# WP2 - Architecture Specification

## Deliverable D2.1 - Architecture and Components Integration

**WP2 - Architecture Specification : Deliverable D2.1 - Architecture and Components Integration**

by Adrian Matellanes, Sergio Sáez, Patricia Balbastre, Luca Abeni, Pavel Pisa, Frantisek Vacek, Petr Smolik, Zdenek Sebek, Jan Krakora, Zdenek Hanzalek, and Agnes Lanusse

Published September 2002

# Table of Contents

# List of Tables

# List of Figures

# Document Presentation

**Table 1 Project Coordinator**

| | |
|---|---|
| Organisation | UPVLV |
| Responsible person | Alfons Crespo |
| Address | Camino Vera, 14 |
| | 46022 Valencia, Spain |
| Phone: | +34 963877576 |
| Fax: | +34 963877576 |
| Email: | alfons@disca.upv.es |

**Table 2 Participant List**

| Role | Id. | Participant Name | Partcipant acronym |
|---|---|---|---|
| CO | 1 | Universidad Politecnica de Valencia | UPVLC |
| CR | 2 | Scuola Superiore Santa Anna | SSSA |
| CR | 3 | Czech Technical University in Prague | CTU |
| CR | 4 | CEA/DRT/LIST/DTSI | CEA |
| CR | 5 | Unicontrols | UC |
| CR | 6 | MNIS | MNIS |
| CR | 7 | Visual Tools S.A. | VT |

**Table 3 Document version**

| Release | Date | Reason of change |
|---|---|---|
| 1_0 | 15/01/2003 | First release |
| 2_0 | 15/04/03 | Second release |

# I. Architecture

## Table of Contents

# Chapter 1. Introduction

## 1.1. Goals

The main objective of the project is the creation of a library of free-software components for the design and development of real-time and embedded systems (RTES).

To help development we will setup a framework encompassing all components, kernels and tools; we will call this framework the OCERA Framework (OF).

To reduce the difficulty of development within the OCERA framework we have to provide it in an easy and consistent way so people will be able to develop with not too much trouble.

It is reasonable to make an effort to avoid, as much as possible, the usual problems developers face when they begin using new libraries and tools. It is common to have dependency problems, unresolved paths, header files not found, etc... and those problems are many times what prevent people from using those tools. We expect to avoid those problems with the OF.

If someone get the OCERA software package, what should she be able to do with it? What the common use of this software would be? It is important to answer these questions in order to study and define how OCERA components will be built.

We try to provide a framework so developers can:

- Choose and Configure the OCERA components they need
- Develope their real-time/embedded software
- Use (possibly third parties') debugging tools
- Build effectively the embedded system image file

## 1.2. Real-Time in Linux

Linux is a full-featured UNIX implementation, conforming to the POSIX standard. The Linux kernel was not originally designed for real-time applications. Althought kernel developers are actively working in improving the responsiveness of the kernel, Linux is still not suited to support hard real-time applications with tight timing requirements for at least two reasons:

- Kernel responsiveness. The default mechanims used by Linux for protecting its internal kernel structures can cause long non-preemptable sections.
- The lack of real-time mechanisms, like priority inheritance, sufficient timing resolution, etc.

Several mechanisms have been proposed for supporting real-time in Linux. They can be divided in two distinct classes [Dankwardt00]:

- Mechanisms that use Linux for accessing the hardware. Following this approach, the latency of the Linux kernel must be reduced as much as possible. It is also necessary to modify the kernel by introducing real-time mechanisms like priority inheritance, dynamic scheduler, high resolution timers etc.
- Mechanisms that by-pass Linux for accessing the hardware. In this case, the hardware interrupts must be virtualized. The interrupts that are used by critical hard real time tasks are directly handled by-passing the standard Linux kernel. The other interrupts are forwarded to Linux when no real-time task is active. Also, cli and sti instructions must be modified to avoid that Linux disables interrupts for long intervals of time.

There are several on-going projects in both approaches. The so-called "low latency patch" and "preemption patch" belong to the first one. Also, several research groups proposed modifications to Linux for introducing real-time mechanisms. Examples of this approach are Linux/RK by TimeSys, RED-Linux, etc.

The second solution can be implemented in different ways. RTLinux [RTLinux.org] and RTAI [RTAI] implement a real-time executive (which consists of a interrupt handler plus a scheduler and some real-time mechanism) and real-time tasks as kernel modules that are dynamically linked with the Linux kernel and run in kernel space.

A different approach is used by other systems, like $L^4$-Linux [L4], in which a small microkernel is used for running both a modified version of the Linux kernel and real-time tasks in user space.

RTLinux and RTAI use similar mechanisms to acquire direct control of the hardware, by intercepting the interrupts and modifying a small portion of the Linux kernel code. On the other hand, $L^4$-Linux differs from previous approaches in that Linux runs completely in user space as a task on the $L^4$ microkernel. Therefore, the $L^4$ microkernel has full control of the "guest" operating system. As a consequence, by introducing a small overhead, it is possible to implement memory protection and other security mechanisms.

The OCERA architecture is based on the RTLinux architecture. Figure 1-1 presents a schematic overview of the relations between Linux kernel and RTLinux. RTLinux is located just above the hardware to represent the tight control that RTLinux has on it. Also it is important to note that while RTLinux has direct control of the interrupts (dark angled arrows), Linux works with virtual interrupts delivered by RTLinux. Hardware devices can trigger an interrupt which will be received by RTLinux, but will not be delivered to Linux until all real-time tasks are idle.

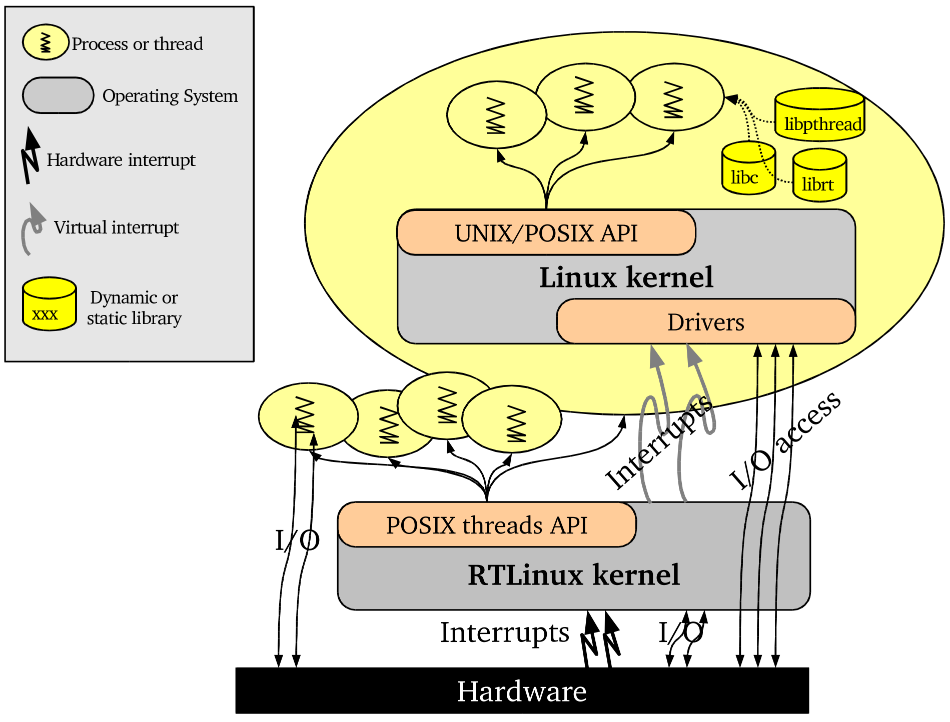


**Figure 1-1. General Linux and RTLinux overview**

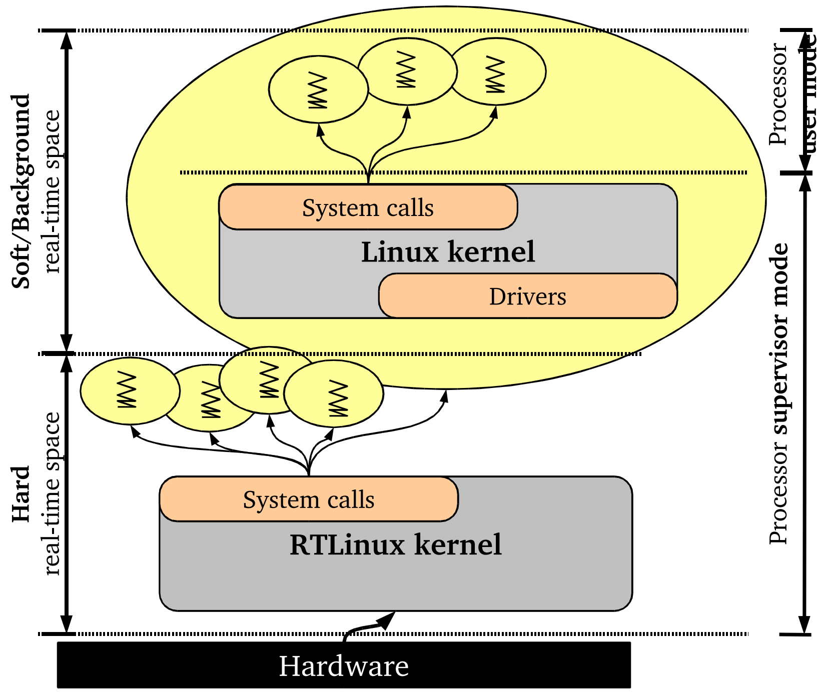


**Figure 1-2. Different view of the execution environments**

Although RTLinux and Linux run both in kernel space, when RTLinux is installed, the code of Linux is modified (applying a patch to the sources files and recompiled) to prevent Linux from disabling interrupts. As a result, Linux has no direct control of the interrupt and timer subsystems. For this reason, RTLinux is used for hard real-time execution, and Linux jointly with its user applications is used for background (or non-hard) real-time activities. In OCERA, we plan to provide components both at the RTLinux level and at the Linux level.

As a summary of the pros and cons of the RTLinux approach to real-time: RTLinux applications run faster; can work with the hardware like most programmers are costumed to do (MSDOS® style); but buggy task can compromise the system integrity due to the lack of protection.

## 1.2.1. Concurrent execution paradigm

An operating system can provide concurrent execution in two different forms: processes and threads. In general, each process has a protected virtual address space, a set of open files, one single execution flow (program counter), etc. Processes are self-contained units. A thread, on the other hand, consists of an execution context and a state (i.e., a program counter, a stack, and local variables). Usually, a thread is contained within a process; thus, a thread has global access to all states within its containing process. A process can contain more than one thread. Threads contained by different processes may only communicate through inter-process communication mechanisms.

The main differences between processes and treads as real-time is concerned are summarised in the following table:

**Table 1-1. Process versus thread**

| Process | Thread |
|---|---|

| | |
|---|---|
| * Memory protection among processes.<br>* Easy to distribute in a distributed system.<br>* Well defined and complete programming API. | * Easy and small support implementation.<br>* Fast context switch.<br>* Intrinsic shared memory.<br>* Efficient communication.<br>* Finer grain parallelism. |

It is common to use both methodologies in a real-time embedded application. Threads are better suited to program the low level, hard real-time activities, while processes can be used for human interaction and tasks that do not require short timing response.

Most of the POSIX real-time extensions are based on the thread model, an example of this is that solution to priority inversion is only available in thread API (`PTHREAD_PRIO_PROTECT` and `PTHREAD_PRIO_INHERIT` protocols). POSIX profiles define subsets of the operating system API which fulfil the requirements of specific targets. There are four profiles: "Minimal System", "Controller", "Dedicated System" and "Multipurpose System". The two first profiles do not have process support, only thread support is required. RTLinux follows the "Minimal System" standard.

There are three different libraries that provide thread support on Linux. The first implementation was done By Xavier Leroy and is known as the LinuxThreads. It is integrated and distributed jointly with the standard "C" library. Currently there are two new competing implementations [Cooperstein02] of the POSIX thread standard called: “New Generation POSIX Threads” (NGPT) and “Native POSIX Thread Library” (NPTH). These implementations improve both, compatibility with POSIX standard and performance.

OCERA components will assume the thread model if not explicitly stated otherwise. The term “task” will be used to refer to both process or thread.

## Bibliography

[Cooperstein02] Jerry Cooperstein, 07/11/2002, The O’Reilly Network, *Linux Multithreading Advances*.

[RTLinux.org] Der Hofrat, *Open Source RTLinux Repository*.

[RTAI] Paolo Mantegazza, *RTAI Home page*.

[L4] *DROPS - The Dresden Real-Time Operating System Project*.

[Dankwardt00] Kevin Dankwardt, 11/2000, Linux Devices.com, *Comparing real-time Linux alternatives*.

# Chapter 2. Software Architecture

## 2.1. A Tale of two Levels

A real-time application is normally composed by multiple tasks with different levels of criticality. Although losing deadlines is not desirable in a real-time system, soft real-time tasks could lose some deadlines and the system could still work correctly. However hard real-time tasks cannot lose any deadline or undesirable/fatal results can be produced in the system. From this point of view, a real-time application can be organised in two levels: hard real-time tasks and soft real-time tasks.

Within the OCERA project we will regard, thus, real time systems from these two levels of criticality. Both hard and soft real time software will run with the same customized kernel, from now on the **OCERA Kernel** (OcK).

### 2.1.1. Hard Real-Time System Configuration

Hard real time tasks will give precise and deterministic control of the system, very high time accuracy and very low latency. Hard real time systems will be provided through a RT-Linux kernel and modules. As explained in "D1.1 RTOS Analysis" hard real-time tasks are implemented in kernel modules, and as such, development of hard real time systems will resemble the development at kernel level: no memory protection, no filesystem, etc... Neither the common user space tools and libraries nor the Kernel services are available for hard rt-tasks.

In this configuration the whole application runs in the RTLinux layer, where hard real-time performance can be guaranteed. OCERA Linux will be a stripped kernel version with the minimal functionality and memory footprint just to boot the system and execute simple background tasks.

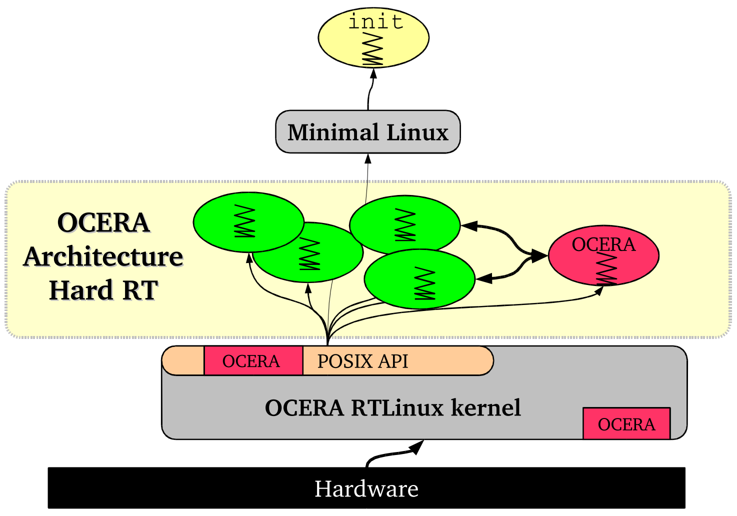


**Figure 2-1. Hard Real-Time applications**

### 2.1.2. Soft Real-Time System Configuration

Soft real time tasks, on the other hand, do not provide so much time accuracy, they depend on the Linux scheduler (or other scheduler we might install) and they are likely to miss some deadlines. To minimize latency and deadline misses we will patch the Linux kernel in a way that minimizes scheduler latency, give preemptability to the kernel, provide resource management and Quality of Service (QoS), etc... Soft real time systems will be developed at user space and then the usual mechanisms for their development are available.

The RTLinux layer is also optional in the OCERA architecture. It can be removed if it is not required, which results in a embedded system with a single enhanced Linux OS.

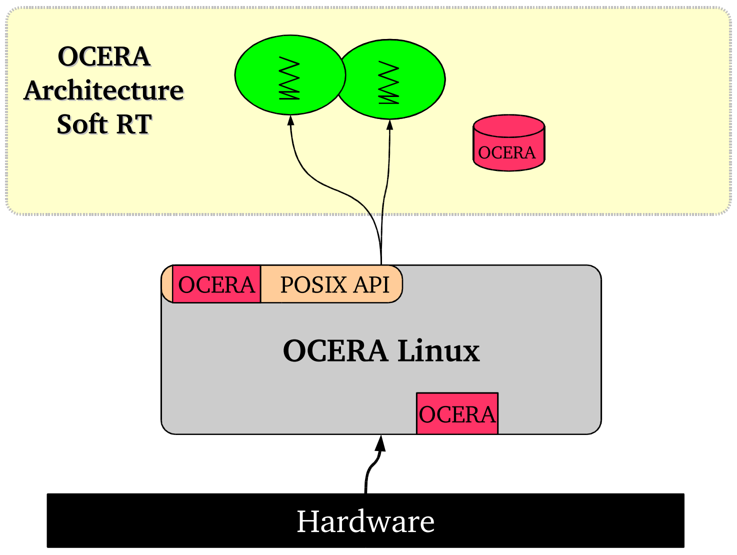


**Figure 2-2. Soft Real-Time applications**

This architecture should be the first choice when starting a new application design since it is closer to the standard development methodology of general Linux applications (applications that do not directy handle hardware and have no timing requirements), and also it is a more robust execution and debugging environment.

## 2.1.3. Hard and Soft Real-Time System Configuration

Most complex applications require a hybrid solution, requiring at the same time both hard and soft real-time, this configuration of the OCERA architecture allows the user to organise the real-time tasks in two groups: hard real-time tasks and soft real-time tasks. This special feature is provided by a QoS component which reserves a fraction of the processor time for the Linux layer without endangering the the correct and predictable execution of the hard-real-time tasks.

Hard real-time tasks will give precise and deterministic control of the system, very high time accuracy and very low latency. Hard real time performance will be provided at the RTLinux layer. The main design criteria at this level is predictability and low overhead (what is necessary to build a real-time system); therefore, this layer lacks most of the general facilities found in conventional OS but provides as much determinism as the underlying hardware provides.

The Linux layer, on the other hand, does not provide so much time accuracy, its execution depends on both the RTLinux work load and on the Linux unbounded behaviour. To minimise latency and deadline misses the OCERA Linux kernel will integrate several patches (low latency patch, preemptable patch, etc.), as well as the OCERA developed components.

The Linux layer is mainly used for serving user-space applications: in user-space, tasks can access the full range of Linux services, like drivers, debugging tools, network, graphics, file system, etc.

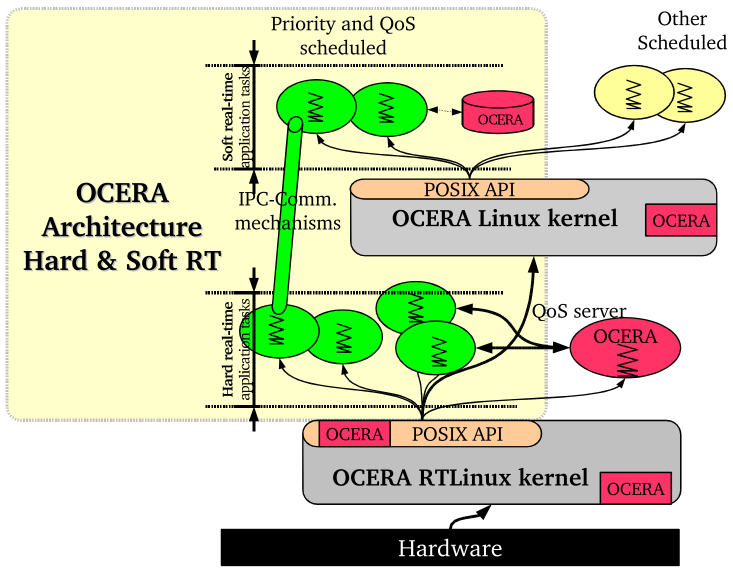


**Figure 2-3. Hard & Soft Real-Time applications**

Although the same API cannot be provided at both levels, some OCERA components are aimed at improving the API compatibility of both execution layers; being, at the same time, as close as possible to the POSIX real-time extensions. The benefit will be twofold:

1. It will be possible to develop real-time applications in user-space. At this level, we can use many debugging facilities like gdb, dynamic memory debuggers, etc., and the system will not hang on application bugs.
2. For some application, like applications with soft or firm timing requirements, the decision about which part of the application will run in the kernel space or in user space has not to be taken at the initial design phase, but can be delayed after the implementation, at the deployment phase.

### 2.1.4. Distributed Architecture

Real-time distributed system under consideration consists of application tasks, operating system and communication support. Hard and soft real-time application requirements have strong impact on the architecture of such systems. If given real-time requirements have to be met then communication support has to be able to guarantee specific characteristics of the message delivery paradigm. The matter is quite complex since the message delivery is realized via shared media.

Specifically the deterministic protocol behaviour, the message priority and the guaranteed message delivery time are required in hard real-time distributed systems. Concise verification of such systems considering all possible states is needed when analysing hard real-time applications. Consequently CAN bus is well-suited communication support for hard real-time distributed systems and it is widely used in this application area. "OCERA Hard RT CAN architecture" is accessed via POSIX compliant VCA (Virtual CAN API). VCA offers minimal set of functions enabling to open/close/configure CAN device and to send/receive CAN message.

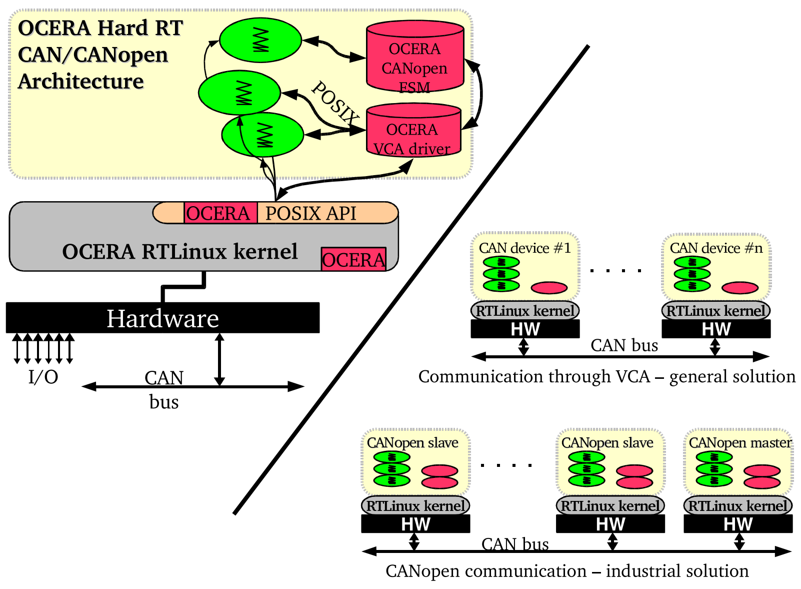


**Figure 2-4. Distributed Architecture**

Since a complex distributed application possibly consists of non-Linux products (e.g. sensor - driven by programmable 8-bit micro-controllers) and various third party products (e.g. operator interface - closed non-Linux system that can be parameterised but not programmed), it is needed to use application level protocol in OCERA architecture. Choice of CANopen is based on a fact, that it is open, well documented and widely used in factory automation. "OCERA Hard RT CANopen architecture" is formed by CANopen communication standard in the same manner as it is used in typical factory automation areas. It is based on Finite State Machine (FSM), which enables to access data stored in device Object Dictionary (OD) to other CANopen devices connected to CAN bus. VCA is used to connect CAN bus and FSM.

On the other hand the network throughput and message delivery time given as statistic values (e.g. probability distribution functions) are sought in soft real-time distributed systems. Performance measures based on simulations or stress tests are usually appropriate for evaluation of these systems. Consequently RT-Ethernet is well-suited communication support for soft real-time distributed systems since it has high performance, guaranteed throughput (isolated from external traffic, publisher/subscriber architecture resulting in restricted traffic from upper layers), and wide range of supporting hardware. "ORTE (Ocera Real-Time Ethernet) architecture" is implementation of RTPS protocol originally developed by RTI. As a consequence it is composed of one "Manager" per each node and one "Managed application" per each local or remote user application. Publisher/subscriber mechanism is used to share data between user applications.

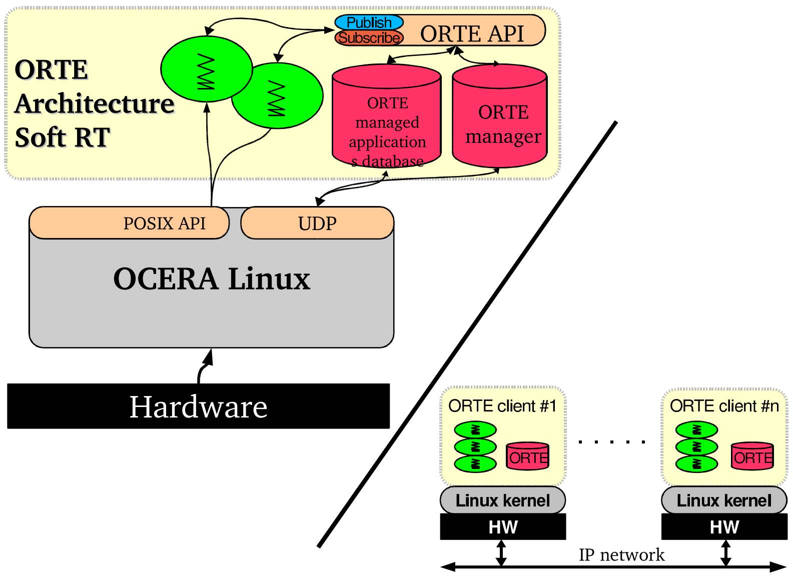


**Figure 2-5. ORTE architecture**

## 2.2. Fault-Tolerance Management

The goal of the fault-tolerance in OCERA is intended to provide basic functionalities to develop robust applications. In order to achieve this objective, specific components will be developed allowing task monitoring, redundancy management and dynamic reconfiguration.

### 2.2.1. Applications characteristics

OCERA applications can be deployed to one or more nodes configuring a distributed system, where nodes can be connected through CAN/Open bus or RT-Ethernet network.

The application properties tackled within the project will concern: safety, liveliness and timeliness.

- Safety means that nothing bad can happen, or in other words that if something goes wrong, we can insure that a safe state can be and is reached.
- Liveliness means that the system required services are provided.
- Timeliness means that services are delivered on time.

We will not consider security issues.

### 2.2.2. Faults considered

There are several sources of faults or errors in this kind of embedded applications.

The following hypotheses will be made concerning hardware devices:

- they will be fail-silent,
- byzantine behaviours will not be considered,
- critical hardware components will have a mechanism allowing for safe stop in case of silent node.

Hypotheses on communications

- communications are atomic,
- in case of a silent node. Communications are silent.

Additionally, it is assumed that kernel crashes results in a silent node.

Prevention of such errors can be done by providing good practice programming rules that can be used for the development of tasks or drivers. Code analysis tools and specific compilation techniques with appropriate segmentation strategies can also reduce this risk. By the end of the project guidelines will be written to help developers write safe rt-tasks.

Fault-tolerance in OCERA will focus on the management of timing errors and on errors raised by the RTLinux executive. By timing errors we consider missed deadlines for time-critical tasks and watchdogs fires. We will mostly handle deadline missing and provide patterns for default actions (for example provide a default value for a periodic acquisition task restart for the next period and controlling that this does not occur more than a given number of successive iterations). Other patterns will be developed for different types of tasks. Specific watchdogs will be used to detect fail-silent components and will thus reveal a miss-functioning in the system.

The fault-tolerance facilities will analyse the possible consequences of such errors and apply actions decided by the developer at design time and stored in run-time databases.

An application can define several operational modes that will be predefined at design time. An operational mode is an attribute of the on-board software as a whole, which purpose is to adapt the system (set of tasks) behaviour to various sets of external conditions. Tasks may have several alternative behaviours. A mode will encompass a set of tasks and a statement on which behaviour must be adopted for each task. For instance some applications may have an initialisation mode and then several possible functioning modes. Switching from one mode to another can be decided by the application on reaching a certain state or can be decided by the Application fault-tolerant monitor.

With respect to timing fails, several strategies are proposed: the task is late because of temporary overload, a task is blocked, a task is waiting for a failed node. They will also depend on the nature of the task: periodic acquisition task (in this case a default strategy can be defined by the developer), command task towards an actuator is not responding (failure can be assumed), computational task (imprecise computation strategy can be implemented). Within OCERA a global framework consisting of a set of building blocks will be developed. These building blocks will implement basic strategies in the beginning of the project and will be enriched as the project progresses.

## 2.2.3. Fault-Tolerance in OCERA

The approach chosen in the project is to provide a declarative way of managing fault-tolerance. It is based on a first stage which consists in collecting fault-tolerance requirements and default strategies at design and use this information to instantiate run-time fault-tolerance components that will monitor the application and take appropriate decisions at run-time on abnormal situations. The gathering of information and the configuration phase will be supported by a design/build tool set that will settle the fault-tolerance infrastructure.

As stated in the introduction, we will focus on the development of a framework offering basic building blocks aimed at two well identified goals. The first goal will be to provide mechanisms allowing mode management in order to increase applications robustness and achieve the safety requirements. The second goal will be to implement transparent redundancy mechanisms between nodes, which will provide continuity of service.

Two sets of components will thus be developed:

- The first one devoted to mode management consisting of two complementary components: i) The FT controller will collect low level information on ongoing tasks and notify the application monitor on tasks completions or abnormal situations. The FT controller will apply local emergency actions on faulty tasks. ii) The Application FT monitor will take global decisions on tasks reconfiguration decisions. These decisions will result in new constraints for the QoS scheduler (stop tasks, start new tasks implementing alternate behaviours, etc...)

- The second set of components will implement transparent redundancy facilities for critical tasks (declared critical by the user). Such tasks will be automatically replicated, using a passive replication method) and a specific protocol will be defined to permit synchronisation of replicas states. The controlling structure for the management of redundancy will rely on two kinds of components.

  These components will consist on: a task redundancy manager that will monitor redundancy of a cluster of replicas and decide when to activate or deactivate a replica; and a task replica manager that is charge of synchronization and communication of a particular replica with its master for check-pointing.

As a conclusion, the expected results from the fault-tolerance work_package will consist in a set of design/configuration and run-time components permitting to enhance applications robustness. Fault-tolerance characteristics targeted are the following :

- A faulty-task will have a degraded behaviour defined at design time and implemented as a dormant task (with state updates)that will possibly be activated on error detection. This degraded behaviour will not support a subsequent failure but can be used to implement a graceful stop.
- A faulty-task or a faulty component that compromise a service may activate a degraded service implemented through a mode change activation of several tasks. Several cases will be considered : the service can be fulfilled at least partially by a different combination of tasks or not, the service is mandatory for the overall application or not. Different strategies of reconfiguration will be implemented.
- Redundancy mechanisms will permit to insure the continuity of service of a redundant task. In this case, if the master is faulty, the control will be passed onto the slave and no degraded mode will be enabled unless the last copy becomes faulty.

## 2.3. OCERA Kernel

As we said in the previous section, OCERA will produce a kernel that will be composed by a Linux kernel together with the RT-Linux extension. This unique kernel (per supported architecture) will allow us to run both kinds of tasks, hard and soft.

The characterization of the OCERA kernel that will give us **hard** real-time capabilities is done through the RT-Linux kernel. The RT-Linux kernel guarantees the hard real-time behavior as explained in the previous section. We will also add some low-level components that will complement the standard RT-Linux funtionalities.

On the other hand, the characterization of the OCERA Kernel to support **soft** real-time capabilities is done through the application of several patches to the standard Linux kernel and some low-level components we will present later. The patches that we will include in the OcK are:

- High Resolution Timer Patch
- Preemptable Kernel Patch
- Low Latency Kernel Patch
- BigPhysArea Patch

These patches are currently available but we have to make a considerable effort integrating all of them and making them working correctly on top of a RT-Linux kernel.

As starting point we will use a standar Linux kernel version 2.4.18 and the RT-Linux extension version 3.1. We will evolve with the following kernels in the 2.4.x series but we will also be very interested in the development of the 2.5.x series (still in a very preliminar development status) since some of the patches and features that we have to add now to the 2.4.x kernels are to be included into the mainstream kernel development.

## 2.4. OCERA Framework

In order to facilitate the development of Real-Time and Embedded Systems (RTES) we will provide a bundle of applications, libraries and source files that will help in the development, testing, debuging and deployment of the RTES application, this framework is what we will call OCERA Framework (OF).

There will be just one framework and it will permit the development of any kind of RTES, hard, soft or both and for the targeted architecture.

When developing RTES one have to face the tedious tasks of compiling, generating the binary image, copying it into the target device (usually flash memory), testing, debuggung...and back again. In OCERA we are concern about the effort that developers have to make to debug and deploy their applications and the amount of time spent in these tasks.

The OF will let us:

- Generate (compile) hard and/or soft real-time applications

  We will setup the appropriate directories and compilation rules to make compilation of RTES a trivial task.

- Configure the target system and utilities.
- Monitor, test and debug the application.

  We will adopt a free software tracing tool and integrate it with the OCERA Kernel and Components; we are still analyzing different possibilities but it is very likely that we will take LTT (Linux Trace Toolkit) which already has real-time support and provide it with the POSIX Trace API.

- Generate the final binary image for deployment

## 2.5. Components

The main goal of this project is to deliver a structured set of components to ease the development of RTES providing new services not available in current RTOSes. These components are responsible for providing those new services or functionalities. This will be mainly performed by means of modifying the current Linux and RTLinux kernels or adding new software layers over them using their current POSIX APIs.

Most computer applications can be designed following a clear and standard software engineering methodology like object oriented methodology or client/server model. The code that is part of the operating system can not be easily categorised or described in a unique way, specially when the type of operating system is an RTOS. In this case, the intrinsic characteristic of an RTOS: efficiency and predictability of the code are mostly opposed to clearly divided and well organised code. It do not mean that the internal code of the RTOS is bad written or chaotic, what it means is that the RTOS can not be structured in a well defined and clearly separated blocks of code with clear inputs and outputs. For example, when RTLinux is loaded (inserted as a kernel module) it takes the control of the interrupt system by modifying the Linux code while Linux is running, which is one of the less advisable programming methods but provides the fastest result.

One of the definitions of what is an operating system is that it is the piece of software that hides all the complexity of the underlaying hardware to provide a clear and orthogonal programming environment to user applications.

### 2.5.1. Definition

An **OCERA component** is a piece of software that brings some new functionality or feature in some of the fields of the Embedded and Real-Time Systems that are of interest for OCERA: Scheduling & RT-Kernels, Quality of Service, Fault-Tolerance and Communications.

As a piece of software we mean:

- a modification of the Linux kernel or RTLinux executive, which will be released as a patch file against a specific kernel version or integrated into the final OCERA kernel;
- a module which can be loaded (with the `insmod/modprobe` commands) and provides new functionalities and may use some of the already installed services;
- a library, dynamic or static, which can be linked with the user application;
- or a standalone thread or process (for example a debugging program).

### 2.5.2. Component Classification

OCERA components can be classified according to two criteria:

- Protection
- Level

According to the “protection” criteria, we can identify components that run in user space and components that run in kenel space. User space components are Linux applications running in their own address spaces at the lowest privilege level. Because of the protection mechanism enforced by the Linux kernel, these components are not able to crash the system, even if they misbehave and try to access random memory locations.

Components running in kernel space are: the (patched) Linux kernel (including its own device drivers), the RTLinux executive, and the hard real-time tasks (also referred as RTLinux tasks or RTLinux applications).

According to the “level” criteria, kernel space components can be located in the RTLinux layer (as modifications or additions to the RTLinux executive) or in the Linux layer (as modifications to the Linux kernel, which are independent from RTLinux).

Note that some functionalities can be provided both in user space and in kernel space (for example, the QoS manager developed by SSSA will be available both as a kernel module or as an user space daemon), and other functionalities can be implemented at different levels in kernel space (for example, the CBS scheduler will be implemented both in the RTLinux layer - by UPVLC - and in the Linux layer - by SSSA). For this reason, each component documentation will contain precise information about its protection level (user space / kernel space) ant its location (RTLinux layer / Linux layer).

It is worth to note that the RTLinux layer can be logically divided in three sub-layers:

1. **Low-level RTLinux**: This kind of components are highly related with the current RTLinux capabilities or internal algorithms, thus *it requires to modify the current RTLinux source* code in order to provide the new functionality or to improve an available one.

   This kind of component will be distributed in a patch-form and hopefully incorporated in the main stream of the implied kernel source.

2. **High-level RTLinux**: *It only needs the current API of the RTLinux kernel* or an extended API offered by other kernel component in order to implement its new functionality. It does not require to modify the existing kernel source code or any low-level kernel component.
3. **RTLinux Applications**: This kind of component uses the kernel API to provide a new service. It does not require to modify the existing kernel source code or any kernel component. The main characteristic of these components are that they are

implemented as an application-level processes/threads, offering some kind of service to other processes/threads (as a kind of a classical UNIX daemon).

In a similar way, the Linux layer is also split in two layers:

4. **Low-level Linux** : Like the low-level RTLinux executive components, these components modify the current kernel and has to be distributed as patch files.
5. **High-level Linux** : Components located inside the Linux kernel that use but do not modify the Linux kernel code. Device drivers are components of this category.

The third level (Linux applications) coincides with user space applications.

Most of the components will fit in one of these categories, but others will require the modification of several layers. For example, the CBS component at the RTLinux executive will also require small modifications of the Linux kernel.

## 2.5.3. Characteristics

Lets enumerate some of the general characteristics of the OCERA components.

- **Open Source**

  **OC** will be released under GPL or GPL like licenses. Some components may be incorporated into the RTLinux/Open distribution and in that case they will be covered by the RTLinux/Open License.

- **Unique Level of Application**

  Each OC will be developed just for one level of criticality, either hard or soft real-time. This will let us separate configuration and parameterization and get smaller footprints.

  We can have, nevetheless, the same feature or service in both levels, hard and soft, but this would be provided by different components, whether they share API (which will be the common case) or not.

- **Uniformized API**

  All OC will share a common API structure. They will adopt POSIX if available and a POSIX draft or POSIX-like API if not. All OC will have their API uniformized: names, profiles, etc...

- **Compatibility with OCERA kernel**

  Although the components can be developed for any architecture, kernel, etc... They at least must be provided compatible with the OCERA Kernels. Notice that there will be several kernels depeding on the target architectures

- **Small footprint**

  Having in mind the development of Embedded Systems, the OCERA components footprint will be kept as small as possible

  Although there is not specific maximum size for an OCERA basic system (just the system and libraries), we should strongly limitate its footprint to allow the development of embedded systems where non-volatile memory or disk capacity are very reduced.

- **Uniformity**

  All OCERA components will be provided in a standard way, including documentation, examples, etc... This is important for several reasons, first to ease development with OCERA components, second to allow people to contribute (to the existing components or adding new ones)

  □ Well Documented

Well documented. The following questions will be considered when preparing the documentation:

- What is the component useful for.
- Small review of similar or related facilities in other RTOS.
- How it can be used: is it a patch, a stand alone module, a thread.
- Configuration parameters if any.
- Complexity analysis, both temporal and spatial memory worst case analysis.
- How it is installed. In the case that the component could be used independently of the OCERA distribution, dependencies with other components from OCERA or external developers will be specified.

□ Usage examples
Basic usage examples will come with each component.

□ Regression Tests
Regression tests, used to validate the correct implementation of the component.

□ Cross partner validation
Every component will be developed by a partner and reviewed by a different one.

- **Minimum Dependencies**

  There can be dependencies between components, but these dependencies have to be maintained to a minimum

  Sometimes we will develop components that make use of features provided by other components. Separating components increases modularity and eases development, but we have to take care not to add artificial dependencies. Frequently developers do not take much care about the dependencies of their software. Keeping dependencies to a minimum allows exchangeability, minimize the use of memory and system footprint, etc... so we will develop the components with a “just what needed” philosophy.

### 2.5.4. External Structure

From the point of view of an external developer that wants to build her RTES using the OCERA framework, an OCERA component is simply a feature or service that can be activated or deactivated, and optionally configured or customised for the target application.

It might be the case that a selected component modifies the kernel source code but this should not be an issue for an external development. On the contrary, an external developer will not notice whether the components she has selected are kernel modules or they modify kernel sources; for her it will be completely transparent.

The only difference an external developer should notice is that some components will be activated/deactivated as a kernel feature, and configured in the same way, and others will be activated as a user process/thread that offers some new service, i.e., as a kind of a classical UNIX daemon.

### 2.5.5. List of Components

Though they will be explained in detail later in, we present here a complete list of components that will be develop within OCERA:

- **Kernel and Scheduling**
  - RTL-ADA: Porting of Runtime support of the ADA (GNAT) system to RT-Linux
  - POSIX Timers in RT-Linux
  - POSIX Non-realtime signals in RT-Linux
  - POSIX Barriers in RT-Linux
  - POSIX Message queues in RT-Linux
  - Application defined scheduler in RT-Linux
  - POSIX Trace support
  - Dynamic Memory Management support in RT-Linux
  - Constant Bandwidth Server (CBS) in RT-Linux
- **Quality of Service**
  - Scheduler for resource reservation in Linux
  - Quality of Sercive (QoS) manager
  - User library for accessing previous components
  - A series of patches to the RT-Linux/Linux kernel to solve problems with preemption, low-latency and high-resolution timers patches.
- **Fault-Tolerance**
  - Design & Building Tool
  - Application fault-tolerance monitor
  - Fault-Tolerance controller in kernel level
  - TaskReplicaAgent
- **Communications**
  - Virtual CAN API
  - CANOpen device
  - EDS parser and CAN/CANOpen analyzer
  - Real-Time Ethernet (ORTE) device
  - Real-Time Ethernet analyzer
  - CAN model by timed automata/ Petri Nets
  - Verification of cooperative scheduling and interrupt handlers

# Chapter 3. Hardware Architecture

The mainstream development of OCERA will be done in x86 platforms. Although we have still to face some uncertainties, we have chosen two other architectures OCERA will support. These architectures are Motorola PowerPC and Intel StrongArm, mainly motivated by the embedded systems character of the project.

Here we explain with some more details why we have decided to support such architectures.

## 3.1. x86

Despite the fact that the OC will be mainly developed in x86 environments, the fact that RT-Linux is originally being developed in x86 (and then ported to other architectures) and that x86 is the architecture most people have at hand when going to develop, we find that there is an increasing interest in the development of x86-based embedded systems.

The appearance of smaller and cheaper x86 motherboards together with the commercialization of low-consuming processors (e.g. VIA) and the demand of more performance makes x86 an architecture appropriate for the real time and embedded market.

## 3.2. PowerPC

The customers from the critical applications sectors (including namely transport, gas industry, power stations, ...) usually require Motorola based hardware for security and reliability reasons.

Important part of embedded applications is based on Motorola processors as they are designed for control applications (while x86 is mainly focused on PC) and they have many peripherals integrated on the chip (CAN, Ethernet, timers...)

Developers of real time applications prefer this family for the architecture and free development tools. PowerPC seems to be a perspective candidate for the further development of embedded applications using RT-Linux.

## 3.3. StrongArm

StrongARM [SA1100] is a family of high speed, low power processors specifically designed for portable and embedded systems, such as handheld devices. The processors, which were jointly developed by ARM [ARM] and Digital Equipment Corporation, are now available from Intel. ARM's microprocessor cores [Jagger95] are rapidly becoming the volume RISC standard in such markets as portable communications, hand-held computing, multimedia digital consumer and embedded solutions.

One of the most used processors of the StrongARM family is the Intel SA-1110. It provides the performance, low power, integration and cost benefits of its predecessor (Intel SA-1100), plus a high-speed memory bus, flexible memory controller and the ability to handle variable-latency I/O devices such as high performance graphic devices. The Intel XScale processors are an improvement of Intel StrongARM family. It combines high performance, small size, low power and modest cost.

A wide variety of PDA models used StrongArm processors, such as Sharp Zaurus [Sharp], Intrynsic Cerf [IntrinsycCerf]. All iPaq handhelds of Compaq [iPaq] contain XScale or StrongARM processors (depending on products).

This architecture is supported by several embedded operating systems, such as Windows CE and VXWorks. Linux has been ported to ARM processors (included SA 110 and 1100) [ArmLinux], and a wide variety of tools and distributions, such as GNU, Debian, have been developed to support it. It is also available a straightforward port of the XFree86 implementation of the X Window system [HandHelds]. Furthermore, commercial embedded linux distributions [Montavista] support both StrongARM and Xscale.

There is also a patch for RT-Linux (v 3.0) and Linux 2.4.16 that ports SA 1100 processor [Imec].

Therefore, since the basic support has been already developed, the main effort will be focused on adding tools and components for real-time features. This will be very necessary if consumer electronics market place consolidates for domotic applications.

## Hardware Bibliography

## StrongARM

[Montavista] *Monta Vista http://www.mvista.com/*.

[ARM] *ARM http://www.mvista.com/*.

[Sharp] *Sharp Zaurus handhelds http://www.sharpsec.com/*.

[IntrinsycCerf] *Intrinsyc Cerf Products http://www.intrinsyc.com/products/compare.asp/*.

[iPaq] *iPaq Compaq handhelds http://www.compaq.com/products/handhelds/pocketpc/*.

[HandHelds] *Open source software for use on handheld and wearable computers http://www.handhelds.org/*.

[Imec] *RTLinux patch for StrongARM http://www.imec.be/rtlinux/*.

[ArmLinux] *ARM Linux Project http://www.arm.linux.org.uk/*.

[SA1100] Intel Corporation, *SA 1100 Microprocessor Technical Reference Manual*, September, 1998 .

[Jagger95] Edited by D. Jagger, *Advanced RISC Machines Architecture Reference Manual*, Prentice Hall, July, 1995.

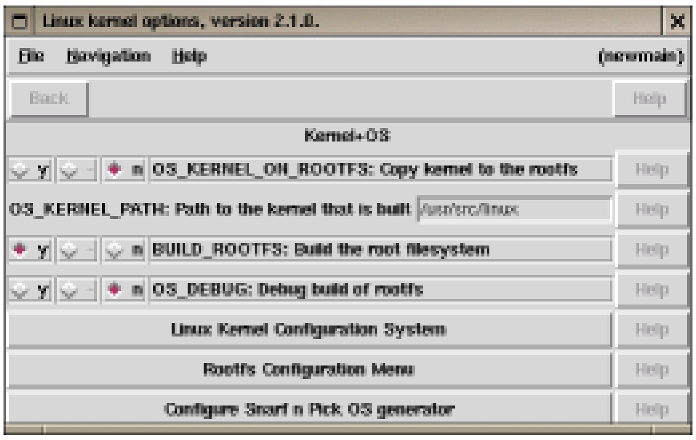
# Chapter 4. Embedded system generation

The final stage of the development of RTES is the generation of the embedded system. As we outlined in a previous chapter, the OF will permit us to generate the embedded system image.

Emdebian is a project that gives a tool for the configuration, setup and generation of embedded systems. It is a very intuitive and easy-to-use graphical tool. We will extend this tool to include the configuration, parameterization and compilation of the OC.

As we have seen, a component have the possibility to be activated or deactivated. But how can I activate or deactivate a specific component? The Emdebian-OCERA system allows us, according to the application needs, to activate or deactivate components, configure the kernel, and finally compile the kernel and components.

With this tool we will also be able to configure the target system: shells, services to include, root tree, users, binary files, booting, etc...


Linux kernel options, version 2.1.0.
File
Navigation
Help
(newmain)
Back
Help
Kernel+OS
Linux Kernel Configuration System
Rootfs Configuration Menu
Configure Snarf n Pick OS generator

y - [✓ n] OS_KERNEL_ON_ROOTFS: Copy kernel to the rootfs

OS_KERNEL_PATH: Path to the kernel that is built /usr/src/linux

[✓ y] - n BUILD_ROOTFS: Build the root filesystem

y - [✓ n] OS_DEBUG: Debug build of rootfs

**Figure 4-1. Emdebian Tool Screenshot**

Once we have the system configured and compiled, we will automatically generate the binary image to be copied into the target. This image will include the corresponding Linux kernel and system utilities, the OC chosen, the RT-Linux kernel if applicable and the real-time application developed.

We are still evaluating if it will be possible to deliver some tool to effectively copy the binary image into the target, through serial port (JTAG), or provide the kernel through the BOOTP protocol, etc...

# II. Components Specification

## Table of Contents

# Chapter 5. Kernel Components

## 5.1. Introduction

In the framework proposed in this project, the division of an application in hard/soft real-time parts is also used to structure the execution platform. Then, the execution platform is organised in two levels: a soft real-time level, where all soft real-time tasks will be allocated, and a hard real-time level, where critical tasks will be executed. Each of this execution levels is going to be managed by a different operating system kernel. The Linux kernel will control the execution of all soft real-time tasks and the non-real-time tasks, as graphical analysers, etc. Under the Linux kernel, and with a more strict control over the hardware, the RTLinux will execute all critical tasks of the developed application.

Next, the components organisation from the point of view internal or external developer is detailed.

## 5.2. Kernel Components Architecture

From the point of view of a developer that is building new kernel components for the OCERA project, the kernel components can be classified in two types, depending on where the source code of that component will be finally allocated.

- *Low-level kernel component*: This kind of component is highly related with the current kernel capabilities or internal algorithms, and therefore it requires to modify the current kernel source code in order to provide the new functionality or to improve an available one.

  This kind of component will be distributed in a patch-form and hopefully incorporated in the main stream of the implied kernel source.

- *High-level kernel component*: It only needs the current API of the kernel or an extended API offered by other kernel component in order to implement its new functionality. It does not require to modify the existing kernel source code or any low-level kernel component.

  The distribution of this kind of components is quite easier. It can be distributed as a separated source code tree, while the required API is ensured to be available in the corresponding kernel.

- *Application-level kernel component*: This kind of component uses the kernel API to provide a new service. It does not require to modify the existing kernel source code or any kernel component. The main characteristic of these components is they are implemented as application-level processes/threads, offering some kind of service to other processes/threads (as a kind of a classical UNIX daemon).

  This kind of components is quite easy of being distributed. It is also implemented as a separated source code tree and distributed in the same manner of high-level kernel components.

These kind of components will be developed for both kernels present in the execution platform: Linux and RTLinux. In fact, one of the goals of this project, at kernel level, is to provide the same API in Linux and RTLinux, from the real-time point of view.

The term high-level, low-level or application-level used to classify the components is not related with the complexity or criticality of the component. It only refers to the fact that requires or not to modify the current kernel source. This classification can be applied to soft and hard real-time components, being the low-level components for the Linux kernel probably the most complicated to develop and maintain, due to the complexity of this kernel.

## 5.3. Kernel Components Description

In this section, a description of the kernel components that will be developed and where they are situated in the kernel components architecture is presented.

The kernel components presented bellow will be developed mainly for the RTLinux kernel, because some of these services are already available in the Linux kernel.

### 5.3.1. Low-level kernel components

As it was described above, these components require the modification of the currently available kernels in order to extends their current API with new real-time functionalities.

- POSIX Signals
  - Description: This component will provide a mechanism by which a process or thread may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes.

    The Real-time signals is a deterministic signal extension that allows asynchronous signal notifications to an application to be queued without impacting compatibility with the existing signal functions.
  - Responsable: UPVLC
  - Validator: CEA
- POSIX Timers
  - Description: A mechanism that can notify a thread when the time as measured by a particular clock has reached or passed a specified value, or when a specified amount of time has passed.

    This component needs POSIX signals to notify the timer owner of its expiration and it has to modify the kernel scheduler to manage timer events.
  - Responsable: UPVLC
  - Validator: CEA
- POSIX Barriers
  - Description: Barriers, are defined in the advanced realtime POSIX (IEEE Std 1003.1-2001). A barrier is a simple and efficient synchronisation utility. A barrier can be implemented inefficiently by mean of a mutex and condition variables, but the proposed implementation will relay on special processor instructions to achieve low overhead.
  - Responsable: UPVLC
  - Validator: CEA
- POSIX Tracing facilities
  - Description: Recently, IEEE Std 1003.1-2001, POSIX provides a new debugging facility: POSIX tracing. POSIX trace implementation jointly with a tool to display and analyse the logged data will be a very useful tool for realtime developers.
  - Responsable: UPVLC
  - Validator: CTU

- Application-defined Scheduler support (ADS)
  - Description: It is an RTOS API proposal to allow the user application to define the scheduling algorithm for its system.

    This new API will be provide the capability of implementing new scheduling policies without modifying the kernel scheduler. Using this new functionality, an application thread can decide how other threads of the same process should be scheduled. It is also possible that multiple application schedulers coexist in the same real-time application.
  - Responsable: UPVLC
  - Validator: SSSA

These components provide basic mechanisms for process/thread synchronisation, timed execution and system monitoring. All of them are desirable to build real-time applications and to develop more powerfull real-time components.

### 5.3.2. High-level kernel components

These kind of components provide a new API that extends the real-time API of Linux and RTLinux kernels. As they does not require to modify the kernel source code, it can be provided as separated loadable modules, application libraries or incorporated into the kernel mainstream, if preferred.

- POSIX Message Queues
  - Description: The IEEE Std 1003.1-2001 specification defines a prioritised message passing facility for the realtime and embedded environments. Among others features, this message system will provides: prioritization of messages, asynchronous notification, and fixed size messages.

    This powerfull message passing facility is not available in any of the two kernels, Linux and RTLinux. This component should provide such facility using only the API provided by the kernel plus the API provided by the low-level kernel components, more precisely, using POSIX signals and timers.
  - Responsable: UPVLC
  - Validator: VT

- RTLinux Dynamic Memory Management
  - Description: The RTLinux kernel lacks of a dynamic memory manager, and therefore, hard real-time threads and drivers that should be allocated in the RTLinux kernel cannot use any kind of dynamic memory allocation. This is a very strong restriction in same cases.

    This component should provide a highly customisable and fully deterministic manager that allows hard real-time threads and low-level drivers to allocate memory dynamically.
  - Responsable: UPVLC and SSSA
  - Validator: CTU

- RTL-ADA Porting
  - Description: RTLinux applications can be programmed in C and C++ . Another important programming language for realtime systems is ADA. The ADA run time support has to be ported to RTLinux in order to run ADA programs. This component will provide such support in the RTLinux kernel.

- □ Responsable: UPVLC
- □ Validator: VT

These components and the low-level kernel components will be provide the required functionalities to develop more complex components, such QoS, communication and fault-tolerance components and to build reliable real-time embedded applications.

### 5.3.3. Application-level kernel components

This kind of components is not the typical kernel component, and its existence is reduced to a small set of threads that provide scheduling and monitoring support to the other threads. This kind of components will be mainly developed as a QoS and fault-tolerance components.

- Earliest Deadline first (EDF)
  - □ Description: The EDF is a basic scheduling algorithm with a solid theory background, mainly used in multimedia applications. This implementation relies on the Application-defined Scheduler component, which makes it almost independent of RTLinux code, and very easy to port to other OS.
  - □ Responsable: UPVLC
  - □ Validator: SSSA

- Constant Bandwidth Server (CBS)
  - □ Description: Using the facilities provided by the Application-defined Scheduler component, this component implements the CBS scheduling algorithm explained before. As was the case with the previous component, this implementation is highly portable since it do not modify the RTLinux executive.
  - □ Responsable: UPVLC
  - □ Validator: SSSA

Next figure shows the kernel components distributions inside the OCERA software architecture.

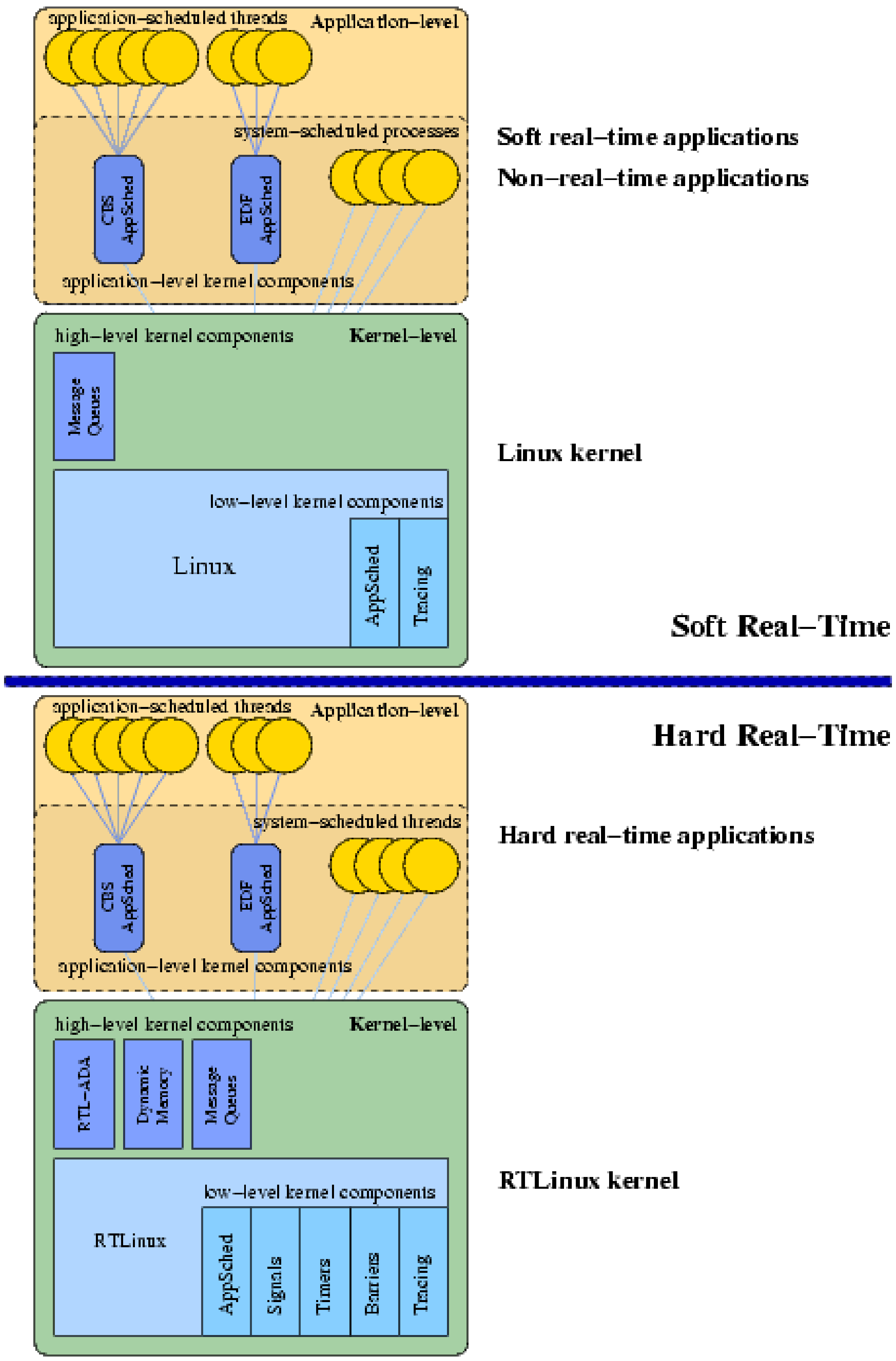

application–scheduled threads
Application–level
system–scheduled processes
CBS AppSched
EDF AppSched
application–level kernel components
Soft real–time applications
Non–real–time applications
high–level kernel components
Kernel–level
Message Queues
low–level kernel components
Linux
AppSched
Tracing
Linux kernel
Soft Real–Time
Hard Real–Time
application–scheduled threads
Application–level
system–scheduled threads
CBS AppSched
EDF AppSched
application–level kernel components
Hard real–time applications
high–level kernel components
Kernel–level
RTL–ADA
Dynamic Memory
Message Queues
low–level kernel components
RTLinux
AppSched
Signals
Timers
Barriers
Tracing
RTLinux kernel

**Figure 5-1. Kernel components**

# Chapter 6. Quality of Service Components

One of the goals of the OCERA QoS component is to provide a predictable Quality of Service level to both kernel level (RTLinux) and user level (regular Linux tasks). A correct resource allocation is fundamental to guarantee a predictable QoS, and in order to precisely allocating resources two requirements must be fulfilled:

1. Tasks must be scheduled with a *low latency*;
2. A *proper scheduling algorithm* must be used to guarantee real-time performance to tasks.

At the kernel level, RTLinux provides low latencies, and the extended scheduler developed by UPVLC can be used to guarantee the correct CPU allocation.

At the user level, a proper resource allocation can be obtained by combining techniques analysed in Deliverable D1.1 (the low-latency or kernel preemptability patches) and novel scheduling strategies. In a traditional RTLinux system, the Linux kernel is scheduled in background respect to all the real-time tasks, and as a result user-level tasks experience big latencies and non predictable delays. This problem can be solved by scheduling Linux as a real-time task, and not as a background one.

Since the Linux kernel is scheduled by the low-level flexible scheduler implemented in RTLinux, the guarantees provided by the two schedulers must be combined in a hierarchical fashion to control the QoS delivered to the application.

Resource Reservation techniques (described in Deliverable D3.3) are particularly useful for hierarchically combining the two schedulers, since they permit to reserve a fraction of the CPU time to the scheduled entities. Hence, this combination allows to reserve a fraction Bl of the total CPU time to the Linux kernel and then a fraction Bi of the Linux time to each user-level task. An approximate guarantees can intuitively be provided by multiplying the two fractions, and an user-level task is reserved a fraction of the total CPU time equal to about Bi * Bl [1].

Hence, the QoS component will be based on:

1. Preemptability or low-latency patches that permit to decrease the scheduling latency experienced by user level applications;
2. A high-resolution timers patch that increases the kernel timing accuracy;
3. A loadable kernel module that permits to schedule user-level tasks with a proper algorithm;
4. Mathematical tools that permit to combine the kernel-level and user-level guarantees.
5. A QoS Manager, which uses the patches and modules described above to properly allocate CPU to each application.

As a result, the QoS component will use features provided by the Real-Time Scheduling component, and will depend on it to implement hierarchical scheduling. Note, however, that the QoS component can be used also if the Real-Time Scheduling component is not used (i.e., in a non RTLinux-based system). In this case, the loadable scheduler provided by the QoS module will be the only real-time scheduler in the system, and all the real-time/time-sensitive tasks will run at user level.

As a final note, we would like to stress that the QoS component gives to the user the freedom to choose which time-sensitive tasks to run at kernel level, and which ones to run at user level, allowing the implementation of three different kinds of solutions:

1. A “pure RTLinux” solution, in which all the time-sensitive tasks run at kernel level;
2. A “pure user-level” solution, in which all the time-sensitive tasks run in user space;
3. A “mixed” solution, in which some time-sensitive tasks run at kernel level (as RTLinux tasks), and the other ones run at user level. OCERA will allow to easily move a task from one level to the other without having to change the application too much.

A description of the Resource management components follows.

**Low level Linux components**

- Generic Scheduler Patch
  - □ Description: It is a small patch for the Linux kernel that provides useful hooks to the Linux scheduler. These hooks will then be used by our scheduling module for implementing sophisticated real-time scheduling policies. This patch has to be minimally invasive for to limit the overhead and to minimise the need to upgrade it for new versions of the Linux kernel.
  - □ Responsable: SSSA
  - □ Validator: UPVLC

- Integration patch
  - □ Description: This patch will take into account the introduction of the "preemtpion patch" and "high resolution timers" in the Linux kernel. These new services are very useful for improving the responsiveness of time-sensitive applications in Linux: however, they are not compatible with the RTLinux patch. Since these new services are most likely to be introduced in the next Linux stable release, there is the need to take into account this problem by modifying the RTLinux executive. Therefore, this component will be a patch to the RTLinux executive that makes it compatible with the future versions of Linux.
  - □ Responsable: SSSA
  - □ Validator: UPVLC

**High level Linux components**

- Resource Reservation Scheduling module
  - □ Description: It will be a dynamically loadable module for the Linux kernel, that will provide a resource reservation scheduler for soft real-time tasks in the user space. It will be based on the Constant bandwidth Server. This module is the core of Workpackage 4, and it will provided in different versions during the course of the project: a first version will simply provide the CBS scheduler for uniprocessors; a second version will include mechanisms for reclaiming the spare capacity; a third version will take into account SMP (simmetric multi-processor systems).
  - □ Responsable: SSSA
  - □ Validator: UPVLC, CEA

- Quality of service Manager
  - □ Description: This component will provide a mechanism for identifying the temporal characteristics of a task and to adjust its scheduling parameters so to maximise its quality of service. It will be based on the concept of "feedback scheduler", that is a controller that measure the QoS experienced by the task and modify its parameters accordingly. It will be provided in two versions, as a dynamically loadable module and as a daemon process running in the Linux kernel (i.e. as a Linux application).
  - □ Responsable: SSSA
  - □ Validator: VT, UPVLC

**Linux applications**

- User API
  - □ Description: This component is a set of one or more libraries that will provide a convenient API to the user to access the Resource Management services. This API will be as similar as possible to the POSIX API provided by the RTLinux executive:

in this way, it will be possible to move a RTLinux thread from the hard real-time level to the soft real-time level, and vice versa, with little effort.

- □ Responsable: SSSA
- □ Validator: UPVLC

## Notes

1. The correct guarantee is a little bit more complex, but the intuitive product Bi * Bl gives an immediate idea of how to compose the guarantees. The correct and precise analysis will be developed in WP 4.

# Chapter 7. Components for fault-tolerance

## 7.1. Introduction

Fault-tolerance can be provided by the combine use of a set of methods and mechanisms starting from design and covering all the development process until run-time.

Targeted objectives of Fault-tolerance in the project are twofold : implementation of tools and mechanisms to support degraded mode management and dynamic reconfiguration of tasks (based on dynamic scheduling policies) on a local basis (one node) during the first phase of the project and, redundancy mechanisms to support dynamic reconfiguration in a distributed system in the second phase of the project.

In the project we will not consider that fault-tolerance can only be provided by isolated separate fault-tolerant components. The methodology we will adopt will be to build a fault-tolerant system using as far as possible the other OCERA components (scheduling, resource management and communication) and exploiting in an appropriate way the new high level facilities they will implement.

The Fault-Tolerance work package will thus not only provide some basic specific run-time FT components, but it will also provide other OCERA components with requirements for new features so that these components can contribute to the overall application fault-tolerance. It will also provide a methodology and associated supporting tools to help application users specify and implement fault-tolerance.

Once specialized, OCERA components will contribute to fault-tolerance provided a few additional features (such as temporal fault signaling), or specific configuration (within QoS scheduler) is undertaken, or specific protocol is implemented (in communication).

The main reasons for splitting objectives in two sets is related to the project planning . We have to reach rapidly a first objective so that a consistent set of components can be demonstrated at the end of the first phase. Local management of degraded modes can be achieved in a such a short period of time and still provide the basic building blocks for the more complete fault-tolerance set of mechanisms. Full implementation of fault-tolerance will require cooperation of almost every other OCERA component and will thus take time to specify precisely all the needed requirements and implement them, this could not reasonably reached within the first period of the project.

While most OCERA components will be run-time components, we have identified a need for design help tools related to fault-tolerance. These design help tools will be devoted first to provide the user to specify non-functional features for its application namely, temporal constraints, declare critical tasks, specify exception handling and alternative behaviors. It will support also the specification of mode-change. A second help will be related to configuration of the target system

Indeed, since fault-tolerance will require cooperation of several interrelated components, it is important that proper configuration and use of the various OCERA components is consistent. Moreover some additional code will have to be added to provide support for dynamic reconfiguration. So a building tool will be developed that will configure OCERA components, instantiate fault-tolerance policies and generate additional code to support fault-tolerance.

Fault-tolerance will thus be considered at three steps in the life-cycle of an application :

*Design*

- Specify dependability criteria (temporal and fault-tolerance) related to application
- Feasibility evaluation Off-line analysis

*Application building*

- Build tasks
- Configure schedulers
- Instantiate specific fault-tolerance mechanisms

*Runtime*

- Monitoring and control
- Collecting logs (for tuning)

## 7.2. Tools and components to be developed

### 7.2.1. Tools and components to be developed during first phase of the project

#### 7.2.1.1. Tools

*Design tool*

The design tool will help user for the

- Specification of tasks
- Specification of real-time constraints
- Specification of fault-tolerance constraints

*Building tool*

The building tool will permit to

- Configure OCERA components
- Instantiate FT components
- Produce additional tasks and/or wrap user code

Starting from information gathered with design tool

#### 7.2.1.2. Components

The specific fault-tolerance components that will be developed within the first period of the project are dispatched over the user level and the kernel level.

*User-level*

*AFT-monitors*

The Application FT-monitor will decide of dynamic reconfiguration of tasks on abnormal situations. It will receive notification of errors from the kernel level and will get information on load from the QoS scheduler. Depending on the situation, it will apply degraded mode management policies in order to keep the system in a safe state. For that it will ask the kernel level to stop certain tasks and ask the QoS scheduler to reschedule the new configuration of tasks. Dynamic behavior change can also be activated by application on detection of alarm conditions on certain tasks.

- First implementation will provide for emergency stop
- Second implementation will provide for degraded mode management
- Third implementation will provide for dynamic mode management depending on load

*Kernel level*

*FT-controller*

The FT controller will collect low level information on tasks progress and survey their lifeliness. It will signal abnormal behaviors detected and possibly activate emergency tasks.

#### 7.2.1.3. Interactions with other OCERA components

In this first step, active collaboration with the development of scheduling and QoS components in WP4 and WP6 will be necessary.

WP4 will provide error signaling and deadline miss at kernel level

WP5 will provide high level dynamic scheduling necessary to implement dynamic reconfiguration, it will also provide information about anticipated scheduling miss so that dynamic reconfiguration can alleviate load.

### 7.2.2. Tools and components to be developed during second phase of the project

#### 7.2.2.1. Tools

*Design tool*

Functioanalities will be added in the design tool :

- Specification of fault-tolerance constraints for redundancy

*Building tool*

The building tool will be extended in order to permit the implementation of distributed redundant tasks. It will also instanciate communication controllers to be developed in common with the OCERA communication components

- Code generation for redundancy management will be added

#### 7.2.2.2. Components

*User-level*

The Application FT-monitor will be enriched and a new component a task redundancy manager will be added

*AFT-monitors*

- 4th implementation will provide for dynamic redundancy management (activation or deactivation of redundancy for certain tasks depending on workload)

*Task Redundancy Manager*

- The task redundancy manager will monitor redundancy management. It will synchronize, activate and deactivate replicas. Passive redundancy will be implemented

*Kernel level*

*FT-controller*

The FT controller will collect low level information on tasks progress and survey their lifeliness. It will also provide reflexive information on the tasks implementing fault-tolerant mechanisms.

*TaskReplicaAgent*

This component will locally monitor interactions of a local replica of a task with the Task Redundancy manager. It will operate chekpointing, and local activation / deactivation of a replica.

#### 7.2.2.3. Interactions with other OCERA components

The implementation of redundancy mechanisms requires that data synchronisation over a distributed network can be achieved. This implies either a specific design approach

(Time Triggered Systems) or the implementation of specific distributed synchronisation protocols between replicas.

In this second step active collaboration with the development of communications components in WP7 will be necessary in addition to already existing collaboration with other components. In particular fail-safe communications will have to be implemented and a checkpointing algorithm will have to be chosen and implemented. A temporal synchronization model will have to be defined in collaboration with WP4.

## 7.3. Components and tools descriptions

A brief description of each component is given in this section. Only general overview is given since fault-tolerance is deeply inter-related with other components. This interaction will be analysed further within the WP6 workpackage but is still on-going.

**Components at the Low and High level RTLinux**

At this level no fault-tolerant component will be specifically developed but the POSIX Tracing, POSIX Signals, POSIX Timers and Application Defined Scheduling components described in the Scheduling section will contribute to fault-tolerance thanks to the logging and signalling facilities they will offer.

**Components at the Application-level RTLinux**

- FT controller
  - □ Description: The FT controller will be the low level module that will permit transfer of FT information from the RT level to the user level and that will activate emergency actions when required. In connection with the RT Kernel it will detect fail silent situations through watchdogs and transmit deadline miss to the Application FT monitor.
  - □ Responsable: CEA
  - □ Validator: UPVLC

- Application FT monitor
  - □ Description: The AFT monitor will consist of a module that will memorize Tasks Information and define reconfiguration strategies. It will collect information from both RT Scheduler and QoS Scheduler. It might also receive alarms from applications.It will decide of reconfiguration and inform RT scheduler by stopping tasks and QoS Scheduler by giving a new tasks set (this set will possibly include already running tasks)
  - □ Responsable: CEA
  - □ Validator: UPVLC

- Task Replica Manager
  - □ Description: The task replica manager is a low level module that is in charge of synchronization and communication of the replica during checkpointing. It will also detect possible communication failure
  - □ Responsable: CEA
  - □ Validator: SSSA

- Task Redundancy manager
  - □ Description: The task redundancy manager will monitor redundancy of a cluster of replicas, decide when to activate or deactivate a replica.
  - □ Responsable: CEA
  - □ Validator: SSSA

**Components at the Application-level Linux**

- Design tool
  - ☐ Description: The design tool whose goal is to permit the expression of non-functional features from the user will be developed. A first step will consist in defining a description language in order to specify explicitly timing characteristics and constraints of tasks, possible alternatives for tasks, actions to be done on temporal faults, on errors. The possibility to specify tasks graphs will be offered. Support for imprecise computation will be offered. A way to specify critical tasks requiring redundancy will also be defined. The tool will permit to gather this information along with information on mapping requirements for tasks. Ways of specifying modes at task and application level will be considered. A possible result of this language definition task might be a UML profile for fault-tolerance. The acquisition tool itself will be developed using standard GUI programming tool.
  - ☐ Responsable: CEA
  - ☐ Validator: CTU

- Buiding tool
  - ☐ Description: The building tool will use information gathered by the design tool and configure the OCERA platform, it will provide tasks information to kernel schedulerand QoS Scheduler, instantiate FT mechanisms (AFT monitor and FT-controllers), and adapt tasks code. When redundancy will be tackled it will produce code for tasks duplication and checkpointing mechanisms.
  - ☐ Responsable: CEA
  - ☐ Validator: VT

# Chapter 8. Communications Components

## 8.1. The Real-Time Ethernet Architecture

### 8.1.1. Preface

With the explosion of the Internet, the TCP/UDP/IP protocol suite has become the underlying framework upon which all Ethernet communications are built. Their success attests to the generality and power of these protocols. However, these transport-level protocols are too low level to be used directly by any but the simplest applications. Consequently, higher-level protocols such as HTTP, FTP, DHCP, DCE, RTP, DCOM, and CORBA have emerged. Each of these protocols provides well-tuned functionality for specific purposes or application domains, but none of them is suitable for real-time distributed applications, where developers need some methods to control data traffic on bus. For example, none offers deterministic communications, time-aware notifications, heartbeats, transparent hot-swap substitution, or quality of service control. Modification of publish-subscribe protocol (Real-Time Publish-Subscribe) adds parameters, that offers application developers an easy way to manage communication on bus with different deadline requirements.

### 8.1.2. Communication architectures

Distributed application developers have several choices for writing communication. There are three main communication architectures (Point-to-Point, Client-Server, Publish-Subscribe).

**Client-Server (CS) :** data exchange is done between group of client nodes and one server node. It works well when all data are located on central server (database applications). CS architecture is inefficient when data are produced by many nodes and are consumed by multiple nodes (two transactions are needed), which is typical for distributed application.
**Publish-Subscribe (PS) :** nodes "subscribe" to data they need and "publish" data they produce. Messages are transferred directly between communication nodes. Data are beeing sent without exact destination address, they are received by all nodes and each node decides itself (with respect to topic identification) whether it is interested in this message or not. PS architecture not suited for request/response traffic, such as file transfers.

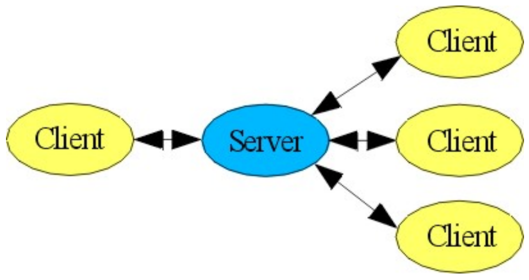


**Figure 8-1. Client-server model**

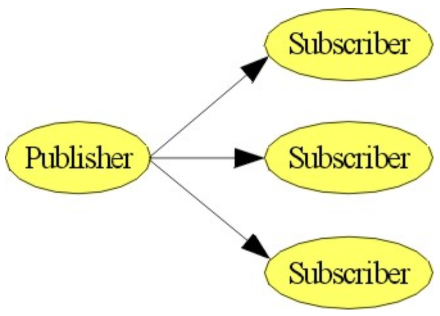


**Figure 8-2. Publish-subscribe model**

### 8.1.3. Real-Time Publish-Subscribe (RTPS)

Real-time applications require more functionality then the one provided by traditional publish-subscribe architectures. RTPS adds publication and subscription timing parameters and properties so the developer can control different types of data flows and achieve their application's performance and reliability goals.

**Table 8-1. Publication parameters**

| **parameter** | **description** |
|---|---|
| topic | A string which uniquely identifies the issues to be distributed. |
| type | A string which uniquely identifies the issues's data format. |
| strength | This value allows to arbitrate among issues of the same topic sent by multiple publishers. It express the relative priority of one publisher to another. A primary publisher would have a higher strength than a secondary publisher. |
| persistence | specifies duration of time for which an issue is valid after it is published. |

**Table 8-2. Subscription parameters**

| **parameter** | **description** |
|---|---|
| topic | A string which uniquely identifies the issues to be received. |
| type | A string which uniquely identifies the issues's data format. |
| minimum separation | Fastest rate at which issues should be sent by the ORTE network stack |
| deadline | Time period after which the ORTE stack should notify the subscriber if no issues have been received. |

Figure 8-3 illustrates how RTPS uses the deadline, minimum separation, strength and persistence properties to provide network communication for real-time applications and simplify system design. Features shown on the figure are:

**Subscription Issue data flow:** If a new issue does not arrive by the deadline, the application is notified.

**Redundant Issue hot-swap:** During persistance period ORTE stack accepts issues from publiccations with equal or higher strength. After persistance, it accepts the first issue, regardless of the publication's strength.

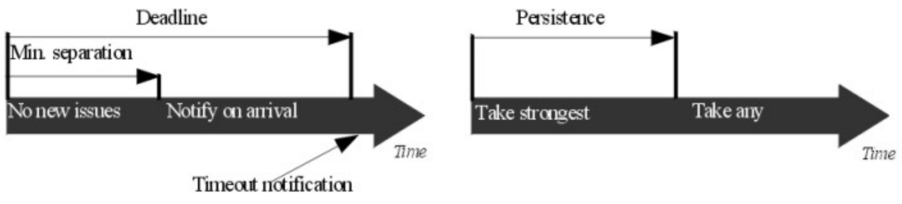


**Figure 8-3. RTPS timing parameters**

### 8.1.4. ORTE architecture

The OCERA Real-Time Ethernet (ORTE) will be open source implementation of RTPS communication protocol. This protocol is being to submit to IETF as an informational RFC and has been adopted by the IDA group. Figure 8-4 shows the network stack layers. Non Real-Time applications, which are using standard protocols such as HTTP, FTP, DCOM etc., are running on top of standard TCP or UDP stack. ORTE is new application layer protocol, which is build on top of standard UDP stack. Since there are many TCP/IP stack implementations under many operating systems and ORTE protocol does not have any other special HW/SW requirements, it should be easily ported to many HW/SW target platforms. It doesn't use or require TCP, so it retains control of timing and reliability.

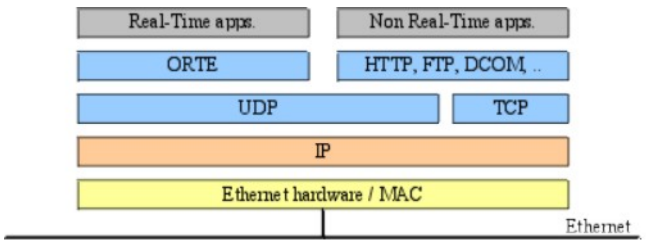


**Figure 8-4. ORTE layers**

The ORTE is composed from three main components, as shown on Figure 8-5:

**Database:** stores parameters describing both local as well as remote node's objects.
**Processes:** perform message processing, serialization/deserialization and communication between objects.
**API:** application interface.

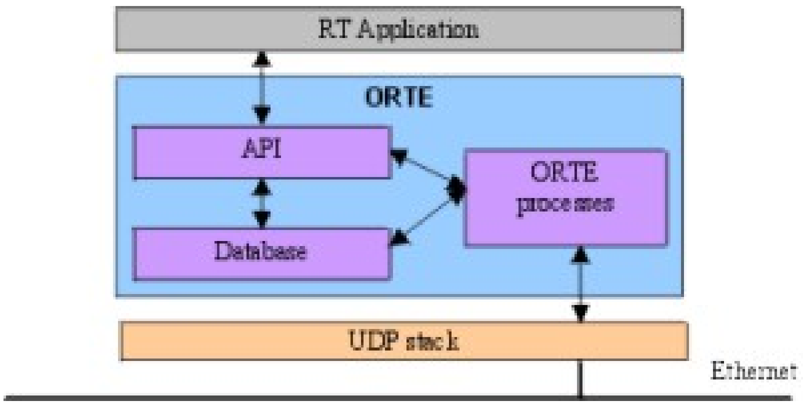


**Figure 8-5. ORTE structure**

The RTPS protocol is implemented as a set of objects. Objects are of the following types:

- Manager (M)
- ManagedApplication (MA)

- Services
- Writers (Publication, CSTWriter)
- Readers (Subscription, CSTReader)

A Manager is a special object that facilitates the automatic discovery of other Managers. There is one Manager on each participating network node. A ManagedApplication is an applciation that is managed by one or more Managers. The Publication is used to publish issues to matching Subscription. The CSTWriter and CSTReader are the equivalent of the Publication and Subscription, respectively, but are used solely for the state-synchronization protocol. Each object on network is characterized by GUID (Globally Unique Id).

The RTPS protocol uses five logical messages:

**ISSUE:** Contains the application's user data. ISSUES are sent by Publications to one or more Subscriptions.
**VAR:** Contain information about attributes of state of objects.
**HEARTBEAT (HB):** Describes the information which is available in a writer.
**GAP:** Describes information which is no more relevant to readers.
**ACK:** Provides information on the state of a reader to a writer.

Each of these logical messages are sent between specific readers and writers as follows:

**Publication to subscription(s):** ISSUE, HEARTBEAT
**Subscription to publication:** ACK
**CSTWriter to a CSTReader:** VAR, GAP, HEARTBEAT
**CSTReader to a CSTWriter:** ACK

## 8.1.5. Database implementation

Status of all objects is stored in ORTE database. The database is set data structures containing parameters of all known objects. There are not only data describing local node's objects, but also description of all known remote nodes' objects. Since access to this database should be granted to both ORTE stack processes as well as to applications, there should be implemented some access control algorythm allowing concurrent access. Applications use an API call to access the database.

Database contains two types of information. The first type is description of all known objects without any specification of relationship among them. This information is stored using a binary tree structure with object's GUID used as the key. The second type of information describes relationship among objects. Such relationship is for example association between Manager and ManagedApplications or between publishers and subscribers. This information is stored as a (double) linked list of GUID of objects which belong to certain owner. As an example, such owner is a publisher, the linked list contains list of GUIDs of its subscribers. The communication objects can be represented as the branch of a tree. Each of the nodes in the branch and its descendants represents an element of the complex data-structure.

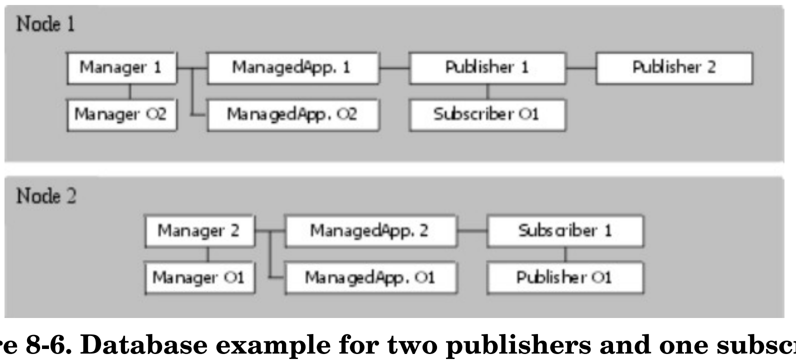


**Figure 8-6. Database example for two publishers and one subscriber**

Example of content of a database is shown on Figure 8-6. There are two nodes `Node 1` and `Node 2`. The figure shows content of database on both these nodes. The situation on `Node 1` is as follow:

- there is local manager `Manager 1`
- `Manager O2` is local copy of parameters describing `Manager 2`, which is located on remote node `Node 2`
- there is local managed application `ManagedApp 1`
- there is local copy of remote application `ManagedApp O2`
- there are publishers `Publisher 1` and `Publisher 2`
- there is remote subscriber `Subscriber O1` which is subscribed to data published by `Publisher 1`
- there is currently no subscription to `Publisher 2`

Situation in node 2's database is similar.

Note: letter O, which is placed before numerical index of an object means, that this database object represents object which belongs to other node than local. For example `node 1`'s object `ManagedApp O2` is local representation of `node 2`'s object `ManagedApp 2`.

## 8.1.6. ORTE processes implementation

The ORTE system spawns several processes during its lifetime as shown on Figure 8-7. Generaly there are two groups of processes. The first group represents processes involved in real-time application's data exchange between local and remote nodes. These processes are responsible for publishing of local application's data if there are any remote subscribers and also for receiving of remote application's data if there are any local subscribers. Another group are processes involved in network management.

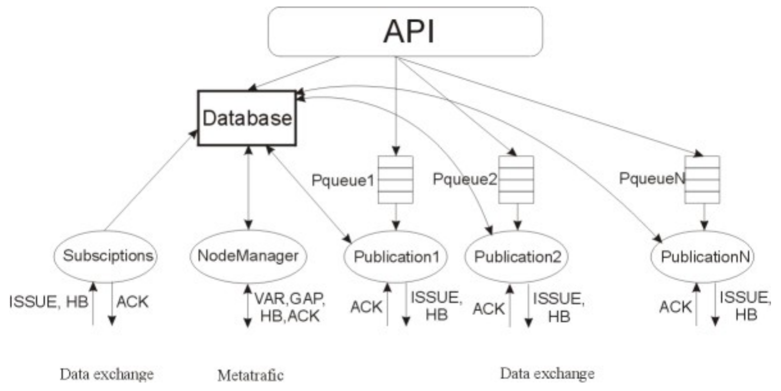


**Figure 8-7. ORTE processes**

Process `Subscription` which is responsible for processing of all local subscriptions, i.e. it receives all application data from all remote publishers to which local applications want to be subscribed. It performs whole processing of received data including storing of all data changes into database and, if requested by application, also calling an application's callback functions. This process can listen on more UDP ports where each port belongs to one local subscription, or on a single UDP port, which is common to all subscriptions. In any case, this process will never listen on default NDDS port 7400.

Processes `Publication1, Publication2, ..., PublicationN` are responsible for proper publication of local data to all its subscribers across the network. Each process is associated with queue `Pqueue1, Pqueue2,... PqueueN`. Data changes are stored (using proper API call) into certain queue. Process which is associated with this queue reads its content and sends ISSUE message to all subscribers and waits for their ACK. Each process in this group use its own UDP port, they will never use default NDDS port 7400.

Process `NodeManagement` is responsible for whole management traffic. It is the only process which uses default NDDS port. Since management traffic does not have real-time requirements, it is designed as single process which processes both incomming as well as outgoing requests. During evaluating phase it may be split to two separate processes, one for incomming requests and second for outgoing requests.

### 8.1.7. API implementation

First implementation will be designed like one application running in user space using standard Linux 2.4 kernel. Since the main purpose of this version will be to test this implementation against another commercially available implementation, there will be no any standard API provided. The real-time capabilities will not be focused during this phase too. The next version will be written as a Linux kernel module. Interface between this module and an application will use standard `ioctl` function and read/write operation. The function will be divided into three categories:

- Administration - create and destroy database (InitDB, DestDB, SetParamDB, GetParamDB,...)
- Publish - create and destroy publishers, sending data (CreatePub, DestroyPub, SndData,...)
- Subscribe - create and destroy subscribers, receiving data (CreateSub, DestroySub, RecDataPoll, @RecDataCallBack, ...)

There are two type of subscribers - polling or callback. Polling subscriberes have to ask ORTE layer by `RecDataPoll` API call whether new data are available. Callback subscriber creates an callback function, which will be passed to ORTE stack. ORTE stack will call this function every time when new data will be available.

## 8.2. RT-CANopen architecture

### 8.2.1. Overview

#### 8.2.1.1. List of abbreviations

**List of abbreviations used in the document**

CAN
: Controller Area network

COB
: Communication object (CAN message). A unit of transportation in a CAN network. Data must be sent across a network inside a COB.

COB-ID
: COB-Identifier.

  Unique ID of the COB. The identifier determines the priority of the COB in the MAC sub-layer too.

EDS
: Electronic Data Sheet

FSM*
: Finite State Machine

  CANopen FSM in this article means a state automat processing CAN messages.

HDS*
: Handlers Definition Sheet

NMT

Network ManagemenT

COBs designated for network management ie. initialize, start, stop nodes etc.; this service is implemented according to master-slave concept.

OD

Object Dictionary

Common representation of device parameters, process variables, configuration, communication settings and device data types. Accesible as SDO objects from CANopen network. The textual description of OD is called EDS file.

PDO

Process Data Object

CANopen objects designed for real-time process data exchange.

RPDO

Receive PDO

PDO received by device in order to set appropriate PDO mapped objects value.

RTR

Remote Transmission Request

CAN message to initiate RPDO object sending.

SDO

Service Data Object

CANopen service objects designated for manipulation with slaves object dictionaries.

SFO

Special Function Object

Special purpose messages ie. SYNC, time stamp, emergency, node/life guarding, boot-up objects.

TPDO

Transmit PDO

PDO transmitted by device when device specific event occurs (ie. timer or object value is changed) or as a response to the RTR or after SYNC object.

VCA*

Virtual RT-CAN API

* - this abreviations are not standardised yet and they are valid only in OCERA project

#### 8.2.1.2. Goal of the work

The goal of our team work is to develop set of software tools and real-time modules/applications necessary for building of CANopen network solutions. We also want to make tools to facilitate the CANopen slaves and master configuration using the textual EDS files and the bus monitoring. By developing the CANopen real-time driver, software slave and master we will get basic blocks to connect to working CANopen network. Any industrial CANopen device can be plugged to this network.

To reach that goal, the next points have to be fulfilled.

- to develop CAN API called the virtual CAN API (VCA), usable for user space threads and also for the RT-Linux ones.
- to develop a RT-Linux CAN driver providing VCA interface to the application RT-Linux threads and also to the user space ones.
- to develop software CANopen slave.
- to develop software CANopen master.
- to develop EDS parser tool for setting software CANopen master and slave parameters in the convenient way.

## 8.2.2. Architecture

### 8.2.2.1. Virtual CAN API (VCA)

The virtual CAN API is the interface to connect application thread either with the CAN hardware card or with other software layers substituting CAN bus. The application thread can be in RT_Linux/kernel space or in the user space.

#### 8.2.2.1.1. RT-CAN driver

RT-CAN driver resides in the RT-Linux address space as the real-time thread. It means, that any CAN message can be handled in a deterministic time horizon.

Two interfaces are exported. First one is the set of handlers needed for device /dev/canxx. The second one is the *Virtual CAN API (VCA)* designed for driver communication with POSIX threads.

As you can see in the following figures, RT-CAN driver contains the table of buffers for transmitted and received CAN messages. This buffers can be considered, from the CANopen point of view, to serve as the RPDO and TPDO buffers. The COB-ID mask can be registered to each buffer. Driver also contains the FIFO for receiving and transmitting the CAN messages. This FIFO will be used for the SDO communication.

Next figures show possible usage of the VCA.

#### 8.2.2.1.2. RT-Linux space VCA usage

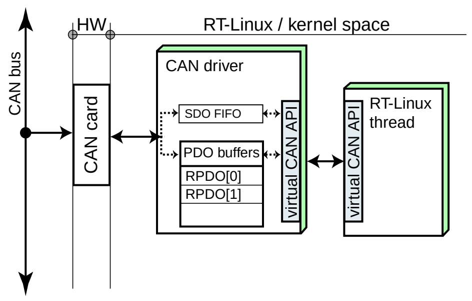


**Figure 8-8. RT-Linux space VCA usage**

Real-time application thread uses directly VCA of RT-CAN driver for the communication with a CAN bus.

#### 8.2.2.1.3. User space VCA usage

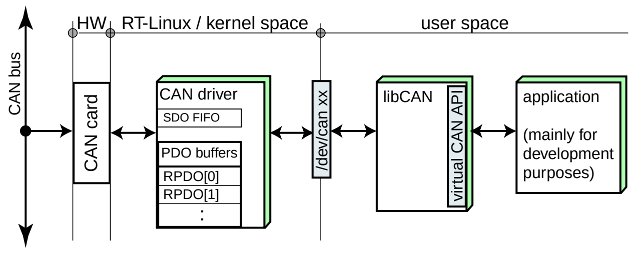


**Figure 8-9. User space VCA usage**

In Figure 8-9 the application thread and CAN driver reside in different address spaces. To connect kernel space and user space, we use device /dev/can xx. The applications which will not use VCA can use directly /dev/can and the standard set of I/O operations (read, write, ioctl). A libCAN is needed to grant the same API to user space and kernel space threads. This approach supports portability of application threads between the kernel and user spaces on the source code level.

#### 8.2.2.1.4. CAN bus substituting layer with VCA

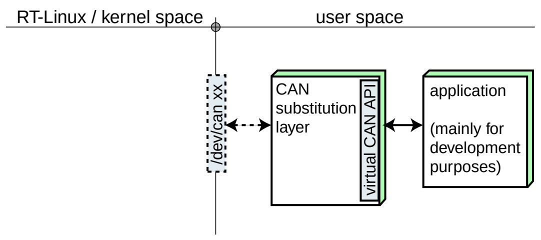


**Figure 8-10. CAN bus substituting layer with VCA**

On Figure 8-10 we can see option, that gives us an opportunity to exploit the VCA even if no real CAN hardware is used. This solution is designed mainly for development purposes and testing. The dashed line in the figure means that CAN substitution layer can optionally operate on some other device than /dev/can (ie. Ethernet). CAN substitution layer can be also a kind of user space application providing the VCA. This can be very useful during development or for educational purposes.

### 8.2.2.2. RT-CANopen slave

#### 8.2.2.2.1. Overview

RT-CANopen slave (only *slave* or *CANopen slave* will be used instead in rest of this section) is the software solution based on the hardware, RT-CANopen FSM threads, EDS file and HDS file (see Figure 8-11).

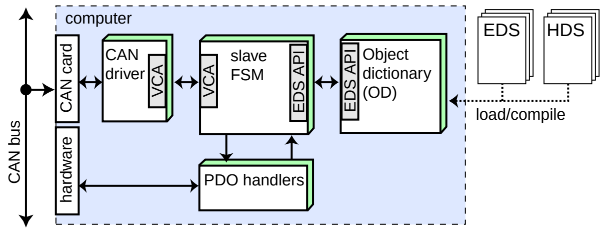


**Figure 8-11. RT-CANopen slave architecture**

### CANopen slave components description

**Slave FSM:** Slave FSM means set of RT-Linux threads providing PDO and SDO communication via CAN driver VCA. Slave FSM also calls appropriate handler PDO communication and looks into the slave's object dictionary in case of SDO request.

**PDO handlers:** User written module containing handlers for reading/writing PDO mapped object data from/to hardware.

**EDS:** EDS means the *Electronic Data Sheet*, text file describing all objects in the slave object dictionary. EDS is parsed in order to create the slave OD.

**HDS:** HDS means the *Handler Definition Sheet*, text file describing the linking PDO's COB-ID with required handler in order to grant correspondence between the CANopen object value and technological process data from the hardware. For example a thermometer with the analog output connected to PC A/D convertor card needs handler which reads temperature from the card output port and gives it to the FSM. The slave designer have to write this handler code while the FSM source code remains always the same, OCERA written.

#### 8.2.2.2.2. User space OD slave architecture

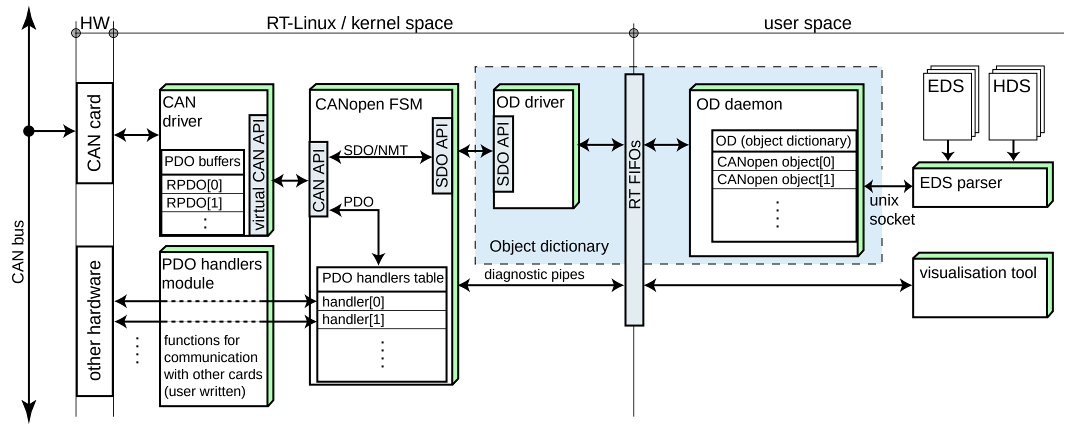


**Figure 8-12. User space OD slave architecture**

As can be seen on figure above CAN driver sends the CAN messages to CANopen FSM via VCA. FSM handles messages of two main categories, process data (PDO) and service data (SDO, NMT, SFO).

The process data (PDO objects) are handled separately of the service ones. Slave FSM exploits CAN driver message buffers as the buffers for the slave PDOs. This approach is necessary because some CAN chips have such buffers integrated. On the other hand this can speed up PDO object handling. Slave FSM role lies in updating this buffers after device specific event such is timer event or process object value change. The CAN driver sends objects from its buffers when needed (after SYNC object or as a response to RTR object). Consequently slave FSM has to read this buffers after WPDO object arrival.

The PDO handler module is used to synchronize PDO mapped objects values and real world data examined or set by computer hardware. Every PDO mapped object has assigned its reading and writing routine called PDO handler. These handlers are written by control system developer in order to fit general FSM concept to specific hardware and real world process.

Some of SFO are handled directly by CAN driver. Such objects are the SYNC or RTR frames

Other objects (SDO, NMT) are sent through the SDO API to OD driver. OD driver is responsible for all object dictionary manipulations, that means getting and setting object values. If the PDO mapping change occurs due to SDO object processing, OD driver informs slave FSM (via SDO API) to correct PDO handlers table to reflect PDO mapping status in OD properly.

OD driver communicates with OD daemon, which resides in the user space, through RT FIFOs. OD daemon offers set of primitives to provide basic manipulations with OD like get/set object value, add object, delete object etc. OD daemon also owns slave OD in its memory space.

Slave OD can be loaded onto OD daemon memory by EDS parser. EDS parser is the graphical front end to the OD daemon connected with it via Unix socket. This gives us the opportunity to control the daemon and slave OD remotely using TCP/IP. EDS parser is also responsible to read HDS and make appropriate changes in daemon OD and also (via daemon SDO API) in slave FSM PDO handler table. This ensures that the proper handlers will be called for certain PDO objects.

For diagnostic and visualization purposes the diagnostic couple of pipes exists. This pipes can be connected to the diagnostic and monitoring application.

#### 8.2.2.2.3. Kernel space OD slave architecture

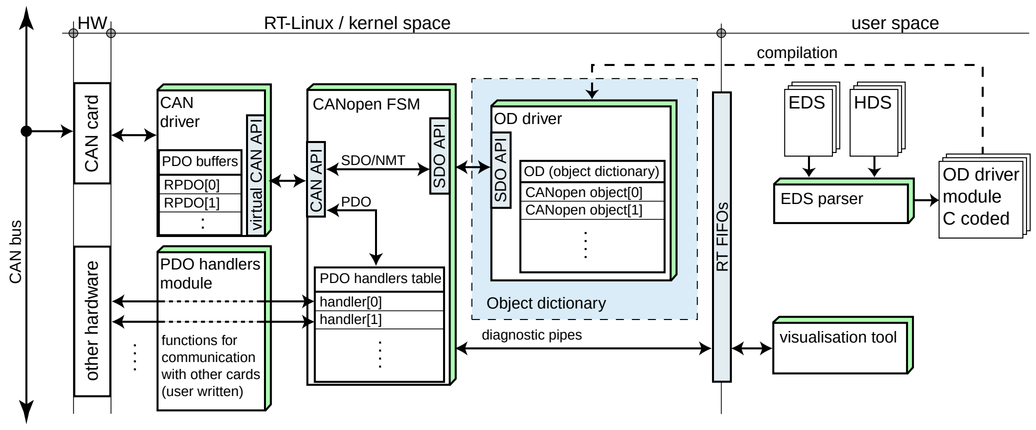


**Figure 8-13. Kernel space OD slave architecture**

This architecture is very similar to the previous one. The main difference lies in OD position which is a part of OD driver module now. Every other part of slave remain the same. OD driver module is compiled from source code generated by EDS parser from the slave EDS, HDS and the empty OD driver template.

**Benefits of the kernel space solution**

- Faster SDO object processing.
- Slave does not need user space applications to work properly.
- Slave can be implemented to other POSIX compliant real-time OS like RTEMS.
- Suitable for CANopen slave realization in embedded systems.

**Disadvantages of the kernel space solution**

- OD is static, no objects can be added or removed.
- EDS parser can not explore OD any more. The diagnostic pipe has to be used for that purpose and all the information must be communicated through SDO API and slave FSM.

## 8.2.2.3. RT-CANopen master architecture

RT-CANopen master architecture is very similar to the user space slave one. CAN driver, FSM, OD driver and OD daemon are the same. New blocks, PDO processor and PDO object table are introduced and OD daemon contains OD of all slaves connected till now.

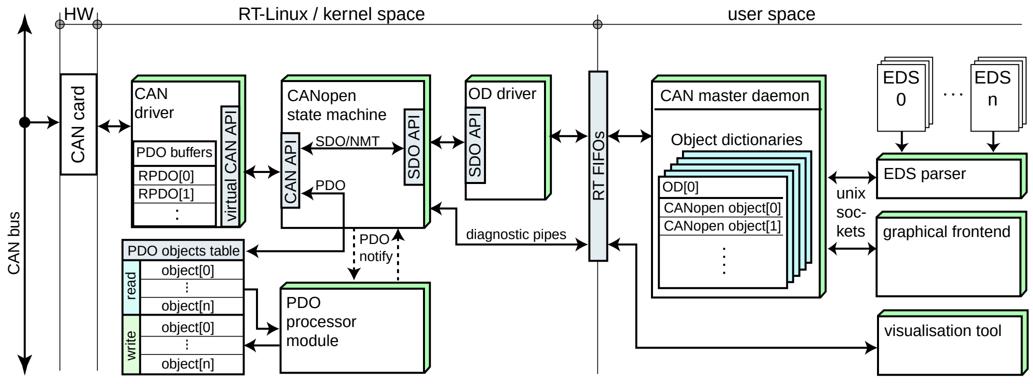


**Figure 8-14. RT-CANopen master architecture**

PDO objects table is a memory mirror for transmitted and received PDOs. When a new RPDO comes, it is written into the table and PDO processor is notified by master FSM about this event. On the contrary, when PDO processor updates some object in the write part of PDO table, the FSM should be notified to allow it to transmit object change across the network.

PDO processor is an user written set of functions designated for processing objects from read part of PDO table and generating new value of objects in write part of PDO table. New generated write objects can be sent across the CAN, if the processor notifies master FSM. This way the RPDO-TPDO mapping rules or control algorithms can be realized.

We expect to design PDO processor as a special PDO handler (from slave FSM point of view). This approach gives us the opportunity to have the same FSM for slave and master.

The master OD daemon holds one or more slave EDS. That means, that the slave OD daemon is a special case of the master one. Thanks to this generalization we can have also one code for the master and slave OD daemon.

When we look at the paragraphs above, we can see, that the User space CANopen slave and the CANopen master share the same code, which can be any time configured to run as a master or as a slave.

## 8.2.3. Conclusion

The RT-CANopen architecture design is based upon 4 basic elements.

The Virtual CAN API introduces either real CAN network or any substitution of that sublayer to the application threads. Application threads can reside in user space or in kernel/RT-Linux space.

The CANopen FSM is an kernel space state machine connected to VCA, which can handle the process data (PDOs) and service data (SDO, NMT, SFO). The PDOs are handled by PDO handler functions. The service data objects are resent through a SDO API to OD driver for other processing and its response is sent back if necessary.

The OD driver manages a device OD. It is also able to communicate with FSM via EDS API to provide it information about every particular object. The OD driver can be realized either whole in the kernel space or partially in the user space (OD daemon).

The EDS parser serves for processing EDS and configuration files. Processed data are loaded into OD daemon (master, user space slave). EDS parser can also generate OD driver module source code in C (kernel slave concept).

This philosophy gives us the opportunity to reuse common elements to build the CANopen master or slave even to define it in process of configuration.

## Bibliography

[CANOpen] CiA *CANopen - Application Layer and Communication Profile, CiA Draft Standard 301* CAN in Automation e. V. 1 June 2000

[[EDS]] CiA, *CANopen: Electronic Data Sheet Specification for CANopen, CiA Draft Standard Proposal 306*, CAN in Automation e. V., 31 May 2000.

[[CANBus]] H. Boterenbrood, *CANopen: high-level protocol for CAN-bus*, NIKHEF, Amsterdam, 20 Mar 2000.

## 8.3. Verification of Distrubuted Systems

### 8.3.1. Problem statement

Figure 8-15 illustrates mayor topic of verification of distributed systems . Figure shows a control system consisting of *n* independent processors and communication bus with general communication protocol. Let us consider the parallel machines running applications in the real-time operating system (RTOS) environment and further let us consider the communication protocol behaving in Real-time manner. The crucial problem is whether the general real-time control system (RTCS) [Buttazzo97] behaves in this manner too.

For simplicity we suppose that each processor is running only one task so there is no problem with scheduling. And further let us suppose the communication protocol to be deterministic (CAN, Token ring etc.).

To resolve the above mentioned problem we use a mathematics formalisms based on Petri Nets and communicating automata theory. Especially we use the tools that have been already used to solve similar problems (PEP tool, UPAAL etc.). The tools are able to model concurrent systems and mainly verify it. To reach the mentioned goal we need to create model of RTCS. The work includes the model of the communication protocol, the model of RTOS and the model of the application. After that we create model of RTCS. Finally we need correctly define dangerous system properties that can assign the system to undesirable states (Deadlock, missing deadline etc.).

Such model plus properties to be verified will be passed to the verification tools, exactly verified with included methods (SPIN algorithm, temporal logic, etc.). Due to the model checking we can say that the system can avoid the states.

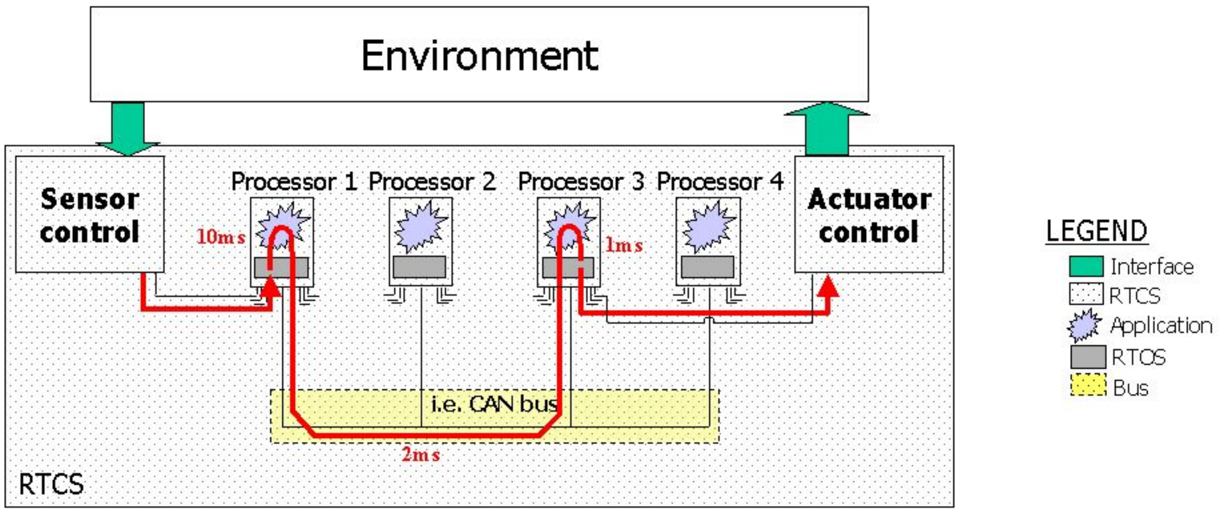


**Figure 8-15. Real time control system structure with denotation of computation/communication times**

### 8.3.1.1. Modelling of communication protocol

We are modelling and verifying a communicating system we have to understand the idea of communication protocol design [Holzmann91]. At the start of the design we must answer the next five questions:

- What service should be provided by the protocol?
- What assumptions are about the environment in which the protocol is executed?
- What vocabulary of messages should be used to implement the protocol?
- What encoding (format) of each message in the vocabulary should be used?
- What procedure rules are guarding the consistency of message exchanges?

If we satisfactorily answer the questions we can suggest concrete structure of communication protocol. During solving of the question problem we have to ask for kind of properties of the protocol. For example type of transmission of bits over a physical circuit, error-control problem, flow-control problem and so on. It is evident that the most complicated design problem the most difficult his resolving. Partitioning the problem to small subproblems is reason that tell us common sense. The subproblems can be either easy to solve or they have been solved before. One of them can be ISO/OSI model of protocol layers, encoding of messages (e.i. CRC) or access control at physical media (i.e. MAC).

In OCERA project we are interested in CAN bus communication protocol due to its real-time properties (see "Survey on RTOS"). The protocol will be modelled and it will be included to RTCS as communication part.

#### 8.3.1.2. Model of RTOS

The term real-time is frequently used in many application fields. The definitions [Buttazzo97] adopted by us is that the main difference between a real-time and non-real-time task is that the real-time is characterised by a deadline, which is the maximum time within which it must complete its execution. In critical applications, a result produced after the deadline is not late but wrong! RTOS is operating system in manner of the definition.

We need RTOS model in this point of view. The model connects application part and communication part of RTCS verified by verification tools.

#### 8.3.1.3. Tools

To modelling and verifying the models above we use a tools that are allowed do that with sophisticated mathematical formalisms. This tools are based on Petri Net or communicating automata and they proved their qualities in research community. Our intention is to use them in applications based on RTOS and fieldbus systems.

##### 8.3.1.3.1. PEP tool

PEP tool (Programming Environment based on Petri nets) [Best98] [PEP] is able to model concurrent systems and to verify them by partial model checking based on a compositional denotation Petri nets semantics. The language supported by the tools covers block structuring, parallel and sequential composition, synchronous and asynchronous communications and so on.

Modelling allows to create either graphical version of Petri net model or structured program code of the model in B(PN)$^2$ (Basic Petri Net Programming Notation) [Best83] or SDL (Specification and Description Language) [PEP].

PEP contains those verification components:

- FC2Tools (verification based on networks automata)
- SMV (CTL model checking)
- SPIN (linear temporal logic with optional partial order reduction)Deadlock free checker

##### 8.3.1.3.2. UPPAAL

UPPAAL [UPPAAL]allows modelling, simulation and verification of real-time systems. It is appropriate for systems that can be modelled as collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables.

Typical application areas of the tool include real-time controllers and communication protocols in particular, those where timing aspects are critical.

#### 8.3.1.4. Remarks

The goal of our work is to use the proved tools to create some models of OCERA software components. A future application developer will need to create only a model of his application tasks and to define the model properties to be verified.

The essential part of the work will be an example of the RT application, its model and properties verification.

## Bibliography

[[Holzmann91]] Gerard J. Holzmann, 1991, Prentice Hall, *Design and validation of computer protocols*.

[[Buttazzo97]] Giorgios C. Buttazzo, 1997, Kluwer Academic Publisher, *Hard Real-time computing systems: Predictable Scheduling Algorithms and Applications*.

[[Best98]] Eike Best and Bernd Grahlmann, 1998, *Programming Environment based on Petri nets: Docummentation and User Guide Version 1.8*.

[[Best83]] Eike Best and R. P. Hopkins, 1993, *B(PN)$^2$ - A Basic Petri Nets Programming Notation*.

[[UPPAAL]] *UPPAAL tool: http://www.docs.uu.se/docs/rtmv/uppaal/*.

[[PEP]] *PEP tool: http://theoretica.informatik.uni-oldenburg.de/~pep/*.

## 8.4. Communications Components Summary

**Components at the High-level RTLinux**

- CANopen device
  - □ Description: OCERA RT CANopen device is a software solution based on OCERA RT Linux and VCA capable to exchange its data with any industrial CANopen device following the CANopen communication standard. It can be configured to work as CANopen master, CANopen slave or CANopen NMT master. Type of CANopen device is specified by loading appropriate Electronic Data Sheet (EDS) into device Object Dictionary (OD).
  - □ Responsible: CTU
  - □ Validator: UC

**Components at the Application-level RTLinux**

- Virtual CAN API (VCA)
  - □ Description: The Virtual CAN API introduces CAN network to the application threads. Application threads can reside in the soft real-time space or in the hard real-time space. VCA offers minimal set of functions enabling to open/close/configure CAN device and send/receive CAN messages.
  - □ Responsible: CTU
  - □ Validator: UPVLC

- EDS parser and CAN/CANopen analyzer
  - □ Description: This component captures the traffic on CAN bus and analyze it on the level of CAN massages. A CANopen device Electronic Data Sheet (EDS) can be loaded into analyzer. In that case the analyser can send and receive SDO communication objects and show its impact in a device Object Dictionary (OD). The analyser olso offers basic CAN open NMT functionality.
  - □ Responsible: CTU
  - □ Validator: UC

**Components at the Application-level Linux**

- Real Time Ethernet (ORTE) device
  - □ Description: The Ocera Real-Time Ethernet (ORTE) is open source implementation of RTPS communication protocol. This protocol has already been submit to IETF as an informational RFC and has been adopted by the IDA group. RTPS is new application layer protocol, which is built on top of standard UDP stack. This protocol stack adds real-time capabilities to standard Ethernet technology. Publisher/subscriber mechanism is used to share data between user applications. The component will be a library linkable against the user applications.
  - □ Responsible: CTU
  - □ Validator: SSSA

- Real Time Ethernet analyzer
  - □ Description: This component enables to developer to capture network traffic and to analyse it on the level of Ethernet frames, IP and UDP datagrams and RTPS messages. Real Time Ethernet analyzer is not stand alone application, but it is a plug-in module for Ethereal network analyzer.
  - □ Responsible: CTU
  - □ Validator: SSSA

**Verification**

- CAN model by timed automata /Petri Nets
  - □ Description: This component is theoretical study offering methodology tool support for analysis of distributed system consisting of *n* independent processors and deterministic communication bus (CAN). In order to verify distributed RT system, application designer needs to create model of application tasks and to interconnect this model with communication bus model provided by this component. Finally he/she needs to define system properties to be verified (deadlock, missed deadline etc.). The approach is illustrated in the form of examples in PEP verification tool.
  - □ Responsible: CTU
  - □ Validator: CEA

- Verification of cooperative scheduling and interrupt handlers
  - □ Description: This component is theoretical study offering methodology and tool support for model checking of real-time applications running under multitasking operating system. Theoretical background is based on timed automata by Allur and Dill. As this approach does not allow to model pre-emption we focus on cooperative scheduling. The cooperative scheduler under assumption performs rescheduling in specific points given by "yield" instruction in the application processes. In the addition, interrupt service routines are considered, and their enabling/disabling is controlled by interrupt server considering specified server capacity. The server capacity has influence on the margins of the computation times in the application processes. Such systems, used in practical real-time applications, can be modelled by timed automata and further verified by existing model checking tools. The approach is illustrated in the form of examples in the real-time verification tool UPPAAL.
  - □ Responsible: CTU
  - □ Validator: CEA

# Chapter 9. Glossary of terms

Please, add all the terms that you think it may be useful to be here:

**User space**

The execution environment (characterized by restricted privileges, addres-space protection, etc.) in which Linux applications run.

**Kernel space**

The execution environment of the Linux kernel (maximum privilege, no address-space protection, etc.)

**Linux module**

A object file which can be dynamically linked (and unlinked) into the running Linux kernel with the `insmod` command. A module can access to all the Linux kernel functions and data structures (if they are exported).

**Linux kernel**

The kernel, as released by Linus Torvalds at kernel.org. The Linux kernel version currently used in the OCERA project to is 2.4.18. At the end of the project, the version will probably be upgraded (depending in the kernel evolution) and all the components will be ported to the new version.

**RTLinux layer**

The term "RTLinux layer" is used for identifying the RTLinux exectuive and the set of real-time tasks using it.

**Linux layer**

The term "Linux layer" is used for identifying all the code running in kernel space that does not depend on RTLinux.

**OCERA component**

A piece of software that brings some new functionality or feature in some of the fields: Scheduling, Quality of Service, Fault-Tolerance and Communications. Depending on the type of facility and its role, a component can be: a patch, a stand-alone module, a library, or a thread.

**OCERA framework**

The development environment provided to the final user for building and installing applications using OCERA components.

**RTLinux executive**

The Linux patch and the set of kernel modules that provide the RTOS functionality out of the scope of Linux kernel. In the strict sense, it is not an operating system, since it can not boot nor have many of the facilities required take full control of a computer. RTLinux only manages the set of hardware devices required to provide deterministic timed behaviour.

**Open RTLinux or RTLinux/Open**

The version of RTLinux released by FSMLabs covered by the Open RTLinux license. It is a small RTOS which coexists with Linux kernel and intercepts the low level interrupts and processor control instructions which allows to have the control of the computer at any time independently of the Linux kernel state.

**OCERA Linux kernel**

The Linux kernel containing existing, as well as the OCERA developed, patches to enhance the real-time capabilities. This kernel will be considered as a Soft Real-Time system.

**RTLinux/GPL**

Same than RTLinux/Open. FSMLabs use both terms interchangeably.

**RTLinux/Pro**

The commercial version of RTLinux developed and distributed by FSMLabs.

**Task**

A executing unit, which can be a normal process or a thread.

**User Space Task (or User Space Application)**

Task (or application) running in user space, that uses the Linux services only by invoking system calls.

**RTLinux Task (or RTLinux Application)**

Task (or application) running in kernel space, that directly uses the RTLinux services