Real-Time Communication & Execution

1. Real-Time Communication
2. Input/Output
3. Real-Time Operating Systems
4. Real-Time Scheduling

Real-Time Communication

1. Requirements
2. Flow Control
3. PAR Protocol
4. Architectures
5. Contradictions in Design

Requirements 1

The **protocol latency** is the time interval between the start of transmission of a message at a sending node, and the delivery of this message across the CNI of the receiving node. A real-time 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**. A message should be delivered at all receiver CNI within a short and known time interval.

Composability is the most important property of a real-time system architecture. The communication system should exercise flow control over the requests from the clients to fulfill the temporal obligations (e.g., not to overload clients)

Requirements 2

The communication system should establish a **temporal firewall** around the operation of a host, forbidding the exchange of control signals across the CNI. Thus, the communication system becomes autonomous and can be validated independently from the application software in the host.

The timing properties of the application software in a temporally encapsulated host can also be validated in isolation.

Many real-time communication systems must support different system configurations that change over time. A real-time protocol should be flexible to accommodate these changes without requiring a software modification and retesting of the operational nodes that are not affected by the change.

Flexibility is also required to service important **sporadic** messages (e.g., an emergency shutdown) with minimal delay.

Requirements 3

The communication system must provide predictable and dependable service. Errors that occur during the message transmission must be **detected**, and should be **corrected** without increasing the jitter of the protocol latency. In a real-time system, the detection of message loss by the receiver of a message is of particular concern.

The physical structure of a real-time communication system is determined by technical and economic considerations. The **multicast** communication requirement suggests a communication structure that supports multicasting at the physical level, e.g., a bus or a ring network.

The high costs of a fully connected **point-to-point** communication network are prohibitive in most applications.

End-to-End Protocol

The effect of a desired action should always be monitored by an independent sensor. The results of this observation ensure that the desired action of the message has actually been achieved.

Such a protocol is called **end-to-end protocol**.

Flow Control

Flow control is concerned with the control of the speed of the information flow between a sender and a receiver in such a manner that the receiver can keep up with the sender. In any communication scenario, it is the receiver rather than the sender, which determines the maximum speed of communication.

In **explicit flow control**, the receiver sends an explicit acknowledgment message to the sender, informing the sender that the sender's previous message has arrived correctly, and that the receiver is now ready to accept the next message.

Explicit flow control requires that the sender is in the sphere of control of the receiver, i.e., the receiver can exert **back pressure** on the sender.

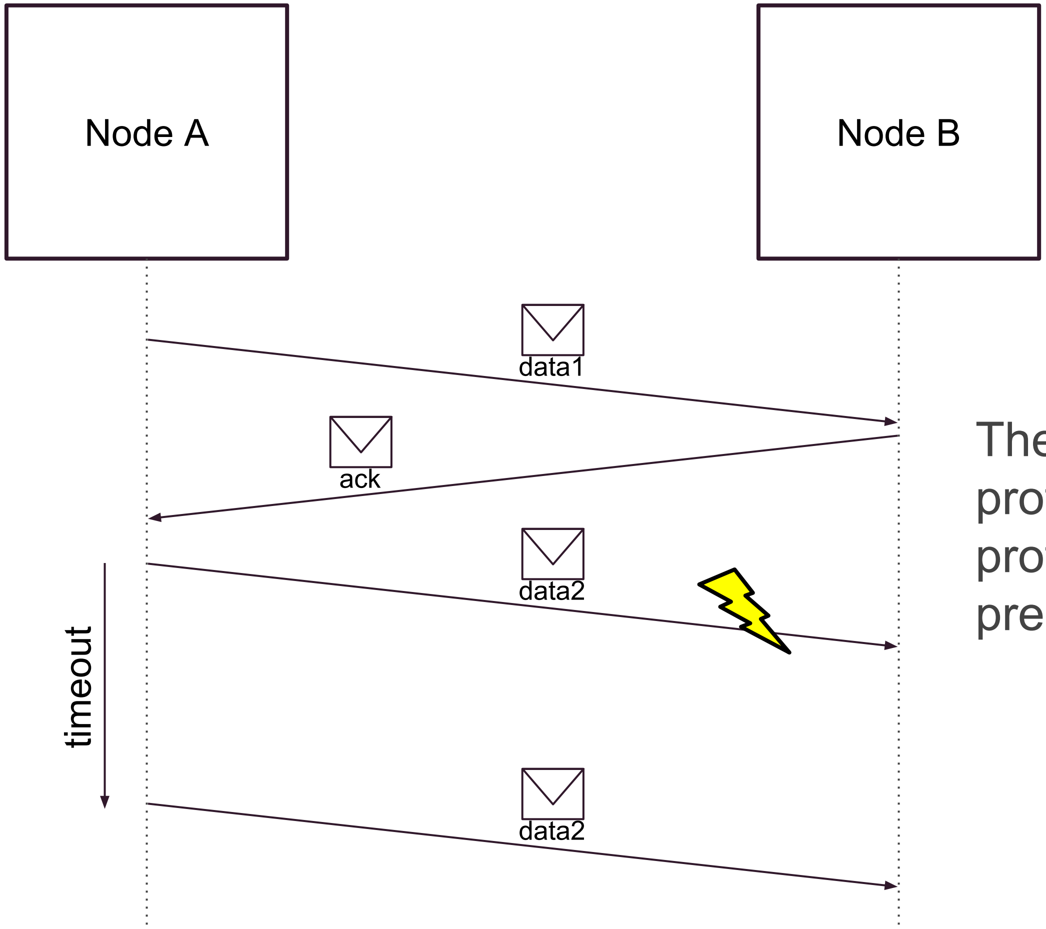
The PAR Protocol

The most important protocol with explicit flow control is the well known **Positive-Acknowledgment-or-Retransmission** (PAR) Protocol. It is event-triggered.

For a successful transmission, the receiver must inform the sender within the timeout interval of the successful transmission.

If the sender fails to transmit the message multiple times (retries), the sender aborts the communication.

PAR



The major concern with a PAR protocol is the **increase** of protocol execution time in the presence of transmission errors.

Implicit Flow Control

In implicit flow control, the sender and the receiver **agree a priori**, i.e., at system startup, about the points in time when messages will be sent.

No acknowledgment messages are exchanged during run time.

Error detection is the responsibility of the receiver, which knows when an expected message fails to arrive.

Trashing

The often observed phenomenon of the throughput of a system decreasing abruptly with increasing load, is called **trashing**. Mechanisms that can cause trashing are, e.g., retry mechanisms or dynamic scheduling.

Trashing

Real-time systems **must be free of trashing** phenomena. If a real- time system contains a mechanism that can cause trashing, then, it is likely that the system fails in the important rare-event scenarios.

A technique to avoid trashing in explicit flow-control schemes is

1. to monitor the resource requirements of the system continuously and
2. to exercise a stringent back-pressure flow control as soon as a decrease in the throughput is observed.

Flow Control Comparison

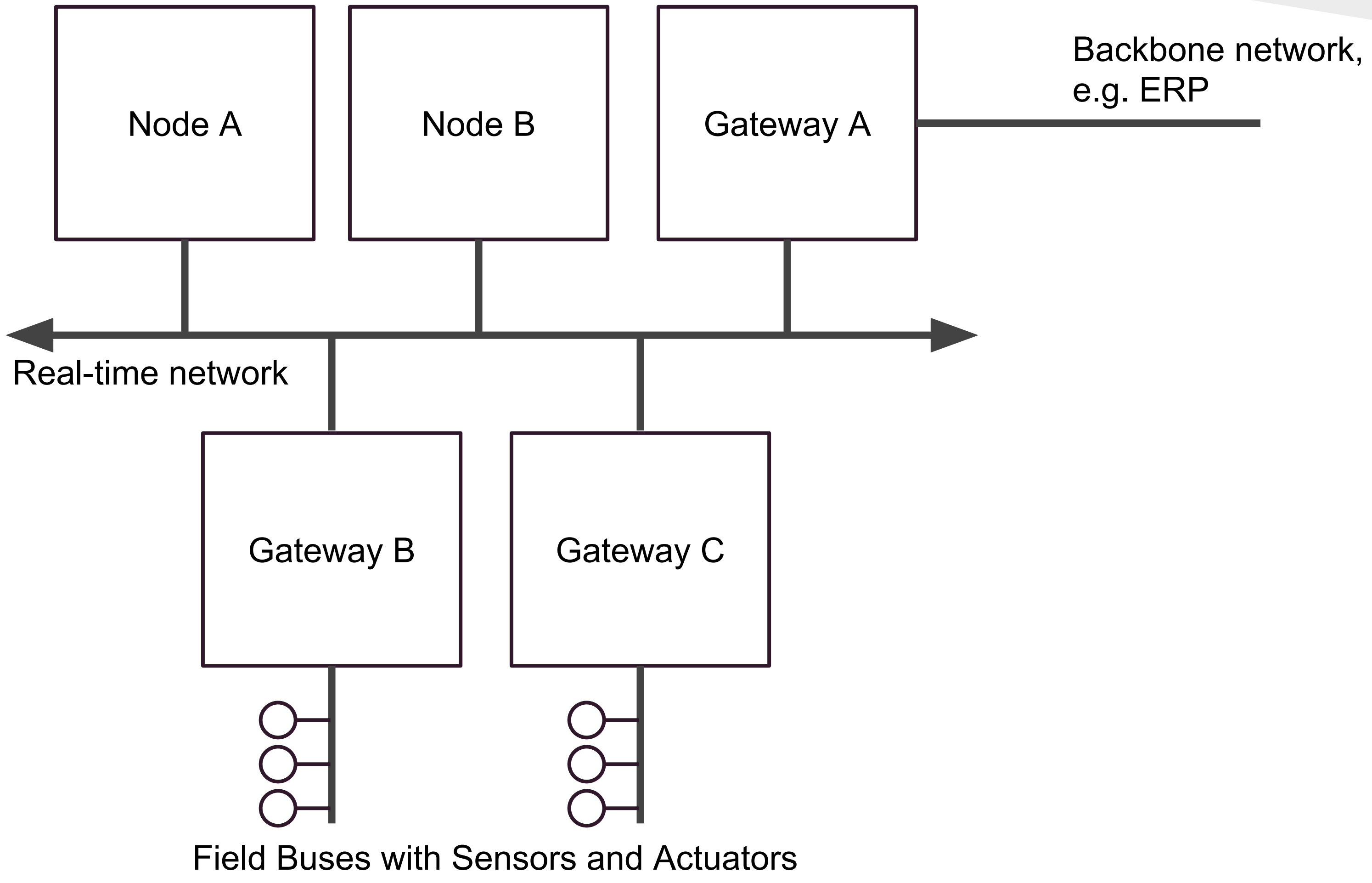
Mechanism	Explicit Flow Control	Implicit Flow Control	Hard Real-Time System
Control Signal	receiver must control the sender	time-triggered, predefined	time-triggered, predefined
Trashing	yes	no	must be avoided
Multicast	difficult	yes	yes

Flow Control Gateway

It is difficult to design the interface between a producer subsystem that uses implicit flow control and a consumer subsystem that uses explicit flow control.

Since the consumer subsystem can consume information only at the speed determined by its receiver, **adequate buffering** must be provided (which is a delicate design issue of a gateway).

Real-Time Communication Architecture


Node A
Node B
Gateway A
Backbone network, e.g. ERP
Real-time network
Gateway B
Gateway C
Field Buses with Sensors and Actuators

Real-Time Network

The real-time network must provide reliable and temporally predictable message transmission with small latency and minimal latency jitter, clock synchronization and support for fault-tolerance.

For fault-tolerance, the system must be designed so that e.g. a failure in any single unit cannot cause a crash of the total system (single point of failure).

Examples: FlexRay, TTP/C, TT-CAN, ARINC protocols, ethernet derivatives

Field Busses

The purpose of the fieldbus is to interconnect a node of the distributed real-time computer system to the sensors and actuators in the controlled process.

The main concern at the field bus level is low cost, both for the controllers and for the cabling.

Examples: ProfiBus, DeviceNet, LIN, CAN, 1-Wire, I2C, SPI, ...

Backbone Network

The purpose of the backbone network is the exchange of non time-critical information between the real-time cluster and the data-processing system of an organization.

Information can be production schedules, production reports, quality data, ...

Examples: Networks based on OSI architecture, e.g., TCP/IP

Contradictions in Design

A fundamental conflict exists between the requirement for **flexibility** and the requirement for **error detection**.

Flexibility implies that the behavior of a node is not restricted a priori. Error detection is only possible if the actual behavior of a node can be compared with some a priori known expected behavior.

In a system with unbound maximum communication delay, it can never be determined if a message was lost or is just delayed.

Tradeoffs

Event-triggered

1. Flexibility
2. Immediate response
3. Sporadic data

Time-triggered

1. Composability
2. Error detection
3. Periodic data

Comparison

Attribute	Fieldbus	Real-Time Network	Backbone Network
Message semantics	state	state	event
Latency/jitter control	yes	yes	no
Typical data length	1-10 bytes	1-64 bytes	>= 1,5kbytes
Clock synchronisation	yes	yes	optional
Fault-tolerance	limited	yes	limited
Communication control	single master	distributed	distributed
Flow control	explicit/implicit	implicit	explicit
Cost	very important	important	not very important

Input / Output

1. The Dual Role of Time
2. Agreement Protocols
3. Sampling / Polling / Interrupts
4. Smart Sensors
5. Installation

I/O Signals

Every I/O signal has two dimensions, the **value dimension** and the **temporal dimension**.

The value dimension of an I/O signal relates to the value of the signal. The temporal dimension relates to the moment when the value was recorded from the environment, or released to the environment.

Dual Roles of Time

An event that happens can be looked upon from two different perspectives.

1. It defines the point in time of a value change of an RT entity. The consequences of the event are analyzed later (time as data).
2. It may demand immediate action by the computer system to react at a defined point in time to this event (time as control).

Time as data: Consider a computer system that must measure the time interval between “start” and “finish” of a downhill skiing competition. Here, it is sufficient to record the precise timestamps of both events for later processing.

Time as control: Time as control is e.g. realized in a train scheduling plan. Control signals are derived from time.

Agreement Protocols

Sensors and actuators have failure rates that are considerably higher than those of single-chip microcomputers. It may be necessary to observe the controlled object by a number of different sensors and to relate these observations to detect erroneous sensor values, to observe the effects of actuators, and to get an **agreed** image of the state of the controlled object.

A data element that is measured at the I/O interface of a sensor, is called a raw data element. A **raw data** element that is calibrated and converted to standard technical units is called a **measured data** element. A measured data element that is consistent with other measured data elements - from different sensors - is called an **agreed data** element.

Two types of agreement protocols exist: syntactic and semantic

Syntactic Agreement

Assume that a single RT entity is measured by two independent sensors. When transformed from the analog values domain to the domain of discrete values, a slight difference between both raw values caused by measurement and digitization errors is unavoidable.

In **syntactic agreement**, the algorithm computes the agreed value without considering the context of the measured values (e.g., takes the average).

If one of the sensor readings can be erroneous, then the assumed failure model of the failed sensor determines how many measured data values are needed to detect the erroneous sensor. Syntactic agreement without any restrictions of the failure model of the sensor is the most costly form of agreement among a set of sensor values. Therefore, failure modes should be restricted (e.g., fail-silent).

Semantic Agreement

In **semantic agreement**, different RT entities are observed by different sensors and these sensor readings are related to each other to find a set of plausible agreed values, and to locate implausible values that indicate a sensor failure.

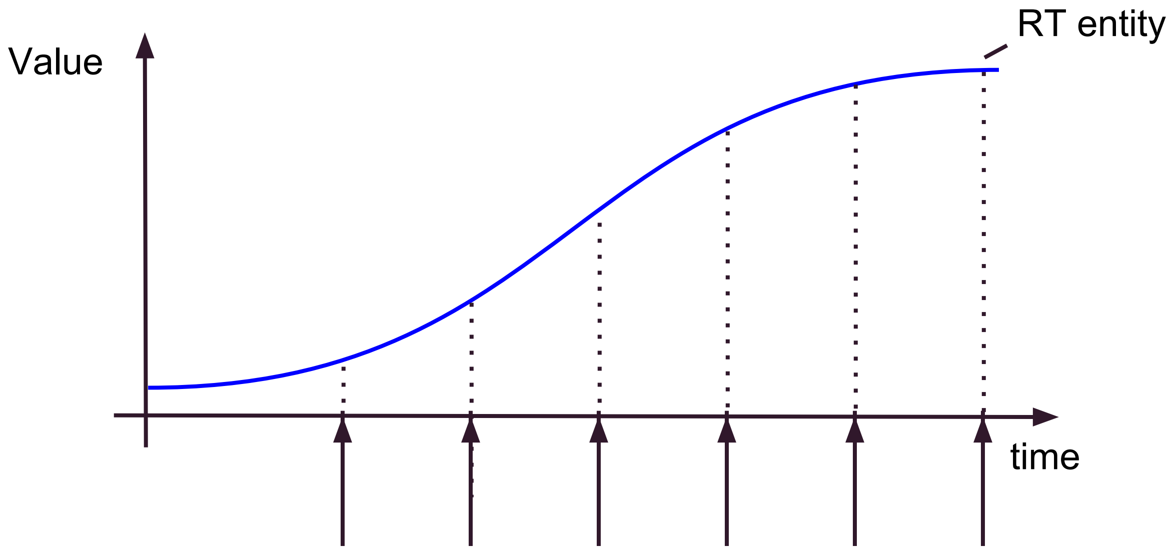
Such erroneous sensor values must be replaced by calculated estimates, based on the semantic redundancy in the set of measurements. It is not necessary to duplicate or triplicate every sensor.

Example: measuring the filling level of a vessel by a weight sensor and a distance gauge between ceiling and fluid level.

Sampling

In sampling, the state of an RT entity is periodically interrogated by the computer system at points in time called the sampling points.

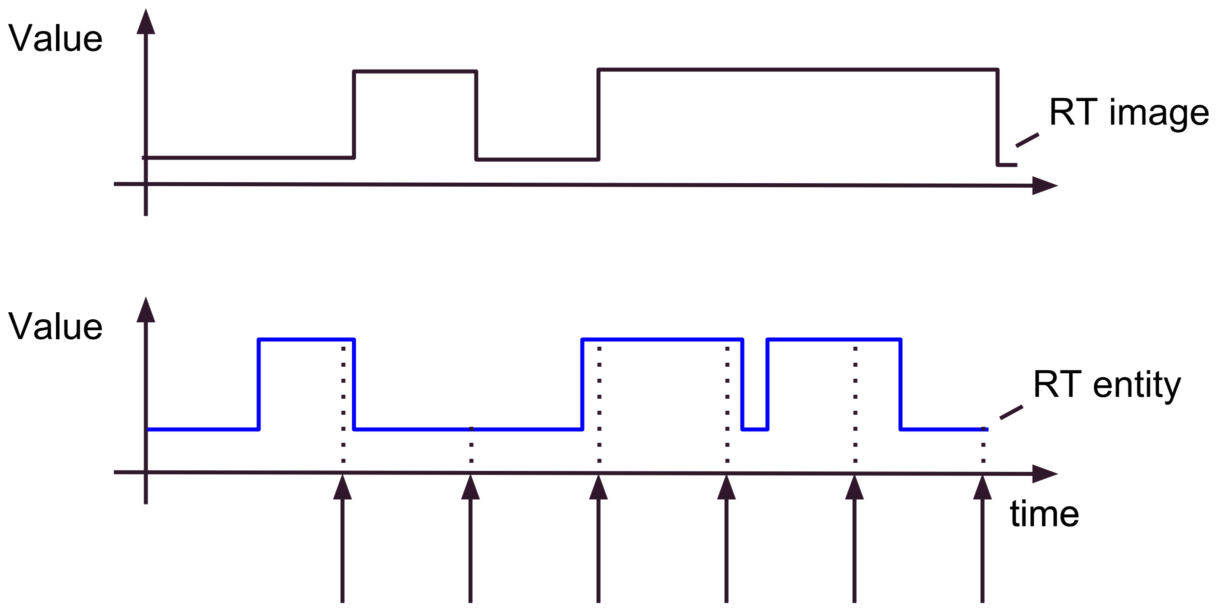
The temporal control always remains within the computer system. The constant time interval between two consecutive sampling points is called the sampling interval.



sampling of analog value

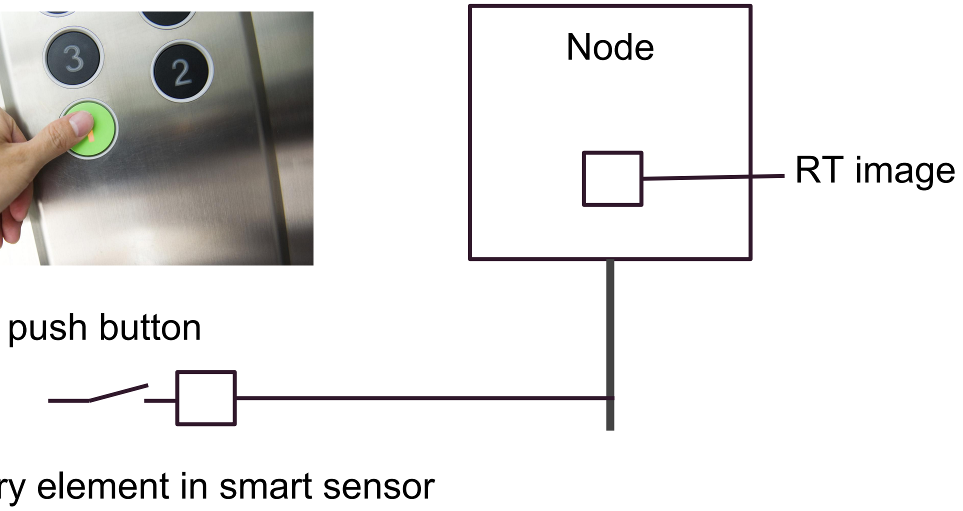
Sampling of Digital Values

When sampling of digital values, the current state and the temporal position of the most recent state change are often of interest. If the state of the RT entity changes more than once within a single sampling interval, some state changes will evade the observation. A sampling system acts as low pass filter and protects the node from more events than specified.



Sampling with Memory Element

If every event in the RT entity is significant (e.g., a elevator controller), a memory element must be implemented that stores any state change. It can be reset after is has been read.



Sampling with Memory Element

From the fault-tolerance point of view, sampling with a memory element is more robust than sampling without memory element in the following two situations.

1. Transient disturbances on the transmission line between the sensor and the computer.
    a. Sampling with smart sensor: Low probability
    b. Sampling without memory element: Error in data
2. Node shutdown and restart
    a. Sampling with smart sensor: Read sensors again;
    b. Sampling without memory element: Values lost

Smart Sensors

There is a trend to encapsulate a sensor / actuator and the associated microcontroller into a single physical housing to provide a standard abstract message interface to the outside world that produces measured values at the field bus. Such a unit is called an **smart sensor** and hides the concrete sensor interface.

Smart sensors provide better error detection and have a higher robustness against transient disturbances on the signal transmission line.

Interrupts

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 and potentially dangerous mechanism that must be used with care.

From a fault-tolerant point of view, an interrupt mechanism is even less robust than a polling mechanism.

Interrupts are needed when an external event requires such a short reaction time from the computer that it is not possible to implement this reaction time efficiently with sampling.

Monitoring Interrupts

A transient error on the interrupt line may upset the temporal control pattern of the complete node and may cause the violation of important deadlines.

Therefore, the time interval between the occurrences of any two interrupts must be continuously monitored and compared with the specified minimum duration between interrupting events by a sufficiently independent mechanism.

Example: external circuit to filter interrupts before entering the CPU

Fault-Tolerant Actuators

In a fault-tolerant system, the actuators must also be fault-tolerant to avoid a single point of failure.

A **fail-silent actuator** will either produce the intended (correct) output action or no result at all. In case a fail-silent actuator fails to produce an output action, it may not hinder the activity of the replicated fail-silent actuator.

A **triple-modular redundant (TMR) actuator** consists, e.g., of three motors. The force of any two motors must be strong enough to override the force of the third motor. However, any single motor may not be strong enough to override the other two. The TMR actuator can be seen as a “mechanical” voter, determining the majority of the three channels and outvoting the disagreeing channel.

Physical Installation

Many real-time systems fail because of a deficient physical installation. Some critical issues are:

1. Power supply failures. A reliable and clean power source is of importance
2. Grounding (high quality true ground point, no loops)
3. Electric isolation of the sensor signals (opto coupler)
4. Corrosion in plugs (signal interruption, signal degradation)
5. Insufficient shielding (EMI)
6. Insufficient heat transport
7. Insufficient fault-containment
   (e.g. fire in cable duct)

Real-Time Operating Systems

1. Task Management
2. Explicit / Implicit Synchronization
3. Non-Blocking Write (NBW) Protocol
4. Error Detection

Requirements

A real-time operating system must provide **predictable** service to the application tasks executing within the host.

The worst-case administrative overhead (**WCAO**) of every operating system service must be known a priori, so that the temporal properties of the behavior of the complete host can be determined analytically.

To make such an analytic analysis of the WCAO feasible, a hard real-time operating system must be very careful in supporting dynamic services that are common in standard workstation operating systems (e.g., Windows, Linux, MacOS, Unix, ..)

- dynamic task creation during runtime
- virtual memory management
- and dynamic queue management.

Task Management

Task management is concerned with the provision of the dynamic environment within a host for the initialization, execution, and termination of application tasks.

We will analyze task management for TT systems, ET systems with S-tasks, and ET systems with C-tasks.

In a time-triggered system, the temporal control structure of all tasks is established a priori by off-line support tools. This control structure is encoded in a **Task-Descriptor List** (TADL).

This schedule must consider the required sequence and mutual exclusion relationships between the tasks such that an explicit coordination of the tasks at run-time is not necessary.

Task Management

In an event-triggered system, the sequence of task executions is determined dynamically by the evolving application scenario. Whenever a significant event happens, a task is released to the active (ready) state, and the dynamic scheduler is involved.

It is up to the scheduler to decide at run-time which ready task is next to be serviced. The WCET of the scheduler contributes to the WCAO of the operating system.

Non-Preemptive S-Tasks

A non-preemptive S-task is the most common solution in hand-made operating systems. It can be established using a simple loop plus a switch statement.

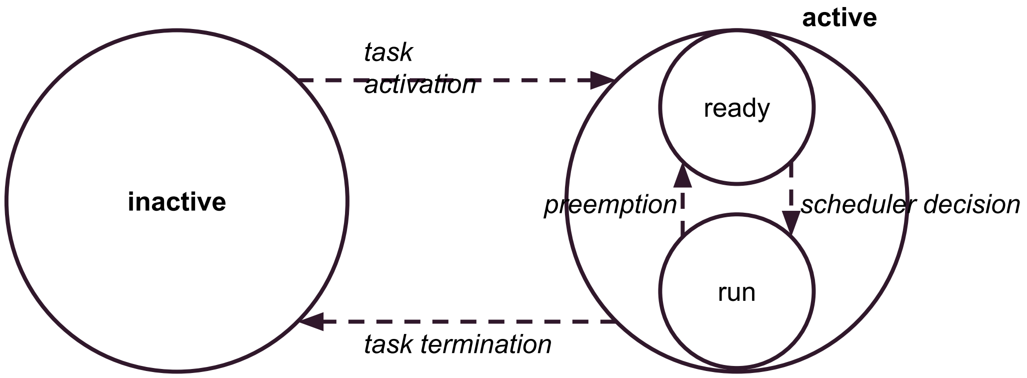
An ET operating system that supports non-preemptive S-tasks will take a new scheduling decision after the currently running task has terminated. This simplifies the task management but severely restricts its responsiveness.

If a significant event arrives after the longest task has been scheduled, the event will be considered only after task completion of the running task.

Preemptive S-Tasks

In a RT system that supports task preemption, each occurrence of a significant event can potentially activate a new task and cause an immediate interruption of the currently executing task to invoke a new decision by the scheduler.

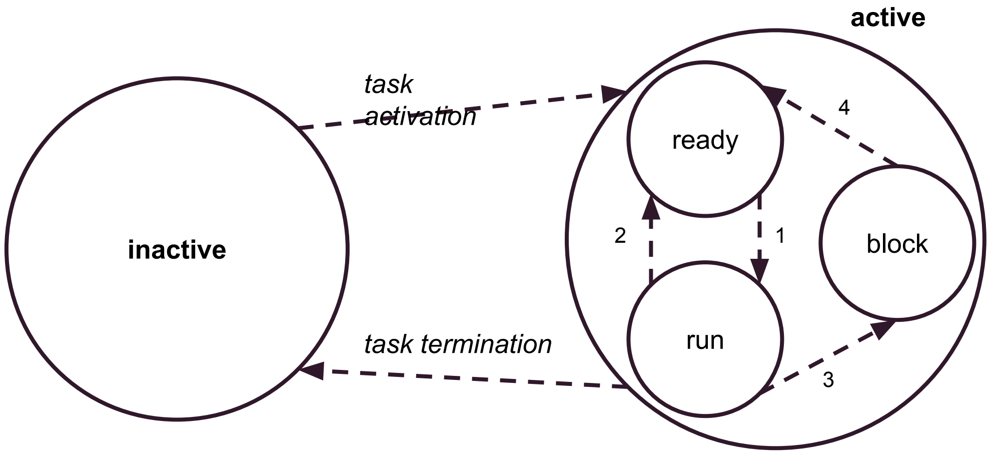
Depending on the outcome of the dynamic scheduling algorithm, the new task will be selected for execution or the interrupted task will be continued.



Complex Tasks

The WCET of a C-task cannot be determined independently of the other tasks in the node. It can depend on the occurrence of an event in the node environment, e.g. waiting for an input message.

The timing analysis is not a local of the C-task, but a system issue. In the general case it is impossible to give an upper bound for the WCET of a C-task.



1 scheduler decision
2 preemption
3 task calls wait on event
4 wait event occurs

Software Portability

The complexity of the Application Program Interface (API) determines the portability of the application software.

The simplest application program interface is the API of a time- triggered S-task. The coupling between the application software and the operating system increases with the number / variety of operating systems calls.

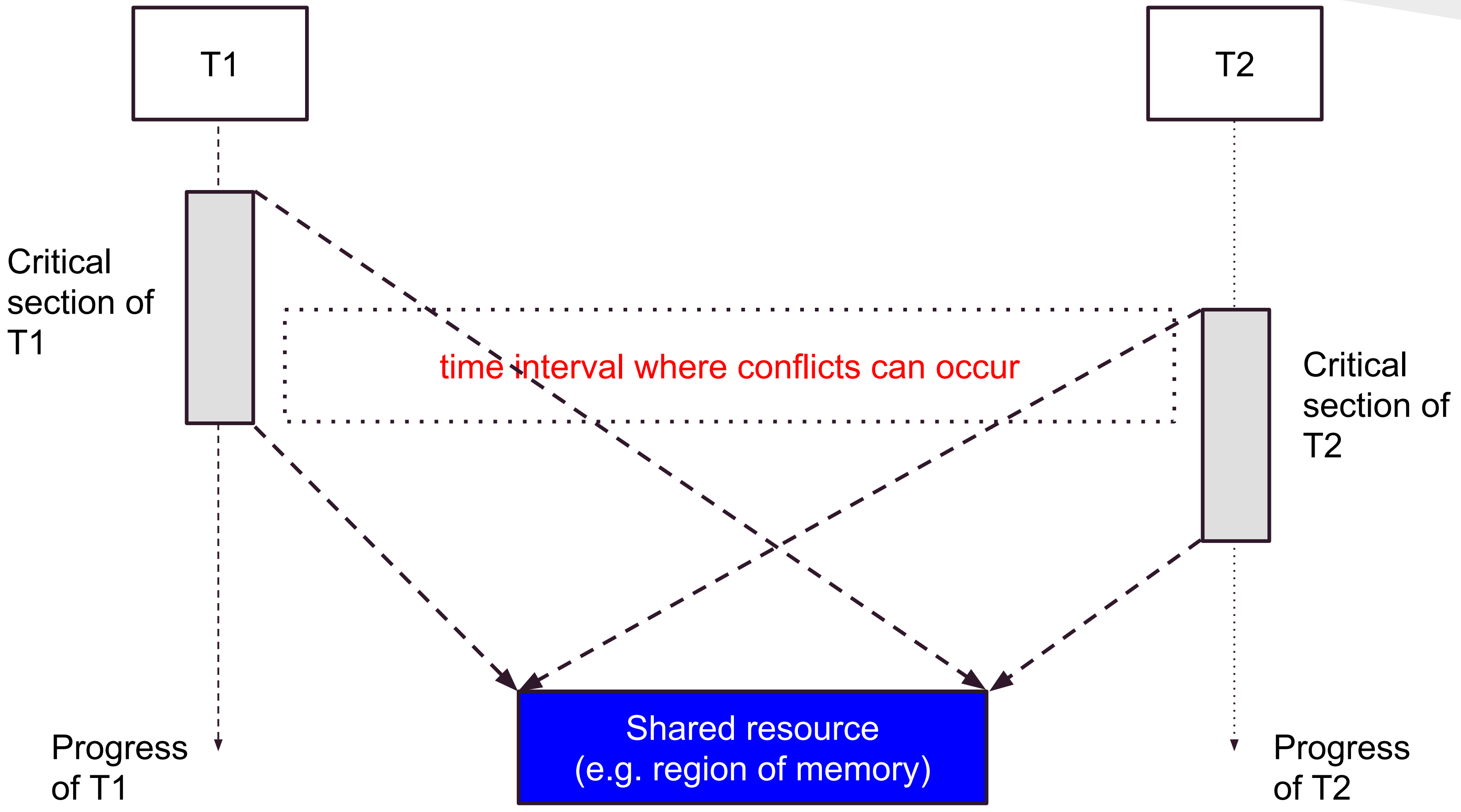
Interprocess Communication

Interprocess communication is needed to exchange information among concurrently executing tasks. There are two possible types of information exchange – the direct exchange of messages among the involved tasks and the indirect exchange of information via a common region of data (e.g. shared memory).

If interprocess communication is based on messages, a choice must be made between event-message semantics and state-message semantics.

State messages support the information exchange among tasks of differing execution periods. Therefore, state message semantics match better the need of real-time applications.

Critical Sections


T1
T2
Critical section of T1
time interval where conflicts can occur
Critical section of T2
Shared resource (e.g. region of memory)
Progress of T1
Progress of T2

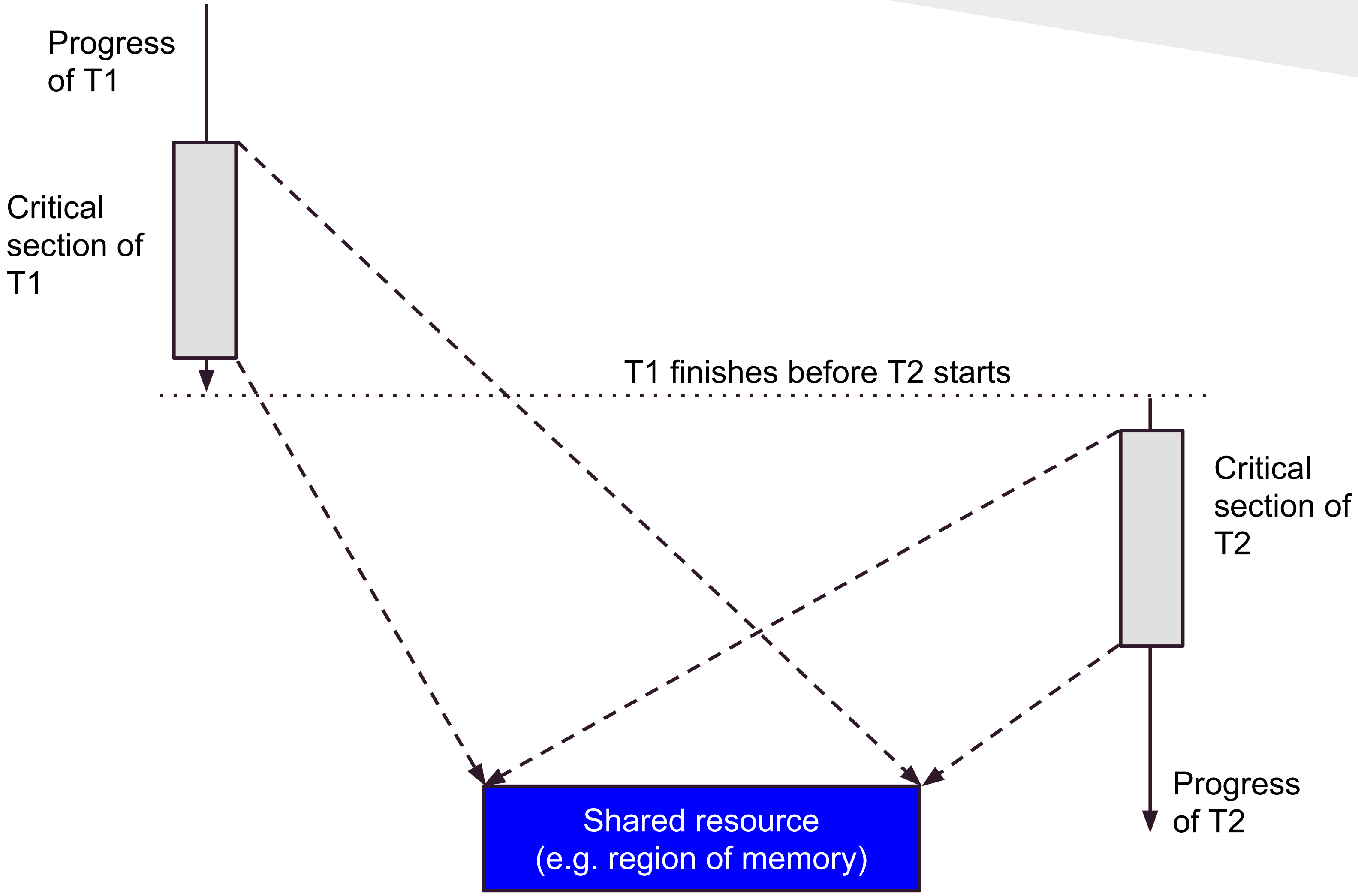
Interprocess Communication

The indirect exchange of information by a common region of data is related to the state message mechanism. The main issue is the missing atomicity property of common memory. It is thus a low-level concept of data sharing, leaving it up to the application tasks to implement data consistency. The “classic” mechanism is to enforce mutual exclusive access via semaphores.

Explicit synchronization of tasks by semaphore operations can be very costly if the protected region of data is small (this is the case in many real-time applications). Then, the processing time for the semaphore operations take hundreds of times longer than the actual access to the data.

Implicit synchronization by properly designed state schedules – accessing the same region of data will never overlap – is order of magnitude cheaper.

Implicit Synchronization



Non-Blocking Write Protocol

The implementation of the atomicity property of state messages that are exchanged between the host computer and the communication controller at the Communication Network Interface (CNI) requires special consideration.

Data exchange at the CNI can be protected by the efficient non-blocking write (NBW) protocols because it can be difficult to exercise back-pressure flow control on the sender.

In this example, there is one writer (the communication system) and many readers (the tasks of the host). A reader does not destroy the information, but the writer can interfere with the operation of the reader. In the NBW protocol, the writer is never blocked. Thus, the reader can read an inconsistent message. If the reader is able to detect the interference, the reader can retry the operation. It exists an upper bound for the number of retries.

Non-Blocking Write

```
/* counter volatile init with 0 */
```

```
Writer:
     counter_old= counter;
     counter= counter_old + 1;

     /* write to shared memory */

     counter= counter_old + 2;
```

```
Reader:
start:
     counter_start= counter;

     if ( is_odd ( counter_start ) )
          goto start;

     /* read from shared memory */

     counter_end= counter;

     if ( counter_end != counter_start )
          goto start;
```

Error Detection

A real-time operating system must support error detection in the temporal domain and error detection in the value domain by generic methods.

Some important methods are

1. Monitoring task execution times
2. Monitoring frequency of interrupts
3. Double execution of tasks
4. Watchdog
5. Exception Handling

Monitoring Task Execution Times

A tight upper bound on the worst-case execution time (WCET) of a real-time task must be established during software development. This WCET must be monitored at runtime to detect transient or permanent hardware (or software) errors.

In case a task does not terminate its operation within the WCET, the execution must be terminated by the operating system and an **error handling strategy** must be enforced.

Monitoring Interrupts

Erroneous external interrupts are very dangerous. Therefore, at design time, the minimum inter-arrival periods must be known to be able to estimate the peak load that must be handled by the software system.

At run time, this minimum inter-arrival period must be enforced by the operating system by disabling the interrupt line to reduce the probability of erroneous sporadic interrupts.

Double Execution of Tasks

Fault-injection experiments have shown that the double execution of tasks and the subsequent comparison of the results is a very effective method for the detection of **transient hardware faults** that cause undetected errors in the value domain.

An operating system can provide the execution environment for the double execution of application tasks without changing the application task per se.

Watchdog

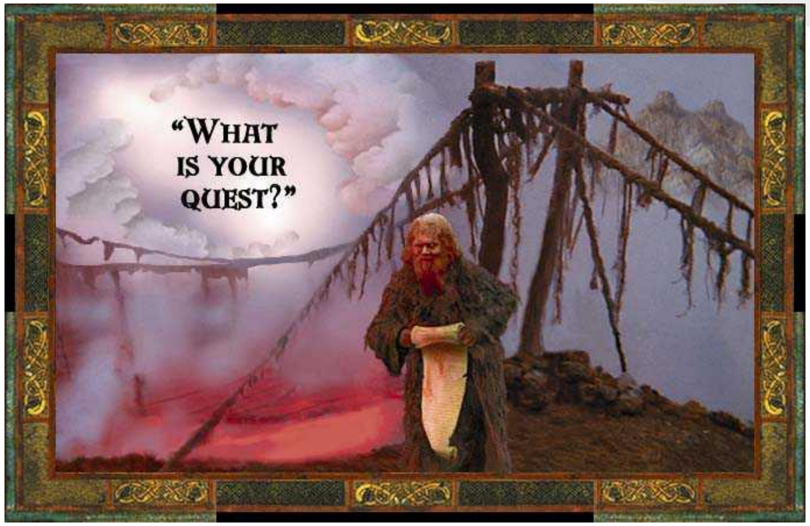
A standard technique for error detection is the provision of a **watchdog signal** (heartbeat) that must be periodically produced by the operating system of the node. An independent outside observer (e. g. watchdog chip) can detect the failure of the node as soon as the watchdog signal disappears.

An end-to-end monitoring is possible if the watchdog signal is produced by the application task instead of the operating system. Then also the execution of the application task is covered.

Challenge Response Protocol

A more sophisticated error detection mechanism that also covers part of the value domain is the periodic execution of a **challenge-response protocol** by the operating system of a node.

An outside error detector provides an input pattern to the node and expects a defined response pattern within a specified time interval. If there is a deviation from the a priori known result, a node error is detected.



Exception Handling

Exception handling is a well-known technique for handling errors that are detected within a task. In real-time systems exception handling must be used with care. The WCET of a task is extended by the WCET of all exception handlers that can be possibly be activated during the execution of the task.

This may result is very pessimistic WCET values.

A long lasting exception handling routine for a low priority task may steal computation time from a high priority task leading to the violation of critical deadlines (see Ariane-5 example).

Real-Time Scheduling

1. The Schedulability Test
2. Dynamic Scheduling
3. Examples
4. Priority Ceiling Protocol
5. Static Scheduling

Overview

Real-time scheduling is concerned with the optimal activation of tasks with real-time constraints.

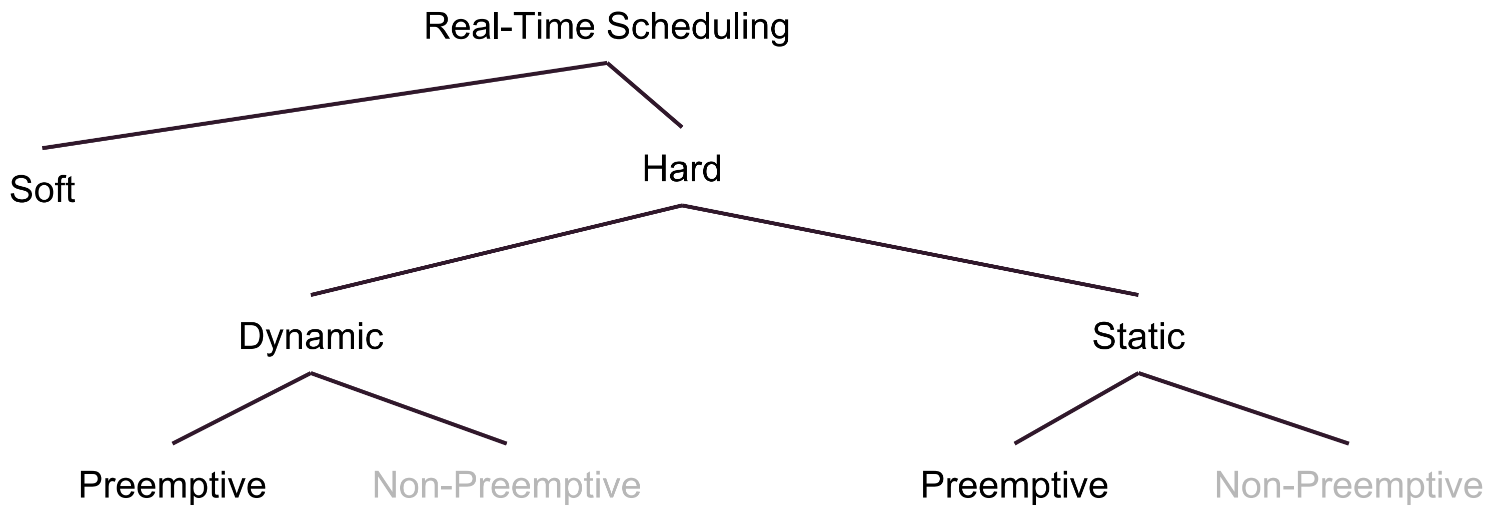
Many thousand of research papers have been written about how to schedule a set of tasks in a system with a limited amount of resources such that all tasks will meet their deadlines.

This module focuses on the foundation of real-time scheduling and presents some well-known algorithms. Furthermore, it discusses scheduling for time-triggered systems.

The Scheduling Problem

A hard real-time system must execute a set of concurrent real- time tasks in such a way that all time-critical tasks meet their specified deadlines.

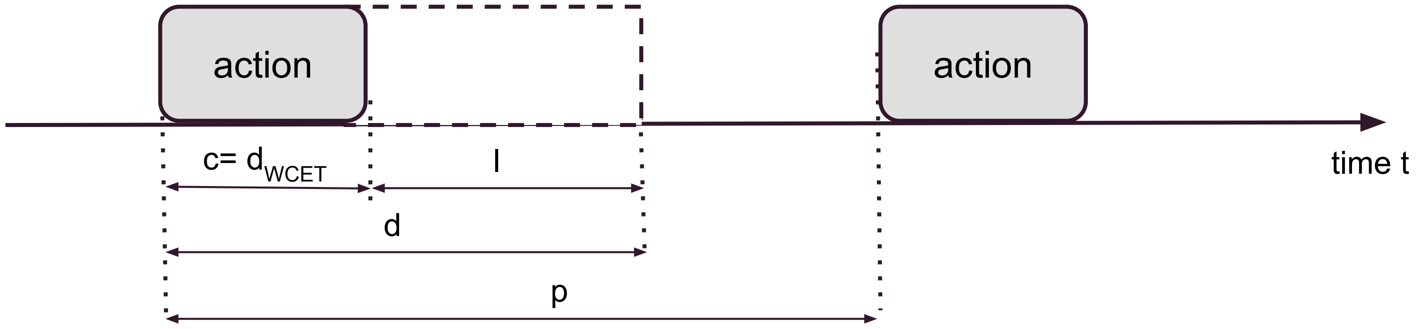
Every task needs computational and data resources to proceed. The scheduling problem is concerned with the allocation of these resources to satisfy all timing requirements.



New Parameters of Action

Additional quantities of actions result from the periodic or sporadic execution e. g. in a task:

1. Old quantities: $d_{act}(a,x)$, $d_{min}(a)$, $d_{WCET}(a)$, jitter: $d_{WCET}(a) - d_{min}(a)$
2. Computation time $c = d_{WCET}(a)$
3. Period p
4. Deadline d
5. Laxity l



Classification

A scheduler is called **dynamic** (or online) if it makes its scheduling decisions at run time, selecting one out of the current set of ready tasks. Dynamic schedulers are flexible, but require substantial run time effort (WCAO).

A scheduler is called **static** (or pre-runtime) if it makes its scheduling decisions at compile time. It generates a dispatching table for the run-time dispatcher offline. The run-time overhead of this dispatcher is small.

In **preemptive** scheduling the currently executed task may be preempted, i.e., interrupted, if a more urgent task requests service. In non-preemptive scheduling, the currently executing task will not be interrupted until it decides on its own to release the allocated resources – normally after completion.

Non-preemptive scheduling is reasonable in a task scenario where many short tasks must be executed.

Schedulability Test

A test that determines whether a set of ready tasks can be scheduled so that each task meets its deadline is called a **schedulability test**. We distinguish between **exact**, **sufficient**, and **necessary** schedulability tests.

A scheduler is called optimal, if it will always find a schedule provided provided an **exact** schedulability test indicates the existence of such a schedule. In nearly all cases of task dependency, even if there is only one common resource, the complexity of an exact scheduling algorithm is computationally intractable.

Sufficient schedulability test algorithms can be simpler at the expense for a negative result of some task sets that are in fact schedulable.

A task set is definitely not schedulable, if a **necessary** schedulability test gives a negative result. But, if positive, it may not be schedulable.

Example

A **necessary** schedulability test for a set of periodic tasks states that the sum of utilization factors

$$\mu=\sum c_i/p_i \leq n$$

(c .. execution time, p .. period)

must be less or equal to n, where n is the number of available processors.

This is evident because the utilization factor μ_i denotes the percentage of time the task T_i requires service from the processor.

The Task Request

The moment when a request for a task execution is made is called the task request time. Starting with an initial request, all future request times of a periodic task are known a priori by adding multiples of the known period to the initial request time.

Of a sporadic task, only the minimum inter-arrival time is known in advance – the actual points in time to service the task are unknown. An aperiodic task has no constraints.

Dynamic Scheduling

After the occurrence of a significant event, a dynamic scheduling algorithm determines online which task out of the ready task set must be serviced next.

The algorithms differ in the assumptions about the complexity of the task model and the future task behavior. Scheduling can be done either for independent or for dependent tasks.

Rate Monotonic Algorithm

The Rate Monotonic (RM) Algorithm is the classic algorithm for scheduling a set of periodic independent hard real-time tasks in a system with a single CPU. The RM algorithm assigns static priorities based on the task periods. The task with the shortest period gets the highest static priority, etc. If all assumptions are satisfied, RM guarantees that all task will meet their deadlines.

The rate monotonic algorithm is a dynamic preemptive scheduling algorithm based on static task priorities. It assumes a set of periodic and independent tasks with deadlines equal to their periods and constant computation times.

The sum of utilization factors μ of n tasks is given by

$$\mu=\sum c_i/p_i \leq n(2^{1/n} - 1)$$

The term $n(2^{1/n}-1)$ approaches $\ln 2 \approx 0.7$, as n goes to infinity.

Earliest-Deadline-First Algorithm

The Earliest-Deadline-First (EDF) Algorithm is an optimal dynamic preemptive algorithm in a single processor system, which is based on dynamic priorities. After any significant event, the task with the earliest deadline is assigned the highest dynamic priority.

The assumptions of the RM algorithm (except μ) must hold. The processor utilization can go up to 1.

Least-Laxity Algorithm

In a single processor system, the Least-Laxity (LL) Algorithm is another optimal algorithm. It makes the same assumptions as the EDF algorithm. At any scheduling decision point the task with the shortest laxity l,

$$l = d - c,$$

i.e., the difference between the deadline interval d and the computation time c is assigned the highest dynamic priority.

Other scheduling algorithms

The **round-robin** algorithm is not optimal since it does not take any real-time parameters into account but only the order of execution in the past.

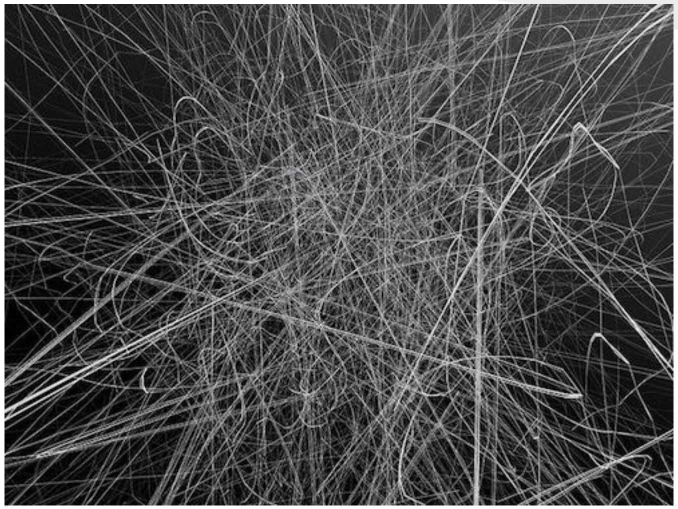
The current Linux scheduling algorithm (**Completely Fair Scheduler**) is not optimal since it takes into account only the duration of execution in the past.

Scheduling Dependent Tasks

From a practical point of view, results on how to schedule tasks with precedence and mutual exclusion constraints are much more important than the analysis of independent task models.

It is prohibitively expensive to look for an optimal schedule for a set of dependent tasks.

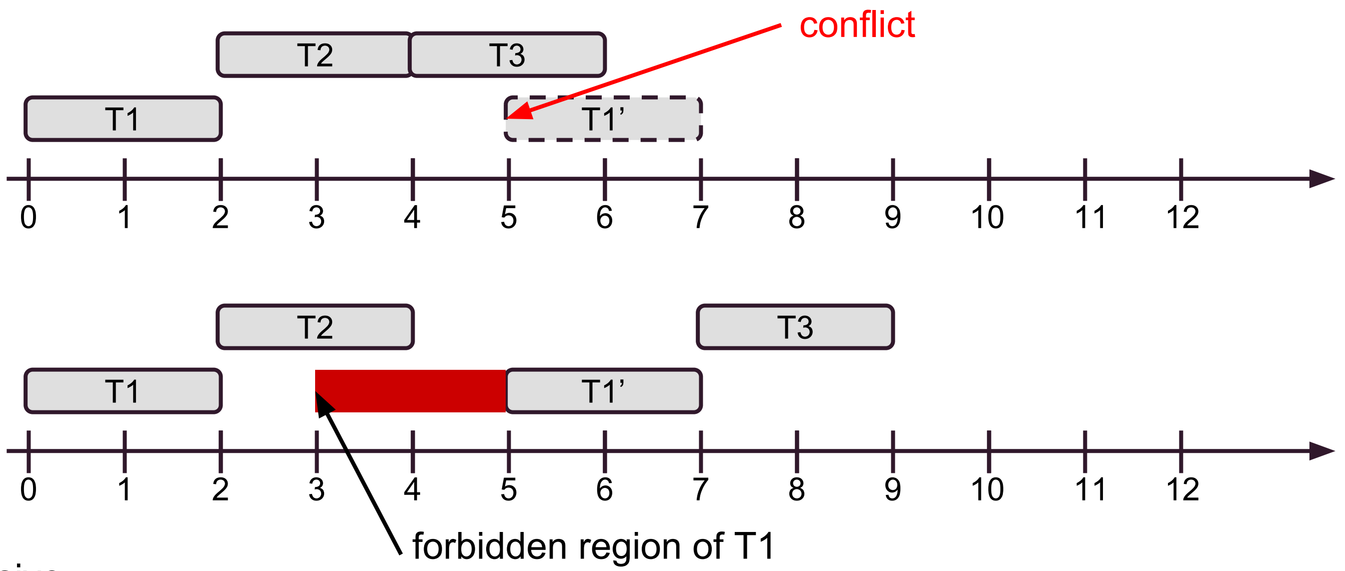
The more resources are spent for scheduling, the fewer resources remain available to perform the actual work (WCAO).



Kernelized Monitor Algorithm

The Kernelized Monitor Algorithm [3] allocates the processor time in uninterruptable quota of duration q, assuming that all critical sections can be started and completed within this time quantum (preemption time quantum).

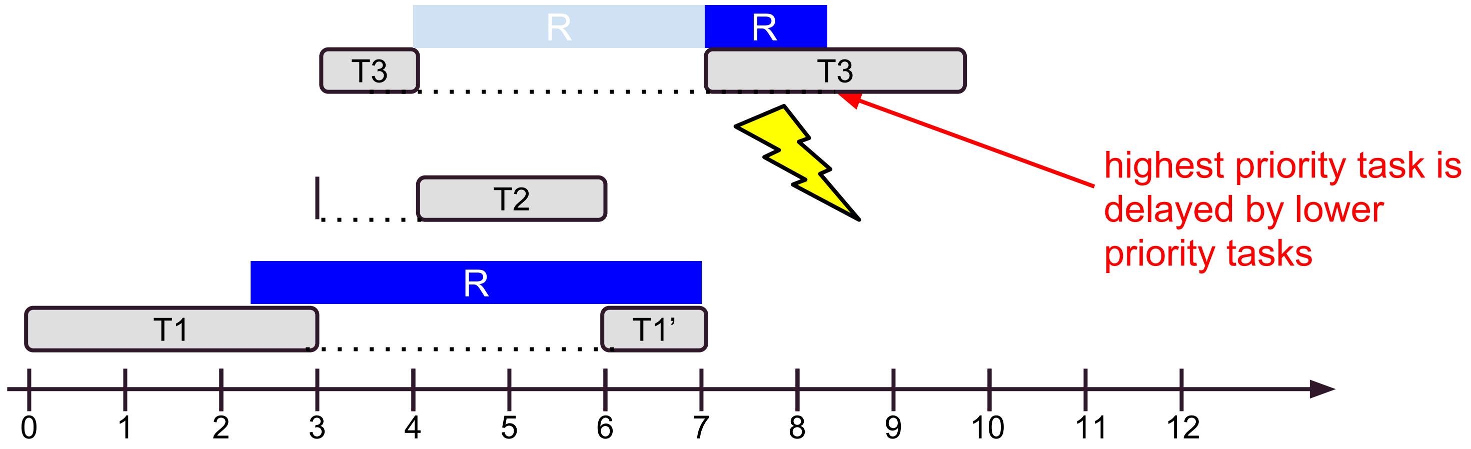
A **forbidden region** must be generated to guarantee on-time service of requests. During compile time, all forbidden regions must be determined, and passed to the dispatcher to make sure no unwanted critical sections are scheduled inhere.



T1: c=2,d=2,P=5
T2: c=2, d=10, P=10
T3: c=2, d=10, P=10
T3 and T1 mutually exclusive

Priority Inversion

Priority inversion takes place if a low priority task is preempted by a higher priority task which wants to access a shared resource that is blocked by the low priority task. In that situation, a medium priority task may prevent the low priority task to complete the critical section and the high priority task cannot run until the medium priority and the low priority task have completed.



T1: Prio=1
T2: Prio=2
T3: Prio=3
T3 and T1 access resource R

Priority Ceiling Protocol

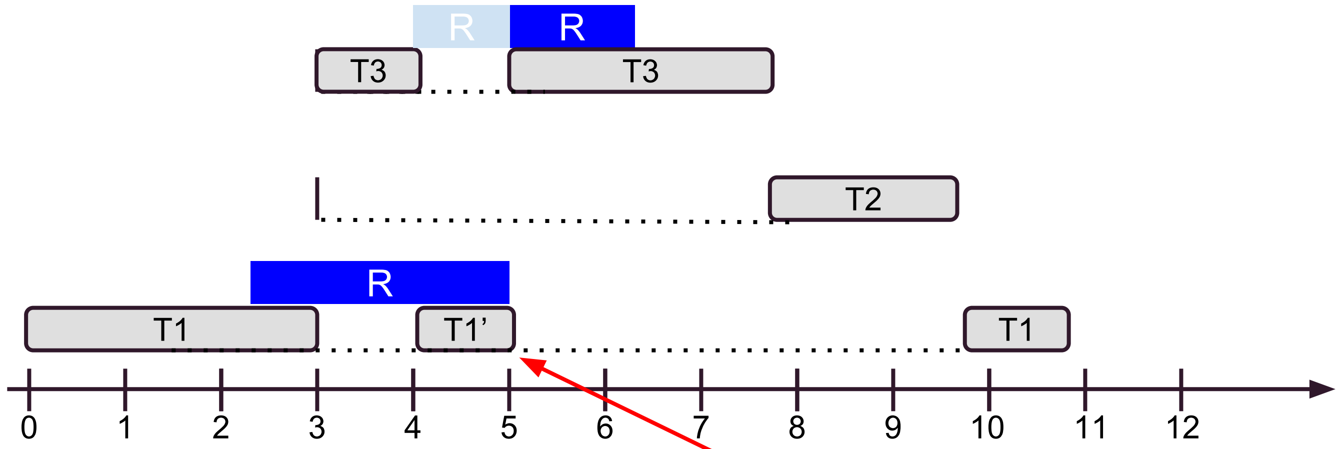
The priority ceiling protocol is used to schedule a set of periodic tasks that have exclusive access to common resources protected by semaphores. It was developed to solve, e.g., the problem of **priority inversion**.

The **priority ceiling** *PC(S)* of a semaphore *S* is defined as the priority of the highest priority task that may lock this semaphore.

1. A task T is allowed to enter a critical section only if its assigned priority is higher than the priority ceilings of all semaphores currently locked by tasks other than T.
2. Task T runs at its assigned priority unless it is in a critical section and blocks higher priority tasks. Then, it gets the priority of the task he blocks.
3. When exiting a critical section, it resumes the priority it had at the entry point into the critical section.

Priority Ceiling Example

This example shows the same setting as before with priority ceiling protocol in action.



T1 runs at Prio of T3 until T1 releases R

T1: Prio=1
T2: Prio=2
T3: Prio=3
T3 and T1 access resource R, PC(R)=3

Scheduling in COTS Systems

COTS … commercial-off-the-shelf

System	Scheduling Policy	Comment
vxWorks [4]	priority based round robin	
RTLinux [4]	FIFO, EDF, rate monotonic	API for extensions
AUTOSAR [5]	priority based and time tables	
eCos [6]	priority based	API for extensions

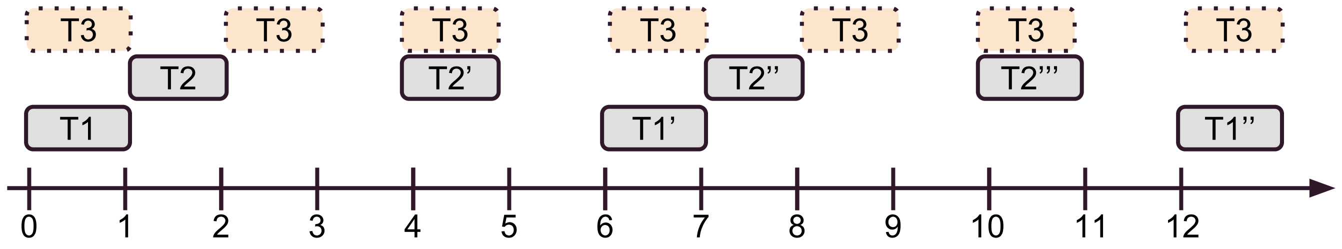
Static Scheduling

In static scheduling, a feasible schedule of a set of tasks that guarantees all deadlines, considering the resources, precedence, and synchronization requirements of all tasks, is calculated off-line before the runtime of the system.

A static schedule is a **periodic time-triggered** schedule that is repeated after the schedule period. The timeline is partitioned into a sequence of basic granules, the basic cycle time.

There is only one interrupt in the system: a periodic clock interrupt denoting the start of a new basic granule.

Patterns for Static Schedules



T1: p=6 (period)
T2: p=3
T3: p=2 (not feasible)

one more task with p=6 and one more task with p=3 would fit.

Increasing Flexibility

One of the weaknesses of static scheduling is the assumption of strictly periodic tasks. Although the majority of tasks in hard real-time applications is periodic, there are also sporadic requests for service that have hard deadline requirements.

Three techniques can increase the flexibility in static scheduling:

1. transformation of sporadic tasks into periodic tasks,
2. periodic server tasks,
3. and mode changes.

Transform into Periodic Task

It is possible to find solutions to the scheduling problem if an independent sporadic task has a **laxity**. The transformation to a **pseudo-periodic task** will always meet its deadline if the pseudo-periodic task can be scheduled (similar to trigger task).

A **periodic server task** is a placeholder for aperiodic tasks. Whenever a sporadic request arrives during the period of the server task, it will be serviced with the high priority of the server task.

Parameter	Sporadic Task	Pseudo-Periodic Task
Computation Time c	c	c’=c
Deadline d	d	d’=c
Period p	p	p’ < min(l, p)

Mode Changes

During the operation of most real-time applications a number of different **operating modes** can be distinguished (e.g., plane is taxiing, plane is flying).

During system design, all possible modes are identified and static schedules that meet all deadlines are calculated off line. Whenever a mode change is requested in runtime, the new schedule will be activated immediately.

A major challenge with mode changes is the switch between the execution of one mode to a new mode. Basically such a switch is only possible in the ground-state of the application.

Sources

[1] Real-Time Systems: Design Principles for Distributed Embedded Applications, Hermann Kopetz, 1997, Springer Verlag

[3] Real-Time Systems: Scheduling, Analysis, and Verification, Albert M. K. Cheng, John Wiley and Sons Inc. 2002

[4] http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=6BB810DD09C3305E287D0AB7065C0D1E?doi=10.1.1.114.8870&rep=rep1&type=pdf

[5] http://www.it.fht-esslingen.de/~zimmerma/automotive/kapitel7.pdf

[6] http://ecos.sourceware.org/ecos/docs-latest/ref/kernel-overview.html