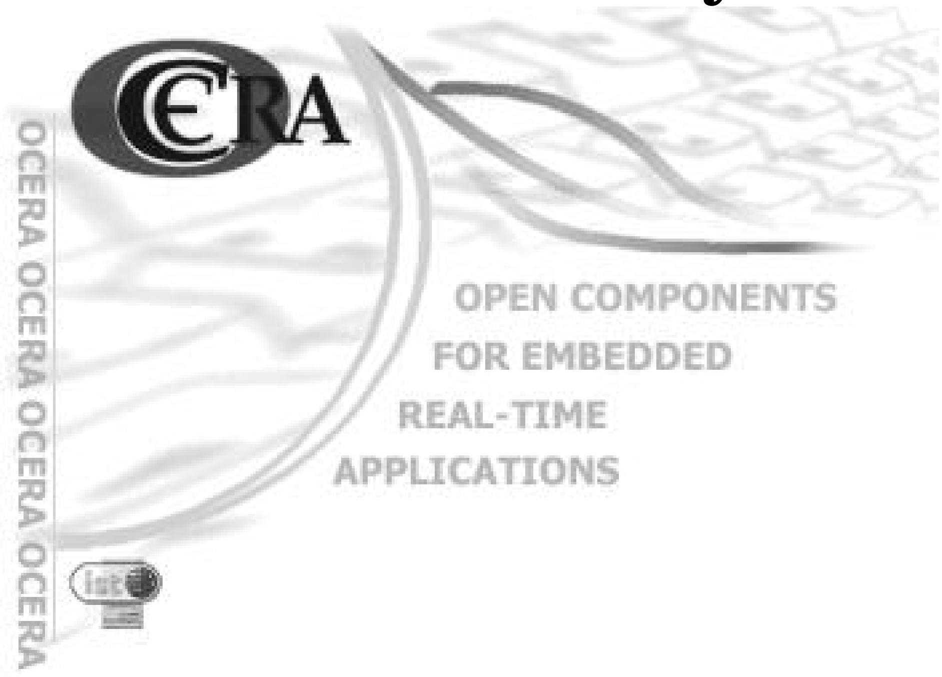
WP3 - Market Analysis



Deliverable D3.3 - New Approaches from Academia

WP3 - Market Analysis: Deliverable D3.3 - New Approaches from Academia

by Luca Abeni, Patricia Balbastre, Adrian Matellanes, Agnes Lanusse, Ondrej Dolejs, and Zdenek Hanzalek

Published September 2002

Table of Contents

List of Tables

List of Figures

Document presentation

Table 1. Project Co-ordinator

Organisation:	UPVLC
Responsible person:	Alfons Crespo
Address:	Camino Vera, 14. CP: 46022, Valencia, Spain
Phone:	+34 9877576
Fax:	+34 9877579
E-mail:	alfons@disca.upv.es

Table 2. Participant List

Role	Id.	Name	Acronym	Country
CO	1	Universidad Politécnica de Valencia	UPVLC	E
CR	2	Scuola Superiore S. Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	CEA	CEA	FR
CR	5	UNICONTROLS	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	VISUAL TOOLS S.A.	VT	E

Table 3. Document Version

Release	Date	Reason of change
1.0	November,	First Release

Chapter 1. Introduction

In this document we will present the different tendencies of current Real-Time systems research. The purpose of this analysis is to consider the maturity and feasibility of those different tendencies and evalute if they are appropriate for their inclusion in the OCERA project.

Of course, we will not cover all areas of current Real-Time research, but we aim to present the most important results that are relevant for the areas covered by the OCERA project.

This document is structured as follows, in Chapter 2, *A Survey of Real-Time Scheduling* we will present current trends in task scheduling, in Chapter 3, *Resource Reservation* we analyzed resource reservation, in Chapter 4, *Fault-tolerance analysis* we will consider the future of fault-tolerance in embedded and real-time systems and finally we will consider communications.

Chapter 2. A Survey of Real-Time Scheduling

2.1. Introduction

Scheduling theory allows to know whether a real-time system is feasible (schedulable) or not, that is, if tasks will meet its timing requirements. A task is said to be feasible if, in the worst case, its output is delivered before the deadline. Depending on the consequences of a missed deadline, real-time systems can be classified in hard and soft. In a hard real-time system a missed deadline can cause catastrophic damage, while in a soft real-time system deadlines can be missed without compromising the integrity of the system. Note than in real-time systems scheduling, specially in hard real-time, the main objective is that all tasks meet its deadlines, so an important term is the worst case, not the average case.

There are two main steps related to real-time scheduling problem:

- Off-line feasibility analysis. Before the execution of the system, designers must know if timing requirements will be met with a feasibility test of the system. If not, task parameters have to be re-evaluated.
- On-line scheduling algorithm. Once the task set is feasible, the system can execute, and the scheduling algorithm decides what task to execute.

2.2. Scheduling algorithms

A *scheduler* provides a policy for ordering the execution of tasks on the processor, according to some criteria. Schedulers produce a schedule for a given set of processes. There are several classifications of schedulers. Here are the most important:

- Optimal or non-optimal. An optimal scheduler can schedule a task set if the task set is schedulable by some scheduler.
- Preemptive or non-preemptive. A preemptive schedul er can decide to suspend a task (before finishing its execution) and restart it later, generally, because a higher priority task becomes ready. Non-preemptive schedulers do not suspend tasks in this way. Once a task has started, it can not be suspended involuntarily.
- Static or dynamic. Static schedulers calculate the execution order of tasks before run-time. It requires knowledge of task characteristics but produces little run-time overhead. However, it can not deal with aperiodic or non-predicted events. Some references about this kind of schedulers can be found in [Locke92]. Dynamic schedulers, on the contrary, make decisions during the run-time of the system. This allows to design a more flexible system, but it means some overhead.

  Priority-driven schedulers are dynamic schedulers which decide what task to execute depending on the importance of the task, called *priority*. This kind of schedulers can be also fixed or dynamic, depending on whether the task priority can vary during run-time. Because Priority-driven schedulers have been widely used in real-time systems, they will be described in more detail.

Figure 2-1, *Schedulers classification* shows a classification for real-time schedulers.

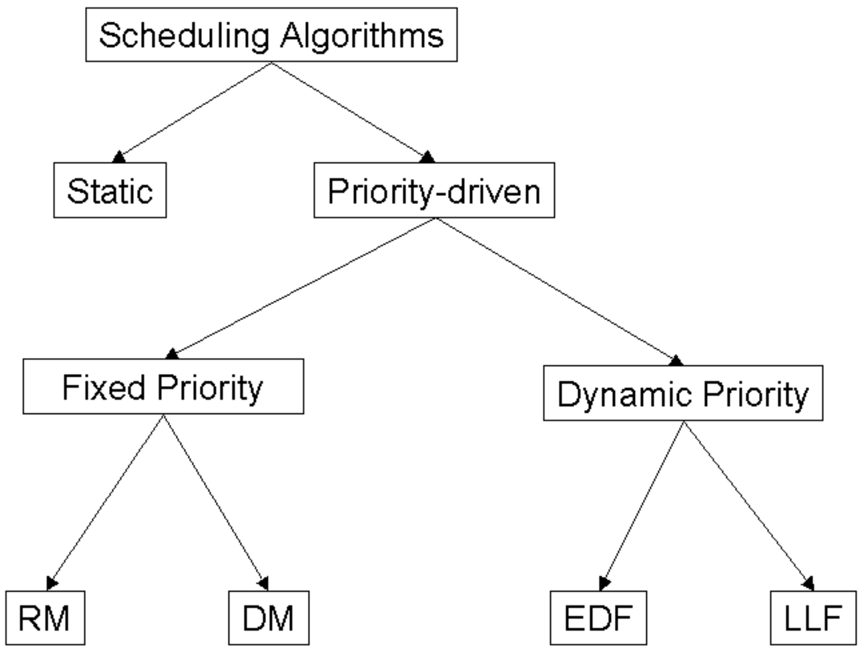


Figure 2-1. Schedulers classification

According to [Sprunt89], the quality of a scheduling algorithm is measured wether it meets the objectives below:

- It guarantees hard deadlines.
- It achieves a high degree of processor utilization.
- It provides fast average response times for soft tasks.
- It ensures scheduling stability for transient overloads.

2.2.1. Priority-driven algorithms

2.2.1.1. Fixed-priority scheduling

The most important scheduling algorithms in this category are Rate Monotonic (RM) [Liu73] and Deadline Monotonic (DM) [Leung82]. The former assigns the higher priority to the task with the shortest period, assumming that periods are equal to deadlines. The latter assigns the highest priority to the task with the shortest deadline. Both algorithms are optimal. Fixed-priority scheduling has been widely studied and the most important real-time operating systems have a fixed-priority scheduler..

2.2.1.1.1. Schedulability analysis

The first test was provided by Lui and Layland [Liu73]. It is based on the processor utilization of the task set. The total utilization of the set is the sum of the utilizations of all tasks in the set, which is obtained as the quotient of execution time by the period. This utilization is compared with the utilization bound (that depends on the number of tasks). Thus, if the utilization of the set is less or equal to the utilization bound, the task set is schedulable. This schedulability constraint is a sufficient, but not a necessary condition. That is, there are task sets that can be scheduled using a rate monotonic priority algorithm, but which break the utilization bound.

A sufficient and necessary condition was developed by Lehoczky [Lehoczky89] and Audsley [Audsley93] . This test is based on the worst case response time of every task. If, in the worst case, a task finish its execution before its deadline, the task will be schedulable. The worst case response time of a task occurs in the first activation. Moreover, this test is valid for any priority assignation, and it informs not only whether the set is feasible or not, but the task or tasks that miss its deadline.

2.2.1.2. Dynamic-priority scheduling

Within this cathegory, Earliest Deadline First (EDF) [Liu73] and Least Laxity First (LLF) [Mok78] are the most important. Both are optimal, t if any algorithm can find a schedule where all tasks meet their deadline then EDF can meet the deadlines.

In EDF, the highest priority task is the task with the nearest absolute deadline. The absolute deadline is the point in time in which it arrives the deadline of the current activation of the task.

LLF assigns priorities depending on the *laxity*, being the task with the lower laxity, the highest priority task. The term *laxity* refers to the interval between the current time and the deadline, minus the execution time that remains to execute.

Dynamic-priority algorithms have interesting properties when compared to fixed-priority. They achieve high processor utilization, and they can adapt to dynamic environments, where task parameters are unknown. On the contrary, real-time systems community is reluctant to use dynamic-priority algorithms mainly because of the unstability in case of overloads. It is also not possible to known what task miss its deadline if the system is not feasible.

2.2.1.2.1. Schedulability analysis

In [Liu73] it is proved that EDF can guarantee schedulability of tasks when the processor utilization is less than 100%. In this case, deadlines have to be equal than periods, but Dertouzos also proved that EDF is optimal when deadlines are less than periods.

Other schedulability test was presented in [Leung80]. The idea is to check if at any instant `t` the amount of computation time that has been requested by all activations (with deadline less than or equal to `t`) is less than or equal to `t`. This condition must be checked throughout the *`hyperperiod P`*, that is, the point in which the execution order is repeated. It is calculated as the LCM of task periods.

An improvement of this test is presented by Baruah et al in [Baruah90] and later by Ripoll et al in [Ripoll96]. These test reduce the interval in which the condition is evaluated.

Other feasibility test is based on the *Initial Critical Interval* [Ripoll96], that is the first interval in which the processor is not idle. If no dealines are missed in this interval, then the task set is feasible.

2.2.1.3. Resource sharing

In a priority scheduled system, when several tasks can access shared resources, one of the problems is the *priority inversion*. This occurs when a higher priority task is blocked, not only by the task that has the requested resource, but by other medium priority tasks. There are several protocols that try to reduce the priority inversion problem.

The first of these protocols is the *Priority Inheritance Protocol* (PIP) [Sha90]. When a task wants a resource that is locked, the task that owns (has locked) the resource inherits the priority (RM scheduling policy) of the task that wants it. This protocol gives an upper bound of the number of blocks. However, this bound is high and can lead to pessimistic worst case calculation. It also does not avoid deadlock or chained blocking. This protocol was included in the realtime POSIX standard.

The *Priority Ceiling Protocol* (PCP) [Sha90] assigns a priority to each resource, this priority is called the priority ceiling. When a task is request a resource, the requesting task inherits the ceiling of the resource. This policy avoids both deadlocks and chained blocking, and it ensures that a task can be blocked only once. This protocol can only be used with fixed priority scheduling policies.

The dynamic version of PCP is the *Dynamic Priority Ceiling Protocol* (DPCP) [Chen90]. This algorithm was developed to be used with dynamic priority scheduler like the EDF. With this protocol, the priority ceiling of the resources change dynamically, depending on the priority of the active tasks that can access a resource. The main drawback of DPCP is the high cost of re-evaluating ceilings every time a task gets active and suspended.

The *Stack Resource Protocol* (SRP) [Baker91] reduces the implementation cost of DPCP, and it has the same advantages. It can be used both with fixed and dynamic priority schedulers. The main idea of the SRP is to consider that resource ceilings are static, that is, its value do not change during run-time. However, tasks priorities can change. A task is allowed to start its execution only when all the resources that will require can be granted. The blocking factor and the schedulability analysis is the same than that of the PCP algorithm.

Another interesting protocol is the Ceiling Semaphore Protocol (CSP, also called *Inmediate Priority Ceiling Protocol*, IPCP) [Klein90]. When a task successfully locks a resource, it inherits immediately the ceiling of the resource, in contrast to the ordinary PCP which only raises process priorities when a process actually blocks a higher priority process. CSP can only be used with fixed-priority schedulers. This protocol included in the realtime extensions of the POSIX standard as *Priority Protect Protocol* and in Ada95 as *Ceiling Locking*.

2.2.1.4. Aperiodic scheduling

Since not all the tasks of a realtime system are periodic, nor have strong deadlines to meet, the system scheduler has to provide some non-realtime scheduling facilities for these type of tasks. Although aperiodic tasks do not have deadline to meet, they benefit from being server as soon as possible. The common approach is to add a special task, called aperiodic server, to the system that is the responsible to execute the aperiodic requests. It may happen that when the server is ready there are no process to handle. On the contrary, if a lot of events arrive at the same time, the server could not deal with these great number of arrivals. To overcome these problems three main algorithms are proposed, all of them based on the bandwidth preservation.

Priority exchange [Sprunt88]

> When no aperiodic requests exists, periodic tasks can execute changing its priority with the priority of the server, that is a high priorty task. Although the server's priority decreases, the reserved time for aperiodic tasks is maintained. The main drawback is the unpredictability when deadlines are missed.

Deferrable server [Lehoczky87]

> When the server is active but with no requests, it does not execute but it defers its assigned computation time at its initial priority. When an aperiodic event arrives, the server can execute. The server capacity is replenished at the start of its period. It provides its computation time at a fixed level, but the priority exchange algorithm allows for more aperiodic tasks to be serviced.

Sporadic server [Sprunt89]

> This server combines the advantages of the previous algorithms, by varying the points at which the computation time of the server is replenished, rather than at the start of each server period. Therefore, the sporadic server increases the number of aperiodic requests that can be serviced. The Sporadic Server was implemented in RTLinux by [Shi01] and it is defined in the POSIX 1003.1 standard.

The dynamic versions of the previous three algorithms were developed by Ghazalie and Baker [Ghazalie95] and Spuri and Buttazzo [Spuri96].

Other methods consist of using the available slack of the system to execute aperiodic tasks. This way, in every aperiodic arrival it is calculated the available slack (without jeopardizing periodic tasks schedulability). In the static method [Ramos93], a table with the instants in which aperiodic events can execute is obtained off-line. The main disadvantage is the size of the table. In the dynamic version [Davis93], slack instants are obtained on-line, so this may cause a considerable overload. The slack stealing methods for dynamic-priority algorithms were proposed by Chetto and Chetto [Chetto89] with *Earliest Deadline Last* algorithm (EDL), and Ripoll et al [Ripoll96].

Chapter 3. Resource Reservation

3.1. Reservation Guarantees

The problem of integrating flexible Quality of Service (QoS) guarantees in real-time systems has been widely studied in the last years, resulting in some interesting proposals. Probably, the most important theoretical result that emerged in this work is the idea that in order to provide a predictable QoS to different applications running on the same system, the OS kernel must provide *temporal isolation* between different applications or tasks. Temporal isolation (also known as temporal protection), requires that the temporal behaviour of a task is not influenced by the temporal behaviour of other tasks in the system.

In other words, if a task requires "too many" resources, it is slowed down so that it does not jeopardize the other tasks' guarantees. This property is very important, since it permits to provide different kind of guarantees to different tasks: for example, it is possible to perform a hard guarantee on a critical task, while other tasks are provided a probabilistic guarantee, or no guarantee at all.

A way to provide temporal isolation is to guarantee that every task will execute only for a reserved amount of time in every time interval. If exec(Ti, t1, t2) is the amount of time executed by task Ti in the interval (t1, t2), temporal protection can be ensured by enforcing that exec(Ti, t1, t2) / (t2 - t1) = Fi for every time interval (t1, t2). However, this requirement (that lead to the *Generalized Processor Sharing* - GPS - model used by the proportional share schedulers) can over-constrain the system: a more realistic requirement would be to enforce that the ratio exec(Ti, t1, t2) / (t2 - t1) is constant only *over well specified intervals*, for example between deadlines in a real-time task. This is the essence of the reservation guarantee. More formally, *a reservation (Q, P, D) guarantees that an amount Q of a resource will be available to the reserved task every period P, within a relative deadline D from the beginning of the period.*

If P = D, the reservation simplifies to a (Q, P) model, and the guarantee becomes: *A task Ti attached to a reservation (Q, P) is guaranteed to execute for Q execution time units every P* .

Some authors tend to distinguish between *hard reservation guarantees* and *soft reservation guarantees*: following this definition, a soft reservation guarantee ensures that the attached task will execute for at least Q time units every P, whereas a hard reservation guarantees that it will execute for exactly Q time units every P.

3.2. Reservation Based Scheduling

Based on classical real-time scheduling (EDF or RM priority assignment), it is possible to implement a reservation guarantee by simply enabling a task to execute as a real-time task (scheduled, for example, by EDF or RM) for the reserved time Q, and then blocking it (or scheduling it in background as a non real-time task) until the next reservation period. In this way, a task is *reshaped* so that it behaves like a periodic real-time task with parameters (Q, P) and can be properly scheduled by a classical real-time scheduler. A similar technique is used in computer networks by the traffic shapers, such as the leaky bucket or the token bucket.

More formally,

- a reservation is characterized by two or more parameters (Q, P), or (Q, P, D), where Q is the reserved time per period, P is the reservation period, and D is the reservation relative deadline
- a *budget*, or *capacity* is also associated to each reservation

- at the beginning of each reservation period (every P time units), the budget is recharged to Q
- if present, the parameter D is used to compute the priority with which the reserved task is scheduled
- a real-time scheduling algorithm is used to select a task *with budget greather than 0* for execution. When the reserved task executes, the budget is decreased accordingly
- when the budget arrives to 0, the reservation is said to be *depleted*, and an appropriate action should be taken (see below).

As previously said, when a reservation is depleted the reserved task can be blocked, or it can be “downgraded” to be a non real-time task. By blocking the task, it is possible to implement a hard reservation guarantee. If, on the other hand, a depleted task is downgraded to non real-time, then a soft reservation guarantee can be implemented.

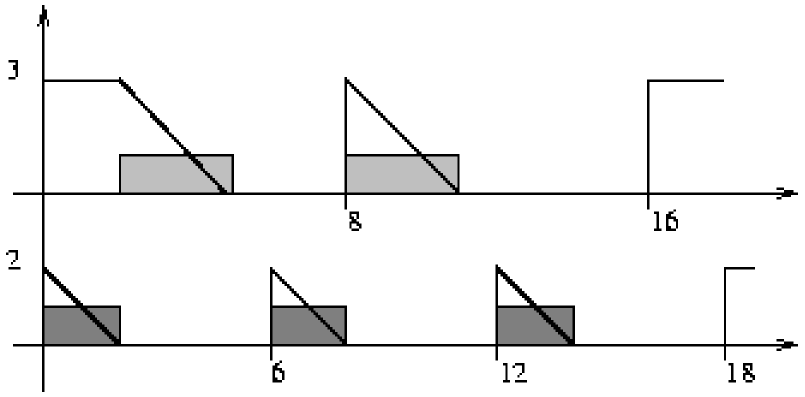


Figure 3-1. Scheduling Example

The previous figure illustrates an example of reservation-based scheduler based on RM. Two tasks are attached to two reservations (2, 6, 6) and (3, 8, 8), and for the sake of simplicity both the two tasks are assumed to be always backlogged. According to RM, the first task is scheduled first; since after two time units its budget arrives to 0, the task the at time 2 the first reservation is depleted, and the task is blocked (hard reservation). Hence, the second task can executed until time 5, when its reservation is depleted. At time 6, the budget of the first task is replenished, and the first task can execute until time 8, and so on...

Note that the reservation parameters (Q, P) are different from the task parameters (for example, from the (C, P) parameters of a periodic task), and this separation can be useful to control the tasks’ QoS.

3.3. State of the Art

Resource Reservations have been probably proposed for the first time by Mercer and Tokuda[Mer93-2] in Real-Time Mach to implement temporal protection in the context of a real-time multimedia system [Mer94, Mer93]. A more rigorous formalization of the reservation approach has been elaborated by Rajkumar and friends resulting in the Resource Kernel (RK) approach [Raj98].

Because of their goal of providing temporal protection to integrate real-time and non real-time activities, resource reservations have been initially implemented in “hybrid” systems that offer support to both time sensitive and general purpose applications, such as Real-Time Mach[Tok90], but also Rialto[Jon95, Jon97], Nemesis with its Atropos scheduler[Les96]. Moreover, the RK approach is fairly portable and has been successfully applied to Linux[Oik98, Oik99, Raj00]. In addition, reservation scheduling has been proven to be effective also in serving control tasks[Pal00, Pal], hence we believe that reservation techniques can be successfully applied to embedded systems (as already suggested by RED Linux[Wan99]).

Although CPU reservation were initially implemented using a fixed priority (Rate Monotonic or Deadline Monotonic) scheduler, they have successively been adapted to dynamic priorities, to obtain better CPU utilization and easy reclaiming of the unused CPU time. Examples are the already cited Atropos scheduler, the Constant Bandwidth Server (CBS) [Abe98], and some algorithms like GRUB[Lip00] that implement reclaiming of unused CPU time. The problem of coping with reserve underruns or overruns is conceptually similar to CPU reclaiming, and has been extensively studied at the Real-Time Systems Laboratory of University of Illinois at Urbana Champaign[Gar98].

One of the biggest problems with the reservation approach is to correctly reserve the proper amount of resources to achieve the specified QoS. If enough a-priori information are provided, it is possible to perform a probabilistic guarantee[Abe99, Abe01], otherwise some kind of on-line estimation must be used for implementing a feedback scheme and dynamically adapt the reservation parameters[Abe99-2]. To do this, it is necessary to do some performance monitoring, that can be either based on the knowledge of the application structure (for example, a pipeline) or of the task model (for example, the real-time task model[Abe99-2]).

Adaptive systems have been mainly developed for serving multimedia applications, but we believe that they can also be applied in embedded systems used in data-intensive contexts, where relatively high volumes of sensor data are flowing and must be processed and analysed in real time.

Chapter 4. Fault-tolerance analysis

4.1. Introduction

Over the last decade, the software industry has been confronted to a major revolution in uses and habits with, firstly the use of COTS instead of ad hoc proprietary solutions, then the use of open source operating systems and components.

In this context however, developers of real-time systems and, above all, safety-critical systems have long been reluctant to follow these tracks. Even the introduction of software was still questionable a few years ago in critical domains such as nuclear energy. In these domains indeed, the dynamic behavior of systems must be perfectly controlled at anytime, so their development is strictly controlled with specific design and coding rules. Even the use of multi-tasking programming was prohibited in safety-critical applications. Proprietary hardware and software solutions were developed within companies. Even less demanding applications used specific solutions.

Things have changed and more and more software is embedded in systems (avionics, automobile, transportation, industrial control, etc...). If traditional safety critical applications still use proprietary solutions, they appears to be to costly for more competitive industry. On an other hand, open source software can be tested and debugged by a very large community which can increase the confidence we can place into the system. There is thus an increasing demand for open source fault-tolerant components

In this chapter we remind the main issues related to dependability of real-time systems. We then focus on fault-tolerance and existing associated mechanisms. Finally we analyze basic requirements necessary in order to implement fault-tolerance in a system.

4.2. Dependability: a few definitions

Quite early high development costs of safety critical applications have been a major concern and industry has been eager to find more generic solutions. While international working groups were created such as IFIP WG10.4 named Dependable Computing and Fault Tolerance, the European Community encouraged research activities in the domain of formal methods, real-time systems development, and dependability issues.

Large projects such as SEDOS for formal methods and verification and validation issues, REAKT for the development of Knowledge-based real-time systems, Delta-4, PDCS, PDCS-2 for distributed fault-tolerant systems were initiated and brought to the community innovative solutions. If few of these results are available in the open source world, generic patterns have been clearly analyzed and described, they are today well known solutions that could be implemented as open source components. However, as we will see it, dependability is more a design issue than a component issue. It is thus important to have a deep understanding of what dependability means and clearly identify the various aspects involved.

In this section we thus introduce a few basic definitions extracted from [Laprie92].

4.2.1. dependability

J-C Laprie defines dependability as : "that property of a computer system such that reliance can justifiably be placed on the service it delivers. The service delivered by a system is its behavior as it is perceived by its user(s); a user is another system (physical, human) which interacts with the former"

He groups notions related to dependability into three classes : the attributes of dependability, the impairments to dependability and the means for dependability.

4.2.2. Attributes of dependability

Several properties may be expected from a dependable system, this leads to the notion of "attributes" of dependability. Depending on applications some of them take more or less importance. We will consider in the following five attributes :

- reliability
- availability
- safety
- security
- maintainability

defined as follows.

availability

The property readiness of usage is insured

reliability

The property continuity of service delivery is insured

safety

The property non-occurrence of catastrophic consequences is insured

security

Security regroups attributes related to integrity, confidentiality and availability and their associated properties not detailed here.

maintainability

The property aptitude to undergo repairs + evolutions is insured

4.2.3. Impairments to dependability

Impairments to dependability are faults, errors, failures. They are undesired but not in principle unexpected.

failures

A system failure occurs when the delivered service deviates from fulfilling the system function(what the system is intended for).

errors

Part of the system state which is liable to lead to subsequent failure: an error affecting the service is an indication that a failure occurs or has occurred.

faults

The adjudged cause or hypothesized cause of an error is a fault.

4.2.4. Means for dependability

Dependability property can be insured by an appropriate use of a combination of the following classes of methods

- fault prevention
- fault tolerance
- fault removal
- fault forecasting

fault prevention

How to prevent fault occurrence or introduction

fault tolerance

How to provide a service capable of fulfilling the system function in spite of faults

fault removal

How to reduce the presence (number, seriousness) of faults

fault removal

How to estimate the present number, the future incidence, and the consequences of faults. related both to value failures and timing failures.

4.3. Fault-tolerance issues

Fault-tolerance can be achieved by the combined use of *error processing* and *fault-treatment* [Anderson] [Avizienis].

4.3.1. Error processing

Error processing can be carried out by :

- error detection
- error diagnosis
- error recovery

error detection

error detection enables an erroneous state to be identified as such

error diagnosis

error diagnosis enables assessment of the damages caused by the detected error, or by errors propagated before detection.

error diagnosis

error diagnosis enables assessment of the damages caused by the detected error, or by errors propagated before detection.

error recovery

error recovery corrects or masks the effect of errors.

backward recovery

Backward recovery is used where the erroneous state transformation consists of bringing the system back to a state already occupied prior to error occurrence; this involves the establishment of recovery points, which are points in time during the execution of a process for which the then current state may subsequently need to be restored;

forward recovery

Forward recovery is used where the erroneous state transformation consists of finding a new state, from which the system can operate (frequently in degraded mode)

compensation

Compensation is used where the erroneous state contains enough redundancy to enable its transformation into error-free state.

The association into a component of its functional processing capability together with error detection mechanisms leads to the notion of a *self-checking* component, either in hardware or in software. One important benefits of the self-checking component approach is the ability to give a clear definition of error confinement areas.

When error compensation is performed in a system made up of self-checking components partitioned into classes executing the same tasks, then state transformation simply consists of switching within a class from a failed component to a non failed one.

On the other hand, compensation may be applied systematically, even in the absence of errors, so providing *fault masking* (e.g. majority vote). However, this can at the same time correspond to a redundancy decrease whose extent is not known. So, practical implementations of *masking* generally involve error detection, which may then be performed after the state transformation.

As opposed to fault masking, implementing error processing via error recovery after error detection has taken place, is generally referred to as error detection and recovery

4.3.2. Fault treatment

Fault treatment consists of several steps: fault-diagnosis, fault-passivation and when necessary reconfiguration;

4.3.2.1. Fault diagnosis

The first step of fault treatment is fault diagnosis, which consists of determining the causes of errors, in terms of both location and nature.

4.3.2.2. Fault passivation

Then comes the actions aimed at fulfilling the main purpose of fault-treatment: preventing the faults from being activated again, thus aimed at making them passive, i.e. fault passivation. This is carried out by preventing the components identified as being faulty from being invoked in further executions.

4.3.2.3. Reconfiguration

If the system is no longer capable of delivering the same service as before, then a reconfiguration may take place, which consists of modifying the system structure in order that the non-failed components enable the delivery of an acceptable service, although degraded; a reconfiguration may involve some tasks to be given up, or re-assigning tasks among non failed components.

If it is estimated that error processing could directly remove the fault, or if its likelihood of recurring is low enough, then fault passivation need not be undertaken. As long as fault passivation is not undertaken, the fault is regarded as a soft fault; undertaking it implies that the fault is considered as hard or solid.

4.3.3. Remarks

The preceding definitions apply to physical faults as well as to design faults. The classes of faults that can actually be tolerated depend on the fault hypothesis that is being considered in the design process, and thus relies on the independence of redundancies with respect to the process of fault creation and activation.

Example of tolerance of physical faults:

A widely used method of achieving fault-tolerance is to perform multiple computations through multiple channels. When tolerance of physical faults is foreseen, the channels may be identical, based on the assumption that hardware components fail independently. Such an approach is not suitable for providing tolerance to design faults where channels have to provide identical services through separate designs and implementations through design diversity.

An important aspect of the coordination of the activity of multiple components is that of preventing error propagation from affecting the operation of non failed components.

This aspect becomes particularly important when a given component needs to communicate some information to other components that is private to that component.

Typical examples of such single-source information are local sensor date, the value of a local clock, the local view of the status of other components, etc.. The consequence of this need to communicate single source information from one component to other components is that non-failed components must reach an agreement as to how the information they obtain should be employed in a mutually consistent way.

Specific attention has been devoted to this problem in the field of distributed systems (atomic broadcast, clock-synchronization, or membership protocols).

It is important to realize, however, that the inevitable presence of structural redundancy in any fault-tolerant system implies distribution at one level or another, and that the agreement problem therefore remains in existence.

Fault-tolerance is also a recursive concept. It is essential that the mechanisms aimed at implementing fault-tolerance be protected against the faults which can affect them. Examples are voter replications, self-checking checkers, stable memory recovery programs and data.

4.3.4. Real-time and fault-tolerance

Even if it has not been evocated explicitly in the preceding sections, time management and timeliness are tow major challenging issues for fault-tolerant systems.

Real-time systems are often used for control applications with hard deadlines imposed by the environment. Therefore the computing system has to meet real-time requirements. Timeliness is thus a major property to be verified by the system. Results or actions must occur in time, if not, it is considered as a faulty behavior.

The strict observance of these deadlines complicates the solutions of many problems that are well understood if no time constraints have to be considered.

Besides, as we have seen it a fault-tolerant system is inherently a distributed system. A first objective is thus to maintain temporal consistency between tasks. The implementation of recovery techniques based on redundancy impose synchronization. This supposes the existence of a consistent temporal framework.

While the timeliness requirement is a service attribute at the user/system interface, distribution and fault-tolerance are implementation characteristics.

So it seems natural to design a real-time system on a temporal basis. This offers two main advantages: a clear specification of temporal features of tasks, and a suitable framework for implementing fault-tolerance mechanisms. It is the Time Triggered approach promoted by [Kopetz] (see also [David & al98]).

A time triggered architecture observes the state of the environment periodically and initiates system activities at recurring predetermined points of the globally synchronized time. It is the kind of approach generally used for safety critical systems where determinism and predictability are required[1].

The other class of approaches is called event-triggered . It is the most commonly used for non safety-critical systems. However it is not well adapted to ensure synchronization of activities.

In an event-triggered architecture, a system activity is initiated as an immediate consequence of the occurrence of a significant event in the environment or the computer system. There is no explicit temporal model of execution, events are handled according to priorities. In such systems real-time constraints were generally analyzed off line with RMA techniques and priorities were then computed. More recently a large number of dynamic scheduling algorithms have been made available [Chetto90], there is thus a hope that in the near future programmers will be able to directly specify their time constraints at the task programming level.

4.4. Existing mechanisms study

We will start this analysis by the study of two famous academic realization of fault-tolerant architectures implemented within the European Delta-4 project (Definition and Design of an open Dependable Distributed system Architecture) and the MARS project that has been developed at the Vienna University of Technology within the PDCS project.

The first one Delta-4 illustrates most of basic features of fault-tolerance in a distributed system. It's most important characteristics is that it supports transparent redundancy features. The second one, the MARS project is a Time-Triggered approach that pilots all synchronization issues of the system, it gives a very good illustration of the constraints imposed by real-time requirements especially in the case of safety-critical systems.

4.4.1. Examples of fault-tolerant architecture

4.4.1.1. Delta-4 experience

The Delta-4 project started in March 1986 and terminated in January 1992. DELTA4 architecture [Powell] was an open architecture that could employ off-the-shelf hardware to provide fault-tolerance. Fault-tolerance was achieved by the replication of code and data on different computational nodes interconnected by a local area network. Off-the-shelf hardware is assumed to be either fail-silent or fail-uncontrolled. In order to face the risk of saturation of channel in case a fail-uncontrolled node connected to multiple channels would fail; the Delta-4 architecture follows a hybrid approach were each node is split into two subsystems:

- An off-the-shelf computation component called a host that may be fail-uncontrolled
- A communication component called network attachment controller (NAC) that is assumed to be fail-silent.

The NACs of each station are interconnected by a standard LAN, they are the only specialized hardware components in Delta-4. Other Delat4 functionalities are handled by system software on top of either the hosts local operating systems or the NACs real-time kernels.

The system software can be split into three main components:

- A host resident infrastructure for supporting distributed applications;
- A computation and communication administration system (running partly on the hosts and partly on the NACS)
- A multi-point communication protocol stack (executing on the NACs).

An Application Support Environment (Deltase) was developed in order to facilitate the use of heterogeneous languages. A runtime environment made transparent the underlying Operating System.

The administration system carries out three basic tasks:

- Configuration management, which provides support for planning and integration of redundancy and distribution;
- Performance management which includes system status monitoring by event counting and polling;
- Fault management, including automatic fault treatment and support for maintenance interventions.

The multipoint communication protocol stack provides two major features:

- The provision of multi-point associations for connection-oriented communication between groups of communication end-points;
- The ability to coordinate communication to and from replicated communication end-points.

The core of the Delta-4 group communication mechanisms is the atomic multicast protocol(a two phase accept protocol). It ensures unanimity (frames are either delivered to all addressed gates on non faulty nodes or to none) and the that frames are delivered to all addressed gates in a consistent order. Messages are never lost and they are delivered in bounded time. Remote crashes can be reliably detected by time-outs on response to protocol frames.

The heart of Delta-4 fault-tolerant mechanisms is the inter-replica protocol. It coordinates communication to and from endpoints that are replicated on different nodes such that replication is hidden from the sources and the destination of the messages. This involves the transparent delivery of messages to all endpoint replicas but also arbitration between send events across the set of endpoint replicas such that destination only receive a single message.

Fault-tolerance

Capsules, the runtime representations of Deltase objects are the units of replication for achieving fault-tolerance. They may be replicated independently so that fault-tolerance can be specified service by service.

Error processing

Three complementary techniques have been investigated :

- Active replication
- Passive replication
- Semi-active replication

Fault treatment

Policies have been defined to properly handle fault diagnosis, fault passivation and system reconfiguration. This latter entails the allocation and initialization of new replicas to replace failed ones and thereby restore the level of redundancy so that further faults can be tolerated.

This work had been followed by the GUARDS project that investigated Real-time dependable systems and ended in 1999 [Powell].

4.4.1.2. The MARS experience

An other well known implementation of fault-tolerant mechanisms concerns the MARS project that has been developed at the Vienna University of Technology. MARS system supported deterministic timing behavior.

The time-triggered approach has been applied to the MARS architecture a project developed at the Technical University of Vienna. MARS was a fault-tolerant distributed real-time system build of a number of autonomous, fail-silent processing nodes which were interconnected by a redundant real-time network. In MARS all active and passive components were replicated in order to prevent a single failure from crashing the system. Moreover processing activities are replicated at the node level.

Timeliness

Synchronization followed strict time trigger policy. Every relevant action of the system was scheduled before operation. The actions to schedule included:

- The point in time when a node is allowed to send a message (as a consequence MARS used a TDMA protocol for communication).
- The start times and deadlines for all processes
- The point in time when sensors values are read and actuator values are written
- The actions for recovery and reintegration of failed nodes (these actions must be included into the schedule in order to prevent a properly detected fault from affecting the correct timing behavior of the system.

In order to insure timeliness , the system designer had to carefully investigate the timing behavior of all parts of the system. Time bounds for all processing and communication activities had to be known.

- The maximum execution time of each process, considering the architecture and speed of the processor, pipelining, caching, and memory wait states
- The maximum time of communication
- The operating system overhead
- The overhead of hardware activities which influence the timing behavior of the node

Fault-tolerance

The fault-tolerance of MARS was based on a two layered mechanism. The bottom layer (node layer) was responsible for error detection and error confinement. The task of fail-silent nodes was two detect internal errors and to prevent propagation of the errors by stopping to produce data. This way, the top layer didn't had to care about erroneous data, but had to provide enough redundancy to tolerate crash failures of parts of the system. The major functions of the top layer were handling of redundant data and reconfiguration of the system in case of node failure.

A micro-kernel was developed and implemented on each node. Its functionality was reduced to a minimum, the execution of context switches, and maintenance of the global time base. Context switches were performed using tables created by the pre-runtime scheduler. Management of the global time-base was done by the clock interrupt handler.

4.4.2. Fault-tolerant basic mechanisms

Well known mechanisms are being used to implement fault-tolerance in distributed systems. However, very few existing operating systems handle transparently fault-tolerance. Most of time specialized company provide proprietary solutions based on specific hardware and associated middleware. We cite in this section some of the most common techniques used.

The implementation of fault-tolerant distributed applications largely depends on the computing environment available. The ideal case is when the underlying operating system provides fully transparent error processing protocols such as in Delta-4 and in MARS.

When the operating system doesn't provide such facilities, the application programmer is forced to integrate in the functional part of the application statements to initialize, invoke appropriate non-functional mechanisms for error processing. This can be done using library calls to pre-defined mechanisms embedded in a specific environment such as in Isis [[Birman]].

Other approaches, used by systems like Avalon/C++ [[Detlefs]] and Arjuna [[Arjuna]], consists of using properties of object-oriented languages, such as inheritance, to make objects recoverable. The object model seems appropriate for introducing fault-tolerance into applications, but there are significant problems with such an approach for implementing various replication techniques in distributed applications. Promising solutions rely on the use of reflection. system-based fault-tolerance.

4.4.2.1. System-based fault-tolerance

In this approach, the underlying runtime system may offer a set of transparent error processing protocols, for instance based on replication as in Delta-4 [[Powell93]]. Delta-4 provides several replication strategies : passive, semi-active and active replication.

Passive replication is a technique in which only one of the replicas (the primary replica) processes, the input messages and provides output messages (in the absence of faults), the other replicas (the standby replicas) do not process input messages and do not produce output messages; their internal states are however regularly updated by means of checkpoints from the primary replica.

Semi-active replication can be viewed as a hybrid of both active and passive replication; only one of the replicas (the leader replica) processes the input messages and provides output messages (in the absence of faults), the other replicas (the follower replicas) do not produce output messages; their internal state is updated either by direct processing of input messages or, where appropriate, by means of modifications or mini-checkpoints form the leader replica.

Active replication is a technique in which all replicas process all input messages concurrently so that their internal states are closely synchronized (in the absence of faults), outputs can be taken from any replica; several inter-replica protocols are available to synchronize replicas.

These techniques however required specific hardware and advanced specialized protocols to insure the expected properties from the system.

4.4.2.2. Libraries of fault tolerant mechanisms

This approach is based on the use of pre-defined library functions and basic primitives. A good example is Isis [[Birman]]. This generic software construct enables the computation to be organised using groups of processes (tasks) according to various objectives: parallel computation, partitioned passive replication, the updated states of the primary copy (coordinator) must be sent to the standby copies (cohorts). When a coordinator fails a new coordinator is elected and loaded with the current state of the computation A new member can be inserted in the group of replicas and its state initialized using a state transfer primitive. All this must be taken into account when programming the replicas. Different check-pointing strategies are left open to the application programmer.

In this approach, error processing and application programming are done at the same level using specific programming constructs. This means that specific function calls (state transfer, synchronization, sending results, voting) must be introduced into the application programs at appropriate points, for instance for sending updates (passive replication), token management (semi-active replication), or decision routines (active replication). The advantage of this approach is that application programmer can tailor and optimize his own fault-tolerant mechanisms. The main drawback is that functional and non functional programming are mixed.

4.4.2.3. Adaptive Fault Tolerance and graceful degradation

More recent work attempts to provide more flexibility to systems than traditional statically predefined systems such as MARS. Adaptive Fault Tolerance is one of them. Depending on the dynamic situation more or less levels of redundancy may be necessary over a system and the repartition of these resources between critical and less critical tasks may change. The idea is to relax constraints on less critical tasks in demanding situations. This leads to the use of dynamic real-time scheduling algorithms that must not only meet timing but also fault tolerance requirements. This can be very efficient when combined with a reflexive real-time OS were reflexive information encompasses importance, deadline, precedence constraints, fault-tolerance requirements. Such an approach has been experimented at U Mass at Amherst[Gonzalez & al.]. Several strategies of fault tolerance techniques considered as alternatives have been used Triple Modular Redundancy (TMR), Primary /Backup (PB), and Primary /Exception (PE). When a task arrive in the system, the scheduler uses information about the task has an alternative list, then the scheduler attempts to build a feasible schedule using the first fault tolerant alternative on the list. If no guarantee can be provided using the alternative, the next alternative is selected and the scheduler makes an other attempt. These experiments have been done with the Spring system. They take benefit of the deep experience in the laboratory in Scheduling and Real-time systems [Ramamritham 94].

Other techniques are used to provide graceful degradation, where alternatives do not concern alternative fault-tolerance mechanisms associated to tasks, but alternative tasks behaviors that can be triggered when faults are detected or when the system is overcharged. Such an approach has been tested by [Delacroix & al] , her approach uses an EDF like algorithm. Tasks are described by time constraints, importance and a set of possible modes. Four modes are defined for each task: nominal mode, deferred mode, revocation mode and temporal fault mode. A specific module above the scheduler detects surcharge situations and possibly stop tasks depending on their importance and decides for other tasks if they must be maintained in their normal mode or in a degraded one. The ROSE project at Carnegie Mellon has also the goal of implementing gracefully degraded modes [Nace & al.].

Those two classes of approaches have in common a declarative way of describing constraints attached to tasks. They both need a reflexive approach (knowledge on the state of the system and on importance of tasks as well as strategies for relaxing constraints) and above all they require advanced scheduling techniques.

4.4.2.4. Object oriented approaches and inheritance of fault-tolerance mechanisms

In this paper we mostly spoke about real-time systems developed as a set of classical tasks. But it is important to notice that object-oriented approaches to real-time have made important progress during the last decade and design tools are now supporting implementation of active objects (animated by one or more threads) . The object concept is very interesting for fault-tolerance since it represents a unit of encapsulation to which generic methods can be added to support fault-tolerance mechanisms as for example transaction mechanisms within active object models. It is also possible to design specific software components such as a captor that transparently handled data acquisition and provide high level service to the user application

Moreover the inheritance mechanisms of objects permit the implementation of such services in a very flexible way. European projects such as WOODDES or EAST are good examples of recent advances in object-oriented real-time programming. An other reason to have a look at it is that CORBA developments are object-oriented.

4.5. Conclusion

As a conclusion we can say that while fault-tolerance basic mechanisms are well known, there is no generic of the shelf component that can be used. Replication for example usually relies on specific hardware components associated to particular protocols.

While traditional safety-critical fault-tolerant systems were build upon a static analysis of temporal constraints. More recent approaches try to introduce more flexibility through the use of dynamic scheduling approaches taking into account importance criteria and multi-mode functioning.

This make obvious the deep imbrication of fault-tolerance and scheduling and communication issues, there thus be a necessity to provide not only seperate services but a global framework with well defined roles for each component.

Design issues constitutes also a very important step to build dependable systems. Adapted design environments can help specifying the systems characteristics and evaluating its feasibility or the adequate dimensioning for the system hardware.

We signaled also the growing importance of object oriented development in the real-time community. UML state charts offer a way of specifying active objects behaviors and are thus a good design support to introduce constraints over tasks (actually threads associated to objects.

Notes

1. Other approaches based on synchronous languages have been proposed for the development of safety critical systems (SIGNAL, LUSTRE, ESTEREL).

Chapter 5. Survey on real-time communications

5.1. ISO/OSI communication model

The communication model was standardised by ISO (International Organization for Standardization) as ISO/OSI communication model (IS 7498) in 1984. Problems linked with communications have been divided into standard seven layers. Each layer solves a given group of basic problems interfaced via services with other layers. Each layer is defined by protocol and service. The protocol specifies rules and conventions of dialog among hosts on given layer. The services cover specification of interface between neighbour layers in one host.

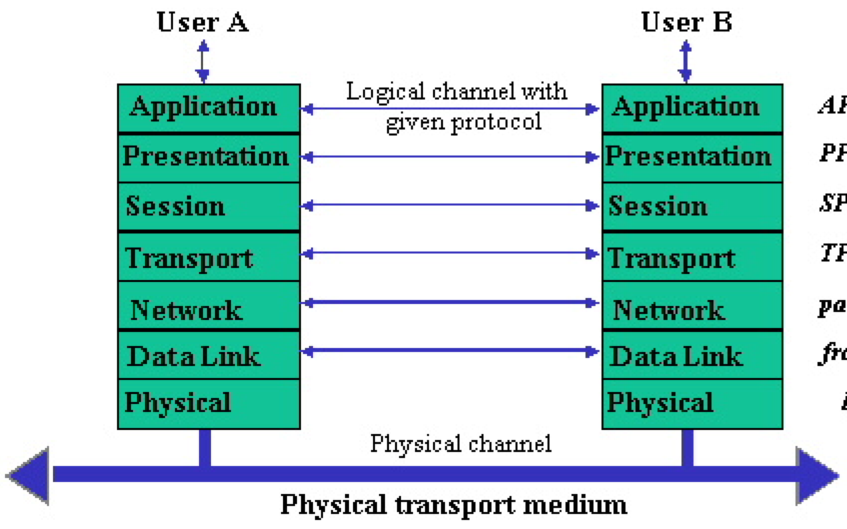


Figure 5-1. ISO/OSI communication reference model

5.1.1. Physical layer

Physical layer specifies electrical, mechanical and logical interface between devices. The communication media are:

- wire
- optical link
- wireless - different frequence bands

The voltage coding, modulation and connectors are specified by physical layer.

5.1.2. Data link layer

Data link layer controls media access (MAC-Media Access Control), data transfer, error detection and packet retransmission. The constructed data link packet is composed of header, data (payload) and trailer.

5.1.3. Network layer

Network layer defines roots in multiple networks, collects function to interconnect different characteristics of different subnetworks. The higher layer are independent of rooting.

5.1.4. Transport layer

Transport layer is between the three upper layers (oriented to data processing) and the three lower layers (oriented to communication). The long messages are divided into packets on the sender side and they are collected on the receiver side.

5.1.5. Session layer

Session layer ensures data exchange between applications e.g. synchronization. A session exists all the time, when the application allocates resources. When some data are prepared to be send, the transport layer is contacted and communication link is opened.

5.1.6. Presentation layer

Presentation layer is responsible for data representation (coding), compression-decompression and protection. Data representation can be different at different devices e.g. location of most significant bit in byte. Data protection means its encryptions, integrity etc.

5.1.7. Application layer

Application layer specifies data format between application program and network.

5.1.8. Reduction of ISO/OSI model for control applications

Most of the fieldbuses use only 1st,2nd and 7th layer for shorter time response. In many control applications there is not required for example to interconnect different subnetworks.

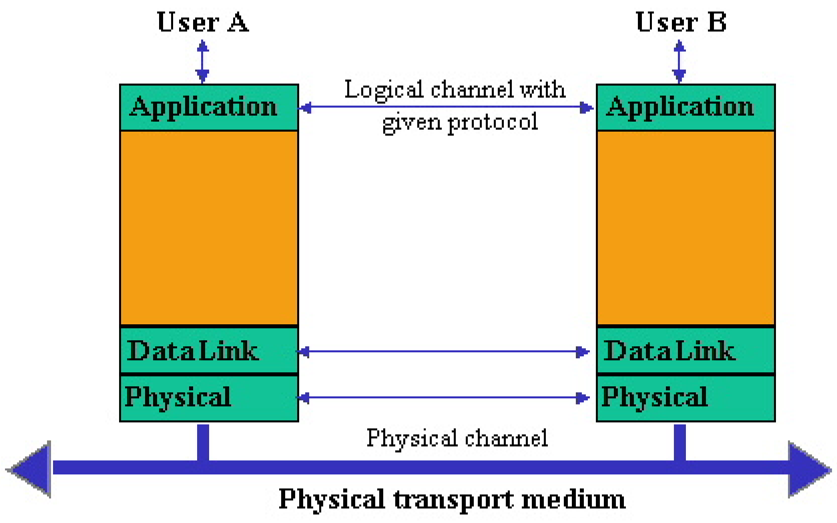


Figure 5-2. ISO/OSI model for fieldbus systems

5.2. Network Topology

The topology characterizes the connection between the nodes. The basic topology structures depicted in Figure 5-3, *Examples of the network topologies* are:

- peer-to-peer
- bus
- star
- ring
- free topology

The network topology influences important network parameters as:

- connectivity
- diameter
- reconfigurability
- reliability

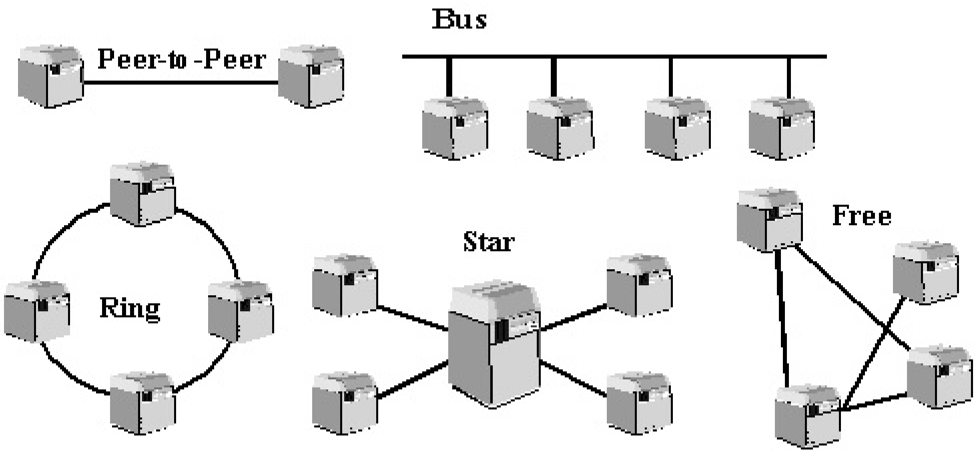


Figure 5-3. Examples of the network topologies

5.2.1. Peer-to-peer

Two devices are connected together and data are exchanged only between those two devices. It is the simplest communication topology sometimes called point-to-point topology.

5.2.2. Bus

All devices are connected to one line (multipoint connection). The extensibility is very simple and the cost of this topology is smaller than the cost of other ones. The crucial problem here is the access to the media, (description different access methods is given further). This topology is mainly used in fieldbus systems.

5.2.3. Star

All devices are connected to the central unit, the type of connection is peer-to-peer. The problems are extensibility, higher cost and the situation, when the central unit fails.

5.2.4. Ring

The devices are connected in the ring in point-to-point manner. The data are usually regenerated in the device and send to the next device. The flow control is simple, but the extensibility is difficult. Another problem is the fault of one device, which collapses whole network.

5.2.5. Free topology

Free connections among devices are possible, as a consequence the extensibility is very simple. The reliability can be improved by adding another devices in the network.

5.3. Communication delay

For real-Time applications the response time is a crucial problem. Therefore, it is needed to analyse whole system in order to know, where the delays arise. This chapter focuses on the communication delays (other delays are given by the operating system and the application tasks). The communication delay depends on topology of the network too. We focus on the delay model of typical topologies used in automation - the peer-to-peer topology and the bus topology.

5.3.1. Peer-to-peer

The complex communication process is shown in Figure 5-4, *Communication delay on peer-to-peer topology*. The transmission delay is calculated as time difference between the beginning of transmission of the first bit and the end of transmission of the last bit of the packet. The propagation delay depends on parameters such as the type of physical medium, the distance between the source and the destination, and baudrate. After the entire packet has completed reception at the destination, the number of bit errors that occurred during communication are allocated. This result is generally dependent on a bit-error probability which reflects the quality of the link, and the packet length. Depending on error allocation (how many errors occurred), the error correction procedure is executed.

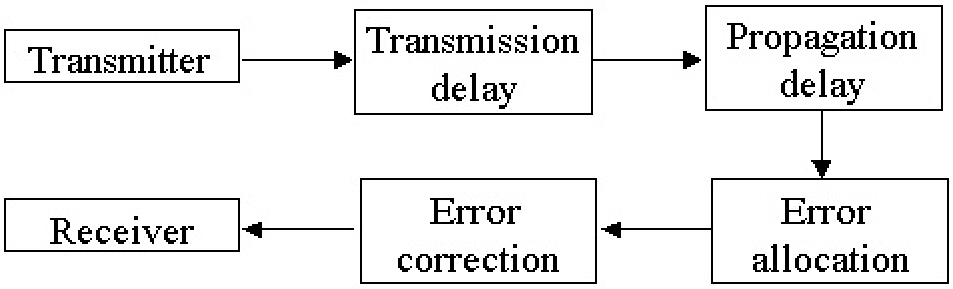


Figure 5-4. Communication delay on peer-to-peer topology

5.3.2. Bus

In the bus topology the collision delay is added, because the media is shared by all devices on the bus, and only one of them can transmit at a given time. This delay has the most importance to the total communication delay. Up to the MAC method it is resolved, which device wins the media for its transmission. If the media access is deterministic, the calculation of the collision delay is relatively easy. But the analysis tends to be infeasible when the media access is non- deterministic (e.g. CSMA/CD).

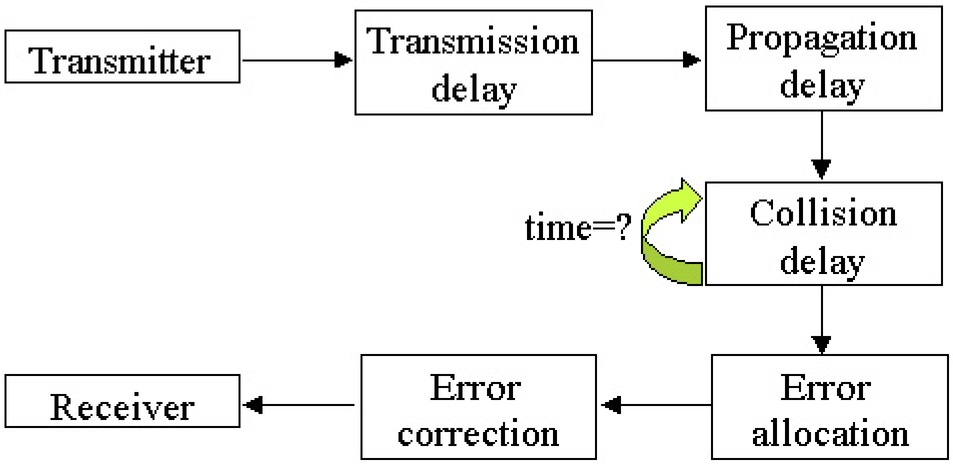


Figure 5-5. Communication delay on bus topology

5.4. Media Access Control (MAC)

The media access control on the bus topology is important as the communication media is shared by devices, and only one of them can transmit at a given time. Figure 5-6, *MAC methods* divides this problem into various cathegories. The first division is up to the synchronicity of the media access. The synchronous access means, that the time is divided into slots and each of devices have predefined slot, when it can transmit to the media. No time division is made by asynchronous access. This control method can be further divided into random and deterministic access methods. The random access method is typical for even-driven application, where the event activates the transmitter. Two main methods belong to the random access, the carrier sense with collision detection (e.g. CSMA/CD) and the carrier sense with collision resolution (e.g. CSMA/DCR - deterministic resolution by priority). The deterministic access method can be divided in the centralized and decentralized control methods. The typical centralized control method is Master-Slave communication, which is used in many fieldbus systems. The decentralized control method is represented by token passing, where one token circulates in a logical ring and when the device gets the token, then it makes its own transmission and passes the token to the next device in the ring.

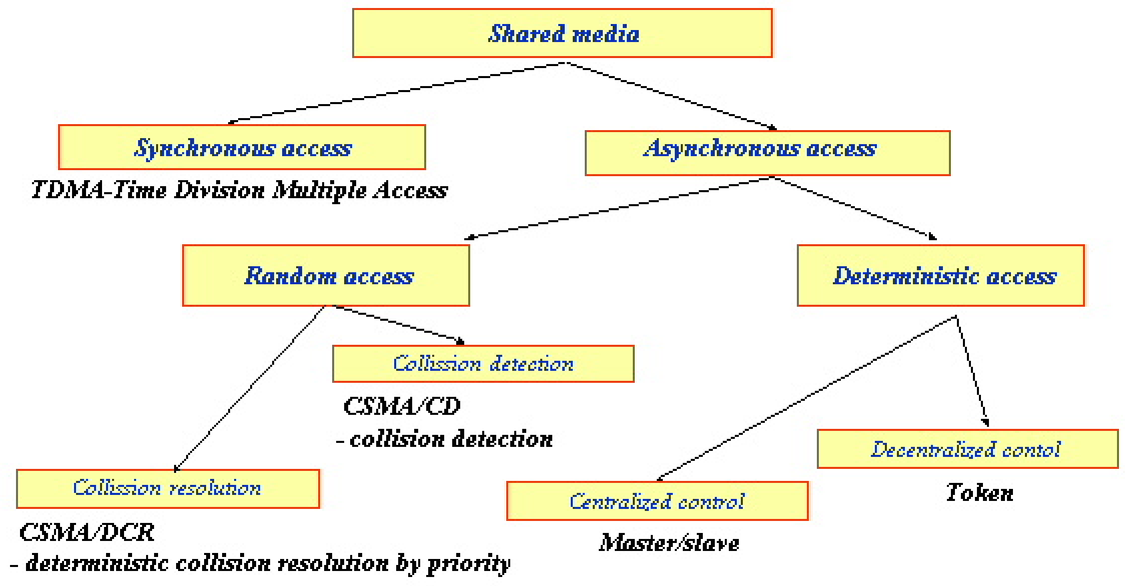


Figure 5-6. MAC methods

5.5. Buses in RT applications

The busses used in automation are discussed in this chapter. The main points of view are the topology and the media access control (MAC). This brief overview is concerned to the following busses: Profibus, CAN (CANopen), Fieldbus Foundation, LON Time-Triggered-Protocol (TTP), Train Communication Network (TCN) , Ethernet - Modbus TCP, Ethernet - RTI and Ethernet - RTNet.

5.5.1. Profibus (Process Field Bus)

PROFIBUS is a consistent, open, digital communication system with a wide range of applications, particularly in the fields of factory and process automation. PROFIBUS communication is anchored in the international standards IEC 61158 and IEC 61784. Profibus is a bus (line) from the topology point of view. The devices are divided into two groups: Masters and Slaves see Figure 5-7, *Mixed MAC in Profibus*. The Masters pass the token in a logical ring. The Master, which owns the token, can communicate with its Slaves. The Master requests step by step the Slaves (output data), which has been configured by this Master, and Slaves give response (input data) to the Master. It can be seen, that this MAC mechanism between Masters belongs to the decentralized control and between Master and Slaves to the centralized control on the other hand (according to Figure 5-7, *Mixed MAC in Profibus*).

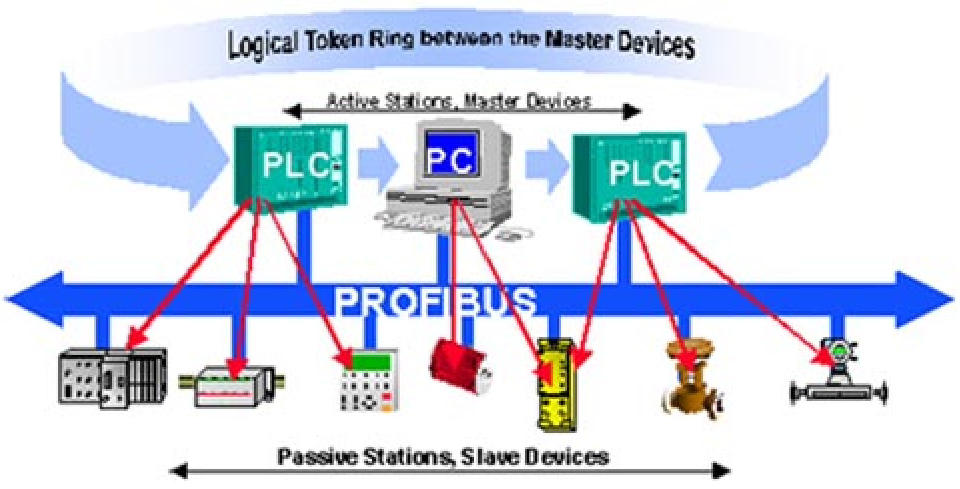

Logical Token Ring between the Master Devices
Active Stations, Master Devices
PLC
PC
PLC
PROFIBUS
Passive Stations, Slave Devices

Figure 5-7. Mixed MAC in Profibus

Profibus specifies layer 1,2 and 7 according to the ISO/OSI model described above. Some profiles are available for special application (process control, motion control, safety application etc.).

5.5.2. CAN (Controller Area Network)

The CAN protocol is an international standard defined in the ISO 11898. The devices are connected parallel to the wire, it is bus topology. CAN is based on the so- called broadcast communication mechanism. This broadcast communication is achieved by using a message oriented transmission protocol. Thus not defining stations and station addresses, it only defines message. The CAN protocol is producer/consumer oriented (see Figure 5-8, *Message filtering in CAN*). Station 2 produces message and filtering on the consumer site allowing accept (station 1 and 4) or discard message (station 3). These messages are identified by a message identifier. Such a message identifier has to be unique within the whole network and it defines not only the content but also the priority of the message.

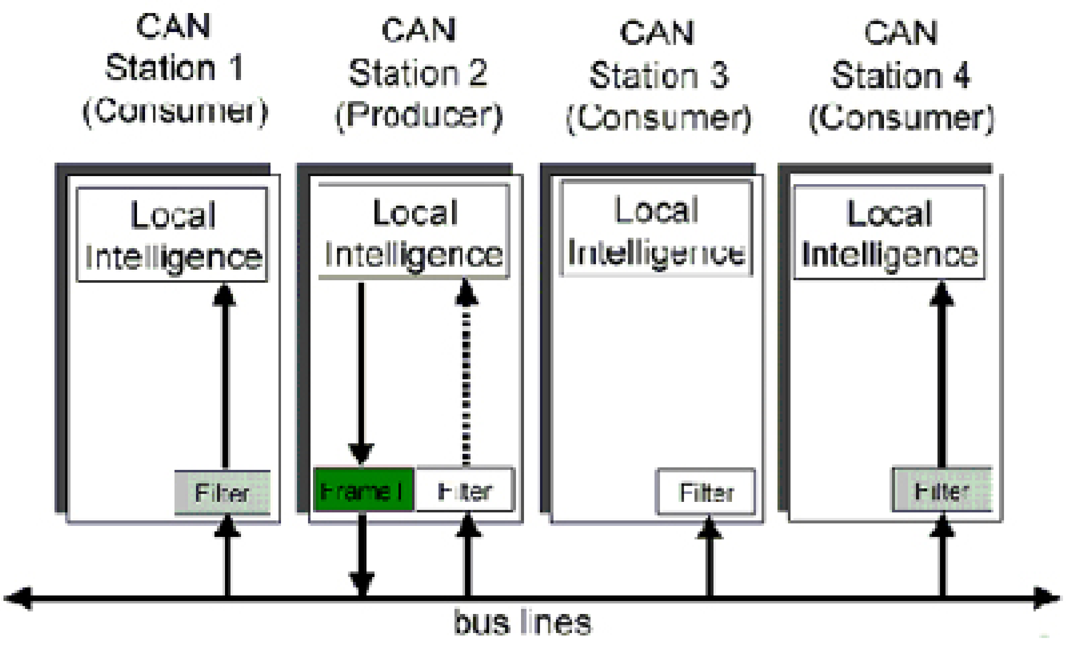


Figure 5-8. Message filtering in CAN

The MAC is made by priority corresponding to the message identifier. The identifier with the lowest binary number has the highest priority. The example of the bus arbitration is given in Figure 5-9, *Media access control in CAN*. The node 1 and 2 have higher identifier (smaller priority, i.e. the identifier starts with more logical ones) than node 3 (the identifier starts with more logical zeros), therefore the node 3 wins arbitration and can send data.

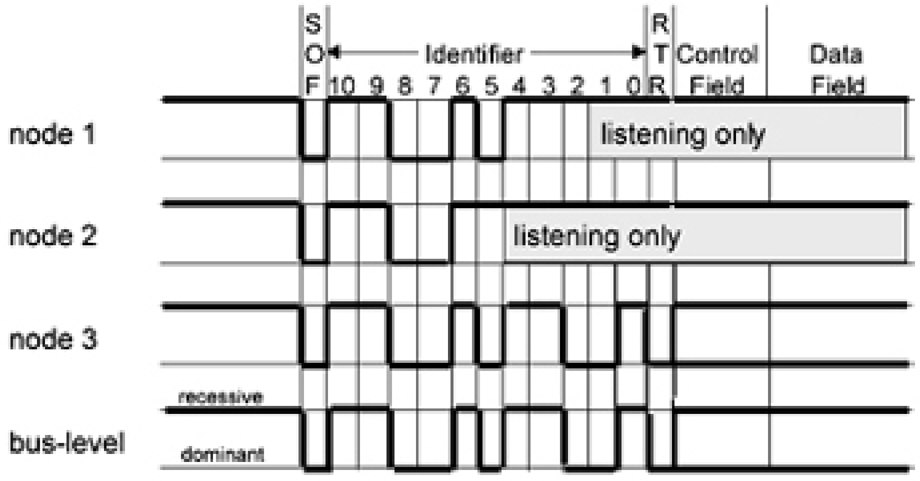


Figure 5-9. Media access control in CAN

Figure 5-10, *Wired AND in CAN MAC* is illustrative representation of the hardware implementation of CAN arbitration. If the transistor is open, the dominant state at the bus is set. In accordance with the "wired and" mechanism, by which the dominant state overwrites the recessive state.

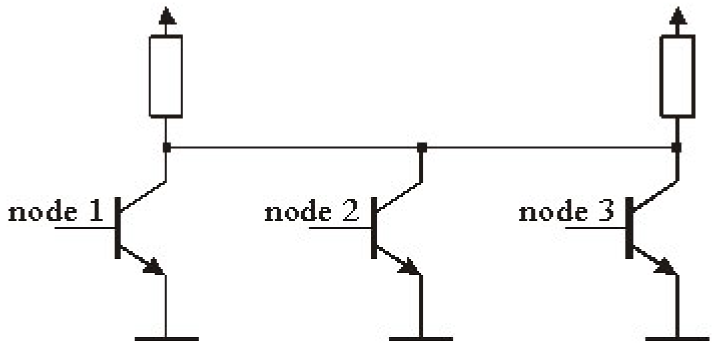


Figure 5-10. Wired AND in CAN MAC

CAN specifies the 1st and the 2nd layer according to the ISO/OSI model described above. The time analysis of CAN is given in [1]. In RT applications there are many systems, using CAN. Those fieldbus systems usually implement their own 7th layer. As the example of this fieldbus systems we can mention: CANopen, DeviceNet, SDS and TTCAN (Time-Triggered CAN).

5.5.3. TTCAN (Time Triggered CAN)

TTCAN is based on a time triggered and periodic communication which is timed by a master's reference message. The time is divided into windows (slots). The windows can be used for periodic messages (called an exclusive time window) and other windows for sporadic message (called an arbitrating time window). Prior to the communication it is decided off-line which message must be send at which exclusive window. Any message that is sent has the CAN data format and is a standard CAN message.

5.5.4. Fieldbus Foundation

Fieldbsus Foundation (FF) is based on IEC 1158-2, process oriented and intrinsically safe physical layer. The topology is bus, which transfer data as well as power. Only three types of devices are connected to the bus, the Basic Devices, Link Master and Bridges, which is used for interconnection of individual devices. The Link Master has a list of transition times for all data buffers in all devices that need to be cyclically transmitted. When it is time for device to send a buffer, the LAS issues a Compel Data (CD) message to the device. Upon receipt of the CD message, the device broadcasts or publishes the data in the buffer to all device on the bus. Any device that is configured to receive the data is called a subscriber.

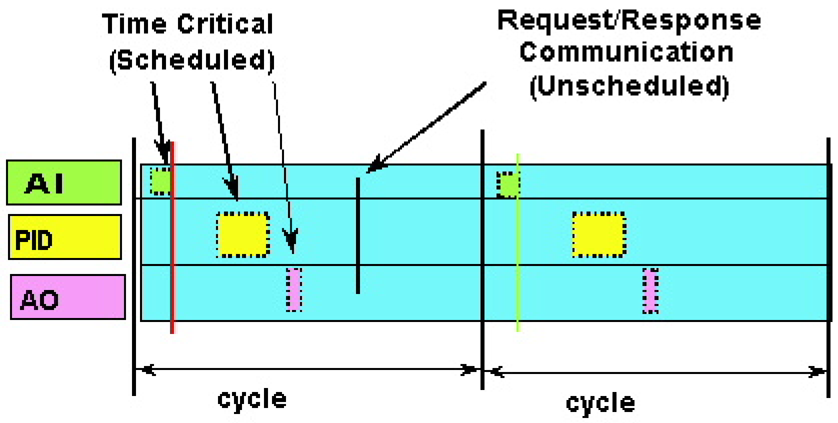


Figure 5-11. Communication scheduling

This model is called publisher/subscriber. All of the devices on the fieldbus are given a chance to send unscheduled messages between transmission of scheduled messages. In the other words the communication time is divided into two section. The first is for scheduled (deterministic) data - cyclic communication and the second is for unscheduled - acyclic data.

5.5.5. LON

LonWorks is one of the most important fieldbus standards in building automation. It was developed by Echelon® and in addition to many others, LonWorks uses wide variety of physical layers including power lines. Free topology is used in LON network. LonTalk, which is a communication protocol of LonWorks, uses non-deterministic media access method (MAC) called predictive p-persistent CSMA (Carrier Sense Multiple Access). Despite of non-deterministic MAC method LonWorks is used in soft real-time applications with moderate traffic. The advantage of the protocol is that it keeps the collision ratio independent of the channel utilisation and it uses technique for partial predication of the channel backlog. Like CSMA, Predictive p-persistent CSMA senses the medium before transmitting. A node attempting to transmit monitors the state of the channel (see Figure 5-12, *Communication scheduling*), and when it detects no transmission during β_1 period, it asserts the channel is idle. Then it generates the random delay Δ_T before transmission. If the channel is idle when the delay Δ_T expires, the node transmits; otherwise, Link layer receives an incoming packet and the algorithm repeats. Predictive p-persistent CSMA generates the delay Δ_T as an integer number of discrete time slots of width β_2. The delay Δ_T is generated from the randomising window (0..W_Base), which changes with respect to actual and predicated traffic on the channel. In other words W_Base is defined as product of BL (an integer estimate of the current channel backlog) and a basic window size. If there is no traffic on the channel or if the traffic is very low, then BL is equal to 1. With growing utilisation of the channel the BL grows and the randomising window (0..W_Base) enlarges. β_1 and β_2 are time constants given by Physical layer parameters and respect propagation delay defined by the media length, detection and turn-round delay within MAC sublayer. In Figure 5-12, *Communication scheduling* $\Delta_{T_}$Mean is given as W_Base/2 because variable Δ_T is uniformly distributed.

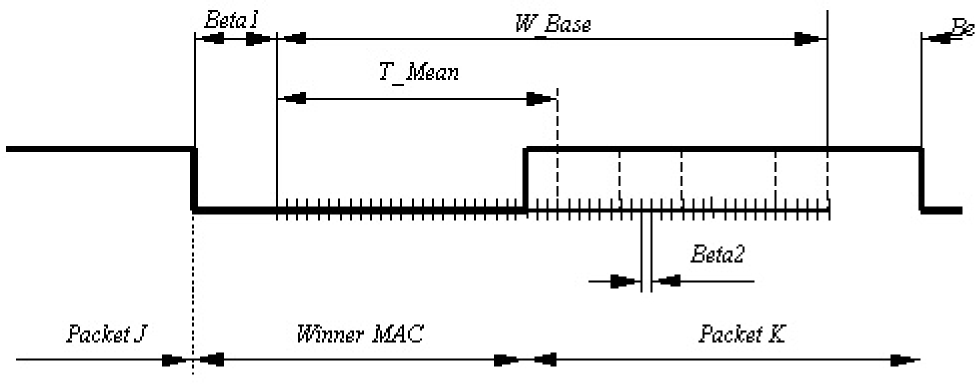


Figure 5-12. Communication scheduling

The MAC algorithm predictability is based on backlog estimation. Each node maintains an estimate of the current channel backlog BL, which is incremented as a result of the packet transmission/reception and decremented periodically every packet cycle. Each packet of MAC sublayer contains a variable representing prediction of the traffic arising as a result of processing this packet (the variable represents the number of messages that the packet shall generate upon reception). By adjusting the size of the randomising window as a function of the predicated load, the algorithm keeps the collision ratio constant and independent of the load.

5.5.6. Time-Triggered-Protocol (TTP/C)

TTP/C is a time-triggered communication protocol for safety-critical distributed real-time control systems. Its intended application domains are automotive control systems, aircraft control systems, industrial and power plants, or air-traffic control. This system is fault- tolerant, each node is connected to the bus with two replicated channels (Channel 0 and Channel 1). The bus, star or multi-star topology can be used. Access to the transmission medium is controlled by a static TDMA scheme (Time Division Multiple Access). Each node get predefined time to send messages, called slot. The scheduling scheme is made offline, all needed information about distributed system is known before start-up. All TDMA rounds are time identical. However, the length and contents of the messages (the application data) may differ.

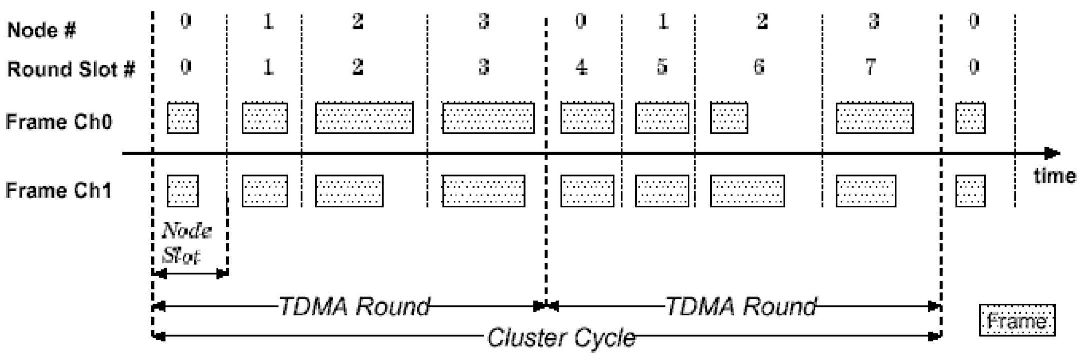


Figure 5-13. Media access control in TTP/C

Figure 5-12, *Communication scheduling*, shows that the slot can be used by different nodes in different TDMA rounds. In this figure a cluster cycle consists of four TDMA rounds. The last slot is shared by the multiplexed nodes 3,4 and 5, with node 3 sending in the TDMA round 0 and 2, node 4 sending in TDMA round 1 and node 5 in TDMA round 3.

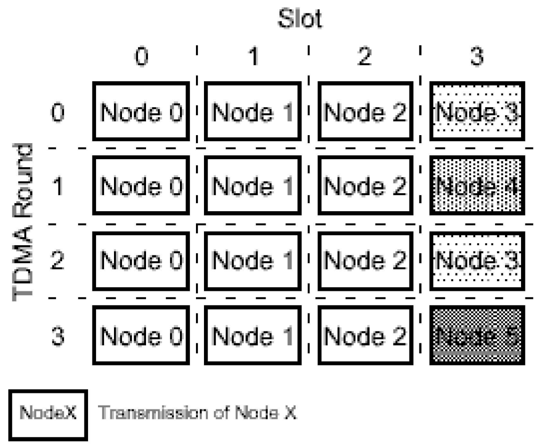


Figure 5-14. TDMA rounds in TTP

5.5.7. TCN (Train Communication Network)

Train Communication Network IEC 61375-1 is designed for train applications. The MAC control is made by TDMA and the basic period is divided into two phase, the periodic phase and sporadic phase. The periodic phase is controlled by the Master. Slaves cannot send spontaneously, even during the sporadic phase. Polled slave responses by broadcast frame.

5.5.8. Ethernet

Standard Ethernet networks use CSMA/CD (Carrier Sense Multiple Access / Collision Detection). This standard enables devices to detect a collision. After detecting a collision, the device waits a random delay time form a given interval and then it attempts to re-transmit the message. If the device detects a new collision again, the interval is doubled, new random delay is generated and the device tries to re-transmit the message. This is known as exponential back off. After 10 attempts the interval is no more doubled, and after the 16th attempt the message is rejected. More and more resources are investigated to exploit Ethernet in automation in the last years. Therefore, Ethernet technology infiltrates into automation area, where the fieldbuses are dominant today. Further are described communication protocols based on Ethernet: Modbus TCP and RTI.

5.5.9. Modbus TCP

The Modbus TCP was one of the first fieldbuses, taking advantage of Ethernet. The simplicity of standard Modbus allows the packing the Modbus frame into standard TCP segment. Then the data are sent through IP and Ethernet layer. A receiver decomposes the original data from the TCP packet. The advantage is very simple implementation, but on the other hand this protocol does not guarantee the delivery time due to its non-deterministic behaviour.

5.5.10. RTI (Real-Time Innovations)

The RTI uses standard IP stack for communication. It is based on the Publisher-Subscriber scheme. For time critical task, the NDDS (Network Data Delivery Service) is used. The NDDS is network middleware for distributed real-time applications that requires the fast and deterministic delivery of control and streaming data. NDDS uses UDP/IP and provides an API with publisher-subscriber and client-server services that eliminate socket programming. The publisher-subscriber model is extended to give subscribers flow control, application-transparent fault-tolerance, and determinism on a per subscription basis. The number of messages send to the network is limited, so the network throughput is guaranteed and NDDS is able to guarantee the probability of delivering the message before a given deadline. For non real-time applications the TCP protocol is used (see Figure 5-15, *RTI communication model*).

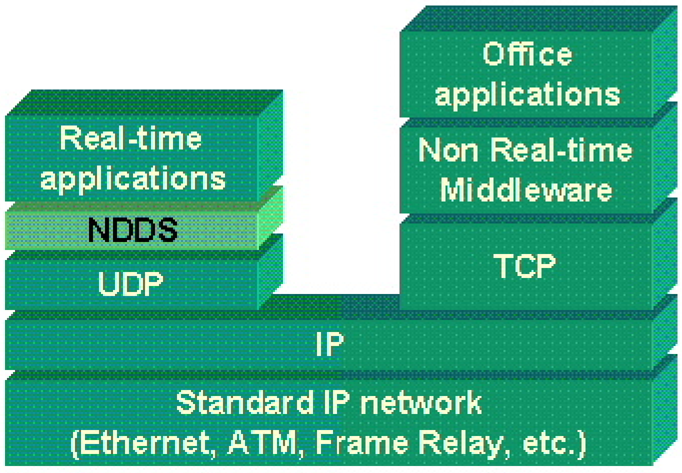


Figure 5-15. RTI communication model

5.5.11. RTNet

The RTNet uses some changes in IP and UDP stack for better real-time behaviour. The disadvantage is that the implementation is made only for RTLinux ver. 2.3 and RTAI ver. 1.3.

5.6. Concluding remarks

In general we can say that time-triggered systems are more robust and testable than event-triggered systems, but they require more planning and are less flexible (for more details see *TTP-A Protocol for Fault-Tolerant Real Time Systems*). Communication systems based on priority access to the media are probably a good candidate for reliable RT applications based on RT OS. In order to ensure a response time of distributed applications it is needed to verify them by suitable verification tools (unfortunately a classical verification approach by timed automata does not support preemption).

On the other hand CSMA/CD methods are non-deterministic and it is difficult to implement priority access to the media in this case. As a consequence can be used as best effort approach and they are not suited for hard real-time applications.

For further detailed information about fieldbus systems see reference list at http://ourworld.cs.com/rahulsebos/.

Bibliography

Scheduling

[Audsley93] N. Audsley, A. Burns, K. Tindell, M. Richardson, and A. Wellings, Software Engineering Journal, 5, 284-292, *Applying a new scheduling theory to static priority preemptive scheduling*.

[Baruah90] S. Baruah, A. Mok, and L. Rosier, 1990, IEEE Real-Time Systems Symposium, 182-190, *Preemptively scheduling hard real-time sporadic tasks on one processor*.

[Chen90] M. Chen and K. Lin, 1990, The Journal of real-time systems, 325-346, *Dynamic Priority Ceilings: A concurrency control protocol for Real-Time Systems*.

[Chetto89] H. Chetto and M. Chetto, 1998, IEEE Transactions on Software Engineering, 15, 1261-1269, *Some results of the Earliest Deadline Scheduling Algorithm*.

[Davis93] R. Davis, K. Tindell, and A. Burns, 1993, IEEE Real-Time Systems Symposium, 222-231, *Scheduling slack time in fixed priority preemptive systems*.

[Ghazalie95] T. Ghazalie and T.P. Baker, 1995, The Journal of real-time systems, 31-67, *Aperiodic servers in a deadline scheduling environment*.

[Lehoczky89] J. Lehoczky, L. Sha, and Y. Ding, 1989, IEEE Real-Time Systems Symposium, 166-171, *The rate monotonic scheduling algorithm: Exact characterization and average case behaviour*.

[Lehoczky87] J.P. Lehoczky, L. Sha, and K. Strossnider, 1987, IEEE Real-Time Systems Symposium, 261-270, *Enhanced aperiodic responsiveness in hard real-time environments*.

[Mok78] A. Mok and M. Dertouzos, 1978, 7th Texas Conference on Computing Systems, *Multiprocessor scheduling in a hard real-time environment*.

[Sha90] L. Sha, R. Rajkumar, and J.P. Lehoczky, 1990, IEEE Trans. on Computers, 39, 1175-1185, *Priority Inheritance Protocols: An Approach to Real-Time Synchronisation*.

[Baker91] T.P. Baker, 1991, The Journal of Real-Time Systems, 3, 67-100, *Stack-Based Scheduling of Realtime Processes*.

[Klein90] M.H. Klein and T. Ralya, Technical Report, CMU/SEI-90-TR-19, Carnegie Mellon University, Software Engineering Institute, 1990, *An analysis of input/output paradigms for real-time systems*.

[Leung82] J. Leung and J. Whitehead, 1982, Performance Evaluation, *On the complexity of fixed-priority schedulings of periodic, real-time tasks*.

[Leung80] J. Leung and R. Merrill, 1980, Information Processing Letters, 115-118, 18, *A note on the preemptive scheduling of periodic, Real-Time tasks*.

[Liu73] C.L. Liu and J.W Layland, 1973, Journal of the ACM, *Scheduling algorithms for multiprogramming in a hard real-time environment*.

[Locke92] C.D. Locke, 1992, Journal of Real-Time Systems, 4, 37-53, *Software architecture for hard real-time applications: cyclic executives vs. priority executives*.

[Ramos93] S. Ramos and J.P. Lehoczky, 1993, IEEE Real-Time Systems Symposium, 160-171, *On-Line scheduling of hard deadline aperiodic tasks in fixed-priority systems*.

[Ripoll96] I. Ripoll, A. Crespo, and A. Mok, 1996, Journal of Real-Time Systems, 11, 19-40, *Improvement in feasibility testing for real-time tasks*.

[Ripoll96] I. Ripoll, A. Crespo, and A. Garcia-Fornes, 1995, IEEE Transactions on Software Engineering, 23, 388-400, *An optimal algorithm for scheduling soft aperiodic tasks in dynamic-priority preemptive systems*.

[Spuri96] M. Spuri and G. Buttazzo, 1996, Journal of Real-Time Systems, March, 179-210, *Scheduling Aperiodic Tasks in Dynamic Priority Systems*.

[Sprunt88] B. Sprunt, L. Sha, and J.P. Lehoczky, 1988, IEEE Real-Time Systems Symposium, 251-258, *Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm*.

[Sprunt89] B. Sprunt, L. Sha, and J.P. Lehoczky, 1989, Journal of Real-Time Systems, 27-60, *Aperiodic task scheduling for hard real-time systems*.

[Shi01] W. Shi, 2001, Technical Report, Dep. of Computer Science, FSU, *Implementation and Performance of POSIX Sporadic Server Scheduling in RTLinux*.

Resource Reservation

[Tok90] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao, *Real-Time Mach: Towards a Predictable Real-Time System*, October 1990, USENIX Mach Workshop.

[Mer93] Clifford W. Mercer, Ragunathan Rajkumar, and Hideyuki Tokuda, *Applying Hard Real-Time Technology to Multimedia Systems*, December 1993.

[Mer93-2] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda, *Processor Capacity Reserves for Multimedia Operating Systems*, 1993, May 1993.

[Mer94] Clifford W. Mercer, Ragunathan Rajkumar,, and Jim Zelenka, *Temporal Protection in Real-Time Operating Systems*, May, 1994, IEEE Workshop on Real-Time Operating Systems and Software.

[Jon95] Michael B. Jones, Paul J. Leach, Richard P. Draves, and Joseph S. Barrera, *Modular Real-Time Resource Management in the Rialto Operating System*, May 1995, Fifth Workshop on Hot Topics in Operating Systems (HotOS-V) .

[Jon97] Michael B. Jones, Daniela Rosu, and Marcel-Catalin Rosu, *CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities*, October 1997, ACM Symposium on Operating Systems Principles.

[Raj98] Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano, and Shui Oikawa, *Resource Kernels: A Resource-Centric Approach to Real-Time Systems*, January 1998, SPIE/ACM Conference on Multimedia Computing and Networking.

[Les96] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden, *The Design and Implementation of an Operating System to Support Distributed Multimedia Applications*, September 1996, IEEE Journal on Selected Areas In Communications.

[Abe98] Luca Abeni and Giorgio Buttazzo, *Integrating Multimedia Applications in Hard Real-Time Systems*, December 1998, IEEE Real-Time Systems Symposium.

[Gar98] M. K. Gardner, J. W.S. Liu, *Performance of Algorithms for Scheduling Real-Time Systems with Overrun and Overload*, June 1999, Eleventh Euromicro Conference on Real-Time Systems.

[Abe99] Luca Abeni, Giorgio Buttazzo, *QoS Guarantee Using Probabilistic Dealines*, June 1999, IEEE Euromicro Conference on Real-Time Systems.

[Abe99-2] Luca Abeni and Giorgio Buttazzo, *Adaptive Bandwidth Reservation for Multimedia Computing*, December 1999, IEEE Real-Time Computing Systems and Applications (RTCSA).

[Oik98] Shui Oikawa and Ragunathan (Raj) Rajkumar, *Linux/RK: A Portable Resource Kernel in Linux*, December 1998, IEEE Real-Time Systems Symposium Work-In-Progress.

[Oik99] Shui Oikawa and Ragunathan (Raj) Rajkumar, *Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior*, June 1999, IEEE Real-Time Technology and Applications Symposium.

[Wan99] Yu-Chung Wang and Kwei-Jay Lin, *Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel*, December 1999, Real-Time Systems Symposium.

[Raj00] Ragunathan (Raj) Rajkumar, Luca Abeni, Dionisio De Niz, Sourav Ghosh, Akihiko Miyoshi, and Saowanee Saewong, *Recent Developments with Linux/RK*, December 2000, Second Real-Time Linux Workshop.

[Lip00] G. Lipari and S.K. Baruah, *Greedy reclaimation of unused bandwidth in constant bandwidth servers*, June 2000, Euromicro Conference on Real-Time Systems.

[Pal00] Luigi Palopoli, Luca Abeni, Fabio Conticelli, Marco Di Natale, and Giorgio Buttazzo, *Real-Time control system analysis: an integrated approach*, December 2000, Real-Time Systems Symposium.

[Abe01] Luca Abeni and Giorgio Buttazzo, *Stochastic Analysis of a Reservation Based System*, April 2001, 9th International Workshop on Parallel and Distributed Real-Time Systems.

[Pal] Luigi Palopoli, Luca Abeni, Gabriele Bolognini, Benedetto Allotta, and Fabio Conticelli, *Novel scheduling policies in real-time multithread control system design*, to be issued, Control Engineering Practice.

Fault-tolerance

T. Anderson and P.A. Lee, *Fault Tolerance - Principles and Practice.* , Prentice Hall, 1981.

A. Avizienis and J.P.J. Kelly, *Fault Tolerance by design diversity: Concepts and Experiments* , pp 67-80, *IEEE Computer Magazine* , 17, (8), 1984.

K. P., Birman, *Replication and fault-tolerance in the ISIS system* , pp 79-86, *ACM Operating Systems Review* , 21, (5), 1984.

H., Chetto, M., Silly, and T., Bouchentouf, *Dynamic scheduling of real-time tasks under precedence constraints* , pp 184-194, *Real-Time Systems* , 2, (3) , 1990.

D., David, J., Delcoigne, E., Leret, A., Ourghanlian, P., Hilsenkopf, and P., Paris, *Safety Properties Ensured by the OASIS Model for Safety Critical Real-Time Systems* , pp 47-59, *Computer Safety, Reliability and Security : Proc. 17th International Conference, SAFECOMP'98* , 1998.

D. L., Detlefs, M. P., Herlihy, and J. M., Wing, *Inheritance of synchronization and recovery properties in Avalon/C++* , pp 57-69, *Computer* , Dec. 1988.

J-C. Laprie, *Dependability: basic concepts and Terminology* , pp 265, Edited by J-C. Laprie, Springer-Verlag, 1992.

J-C. Laprie, *Dependability - Its attributes, impairments, and means*, pp 3-24 in , *Predictably Dependable Computing Systems* , Edited by B. Randell, Springer-Verlag, 1995.

J-C. Fabre, V. Nicomette, T. Perennou, R. Stroud, and Z. Wu, *Implementing Fault-Tolerant Applications Using Reflective Object-Oriented Programming* , pp 190 - 208 in , *Predictably Dependable Computing Systems* , Edited by B. Randell, Springer-Verlag, 1995.

W. Nace and P. Koopman, *Graceful Degradation in Distributed Embedded Systems* , pp 55-67 , *Dr.Dobb's Embedded Systems, inaugural feature for embedded system webzine, http://www.ddjembedded.com/resources/articles/2001/0106em001/0106em001a.htm* , 2001.

O. Gonzalez, H. Shrikumar, J. A. Stankovic, and K. Ramamritham, *Adaptive Fault Tolerance and Graceful Degradation Under Dynamic Hard Real-time Scheduling* , pp 55-67 , *Proceedings of the 18th IEEE Real-time Systems Symposium, San Francisco* , Dec. 97.

D. Powell, *Distributed Fault Tolerance - Lessons Learnt from Delta-4* , pp 16, Research Report 93192 , LAAS-CNRS, Toulouse,, 1995.

D. Powell, *A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems* , pp 260 , Edited by D. Powell, Kluwer Academic Publishers, Boston, 2001.

H. Kopetz, *The Time Triggered Approach to Real-Time System Design* , pp 54-66 in , *Predictably Dependable Computing Systems* , Edited by B. Randell, Springer-Verlag, 1995.

J., Reisinger, A., Steininger, and G. Leber, *The PDCS Implementation of MARS Hardware and Software* , pp 209-224 in , *Predictably Dependable Computing Systems* , Edited by B. Randell, Springer-Verlag, 1995.

K. Ramamritham and J. Stankovic, *Scheduling Algorithms and Operating Systems Support for Real-Time Systems* , pp 55-67 , *Proceedings of the IEEE* , Jan. 94.

S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, *An overview of the Arjuna Distributed Programming System* , pp 66-73 , *IEEE Software* , 8, (1), Jan. 91.

Communications

K. Tindell, A. Burns, and A.J. Wellings, *Calculating Controller Area Network (CAN) Message Response Time*, Control Eng. Practice, Vol. 3, No. 8, pp. 1163-1169, Elsevier, 1995.

Profibus specification EN 50170 Volume 2, Profibus International, Germany, 1998.

CAN Specification 2.0 Part B, CAN in Automation, Germany, .

Lontalk Protocol Specification ver. 3.0, Echelon Corporation, USA, 1994.

Time-Triggered Protocol TTP/C, High-Level Specification Document ver 1.0.0, TTA-Group, Austria, 2002.

H. Kopetz and G Grünsteidel, *TTP-A Protocol for Fault-Tolerant Real Time Systems*, IEEE, 1994.

Train Communication Network (TCN), norm IEC 61375-1, IEC, Switzerland, 1999.

Open Modbus/TCP specification rel. 1.0, Schneider Electric, 1999 .

NDDS Getting Started Guide,ver. 3.0, Real-Time Innovations, Inc., USA, 2002.

T. Führer, *Time Triggered Communication on CAN*, technical report, Robert Bosch GmbH, .

N. Navet and Ye-Q. Song, *Validation of in-vehicle real-time applications*, Computers in Industry, Elsevier,, 2000.

E. Tovar, *Supporting Real-Time Communications with Standard Factory-Floor Networks*, doctor thesis, 1999.

K.W. Tindel, H. Hanson, and A.J. Wellings, *Analysing Real-Time Communications: Controller Area Network (CAN)*, pp. 259-263, IEEE Real-Time System Symposia, 1994.