IST-2001 34140

Operating System Primitives: implementation under MaRTE OS

Deliverable D-OS.1v3

Responsible: Universidad de Cantabria

Julio L. Medina Pasaje, Michael González Harbour,
Mario Aldea Rivas, José Carlos Palencia Gutiérrez

12 April 2005

Contents

1 Introduction 2

2 The FIRST scheduling framework API 4

2.1 Data structures and configuration parameters 4
2.2 Contract management functions 7
2.3 Server administration 7
2.4 Quality management functions 8
2.5 Bounded workload management 8
2.6 Services for distributed applications 8
2.7 Support for shared resources 8
2.8 Implementation-specific services 8

3 Internal architecture of the implementation 9

3.1 Scheduling information data structures 9
3.1.1 Scheduler and Service Thread shared information 9
3.1.2 Server specific data: 11
3.1.3 Thread specific data 11
3.1.4 Service jobs table and index queue 12
3.1.5 Synchronization objects 12
3.1.6 App-sched parameters 13
3.1.7 Constants for signalling the scheduler in the info field 13
3.1.8 Explicit data exchanged with the scheduler 13
3.2 Scheduler operation 14
3.3 Server operation 14
3.4 Threads states 15
3.5 Background round-robin scheduler 16
3.6 Getting the OS configuration information 16

4 Design of the FSF_Contract_API methods 17

5 Acceptance test and reclamation algorithm 18

5.1 Automatic priority assignment 19
5.2 Acceptance test when adding a new server 19
5.3 Acceptance test structures maintenance when deleting a server 20
5.3.1 Spare capacity sharing algorithms 21

6 Shared objects management 22

7 Implementation in Ada 22

8 Organization of the code 22

9 References 24

Annex 1: Architecture of the distributed FSF implementation 25

1 Introduction

This document describes an implementation of the FIRST Scheduling Framework (FSF) as it is proposed in the deliverable D-SI.1v3. This implementation has been developed to run on a platform conforming to the Minimum Real-time POSIX profile, and using the non-standard application defined scheduling extensions ("APP_SCHED") [AG02]. The particular platform is using the MaRTE OS developed at the University of Cantabria [AG01],

A graphical presentation of usages and users of the framework is shown in Figure 1. Multi-threaded and distributed applications can be managed to fit in the contract-server model, by assigning the different management responsibilities to a concrete thread or to some coordinated group of them.

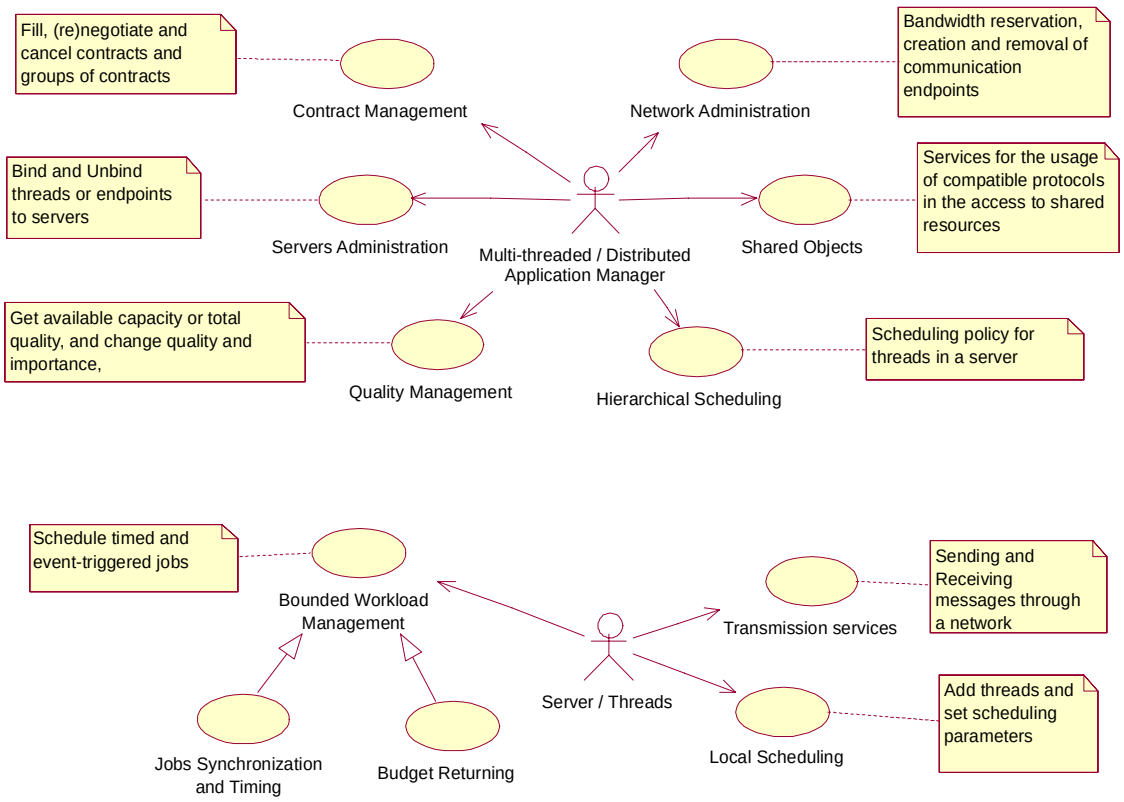

Figure 1. A presentation of the FSF services according to its expected usage.

The main architecture of the solution implemented is shown in Figure 2 by means of a class diagram. In this diagram two separate memory areas are considered, one for the application threads, labelled in Figure 2 as *User Memory Space* and the other one for the application scheduler related objects, called *Scheduler Memory Space*. This separation is caused by the implementation-independent nature of the application scheduling interface, which must allow for the application scheduler to be eventually implemented inside the kernel's memory space, although the current implementation in MaRTE OS uses a special thread at the application level.

The link between the user threads and the underlying scheduler is made in the implementation of the *FSF_Service_Contract_API*. This API is common to both target platforms of the project: S.Ha.R.K OS and MaRTE OS, and is briefly presented in the next section. As it is shown in Figure 2, all the scheduling responsibilities are solved by two internal threads, the *scheduler* and the *service thread*. The scheduler runs at a priority higher than any FSF scheduled task and consequently its effects over the schedulability of the application must be considered as an additional preemption term. The service thread, instead, is

scheduled by the underlying operating system as a normal sporadic server with a budget and period assigned by the system developer.

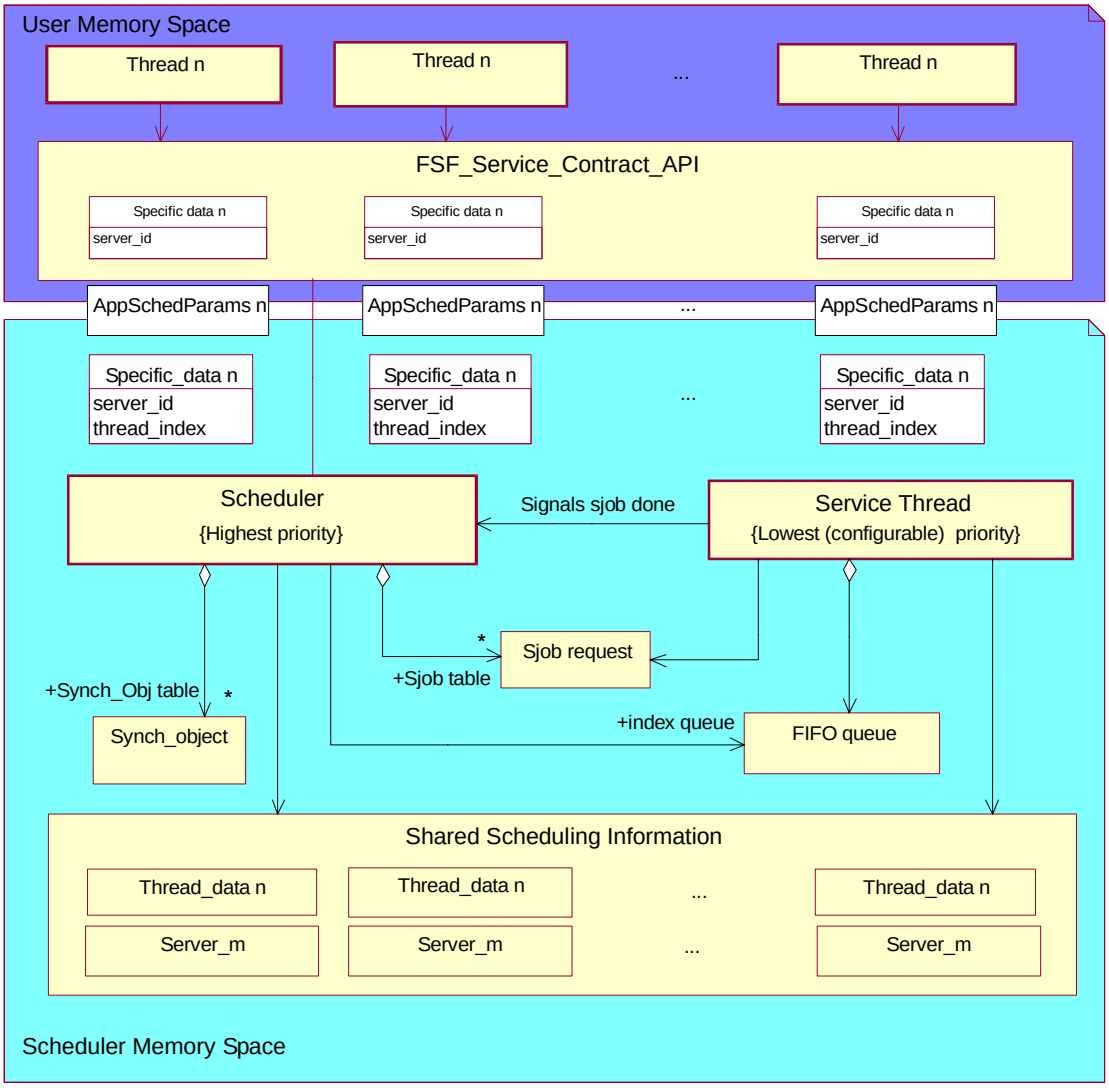

Figure 2. Main architecture of this implementation

The detailed architecture of this scheduling mechanism, as well as the scheduling information data structures used to implement it are presented in Section 3. The detailed design of the methods provided by the *FSF_Service_Contract_API* is presented in Section 4. Section 5 describes the way in which the reclamation algorithm and the acceptance test are implemented. Section 6 presents the implementation of the shared objects management. Section 7 shows the Ada95 version of the FSF API services. Section 8 shows the organization of the code and the way of using from an end-user application, using the appropriate version of MaRTE OS. Finally the last section presents the referenced bibliography.

The detailed description of the Distributed FSF module and its implementation appears in Annex A of this document.

In a separate compressed file, which is available in the web page presented in deliverable D-EX.2v3, it is possible to find the complete source code of the implementations in C and Ada, the *Makefile*, and all the necessary header (*include*) files for the implementation presented in this document.

2 The FIRST scheduling framework API

All the functionality defined in the FIRST Scheduling Framework (FSF) is accessible through its proposed *service contract application programming interface*. To facilitate its usage and the selective implementation of the wide functionality proposed by the FIRST project architectural framework requirements, this API is divided into several modules. The precise description of the API and the rationale on the purpose of each module is made in the D-SI.1v3 and D-AF.2v2 deliverables. Nevertheless, for convenience, a graphical summary of the API is presented here. Figure 3 shows the modules of the API and their dependencies by means of a visibility graph depicted in a UML components diagram. The dependencies shown correspond to the visibility required between the Ada95 specification packages used in the ADA version of the FSF API.

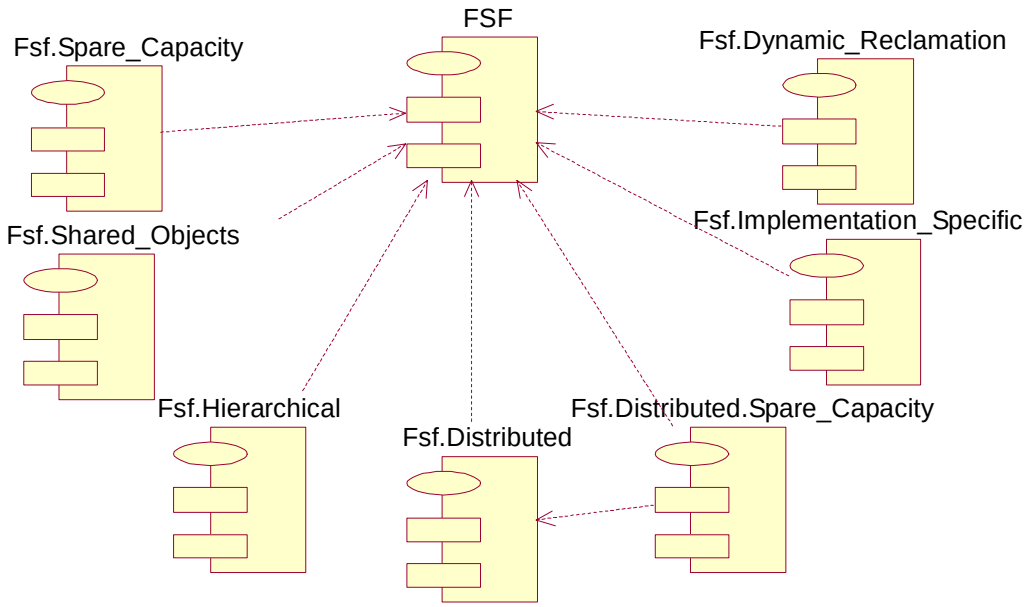

Figure 3. Modules of the service contract API

The modules implemented in this deliverable are the Core (FSF), Spare_Capacity, Shared_Objects, Implementation_Specific, and Distributed. Other modules such as hierarchical scheduling and dynamic reclamation are implemented in the Shark version of FSF.

Figure 4 presents in a UML class diagram, a language-independent view of the data structures and methods exported and used by the Core module of this API, as well as other operations exported by other modules; the diagram shows its decomposition considering an object oriented approach. The actual data structures used and a short reference to the functionality offered by the FSF API are then briefly mentioned.

2.1 Data structures and configuration parameters

The main abstract data structure defined in the FSF API is the **`fsf_contract_parameters_t`** structure, which effectively contains the contract parameters described in Section 4 of the D-SI.1v3 deliverable. Here we present this structure and its parameters from its implementation point of view. In following sub-sections some semantic issues are described when necessary as some other capabilities are introduced. This structure is proposed as an opaque type, whose attributes are written and read by means of a set of functions. Most of the parameter names and the types shown in Figure 4 are self-explanatory and their C versions are included in Annex A of D-SI.1v3. The concrete implemented C ver-

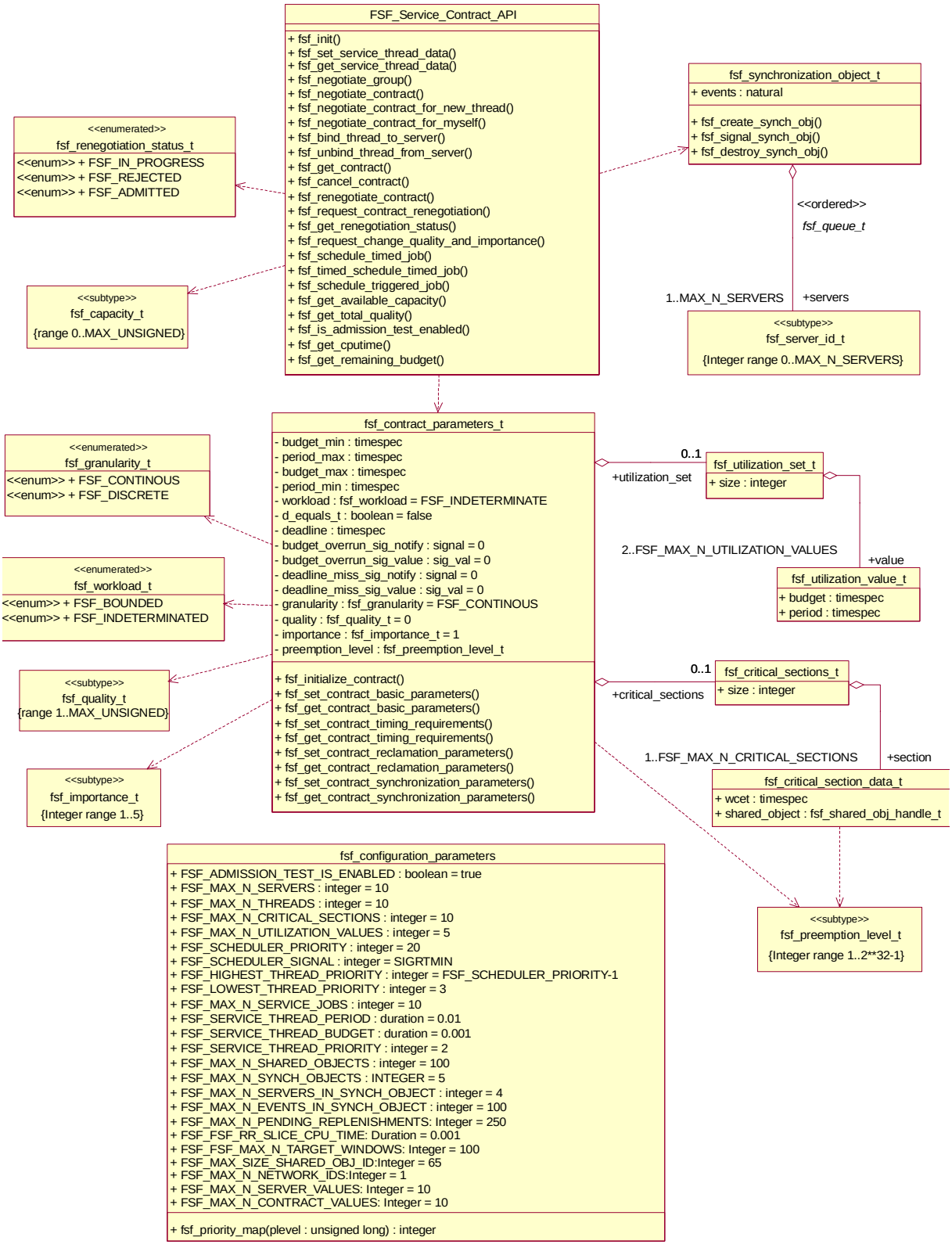

Figure 4. Abstract view of the service contract API

sion of this structure as well as some other types used to define it (also shown in Figure 4) are presented here for convenience.

```
typedef struct {
  struct timespec         budget_min;
  struct timespec         period_max;
  struct timespec         budget_max;
  struct timespec         period_min;
  fsf_workload_t          workload;
  bool                    d_equals_t;
  struct timespec         deadline;
  int                     budget_overrun_sig_notify;
  union sigval            budget_overrun_sig_value;
  int                     deadline_miss_sig_notify;
  union sigval            deadline_miss_sig_value;
  fsf_granularity_t       granularity;
  fsf_utilization_set_t   utilization_set;
  int                     quality;
  int                     importance;
  fsf_preemption_level_t  preemption_level;
  fsf_critical_sections_t critical_sections;
  fsf_sched_policy_t      policy;
  fsf_network_id_t        network_id;
  bool                    granted_capacity_flag;
} fsf_contract_parameters_t
```

In addition to the attributes described in Annex A of D-SI.1v3, the members **`budget_overrun_sig_value`** and **`deadline_miss_sig_value`**, were included to allow for the possibility of sending an information field configured by the user attached to the indicated signal, when the respective notification is required.

The data types **`fsf_utilization_set_t`** and **`fsf_critical_sections_t`** are implemented as structures with fixed-size arrays. Their size as well as all other parameters are configurable by the user, and are shown in Figure 4 as the **`fsf_configuration_parameters`** class, which corresponds to the **`fsf_configuration_parameters.h`** header C file

The utilization set, used when the granularity of the application timing is discrete, is introduced in the contract by the types **`fsf_utilization_value_t`** and **`fsf_utilization_set_t`**.

```
typedef struct {
  struct timespec    budget;    // Execution time
  struct timespec    period;    // Period
} fsf_utilization_value_t;

typedef struct {
  int                          size; // = 0
  fsf_utilization_value_t      value[FSF_MAX_N_UTILIZATION_VALUES];
} fsf_utilization_set_t;
```

The concrete C types used for introducing the critical sections are:

```
// Critical section data
typedef struct {
   fsf_shared_obj_handle_t     obj_handle;
   struct timespec             wcet;  //Execution time
} fsf_critical_section_data_t;

// List of critical sections
typedef struct {
  int                          size; // = 0
```

```
  fsf_critical_section_data_t  section[FSF_MAX_N_CRITICAL_SECTIONS];
} fsf_critical_sections_t;
```

The service contract API defines also a number of constants for the assignment of default values and for the error codes possibly returned by the functions.

2.2 Contract management functions

Here we distinguish two groups of functions. A first group embodies the functions for the composition and interrogation of the passive structure that holds a contract. These are the methods of the **`fsf_contract_parameters_t`** class shown in Figure 4. The second group considers functions for the negotiation and cancellation of contracts, which effectively reserve and release execution capacity from the system, creating the servers for the contract parameters and also assigning the initial thread to the servers. They are presented in Figure 4 as methods of the **`FSF_Service_Contract_API`** class.

The first function that must be called to initialize and create all the internal management structures that are necessary for the FIRST Scheduling Framework to operate properly is the **`fsf_init`** function.

The contract management functions that operate over the passive data structure are spread across the different modules of the API and are characterized by having the contract parameters structure as their first argument, and being presented as pairs of operations, one that sets and another one that gets the corresponding parameter.

The default values stored in a contract parameters object after its initialization are:

```
//  budget_min                  => {0,0};
//  period_max                  => {0,0};
//  budget_max                  => {0,0};
//  period_min                  => {0,0};
//  workload                    => DEFAULT_WORKLOAD; (FSF_INDETERMINATE)

//  d_equals_t                  => DEFAULT_D_EQUALS_T;           (false)
//  deadline                    => DEFAULT_DEADLINE;               (0,0)
//  budget_overrun_sig_notify   => 0;                     (signal number)
//  budget_overrun_sig_value    => {0};
//  deadline_miss_sig_notify    => 0;                     (signal number)
//  deadline_miss_sig_value     => {0};
//
//  granularity                 => DEFAULT_GRANULARITY;
//  utilization_set;            => size = 0

//  quality                     => DEFAULT_QUALITY;(0)(range 0..2**32-1)
//  importance                  => DEFAULT_IMPORTANCE;   (1)(range 1..5)
//
//  preemption_level            => 0;                 (range 1..2**32-1)
//  critical_sections;          => size = 0
//
//  policy                      => DEFAULT_SCHED_POLICY;      (FSF_NONE)

//  network_id                  => FSF_NULL_NETWORK_ID;              (0)

//  granted_capacity_flag       => false;
```

2.3 Server administration

After a successful negotiation, the reserved execution capacity demanded by a contract is managed by the scheduling system under the concept of a server. The basic operations allowed for the user to do with the servers are its assignment or *binding* to different operating system threads and the synchronization of the jobs to be performed using the capacity reserved by the servers. The binding and unbinding of threads to and from servers allows for the possibility of having servers that are not bound, which means

servers not in use (wasting capacity) as well as threads suspended in a dormant state, waiting to be bound to a server.

The FSF hierarchical module is not supported in the present MaRTE OS implementation, but it is feasible to be implemented in the future, having several threads bound at the same time to a server. This implies an implementation-dependent underlying scheduling mechanism among these threads that can be implemented in the first level scheduler. The ability of having CPU clocks attached to groups of threads would be a key feature for the target operating system to realize this enhancement to the binding mechanism.

On the other hand the event-driven synchronization of servers proposed by the synchronization objects, mentioned in Section 4.1 and included in Annex A of the D-SI.1v3 deliverable, is implemented with a structure that effectively realizes the **`fsf_synch_object_t`** class shown in Figure 4.

2.4 Quality management functions

The sharing of the capacity that is beyond the minimum required for the acceptance of the contracts is performed over the basis of the server's quality and importance parameters, which are set and managed using the *spare_capacity* module operations. The functions listed in that module are used by a server or a quality management centralized application to control the relative position of a server in the context of the rest of the servers in the same importance level.

2.5 Bounded workload management

These functions are dedicated to the control of each individual piece of work that must be performed by bounded workload servers. They ask the scheduler to activate the server on a per-job or per-instance basis. There are three operations for this purpose: **`fsf_schedule_timed_job`**, **`fsf_schedule_triggered_job`**, and **`fsf_timed_schedule_triggered_job`**.

2.6 Services for distributed applications

The Distributed FSF modules brings the possibility of reserving and sharing network bandwidth in the same manner it is done for processing capacity. Besides it offers a basic communications programming interface that abstract away the network provider implementation dependent mechanisms used. The implementation of the Core Distributed module is presented in an addendum to this document, but the distributed spare capacity module was not implemented due to time limitations.

2.7 Support for shared resources

The Shared objects module provides services to introduce the effect of mutually exclusive (passive) resources in the system and to set the corresponding critical sections in the contracts. The run time synchronization of servers in the usage of common (shared) resources is expected to be realized with the usual primitives provided by the operating system, like **`pthread_mutex_lock`**, **`pthread_mutex_trylock`** and **`pthread_mutex_unlock`**, but to assure that they will implement a priority inversion avoiding protocol that is appropriate for the FSF scheduler, the mutexes to use are initialized by means of the primitives offered by the shared objects module, whose implementation is presented in detail in Section 6. What it is important to note in this situation, considering the fixed priority based implementation in MaRTE OS, is that the preemption level assigned to each shared object (actually the ceiling), must be set equal to the highest preemption level among those of the servers that will make use of the resources protected by the critical sections included in those servers contracts.

2.8 Implementation-specific services

To keep the API as general as possible, the usage of preemption levels in contracts and shared objects is encapsulated in a separate module. The services offered in this module allow the user to set the concrete priorities at which the tasks in the servers will run, as well as the priority ceilings to be set for the shared

object mutexes. If these values are not set by the user, an automatic priority assignment will be provided for the servers, and the maximum possible priority will be used as the ceiling of shared objects, thus making the system possibly less schedulable.

3 Internal architecture of the implementation

The main architecture of the implemented solution is shown in Figure 2. The scheduling mechanism involves the collaboration of two threads, one is the called *Service Thread*, which is dedicated to perform admission tests and recalculating the priorities, budgets, and periods for the servers when necessary, and the other is the *Scheduler* itself, which is in charge of attending the timing and signalling related events, with low latencies. The scheduling information data structures necessary to implement this collaboration are described next, then the operation of these two threads is presented. Finally the state-space that shapes the operation of the servers, the threads, and the background round-robin scheduler are described.

3.1 Scheduling information data structures

Figure 5 shows an abstract view of the basic components of the scheduling solution given in the present implementation. Each part of this structure is mentioned and described in more detail in the remainder of this section.

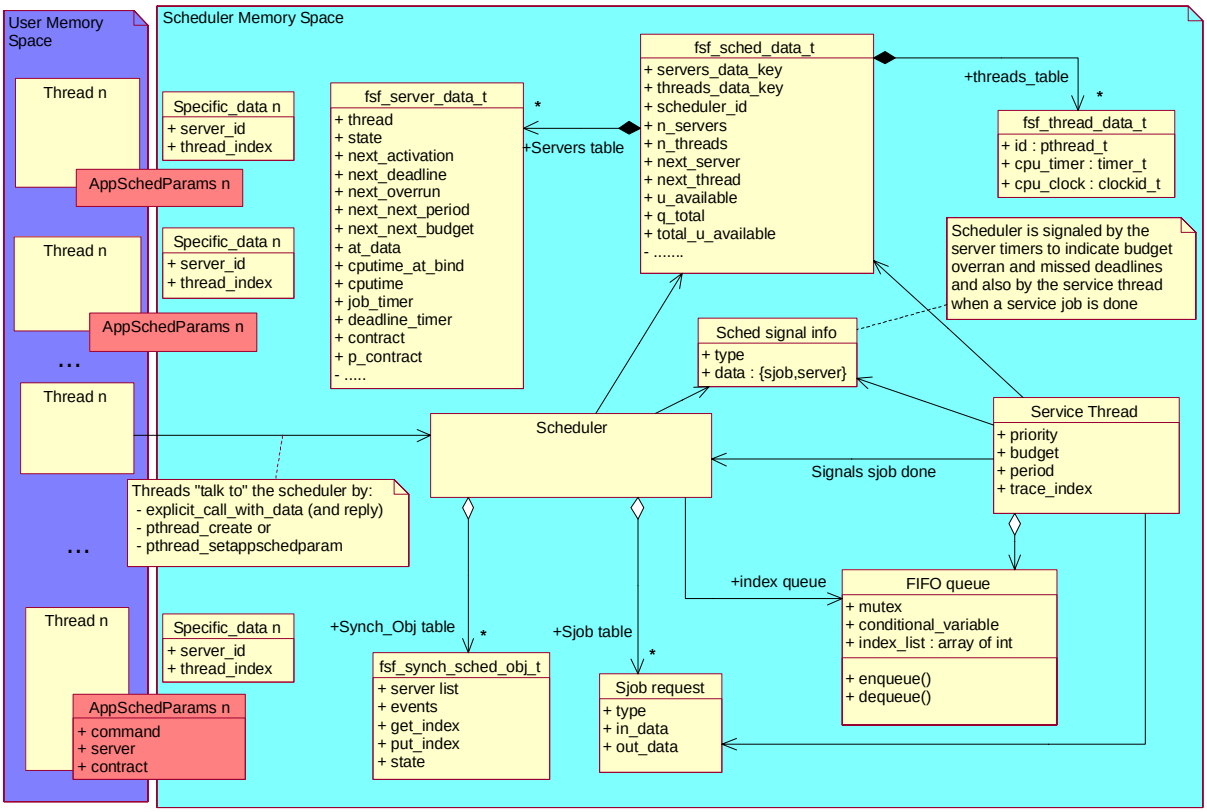

Figure 5. Static abstract view of the scheduler internal data structures and threads

3.1.1 Scheduler and Service Thread shared information

The scheduling mechanism uses a static global data structure to store all the scheduling information. This global structure is shared by the scheduler thread and the service thread, and contains all the data structures shared among these two threads. A summarized UML representation of this structure is shown in Figure 6. The C code of this structure is extracted from the **`fsf_types.h`** header file and is included

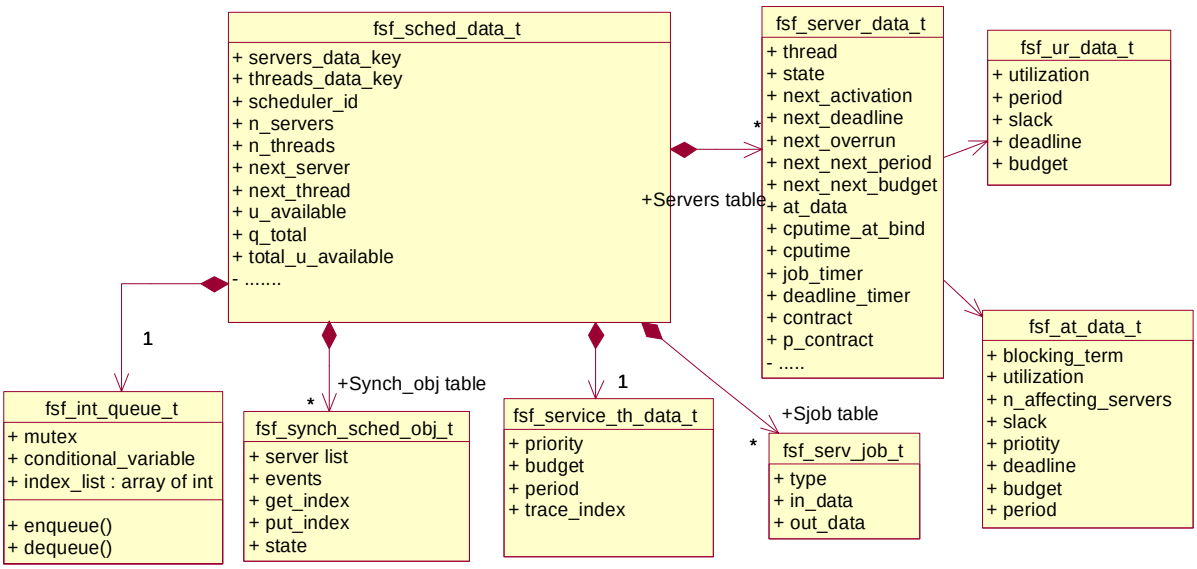

Figure 6. Simplified view of the global scheduler data structure

below for convenience. A common approach for the treatment of lists in this implementation, is the usage of fixed-size arrays and a state variable that indicates whether each record is in use or it is empty.

General global control structure:

```
typedef struct {
  pthread_key_t           servers_key;
  pthread_key_t           threads_key;
  pthread_t               scheduler_id;
  size_t                  msg_type_size;
  size_t                  reply_error_size;
  size_t                  appschedparam_command_size;

  fsf_service_th_data_t   service_th;
  timer_t                 service_th_job_timer;   //activation & refill
  timer_t                 service_th_cpu_timer;   //budget overrun
  fsf_thread_state_t      service_th_state;       //ACTIVE or TIMED

  int                     n_threads;
  int                     next_thread;
  fsf_thread_data_t       threads[FSF_MAX_N_THREADS];

  int                     n_servers; // = 0;
  int                     next_server; // = 0;
  fsf_server_data_t       servers[FSF_MAX_N_SERVERS];

  int                     n_sjobs;
  int                     next_sjob;
  fsf_serv_job_t          sjobs[FSF_MAX_N_SERVICE_JOBS];

  fsf_int_queue_t         queue;

  int                     n_synchobjs;
  int                     next_synchobj;
  fsf_synch_sched_obj_t   synchobjs[FSF_MAX_N_SYNCH_OBJECTS];

  double                  u_available[FSF_N_IMPORTANCE_LEVELS];
  int                     q_total[FSF_N_IMPORTANCE_LEVELS];

  double                  total_u_available;
  double                  next_total_u_available;

  fsf_server_data_t      *at_list;
```

```
  double                final_t_u_a;
  double                trying_t_u_a;

} fsf_sched_data_t;
```

Service thread configuration data structure:

```
typedef struct {
  pthread_t       thread;
  struct timespec period, budget;
  int             priority;
} fsf_service_th_data_t;
```

Other types contained in the global structure are shown next.

3.1.2 Server specific data:

```
typedef struct fsf_server_data_t {
  struct fsf_server_data_t   *next;
  struct fsf_server_data_t   *at_next;
  fsf_server_id_t             server_id;
  struct timespec             last_activation; /* absolute time */
  struct timespec             next_activation; /* absolute time */
  struct timespec             next_deadline;   /* absolute time */
  struct timespec             period;          /* relative time */
  struct timespec             budget;          /* relative cpu time */
  struct timespec             next_period;     /* relative time */
  struct timespec             next_budget;     /* relative cpu time */
  struct timespec             pending_budget;  /* relative cpu time */
  struct timespec            remaining_budget_at_refilling;  /* relative cpu time */
  struct timespec             cputime;         /* absolute cpu time */
  struct timespec             cputime_at_bind; /* absolute cpu time */
  timer_t                     job_timer;   //activations & periodic deadlines
  timer_t                     deadline_timer;  //non periodic deadline misses
  fsf_thread_data_t          *thread;
  fsf_contract_parameters_t  *p_contract;
  fsf_contract_parameters_t   contract;
  double                      u_differential[FSF_MAX_N_UTILIZATION_VALUES-1];
  fsf_at_data_t               at_data;
  fsf_at_data_t               next_at_data;
  fsf_ur_data_t               ur_data;
  double                      tmp_u;
  double                      tmp_a;
  fsf_ur_data_t               try_ur_data;
  char                        state;                    //fsf_server_state_t
  char                        deadline_missed;
  char                        budget_overran;
  char                        reply_deadline_missed;
  char                        sub_state;
  char                       last_renegotiation_status; //fsf_renegotiation_status_t
  char                        synch_obj_index_plus;
  char                        synch_obj_server;
} fsf_server_data_t;
```

3.1.3 Thread specific data

```
typedef enum {FSF_ACTIVE_THREAD, FSF_TIMED_THREAD,
              FSF_UNBOUND_THREAD, FSF_EMPTY_THREAD} fsf_thread_state_t;

typedef struct {
  fsf_thread_state_t          state;
  pthread_t                   id;
  timer_t                     cpu_timer;
  clockid_t                   cpu_clock;
```

```
} fsf_thread_data_t;
```

3.1.4 Service jobs table and index queue

The service job table is filled by the scheduler with those jobs that must be done by the service thread. Then, the index of the job is sent to the service thread through a FIFO queue implemented with a conditional variable. It is important to observe that after finishing the job, the service thread sends a signal to the scheduler with the information field asserted to the *sjob* index already done. The field `event_$_type_queuing_point` indicates what was the kind of event and user function combination that caused the *sjob* to be invoked, and in that way it is possible to select the pending actions to accomplish before returning control to the calling thread if it is the case, or simply finishing the action.

```
typedef enum {FSF_IN_COURSE_SJOB, FSF_DONE_SJOB, FSF_EMPTY_SJOB} fsf_sjob_state_t;

typedef enum {FSF_NOTHING_ELSE,
              FSF_NEW_$_NEGOTIATE,
              FSF_CHANGE_SCHED_PARAM_$_NEGOTIATE,
              FSF_EXPLICIT_CALL_WITH_DATA_$_NEGOTIATE,
              FSF_EXPLICIT_CALL_WITH_DATA_$_CANCEL,
              FSF_EXPLICIT_CALL_WITH_DATA_$_RENEGOTIATE
} fsf_event_type_t;

typedef struct fsf_serv_job_t {
  fsf_sjob_state_t       state;
  pthread_t              thread_id;
  fsf_event_type_t       event_$_type_queuing_point;
  fsf_msg_type_t         type;
  /*admited types are:   FSF_NEGOTIATE_CONTRACT,
                         FSF_CANCEL_CONTRACT,
                         FSF_RENEGOTIATE_CONTRACT,
                         FSF_REQUEST_CONTRACT_RENEGOTIATION,
                         FSF_REQUEST_CHANGE_QUALITY_AND_IMPORTANCE,
                         FSF_RETURN_SPARE_CAPACITY  */
  int                    error;
  union {
    fsf_renegotiate_contract_in_t              negotiate_contract;
    fsf_cancel_contract_in_t                   cancel_contract;
    fsf_renegotiate_contract_in_t              renegotiate_contract;
    fsf_request_contract_renegotiation_in_t    request_contract_renegotiation;
    fsf_request_change_quality_and_importance_in_t
                                              request_change_quality_and_importance;
    fsf_return_spare_capacity_t                return_spare_capacity;
  }                      in;
} fsf_serv_job_t;
```

Queue for the indexes of service jobs required by the scheduler that must be attended by the service thread :

```
typedef struct {
  int              max_size;  //FSF_MAX_N_SERVICE_JOBS
  int              size;
  int              index;
  pthread_mutex_t  mutex;
  pthread_cond_t   cond;
  int             *buffer;  //FSF_MAX_N_SERVICE_JOBS
} fsf_int_queue_t;
```

3.1.5 Synchronization objects

This structure is managed in the application scheduler. Only a handle (the index in the array conveniently complemented with a pseudo-unique mask) is returned to the user. The servers array of this structure is managed by the *put_index* and *get_index* fields.

If a synchronization object has servers queued at the time of destroying it, the servers are unbound and they become ready to start when they are bound again (as if they would have been signalled). In this way both threads and servers are reusable independently from each other.

The signalling time is not stored with the event. Events are just accumulated in a counter. This makes the signalling time independent from the activation time. They happen to be the same just when the server is already waiting for the signal, but not when there are events already accumulated in the synchronization object. In this case the activation time used to evaluate the possible miss of a deadline is the time at which the server asks for the synchronization object to activate it (calling *fsf_schedule_triggered_job* or *fsf_timed_schedule_triggered_job*).

```
typedef struct {
  int                handle;
  char               servers[FSF_MAX_N_SERVERS_IN_SYNCH_OBJECT];
  char               events;
  char               get_index;
  char               put_index;
  char               state; // EMPTY(0) or ACTIVE(1)
} fsf_synch_sched_obj_t;
```

3.1.6 App-sched parameters

The communication between the service contract API in the user space and the scheduler is realized by means of these three functions : `explicit_call_with_data()`, `pthread_setappschedparam()` and `pthread_create()`. In these two latter cases it is necessary to fill a structure of the type `fsf_appschedparam_t` to instruct the scheduler with the kind of action to take.

```
typedef enum { FSF_BIND, FSF_UNBIND, FSF_NEGOTIATE} fsf_command_t;

typedef struct {
  fsf_command_t     command;
  union {
    fsf_server_id_t              server;
    fsf_negotiate_contract_in_t  negotiate_contract;
  }                  val;
} fsf_appschedparam_t;
```

3.1.7 Constants for signalling the scheduler in the info field

Since the scheduler is waiting to receive only one signal, each different action to execute is selected by means of the *info* field that is attached to the signal reception. This field is divided in two parts: the upper 16 bits hold the kind of action to execute, and the lower 16 bits have the index of the target entity in its corresponding list (*service_job* or server). The constants that may be sent in the upper 16 bits part are the following:

```
#define FSF_SERVICE_JOB_DONE          0x00010000
#define FSF_REPLENISH_SERVER          0x00020000
#define FSF_SIGNAL_SERVER             0x00030000
#define FSF_BUDGET_EXPIRED            0x00040000
#define FSF_DEADLINE_NEWJOB_TIMEOUT   0x00050000
```

3.1.8 Explicit data exchanged with the scheduler

When the service contract API in the user space communicate with the scheduler by means of the function `explicit_call_with_data()`, the structures used are of the type `fsf_in_msg_t` and `fsf_out_msg_t`.These are basically union structures that may hold a large number of data types and implement most of the calls that are made to the scheduler. These calls and their inner types can be seen at the beginning of the file **`fsf_types.h`**

3.2 Scheduler operation

As it is usual, the core of the scheduler is an infinite loop waiting for events to occur. As it can be seen in Figure 7, after initialization, the scheduler waits for tasks to be created. At task creation time the scheduler receives the prospective contract parameters, send them to the service thread and after a successful acceptance test and redistribution of the spare capacity, the contract is activated and the control returns to the scheduler. Then, the scheduler programs the timers associated to the server and the timer associated to the CPU clock of the thread. The signal sent by these timers is sent along with the server index as part of the additional information field

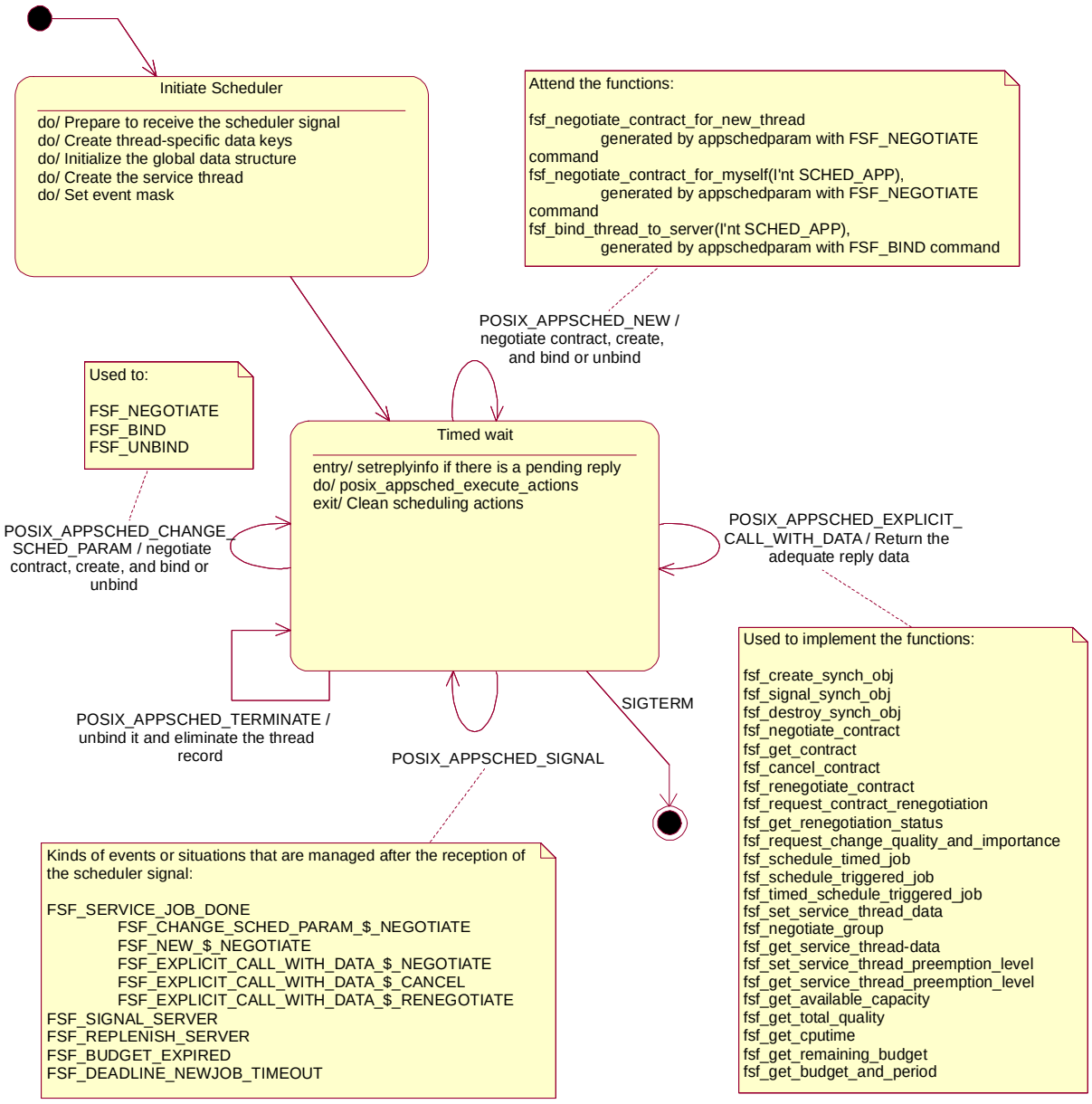

Figure 7. State chart of the FSF_FP scheduler

3.3 Server operation

In this implementation, servers behave basically as the POSIX sporadic servers. Depending on whether their workload is indeterminate or bounded they may go to the background or not, respectively. Figure 8 shows the state chart of an FSF server. It is only when the server is in the active state that its budget is

consumed. If the server has a thread bound to it, it may only execute if the server is in the active state or if being in the timed state the thread gets into the *rr_slice* state (see the following section on the thread states).

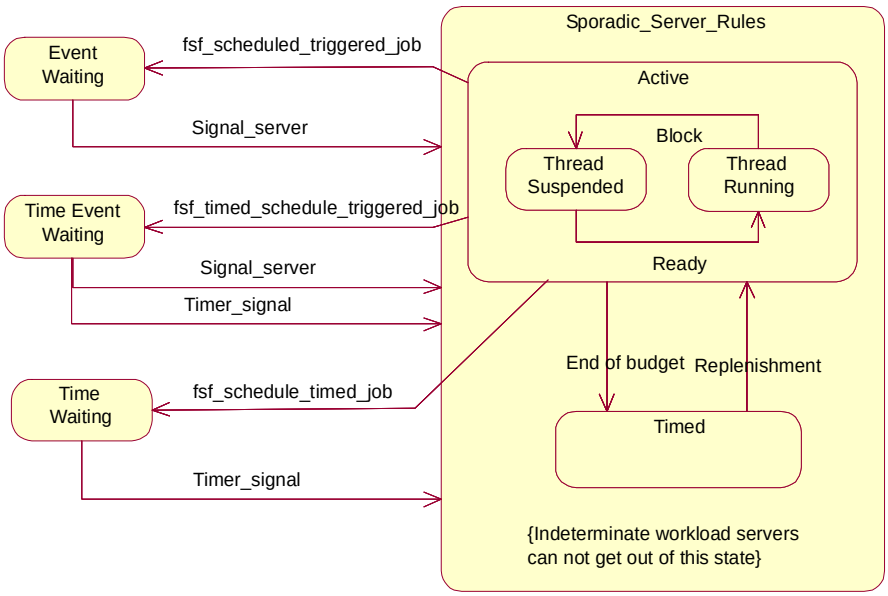

Figure 8. State chart of an FSF server

3.4 Threads states

Figure 9 shows the possible states of a thread by means of a state chart. The Timed state is reached due to exhaustion of the budget assigned by the server to which the thread is bound, while the Blocked state is reached due to auto suspension, or by blocking waiting for any kind of resource. The Unbound state is forced by the manager using the FSF corresponding primitives **`fsf_unbind_thread_from server`** or **`fsf_cancel_contract`**.

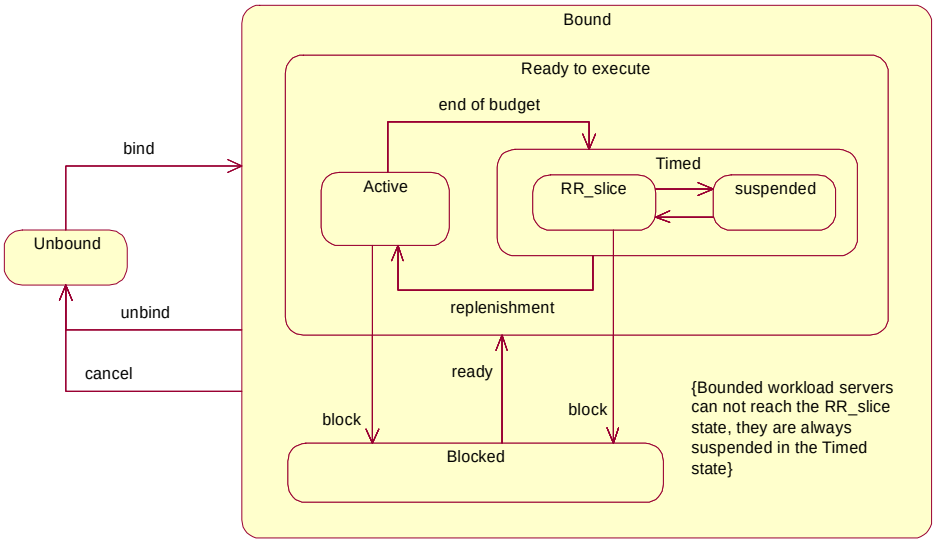

Figure 9. State chart of a thread

3.5 Background round-robin scheduler

Whenever an indeterminate workload server is in the Timed state its bound thread is inserted in a circular singly linked list. Each thread in this list is able to execute for an interval given by the `FSF_RR_SLICE_CPU_TIME` constant (which is in the **`fsf_configuration_parameters.h`** file). The threads bound to specific background servers are also in this list and share the background capacity. All the threads in the background are at the background priority, which is 2 levels below the lowest normal FSF thread priority, given by the `FSF_LOWEST_THREAD_PRIORITY` constant. The background priority value is the lowest priority used in the implementation.

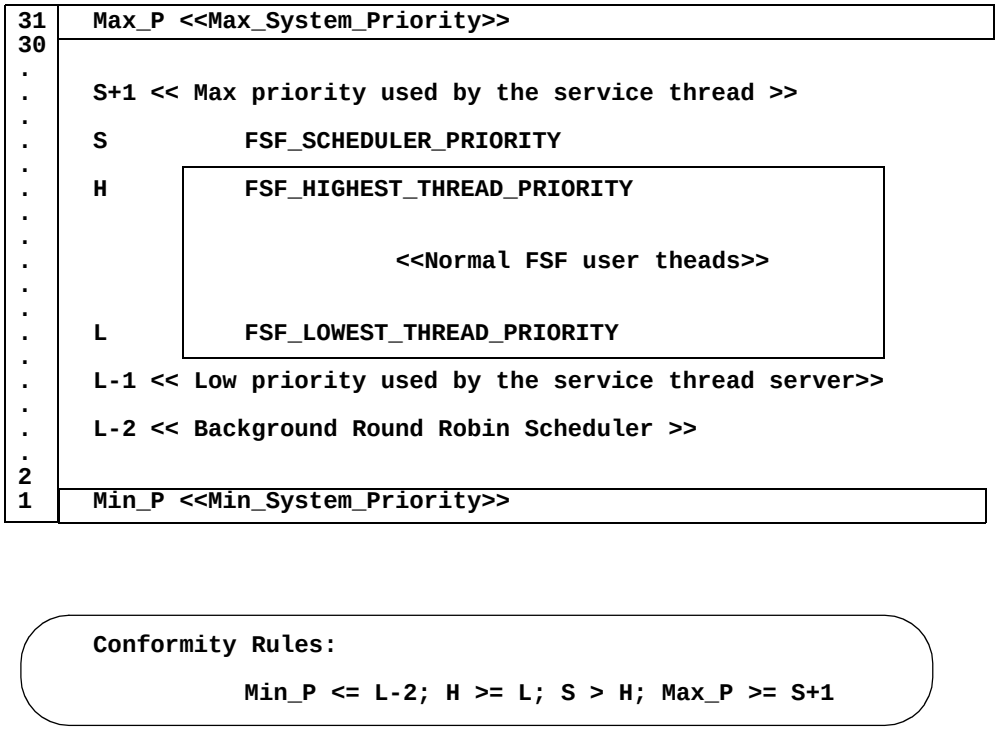

Figure 10. Map of priorities used in the FPS scheduler

3.6 Getting the OS configuration information

The values written in the MaRTE OS **`configuration_parameters.ads`** file are critical to the system operation. To be able to chose the appropriate values in each configuration circumstance, a program that calculates the most sensible values using the **`fsf_configuration_parameters.h`** file is provided. The program is in the **`sizes.c`** file and may be compiled and run in any linux distribution. It uses the **`fsf_configuration_parameters.h`** and the **`fsf_types.h`** files so these headers must be visible from the compilation directory.

During the initialization of the system the coherence of the configuration values given in the file **`fsf_configuration_parameters.h`** is verified, in particular the conformity rules shown in Figure 10 are tested and an error is returned if they are not satisfied.

4 Design of the FSF_Contract_API methods

Table 1 summarizes the way in which each method of the service contract API is implemented.

Table 1: FSF Service Contract methods (library functions)

Method name	Done by	Comments on the implementation
fsf_init	S & ST	initialize the system, creates the scheduler and the service thread
fsf_initialize_contract	CT	assigns default values to a user declared object
fsf_set_contract_basic_parameters fsf_get_contract_basic_parameters fsf_set_contract_timing_requirements fsf_get_contract_timing_requirements fsf_set_contract_reclamation_parameters fsf_get_contract_reclamation_parameters fsf_set_contract_synchronization_parameters fsf_get_contract_synchronization_parameters fsf_set_contract_preemption_level fsf_get_contract_preemption_level	CT	passive library functions that set and get parameters of an already initialized contract object
fsf_create_synch_obj fsf_signal_synch_obj fsf_destroy_synch_obj	S	use *posix_appsched_invoke_withdata*
fsf_negotiate_contract	S & ST	uses *posix_appsched_invoke_withdata*
fsf_negotiate_contract_for_new_thread		uses *pthread_create* with adequate appschedparams
fsf_negotiate_contract_for_myself		*pthread_setappschedparam* commands: negotiate
fsf_bind_thread_to_server	S	*pthread_setappschedparam* commands: bind
fsf_unbind_thread_from_server		*pthread_setappschedparam* commands: unbind
fsf_get_server		uses *pthread_getspecific_from*
fsf_get_contract		use *posix_appsched_invoke_withdata*
fsf_cancel_contrac	S & ST	
fsf_renegotiate_contract		
fsf_request_contract_renegotiation		
fsf_set_service_thread_data		
fsf_negotiate_group		
fsf_get_renegotiation_status	S	
fsf_get_service_thread_data		
fsf_set_service_preemption_level		
fsf_get_service_preemption_level		
fsf_request_change_quality_and_importance	S & ST	
fsf_schedule_timed_job		use *posix_appsched_invoke_withdata,* and the service thread is called for returning the unused capacity
fsf_schedule_triggered_job		
fsf_timed_schedule_triggered_job		
fsf_get_available_capacity	S	use *posix_appsched_invoke_withdata*
fsf_get_total_quality		
fsf_is_admission_test_enabled fsf_strerror	CT	read the configuration parameter
fsf_get_cputime	S	use *posix_appsched_invoke_withdata*
fsf_get_remaining_budget		
fsf_get_budget_and-period		
fsf_init_shared_object fsf_get_shared_object_handle fsf_get_shared_object_mutex fsf_set_shared_obj_preemption_level fsf_get_shared_obj_preemption_level	CT	use a global table shared and protected with a mutex
(S: FSF Scheduler, ST: Service thread, CT: Calling thread)		

Figure 11 shows by means of a sequence diagram the threads that come into play in response to each kind of interaction with which the different functions are implemented. These kinds of function calls are used in Table 1 to describe the API operations.

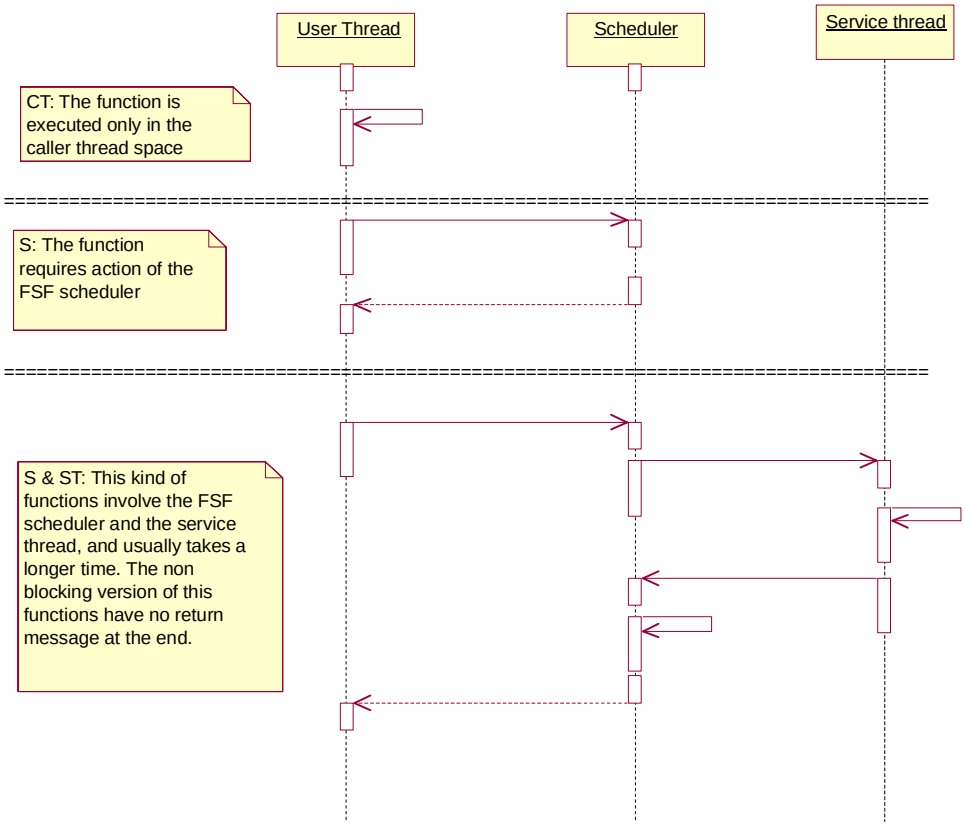

Figure 11. Interaction according to the kind of operation

5 Acceptance test and reclamation algorithm

In order to calculate the schedulability of the admitted servers, a “pessimistic” though simple utilization bound algorithm has been implemented, that can handle tasks with synchronization and deadlines smaller than or equal to the task periods [1]. The test is applied each time a contract is negotiated and maintains or recalculates as necessary a number of accumulative values, which are updated whenever the test is made and a contract admitted, after a successful renegotiation or when a server is to be cancelled. Among these values we calculate the “utilization slack” of each server; the minimum of these slacks is the utilization that will be safely shared by the reclamation algorithm.

The terms that are used in the description of the algorithms, which are also held in a data structure associated to each server, are:

- *U* Utilization, due to its own budget and period and forced by higher or equal priority servers
- *B* Maximum blocking caused by lower priority servers
- *P* Priority considered in the analysis, calculated using the `fsf_priority_map()`function
- *T* Period considered in the analysis
- *D* Deadline considered in the analysis, if it was not specified the maximum period is used.
- *A* Utilization slack or available. It is the difference between U and the server’s “utilization limit”
- *N* One plus the number of higher or equal priority severs

Other terms used:

Servers_set	the set of servers already admitted, it can be empty
i	indicates one server in the set if it is not empty
Critical_sections_set	is the set of critical sections of a certain server
s	a critical section in a set
C	the worst case execution time in a certain critical section
U_available	the total available utilization (the minimum of all the slacks)

The formulation of the modified utilization bound test used is the next:

For the analysis of the server *i* consider all the servers with priority equal to or higher than P_i and group them into the following two sets:

H1, singly preemptive tasks: servers with periods >= T_i

Hn, multiply preemptive tasks: servers with periods < T_i

The effective utilization for server *i* is:

$$f_i = \sum_{j \in H_n} \frac{C_j}{T_j} + \sum_{k \in H_1} \frac{C_k}{T_i} + \frac{B_i + C_i}{T_i}$$

The utilization bound for the server *i* is:

$$n = num(Hn) + 1$$

$$\Delta_i = \frac{D_i}{T_i} \leq 1$$

$$U_{bound}(n, \Delta_i) = \begin{cases} n((2\Delta_i)^{1/n} - 1) + 1 - \Delta_i & ,0,5 \leq \Delta_i \leq 1 \\ \Delta_i & ,0 \leq \Delta_i \leq 0,5 \end{cases}$$

And the schedulability test is :

$$f_i \leq U_{bound}(n, \Delta_i)$$

5.1 Automatic priority assignment

When the *Implementation_Specific* module operations for setting preemption levels of servers or shared objects are not used, the system will use a simple deadline monotonic algorithm to calculate the running priorities of the servers. In this case the maximum priority will be considered for all the critical sections of the system. If the user assigns the preemption level of any server or shared object in the system, it is the user responsibility to do the same for all the other servers and shared objects, and in this case the automatic priority assignment capability is disabled.

5.2 Acceptance test when adding a new server

When a new contract is evaluated, a prospective acceptance test data structure is used to perform the test and calculate all the server values, which includes the reassignment of deadline monotonic priorities if preemption levels are not specified. If the calculated utilizations are below their utilization bounds and correspondingly the available utilization for all the servers is greater than or equal to zero, the contract is admitted and the temporary structure becomes permanent. Then the remaining spare utilization is shared among the servers. To impose the new values, the service thread lowers its own priority level just above the background, waiting for no other server to execute. Then, it immediately increases its priority to a level higher than the scheduler's priority and assigns new budgets and periods to the servers; if the automatic priority assignment capability is enabled, the new priorities are also set for all the servers; this

makes effective the recently accepted contract. Finally, the service thread simply returns to its normal operation priority.

For adding a new server *a* to the current *Servers_set*, the following algorithm is used to update the acceptance test **`fsf_at_data_t`** accumulative structures of each server:

```
-- Recalculate the blocking terms of the present servers:
For i in Servers_set loop
    if P_a  < P_i then
        for s in the Critical_sections_set of a, loop
            if P_s >= P_i then
                B_i = Max(B_i, C_s)
            end if
        end loop
    end if
end loop

-- Calculate the blocking term for the new server:
B_a = 0
For i in Servers_set loop
    if P_i  < P_a then
        for s in the Critical_sections_set of i, loop
            if P_s >= P_a then
                B_a = Max(B_a, C_s)
            end if
        end loop
    end if
end loop


-- Introduce the effect of the new one over the former servers'utilization
For i in Servers_set except for background servers or the service thread loop
    if P_a  >= P_i then
        U_i = U_i + C_a / Min(T_i, T_a)
        n_i = n_i + 1
        A_i = U_bound(n_i, D_i/T_i) - [U_i + B_i/T_i]
        U_available = Min(U_available, A_i)
    end if
end loop

-- Finally calculate the utilization of the new server
U_a = C_a/T_a
n_a = 1
For i in Servers_set loop
    if P_i >= P_a then
        if the new one is not a background server
            U_a = U_a + C_i / Min(T_i, T_a)
            n_a = n_a + 1
        end if
        if server i is neither a background server, nor the service thread
            A_i = U_bound(n_i, D_i/T_i) - [U_i + B_i/T_i]
            U_available = Min(U_available, A_i)
        end if
    end if
end loop
A_a = U_bound(n_a, D_a/T_a) - [U_a + B_a/T_a]
U_available = Min(U_available, A_a)
```

5.3 Acceptance test structures maintenance when deleting a server

For removing a server *a* from the present *Servers_set* the following algorithm is used:

```
-- First take the server a out of the Servers_set 1..n
-- Recalculate the blocking terms:
if the Critical_sections_set of a is not empty then
    for i in Servers_set loop
        CSmax = 0
        if P_a  < P_i then
            for s in the Critical_sections_set of a, loop
                if P_s >= P_i then
                    CSmax = Max(CSmax, C_s)
                end if
            end loop
        end if
        if CSmax >= B_i then
            for j in Servers_set loop except i
                if P_j < P_i then
                    for s in the Critical_sections_set of j, loop
                        if P_s >= P_i then
                            B_i = Max(B_i, C_s)
                        end if
                    end loop
                end if
            end loop
        end if
    end loop
end if

-- Recalculates the utilization for all the servers
U_available = 1.0
For i in Servers_set loop
    if P_a  >= P_i and i is neither a background server nor the service thread then
        U_i = U_i - C_a / Min(T_i, T_a)
        n_i = n_i - 1
        A_i = U_bound(n_i, D_i/T_i) - [U_i + B_i/T_i]
    end if
    U_available = Min(U_available, A_i)
end loop
```

5.3.1 Spare capacity sharing algorithms

When the *U_availabe* obtained after the application of the shown utilization test is greater than zero, this means that there is some capacity beyond the minimum accepted that can be shared among the servers. The way of calculating this capacity is similar to the acceptance test in the sense that it uses an auxiliary **`fsf_ur_data_t`** structure for trying the values to assign, and consolidates the values as they prove to be acceptable by the utilization bound test. The algorithm to do this is:

```
for each importance level in reverse order loop
    calculate the total quality Q_t for this level
    for each server i in this level with discrete granularity in reverse quality
    order loop
        U_share = U_available * Q_i / Q_t
        ([A]) Reduce U_share to adapt to granularity and also calculate the new
        U_available, it must be at least: U_available = U_available - U_share
        Q_t = Q_t - Q_i
    end loop
    for each server i in this level with continuous granularity in reverse quality
    order loop
        U_share = U_available * Q_i / Q_t
        ([B]) Calculate new budget and period for U_share, limit them to Cmax and
        Tmin, and get sure the extra utilization is under the U_share limit, finally
        calculate the new U_available: at least U_available = U_available - U_share
        Q_t = Q_t - Q_i
    end loop
end loop
```

The algorithms used in the sections **([A])** and **([B])** are based in the re-calculation of the available utilizations (applying the acceptance test as necessary) for each of the different values assigned to the budget and period of each server. In **([A],** the algorithm is applied for the values given in the utilization set of each contract, In **([B])**, instead, it is applied first for the evaluation of the smaller possible period and then it there is still extra utilization for the increment of the server's budget. This code is in the **fsf_service_th.c** file and it is has enough comments to understand the successive approximation algorithms.

6 Shared objects management

The initialization of the shared objects and their associated mutexes is performed in the user space. They are inserted in a shared table that is accessed in a mutually exclusive way by means of a protecting mutex. The scheduler is not involved in these operations. The protocol given to the user mutexes is the priority ceiling protocol (the POSIX `PTHREAD_PRIO_PROTECT` protocol), and the ceiling is raised to the maximum of the possible user priorities (`FSF_HIGHEST_THREAD_PRIORITY`). This ceiling may be changed by the user using the `fsf_set_shared_obj_preemption_level` operation of the *Implementation-Specific* module, but in this case the preemption level of all the shared objects and all the servers must be assigned by the user. It is the user responsibility to decide whether to use pre-calculated preemption levels for all servers and shared objects or leave the automatic assignment for them. As it may be expected, even considering that the system's automatic priority assignment uses deadline monotonic order for the servers, since the ceilings are always high, schedulability may be compromised if very long critical sections are used.

7 Implementation in Ada

The Ada implementation of the FSF Ada API, which is presented in the D-SI.5v3 deliverable, is realized as an adaptation layer over the C version. In this way, applications that mix both languages may coexist, and use the FSF services in a coherent way from both programming languages. To keep the configuration parameters centralized in a single file, the Ada source code of the core package specification of the API, the file **fsf.ads**, is obtained by the C preprocessing of the **fsf_ads.c** file. Also the **fsf_c.ads** package is got from the **fsf_c_ads.c** file in this way. The important issue around this mechanism to get the Ada source code is that if it is necessary to make any change to a source code file, the makefile must be updated with the new number of lines of the destination file (see the corresponding rules in the **Makefile** file).

The distributed services module is implemented only in the Ada 95 version of the API, but a similar approach can be used to implement the C API as an adaptation layer on top of the Ada implementation.

8 Organization of the code

The best way to get familiar with the code is by taking a look into the makefile that compiles the FSF and the test applications. Considering that the fsf_v3 directory is the place for decompressing the distribution file, here we show a list of the directories and files that it creates. The actual number of bytes and the dates may change:

```
 Directory of C:\FIRST\fsf_v3

01/04/2005  17:10             4.365 Makefile
05/04/2005  18:33    <DIR>          ada
05/04/2005  18:33    <DIR>          fsf
10/04/2005  16:05    <DIR>          tests
11/04/2005  03:05    <DIR>          files-to-change
               1 files              4.365 bytes
```

```
 Directory of C:\FIRST\fsf_v3\ada

22/03/2005  11:58            10.185 btm.adb
22/03/2005  11:58            13.752 btm3.adb
22/03/2005  11:58            10.681 btm_v3.adb
22/03/2005  11:58             6.007 example.adb
22/03/2005  11:58             3.241 fsf-distributed-spare_capacity.adb
22/03/2005  11:58             2.930 fsf-distributed-spare_capacity.ads
22/03/2005  11:58             5.931 fsf-hierarchical.ads
22/03/2005  11:58             4.995 fsf-implementation_specific.adb
22/03/2005  11:58             2.753 fsf-implementation_specific.ads
22/03/2005  11:58             6.467 fsf-shared_objects.adb
22/03/2005  11:58             2.783 fsf-shared_objects.ads
22/03/2005  11:58            11.983 fsf-spare_capacity.adb
22/03/2005  11:58             6.020 fsf-spare_capacity.ads
22/03/2005  11:58            52.422 fsf.ads
22/03/2005  11:58             2.441 fsf_ada_preprocessing.h
22/03/2005  11:58            20.148 fsf_btm_contract.adb
22/03/2005  11:58             1.066 fsf_btm_contract.ads
04/04/2005  20:12             4.173 fsf_c.adb
22/03/2005  11:58             2.970 fsf_c.ads
22/03/2005  11:58            12.745 fsf_conf_pars.c
22/03/2005  11:58             4.064 new_execution_load.adb
22/03/2005  11:58             2.187 new_execution_load.ads
04/04/2005  16:31            15.086 net_functions.adb
01/04/2005  19:15            10.294 net_negotiation.adb
04/04/2005  03:51            14.026 fsf-distributed.ads
04/04/2005  17:14             9.752 net_tx_time.adb
04/04/2005  16:59             6.504 net_mirror.adb
04/04/2005  20:08             5.025 fsf_ada_preprocessing.c
04/04/2005  20:10             3.787 fsf_c_ads.c
07/04/2005  16:30            52.997 fsf_ads.c
05/04/2004  15:30            63.777 fsf_soloproc.adb
05/04/2004  15:31            16.062 fsf-distributed_soloproc.adb
05/04/2005  17:44             6.705 net_mirror_tasks.adb
03/04/2005  17:17            63.516 fsf_completa.adb
05/04/2004  15:30            63.777 fsf.adb
04/04/2005  21:11            28.365 fsf-distributed_completa.adb
05/04/2004  15:31            16.062 fsf-distributed.adb
              37 files            565.679 bytes

 Directory of C:\FIRST\fsf_v3\fsf

05/04/2005  18:33    <DIR>          include
05/04/2005  18:33    <DIR>          lib
05/04/2005  18:33    <DIR>          source
               0 files                  0 bytes

 Directory of C:\FIRST\fsf_v3\fsf\include

29/11/2004  12:45             3.156 fsf.h
01/04/2005  03:17             6.591 fsf_configuration_parameters.h
10/02/2005  23:04             3.073 fsf_distributed_spare_capacity.h
20/09/2004  00:00               644 fsf_dynamic_reclaiming.h
28/10/2004  09:22             9.042 fsf_hierarchical.h
09/02/2005  09:48             4.121 fsf_implementation_specific.h
14/10/2004  22:17             7.144 fsf_spare_capacity.h
30/03/2005  00:59             2.366 fsf_ss.h
31/03/2005  01:25            23.053 fsf_types.h
17/02/2004  00:00             1.686 stdbool.h
17/02/2004  00:00             3.739 timespec_operations.h
02/04/2005  19:38            45.372 fsf_core.h
02/04/2005  19:54             6.576 fsf_basic_types.h
02/04/2005  21:50             3.745 fsf_opaque_types.h
02/04/2005  22:51             5.602 fsf_shared_objects.h
04/04/2005  03:56            11.920 fsf_distributed.h
              16 files            137.830 bytes

 Directory of C:\FIRST\fsf_v3\fsf\source
```

```
08/04/2005  01:17            30.086 fsf_ss.c
09/04/2005  19:59           162.510 fsf_scheduler.c
02/04/2005  19:18            70.761 fsf_service_th.c
04/04/2005  20:07           108.609 fsf_contract.c
               4 files            371.966 bytes

 Directory of C:\FIRST\fsf_v3\tests

10/03/2005  01:12             1.857 sizes.c
30/03/2005  17:49             7.984 jitter_test.c
24/02/2005  15:43            18.059 fsf_negotiation_test.c
24/02/2005  15:28            14.584 fsf_synch_obj_test.c
23/02/2005  16:06            19.760 context_switch.c
21/02/2005  19:59            48.908 functions.c
               6 files            111.152 bytes

 Directory of C:\FIRST\fsf_v3\files-to-change

19/02/2005  12:57            14.252 configuration_parameters.ads
13/03/2005  03:44             3.599 execution_load.adb
11/03/2005  13:56            16.944 k-file_system.adb
19/02/2005  12:53            18.782 k-mutexes-internals.adb
19/02/2005  12:49            25.458 k-tasks_operations.adb
13/03/2005  03:44             2.848 marte_os.ads
30/07/2004  11:46             2.709 mgnatmake
29/07/2004  19:03            37.073 posix-signals.adb
04/03/2004  00:00             6.645 kernel_console.adb
04/03/2004  00:00            14.727 kernel_console.ads
              10 files             143.037 bytes

     Total files in list:
               74 files      1.334.029 bytes
```

IMPORTANT NOTE: For this deliverable to be implemented in MaRTE OS several changes have been introduced since its original distribution. The version of MaRTE OS that must be used with the D-OS.1v3 accompanying code is MaRTE OS version 1.42a, plus the files included in the `files-to-change` folder.

9 References

[AG01] MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. Mario Aldea Rivas and Michael González Harbour. International Conference on Reliable Software Technologies, Ada-Europe-2001, Leuven, Belgium, LNCS, May 2001.

[AG02] POSIX-Compatible Application-Defined Scheduling in MaRTE OS. Mario Aldea Rivas and Michael González Harbour. Proceedings of 14th Euromicro Conference on Real-Time Systems, Vienna, Austria, IEEE Computer Society Press, pp. 67-75, June 2002.

[KL93] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. González Harbour. “A practitioner’s Handbook for Real-Time Analysis”. Kluwer Academic Pub., 1993.

IST-2001 34140

Architecture of the Distributed FSF Implementation

Annex 1 to Deliverable D-OS1.v3 (Operating Systems)

Responsible: Universidad de Cantabria

Authors: Michael González Harbour, José Javier Gutiérrez,
Julio Medina, José María Martínez and Juan López

12 April 2005

1 Introduction

This document describes the architecture and design of the Distributed First Scheduling Framework (DFSF). This is the first implementation of FSF on a network, and for simplicity reasons it focuses only on the core FSF module, and makes some restrictions. This approach will allow us to rapidly develop and evaluate the core features of the distributed FSF. A future version will eliminate the restrictions and explore the implementation of the distributed spare capacity module.

The scenario we are handling in this document is a single ethernet network using a modified version of the RTEP protocol [4], that we will call DFSF_RTEP. To use more networks, it is possible to replicate the architecture described in this paper.

Some restrictions have been imposed to facilitate the first implementation of DFSF:

- The contract must be negotiated in the node where the *send_endpoint* that is bound to it is created. Otherwise, we would need a distributed synchronization to check that two endpoints are not bound to the same contract.
- Asynchronous negotiations and renegotiations are not implemented, to avoid the need for a special network service thread.
- Deadlines are considered equal to periods.
- Only indeterminate workloads are considered.
- It is mandatory to set the priority (i.e., preemption level) of the message.
- Budget overruns are not reported. An overrunning message is lowered to the background priority until the next replenishment.
- Deadline misses are not reported.
- No hierarchical scheduling; only one send endpoint can be bound to a server.
- If a station is excluded from the network because it does not respond, its contracts are not deleted.

Figure 1 shows an overall picture of the architecture of this implementation. It contains the following modules, whose description appears in the following sections.

- `DFSF.Shared_Info`: This package contains a protected object called `Table`, with the information about the contracts that is shared among all the nodes. The information allows a node to negotiate a new contract, or renegotiate a previous one.
- `DFSF.Servers`: This package contains a protected object called `Table` that stores the information relative to a server that is local to the node where it is created, and therefore needs not be shared among the different nodes. The most important piece of this information is the current budget of each server.
- `DFSF.Negotiation`: This package contains a protected object called `Engine` that implements the state machine associated with the contract negotiations, and all the associated information. To negotiate a contract there is a negotiation token that must be acquired to ensure mutual exclusion. Once the negotiation is finished the infromation must be propagated to all the oher nodes. The information associated with these operations is circulated in the token ring, and this package manages it as a function of the current negotiation state of each node.
- `DFSF_RTEP.Protocol`:

The implementation language is Ada, to ease the development of this complex piece of concurrent software. The implementtaion will be tested in a distributed system with nodes running MaRTE OS [1], and the target application is the FIRST project's robot arm case study [2].

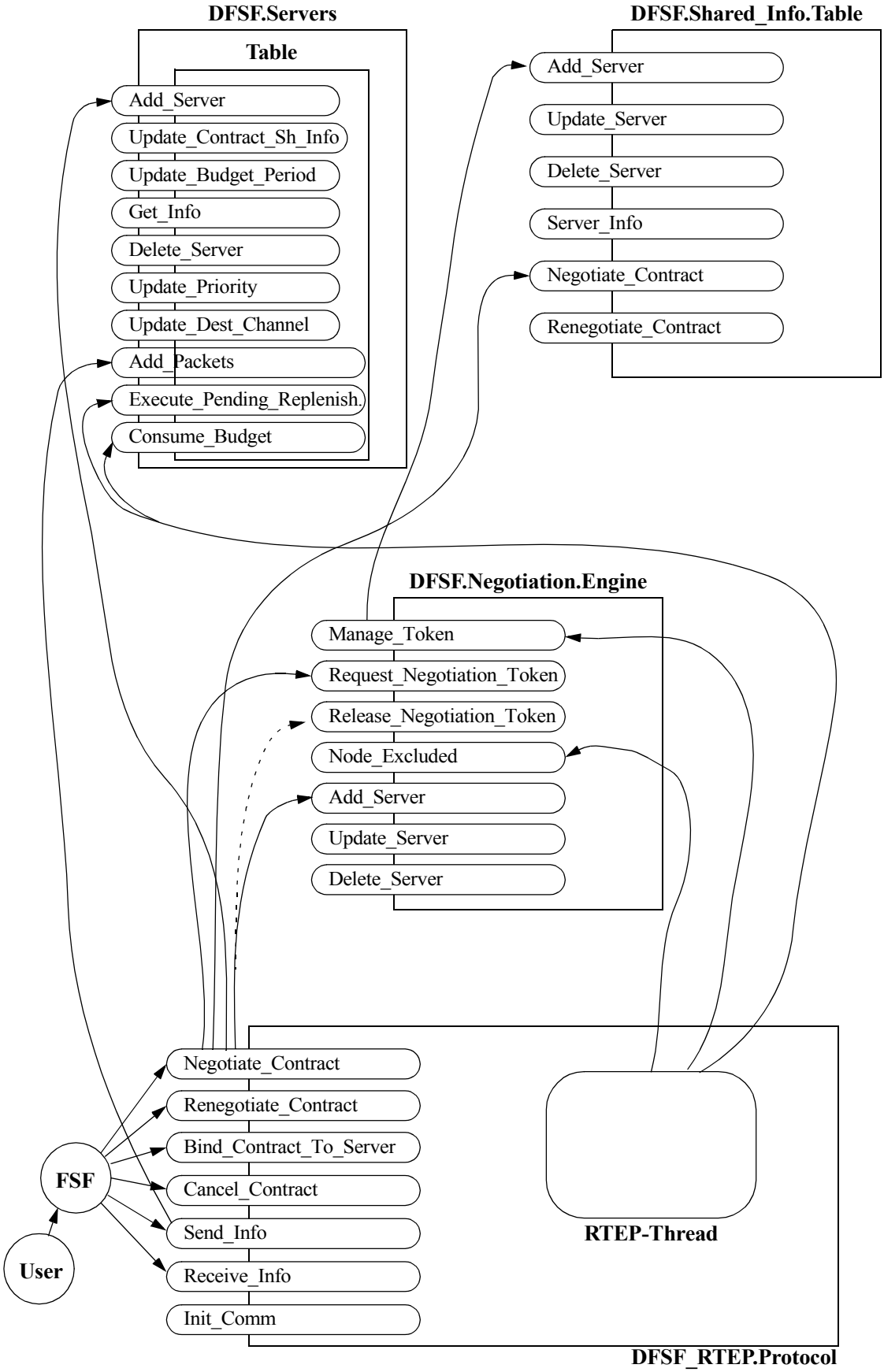

Figure 1. Architecture of the Distributed FSF implementation, using RTEP, and showing the main operations and usage scenarios

2 Information

The Distributed FSF implementation over RT-EP is organized as a set of packages, with a parent called DFSF_RTEP that defines some data types and exceptions. The interface of this package is as follows:

```
-- This is the parent of the Distributed FSF implementation
-- It contains basic types and exceptions

with DFSF_RTEP;

package DFSF is

   pragma Pure;

   -- The network budget is measured in number of packets of maximum size
   -- A small packet consumes the same budget as a large one (one unit)
   type Network_Budget is range 0..2**16-1;
   for Network_Budget'Size use 16;  -- 16 bits

   -- The packet transmission time relates the budget with the time
   -- for the network utilization calculations (nanoseconds)
   Packet_Tx_Time : constant := 500_000; -- nanoseconds

   -- The network max blocking time is used to model the effect of the
   --  non-preemtibility of a packet in the response time of a packet
   --  (nanoseconds)
   Network_Max_Blocking : constant := 500_000; -- nanoseconds

   -- Budget_Error : an attempt was detected to use more budget than
   --                there is available
   Budget_Error : exception;

   -- No_Space: There is no more space for a new server
   No_Space     : exception;

   -- Inexistent: The server or contract specified does not exist
   Inexistent   : exception;

   -- Wrong_State: The operation invoked is not compatible with the
   --              current state of the negotiation
   Wrong_State  : exception;

   -- Bad_Packet_Count: An internal error occurred with the packet accounting
   Bad_Packet_Count : exception;

   -- Not_Supported: An attempt was made to use an unsupported feature
   Not_Supported : exception;

   -- Already_Bound: If attempting to bind an already bounded contract.
   Already_Bound : exception;

   -- Not_Bound: If attempting to un bind an already bounded contract.
   Not_Bound     : exception;

end DFSF;
```

2.1 Shared data structure

There is a data structure containing all the information related to the network contracts. It is replicated in all the nodes. The protocol has a facility for transferring changes to the data structure. An explicit error recovery mechanism is not included because it is already built into the protocol. If the information gets corrupted, the ethernet CRC will be wrong, the packet will be discarded, and a retransmission will occur. If a packet gets lost, a retransmission will occur after a timeout. If a node fails, it will be excluded from the network, and the ring will be reconfigured. Therefore, all the errors that are considered probable are already covered by the RTEP protocol.

The information that is required to be stored is:

- List of server network information. Each element of the list has:

Table 1: Information to be stored for each network contract

Budget_Min: Number of packets
Period_Max: Time_Span
Priority: Integer
Deadline: Time_Span
Server_Id: Integer
Sender_Node_Id: Integer

- Global info, with Service thread info in the future. No service thread will be created for the core FSF, because there is no capacity sharing needed, and the operations will be very fast. In the future, is spare capacity is added, a special service thread different from the regular one should be created, to avoid that delays that can occur in the distributed negotiation could affect the regular negotiation.

The list of server network information items is stored in an array in which each element is identified by a *Server_Network_Info_Id* that acts as the array index. Each array element has a flag indicating whether it is empty or not. New elements are allocated in the first empty element.

The *Server_Id* gives access to all the rest of the information required on the sending node.

Interface. The shared data structure is implemented with a package containing a protected object. It has the following interface:

```
with Ada.Real_Time;
with DFSF_RTEP;
with Generic_Table;
pragma Elaborate_All(Generic_Table);
use Ada;
with Interfaces;
-- This package contains the data structure that contains the
--  information of all the contracts in the network. It is updated by
--  the DFSF negotiation engine, that propagates the information
--  through the RTEP packets.

package DFSF.Shared_Info is

   -- Information required for specifying each contract
   type Contract_Shared_Info is record
      Budget_Min : Network_Budget;
      Server_Id  : DFSF_RTEP.Internal_Server_Handle;
      Period_Max : Real_Time.Time_Span;
      Deadline   : Real_Time.Time_Span;
      Sender_Node_Id : DFSF_RTEP.Station_Id;
      Priority   : DFSF_RTEP.Priority;
   end record;
   for Contract_Shared_Info'Size use 16 + 16 + 64 + 64 + 8 + 16; -- 23 Bytes

   Null_Info : constant Contract_Shared_Info:=
     (0,  DFSF_RTEP.Internal_Server_Id'First,
      Real_Time.Time_Span_Last,
      Real_Time.Time_Span_Last,
      DFSF_RTEP.Station_Id'First,
      DFSF_RTEP.Priority'First);

   -- Maximum number of network contracts (global for all the system)
```

```
   Max_Contracts : constant:=
     DFSF_RTEP.Number_Of_Internal_Servers*DFSF_RTEP.Number_Of_Stations;

   -- Identifier of the contract information in the table

   type Contract_Shared_Info_Handle is range 0..Max_Contracts;
   for Contract_Shared_Info_Handle'Size use 16;  --  use only up to 14 bits
   subtype Contract_Shared_Info_Id is Contract_Shared_Info_Handle
     range 1..Max_Contracts;


   Null_Contract_Shared_Info_Handle : constant
Contract_Shared_Info_Handle:=0;

   package Shared_Table_Pkg is new
     Generic_Table(Contract_Shared_Info_Id,Contract_Shared_Info);

   -- Protected object that contains the table with all the information
   -- for each contract

   protected Table is

      -- Add a server contract that was negotiated in some other node
      -- may raise No_Space. The Id of the new server is returned
      -- It raises No_Space if there is no space for the new server
      procedure Add_Server
        (Info : in Contract_Shared_Info;
         Id   : out Contract_Shared_Info_Id);

      -- Update the server contract identified by Id with the new info.
      --  It raises Inexistent if the Id does not reference
      --  a valid contract in the table
      procedure Update_Server
        (Info : in Contract_Shared_Info;
         Id   : in Contract_Shared_Info_Id);

      -- Delete the server contract identified by Id from the table
      --  It raises Inexistent if the Id does not reference
      --  a valid contract in the table
      procedure Delete_Server
        (Id   : in Contract_Shared_Info_Id);

      -- Return the contract information of the the server contract
      --  identified by Id
      --  It raises Inexistent if the Id does not reference
      --  a valid contract in the table
      function Server_Info
        (Id   : in Contract_Shared_Info_Id)
        return Contract_Shared_Info;

      -- Negotiate a new server contract. Accepted returns whether the
      --  new contract was accepted or not. If accepted, the Id of the
      --  new server is returned.
      --  It raises No_Space if there is no space for the new server
      procedure Negotiate_Contract
        (Info     : in Contract_Shared_Info;
         Id       : out Contract_Shared_Info_Id;
         Accepted : out Boolean);
      -- may raise No_Space

      -- Renegotiate the server contract identified by Id. Accepted
      --  returns whether the new contract info was accepted or not.
      --  It raises Inexistent if the Id does not reference
      --  a valid contract in the table
      procedure Renegotiate_Contract
        (New_Info : in Contract_Shared_Info;
         Id       : in Contract_Shared_Info_Id;
         Accepted : out Boolean);

      -- Return the total network utilization by the current servers
```

```
      function Total_Utilization return Float;

   private
      T : Shared_Table_Pkg.Element_Table;
      U_Total : Float:=0.0;
      N_Total : Natural:=0;

   end Table;

end DFSF.Shared_Info;
```

2.2 Server information

Each node has a list of servers, each identified with a *Server_Id* that is an integer value valid for its node. The list is as the list of server network information. It is stored in an array in which each element is identified by a *Server_Id* that acts as the array index. Each array element has a flag indicating whether it is empty or not. New elements are allocated in the first empty element.

Each server contains the information described in the following table:

Table 2: Information to be stored for each server

Server_Network_Info_Id: integer
Current_Budget: integer (number of packets)
Max_Allocated_Budget: integer (number of packets)
Server_Period : time span
Destination_Node: node_id
Port: port_id

The *Server_Network_Info_Id* is a reference to the information in the shared distributed information.

Budgets are expressed as a number of maximum-size packets. A smaller packet consumes one unit, the same as a maximum-size packet. The server stores the maximum budget, and the current budget. Initially, the current budget is set equal to the maximum. It also stores the *Server_Period*, which must be made equal to the *Period_Max* value of the server.

The *Destination_Node* and *Port_Id* information are stored in the endpoint data structure, and replicated here for convenience.

Interface. The servers data structure is implemented with a package that has the following interface

```
with Ada.Real_Time;
with DFSF_RTEP;
with DFSF.Shared_Info;
with Generic_Table;
with Priority_Queues;
pragma Elaborate_All (Priority_Queues);
with Queues;
pragma Elaborate_All (Queues);
use Ada;

-- This package contains the table with information associated with each
--  of the distributed FSF servers in a specific station.  The
--  information includes the current budget of the server, and the
--  associated replenishment queue. Operations are included to manage
--  the budgets

package DFSF.Servers is
```

```
-- Identifies a distributed FSF server inside the station where it
--  was created
subtype DFSF_Server_Id is DFSF_RTEP.Internal_Server_Id;

-- Information associated with each of the distributed FSF servers
--  in a specific station
type DFSF_Server_Info is record
   Contract_Info        : DFSF.Shared_Info.Contract_Shared_Info_Id;
   Max_Allocated_Budget : Network_Budget;
   Server_Period        : Real_Time.Time_Span;
   Destination          : DFSF_RTEP.Station_Handle;
   Channel_Id           : DFSF_RTEP.Channel;
   Priority             : DFSF_RTEP.Priority;
end record;

-- Implementation-specific packages and types

package Server_Table_Pkg is new
  Generic_Table(DFSF_Server_Id,DFSF_Server_Info);

type Repl_Operation is record
   Amount  : Network_Budget;
   At_Time : Real_Time.Time;
end record;

package Repl_Times is new Priority_Queues
  (Size     => DFSF_RTEP.Number_Of_Internal_Servers,
   Element  => DFSF_Server_Id,
   Priority => Real_Time.Time,
   ">"      => Real_Time."<",
   "="      => DFSF_RTEP."=");

package Server_Queues is new Priority_Queues
  (Size     => DFSF_RTEP.Number_Of_Internal_Servers,
   Element  => DFSF_Server_Id,
   Priority => DFSF_RTEP.Priority,
   ">"      => DFSF_RTEP."<",
   "="      => DFSF_RTEP."=");

package Repl_Queues is new Queues
  (DFSF_RTEP.Max_RTEP_Packet_At_A_Time,Repl_Operation);

type Budget_Array is array (DFSF_Server_Id) of Network_Budget;

type Repl_Queue_Array is array(DFSF_Server_Id) of Repl_Queues.Queue;

-- Protected object containing the server information that may be
--  shared among several threads. The budget information is outside
--  the protected object, beccause it is used only by the RTEP
--  thread

protected Table is

   --  Add a new server with the specified attributes. The Id of the
   --  new server is returned in Id.
   --  It raises No_Space if there is no space for the new server
   procedure Add_Server
     (Max_Allocated_Budget : in Network_Budget;
      Server_Period        : in Real_Time.Time_Span;
      Server_Priority      : in DFSF_RTEP.Priority;
      Id                   : out DFSF_Server_Id);

   -- Delete the server identified by Id
   -- It raises Inexistent if Id does not refere to a valid server
   procedure Delete_Server
     (Id : in DFSF_Server_Id);

  -- Update the contract info id attribute of the server identified by Id
   -- It raises Inexistent if Id does not refer to a valid server
```

```
      procedure Update_Contract_Shared_Info
        (Id       : in DFSF_Server_Id;
         Net_Info : in DFSF.Shared_Info.Contract_Shared_Info_Id);

      -- Update the priority attribute of the server identified by Id
      -- It raises Inexistent if Id does not refer to a valid server
      procedure Update_Priority
        (Id              : in DFSF_Server_Id;
         Server_Priority : in DFSF_RTEP.Priority);

      -- Update the budget and period attributes of the
      -- server identified by Id
      -- It raises Inexistent if Id does not refer to a valid server
      procedure Update_Server_Budget_And_Period
        (Id                   : in DFSF_Server_Id;
         Max_Allocated_Budget : in Network_Budget;
         Server_Period        : in Real_Time.Time_Span);

      -- Update the destination and channel id attributes of the server
      -- identified by Id
      -- It raises Inexistent if Id does not refer to a valid server
      procedure Update_Station_And_Channel
        ( Id : in DFSF_Server_Id;
          Destination : in DFSF_RTEP.Station_Handle;
          Channel_Id  : in DFSF_RTEP.Channel);

      -- Return the information of the  server identified by Id
      -- It raises Inexistent if Id does not refer to a valid server
      function Get_Info
        (Id : in DFSF_Server_Id)
        return DFSF_Server_Info;


      -- Return the current budget of the server identified by Id
      -- It raises Inexistent if Id does not refer to a valid server
      function Current_Budget
        (Id : in DFSF_Server_Id)
        return Network_Budget;

      -- Execute all the pending replenishments, and return the current
      -- budget of the server identified by Id
      -- It raises Inexistent if Id does not refer to a valid server
      procedure Execute_Pending_Replenishments
        (Highest_Priority_Server : out DFSF_RTEP.Internal_Server_Handle;
         Highest_Priority        : out DFSF_RTEP.Priority);

      -- Add the specified amount of packets to the Packet_Count,
      -- for the server identified by Id
      -- It raises Inexistent if Id does not refer to a valid server
      procedure Add_Packets
        (Id        : in DFSF_Server_Id;
         Amount    : Network_Budget);

      -- Consume the specified amount of budget (limited to the
      --  current budget), programming its corresponding
      --  replenishment operation, for the server identified by Id
      --  It also decrements from the packet count
      --  It raises Inexistent if Id does not refer to a valid server
      procedure Consume_Budget
        (Id        : in DFSF_Server_Id;
         Amount    : Network_Budget;
         Timestamp : Real_Time.Time);

   private

      -- Table with the servers
      T : Server_Table_Pkg.Element_Table;

      -- Current budget for every server
      Budget : Budget_Array:= (others => 0);
```

```
      -- Current number of packets pending to be sent for every server
      Packet_Count : Budget_Array:= (others => 0);

      -- Replenishment queues
      Repl_Queue : Repl_Queue_Array;

      -- Queue of first replenishment time for each server
      Server_Repl_Times : Repl_Times.Queue;

      -- Queue of active servers (with packets pending) ordered by their
      -- active priority
      Server_Priorities : Server_Queues.Queue;

   end Table;

end DFSF.Servers;
```

Internally, each server has a replenishment queue associated. It is a FIFO queue of pending replenishment operations, each with a natural number representing the amount of budget to replenish, and with the replenishment time. Its size is the same as the size of the send queue (configuration value). Its operation is as follows:

- If an item is inserted that has the same replenishment time as the last item, instead of creating a new entry in the queue, the budget amounts are added together.
- If the queue is full and a new replenishment is queued, its budget is added to the last item in the queue, and its replenishment time is made equal to that of the new element.
- There is a time value called “server_replenished”; it is made equal to the current time during initialization.
- To execute a pending replenishment (one that has a replenishment time less than or equal to the current time), the current budget is increased by the associated amount; in addition, if the current budget was zero, the time value called “server_replenished” is made equal to the replenishment time of the executed replenishment.
- To consume the budget, it is decremented by the specified amount, and a replenishment operation is queued, for that amount and for a replenishment time equal to the sum of the server’s period (*Period_Max*), and the maximum of the message’s timestamp and the “server_replenished” time.

3 Mechanism to communicate changes to the scheduling table

The packets circulating under RTEP (including tokens, transmit tokens, and info packets) have a fixed-length field for FSF information, which contains the following information

Table 3: FSF Information to be sent with the packet

FSF_Info_Code: integer with the following possible values: • *No_Op* • *Claim_Negotiation_Token* • *Wait_For_Action* • *Release_Negotiation_Code* • *Add_Server* • *Delete_Server* • *Update_Server*
Executive : enumeration of: (initial ,executive_power,final)
Negotiation_Token_Holder

Table 3: FSF Information to be sent with the packet

Server_Network_Info_Id
Server_Network_Info:

The *Executive* flag exists because the node acquiring the negotiation token does not generally coincide with the node acting as the *Token_Master* in the regular RTEP behaviour. The flag is used to add executive power to the *FSF_Info_Code*. Except for the *No_Op* and *Wait_For_Action* codes, if the executive flag is *initial* or *final*, the FSF information is passed to the next node and no further action is taken; if its value is *Executive_Power*, the *FSF_Info_Code* is interpreted and the corresponding actions appropriate for the node's state are executed, according to the description below. Initially, except for a node acting as a *Token_Master*, when an *FSF_Info_Code* different from *No_Op* or *Wait_For_Action* is generated, it is passed with the *Executive* flag set to *Initial* (we will call it a *non-executive* code). When a new *Token_Master* receives such a code for the first time or if it is the new *Token_Master* itself the node generating the code, it sets the *Executive* flag to *Executive_Power* (thus converting it to an *executive* code). When it receives it again after a full token rotation, it sets it to *Final*. In this way, we can guarantee that each code requiring an action is received by all the servers with *Executive_Power* during a full token rotation, thus guaranteeing that all the nodes receive it. The *No_Op* and *Wait_For_Action* codes are always sent with the flag set to *Initial*.

The *Negotiation_Token_Holder* field is set equal to the node acquiring the negotiation token, and it is set back to indicate a *null* node when the negotiation token is released.

The *Server_Network_Info_Id* field is only set for *Add_Server*, *Replace_Server*, or *Delete_Server* operation codes, to indicate which server to affect.

The *Server_Network_Info* field is only set for *Add_Server* or *Update_Server* operation codes, to indicate the server network info of the corresponding server (see Table 1).

The RTEP regular operation, including any error recovery actions, is not modified by the negotiation to-

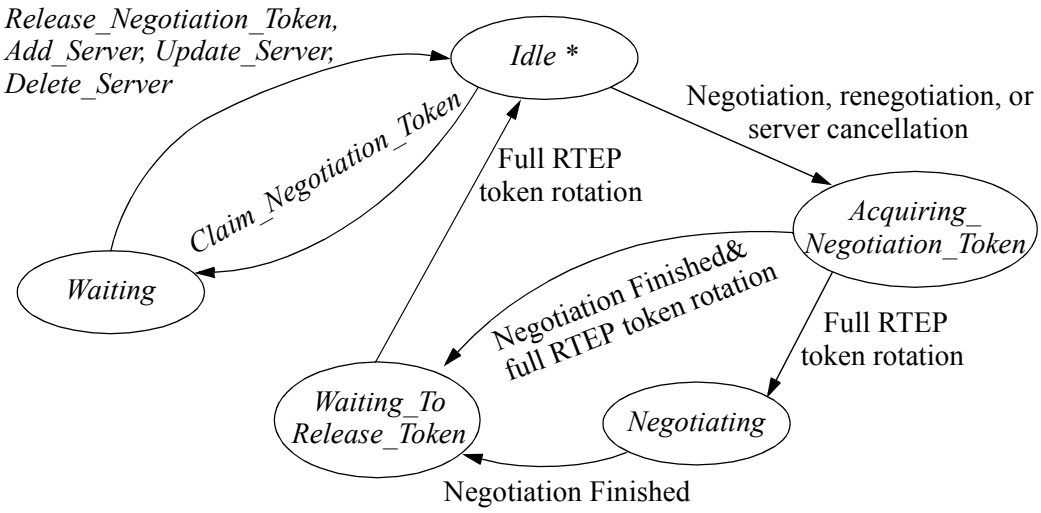

Figure 2. State diagram of the FSF distributed negotiation.
All the operation codes shown are "executive".
State changes are only allowed right after sending a packet

ken or state. The network nodes can be in five states regarding the FSF information exchange. Figure 2 shows the state diagram. The state changes are only allowed right after sending a packet. The operations of each state are as follows:

- *Idle*: This is the initial state. The first *Token_Master* sends a *No_Op* code. When a node receiving a *No_Op* code wishes to perform a negotiation, renegotiation, or server cancellation operation it sends a *Claim_Negotiation_Token*, and switches to the *Negotiation_Token_Holder* state. Otherwise, a *No_Op* code is sent. If a *Claim_Negotiation_Token* code is received, the node retransmits the token and switches to the *Waiting* state.
- *Waiting*: A node in this state recirculates the operation code received in the packet with no modification, except for the case of excluding a node (see below). After an *executive Release_Negotiation_Token* code is received and retransmitted, the node switches to the *Idle* state. If an *executive Add_Server*, *Update_Server* or *Delete_Server* code is received, the node performs the requested action, retransmits the same operation code, and then switches to the *Idle* state. If a *Claim_Negotiation_Token* or a *Wait_For_Action* code is received and the following node in the ring has to be excluded because it does not respond (see the description of the RTEP protocol), and it is the *Negotiation_Token_Holder*, a *Release_Negotiation_Token* code is sent instead of the received operation code.
- *Acquiring_Negotiation_Token*. In this state the node is holding the negotiation token while the other nodes are being informed about it. Therefore, in this state the node can perform a negotiation with guarantee that the shared information is not being changed in other nodes. If a *non-executive Claim_Negotiation_Token* code is received before having received an *executive* one, it is retransmitted with no other effect. If an *executive Claim_Negotiation_Token* code is received and the node is not the *Token_Master*, it is also retransmitted with no other effect

  If a *non-executive Claim_Negotiation_Token* code is received after having received an *executive* one, or if an *executive Claim_Negotiation_Token* code is received and the node is the current *Token_Master*, the following happens:

  - If the negotiation, or renegotiation operation is finished, if the new server is not accepted, a *Release_Negotiation_Token* code is sent, and the node switches to the *Waiting_To_Release_Token* state.
  - If the negotiation or renegotiation is finished and the new server is accepted, or if it is a server cancellation operation, the corresponding *Add_Server*, *Update_Server* or *Delete_Server* code is sent, and the node switches to the *Waiting_To_Release_Token* state.
  - If the negotiation or renegotiation is in progress, a *Wait_For_Action* code is sent and the node switches to the *Negotiating* State
- *Negotiating*. In this state the node can continue the negotiation or renegotiation started during the *Acquiring_Negotiation_Token* state. While the negotiation is in progress, if a packet has to be sent, another *Wait_For_Action* code is sent. When the negotiation, or renegotiation operation is finished, upon receiving a packet, the following happens:

  - If the new server is not accepted, a *Release_Negotiation_Token* code is sent, and the node switches to the *Waiting_To_Release_Token* state.
  - If the negotiation or renegotiation is accepted, the corresponding *Add_Server* or *Update_Server* code is sent, and the node switches to the *Waiting_To_Release_Token* state.
- *Waiting_To_Release_Token*. In this state, when an non executive packet is received after having received an executive one, or if an *executive* packet is received and the node is the current *Token_Master*, the node sends a *No_Op* packet and then switches to the *Idle* state. Otherwise, the received packet is retransmitted.

Interface. The state diagram and the operations described in this section are executed inside the following protected object:

```
with DFSF_RTEP;
with DFSF.Shared_Info;
```

```
-- This package contains the negotiation engine for the distributed
-- FSF It contains the operations required to manage the negotiation
-- state, and manages the DFSF information that is sent with each
-- RTEP packet

package DFSF.Negotiation is

   -- Operation codes transmitted with the RTEP packets
   type DFSF_Op_Code is
     (No_Op, Claim_Negotiation_Token, Wait_For_Action,
      Release_Negotiation_Code, Add_Server, Delete_Server, Update_Server);
   for DFSF_Op_Code'Size use 8;
   for DFSF_Op_Code use (No_Op    => 0,
                         Claim_Negotiation_Token => 1,
                         Wait_For_Action => 2,
                         Release_Negotiation_Code => 3,
                         Add_Server  => 4,
                         Delete_Server => 5,
                         Update_Server => 6);

   -- State of the negotiation engine for each station
   type DFSF_Negotiation_State is
     (Idle, Waiting, Acquiring_Negotiation_Token,
      Negotiating, Waiting_To_Release_Token);

   -- Executive power of the DFSF operation code
   type Executive_State is (Initial, Executive_Power, Final);
   for Executive_State'Size use 8;
   for Executive_State use (Initial => 0,
                            Executive_Power => 1,
                            Final => 2);


   -- Information transmitted with the RTEP packets
   type DFSF_Packet_Info is record
      Op_Code                  : DFSF_Op_Code;
      Executive                : Executive_State;
      Negotiation_Token_Holder : DFSF_RTEP.Station_Handle;
      Id                       : DFSF.Shared_Info.Contract_Shared_Info_Handle;
      Info                     : DFSF.Shared_Info.Contract_Shared_Info;
   end record;
   for DFSF_Packet_Info'Size use 8 + 8 + 16 + 16 + 23 * 8; -- 29 bytes
   pragma Pack(DFSF_Packet_Info);

   type Negotiation_Result_Type is
     (Idle, In_Progress, Rejected, New_Server_Accepted,
      Update_Server_Accepted, Delete_Server_Accepted);

   ---New...
   type Token_Master_Status is
     (Start_Rotation, End_Rotation, Not_A_Token_Master);

   -- Protected object that contains the negotiation state and manages
   -- the DFSF information sent with the RTEP
   protected Engine is

      -- Initialize the Engine with the Current Node Id
      procedure Init
        (Current_Node : DFSF_RTEP.Station_Id);

      -- Get the initial DFSF packet information, for the first token sent
      procedure Get_Initial
        (Info : out DFSF_Packet_Info);

      -- Manages the DFSF packet information. The information received
      --  in the incoming packet is provided to this operation, and is
      --  updated according to the state. The updated information is
      --  sent with the outgoing packet. The station has to indicate
      --  whether it has the role of the current Token Master or not.
      procedure Manage_Token
```

```
         (Info : in out DFSF_Packet_Info;
          Token_Master : in Token_Master_Status);

      -- This entry suspends the calling thread until the negotiation
      --  token is acquired. The identifier of the current node is
      --  provided in the call
      entry Request_Negotiation_Token;

      -- This procedure releases the negotiation token. It is used
      --  after a rejected negotiation or renegotiation to return the
      --  token
      --  It raises wrong_state if node does not hold the negotiation token
      procedure Release_Negotiation_Token;

      -- This procedure is invoked by the RTEP thread to indicate that
      --  a station -- was excluded from the ring, because it does not
      --  respond. The packet information is updated in the call, if
      --  necessary
      procedure Node_Excluded
        (Excluded_Node : DFSF_RTEP.Station_Id;
         Info          : in out DFSF_Packet_Info);

      -- This function indicates whether the current station is
      --  holding the negotiation token or not.
      function Has_Negotiation_Token return Boolean;

      -- This procedure is invoked to indicate that a negotiation was
      --  accepted, and therefore its information must be propagated
      --  to all the nodes in the system, and then the negotiation
      --  token must be released.
      --  It raises wrong_state if node does not hold the negotiation token
      procedure Add_Server
        (Id   : DFSF.Shared_Info.Contract_Shared_Info_Id;
         Info : DFSF.Shared_Info.Contract_Shared_Info);

      -- This procedure is invoked to indicate that a renegotiation was
      --  accepted, and therefore its information must be propagated
      --  to all the nodes in the system, and then the negotiation
      --  token must be released.
      --  It raises wrong_state if node does not hold the negotiation token
      procedure Update_Server
        (Id   : DFSF.Shared_Info.Contract_Shared_Info_Id;
         Info : DFSF.Shared_Info.Contract_Shared_Info);

      -- This procedure is invoked to indicate that a server was
      --  cancelled, and therefore this event must be propagated
      --  to all the nodes in the system, and then the negotiation
      --  token must be released.
      --  It raises wrong_state if node does not hold the negotiation token
      procedure Delete_Server
        (Id : DFSF.Shared_Info.Contract_Shared_Info_Id);

      -- Return the state of the negotiation engine
      function State return DFSF_Negotiation_State;

   private

      Self_Id : DFSF_RTEP.Station_Handle:=0;
      Current_State : DFSF_Negotiation_State:=Idle;
      Holding_Negotiation_Token : Boolean:=False;
      Negotiation_Result : Negotiation_Result_Type:=Idle;
      Shared_Id : DFSF.Shared_Info.Contract_Shared_Info_Id;
      Shared_Info : DFSF.Shared_Info.Contract_Shared_Info;

   end Engine;

end DFSF.Negotiation;
```

4 RT-EP with budget control

Currently, RTEP has the following user-level operations:

- Init_Comm
- Send_Info
- Receive_Info
- Try_Receive_Info

To support the FSF contracts, in the `Send_Info` procedure the *Destination_Station_Id* and the *Channel_Id* parameters are replaced with a *Server_Id* parameter, which contains the information necessary to identify the send endpoint (including the destination and channel).

In addition, we have to add new operations to perform the negotiations, renegotiations, and cancellation of servers. These are blocking operations that suspend the calling thread while the negotiation is in progress.

The new user interface of RTEP is therefore as follows. There is a parent package that contains the configuration data, exceptions, and basic types. Its specification is:

```
with Interfaces;

package DFSF_RTEP is

   pragma Pure;

   -- RTEP_Task_Prio: Specifies the priority of the RT-EP internal
   --                 communication task. It also sets the ceiling of the
   --                 protected objects involved.
   RTEP_Task_Prio: constant := 30;

   -- Maximum number of user bytes per packet (29 bytes for the DFSF headers)
   Max_Rt_Ep_MTU : constant := 1492 - 29;

   -- Number of reception channels
   Number_Of_Channels : constant := 10;

   -- Number of internal DFSF servers per station
   Number_Of_Internal_Servers : constant:=10;

   -- Maximum number of packets that can be queued with the same priority
   Max_Queued_Element_Same_Priority : constant Integer := 10;

   -- Maximum number of simultaneous pending packets
   Max_RTEP_Packet_At_A_Time : constant Integer :=
     Max_Queued_Element_Same_Priority * 256 * Number_Of_Channels;

   -- In Number_Of_Stations we define the number of configured stations.
   Number_Of_Stations : constant := 2;

   -- identifier of the RTEP protocol
   Rt_Ep_Protocol_Number : constant := 16#1000#;

   -- Device name
   Device_Name : constant String := "/dev/eth0";

   -- Max_retries for packet retransmission, before excluding a station:
   RTEP_Error_Max_Retries : constant := 3;

   ---------------
   --  Timeouts --
   ---------------
   -- Timeout to determine that a packet has been lost
   --  The time MUST be in nanoseconds.
```

```
RTEP_Communication_Timeout : constant := 250_000_000; --  250; -- 250 usec
--  The time MUST be in nanoseconds.
RTEP_Comunication_Initialitation_Timeout : constant
  := 10_000_000_000;-- 1 sec.


------------
--  Delay --
------------
--  Delay between receiving and sending a token.
--  With this delay we can reduce the processor overhead.
--  The Time MUST be in nanoseconds.

subtype Enable_RTEP_Delay is Boolean range True .. True;

RTEP_Delay : constant := 80_000;  --  80 usec

------------------
--  Debug_Modes --
------------------

subtype Enable_RTEP_Core_Debug is Boolean range False .. False;


--  Exceptions :
--  Station_Not_Valid : If the station is no longer in the ring.
Station_Not_Valid : exception;
--  Station_Not_Found : If the station isn't in the logical ring
Station_Not_Found : exception;
--  Invalid_Channel : If the channel is not available
Invalid_Channel : exception;
-- Info_Length_Overflow : If we try to send more than Max_Rt_Ep_MTU
Info_Length_Overflow : exception;
--  Creation_Error : If not being able creating the queues
Creation_Error : exception;
--  Unexpected_Error : If an unknown error has occurred
Unexpected_Error : exception;
--  Initialization_Error :If an error initializing the protocol.
Initialization_Error : exception;

----------------------
-- RT-EP Station_ID --
----------------------
-- The Station Identifier within the protocol.
type Station_Handle is new
  Interfaces.Unsigned_16 range 0..Number_Of_Stations;
for Station_Handle'Size use 16; --  16 bits for the Station_ID
subtype Station_ID is Station_Handle range 1..Number_Of_Stations;


Null_Station_Handle : constant Station_Handle:=0;

-- The station position in the ring
subtype Position is Integer range 1..Number_Of_Stations;

--------------------
-- RT-EP Priority --
--------------------
-- The priority of the messages
type Priority is new Interfaces.Unsigned_8
  range 1..Interfaces.Unsigned_8'Last;
for Priority'Size use 8;
-------------------
-- RT-EP Channel --
-------------------
-- Identifier of a reception channel
type Channel is new Interfaces.Unsigned_16 range 1 .. Number_Of_Channels;

-------------------------------
-- RT-EP Internal Server Id  --
```

```
   -------------------------------
   -- This is an internal server id for the purpose of DFSF_RTEP only
   -- Do not confuse with the regular FSF Server_Id
   type Internal_Server_Handle_T is range  0..2**16-1;
   for Internal_Server_Handle_T'Size use 16;

   subtype Internal_Server_Handle is Internal_Server_Handle_T
     range 0..Number_Of_Internal_Servers;
   subtype Internal_Server_Id is Internal_Server_Handle_T
     range 1..Number_Of_Internal_Servers;

   Null_Server_Handle : constant Internal_Server_Handle:=0;

end DFSF_RTEP;
```

With the exception of the initialization operation, the main user operations for RTEP are in the following package:

```
with Ada.Streams; use Ada.Streams;
with Ada.Real_Time; use Ada;
with DFSF.Shared_Info;

package DFSF_RTEP.Protocol  is

   ---------------------
   --  Get_Station_ID --
   ---------------------
   --  Will return the station identifier of the current station.
   --  On error will raise Station_Not_Found.

   function Get_Station_ID return Station_ID;


   ----------------------------
   --  Get_Station_ID_By_Name --
   ----------------------------
   --  Will return the station identifier of the station labeled
   --  by Station_Name in the ring configuration.
   --  On error will raise Station_Not_Found.

   function Get_Station_ID_By_Name
     (Station_Name : in String)
     return Station_ID;


   --------------------------------
   --  Get_Station_ID_By_Position --
   --------------------------------
   --  Will return the station identifier of the station placed in
   --  position Pos in the logical ring.

   function Get_Station_ID_By_Position
     (Pos : in Position)
     return Station_ID;


   ----------------
   --  Send_Info --
   ----------------

   --  Send_Info is used to send Data over the network.
   --  You have to provide the destination MAC addres (Destination_Address)
   --  the channel identifier of the reception task (Channel_ID)
   --  the priority of the packet.
   --  and the length of the data in bytes.

   --  Raise on Error:
   --  Station_Not_Valid
```

```
   --  Station_Not_Found
   --  Invalid_Channel
   --  Unexpected_Error
   --  Info_Length_Overflow
   --  Inexistent : If the Internal_Server_Id does not refere to a valid server
   ------------------------
   --  Generic Send Info --
   ------------------------

   generic
      type Data_Type is private;
   procedure Generic_Send_Info
     (Server_ID     : in Internal_Server_Id;
      Data          : in Data_Type;
      -- Data_Priority : in Priority;
      Timestamp     : in Real_Time.Time := Real_Time.Clock);


   ---------------
   -- Recv_Info --
   ---------------
   --  Recv_Info reads from the Chanel_ID the highest information Data
   --  and store it in Data. The length is stored in Size in bytes and
   --  the priority of the message is stored in Data_Priority.

   --  Raise on Error:
   --  Invalid_Channel
   --  Unexpected_Error

   ------------------------
   -- Generic Recv_Info --
   ------------------------

   generic
      type Data_Type is private;
   procedure Generic_Recv_Info
     (Source_Station_ID : out Station_ID;
      Channel_ID        : in Channel;
      Data              : out Data_Type;
      Data_Priority     : out Priority);


   ---------------------------
   -- Generic Try_Recv_Info --
   ---------------------------
   --  if No Elements in the queue, Received will set to false.
   generic
      type Data_Type is private;
   procedure Generic_Try_Recv_Info
     (Source_Station_ID : out Station_ID;
      Channel_ID        : in Channel;
      Data              : out Data_Type;
      Data_Priority     : out Priority;
      Received          : out Boolean);


   --------------
   -- Any_Info --
   --------------

   --  Any_Info checks if there is any data to be received in the specified
   --  Channel_ID. The caller process MUST assign its ID to
   --  the channel_id variable. Returns: True if threre is Data to be received.

   --  Raise on Error:
   --  Invalid_Channel
   --  Unexpected_Error

   function Any_Info
     (Channel_ID : in Channel)
```

```
   return Boolean;


-----------------------------------------------------------------------
--- RT-EP with Streams  -----------------------------------------------
-----------------------------------------------------------------------

-----------------
--  Recv_Info  --
-----------------
--  Recv_Info reads from the Chanel_ID the highest information Data
--  and store it in Data. The length is stored in Size in bytes and
--  the priority of the message is stored in Data_Priority.

--  The Recv_Info function updates the written offset to the size of the
--  receiving packet and the read offset to 0, so you can start reading
--  from the beginning.

--  Raise on Error:
--  Invalid_Channel
--  Unexpected_Error

procedure Recv_Info
  (Source_Station_ID : out Station_ID;
   Channel_ID : in Channel;
   Data : out Stream_Element_Array;
   Last : out Stream_Element_Offset;
   Data_Priority : out Priority);

--------------------
-- Try_Recv_Info  --
--------------------
--  if No Elements in the queue, Received will set to false.
procedure Try_Recv_Info
  (Source_Station_ID : out Station_ID;
   Channel_ID : in Channel;
   Data : out Stream_Element_Array;
   Last : out Stream_Element_Offset;
   Data_Priority : out Priority;
   Received      : out Boolean);

---------------
-- Send_Info --
---------------
--  Send_Info is used to send Data over the network.
--  You have to provide the destination MAC addres (Destination_Address)
--  the channel identifier of the reception task (Channel_ID)
--  the priority of the packet.and the length of the data in bytes.
--

--  Raise on Error:
--  Station_Not_Valid
--  Station_Not_Found
--  Invalid_Channel
--  Unexpected_Error
--  Info_Length_Overflow
--  Inexistent : If the Internal_Server_Id does not refere to a valid server
procedure Send_Info
  (Server_ID     : in Internal_Server_Id;
   Data          : in Stream_Element_Array;
   -- Data_Priority : in Priority;
   Timestamp     : in Real_Time.Time := Real_Time.Clock);


------------------------
-- Negotiate_Contract --
------------------------
--  Negotiate a new contract for the specified destination station,
--  channel id, and contract information. If accepted, create a new
--  server, and propagate the information to all the stations.
```

```
   --  Accepted indicates whether the renegotiation is accepted or
   --  not. If accepted, the id of the server is returned in Id It
   --  raises DFSF.No_Space if the maximim number of servers had been
   --  reached

   procedure Negotiate_Contract
     (Info                 : in DFSF.Shared_Info.Contract_Shared_Info;
      Id                   : out Internal_Server_Id;
      Accepted             : out Boolean);

   -----------------------------
   -- Bind_Contract_To_Server --
   -----------------------------
   --  It raises No_Space if there is no space for the new server
   procedure Bind_Contract_To_Server
     (Destination_Station  : in Station_ID;
      Channel_Id           : in Channel;
      Id                   : in Internal_Server_Id);

   --------------------------
   -- Renegotiate_Contract --
   --  ------------------------
   --  Renegotiate the contract associated with the server identified
   --  by Id, using the contract information in New_Info.
   --  Accepted indicates whether the renegotiation is accepted or
   --  not.  If accepted, the new information is propagated to all
   --  the stations and used for all subsequent message transmission
   --  through the server.
   --  It raises DFSF.Inexistent if Id does not reference a valid server

     procedure Renegotiate_Contract
        (New_Info : in DFSF.Shared_Info.Contract_Shared_Info;
         Id       : in Internal_Server_Id;
         Accepted : out Boolean);

   ---------------------
   -- Cancel_Contract --
   ---------------------
   --  Delete the server identified by Id, propagating the change to
   --  all the stations
   --  It raises DFSF.Inexistent if Id does not reference a valid server

     procedure Cancel_Contract
       (Id : Internal_Server_Id);

   ------------------
   -- Get_Contract --
   ------------------
   --  It raises Inexistent if the Id does not reference
   --  a valid contract in the table
   function Get_Contract
     (Id : Internal_Server_Id)
     return DFSF.Shared_Info.Contract_Shared_Info;


   -----------------------------
   -- Unbind_Contract_To_Server --
   -----------------------------
   procedure Unbind_Contract_From_Server
     (Id : in Internal_Server_Id);


--  private

--     Tx_Channel : constant Channel := 0;

end DFSF_RTEP.Protocol;
```

Internally, the following modifications are needed for implementation of sporadic servers in RTEP, to control budget consumption:

- The send queue is as before, but with the addition of a timestamp field.
- When a message is queued into a send queue, the associated timestamp value is stored with it in the queue.
- When the priority-arbitration token arrives at the node, the pending replenishments, if any, are executed. Then, if the current budget is larger than 1, the priority of the message to be sent is the message's priority; otherwise, it is the background priority (the minimum priority, by default).
- When the message is sent with its own priority, a budget of one unit is consumed.

5 Message partitioning

Message partitioning is done outside RT-EP, with a procedural interface that is executed by the user thread when sending and receiving messages. Therefore, it has no impact on the design and operation of the protocol. For efficiency, all the packets corresponding to the same message will have the same timestamp.

6 References

[1] M. Aldea and M. González. "MaRTE OS: An Ada Kernel for Real-Time Embedded Applications". Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe 2001, Leuven, Belgium, in Lecture Notes in Computer Science, LNCS 2043, May 2001.

[2] Michael González Harbour, José María Martínez, "FIRST Deliverable D-CS1.1-v1 (Teleoperated Robot Arm)", 8 May 2004

[3] Juan López Campos, J. Javier Gutiérrez, and Michael González Harbour. "The Chance for Ada to Support Distribution and Real-Time in Embedded Systems". Intl. Conference on Reliable Software Technologies, Ada-Europe-2004, Palma de Mallorca, Spain, June 2004.

[4] J.M. Martínez, M. González Harbour, and J.J. Gutiérrez. "RT-EP: A Fixed-Priority Real Time Communication Protocol over Standard Ethernet". International Conference on Reliable Software Technologies, Ada-Europe, York, UK, June 2005.