WP6 - Fault-Tolerance Components

Deliverable D6.1 - Design of fault-tolerance components

WP6 - Fault-tolerance components : Deliverable 6.1 - Design of fault-tolerant components
by A. Lanusse

Published February 2003

Table of Contents

Document Presentation

Project Coordinator

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

Participant List

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

Document version

Release	Date	Reason of change
1_0	20/02/2003	First release

Chapter 1. Introduction

With the increasing development of complex real-time embedded systems, there is a growing demand for low cost real-time development environments proposing new facilities for QoS management and fault-tolerance. However, This is a very challenging objective that is the integration within open source and free real-time operating systems of fault-tolerance management features.

Fault-tolerance has been largely studied in the past twenty years and a rich background exists in that domain. However, if general principles are well known and generic mechanisms have been described very well in the literature, very few have been made available within open source community. The goal of the fault-tolerance work package in OCERA is precisely to bring to the open source community basic functionalities useful for user's willing to improve applications robustness.

1.1. Main objectives

The main objective of the fault-tolerant work-package in OCERA is to provide two types of facilities : degraded mode management in mono-node applications and redundancy management in distributed applications.

The development of the fault-tolerant facilities will be decomposed into two major classes:

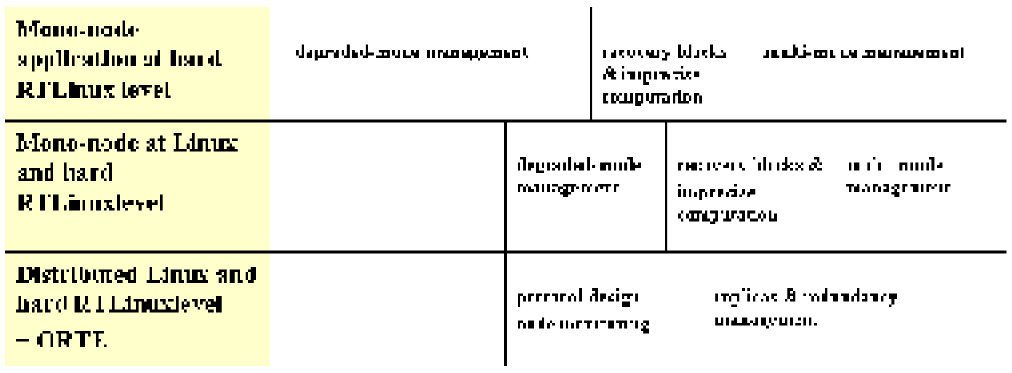
Mono-node applications.

In this context fault-tolerance will consist in providing facilities for handling two types of abnormal situations : deadline miss for hard-real-time tasks and task abortion.

The first range of facilities will implement degraded mode management at the task level and application level.
The second range of facilities will cover recovery blocks management and imprecise computation management.
Moreover, the implementation of degraded mode management will provide a basis for the definition of a more general multi-mode application management scheme.

Distributed architectures.

In this context, we will provide support for redundancy management in multi-node architectures in collaboration with developers of the communication components. Additional abnormal situations handled will concern node failures.



Mono-node application at hard RTLinux level	degraded-mode management	recovery blocks & imprecise computation	multi-mode management
Mono-node at Linux and hard RTLinux level	degraded-mode management	recovery blocks & imprecise computation	multi-mode management
Distributed Linux and hard RTLinux level – ORTE	protocol design node monitoring	replicas & redundancy management	

Figure 1. OCERA targeted fault-tolerant facilities

These objectives imply the definition of an overall fault-tolerant framework well integrated in the RTLinux architecture where specific fault-tolerant components cooperate with other OCERA components; they also imply the definition of an associated user oriented development methodology for fault-tolerant applications.

1.2. Development steps

The development strategy will follow an incremental process.

A first version (V1) of the fault-tolerance components will provide basic degraded mode management features; the main objective of the development related to this phase is to set up the overall framework required to handle fault-tolerance and propose an associated development methodology for users. This will concern applications limited to hard-real-time tasks.

Two application level RTLinux components will be developed: an application fault-tolerance monitor and a fault-tolerance controller (components with dashed borders in fig.2). They will control the application behaviour and will be respectively in charge of handling error recovery at the application level and at the task level.

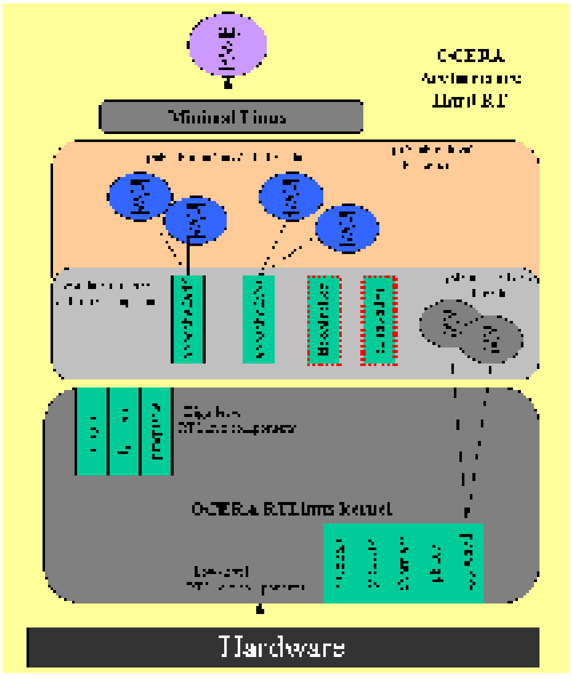


Figure 2 . Fault-tolerant components in OCERA architecture (V1)

Next developments leading to the second version (V2) will concern both the enrichment of basic fault-tolerance strategies and the support for more complex application architectures. Extensions to first components will be provided so as to handle both hard-real-time and soft real-time tasks.

This will require that the previous fault-tolerant components be redefined as two cooperating components located at the Linux application level and at the application RTLinux level.

Moreover, this cooperation requires that bounded reaction time can be guaranteed between the detection of an abnormal situation implying an application mode change and the effective mode change activation by the Linux application level **aftmonitor**. The implementation of OCERA components providing a hierarchized CBS scheduling is thus a prerequisite for this step.

In addition to that, specific new components will be added in order to provide new facilities to handle redundancy management in distributed environments. The V2 version will require additional components for redundancy management : a fault-tolerant redundancy manager and a fault-tolerant replica manager. They will respectively control redundancy at the application level and at the task level on each node. The design of these components will start with the second phase of the project.

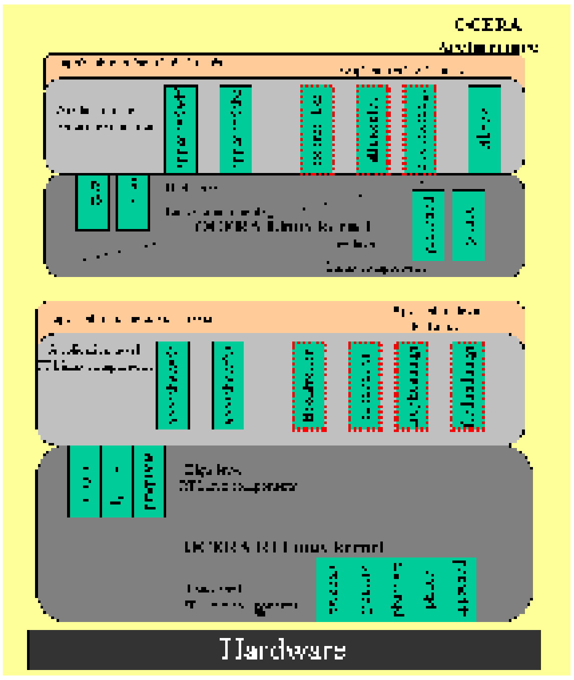


Figure 3 . Fault-tolerance components in OCERA architecture (V2)

1.3. General Philosophy

Defining a well integrated consistent architecture requires the definition of a global programming model and a smooth integration within the current one. It is important to notice that fault-tolerance implementation requires not only the development of specific fault-tolerance components but also the adaptation or use of almost all other OCERA components and in particular scheduling and QoS components.

The changes to current other OCERA components must be as light as possible and the user must not be obliged to rewrite its code with specific primitives.

We have thus opted for a declarative approach for the definition of fault-tolerant behaviour of the application . Only few fault-tolerant primitives will have to be added to the existing code and a specific tool will help completing fault-tolerance information (related to tasks and application) that will be used to instantiate the various run-time components in charge of the behavioural control of the application.

In this approach, non functional requirements related to fault-tolerance will be collected through a design tool and then used by a building tool to instantiate the fault-tolerant components and to create the fault-tolerance tasks.

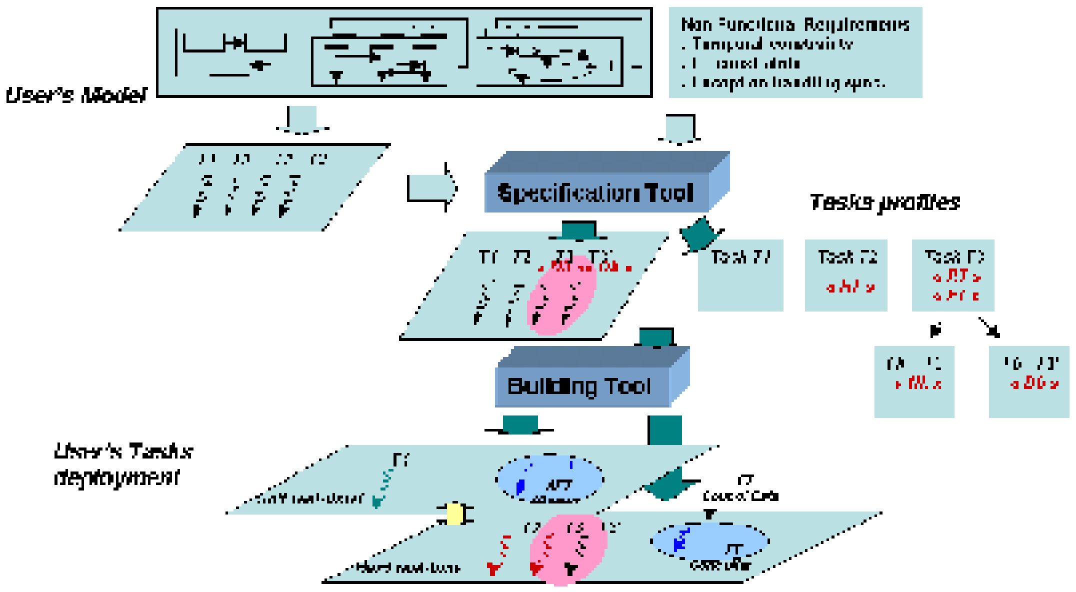


Figure 4 . Declarative approach and supporting tools

The specification tool will permit the user to specify for each task the related temporal constraints, the different possible alternative behaviours (functions to be activated in the threads), and the transition conditions for switching behaviour. It will also describe application modes and transition conditions for application mode switch. The modeling of the application will rely on a generic task model described in the next section.

In the second phase of the project, it will also help define which tasks will be redounded and their level of redundancy. This will imply also that deployment information is given in order to specify tasks location.

Two fault-tolerant components will gather all the knowledge concerning the abstract levels tasks, and associated control information for their management so that other OCERA components can continue to work only with **regular rt-Linux** threads. Very few adaptations will be necessary to allow for the notification of specific events in order for the fault-tolerant components to be kept informed on the current status of execution of the threads.

Chapter 2. Methodological approach

As introduced in the previous section, we have made the choice of handling fault-tolerance on the basis of a declarative approach combined with transparent error handling mechanisms. An other possibility would have been to define specific primitives that would have been used inside the application code directly by the programmer.

This choice is driven by the fact that we consider fault-tolerance as additional non-functional requirements that must not interfere with application core coding since it may be subject to change and it must be handled in a consistent manner which is not easy to control if it is embedded in the code.

This declarative approach forces the user to specify transition conditions both at application level and at task level to handle properly reactions to abnormal events. These transition conditions are used to instantiate specific error handling hooks.

The goal of ft components is to insure a transparent and safe management of such transitions at task and application level. A particular attention must be paid to the overall application logical and temporal consistency and to a clean resource management so that aborting a task does not produce subsequent tasks blocking.

The basic principles of degraded mode management according to this approach are the following: when an error is detected at task level, it triggers a task behaviour change to a degraded mode and propagates the notification of abnormal event at the application level where a decision is taken to apply or not an application mode change.

In the next sections we present : the application and task models used and the errors handling principles to be used in the first implementation step.

2.1. Application model

- An application consists of a set of tasks.
- Each task may have several behaviours.
- An application may have several modes of functioning
- An application mode is defined by :
    - the set of all the tasks mandatory for the correct functioning of the mode.
    - The QoS requirements for each task (scheduling parameters and behaviour) of the mode

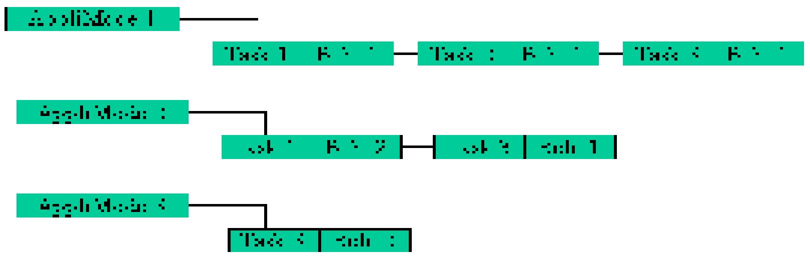


Figure 5 . Application model for degraded modes management

- An application mode change can be triggered when
- A user condition is verified
- An abort or deadline miss occurs for one of the tasks present in the mode an triggers a behaviour change
- An application mode switch results in triggering a new mode which implies
    - Stop some tasks
    - Start new ones
    - Switch mode for some tasks
    - Reschedule
- A state machine describes the transition conditions for mode change triggering
  As an example, the application described above will have an associated statechart that will describe its transitions. In this particular case, a simplified view shows the transitions triggered by errors occurred on task 1.

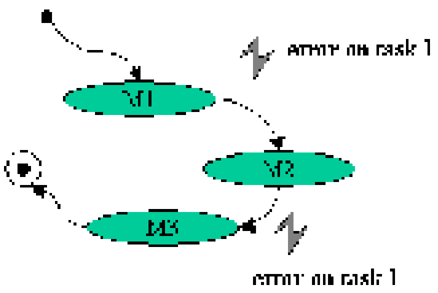
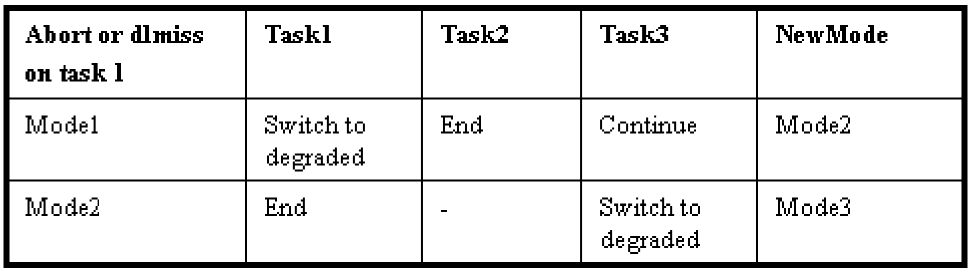


Figure 6 . Application modes transition statechart (filtered view for event error on task1)

Each transition implies possible actions to be applied to tasks, in the case of an abort or deadline miss on task1 the actions on other tasks are the following.



Abort or dlmiss on task 1	**Task1**	**Task2**	**Task3**	**NewMode**
Mode1	Switch to degraded	End	Continue	Mode2
Mode2	End	-	Switch to degraded	Mode3

Figure 7 . Table of actions related to transitions in statechart

2.2. Task Model

We introduce a concept of fault-tolerant task.

- A fault-tolerant task (ft_task) is a real-time task with several possible behaviours. In our current model a ft_task has at least two possible behaviours, a nominal one and a degraded one (the degraded one can simply be ending the task, this particular case corresponds to a regular task).
- QoS parameters are attached to each behaviour. They encompass current rt-linux tasks attributes such as temporal constraints and scheduling policy. In the future, they will also include resource reservation constraints for memory and communication.
- The transition conditions between nominal and degraded behaviours are described in a state machine. An abort or deadline miss triggers a transition to a more degraded mode if no continue condition is true.

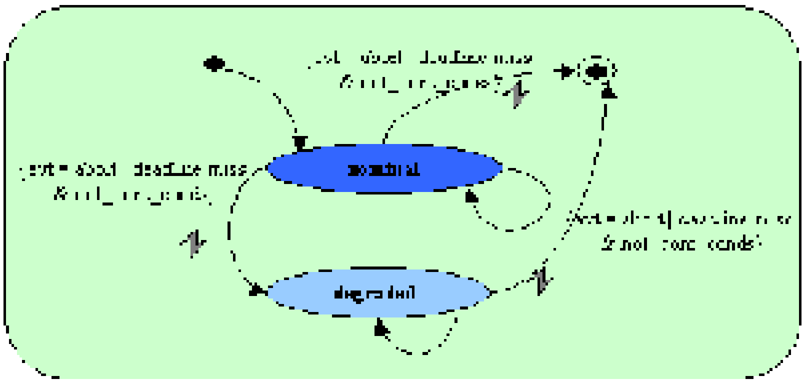


Figure 8 . Task statechart for degraded mode management

- A behaviour is implemented as a thread, (real-time task of rt-linux).
- The two behaviours corresponding to a given task are implemented as two different threads.
- At start-up both threads are created but only one of them, the one corresponding to the nominal behaviour is activated, the second one keeps suspended until a specific fault-tolerant signal-handler make it running.

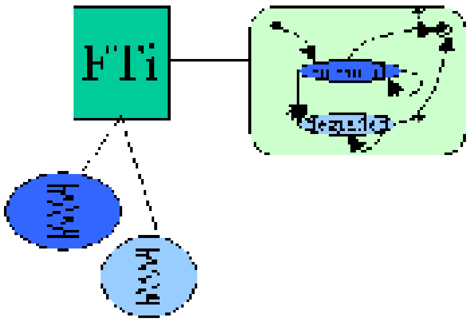


Figure 9 . Fault-tolerant task related entities

2.3. Global view of an application

At a given moment, the application can be viewed as the set of the current active threads corresponding to the tasks behaviours activated in the current application mode.

The transitions from one application mode to an other changes the active threads configuration and associated scheduling parameters.

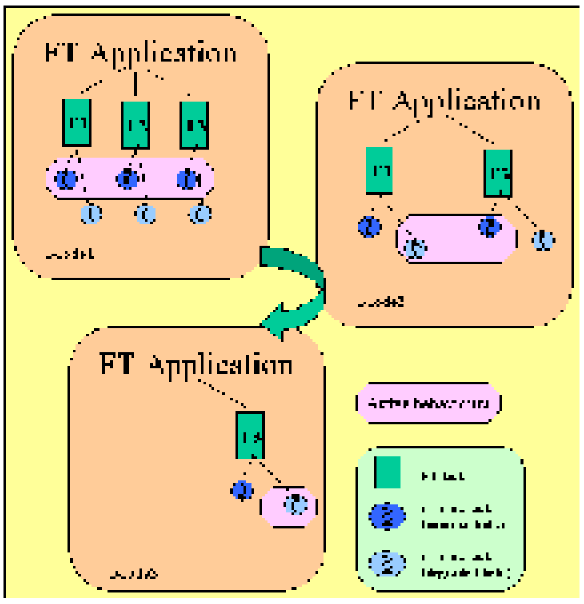


Figure 10 . Global view of an application along its life

Chapter 3. Errors handling

3.1. Types of errors handled

Fault-tolerance in OCERA will first concern mono-node hard real-time applications. In this context, errors handled will be limited to software errors of two major types:

- Timing errors (deadline miss)
- Programming errors leading to Abort default action in POSIX

3.1.1. Timing errors

Timing errors corresponding to deadline miss are a major issue for real-time systems. Normally they should never happen since the system must be designed so that it can react in time. If they occur, they reveal an abnormal situation that can be due to an overloaded situation, blocked threads or a faulty component not responding. Different strategies can be defined to handle these situations. They can be considered at two levels: at the task level and at the application level. A first compensation strategy can be defined at task level while a global strategy can be applied at task level, where it is possible to detect overloads condition requiring tasks reconfiguration and a redefinition of scheduling parameters.

At task level, specific strategies can be defined depending on the type of tasks handled. For example, deadline misses can be tolerated in certain conditions if predefined default actions can compensate them. A deadline miss at task level may be admissible for example for acquisition tasks provided, it does not occur systematically. Such tasks are generally periodic tasks and produce each period a certain result (data acquired during the period). It is possible to associate a default result function that defines a way of delivering a default result in case of deadline miss. It can be decided that the previous result is stored every period in a state variable and can be used instead of real acquired data. More rich strategies can use the mean of the last three periods or any extrapolation function. In this case it can be decided that such default result can be transmitted, once or two consecutive times before the task is definitively considered faulty.

In such cases the strategy is to end the current cycle of the task, transmit the default result with an associated validity value, notify the application level and wait for the next period. Classical real-time applications are generally constituted of a reduced set of types of tasks. Generally acquisition tasks, controlling tasks and action tasks (that command actuators).

In such applications most tasks are independent and shared resources concern only one to one communication with controlling task. So providing in time a result even if it is a default one avoids temporal propagation of abnormal situation and reduces the risks of locked threads.

At application level dynamic reconfiguration techniques can help satisfy timing constraints for most important tasks by stopping less important ones or choosing an alternate less consuming behaviour for some tasks. They can be triggered on timing errors or in a preventive manner from feedback information on load by high level QoS components.

Specific implementation patterns must be provided in order to support such schemes. However in the first implementation, timing errors will be considered as the consequence of a non responding faulty component. The corresponding task behaviour will thus be ended and an alternate one corresponding to a degraded functioning mode will be activated if possible. This default strategy will be common with the abort handling.

3.1.2. Programming errors

Concerning programming errors, the goal is to permit the system to continue the application with a degraded mode when a thread is aborted by the kernel. Indeed, a default Abort or Dump action takes place in many situations when the kernel receives POSIX SIGNALS generated by abnormal situations (SIGSEGV, SIGFPE, SIGPIPE, SIGILL,…). Each of these exceptions may have several possible causes, some may be fixed by the kernel but some others oblige the system to abort the thread.

The strategy adopted will consist in activating an alternate behaviour for the faulty task and propagate this task change to the application level in order to decide if an application mode has to be activated. If it is the case the application mode switch takes place. This is a best effort approach that doesn’t prevent from thread abortion but limits the impact on the rest of the application.

It is important to remind that RTLinux tasks run in the kernel space and that a programming error can lead to kernel crash. It is thus highly recommended that users adopt a defensive programming style in order to limit as far as possible threads abortion by the kernel.

Such a defensive programming relies on a systematic control of return values from system calls and user handling of error. Most programmers are already used to write generic errors management routine for most frequent system calls such as open, close, read, write etc…that can try other solutions when an error occurs than abort the thread. Pointers should systematically be carefully checked.

Some errors can be detected during debugging thanks to the use of the `assert()` macro that can help control that some conditions are verified during execution. Errors like EFAULT, EDOM or ERANGE should not happen in a program well debugged. Overflow and bound checks can also be achieved during debugging.

Other errors are more difficult to detect but are generally due to bad evaluation of resources required for the system. They can lead to errors like ENOMEM where the system can not do anything else than abandon the operation. Careful design and off-line analysis can help reduce the occurrence of such errors.

Specific guidelines for device drivers programmers are being issued in a separate document in order to give hints to avoid risky coding.

3.1.3. Hardware errors

Concerning hardware errors, we will consider two types of situations : errors related to hardware components such as sensors or actuators and errors related to nodes in the distributed architecture.

Hardware components such as sensors and actuators

Such components will be supposed failed silent. Their failure will thus be detected by a timeout. If it were not the case, a specific protocol would be defined in order to implement this behaviour in the corresponding driver.

Nodes in a distributed architecture

In the second phase of the project, we will support fault-tolerance management of distributed architectures composed of several nodes running OCERA kernel and communicating thanks to ORTE communication package.

Two types of facilities will be implemented for fault-tolerance management :

The first one will be the monitoring of the overall architecture in order to detect node failures.

The second one will be devoted to redundancy management.

Nodes failure can be due to hardware failure or to a kernel crash consecutive to a programming error. It is also necessary to make sure that an apparent node failure is really a node failure and not in reality due to a faulty communication system.

In OCERA, a kernel crash will lead to a node failure and a fail silent behaviour of the corresponding node.

A node monitoring protocol will be defined to detect non responding nodes. This protocol will be implemented using ORTE communication components that provide already facilities to monitor the network and connected nodes.

3.2. Errors management

Transition from one application mode to an other will result from an abnormal situation detected in one of the threads by the kernel. In the future it will also possibly be activated by the detection of external conditions decided as abnormal and defined as a transition condition declared by the user.

In the first implementation, transitions will be handled by a specific signal handling pattern that can be briefly described as follows.

To each behaviour is associated a set of signal handlers specific to the treatment of the two types of abnormal situations listed in the introduction of this document (namely deadline miss and abort).

It will be possible to define specific strategies for each one but in a first implementation phase they will both lead to the end of the related thread and will activate the alternate behaviour (actually the other thread) or end the task. This immediate action will be followed by the notification of the behaviour change of the task mode and the propagation to an application controller that will analyze the consequence of this change and possibly switch the application mode to a new one that will require a different configuration of tasks with different QoS attributes. This will possibly imply tasks termination and / or tasks behaviour switch and /or new tasks creation and rescheduling.

Chapter 4. Fault-tolerance components architecture

4.1. Run-time components for degraded mode management

4.1.1. Architecture overview

In the first implementation step we will consider the implementation architecture at the Hard RT level.

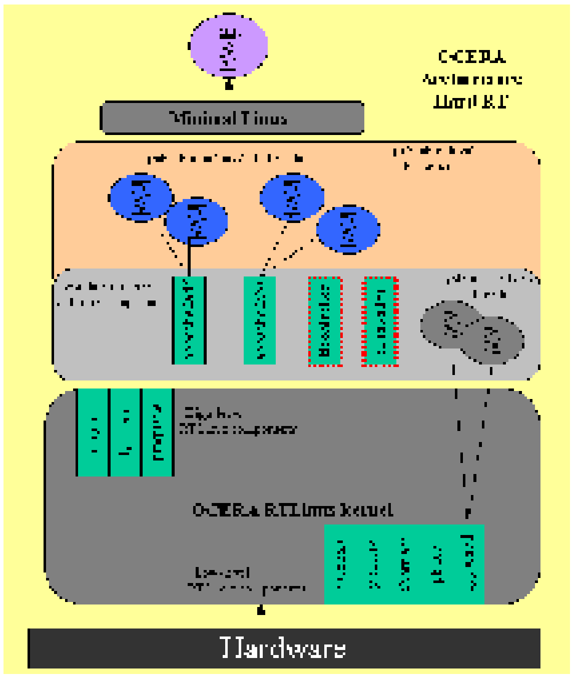


Figure 11 . FT architecture overview

All the application tasks will be hard real-time tasks that will be either periodic or sporadic tasks.

The implementation of degraded mode management will require two OCERA RTLinux components located at the application level:

- an Application Fault Tolerant Monitor (**aftmonitor**) in charge of the global application monitoring and
- a Fault Tolerant Controller (**ftcontroller**) in charge of the low level control of the running threads, the emergency signal handling, the activation of replacement behaviour and propagation of the notification of the behaviour change to the **aftmonitor**.

They cooperate with a specific application scheduler component located also at the application level.

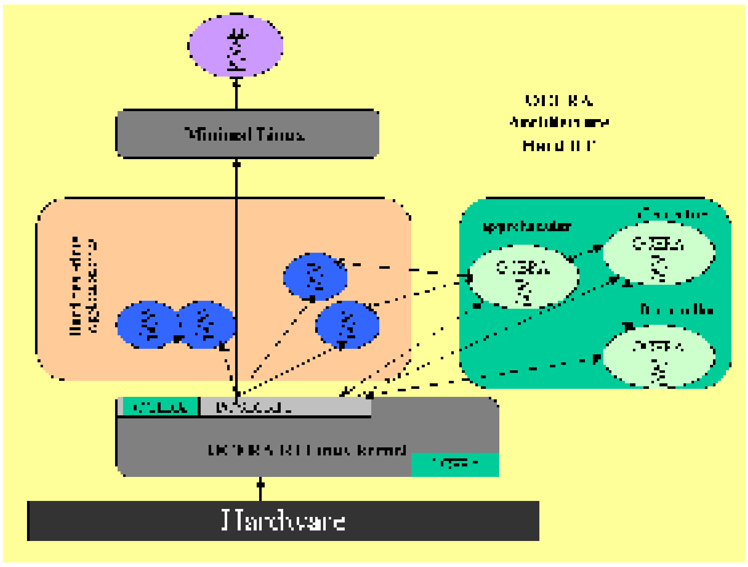


Figure 12 . FT architecture : OCERA components at RTLinux level

4.1.2. Basic interaction between components

The main interactions between components are illustrated in following figure.

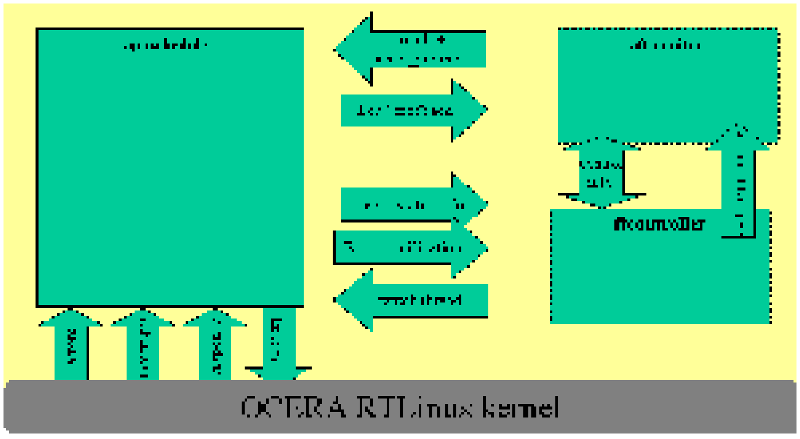


Figure 13 . Interactions with FT components

At init, information on application modes and tasks is dispatched between the different data structures and tables in the **aftmonitor** and the **ftcontroller**.

This concerns the list of all tasks of the application and their associated behaviours along with the transition tables and error hooks.

When the application starts threads are submitted with associated scheduling parameters to the appscheduler. When threads are elected by the scheduler for running a notification is done towards the **ftcontroller** with the deadline parameter.

The **ftcontroller** keeps tracks of all threads started. If the threads end normally, the **ftcontroller** is informed. If an error occurs or a deadline is missed a specific handler stored in a table in the **ftcontroller** applies immediate action. By default the action is to switch the active thread to his alternate suspended thread. Then, the **ftcontroller** notifies the **aftmonitor** of the abnormal event and of task mode switch.

The **aftmonitor** checks if the abnormal event triggers an application mode change condition and activates it if it is the case. This results in a new configuration of tasks behaviours and associated scheduling parameters to be submitted to the application scheduler.

The error management process can be illustrated by the next diagram.

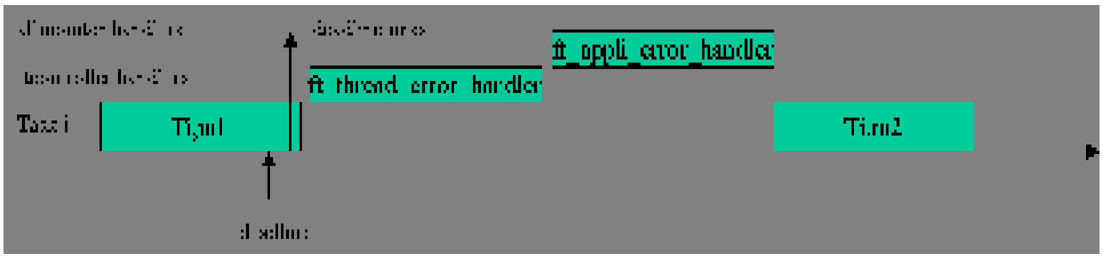


Figure 14 . Error management process

A second type of interactions is directed towards prevention of overloaded situations. It will be supported by cooperation with schedulers that will provide feedback information on scheduled threads and on workload. This will be supported in the second phase of the implementation by the use of two cooperative schedulers (edf and cbs).

The cbs application scheduler will perform resource reservation and workload monitoring , if overload is foreseen, a dialog with the **aftmonitor** will permit to prevent effective overload situation by taking actions on less important threads in order to avoid most important ones to become faulty.

Up to now CBS components take decisions only on the basis of the budget allocated to tasks, cooperation with **aftmonitor** that has knowledge on tasks importance and possible alternative will help prevent timing faults on important tasks.

The edf scheduler will schedule tasks on the basis of deadlines since as long as there is no overload situation it is an optimal algorithm.

4.1.3. Synchronization principles

As said in a previous section, in the beginning of the project, tasks will be considered as periodic.

Two classes of tasks have been identified.

- Simple periodic tasks that are activated by a timing event each new period and execute sequentially a set of actions then wait for the next period. These are representative of usual acquisition or serving tasks.

- Controlling tasks that periodically receive data , perform computation and send control data. These are also periodic tasks, they perform each cycle a set of actions but they can also receive aperiodic requests and react to them during their period.

In this schema simple tasks do not interact with each other; they interact with a controlling task. Several simple tasks may interact with a unique controlling task, however, they use different communication entities (one to one communication). Shared resources are only of the type one producer - one consumer.

Periods values are defined in such a way that consistency can be insured between controlling and simple tasks (usually a same period).

In these conditions, an abnormal event occurring during a period will be taken into account during the period and induce reconfiguration of impacted tasks so that the new configuration can be made operational for the next period. Default strategies will be defined so that the impact of an error during one period can be tolerated.

This blackbox view of tasks is however limited, in the future we intend to extend the fault-tolerance facilities to more detailed tasks models. This will require the description of the body of tasks in terms of actions such as call action, communication action (send, receive, read, write) etc… This will permit to specify possible breakpoints in the code and model tasks behaviours in a more accurate way allowing a finer management of error recovering.

4.1.4. Application building

Application building is a major issue in such an environment. It is the time when all declarative information gathered during design is finally introduced into the system in order to prepare it for handling fault-tolerance. During this step several types of information is entered in the system:

- Information on tasks and application structure
- Control information on application and tasks
- Code of tasks (behaviours and error_handling routines)

As illustrated in the next figure. This information comes from three sources. Structural information comes from application and task descriptions, control information comes from statecharts and code is associated to behaviours function used on thread creation.

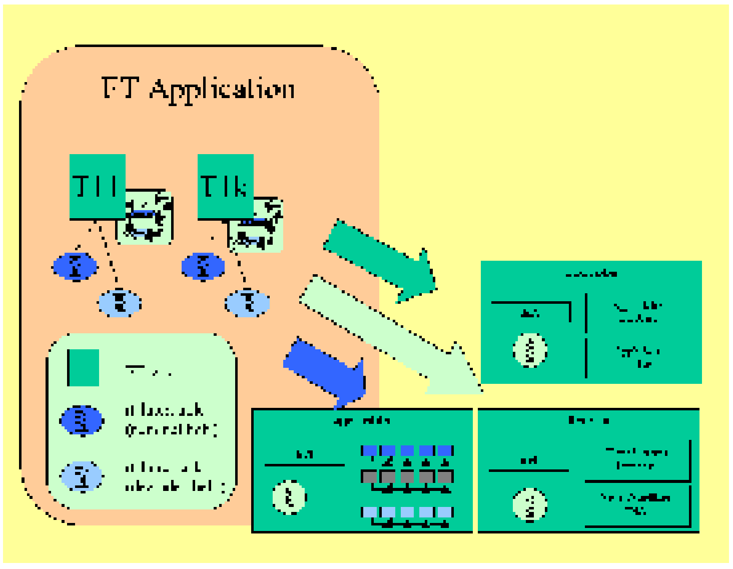


Figure 15 . Application building : FT components instantiation

Part of this information can be get from a design/build tool that will be developed in the project. It concerns the two first types of information. The last one, corresponding to tasks code will be written as any other RTLinux code. A task behaviour implemented as a thread is exactly an RTLinux task.

However, our specific notion of task does not exist in RTLinux, it is an encapsulation that permits to handle fault-tolerance management at the task level. So specific init and delete primitives have to be defined at the task level, these primitives in turn call usual RTLinux thread primitives.

The full declaration of a task consists thus in :

- Enumerating its behaviours
- For each behaviour define
    - FT attributes (mode)
    - QoS parameters (temporal parameters, and resource requirements)
    - Body (a start_routine() to be run at thread creation)
- Init actions
    - Threads creation
- End actions
    - Threads delete
- Error handling on
    - Abort
    - Deadline miss

The tasks and application data structures are preset by the building tool, only effective task init and task creation are done as a programming step with the ft_task_init and ft_task_create primitives described in the **aftmonitor** section.

Chapter 5. FTController

5.1. Description

The **ftcontroller** component consists of one controlling threads a threads status database and a table containing handler routines for tasks mode switch on errors.

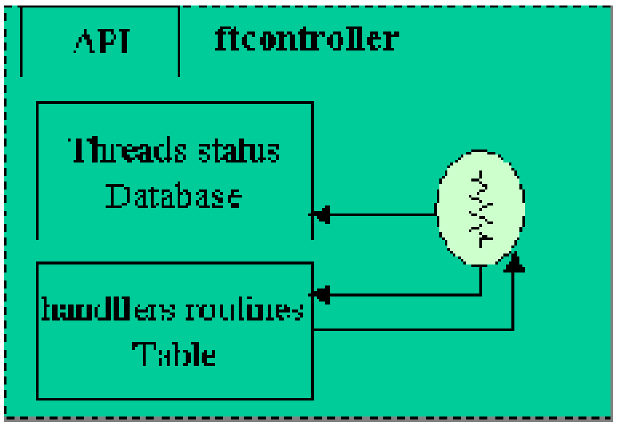


*Figure 16 . **ftcontroller** internal view*

The threads status data base collects the current status of the thread along with information on its start date and deadline from the application scheduler. An alternative implementation solution will be studied in which information will come from the tracing facility (ptrace component).

The handlers routine table contains two entries for each possible thread of the application. These two entries correspond to the two types of errors currently supported.

The routines have all the same structure and are instantiated at application init.

Each routine have four actions :

- Ends the current thread (faulty if not yet aborted)
- Sets the scheduling parameters for the replacement behaviour
- Activates the replacement behaviour
- Propagate event to **aftmonitor**

The need for different routines resides in the fact that in the future, richer error handling strategies will be made available and may be different for different tasks.

5.2. Layer

This component is located at the Application RTLinux Level.

5.3. API / Compatibility

This component uses POSIX compatible RTLinux API for threads manipulation and OCERA components API (ptrace, psignals, ptimers, pbarriers, appsched). A few additional primitives have been defined.

The API can be divided into two subsets. An external API for communication with scheduler and an internal API to be used for communications between **aftmonitor** and **ftcontroller**.

5.3.1. External API

```
ft_notify_thread_started         // Start of cycle

parameters :

in :

          threadId

          executionStartDate

          deadline

Description:

Used by the scheduler to notify that a thread has started its execution
(running and elected). If it is a periodic thread, it must be notified at
each new period.


ft_notify_thread_finished       // Normal end of cycle

parameters :

in : threadId

   executionEndDate

   deadline


Description :

Used by the scheduler to notify the normal end of execution of a thread.
For a periodic thread it notifies the end of a cycle.


ft_notify_deadline_miss

parameters :

in :

          threadId


Description :

Used by the scheduler to notify an error. Actually this should rather be a
signal. If not, the scheduler must set a high priority to ftcontroller
after this call.
```

ft_notify_thread_aborted

parameters :

in :

threadId

Description :

Used by the scheduler to notify an error. Actually this should rather be a signal. If not, the scheduler must set a high priority to **ftcontroller** after this call.

5.3.2. Internal API

ft_notify_task_created // Init of task

parameters :

in :

taskId

task_name

normal_threadId

degraded_threadId

normal_thread_parameters

degraded_thread_parameters

Description :

Used by the **aftmonitor** to notify the creation of a task and transmit related information to the **ftcontroller**. This information is used to init the data structures in the ft_controller (threads status database and errors routines table.

ft_switch_task_mode // Init of task

parameters :

in :

threadId

Description :

Used internally by the **ftcontroller** to activate task_mode switch.

ft_set_task_status

parameters :

in :

taskId

taskStatus

Description :

Used internally by the **ftcontroller** to set the task_status.

The task status tells if a task is created, active or ended.

ft_get_task_status

parameters :

in :

    taskId

out :

    taskStatus

Description :

Used internally by the **ftcontroller** to get the taskStatus.

ft_get_task_mode

parameters :

in :

    taskId

out :

    taskMode

Description :

Used internally by the **ftcontroller** to get the taskMode.

The taskMode indicates the current active mode of the task.

Chapter 6. Aftmonitor

6.1. Description

The **aftmonitor** component consists of one controlling threads, an application control database and an application status database.

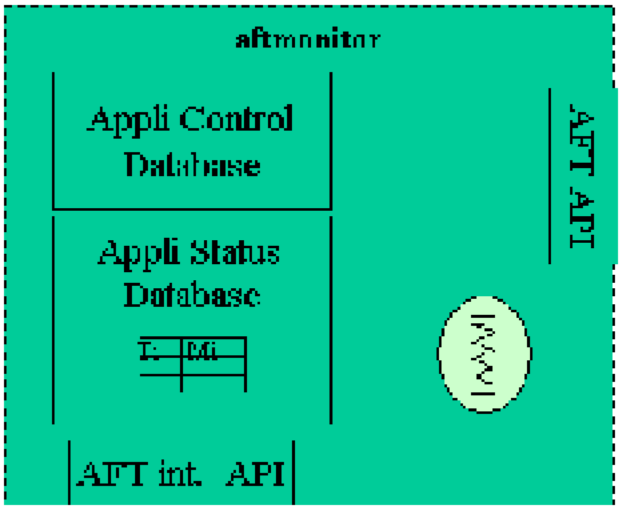


*Figure 17 . **aftmonitor** : internal view*

The application control database contains the transition conditions for application mode change and the related actions.

The application status database contains information on the current configuration of the application, i.e. for each task its status and active mode.

The controlling thread receives notification of errors and task mode changes, analyses the situation and if necessary activates an application mode change.

Depending on the error and the current application mode, the process detects which transition must be fired. Then it compares the current configuration and the new targeted one.

For each task in the current configuration,

If the task is present in the new configuration with the same behaviour nothing is done for the task and its current running thread.

If the task is present in the new configuration with a different behaviour, the current thread of the task is aborted, and the alternate behaviour is made active for the next period .

For each task not in the current configuration new tasks are created with appropriate behaviour.

Synchronization is ensured in order to respect the periodic functioning of the application. Tasks interrupted during the period in which the error occurs will have misbehaving during this period that will be tolerated by tasks with which they communicate.

6.2. Layer

This component is located at the Application RTLinux Level.

6.3. API / Compatibility

This component uses POSIX compatible RTLinux API for threads manipulation. A few additional primitives have been defined. Scheduling parameters will follow the structure required by the OCERA scheduling components (CBS and / or EDF) when used.

6.3.1. External API

```
ft_task_init

parameters :

in

          task_name

          normal_behaviour_function

          degraded_behaviour_function

          normal_behaviour_scheduling_parameters

          degraded_behaviour_scheduling_parameters

out

          taskId

Description :

Used internally by the aftmonitor to instantiate its task database and to
setup the data structures relative to a task in order to be able to handle
related threads.
```

```
ft_task_create

parameters :

in

          taskId

          mode <normal | degraded>

Description :

Creates the two threads related to the task (using pthread_create ) the
thread corresponding to the selected mode is made active while the other
is suspended. The internal tasks status database is updated.
```

6.3.2. Internal API

`ft_notify_failed_thread`

```
parameters :

in

          thread_id

          taskId

          failure_parameters

          failure_cause  < error | deadline_miss >

          mode_when_failed

          current_mode

          scheduling_parameters_when_failed

          current_scheduling_parameters

Description :
```

Informs the **aftmonitor** that an abnormal situation occurred on the thread threadId related to taskId and that a new behaviour has been activated. The **aftmonitor** updates its database and checks if this failure triggers a need for mode change. If it the case it calls the ft_switch_appli_mode function.

`ft_switch_appli_mode`

```
parameters :

in

          thread_id

          taskId

          failure_parameters

          failure_cause  < error | deadline_miss >

          task_mode_when_failed

          current_task_mode

          scheduling_parameters_when_failed

          current_scheduling_parameters


Description :
```

Is used internally by the **aftmonitor** on notification of a failed thread. This function applies the algorithm described in the description section of the **aftmonitor** to make effective the new mode and updates its database.

ft_get_appli_mode

parameters :

out :

appliModeId

Description :

Used internally by the **aftmonitor** to get the current active application mode.

AppliModeId is a pointer to a structure that describes an application mode.

The appliMode structure contains two fields :

appli_mode_name

active_tasks_list

(list of tupples(task,behabiour) active in the mode)

Chapter 7. Ftbuilber

7.1. Description

The ftbuilder is an off/line tool. As said the general philosophy section, it will help the user specify the non-functional features of its application in a declarative way. This tool will thus help gathering information about application and fault-tolerant tasks and help build and instantiate the data structures needed for run-time management of fault-tolerance. It will be a simple TCL/TK tool used to enter textual information that will produce files related to tasks.

This information will possibly be used also for off-line analysis by verification components developed by CTU.

7.2. Layer

The tool is at Linux application level.

7.3. API / Compatibility

The tool will use TCL/TK

Chapter 8. Perspectives for the next version

This first version is intended to build a basic framework for fault-tolerance support. Facilities offered are still limited, but the main goal was to set up a consistent environment well integrated inside OCERA architecture and as far as possible compatible with existing RTLinux.

It must thus be considered as a first laboratory to test different strategies of cooperation between OCERA components, RTLinux kernel and user's applications for a proper handling of errors.

The next version will offer richer fault-tolerant strategies and offer new facilities. The main expected improvements will concern :

- More detailed task model
  - Up to now tasks are considered as blackboxes
  - It is necessary to model also communication and access to resources
- Taking into account Linux level tasks
  - Difficulty for insuring predictability of reaction time of a linux level **aftmonitor** requires guarantied bandwidth
- Taking into account distribution
  - Need to integrate node control with ORTE components
  - Integrate redundancy management
  - Introduce synchronization mechanisms