Embedded Information Systems

(Lecture notes covering lectures 2-17)

Lecturer: Gábor Péceli

2017.

1. Introduction (cont.)

Real-time quantities, variables, images

Real-time variables (*RT* entities/quantities): state variables of relevance for the given purpose, like the flow of a liquid in a pipe, the set-point of a control loop. It has static attributes that do not change during the lifetime of the *RT* variable, like the name, the type, the value domain, the maximum rate of change, and has dynamic attributes that change with time, like the value set at a point of time. Every *R*T variable is in the sphere of control (SOC) of a subsystem that has the authority to set the value of the *RT* variable. Outside its SOC the *RT* variable can only be observed, but not modified. For this very reason, it is a must to measure the proper operation of an actuator. An *RT* variable can have a discrete value set (discrete *RT* variable) or a continuous value set (continuous *RT* variable). A discrete *RT* variable can be undefined. *Example:* a garage door while it is opening is neither open, nor closed. In contrast, the set of values of a continuous *RT* variable is always defined.

Observations: the information concerning an *RT* variable at a given point in time.

Observation =<name, observation time, value>

Observations in distributed systems: if the global time is not available, then the usability of timestamps is limited, therefore the observation time is often replaced by the arrival time. This can cause considerable error in state estimation.
Indirect observations: In some situations, the direct observation of an *RT* variable is not possible. As an example, consider the temperature measurement within a slab of steel. The internal temperature measurement is replaced by measurement on the surface together with the mathematical modelling of the heat transfer.
Such measurements are called indirect observations.
State observations: Every observation is self-contained because it carries an absolute value. In many cases equidistant sampling is applied, i.e. periodic time-triggered readings.
Event observations: an event is a state-change at a point of time. Since an observation is also an event, therefore it is not possible to observe and event in the controlled object directly. *Problems:* If the observation is time-triggered, the time of event occurrence is the rising edge of the interrupt signal. Any delayed response to this interrupt signal will cause an error in the timestamp of the event observation. If the event observation is time triggered, then the time of the event occurrence can be at any point within the sampling interval. Since the value of an event observation contains the difference between the old and the new state, the loss or duplication of a single event observation may cause problems. An event observation is sent if the *RT* variable changes its value. The latency of the detection of a failure at the observer node cannot be bounded, because if no new message arrives, the receiver assumes that the *RT* variables did not change.

RT images: it is the picture of an *RT* variable within the computer program, if it is accurate copy of the *RT* variable both in the value and the time domains. We define the notion of temporal accuracy. While an observation records a fact that remains valid forever, since it was observed at a given point of time, the validity of an *RT* image is time-dependent, and will be invalid with time. An *RT* image can be an up-to-date state observation, or an up-to-date event observation, or a state estimation.

RT objects: An *RT* object is analogous to a container within a node of a distributed computer system holding an *RT* image or an *RT* variable. A given granularity *RT* clock is associated with every *RT* object. Whenever this clock ticks, an object procedure is activated. If this is periodic, we are talking about a synchronous *RT* object. If within a distributed system the copies of an *RT* object provide specific local services, we are talking about distributed *RT* objects. A good example is the global clock, because every node has a local copy, which operates with a prescribed precision.

Temporal accuracy: The time of the observation and the time of the use of this information differ, this latter is delayed. This delay is caused by the time needed to create a message containing the information, the time

of sending this message to the receiver node, and the time to decode the message. If during this delay the information sent loses its validity, then the observation cannot be utilised directly. We define temporal accuracy by an interval of duration $a_{ccuracy}$, during which the error in value is still tolerable. As an example let us investigate the required temporal accuracy intervals of the *RT* images that are used in a controller of an automobile engine.

RT image within the computer	max. change	accuracy	temporal accuracy
Piston position	6000 rpm	0.1°	3µsec
Accelerator pedal position	100%/sec	1%	10 msec
Engine load	50%/sec	1%	20 msec
Temperature of oil and water	10%/minute	1%	6 sec

There is a difference of more than six orders of magnitude in the temporal accuracy of these *RT* images. Obviously, the position of the piston within the cylinder requires the use of state estimation.
The time between of observation and utilisation in case of a variable $v(t)$ causes the following value error:

$$error(t) \cong \frac{dv(t)}{dt}[C(t_{use}) - C(t_{observation})],$$

$C(t)$ stands for the clock function of the corresponding node.

If we use image of specified temporal accuracy, the *worst-case* error:

$$error = \underbrace{max}_{\forall t}\left|\frac{dv(t)}{dt}\right| d_{accuracy}$$

If the design is properly balanced, this error should be in the range of measurement error of the magnitude. To get calculations of acceptable accuracy, we must meet the requirement:

$$[C(t_{use}) - C(t_{observation})] \leq d_{accuracy}.$$

Example: *Validity of an observation:* On September 14, 1993, a Lufthansa A320 overran the runway at Warsaw Airport, killing a crew member and a passenger, and injuring 54. The validity problem related to the fact, that the A320 control logic required the airplane to be settled on both main landing gears before the brakes, ground spoilers and thrust reversers can be activated. Due to the side-wind the airplane did not settle on its second main landing gear for nine seconds, and was travelling to fast: → The instrumentation logic was wrong.

A periodically updated *RT* image is called parametric or phase insensitive, if

$$d_{accuracy} > \left(d_{update} + WCET_{message\ forwarding}\right).$$

Here *message forwarding* includes the composition of the message at the sending node, the communication, and the message decoding at the receiving node. A parametric *RT* image can be used any time without considering the phase conditions, because the update always arrives in time.

A periodically updated RT image is called phase sensitive, if

$$WCET_{message\ forwarding} < d_{accuracy} < \left(d_{update} + WCET_{message\ forwarding}\right)$$

In this case it is not sure that the update will arrive within the interval of the temporal accuracy, therefore the time of update and use should be monitored.

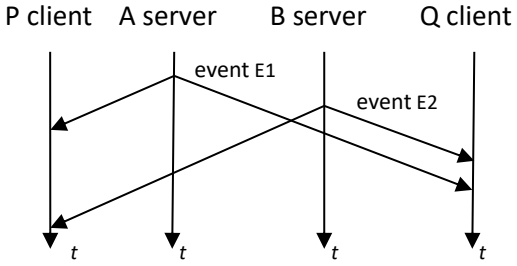
Example: Imagine that the information about the accelerator pedal needs $WCET_{message\ forwarding} = 4ms$. If $d_{update} < 6ms$, then the Rt image is parametric, if e.g. $d_{update} = 8ms$, then it is phase sensitive.

Phase sensitivity can be avoided by proper updating frequency, or by applying state estimation.

Important comment: The updating frequency here is completely different from the sampling frequency required when we address the reproducibility of continuous signals from its samples. It is a quite different problem! However, it might happen that both conditions are to be considered!

Examples of specific time relations in embedded systems:

- **relativistic effect:** the time conditions of the communication through channels may change the order of the event at the receiving node:



The figure above illustrates, that in the case of client Q the message about event E2 precedes the message about event E1, which occurred earlier. Such a situation might cause problems, if the decisions made at client Q depend on the order of the messages. If the events E1 and E2 are not independent, after the arrival of the message about E2 to server Q, it might be reasonable to propose to wait for all those messages which were sent possibly at the same time instant or earlier as the message about E2. This waiting time is called ***action delay***, which is the worst-case value of the possible message forwarding time for the case described above. The necessary action delay can be calculated if the minimum and the maximum of the message forwarding time is known, i.e. for the message forwarding time the following is valid:

$$d_{\min} \le d \le d_{\max}\,.$$

If the global time is known for the nodes, i.e. a timestamp can be attached to the massage, the client can calculate, based on the time of the arrival and the timestamp, what is the worst-case arrival time for all simultaneous or previous messages. In this case the arrival time will be the sum of the sending time and d_{max}, and therefore the action delay equals d_{max}.

If for the nodes considered the global time is not known or their local clocks are not synchronized, the use of timestamps might cause problems, therefore only the time of the arrival is to be used to determine the action delay. Since the actual message forwarding time cannot be measured, let us suppose the shortest forwarding time d_{min} before the arrival, then we must wait $d_{max} - d_{min}$ before starting the action. However, since it might happen, that the arrived message was travelling for the worst-case value d_{max}, then relative to the sending time till the action, there will be a waiting time of $2d_{max} - d_{min}$ i.e. the action delay is $2d_{max} - d_{min}$.

Comments:

1. The second case might be far more disadvantageous, if $d_{max} - d_{min}$ is large. It worth keeping it low.
2. Certain communication protocols may have large $d_{max} - d_{min}$ differences: for example for a token controlled bus, if the token round takes $10\,ms$, and the message forwarding is $1\,ms$, then $d_{max} = 11ms$, and $d_{min} = 1ms$, because in the worst-case situation the sending node should wait $10\,ms$ for the token, and then send the message.

3. If we wait with the action as above, then for the given node it can be stated, that relative to the message arrived, all the simultaneously or previously sent messages are arrived, or will never arrive. This relation is called permanence, and such a message is called permanent.
4. In case of irrevocable actions, neglecting the action delay/permanence can have serious consequences. Imagine a shooter with a machine gun or a catapulting pilot.
5. In case of embedded systems the communication of the different nodes involves the technological devices: the changes due to the operation of actuators are detected by sensors. The physical processes among the actuators and the sensors act as hidden communication channels the timing properties of which must be remembered while we are calculating action delay.

   Example: On the figure below the nodes A, B, C and D operate using processors. A: alarm monitor, B: actuator (setting of the valves), C: sensor (e.g. pressure or level), D: workstation of the operator.

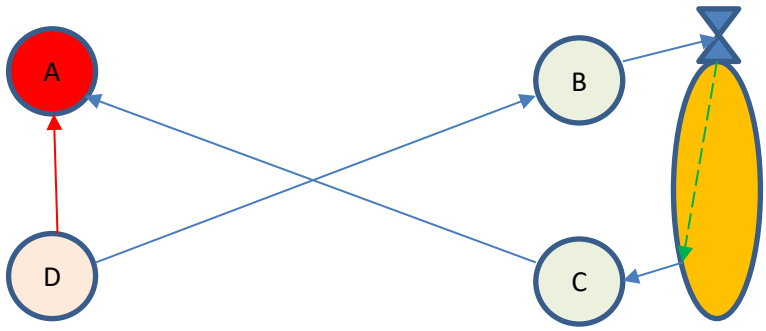
   If the alarm monitor receives messages from two other nodes, then the order of these messages is not indifferent. If the operator would like to prevent the action of the alarm monitor in case of exceeding the limit, then this message should arrive earlier to the alarm unit, i.e.:

   $$t_{DA} < t_{DB} + t_{BC} + t_{CA}$$

   Otherwise the alarm unit will act, unless action delay is applied.

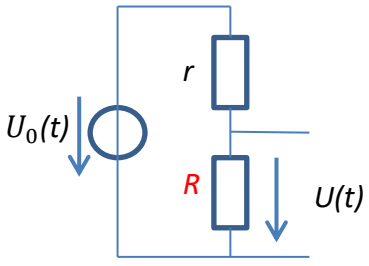
   Comments:

   1. In 1986 the nuclear accident in Chernobyl was the consequence of an experiment on the technological process, during which the automatic protecting mechanisms were switched off.
   2. The green line indicates hidden communication channel.



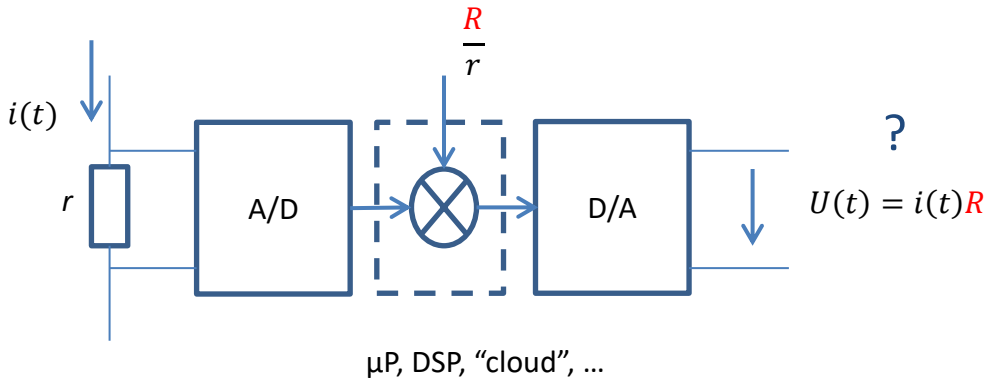
6. The value of the action delay may cause problems concerning the validity of the sent information: a measured value can become easily invalid, if it varies relatively fast.

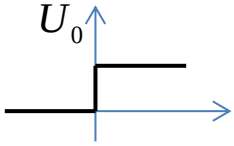
- **Idempotency:** If – to improve fault tolerance - the very same message arrives several times to a particular node, then this set of messages is called idempotent, if the effect is the same as in the case of a single one. This concept is important, because if the message is only a change, then its multiple application will result in multiple corrections, while the intention was only a single one. *Example:* setting the valve to 45° (state message) ↔ changing the valve by 5° (event message).
- **Programmable voltage divider:**



$$U(t) = U_0(t)\frac{R}{r+R}, \qquad i(t) = \frac{U_0(t)}{r+R}, \qquad U(t) = i(t)R$$

R should be programmable! Let us replace R by the programmable unit below!





The response of this system, due to the unavoidable delays, is: $U(t) = Ri(t-\Delta t)$. If U_0 is a step value, then $i(t=0) = \frac{U_0}{r}$, since: $U(t=0) = 0$. After Δt the current will reach

$$U(t=\Delta t) = R\frac{U_0}{r}, \; i(t=\Delta t) = \left(U_0 - R\frac{U_0}{r}\right)\frac{1}{r} = \left(1-\frac{R}{r}\right)\frac{U_0}{r}.$$

After another Δt the current steps in, resulting in:

$$U(t=2\Delta t) = R\left(1-\frac{R}{r}\right)\frac{U_0}{r}, \qquad i(t=2\Delta t) = \left[U_0 - R\left(1-\frac{R}{r}\right)\frac{U_0}{r}\right]\frac{1}{r} = \left[1-\frac{R}{r}+\left(\frac{R}{r}\right)^2\right]\frac{U_0}{r}$$

It can be seen that at Δt the voltage will increase, while the current decreases, at $2\Delta t$ the voltage decreases, and the current increases. The magnitude of the growth and the decrease is influenced by $\frac{R}{r}$. After $n\Delta t$, if $\frac{R}{r} < 1$:

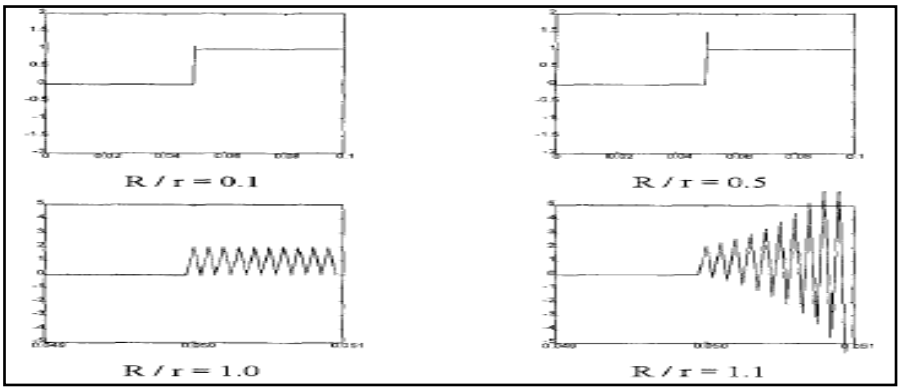
$$U(t=n\Delta t) = R\left[1-\left(\frac{R}{r}\right)+\left(\frac{R}{r}\right)^2 \mp \cdots \pm \left(\frac{R}{r}\right)^{n-1}\right]\frac{U_0}{r} \xrightarrow[n\to\infty]{} U_0\frac{R}{r+R}$$

$$i(t=n\Delta t) = \left[1-\left(\frac{R}{r}\right)+\left(\frac{R}{r}\right)^2 \mp \cdots \mp \left(\frac{R}{r}\right)^{n}\right]\frac{U_0}{r} \xrightarrow[n\to\infty]{} \frac{U_0}{r+R}.$$

Here we utilized the properties of the geometric series. Both the current and the voltage reach their steady state by a decreasing magnitude oscillation. This process is slower, if the ratio $\frac{R}{r}$ is closer to one. In case of equality the oscillation is undamped. If $\frac{R}{r} > 1$, then the magnitude of the oscillation will increase with time.

Comments:

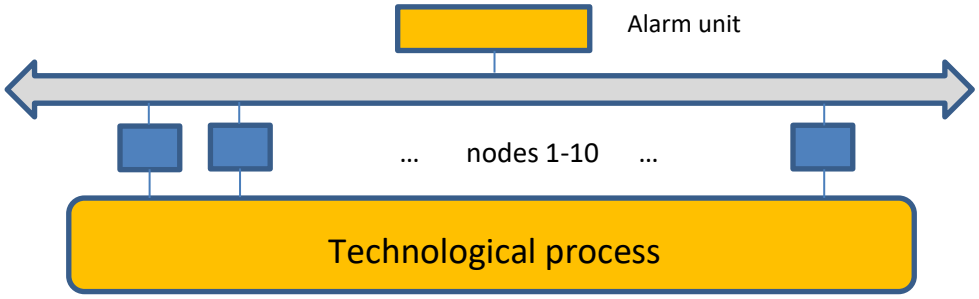
1. The above example emphasises the fact of a delay. The magnitude of the delay here does not influence the amplitudes, because the components are resistive. In case of capacitive or resistive components the magnitude of the voltage and current values would depend on the delay.
2. In this example, the stability of the system is a function of the ratio of the resistances.



- **Event triggered (ET) and time triggered (TT) systems:**

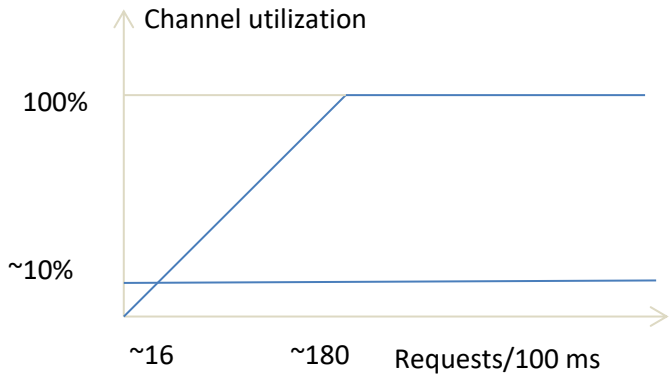
The event triggered systems execute the program associated with the event immediately after the arrival of the request. With this approach, we can get good response times, but if the number of (almost) simultaneous events increase, the throughput/capacity of the system might be insufficient, therefore to meet the deadlines will be impossible. Within the time-triggered systems a separate time-slot is assigned to every task in design time, therefore in the case of a priori known response times, the program execution can be guaranteed.

Example: A technological process is supervised by 10 nodes. Each node monitors 40 binary signals (alarm signals, e.g. limit crossings, etc.). The communication of the nodes is solved via a bus. To this bus a system-level alarm unit is also connected. The speed of the bus is 100 kbit/s. The monitoring nodes should send an alarm message to the alarm unit within 100 ms.



Event triggered operation: ET/CAN protocol is applied. The shortest message is one byte. According to the protocol the message will contain: 44 bits overhead, 1-byte data, followed by a 4-bit length inter-message gap. The total size is 56 bits. 100 kbit/s means, that within 100 ms 10 000 bits can get through. If the messages are of 56 bits, then 10 000/56 ~ 180 messages can arrive to the alarm unit within the specified time. Since 180 < 400, therefore it is not possible to send all the possible changes, the communication channel for ~180 simultaneous messages will completely saturate.

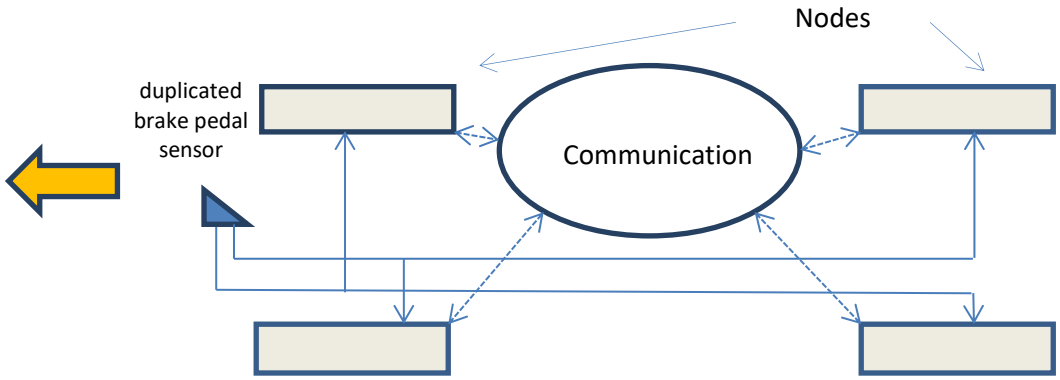
Time triggered operation: TT/CAN protocol is applied. The nodes periodically send all the signalling bits to the alarm unit. This can be performed for every 40 binary signals using a single message. According to the protocol the message will contain: 44 bits overhead, 5-byte data, followed by a 4-bit length inter-message gap. The total size is 88 bits. 100 kbit/s means, that within 100 ms 10 000 bits can get through. If the messages are of 88 bits, then 10 000/88 ~ 110 messages can arrive to the alarm unit within the specified time. Since 110 > 10, therefore all the signalling bits will arrive to the alarm unit, and what is more, at a load level of ~ 10%.


Channel utilization
100%
~10%
~16
~180
Requests/100 ms

1. Introduction (cont.)

- The importance of agreement protocols

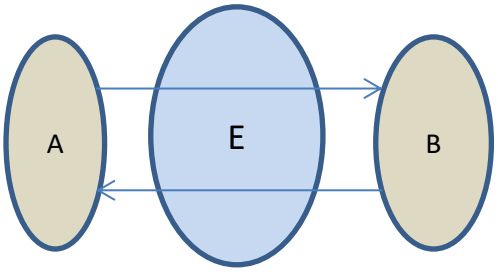
Example: brake-by-wire:



In this example, for safety reasons, duplicated brake pedal sensors are applied. The brakes of each wheel have separate control nodes. The nodes inform each other about their knowledge of the actual sensor value, and calculate the braking force. If a node is violated and fails, the corresponding wheel will run free automatically, and braking force will not be provided. The other three nodes, after observing this situation, recalculate the braking forces, and will brake safely.

In distributed systems there are several situations where run-time agreements are needed: time synchronisation, consistency of distributed states, distributed mutual exclusion, distributed transactions, distributed completion, distributed election, etc. A further problem is that even in case of errors, an agreement would be needed. This is not always possible.

Example: Two armies' problem: The allied armies, A and B together have more soldiers than the enemy, but separately each has less. A and B have decided the attack, but an agreement upon the time is still needed. The agreement needs communication, e.g. a messenger (M) should be send, but the messenger can be captured by enemy E, i.e. the communication is not error-free.



If the general of army A sends a messenger to the general of army B with the message: „Let's attack tomorrow afternoon at four o`clock", an acknowledgement is needed, since the communication channel is not error-free. (And what is more, it is also possible, that the general of army B sends a message to A with a different timing proposal.)

The problem is obvious:

- If M does not return to A, what is the conclusion? If M is captured on the way to B, A will be in danger, if acts alone. If M is captured on the way back to A, then B might depart with a given probability, but A will not, because acknowledgement was not returned.
- If M returns to A, there is a given probability that B will not depart, because B does not know whether the messenger returned. To avoid such a problem B might send his own messenger to A to check the arrival of the acknowledgement.

If we send newer messengers, the probability of an acknowledgement will increase, however the problem will not be solved, because there is always a finite probability of capturing the messenger.

Impossibility Result: It can be proven by formal methods, that to reach to an agreement of two or more distributed units in limited time, and through an asynchronous medium, which is lossy, cannot be guaranteed. What we can do: to increase the probability of the agreement.

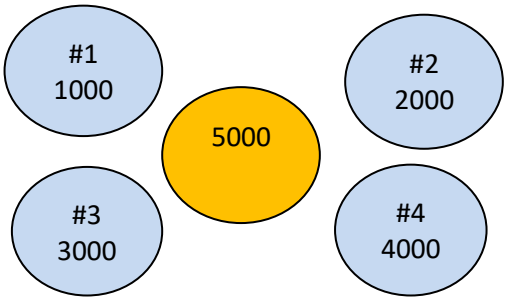
Agreement in case of Byzanthine errors:

Example: Synchronisation of clocks:

The position of clock A is 4h00m, and that of B is 4h05m. Clock C does not operate properly, because communicating with A sends 3h.55m, and to B 4h10m. This type of error is called Byzanthine error. In such a situation the agreement is not possible, because both clocks A and B realize that their value is the arithmetic mean of the two other clocks, thus there is no reason to change. To filter out a node with Byzanthine error is possible only if at least $3k+1$ nodes participate in the synchronization, where k is the number of nodes having Byzanthine error. In our case one further correctly operating clock-node (D) is required.

Example: Problem of the Byzanthine generals:

According to the figure below, the generals of the "blue" armies try to agree the total number of soldiers available for a joint action. In the meantime it turns out that one of the generals sends wrong information (there is a bug in the software). The enemy has 5000 soldiers.



The generals of the allied armies inform each other about the number of the available soldiers. Suppose that thez can communicate without any error. At the different nodes the following data are available:

#1: (1K, 2K, xK, 4K), #2: (1K, 2K, yK, 4K), #3: (1K,2K,3K,4K), #4: (1K, 2K, zK, 4K), where x, y, z are values which differ from the true values and from each other, because the general of node #3 sends wrong data (software bug). For nodes #1, #2 and #4 this error is not known, only the given data are available.

To check the values, all the nodes send their information vector to the other nodes. The node with Byzanthine error will send wrong values. Finally the information available at the correctly operation nodes (in units of 1000 soldiers):

$$\#1:\begin{bmatrix}1 & 2 & y & 4\\ a & b & c & d\\ 1 & 2 & z & 4\end{bmatrix}\quad \#2:\begin{bmatrix}1 & 2 & x & 4\\ e & f & g & h\\ 1 & 2 & z & 4\end{bmatrix}\quad \#4:\begin{bmatrix}1 & 2 & x & 4\\ 1 & 2 & z & 4\\ i & j & k & l\end{bmatrix}$$

The generals of the three properly operating nodes will get the same information from two nodes, but from the third, from general #3, the information is different. Their conclusion is $[1 \quad 2 \quad unknown \quad 4]$, i.e. minimum 7000 soldiers will participate, and the general of node #3 is the source of wrong information.

Hard RT Systems versus Soft RT Systems:

- hard real-time system (HRT): which must produce the result at the correct instant, because if we do not meet the time limitation, it might result in catastrophic consequences. (See e.g. the electronic control of vehicles).
- soft real-time system (SRT), online system: the result hase value also if we do not meet the time limitation, only the quality of the service will degrade (See e.g. transaction processing systems).

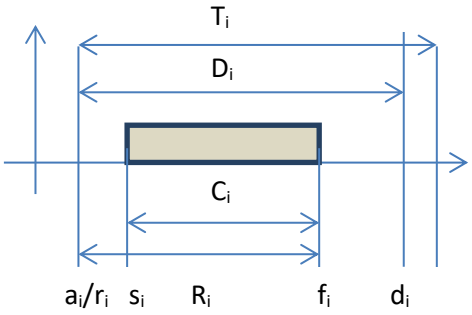
Characterisation of HRT and SRT systems:

characteristic	hard real-time	soft real-time (on-line)
response time	hard-required	soft-desired
peak-load performance	predictable	degraded
control of pace	environment	computer
safety	often critical	non-critical
size of data files	small/medium	large
redundancy type	active	checkpoint-recovery
data integrity	short-term	long term
error detection	autonomous	user assisted

- *Response time*: In case of HRT systems often int he order of millisceconds or less, preclude direct human intervention during normal operation and in critical situations. A HRT system must be highly autonomous to maintain safe operation of the process. In contrast, the response time requirements of SRT and on-line systems are often in the order of seconds. Furthermore, if a deadline is missed in a SRT system, no catastrophe can result.
- *Peak-load performance*: In a HRT system, the peak-load scenario must be well-defined. It must be guaranteed by design that the computer system meets the specified deadlines in all situations, since the utility of many hard RT applications depend on their predictable performance during rare event scenarios leading to peak load. This is in contrast to the situation in a SRT system, where the average performance is important, and a degraded operation in a rarely occurring peak load case is tolerated for economic reasons.
- *Control of pace*: A HRT system must remain synchronous with the state of the environment (the controlled object and the human operator) in all operational scenarios. It is thus paced by the state changes occurring in the environment. This is in contrast to an on-line system, which can exercise some control over the environment in case it cannot process the offered load. Consider the case of a transaction system. If the computer cannot keep up with the demands of the operators, it just extends the response time and forces the operator to slow down.
- *Safety*: The safety criticality of many RT applications has a number of consequences for the system designer. In particular, error detection must be autonomous so that the system can initiate appropriate recovery actions within the time intervals dictated by the application.
- *Size of data files*: RT systems have small data files, which constitute the RT database that is composed of the temporally accurate images of the RT-entities. The key concern in HRT systems is on the short-term temporal accuracy of the RT database that is invalidated due to the flow of real-time. In contrast in on-line transaction processing systems, the maintenance of the long-term integrity of large data files is the key issue.
- *Redundancy type*: After an error has been detected in a SRT system, the computation is rolled back to a previously established checkpoint to initiate a recovery action. In HRT systems, roll-back/recovery is limited utility for the following reasons: (1) It is difficult to guarantee the deadline after the occurrence of an error, since the roll-back/recovery action can take an unpredictable amount of time, (2) An irrevocable action which has been effected on the environment, cannot be undone, (3) The temporal accuracy of the checkpoint data is invalidated by the time difference between the checkpoint time and the instant now.

2. Scheduling

Problem: the processors should execute various tasks with different timing requirements. The timing conditions can be interpreted using the following figure:



Here a_i or r_i is the arrival/release/request time, s_i is the start time of execution, its finishing time is f_i, d_i stands for deadline, T_i is the period time, D_i=d_i-a_i is the deadline relative to the request time, C_i stands for computation time, and finally R_i=f_i-a_i is the response time.

1. <u>Periodic scheduling</u>: this is the simplest method: in design time fix time slots are assigned for the completion of the periodic requests, and this is repeated periodically. The assignment is typically clock-driven; therefore these types of scheduling are called time-triggered. They have different versions, but what is common: the decisions concerning schedules are made in design-time, thus their run-time overhead is low. A further feature is that the parameters of the HRT tasks are known and fixed in advance.

Example: To each task there is assigned a frame of 10 ms. We implement 4 functions in the following way: The first function operates with a periodicity of 50 Hz, i.e. it receives 10 ms in every 20 ms. The second function operates with a periodicity of 25 Hz, i.e. it receives 10 ms in every 40 ms. The third function at a rate of 12.5 Hz, i.e. in every 80 ms receives 10 ms, and finally at a rate of 6.25 Hz, i.e. in every 160 ms 10 ms is assigned to the fourth function.

Obviously the assignment of the frames can be different; however, it can be done only in design time. Such a scheduling can be rather unpleasant and rigid.

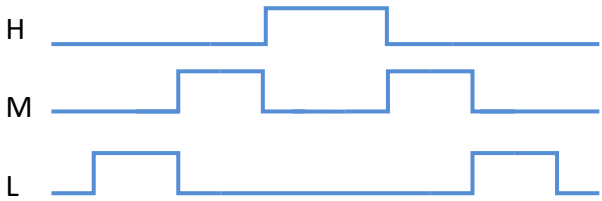
Comment: In the example above the first function utilises one half of the processor time, the second the one fourth, the third the one eights, etc. It worth bringing back the result:

$$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots\rightarrow 1,$$

i.e. the number of the functions can be increased to the infinity, if the required overall processor time is always one half of the previous. This property was utilised by the designers of the first real-time 1/3 octave spectrum analyser (Brüel & Kjaer 2131) based on digital filters in 1977! This device performs 1/3 octave analysis in the frequency range of 1.6 Hz and 20 kHz, altogether in 42 bands. The hardware is based on octave filters, for which the ratio of the frequencies at 3 dB attenuation is 1:2. This property is utilised in the following way: imagine an octave filter with a centre frequency of 16 kHz and sampling frequency of f_m=66.667 kHz. If the signal is properly bandlimited, then the octave filter with a centre frequency of 8 kHz can be successfully operated at a sampling frequency of f_m/2, etc. The evaluation of the highest frequency range takes one half of the time, all the other ranges share the second half of the time.

2. <u>Time-shared/round-robin scheduling</u>: The tasks ready to run are placed into a FIFO (First-In First-Out), and the task first on the list will get the processor for a fixed amount of time. This time slot is typically few times 10 ms, and independent of the tasks. If the given task is not completed within the slot, it will be interrupted, and placed to the last position of the FIFO.

3. <u>Priority based scheduling</u>: From the set of tasks ready to run the one having the highest priority will run. Priority assignment can be performed either in design or in run-time. For simplicity imagine that to every task a different priority level is assigned. The operation is illustrated by the figure below. We have one low priority (L=low), one medium priority (M=medium) and one high priority (H=high) task. This assignment happened in design-time. All the tasks start running immediately after the request, if their priority is the highest among the tasks ready to run:



The response time of the lowest priority task on the figure is: $R_L=C_L+C_M+C_H$. If the medium and/or the high priority task are released periodically, then depending on the time relations, it might happen, that these tasks will run more than one time during R_L. In a more general case, for a task at priority level i, the worst-case response time can be calculated using the following formula:

$$R_i = C_i + I_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k ,$$

where I_i is the so-called interference. The interference time is the total computation time of those higher priority tasks, which prevent task i to complete its actual run. $\forall k \in hp_i$ refers those tasks, which have higher priority than i (hp=higher priority). The $\lceil\ \rceil$ sign is the operator of assigning the upper integer. $\lceil 1.02 \rceil = 2$, $\lceil 2.0 \rceil = 2$. Since in the above formula the unknown R_i on the left-hand side is present also in the argument of the highly nonlinear function on the right-hand side, it can be evaluated only via an iterative procedure:

$$R_i^{n+1} = C_i + I_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i^n}{T_k} \right\rceil C_k$$

The iteration will stop at step n_0 where $R_i^{n_0+1} = R_i^{n_0}$. The name of this method in the literature is Deadline Monotonic Analysis (DMA). It supposes priority assignment according to the deadlines: tasks with larger D_i will have in lower priority. Only such cases are considered, where $D_i \le T_i$. The method is suitable both for periodic and sporadic tasks.

Periodic task: is characterised by known and fixed period T_i.

Sporadic task: the requests are not periodic, but there is a known and fixed T_i value that is the minimum time between two subsequent requests.

Aperiodic task: the requests are not periodic, and there is no specified T_i between two requests, i.e. a request can be followed immediately by a second request. Obviously in the case of aperiodic tasks the DMA methos cannot be applied.

It might be important to emphasize, that by the application of the DMA method not the response time but the worst-case response time will be derived.

Example: A system with four tasks can be characterized with the following time values (the time is measured in miliseconds):

Task	T	C	D
1	250	5	10
2	10	2	10
3	330	25	50
4	1000	29	1000

The priority order corresponds to the order of the tasks in the list. If the deadline values are equal secondary conditions are used to decide priority order. In the example, the first task has higher computation time, i.e. its laxity is smaller, therefore the higher priority might be a better choice. Let us calculate the worst-case response time of task 3. The iterative procedure:

step	R^n	I	R^{n+1}
1	0	0	25
2	25	5+3*2	36
3	36	5+4*2	38
4	38	5+4*2	38

Comments:

1. 38<50, thus task 3 will meet the deadline also under worst case conditions.
2. Note that the data of task 4 were not utilized at all. They are not required.
3. Note that the tasks up till now were considered independent from each other. However, if they are not independent, i.e. they are communicating. In this case might happen, that higher priority tasks should wait for data provided by lower priority. This additional waiting time will increase both the response time and its worst-case version.

2. Scheduling (Cont.)

Example: An embedded system, devoted to executing requests of four periodic/sporadic tasks and one periodic/sporadic interrupt, has the following parameters (time values are in ms):

Task	T	C	D
i_1	10	0.5	3
t_1	3	0.5	3
t_2	6	0.75	6
t_3	14	1.25	14
t_4	50	5	50

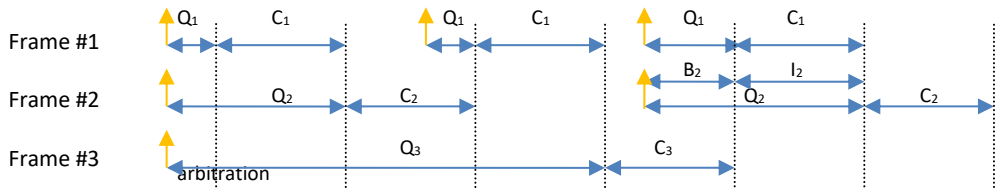
Let us calculate the worst-case response time of task t_4 using the iterative procedure! The interrupt will be served on the highest probability, otherwise the priority level of the tasks follows the deadline monotonic assignment. The iterative procedure:

Step	R^n	I	R^{n+1}
1	0	0	5
2	5	0.5+1.0+0.75+1.25	8.5
3	8.5	0.5+1.5+1.50+1.25	9.75
4	9.75	0.5+2.0+1.50+1.25	10.25
5	10.25	1.0+2.0+1.50+1.25	10.75
6	10.75	1.0+2.0+1.50+1.25	10.75

Since 10.75<50, the deadline is met.

Example: A modified version of DMA method can be used also if the operation is not pre-emptive, i.e. when the running task will not be pre-empted. The example is the response time analysis of the priority-based CAN bus.

The key features of the communication through the CAN (Control Area Network, ISO 11898, Bosch) bus can be seen on the figure below. Here the forwarding of three messages of different priority is to be solved.



The vertical dashed lines on the figure indicate the so-called arbitration points. At these time locations is decided which message (frame) will be forwarded next. The priority is decreasing from the top. The first requests are simultaneous. Till the arbitration time all should wait. At the arbitration, the forwarding of frame#1 is decided. C_1 stands for the communication time that corresponds to the computation time. In this example, the communication time is the same for every frame. After the second arbitration comes the forwarding of frame#2. Within this communication interval the next highest priority request arrives. It should wait till the arbitration, but after the arbitration this message is immediately forwarded. At the fourth arbitration, it turns out that only the request concerning frame#3 is on the list, therefore it will be forwarded. In the meantime, simultaneous high and medium priority requests arrive. The waiting time of the medium priority can be divided into two parts: one is the so-called blocking time B_2, during which lower priority frame is forwarded, the other is the so-called I_2, during which higher priority frame is forwarded. Based on these components, the response time calculation is based on the following formulas:

$$R_i = C_i + Q_i \text{, where } Q_i = B_i + \sum_{\forall k \in hp_i} \left\lceil \frac{Q_i}{T_k} \right\rceil C_k .$$

Here Q_i is the worst-case waiting time of the frame denoted by i. B_i is blocking time, which is the longest message forwarding time of any lower priority frame. From the figure, the worst-case blocking time is the time between two arbitrations. This is the case if the request arrived immediately after the previous arbitration.

Let us have the following set of messages:

Message	T [ms]	C[ms]
1	3	1.35
2	6	1.35
3	10	1.35
4	30	1.35
5	40	1.35
6	40	1.35
7	100	1.35

The messages are periodic, and their priority is decreasing from the top. The requests are asynchronous. The 7th message is related to braking, it should be received in 100 ms. The iteration for the waiting time:

Step	Q^n	I							B	Q^{n+1}
		1	2	3	4	5	6	Sum		
1	0	-	-	-	-	-	-	0	1.35	1.35
2	1.35	1	1	1	1	1	1	8.1	1.35	9.45
3	9.45	4	2	1	1	1	1	13.5	1.35	14.85
4	14.85	5	3	2	1	1	1	17.55	1.35	18.9
5	18.9	7	4	2	1	1	1	21.6	1.35	22.95
6	22.95	8	4	3	1	1	1	24.3	1.35	25.65
7	25.65	9	5	3	1	1	1	27	1.35	28.35
8	28.35	10	5	3	1	1	1	28.35	1.35	29.7
9	29.7	10	5	3	1	1	1	28.35	1.35	29.7

The worst-case waiting time is 29.7 ms, thus the worst-case response time: 29.7ms+1.35ms=31.05 ms. This value is smaller than the specified 100 ms: the deadline is met.

Comment: The different versions of the DMA analysis are widely used for worst-case response time analysis to optimize products concerning the necessary clock frequencies/bandwidths to increase noise immunity and reduce costs. (Volvo Corporation has introduced this technique already in 1995, first regarding S80.)

Schedulability, schedulability tests:

- *necessary:* if the necessary condition is not met, then no schedule exists.
- *sufficient:* if the sufficient condition is met, then a schedule always exists.
- *exact:* gives the necessary and sufficient conditions, and shows the existence of the schedule. The complexity of the exact schedulability test is high, these are so called NP-complete problems, which are hard to handle, and therefore they will not be considered.

For periodic tasks among the necessary conditions the processor utilization factor can be mentioned, which is the sum of the processor demands relative to the unit of time:

$$\mu = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

For a single processor system if $\mu \le 1$ is not met, then the tasks are not schedulable, i.e. $\mu \le 1$is a necessary condition.

Scheduling strategies:

Rate-monotonic (RM) (1973): For periodic and independent tasks if D_i=T_i and C_i are known and constant. The highest priority is assigned to the task with the shortest period. The procedure is pre-emptive. We assume that the time of context switching between tasks is negligible. For the RM algorithm sufficient test is available. If the above conditions hold, and for the processor utilisation factor the following inequality is met

$$\mu = \sum_{i=1}^{n} \frac{C_i}{T_i} \le n\left(2^{\frac{1}{n}} - 1\right) \xrightarrow[n\to\infty]{} \ln 2 \sim 0.7,$$

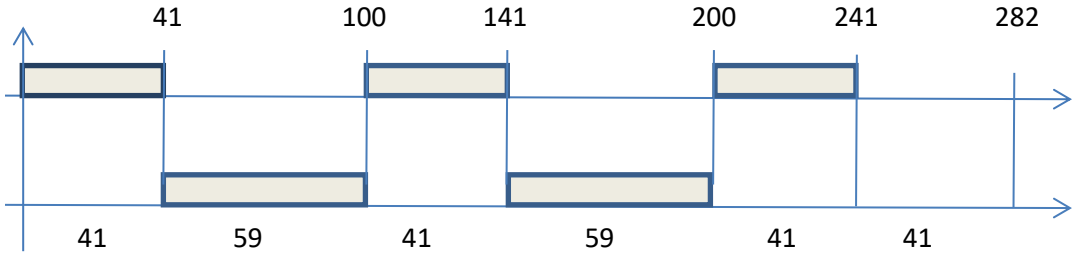
(Where n denotes the number of the tasks to be scheduled), then the tasks are schedulable. It might happen that the actual set of tasks is schedulable with the RM strategy even at higher processor utilisation; however there is no guarantee for it. Simulations with randomly selected T_i and C_i values were reported successful up to $\mu = 0.88$. To achieve 100% utilization when using fixed priorities, assign periods so that all tasks are harmonic. This means that for each task, its period is an exact multiple of every other task that has a shorter period.

Example: This example illustrates that under what period and computation time conditions reaches the RM strategy the limits of schedulability. If n=2, setting

$\frac{T_2}{T_1} = \sqrt{2}$, $C_1 = T_2 - T_1$, and $\frac{C_1}{T_1} = \frac{T_2 - T_1}{T_1} = \frac{C_2}{T_2}$, we have: $\mu = 2\left(\sqrt{2} - 1\right)$, or for arbitrary i if

$\frac{T_{i+1}}{T_1} = 2^{\frac{1}{n}}$, $C_i = T_{i+1} - T_i$, then $\mu = n\left(\frac{T_{i+1}}{T_i} - 1\right) = n\left(2^{\frac{1}{n}} - 1\right)$.

For a system consisting of two tasks we have: $T_1 = 100$, $C_1 = 41$, $T_2 = 141$, $C_2 = 59$, all in *ms*. The processor utilisation factor $\frac{41}{100} + \frac{59}{141} = 0.41 + 0.4184 = 0.8284$, i.e. nearly the value given by the above formula. If the requests are simultaneous, the time conditions are the following:



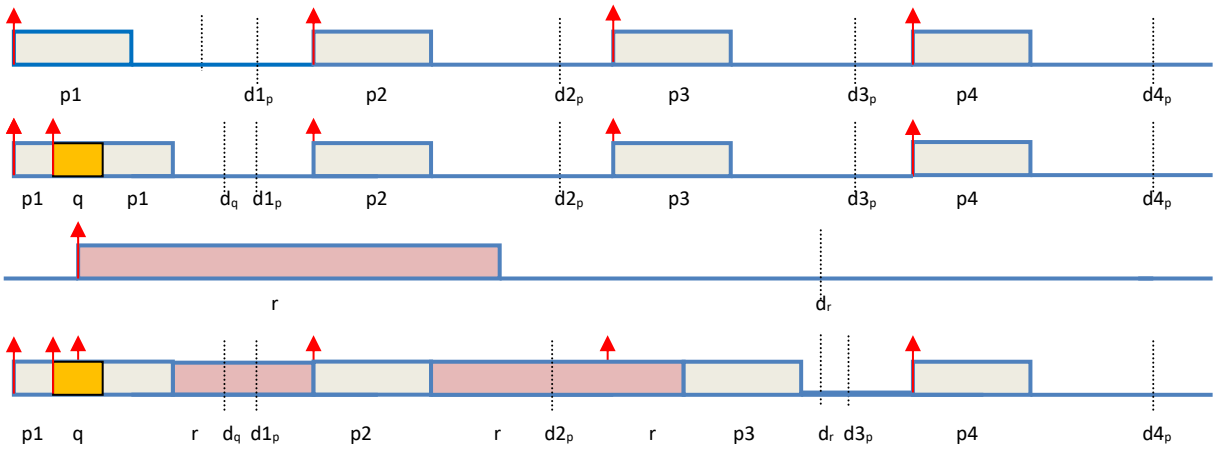
If there is slight increase in the computation time, then the RM strategy will fail. At the same time between 241 and 282 there is no schedulable task, thus the processor utilisation cannot be increased.

Comments:

1. If the RM scheduling strategy is applied, the most disadvantageous is the case when all the tasks start with zero phase, i.e. the first requests are simultaneous. From schedulability point of view it is advantageous, if the starting phases of the tasks are different, i.e. non-simultaneous.
2. If the RM scheduling strategy is applied, and the necessary condition is met, but the sufficient not, then the schedulability analysis should be performed for smallest common multiple of the periods that can be extremely large.

2. Scheduling (Cont.)

Earliest Deadline First (EDF) strategy: We assume that the tasks are periodic, independent of each other, $D_i \le T_i$ and C_i are known and are constant. Priority assignment is in run-time, and the processor is given to the task having the earliest deadline. The operation is pre-emptive. Here we also assume that the time of context switching is negligible. For the EDF algorithm sufficient schedulability test can be given: tasks meeting the above conditions are schedulable up to $\mu \le 1$, i.e. 100% processor utilisation is possible. The operation is illustrated by the following figure:



In the first line, we can see the periodic requests and runs of task p (p…) and the corresponding deadlines (d…$_p$). During the run of p1, indicated in the second line, the request of task q arrives. Since its deadline is earlier than that of p1, therefore q will run. The third line shows the request and the deadline of task r. The fourth line gives the summary of the runs: after completing the runs of q and p1, task r will run, since there is no other task ready to run which has earlier deadline. As the request of p2 arrives it will have the earliest deadline, thus r will be pre-empted, and p2 will run. After completing its operation r will be resumed. The deadline of p3 is later than that of r, therefore first r will finish, but p3 will also be completed before deadline.

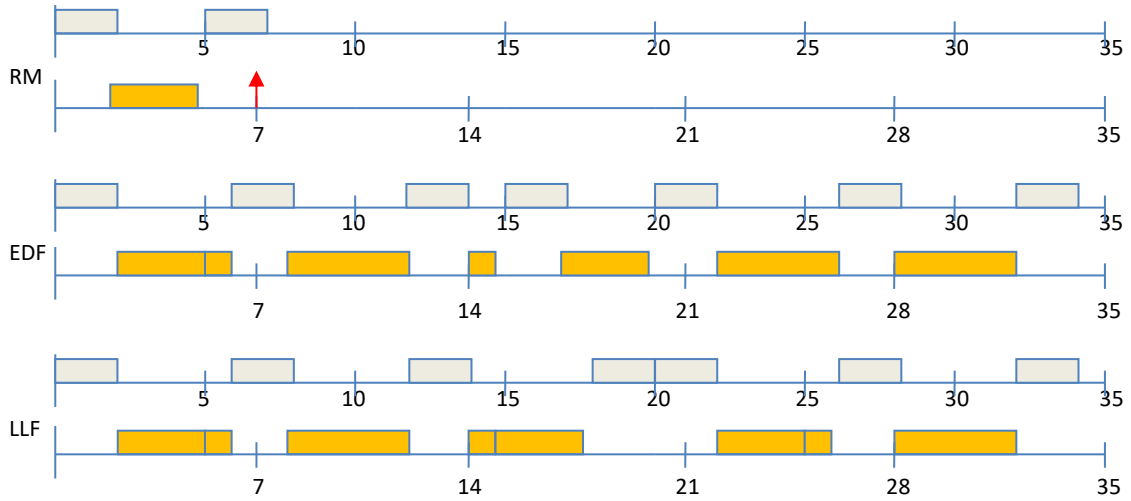
Least Laxity First (LLF) strategy: Like EDF. The conditions of its application are the same, but instead of the task having the earliest deadline, the processor is assigned to the task having the smallest laxity. This is the difference of the deadline and the remaining computation times at the time instant of investigation. For the LLF algorithm sufficient schedulability test can be given: tasks meeting the above conditions are schedulable up to $\mu \le 1$, i.e. 100% processor utilisation is possible.

Comment: The EDF and LLF strategies are applicable also for aperiodic tasks, but since the processor utilisation factor can only be interpreted in a different way, the sufficient condition above cannot be used.

Example: Comparison of the RM and EDF algorithms. We have two tasks. The deadlines equal the periods. T_1=5 ms, C_1=2 ms, T_2=7 ms, C_2=4 ms. The processor utilisation factor:

$$\frac{2}{5}+\frac{4}{7}=0.4+0.57=0.97.$$

Here the necessary condition of the schedulability is met, but the sufficient condition only for the EDF strategy. If at the beginning the requests are simultaneous, the RM, the EDF and the LLF algorithms will give the following schedules:



If the RM strategy is applied, then the second task will miss the deadline at 7 ms, but using EDF or LLF the tasks are schedulable. Both for EDF and LLF it is obvious that if the deadlines are equal, the applied schedule should result in less context switching, because context switching take time. All the task-specific information of the pre-empted task must be saved: typically, the content of the processor's registers must be copied into the task-specific Task Control Block (TCB), while TCB of the task decided to run should be loaded into the registers of the processor. These copying are supported by fast mechanisms, but still they need time.

Proof of the EDF schedulability

The proof is given for periodic tasks with $D_i=T_i$. The statement is the following: A set of periodic tasks is schedulable with EDF if and only if

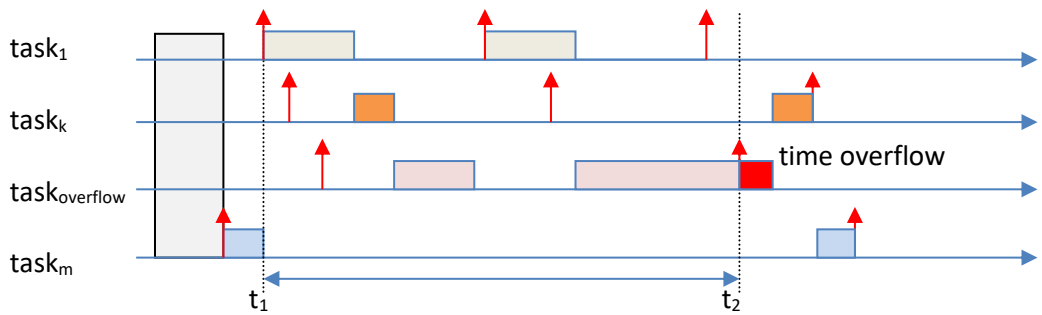
$$\mu = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.$$

Proof: *Only if*: We show that a task set cannot be scheduled if $\mu > 1$. In fact, by defining $T = T_1 T_2 \ldots T_n$, the total demand of computation time requested by all tasks in T can be calculated as

$$\sum_{i=1}^{n} \frac{T}{T_i} C_i = \mu T.$$

If $\mu > 1$ – that is, if the total demand μT exceeds the available processor time T – there is no feasible schedule for the task set.

Proof: *If*: We show the sufficiency by contradiction. Assume that the condition $\mu < 1$ is satisfied and yet the taskset is not schedulable. The next figure helps to understand the proof. Here we can see the schedule of periodic tasks according to EDF.



If our assumption is that the task-set is not schedulable, then there must be such a task, which misses its deadline. Let t_2 be the instant at which the time/overflow occurs and let $[t_1, t_2]$ be the longest interval of continuous utilisation before the overflow, such that only instances with deadline less than or equal to t_2 are

executed in $[t_1, t_2]$. Note that t_1 must be the release time of some periodic instance. Let $C_P(t_1, t_2)$ be the total computation time demanded by periodic tasks in $[t_1, t_2]$, which can be computed as:

$$C_P(t_1,t_2) = \sum_{r_k \le t_1, d_k \le t_2} C_k = \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i ,$$

where $\lfloor ... \rfloor$ denotes the lower-integer function. (Note that for task$_1$ the response to the third request is not considered, therefore the assignment of the lower-integer is correct.) Now, observe that:

$$C_P(t_1,t_2) = \sum_{r_k \le t_1, d_k \le t_2} C_k = \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i \le \sum_{i=1}^{n} \frac{t_2 - t_1}{T_i} C_i = (t_2 - t_1)\mu .$$

Since a deadline is missed at t_2, $C_P(t_1, t_2)$ must be greater than the available processor time $((t_1 - t_2))$; thus, we must have:

$$(t_2 - t_1) < C_P(t_1,t_2) \le (t_2 - t_1)\mu ,$$

that is, $\mu > 1$, which is a contradiction, i.e. the original statement is false.

Combined Scheduling of hard RT and soft RT tasks:

Two rules are applied:

Rule#1: Every task should be schedulable with average execution and average arrival times.
Rule#2: Every hard RT task should be schedulable with worst-case execution and worst-case arrival time.

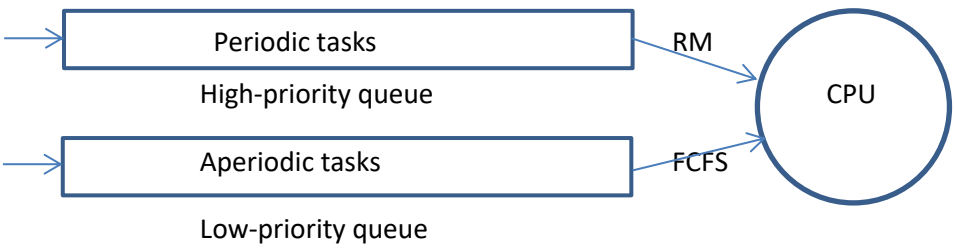
Combined Scheduling of periodic and aperiodic tasks: Fixed Priority Servers

We concentrate on hard RT systems, and soft aperiodic systems, but soft RT systems can also be considered.

The algorithms presented here rely on the following assumptions:

1. Periodic tasks are scheduled based on a fixed-priority assignment; namely, the RM algorithm;
2. All periodic tasks start simultaneously at time t=0 and their relative deadlines are equal to their periods;
3. Arrival times of aperiodic requests are unknown;
4. When not explicitly specified, the minimum interarrival time of a sporadic task is assumed to be equal to its deadline.

Background Scheduling:



The major advantage of background scheduling is its simplicity. Its drawback is that the response time of the aperiodic tasks can be very large. (FCFS=First-Come-First-Served.)

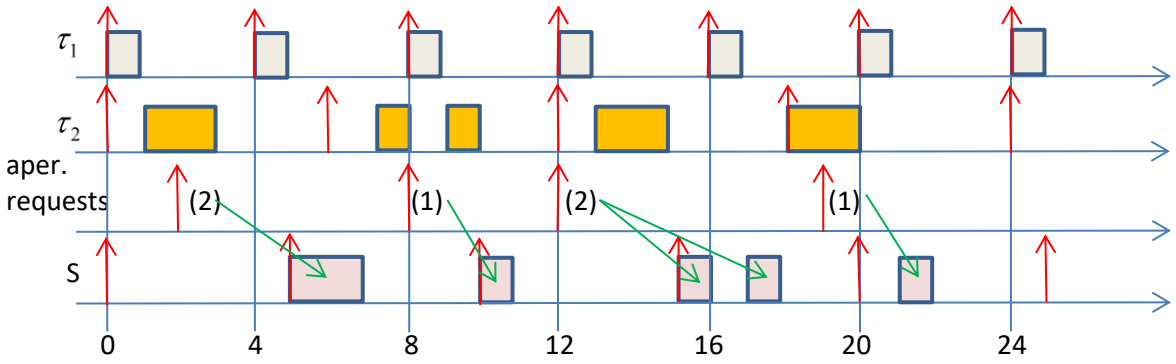
If the response time of the aperiodic tasks is critical, the so-called server methods give better result. The server method provides processor time for the aperiodic tasks in a separated way. The tool of this solution is the server task, which is scheduled together with the periodic tasks.

Polling Server (PS): The aperiodic requests are scheduled with the help of the so-called server task (S) using the server capacity (T_S,C_S), and a separated scheduling mechanism. Is there is no aperiodic request while the server task could run, the server task suspends itself, and its capacity will not be preserved.

Example: Let us have T_S=5, C_S=2. There are two additional tasks to be scheduled:

	C	T
τ_1	1	4
τ_2	2	6

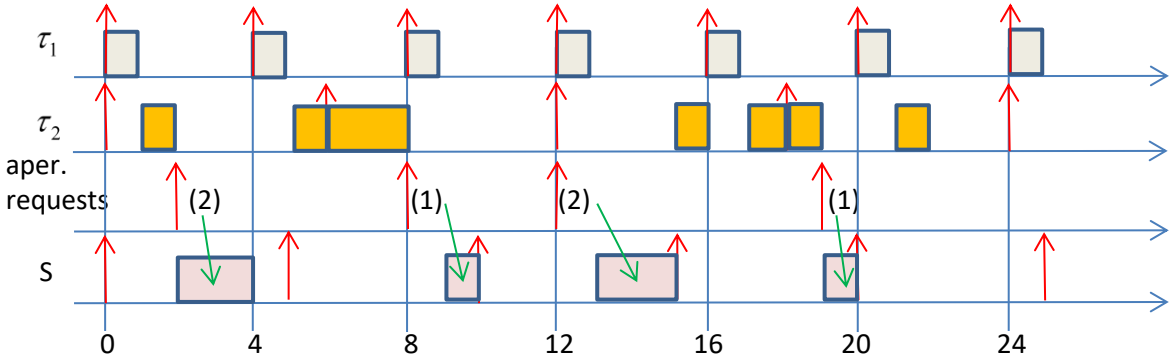
The server task (according to RM) will have medium priority. Assuming simultaneous start, the schedule will be the following:



In worst case situations, the fulfilment of the aperiodic request will occur only after an almost complete server period.

Deferrable Server (DS): The aperiodic requests are scheduled with the help of the so-called server task (S) using the server capacity (T_S,C_S), and a separated scheduling mechanism. Is there is no aperiodic request while the server task could run, the run of the DS will be postponed, its capacity is preserved till the end of the period. With this method, much better response times to aperiodic requests can be achieved.

Example: For the previous example the schedule will be the following:



The response times, obviously depending also on the priority level of the server task, are much better.

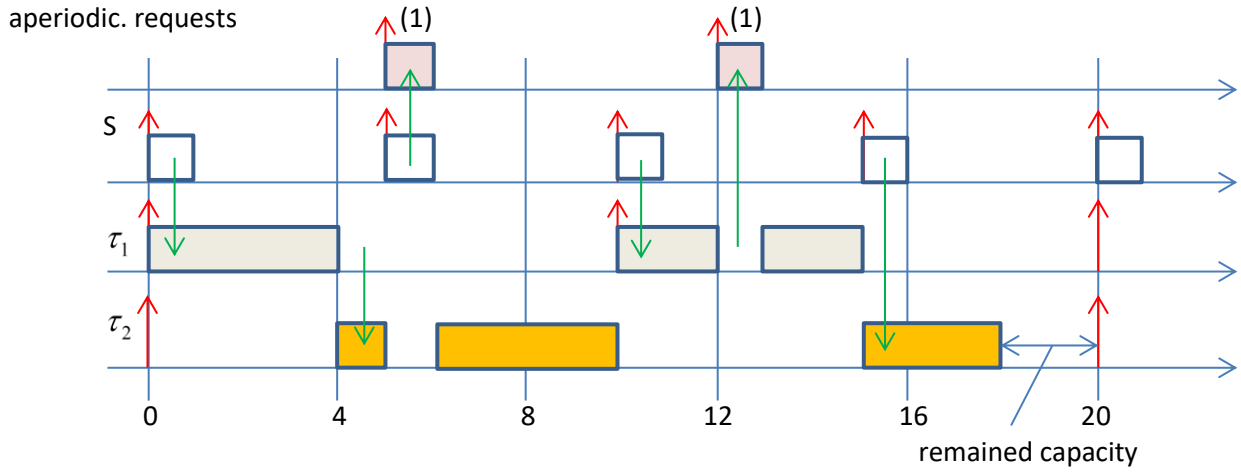
2. Scheduling (Cont.)

Priority Exchange Server (PE): Like DS, the PE algorithm uses a periodic server (usually at a high priority) for servicing aperiodic requests. However, it differs from DS in the manner in which the capacity is preserved. Unlike DS, PE preserves its high-priority capacity by exchanging it for the execution time of a lower-priority periodic task. When a priority exchange occurs between a periodic task and a PE server, the periodic task executes at the priority level of the server while the server accumulates a capacity at the priority level of the periodic task. Thus, the periodic task advances its execution, and the server capacity is preserved at a lower priority.

Example: The PE server has T_S=5, C_S=1. The further tasks to be scheduled:

	C	T
τ_1	4	10
τ_2	8	20

The server task has the highest priority (RM strategy). Note, that the processor utilization factor is: $\mu = \frac{1}{5} + \frac{4}{10} + \frac{8}{20} = 1$. Supposing simultaneous start the schedule is the following:

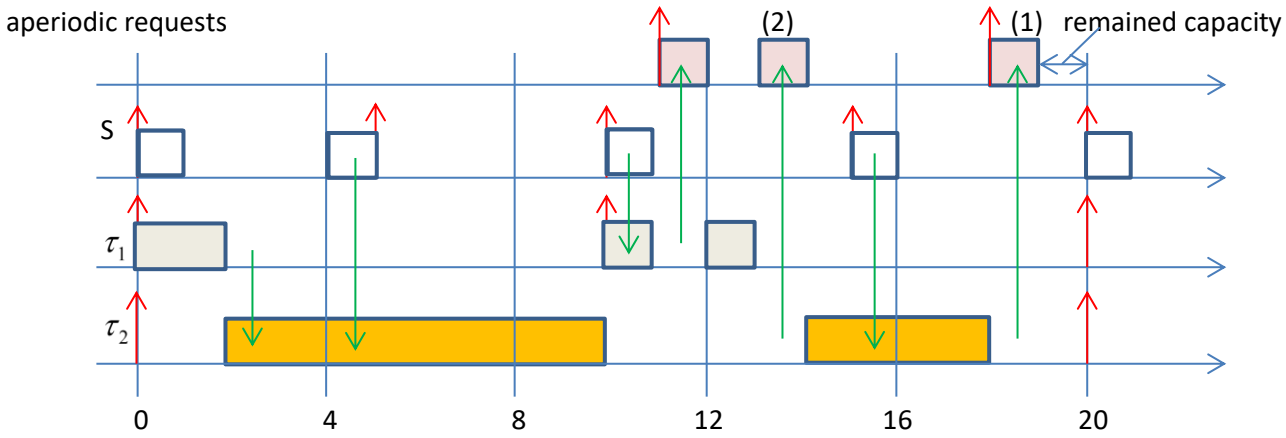


Since there is no aperiodic request to process, the server capacity is used by task τ_1. As a consequence task τ_2 can run earlier, i.e. the server capacity will be used at this level. The server capacity of the second period is used immediately. The server capacity of the third period is used by τ_1, but it is given back to fulfil the second aperiodic request. The server capacity of the fourth period is used by τ_2. Between [18-20], at the priority of τ_2, remaining server capacity is available, which could be used to serve further aperiodic requests.

Example: The PE server has T_S=5, C_S=1. The further tasks to be scheduled:

	C	T
τ_1	2	10
τ_2	12	20

The server task has the highest priority (RM strategy). Note, that the processor utilization factor is: $\mu = \frac{1}{5} + \frac{2}{10} + \frac{12}{20} = 1$. Supposing simultaneous start the schedule is the following:



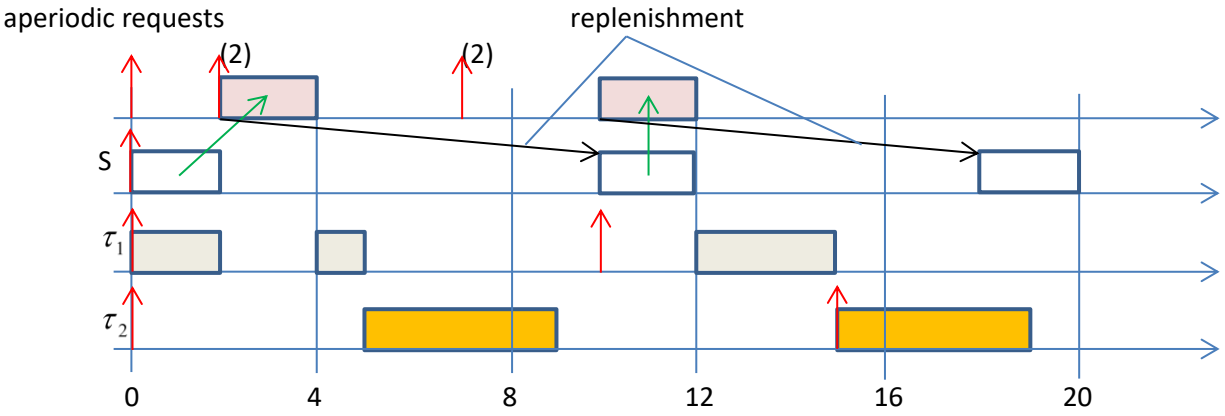
On the figure we can see, that if we pass capacity to another task, then it can be utilized at the priority of the receiving task. At time instant 11 the first unit of the requested two can be found at τ_1, while the second at τ_2. Therefore at 12 the execution of τ_1 is continued, and the aperiodic task should wait. At 18 the aperiodic task will get processor time from τ_2. Between [19-20], at the priority of τ_2, remaining server capacity is available, which could be used to serve further aperiodic requests.

Sporadic Server (SS): Allows enhancing the average response time of aperiodic tasks without degrading the utilization bound of the periodic task set. The SS algorithm creates a high-priority task for servicing aperiodic requests and, like DS, preserves the server capacity at its high-priority level until an aperiodic request occurs. However, SS differs from DS in the way it replenishes its capacity. Whereas DS and PE periodically replenish their capacity to its full value at the beginning of each server period, SS replenishes its capacity only after it has been consumed by aperiodic task execution.

Example: The SS server has T_S=8, C_S=2. The further tasks to be scheduled:

	C	T
τ_1	3	10
τ_2	4	15

The server task has the highest priority (RM strategy). Supposing simultaneous start the schedule is the following:



Slack stealing: Offers substantial improvements in response time over the previous methods. Unlike these methods, the Slack Stealing algorithm does not create a periodic server for the aperiodic service. Rather it creates a passive task, referred to as the Slack Stealer, which attempts to make time for servicing aperiodic tasks by "stealing" all the processing time it can from the periodic tasks without causing their deadlines to be missed. If $C_i(t)$ is the remaining computation time at time *t*, then the slack of a task τ_i is

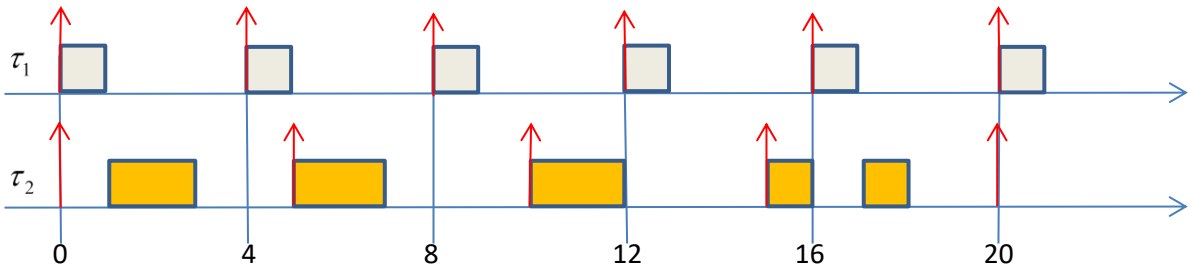
$$Slack_i(t) = d_i - t - C_i(t)$$

The price to be paid for the good response times is: larger computational and implementation complexity, larger memory requirement.

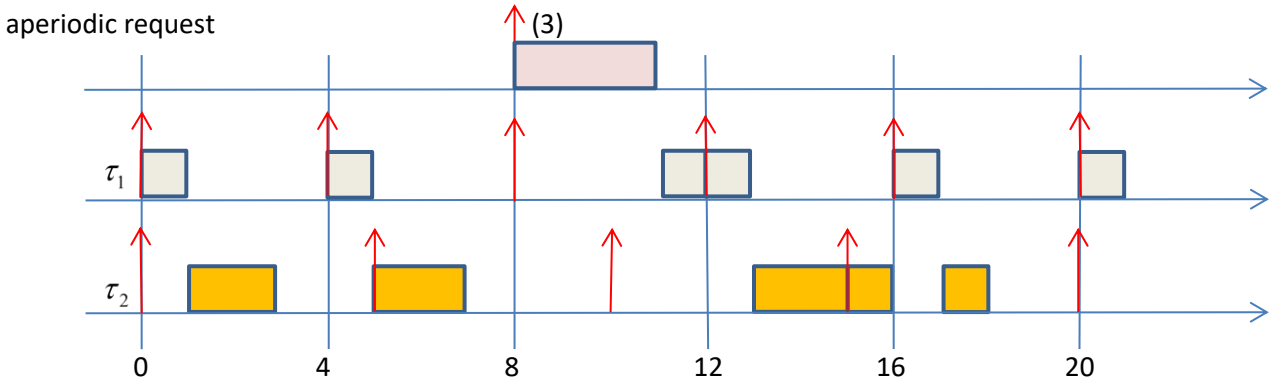
Example:

	C	T
τ_1	1	4
τ_2	2	5

According to the normal RM strategy:



Upon arrival of aperiodic request, the slack is calculated, and this amount of processor time is given to the aperiodic task at the highest priority:



Dual Priority Scheduling: Idea: there is no benefit in early completion of hard tasks. Use three ready queues: High, Middle and Low. The hard RT tasks start running at Low priority. The soft RT and the aperiodic tasks run at middle priority. The hard RT tasks at approaching the so-called promotion time (x_i) before their deadline (d_i) are promoted and put in the High queue just to able to meet their deadline. The promotion time can be calculated as follows:

$$x_i = d_i - R_i$$

(where $R_i = B_i + C_i + I_i$). Obviously the three priority queues can be subdivided into further priority levels.

Comment: The server tasks introduced above were scheduled using the RM strategy. Similar solutions can be derived in the case of the EDF strategy. These are dynamic priority servers.

Schedulability if $Di < Ti$: almost all the methods, statements and proofs discussed up till now cover cases where $D_i = T_i$. If the deadline is earlier than the period, then the priority can be assigned according to the deadlines. One such a technique is the *Deadline Monotonic* (DM) algorithm, where the highest priority is assigned to the task having earliest deadline relative to the request time. Obviously the condition

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \le n\left(2^{\frac{1}{n}} - 1\right)$$

can be a sufficient schedulability test, however, this is not necessary, and sometimes rather pessimistic. Less pessimistic, if assuming simultaneous start (since concerning processor demand this is the most disadvantageous) for all the tasks we investigate the fulfilment of the condition $C_i + I_i \le D_i$. Here $I_i = \sum_{\forall k \in hp_i} \left\lceil \frac{D_i}{T_k} \right\rceil C_k$. This condition is sufficient but not necessary. The necessary and sufficient condition is given by the already discussed worst/case response time analysis:

$$R_i = C_i + I_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k < D_i .$$

If the EDF strategy is applied while $D_i < T_i$, then the processor utilisation factor cannot be used. Instead the so-called processor demand approach can be suggested. First this will be introduced for the $D_i = T_i$ case. In general within an arbitrary interval $[t, t+L]$the processor demand of a task τ_i is the time needed to become completed till the time instant $t + L$ or before. In the case of such periodic tasks, which start running at $t = 0$, and for which $D_i = T_i$, the total processor time in any $[0, L]$ interval is:

$$C_p(0,L) = \sum_{k=1}^{n} \left\lfloor \frac{L}{T_k} \right\rfloor C_k .$$

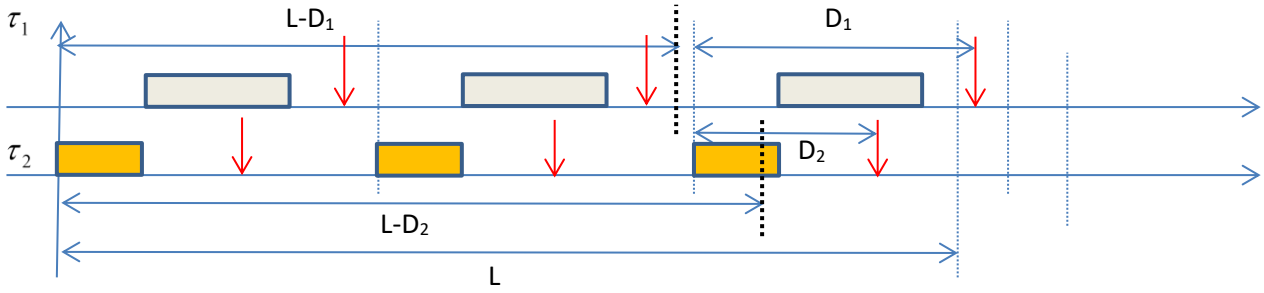
Statement: A periodic task set can be scheduled by EDF if and only if for any $L > 0$:

$$L \ge \sum_{k=1}^{n} \left\lfloor \frac{L}{T_k} \right\rfloor C_k . \qquad (*)$$

Proof: On one hand, since $\mu = \sum_{i=1}^{n} \frac{C_i}{T_i} \le 1$, therefore $L \ge \mu L = \sum_{k=1}^{n} \left(\frac{L}{T_k} \right) C_k \ge \sum_{k=1}^{n} \left\lfloor \frac{L}{T_k} \right\rfloor C_k$. On the other, if $\mu > 1$, then there exists such $L > 0$, for which (*) does not hold, since if e.g. $L = lcm(T_1 T_2 \ldots T_n)$, then:

$$L < \mu L = \sum_{k=1}^{n} \left(\frac{L}{T_k} \right) C_k = \sum_{k=1}^{n} \left\lfloor \frac{L}{T_k} \right\rfloor C_k .$$

If $D_i < T_i$, then the calculation of $C_p(0,L)$ is different. To see this, consider the case of the two tasks on the next figure. For simplicity let us have the same period but different deadlines:



Based on the figure, since deadline of the third period is out of the range of the interval of length L, the processor demand of τ_1: $C_1(0,L) = \left\lfloor \frac{L}{T_1} \right\rfloor C_1$, while for τ_2 this can be given by $C_2(0,L) = \left(\left\lfloor \frac{L}{T_2} \right\rfloor + 1 \right) C_2$. Using the figure, it is easy to understand that the two cases can be handled with a single formula of the form:

$$C_i(0,L) = \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i .$$

With this formula: A periodic task/set can be scheduled by EDF if and only if for every $L > 0$

$$L \geq \sum_{k=1}^{n} \left(\left\lfloor \frac{L - D_k}{T_k} \right\rfloor + 1 \right) C_k .$$

Summary:

	$D_i=T_i$	$D_i<T_i$
static priority	**RM** processor utilisation approach $\mu \leq n\left(2^{\frac{1}{n}} - 1\right)$	**DM** response time approach $\forall i - re \quad R_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k \leq D_i$
dynamic priority	**EDF** processor utilisation approach $\mu \leq 1$	**EDF** processor demand approach $\forall L > 0 \quad L \geq \sum_{k=1}^{n} \left(\left\lfloor \frac{L - D_k}{T_k} \right\rfloor + 1 \right) C_k$

Extensions of the response time calculation:

1. *Cooperative scheduling*: At a given point of the task execution it might be a requirement the completion of the task as early as possible. This can be achieved if the pre-emption of the task is prohibited till the end it's run. If this takes time F_i, then the response time can be written in the form of $R_i = R_i' + F_i$, where

   $$R_i^{'} = B_i + C_i - F_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i^{'}}{T_k} \right\rceil C_k .$$

   In this case the last part of the execution if runs, it will run on the highest priority.

2. *Fault tolerance*: exception handlers, recovery blocks, etc., generally require additional computation time: C_i^f extra computation time for every task.

   For a single fault the extended formula:

   $$R_i = B_i + C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k + \underbrace{\max}_{k \in hep_i} C_k^f .$$

   Here, since we do not know which higher-priority task execution was faulty, for the worst-case calculation we select the longest computation time. (*hep=higher or equal priority*)

   For *F* faults:

   $$R_i = B_i + C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k + \underbrace{\max}_{k \in hep_i} \left(F C_k^f \right).$$

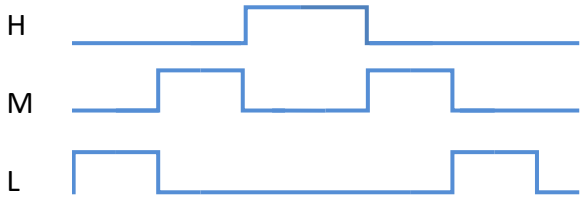
   If T_f denotes the shorter inter arrival time between two faults, then:

   $$R_i = B_i + C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k + \underbrace{\max}_{k \in hep_i} \left(\left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right).$$

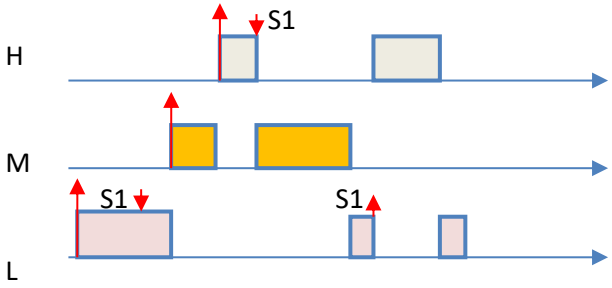
3. The time demands of the *clock handler* and that of the *context switches*:
   - In many applications the scheduler is triggered by a clock interrupt (tick scheduling). In this case the response time should be increased by the worst case time difference of the arrival and the clock tick. If the time of the arrival is not measurable, then the time between two clock ticks is the correcting value.
   - If the scheduler decides a task to run, then first the registers of the processor should be saved, after this the context of the new task should be loaded into the registers, and then comes the execution of the task. The response time should be increased by time of this „context switch". The computation time of the higher priority tasks, which pre-empt the execution of an actual task, should be increased by the time needed to perform context switching, as well.

Scheduling if the tasks are not independent: Resource Access Protocols

Except for the time-sharing systems, where the processor's capacity is shared among independent users, for most of the applications the runs of the different tasks are not completely independent. Tasks are communicating with each other, exchange data, they are waiting for results from other tasks, they use common resources, and it can happen, that higher priority tasks are blocked by runs of lower priority tasks. Let us recall the illustration of the priority based scheduling! If here task L would use such a resource, which is later also used by task H, then it might happen that task H should wait until the resource will be released.



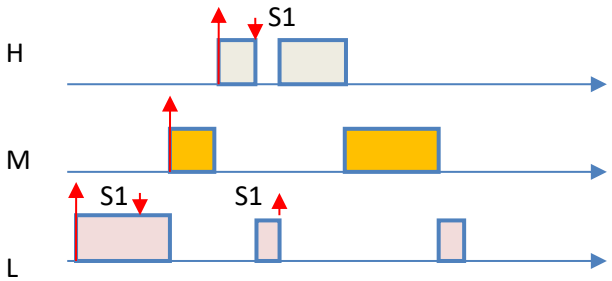
This is illustrated by the following figure:



Task L locks a common resource using semaphore S1, and starts its critical section. After the request of task M task L will be pre-empted. The execution of M is pre-empted by task H upon its arrival. Task H would like to use the common resource locked by task L. Task H should wait for unlocking semaphore S1. This type of waiting is called blocking, because lower priority task forces higher priority task to wait. To unlock semaphore S1 task L should get back the processor. This is possible only after completing task M. Thus, task H can be considerably delayed. This situation is called **priority inversion**, because seemingly the priorities of task M and H are inverted. With semaphore S1 we implement mutual exclusion: if a resource is locked, it will be unlocked after completing the critical section.

Priority Inheritance Protocol (PIP)**:**

To avoid priority inversion, task L should dynamically inherit the priority of task H upon its request to enter the critical section. Thus, task L can complete the critical section much earlier and unlock semaphore S1. The inherited priority is called dynamic priority valid only for the critical section. After unlocking semaphore S1 the static priority will be restored. The effect of this modification is indicated by the following figure:



The response time of task H will be much shorter, and the worst-case blocking time equals the duration of the critical section of task L.

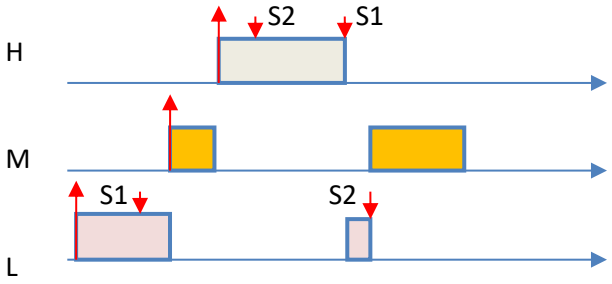
The worst-case response time will increase with the worst-case blocking time:

$$R_i = C_i + B_i + I_i = C_i + B_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

2. Scheduling (Cont.)

Deadlock avoidance

The Priority Inheritance Protocol should be extended/modified if more common resources are to be handled. This is illustrated by the following figure:



Task L by locking semaphore S1 enters a critical section. Within this critical section semaphore S2 will be also locked by task L. These two resources – with the given timing – are used by task H, as well. As task H would like to lock semaphore S1, it will be blocked. Task L inherits priority H, but trying to lock semaphore S2 it will also block. Both task H and L will wait for the other. This situation is called: deadlock. To avoid deadlock priority ceiling protocols are used.

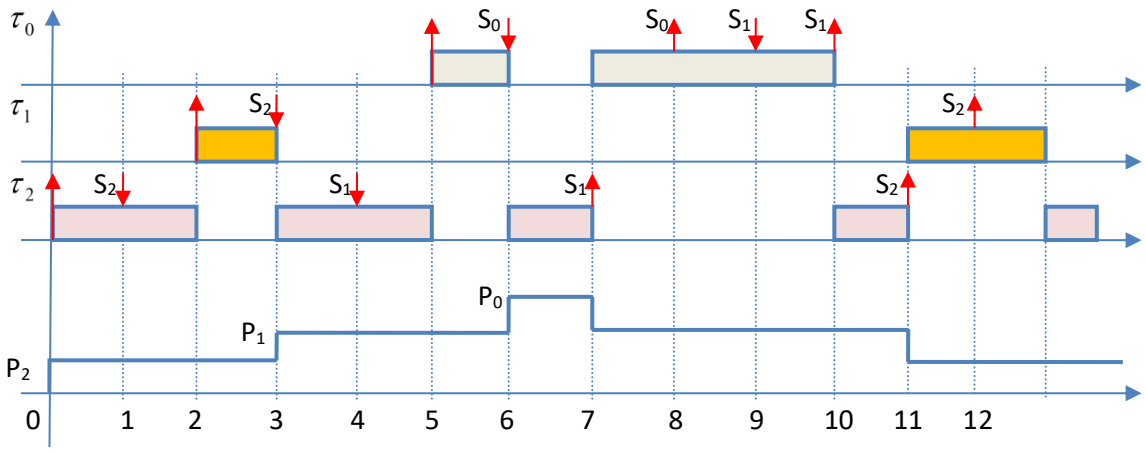
Priority Ceiling Protocol (PCP): The basic idea of this method is to extend the PIP with a rule for granting a lock request on a free semaphore. To avoid multiple blocking, this rule does not allow a task to enter a critical section if there are locked semaphores that could block it. This means that, once a task enters its first critical session, it can never be blocked by lower-priority tasks until its completion.

To realize this idea, each semaphore is assigned a priority ceiling equal to the priority of the highest-priority task that can lock it. Then, a task i can enter a critical section only if its priority is higher than all priority ceilings of the semaphores currently locked by tasks other than i.

The PCP protocol:

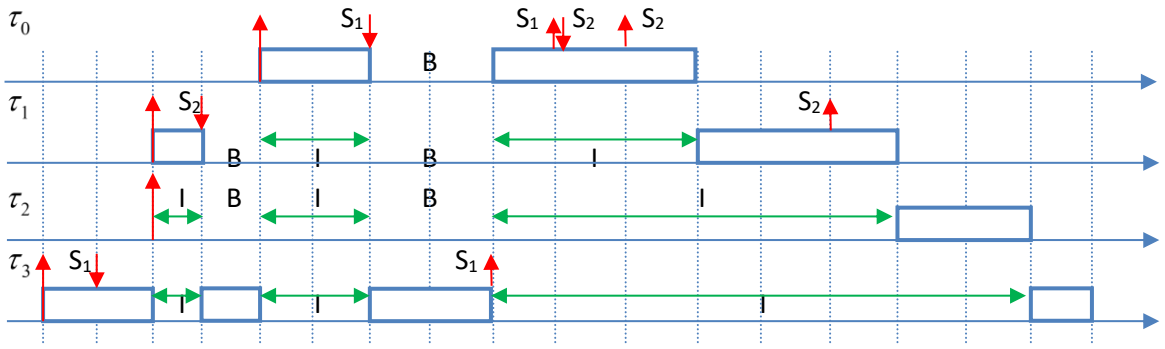
- Each semaphore S_k is assigned a priority ceiling $C(S_k)$ equal to the priority of the highest-priority task that can lock it. Note that $C(S_k)$ is a static value that can be computed offline.
- Let τ_i be the task with the highest-priority among all tasks ready to run; thus, τ_i is assigned to the processor.
- Let S^* be the semaphore with the highest-priority ceiling among all the semaphores currently locked by tasks other than τ_i, and let $C(S^*)$ be its ceiling.
- To enter a critical section guarded by a semaphore S_k, τ_i must have a priority (P_i) higher than $C(S^*)$. If $P_i \leq C(S^*)$, the lock request is denied and τ_i is said to be blocked on semaphore S^* by the task that holds the lock on S^*.
- When a task τ_i is blocked on a semaphore, it transmits its priority to the task, say τ_k, that holds that semaphore. Hence, τ_k resumes and executes the rest of its critical section with the priority of τ_i. In general, a task inherits the highest priority of the task blocked by it.
- When τ_k exits a critical section, it unlocks the semaphore and the highest-priority job, if any, blocked on that semaphore is awakened. Moreover, the active priority of τ_k is updated as follows: if no other jobs are blocked by τ_k, its priority is set to the nominal (static) priority; otherwise it is set to the highest-priority of the tasks blocked by τ_k.

Example: The tasks to be scheduled with descending priority are: τ_0, τ_1, τ_2. Their priorities: P_0, P_1 and P_2. The resources are guarded by semaphores S_0, S_1 és S_2. Their priority ceilings: $C(S_0) = P_0$, $C(S_1) = P_0$, $C(S_2) = P_1$.



Note that task τ_0 will be blocked even though the requested resource is not blocked. The reason of this blocking is that task τ_2 is within a critical section guarded by semaphore S_1 the priority of which is equal of that of τ_0.

Example: The tasks to be scheduled with descending priority are: $\tau_0, \tau_1, \tau_2, \tau_3$. Their priorities: P_0, P_1, P_2 and P_3. The resources are guarded by semaphores S_1 and S_2. Their priority ceilings: $C(S_1) = P_0$, $C(S_2) = P_0$.

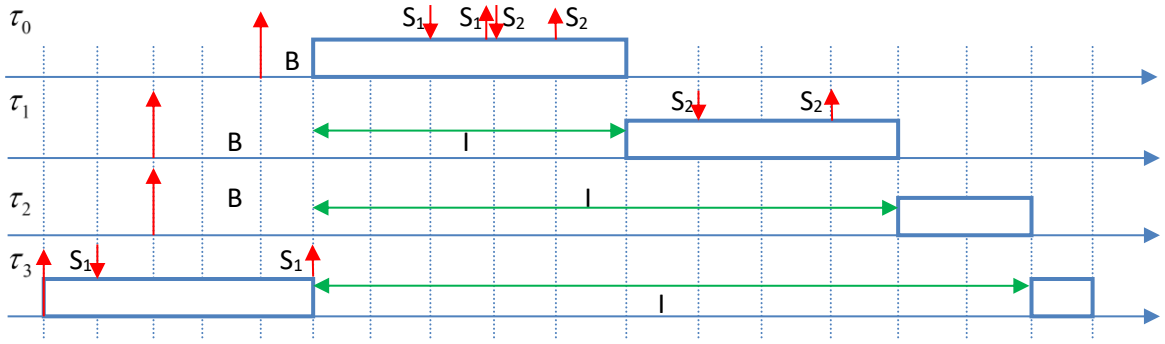


On the figure it is easy to follow the operation of the PCP protocol. "I" denotes the interference intervals, while B stands for the blocking intervals. The sum of these latter gives the effective blocking time, the worst-case value of which equals length of the critical section of task τ_3.

Properties of the PIP:

- Priority inheritance is transitive; that is, if a task τ_3 blocks a task τ_2, and τ_2 blocks task τ_1, then τ_3 inherits the priority of τ_1 via τ_2.
- If a task τ_k is pre-empted within a critical section by a task τ_i that enters another critical section, then, under the PCP, τ_k cannot inherit a priority higher than or equal to that of task τ_i until τ_i completes.
- The PCP prevents transitive blocking.
- The PCP prevents deadlocks.
- Under PCP, a task τ_i can be blocked for at most the duration of one critical section.
- The maximum blocking time B_i of task τ_i can be computed as the longest critical section among those belonging to tasks with priority lower than P_i and guarded by a semaphore with ceiling higher than or equal to P_i.

Immediate Priority Ceiling Protocol (IPCP): The essence of the protocol is that the tasks entering a critical section immediately inherit the ceiling priority of the semaphore which guards the critical section. Thus, on the figure below, task τ_3 at entering the critical section receives as dynamic priority P_0, and will operate at this priority level till the end of the critical section. The implementation of IPCP is easier than that of the PCP, and there are less task-switching, and consequently context switching. It is interesting to note that the semaphores do not need implementation because after leaving the first critical section they are and remain unlocked. It is also interesting to realize that using IPCP the response time of the highest priority task became shorter.



The name IPCP in POSIX is Priority Protect Protocol, and in Real-Time Java: Priority Ceiling Emulation.

3. Memory management

Scheduling not independent tasks we faced some problems of handling resources. Here we discuss the problems of memory management from the viewpoint of embedded systems. In the case of embedded systems, it is typically not possible to eliminate the side-effects of not completely correct resource handling time-to-time by resetting the device. We must design such systems, where the performance of the resources remains stable, degradation is not possible.

- **Static memory allocation:** If the memory is allocated statically, then it can be established at compile time exactly how each byte of RAM will be used during the running of the program. This has the advantage, for embedded systems, that the whole issue of bugs due to leaks and failures due to fragmentation simply does not exist. The global and static data is allocated in a fixed location, since it must remain valid for the life of the program. This approach prohibits the use of recursion, function pointers, or any other mechanisms that require re-entrant code. For example, and interrupt routine cannot call a function that may also be called by the main flow of execution.
- **Stack based management:** The next step up in complexity is to add a stack. Now a block of memory is required for every call of a function, and not just a single block for each function in existence. The blocks are now stored on a stack, which usually has some hardware support including special instructions in the processors instruction set.

  The stack grows and shrinks as the program executes, and for many programs it is not possible to predict, at compile time, what the worst-case stack size will be. In multitasking system, there will be one stack per task to manage, plus possibly an extra one for interrupts. Some judgement must be exercised to make sure that each stack is big enough for all its activities. It is awful shame to suffer from an untimely stack overflow, when one of the other stacks has reserve of space that it never uses. Unfortunately, most embedded system does not support any kind of virtual memory management that would allow the tasks to draw from a common pool as the need arises.

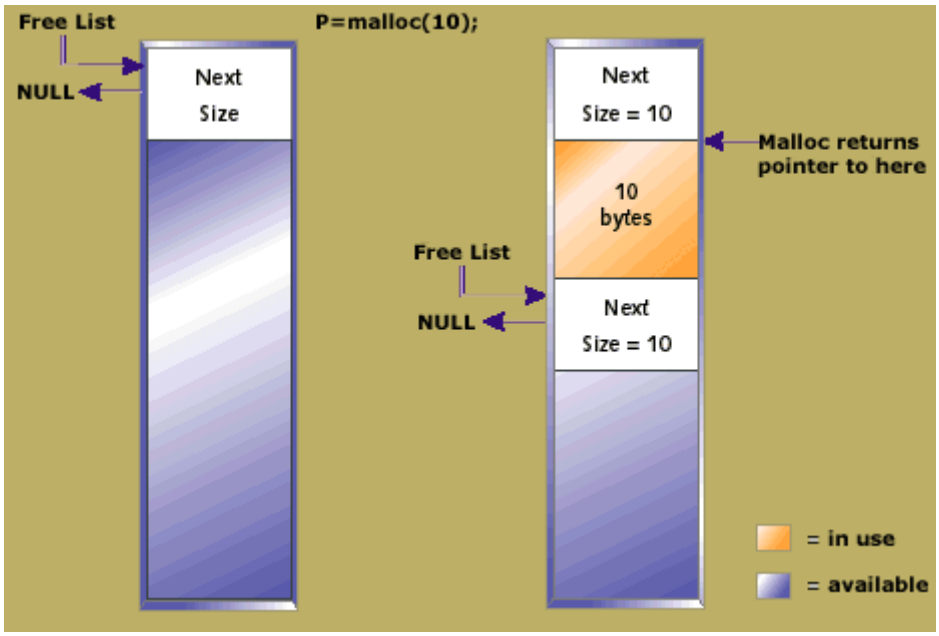
  One rule of thumb is to make each stack 50% bigger than the worst case seen during testing. One simple technique is to paint the stack space with simple pattern. As the stack grows and shrinks, it will overwrite the area with its data. At a later time, a simple loop can run through the stack's predefined area to detect the furthest extent of the stack. This technique is called watermarking. Many RTOS's support this mechanism. It's worthwhile to bind this testing with the start of a watchdog timer.

- **Heap based management:** In C programs heap management is carried out by the malloc() and free() functions. Malloc() allows the programmer to acquire a pointer to an available block of memory of a specified size. Free() allows the programmer to return a piece of memory to the heap when the application has finished it. In this way, a piece of memory that is used to store of data from a serial port at one point in time may be used to store a structure controlling a graphics window at another time. The programmer has simple interface to the heap so it is not necessary for the programmer to establish at design time which items are not going to be in use simultaneously.

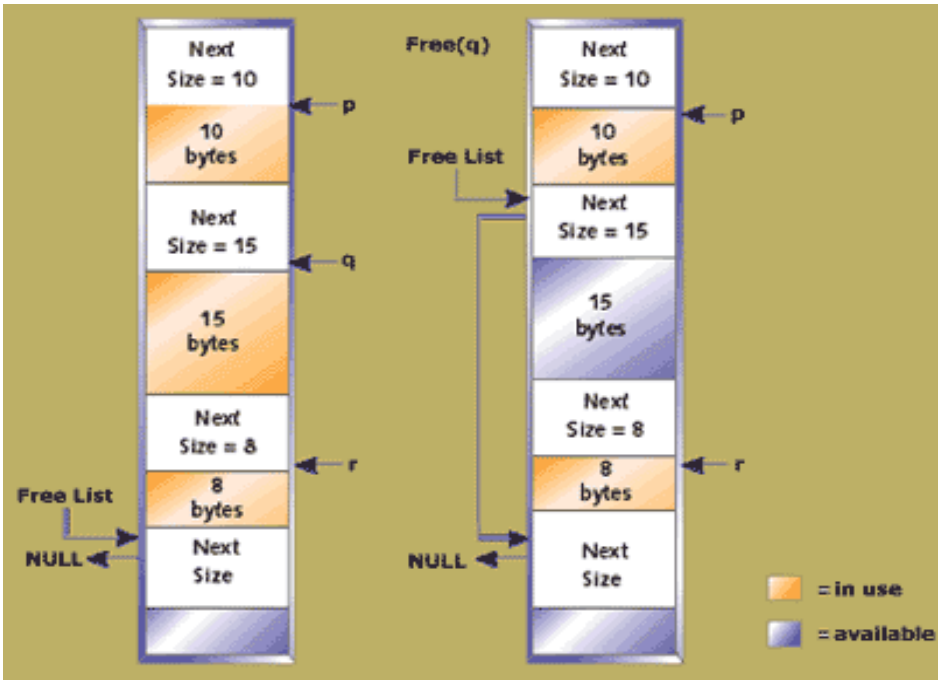
  While stack management is handled by the computer, using heap management requires care by the programmer, or many particular devious bugs can creep into the program.

  At a certain point in the code you may be unsure if a particular block is no longer needed. If you free() this piece of memory, but continue to access it (probably via a second pointer to the same memory), then your program may function perfectly, until that particular piece of memory is reallocated to another part of the program. Then two different parts of the program will proceed to write over each other's data. If you decide not to free the memory, on the grounds that it may still be in use, then you may not get another opportunity to free it, since all pointers to the block may have gone out of scope, or been reassigned to point elsewhere. The result is: *memory leakage*. In this case the program logic will not be affected, but if the piece of code that leaks memory is visited on a regular basis then the leak will tend toward infinity, and the execution time of the program increases.

  Any leak is a bug, which can be rectified by correcting the logic of the program. There is another problem called *fragmentation*, which cannot be corrected at the application program level. This is a property inherent in most applications of malloc(). It is caused by blocks of memory available being broken down into smaller pieces as many allocations and frees are performed.



The figure shows the allocation of the first 10 bytes. (These two figures are from the lecture of Niall Murphy (Panelsoft) entitled Memory Management, presented among others at the Embedded Systems Conference Europe in 2000.)



On the left-hand side of the figure the allocation of 10, 15 and 8 bytes is illustrated. On the right-hand side the freeing of the block of 15 bytes is presented.

Example*:* In a survey of a number of Unix applications it was found that 90% of allocations were covered by 6 sizes. 99.9% of allocations were covered by 141 sizes. In embedded systems this range is far smaller, since file and string handling is much rarer. It seems to advantageous to have only few sizes.

Fragmentation can also be reduced by using the appropriate policy when allocating and freeing blocks. Allocation policies include:

- Allocate (and possibly split) first block found, larges than the request (First Fit).
- Allocate the best fit after exhaustive search (Best Fit).

Free list policies include:

- Maintaining the list in order of address, to simplify merging of free blocks.
- Maintain the list in most recently used order, to match patterns of use where similar sizes are allocated and freed in bursts.

Seeing the difficulties, the conclusion is that mission critical project cannot afford these dynamic memory allocation mechanisms. Systems that need to be very reliable, but not 100% reliable, can use a heap, if appropriate testing and measurement is performed.

- **Recommendation:** limited heap functionality: static allocation: (1) malloc() is used only during initialization, and freeing is not applied. (2) It worth writing a separate program which does not have the overhead of the block headers. (E.g. salloc( ) (static allocation)). (3) Following the initialization salloc( ) is inhibited.
- **Recommendation:** dynamic allocation but fixed block size. (Partitions or pools of fixed size memory blocks.)
- **Multitasking:** While each task must have its own stack, it may or may not have its own heap, regardless of whether the heap is based on the static scheme, pools, or general purpose allocation scheme. Having more than one heap means that you have to tune the size of a number of heaps, which is a disadvantage. However one heap for many tasks must be re-entrant, which means adding locks that will slow down each allocation and deallocation.

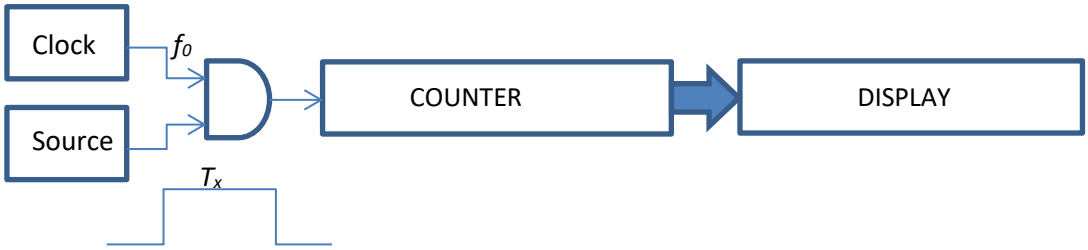
A single heap also allows one task to allocate a piece of memory which may be freed by another task**.** This is useful for passing inter-task messages. When memory is passed between tasks in this way, make sure that it is always well defined who owns the memory at each point. It is obviously important that two tasks do not both believe that they own a piece of memory at the same time leading to two calls to free the memory.

- **Libraries:** Problems: (1) Memory should be allocated by the library. (2) Freeing memory is the role of the application program. (3) We can assign static memory to the library; however, this is not suitable for re-entrant code, which is so essential to many embedded systems. (4) All these problems should be considered by the programmer of the library: possibly by offering library routines for freeing memory. (So-called Pluggable memory management).
- **Automatic garbage collection:** e.g. Java, LISP, Smalltalk offer such a service. Two basic mechanisms: (1) The pointers can be objects in their own right, which have destructors that are called when the pointer goes out of scope. In C++ this is possible with *smart pointers*. (2) To test whether a piece of memory is free a search is performed within the memory in use to find a pointer to that block. If no pointers to the block are found then the block is free. This is obviously an expensive way to check for available memory.

4. Measuring time, clocks, clock synchronization

Concept/knowledge of global clock: GPS, …
Local clocks: synchronization to the global clock, synchronization to each other → consequences..

Measuring time:
(1) *Using a single electronic counter:* We count the clock ticks during the time interval to be measured:



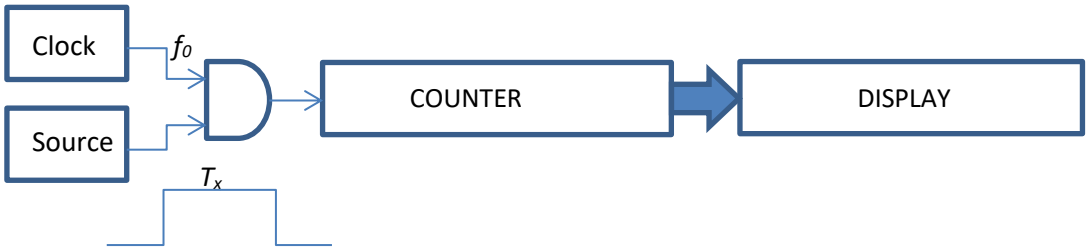
The gate time T_x generated by the source is the time duration to be measured. Before starting the measurement the counter is zeroed. The counter counts the impulses during the gate time. $T_x \cong \frac{N}{f_0}$, where N is the content of the counter, and f_o stands for the clock frequency. The approximate equality refers that N is always integer, while $T_x f_0$ is not necessarily. The difference is the quantization error. The measurement is accurate if T_x is the integer multiple of $\frac{1}{f_0}$. The worst-case relative error of the time measurement is:

$$\left|\frac{\Delta T_x}{T_x}\right| \cong \left|\frac{1}{N}\right| + \left|\frac{\Delta f_0}{f_0}\right|$$

4. Measuring time, clocks, clock synchronization (cont.)

Concept/knowledge of global clock: GPS, …
Local clocks: synchronization to the global clock, synchronization to each other → consequences.

Measuring time:
(1) *Using a single electronic counter:* We count the clock ticks during the time interval to be measured:



The gate time T_x generated by the source is the time duration to be measured. Before starting the measurement, the counter is zeroed. The counter counts the impulses during the gate time. $T_x \cong \frac{N}{f_0}$, where N is the content of the counter, and f_o stands for the clock frequency. The approximate equality refers that N is always integer, while $T_x f_0$ is not necessarily. The difference is the quantization error. The measurement is accurate if T_x is the integer multiple of $\frac{1}{f_0}$. The worst-case relative error of the time measurement is:

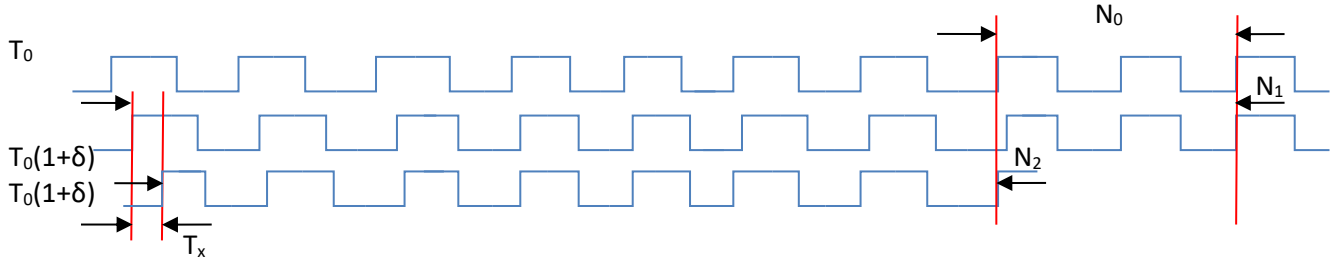
$$\left|\frac{\Delta T_x}{T_x}\right| \cong \left|\frac{1}{N}\right| + \left|\frac{\Delta f_0}{f_0}\right|$$

i.e. to get an accurate value we need a relatively high frequency clock, since otherwise the value of N will not be high enough. The above equation can be derived from the complete differential:

$$dT_x = \frac{\partial T_x}{\partial N} dN + \frac{\partial T_x}{\partial f_0} df_0 = \frac{1}{f_0} dN - \frac{N}{f_0^2} df_0,$$

which divided by $T_x = \frac{N}{f_0}$ will result in $\frac{dT_x}{T_x} = \frac{dN}{N} - \frac{df_0}{f_0}$. Obviously, N can take only integer value, therefore it can change only by integer multiple of ± 1. In principle the relative change of N and f_0 could result in a compensating effect, however the sign of the changes is not known, therefore we use the absolute value of the changes and express the worst case relative error.

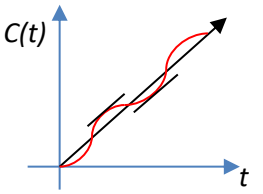
(2) *The dual vernier method:* This method applies three oscillators: the free-running reference oscillator, and two phase-startable phase-lockable oscillators (PSPLO). These latter oscillators have a slightly longer period of oscillation than the reference oscillator such that once started, they will reach coincidence with the reference oscillator some number of cycles later dependent on the initial phase. The beginning (sliding edge) of the time interval to be measured and the end (falling edge) of the time interval start the phase-startable phase-lockable oscillators having periods of $T_0(1+\delta)$.



The time from the start till the coincidence is $N_1T_0(1+\delta)$, while from the end till the coincidence it is $N_2T_0(1+\delta)$. The time between the two coincidences is N_0T_0. Based on these values

$$T_x = T_0\left[\pm N_0 + (N_1 - N_2)(1+\delta)\right],$$

where the sign before N_0 is determined by the order of the two coincidences. If T_0=5 nsec and δ=0.004, then the shortest measurable duration is 20psec. Remark: it is a challenging task to implement startable oscillators with an accuracy of quartz. Similarly, it is a great challenge to detect coincidences at these frequencies.



Clocks are the sources of the knowledge of time with a given accuracy:

The source of the knowledge of time is called clock. Clock *k* is a function $C_k(t)$ of time, which maps real time to the time at clock *k*.

Reference clock or standard clock: if $C_k(t) = t;\ \forall t$.

The *correctness* of the clock, at any point of time, depends on the difference between its readout and that of the standard clock, at that point of time. The quality of this correctness is one characteristic of the quality of knowledge of time t localities for which this clock is the source of this knowledge.

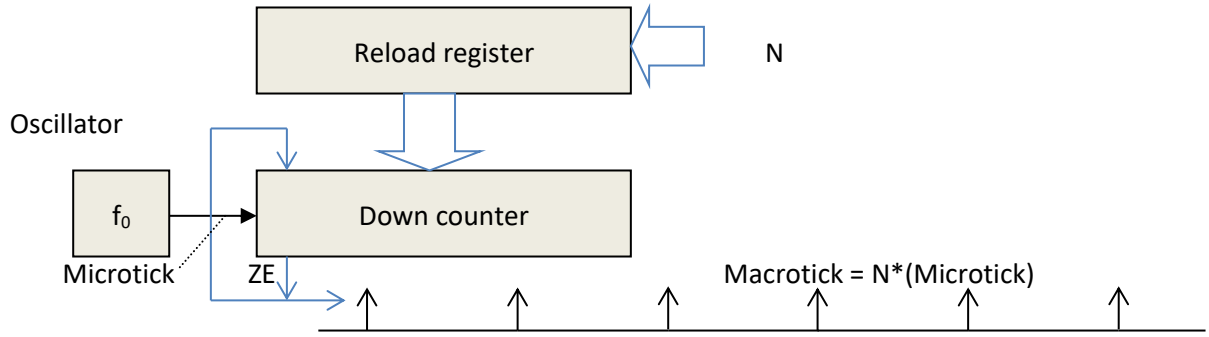
Correct clock: Clock *k* is correct in t_0, if $C_k(t_0) = t_0$.

Another characteristic of the quality of knowledge of time at the localities for which some clock is the source of this knowledge is the quality of this clock's rate.

Accurate clock: Clock *k* is is accurate in t_0, if $\frac{\partial C_k(t)}{\partial t} = 1;\ t = t_0$.

If a clock is inaccurate at some point of time, we say that the clock *drifts* at that point of time.

Physical clock: Oscillator + counter, its granularity $g = \frac{1}{f}$ is the time between two microticks.



The figure above shows the structure of a digital clock: the output of the high-frequency oscillator is divided by a down counter, which outputs an impulse at its ZE output as it reaches zero. This gives a macrotick, and simultaneously loads the content of the reload register into the counter. The rate of the macrotick is controlled by *N*.

The physical reference clock: denoted by *C*, its granularity is g^C. E.g.: 10^{15} microticks/sec $\rightarrow$ $g^C = 10^{-15}$ sec. The frequency of this reference clock is in perfect agreement with the international standard of time.

Timestamp: $C(e)$ is the absolute timestamp of the event *e*.

The *duration* between two events is measured by counting the microticks of the reference clock that occur in the interval between these two events. The *granularity* g^k of a given clock k is given by the nominal number n^k of microticks of the reference clock C between two microticks of this clock k.

Clock drift: The drift of a physical clock k between microtick i and microtick $i+1$ is the frequency ration between this clock k and the reference clock, at the instant of microtick i. The drift is determined by

measuring the duration of a granule of clock k with the reference clock C and dividing it by the nominal number of n^k of reference clock microticks in a granule:

$$drift_i^k = \frac{C\left(microtick_{i+1}^k\right) - C\left(microtick_i^k\right)}{n^k}.$$

A perfect clock has a drift of 1. Therefore, the so-called *drift rate* ρ_i^k is also introduced:

$$\rho_i^k = \left|drift_i^k - 1\right| = \left|\frac{C\left(microtick_{i+1}^k\right) - C\left(microtick_i^k\right)}{n^k} - 1\right|.$$

A perfect clock will have a drift rate of 0. Real clocks have a varying drift rate that is influenced by environmental conditions, e.g., a change in the ambient temperature, a change in the voltage level that is applied to a crystal oscillator, or aging of the crystal. Within specified environmental parameters, the drift rate of an oscillator is bounded by the maximum drift rate ρ_{max}^k, which is documented in the data sheet of the oscillator. Typical maximum drift rates ρ_{max}^k are in the range of 10^{-2} to $10^{-7}\ sec/sec$, or better. Because every clock has a non-zero drift rate, free-running clocks, i.e., clocks that are never resynchronized, leave any bounded relative time interval after a finite time, even if they are fully synchronized at start-up.

Example: During the Gulf war on February 25, 1991 a Patriot missile defence system failed to intercept an incoming scud rocket. The clock drift over 100-hour period (which resulted in a delay of 0.3433 sec causing a tracking error of 678 meters) was blamed for the Patriot missing the scud missile that hit an American military barrack in Dhahran, killing 29 and injuring 97.
The explanation of the error mentions that originally the Patriot systems were designed against muck slower devices, and they were further developed against scud systems only during the Gulf war. The error causing this tragedy was identified already during the early days of February, and the modified software was released on 16th of February, but unfortunately it was not forwarded to the system activated on February 25.

Offset: The *offset* at microtick i between two clocks j and k with the same granularity is defined as:

$$offset_i^{jk} = \left|C\left(microtick_i^j\right) - C\left(microtick_i^k\right)\right|.$$

The offset denotes the time difference between the respective microticks of the two clocks, measured in the number of microticks of the reference clock.

Precision: Given an ensemble of clocks $(1, 2, \ldots, n)$, the maximum offset between any two clocks of the ensemble

$$\Pi_i = \max_{\nabla 1 \le j,k \le n} \left\{offset_i^{jk}\right\}$$

is called the *precision* Π_i of the ensemble at microtick i. The maximum of Π_i over an interval of interest is called the *precision* Π of the ensemble. The process of mutual resynchronization of an ensemble of clocks to maintain a bounded precision is called *internal synchronization.*

Accuracy: the offset of a clock k with respect to the reference clock C at microtick i is called $accuracy_i^k$:

$$accuracy_i^k = offset_i^{k,ref} = \left|C\left(microtick_i^{jk}\right) - C\left(microtick_i^{ref}\right)\right|.$$

The maximum offset of all microticks *i* that are of interest is called the $accuracy^k$ of clock *k*. To keep the clock within a bounded interval of the reference clock, it must be periodically resynchronized with the reference clock. This process of resynchronization of a clock with the reference clock is called external synchronization.

Example: If all clocks of an ensemble are externally synchronized with accuracy *A*, then the ensemble is also internally synchronized with a precision of at most 2*A*. The converse is not true. An ensemble of internally synchronized clocks will drift from the external time if the clocks are never resynchronized with the external time base.

Time measurement using more clocks

Global time: In a distributed system all nodes have their own clock C^k that ticks with granularity g^k. Assume that all the clocks are internally synchronized with a precision Π, i.e., for any two clocks j, k and all microticks i

$$\left|C\left(microtick_i^j\right) - C\left(microtick_i^k\right)\right| < \Pi.$$

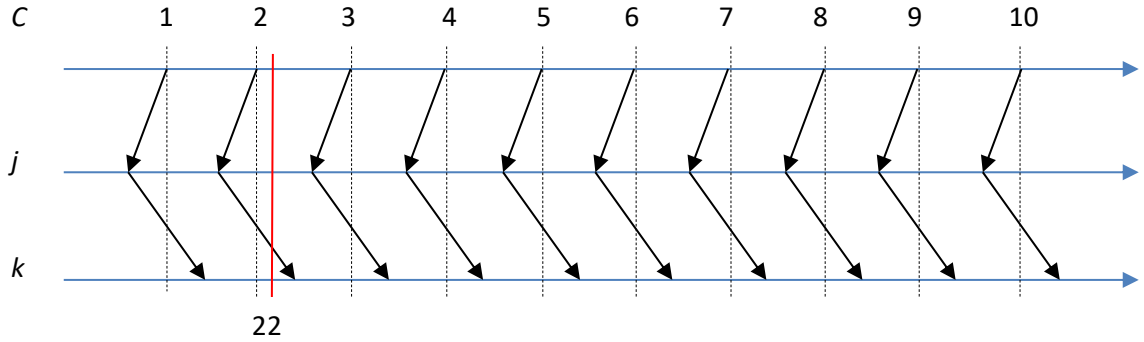
It is then possible to select a subset of the microticks of each local clock k for the generation of the local implementation of a global notion of time. We call such a selected local microtick i a *macrotick* (or *tick*) of the global time.

The global time is of the same accuracy as the reference clock, however its granularity is worse: it produces macroticks. These macroticks can be properly used, if $g > \Pi$, i.e. the synchronisation error is smaller than the resolution. The difference of the timestamps of an event *e*, at nodes j and k, differ at most one tick.

$$\left|C_j(e) - C_k(e)\right| \leq 1.$$

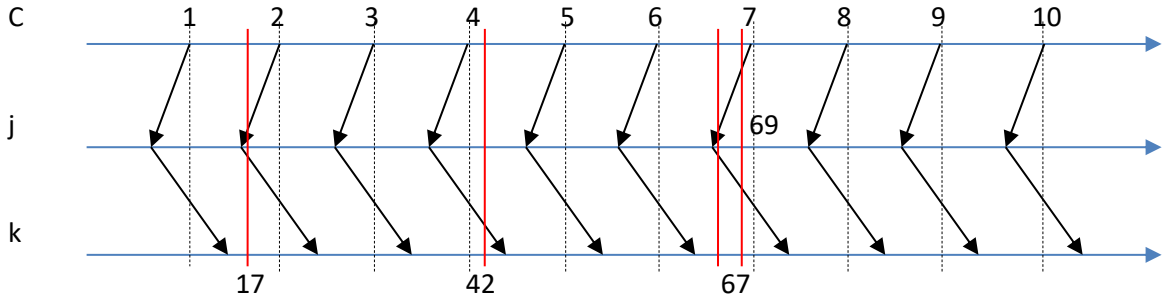
This is the maximum we can achieve, since it is always possible that first clock *j* ticks, event *e* occurs, clock *k* ticks. In this case the event will be timestamped with one tick difference.

Example: (every macrotick corresponds to 10 microticks):



At microtick *e*:22 clock *j* shows 2, while clock *k* shows 1.

One tick difference: what does it mean?



At microtick *e*:17 *j*:2, *k*:1. At microtick *e*:42 *j*:4, *k*:3. If the time difference of *e*:42 and *e*:17 is expressed by the difference of the C_k and C_j values, then the measurement will give 1 expressed in global time, while the real difference is 25 microticks.

At microtick *e*:67 *j*:7, *k*:6. At microtick *e*:69 *j*:7, *k*:6. If the time difference of events *e*:69 and *e*:67 are measured by the difference of C_j and C_k, then it gives 1, the actual difference is only 2 microticks.

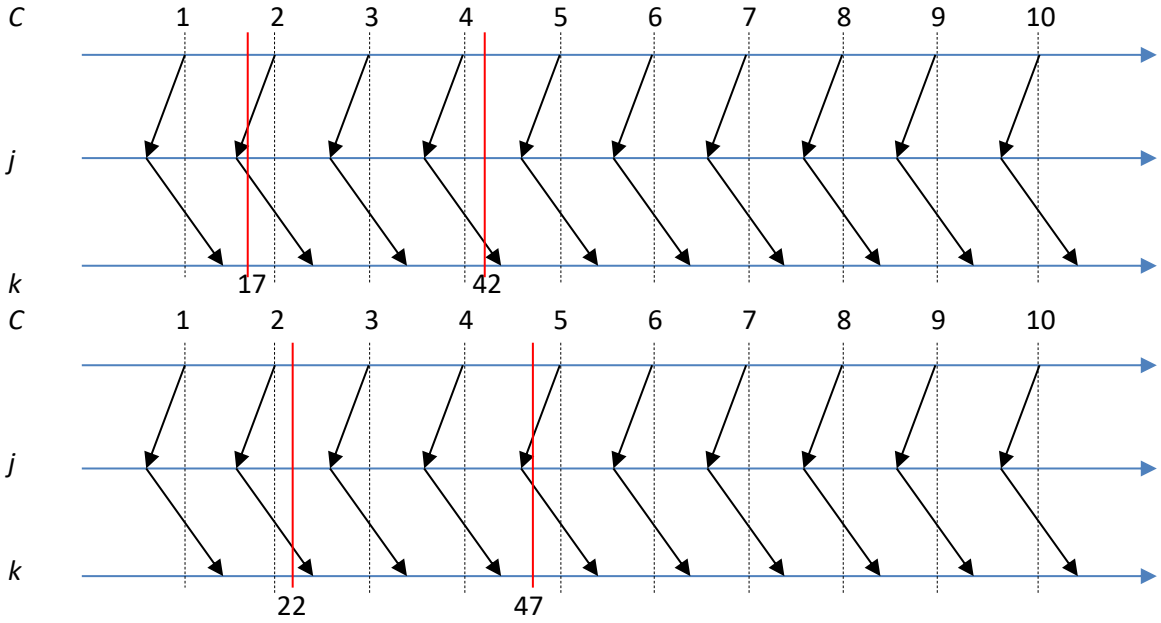
Problem: The order of time is not decidable: at microtick *e*:67 j:7, at microtick *e*:69 k:6!

An interval is delimited by two events: the start event and the terminating event. The measurement of these two events relative to each other can be affected by the synchronization error and the digitalization error. The sum of these two errors is less than $2g$ because the local implementation of the global time satisfies the condition $g > \Pi$. Thus, if the difference is two macroticks then the order of time is decidable.

Measurement of time interval:

$$(d_{observed} - 2g) < d_{true} < (d_{observed} + 2g),$$

where d_{true} is the true duration of the interval $d_{observed}$ is the observed difference of the start event and the terminating event. See illustration:



On the upper figure if we measure the difference of microticks *e*:42 and *e*:17 as the difference of clocks C_k and C_j, then the result in macroticks is 1, in contrast to the fact that the true difference in microticks is 25. On the lower figure the time difference of microticks *e*:47 and *e*:22 is measured by the difference of clocks C_j and C_k: the result in macroticks is 4in contrast to the true difference in microticks is 25.

Types of clock systems:

- *Central clock systems*:
  - One accurate clock provides the time knowledge to the whole system. The existence of other clocks in the system is „ignored” if there is no detectable failure of the central clock.
  - For fault tolerance a standby redundancy for the central clock is used.
  - The method is accurate (within ns to ms) and expensive.
  - This method needs special purpose integrated into the processor. The central clock sets this hardware to the proper value and any executing process can read it.
  - The communication cost of this category is very low: Only one message is required for synchronizing a clock at any site. In a broadcasting environment, one message is required for synchronizing a group of sites.
  - An example of this category is the GPS (Global Positioning System), which uses 4 broadcasting satellites and achieves a clock synchronization with correctness within a few nanoseconds.
- C*entrally controlled clock systems*: In this category we distinguish between two types of nodes: master nodes and slave nodes.
  - A nominated master clock polls slave clocks.
  - Clock differences are measured, and the master dictates corrections to the slaves.
  - If the master clock fails, an election of a new master (from the slaves) is initiated.

- Transmission times and delays are estimated, since they affect significantly the clock difference measured.
- Communication costs are higher than in the previous case.

- *Distributed clock systems*
  - All the nodes are homogeneous and each runs the same algorithm.
  - Each node updates its own clock after receiving the time from other clocks and after estimating their correctness.
  - The fault tolerance is protocol based. If a node fails, the other nodes are not affected; they only detect the failure and ignore the failed node thereafter.
  - Generally, relatively heavy communication traffic is involved in this category, especially when robustness in the presence of malicious faults is required.

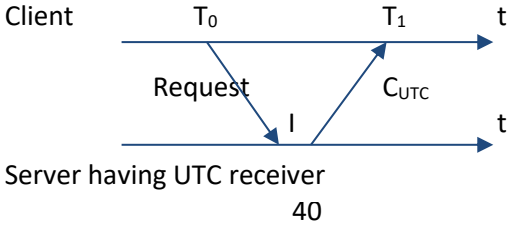
Time standards

Two are used in distributed real-time systems:

- *International Atomic Time (Temps Atomique Internationale, TAI)*: TAI defines the second as the duration of 9 192 631 770 periods of the radiation of a specified transition of the Cesium atom 133. The intention was to define the duration of the TAI second so that it agrees with the second derived from astronomical observations. TAI is a *chronoscopic* timescale, i.e., a timescale without any discontinuities.
- *Universal Time Coordinated (UTC)*: UTC is a time standard that has been derived from astronomical observations of the rotation of the earth relative to the sun.
  - The UTC time standard was introduced in 1972, replacing the Greenwich Mean Time (GMT) as an international time standard. The duration of the second conforms to the TAI standard.
  - Because the rotation of the earth is not smooth, but slightly irregular, occasionally a leap second is inserted into the UTC to maintain synchrony between the UTC and astronomical phenomena, like day and night. Because of this leap second, the UTC is not a chronoscopic time scale, i.e., it is not free of discontinuities.
  - It was agreed that on January 1, 1958 at midnight, both the UTC and the TAI had the same value. Since then the UTC hs deviated from TAI about 30 seconds.
  - The US National Institute of Standards and Technology: NIST provides a short-wave radio broadcast of continuous frequency and time signal at frequencies 2.5, 5., 10, 15 és 20 MHz. The accuracy of these time signals is ± 1 msec, but due to random atmospheric disturbances at the receiver this accuracy will be about ± 10 msec. (In case of geostationary satellites: ± 0.5 msec.)
- *Time Format*: Network Time Protocol (NTP). This time format with a length of 8 bytes contains 2 fields: a 4-byte full second field, where the seconds are represented according to UTC, and a fraction of a second field, where the fraction of a second is represented as a binary fraction with a resolution of about 232 picoseconds. On January 1, 1972, at midnight the NTP clock was set to 2 272 060 800.0 seconds, i.e., the number of seconds since January 1, 1900 at 00:00h. This ranges up to the year 2036, i.e., it has 136 years wrap around cycle.

Synchronizing clocks: Cristian algorithm

Synchronization is initiated by the client at time T_0 by requesting a server, which has a UTC receiver. After the arrival of the request and interrupt routine is executed and the UTC radio is requested. Finally, the value of the UTC clock is send to the client. The message arrives at T_1. The received clock value must be corrected by the time needed for communication. If the communication requires nearly the same amount of time in both directions, then a good approximation of this correction is: $\sim \frac{T_1 - T_0 - I}{2}$.



Comment: It might cause problems if C_{UTC}+correction < T_1, i.e., the clock of the client is to be reset to an earlier time. If the client clock is just timestamping subsequent events, it might happen that due to the reset a later event receives earlier time, i.e., seemingly changes the order. If such a situation is a real danger, then it is not allowed to reset the clock, only the slowing of the clock is permitted until its run will be synchronous with the UTC clock.

Synchronizing clocks: Master-slave algorithms

1. The master clock *i* initiates synchronization at T_1. The error of the timestamp is e_1. ($T_1 = C_i(T_1) + e_1$). The slave is node *j*. The message sent in T_1 will travel for μ_i^j time before it arrives to *j* at time T_2. The timestamp is $C_j(T_2)$, and $T2 = C_j(T_2) + e_2$.

2. The slave computes the difference:

$$\boxed{d_1 = C_j(T_2) - C_i(T_1)}$$

Comparing the times of sending and receiving the message:

$$C_i(T_1) + e_1 + \mu_i^j = C_j(T_2) + e_2 \rightarrow d_1 = C_j(T_2) - C_i(T_1) = \mu_i^j + (e_1 - e_2)$$

If ε_j denotes the mean of the difference of the slave clock and the master clock, then

$$\varepsilon_j = (e_1 - e_2) + E_j^1,$$

where E_j^1 represent noise. ($(e_1 - e_2)$ is difference of actual values.) Replacing this difference by the mean value + noise:

$$d_1 = C_j(T_2) - C_i(T_1) = \mu_i^j + (e_1 - e_2) = \mu_i^j + \varepsilon_j - E_j^1$$

3. The slave sends its clock value at $T_3 = C_j(T_3) + e_3$ to the master. This message arrives at $T_4 = C_i(T_4) + e_4$ following a travel of μ_j^i duration. The master computes the

$$\boxed{d_2 = C_i(T_4) - C_j(T_3)}$$

difference. Again, comparing the actual times of the events:

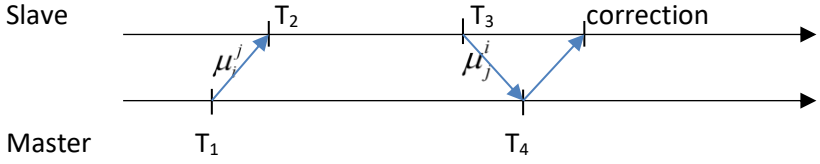
$$C_j(T_3) + e_3 + \mu_j^i = C_i(T_4) + e_4 \rightarrow d_2 = C_i(T_4) - C_j(T_3) = \mu_j^i + (e_3 - e_4).$$

The mean difference of the slave and the master clocks ε_j can be expressed by: $-\varepsilon_j = (e_3 - e_4) + E_j^2$ where E_j^2 represents noise. Thus, $d_2 = \mu_j^i - \varepsilon_j - E_j^2$.

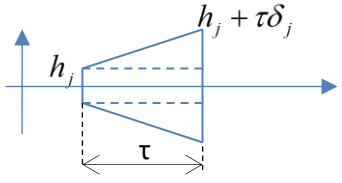
The half of the difference of $d_1 - d_2$ can be used to give the necessary correction value at the slave node:

$$\boxed{(d_1 - d_2)/2 = \varepsilon_j + (\mu_i^j - \mu_j^i)/2 - (E_j^1 - E_j^2)/2 = \varepsilon_j + h_j},$$

where h_j stands for errors coming from the difference of the communication times and the quantization effects. The random components can be reduced by averaging.



After the synchronization the remaining error will increase due to the drift. See the figure below:



The figure illustrates that due to the remaining error and the drift, after a time duration of τ, the worst-case time distance of the clocks will be: $2\left(\tau \underbrace{\max}_{j}\delta_j + \underbrace{\max}_{j}|h_j|\right)$.

Example: Tempo algorithms: master-slave synchronization in the distributed Berkeley Unix.

Comment: (1) The accuracy of the correction has improved after repeating the measurement several times, and averaging the results. (2) If the details of the communication between master and slave are known (e.g. in LAN environment), the correction can be further refined. (3) If the requirement is the synchronization of n processors, and every slave is requested p times, then the communication demand of the master-slave synchronization can be characterized by $(2p+1)n$ in every τ period.

Master side:

The basic algorithm is repeated N times:

for k=1 to N

do

Initialization:

do

$T_A \leftarrow C_i(now)$

$\forall j \neq i$ Send T_A to j

endo

Processing data received from the slaves:

$\forall j \neq i$:

do

$d_2^j \leftarrow C_i(now) - T_B$

$\Delta_j(k) \leftarrow \left(d_1^j - d_2^j\right)/2$

endo

endo ; *N differences are available for* $\forall j$*:*

$\forall j \neq i$:

do

$\Delta_j = (1/N)\sum_{k=1}^{N}\Delta_j(k)$

Send (Δ_j) to j

endo

Slave side:

do

$d_1^j \leftarrow C_j(now) - T_A$

$T_B \leftarrow C_j(now)$

Send (T_B, d_1^j) to i

endo

do

$C_j(t) \leftarrow C_j(t) - \Delta_j$

end

4. Measuring time, clocks, clock synchronization (cont.)

Clock synchronization: Distributed clock algorithms

The major advantage of the distributed approach is the higher degree of fault tolerance it achieves. This achievement increases the cost mainly in communication rather than in special hardware. The load imposed on the communication network by the distributed approach is therefore expected to be higher than that of the MS approach.

In the distributed clock systems all the time servers use uniform approach with the following characteristics:
- Each node polls the rest of the clocks or a subset of them.
- Each applies a specific algorithm to the responses of the poll.
- Each node updates the local clock accordingly.

Some examples:

I. A Fundamental Ordering Approach

Let us consider an ordering approach based on message timestamping with the following properties:
- The accuracy of clock *i* is bounded by a drift rate δ: $\forall t$: $\left|1 - \frac{d}{dt} C_i(t)\right| < \delta_i \ll 1$.
- The communication graph of the algorithm is closely connected (every vertex/node can send synchronization messages to the rest of the vertices/nodes) with a diameter *d* (minimum number of hopes).
- The network imposes an unpredictable (yet bounded) message delay *D*. In other words, $\mu < D < \eta$ holds, where μ and η are the lower and upper bounds on *D*.

Each clock implements the following algorithm:

- On every local clock event occurrence, increment the local clock $C_i(t) \leftarrow C_i(t) + 1$.
- Each node with a clock sends messages to the others at least every τ seconds. Each message includes its timestamp T_m.
- Upon reception of an external T_m, the receiver sets its clock $C_i(t) \leftarrow max(C_i(t), T_m + \mu)$.

The communication cost of one update of the whole network is $n(n-1)$ messages. The correctness of each clock due to this synchronization algorithm is $\forall i: \forall j: \left|C_i(t) - C_j(t)\right| < d(2\delta\tau + \eta)$ for all *t*. This algorithm achieves only the ordering goal, bounding clock differences between sites. The algorithm results in updates according to the fastest clock in the system, and not necessarily the most accurate one.

II. Minimize Maximum Error

Every clock *i* "knows" it is correct within the interval: $[C_i(t) - E_i(t), C_i(t) + E_i(t)]$ where $E_i(t)$ is a bound on the error of clock *i*. The error interval is constructed from the following contributors:

- The error that comes into effect right on the clock reset time (ρ_i), as discretization and other constant errors (ε_i).
- The delay from the time this clock *i* is read until another clock *j* uses the readout for its update (μ_i^j).
- The degradation of time-counting that develops between consecutive resets (δ_i).

The algorithm consists of two rules: a response rule and a synchronizer rule. A request is transmitted by the synchronizer rule at node *j* activates *i*'s response. In this response, node *i* first updates its bound on the error, $E_i(t)$. It then replies its clock value $C_i(t)$ and the above bound on that clock's error. This message expresses a time interval within which the clock is correct.

The synchronizer rule is periodic, performed at least every τ time units. Its first step is a request for responses which it sends to the rest of the nodes. Then, for each of these nodes, it performs a response reception and a conditional clock reset. Two conditions must hold for a reset:

1. The interval $[C_i(t) - E_i(t), C_i(t) + E_i(t)]$ that expresses the local knowledge of time must be consistent with the incoming interval $\left[C_j(t) - E_j(t), C_j(t) + E_j(t)\right]$. The consistency requires a nonempty intersection of these two intervals.
2. The error of the response, $E_j(t)$, plus the error of the response delay, $(1 + \delta_i)\mu_j^i$ generate an error smaller than the local one.

If these two conditions hold, the node can reset its clock and enhance its knowledge of time. The reset involves three parameters. The local clock $C_i(t)$ is set to the value of the response clock. The error at local clock reset, ε_i, is set to the value of the response error and the delay combined. The time-of-reset record, ρ_i, is also set to the value of the response clock. If on the other hand one of these conditions does not hold, the algorithm ignores the response. The algorithm's two rules are given below:

Upon receiving a time Request from $j \neq i$:

do

$E_i(t) \leftarrow \varepsilon_i + (C_i(t) - \rho_i)\delta_i$

Send $(C_i(t), E_i(t))$ to *j*.

endo

} Rule#1 (from the viewpoint of clock *i*).

At least once every τ time units:

$\forall j \neq i$: Request$(C_j(t), E_j(t))$;

for $j \neq i$ do begin

Receive$(C_j(t), E_j(t))$;

if $(C_j(t), E_j(t))$ is consistent with $(C_i(t), E_i(t))$

then if $E_j(t) + (1+\delta_i)\mu_j^i \leq E_i(t)$

then begin

$C_i(t) \leftarrow C_j(t)$

$\varepsilon_i \leftarrow E_j(t) + (1+\delta_i)\mu_j^i$

$\rho_i \leftarrow C_j(t)$

end

else ignore it

end

endo

} Rule#2

III. Intersection of time intervals

This algorithm also consists of two rules: a response rule and a synchronizer rule. The response rule is identical to the response of the previous algorithm. The synchronizer rule here is also periodic, performed at least every τ time units.

The first step of the synchronizer rule is a request for responses, which the algorithm sends to the rest of the nodes. The similarity to the previous algorithm ends here. The second step of this algorithm is to receive all the responses. Each response interval has a left boundary $L_j(t)$ and a right boundary $R_j(t)$ and the algorithm calculates both of them. Then the algorithm selects the highest left boundary in the responses, α, and the lowest right boundary, β. If the responses are consistent, there must be a nonempty intersection of them all, and thus, $\alpha < \beta$. Otherwise the responses are considered inconsistent and therefore ignored. If they are consistent, we can conclude that the real-time clock is within the interval $[\alpha, \beta]$. Therefore the algorithm sets its error to equal half this interval and the local clock to equal the interval's midpoint.

The first rule is exactly the same as in the previous algorithm. The continuation:

At least once every τ time units:

$\forall j \neq i$: Request$\left(C_j(t), E_j(t)\right)$;

$\forall j \neq i$: Receive$\left(C_j(t), E_j(t)\right)$;

$\forall j \neq i$: $L_j(t) \leftarrow \left(C_j(t) - E_j(t)\right)$; *the left boundary of j*

$\forall j \neq i$: $R_j(t) \leftarrow \left(C_j(t) + E_j(t)\right) + \left(1 + \delta_i\right)\mu_j^i$; *the right boundary of j*

$\alpha \leftarrow \max\left(L_j\right)$; $\beta \leftarrow \min\left(R_j\right)$

if $\alpha < \beta$

then

$$\varepsilon_i \leftarrow \frac{1}{2}(\beta - \alpha);$$

$$C_i(t) \leftarrow \frac{1}{2}(\alpha + \beta);$$

$$\rho_i \leftarrow \frac{1}{2}(\alpha + \beta)$$

end

else ignore them all

endo

Comments:

(1) The intersection algorithm is superior in its accuracy, however, less robust. It may ignore the responses of all the participants because of an erroneous response from one participant.

(2) The communication demand of the distributed algorithm in case of *n* clocks is $2n(n-1)$ in every τ time units.

(3) The distributed algorithms require good knowledge of time.

Jitter of the synchronization message:

Jitter: $d_{\max} - d_{\min}$

- at the application software level: $500\mu s \ldots 5ms$
- in the kernel of the operating system: $10\mu s \ldots 100\mu s$
- int he hardware of the communication controller: $< 10\mu s$.

The important role of the latency jitter ε for internal synchronization is emphasized by an impossibility result: It is not possible to internally synchronize the clocks of an ensemble consisting of *N* nodes to a better precision than

$$\Pi = \varepsilon\left(1 - \frac{1}{N}\right).$$

***IV. Fault-Tolerant-Average (FTA) algorithm*:**

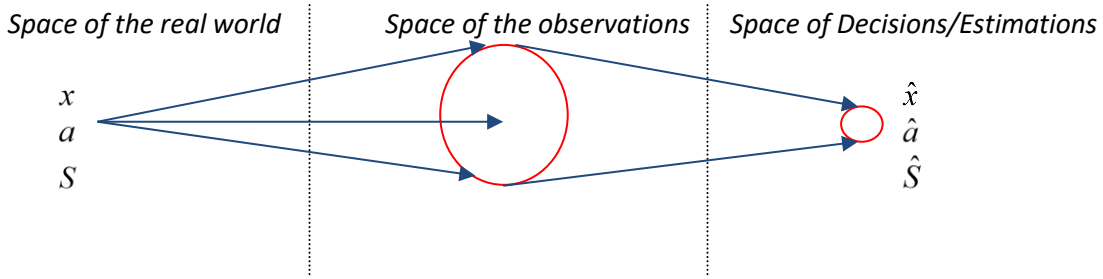
In a system with *N* nodes *k* Byzantine faults should be tolerated. The FTA algorithm is a one-round algorithm that works with inconsistent information, and bounds the error introduced by inconsistency. At every node, the *N* measured time differences between the node's clock and the clocks of all other nodes are collected (the node considers itself a member of the ensemble with time difference zero). These time differences are sorted by size. Then the k largest and the k smallest differences are removed (if the erroneous time value is either larger or smaller than the rest). The remaining $N - 2k$ time differences are, by definition, within the precision window. The average of these remaining time differences is the correction term for the node's clock.

5. Quantities and variables in real-time systems (cont.)

Modelling of the recipient environment:

Measurement process: part of the cognition process, in which the a priori knowledge is improved and extended. The figure below helps the interpretation. While taking measurement, we try to grasp the different phenomena of the real world. This is made preferably by quantities which show stability. Obviously, such quantities are results of abstractions. The following quantities/features play key role:

- *state variables* (***x***), the changes of which follow energy processes (voltage, pressure, temperature, speed, etc.) due to interactions;
- *parameters* (***a***), which characterize the strength of the interactions; and
- *structures* (***S***), which describe the relations of the system components.

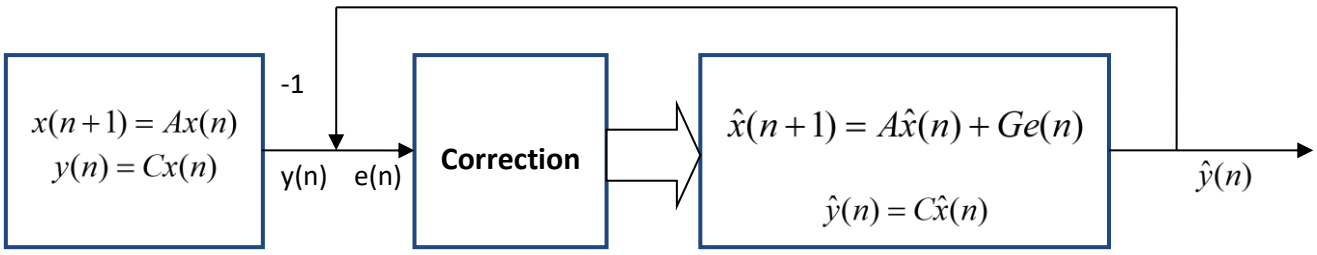


The *Space of the real world* is such an abstraction, where the values of the investigated features correspond to one point of the space. The coordinates of this points are unknown before the measurement. With the measurement we try to determine these coordinates. It is well known, that due to measurement errors, only an estimate of the measurand can be provided. Further difficulty, that there is no direct access to the quantity to be measured, only some kind of indirect mapping is possible. This mapping is called observation. The path between the quantity to be measured and the observation is called measuring channel.

Observation in case of deterministic channel: the illustrative example below presents a discrete observer. The observed reality is described by a discrete model, and it is supposed to be an autonomous system. The state equations and the observation equation describing the reality and the observation:

$$x(n+1) = Ax(n), \tag{1}$$

$$y(n) = Cx(n), \tag{2}$$



where the state variable $x(n)$ is of N dimension, the state-transition matrix is of $N*N$ dimension, the observation vector $y(n)$ is of $M \leq N$ dimension, and finally the observation matrix C is of $M*N$ dimension. Our aim is the estimation of the state variable $x(n)$. The tool of this estimation is the observer mechanism, which tries to produce a copy of the reality. This performed by a computer program capable to follow the reality as a result of a correction/training/adaptation process. The result of this process is the estimation of the value to be measured. After convergence, this estimator $\hat{x}(n)$ can be read from the observer. The state and the observation equations of the observer are:

$$\hat{x}(n+1) = A\hat{x}(n) + Ge(n), \tag{3}$$

$$\hat{y}(n) = C\hat{x}(n), \tag{4}$$

where correction matrix G is of $N*M$ dimension, $e(n) = y(n) - \hat{y}(n)$. Matrix G is to be designed in such a way that $\hat{x}(n) \to x(n)$. The difference of (1) and (3):

$$x(n+1) - \hat{x}(n+1) = Ax(n) - A\hat{x}(n) - Ge(n) = (A - GC)(x(n) - \hat{x}(n)). \tag{5}$$

Introducing notations: $\varepsilon(n+1) = x(n+1) - \hat{x}(n+1)$, and $F = A - GC$, the state transition matrix of the so-called error system is:

$$\varepsilon(n+1) = F\varepsilon(n). \tag{6}$$

The correction matrix G is designed to result in $\varepsilon(n) \xrightarrow[n\to\infty]{} 0$, possibly with $\|\varepsilon(n+1)\| < \|\varepsilon(n)\|$, for $\forall$ n, i.e. matrix F reduces the size of vector, i.e. it is „contractive".

Comment:

Obviously it is not a necessary condition to decrease the state error in every step: only the stability of the error system is required, i.e. the convergence of the error to zero for zero input. This property can be interpreted also in such a way that the internal energy of the error system is dissipated. If this is the case in every step then the decrease of the size of the error vector will be a monotonic process.

Special cases:

1. $F = A - GC = 0$. In this case $G = AC^{-1}$. This is possible if C is a square matrix, i.e. the observation has as many components as the state vector itself. In this case the observer, and the copy of the system investigated can follow the system without iteration, in one step.
2. $F^N = (A - GC)^N = 0$. In this case the error system converges in N steps:

$$x(N) - \hat{x}(N) = (A - GC)^N (x(0) - \hat{x}(0)) = 0 \tag{7}$$

   The matrices with the property $F^N = 0$ can be characterized by the fact that all they eigenvalues are zero. Systems having state transition matrix of this property are of finite impulse response (FIR) systems, since the initial error will disappear in finite steps. (Comment: if $F^M = 0$, where $M<N$, then the error system will converge in M steps.)
3. If $F^N = (A - GC)^N \neq 0$, then the size of the state vector of a stabile error system will decrease exponentially. The error system is stable, if all its eigenvalue is within the unit circle. Systems having state transition matrix of this property have infinite impulse response (IIR systems), because the initial error will disappear in infinite steps.

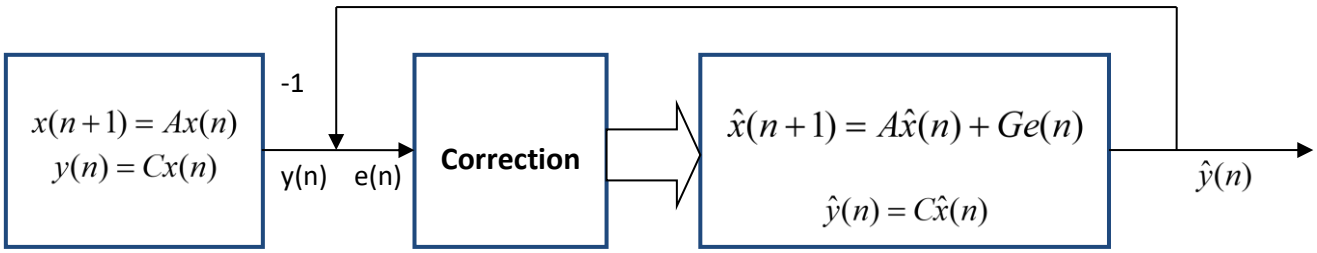
5. Quantities and variables in real-time systems (cont.)

Modelling of the recipient environment:

Observation in case of deterministic channel: the illustrative example below presents a discrete observer. The observed reality is described by a discrete model, and it is supposed to be an autonomous system. The state equations and the observation equation describing the reality and the observation:

$$x(n+1) = Ax(n), \tag{1}$$

$$y(n) = Cx(n), \tag{2}$$



where the state variable $x(n)$ is of N dimension, the state-transition matrix is of $N*N$ dimension, the observation vector $y(n)$ is of $M \leq N$ dimension, and finally the observation matrix C is of $M*N$ dimension. Our aim is the estimation of the state variable $x(n)$. The tool of this estimation is the observer mechanism, which tries to produce a copy of the reality. This performed by a computer program capable to follow the reality as a result of a correction/training/adaptation process. The result of this process is the estimation of the value to be measured. After convergence, this estimator $\hat{x}(n)$ can be read from the observer. The state and the observation equations of the observer are:

$$\hat{x}(n+1) = A\hat{x}(n) + Ge(n), \tag{3}$$

$$\hat{y}(n) = C\hat{x}(n), \tag{4}$$

where correction matrix G is of $N*M$ dimension, $e(n) = y(n) - \hat{y}(n)$. Matrix G is to be designed in such a way that $\hat{x}(n) \to x(n)$. The difference of (1) and (3):

$$x(n+1) - \hat{x}(n+1) = Ax(n) - A\hat{x}(n) - Ge(n) = (A - GC)(x(n) - \hat{x}(n)). \tag{5}$$

Introducing notations: $\varepsilon(n+1) = x(n+1) - \hat{x}(n+1)$, and $F = A - GC$, the state transition matrix of the so-called error system is:

$$\varepsilon(n+1) = F\varepsilon(n). \tag{6}$$

The correction matrix G is designed to result in $\varepsilon(n) \xrightarrow[n\to\infty]{} 0$, possibly with $\|\varepsilon(n+1)\| < \|\varepsilon(n)\|$, for $\forall$ n, i.e. matrix F reduces the size of vector, i.e. it is „contractive".

Comment:

Obviously it is not a necessary condition to decrease the state error in every step: only the stability of the error system is required, i.e. the convergence of the error to zero for zero input. This property can be interpreted also in such a way that the internal energy of the error system is dissipated. If this is the case in every step then the decrease of the size of the error vector will be a monotonic process.

Special cases:

1. $F = A - GC = 0$. In this case $G = AC^{-1}$. This is possible if C is a square matrix, i.e. the observation has as many components as the state vector itself. In this case the observer, and the copy of the system investigated can follow the system without iteration, in one step.

2. $F^N = (A-GC)^N = 0$. In this case the error system converges in *N* steps:

$$x(N) - \hat{x}(N) = (A-GC)^N (x(0) - \hat{x}(0)) = 0 \tag{7}$$

The matrices with the property $F^N = 0$ can be characterized by the fact that all they eigenvalues are zero. Systems having state transition matrix of this property are of finite impulse response (FIR) systems, since the initial error will disappear in finite steps. (Comment: if $F^M = 0$, where *M*<*N*, then the error system will converge in *M* steps.)

3. If $F^N = (A-GC)^N \neq 0$, then the size of the state vector of a stabile error system will decrease exponentially. The error system is stable, if all its eigenvalue is within the unit circle. Systems having state transition matrix of this property have infinite impulse response (IIR systems), because the initial error will disappear in infinite steps.

Examples:

1. Example: Given $A = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$; $C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. How to set *G*-t? $G = AC^{-1} = A = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

2. Example: Given $A = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$; $C = \begin{bmatrix} 1 & 1 \end{bmatrix}$. How to set *G*-t? $G = \begin{bmatrix} g_0 \\ g_1 \end{bmatrix} = ?$

$GC = \begin{bmatrix} g_0 \\ g_1 \end{bmatrix} \begin{bmatrix} 1 & 1 \end{bmatrix} = \begin{bmatrix} g_0 & g_0 \\ g_1 & g_1 \end{bmatrix}$. $[A-GC] = \begin{bmatrix} 1-g_0 & -g_0 \\ -g_1 & -1-1g_1 \end{bmatrix}$. $[A-GC]^2 = 0$ is the condition to be fulfilled by *G*:

$$\begin{bmatrix} 1-g_0 & -g_0 \\ -g_1 & -1-g_1 \end{bmatrix} \begin{bmatrix} 1-g_0 & -g_0 \\ -g_1 & -1-g_1 \end{bmatrix} = \begin{bmatrix} 1-2g_0+g_0^2+g_0g_1 & -g_0+g_0^2+g_0+g_0g_1 \\ -g_1+g_1^2+g_1+g_0g_1 & 1+2g_1+g_1^2+g_0g_1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

By replacing the values of the side diagonal into the main diagonal we have: $1-2g_0 = 0$, and $1+2g_1 = 0$, where from: $g_0 = 0.5$ and $g_1 = -0.5$. As a control we can write:

$$\begin{bmatrix} 0.5 & -0.5 \\ 0.5 & -0.5 \end{bmatrix} \begin{bmatrix} 0.5 & -0.5 \\ 0.5 & -0.5 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

3. Example: Let us compute the eigenvalues of $[A-GC]$ using the results of *Example* 2:

$$\det[\lambda I - A + GC] = 0 = \det \begin{bmatrix} \lambda - 0.5 & 0.5 \\ -0.5 & \lambda + 0.5 \end{bmatrix} = (\lambda - 0.5)(\lambda + 0.5) + 0.25 = \lambda^2 - 0.25 + 0.25 = 0.$$

Both eigenvalues are zero.

Comments:

1. This property is valid in every system capable to converge in finite steps.
2. The transfer function of such systems is a rational function having all its poles at the origin:

$$H(z) = a_1 z^{-1} + a_2 z^{-2} + ... + a_N z^{-N} = \frac{a_N + a_{N-1} z + a_{N-2} z^2 + ... + a_1 z^{N-1}}{z^N} \tag{8}$$

These are the so-called Finite Impulse Response (FIR) filters. The time-domain equivalent of (8):

$$y(n) = a_1 x(n-1) + a_2 x(n-2) + ... + a_N x(n-N), \tag{9}$$

where due to computability reasons only previous samples of $x(n)$ are used.

3. The condition for the eigenvalues (See *Example 3*) can be used to compute g_0 and g_1:

$$\det[\lambda I - A + GC] = 0 = \det \begin{bmatrix} \lambda - 1 + g_0 & g_0 \\ g_1 & \lambda + 1 + g_1 \end{bmatrix} = \lambda^2 + \lambda(g_0 + g_1) + g_0 - g_1 - 1 = \lambda^2 = 0$$

Where from: $g_0 + g_1 = 0$, and $g_0 - g_1 = 1$, Thus: $g_0 = 0.5$ and $g_1 = -0.5$.

Comments:

1. Thanks to the principle of superposition, the system observed and the observer itself can have an additional, common external excitation without any change in the convergence properties.
2. The observer of Figure 2 is called Luenberger observer. According to Luenberger almost any system is an observer. The only requirement is that the observer should be faster than the observed system; otherwise it will not be able to follow its changes.
3. The bridge-branch containing the impedance to be measured within an impedance measuring bridge implements the physical model of the realty, while the tuneable bridge-branch correspond to the model built into the observer. The difference between the outputs of the voltage divider bridge-branches controls the correction mechanism. Finally the value of the unknown impedance will be computed from the correction value. This setup, together with the operator responsible for tuning, implements an observer.

Observation in the case of noisy observation channel: In this case our expectation is not $\|\varepsilon(n)\| \xrightarrow[n\to\infty]{} 0$, but the trace of $E\{[\![\varepsilon(n)\varepsilon^T(n)]\!]\} \xrightarrow[n\to\infty]{} min$. The state equation (6) will be replaced by

$$E[\varepsilon(n+1)\varepsilon^T(n+1)] = FE[\varepsilon(n)\varepsilon^T(n)]F^T. \tag{8}$$

This matrix will play a central role in the famous Kalman predictor and filter.

Linear Least Squares (LS) Estimation/Estimator: No a priori information is available neither from the parameter to be measured, nor from the channel characteristics/noise. Let us suppose that the observation equation is linear: $z = Ua + n$, where *z* stands for the observation vector, *a* for the unknown parameter, *U* is of full rank, and *n* represents the additive noise vector. We assume that parameter *a* takes the value $\hat{a}$. The model of the observation is: $U\hat{a}$. We compare this value with the observation, and we are looking for the best value of $\hat{a}$ by minimizing the LS cost:

$$C(a,\hat{a}) = (z - U\hat{a})^T (z - U\hat{a}) = z^T z - z^T U\hat{a} - \hat{a}^T U^T z + \hat{a}^T U^T U\hat{a} = z^T z - 2\,\hat{a}^T U^T z + \hat{a}^T U^T U\hat{a} \tag{11}$$

The gradient is: $\left.\frac{\partial C(a,\hat{a})}{\partial \hat{a}}\right|_{\hat{a}=\hat{a}_{LS}} = -2U^T z + 2U^T U\hat{a} = 0$, therefore

$$\boxed{\hat{a}_{LS} = [U^T U]^{-1} U^T z} \tag{12}$$

Comment:

The LS criterion can be modified by including a positive definite (symmetric) weighting matrix *Q*:

$$C(a,\hat{a}) = (z - U\hat{a})^T Q (z - U\hat{a}), \tag{13}$$

that leads to the following estimator:

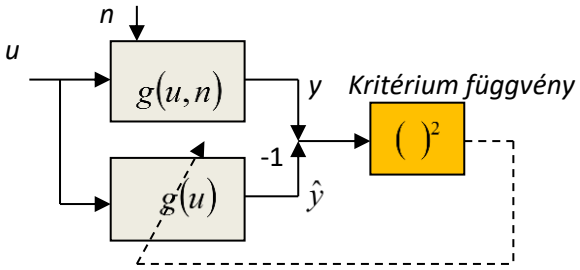
$$\boxed{\hat{a}_{LS} = [U^T QU]^{-1} U^T Qz}. \tag{14}$$

Model fitting

In the case of LS estimators, we do not have priori information about parameter to be measured, therefore what we do is model fitting. The problem of model fitting is manifold. A classical version is regression calculus.

Regression calculus: The determination of a possibly deterministic relation of independent and dependent variables can be considered as a special case of model fitting. On the figure below the function to be modelled $y = g(u,n)$ has two types of independent variables: the one denoted by *u*(*n*), is known and can be influenced, while the other, denoted by *n*(*n*), is unknown, and cannot be influenced. This latter is typically a noise process, or disturbance modelled as a noise process. In the argument of the independent variables *n*

stands for discrete time index, that can be considered as a timestamp assigned to an actual value of the variable.



Comments:

1. In the following sections the independent discrete variable *n* will identify an iteration index or a discrete time index, which appears time to time as explicit index, as well. Consequently, $u(n)=u_n$, and $y(n)=y_n$ are equivalent.
2. Note the double use of *n*: it can be an iteration or time index, or standing alone, it denotes a noise process.

For modelling we use a "tuneable" function $\hat{y}=\hat{g}(u)$ the free parameters of which are tuneable. The in least squares sense optimal setting of the parameters might be a useful strategy:

$$\varepsilon = E\left\{(y-\hat{y})^T(y-\hat{y})\right\} \tag{15}$$

Linear regression: The function to be fitted is a scalar linear function $\hat{g}(u)=a_0+a_1u$**,** the parameters of which are set to minimize $E\left\{(y-\hat{g}(u)^2\right\}$. Let us denote the expected value and the standard deviation of *u* and *y* by $\mu_u,\mu_y,\sigma_u,\sigma_y$, and the normalized cross-correlation by: $\rho=\dfrac{E\left\{(u-\mu_u)(y-\mu_y\right\}}{\sigma_u\sigma_y}$. If we minimize

$$\varepsilon = E\left\{(y-a_0-a_1u)^2\right\}=E\left\{y^2\right\}+a_0^2+a_1^2E\left\{u^2\right\}-2a_0E\left\{y\right\}-2a_1E\left\{uy\right\}-2a_0a_1E\left\{u\right\} \tag{16}$$

by a_0 and a_1:

$$\frac{\partial\varepsilon}{\partial a_0}=2a_0-2\mu_y+2a_1\mu_u=0\text{, where from } a_0=\mu_y-a_1\mu_u\text{,} \tag{17}$$

which is replaced into $\dfrac{\partial\varepsilon}{\partial a_1}=2a_1(\sigma_u^2+\mu_u^2)-2(\rho\sigma_u\sigma_y+\mu_u\mu_y)+2a_0\mu_u=0$, then:

$$\boxed{\hat{a}_0=\mu_y-\mu_u\rho\frac{\sigma_y}{\sigma_u},\ \hat{a}_1=\rho\frac{\sigma_y}{\sigma_u}} \tag{18}$$

Comments:

1. Deriving (18) we utilized the relations
$$E\left\{(u-\mu_u)^2\right\}=\sigma_u^2=E\left\{u^2\right\}-\mu_u^2\text{, and } E\left\{(u-\mu_u)(y-\mu_y)\right\}=E\left\{uy\right\}-\mu_u\mu_y\text{.}$$
2. A possible generalization of the linear regression problem is the polynomial regression. The modelling function is
$$\hat{g}(u)=\sum_{k=0}^{N}a_ku^k\text{,} \tag{19}$$
which is linear in its parameters. We prefer models linear in their parameters, because in case of squared error criterion, finding the optimum requires the solution of a set of linear equations.

Linear regression based on measured data: The above development can be carried out also for the case where no a priori information is available. Here for the measured values we have $y_n=a_0+a_1u_n+w_n$, where

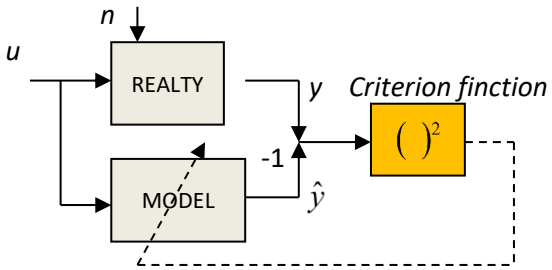
w_n denotes additive noise, n=0,1, …, N-1. With vector notation: $z = Ua + w$. Using the least squares estimation (LS) method:

$$z = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & u_0 \\ 1 & u_1 \\ \vdots & \vdots \\ 1 & u_{N-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} + \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{N-1} \end{bmatrix}, \quad [U^T U] = \begin{bmatrix} N & \sum_{n=0}^{N-1} u_n \\ \sum_{n=0}^{N-1} u_n & \sum_{n=0}^{N-1} u_n^2 \end{bmatrix}, \quad U^T z = \begin{bmatrix} \sum_{n=0}^{N-1} y_n \\ \sum_{n=0}^{N-1} u_n y_n \end{bmatrix}.$$

$$\begin{bmatrix} \hat{a}_0 \\ \hat{a}_1 \end{bmatrix} = \frac{1}{\frac{1}{N}\sum_{n=0}^{N-1} u_n^2 - \left(\frac{1}{N}\sum_{n=0}^{N-1} u_n\right)^2} \begin{bmatrix} \frac{1}{N}\sum_{n=0}^{N-1} u_n^2 & -\frac{1}{N}\sum_{n=0}^{N-1} u_n \\ -\frac{1}{N}\sum_{n=0}^{N-1} u_n & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{N}\sum_{n=0}^{N-1} y_n \\ \frac{1}{N}\sum_{n=0}^{N-1} u_n y_n \end{bmatrix}.$$

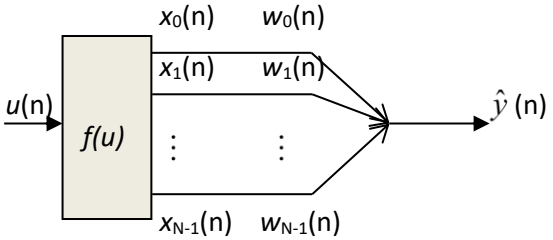
Comment: In the result above we can identify the estimators of the statistical descriptors used in (16), and by converting one to the other, we can find a complete correspondence. Do it!

Generalization of the regression scheme: On the figure below the model fitting problem is presented as a regression scheme:



The response y to the input u is to be compared with the response $\hat{y}$ of the model. (It worth comparing this structure with the observer scheme: the similarity is obvious; we are fitting a model in both cases. For the observer we know the parameters, and the state variables are to be estimated, while for the regression scheme the state variables are known and the parameters are to be estimated. Both schemes are parallel in the sense that both input signals enter parallel.

Adaptive linear combinator: A frequently used model-family. The model consists of two parts: (1) a fix, multiple output function, and (2) a linear combinatory with variable/tuneable weights:



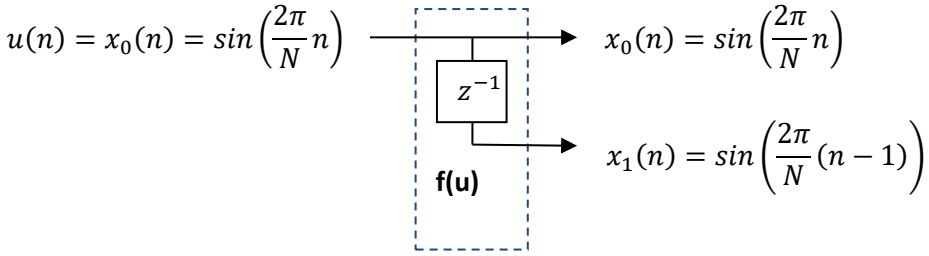
The fix part generates a sequence of values $X^T(n) = [x_o(n) \quad x_1(n) \quad \ldots \quad x_{N-1}(n)]$ from $u(n)$, and the linear combination of these values results in $\hat{y}(n)$. We are looking for the minimum squared error by searching the optimum $W^T(n) = [w_0(n) \quad w_1(n) \quad \ldots \quad w_{N-1}(n)]$ parameters. We are minimizing

$$\varepsilon(n) = E\{[y(n) - X^T(n)W(n)]^T [y(n) - X^T(n)W(n)]\} =$$
$$= E\{y^T(n)y(n)\} - 2W^T(n)E\{X(n)y(n)\} + W^T(n)E\{X(n)X^T(n)\}W(n). \qquad (20)$$

Let us introduce the following notations: $E\{X(n)y(n)\}=P$, and $E\{X(n)X^T(n)\}=R$! The minimum will be obtained at $\frac{\partial\varepsilon(n)}{\partial W(n)}=-2P+2RW(n)=0$, thus the optimum setting is: $\boxed{W^*=R^{-1}P}$ (21)

Expression (21) is called Wiener-Hopf equation.

Example: Imagine $X^T(n)=[\sin(2\pi n/N) \quad \sin(2\pi(n-1)/N]$**,** i.e**.** two subsequent sample of a sine wave:



In this example the regression and the parameter vectors are of two dimensions. Here *N* denotes the number of samples taken from one period. The signal to be approximated let $y(n)=2\cos(2\pi n/N)$. How to select parameters

$$W^T(n)=[w_0(n) \quad w_1(n)] \tag{22}$$

to minimize the mean square error? The matrices *R* and *P* can be computed by averaging sine and cosine waveforms for the complete period (*N*>2):

$$R=\begin{bmatrix} 0.5 & 0.5\cos\frac{2\pi}{N} \\ 0.5\cos\frac{2\pi}{N} & 0.5 \end{bmatrix},\ P=\begin{bmatrix} 0 \\ -\sin\frac{2\pi}{N} \end{bmatrix}. \tag{23}$$

$$E\{\sin^2(2\pi/N)\}=E\{\sin^2(2\pi(n-1)/N)\}=0.5,\ E\{\sin(2\pi n/N)\sin(2\pi(n-1)/N)\}=\frac{\cos(\frac{2\pi}{N})}{2},$$

$$E\{2\sin(2\pi n/N)\cos(2\pi n/N)\}=0,\qquad E\{2\sin(2\pi(n-1)/N)\cos(2\pi n/N)\}=-\sin\frac{2\pi}{N}.$$

$$R^{-1}=\frac{1}{0.25\sin^2\frac{2\pi}{N}}\begin{bmatrix} 0.5 & -0.5\cos\frac{2\pi}{N} \\ -0.5\cos\frac{2\pi}{N} & 0.5 \end{bmatrix},\ \boxed{W^*=R^{-1}P=\begin{bmatrix} \frac{2}{\tan(2\pi/N)} \\ -\frac{2}{\sin(2\pi/N)} \end{bmatrix}} \tag{24}$$

Comments:

1. $X^T(n)W^*=2\frac{\sin(2\pi n/N)}{\tan(2\pi/N)}-2\frac{\sin(2\pi(n-1)/N)}{\sin(2\pi/N)}=2\cos(2\pi n/N)$.

2. Since cosine waveforms can be generated as linear combination of different phase sine waves, therefore for the case of the example $\varepsilon_{\min}=0$, i.e. the lowest point of the error surface (paraboloid) reaches the hyper-plane of the parameters.

Towards adaptive procedures: Based on (20) and (21): $W^*=R^{-1}P$, $\nabla(n)=2(RW(n)-P)$. Multiplying both sides by $\frac{1}{2}R^{-1}$: $\boxed{W^*=W(n)-\frac{1}{2}R^{-1}\nabla(n)}$. (25)

If we have only approximate knowledge about matrix *R*, and consequently about the gradient, (25) can be rewritten to an iterative form: $W(n+1)=W(n)-\frac{1}{2}\hat{R}^{-1}\hat{\nabla}(n)$, or by introducing a convergence factor and assuming the perfect *R* and the perfect gradient, we have:

$$\boxed{W(n+1) = W(n) - \mu R^{-1}\nabla(n)}\ . \qquad (26)$$

Comments:

1. If matrix *R* matrix and the gradient are perfectly known, then setting $\mu = \frac{1}{2}$ provides one-step convergence from an arbitrary (but finite) initial value W(*n*).

2. Since $\nabla(n) = 2R[W(n) - W^*]$, therefore by introducing this into (26), and subtracting W^* from both sides of the equation:
$W(n+1) - W^* = (1-2\mu)(W(n) - W^*) = V(n+1) = (1-2\mu)^{n+1}V(0)$, i.e. the initial parameter error will decrease exponentially, if $\mu \neq \frac{1}{2}$. If $0 < \mu < 0.5$, then the error will decrease monotonically, otherwise with oscillating sign.

3. The gradient methods of model fitting are distinguished by the a priori knowledge available to evaluate (26).
If the R and P matrices are known, the equations describing the behaviour of adaptive linear combinator are as follows:

$$\boxed{W(n+1) = W(n) - \mu R^{-1}\nabla(n)}, \text{ and } \boxed{V(n+1) = (1-2\mu)V(n)}\ . \qquad (27)$$

5. Quantities and variables in real-time systems (cont.)

Replica determinism

The reliability of a system can be improved by active redundancy, i.e. by replicated physical components. A set of replicated RT objects is *replica determinate* if all the members of this set have the same externally visible RAM state, and produce the same output messages at points in time that are at most an interval of *d* time unit apart (as seen by the omniscient outside observer with the reference clock *C*). In a fault-tolerant system, the time interval d determines the time it takes to replace a missing message or an erroneous message from a node by a correct message from redundant replicas. This time must be derived from the dynamics of the application.

A *major decision point* is a decision point in an algorithm that provides a choice between a set of significantly different courses of actions.

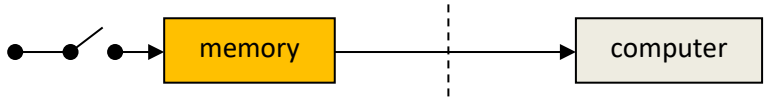
Example: Consider an airplane with a three-channel flight-control system and a majority voter. Each channel has its own sensors and computers to minimize the possibility of a common-mode error. Within a specified time interval after the event "start of take-off", the control system must check whether the plane has attained the take-off speed. In case the take-off speed has been attained, the lift-off procedure is initiated, and the engines are further accelerated. In case the take-off speed has not been reached within this specified time interval, the take-off must be aborted, and the engines must be stopped. The decision whether or not to take off occurs at a major decision point. The table below describes such a situation, where the condition of replica determinism is not met, and the faulty channel governs the decision.

Channel	Decision	Action
Channel 1	Take off	Accelerate engine
Channel 2	Abort	Stop engine
Channel 3	Abort	Accelerate engine

Assume that the speed of the plane at the major decision point is about the same as the specified limit of the take-off speed. Because of random effects (deviation in the sensor calibration, digitalization error, slightlz different points in the time of speed measurement), channels 1 and 2 reach different conclusions: channel 1 decides that the take-off speed has been reached and that the plane should take off. Channel 2 decides that the take/off speed has not been reached and the take-off speed and the take-off should be aborted. Both channels take the correct decision, although the decisions are *not replica determinate*. Channel 3 is faulty and decides to abort, and to accelerate the engine. In the majority vote of the action, the faulty channel wins, because the correct channels are not replica determinate.

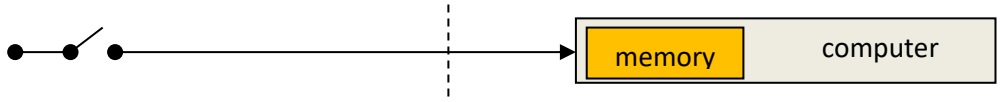
Sampling and polling:

We use the term sampling, if the data is written into a memory element at the sensor:



From the point of view of system specification, a sampling system can be seen as protecting a node from more events in the environment than are stated in the system specification. The memory is at the sensor and thus outside the sphere of control of the computer.

We use the term polling, if the data memory is resided inside the computer:

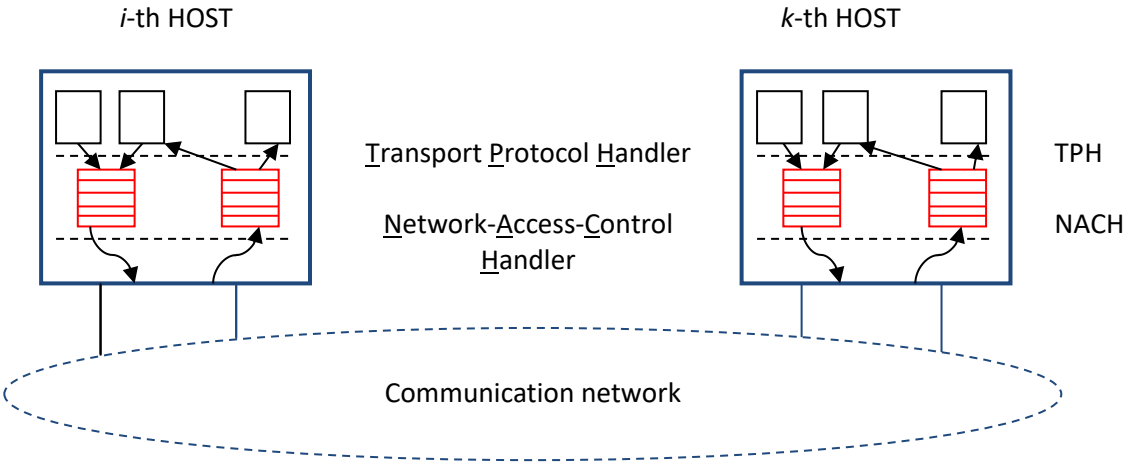


From functional point of view, there is no difference between sampling and polling as long as no faults occur. Under fault conditions, the sampling system is more robust than the polling system.

Comment: The interrupt mechanism can be characterized by the figure introduced for polling. The interrupt mechanisms empower a device outside the sphere of control of the computer to govern the temporal control pattern inside the computer. This is a powerful a potentially dangerous mechanism that must be used with great care. From the fault-tolerance point of view, an interrupt mechanism is even less robust than the already denounced polling mechanism. Every transient error on the transmission line will interfere with the temporal control scheme within the computer. It will generate an additional unplanned processing load for the detection of a faulty sporadic interrupt, making it more difficult to meet the specified deadlines.

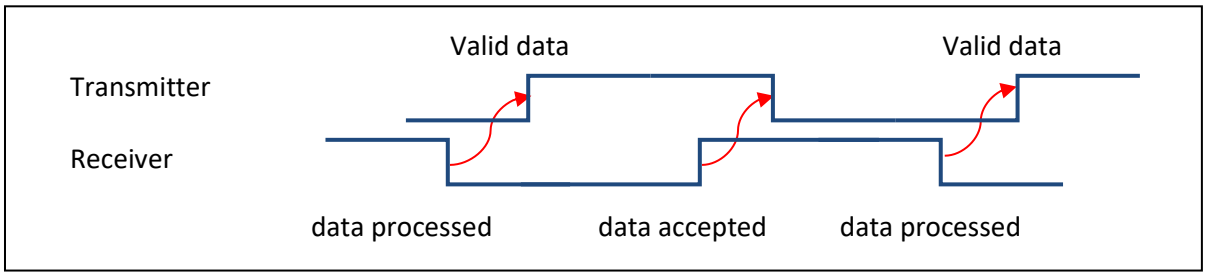
6. Real-Time (RT) communication

The general scheme:



Comment: In general complicated mechanisms, different queues. The RT requirements are hard to meet.

The criticality of time conditions can be identified even on the physical level. In case of asynchronous communication, some kind of synchronization or handshaking is unavoidable. Handshaking with two wires:



The speed and time conditions of the asynchronous communication are determined by both actors, since till the processing of the received data, the transmitter cannot forward the next data.

Real/time communication requirements:

1. A RT communication protocol should have a predictable, and small maximum protocol latency and a minimal jitter. The standard communication topology in distributed real/time systems is multicast, not point-to-point. A message should be delivered to all receivers within a short and known time interval.
2. Support for Composability: (1) Temporal encapsulation of the nodes: the communication system should erect a temporal firewall around the operation of the host, forbidding the exchange of control signals across the Communication Network Interface (CNI). Thus, the communication system becomes autonomous and can be implemented and validated independently of the application software in the host.

(2) Fulfilling the obligations of the Client: a host implementing server functions can guarantee its deadlines if the clients fulfill their obligations, and do not overload the host with too many, uncoordinated service requests.

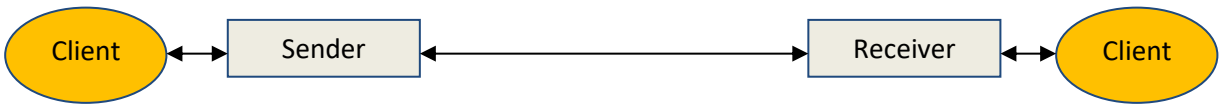
3. Flexibility: An RT protocol should be flexible to accommodate different system configurations without requiring a software modification and retesting of the operational nodes that are not affected by the change. As an example, imagine a car with and without extras.
4. Error detection: The communication system must provide predictable and dependable services. Errors must be detected and corrected without increasing the jitter of the protocol latency. If the errors cannot be corrected, the receivers should be informed about the error with low latency. Loss of information is of particular concern. Consider a node, at a control valve, that receives output commands from another node. In case the communication is interrupted because the wires are cut, the control valve node should enter a safe state autonomously, e.g. it should close the valve. The communication system must inform the control valve node about the loss of communication with low error detection latency. End-to-end protocols are needed (Three Mile Island Nuclear Reactor #2 accident on March 28, 1978).
5. Physical structure: point-to-point communication can easily result in high costs. Physical networks should be based on a bus or a ring structure.

Flow control:

Explicit Flow Control:

Example: PAR (Positive Acknowledgement or Retransmission) protocol: Many variants of the basic PAR protocol are known, but they all rely on the following principle:

(1) The client at the sender's site initiates the communication. (2) The receiver has the authority to delay the sender via the bi-directional communication channel. (3) The communication error is detected by the sender, and not by the receiver. The receiver is not informed when a communication error has been detected. (4) Time redundancy is used to correct a communication error, thereby increasing the protocol latency in case of errors.



Program of the sender:

(1) The sender initializes a retry counter to zero.
(2) The sender starts a local time-out interval.
(3) The sender sends the message to the receiver.
(4) The sender receives an acknowledgement message from the receiver within the specified time-out interval.
(5) The sender informs its client about the successful transmission, and duly terminates.

If the sender does not receive a positive acknowledgement message from the receiver within the specified time-out interval:

(a) The sender checks the retry counter to determine whether the given maximum number of retries has already been exhausted.
(b) If so, the sender aborts the communication, and informs its client about the failure.
(c) If not, the sender increments the retry counter by one, and returns to (2).

Program of the receiver:

(1) If new message arrives at the receiver, the receiver checks whether this message has already been received.
(2) If not, the receiver sends an acknowledgement message to the sender, and delivers the message to its client.
(3) If yes, it just sends another acknowledgement message back to the sender. (In this case the previous acknowledgement message has arrived at the sender out of the specified time-out interval, or failed to arrive.

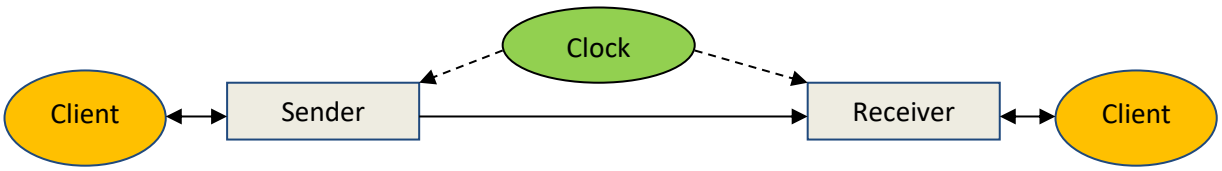
Comment:
The point in time at which the sender's client is informed about a successful transmission, can be significantly different from the point in time at which the receiver's client accepts the delivery of the message.

Example: Consider a bus system where a token protocol controls media access to the bus. The maximum token rotation time (TRT) is 10 ms. The time needed to transport the message on the bus is 1 ms.
The length of time-out interval: 10+1+10+1=22 *ms*, since in worst-case the sender should wait 10 *ms* for the token, the message takes further 1 *ms*, and on the way back the worst case is the same. Here d_{min}=1 *ms*, d_{max}= (number of retries) *time-out+10 *ms*+1 *ms*. If the number of retries is two (i.e. we apply three trials), then d_{max}= 55 *ms*.

- The jitter of this PAR protocol: jitter= d_{max}- d_{min}=54 *ms*.
- The action delay, if we have global clock: d_{max} = 55 *ms*. If the granularity of the global clock is 100 μs, then the message becomes permanent after $d_{max} + 2g = 55.2\mu s$.
- The action delay, if the global clock is not available: 2* d_{max} - d_{min}= 109 *ms*.
- The error detection latency: 3*time-out: 66 *ms*.

This example illustrates that the explicit flow control in RT applications might be disadvantageous due to the large jitter and error detection latency.

Implicit flow control:



The communication is time-triggered. Both the sender and the receiver have a message scheduling time table, which were fixed in design-time. From this schedule it is clear at what time the message is to be sent and received. The sender at the corresponding clock tick pushes the message, while at the same time the receiver pulls the message (push-pull mechanism). This approach fits better to the real-time requirements in many cases. E.g. error detection by the receiver is immediately possible, if the expected message fails to arrive. (For the sender this means a so-called fail-silent mode, where the absence of the message means the error state of the sender.)

Global time-base is needed. The sender transmits only at fixed time instants, no handshaking is applied, error detection is the role of the receiver, since it knows when a message should arrive. Fault tolerance is solved by active redundancy: *k* copies of the message are transmitted, and the transmission is successful unless at least one message arrives. Summary of implicit flow control:

(1) The communication is initiated by clock tick.
(2) The receiver is expecting the message following the clock tick.
(3) Error is detected by the receiver, typically by realizing the absence of the message.
(4) For error correction active/hardware redundancy is applied.

The Time Triggered Architecture (TTA) and the Time-Triggered Protocols (TTPs)

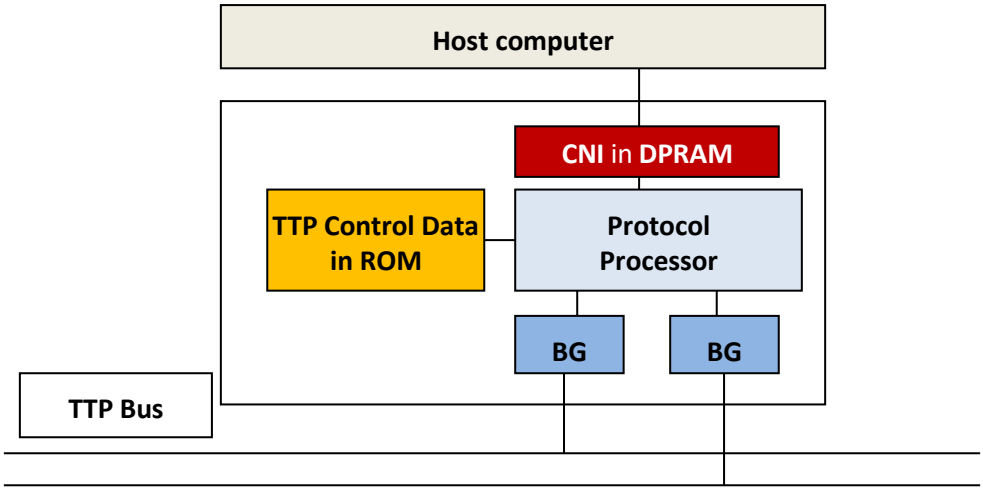
A detailed description can be found in

HERMANN KOPETZ, GÜNTHER BAUER: *The Time-Triggered Architecture*, PROCEEDINGS OF THE IEEE, VOL. 91, NO. 1, JANUARY 2003, pp. 112-126. (This paper is available also on the homepage of the subject.)

Here follows only a brief description of TTA and TTP:

It is devoted to implement hard real-time (HRT) systems. The basic building block of the TTA is a node. A node comprises in a self-contained unit (possibly on a single silicon die) a processor with memory, an input–output subsystem, a TT communication controller, an operating system, and the relevant application software 3. Two replicated communication channels connect the nodes, thus forming a cluster. The cluster communication system comprises the physical interconnection network and the communication controllers of all nodes of the cluster. In the TTA, the communication system is autonomous and executes periodically an a priori–specified time-division multiple access (TDMA) schedule. It reads a state message from the Communication Network Interface (CNI) at the sending node at the a priori–known fetch instant and delivers it to the CNIs of all other nodes of the cluster at the a priori–known delivery instant, replacing the previous version of the state message. The times of the periodic fetch and delivery actions are contained in the message scheduling table [the message descriptor list (MEDL)] of each communication controller. It has two versions: the TTP/C, which serves fault tolerant solutions, and the TTP/A, which is suitable for cheap field bus applications. Each node consists of a Host computer and a Communication Controller (CC). The CNI is the interface within node between the host and the CC. It is a dual-port RAM (DPRAM). Data integrity is solved by the Non-Blocking Write (NBW) Protocol (see later). The local memory of the CC contains the Message Description List (MEDL), that determines which node can send, and which can receive a message at a given time. The size of the MEDL is determined by the cluster round. The TTP controllers contain - as independent hardware so-called Bus Guardian units, which monitor the bus access patterns of the controlling bus, and stop the operation of the controller, if the timing of the regular access patterns fails.

Important properties: (1) The TTP is a time-division-multiple-access (TDMA) protocol. (2) The CC is autonomous, which is controlled by the MEDL and the global clock. This serves composability. The error of the hosts cannot influence the communication system, because control signal cannot get through the CNI, and neither the MEDL can be accessed from the host. (3) The communication system is decided in design time (it is like a time table at the railways): it knows in advance when a message arrives, and when a message is to be sent. If the message fails to arrive, the error is immediately detected. (4) Naming: the name of the message and the sender should not be part of the message, it can be read from the MEDL. At the same time, we can give different names to a given RT variable within the software of different hosts. (5) Acknowledgement: We know in advance that all correct receivers the message of the correct sender. As one of the receivers acknowledges the message, it can be assumed that all correctly operating host has also received it. (6) Fail-silence in the time domain: TTP assumes that the nodes support the “fail silence” abstraction in the time domain, which means that the node either sends a message at right time, or sends nothing. This property within the TTP controller is solved by the bus guardian. The error handling in the magnitude domain is the responsibility of the host. The TTP provides only CRC.



The basic CNI:

The CNI is the most important interface within a time-triggered architecture, because it is the only interface of the communication system that is visible to the software of the host computer. It thus constitutes the programming interface of a TTP network. The Status Registers are written by the TTP controller, while the Control Registers by the host.

Status Registers	Control Registers
(S1) Global Internal Time	(C1) Watchdog
(S2) Node Time	(C2) Timeout Register
(S3) Message Description List	(C3) Mode Change Request
(S4) Membership	(C4) Reconfiguration Request
(S5) Status Information	(C5) External Rate Correction

S1: The common clock of the cluster on two bytes. S2: The clock of node. S3: MEDL Pointer. S4: As many bits as the number of participants within the cluster. If one of the bits is “TRUE”, then that node was in operation within the last time-slot. C1: The host periodically restarts; the controller checks it. If the restart fails, then the controller – supposing error – stops sending messages. C2: Written by the host. If the time is over, it will cause interrupt. For example, later, the host can synchronize its clock to that of the cluster. C3: Using this a new scheduling can be introduced. C4: In case of error a reconfiguration can be initiated. C5: Makes possible external clock synchronization.

The Message Description List (MEDL)

Node Time	Address	D	L	I	A
When	*What: Pointer to the Message*	*direction*	*length*		

I: specifies whether the message is an initialization message or a normal message. A: contins additional protective information concerning mode changes and mode role changes.

Fault-Tolerant Units: Its role is to mask the failure of a node. If it implements fail-silent abstraction, then it is enough to duplicate the nodes to tolerate a single value failure. If the node does not implement the fail-silent abstraction, and can have value-failure at the CNI, then Triple Modular Redundancy (TMR) is to be applied. If in case of node failure, nothing is known about the behaviour of the node, then byzantine error might also occur, i.e. four nodes can mask the error.

2. Fundamental conflicts in protocol design

A balanced protocol design tries to reconcile many requirements. It is important to understand which requirements are compatible with each other, and which requirements are in fundamental conflict with each other, and cannot be reconciled by any design decisions that are made.

External control ↔ Composability

Consider a distributed real-time system consisting of a set of nodes that communicate with each other. Each node has a host computer with CNI. Composability in the temporal domain requires that:

- The CNI of every node is fully specified in the temporal domain;
- The integration of a set of nodes into the complete system does not led to any change of the temporal properties of the individual CNIs, and
- The temporal properties of every host can be tested in isolation with respect to the CNI.

If the temporal properties are not contained in the CNI specification, e.g. because the moment when a message must be transmitted is external and unknown to the communication system, then it is not possible to achieve composability in the temporal domain. If the temporal properties of the CNI are fully specified, then low-level composability can be achieved. There is, however, always the possibility that the application functions interact in an unpredictable manner that precludes high-level composability.

In an event-triggered system, the temporal signals originate external to the communication system, in the hosts of the nodes. It is thus not possible to achieve low-level composability.

Example: If all the nodes can compete at any point in time for a single communication channel on a demand basis, then, it is impossible to avoid the side effects caused by the extra transmission delay resulting from conflicts regarding the access to this single channel, no matter how clever the medium access protocol may be. These extra transmission delays can invalidate the temporal accuracy of the real-time images that are transported in the message.

Flexibility ↔ Error detection

Flexibility implies that the behaviour of a node is not restricted a priori. In an architecture without replication, error detection is only possible if the actual behaviour of the node can be compared to some a priori knowledge of the expected behaviour. If such knowledge is not available, it is not possible to protect the network from a faulty node.

Example: Consider an event-triggered system with no regularity assumptions, where access to a single bus is determined solely by the message priority: if there is no restriction on the rate at which a node may send messages, it is impossible to avoid the monopolization of the network by a single (possibly erroneous) node that sends a continuous sequence of messages of the highest priority.

Example: If a node is not required to send a “heartbeat message” at regular intervals, it is not possible to detect a node failure with a bounded latency.

Sporadic data ↔ Periodic data

A RT protocol can be effective in either the transmission of periodic data or the transmission of sporadic data, but not with both. The transmission of periodic data (e.g. data exchanges needed to coordinate a set of control loops) must take place with minimal latency jitter. Because the repetitive intervals between the transmissions of periodic data are known a priori, conflict-free schedules can be designed in design time. Sporadic data must be transmitted with minimal delay, on demand, at a priori unknown points in time. If an external event requiring the transmission of a sporadic message occurs at the same time as the next transmission of the periodic data, then, the protocol must decide either to delay the sporadic data, or to modify the schedules of the periodic data. In either case, the latency jitter increases: one cannot satisfy both goals simultaneously.

Single locus of control ↔ Fault tolerance

Any protocol that relies on a single locus of control has a single point of failure. This is evident for a communication protocol that relies on a central master. However, even the access method of token passing relies on a single locus of control at any particular moment, with no consideration of time as the control element. If the station holding the token fails, no further communication is possible until the token loss has been detected by an additional time-out mechanism, and the token has been recovered. This takes time, and also interrupts the real-time communication. In some respects, the nontrivial problem of token recovery is related to the problem of switching from a central master to a standby master in a multi-master protocol.

Probabilistic access ↔ Replica determinism

Another fundamental conflict exists between the property of replica determinism (needed if active redundancy is to be applied) and that of medium access based on probabilistic mechanisms. Is systems that rely on a single winner emerging from fine-grained race conditions (e.g. bit arbitration, conflict resolution based on random numbers), it cannot be guaranteed that the access to the replicated communication channels is always resolved identically by competing nodes. Without replica determinism, each replica can come to different correct result, thereby leading to inconsistency in the system as a whole.

Performance Limits in TT systems

As in any distributed computing system, the performance of the TTA depends primarily on the available communication bandwidth and computational power. Because of physical effects of time distribution and limits in the implementation of the guardians, a minimum interframe gap of about 5 μs must be maintained between frames to guarantee the correct operation of the guardians. If a bandwidth utilization of about 80% is intended, then the message-send phase must be in the order of about 20 μs, implying that about 40 000 messages can be sent per second within such a cluster. With these parameters, a sampling period of about 250 μs can be supported in a cluster comprising ten nodes. The precision of the clock synchronization in current prototype systems is below one microsecond. If the interframe gap and bandwidth limits are stretched, it might be possible to implement in such a system a 100 μs TDMA round (corresponding to a 10-kHz control loop frequency), but not much smaller if the system is physically distributed (to tolerate spatial proximity faults). The amount of data that can be transported in the 20 μs window depends on the bandwidth:
In a 5-Mb/s system it is about 12 bytes: $5*10^{6}*20*10^{-6}$ = 100 bit (~12 byte);
In a 1-Gb/s system it is about 2500 bytes: $1*10^{9}*20*10^{-6}$ = 20 000 bit (2500 byte) bytes.

Synchronizing ET and TT systems

The processor of the host operates in ET mode, while the network in TT mode. This means that the CNI cannot be blocked without consequences. The writing from the network is investigated.
Non-blocking Write Protocol (NBW): At the interface there is one writer, the communication system, and many readers, the tasks of the host. A reader does not destroy the information written by the writer, but the writer can interfere with the operation of the reader. In the NBW protocol, the writer is never blocked. It wiil thus write a new version of the message into the DPRAM of the CNI whenever a new message arrives. If a reader reads the message while the writer is writing a new version, the retrieved message will contain inconsistent information and must be discarded. If the reader is able to detect the interference, then the reader can retry the read operation until it retrieves a consistent version of the message. It must be shown that the number of retries performed by the reader is bounded.
The protocol requires a concurrency control field, CCF, for every message written. Atomic access to the CCF must be guaranteed by the hardware. The CCF is initialized to zero and incremented by the writer before start of the write operation. It is again incremented after the completion of the write operation. The reader starts by reading the CCF at the start of the read operation. If the CCF is odd, then the reader retries immediately because a write operation is in progress. At the end of the read operation the reader checks

whether the CCF has been changed by the writer during the read operation. If so, it retries the the read operation again until it can read an uncorrupted version of the data structure.

Inicialization: CCF:=0

Writer:
Start: CCF_old:=CCF;
CCF:=CCF_old+1;
<write into data structure>
CCF:=CCF_old+2;

Reader:
Start: CCF_begin:=CCF;
if CCF_begin=odd **then goto** Start;
<read data structure>
CCF_end:=CCF;
if CCF_end ≠ CCF_begin **then goto** Start;

It can be shown that upper bound for the number of read retries exists if the time between write operation is significantly longer than the duration of a write or read operation.

6. Real-Time (RT) communication (cont.)

Characteristics of a Communication Channel:

A communication channel is characterized by its bandwidth and its propagation delay.

Bandwidth: The bandwidth indicates the number of bits that can traverse a channel in unit time. It is determined by the physical characteristics of the channel. In a harsh environment, such as a car, it is not possible to transmit more than 10kbit/sec over a single-wire channel or 1 Mbit/sec over an unshielded twisted pair because of EMI constraints. In contrast, optical channels can transport gigabits of data per second.

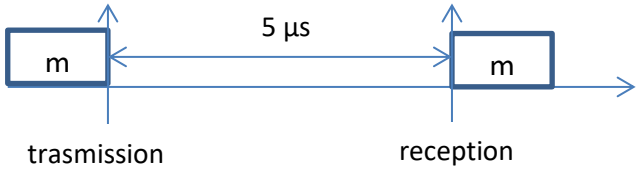
Propagation Delay: The propagation delay is the time interval it takes for a bit to travel from one end of the channel to the other end. It is determined by the length of the channel and the transmission speed of the wave (electromagnetic, optical) within the channel. The transmission speed of an electromagnetic wave in vacuum is about 300 000 km/sec, or 1 foot/nsec. Because the transmission speed of a wave in a cable is approximately 2/3 of the transmission speed of light in vacuum, it takes a signal about 5 µs to travel across a cable of 1 km length.

The term *bit length* of a channel is used to denote the number of bits that can traverse the channel within one propagation delay. For example, if the channel bandwidth is 100 Mbit and the channel is 200 m long, the bit length of the channel is 100 bits, since the propagation delay of this channel is 1 µs.

Limit to Protocol Efficiency: In a bus system, the data efficiency of any media access protocol to a single channel is limited by the need of to maintain a minimum time interval of one propagation delay between two successive messages. Assume the bit length of a channel to be *bl* bits and the message length to be *m* bits. Than an upper bound for the data efficiency of any media access protocol in a bus system is given by:

$$data\ efficiency < \frac{m}{m + bl}$$

Example: Consider a 1 km bus with a bandwidth equal to 100 Mbits/sec. The message length that is transmitted over the channel is 100 bits. It follows that the bit length of the channel is 500 bits, and the limit to the data efficiency is 100/(500+100)=16.6%.



Comments to the topic of real/time communication:

1. Properties of Transmission Codes:

The terms asynchronous and synchronous have different meanings depending on whether they are used in the computer-science community or in the data-communication community.

In asynchronous communication, the receiver synchronizes its receiving logic with that of the sender only at the beginning of a new message. Since the clocks of the receiver and the sender drift apart during the interval of the message reception, the message length is limited in asynchronous communication, e.g. to about 10 bits in a UART (Universal Asynchronous Receiver Transmitter) device that uses a low-cost resonator with a drift rate of $10^{-2} sec/sec$.

In synchronous communication, the receiver resynchronizes its receive logic during the reception of a message to the ticks of the sender's clock. This is only possible if the selected data encoding guarantees

frequent transitions in the bit stream. A code that supports the resynchronization of the receiver's logic to the clock of the sender during transmission is called synchronizing code.

NRZ code: (non-return-to-zero)***:*** non-synchronizing code
11010001: "1" corresponds to high level, "0" correspond to low level

Manchester code: synchronizing code
11010001: "1" corresponds to rising edge: from low to high, "0" corresponds to a falling edge: from high to low. These edges appear in the middle of the clock interval. Always the next bit tells whether the signal should return to the other level at time instant of the clock, or not. If the new bit equals the previous one, then it should return. This code is ideal from the point of view of resynchronization, but it has the disadvantage that the size of a feature element, i.e. the smallest geometric element in the transmission sequence, is half of the bit cell.

Modified Frequency Modulation Code (MFM): The MFM code is a code that has a feature size of one bit cell and is also synchronizing. The encoding scheme requires distinguishing between a data point and a clock point. A "0" is encoded by no signal change at data point; a "1" requires a signal change at data point. If there are more than two "0"s in sequence, the encoding rules require a signal change at clock points.

2. Time Synchronization in Wireless Sensor Networks

Classes of Synchronization:

- *Internal versus external*

  The synchronization of all clocks in the network to a time supplied from outside the network is referred to as external synchronization. NTP performs external synchronization, and so do sensor nodes synchronizing their clocks to a master node. Internal synchronization is the synchronization of all clocks in the network, without a predetermined master time. The only goal here is the consistency among the network nodes.

- *Lifetime: Continuous versus on-demand*

  The lifetime of synchronization is the the period of time during which synchronization is required to hold. If time synchronization is continuous, the network nodes strive to maintain synchronization (of a give quality) at all times. For some sensor-network applications, on-demand synchronization can be as good as continuous synchronization in terms of synchronization quality, but much more efficient. During the (possibly long) periods of time between events, no synchronization is needed, and communication and hence energy consumption can be kept at a minimum. As the time intervals between successive events become shorter, a break-even point is reached where continuous and on-demand synchronization perform equally well. There are two kinds of on-demand synchronizations:

  Event-triggered on-demand synchronization is based on the idea that in order to time-stamp a sensor event, a sensor needs a synchronized clock only immediately after the event has occurred. It can then compute the time-stamp for the moment in the recent past when the event occurred (Post-facto synchronization).

  Time-triggered synchronization is used if we are interested in obtaining sensor data from multiple sensor nodes for a specific time. This means that there is no event that triggers the sensor nodes, but the nodes have to take a sample at precisely the right time. This can be achieved via *immediate* synchronization (where sensor nodes receive the order to immediately take a sample and time-stamp it) or *anticipated* synchronization (where the order is to take the sample at some future time, the *target time*). Anticipated synchronization is necessary if it cannot be guaranteed that the order can be transmitted rapidly ans simultaneously to all involves sensor nodes. This is especially the case if sensor nodes are more than one hop away from the node giving the order.

  Note that for successful *anticipated* synchronization, it is sufficient to maintain a synchronization quality which guarantees that the target time is not missed. This means that the required synchronization quality

grows as the real time approaches the target time. There is no need to synchronize with maximum quality right from the beginning.

Analogously to the event-triggered *post-facto synchronization*, we might refer to time-triggered synchronization as *pre-facto synchronization*.

- *Scope: all nodes versus subsets*

The *scope* of synchronization defines which nodes in the network are required to be synchronized. Depending on the application, the scope comprises all or only a subset of the nodes (where and when synchronization is required). Event-triggered synchronization can be limited to collocated subset of nodes which observe the event in question.

- *Rate synchronization versus offset synchronization*

Rate synchronization means that nodes measure identical time-interval lengths. In a scenario where sensor nodes measure the time between the appearance and disappearance of an object, rate synchronization is a sufficient and necessary condition for comparing the duration of the object's presence within the sensor range of different nodes.

Offset synchronization means that nodes measure identical points in time, that is at some time *t*, the software clocks of all nodes in the scope show *t*. *Offset synchronization* is needed for combining time stamps from different nodes.

- *Timescale transformation versus clock synchronization*

Time synchronization can be achieved in two fundamentally different ways. We can synchronize clocks, that is make all clocks display the same time at any given moment. To achieve this, we must perform rate and offset synchronization (or continuous offset synchronization, which however is costly in terms of energy and bandwidth and requires reliable communication links). The other approach is to transform timescales, that is to transform local times of one node into local times of another node.

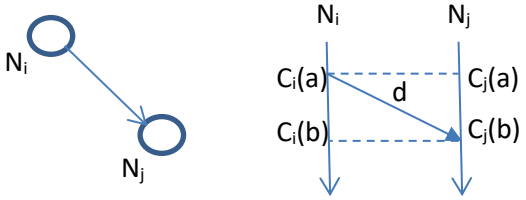
The approaches in that clock synchronization requires either communication across the whole network or some degree of global coordination. Timescale transformation does not have this drawback, but instead requires additional computations and memory overhead, since the received timestamps must be transformed.

- *Time instants versus time intervals*

Time information can be given by specifying time instants (e.g., "t=5") or time intervals ("t∈ [4.5,5.5]"). In both cases, the time information can be refined by adding a statement of quality. E.g., the time information may be guaranteed to be correct with a certain probability, or even probability distributions can be given. Typically, the term *time uncertainty* is used. In sensor networks the use of guaranteed time intervals can be very attractive.

Synchronization techniques: Taking one sample

Unidirectional Synchronization:

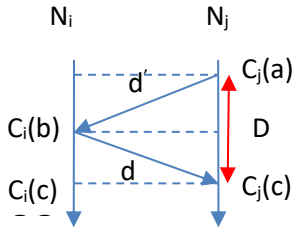


Node N_j is not familiar with d, its knowledge is only the fact, that the clock of node N_i displayed the value $C_i(a)$ before the clock of node N_j displayed $C_j(b)$. To perform synchronization, we must estimate either the value of $C_j(a)$ or $C_i(b)$. If the limits $d_{min} \leq d \leq d_{max}$ are known, then

$$\hat{C}_j(a) \approx C_j(b) - \frac{d_{min}+d_{max}}{2} \text{ or } \hat{C}_i(b) \approx C_i(a) + \frac{d_{min}+d_{max}}{2}.$$

Having these estimates, the clock of node N_j should be modified by $\hat{C}_j(a) - C_i(a)$ or $C_j(b) - \hat{C}_i(b)$. If the communication jitter (d_{max}-d_{min}) is large, then the synchronization will be inaccurate, because the lower bound of $C_j(a)$ will be $C_j(b) - d_{max}$, and the upper bound will be $C_j(b) - d_{min}$, which is a wide range.

Bidirectional (round trip) synchronization:



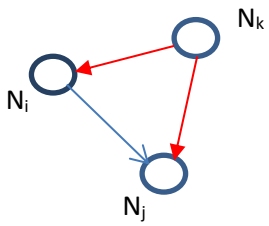
Here node N_j knows that $0 \leq d \leq D$. $D = C_j(c) - C_j(a)$. If $d_{min} \leq d \leq d_{max}$, then max $(D - d_{max}, d_{min})$ and min $(d_{max}, D - d_{min})$ give the limits of d. The estimate that can be computed here.

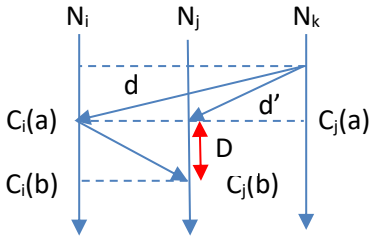
$$\hat{C}_j(b) \approx C_j(c) - \frac{D}{2},$$ having lower bound $C_j(c) - (D - d_{min})$, and upper bound $C_j(c) - d_{min}$.

The clock of node N_j is to be modified by $\hat{C}_j(b) - C_i(b)$. With such a method the quality of the synchronization will be better. The worst-case synchronization error: $\frac{D}{2} - d_{min}$, that can be proved using the above figure. The method can be improved using the so-called probabilistic time synchronization, where node N_j after receiving the timestamp checks weather the value of $\frac{D}{2} - d_{min} <$ than a defined threshold. If not, then the request will be repeated.

Reference broadcasting synchronization:

In this case also a so-called *beacon* node N_k is involved. The beacon sends a broadcast message to the other nodes. The delays are almost equal: $d \approx d'$.





Thus

$$\hat{C}_i(b) \approx C_i(a) + D,$$

i.e., the clock of node N_j should be modified by $C_j(b) - \hat{C}_i(b)$. It is n important property, that the synchronization of node N_j is performed without using it radio channel.

Synchronization of multiple nodes:

(1) Single-hop synchronization with a set of master nodes which are synchronized out of ban (e.g. using GPS); (2) Partitioning the network into clusters: all nodes within a cluster can broadcast messages to all other members of the cluster and thus reference-broadcast techniques can be used to synchronize the cluster internally. Some nodes are members of several clusters and participate independently in ll corresponding synchronization procedures. These nodes act as time gateways to translate time stamps from one cluster to the other.; (3) Tree construction: The most common solution of the multi-hop synchronization problem is to construct a synchronization tree with a single master at the root. The accuracy degrades with the hop distance from the root.

7. Embedded operating systems

7.1. Software aspects of Embedded Systems: Typical software architectures

Concerns: computational capacity, memory size (RAM, ROM), development, flexibility, reaction time to external, asynchronous event, (memory) protection, recursion, re-entrant calls, processor utilization, etc.

Features used in characterization:
- maximum response time,
- handling hardware,
- inter task communication,
- design technologies,
- application area.

Classification of software architectures: periodic, priority-based, event-triggered, time-triggered.
Practical implementations:
- cyclic architecture
- cyclic architecture together with external interrupt
- function queue scheduling
- RTOS

Cyclic architecture
- round-robin
- weighted round-robin
- time-triggered round-robin
- strictly time-triggered
- round-robin with interrupt

Simple cyclic architecture: the processor runs in an infinite loop, even if there is no service request.

```
void main() {
while (TRUE){
        if (DeviceA_Needs_Service()) {Service_A};
        if (DeviceB_Needs_Service()) {Service_B};
        if (DeviceC_Needs_Service()) {Service_C};
        ...
        }
}
```

Properties:
- maximum response time: tA+ tB+ tC+ ..., i.e. maximum cycle time.
- hardware is handled with polling
- inter-task communication with shared variables (non preemptive!)
- development: hard
- hard RT behaviour: slow (e.g. printer task) (however, it can be RT)
- processor utilization:100% (this is wrong!)
- application area: where the time constant of the system is larger than the cycle time (fast and rare events).

Weighted round-robin: the more frequent tasks can be called more than once within the cycle.

```
void main() {
while (TRUE){
        if (DeviceA_Needs_Service()) {Service_A};
        if (DeviceB_Needs_Service()) {Service_B};
```

```
        if (DeviceA_Needs_Service()) {Service_A};
        if (DeviceC_Needs_Service()) {Service_C};
        if (DeviceA_Needs_Service()) {Service_A};
        ...
        }
}
```

Properties:

- maximum response time: tA+ tB+ tA+ tC+ tA+.., but for more frequent tasks it is less than the maximum cycle time.
- hardware is handled with polling
- inter-task communication with shared variables (non preemptive!)
- development: hard
- processor utilization:100% (this is wrong!)
- further property: priority-like behaviour, still not preemptive

Time-triggered round-robin: The boundaries of the cycle are determined by a timer. For every timer interrupt the cycle runs once or several times. The cycle itself can be weighted round-robin.

Properties:

- maximum response time: cycle period.
- hardware is handled with polling
- inter-task communication with shared variables
- development: hard
- hard RT behaviour: slow (e.g. printer task) (however, it can be RT)
- processor utilization: <100% (standby exists)

Strictly time-triggered: Every task starts running at predefined time instant. Administration: the predefined time instants and function references are collected into a table, which is used by a micro run-time system to start the scheduled task at right time.

Properties:

- maximum response time: the scheduling rate of the task + its computation time
- inter-task communication with shared variables
- development: hard
- processor utilization: <100% (standby exists)
- hard real-time behaviour: OK, its application is typical in safety-critical systems

Round-robin with interrupt: signalling is performed with interrupt instead of polling

```
FLAG A, B;
void interrupt A_Handler() { Handle_HW_A(); A=TRUE; }
void interrupt B_Handler() { Handle_HW_B(); B=TRUE; }
void interrupt C_Handler() { Handle_HW_C(); C=TRUE; }
void main() {
while (TRUE){
        if A {A=FALSE; Service_A(); }
        if B { B=FALSE; Service_B(); }
        if C { C=FALSE; Service_C(); }
        ...
        }
}
```

Properties:

- maximum response time: tA+ tB+ tC+ ...(+ IT) only the signalling will be faster, the service not
- hardware is handled with interrupt, priority assignment to interrupts is possible

- inter-task communication with shared variables: no problem. Between interrupt and task: pre-emption (shared variables may cause problems)
- development: concerning interrupts it is good, but by introducing new tasks time conditions will change
- application field: if the execution time of the tasks is nearly the same. This is the most widely used solution.

Function-Queue Scheduling

```
void interrupt A_Handler() { Handle_HW_A(); PutFunction(Service_A); }
void interrupt B_Handler() { Handle_HW_B(); PutFunction(Service_B); }
void interrupt C_Handler() { Handle_HW_C(); PutFunction(Service_C); }
void Service_A();
void Service_B();
void Service_C();
void main() {
        while (TRUE){
                while (IsFunctionQueueEmpty());
                        CallFirstFromQueue();
        }
}
```

Properties:

- maximum response time: execution time of the longest task + the execution time of the *i*th task
- hardware is handled with interrupt
- inter-task communication with shared variables: no problem. Between interrupt and task: pre-emption (shared variables may cause problems)
- development: easy
- The service order from the queue can be: (1) FIFO, (2) priority based
- drawback: non preemptive
- processor utilization:100%

Question: how to modify to have less than 100%?

Software based on Real-time Operating System

```
void interrupt A_Handler() { Handle_HW_A(); Signal_A(); }
void interrupt B_Handler() { Handle_HW_B(); Signal_B(); }
void Service_A();
void Service_B();
void task_A(void) {
        while (TRUE){
                Wait_for_Signal_A(); Service_A();
        }
}
void task_B(void) {
        while (TRUE){
                Wait_for_Signal_B(); Service_B();
        }
}
```

Properties:

- maximum response time: a feature of the operating system (~10 μsec) + the execution time of the task together with the sum of the execution time of higher priority tasks
- hardware is handled with interrupt

- inter-task communication: using RTOS communication functions. This solves synchronization, as well.
- development: easy
- hard real-time behaviour: good
- processor utilization:<100% (In idle state sleep is possible)
- application area: everywhere
- drawback: The Operating System (OS) involves additional code and time

Terms:

embedded OS:	small resource requirement (micro-controllers (μC) are enough)
real-time OS:	provides finite, deterministic response time
task:	series of inter-dependent activities
job:	parts, subunits of the tasks
process:	unit of schedule with separate memory block (implementation of tasks)
thread:	unit of schedule without separate memory block
kernel:	the key components of the OS
scalability:	the services of the OS can be switched on/off in compilation time; availability with source code

The role of the kernel:

- to provide parallel programming environment,
- scheduling,
- inter-task communication,
- handling interrupts,
- handling timers, and timing services,
- (possibly) memory management

Further OS features:

- handling peripherals and system programs (APIs: Application Programming Interfaces)
- handling communication channels
- virtual memory management, file system, etc.

7.2. Comparison of desk-top OSs and embedded OS-s

a. Desk-top OS-s are not suitable in embedded systems, because:

- the services are too extensive;
- non-modular, non-fault tolerant, non-configurable, non-modifiable;
- require large memory;
- not optimized for power consumption;
- are not designed for mission-critical application;
- timing uncertainties are too large.

b. **Configurability** is needed because:
- a single OS is unable to meet all the requirements;
- the overhead caused by the unused functions and data is not tolerable;
- there are many embedded systems not having disc, keyboard, display, mouse.

Typical **tools of configuration**:
- removal of the unnecessary functions e.g. by the linker;
- applying conditional compilation (using #if and #ifdef commands);

Comment: The verification of systems, having operating systems generated by configuration, is difficult:
- every OS generated by configuration should be thoroughly tested.

- E.g. the number of the configuration points of the OS eCos (an open source RT OS of Red Hat) is between 100 and 200.

c. The device handlers of the embedded OS are handled by the tasks, and not by the integrated drivers;
- the predictability is improved, if everything is handled by the scheduler;
- practically there is no such a device, which would be supported by all versions of the OS, apart from the timer.

Embedded RTOS	Standard OS
application software	application software
middleware	middleware
devices drivers	devices drivers
real-time kernel	device drivers

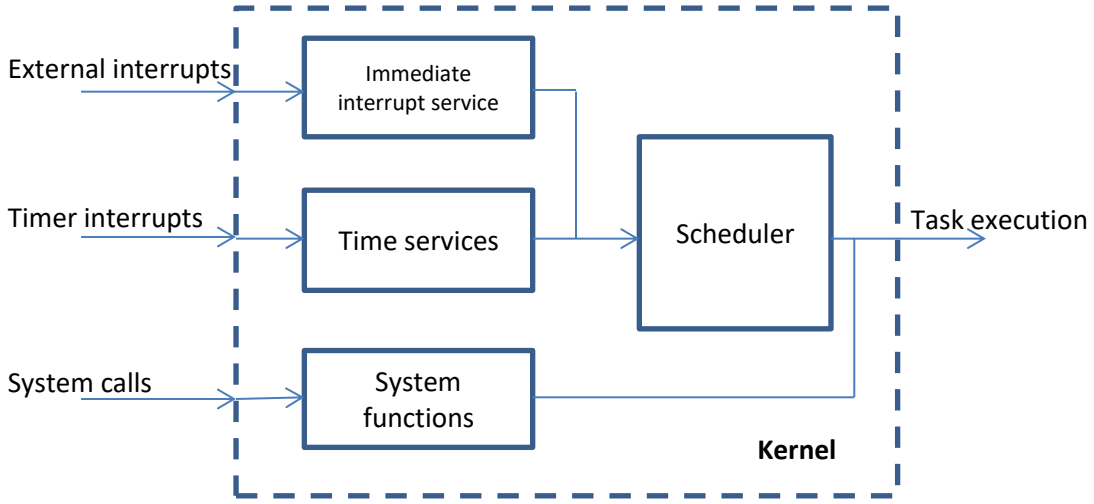
d. In embedded systems **every task can use interrupt**:
- In standard OS this would be a serious source of unreliability;
- Embedded programs are supposed to be tested;
- It is allowable that an interrupt starts or stops tasks by putting the start addresses of the tasks into the interrupt table. This is more efficient and predictable than via OS functions.

Comment:
- However, composability will fail: if the run of a task depends on an interrupt, then it is difficult to add another task to be started by the very same event.
- If RT processing is a concern, then the time required by the interrupt services should also be considered. In this case the interrupts should also be handled by the scheduler.

e. In embedded operating systems **the protecting mechanisms are not necessary** in every case:
- The embedded systems are designed for dedicated purposes, untested programs are rarely used, the software is reliable.
- There is no need for privileged I/O instructions, the tasks can manage the I/O operations related to them.

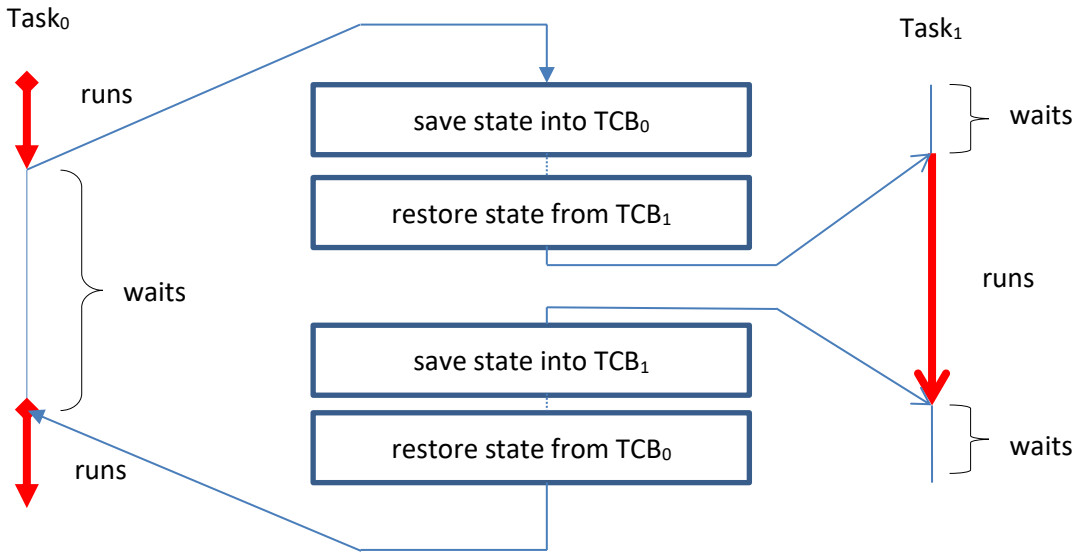
i- However, in case of security concerns protecting mechanisms might be required.

f. **The real-time operating systems (RTOS)** support creating real-time systems. Requirement:
- The time-domain behaviour is predictable: the maximum execution time of every operating system function should be known. The RTOS behaves in a deterministic way, almost all activity is supervised by the scheduler.
- the RTOS manages timing and scheduling: to do this it should familiar with deadlines of the tasks, and should provide high-resolution timing services.
- The RTOS should be fast (due to practical considerations).
- The RTOS should provide process management functions, so-called Application Program Interface (API), like:
  create_thread,
  suspend_thread,
  destroy_thread; …
  create_timer,
  timer_sleep,
  timer_notify; …
  open,
  read; …
  other system calls.



The role of the kernel:

- Execution of concurrent (quasi-parallel) programs in forms of tasks or threads:
  - by handling the states of tasks/threads, and by ordering them into queues;
  - by executing preemptions (fast context switching, see Figure below) and fast interrupt handling;
- Scheduling the CPU (guaranteeing deadlines, minimizing waiting, reasonable distribution of computing power);
- Synchronizing tasks (Critical sections, semaphores, monitors, mutual exclusion);
- Inter-task communication (buffering);
- Supporting RT clock serve as internal reference.



TCB: task control block: contains the run-time environment (context, state) of a task. While it is running, it is typically loaded into the registers of the processor, and saved into the memory in case of pre-emption.

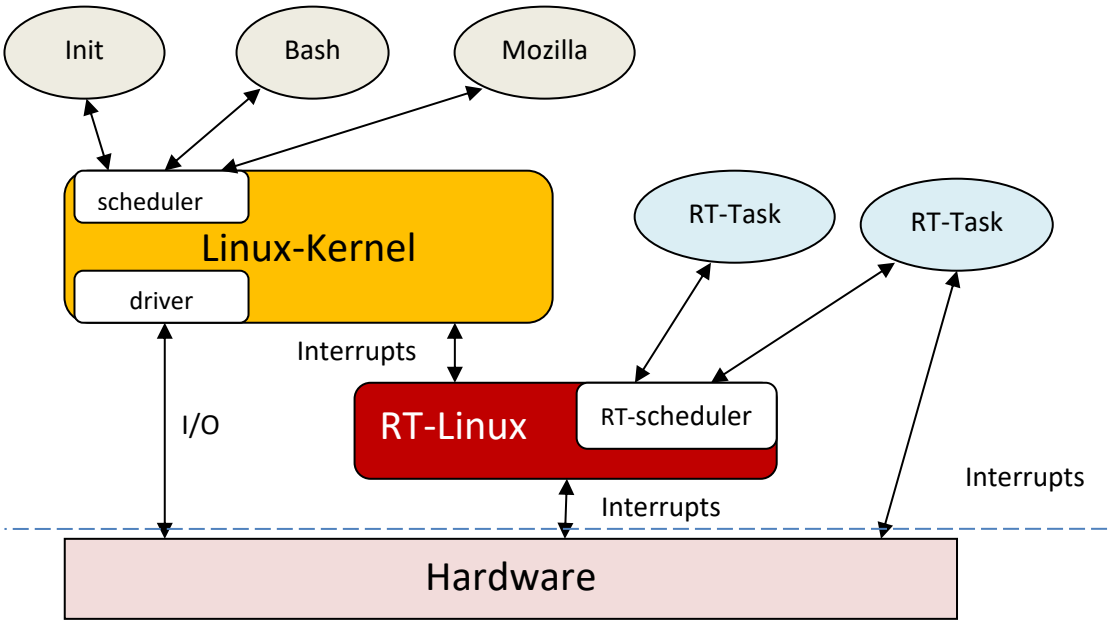
g. **Real-time extensions of standard operating systems**

A possible solution: Every real-time task is executed by a real-time kernel, and the standard operating system is a single task:

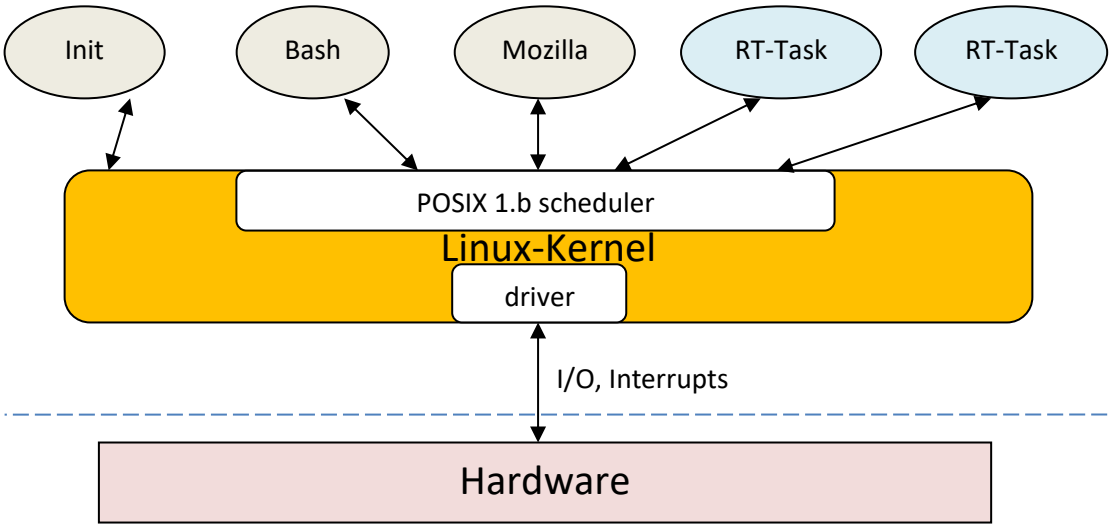
RT-task 1	RT-task 2	non-RT task 1	non-RT task 2
device driver	device driver	Standard-OS	
real-time kernel			

Comments:
- Problems within the standard OS do not influence the execution of the real-time tasks.
- Since the real-time tasks are unable to use the standard OS function this solution is below the expectations.

Example: RT Linux



Example: Posix 1.b RT-extensions to Linux

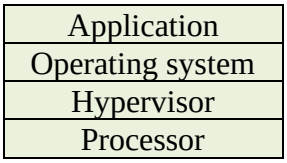


The ordinary Linux scheduler cn be replaced by the POSIX scheduler, which provides priority fro the real-time tasks. Among the standard OS functions special real-time functions are also offered. Programming is

simple, however, there is no guarantee to meet the deadlines. (POSIX: "**P**ortable **O**perating **S**ystem **I**nterface for uni**X**".)

Virtualization in embedded systems

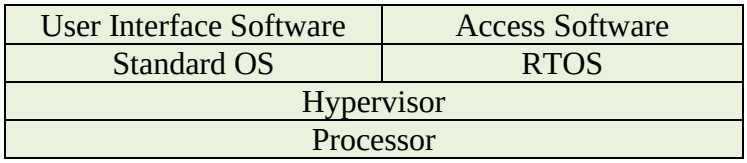
Virtualization: supports portability, i.e. the use of software on different hardware platforms. A so-called virtual machine (VM: Virtual Machine) provides such a software environment for the given software, like it would run on a real hardware in the following structure:



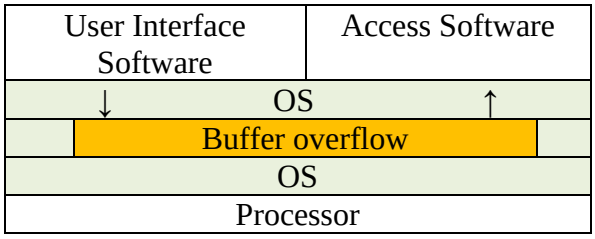
The software layer, which provides the virtual environment is the so-called virtual machine monitor (VMM) or hypervisor. It has three key features:
- provides an identical software environment than the original machine;
- at maximum it runs slower;
- completely supervises the system resources.

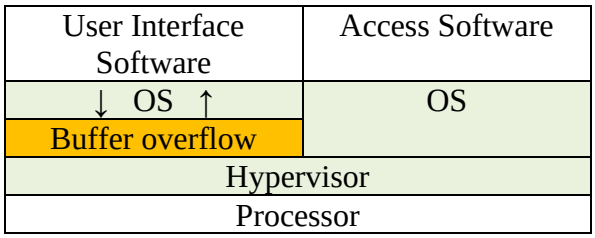
The majority of the VM instructions are immediately executable on the hardware applied, some of them requires interpretation. Among them there are:
- control-sensitive instructions, which modify the privileged machine states, therefore interfere with the supervision of the resources by the hypervisor.
- behaviour-sensitive instructions, which can reach (read) the privileged machine-states.

Concurrent operating systems on virtual machines



Improvement of security using virtualization





The error caused by an application does not propagate towards another, because it has its own operating system.

Licence separation using virtualization

User Interface Software	Access Software
Linux GPL	RTOS
Stub	Driver
Hypervisor	
Processor	

GPL: General Public Licence. Linux is licenced under the GPL which requires open-sourcing of all delivered code. Therefore, all software written using Linux is open-source. Where this would cause problems, the Linux and the dedicated application run on different virtual machines. A stub (or proxy) driver is used to forward Linux driver requests to the real device driver, using hypercalls.

Limits of virtualization in embedded systems:

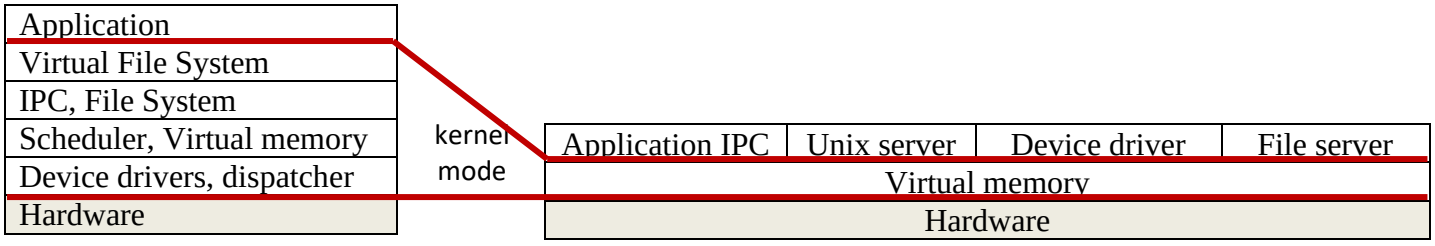
- Using more operating systems together with applications of increasing complexity results in large amount of code, which can be a source of error, requires larger memory, consumes more energy.
- The different subsystems should intensively cooperate: separated implementation does not fit.
- The inter-subsystem communication is not supported by the virtual machine.
- To share common resources among subsystems is hard to organize if more operating systems re running parallel.
- Due to the virtualization scheduling is performed at two levels: (1) Between the hypervisor and the VM, (2) within every operating system running on the VM.
- The fulfilment of critical security requirements is not supported by the virtualization alone. The critical code segments (so-called trusted computing base, TCB) must be executed in privileged mode on the processor. The hypervisor is part of the TCB. The correctness of such a code should be proved.
- Virtualization increases the size of the code.

What kind of supporting software is required by the embedded?

- support for virtualization with all its benefits;
- support for lightweight but strong encapsulation of medium-grain components that interact strongly, to build robust systems that can recover from faults;
- high-bandwidth, low-latency communication, subject to configurable, system-wide security policy;
- global scheduling policies interleaving scheduling policies of threads from different subsystems;
- ability to build subsystems with very small trusted computing base (TCB).

Microkernel (microvisor) technology: a better solution to embedded systems

A microkernel (microvisor) is a minimal privileged software layer that provides only general mechanisms. Actual system services and policies are implemented on top in user-mode components. A microkernel is defined by Liedtke's minimalism principle (1995): A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality. This minimality implies that a microkernel does not offer any services, only the mechanisms for implementing services. The microkernel approach leads to a system structure that differs significantly from that of classical "monolithic" operating systems:



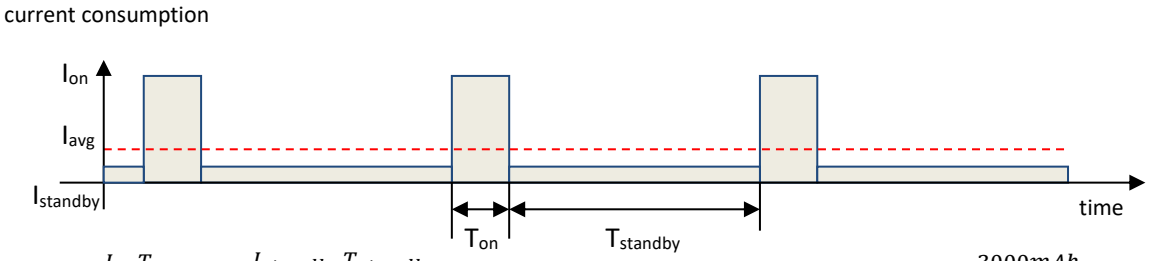
The classical structures have a vertical structure of layers, each abstracting the layers below, a microkernel-based system exhibits a horizontal structure. System components run beside application code, and are invoked by sending messages. In case of a microkernel system there is no real difference between "system

services" and "applications", all are simply processes running in user mode. Each such user-mode process is encapsulated in its own hardware address space, set up by the kernel. It can only affect other parts of the systems (outside its own address space) by invoking kernel mechanisms, particularly message passing (Inter Process Communication, IPC). It can only directly access memory or other resources if they are mapped into its address space via a system call. A more detailed description about microkernels, e.g.:
https://gdmissionsystems.com/cyber/products/trusted-computing-cross-domain/microvisor-products/

8. Power Aware Systems – Low Power Design

Example: The capacity/capability of two AA (Mignon) type battery cells in sensor network applications (microcontroller+radio+sensors):

2 pieces of AA battery cells have an average capacity of 3000 mAh. How long can we use our system in a day, if we require a total availability of services for minimum 1 year (8760 hours), while P_{on}=150 mW (I_{on}=50mA) and $I_{standby}$=50µA.

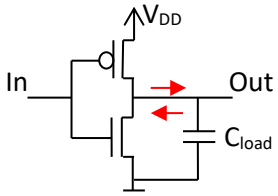


$$I_{avg} = \frac{I_{on}T_{on}}{T_{on}+T_{standby}} + \frac{I_{standby}T_{standby}}{T_{on}+T_{standby}} = I_{on}\lambda + I_{standby}(1-\lambda), \quad I_{avg\,max} = \frac{3000mAh}{8760h} = 342\mu A$$

$$\lambda = \frac{I_{avg\,max}-I_{standby}}{I_{on}-I_{standby}} = 0.0058 \approx 0.6\% \qquad \approx 8\frac{minutes}{day}.$$

If we take measurements in every hour, then they can take 20 seconds.

Power Consumption of a CMOS Gate:



The two FETs are alternately open and closed. Main sources of power consumption: (1) charging and discharging capacitors, (2) short circuit path between supply rails during switching, (3) leaking diodes and transistors (becomes one of the major factors due to shrinking feature sizes in semiconductor technology). Power consumption of CMOS circuits (ignoring leakage):

$$P \sim \alpha C_L V_{DD}^2 f,$$

where V_{DD}: stands for power supply, α: switching activity (for a clock it is 1), C_L: a load capacity, f: clock frequency. Delay for CMOS circuits:

$$\tau \sim C_L \frac{V_{DD}}{(V_{DD}-V_T)^2}, \qquad \text{where } V_T\text{: threshold voltage, } V_T \ll V_{DD}.$$

It can be stated:

- Decreasing V_{DD} reduces P quadratically (f constant);
- The gate delay increases only reciprocally;
- Maximal frequency f_{max} decreases linearly.

Potential for Energy Optimization (Dynamic Voltage Scaling: DVS):

$$P \sim \alpha C_L V_{DD}^2 f\,, \qquad E \sim \alpha C_L V_{DD}^2 f t = \alpha C_{Lt} V_{DD}^2 (\#cycles).$$

Saving energy for a given task:

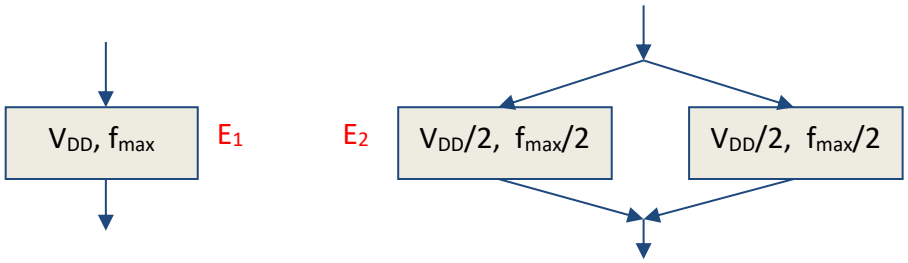
- reduce the supply voltage V_{DD};
- reduce switching activity;

- reduce the load capacitance;
- reduce the number of cycles (#cycles).

Power Supply Gating:

It is one of the most effective ways of minimizing static power consumption (leakage). Power Supply Gating cuts-off power supply to inactive units/components: a header switch provides virtual power, while a footer switch provides virtual ground and thus reduces leakage.

Use of Parallelism: Duplicated hardware with half of the supply voltage, and half of the clock frequency.

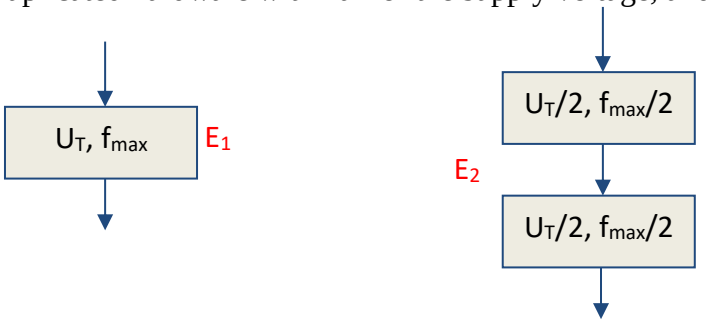


$$E_1 \sim V_{DD}^2(\#cycles), \qquad E_2 = \frac{E_1}{4}.$$

The number of the operations and the delays is the same, the energy consumption is lowered to its fourth.

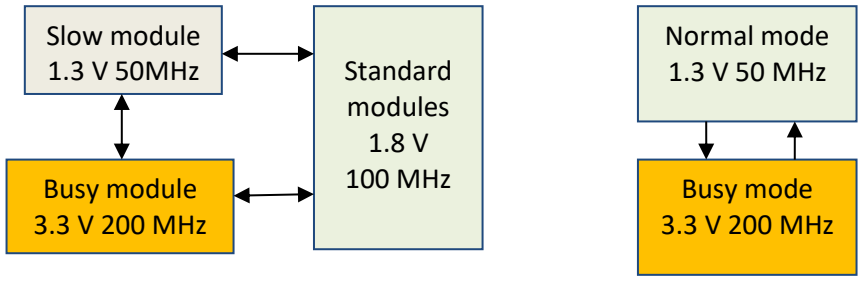
The application of Very Long Instruction Word (VLIW) architectures is an alternative: parallel instruction sets are applied.

Use of Pipelining: Duplicated hardware with half of the supply voltage, and half of the clock frequency.

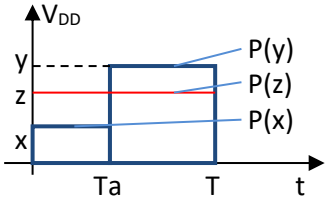


The number of the operations is the same, the energy consumption is lowered to its fourth.

Not all components require same performance. Required performance may change over time



Optimal strategy (Dynamic Voltage Scaling):



Case A: execute at voltage *x* for *Ta* time units, and at voltage *y* for (1-*a*)*T* time units.
The energy consumption: $\boxed{T(aP(x) + (1-a)P(y)).}$
Case B: execute at voltage $z = ax + (1-a)y$ for *T* time units.
The energy consumption: $\boxed{TP(z).}$

Since the power is a convex (quadratic) function of V_{DD}, therefore $P(z) < aP(x) + (1-a)P(y)$, i.e. it worth executing at constant voltage. (The linear combination gives a string above the parabola.)

Example:

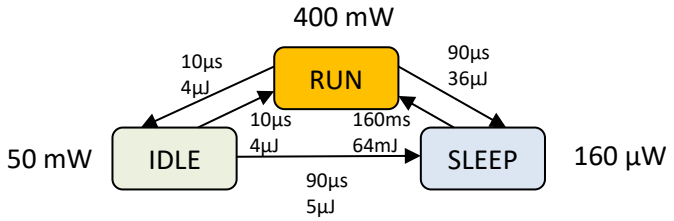
V_{DD} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
cycle time [ns]	20	25	40

Task execution needs 10^9 cycles within 25 seconds.

a. Complete task ASAP: 10^9 cycles @ 50 MHz.
   Energy consumption: $E_a = 10^9 * 40 * 10^{-9} = \mathbf{40}$ [J], time requirement: $10^9 * 20 * 10^{-9} = \mathbf{20}s$.
b. Execution at two voltages: 0.75*10^9 cycles @ 50 MHz + 0.25*10^9 cycles @ 25 MHz.
   Energy consumption: $E_a = 0.75 * 10^9 * 40 * 10^{-9} + 0.25 * 10^9 * 10 * 10^{-9} = \mathbf{32.5}$ [J],
   time requirement: $0.75 * 10^9 * 20 * 10^{-9} + 0.25 * 10^9 * 40 * 10^{-9} = \mathbf{25}s$.
c. Execution at optimal voltage: 10^9 cycles @ 40 MHz.
   Energy consumption: $E_a = 10^9 * 25 * 10^{-9} = \mathbf{25}$ [J], time requirement:$10^9 * 25 * 10^{-9} = \mathbf{25}s$.

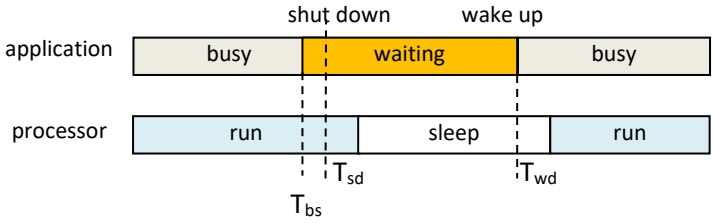
Comment: Obviously some spare-time is always required at task executions.

Dynamic Power Management (DPM): tries to assign optimal power saving states



Requires hardware support.
Example: **StrongARM SA1100**
IDLE: a software routine may stop the CPU when not in use, while monitoring interrupts.
SLEEP: shuts down on-chip activities.

Example: reduce power according to workload:



T_{bs}: time before shutdown
T_{sd}: shutdown delay
T_{wd}: wakeup delay

Comment: Shutdown only if long idle times occur. There is a trade-off between savings and overhead "costs".

Example: Dynamic power management:

Suppose that the power consumption $P(f)$ of a given CMOS processor at frequency f is:

$$P(f) = \left[10\left(\frac{f}{100MHz}\right)^3 + 20\right] mWatt$$

To reduce the power consumption, one can adjust the execution frequency. The maximum/minimum available frequency:

$$f_{max} = 1000MHz / f_{min} = 50MHz$$

Frequency switching has negligible overhead and the processor can operate at any frequency between $50MHz$ and $1000MHz$. One can also apply dynamic power management to turn the processor to the sleep

mode (or turn the processor off) to reduce the power consumption. When the processor is in the sleep mode, it consumes *no power*. However, turning the processor on to the run mode requires additional energy consumption, i.e. $3 \times 10^{-5} Joule$. (Switching from run mode to sleep mode consumes no energy.) Turning on/off the processor can be done instantly.

The system has three jobs to execute:

	arrival time	deadline	execution cycles
τ_1	0	2ms	100000
τ_2	2ms	6ms	100000
τ_3	6ms	7ms	80000

The processor is in the run mode at time 0 and is required to be in the run mode at time 7 ms.

Problem#1:

The energy consumption to execute *C* cycles is $\frac{CP(f)}{f}$. There is a critical frequency f_{crit} between $50MHz$ and $1000MHz$ at which the energy consumption to execute any *C* cycles is minimized. What is the critical frequency of the processor?

Solution#1:

As we lower the frequency the power consumed falls. But beyond a critical frequency, the rate of fall in power is overweighed by the fall in frequency and thus the power consumed per cycle increases. To compute this critical frequency, we have to minimize $\frac{P(f)}{f}$. Let f be normalized to $100MHz$. The first derivative of $\frac{P(f)}{f}$ is $20f - \frac{20}{f^2}$, which equals 0 if $f = 1$. Thus $f_{crit} = 100MHz$.

Problem#2:

When the processor is idle at frequency f_{min} for *t* seconds, then the energy consumption is $P(f_{min}) \times t$. The *break-even time* is defined as the minimum idle interval for which it worth turning the processor off. What is the *break-even time* of the processor?

Solution#2:

Going into the sleep mode must provide sufficient energy saving to compensate for the additional energy consumption (overhead) of $3 \times 10^{-5} + 0\, Joule$.

Energy(idle state, f_{min}) – Energy(sleep state) ≥ Energy(turning from sleep to run state)

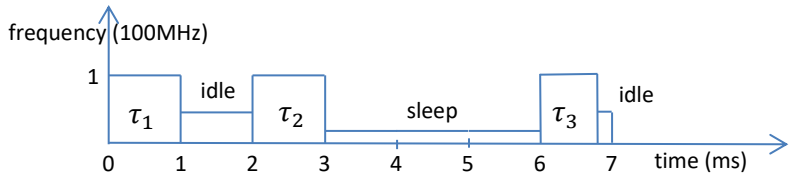
$$P(f_{min}) \times t_{bev} - 0 \geq 3 \times 10^{-5} Joule$$

$$t_{bev} \geq \frac{3 \times 10^{-5} Joule}{10^{-3} \times (10 \times 0.5^3 + 20) Watt} = 1.412ms.$$

Problem#3:

A *workload-conserving* schedule is defined as a schedule which always executes some jobs when the ready queue is empty. Provide the workload conserving schedule for the 3 tasks which minimizes the energy consumption without violating the timing constraints. For this apply the *critical frequency* f_{crit} as the frequency for active task execution. What is the energy consumption of the schedule?

Solution#3:



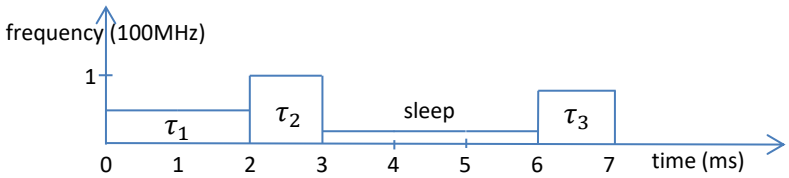
Since $P(f_{crit}) = 30mWatt$, and $P(f_{min}) = 21.25mWatt$, thus the energy consumption of the schedule is $(30 \times 1 + 21.25 \times 1 + 30 \times 1 + 3 \times 10^1 + 30 \times 0.8 + 21.25 \times 0.2)\mu Joule = 0.1395\, mJoule$. Note that the interval [1,2] is shorter than the break/even time, therefore there is no reason to switch to sleep mode.

Problem#4:

Would it be possible to provide another workload-conserving schedule without violating the timing constraints for the 3 tasks, and with less energy consumption?

Solution#4:

Yes, the solution is to use the convex nature of the power consumption: to slow down execution of tasks τ_1 and τ_3 to such extent as to avoid idle times after their executions. Thus, even though we work below the critical frequency, we can save on energy consumption. The energy consumption of the schedule is (see figure):



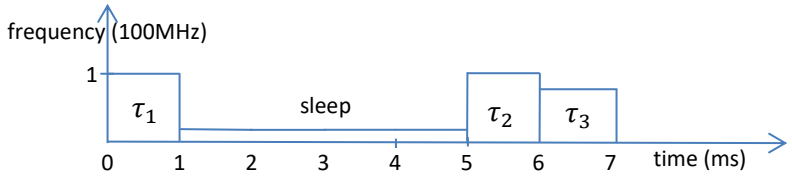
$(21.25 \times 2 + 30 \times 1 + 3 \times 10^1 + (10 \times 0.8^3 + 20) \times 1)\mu Joule = 0.12762mJoule$.

Problem#5:

Would it be possible another schedule without violating the timing constraints for the 3 tasks that is not workload-conserving but the energy consumption is even lower than the optimal work/load conserving schedule?

Solution#5:

Yes, the idea here is to *batch* the sleep mode into one block, and execute τ_1 with critical frequency. This illustrates that to conserve energy; workload-conserving strategies are not necessarily the best. The energy consumption of the schedule is (see figure):



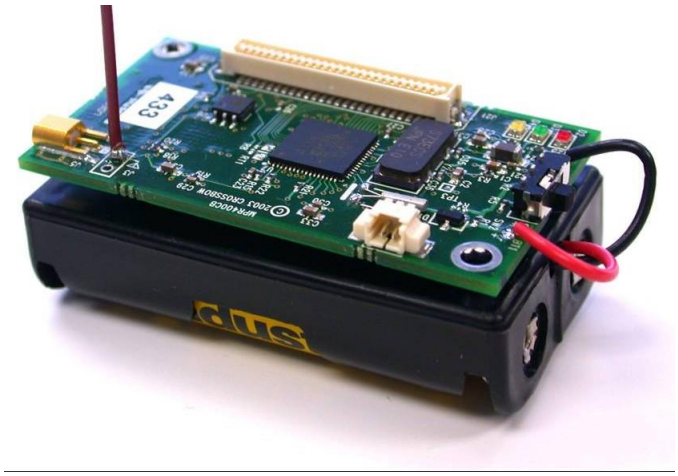
$(30 \times 1 + 3 \times 10^1 + 30 \times 1 + (10 \times 0.8^3 + 20) \times 1)\mu Joule = 0.115120mJoule$.

9. Sensor networks

The availability of cheap sensors to measure almost all possible quantities resulted in systems consisting (sometimes of large amount) of nodes capable to measure, pre-process and communicate data coming from the environment. A typical example of such a node is the Berkeley Mica2 mote (see picture). It size can be estimated by the size of two AA type batteries located underneath the printed circuit board. Concerning this type of devices, very many information is available on the internet. The sensor network applications can be characterized by relatively large spatial scope, large number of nodes, and limited availability of local power.

Examples of applications:

1. Detection of acoustic shockwaves. Shooter localization → Counter-sniper system
2. Detection of shooting at elephants → help authorities to catch poachers
3. Intelligent rock bolt monitoring
4. Active noise control



The TinyOS operating system
(A detailed slide-set is available on the webpage of the subject. Plenty of similar presentations including many references can be found on the internet.)

Why it is needed?
Traditional operating systems have difficulties in case of sensor networks, because multi-threaded architecture cannot be used with sufficient efficiency, they need large memory, and do not support the minimization of power consumption. In case of sensor networks the followings are important: (1) concurrent execution, (2) efficiency of energy utilization, (3) small memory footprint, and (4) the support of various utilization.

Major properties of TinyOS:

The TinyOS is an open-source operating system, which was designed for sensor network applications. It is component-based, written in NesC (Networked embedded system C) language within a cooperation of University of California, Berkeley and Intel Research.

The components-based architecture allows frequent changes while keeping the size of the code minimum. Program execution is event-based, hence support high concurrency. Power efficient, because the processor is sent into sleep mode as soon as possible. It has small footprint, because it applies a FIFO-based non-preemptable scheduling.

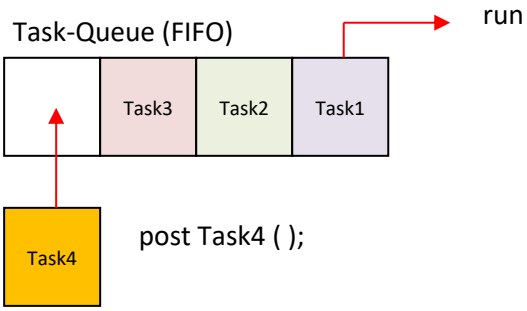
TinyOS uses static memory allocation, memory requirements are determined at compile time. This increases runtime efficiency. Local variables are saved on the stack.

Power-aware, two-level scheduling is applied: (1) Long running tasks and interrupt events, (2) Sleep unless tasks in queue, wakeup on event. Tasks are time-flexible, background jobs, atomic with respect to other tasks, can be pre-empted by events. Events are time-critical, shorter duration program sections, with LIFO (Last-in First-Out) semantic (no priority), can post tasks for deferred execution.

Programs are built out of components, each component specifies an interface, interfaces are "hooks" for wiring components to result in a configuration.

Components should use and provide bidirectional interfaces. Components should call and implement commands and signal and handle events. Components must handle events of used interfaces and also provide interfaces that must implement commands.

Component hierarchy: Command flow "downwards", they are non-blocking requests, and the control returns to the caller. Events flow upwards, post tasks (function queue scheduling), signal higher level events, and call lower level commands. The control returns to signaller.



To avoid cycles: events can call commands, commands can NOT signal events.

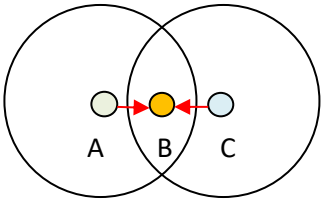
Communication in sensor networks

Standardized solutions: typically, using ISM (Industrial, Scientific, Medical) 2.4 GHz, spread spectrum: ZigBee/IEEE 802.15.4, IEEE 802.11b (Wi-Fi) WLAN (Wireless Local Area Network), Bluetooth WPAN (Wireless Personal Area Network).

Media Access Control: Dynamic (on demand): e.g. CSMA (Carrier Sense Multiple Access). Collision avoidance: Before sending checks the channel. If information flow is detected: waits.

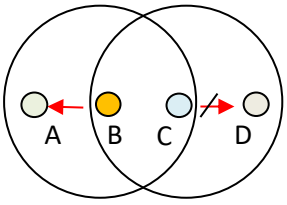
CSMA problems:

Problem of hidden terminal:



- A sends message to B
- C does not hear A!
- C also sends message to B
- B is unable to receive messages

Problem of visible terminal:



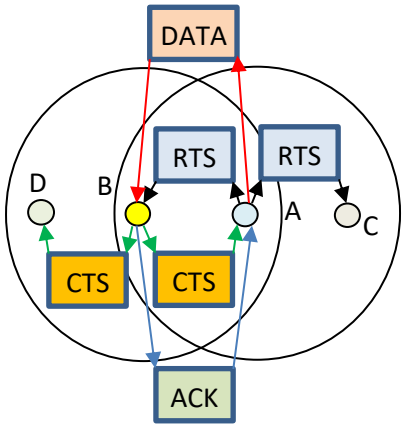
- B send message to A
- C would like to send message to D
- C hears B
- C does not send message, even if it would be possible

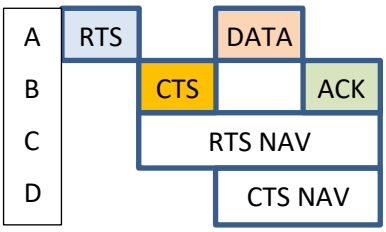
CSMA modifications

CSMA together with availability indication: two channels are used, one for data transmission, and another for indicating availability. The receiver continuously signalizes availability. The sender before transmission checks both channels. The node simultaneously sends and receives that is expensive. Two simultaneous channels require wider bandwidth.

Request To Send/Clear To Send (RTS/CTS): Operates in two phases: (1) Handshake, (2) Data transmission. The basic idea is: collision occurs at the receiver. Avoids the hidden terminal problem. It is advantageous in case of longer messages, otherwise the overhead is too large.

Its operation:





- Sender „A" sends RTS message to "B"
- Receiver „B" replies with CTS message
- Receiving CTS the sender "A" transmits data

The other nodes are not allowed to transmit after receiving RTS or CTS!

(NAV = Network Allocation Vector)

Routing in sensor networks

Sensor networks are ad-hoc. Node distribution is random, connections come into being randomly, they are not reliable (fading), they can be mobile, and systems can consist of large number of nodes.

Typical configurations:

- Single source → multiple (possibly all) destinations. E.g. a central node propagates commands within the network.
- Multiple source → single destination. E.g. data collection and transmission into a central node.
- Single source → single destination. E.g. data exchange between nodes.

Data transmission strategies:

- *Time-triggered:* Sensor activity and data transmission is time-triggered. E.g. data collection. It is advantageous to conserve energy: sleep mode together with synchronized wakeup.
- *Event triggered:* It is used in time-critical applications. Sensor activity is due to some event in the environment. The reduction of energy consumption is more difficult.
- *Polling:* Sensors are activated by a command of a central node.

Typical network structures:

- *Flat:* The nodes have equal rights; Scaling of the network is hard
- *Hierarchical:* Clusters are formed: The communication is solved within the clusters and among the clusters separately. Among the clusters the so-called control nodes communicate. They have extra capabilities. The role of controlling a cluster can be dynamically changed.

Flat: *Flood routing:*

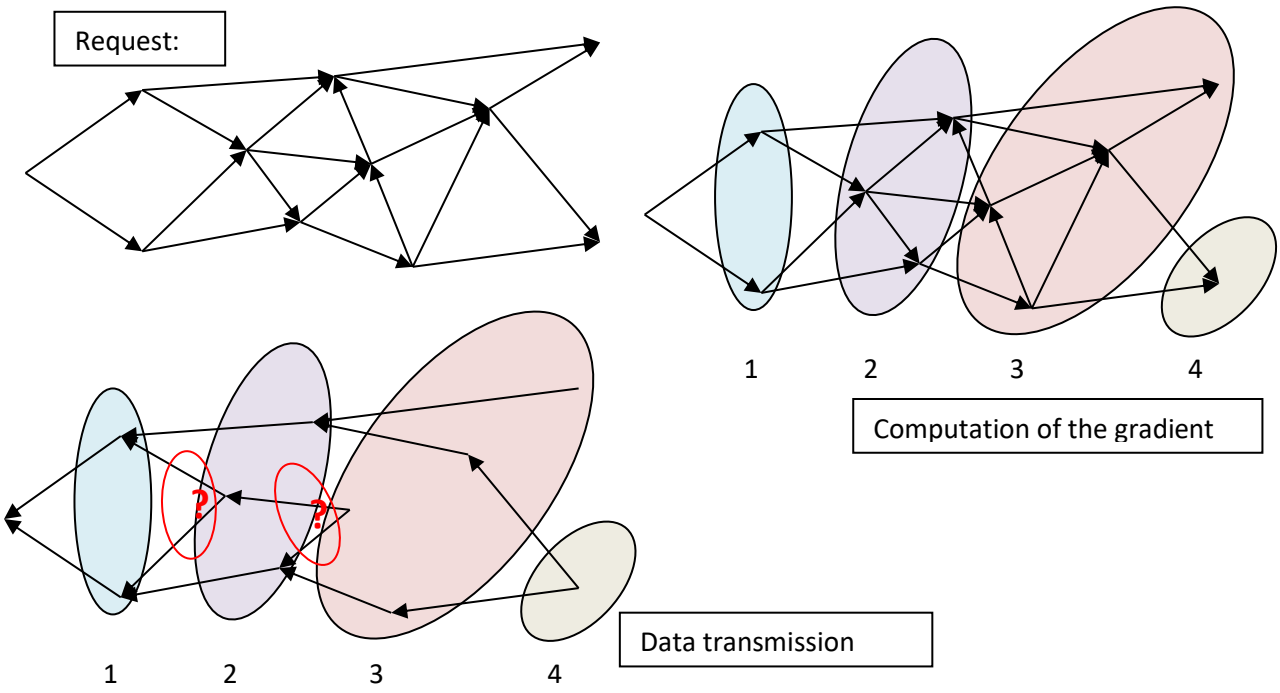
- Message distribution by *broadcasting*.
- The receiver after the first arrival of a message stores the message or only its identifier (if the destination of the message is another node), and broadcasts the message.

- *Typical application:* single source → multiple destination (command is distributed quasi simultaneously).
- *Its advantage:* simple, fault tolerant due to the high level of redundancy.
- *Its disadvantage:* very many superfluous message and energy consumption. In addition, collisions due to hidden terminals.
- *Modifications:*
  - the receiver broadcasts the message only with a probability *p*. The value *p* is topology dependent.
  - to reduce collisions: after receiving the message random waiting time before transmission.

Flat: *Gradient Based Routing (GBR):*

- *Three phases:* (1) Request, (2) Computing the gradient, (3) Data transmission.
- *Typical application:* multiple sources → single destination (data collection).

(1) Request: the central node sends a request into the network by flooding.
(2) Computing the gradient: during the distribution of the request: "measurement" of the gradient → The gradient is the "shortest distance" from the central node: What is the lowest number of hops to reach the central node.
(3) Data transmission: selecting the shortest route, and sending the data. Data aggregation on the route is possible



GBR variants: In case of multiple equivalent route which one to select?

- *stochastic:* random selection
- *energy-based extension*: nodes having low energy can increase their "gradient", thus flow is oriented to another route.

Hierarchical: mentioning only two, descriptions can be found on the internet:

Low Energy Adaptive Clustering Hierarchy (LEACH): Hierarchical, based on dynamically formed clusters.
Geographic and Energy Aware Routing (GEAR): The nodes are familiar with their geographic location, and thus the messages are travelling only towards the targeted region.

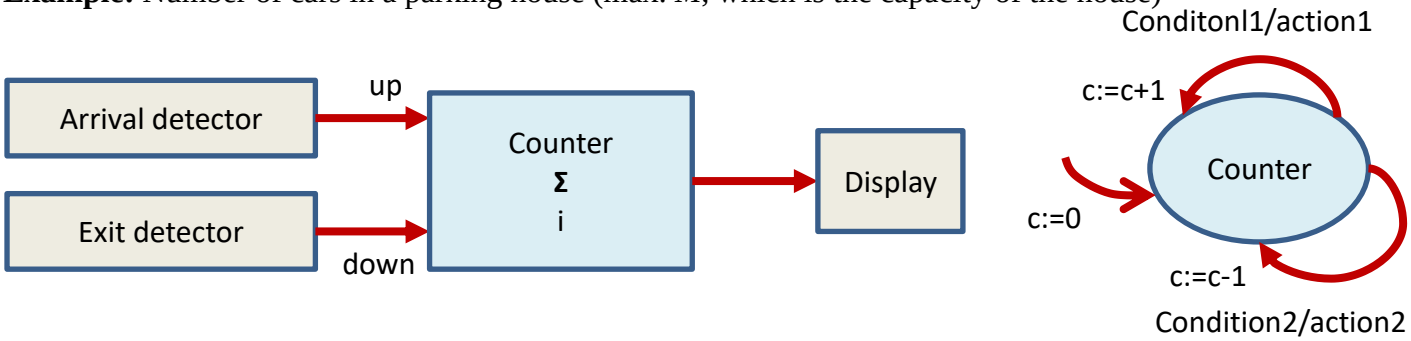
10. Hybrid systems, nonconventional modelling and control, efficient implementation

10.1. Hybrid systems

Hybrid systems combine continuous and discrete dynamics. Sometimes they are called modal systems, because controlled by a Finite State Machine (FSM), they are switched into different modes of operation where they behave as continuous systems. Concerning the mode changes, hybrid systems behave like discrete systems, but between these mode changes time dependency is present.

Discrete systems:

Example: Number of cars in a parking house (max. *M*, which is the capacity of the house)

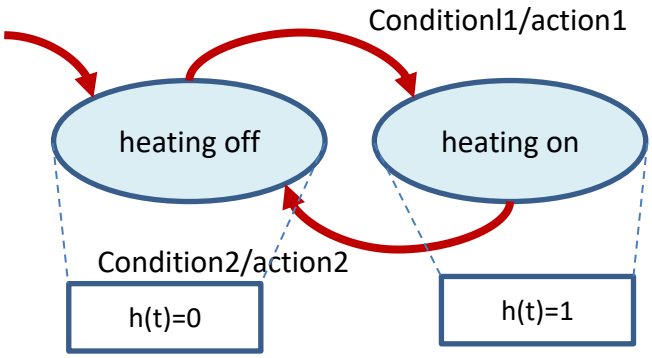


Condition1/action1: $up \wedge \neg down \wedge c < M \,/\, c+1$ Condition2/action2: $down \wedge \neg up \wedge c > 0 \,/\, c-1$

Example: *Thermostat with hysteresis:*

Condition1/action1: Temperature ≤ 18 C°/heating on

Condition2/action2: Temperature ≥ 22 C°/heating off



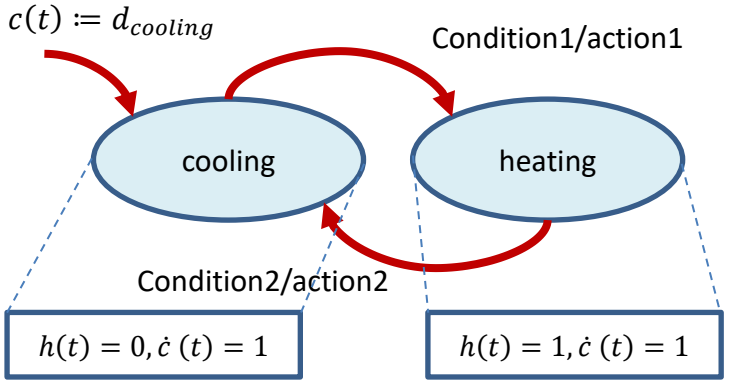
System input: Temperture of the environment

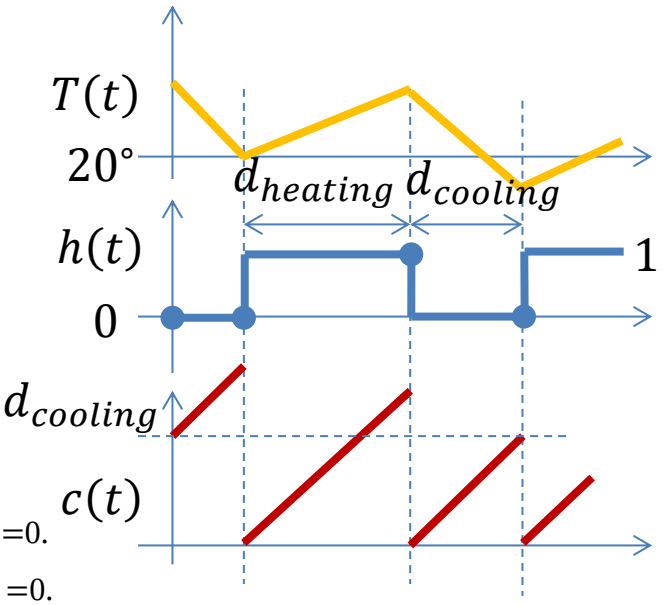
System output: heating on/heatinf off commands:

the corresponding time functions: *h(t)*=1, *h(t)*=0.

Timed automaton:

Example: *Thermostat with timing instead of hysteresis:* this is solved by the so-called **timed automaton**, which is the simplest nontrivial hybrid system. These automata, behind their states measure the evolvement of time for a given value of duration: $\forall t \in d_m$, and the derivative of the clock function is $\dot{c}(t) = a$, i.e. its value changes with the evolvement of time.





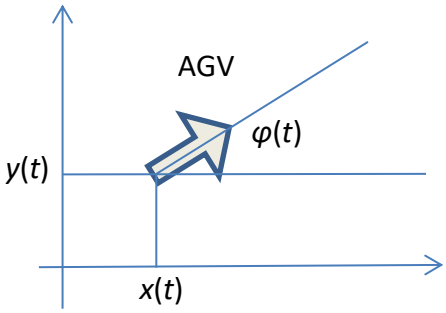
Condition1/action1: $T(t) \leq 20 \wedge c(t) \geq d_{cooling} / c(t) = 0$.

Condition2/action2: $T(t) \geq 20 \wedge c(t) \geq d_{heating} / c(t) = 0$.

Comments: (1) *h(t)* and *c(t)* can be considered as tools of state refinement. They define some details of the operation (Modal systems). (2) On the time diagram T>20 C°. If it would be lower, then immediately the heating mode would start. This is served by the initial condition of the clock.

Example: Automated Guided Vehicle, AGV

It is a vehicle with two degrees of freedom. It can follow a painted stripe. It is moving in every time instant with a velocity of *v*(*t*), where $0 \leq v(t) \leq 10km/h$. It can rotate around its centre with an angular speed $\omega(t)$, where $-\pi \leq \omega(t) \leq \pi\, rad/sec$.



$$\dot{x}(t) = v(t)\cos(\varphi(t))$$
$$\dot{y}(t) = v(t)\sin(\varphi(t))$$
$$\dot{\varphi}(t) = \omega(t)$$

Two-level control: The AGV runs with a constant speed of 10 *km/h*. It has four operational mode: **left, right, straight, stop.**

To every mode of operation, a separate differential equation is assigned.

straight:

$$\dot{x}(t) = 10\cos(\varphi(t))$$
$$\dot{y}(t) = 10\sin(\varphi(t))$$
$$\dot{\varphi}(t) = 0$$

left:

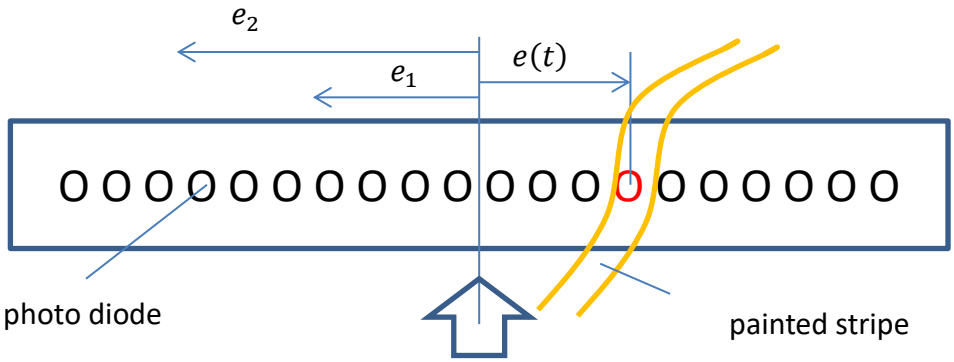
$$\dot{x}(t) = 10\cos(\varphi(t))$$
$$\dot{y}(t) = 10\sin(\varphi(t))$$
$$\dot{\varphi}(t) = \pi$$

right:

$$\dot{x}(t) = 10\cos(\varphi(t))$$
$$\dot{y}(t) = 10\sin(\varphi(t))$$
$$\dot{\varphi}(t) = -\pi$$

stop:

$$\dot{x}(t) = 0$$
$$\dot{y}(t) = 0$$
$$\dot{\varphi}(t) = 0$$



The sensor of the AGV: a set of photodiodes perpendicular to the direction of the movement. Its output signal:

$$e(t) = f(x(t), y(t)).$$

Ha $e(t) > 0$, then it deviates to the left, if $e(t) < 0$, then to the right.

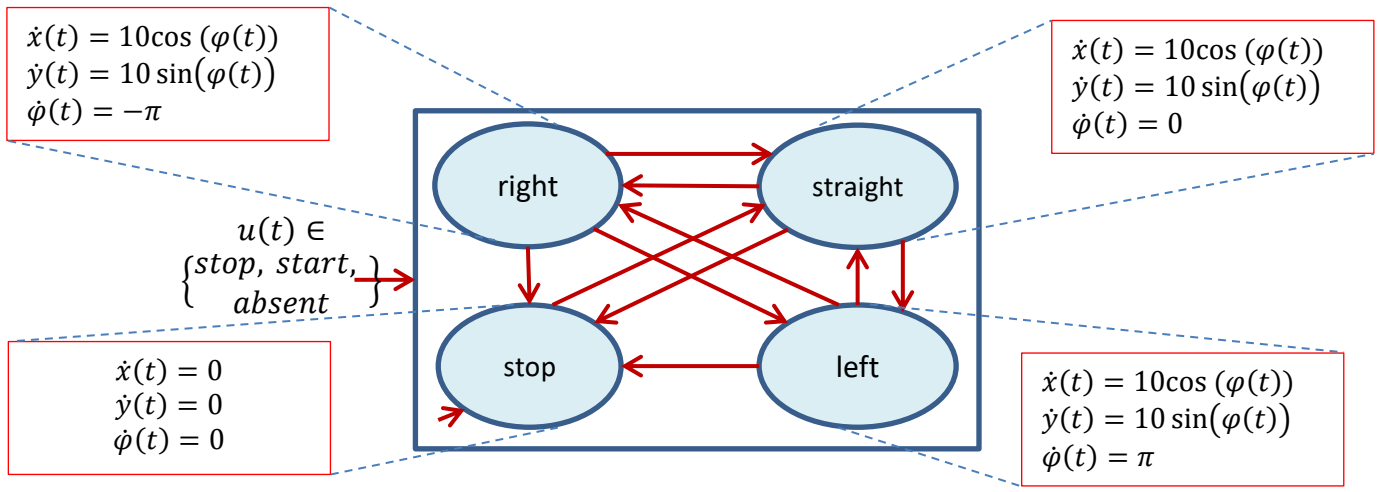
$$(e_1 > 0, e_2 > 0)$$

The control law of the AGV: if $|e(t)| < e_1$, then go straight; if $0 < e_2 < e(t)$, then go to right, if $0 > -e_2 > e(t)$, then go to left.

The set of the input events: $u(t) \in \{stop, start, absent\}$. Since *stop* and *start* are instantaneous events, a*bsent* gives the interpretation for other time instants.

State-transition generating conditions:

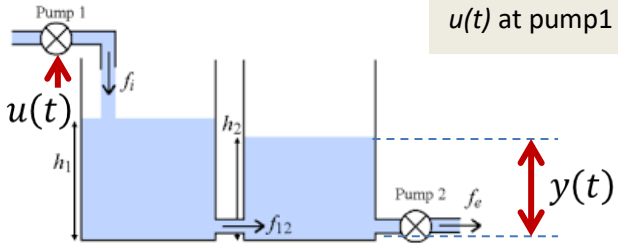
$$start = \{(v(t), x(t), y(t), \varphi(t)) | u(t) = start\}$$
$$go\ straight = \{(v(t), x(t), y(t), \varphi(t)) | u(t) \neq stop, |e(t)| < e_1\}$$
$$go\ right = \{(v(t), x(t), y(t), \varphi(t)) | u(t) \neq stop, e_2 < e(t)\}$$
$$go\ left = \{(v(t), x(t), y(t), \varphi(t)) | u(t) \neq stop, -e_2 > e(t)\}$$
$$stop = \{(v(t), x(t), y(t), \varphi(t)) | u(t) = stop\}$$



10.2. nonconventional modelling and control, efficient implementation

Example #1: *Qualitative modelling and control I.*

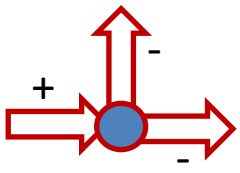
Task: The design of such a controller which keeps the level of the liquid in the second tank at a prescribed level.



This is possible by setting *u(t)* at pump1 properly.

Problems of the quantitative model:
a) The physical limits are not modelled;
b) The equations are linearized;
c) Numerical values are inaccurate and change with time.

Qualitative Reasoning: Only the orientation of the quantities is considered. Possible "values": $\{-,0,+\}$.
The basic physical constraints are kept!



If at branching of a node the liquid flows out in two directions, then through the third tube the liquid should flow in.
The qualitative value of a quantity "Q" with respect to "a": $[Q]_a$
The qualitative value of the change of a quantity "Q" is the qualitative derivative: $[\delta Q]_a$, $[\delta^2 Q]_a$, …

Operation: $(invert\ A)$: Changes the sign.
$(vote\ A_1, A_2, \ldots, A_n)$: Gives back the value in majority.

Qualitative control of the level of tank2: L_2 denotes the level relative to the desired value:

$[L_2] = +$: higher than required.	$[\delta U] = +$: increase pumping rate.
$[L_2] = 0$: equals.	$[\delta U] = 0$: pumping rate is appropriate.
$[L_2] = -$: lower than required.	$[\delta U] = -$: decrease pumping rate.

$[\delta U] = +$: a fixed amount of increase of the pumping rate: ΔU.

The qualitative values exist only at the sampling instants. Between sampling instants there is no level detection.

$$[L_2]_{(k)} = [actual\ level_{(k)} - required\ level_{(k)}]$$

A very simple control law:

$$Q1 \overset{def}{\triangleq} [\delta U]_{(k)} = (invert[L_2])_{(k)}$$

Comment: If ΔU is larger, then larger overshoot and oscillation can be expected, but the reaction is faster. If ΔU is smaller, then the overshoot and the oscillation will be smaller, but also the reaction is slower.

Improved controllers:

Quantities considered:

Level error of tank2: +,0,-
Speed of the level change of tank2: +,0,-
Speed of the level change of tank1: +,0,-

} 3*3*3=27 cases.

$$Q2 \overset{def}{\triangleq} [\delta U]_{(k)} = \left(invert\left(vote\left(vote\left([L_2]_{(k)}, [\delta L_2]_{(k)}\right), [\delta L_1]_{(k)}\right)\right)\right)_{(k)}$$

$$Q3 \overset{def}{\triangleq} [\delta U]_{(k)} = \left(invert\left(vote\left([L_2]_{(k)}, [\delta L_2]_{(k)}, [\delta L_1]_{(k)}\right)\right)\right)_{(k)}$$

Determination of $[\delta L_1]$: $\delta L_1 = \left(L_{2(k)} - L_{2(k-1)}\right) - \left(L_{2(k-1)} - L_{2(k-2)}\right)$ based on measurements.

For the 27 combinations of the possible qualitative values the outputs of the three controllers can be summarized in the table below:

	$[L_2]$	$[\delta L_2]$	$[\delta L_1]$	$Q1$	$Q2$	$Q3$
1	+	+	+	-	-	-
2	+	+	0	-	-	-
3	+	+	-	-	0	-
4	+	0	+	-	-	-
5	+	0	0	-	-	-
…						
20	-	+	0	+	0	0
…						
27	-	-	-	+	+	+

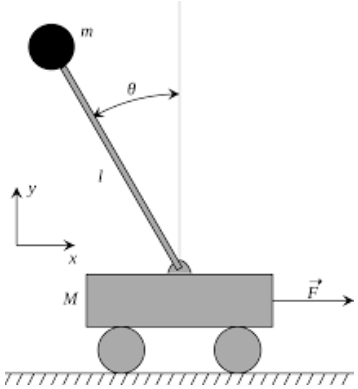
Comments:

(1) A rule-based system was also elaborated for this problem. It could not handle the case: Tank2 shows a constant value above the required level, and the level of Tank1 lowers.

(2) Setting sampling rate and the value of ΔU is a critical issue, and a crucial decision of the designer.

Example#2: Qualitative modelling and control II.

Task: Modelling an inverted pendulum with nondeterministic automaton.
This approach might be required for systems where the about the state vector *x(k)* only quantized [*x(k)*] values are available due to limited precision measurements of angles and angular velocity.

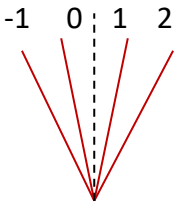


After linearisation of the equations around $\theta = 0$:

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{mg}{M} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{(m+M)g}{Ml} & 0 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \frac{1}{m} \\ 0 \\ -\frac{1}{Ml} \end{bmatrix} u(t)$$

$$M = 1kg,\ m = 0.1kg,\ l = 0.5m,\ g = 9.81\frac{m}{s^2}$$

Measurement insensitivity: 0.0175 rad for θ, and 0.0175/20ms for $\dot{\theta}$.
The pole can no longer be stabilised if $|x_3| > 0.21\ rad$ (12°), és $|x_4| > 0.87$.



For the angle (index 3) and for the angular speed (index 4) the bounds corresponding to the figure:

$$g_{3,-1} = -0.210, g_{3,0} = -0.0175, g_{3,1} = 0.0175, g_{3,2} = 0.210$$
$$g_{4,-1} = -0.870, g_{4,0} = -0.0175, g_{4,1} = 0{,}0175, g_{4,2} = 0.870$$

If we denote staying in one of the two central regions by 0, by - 1 staying in the left-hand-side region, and by +1 staying in the right-hnd-side one, we can define the following qualitative states:

$$z_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, z_2 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, z_3 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, z_4 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, z_5 = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$
$$z_6 = \begin{bmatrix} 0 \\ 1 \end{bmatrix},\ z_7 = \begin{bmatrix} 1 \\ -1 \end{bmatrix},\ z_8 = \begin{bmatrix} 1 \\ 0 \end{bmatrix},\ z_9 = \begin{bmatrix} 1 \\ 1 \end{bmatrix},\ z_{10} = outside,$$

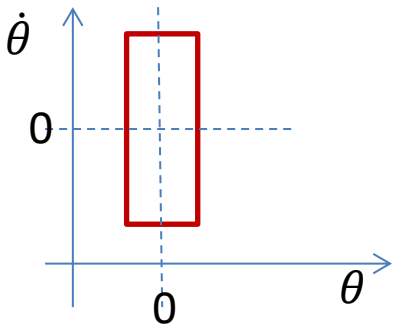
The qualitative values of the force on the vehicle (input signal):

$$u(k) = 10 \Leftrightarrow v(k) = 1, \qquad u(k) = 0 \Leftrightarrow v(k) = 0, \qquad u(k) = -10 \Leftrightarrow v(k) = -1$$

By assigning proper input to the qualitative states, the pole can be stabilized:

$z(k)$	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_8
$u(k)$	-1	0	0	-1	0	1	0	0	1

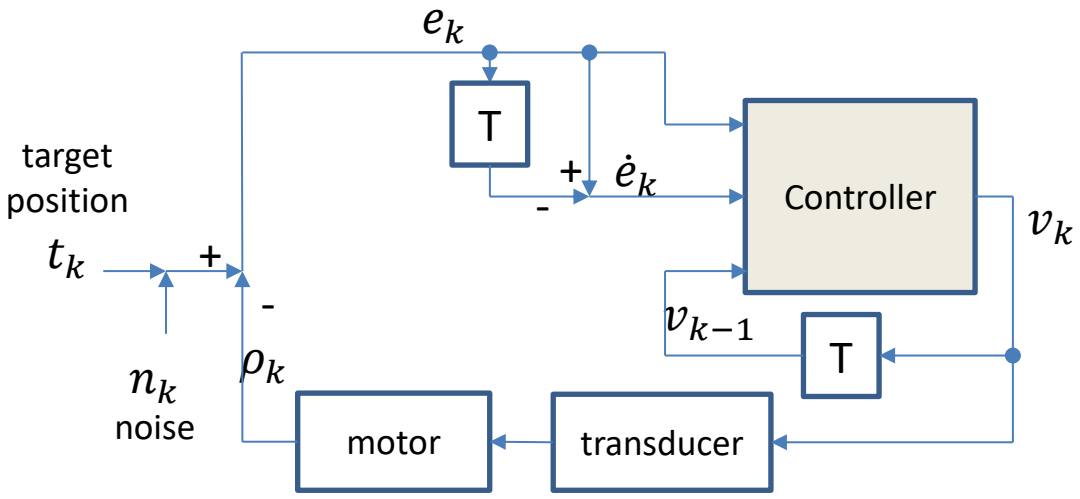
The qualitative controller: $[u(k)] = f([z(k)])$



Comments: Setting sampling rate and the value of ΔU is a critical issue, and a crucial decision of the designer.
The figure shows the idealized trajectories of the motion.
The real trajectories due to noise/disturbance do return to themselves.

Example #3: Adaptive target tracking with fuzzy modelling and control

The target tracking system consists of two channels: it maps azimuth-elevation inputs to motor control outputs. The nominal target moves through azimuth-elevation space. Two motors adjust the platform to continuously point towards the target [azimuth (0 … 180 degrees) elevation (0 … 90 degrees].
Sensor: The platform can be any directional device that accurately points towards the target. The device may be a laser, radar, video camera or high-gain antenna.

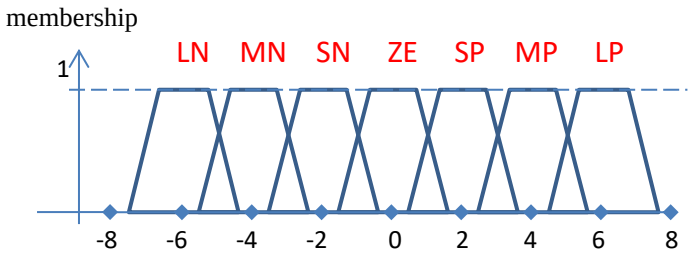


Notation: t_k target position; n_k observation noise; ρ_k tracking position
e_k tracking error; $\dot{e}_k$ tracking error change; v_k estimated angular velocity
T sampling time.

$$\rho_k = \rho_{k-1} + Tv_{k-1} + error$$

error = positioning uncertainty

Fuzzy controller: We restrict the output angular velocity of the fuzzy controller to the interval: [-6,6]. (This is a decision of the designer, a scaling factor).) Since $|v_k| \leq \frac{9.0}{T} degrees/sec$ azimuth, and $|v_k| \leq \frac{4.5}{T} degrees/sec$ elevation, thus the output gains of the channels are: 1.5/T and 0.75/T.
The fuzzy controller uses heuristic control set-level "rules" or fuzzy-associative-memory (FAM) associations, based on quantised values of $e_{k,}$ $\dot{e}_k$ and v_{k-1}. We define seven fuzzy levels by the following library of fuzzy-set values of the fuzzy variables $e_{k,}$ $\dot{e}_k$ and $v_{k-1:}$



LN=Large Negative
MN=Medium Negative
SN=Small Negative
ZE=Zero
SP=Small Positive
MP=Medium Positive
LP=Large Positive

We assign each system input to a fit vector of length 7:

1 → (0 0 0 0.7 0.7 0 0)
-4 → (0 1 0 0 0 0 0)
3.8→ (0 0 0 0 0.1 1 0)

We formulate control FAM rules by associating output fuzzy sets with input fuzzy sets: For example the i-th rule:

$$IF\ e_k = MP \wedge \dot{e}_k = SN \wedge v_{k-1} = ZE\ THEN\ v_k = SP$$

We abbreviate this to: (*MP,SN,ZE;SP*). The scalar value of the i-th FAM rule : $w_i = \min(membership\ values)$.

Example: $e_k = 2.6,\ \dot{e}_k = -2.0, v_{k-1} = 1.8$. The fit vectors of length 7:

LN	MN	SN	ZE	SP	MP	LP
0	0	0	0	1	0.4	0
0	0	1	0	0	0	0
0	0	0	0.1	1	0	0

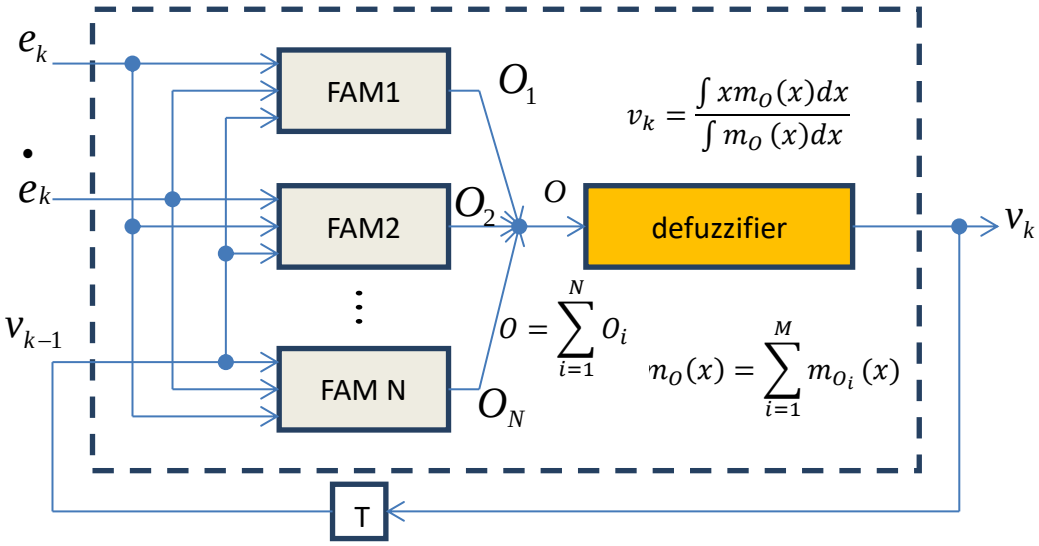
The membership values associated to the rule:

$$m_{MP}(e_k) = 0.4$$
$$m_{SN}(\dot{e}_k) = 1$$
$$m_{ZE}(v_{k-1}) = 0.1$$

The scalar value of the *i*-th rule:

$$w_i = \min(0.4, 1, 0.1) = 0.1$$

The controller:



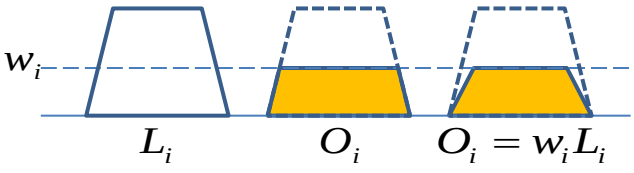
The form of the output fuzzy set depends on the encoding of the FM rule:

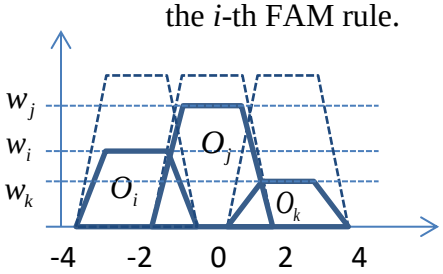
Correlation-product encoding:
$m_{O_i}(x) = w_i m_{L_i}(x)$

Correlation-minimum encoding:
$m_{O_i}(x) = \min(w_i, m_{L_i}(x))$.

Here $m_{L_i}(x)$ stands for the membership function of the output associated to the *i*-th FAM rule.

Two possible encoding strategies of the output fuzzy sets:





The ***defuzzifier*** assigns numerical value to the sum of the output fuzzy sets associated with the FAM rules. This summed set is the sum of weighted trapezoids. It is like the probability density function of probability theory, except the integral of the summed function differs from one. The ***defuzzifier*** computes the v_k value as a centroid, therefore it is called: ***fuzzy centroid***.

The implementation of the fuzzy controller: A FAM_i rule: (*MP,SN,ZE;SP*). At the *k*-th time instant: $e_k = 2.6,\ \dot{e}_k = -2.0, v_{k-1} = 1.8$.

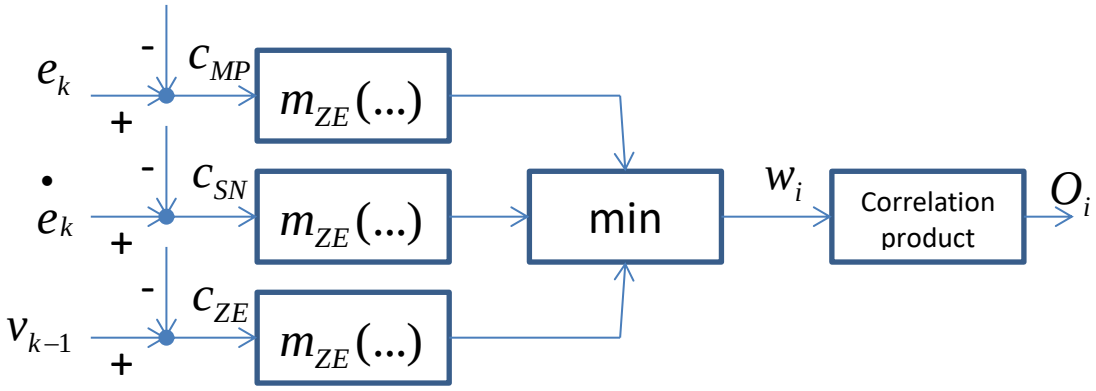
$$w_i = \min\left(m_{MP}(e_k), m_{SN}(\dot{e}_k), m_{ZE}(v_{k-1})\right) = \min(0.4,1,0.1) = 0.1$$

Since in this solution the shape of every fuzzy set is the same, we can write: e.g. $m_{SP}(x) = m_{ZE}(x-2)$. In general $m_{L_i}(x) = m_{ZE}(x - c_{L_i})$, where c_{L_i} is the centroid of the given membership function.

$$w_i = \min\left(m_{ZE}(e_k - c_{MP}), m_{ZE}(\dot{e}_k - c_{SN}), m_{ZE}(v_{k-1} - c_{ZE}) \right)$$

$$w_i = \min\left(m_{ZE}(-1.4), m_{ZE}(0), m_{ZE}(1.8)\right) = \min(0.4,1,0,1) = 0.1$$

In case of correlation-product encoding: $m_{O_i}(x) = w_i m_{ZE}(x - c_i)$, thus the implementation of the i-th FAM rule can have the following form:



11. Safety-critical systems

Concepts and requirements of safety-critical systems

- Risk analysis: Tolerable Hazard Rate (THR)
  - In case of continuous operation: the rate of hazardous error phenomena per hour;
  - In case of discontinuous operation: the probability of the hazardous error phenomena at the time of calling the function
- Categorization: Safety Integrity Level (SIL)

SIL	Error of safety-critical function/hour
1	10^{-6}<THR<10^{-5}
2	10^{-7}<THR<10^{-6}
3	10^{-8}<THR<10^{-7}
4	10^{-9}<THR<10^{-8}

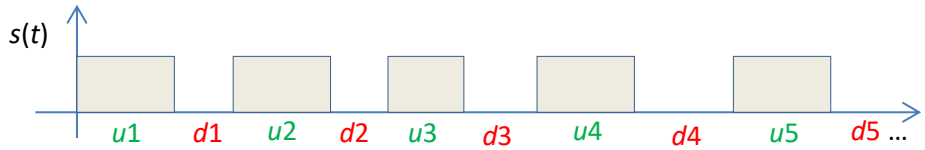
1 year = 8760 hours. Assuming SIL4: $10^8/8760 \cong 11415$ years without error. If the lifetime is 15 years, then during this time out of ~750 equipment the failure of one can be expected, since 15*750=11250.

Dependability requirements (Aspects of dependability):

- Reliability: Probability of continuous correct service (until the first failure). E.g., "After departure the onboard control system shall function correctly for 12 hours";
- Availability: Probability of correct service (considering repairs and maintenance) E.g., "Availability of the service shall be 95%";
- Safety: Freedom from unacceptable risk of harm;
- Integrity: Avoidance of erroneous changes or alterations;
- Maintainability: Possibility of repairs and improvements;

Dependability metrics:

Basis: Partitioning the states of the system s(t). Correct (U, up) and incorrect (D, down) state partitions.



Mean values:

- MTFF=E{u1} Mean Time to First Failure
- MUT=E{ui} Mean Up Time
- MTTF Mean Time To Failure (Same as previous)
- MDT=E{di} Mean Down Time
- MTTR Mean Time To Repair (Same as previous)
- MTBF=MUT+MDT Mean Time Between Failures

Probability functions:

- Availability: a(t)=P{s(t)∈ U} decreases with time, system may fail
- Asymptotic availability: A=$lim_{t\to\infty}$a(t) A=MTTF/(MTTF+MTTR)
- Reliability: r(t)=P{s(t') ∈ U} $\forall t' < t$, system does not fail, →0.

Component attribute:

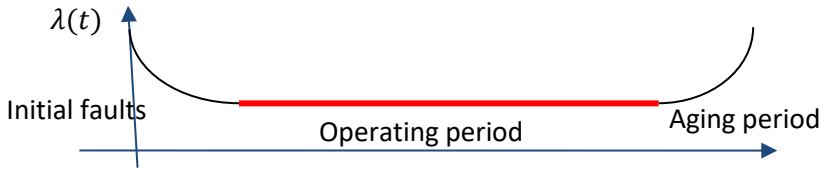
- Fault rate: λ(t) Probability that the component will fail in the interval Δt at time point t given that it has been correct until t is given by λ(t)Δt:

$$\lambda(t)\Delta t = P\{s(t+\Delta t) \in D | s(t) \in U\},\ \Delta t \to 0.$$

- Reliability of a component based on this definition:

$$\lambda(t) = -\frac{1}{r(t)}\frac{dr(t)}{dt}, \text{ thus } r(t) = e^{-\int_0^t \lambda(t)dt}.$$

For electronic components:



In the operating period $\lambda(t) = \lambda$.
Assuming exponential distribution:

$$r(t) = e^{-\lambda t}$$

$$MTFF = E\{u1\} = \int_0^\infty r(t)dt = \frac{1}{\lambda}$$

Comment: Initial errors are filtered by testing at the end of the production lines.

Threats to dependability:

- Fault: Adjudged or hypothesized cause of an error.
- Error: State leading to the failure.
- Failure: The delivered service deviates from correct service.

Example of fault → error → failure chain:

Fault	Error	Failure
Bit flip in the memory due to a cosmic particle	Reading the faulty memory cell will result in incorrect value	The robot arm collides with the wall
The programmer increases a variable instead of decreasing	The faulty statement is executed, and the value of the variable will be incorrect	The result of the computation will be incorrect

Means to improve dependability:

- Fault prevention:
  - o Physical faults: Good components, shielding, ...
  - o Design faults: Good design methodology
- Fault removal:
  - o Design phase: Verification and corrections
  - o Prototype phase: Testing, diagnostics, repair
- Fault tolerance: Avoiding service failures
  - o Operational phase: Fault handling, reconfiguration
- Fault forecasting: Estimating faults and their effects
  - o Measurements and prediction

The development process: e.g. V-model, verification, validation, testing, …

Organisation and independence of roles:

The roles:
Designer (analyst, architect, coder, unit tester) (DES); Verifier (VER); validator (VAL), Assessor (ASS), project manager (MAN), Quality assurance personnel (QUA).

In case of SIL0: DES, VER, VAL can be the same person, ASS should be different person;
In case of SIL1 and SIL2: DES, VER-VAL, and ASS should be different persons;
In case of SIL3 and SIL4: MAN, DES, VER-VAL, and ASS should be different persons, or even VER and VAL.

Architecture design to avoid hazard

Fail-safe operation: (1) fail-stop behaviour (Stopping (switch-off) is a safe state), (2) fail-operational behaviour (Stopping (switch-off) is not a safe state, service is needed). Fault tolerance is required.

Typical architectures for fail-stop operation

- Single channel architecture with built-in self-test;
- Two channels: (a) the same program with comparison, (b) not the same program with independent checker;

Fault tolerance: Providing (safe) service in case of faults (Fail-operational behaviour)
Extra resources: Redundancy: (1) Hardware, (2) Software, (3) Information, (4) Time.

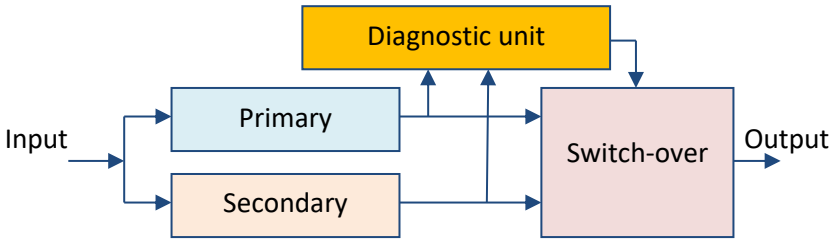
Types of redundancy: cold (inactive in fault-free case), warm (operates with reduced load), hot (active in fault-free case

How to use redundancy?

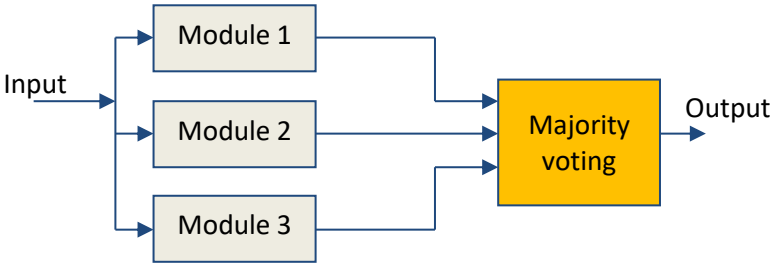
- Hardware design faults (<1%): hardware redundancy with design diversity.
- Hardware permanent operational faults (~20%): hardware redundancy, e.g. redundant processor.
- Hardware transient operational faults: (~70-80%): time-redundancy (e.g. instruction retry); information redundancy (e.g. error correcting codes); software redundancy (e.g. recovery from saved state).
- Software design faults (~10): software redundancy with design diversity

Fault tolerance for hardware permanent faults:

Duplication with diagnostics:



TMR: Triple-modular redundancy: Masking the failure by majority voting. Voter is critical (but simple).



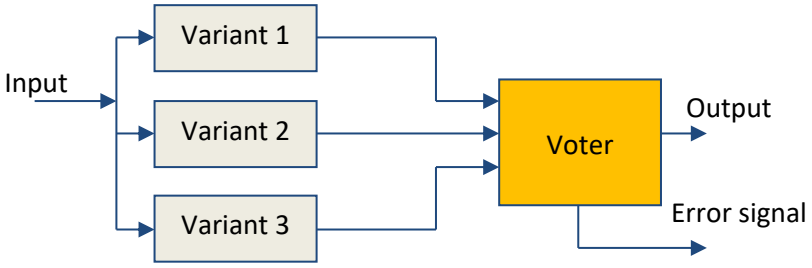
NMR: N-modular redundancy: Masking the failure by majority voting. In mission critical systems surviving the mission time is of primary importance. Following the mission repair is possible. Avionic on-board devices: 4MR, 5MR, sometimes 7MR.

Fault tolerance for software faults:

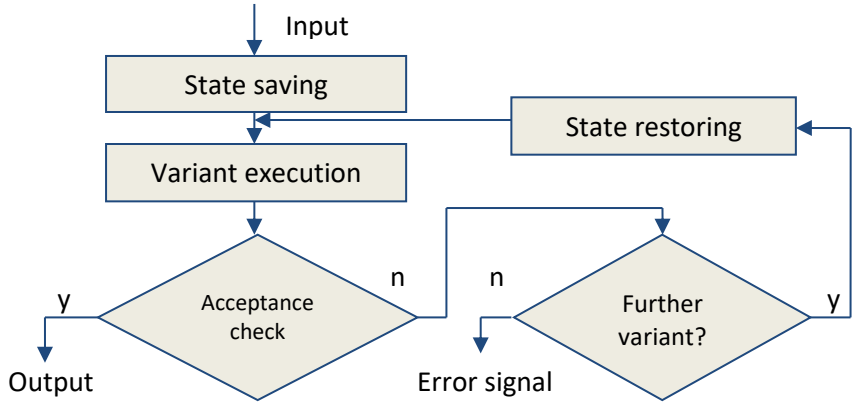
Repeated execution is not effective for design faults. Redundancy with design diversity is required.

Application of variants: redundant software modules with diverse algorithms and data structures, different programming language and development tools, separated development teams.

N-version programming: active redundancy, parallel execution, majority voting. If output acceptance range is specified, the voter will check it. The voter is a critical component (but simple).



Recovery blocks: passive redundancy, activated only in case of faults. (1) The primary variant is executed first; (2) Acceptance check is performed at the output of the variants; (If no acceptance test can be performed, then the method cannot be used.) (3) In case of detected error another variant is executed.

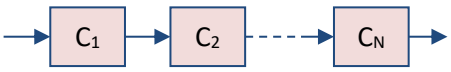


Comparison:

Property/Type	N-version programming	Recovery blocks
Error detection	Majority voting, relative	Acceptance checking
Execution	Parallel	Serial
Execution time	Slowest variant or time-out	Depends on # of faults
Activation of redundancy	Always (active)	Only in case of fault (p)
# of tolerated faults	[(N-1)/2]	N-1
Error handling	Masking	Restoration

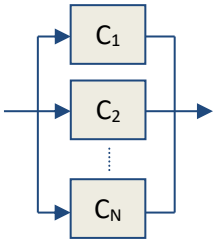
Reliability Block Diagram

1. Serial system: the components follow each other:
The system is faultless, if all the components are faultless.

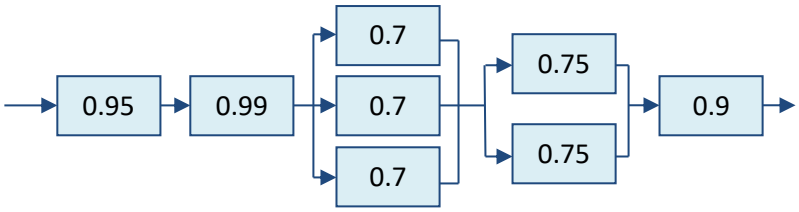


The reliability of the system is the product of the reliability of the components:
$r_S(t) = \prod_{i=1}^{N} r_i\,(t)$. If the fault rate of the components is λ_i, then for the system $MTFF = \frac{1}{\sum_{i=1}^{N} \lambda_i}$.

2. Parallel system: the components are parallelly connected:
The system is faulty, if all the components are faulty.
The probability of faulty behaviour is: (1 – reliability).
$1 - r_S(t) = \prod_{i=1}^{N}(1 - r_i\,(t))$. If the reliability of the components is the same: $r_C(t)$, then $r_S(t) = 1 - (1 - r_C(t))^N$. If the fault rate of the components is λ, then for the system $MTFF = \frac{1}{\lambda}\sum_{i=1}^{N}\frac{1}{i}$.



3. Composite system: can be calculated component by component:



The availability of the system can be computed from the availability of the components:

$$A_S = 0.95 \cdot 0.99 \cdot [1 - (1 - 0.7)^3] \cdot [1 - (1 - 0.75)^2] \cdot 0.9 = 0.95 \cdot 0.99 \cdot 0.973 \cdot 0.9375 \cdot 0.9 = 0.77$$