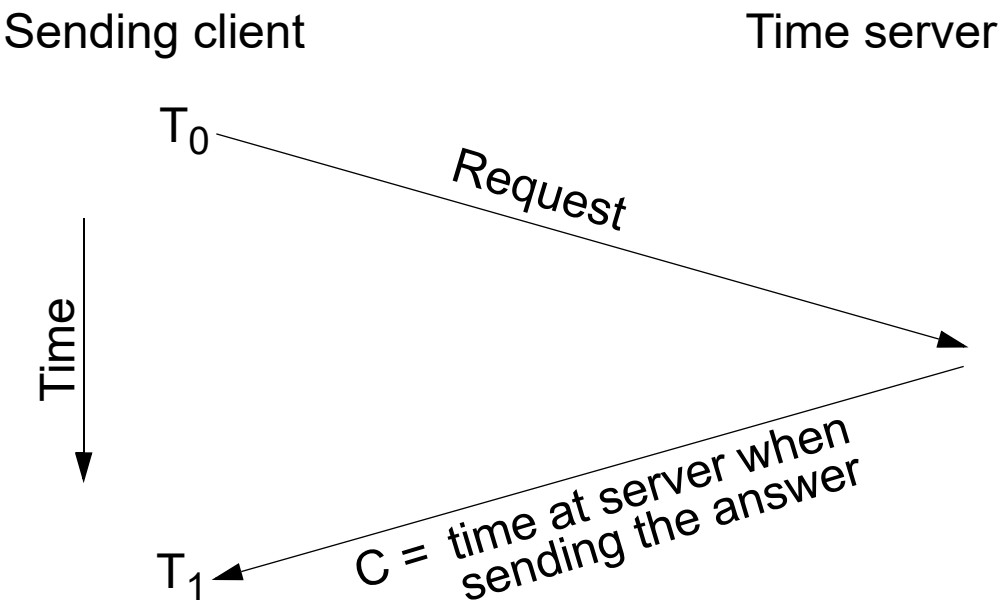
$$T_{rec} = C + T_{trans}$$

How large is T_{trans}?

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

Cristian's Algorithm

- **Cristian's algorithm is a centralized algorithm with passive time server. The server is supposed to deliver the correct time (has an 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 replay to get back to the client:

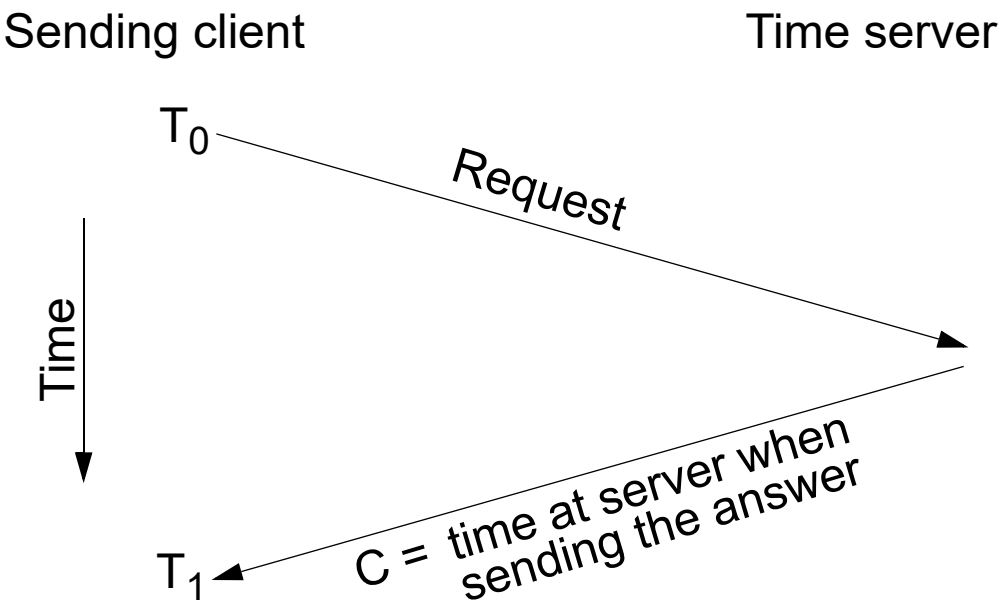
$$T_{rec} = C + T_{trans}$$

How large is T_{trans}? Estimation:

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

Cristian's Algorithm

- **Cristian's algorithm is a centralized algorithm with passive time server. The server is supposed to deliver the correct time (has an 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 replay to get back to the client:

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

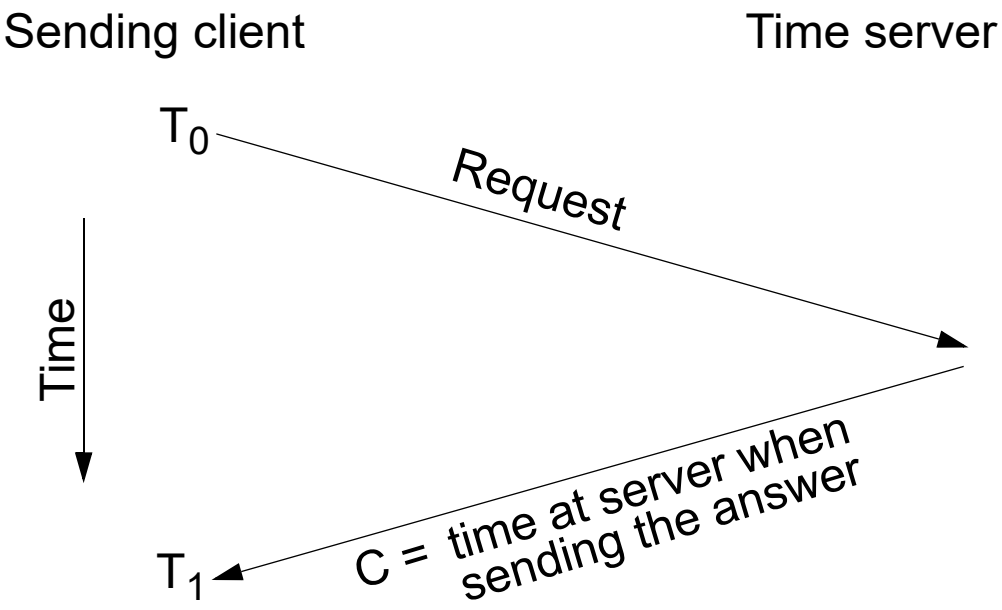
How large is T_{trans}? Estimation:

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

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 an 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:**

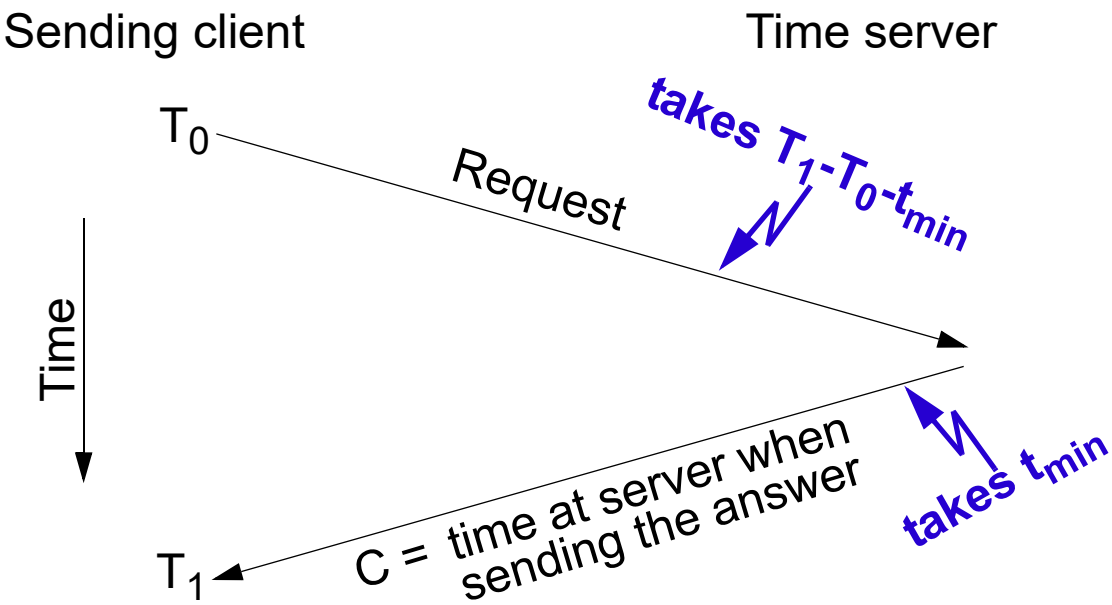


Suppose the *minimum* time t_{min} needed for a communication between the machine and the time server is known:

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

Cristian's Algorithm

- **Cristian's algorithm is a centralized algorithm with passive time server. The server is supposed to deliver the correct time (has an 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:**



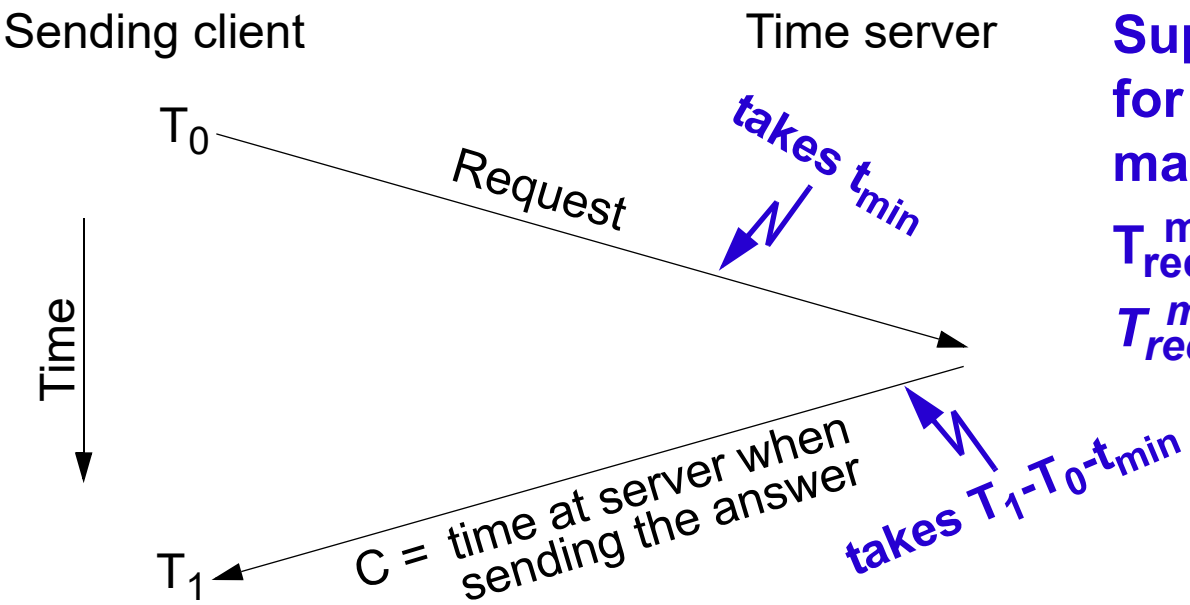
Suppose the *minimum* time t_{min} needed for a communication between the machine and the time server is known:

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

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

Cristian's Algorithm

- **Cristian's algorithm is a centralized algorithm with passive time server. The server is supposed to deliver the correct time (has an 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:**



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

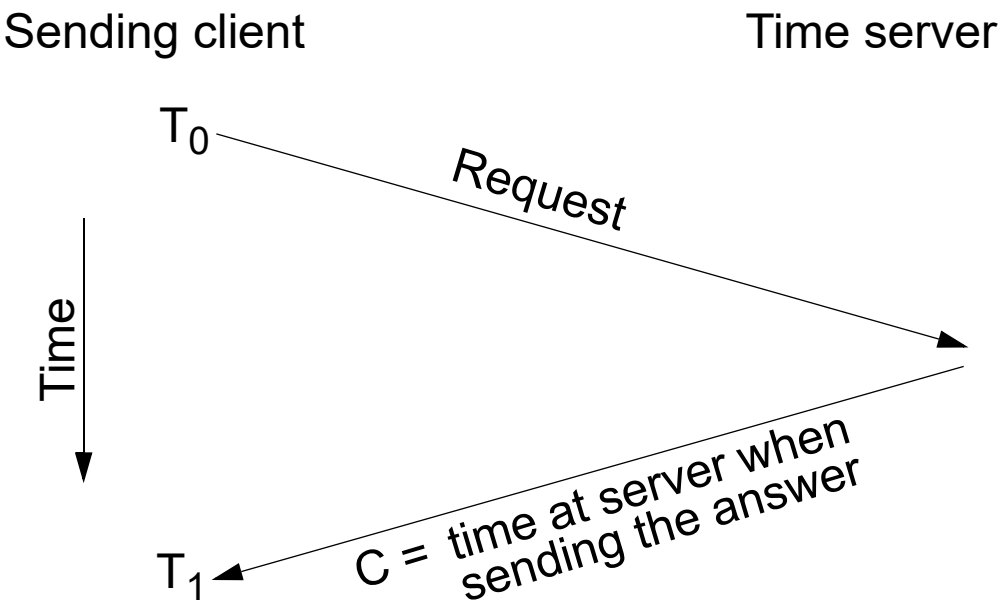
$$T_{rec}^{min} = C + t_{min}$$

$$T_{rec}^{max} = C + (T_1 - T_0) - t_{min}$$

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

Cristian's Algorithm

- **Cristian's algorithm is a centralized algorithm with passive time server. The server is supposed to deliver the correct time (has an 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:**



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

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

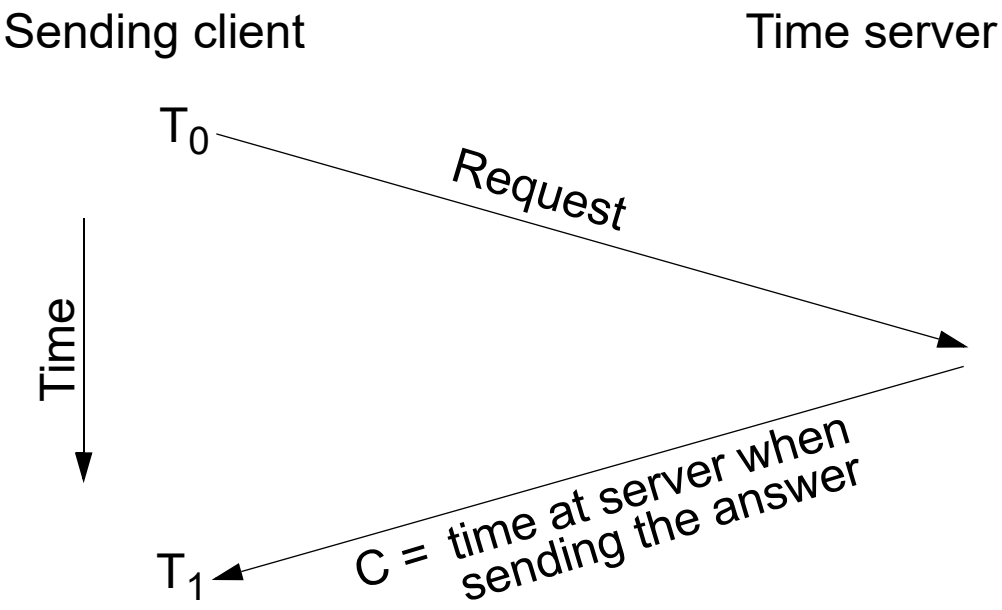
$$T_{rec}^{max} = C + (T_1 - T_0) - t_{min}$$

The range: $T_{rec}^{max} - T_{rec}^{min} = (T_1 - T_0) - 2t_{min}$

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

Cristian's Algorithm

- **Cristian's algorithm is a centralized algorithm with passive time server. The server is supposed to deliver the correct time (has an 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} needed for a communication between the machine and the time server is known:

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

$$T_{rec}^{max} = C + (T_1 - T_0) - t_{min}$$

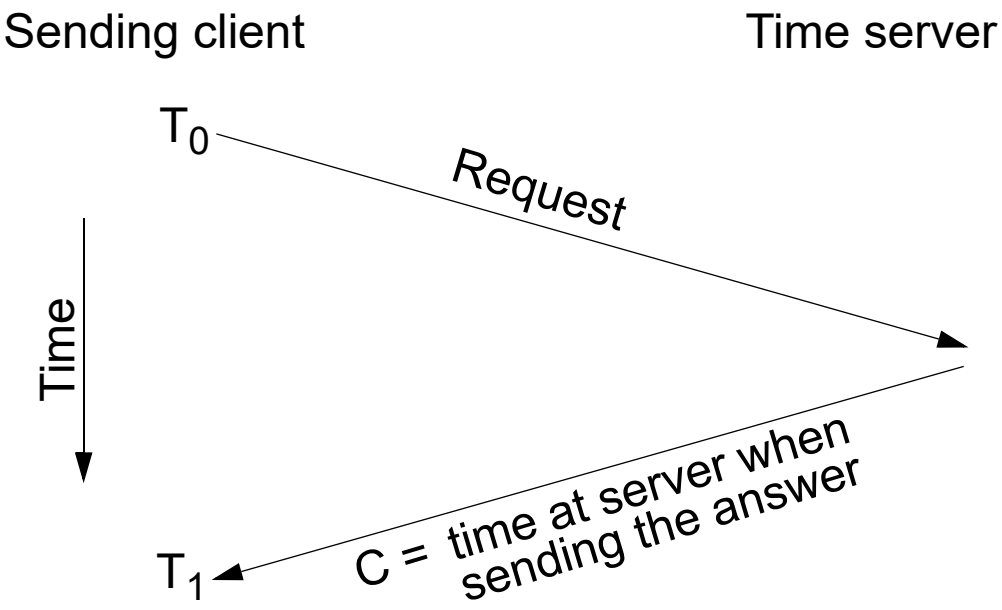
The range: $T_{rec}^{max} - T_{rec}^{min} = (T_1 - T_0) - 2t_{min}$

<u>Time set with an absolute accuracy of</u>:

$$\pm((T_1 - T_0)/2 - t_{min})$$

Cristian's Algorithm

- **Cristian's algorithm is a centralized algorithm with passive time server. The server is supposed to deliver the correct time (has an 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} needed for a communication between the machine and the time server is known:

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

$$T_{rec}^{max} = C + (T_1 - T_0) - t_{min}$$

The range: $T_{rec}^{max} - T_{rec}^{min} = (T_1 - T_0) - 2t_{min}$

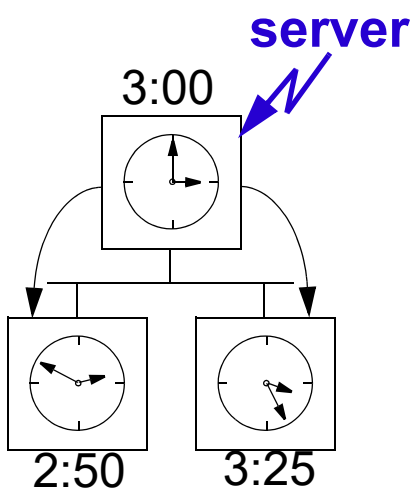
<u>Time set with an absolute accuracy of</u>:

$$\pm((T_1 - T_0)/2 - t_{min})$$

<u>Improve accuracy</u>: several requests can be issued; the answer with the smallest $(T_1 - T_0)$ is used to update the clock.

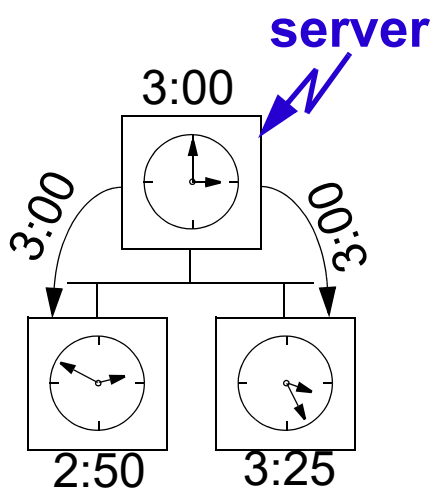
The Berkeley Algorithm

- 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

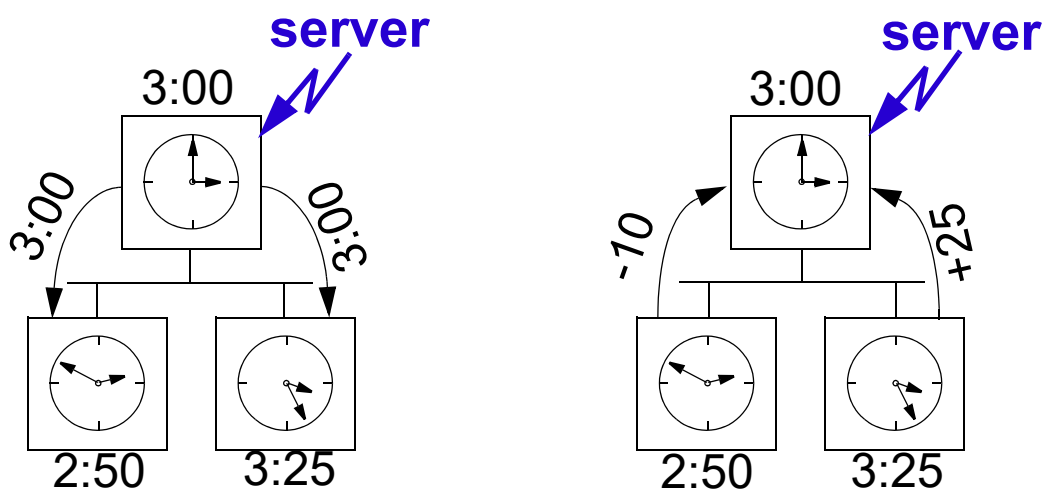
- The Berkeley algorithm is a centralized algorithm with active time server. It tries also to address the problem of possible faulty clocks.



  - The server polls periodically every machine.

The Berkeley Algorithm

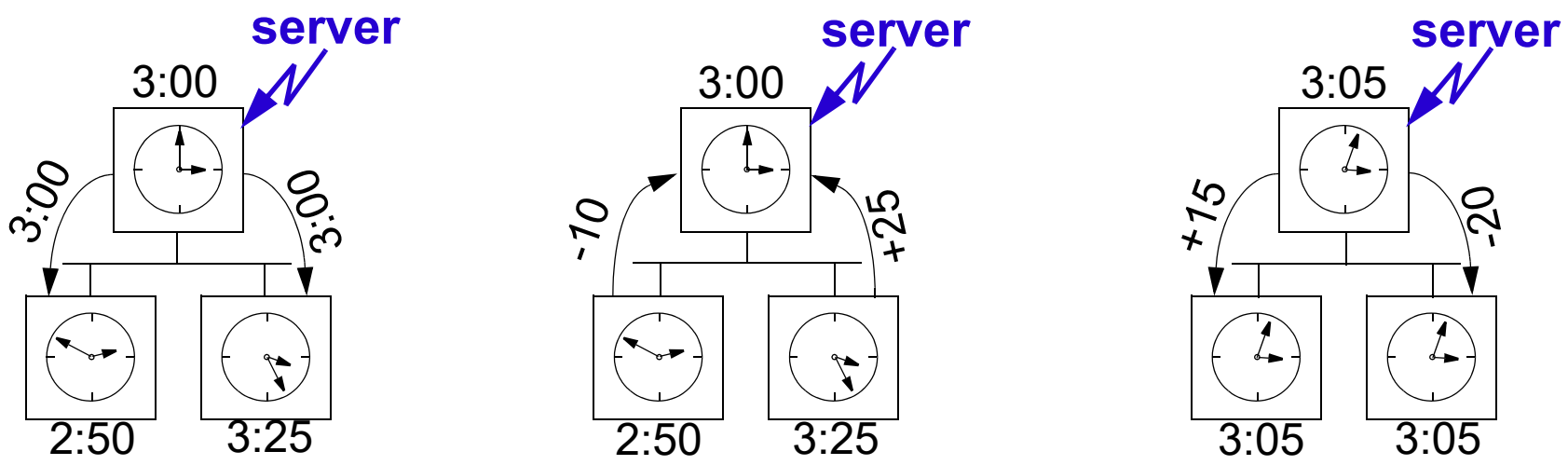
- The Berkeley algorithm is a centralized algorithm with active time server. It tries also to address the problem of possible faulty clocks.



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

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 to ask for the current time.
  - Based on received values, the server computes an average.
  - The server, finally, tells each machine with 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 analysis 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.**

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

Distributed Clock Synchronization

Localized Averaging Algorithm

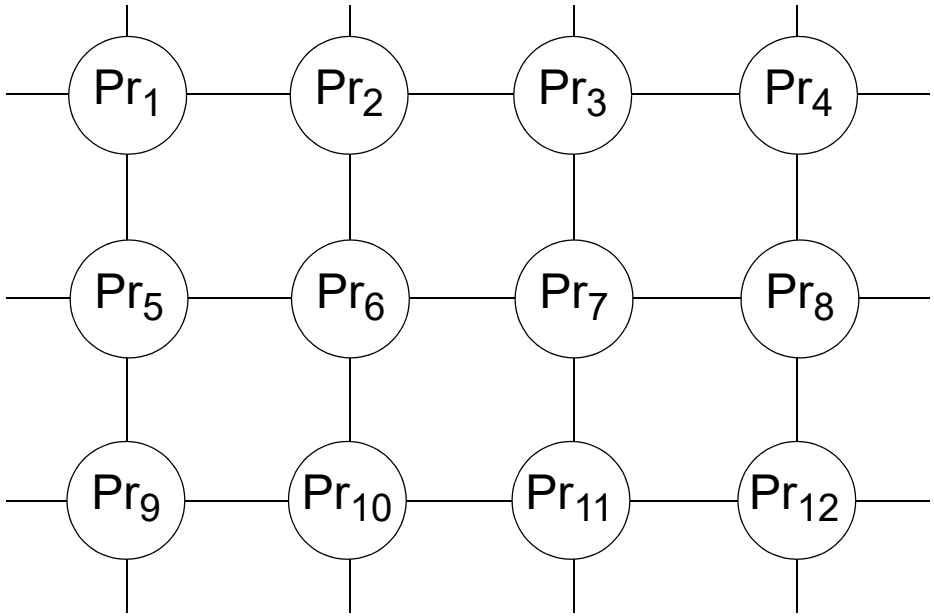
- With a large network it is impractical to perform synchronization among all nodes of the system. Broadcasting synchronization messages from each node to all other nodes generates a huge traffic.
- In large networks, nodes are logically grouped into structures, like grid or ring. Each node synchronizes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

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 a huge traffic.
- In large networks, nodes are logically grouped into structures, like grid or ring. Each node synchronizes with its neighbours in the structure.

  In the grid below:
  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}.

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}.**

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}.**

- **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.**

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}.**

- **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).**

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.**
- **<u>We have to fix *a* and *b*</u> that are to be used during the adjustment period:**

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:

    $T_{curr} = \theta(1 + a) + b \quad (1)$
  - after N ticks
    - the "real" time will be: $T_{new} + N\nu$.
    - the software clock will show: $(\theta + N\nu)(1 + a) + b$

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:

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

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

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:

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

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

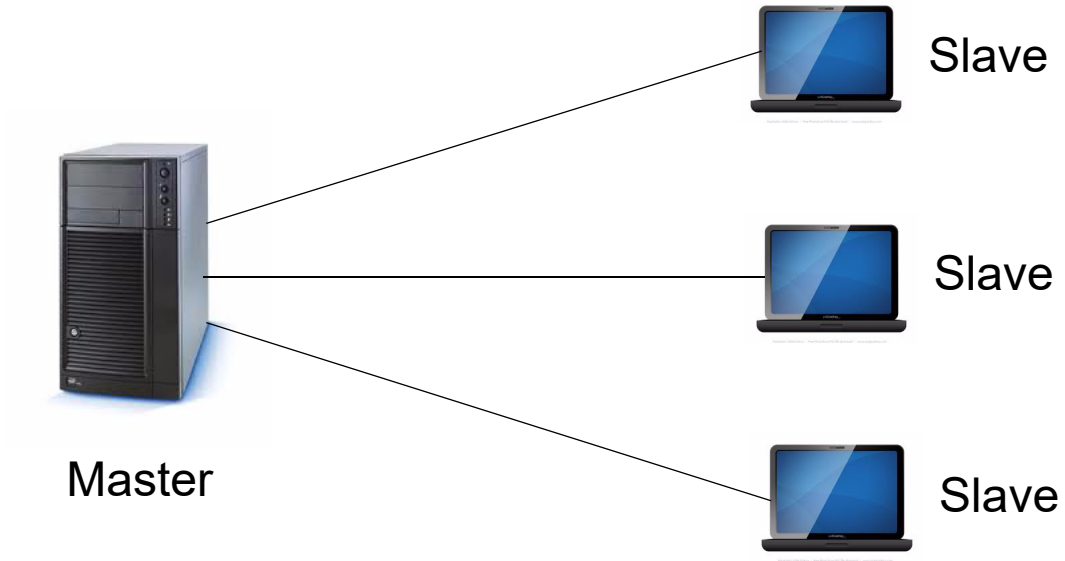
The Precision Time Protocol (PTP)

- **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 synchronises one or several slaves connected to it* (remember Cristian's algorithm).**

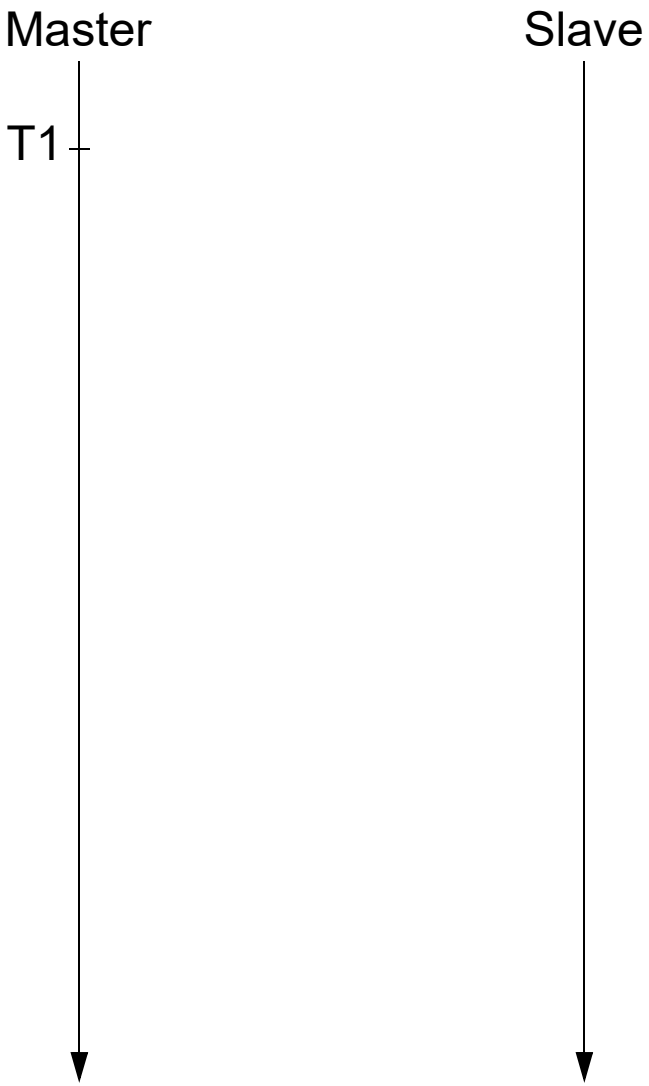
The Precision Time Protocol



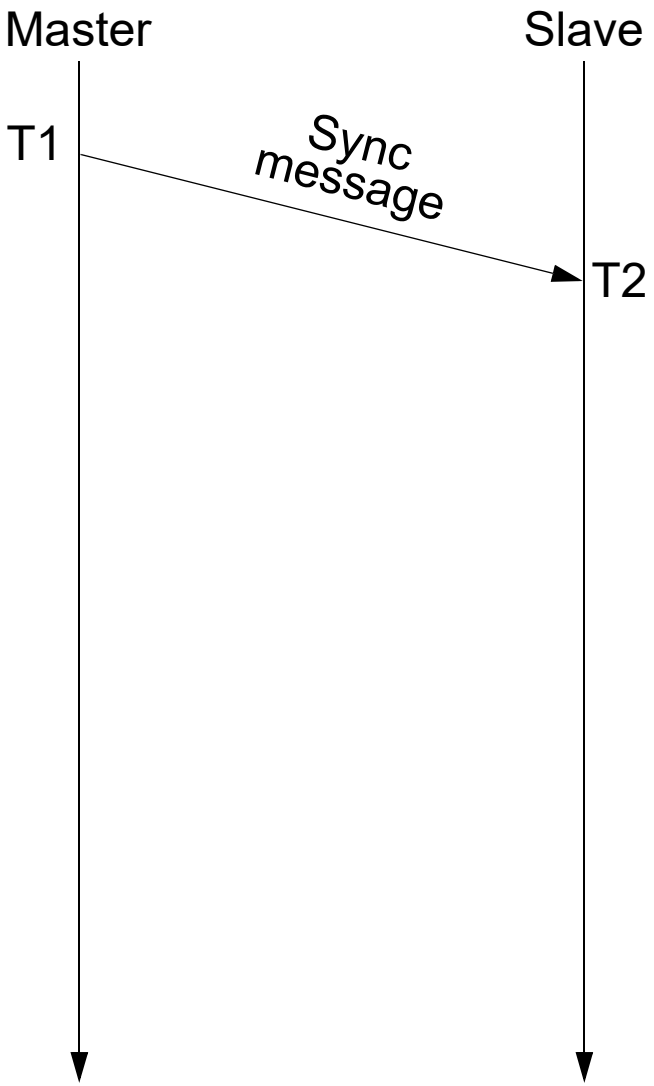
- The *Master* is provider of time; the *Slave* synchronises to the *Master*.

- 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

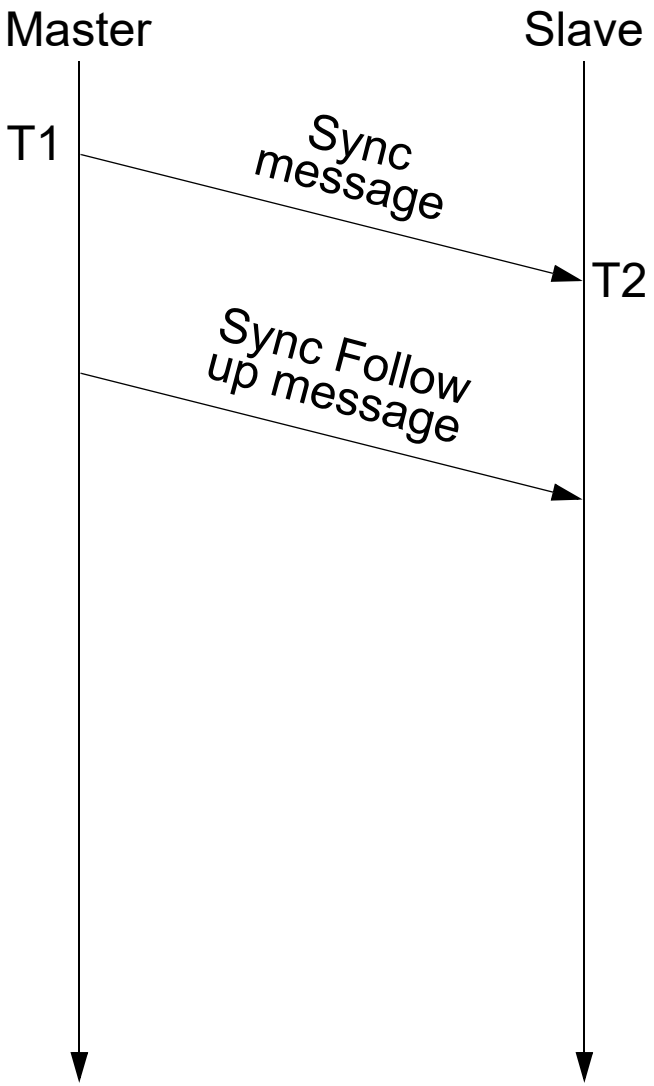


The Precision Time Protocol



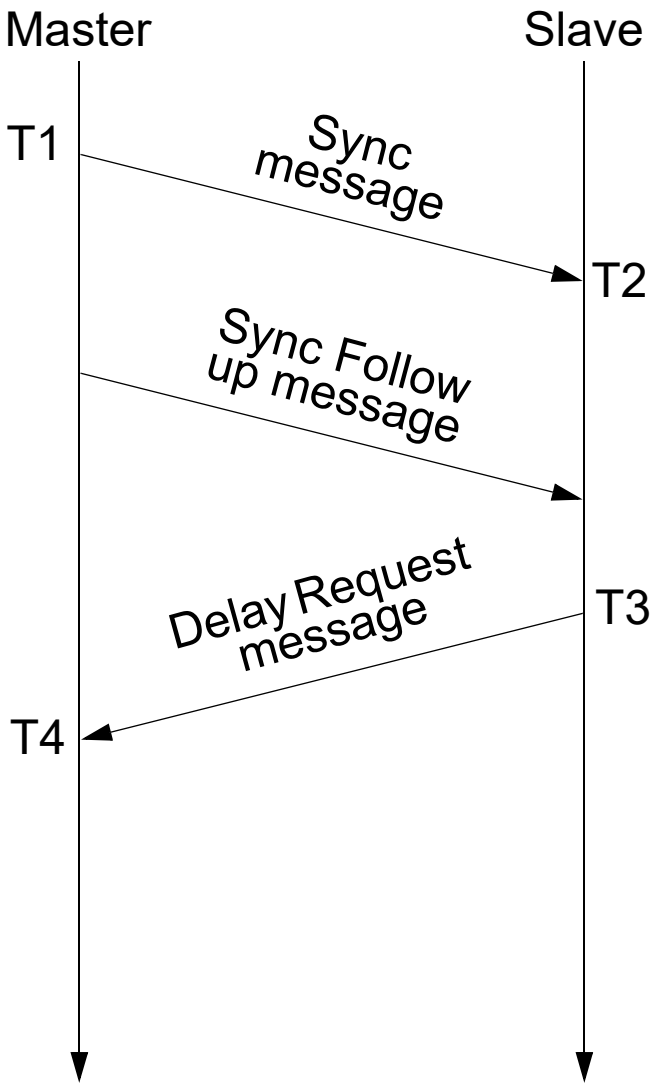
- ***Sync*: issued at T1, arrives at T2;**

The Precision Time Protocol



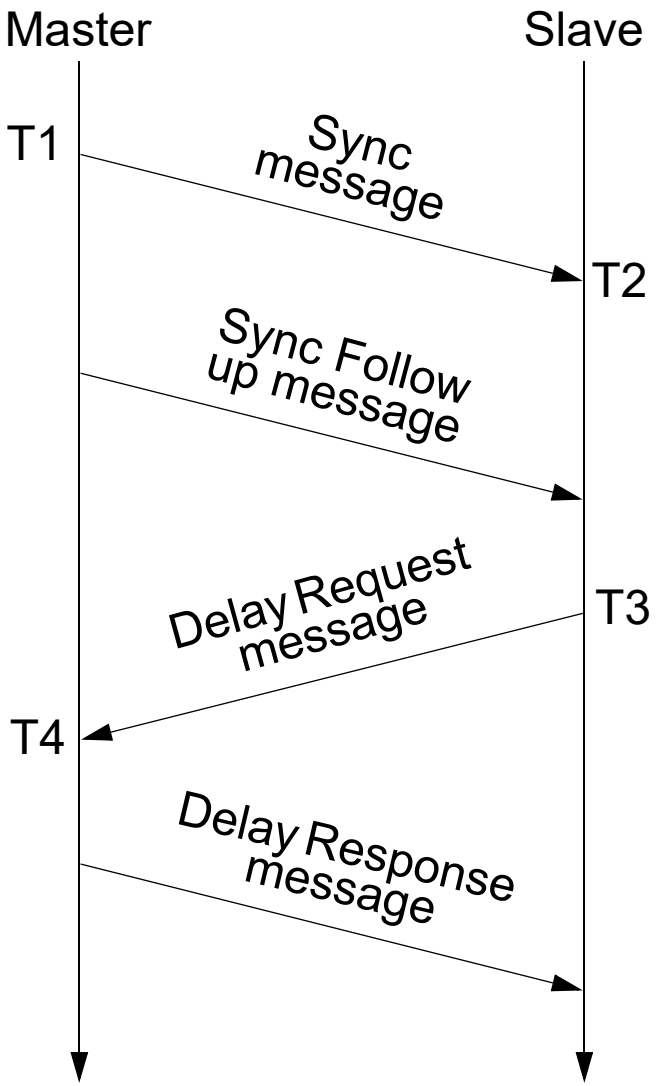
- ***Sync*: issued at T1, arrives at T2;**
- ***Sync Follow up*: carries the value of T1;**

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;**

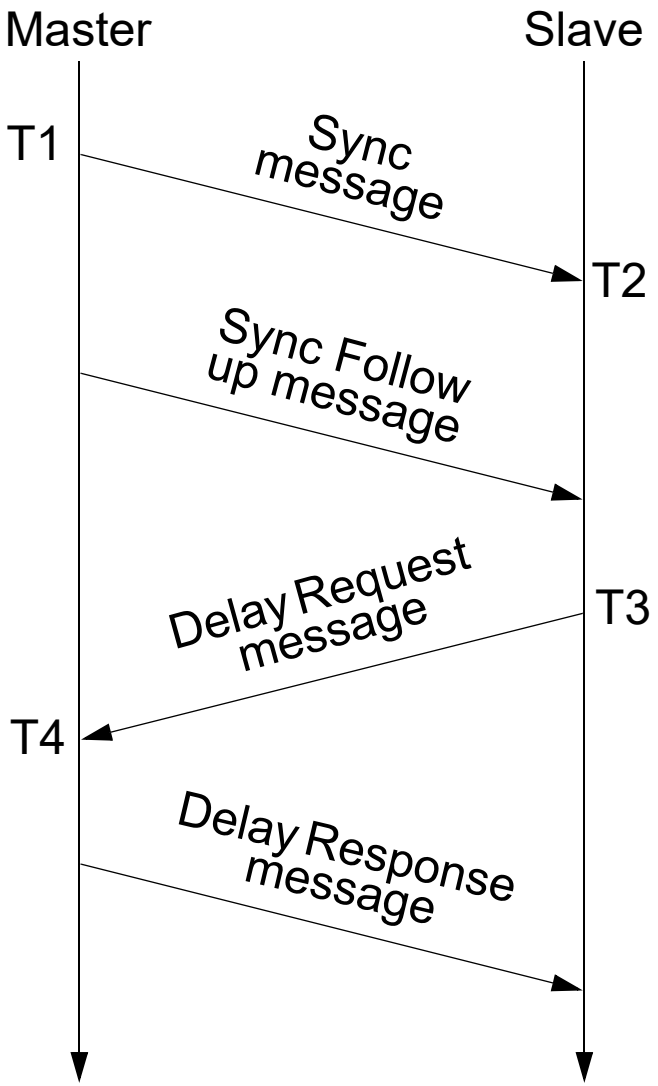
The Precision Time Protocol



- ***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.**

<u>Question</u>: Why does not the *Sync* message carry the value of T1?

The Precision Time Protocol

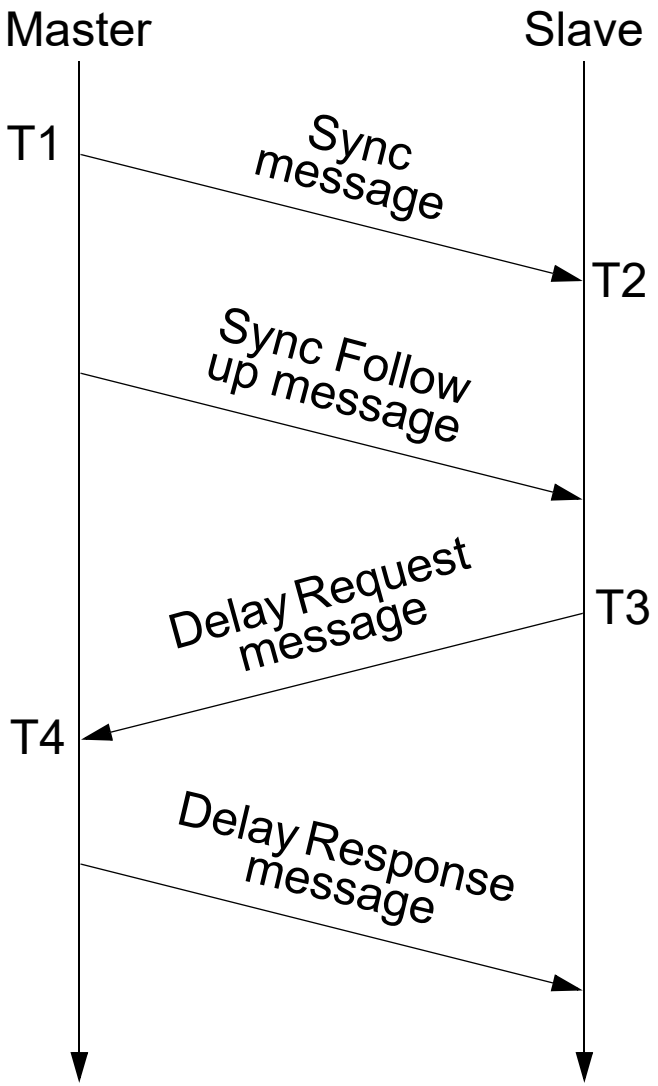


- ***Sync*: issued at T1, arrives at T2;**
- ***Sync Follow up*: carries the value of T1;**
- ***Delay Request*: issued at T3 arrives at T4;**
- ***Delay Response*: carries the value of T4.**

<u>Question</u>: Why does not the *Sync* message carry the value of T1?

<u>Answer</u>: 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 synchronise its clock.**

  Performed in two phases:

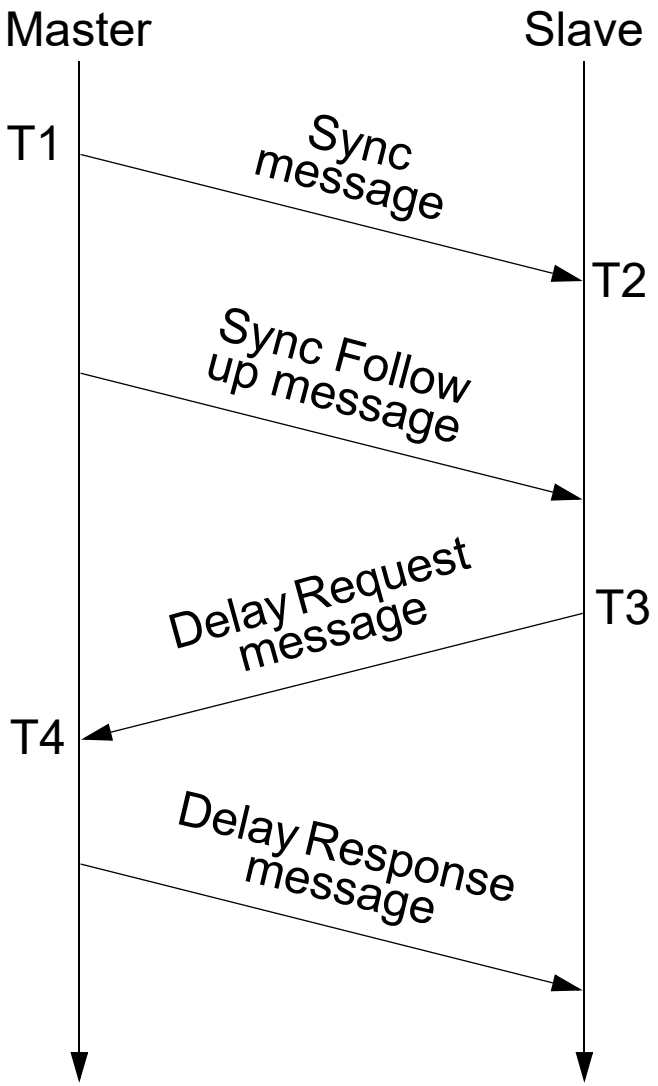
  - **<u>Phase 1</u>: offset calculation**

    Messages *Sync* and *Sync Follow*;

  - **<u>Phase 2</u>: delay measurement**

    Messages *Delay Request* and *Delay Response*;

The Precision Time Protocol



A naive approach:

Calculate the master→slave offset Θ_{MS}:

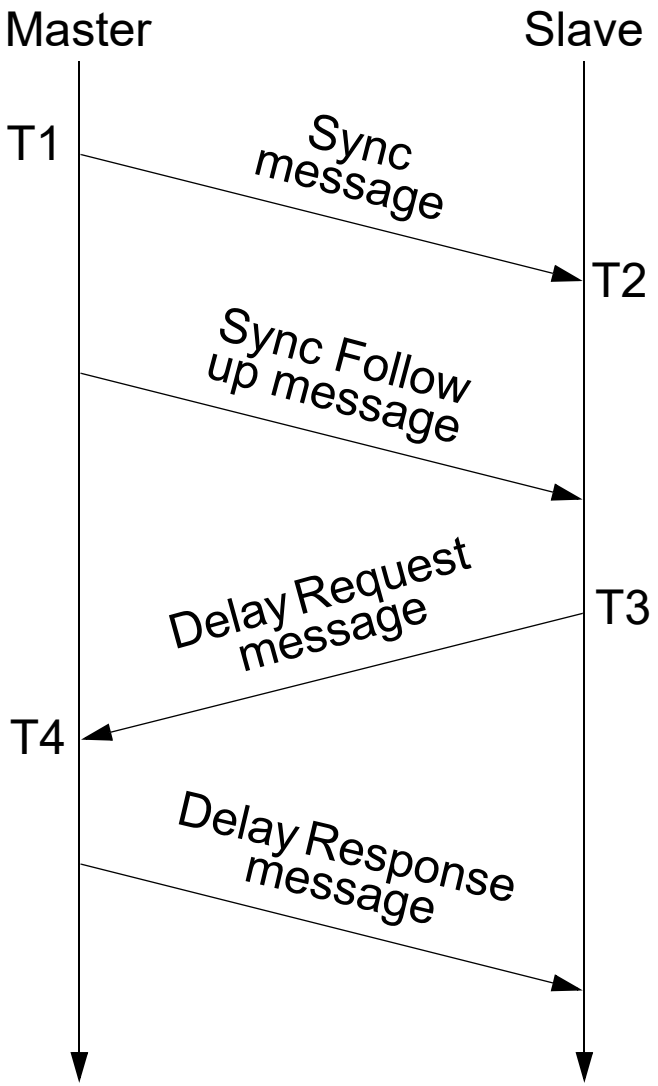
$$\Theta_{MS} = T2 - T1$$

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 synchronisation is accurate *only if this delay is zero* (which, of course, it is not)!**

The Precision Time Protocol



An accurate approach:

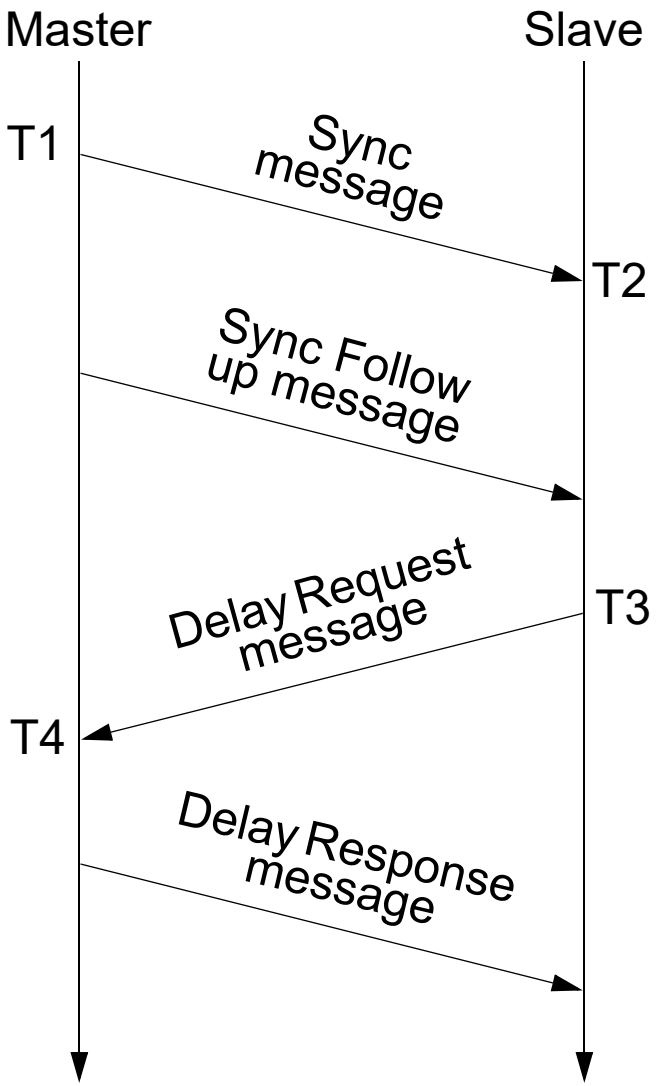
Estimate the communication delay:

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

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

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

The Precision Time Protocol



An accurate approach:

Estimate the communication delay:

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

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

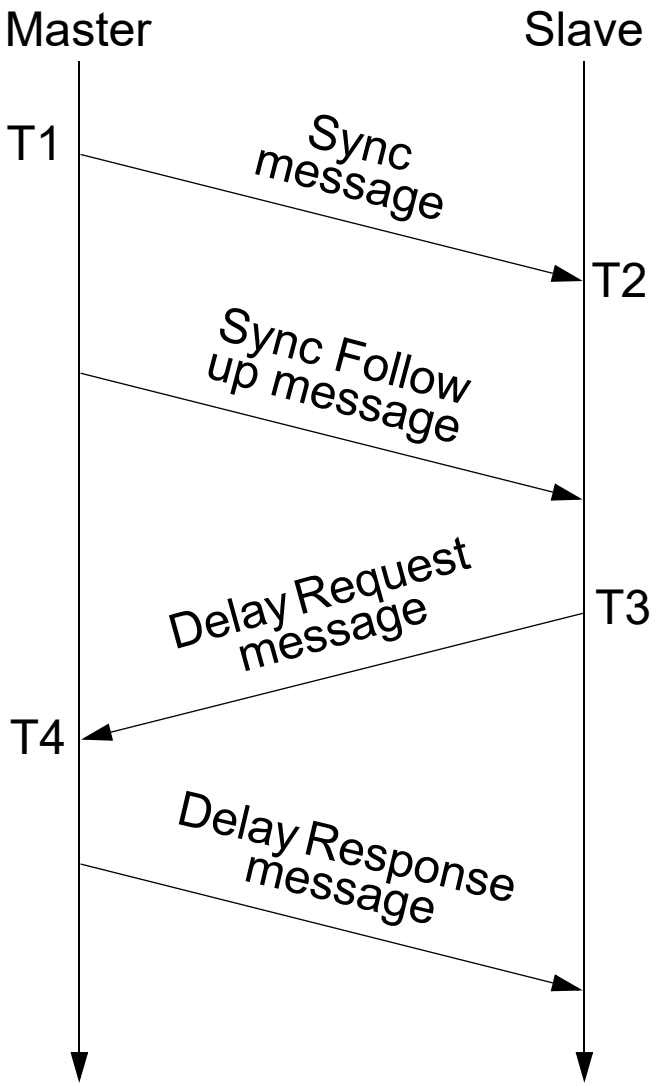
$$\delta = (T2 - T1 + T4 - T3)/2$$

Update the slave clock:

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

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

The Precision Time Protocol



<u>An accurate approach</u>:

Estimate the communication delay:

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

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

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

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.**

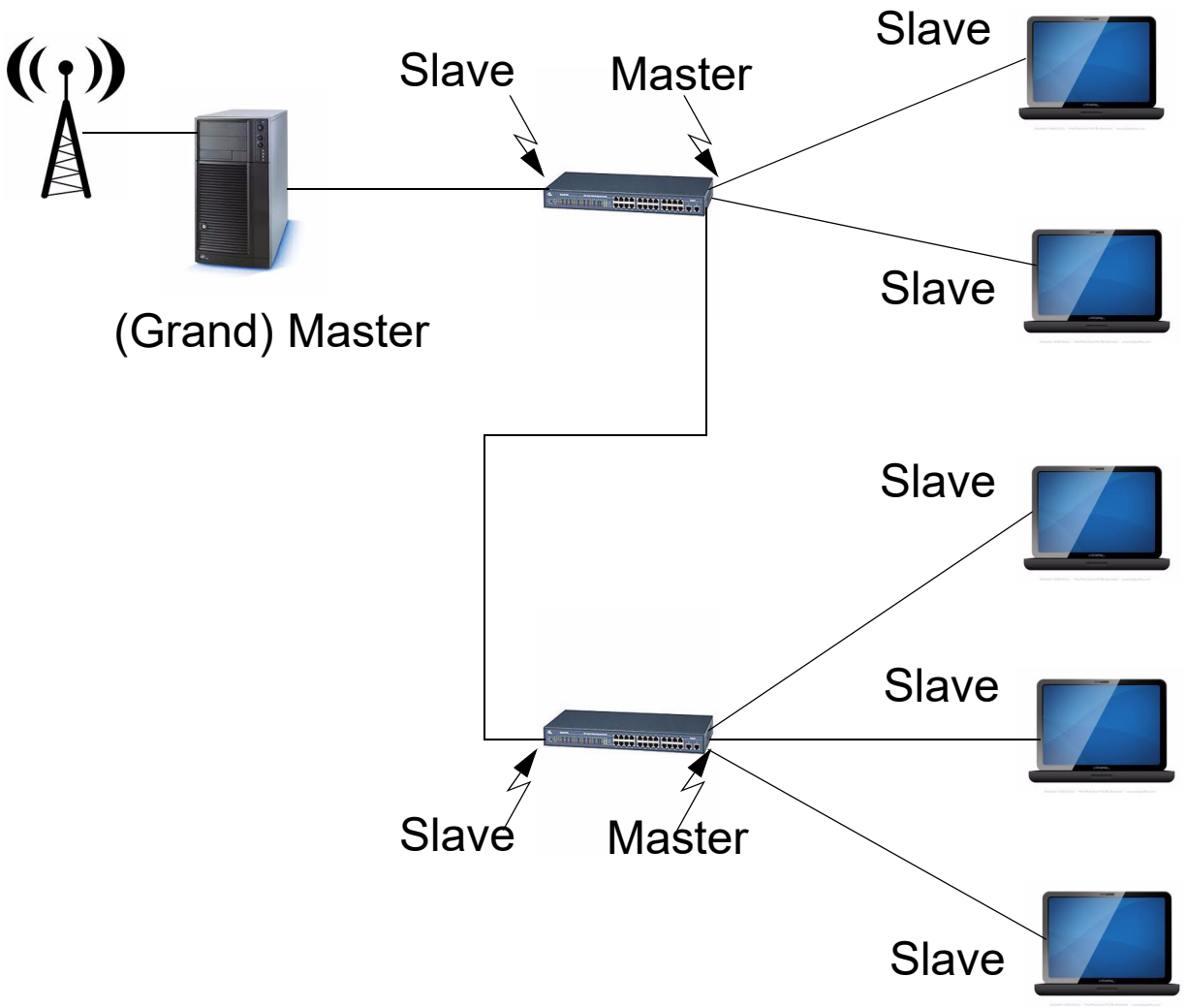
The Precision Time Protocol

- The calculation of δ assumes that the communication delay was identical on the way master to slave and slave to 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 synchronised to a connected master; they themselves behave as masters on all other ports and are used for synchronisation by connected slaves.
- Using BC switches, all synchronisations are over point to point connections.
- A Best Master Algorithm (BMA) 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.

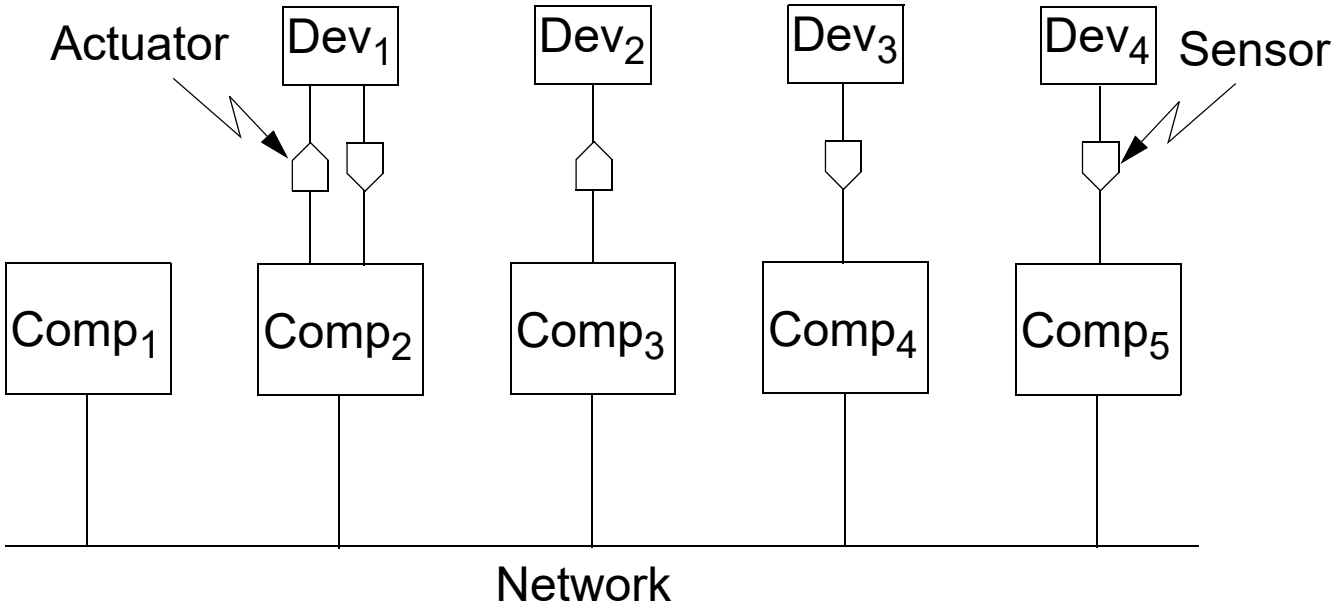
- Software implementation - a PTP software daemon running on non-standard hardware: Synchronisation in the range 10 -100 microseconds is achievable.

- Hardware implementation - hardware timestamping at master and slave plus PTP enabled network switches (with boundary clock): synchronisation in the range 10 - 100 nanoseconds is achievable.

The Network Time Protocol (NTP)

- The NTP has been adopted as a de facto standard for clock synchronisation in general purpose UNIX, Windows, etc. workstations and servers.
  - Connection over global Internet by standard routers and gateways is assumed (no specialised hardware).
  - There are no particular hardware requirements and customised 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 (as speech, video); clock update intervals can be in the range of minutes (even hours).
  - The actual master-slave synchronisation algorithm is, in broad terms, similar to that used in PTP.

Real-Time Communication



Data flows

- from sensors and control panels to processors
- between processors
- from processors to actuators and displays

- In order to achieve predictability: hard real-time systems need communication protocols that allow for the communication overhead to be bounded.

Time/Event Triggered Communication

- **Time-triggered communication**

  The sender and receiver agree on a cyclic, time-controlled, conflict-free communication schedule.

  ↓

  Each message transmission is started at a certain, pre-defined, moment in time; conflicts are avoided per definition.

  Examples:

TDMA	**Predictable; appropriate for real-time**
FlexRay (static phase)	

Time/Event Triggered Communication

- **Event-triggered communication**

  Messages can be sent whenever a significant event happened at the sender (task terminated, interrupt, etc.).

  ↓

  No pre-defined moments in time for communication; potential conflicts for bus access.

  Examples:

  Ethernet - is not predictable;

  CAN
  Token ring
  FlexRay (dynamic phase)
  } **Predictable; appropriate for real-time**

Ethernet Protocol

- **Ethernet is a Carrier Sense Multiple Access/Collision Detection (CSMA/CD) protocol.**
  - **On Ethernet, any device can try to send a frame at any time. Each device senses whether the line is idle and therefore available to be used. If it is, the device begins to transmit.**
  - **If two or more devices try to send at the same time, a collision occurs and the frames are discarded. *Each device then waits a random amount of time and retries until successful in getting its transmission sent*.**

Ethernet is inherently stochastic. It cannot provide a known upper bound on transmission time.

<u>Ethernet is not suitable for real-time applications</u>.

Ethernet Protocol

- **Ethernet is a Carrier Sense Multiple Access/Collision Detection (CSMA/CD) protocol.**
  - **On Ethernet, any device can try to send a frame at any time. Each device senses whether the line is idle and therefore available to be used. If it is, the device begins to transmit.**
  - **If two or more devices try to send at the same time, a collision occurs and the frames are discarded. *Each device then waits a random amount of time and retries until successful in getting its transmission sent*.**

    ↓

    Ethernet is inherently stochastic. It cannot provide a known upper bound on transmission time.

    <u>Ethernet is not suitable for real-time applications</u>.

- **The above is true for the original "vintage" Ethernet.**
  New, Ethernet based solutions have been proposed (e.g. full-duplex switched Ethernet, that avoids collision) which provide support for predictability and could be applied for real-time communication.

Protocols for Real-Time Communication

- CAN protocol
- Token Ring
- TDMA protocol
- FlexRay protocol

- TDMA is mostly suitable for applications with regular data flow (constant rate). It is the most reliable and predictable.
- The CAN protocol provides a higher degree of flexibility in the case of irregular flow.
- FlexRay is a heterogeneous time&event triggered protocol; potentially combines advantages of TDMA and CAN

CAN Protocol

CAN = Control Area Network

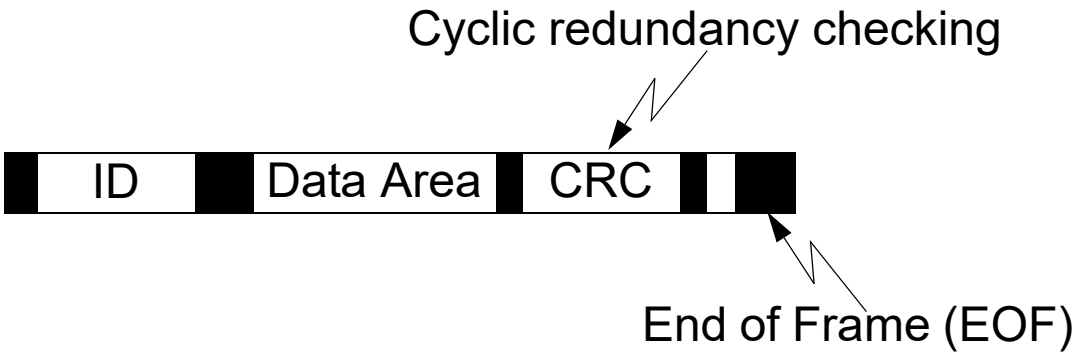
- **CAN is a Carrier Sense Multiple Access/Collision <u>Avoidance</u> (CSMA/CA) protocol.**

  CAN is widely used in automotive applications.

  - **In the CAN protocol, collisions are avoided by arbitration based on *priorities* assigned to messages.**
  - **CAN communication is based on the transfer of packages of data called *frames***

CAN Protocol

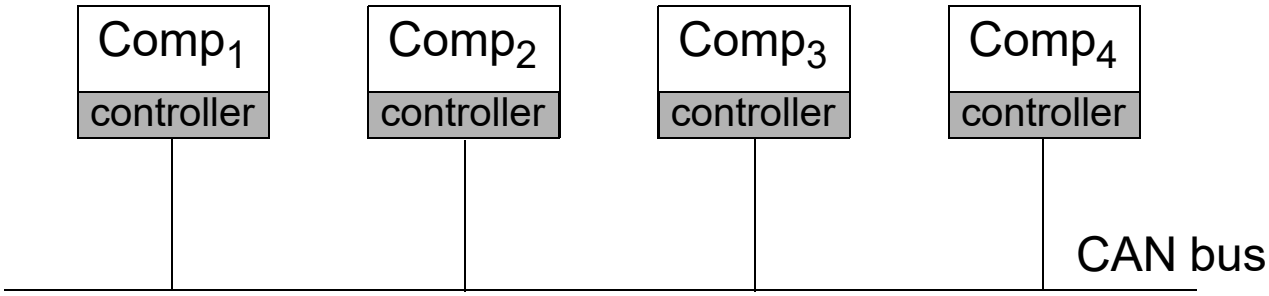
A CAN frame



- **The identifier (ID) field is used for two purposes:**

  1. **To distinguish between different frames.**
  2. **To assign relative priorities to the frames**

CAN Protocol



- **A CAN controller is attached to each processor in the system. It ensures that:**
  1. **The highest priority frame (smallest identifier) waiting to be transmitted from the respective processor is entering the arbitration for the bus.**
  2. **The arbitration procedure performed in cooperation by the controllers, guarantees access to the message with highest priority.**

  - **The arbitration is based on the existence of a *dominant* and a *recessive* bit: 0 is the dominant, 1 is the recessive:**
    - **In CAN, if several controllers transmit and at least one transmits 0, the bus is at 0; if all controllers write 1, the bus it at 1.**

CAN Protocol

Frame node_1:	00001100011	- - - - - - - - - - - - - - - - -	EOF
Frame node_2:	00100010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_3:	00001010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_4:	00001010111	- - - - - - - - - - - - - - - - -	EOF

- **During arbitration, controllers write the frame ID to the bus, bit by bit, *as long as they read from the bus the same value they have written*; once a controller reads different, it continues by writing 1s until it reads EOF from the bus.**
- **After the EOF, nodes which were unsuccessful retry with the same frames.**

CAN Protocol

Frame node_1:	00001100011	- - - - - - - - - - - - - - - - -	EOF
Frame node_2:	00100010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_3:	00001010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_4:	00001010111	- - - - - - - - - - - - - - - - -	EOF

node 1	0	
node 2	0	
node 3	0	
node 4	0	
CAN bus	0	

CAN Protocol

Frame node_1:	00001100011	- - - - - - - - - - - - - - - - -	EOF
Frame node_2:	00100010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_3:	00001010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_4:	00001010111	- - - - - - - - - - - - - - - - -	EOF

node 1	0	0	
node 2	0	0	
node 3	0	0	
node 4	0	0	
CAN bus	0	0	

CAN Protocol

Frame node_1:	00001100011	- - - - - - - - - - - - - - - - -	EOF
Frame node_2:	00100010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_3:	00001010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_4:	00001010111	- - - - - - - - - - - - - - - - -	EOF

node 1	0	0	0	
node 2	0	0	1	
node 3	0	0	0	
node 4	0	0	0	
CAN bus	0	0	0	

CAN Protocol

Frame node_1:	00001100011	- - - - - - - - - - - - - - - - -	EOF
Frame node_2:	00100010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_3:	00001010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_4:	00001010111	- - - - - - - - - - - - - - - - -	EOF

node 1	0	0	0	0	
node 2	0	0	1	1	
node 3	0	0	0	0	
node 4	0	0	0	0	
CAN bus	0	0	0	0	

CAN Protocol

Frame node_1:	00001100011	- - - - - - - - - - - - - - - - -	EOF
Frame node_2:	00100010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_3:	00001010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_4:	00001010111	- - - - - - - - - - - - - - - - -	EOF

node 1	0	0	0	0	1	
node 2	0	0	1	1	1	
node 3	0	0	0	0	1	
node 4	0	0	0	0	1	
CAN bus	0	0	0	0	1	

CAN Protocol

Frame node_1:	00001100011	- - - - - - - - - - - - - - - - -	EOF
Frame node_2:	00100010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_3:	00001010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_4:	00001010111	- - - - - - - - - - - - - - - - -	EOF

node 1	0	0	0	0	1	1
node 2	0	0	1	1	1	1
node 3	0	0	0	0	1	0
node 4	0	0	0	0	1	0
CAN bus	0	0	0	0	1	0

CAN Protocol

Frame node_1:	00001100011	- - - - - - - - - - - - - - - - -	EOF
Frame node_2:	00100010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_3:	00001010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_4:	00001010111	- - - - - - - - - - - - - - - - -	EOF

node 1	0	0	0	0	1	1	1
node 2	0	0	1	1	1	1	1
node 3	0	0	0	0	1	0	1
node 4	0	0	0	0	1	0	1
CAN bus	0	0	0	0	1	0	1

CAN Protocol

Frame node_1:	00001100011	- - - - - - - - - - - - - - - - -	EOF
Frame node_2:	00100010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_3:	00001010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_4:	00001010111	- - - - - - - - - - - - - - - - -	EOF

node 1	0	0	0	0	1	1	1	1
node 2	0	0	1	1	1	1	1	1
node 3	0	0	0	0	1	0	1	0
node 4	0	0	0	0	1	0	1	0
CAN bus	0	0	0	0	1	0	1	0

CAN Protocol

Frame node_1:	00001100011	- - - - - - - - - - - - - - - - -	EOF
Frame node_2:	00100010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_3:	00001010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_4:	00001010111	- - - - - - - - - - - - - - - - -	EOF

node 1	0	0	0	0	1	1	1	1	1
node 2	0	0	1	1	1	1	1	1	1
node 3	0	0	0	0	1	0	1	0	1
node 4	0	0	0	0	1	0	1	0	1
CAN bus	0	0	0	0	1	0	1	0	1

CAN Protocol

Frame node_1:	00001100011	- - - - - - - - - - - - - - - - -	EOF
Frame node_2:	00100010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_3:	00001010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_4:	00001010111	- - - - - - - - - - - - - - - - -	EOF

node 1	0	0	0	0	1	1	1	1	1	1
node 2	0	0	1	1	1	1	1	1	1	1
node 3	0	0	0	0	1	0	1	0	1	0
node 4	0	0	0	0	1	0	1	0	1	1
CAN bus	0	0	0	0	1	0	1	0	1	0

CAN Protocol

Frame node_1:	00001100011	- - - - - - - - - - - - - - - - -	EOF
Frame node_2:	00100010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_3:	00001010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_4:	00001010111	- - - - - - - - - - - - - - - - -	EOF

node 1	0	0	0	0	1	1	1	1	1	1	1
node 2	0	0	1	1	1	1	1	1	1	1	1
node 3	0	0	0	0	1	0	1	0	1	0	1
node 4	0	0	0	0	1	0	1	0	1	1	1
CAN bus	0	0	0	0	1	0	1	0	1	0	1

ID based arbitration:
Frame node_3 has priority

CAN Protocol

Frame node_1:	00001100011	- - - - - - - - - - - - - - - - -	EOF
Frame node_2:	00100010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_3:	00001010101	- - - - - - - - - - - - - - - - -	EOF
Frame node_4:	00001010111	- - - - - - - - - - - - - - - - -	EOF

node 1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	▪	▪	▪	▪	1 1
node 2	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	▪	▪	▪	▪	1 1
node 3	0	0	0	0	1	0	1	0	1	0	1	0	1	0	0	1	1	0	▪	▪	▪		EOF
node 4	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	▪	▪	▪	▪	1 1
CAN bus	0	0	0	0	1	0	1	0	1	0	1	0	1	0	0	1	1	0	▪	▪	▪		EOF

ID based arbitration:
Frame node_3 has priority

CAN Protocol

- If the following assumptions are fulfilled, message communication times can be bounded using techniques similar to those developed for priority based process scheduling:
  - messages are generated periodically, and the period is known.
  - The maximum size of each frame is known.
  - The software overhead connected to handling of messages is known.

Token Ring

- Nodes are logically organized in a ring, on which the right to communicate is continuously passed.

- With a token ring protocol maximum bounds on message delay can be established. The following are the essential parameters:

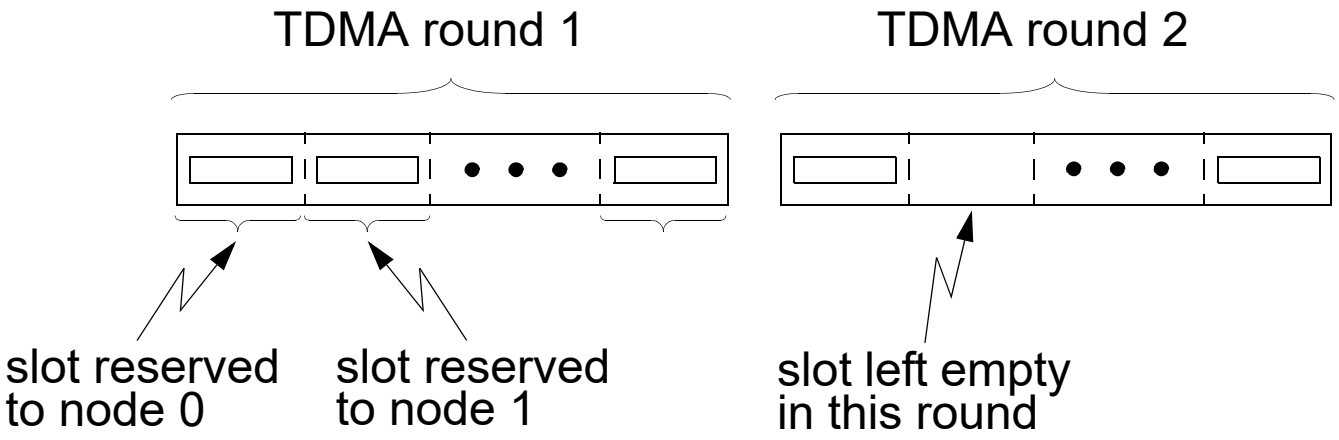
  - <u>The hold time</u>: the longest time a node needs for communicating one message.
    This can be derived from communication speed on bus and the maximum bound on the message length.
  - <u>The full rotation time</u>: the longest time needed for a full rotation over all nodes.
    This can be derived as $k*T_h$, where k is the number of nodes, and T_h is the hold time.

- Fault tolerance can be a problem: if one node fails, the traffic is disrupted.

TDMA Protocol

TDMA = Time Division Multiple Access

- **The total channel (bus) capacity is statically divided into a number of slots. Each slot is assigned to a certain node.**
- **With a system of *N* nodes, the sequence of *N* slots is called a *TDMA round*. One node can send one frame in a TDMA round. The frame is placed into the slot assigned to that processor.**
- **If no frame is to be sent by a node, the slot will stay empty in that round.**
- **The duration of one TDMA round is the TDMA period.**



TDMA Protocol

- TDMA practically means a static partitioning of access time to the bus. Each node knows in advance when and for how long it is allowed to access the communication line.
  TDMA implies the availability of a global physical time base (clock synchronisation).

- Collisions are avoided as nodes know when they have guaranteed exclusive access to the line.

- Message passing delay is bounded.

TDMA Protocol

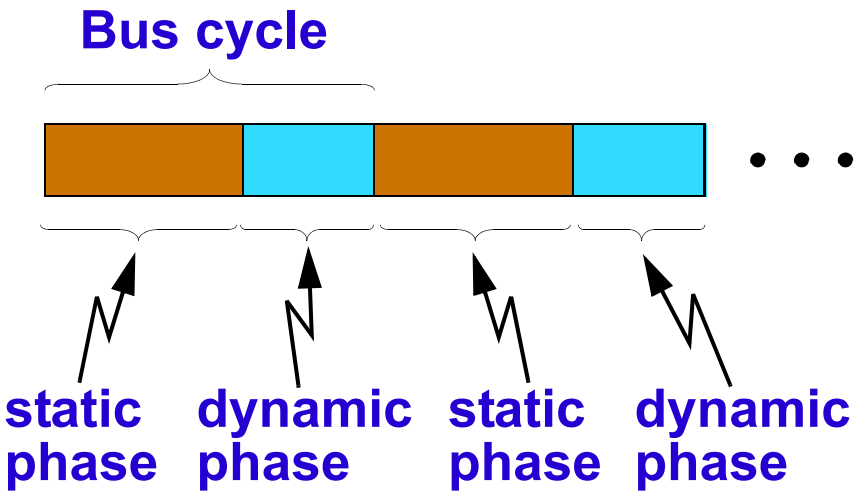
Advantages:

- High degree of predictability
- Well suited to safety critical applications.

Disadvantages:

- Can lead to poor utilisation of bus bandwidth (e.g. empty slots).
- Low degree of flexibility $\Rightarrow$ problems with irregular flows.

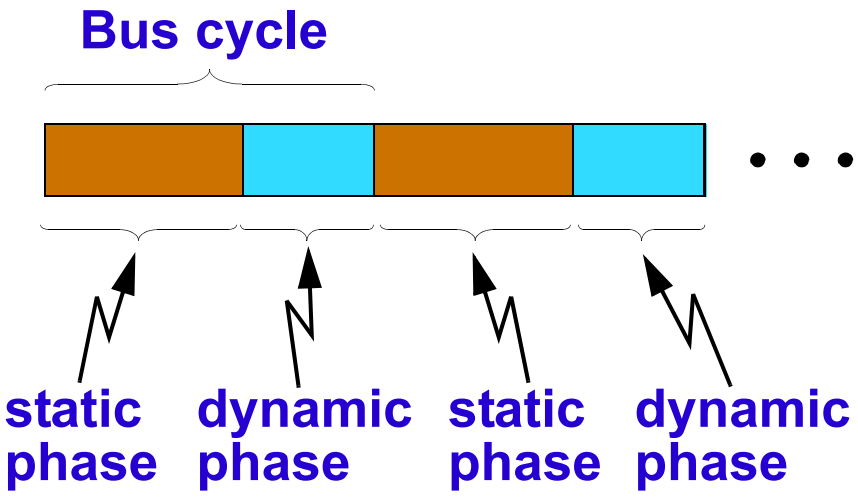
FlexRay
A Heterogeneous Communication Protocol



- FlexRay combines two protocols: an event triggered and a time triggered. It combines some of the advantages of the two approaches.

- FlexRay has been designed by the "FlexRay consortium" for automotive applications.

FlexRay
A Heterogeneous Communication Protocol



- The FlexRay bus cycle is divided into two phases (the length of each phase is fixed by the designer):
  - Static phase

    During the static phase the bus works according to a TDMA policy (the static phase consists of slots assigned to nodes).
  - Dynamic phase

    During the dynamic phase the bus works according to an event triggered protocol, somewhat similar to CAN.
- Combines two predictable approaches: FlexRay is suitable for real-time.