WP6 - Fault-Tolerance Components

Deliverable D6.4_rep - Fault tolerant components V2

WP6 - Fault-tolerance components : Deliverable 6.4_rep - Fault-tolerant components V2
by A. Lanusse and P. Vanuxeem

Published March 2004

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	26/03/04	First release

Chapter 1. Introduction

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 first series of facilities have been described in previous deliverables (see D6.1 and D6.2_rep).

The fault-tolerance components included in this deliverable consist of two complementary components (**ftredundancymgr** and **ftreplicamgr**) that together provide a framework for implementing redundancy management support for user's application. They will respectively control redundancy at the application level and at the task level on each node.

This first implementation is intended to provide a basic framework whose goal is to offer a global set of facilities that permit transparent implementation of redundancy for developers of real-time applications. It offers a passive replication model, the task model is a simplified one (periodic tasks), fault-detection is based on heartbeats and timeouts, consistency of replicas is ensured by periodic checkpointing.

The current implementation is located at Linux user-space level using ORTE component for communication between nodes. However implementation choices have been made in such a way as to facilitate the port to OCERA Hard Real-Time level when ORTE become available at this level. Indeed these facilities can be enriched in the future.

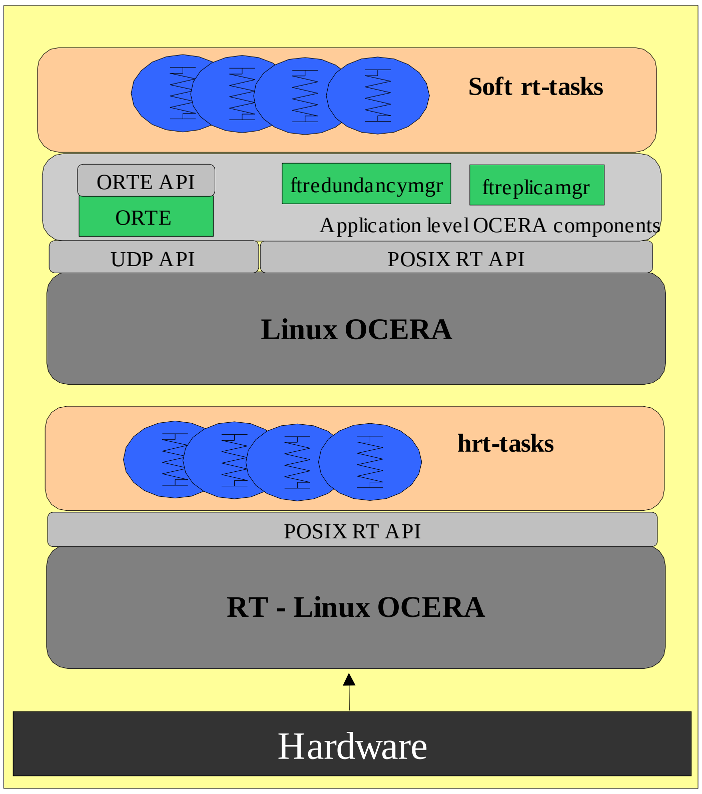


Figure 1.1 FT Redundancy management components location within OCERA architecture

Before entering in the details of the components architecture we describe briefly, the application and tasks model used, then we introduce the main principles of functionning of the overall architecture.

1.1 Application and tasks model

An application consists of a set of ftr_tasks (fault_tolerant redundant tasks).

In order to support data consistency and to facilitate tasks recovering on node crash, a task model must verify synchronisation properties. In the current implementation, we have introduced the following task model.

1.1.1 ftr_tasks

A ftr_task is a real-time periodic task (same parameters as pthread scheduling parameters).

All ftr_tasks are periodic, there is no other temporal synchronization than the periodicity of the task. The basic cycle of a ftr_task instance is the following.

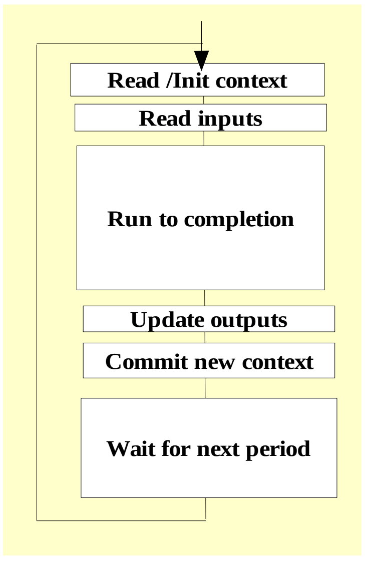


Figure 1.2 ftr_task execution model

A context object is defined for each ftr_task, this context contains static variables which are changed during a period and whose change is significant for the task behavior of next periods. This context is saved at each end of execution of an iteration. It is then broadcasted to the task group of replicas, so that one of them can become the new master task and start with a valid context in case of node crash.

The application developer must define the set of variables which must be part of the context at design time. This context object is automatically updated and broadcasted at each end of cycle.

Communication with other tasks is limited to reading and writing data in predefined shared objects. Reading is done at beginning of the period, writing is done at the end of the period.

These objects have only one writer, visibility of data is enabled to other tasks after the completion of the code of the task, at the beginning of the new period. (Which means that tasks are working on data obtained during the previous period of the writer's task).

When defining ftr_tasks, it is required to specify :

- the structure of its ftr_context object;
- the ftr_shared_data objects that will be used as inputs;
- the ftr_shared_date objects that will be written by the ftr_task (only one writer per ftr-shared_data object).

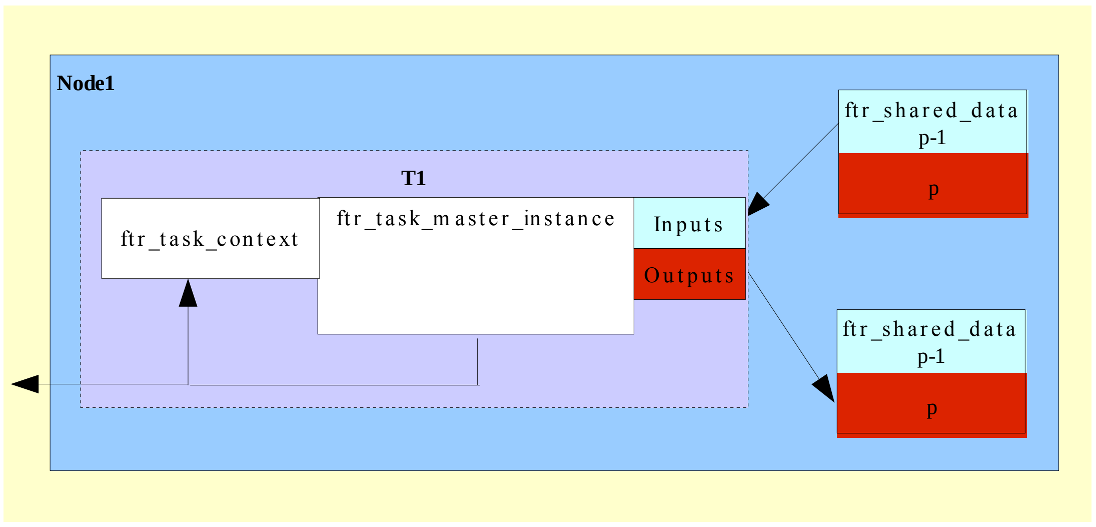


Figure 1.3 ftr_task and ftr_shared_data relationships

1.1.2 ftr_tasks_group

The redundancy management model adopted is a passive replicas management model.

Redundancy parameters have been introduced in the ftr_task data structure. These parameters include :

- the redundancy level required for the ftr_task (minimum and maximum number of replicas);
- the location of each replica;

From these information, and for each ftr_task, an ftr_tasks_group is defined which gathers data on :

- ftr_task_master location and status
- for each ftr_task_replica of the ftr_task_master its location and status
- current valid context of ftr_master_task (from last period)
- temporal information of ftr_task_master (date of cycle start, deadline, date of cycle end).

1.1.3 ftr_tasks group management main principles

Tasks group management is insured by the **ftreplicamgr** which periodically broadcasts the new context emitted by the ftr_master_tasks if execution cycle completed successfully.

Moreover, if the ftr_master_task is writer of a ftr_shared_data, the ftr_shared_data new value is also broadcasted to other nodes at each end of cycle.

Temporal behavior of ftr_master_task execution is controlled and notification of error is done to **ftredundancymgr** in case of deadline miss. If necessary a replica is elected as new master and the previous master is deactivated. The selection of the new master is deterministic, it is simply the ftr_task_replica located on the next available node (in an ordered list of nodes).

1.1.4 Simple example of redundancy management over two nodes

In the following simplified example, the application is composed of two ftr_tasks implemented on two nodes. The two master replicas for tasks T1 and T2 are located on Node1 and two slave replicas are located on Node 2. T1 and T2 periodically (at each end of cycle of each task) transmit their contexts (CT1 and CT2) to **ftreplicamgr** which broadcasts them to members of ftr_tasks_groups of T1 and T2 (in this simple case only to T1:s1 and to T2:s2). Moreover T2 is producer of ftr_shared_data SD1, so SD1 is also propagated to Node2.

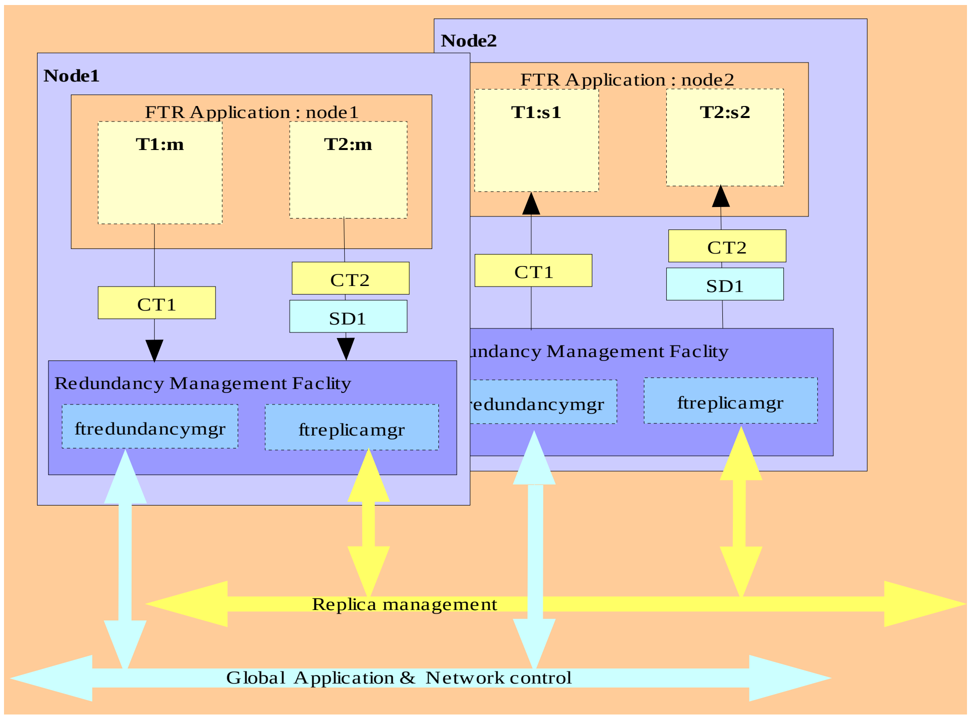


Figure 1.4 Simple example of redundancy management over two nodes

The **ftredundancymgr** controls global network, detects possible node crash and decide of dynamic reconfiguration when such an event happens. Information on application status is thus also replicated within each node. Such transmission of information is totally transparent to the user.

1.2 FT redundancy management architecture overview

As viewed in the previous section, the implementation of redundancy management requires two OCERA RTLinux components located at the application level on each machine of the network.

- a Redundancy manager (**ftredundancymgr**) in charge of the global application monitoring and redundancy policy. This component is in charge of application initialization and control of overall distributed architecture. It also performs node crash detection through lifelines control using heartbeats. On detection of such failure, dynamic reconfiguration of application is activated. New master tasks are elected in order to replace tasks which were located on the faulty node. Low level control of execution of tasks is delegated to a replica manager which is in charge on insuring consistency of groups of redundant tasks (see below).
- a Replica manager (**ftreplicamgr**) in charge of the low level control of the tasks. Tasks groups are defined with a master and several slaves depending on the redundancy level required for the task. Tasks are all periodic tasks, only the master task of a group is active, at each end of cycle, checkpointing is performed. The new context of the task is then broadcasted to all replicas of the task. If a timeout is detected on a periodic task, a notification is issued to **ftredundancymgr** which will then test if corresponding node is still alive and decide about action to be undertaken. (Change current master task to an other one or use default action, or detect node crash and reconfigure all application).

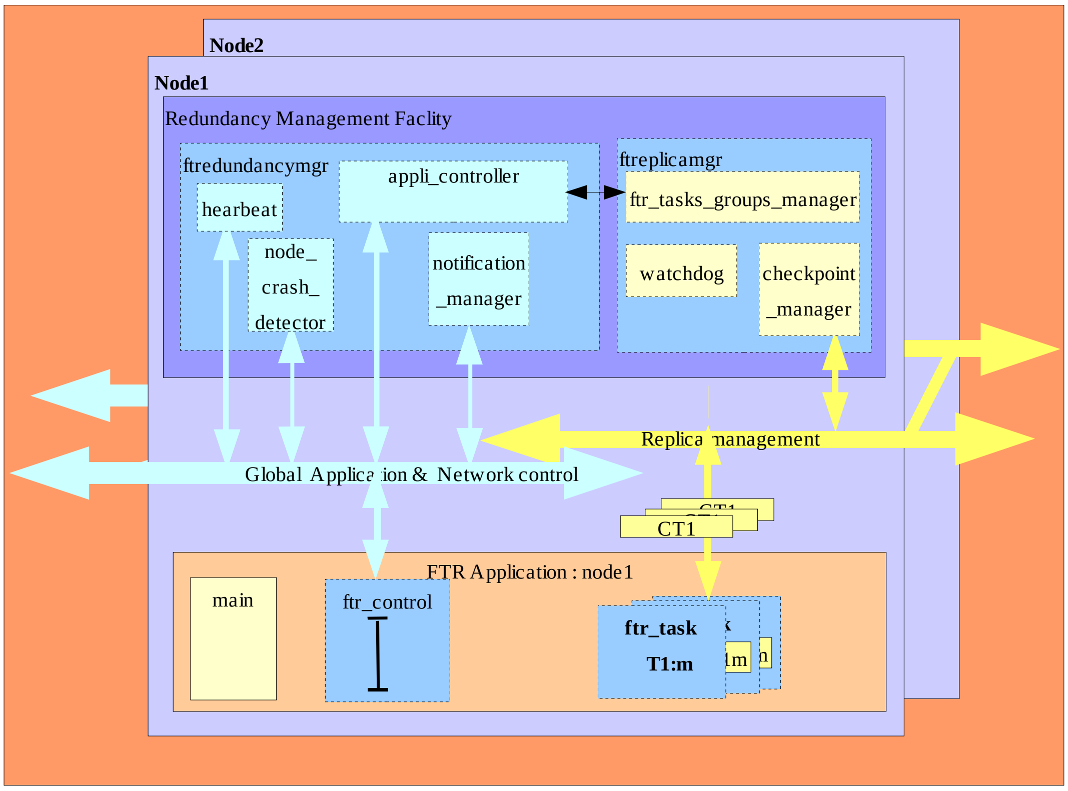


Figure 1.5 Global FT Redundancy Management Architecture overview

We give a rapid overview of this overall framework functioning principles in the following two sections.

1.2.1 Architecture of redundancy management facility on a node

On each node, the two components (**ftreplicamgr** and **ftredundancymgr**) are implemented as two separate threads that cooperate within a Linux process in user's space.

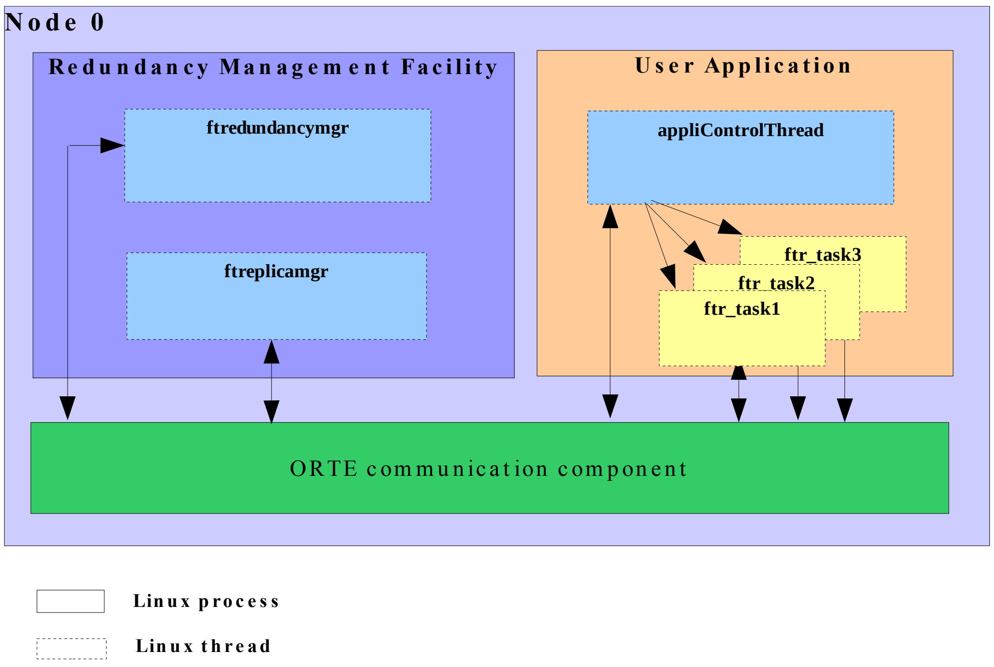


Figure 1.6 Overview of RedundancyManagement Architecture on a node.

The user application runs in a separate Linux process. In the current implementation, all application tasks are implemented as threads within one single application Linux process. (This choice has been made in order to be closer to the future implementation at Hard RTLinux level where all threads share the same space).

Within this process, an application control thread is created at application init, it is in charge of application tasks creation and communication with the ftredundancy management facility. This thread is transparent to the user which uses a dedicated API to create and run application tasks.

An application task is encapsulated within a ftr_task which insures periodic control of the task and checkpointing (communication with the **ftreplicamgr**).

Communication between processes and between nodes are handled by ORTE communication component.

There is one instance of each component on each node of the network. The redundancy manager 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 managers maintain a table of all ftr_tasks_groups and maintain the current status of each member of a group (master, passive_replica, unavailable_replica) and its location. Each active replica, periodically sends its new context which is then broadcasted to all other members of the group. The protocol must insure reliable atomic transfer to all members of the group of replicas. The replica manager, regularly verifies that context checkpoint has been performed on time.

1.2.2 Basic interaction between components

The main interactions between components are illustrated in following figure where the two instances of components located on a node appear as two separate threads within one Linux process, services offered are shown within oval forms. This framework is present on each node of the distributed architecture required for the application.

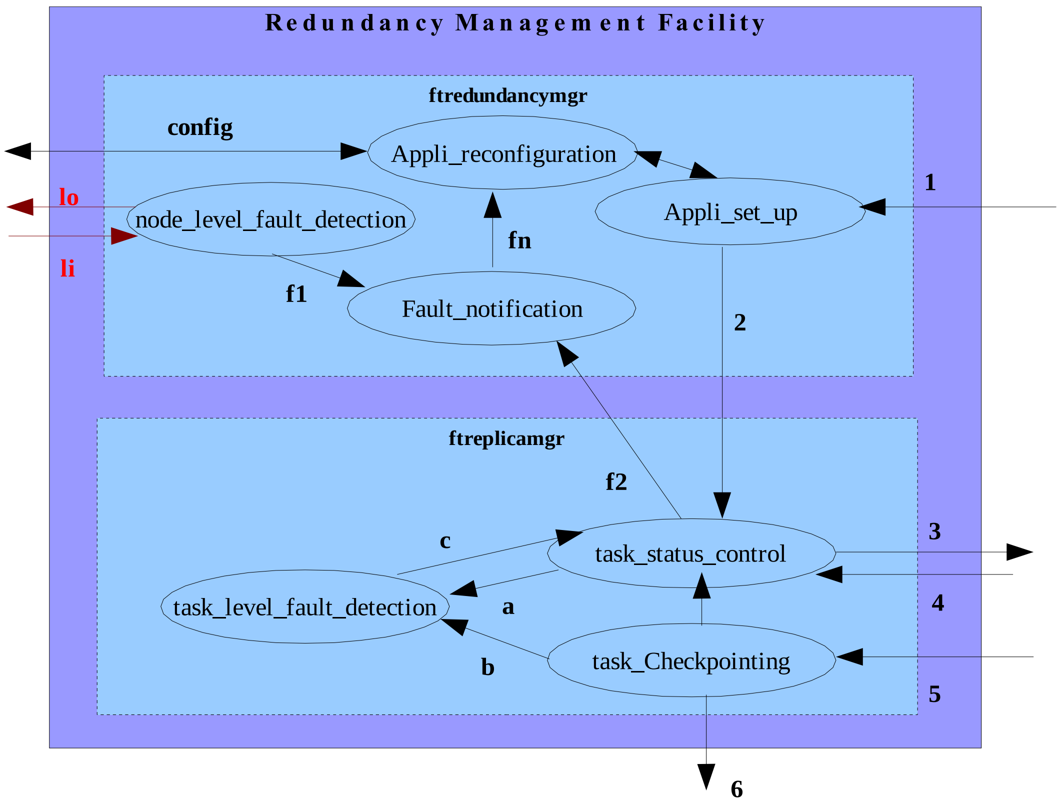


Figure 1.7 . Internal and external interactions of ft_redundancy components

Main functioning loop:

1 : application init

2 : tasks groups creation

3 : task creation, start or end

4 : task cycle started

5 : task cycle ended + context or task end

6 : checkpoint towards passive replicas

Task level fault detection :

a : task started + deadline

b : task ended

c : timeout reached task not responding

f2 : task j on node n not responding

Node level lifeliness control :

li : lifeliness in (node_i is alive) received from each node

lo : current node is alive sent

f1 : node i not responding

In this figure three types of protocols are shown, the first one concerns the main functioning loop, the second concerns node crash detection and related reconfiguration process, the third one concerns timeout detection at task level, the task may be faulty but not the node.

1.2.3 Application life-cycle

Previously to application start, the redundancy management must be made available. A script shell permits the installation of the two components onto a set a specified nodes. Once **ftredundancymgr** and **ftreplicamgr** are installed and ready, one can start an application.

At init, application is started in master mode on one node which is called node_0, information on application configuration is provided to the **ftredundancymgr:**

- number of tasks and tasks descriptions (including redundancy level parameter for each task),
- number of nodes and nodes_Id
- initial mapping of tasks onto nodes
- ordered list of nodes for dynamic reconfiguration on node crash (determines node replacement choice)

According to this information, the **ftredundancymgr instantiates** its internal application description table and remotely starts application on other nodes in slave mode.

It then builds tasks groups (master task + its replicas) and sends information for each group to the **ftreplicamgr** whose role is two control the functioning of each task and to maintain the consistency of all replicas for each tasks group.

The **ftreplicamgr** instantiates its own tasks groups table and task control status table, then enables creation of tasks and tasks replicas on each node (actually task creation itself is achieved within the User Application process by the internal application control thread, the task may be created as active or passive).

When all tasks are created on each relevant node, start of tasks is enabled. Each task thread becomes ready and is then started according to its mealtime parameters and to scheduling policy. At each cycle, start of cycle is notified to **ftreplicamgr**; at each end of cycle, end is notified and the checkpointing of the new task context is achieved. A watchdog verifies timeliness of task completion and a fault notification is issued in case of deadline miss on a task.

On application termination all tasks are terminated, then application instances are finished on each node and application is unregistered. The redundancy management facilities stay available for a new application or may be ended by a specific command.

1.2.4 Faults management at task level

The **ftreplicamgr** located on each node controls the execution of each master replica, namely: start-time, end-time, and timeliness of transmission of context.

If a deadline miss occurs on a ftr_task (the master did not transmit its context on time), a new master is elected amongst the corresponding ftr_tasks_group and the faulty one is terminated.

The **ftreplicamgr** notifies the **ftredundancymgr** of the fault, this latter then updates its new tasks configuration and broadcasts it to each node.

The **ftreplicamgr** located on the node where the new master task will from now on be located, switches the ftr_task_replica on and makes it run in master mode instead of slave mode. It will start at the next period (P+1) of the ftr_task with the last valid context (context P-1) . Several strategies can be envisaged to provide smoother behavior to the application, but for the moment only this rather drastic solution is implemented (one period is lost).

1.2.5 Faults management at node level

The **ftredundancymgr** of each node periodically sends a lifelines message to all other nodes, a node_failure_detection mechanism checks arrival of these messages.

A silent node is considered as faulty and retrieved from the set of available nodes. All active tasks on that node are switched off and a new replacement master task is elected for each one. The process of election is deterministic (using the ordered list of valid nodes). If it is not possible to find a new master task then the current default action is to end the overall application.

As said previously, this first implementation provides a global framework build on top of OCERA Soft RT level, all the application tasks are periodic tasks.

Though the implementation of these components was initially intended to be developed at both Hard and Soft mealtime levels, the current version has been implemented using results of OCERA available at end of phase1 before full integration be ready. The implementation at hard RT level should however be easily ported to Hard real-time level when ORTE components at this level are available.

1.3 User's view

1.3.1 Implementation principles

Implementation principles are driven by the will to make redundancy management as transparent as possible to the application developer. So in order to develop an application, the user can almost forget about underlying ft redundancy management architecture.

To support the approach, two features are introduced and used within the user's process :

- creation of a control thread dedicated to redundancy control (ftr_control_thread)
- encapsulation of application tasks into ftr_tasks_threads

The ftr_control_thread is in charge of initialization and control of application. Created within the user application process it communicates with **ftredundancymgr** and **ftreplicamgr.**

The ftr_tasks_threads are generic encapsulation of redundant tasks. A ftr_task_thread is created for each user's application redundant task. It ensures periodic execution of user's task routine, management of context entity and of shared data entities and communication with **ftreplicamgr** for checkpointing.

Communication with **ftreplicamgr** and **ftredundancymgr** are achieved using ORTE publisher/subscriber mechanisms both within a Node and between nodes, but this is transparent to the user since calls are made either from ftr_control_thread or from ftr_tasks_threads generic part using specific internal APIs that are described in the corresponding component sections below.

1.3.2 User's API

The approach chosen results in a very limited user's API necessary mainly for initialization and termination of user application. Most of user' s application code consists in routines that will be run within ftr_tasks_threads.

```
int ftr_application_register(char *, FTR_APPLI_DESC * ,
                             ManagedApp *);

int ftr_appli_desc_init(FTR_APPLI_DESC *);

int ftr_appli_task_create(FTR_APPLI_TASK_DESC *);

int ftr_appli_task_end(int );

int ftr_application_terminate(char*);
```

FT redundancy management User API

The important issue is to specify the context data and shared resources for each task at design. Concurrency control over such shared data is then automatically insured by the execution model. Then threads routine can be written simply in a usual way.

In the following figure we illustrate on a very simple example how an application is started.

Once the design is done, the resulting architecture on a node is composed of the user's process and of the Redundancy Management Facility process (in the following view we do not show ORTE process).

Within the user's process the yellow (or white) parts concern code written by users and blue (or gray) part concern generic ftr code.

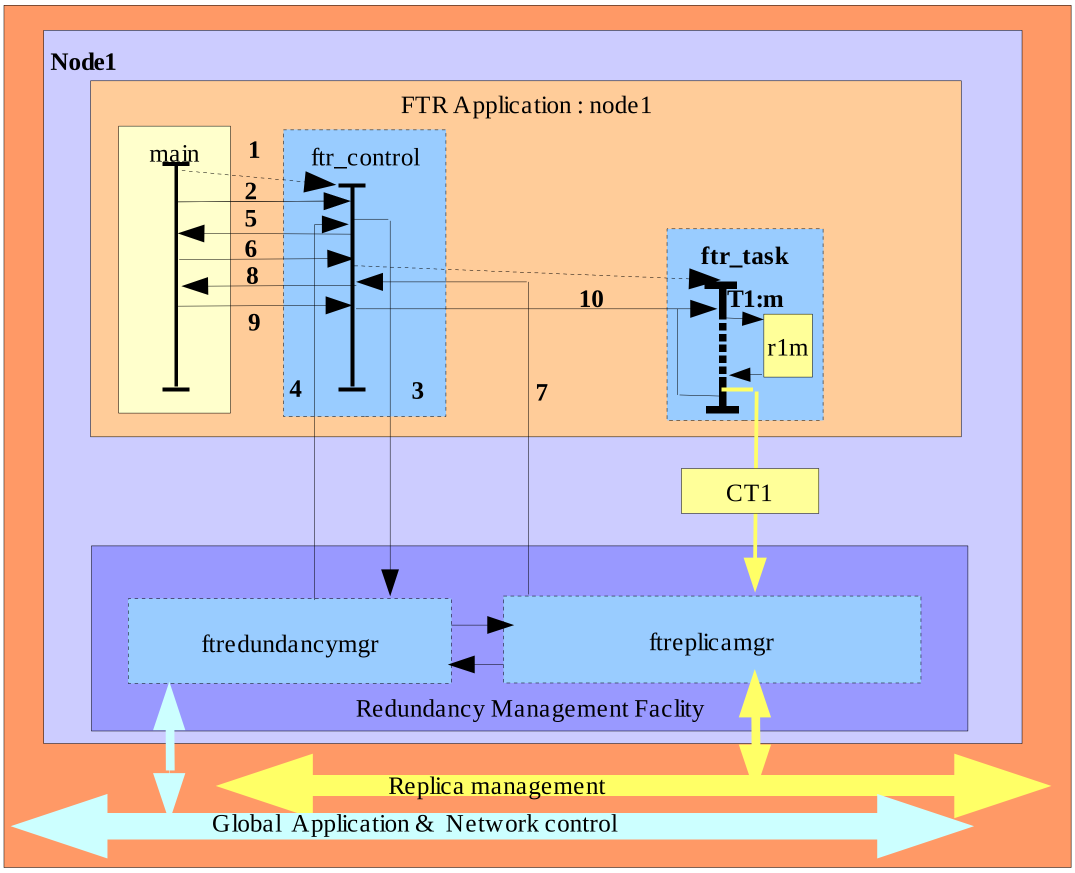


Figure 1.8 Interactions with redundancy Management facility components form User's Application

First the application creates the ftr_control_thread (1), then it calls the **`ftr_application_register`** primitive to register the application (2), the ftr_control_thread then communicates with the **ftredundancymgr** to setup data (3) for the new application, and waits for acknowledgment (4) from it before returning OK (5) to the user main thread.

Then the **`ftr_appli_desc_init`** primitive is called to setup application data structures and ftr_tasks_threads (6). At this step ftr_tasks_threads are created but the corresponding users routines are not started. When all the infrastructure is ready, the **ftreplicamgr** notifies the ftr_control_thread (7) which returns OK (8) to user's main thread.

Finally the user can call the **`ftr_appli_task_create`** primitive to start a ftr_task.(9). The ftr_controller_thread then makes the ftr_task_thread start periodic call to the corresponding user's ftr_task_routine (10).

Two other primitives are available to end an ftr_appli_task (**`ftr_appli_task_end`**) and to terminate the overall application(**`ftr_application_terminate`**).

The user has to define specific data structures, one to describe the overall application structure and one to describe each ftr_task.

It is intended that the **Ftbuilder** tool (already available for the specification of degraded mode management) will assist the designer to determine these features and automatically generate the corresponding data structures. For the moment this facility is not implemented yet, and data is provided in a file read by the **ftr_appli_desc_init** primitive.

1.3.3 Coding steps

An application can be written rather simply following the different generic steps :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <orte.h>
#include <netdb.h>
#include <pthread.h>
#include <simple_appli.h>
#include <ftredundancymgr.h>
#include <appli_controller.h>



ManagedApp *appli;

pthread_t ftr_control_thread;

int main(void)

{

  int res = 0;

  void *ret;

  FTR_APPLI_DESC application_desc;

  FTR_APPLI_TASK_DESC application_task_desc_1;

  FTR_APPLI_TASK_DESC application_task_desc_2;
```

1. Declarations for ftr_application

```
 /* Creation of ftr_control_thread */

  pthread_create(&ftr_control_thread, NULL,
(*ftr_main_control_routine),
                  NULL);

  if (res != 0) {
      perror("Redundancy Management thread creation failure ...
              exiting");
      exit(-1);
    };
```

2. Creation of ftr_control_thread of ftr_application

The ftr_control_thread of the application is created in the beginning of the main thread to install the ftr architecture within the application process. In the future, it will be replaced by a macro. The ftr_main_control_routine, is a generic control loop that monitors events from and to the ftr_process. It also accepts requests form the user main thread.

```
 /* Init appli_desc structure */

  res = ftr_appli_desc_init(&application_desc);

  if (res == -1)    {
  perror("Redundancy Management : application desc init failed ...
             exiting");
      exit(-1);
    };
```

3. Initialazition of application data structures

During this step, data structures describing application and tasks are initialized.

```
  /* Register application */

  res = ftr_application_register(APPLI_NAME,
&application_desc,appli);

  if (res == -1) {
      perror("Redundancy Management : application registration
failed...
              exiting");
      exit(-1);
    };
```

4. Registration of application

Application registration is done towards ftr process which in turn propagate information over network (thanks to ORTE) to other ftr processes. (Application is also registered as ORTE Application). (Internal tables are initialized, groups of replicas are created and instances created on each node).

```
   /* Tasks creation */

  application_task_desc_1 = application_desc.appli_tasks_tab[1];

  application_task_desc_1.appli_task_routine = ft1;

  res = ftr_appli_task_create(&application_task_desc_1);

if (res == -1) {
      perror("Redundancy Management : task creation (1) failed...
              exiting");
      exit(-1);
    };

...
```

5. ftr_tasks creation for ftr_application

During this step each application task is created using the ftr_task_desc of each one. This steps defines mainly the routine to be run within the generic ftr_task_thread and the related real-time parameters (period, estimated_duration, deadline). At the end of each period, the current context is sent to all its replicas on other nodes.

Once this is done for each task, the application runs in a nominal way.

To end a task the following call is necessary.

```
  /* Requiring End of Task 1 */

  res = ftr_appli_task_end(1);

...
```

6. ftr_tasks ending for ftr_application

This ends the corresponding ftr_task (and all its replicas). All ftr_tasks have to be ended before application itself can be ended.

```
 /* Requiring Application Termination */

  ftr_application_terminate(APPLI_NAME);


 /* Waiting for end of control_thread */

  pthread_join(ftr_control_thread,&ret);

  if (ret != PTHREAD_CANCELED) {
      i = (int) ret;
      printf("Main : end of ftr_control_thread ret = %d\n", i);
     };


 printf("\nAppli ending : ");
 return 0;

}
```

7. Termination of ftr_application

Once all the ftr_tasks are ended, resources are freed and the ftr_control_thread is ended, then application terminates.

Obviously, the user must in addition provide the code of the routines that will be run within each ftr_tasks_thread. A pointer to this routine is a member of the ftr_task_desc structure.

In our simple example :

```
int ft1(int i)

{

  printf("Function ft1 running with arg %d\n",i);

  sleep(3);

  return 0;

}
```

The status of the current implementation is still in a testing phase. The example implemented tests application setup, execution and termination.

1.4 Overview of Redundancy Management API

The Fault-Tolerance components described in this document have to be used jointly since they interfere strongly. It is the reason why though each one has its own API described in a distinct section, it may be usefully to get a general overview of them.

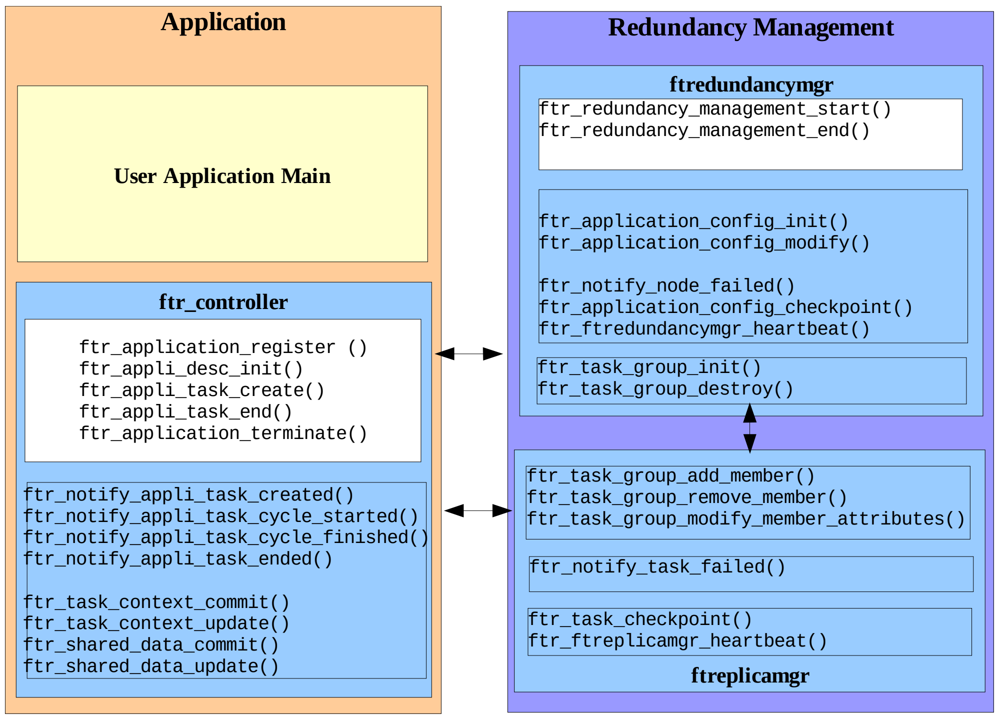
The external user API is actually restricted to very few functions :

- **ftr_application_register()**,
- **ftr_appli_desc_init(),**
- **ftr_appli_task_create()**,
- **ftr_appli_task_end(),**
- **ftr_application_terminate()**

They are called within user's main application thread and handled by the ftr_control_thread (named hereafter **ftr_controller**) running within the application process. Then the **ftr_controller** uses internal API to communicate with **ftredundancymgr** and with **ftreplicamgr**.

The **ftredundancymgr** has a small external API that is used to start or end the redundancy management facility.

In addition, each component has also internal API(s) that permit interactions between them.


Application
User Application Main
ftr_controller
ftr_application_register ()
ftr_appli_desc_init()
ftr_appli_task_create()
ftr_appli_task_end()
ftr_application_terminate()
ftr_notify_appli_task_created()
ftr_notify_appli_task_cycle_started()
ftr_notify_appli_task_cycle_finished()
ftr_notify_appli_task_ended()
ftr_task_context_commit()
ftr_task_context_update()
ftr_shared_data_commit()
ftr_shared_data_update()
Redundancy Management
ftredundancymgr
ftr_redundancy_management_start()
ftr_redundancy_management_end()
ftr_application_config_init()
ftr_application_config_modify()
ftr_notify_node_failed()
ftr_application_config_checkpoint()
ftr_ftredundancymgr_heartbeat()
ftr_task_group_init()
ftr_task_group_destroy()
ftr_task_group_add_member()
ftr_task_group_remove_member()
ftr_task_group_modify_member_attributes()
ftr_notify_task_failed()
ftr_task_checkpoint()
ftr_ftreplicamgr_heartbeat()
ftreplicamgr
Figure 1.9 Global view of ft redundancy management API(s)

The user's API is described hereunder while next chapters describe in details the two FT components.

1.4.1 FT Redundancy Management external API

ftr_appli_desc_init

Parameters :

in

    application_desc

Description :

Used to initialize locally (within user's process) data strutures related to a new application (nodes configuration, ftr_tasks_groups, ftr_tasks_descs).

Prototype :

```
extern int ftr_appli_desc_init(FTR_APPLI_DESC *);
```

ftr_application_register

Parameters :

in

    application_name

    application_desc

    ORTEhandle

out

    application_id

Description :

Used to register the new application and make it handled by ftredundancy management. Among arguments, it is necessary to provide the ORTE handle to the managed application that is used for communications.

Prototype :

```
extern FTR_APPLI_ID ftr_application_register(char *,
                             FTR_APPLI_DESC * ,
                             ManagedApp *);
```

ftr_appli_task_create

Parameters :

in

    ftr_appli_desc

 out

    ftr_task_id

Description :

Creates the ftr_task entity (generic thread implmenting periodic ftr behavior) with real-time and redundancy management parameters provided by the user within the ftr_appli_desc structure.

The internal tasks status database is updated.

Prototype :

```
extern FTR_TASK_ID ftr_appli_task_create(FTR_APPLI_TASK_DESC *);
```

ftr_appli_task_end

Parameters :

in

    ftr_task_id

Description :

Used by application to end an ftr_task and its associated replicas.

Prototype :

```
extern int ft_task_end(FTR_TASK_ID ftr_task_id );
```

ftr_application_terminate

Parameters :

in

    application_id

Description :

Used to terminate application. Norammly all ftr_tasks have been terminated before calling it.

Prototype :

```
extern FTR_APPLI_ID ftr_application_terminate();
```

Chapter 2. ftredundancymgr component

2.1 Summary

- Name : **ftredundancymgr**
- Description : **ftredundancymgr** along with **ftreplicamgr** are two complementary components that provide transparent redundancy management for real-time applications. Redundancy policy implemented is based on a passive replication model. The **ftredundancymgr** is in charge of global application and network initialization and control (including node crash detection). The **ftreplicamgr** is in charge of tasks level control: tasks groups and tasks replicas management, checkpointing.
- Author (name and email) :
  A. Lanusse (agnes.lanusse@cea.fr)
  P. Vanuxeem (patrick.vanuxeem@cea.fr)
- Reviewer :
- Layer : application Linux Level
- Version : V0.1
- Status : test
- Dependencies : ocera V1.0 ORTE component and requires ftreplicamgr component
- Release date : M24

2.2 Description

The **ftredundancymgr** insures redundancy management at global application level.
The role of the **ftredundancymgr** component covers the five following points:

- it sets up the redundancy management infrastructure and installs and configure the **ftreplicamgr**,
- it sets up the initial configuration of application
- it monitors the overall architecture (node crash detection)
- it achieves dynamic reconfiguration of application in case of node crash.
- it terminates and performs clean-up at application end.

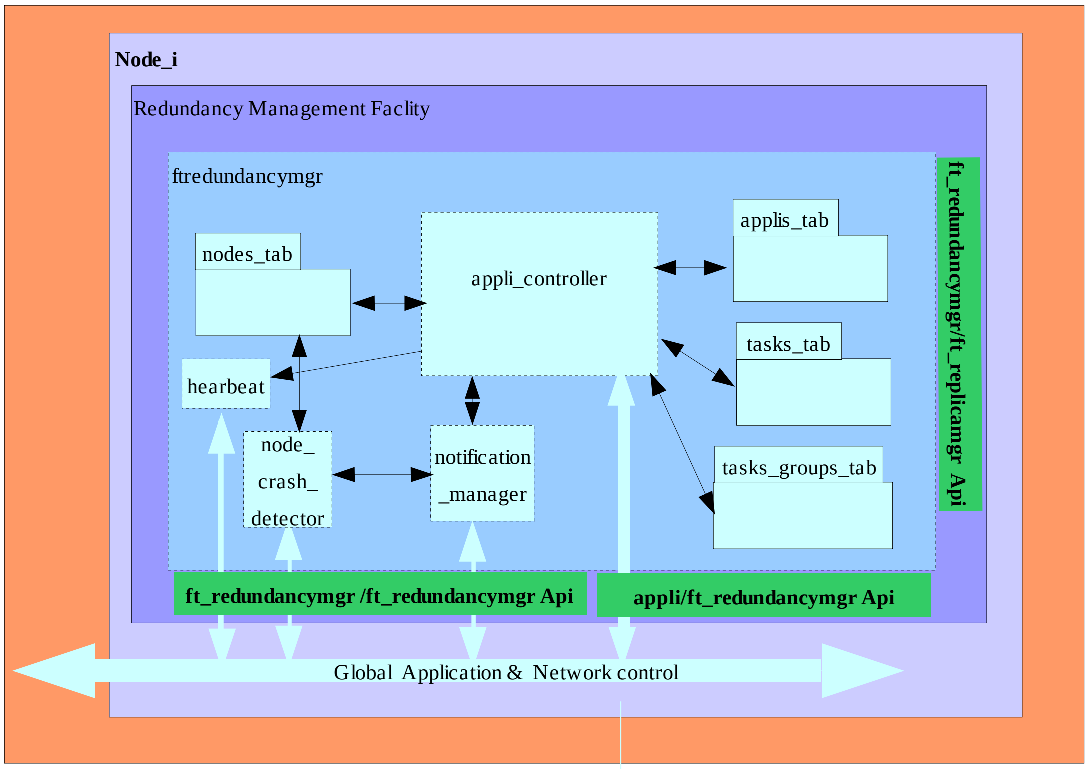
ftredundancymgr and **ftreplicamgr** insure together the redundancy management facility on a node. They have to be started before the application.

They are replicated on all the nodes defined in the configuration at start of the facility. The starting node is considered as master node, the others are semi-active replicas (see below).

Communication between them and with their corresponding component on other nodes relies on ORTE Publisher/Subscriber communication model. This permits easy broadcast of application data or application status over the different nodes. This communication is transparent to the application developer.

2.2.1 Ftredundancymgr internal structure

We describe here the internal structure of the **ftredundancymgr**. For feasibility reasons, entities appear without their prefix (ftr_).



Node_i
Redundancy Management Faclity
ftredundancymgr
nodes_tab
appli_controller
applis_tab
tasks_tab
hearbeat
node_ crash_ detector
notification _manager
tasks_groups_tab
ft_redundancymgr/ft_replicamgr Api
ft_redundancymgr /ft_redundancymgr Api
appli/ft_redundancymgr Api
Global Application & Network control

Figure 2.1 ftredundancymgr : internal view

The **ftredundancy manager** component consists of one main controlling thread that insures applications control and dedicated threads that are in charge of node control. These latter threads insure respectively, heartbeat service, node_crash detection and notifications management.

A nodes database **nodes_tab** contains information on all nodes handled by the redundancy management facility. (That is static nodes description data and nodes control status data).

The nodes_tab is created by the **ftredundancymgr** prior to application start. The database is initialized, heartbeat is started along with node_crash detection and notification management.

The appli_controller thread is a FSM waiting for application events to occur, and reacting accordingly.

On application registration, it creates a new entry in the **applis_tab** and instantiates the corresponding data structure. An application description contains both system configuration data (numer of nodes and their names) and data related to its tasks (number and tasks_descriptions).

From these information entries in other databases are created and completed with the description of the applications tasks and application tasks_groups (location of master and slave replicas for a task).

The **tasks_groups_tab** stores information on replicas groups. For each group, information on task redundancy parameters, task status and on location of replicas is maintained.

Once this is done, information is propagated to **ftreplicamgr** in order to instanciate tables related to groups management.

When **ftreplicamgr** is ready, registration is completed and an acknowledgement is provided to the application by the **ftredundancymgr**.

The application can then start creating tasks. This protocol is handled by the **ftreplicamgr** and task_status and group_status is propagated to **ftreplicamgr** (see chapter3).

If a node_crash is detected, the **notification_manager** analyses the event in relation with information coming from other nodes, if node crash is confirmed, the **ftredundancygr** defines a new configuration of tasks groups (election of new tasks master replicas among replicas groups for each master located on the faulty node) updates its data bases and propagate information to **ftreplicamgr.**

2.2.2 Redundancy management of Redundancy management

As said above, the two **ftredundancy management** components are replicated on all nodes. It is thus necessary to precise how this replication is handled.

The code of each of these component is unique and executed on each node. The only difference between nodes is that one is considered as master for redundancy management facility. A Flag determines the status for each node.

The starting node is the master. It is this node that is responsible for data bases initialization, when a new application is created and for databases updates as long as the node doesn't crash. These databases are maintained consistent by broadcast of changes. Except for this role, the rest of the functioning of the components is the same.

In the same way, the tasks checkpointing process insures the consistency between internal tasks_groups databases of different nodes.

When a node crashes, reconfiguration decisions are made by the master **ftredundancymgr** and propagated to others.

If the master node crashes, the next valid node in the nodes_table becomes the master node and performs dynamic reconfiguration of tasks located on this node.

2.3 API / Compatibility

This component uses posix Linux API for threads manipulation and ORTE OCERA component API for communication between nodes.

We have already introduced the User's API in chapter1, we describe here API that have been defined in order to make the various **ftredundancy management** components cooperate.

The API related to **ftredundancymgr** can be divided into three subsets :

- an API for communication between application and **ftrundancymgr(appli/ftred API).**
- an API for communication between **ftredundancymgr**s (**ftred/ftred API**) located on different nodes,
- an API for communication with the **ftreplicamgr(ftred/ftrep API)**,

Appli/ftred
ftr_redundancy_management_start()
ftr_redundancy_management_end()
ftred/ftred
ftr_application_config_init()
ftr_application_config_modify()
ftr_application_config_checkpoint()
ftr_ftredundancymgr_heartbeat()
ftr_notify_node_failed()
ftred/ftrep
ftr_task_group_init()
ftr_task_group_destroy()

2.3.1 ftredundancymgr external API (Appli/ftred API)

This API is limited to start and end of the ftredundancy management facility.

ftr_redundancy_management_start()

```
parameters :

in :

 master_node_IP

 nodes_list
```

Description:

Used to start the **ftredundancymgr** and **ftreplicamgr on each specified node.** The **ftredundancy** components are started on the master node, then on the others. Heartbeat and network failure detection starts. The components are ready to accept application registration.

Prototype :

```
extern int ftr_redundancy_management_start( FTR_NODE_ID,

                                            FTR_NODES_LIST);
```

ftr_redundancy_management_end()

```
parameters :

in :
```

Description:

Used to end the ftredundancymgr and **ftreplicamgr** on each specified node. The **ftredundancy** components are ended node by node, then the master node terminates. The components are ready to accept application registration.

Normally this call should be done when there is no more application running.

Prototype :

```
extern int ftr_redundancy_management_start( FTR_NODE_ID);
```

2.3.2 The inter ftredundancymgr API (ftred/ftred API)

This API is used to permit coordination between the various **ftredundancymgrs** present in the configuration, the different functions are not seen by the application developers.

`ftr_application_config_init()`

```
parameters :
in :
 appli_id
 appli_config
```

Description:

Used by the **ftredundancymgr** to initialize locally the configuration of an application.

Once this is done, it transmits the new tasks_groups to **ftreplicamgr** and commit the new status of its tables to other nodes using **ftr_application_config_checkpoint()**.

Prototype :

```
extern int ftr_application_config_init( FTR_APPLI_ID,
                                        FTR_APPLI_CONFIG);
```

`ftr_application_config_modify()`

```
parameters :
in :
 appli_id
 appli_old_config
 appli_new_config
```

Description:

Used by the **ftredundancymgr** to update locally the current configuration for an application.

Once this is done, it transmits the new task_groups to **ftreplicamgr** and commit the new status of its tables to other nodes using **ftr_application_config_checkpoint()**.

Prototype :

```
extern int ftr_application_config_init( FTR_APPLI_ID,
                                        FTR_APPLI_CONFIG,
                                        FTR_APPLI_CONFIG);
```

ftr_application_config_checkpoint

parameters :

in :

    application_config

    iteration_number

    date

Description :

Used by the master **ftredundancymngr** to broadcast a new configuration to other **ftredundancymgrs.** This is done at each configuration change, whether it is due to task or node failure.

Prototype :

```
extern int ftr_application_config_checkpoint( FTR_APPLI_ID,
                                              FTR_APPLI_CONFIG,
                                              int,
                                              NtpTime);
```

ftr_ftredundancymgr_heartbeat

parameters :

in : ftredundancymgr_Id

    iteration_number

    date

Description :

Used by the **ftredundancymngr** to control liveliness of the system. A periodic signal is sent on the network and received by all the other f**tredundancymgrs.**

Prototype :

```
extern int ftr_ftredundancymgr_heartbeat( FTR_RED_MGR_ID,
                                          int,
                                          NtpTime);
```

ftr_notify_node_failed

parameters :

in :

    ftreplicamngr_Id

    nodeId

    issue_number

    date

Description :

Used by the **ftredundancymngr** to notify a node crash to other nodes. This detection is followed by a reconfiguration phase and a call to **ftr_application_config_modify** for each application.

Prototype :

```
extern int ftr_ftredundancymgr_heartbeat( FTR_RED_MGR_ID,
                                              FTR_NODE_ID,
                                              int,
                                               NtpTime);
```

2.3.3 API between ftredundancymgr and ftreplicamgr (ftred/ftrepl API)

The **ftredundancymgr** and **ftreplicamgr** share the control of ftr_tasks_groups. The **ftredundancymgr** initializes the data structure for the **tasks_groups_tab** and the **ftreplicamgr** populate and updates it as replicas are started, deleted or change their status from slave to master. Each change is propagated to other nodes by the **ftredundancymgr.**

`ftr_task_group_init`

```
parameters :

in :

         ftr_task_desc

 out :

         ftr_task_group_id

Description :
```

Used by the **ftredundancymgr** to instantiate its internal tasks_groups_tab. This call sets the group configuration for a ftr_task using ftredundancy parameters of the ftr_task (redundancy level, redundancy policy, location of replicas). Each time a ftr_task replica is created, or deleted, the ftr_task_group is updated by the ft_replicamgr using the two calls **ftr_task_group_add_member**, **ftr_task_group_remove_member**.

```
Prototype :

extern FTR_TASK_GROUP_ID ftr_task_group_init(FTR_TASK_DESC *);
```

`ftr_task_group_destroy`

```
parameters :

in :

         ftr_task_group_id

Description :
```

Used by the **ftredundancymgr** to destroy an entry in its tasks_groups_tab when a ftr_task and all its replicas are ended.

```
Prototype :

extern int ftr_task_group_destroy( FTR_TASK_GROUP_ID);
```

2.4 Implementation issues

- Modifications to the existing RTLinux or Linux code

This component is a new one, there is no modification to existing Linux component.

- Data structures created.

Main data structures created concern :

- ftr_node,
- ftr_appli_desc,
- ftr_task_desc,
- ftr_task_group.
- ftr_shared_data,
- ftr_task_context,

Data tables are :

- ftr_nodes_tab,
- ftr_applis_tab
- ftr_tasks_tab
- ftr_tasks_group_tab,

Control events defined are :

- ftr_appli_control_event
- ftr_lifeliness_control_event

Structures :

```
typedef struct
{
  char appli_name[NAME_MAX_LENGTH];
  FTR_NODE_ID  node_id;
  FTR_APPLI_ID node_applis_list;int  appli_tasks_nb;
  FTR_APPLI_TASK_DESC appli_tasks_tab[APPLI_MAX_TASKS_NB];
} FTR_NODE_DESC ;



typedef struct
{
  char appli_name[NAME_MAX_LENGTH];
  FTR_APPLI_ID  appli_id;
  int  appli_tasks_nb;
  FTR_APPLI_TASK_DESC appli_tasks_tab[APPLI_MAX_TASKS_NB];
} FTR_APPLI_DESC ;


typedef struct
{
  char appli_name[NAME_MAX_LENGTH];
  char appli_task_name[NAME_MAX_LENGTH];
  FTR_APPLI_TASK_ID  appli_task_id;
  char appli_task_behavior_name[NAME_MAX_LENGTH];
  int (*appli_task_routine)(int);
  FTR_SCHEDULING_PARAMETERS *scheduling_parameters;
  FTR_REDUNDANCY_PARAMETERS *redundancy_parameters;
} FTR_APPLI_TASK_DESC ;
```

Tables :

The FT R Nodes table :

```
FTR_NODE_DESC ftr_nodes_tab[FTR_NODES_MAX];
```

The FT R Applis table :

```
FTR_APPLI_DESC ftr_applis_tab[FTR_APPLIS_MAX];
```

The FT R Tasks table :

```
FTR_APPLI_TASC_DESC ftr_tasks_tab[FTR_TASKS_MAX];
```

The FT R tasks_groups_ table :

```
FTR_TASK_GROUP_DESC ftr_tasks_groups_tab[FTR_TASKS_MAX];
```

New types defined to describe status of various entities:

```
typedef enum FTR_NODE_STATUS {
  FTR_NODE_STATUS_UNKNOWN,
  FTR_NODE_OK,
  FTR_NODE_NOK
} FTR_NODE_STATUS;

typedef enum FTR_TASK_STATUS {
  FTR_TASK_STATUS_NOT_DEFINED,
  FTR_TASK_CREATED,
  FTR_TASK_RUNNING,
  FTR_TASK_TERMINATED
} FTR_TASK_STATUS;

typedef enum FTR_TASK_REPLICA_STATUS {
  FTR_TASK_REPLICA_STATUS_NOT_DEFINED,
  FTR_MASTER,
  FTR_SLAVE,
  FTR_TERMINATED
} FTR_TASK_REPLICA_STATUS;
```

```
New types defined to describe control events:


  typedef enum
    {
      FTR_TASK_NOP,
      FTR_TASK_CREATION_REQUIRED,
      FTR_TASK_TERMINATION_REQUIRED,
      FTR_APPLICATION_TERMINATION_REQUIRED
    } FTR_APPLI_CONTROL_EVENT;

  typedef enum
    {
      FTR_TASK_NOP,
      FTR_TASK_REPLICA_CREATION_REQUIRED,
      FTR_TASK_REPLICA_TERMINATION_REQUIRED,
      FTR_TASK_REPLICA_CYCLE_STARTED,
      FTR_TASK_REPLICA_CYCLE_ENDED,
    } FTR_TASK_REPLICA_CONTROL_EVENT;
```

2.5 Tests and validation

2.5.1 Validation criteria

Validation criteria concern mainly functional qualitative issues .

Application initialization

Verification that all the threads related to FT_tasks are created correctly and that the data structures are updated.

Application running in nominal conditions

Verification that all the threads related to FT_tasks are executed correctly and that the checkpoints are done in time.

Application termination

Verification that the application terminates correctly (all the threads related to FT_tasks are deleted and resources freed correctly).

Application configuration change is effective and correct after node stopped.

Verification that node failure is detected correctly and that a new configuration of tasks is made available and runs correctly.

2.5.2 Test 1

Application initialisation procedure.

The initialization procedure has been tested by a test example described in the **ftredundancymgr/examples/ftr_appli** directory.

2.5.3 Test 2

Application termination procedure.

The termination procedure has been tested by a test example described in the **ftredundancymgr/examples/ftr_appli** directory.

2.5.4 Test 3

Application execution with redundancy management activated.

The nominal execution of a simple application has been tested by a test example described in the **ftredundancymgr/examples/ftr_appli** directory.

2.5.5 Test 4

Application dynamic reconfiguration

This test must achieve a dynamic reconfiguration on detection of a node_failure.

This test has not been fully achieved yet.

2.5.6 Results and comments

Tests are still on going. At this stage, they are common to the two components.

The first facilities tested concern initialization procedures, nominal functionning and termination over a distributed architecture.

The second range of tests concern the node failure detection and dynamic reconfiguration process. This step is not yet fully achieved.

The initialization of FT redundancy management is working and ftr_tasks are created correctly, they execute then terminate after a given number of iterations.

FTR_tasks are created, periodic behavior is insured, communication with FTR components work.

However tuning of communication parameters using ORTE has still to be done, some messages are lost in the current version. Change to the new version of OCERA should permit a better control of communication and fix this problem.

2.6 Examples

A simple example is provided. It is intended to test the different points cited above and related both to application initialization, event detection, behavior commutation and application termination. Since there is no global example directory for Redundancy Management, all installation and testing is done in the examples subdirectory located within the **ftredundancymgr** component.

2.6.1 How to run the examples

Up to now, the examples developed are common to the two components.

The example directory is located within the **ftredundancymgr** component :

ocera/components/ft/ftredundancymgr/examples/ftr_appli

It is the Makefile located within this directory that builds the test application. In order to run the example it is necessary to compile and start the **ftredundancy management** facility first.

Implementation :

The ***ft/ftredundancymgr/examples/*** directory has the following structure:

```
examples
  ! --- README
  ! --- INSTALL
  ! --- Makefile
  ! --- ftr_appli
  !       !--- README
  !       !--- INSTALL
  !       !--- Makefile
  !       !--- include
  !       !      !---ftr_appli.h
  !       !--- src
  !       !      !---ftr_appli.c
```

The ***ftr_appli*** is a simple application that has been developed to test the **ftredundancy management** facility.

The general OCERA Makefile file permits the compilation of the overall OCERA tree provided options are selected in the configuration step (see OCERA HOWTO for OCERA configuration steps). However examples can be compiled separately afterwards.

Compilation :

In order to compile the example please follow next steps :

```
 - Go to the ft/ftredundancymngr/examples directory:
 $  cd ft/ftredundancymngr/examples


 - Clean the ft/ftredundancymngr/examples directory:
 $  make clean


 - Compile the examples:
 $  make
```

Installation/Execution :

Note that execution of examples requires a distributed architecture. So the ftcomponents and examples must be present on each machine that will be involved in the test. This requires additional operations and controls before the example can be run.

- Install OCERA (or at least ORTE and ftcomponents) on each machine.
- Insure that rights are set so as to allow for remote execution of the code corresponding to both components and application.
- Set up environment variables

(See section 2.7 for details)

The example runs on two nodes N1 and N2. The application has two tasks T1 and T2.

T1 master task is running on node N1 and T2 master task is running on Node2. Node1 is the master node on application start.

To run the application one must :

- start ftredundancy management

  A shell script allows for this, it is located in ***ft/ftredundancymngr/src*** :

  ```
  $ ftrm_start  <Node1> <Node2>
  where <Nodei> is an hostname
  ```

  It starts ORTEManager on each node, then starts ftredundancy components on each node. Actually the two components of a node are linked a single Linux executable named ft_redman.

  The master node is the current node (it must be the same as the first argument , here Node1).

- start application on master node

```
$ cd ftr_appli/src

$ ./ftr_appli
```

The application starts first on Node1 then on Node2. Replicas are created and ftr_tasks started.

After a given number of cycles the application ends.

2.6.2 Description

Up to now, the examples developed are common to the two components.

Up to now , there is only one simple example provided. It runs on two nodes N1 and N2. The application has two tasks T1 and T2.

T1 master task is running on node N1 and T2 master task is running on Node2. Node1 is the master node on application start.

The main objective of this example is to test application registration, ftr_tasks and replicas management, checkpointing and application termination.

2.6.3 Results and comments

The current implementation is still a prototype one. We have adopted an incremental development cycle and some functionalities have still very basic implementation. The main goal of this step was to provide a consistent overall framework for redundancy management. A lot of work has still to be done to make an efficient operational environment of it.

However, the example has permitted to test the ft redundancy management overall structure .

- Ft redundancy framework set-up and functioning
- Application registration
- Application execution
- Application termination.

The next step will cover

- Node crash detection
- Application dynamic reconfiguration.

2.7 Installation instructions

The two FT Redundancy management components provided make part of the OCERA tree under the ft branch. We don't detail here the subtrees corresponding to degraded mode management and ftbuilder which are described in D6.2_rep.

The ft subtree contains the following directories and files

```
ft
!--- ftappmon (not detailed here)
!--- ftcontroller (not detailed here)
!ftbuilder (not detailed here)
!ftredundancymgr
!         README
!         INSTALL
!         Makefile
!         doc
!         examples
!         !     README
!         !     INSTALL
!         !     Makefile
!         !     ftr_appli
!         !           README
!         !           INSTALL
!         !           Makefile
!         !           include
!         !             !---ftr_appli.h
!         !           src
!                       !---ftr_appli.c
!         include
!            !---ft_redundancymgr.h
!            !---ft_controller.h
!         src
!            !---ft_redundancymgr.c
!            !---ft_controller.c
...
!
!
!
!
```

```
!
!ftreplicamgr
!         README
!         INSTALL
!         Makefile
!         doc
!         include
!             !   ft_replicamgr.h
!         src
!             !   ft_replicamgr.c
```

The general OCERA installation procedure compiles and installs the selected components.

In order to be able to use the Redundancy Management facility, at configuration, one must :

- select Soft RT-LINUX in the OCERA configuration tool and select the **ft/ftredundancymgr** and **ft/ftreplicamgr** components.
- select ORTE in the communication components.

During this general installation procedure, all the components and examples are compiled.

The two **ftredundancy** components are compiled into one single executable named **ft_redman**.

Remark:

As said in the 3.6 section, ftredundancy components have to be present on all the nodes of the network devoted to applications. This means that the installation procedure must be done on each node.

If one doesn't want to have full OCERA installation on each node, it is possible to compile only ft and ORTE components separately from OCERA.

<u>ORTE installation</u>

ORTE installation in Linux user's space is rather simple (see extract of readme below or ORTE documentation for more details D7.2 or D7.4) .

```
untar installation package into desired directory, enter this
directory and issue following commands.
UserSpace compilation:
./configure
make
make install
after this procedure ortemanager and orteping are placed in
/usr/local/bin.
```

Ftredundancy components installation

For a separate testing of the FT components :

- copy the ft subtree in the location you want,
- change the ft/Makefile to restrict compilation to **ftredundancymgr** and **ftreplicamgr**
- make

The ft/Makefile normally compiles and install all ftcomponents so you have to change it a bit to restrict it to redundancy management as it is shown below.

1. Change the SUBDIRS line

```
SUBDIRS = ftappmon ftcontroller ftbuilder ftredundancymgr
ftreplicamgr
```

to the following

```
SUBDIRS = ftredundancymgr ftreplicamgr
```

2. Comment out the ocera related stuff

```
ifneq ($(wildcard ../../ocera.mk),)

include ../../ocera.mk

else

all:

        @echo -e "You should go to the ocera/ directory
and do 'make' to generate the ocera.mk file
first.\nThanks."

endif
```

which becomes

```
#ifneq ($(wildcard ../../ocera.mk),)

#include ../../ocera.mk

#else

#all:

#       @echo -e "You should go to the ocera/ directory
#and do 'make' to generate the ocera.mk file
#first.\nThanks."

#endif
```

Installation

The normal installation process is done through OCERA config tool. We have not defined a particular installation process yet, so executable code is located within **ft/ftredundancymgr/src** and **ft/ftredundancymgr/examples/ftr_appli/src**

So you have to define an environment variable named FT_RM_BIN_DIR and copy ft_redman executable and ftrm_start script in it or let $FT_RM_BIN_DIR be **ft/ftredundancymgr/src.**

Chapter 3. ftreplicamgr component

3.1 Summary

- Name : **ftreplicamgr**
- Description : **ftreplicamgr** along with **ftredundancymgr** are two complemetary components that provide transparent redundancy management for real-time applications. Redundancy policy implemented is based on a passive replication model. The **ftredundancymgr** is in charge of global application and network initialization and control (including node crash detection). The **ftreplicamgr** is in charge of tasks level control: tasks groups and tasks replicas management, checkpointing.
- Author (name and email) :
  A. Lanusse (agnes.lanusse@cea.fr)
  P. Vanuxeem (patrick.vanuxeem@cea.fr)
- Reviewer :
- Layer : Linux Level.
- Version : V0.1
- Status : design
- Dependencies : ocera V1.0 requires ORTE component and ftredundancymgr component
- Release date : M2

3.2 Description

The **ftreplicamgr component** is in charge of monitoring local tasks execution and maintaining groups of replicas consistency. Its role is thus to :

- monitor the tasks running on the node (and detects deadline miss on tasks),
- perform periodic checkpointing of tasks contexts and shared data.
- maintain groups of replicas status
- maintain databases related to tasks contexts and shared data.

As **ftredundancygr**, the **ftreplicamgr** must be present on each node of the configuration and must always be associated with the ftredundancy component. They must be started before the application. A script helps starting the two components on a set of nodes.

3.2.1 Ftreplicamgr internal description.

The **ftreplicamgr** component consists of several threads : one main controlling thread insures global group management of tasks located on the node, it cooperates with specialized threads dedicated to checkpointing and error detection at task level. A watchdog thread detects deadline misses.

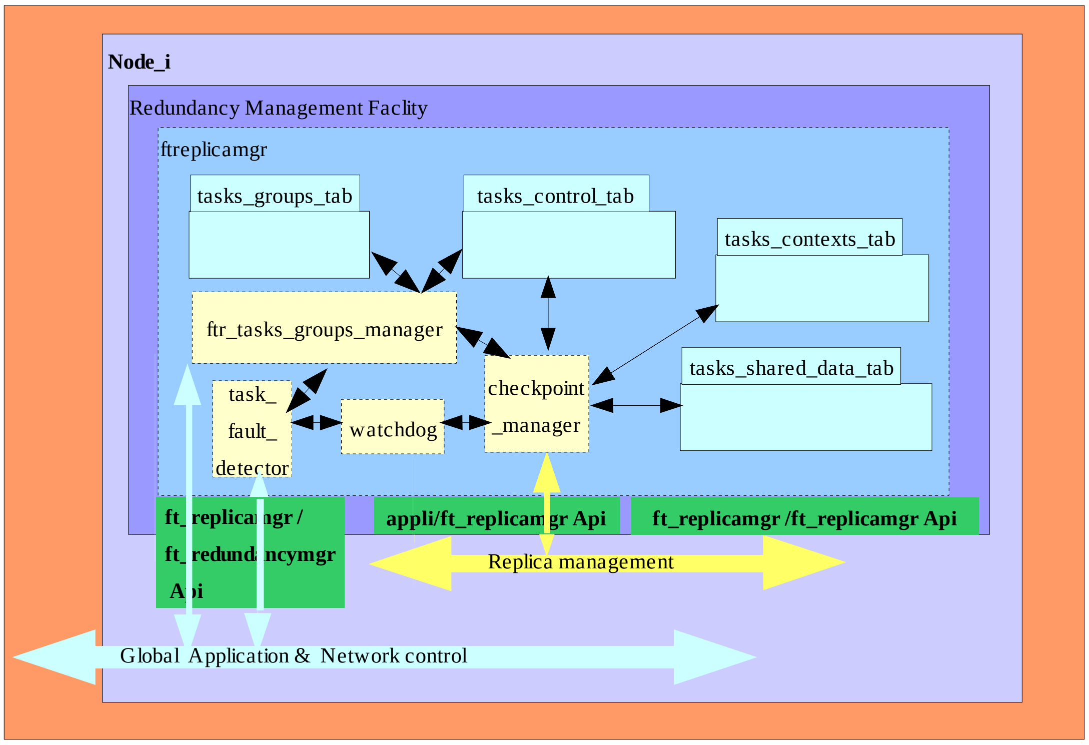


Figure 3.1 ftreplicamgr: internal view

This component maintains several databases and propagates data to other **ftreplicamgrs** when changes related to a local master task occur.

Local databases are :

- **tasks_groups_tab** which contains the description of each group of tasks replicas.
- **tasks_control_tab** which contains information on the current status of each task (state (i.e. created, running, ended), master/slave, start_cycle_time, deadline_cycle_time, cycle_period,...),
- **tasks_contexts_tab** which contains the current ftr_task_context for each task.
- **tasks_shared_data_tab** which contains the current valid values (read/write) for each shared data.

The checkpoint manager receives periodically :

- new context
- new shared data values

at the end of each local master task cycle.

It then propagates these new values to other members of the group.

It also receives periodically in the same manner, new values for slave local tasks replicas. It then updates its local copy of these data.

In parallel, the checkpoint 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 functioning correctly, the replica manager informs the redundancy manager that collects information from other nodes and will decide to change the active replica if possible. The watchdog thread is in charge of detecting such possible deadline miss for reception of new context values from application or from other **ftreplicamgrs**.

A timer is armed on starting a new cycle and reset each time information is received on time by the checkpoint manager. If timeout occurs before, a notification is issued to task fault_detector which then propagates it to the ft_tasks_group manager and to **ftredundancymgr**.

3.3 API / Compatibility

This component uses POSIX Linux API for threads manipulation. A few additional primitives have been defined to handle redundancy management.

As it is the case for the **ftredundancymgr**, this API can be divided into three subsets.

- an API for communication between application (ftr_controller) and **ftrundancymgr (ftr_controller/ftrep API).**
- an API for communication between **ftreplicamgr and ftredundancymgr** (**ftrep/ftred API**) located on different nodes,
- an API for communication between the **ftreplicamgrs (ftrep/ftrep API)**,

ftr_controller/ftrep
ftr_notify_appli_task_created()
ftr_notify_appli_task_cycle_started()
ftr_notify_appli_task_cycle_finished()
ftr_notify_appli_task_ended()
ftr_task_context_commit()
ftr_task_context_update()
ftr_shared_data_commit()
ftr_shared_data_update()
ftrep/ftred
ftr_task_group_add_member()
ftr_task_group_remove_member()
ftr_task_group_modify_member_attributes()
ftr_notify_task_failed()
ftrep/ftrep
ftr_task_checkpoint()
ftr_ftreplicamgr_heartbeat()

3.3.1 API between ftr_controller and ftreplicamgr (ftr_controller / ftrep API)

ftr_notify_appli_task_created

parameters :

in:

    ftr_appli_task_name
    ftr_appli_id

Description :

Used by the ftr_controller thread in the user's process to communicate with the **ftreplicamgr** and signal the creation of a ftr_task. The result gives the ID and the type of replica (master or slave) behavior to adopt for this instance. The type is determined using the group configuration for this task.

Prototype :

```
extern FTR_TASK_REPLICA_ID ftr_notify_appli_task_created
                      (     FTR_APPLI_TASK_NAME,
                            FTR_APPLI_ID);
```

ftr_notify_appli_task_cycle_started

parameters :

in:

    ftr_task_replica_id
    issue_number
    date

Description :

Used by the ftr_controller thread in the user's process to communicate with the **ftreplicamgr** and signal the start of an execution cycle of the ftr_task_replica.

Prototype :

```
extern int ftr_notify_appli_task_cycle_started
                        (FTR_TASK_REPLICA_ID,
                          int,
                          NtpTime);
```

ftr_notify_appli_task_cycle_finished

parameters :

in:

    ftr_task_replica_id
    issue_number
    date

Description :

Used by the ftr_controller thread in the user's process to communicate with the **ftreplicamgr** and signal the end of an execution cycle of the ftr_task_replica.

Prototype :

```
extern int ftr_notify_appli_task_cycle_finished
                        (FTR_TASK_REPLICA_ID,
                          int,
                          NtpTime);
```

ftr_notify_appli_task_ended

parameters :

in:

    ftr_task_replica_id
    issue_number
    date

Description :

Used by the ftr_controller thread in the user's process to communicate with the **ftreplicamgr** and signal the end of a ftr_task.

Prototype :

```
extern int ftr_notify_appli_task_cycle_started
                        (FTR_TASK_REPLICA_ID,
                          int,
                          NtpTime);
```

ftr_task_context_commit

parameters :

in:

    ftr_task_replica_id
    ftr_context_id
    issue_number
    date

Description :

Used by the ftr_controller thread in the user's process to communicate with the **ftreplicamgr** and commit the new context of a master task replica at the end of an execution cycle.

Prototype :

```
extern int ftr_notify_appli_task_cycle_started
                        (FTR_TASK_REPLICA_ID,
                         FTR_TASK_CONTEXT_ID,
                          int,
                          NtpTime);
```

ftr_task_context_update

parameters :

in:

    ftr_task_replica_id
    ftr_context_id
    issue_number
    date

Description :

Used by the ftr_controller thread in the user's process to get the new value of a ftr_task_context before each beginning of cycle. (For a master task, the value is already set, for a slave task, the context is read from the local value stored on **ftreplicamgr**).

Prototype :

```
extern int ftr_task_context_update
                        (FTR_TASK_REPLICA_ID,
                         FTR_TASK_CONTEXT_ID,
                         int,
                         NtpTime);
```

ftr_shared_data_commit

parameters :

in:

```
         ftr_task_replica_id
         ftr_shared_data_id
         issue_number
         date
```

Description :

Used by the ftr_controller thread in the user's process to propagate the new value of a ftr_shared_data whose ftr_task is writer. This is done at each end of cycle and propagated to all **ftreplicamgrs.**

Prototype :

```
extern int ftr_shared_data_commit
                        (FTR_TASK_REPLICA_ID,
                         FTR_SHARED_DATA_ID,
                         int,
                         NtpTime);
```

ftr_shared_data_update

parameters :

in:

```
         ftr_task_replica_id
         ftr_shared_data_id
         issue_number
```

date

Description :

Used by the ftr_controller thread in the user's process to get the new value of a ftr_shared_data before each beginning of cycle. (For a master writer task, the value is already set, for a slave task, or a master reader task the context is read from the local value stored on **ftreplicamgr**).

Prototype :

```
extern int ftr_shared_data_update
                        (FTR_TASK_REPLICA_ID,
                         FTR_SHARED_DATA_ID,
                         int,
                         NtpTime);
```

3.3.2 API between ftreplicamgr and ftredundancymgr (ftrep/ftred API)

ftr_task_group_add_member

parameters :

in :

ftr_task_group_id

ftr_task_id

Description :

Used by the **ftreplicamgr** to add information related to a replica that just started to the ftr_task_group.

Prototype :

```
extern int ftr_task_group_add_member(FTR_TASK_GROUP_ID,
                            FTR_TASK_ID);
```

ftr_task_group_remove_member

parameters :

in :

ftr_task_id

ftr_task_replica_desc

ftr_task_group_id

Description :

Used by the **ftreplicamgr** to remove information related to a replica that just ended to the ftr_task_group. The replica is not available any more.

Prototype :

```
extern int ftr_task_group_remove_member(FTR_TASK_GROUP_ID,
                            FTR_TASK_ID);
```

ftr_task_group_modify_member_attributes

parameters :

in :

ftr_task_group_id

ftr_task_group_id

ftr_task_replica_desc

Description :

Used by the **ftreplicamgr** to update information related to a replica to the ftr_task_group.

Prototype :

```
extern int ftr_task_group_modify_member(FTR_TASK_GROUP_ID,
                            FTR_TASK_ID,
                            FTR_TASK_REPLICA_DESC);
```

ftr_notify_task_failed

parameters :

in :

    ftreplicamngr_Id

    ftr_task_Id

    ftr_node_Id

    issue_number

    date

Description :

Used by the **ftreplicamgr** to notify a deadline miss on a task iteration cycle. This detection is followed by a reconfiguration phase and a call to **ftr_application_config_modify** for each application.

Prototype :

```
extern int ftr_notify_task_failed( FTR_RED_MGR_ID,
                                  FTR_TASK_ID,
                                  FTR_NODE_ID,
                                  int,
                                   NtpTime);
```

3.3.3 API between ftreplicamgrs (ftrepl/ftrepl API)

ftr_task_checkpoint

parameters :

in :

```
     ftr_task_id
     ftr_appli_id
     iteration_number
     date
```

Description :

Used by the **ftreplicamgr** to broadcast a new context and shared data (in writer mode) values at the end of a master task replica.

Prototype :

```
extern int ftr_task_checkpoint( FTR_TASK_ID,
                                FTR_APPLI_ID,
                                int,
                                NtpTime);
```

ftr_ftreplicamgr_heartbeat

parameters :

```
in : ftreplicamgr_Id
     iteration_number
     date
```

Description :

Used by the **ftreplicamgr** to signal its liveliness to the system. A periodic signal is sent on the network and received by all the **ftredundancymgrs.**

Prototype :

```
extern int ftr_ftreplicamgr_heartbeat( FTR_RED_MGR_ID,
                                       int,
                                       NtpTime);
```

3.4 Implementation issues

- Modifications to the existing RTLinux or Linux code

This component is a new one, there is no modification to existing Linux component.

- Data structures created.

Main data structures created concern :

- ftr_task_replica_desc,
- ftr_task_group.
- ftr_shared_data,
- ftr_task_context,

Data tables are :

- ftr_tasks_control_tab
- ftr_tasks_groups_tab,
- ftr_tasks_contexts_tab,
- ftr_tasks_shared_data_tab,

Control events defined are :

- ftr_replica_control_event

```
Structures :


typedef struct
```

```
{
  char appli_name[NAME_MAX_LENGTH];
  char appli_task_name[NAME_MAX_LENGTH];
  FTR_APPLI_TASK_ID  appli_task_id;
  FTR_SCHEDULING_PARAMETERS *scheduling_parameters;
  FTR_REDUNDANCY_PARAMETERS *redundancy_parameters;
  FTR_TASK_REPLICA_STATUS replica_status;
  FTR_LOCATION replica_location;
  FTR_TASK_CONTEXT *context;
  FTR_SHARED_DATA *W_shared_data;
  FTR_SHARED_DATA *R_shared_data;
} FTR_TASK_REPLICA_DESC ;


typedef struct
{
  char group_name[NAME_MAX_LENGTH];
  char appli_task_name[NAME_MAX_LENGTH];
  FTR_APPLI_TASK_ID  master_task_id;
  FTR_LOCATION       master_task_location;
  FTR_TASK_CONTEXT *context;
  FTR_SHARED_DATA *W_shared_data;
  FTR_SHARED_DATA *R_shared_data;
  FTR_TASK_REPLICA_DESC  replicas_tab[NB_MAX_REPLICAS];
} FTR_TASK_GROUP_DESC ;

typedef struct
{
  FTR_TASK_ID     writer_task;
  FTR_DATA_STRUCT published_data;
  FTR_DATA_STRUCT private_data;
  int current_valid_version_number;
} FTR_SHARED_DATA ;



typedef struct
{
  FTR_TASK_ID     writer_task;
```

```
    FTR_DATA_STRUCT published_data;
    FTR_DATA_STRUCT private_data;
    int current_valid_version_number;
  } FTR_CONTEXT ;
```

Tables :

The FT R Nodes table :

```
          FTR_TASK_GROUP_DESC ftr_groups_tab[FTR_TASKS_MAX];
```

The FT R Tasks Control Table :

```
          FTR_TASK_REPLICA_DESC ftr_tasks_control_tab
                                                  [FTR_TASKS_MAX];
```

The FT R Shared Data Table :

```
          FTR_SHARED_DATA ftr_tasks_shared_data_tab[FTR_TASKS_MAX];
```

The FT R Contexts Table :

```
          FTR_CONTEXT ftr_tasks_contexts_tab[FTR_TASKS_MAX];
```

New types defined to describe status of various entities:

```
typedef enum FTR_TASK_REPLICA_STATUS {
  FTR_TASK_REPLICA_STATUS_NOT_DEFINED,
  FTR_MASTER,
  FTR_SLAVE,
  FTR_TERMINATED
} FTR_TASK_REPLICA_STATUS;
```

```
New types defined to describe control events:


  typedef enum
     {
       FTR_TASK_NOP,
       FTR_TASK_REPLICA_CREATION_REQUIRED,
       FTR_TASK_REPLICA_TERMINATION_REQUIRED,
       FTR_TASK_REPLICA_CYCLE_STARTED,
       FTR_TASK_REPLICA_CYCLE_ENDED,
     } FTR_TASK_REPLICA_CONTROL_EVENT;
```

3.5 Tests and validation

3.5.1 Validation criteria

In a first stage validation criteria concern purely functional qualitative criteria.

Verification that in absence of abnormal situation the application runs normally.

Verification that checkpointing works properly.

Verification that faulty events (deadline miss are detected)

Verification that the propagation of an abnormal event to the **ftredundancymgr** is achieved correctly.

Verification that a faulty task commutes correctly to another replica.

In a second stage, we will verify synchronization issues.

Verification that the replacement replica activation is achieved at the right time (next activation period of the previous running task)

In a third stage, if possible, performance issues will be targetted

Verification that commutation times satisfy minimum period requirement from the application.

3.5.2 Test 1

Management of task periodic context and shared data checkpointing

In this test, the periodic checkpointing (of context and shared data) is performed.

3.5.3 Test 2

Detection of abnormal event (deadline miss on task_cycle_end).

The principle of deadline miss event detection is being tested.

Each time a cycle starts, a notification is received by the **ftreplicamgr** with the expected deadline. A timer is armed with this deadline and reset on reception of end of cycle notification. If timeout is reached before, a deadline miss is detected and the **ftrdeundancymgr** is notified of the event. This latter then decides if the faulty task is replaced by one of its replicas.

This mechanism is still under testing.

3.5.4 Results and comments

The testing process is still on going.

Up to now, we have tested the basic communication mechanisms between the components involved in Fault-Tolerance redundancy management and the functioning of basic commutation mechanisms.

This has permitted to set up a global FT framework. The principles of initialization, event detection, commutation and termination have been settled but a lot of work has still to be done.

3.6 Examples

Since there is no global example directory for Redundancy Management, all installation and testing is done in the examples subdirectory located within the **ftredundancymgr** component so for further details related to the next sections please refer to the 2.6 Section.

3.6.1 How to run the examples

Up to now, the examples developed are common to the two components, please refer to the **ftredundancyr** examples section.

The example directory is located within the **ftredundancymgr** component :

ocera/components/ft/ftredundancymgr/examples/ftr_appli

3.6.2 Description

Please refer to section 2.6.2

3.6.3 Results and comments

Please refer to section 2.6.3

3.7 Installation instructions

Please refer to 2.7