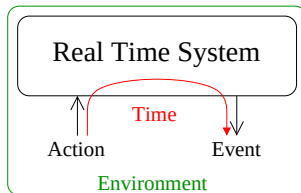


RTS1 Summary

Matthias Jung

22. Februar 2010

1 Realtime



Definition of a Real Time System:

A real-time system is a system that reacts to events in the environment by performing predefined actions within specified time intervals.

Time Correctness:

Correctness of results depends on value and its time of delivery

Deliver the correct value too late (or too early) is wrong! Deadlines must be met.

Time - *Unreal*

- Computer World: (computer time)
 - Computer controls speed of user,
 - User can be halted or annoyed...

Time - *Real*

- Real World: (**real time**)
 - environment controls speed of computer
 - environment cannot be halted
 - catastrophic consequences

Fastness:

Fast enough is dependent on the system and its environment. (e.g. a fly is fast enough to escape a fly swat) The time scale depends on the environment or it is dictated by the environment. In the real world we can not slow down the environment

Average vs. Worst Case

Standard computing is concerned with the average behavior of the system. But real time computing requires timely behavior in all given situations, so the worst case is assumed.

(e.g. Person drowned in river, average depth of 20cm)

Fast is not enough, average is not enough, performance testing is not enough, → worst case guarantees

1.1 Terms

Task:

- Program in execution
- Perform service
- requires resources (e.g. execution time)

Deadline:

- Specified time for completion if e.g. task
- Time interval or absolute point in time.
- Value of result may depend on completion time

Repetition of Tasks

Periodic:

- activity occurs repeatedly
- number of instances
- e.g. to monitor the environment values (temperature...)

Aperiodic:

- can occur any time
- no arrival pattern given

Sporadic:

- can occur any time
- minimum time between arrivals

Dataflow:

- Data are read from environment (e.g. sensors)
- Data are processed by tasks
- Results are processed by other tasks
- Data are transferred to environment (actuators)

Synchronization:

some activities may run only:

- After others have completed (**precedence constraints**)
- While others are not running (**mutual exclusion**)
- Within certain times (**temporal constraints**)

Importance:

- Which parts of the system are important?
- Importance can change over time

Flow Control:

Who has control over speed of processing, who can slow partner down?

- Environment
- Computer System

RT: environment cannot be slowed down!

Types of Real Time Systems:

Hard:

all deadlines (temporal constraints) have to be met.

Soft:

missing single deadline does not jeopardize system behavior

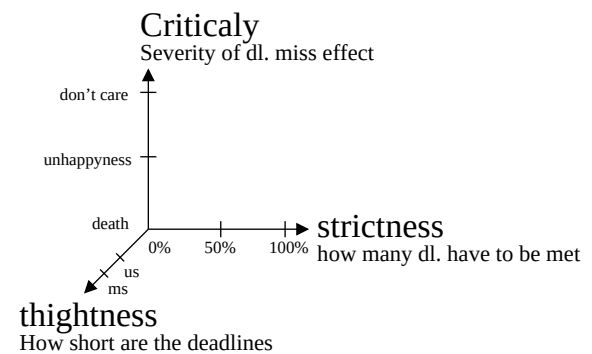
Adaptive:

adjust its internal strategies to respond to environmental changes

Classical View:

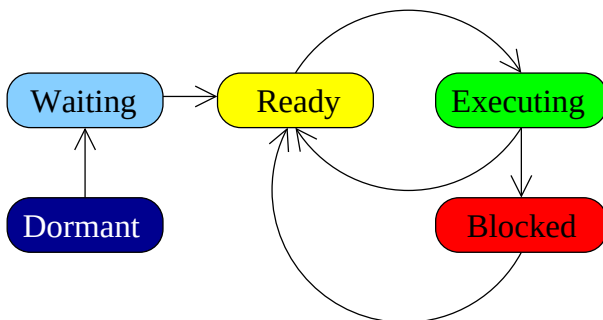
"Hard real-time systems: a single deadline miss can lead to catastrophic consequences, including loss of human life."

Sophisticated View:



2 Scheduling:

Lifecycle of a Task



Dormant:

Sleeping task is in the system but not considered for execution.

Waiting:

Task is waiting for event.

Ready:

Task can execute when the dispatcher decides so.

Executing:

Task is running and uses the CPU

Blocked:

Task cannot execute because it is waiting for resources

Temporal Definition of Periodic Task:

T_i : period of task i

R_i : release time, relative from start of period time at which task i may start to execute

D_i : deadline interval, relative from start of period time at which task i must be completed

C_i : worst case execution time

Utilisation of a Single Task:

$$U_i = \frac{C_i}{T_i}$$

Utilisation of task set with n tasks

$$U = \frac{C_0}{T_0} + \frac{C_1}{T_1} + \dots + \frac{C_{n-1}}{T_{n-1}} = \sum_{i=0}^{n-1} \frac{C_i}{T_i}$$

Schedulability:

A task set is *schedulable* when all tasks meet their deadlines during the lifetime of the task set / system.

Necessary Condition: $U \leq 1$ has to hold, otherwise the taskset is not schedulable, but if it holds it does not mean automatically that it is schedulable.

2.1 Scheduling Policies:

First Come First Serve:

- CPU given to tasks in order of request arrival
 - CPU held by task until it finishes
- non preemtive

Time Sliced:

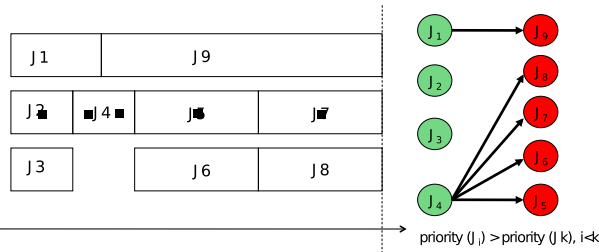
- CPU given to each task for a certain amount of time
- has to be preemptive
- assignment of slices to task with other policies (e.g. *round robin*)

Priority Based:

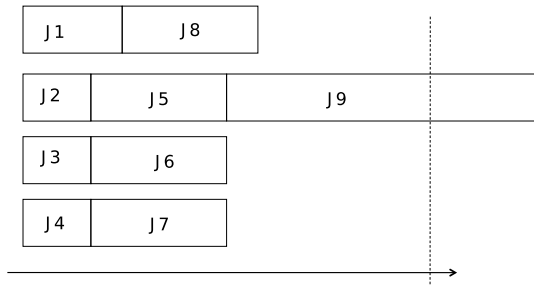
- Priorities are assigned to tasks
- Task with the highest priority gets the CPU next
 - if preemptive: executes directly
 - if non-preemptive: executes after the current running task
- priorities stay the same during lifetime: **fixed priority FPS**
- priorities may change during lifetime: **dynamic priority DPS**

2.2 Scheduling Anomalies

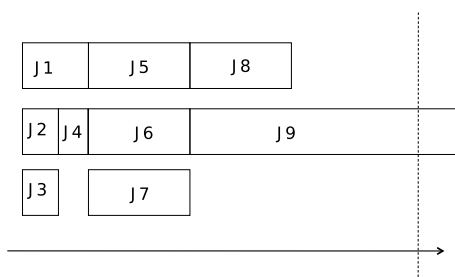
Fixed Priority Scheduling:



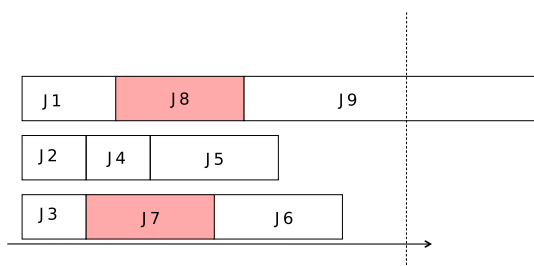
Add More Processors:



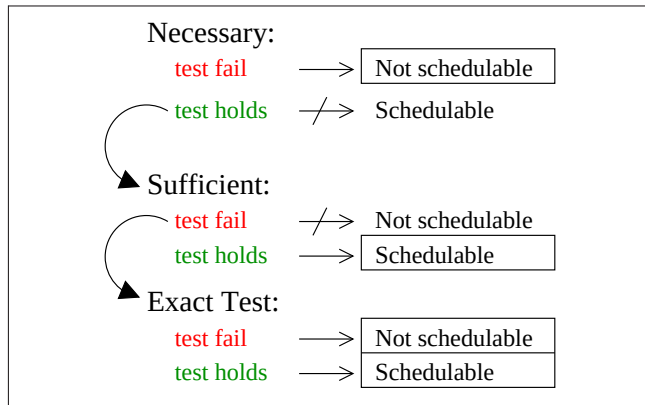
Reduce Computation Times:



Weakened precedence Constraints:



2.3 Schedulability Tests

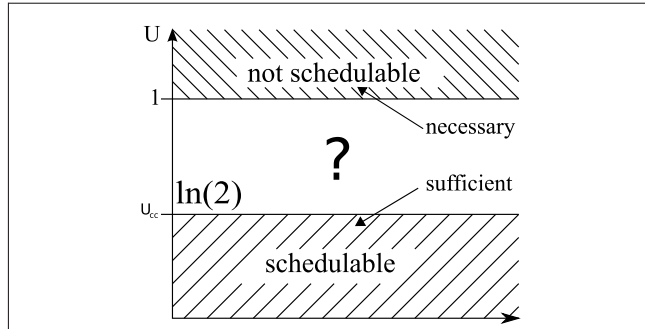


Necessary Test:

A schedulability test is *necessary* if all task sets that fail are definitely not schedulable. But there may exist task sets that pass the test which are not schedulable.

Sufficient Test:

A schedulability test is *sufficient* if all tasks sets that pass are definitely schedulable. But there may exist task sets that fail the test, which are schedulable.



Proof of Schedulability Test

There is some point in time (*critical instant*) where all of the tasks become ready to execute simultaneously. Low priority tasks will be maximally delayed by higher priority tasks

The proof has to show that the critical instance is really worst case and all tasks meet their deadline even in worst case.

3 Fixed Priority Scheduling (FPS)

3.1 Rate Monotonic (Liu and Layland in 1973)

Assumptions:

- Tasks are periodic with deadlines equal to periods. Release time of tasks is the period start.
- Tasks do not suspend themselves
- Tasks have bounded execution time
- Tasks are independent
- Scheduling overhead is negligible

Priorities:

Task priorities are assigned according to their periods. *shorter period means higher priority*

Run Time:

The ready task with the highest priority is executed.

Sufficient Test:

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq n \cdot (2^{\frac{1}{n}} - 1)$$

Necessary Test:

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

Exact Test: (exact analysis by Joseph and Pandya)
Based on critical instance analysis as well. (longest response time of task, when it is released at same time as all higher priority tasks)

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot C_j$$

$hp(i)$: set of tasks with an higher priority than task i

We need iterations for each task and this is not so nice to calculate!

$$?? R_i \leq D_i ??$$

Comments:

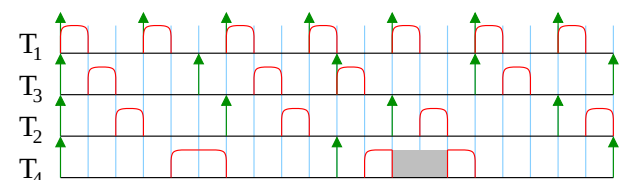
- Rate monotonic is an optimal scheduling policy among all static priority assignments
- Sufficient but not necessary
- Single processor
- artificial correlation *period* and *priority*

Example:

$T_1 = (3, 1), T_2 = (6, 1), T_3 = (5, 1), T_4 = (10, 2)$

$U = 0.9 \leq 0.75683$

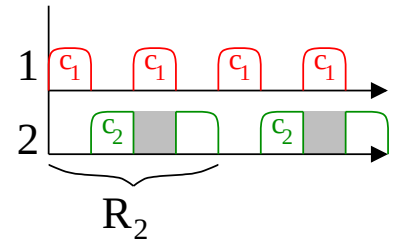
Exact Test:



Example: (1. it.) What is happening at the critical instance?

- Let T_1 be the highest priority task. Its response time $R_1 = C_1$ since it cannot be preempted
- What about a task T_2 ? $R_2 = C_2 +$ 'all the time it is interrupted by T_1 '. That means it will interrupt T_2 at least one time, probably more often. Assume T_1 has half the period of T_2 .

$$R_2 = C_2 + 2 \cdot C_1$$



3.2 Deadline Monotonic:

In rate monotonic the deadlines are at the end of the period. But this is not realistic. Deadline monotonic allows $D < T$. The priorities are assigned according to the deadlines (shorter deadline higher priority). The schedulability test will be done by response time analysis.

4 Dynamic Priority Scheduling

4.1 Earliest Deadline First (EDF)

Assumptions:

Same as rate monotonic

Priorities:

Tasks with the closer deadlines get higher priority

Run Time:

The ready task with the highest priority is executed (the one with the closest deadline). Sort priorities such that tasks with closer deadlines get higher priority at run-time as well (dynamic priorities).

Schedulability Test:

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

Sufficient and necessary condition.

EDF is optimal if utilization is below 1 and performs badly under overload. Optimal means, that if any algorithm can find a schedule, EDF will find as well.

4.2 Earliest Deadline First - Processor Demand Analysis (Baruah in 1990)

if $D < T$ we have to check:

Check all intervals L from 0 to each Deadline until all intervals up to lcm are checked or one interval failed some deadline

$$\forall L \in D : L \leq \sum_{i=1}^n \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) \cdot C_i$$

$$D = \{d_{i,k} | d_{i,k} = kT_i + D_i, d_{i,k} \leq lcm, 1 \leq i \leq n, k \geq 0\}$$

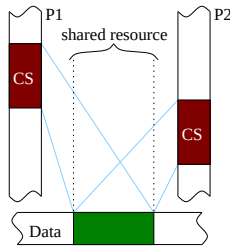
Overview:

	$D = T$	$D < T$
Fixed Priority	$\sum_{i=1}^n \frac{C_i}{T_i} \leq n \cdot (2^{\frac{1}{n}} - 1)$	$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot C_j$
Dynamic Priority	$\sum_{i=1}^n \frac{C_i}{T_i} = U \leq 1$	$\forall L \in D : L \leq \sum_{i=1}^n \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) \cdot C_i$

5 Real-Time Synchronization

Mutual Exclusion (from OS)

Two (or more) tasks access the same memory area (e.g. one task writes, the other reads). If the accesses happen at the same time we got inconsistent data.



Semaphores:

- OS constructs
- Integer values
- Access via atomic functions `wait()`, `signal()`
- Control access to critical section
- Tasks in wait are blocked

Example:

```
wait(S);  
    critical section  
signal(S);  
    remainder section
```

Priority Inversion:

Given:

High priority task H , a medium priority task M and a low priority task L . Task L and H have a critical section.

Assume:

Task H is blocked by L due to semaphors.

Observation:

Task M will preempt task L because it has a higher priority and so task H is delayed (deadline miss...)

5.1 Priority Inheritance Protocol - PIP: (Sha, Rajkumar, Lehoczky in 1990)

Priority Inheritance:

How can we prevent that H waits longer than necessary and how can we ensure that M does not execute during CS of L ?

→ give L during execution of CS the priority of H

L inherits priority of H

Only tasks which could also preempt H can preempt L , priority inversion cannot occur

Priority Inheritance Protocol:

- A task L inherits the priority of the highest priority task **currently blocked** by the same semaphore.
- When L leaves critical section it resumes the priority it had when it entered the critical section.
- Prevents priority inversion
- Task blocked only by critical sections of lower priority tasks and is not delayed.

Comments:

- Priority inheritance does not prevent deadlock
- How many times can H get blocked by lower priority tasks with PIP?
Once for each semaphore it uses (*Chained Blocking*)

5.2 Priority Ceiling Protocol - PCP (Sha, Rajkumar, Lehoczky in 1990)

Assumptions:

- Each task has an assigned static priority (may change at runtime due to priority inheritance)
- Each semaphore has a static priority **ceiling** value defined as:
 - maximum priority of all tasks that may use it: $\text{ceil}(S)$ = highest priority of all tasks that will request S
 - calculated offline

Priority Inheritance Protocol:

It has the same run-time mechanism as in Priority Inheritance Protocol. A request to a semaphore S is granted if:

- S is free and
- The priority of the *requesting task* is higher than all ceilings of semaphores allocated to tasks other than the *requesting task* at the time

Schedulability of PCP:

The basic algorithm assumption (highest priority task executes) is violated. We have to consider additional blocking times B_i . Maximum blocking time in PCP is the duration of the longest critical section.

Response Time Analysis:

$$R_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot C_j$$

Immediate Priority Inheritance Protocol:

Same assumptions as PCP, but when the task locks a semaphore it immediately raises its own priority to the ceiling of the semaphore.

- task will only suffer a blocking at the very beginning of its execution
- once the task starts actually executing:
 - all the resources it needs must be free
 - if they were not, then some tasks would have an equal or higher priority and the tasks's execution would be postponed

Comments on PCP:

- + No priority inversion, deadlocks, chained blocking
- + Task blocked by max one critical section of lower priority task
 - Priorities may be raised unnecessarily
 - Lower priority task delay higher priority tasks without blocking as well

5.3 Stack Resource Policy - SRP (Ted Baker in 1991)**Stack-based scheduling of real-time processes:**

A stack can block processes! when a process preempted, cannot resume until processes holding stack above are completed. To Solve this problem we can use a SRP.

- SRP is a refinement of PCP
- Multi-unit resources
- Sharing of runtime stack-based resources
- Separates priority and preemption level

5.4 Alternatives to Semaphores - Non Blocking:**Dual Port:**

- Keep more than one copy of data
- Data can be accessed concurrently

Non Blocking Write Protocols:

- Counter for the data.
- Every write count +2
(+1 at the begin and +1 at the end)
- A read is inconsistent if:
 - the counter value is uneven at the beginning of the read, or
 - the counter value at the end of the read is unequal to the beginning of the read

TODO TODO TODO

6 Real-Time Systems Communication

System Communication:

Many applications demand a predictable message passing with low latency. Systems are becoming distributed and have small distributed nodes instead of a big central.

Examples:

- Factory automation
- Automobile
- Home entertainment

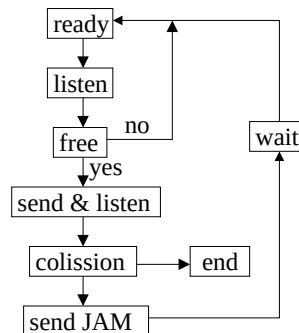
Types of Networks:

- **Special purpose networks**
 - Field-buses
 - Special Protocols
 - Expensive
- **General purpose networks**
 - Local Area Networks
 - Widely used and stable
 - Cheap

Real-time Requirements:

- **Protocol Latency**: time interval between start of sending on sending node and completed receipt on receiving nodes.
- **Latency Jitter**: variation of latencies for instances of messages.
- **Error Detection**: faults during transmission detected and corrected without jitter increase! (detection by receiver (by sender is too slow))
- **End-to-end acknowledgment**: acknowledge from independent source, e.g. sensor. (Never trust an actuator!)
- **Atomic Multicast**: message received by either all or none - not trivial
- **Node failure handling**: node failure detection, consistent notification of all other nodes, membership protocol

CSMA/CD:



- Large Jitter
- Receiver can delay sender
- Sender detect errors
- Good under low load
- Bad under high load

Positive Acknowledge or Retransmit - PAR:

Retransmission is the resending of packets which have been either damaged or lost. The sending node sends the next package until the receiving node has acknowledged the last package.

6.1 Special Purpose Networks:

Event triggered:

An event activates sending of messages. The access strategy determines the waiting time of messages (similar to CPU scheduling... Response time analysis!). There is a distributed access where all nodes can access bus simultaneously.

How can nodes ensure that only one node is sending at the same time and which?

- Collision detection
- Bit Arbitration

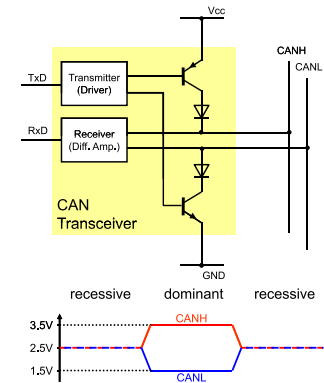
Time Division Multiple Access:

- The network time is divided into slots
- The slots are statically assigned to the nodes
- Implicit flow control
- Time activates sending of messages
- Scheduling table like CPU - **Bus Guardian**
- Only one node sending at time
- Clocks of nodes need to be synchronized
- Often cyclic assignment of slots to nodes
- Simple fault tolerance (active redundancy)
- Simple error detection - message not there at planned time

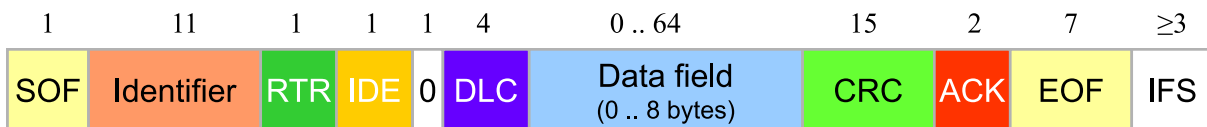
6.1.1 Controled Area Network - CAN (Bosch, Intel in 1983) *Event Triggered*

Properties:

- Broadcast Bus (messages sent to all nodes, only interested node decodes)
- Sender intends to send and puts bit on channel
- Collision detection by CSMA/CD
- Collisions resolved by priorities
- Messages have priorities and an unique identifier
- Two wires (CANH, CANL) reduces effects of electro-magnetic interferences
- Dominant bit '0'
- Regressive bit '1'
- Non-destructive bitwise arbitration



CAN Frame:



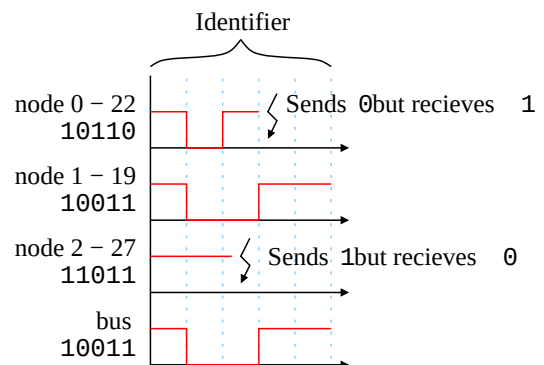
34 bit header and at most 64 bit data

CAN - Access Control - Bit Arbitration

Message priorities by identifier

- Lower numerical value → high priority
- The higher priority message is transmitted as if it were the only one. Collisions don't delay it
- The lower priority message finds out by itself it is not supposed to send stops and retries later
- Sender checks consistency bit by bit
- If monitoring shows inconsistency sender stops

Example for Bit Arbitration



Error Handling:

Upon error detection a node will signal it with a broadcast error flag (six bits with same polarity 000000 or 1111111).

Other nodes will notice and suspend their current reception. There exists mechanisms to disconnect defective nodes to avoid false alarms. The error flag must be avoided anywhere else in the message!
(→ bit stuffing)

Bit Stuffing:

Force one bit of opposite polarity after a sequence of five consecutive bits of equal polarity for:

- to avoid error flag patterns in frame
- for synchronization purposes (many butts of one polarity might desynchronize nodes)

$$S_{Bits} = \left\lceil \frac{34 + 8n - 1}{4} \right\rceil$$

Example:

11111 0000 1111 0000 1111
11111000001111100000111110

6.1.2 Timing Analysis CAN

CAN is scheduled using priorities so we use modified response time analysis in consideration of the WCTT (Worst Case Transmission Time)

Transmission Time:

Equivalent to C_i in tasks: the worst case time that is needed to send a n-bit message.

$$C_i = \left(47 + 8n + \left\lfloor \frac{34 + 8n - 1}{4} \right\rfloor \right) \cdot t_{bit}$$

Worst case: $C_i|_{n=8} = 135 \cdot t_{bit}$

t_{bit} : transmission time of one bit ($1/\text{bus speed}$)

Blocking:

The message sending is non preemptive: transmission not preempted by higher priority message after the arbitration starts. Also blocking can only occur if a low priority message (lpm) starts just before a high priority message (hpm) is ready. ***A hpm cannot preempt a lpm after the first bit of lpm has been transmitted (arbitration started)*** After the lpm finishes hpm will get the bus.

Jitter:

The jitter is the variation on the sending time due to the local scheduling of the sending task. At the worst case the jitter is equivalent to the response time of the sending task:

$$J_i = R_{Task(i)}$$

Time to send the first bit fpr lpm:

Worst case response time:

$$R_i = J_i + w_i + C_i$$

$$w_i = B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j + t_{bit}}{T_j} \right\rceil \cdot C_j$$

- w_i is the time to wait until the first bit is sent
- R_i is the response time for the whole message

6.1.3 Time Triggered Protocol - TTP (Kopetz & Grünsteidl in 1994) TDMA

TTP:

- Time triggered
- Redundant buses and messages for fault tolerance
- Basic idea:
Controller state (time, modes, membership) has to be the same at sender and receiver ("with a CRC check") if not, e.g., different membership, message not even read

Fault Tolerance:

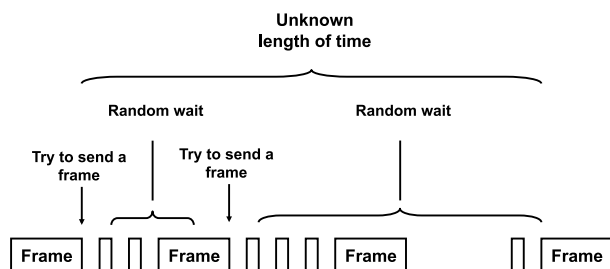
- **1 bus, 1 copy of msg:**
one message error $\rightarrow$ lost
- **1 bus, 2 copy of msg:**
one message error $\rightarrow$ no loss
- **+ 2nd bus:**
Bus errors also tolerated (cut or electrical problem)
- **+ 2nd node: (active redundancy)**
tolerates also node failures, silent fail nodes (if there is an error in the node the node shuts down)
- **+ 3th node: (shadow mode)**
the shadow node normally passive and ist just listening. If another node fails the shadow node get active.

Comparison between CAN and TTP

	TTP	CAN
Temporal Predictability:	Offline - easy and deterministic	response time analysis
Protocol Latency Jitter:	very low because the messages have the same points in time	good jitter far high priority messages but for low priority messages high
Synchronization:	necessary	not necessary
Fault Tolerance:	simple tolerance, reveiver based detection	possible with PAR, sender based detection
Flexibility:	none	possible

6.2 General Purpose Networks

Ethernet in Real-Time:



Appealing to industry:
Same network type on all levels

Ethernet in Non-Real-Time applications:

- De facto standard
- Up to gigabit speed
- Low cost
- Easy to integrate and Stable

Ethernet and Real-Time:

Problems on MAC Layer:

Binary Exponential Backoff (BEB):

When a collision occurs the sender wait for a random amount of time. If the next try fails the time will be increased exponentially. → There is no way to know the delay of a packet (WC: packet is cancelled)

6.2.1 Replace BEB:

Deterministic Algorithm:

Make changes on the MAC layer and replace the BEB by an deterministic algorithm (e.g. assign each node a static wait time (priority) so it is known which node sends next in a case of a collision)

Bad:

The MAC layer is in hardware and the results are expensive non standard NW cards.

6.2.2 Switched Ethernet:

Ethernet Switches:

- Provide a private collision domain for each port
- Each port has full duplex capability
- IEEE 802.1p with 8 different message priorities

Good:

Simple scalable solution (standard network cards and protocols...)

Bad:

Ethernet switches have a limited queue memory. High traffic rate can cause overflow and so messages will be dropped. Also there are not enough priority levels for a efficient RT-Scheduling.

6.2.3 Traffic Smoothing

Idea:

The Traffic is separated into *Real-Time* and *Non-Real-Time* traffic. The Traffic Smoother is based between transport layer and MAC layer and intercepts non real-time packets.

Non-Real-Time Messages:

Bursty traffic in irregular intervals

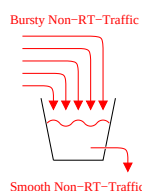
Real-Time Messages:

Are generated periodically every few hundred ms

Goal:

A low delay for RT messages but also a high throughput for non RT messages.

Leaky Bucket:



Credit Bucket Depth (CBD):

maximum nr of credits in bucket

Refresh Period (RP):

how often the bucket is replenished

By changing CBD and RP we can control the *burstiness* of the traffic

