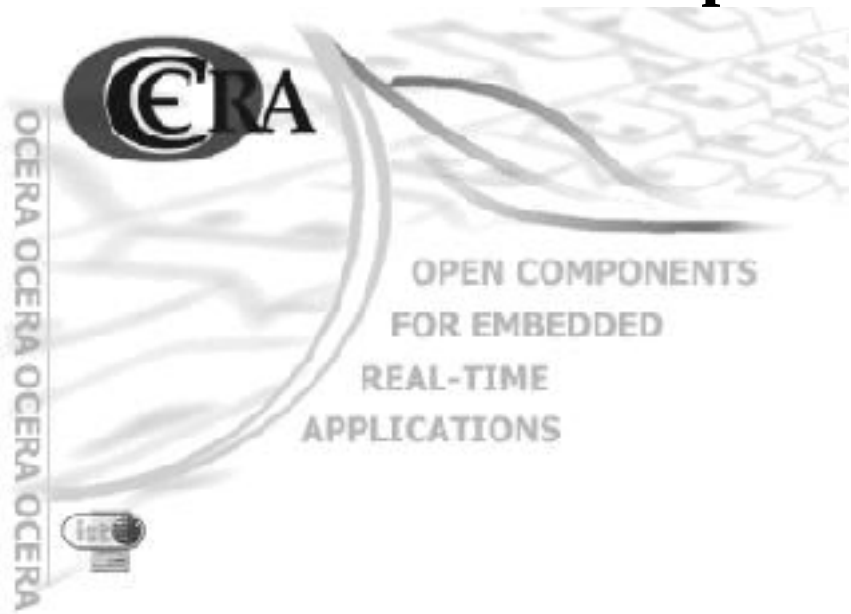
WP6 - Fault-Tolerance Components



Deliverable D6.3 - Design of new fault-tolerance facilities

WP6 - Fault-tolerance components : Deliverable 6.3 - Design of fault-tolerant components
New functionalities by A. Lanusse

Published July 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

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 litterature, 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.

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. During the first phase of the project, the basic fault-tolerant facilities have been settled. During the second phase. New facilities will be added. These new facilities will mainly concern the implementation of redundancy management

1.1. Main objectives

As specified in D6.1 report, 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 has been decomposed into two major classes and will follow the roadmap presented in figure 1:

Mono-node applications.

In this context fault-tolerance consists 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 has been implemented. Tow components, the **ftappmon** and the **ftcontroller** implement degraded mode management respecively at the application and task level.
The second range of facilities will cover recovery blocks management and imprecise computation management. These functionalities will be developed within the second phase of the project.

Distributed architectures.

In this context, we will provide support for redundancy management in multi-node architectures. These developments will exploit the communication components developed within OCERA during the first phase of the project. These new facilities will permit to handle a new type of abnormal situations (nodes failure).

The dispatching of these developments over the time of the project is summarized in fig.1.

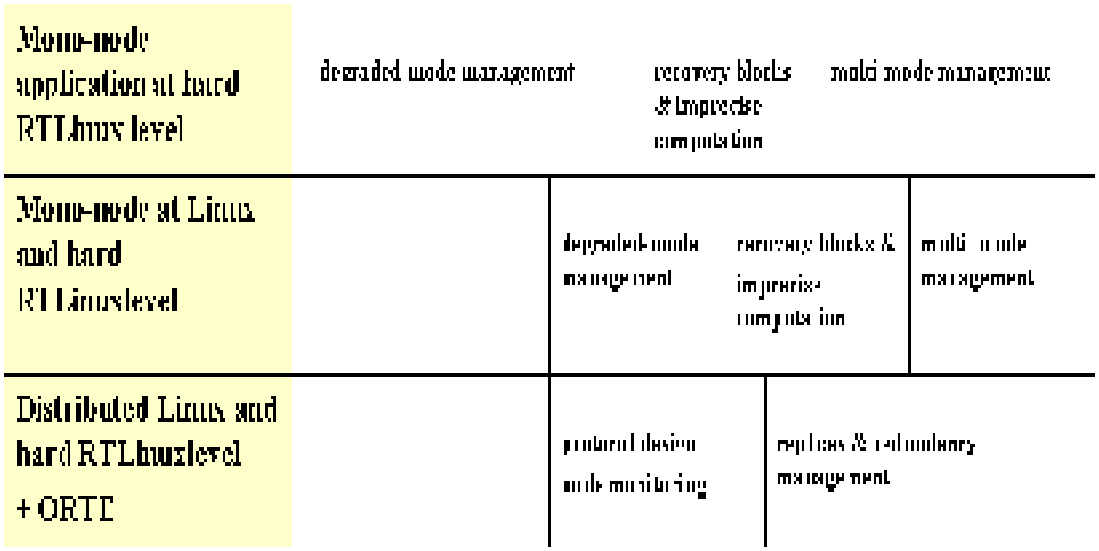


Figure 1. OCERA targeted fault-tolerant facilities

1.2. Current development status

The first stage of the development has consisted in defining an overall fault-tolerant framework well integrated in the RTLinux architecture and two basic FT run-time components. Then a design/build tool has been developed to support the methodology and assist the user to specify its application.

run-time components
Two specific run-time fault_tolerance components have been developed (**ftappmon** and **ftcontroller**) as said above. They provide degraded mode management for applications consisting of hard real_time RT Linux tasks. They are described in document D6.2_rep.

These two components support the notion of FT_task (Fault-tolerant task) an encapsulation of Real-Time task that permit to handle abnormal situations in a smooth and controlled way.
Thanks to predefined alternate behaviors for a FT_task, the service offered by this task can be pursued (even if degraded) in spite of faulty events occurring on the task. Actually the task is implemented through several threads that run alternately. The detailed principles of this computation model have been described in previous documents (see D6.1 and D6.2_rep).

The events currently handled by the implementation are KILL on threads. This implementation uses the **Ptrace** component developed within OCERA in workpackage (WP5). Kill events are periodically checked by the **ftcontroller** which triggers a task behavior switch and notify the **ftappmon** that may in turn trigger an application mode change.
Detection of deadline misses is still under development since it requires the use of components developed during the first stage of the project within the scheduling workpackage (**PosixTimers** and **Application Defined Scheduler**).

The current implementation offers the advantage of being independant from the kernel code itself and developmentscan be done in parallel without deep inter-relation between the two types of components. This will facilitate the maintainability of the overall system.

The fault-tolerance management offered within WP6 is based on the exploitation of

declarative information about the application provided by the user. While the actual application code is almost left the same as for a pure real-time RTLinux application, additional information is required at init time in order to instanciate tables used by the run-time FT components to manage properly abnormal situations. This provides orthogonality between fonctional and non functional features of an application. This way fault-tolerant features or even real-time features can be declared separately from the code which permits easily to test several combinations of parameters without having to rewrite the application (or modify it) each time.

This declarative approach is supported by the introduction of a small number of primitives used to specify these features and init the internal databases of FT components (see rep. D6.2_rep). But the full description of the application using such primitives can easily become fastidious and error prone. It is the reason why a design tool component is required.

off_line design component

A first prototype of design/build tool, the **ftbuilder** component has been implemented. It permits the description of the application tasks and modes and the specification of transitions conditions. Information gathered with the tool is then used to generate application description files used to compile application. The two currently generated files namely the **ft_appli_control_model.h** and **ft_appli_control_model.c** are used respectively to declare tasks, modes and transitions and to init databases using specific API primitives.
Moreover, the **ftbuilder** permits to check the consistency of tasks and modes declarations.

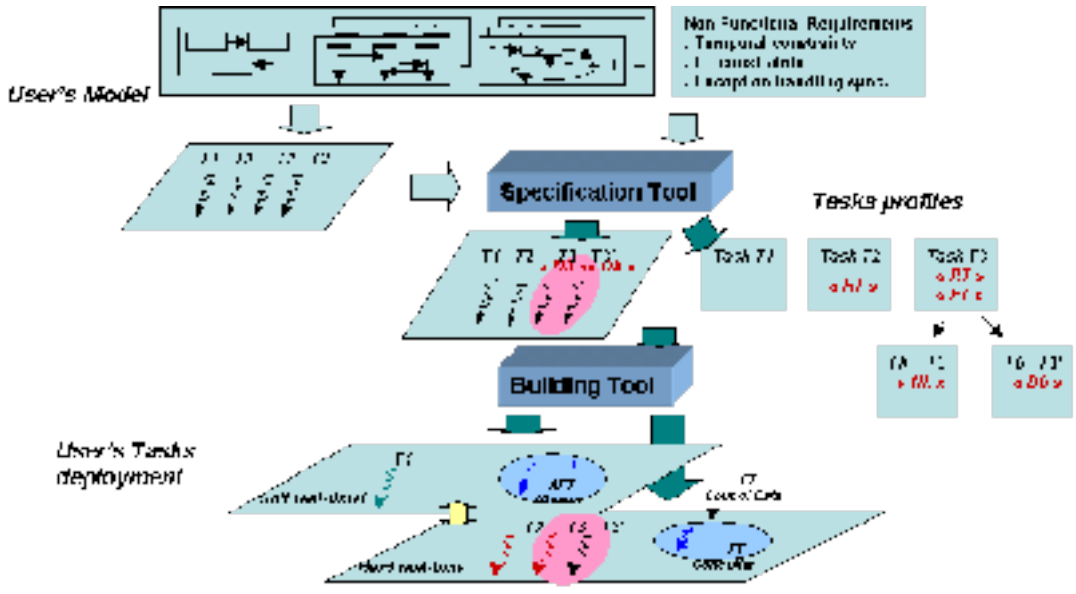


Figure 2 . Declarative approach and supporting tools

The specification tool permits the user to specify for each task the related temporal constraints, the different possible alternative behaviours and the transition conditions for switching behaviour. It also describes application modes and transition conditions for application mode switch. The modelling of the application relies on the generic task model described in document D6.1. The development of this component is still undergoing but a first prototype version is available in the ocera archive. A detailed description of the tool will be provided within a revised version of D6.2_rep that will be delivered at the end of September.

1.3. New FT functionalities for version V2

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 V1 components

Extensions to first components will be provided so as to handle both hard-realtime and soft realtime tasks. They will concern three apsects :

Handling of both Hard real_time RTLinux level and Linux level tasks.

This will require that each of 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 **ftappmon**. The implementation of OCERA components providing a hierarchized CBS scheduling is thus a prerequisite for this step.

Higher level strategies of degraded mode management will be tested

Mainly recovery blocks and imprecise computation management.

Extensions do design/build tool

In the second phase of the project, the design tool will be enriched in order to help define which tasks will be redunded and their level of redundancy. This will imply also that deployment information is given in order to specify tasks location over nodes. Extended code generation will also be provided to support applications implementation and distribution management.

Development of new V2 components

In addition to that, specific new components will be added in order to provide new facilities to handle redundancy management in distributed environments.

They will consist in : a fault-tolerant redundancy manager (**ftredundancymngr**) and a fault-tolerant replica manager (**ftreplicamngr**). They will respectively control redundancy at the application level and at the task level on each node.

They will be implemented both at Hard real-time and Linux level as shown in fig. 3 . They are described in the next sections.

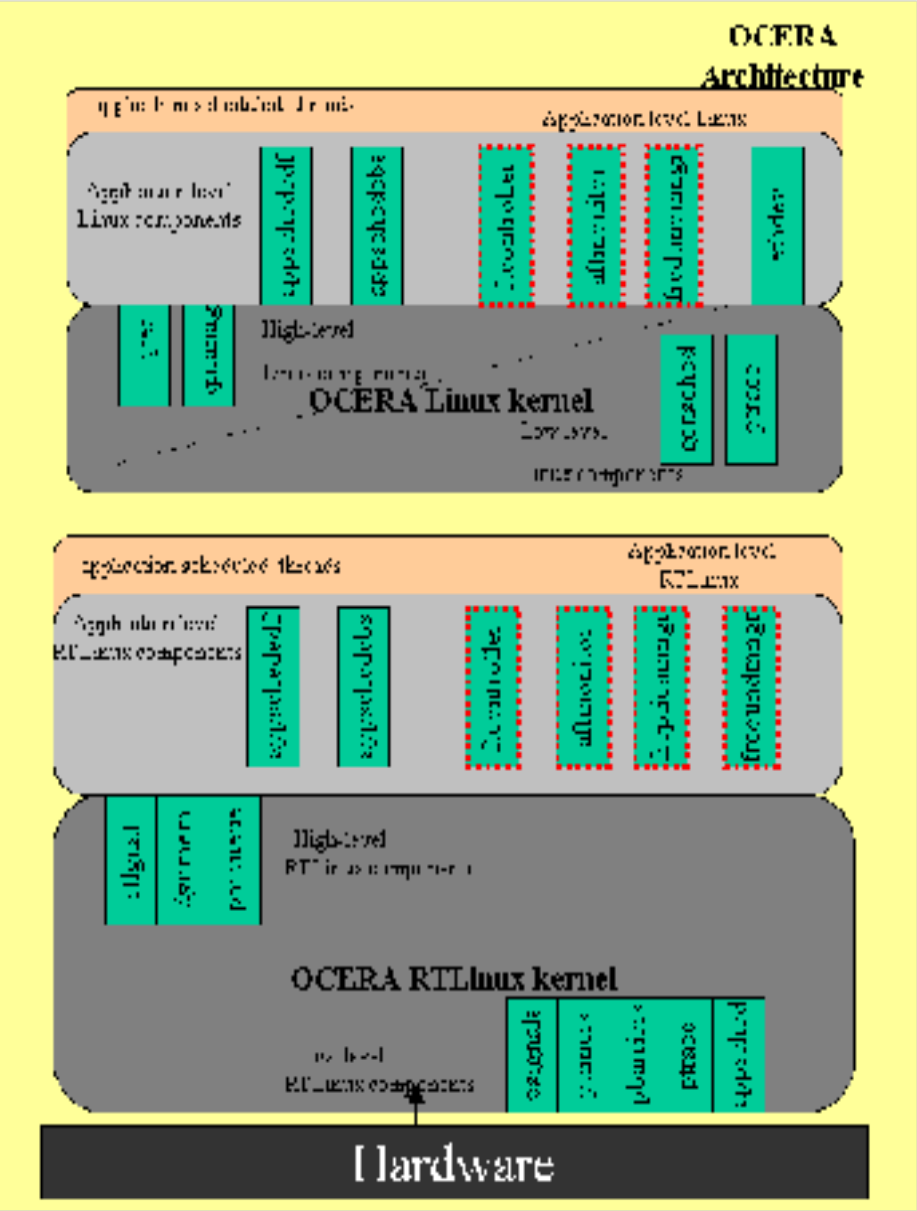


Figure 3 . Fault-tolerance components in OCERA architecture (V2)

Chapter 2. Redundancy management in OCERA : main principles

Redundancy management is a very classical way of providing fault-tolerance to applications (see state of the art on fault-tolerance in D3.3 report).

There are mainly three types of redundancy management currently used :

active redundancy model

passive redundancy model

semi-active redundancy model

The most well known and also most widely used model is the active M:N model where M replicas of a same task run in parallel and a voting system validates the results as long as at least N replicas are still valid.

In semi-active redundancy model replicated tasks run in parallel with the same interactions inputs from other tasks or environment, but only one task is said to be active and can propagate results and interact with other tasks. When the active task fails one of its replicas take over the activeness. Since all the redundant tasks run the same code in parallel, the replica context and state is immediately valid.

In the passive redundancy model, only one task is active and interacts with other tasks, other replicas do not run, but the state of the active task is transmitted to replicas in order to maintain global state consistency over an observability criteria.

2.1. Design choices for redundancy management

Within the OCERA project, we have chosen to implement the last redundancy model, that its the passive model. This choice is driven by the fact that we are targetting real-time embedded applications with restricted resources. This type of redundancy management however implies that a deterministic synchronization mechanism insures state consistency between replicas. The concept of replica context is then a very important notion.

In OCERA, the fault-tolerant model has been build in order to satisfy constraints of certain classes of real-time applications where a large part of the tasks are periodic acquisition or control tasks and where execution time of a task is relatively small regarding period. It is thus possible to define a synchronization mechanism exploiting these characteristics which are not necessary true for any kind of real-time applications but represent a large percentage of real-time applications.

The model retained assumes that one period of execution of a task lost is not dramatic for the application. The idea is that we can take as much as one period of a task to recover from a failure. This assumption which may seem rather dangerous is coupled with a default strategy implementation, that is that if a task must produce a given result at a given time during a period, a default value will be provided if the task fails and a replacement behaviour will take place for the next and further periods. When redundancy is implemented, a replica will take over the failed task and will be ready for the next period.

This strategy give us some laxity to implement checkpointing mechanisms which are generally time consuming and difficult to implement in a more general asynchronous

context.

2.2. Redundancy management model

Introducing redundancy consists in duplicating some or all the application tasks in order to be able to maintain a quality of service in case of faults or failures.

If we consider software faults, redundancy may be achieved through source diversification of code; that is multiple implementation of a same service through different versions. This type of redundancy is not treated here.

When hardware failure or kernel crash is feared, hardware is redunded as well as network and tasks code. This is this type of redundancy that will be tackled in the second phase of the project. It is thus intrinsiquely a distributed system.

Redundancy model

- An application consists of a set of tasks.
- Each task may have several replicas (that constitutes a group)
- Replicas of a task are located on different nodes (only one replica of a given task on a given node).
- A node is a computer plus an OCERA kernel
- Communication between nodes is deterministic (bounded communication time)

Types of faults handled

Two types of faults will be considered :

- kernel crash
- node or communication crash

The software failure of a task is handled by the mode change mechanism developed in the first phase of the project and supported by the **ftcontroller** and the **ftappmon**. The replica of such a failed task is not activated (but the mode change is propagated).

The two types of faults are fail silent failures. In both cases it can be detected by a timeout. A typical way of handling this is to send periodically lifeness signals, that if not received in time reveal a misfunctionning either of the distant node or of the communication mean.

Byzantine errors are not considered within the scope of the project.

Application development using redundancy facilities

During design two important steps are followed :

first the specification of required redundancy is done for each task. Redundancy may be required for some important tasks and not for other

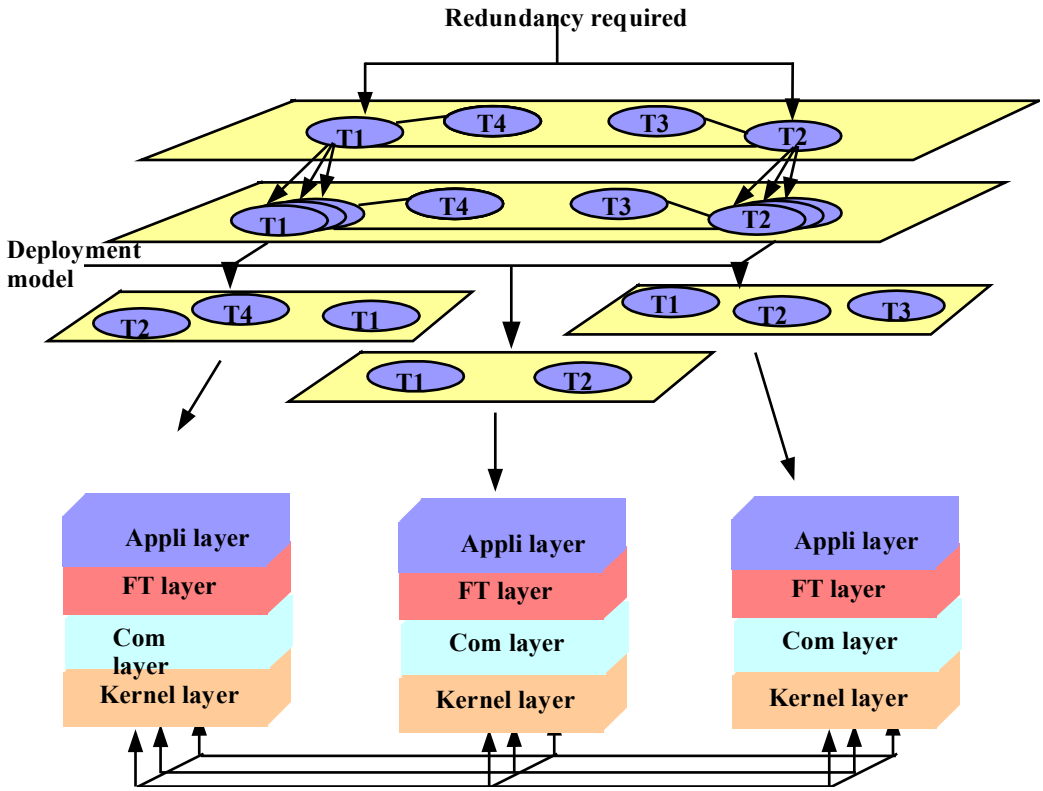
then, the specification of deployment is done. The deployment consist in partitioning the application into separate spaces that will be implemented onto separate machines.

These components rely on the communication components that insure deterministic and reliable real_time communications.

Indeed, the specification of fault-tolerant tasks must be consistent with task dependencies

and location.

The general design process is illustrated hereunder in fig. 4.



Figuure 4. Specification to deployment of redundant application.

With information gathered during design, , the data bases of the redundancy components included in the FT_layer will be instanciated.

The FT layer for redundancy insures several facilities :

It controls the state of the system and a notification mecanism is in charge of the detection of node or network failure.

It insures the periodic update of tasks state of groups of redundant tasks.

It maintains the knowledge of current application configuration and is in charge of the reconfiguration of redundancy in case of partial failure of the system. When a node crashes, the redundant tasks that were active on the crashed node are replaced by one of the remaining redundant tasks of the group. Application topology is updated, and state recovery for crashed tasks is performed. The choice of the new active task is done arbitrarily.

The two first functionalities are insured by the **ftreplicamnger** while the last one is insured by the **ftredundancymngr**.

Chapter 3. Fault-tolerance components architecture V2

3.1. Run-time components for redundancy mode management

3.1.1. Architecture overview

The redundancy management involves implementation at both Hard RT level and Linux level

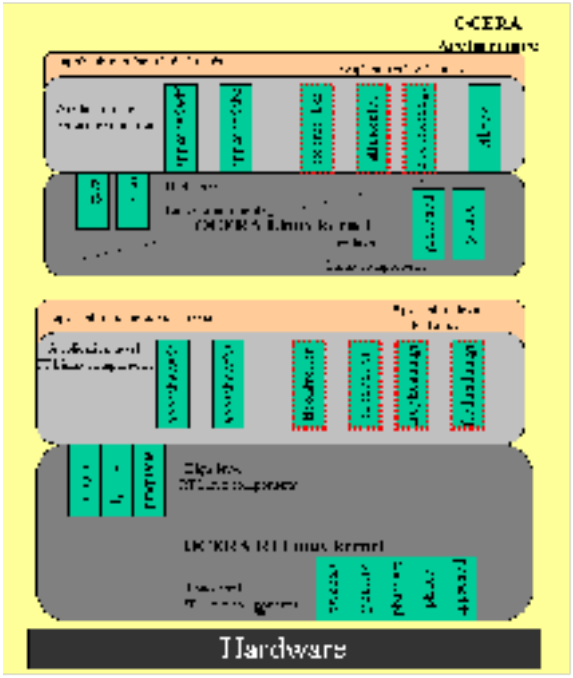


Figure 5 . FT architecture overview

The implementation of redundancy management requires two OCERA RTLinux components located at the application level on each machine of the network.

- a Redundancy managar (**ftredundancymngr**) in charge of the global application monitoring and redundancy policy for the groups of tasks controlled
- a Replica manager (**ftreplicamngr**) in charge of the low level control of the checkpointing of tasks contexts and the monitoring of system state. It is this component which also notify the ftredundancy manager of abnormal situations.

These two components cooperate, especially when situations evolves due to a kernel or a node crash.

There is one instance of each component on each node of the network. The redundancy managers on each machine has a complete knowledge of application current configuration and constraints, so that it can take decisions in an autonomous way if necessary.

The replica manager maintains a table of all duplicated objects and the role of the current

instances located on the node. If the instance is a passive replica. The replica manager, regularly checks that context checkpoint is performed. If the instance is an active replica, it initiates the protocole for state context transfer. This protocole must insure reliable atomic transfer to all memebers of the group of replicas.

3.1.2. Basic interactions between components

At init the **FT redundancy manager** set ups tables of configuration of the application.

The application is a set of FT_tasks. Some of them are requiring redundancy. The set of replicas of a same task constitutes a group that will have to be managed. The FT redundancy manager stores them along with the configuration of tasks (i.e. their location and their belonging to a group).

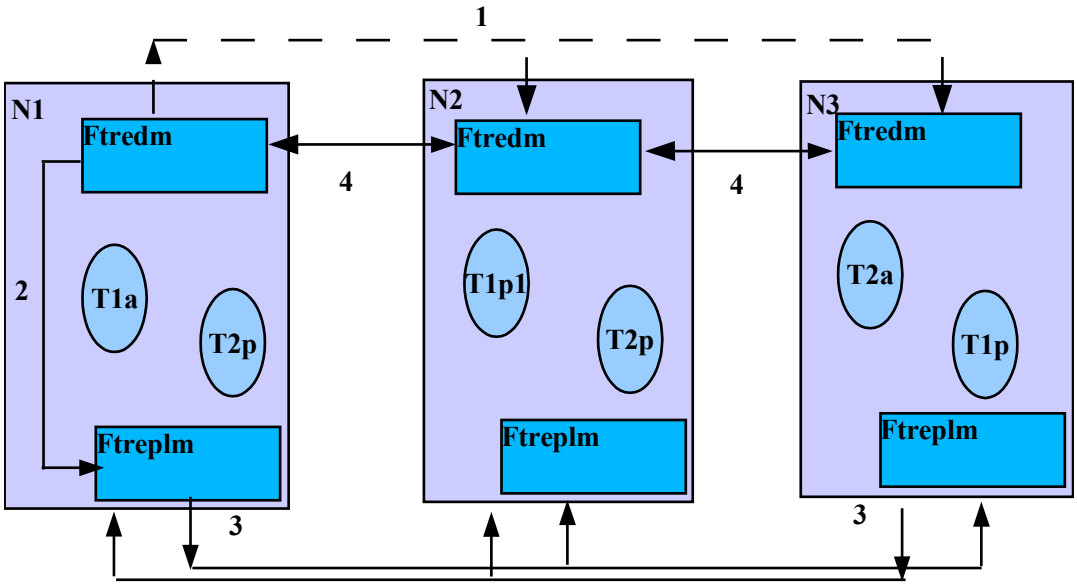
The temporal characteristics of the tasks are also stored since synchronization of checkpoints depends on these characteristics.

Then the FT redundancy manager determines the active redundant replicas. This is done on on the main node.

Then the setup of the network is done and checked.

Then information is propagated to other replica managers other the network.

Each FT redundancy manager sets up then starts its local replica manager .



1. **Init : configuration of Ftredm on each node**
2. **Init : configuration of Ftreplm locally**
3. **Checkpointing from active task to passive replicas**
4. **Lifeliness control**

Fig. 6. Interactions at init and on normal situations

Once this is done, the system is ready to start the application. In this example two redundanst tasks T1 and T2 are running. The active replica for T1 is on Node N1 while the

active replica for T2 is on node N3.

This initialization step of the overall infrastructure, must be carefully designed and a specific protocole is being defined to control it properly.

Once the application is running, the two components almost do not interact except for a lifeliness notification periodically done in order to verify that the other component is still alive.

The **FT replica manager** reguraly insures checkpointing between replicas for all the replicas present on the node. It also monitors the availability of the network and the lifeliness of distant connected nodes.

If a node crashes, notification is made by the Ftreplicamngr or diectly detected by the ftredundancymngr and tasks reconfiguration takes place. The ftredundancymngrs installed on the surviving nodes agree on new actives replicas and reset control data of the ftreplicamngr.

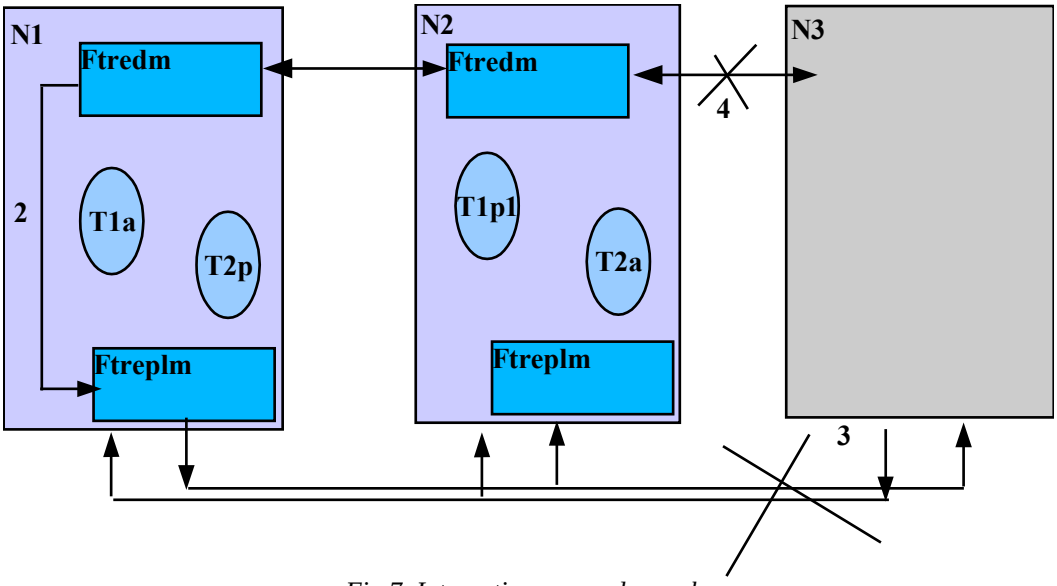


Fig.7 Interactions on node crash

The crash is detected either by ftreplicamangr by a deadline on checkpointing or by the lifeliness management in ftredundancy managers. The redundancy managers check the configuration of tasks located on crashed node and update their group database. Then they agree on a new active replica for each active task previously present on the faulty node.

Finally they update information of ftreplicamngrs so that groups definition is accurateand active tasks are specified.

The ftreplicamngr can then switch passive tasks to active and refresh diffusion lists of context for a given task.

In the example above , the replica of T2 located on N2 becomes active and application continue with two nodes and only one passive replica for each task..

Interactions with other FTcomponents.

The **ft appmon** and the **ftcontroller** also reside on each node. The **ftappm**on has a global view of application and application modes. It is replicated on each node.

The **ftcontroller** is in charge of controlling the whealthiness of the tasks threads of the system. When a thread crashes, this is handled through task behaviour switch and not by the take over of a replica. A degraded mode is activated, and this behaviour change is notified to other replicas at checkpointing time. The replicas will also change their behaviour.

3.1.3. Synchronization principles

Synchronization is the main point as far as replica management is considered. Most critical systems are based on time_triggered d eterministic synchronization model. A purs asynchronous model is eligible on applications such as telecom or networks where time determinism is not so important. However in this case it implies more complex algorithms to insure determinist c atomic broadcast.

In our model we are between those two extremes . The periodical model gives us a bounded synchronization network. Replica take over can only occur at the period following the failure of tha active replica. Checkpointing is performed after the end of the execution of the current iteration of the task and befor the next period.

Detection of node crash is done trhough a mechanism of watchdog implmented in the replica manager.

As said in document D6.1, 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 servoing 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.

3.1.4. Task context

As said earlier the passive replica management relies on the principle that there is an active replica that runs on one node and that this active replica synchronize with other replicas in order to maintain a consistent context so that if a replica has to take over the leadership on failure it can do it starting from a consistent and up-to-date context.

In our model we have made the choice to guarranty that observable state of a task is consistent.An observale state is a state reached at the end of the execution of an instance of a periodic task. The internal intermediate evolution of local variables is not considered. Thus the state is constituted of the set of values of its global variables.

Communication between tasks is supposed to be performed at the end of the excution of the instance of the task. During an execution the task possibly uses results from other tasks produced during previous periods. This model though asynchronous is close to time-triggered approaches.

Chapter 4. FTRedundancy manager

4.1. Description

The **ftredundancy manager** component consists of one controlling thread and two databases.
A database, the AppliControl Database stores information on replicas groups. This database stores the location of each replica of a group.
The other one stores the system configuration satus (nodes and status of nodes and the identifiers of distant replica managers)

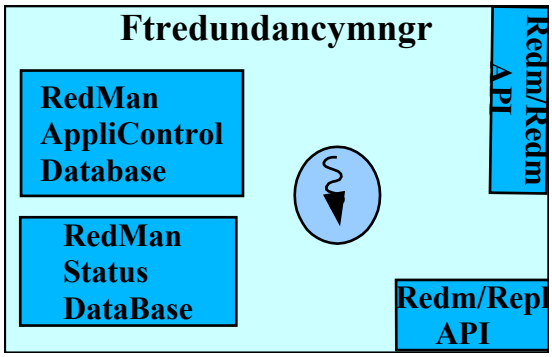


Figure 8 . ***ftredundancymngr*** *internal view*

The main role of the redundancy manager is achieved during init, it sets up the system infrastructure and installs and configure the ft replica manager. During execution its role is less active. It reacts to node crash by defining the next active replica for a replicas group. As said above, it also notify its wealthiness periodically.

4.2. Layer

This component is located at the Application RTLinux Level.

4.3. API / Compatibility

This component uses posix compatible RTLinux API for threads manipulation and OCERA components API (ptrace, psignals, ptimers, pbarriers, appsched).

It also uses WP7 communication components that will ensure predictability of communication.

The API can be divided into two subsets. An API for communication between **ftredundancymngr**s (ftred/ftred API) located on different nodes and an API for communications with the **ftreplicamngr(FTred/ftrep API).**

This API will be refined during the detailed design of the component.

4.3.1. The inter ftredundancymngr API (ftred/ftred API)

ft_set_init_ControlDataBase

parameters :

in :

ftredundancymngrId

Description:

Used by the ftrepalicamngr to setup the Contro DataBase

ft_notify_I_am_alive

parameters :

in : ftreplicamngr_Id

date

Description :

Used by the ftredundancymngr to control liveliness of the system. Periodic signal sent

ft_update_ControlDataBase

parameters :

in : DataStructure

Description :

allows database update

ft_notify_node_crash

parameters :

in :

nodeId

ftreplicamngr_Id

Description :

Used by the ftredundancymngr to notify a node crash

ft_notify_node_started

parameters :

in :

node_Id

ftreplicamngr_Id

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.

4.3.2. API between ftredundancy mngr and ftreplicamngr (ftred/ ftrepl API)

ft_init_ft_task_group

parameters :

in :

task_group structure

Description :

Used by the **ftreplicamngr** instanciate internal DataBase of ftreplicamngr. This call sets the group configuration for a FT_redunadnt_task. It specifies the members of the group and their location and their current activity status(active or passive).

ft_switch_ft_task_group_config // Init of task

parameters :

in :

task_group structure

Description :

Used internally by the **ftreplicamangr** to change active and passive configurations of replicas.

ft_set_task_activity_status

parameters :

in :

taskActivityStatus

Description :

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

The task status tells if a task is an active or a passive replica.

ft_get_task_activity_status

parameters :

in :

task_Id

out :

taskActivityStatus

Description :

Used get the task activity Status.

Chapter 5. FT replica manager

5.1. Description

The **ftrepliacamngr** component consists of one controlling thread, a data base for replica control management, this database is separated int two parts : one for the local active replicas and one for local passive replicas. This DataBase contains information on active replicas temporal application and context.

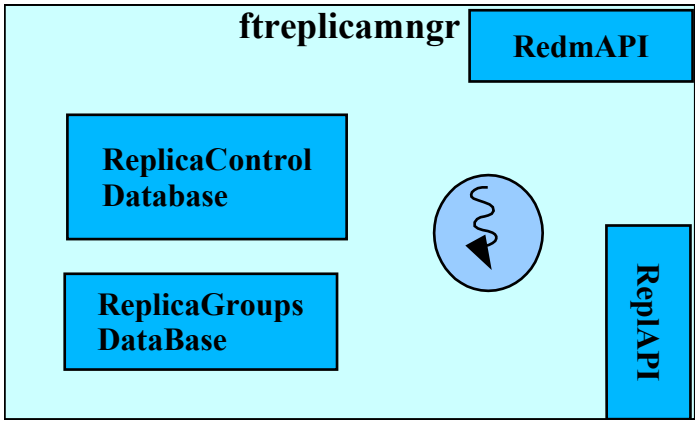


Figure 9 . ***ftreplicamanager****: internal view*

The ReplicaGroups Database is used to specify groups characteristics members , location and active or passive status.

At each end of a periodic execution, the ending replica awakes the replica manager that will transmit the new context to other members of the group.

In parallel, the replica manager arms a timer corresponding to each deadline of passive replicas. If this deadline is reached while no context has been received, a specific connexion check is performed. If network is functionning correctly, the replica manager informs the redundancy manager that collects information from other nodes and will decide to change the active replica if possible.

Finally, the replica manager surveys the status of the network and notify the **ftredundancymngr** in case of pbs.

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. A few additional primitives have been defined.

As it is the case for the ftredundancymngr, this API can be divided into two subsets. An API for communication between **ftreplicamngmngr**s (**ftrepl/ftrepl API**) located on different nodes and an API for communications with the **ftredundancymngr(FTrepl/ftredAPI)**

5.3.1. API between ftreplicamngrs ftrepl/ftrepl API

ft_task_new_context

parameters :

in

task_name

context struture

date

out

Description :

Used to send a new context at the end of active replica task execution

5.3.2. API between ftreplicamngr and ftredundancymngr (ftrep/ ftred API)

ft_notify_failed_node

parameters :

in

node_id

Description :

Informs the **ftredundancymngr** that a failed node has been detected(failure can be detected at both levels).

ft_notify_replicamngr_ready

parameters :

in

Description :

Is used to inform the ftredundancymngr that the replicamngr is ready to start working.

Chapter 6. Ftbuilber

6.1. Description

The ftbuilder is an off/line tool. As said in 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.

A first prototype version has been implemented and is used to enter tasks, modes and mode transitions specifications.

6.2. Layer

The tool is at Linux application level.

6.3. API / Compatibility

The tool uses TCL/TK.