DISTRIBUTED REAL-TIME SYSTEMS

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*.

Examples:

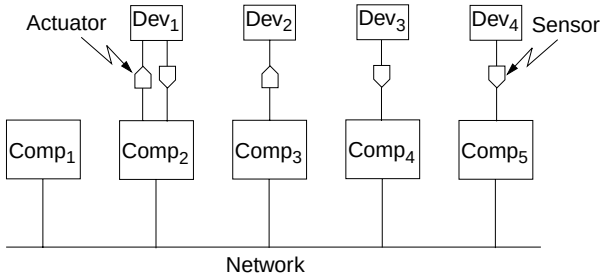
- Process control systems
- Computer-integrated manufacturing systems
- Aerospace and avionics systems
- Automotive electronics
- Medical equipment
- Nuclear power plant control
- Defence systems
- Consumer electronics
- Multimedia
- Telecommunications

Distributed Real-Time Systems

Real-time systems very often are implemented as distributed systems. Some reasons:

- Fault tolerance
- Certain processing of data has to be performed at the location of the sensors and actuators.
- Performance issues.

Distributed Real-Time Systems (cont'd)



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).
- The 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.
- Very often, such systems are dedicated to a specific functionality.
  They perform a certain specific job which is fixed.
  This is opposed to standard, general purpose computing, where several different programs are run on a computer to provide different services.
- 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 deadlines: processes can be finished after its deadline, although the value provided by completion may degrade with time.
- Firm deadlines: 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 concerning predictability:

- Determine worst case execution times for each process.
- Determine worst case communication delays on the interconnection 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?

☞ 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

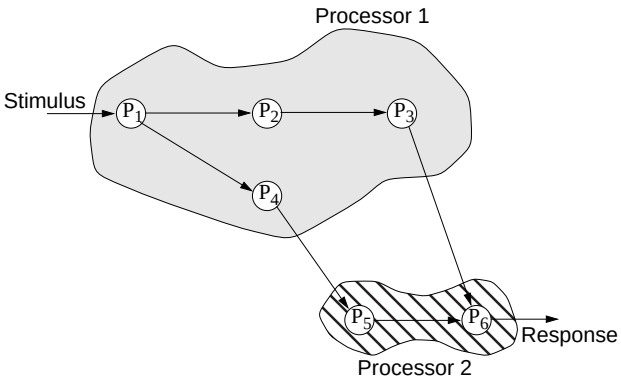
Are processes preempted?
- preemptive scheduling
- non-preemptive scheduling

Scheduling Policies (cont'd)

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



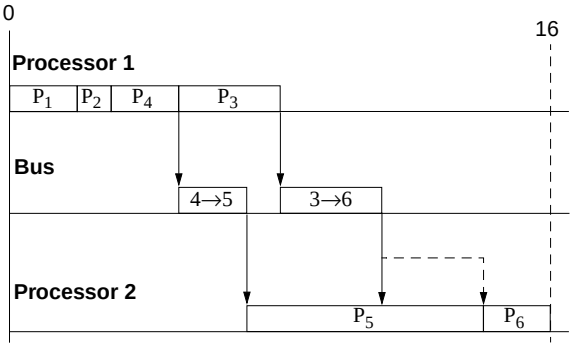
Deadline on *response*: 15

Worst case execution times:

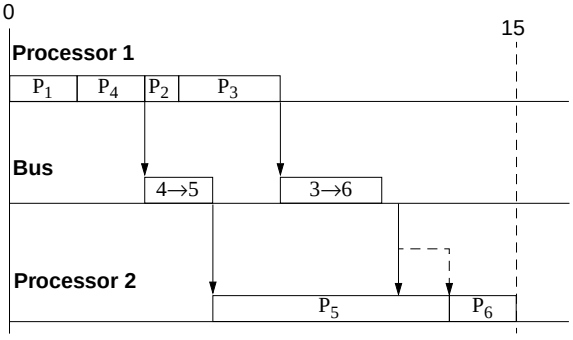
P_1	2
P_2	1
P_3	3
P_4	2
P_5	7
P_6	2
$4 \rightarrow 5$	2
$3 \rightarrow 6$	3

Static Scheduling (cont'd)

A first alternative



A second alternative



Static Scheduling (cont'd)

With the second alternative, the deadline is fulfilled.

Schedule table corresponding to second alternative:

Processor 1

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

Processor 2

6	read from bus message 4→5
6	P_5
11	read from bus message 3→6
13	P_6

We assume that the processors are able to process I/O and programs in parallel (there is an I/O coprocessor).

Static Scheduling (cont'd)

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 (cont'd)

- Processes are activated as response to events (e.g. 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 (cont'd)

☞ With certain restrictions in the process model, schedulability analyses can be performed:

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.
- All processes are independent (no resources shared and no precedence).
- 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 the following is true:

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

Dynamic Scheduling (cont'd)

As result of research in the area of real-time systems, schedulability tests have been developed for situations in which some of the restrictions above are relaxed:

- Several processors.
- Precedence constraints and communications between processes.
- Processes are sharing resources.
- Deadlines can be different from period.

☞ All schedulability tests require worst case execution times for processes and communications to be known!

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 certain 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 (cont'd)

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 processors of the distributed system show different time.

"Universal" Time

- The standard for measurement of time intervals: *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.

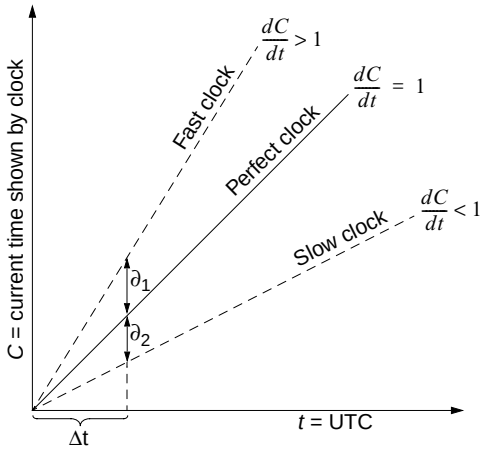
- One processor in the system (possibly several) is equipped with UTC receivers (time providers).
- By external synchronization the system is kept synchronous with the "real time". This allows to exchange consistently timing information with other systems and with users.

Internal Synchronization

Synchronization among processors of the system.

- It is 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 (cont'd)

In an ideal case, the clock shows UTC: $C = t$.

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

In reality: $\frac{dC}{dt} = 1 \pm \rho$

ρ is the maximum drift rate, and should be specified by the manufacturer.

Two processors with similar clocks could be apart with:

$S = 2\rho\Delta t$

If we have to guarantee a skew less than S_{max}, the clocks have to be synchronized at an interval:

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

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. The goal is to keep the clocks of all other nodes synchronized with the time server. This is 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.

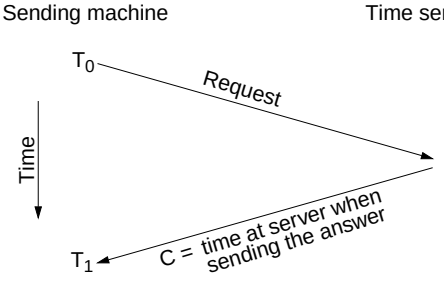
☞ Distributed Algorithms

- 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 is needed). Only internal synchronization is performed.
  - Several processors (possibly all) have an UTC receiver. However, 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 be in possession of the correct time (has an UTC receiver).

- Periodically (with period less than $S_{max}/2\rho$) each machine 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 sending machine when sending the request and receiving the answer, respectively.

Cristian's Algorithm (cont'd)

As a first possible approximation, the receiver could set its clock to:

$T_{rec} = C$

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

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

How large is T_{trans}? Possible estimation:

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

Problem:

The time to receive the answer can be very different for the one needed to transmit the request (because of congestion on network, for example).

Can we, at least, determine the accuracy of the time estimation?

Cristian's Algorithm (cont'd)

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

The exact value for T_{rec} is between:

$T_{rec}^{min} = C + t_{min}$, when the answer has been transmitted in minimum time,

and

$T_{rec}^{max} = C + (T_1 - T_0) - t_{min}$, when the request has been transmitted in minimum time.

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

⇩

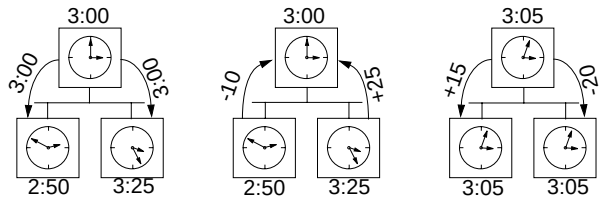
The time is set with an absolute accuracy of:

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

- In order to improve accuracy, 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 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 (cont'd)

☞ The situation is more complicated than the figure on the previous slide shows:

- 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 (see slide 35).
- 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.

- If the master fails, another one has to be elected.

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.

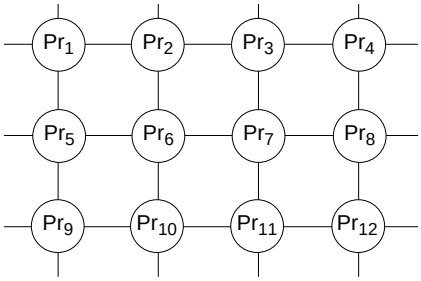
- The typical algorithm used at point 2 preforms the following:
  - The correction value for the local clock is based on an average of the received clock values.
  - When receiving time values from the other nodes, corrections are performed taking into consideration estimated delays due to message passing.
  - Only clock values are considered that do not differ from one another by more than a certain amount.

Distributed Clock Synchronization (cont'd)

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 structured 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}.

- If $T_{new} > T_{curr}$, the solution is simple. Advance the local clock to the new value T_{new}.
- If $T_{new} < T_{curr}$, we are not allowed to simply 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. This is performed by the software which handles the clock tick (see also slide 21).

Adjusting Drifted Clocks (cont'd)

At each clock tick, and 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 running on the processor, 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, $a = b = 0$.

- 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 (cont'd)

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 has to be performed "smoothly" over a period of N clock ticks.

Now 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: $T_{new} + N\nu$.
- after N ticks, 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) above we get:

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

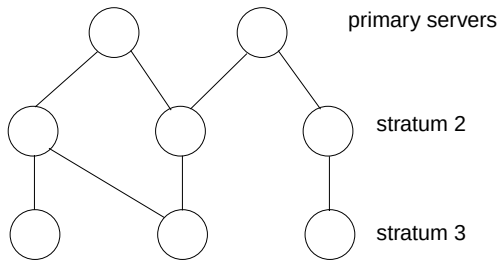
Adjusting Drifted Clocks (cont'd)

- The above 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 directly by updating the clock.
- If the adjustment is backward the clock has to be changed smoothly, like shown above.

The Network Time Protocol

- The NTP has been adapted as a standard for clock synchronization through Internet.
- NTP allows for the implementation of a network architecture in which primary and secondary time servers are defined.
  - Primary time servers are directly connected to a time reference source (e.g. UTC receiver).
  - Secondary time servers synchronize themselves (possibly via other secondary servers) to primary servers over the network.

Based on the accuracy of each time server, a structure of servers is defined, with primary servers at the top, and secondary servers on the levels (*strata*) below.



The Network Time Protocol (cont'd)

- Primary servers have the highest accuracy of their clocks. As the stratus level increases, the accuracy degrades.
- A lower accuracy of a certain server can be caused by the network paths towards the higher accuracy stations, and/or by the stability of its local clock.
- There are several operating modes allowed by NTP, similar to centralized algorithms with active or passive server, and to distributed algorithms.

The Network Time Protocol (cont'd)

One of the principal goals of NTP is to achieve *robustness*. This implies resistance in the case of:

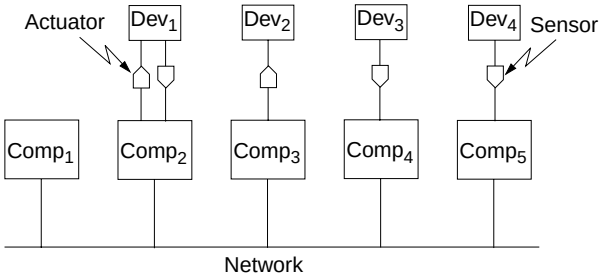
- Faulty clocks.
- Damaged network paths.
- Broken servers
- Malicious intruders (protection, for example, by authentification).

In order to maintain acceptable accuracy (even in the case of faults):

1. Filtering algorithm:
   By this, the estimation of the offset of the local clock, relative to a certain other server (called peer) is improved. Repeated synchronization attempts are performed and the low quality estimates are dropped out.

2. Selection algorithm:
   Periodic attempts to find, for a given server, the most reliable servers (peers) to be used as clock source for synchronization.

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 the communication overhead to be bounded.

Real-Time Communication (cont'd)

Traffic in real-time systems:

- Constant rate:

  Fixed-size packets are generated at periodic intervals. This is typical for many control applications, where the sensors generate a regular traffic.

- Variable rate:

  Can be produced by variable sized packets and/or irregular intervals of generation. Voice and video traffic usually exhibits variable rate.

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 have tried to send at the same time, a collision is said to occur 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.

Ethernet is not suitable for real-time applications.

Protocols for Real-Time Communication

- CAN protocol
- Token Ring
- TDMA 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.

CAN Protocol

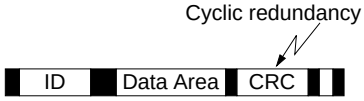
CAN = Control Area Network

☞ CAN is a Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA) protocol.

CAN is widely used in automotive applications, for example in the *Volvo S80*.

- 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*.

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 (cont'd)

A CAN controller is attached to each processor in the system. It ensures that:

- The highest priority frame waiting to be transmitted from the respective processor is entering the arbitration for the bus.
- The arbitration procedure performed in cooperation by the controllers, guarantees access to the message with highest priority.

If the following assumptions are fulfilled, message communication times can be bounded using techniques similar to those developed for priority based process scheduling (see slide 16):

- A given message is generated periodically, and a worst case (minimal) period is known.
- The maximum size of each frame is known.
- The software overhead connected to handling of messages is known.
- The maximum overhead due to errors and re-transmission is estimated.

Token Ring

- The right to transmit is contained in a special control message, the token. Whoever has the token, is allowed to transmit.
- Processors are logically organized in a ring, on which the token is continuously passed.

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

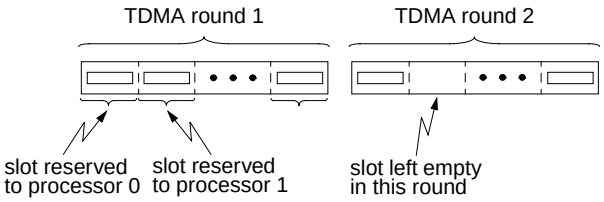
- The token hold time: the longest time a node may hold the token.
  This can be derived from communication speed on bus and the maximum bound on the message length.
- The token rotation time: the longest time needed for a full rotation of the token.
  This can be derived as $k*T_h$, where k is the number of processors, and T_h is the token 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 (processor).
- With a system of *N* processors, the sequence of *N* slots is called a *TDMA round*. One processor 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 processor, an empty slot will be sent in that round.
- The duration of one TDMA round is the TDMA period.



TDMA Protocol (cont'd)

- TDMA practically means a static partitioning of access time to the bus. Each processor knows in advance when and for how long it is allowed to access the bus.
- Collisions are avoided as processors know when they have guaranteed exclusive access to the bus.
- Message passing delay is bounded: a message is split into a certain number of frames which are transmitted in successive slots (one per TDMA round).
- Not all slots have to be of identical length. The slot length corresponding to a given node is however identical for all rounds. This length is determined by the designer, depending on the particularities of the processes running on each node (considering, for example, the length and number of messages generated)

TDMA Protocol (cont'd)

Advantages:

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

Disadvantages:

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

Summary

- 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.
- Many real-time applications are implemented as distributed systems.
- Deadlines in distributed systems can be soft and hard. Hard deadlines have to be met strictly.
- Predictability is the most important property of a real-time system.
- By scheduling it has to be determined which process or communication to be executed at a certain moment.
- With static scheduling, activation times are determined a-priori.
  Advantage: high predictability.
  Problem: no flexibility.
- With dynamic scheduling it is more difficult to guarantee that time constraints are fulfilled.
  Under certain restrictions, however, this is possible.
- Specific issues with distributed embedded systems: clock synchronization and real-time communication.
- External clock synchronization: synchronization with time sources external to the system.
  Internal clock synchronization: synchronization among processors of the system.

Summary (cont'd)

- Centralized algorithms for clock synchronization are based on the existence of a time server node. Such are: Cristian's algorithm and the Berkeley algorithm.
- Disitributed algorithms for clock synchronization don't need the presence of a particular server node.
- The Network Time Protocol is used as a standard for time synchronization in the Internet.

- The main problem with communication in distributed real-time systems is predictability.
- The Ethernet is an example of protocol which cannot be used with real-time applications because no bounds can be derived on communication time.
- The CAN protocol is used in many automotive applications. Predictability is provided by a priority based handling of messages.
- The token ring protocol provides an upper bound on communication time. Efficiency is low for many applications. Fault tolerance can be a problem (if one node fails, the traffic is disrupted).
- TDMA is based on a division of the channel capacity into time slots which are assigned to the particular processors.
  TDMA provides high predictability and the potential for safety critical applications.