FRESCOR network contracts

In this section, we describe the specification of the FNA (FRESCOR Network Adaptation) layer, aimed at integrating network protocols in the FRESCOR framework, and the implementation of FRESCOR network contracts using the Real-Time Ethernet Protocol (RT-EP). [5]

1. Introduction

One of the main keys in FRESCOR is the concept of an integrated view of the different resources involved in a transaction, like the processor, network, memory or disk resources, that eases the deployment of complex distributed applications with a variety of real-time requirements, including hard real-time behaviour as well as soft requirements. For doing that, the framework is based on the notion of contracts that are negotiated between the application and the system. The most important FRESCOR modules concerning to networks are the Core, Spare capacity and Distributed modules:

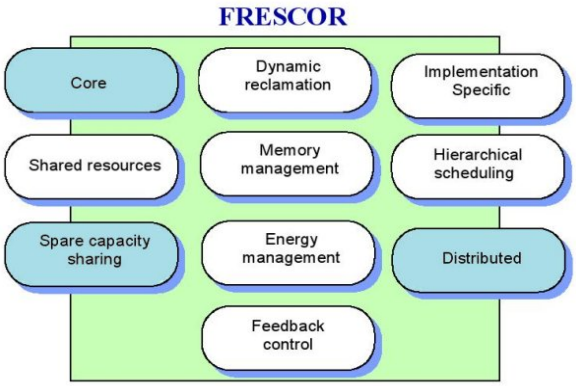


Fig 1.- FRESCOR modules and networks

1.1 Core services

Similar to the core FRESCOR module, the contracts on the network allow the application to specify its minimum utilization (bandwidth) requirements, so that the implementation can make guarantees or reservations for that minimum utilization. We reuse the contract structure that is used for processing nodes, where the main core attributes that apply for distribution are shown in the Fig. 2.

Name	Description
label	Global ID
resource type	Processor, network, memory, ...
resource id	An identifier of the resource
min budget	Min execution capacity per vres period
max period	Maximum virtual resource period
deadline	The deadline of the virtual resource

Fig 2.- Core attributes for networks

In addition to the core attributes, a new protocol-dependent attribute is created, that identifies extra information that might be required by a particular network protocol. In the case of RT-EP, although multicast packets could be treated slightly differently in the schedulability analysis, we finally decided not to specify any protocol-dependent attribute and treat them as normal messages.

1.2 Distributed model

Once the application negotiates a contract specifying that it is for a network resource, the FRESCOR implementation will create a network virtual resource for that contract, in a way similar to the creation of processor virtual resources for processor-related contracts. This network virtual resource is an implementation object that contains a copy of the relevant contract parameters, keeps track of the consumed resources, guarantee the allocation of the minimum resources, and prevent the application from using more resources than those declared in the contract.

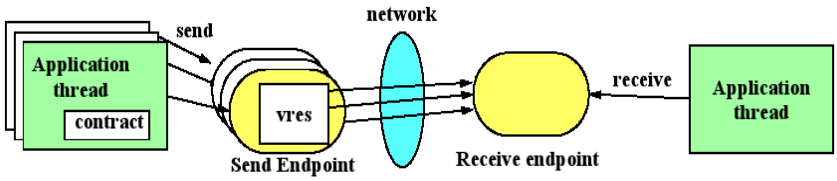


Fig 3.- Communication elements in FRESCOR

To keep track of consumed network resources and to enforce budget guarantees, it is necessary that the information is sent and received through specific FRESCOR services. Therefore, we need to create objects similar to the sockets that we call communication endpoints (see Fig 3).

A send endpoint contains information about the network to use, the destination node, and the network-wide message stream identifier that is used to establish the link between the send endpoints and the associated receive endpoints. The send endpoint is bound to a network virtual resource that specifies the scheduling parameters of the messages sent through that endpoint, keeps track of the resources consumed, and limits the bandwidth to the amount reserved for it by the system.

A receive endpoint contains information about the network and message stream id to use. It may get messages sent from different send endpoints, possibly located in different processing nodes.

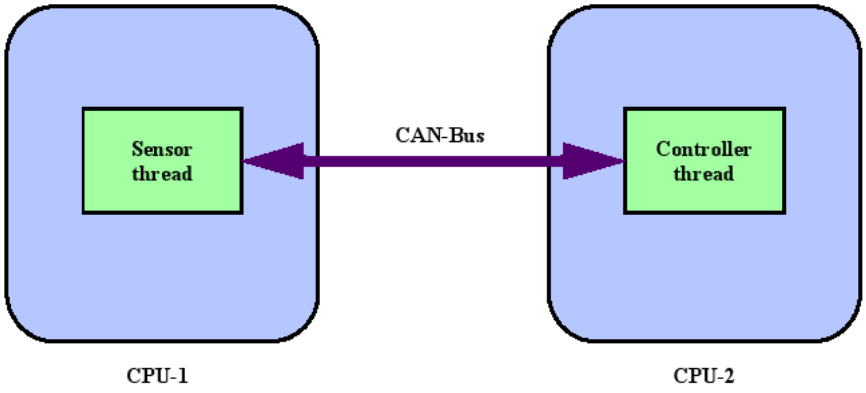


Fig 4.- Example of distributed transaction

The following example illustrates the use of the distributed FRSH module through a very simple distributed transaction in a system containing a task that is triggered by the periodic arrival of a message from a remote sensor. Figure 4 shows the architecture of this transaction. Figure 5 shows the pseudocode of each of the two tasks in the transaction.

Sensor thread

```
create CPU contract
negotiate the CPU contract
create Network contract
negotiate the Network contract
create send_endpoint
bind vres to send_endpoint

while (1) {
     read sensor
     send message
     frsh_schedule_timed_job
}
```

Controller thread

```
create contract
negotiate the contract
create receive endpoint

while (1) {
     read message
     process message
}
```

Fig 5.- Pseudocode of distributed transaction

1.3 Spare Capacity services

In the distributed FRESCOR we want to provide the same level of support for spare capacity sharing that is provided for processing nodes. This is a difficult task in the case of a distributed system, because the decisions made in one node may affect another one, requiring distributed consensus in the whole transaction. We do not want to embed all that complexity into the underlying scheduling framework implementation. Therefore, we have chosen to give a minimum support for spare capacity distribution inside the scheduling framework, and leave the consensus problem to some higher-level transaction manager that would make the negotiations for the application.

Name	***Description***
max budget	maximum usable budget
min period	minimum useful period
granularity	discrete or continuous
discrete gran set	pairs {budget,period} discrete gran
importance	priority to distribute extra cap.
weight	for vres with the same importance
stability time	minimum time vres must not change

Fig 6.- Spare capacity parameters

In the figure 6, the main parameters for spare capacity are described. First, the parameters that identify resources that could be used if there is enough capacity to provide better quality results. These parameters are distributed according to an importance and a weight. The idea is that we distribute the spare capacity starting from the most important level to the lowe

For this minimum support there is a stability attribute in the service contract, which has the implication that during the stability period the period or budget of the virtual resource can only change if a renegotiation is requested for it; it may not change automatically, for instance because of negotiations for other virtual resources. This provides a stable framework while performing the distributed negotiation.

The following example illustrates the use of the FRESCOR framework and the Spare Capacity services through a very simple distributed transaction and is modeled after a welding system that the University of Cantabria is developing. We have an FTT-SE [4] networked system with several nodes and a specific load and we want to negotiate a new transaction in which one of the nodes will

capture an array of points from a laser profiler representing the depth of the welding area and then send it to another node where a controller will activate an actuator, for moving the welding torch.

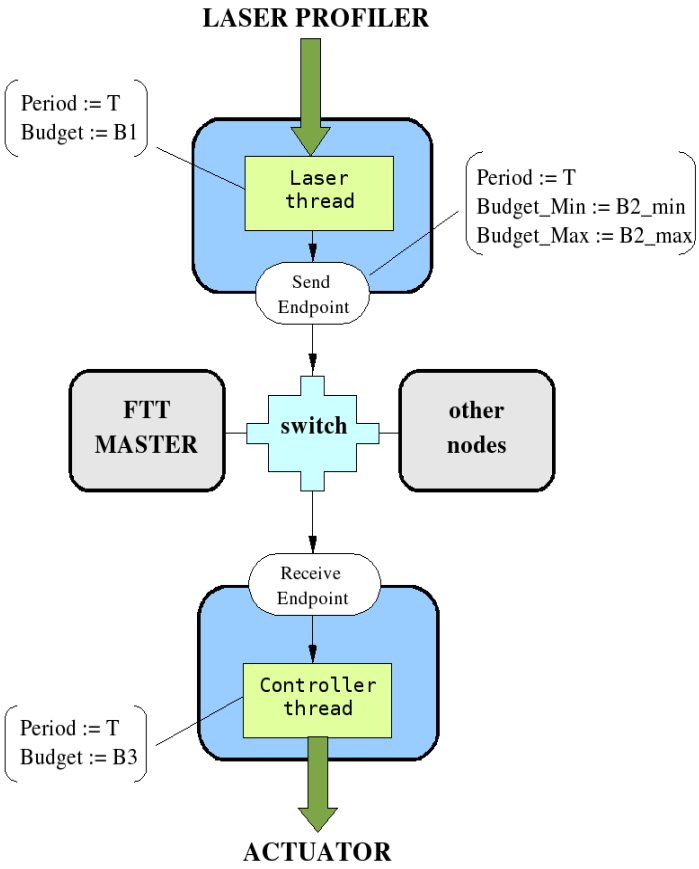


Fig 7.- FTT-SE distributed transaction with Spare Capacity

As shown in Fig. 7, three contracts must be negotiated:

- **Network contract**: this is the contract needed to send the array of points to the controller. We specify a range of budgets so that if the network has enough capacity we can send all the points, but if the network already has a big workload or, later, new contracts are negotiated, a lower capacity would be granted by the system, although always over the minimum required. This is the functionality provided by the Spare Capacity module. Note that for the receiving endpoint it is not needed to negotiate any contract.
- **Laser thread CPU contract**: this is the thread in charge of capturing, processing and sending the points obtained from the laser profiler. In each period it will read the current budget assigned to the network contract to see how many points can be sent. Depending on this information it will process and send more or less points.
- **Controller thread CPU contract**: this is the thread in charge of receiving the data from the network and actuating on the motors controlling the movement of the welding torch. Its budget must be prepared for processing the maximum size of the array.

On the other hand, CPU budgets are fixed and must cover the requirements for processing the whole set of points of the laser. When only a subset of the points is needed to be used in the transaction, threads will use less capacity and the FRESCOR dynamic reclamation module will give that capacity to other threads.

The pseudocode of the threads involved in the transaction would look like this:

Main threads

```
vres := negotiate cpu contract
create thread & bind to vres
```

Laser thread

```
n_vres := negotiate network contract
create send_endpoint & bind to n_vres

loop
     c := get budget (n_vres)
     read laser profiler
     process points (c)
     send points (c)
     frsh_timed_wait
end loop
```

Controller thread

```
create receive_endpoint

loop
     read points
     process points
     send command to actuator
end loop
```

In this simple example, where FRESCOR is used at a low level of abstraction, all the contracts involved in the transaction have a fixed period. If renegotiations in the transaction were necessary to switch to different periods, they should be made through the FRESCOR highlevel transaction manager which will provide the necessary distributed coordination for a global negotiation of the diverse resources involved in the transaction (including CPU, networks, memory or even bus accesses), and whose description is outside the scope of this deliverable.

1.4 RT-EP (Real-Time Ethernet Protocol)

RT-EP is a software based fault tolerant token-passing Ethernet protocol for multipoint communications in real-time applications, that does not require any modification to existing Ethernet hardware. This protocol allows the designer to model and analyze the real-time application using it, because it is based on fixed priorities and well-known schedulability analysis techniques can be applied.

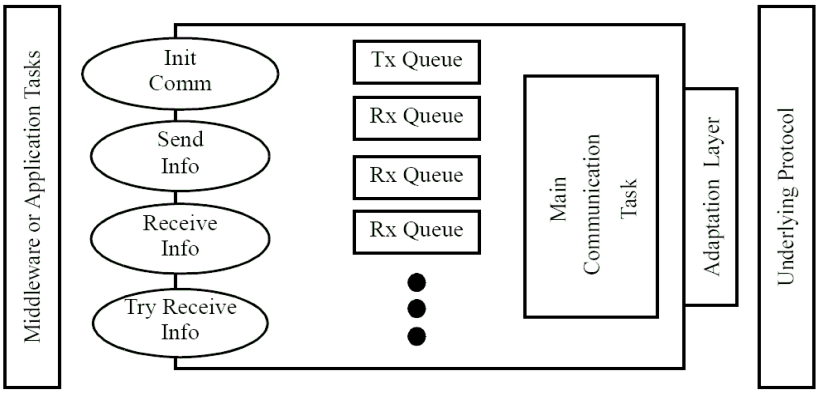


Fig 8.- RT-EP architecture

The Figure 8 shows the architecture of RT-EP. Each station (processing node or CPU) has a transmission queue, which is a priority queue where all the packets to be transmitted are stored in priority order. Packet information size is limited to 1492 bytes and fragmentation of messages is not allow at this layer. Each station also has a set of reception queues that are also priority queues. Packets with the same priority are stored in FIFO order. The number of reception queues can be configured depending on the number of application threads (or tasks) running in the system and requiring reception of messages.

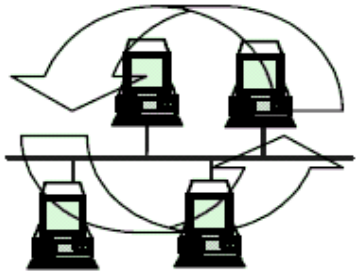


Fig 9.- RT-EP stations conform a logical ring

Each application thread should have its own reception queue attached. The application has to assign a number, the channel ID, to each application thread that requires communication through the protocol. The network is logically organized as a ring. Each station knows which other station is its predecessor and its successor, so the logical ring can be built. The protocol works by rotating a token in this logical ring to avoid collisions in the Ethernet media.

The token holds information about the station having the highest priority packet to be transmitted and its priority value. The network operates in two phases. The first phase corresponds to the priority arbitration, and the second phase to the transmission of an application message. During the priority-arbitration phase the token travels through the whole ring, visiting all the nodes. Each station checks the information in the token to determine if one of its own packets has a priority

higher than the priority carried by the token. In that case, it changes the highest priority station and associated priority in the token information; otherwise the token is left unchanged. Then, the token is sent to the successor station. This process is followed until the token arrives at the token_master station, finishing the arbitration phase. In the message-transmission phase the token_master station sends a message to the station with the highest priority message, which then sends the message. The receiving station becomes the new token_master station.

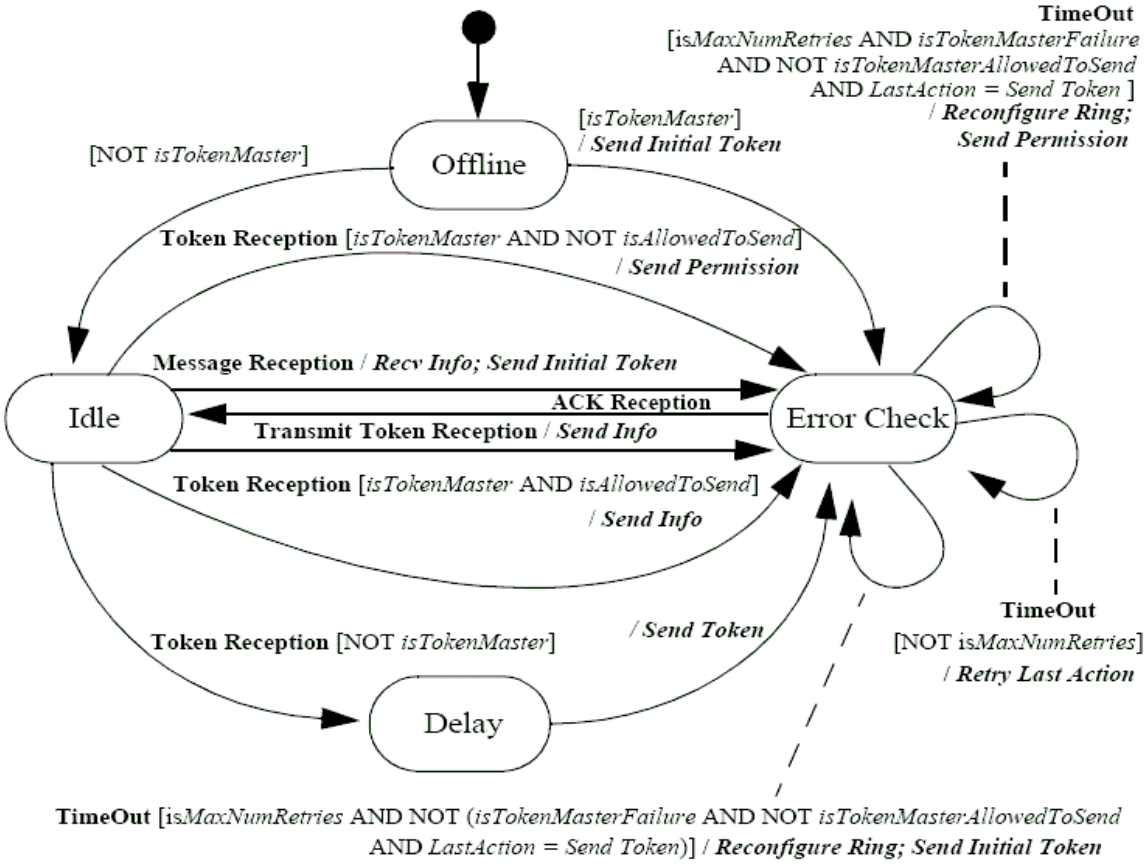


Fig 10.- RT-EP State machine

The recovery method is based on simultaneous listening to the media by all the stations. Each station, after sending a packet, listens to the media for an acknowledge, which is the transmission of the next frame by the receiving station. If no acknowledge is received after some specified timeout, the station assumes that the packet is lost and retransmits it. The station repeats this process until an acknowledge is received or a specified number or retransmissions is produced. In the latter case the receiving station is considered as a failing station and will be excluded from the logical ring. Because retransmission opens the door to duplicate packets if a station does not respond in time, a sequence number is used to discard duplicates at the receiving end.

1.5 DFSF (Distributed First Scheduling Framework)

DFSF is the previous proof-of-concept implementation of contract-based reservations on a network, and it is described in an annex of the deliverable D-OS.1v3 of the previous european project FIRST. For simplicity reasons it focused only on the core module, and made some restrictions. The scenario was a single ethernet network using a modified version of the RTEP protocol that we was called DFSF_RTEP.

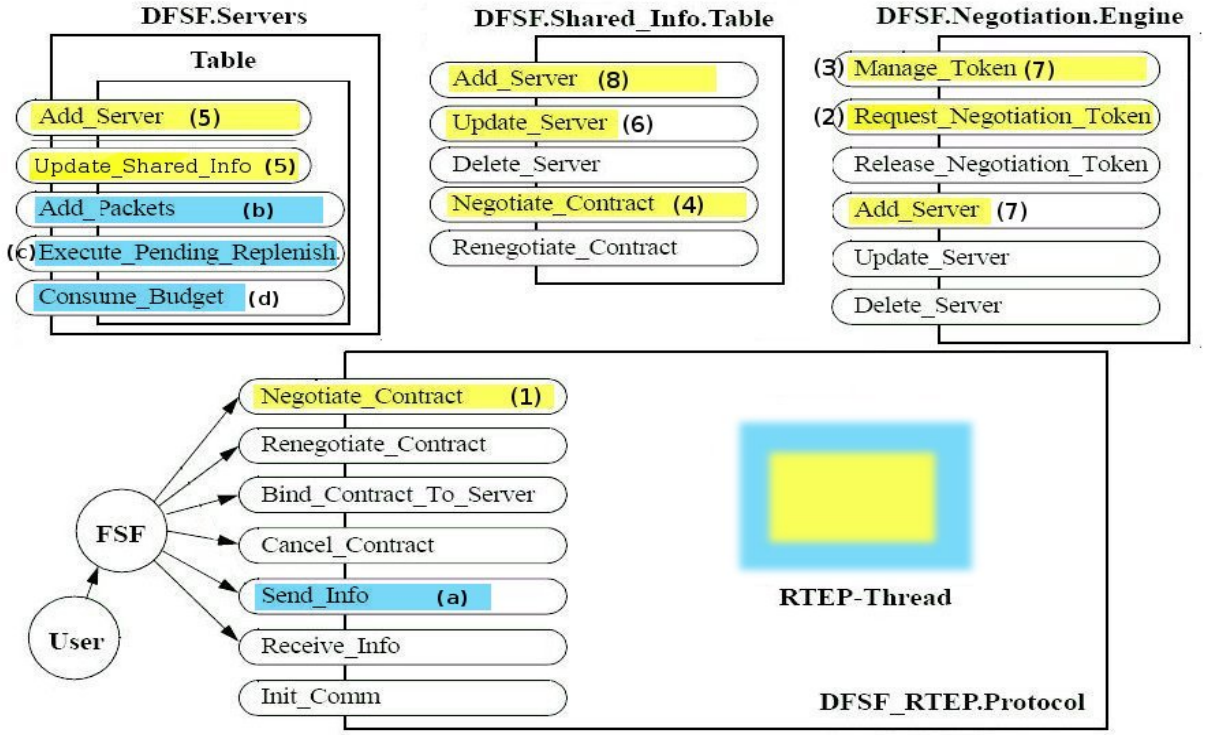


Fig 11.- DFSF achitecture

Figure 11 depictes the architecture of DFSF_RTEP and it contains the following modules:

• *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*: This package implements the interface offered to the user, which in this case is the FSF layer.

Two example sequences are shown in the figure 11:

- In the numbered sequence (in yellow) we see the flow of a negotiation. Negotiation.Engine implements the State Machine shown in Fig. 10, Shared_Info stores the network contracts of all the stations in the ring so we can make a utilization test, and the Servers module contains our server's information including the shared info and the track of the consumed resources and budget available.
- In the Lettered sequence, we see the flow for the delivery of a packet.

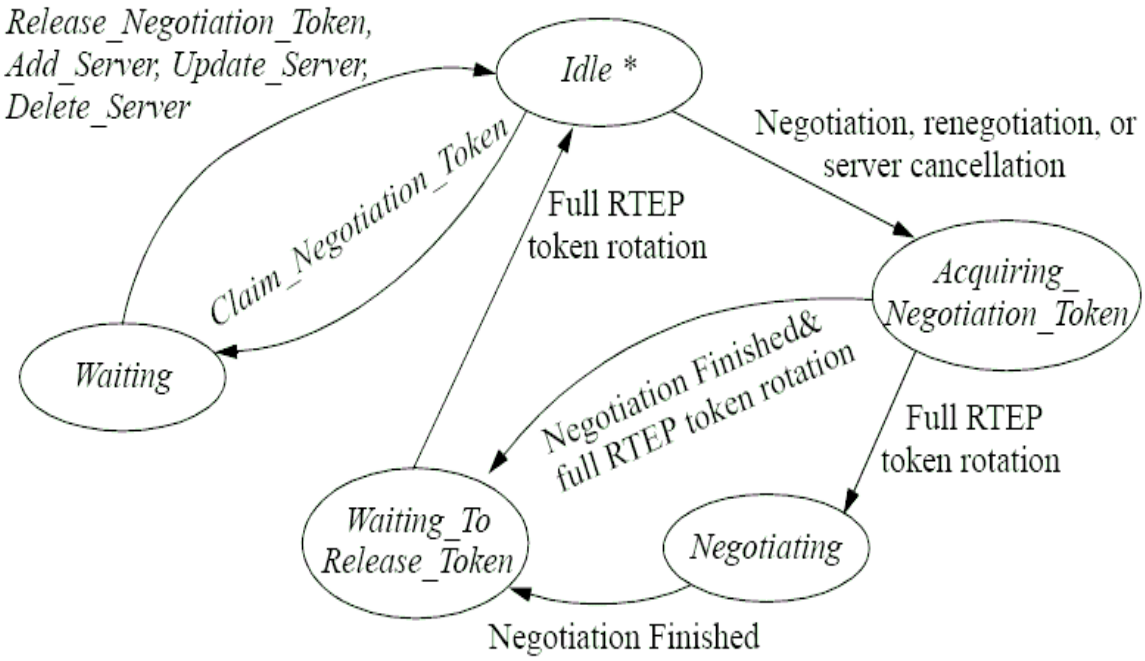


Fig 12.- DFSF state machine

In order to implement FSF with RTEP a new fixed field of 29 bytes had to be included in every packet of the protocol and the Main Task algorithm was changed to introduce the State Machine shown in figure 12. The node starting the negotiation successively switches from the Idle state to Acquiring_Negotiation_Token, Negotiating and Waiting_To_Release_Token. Meanwhile, the rest of the nodes, switch to the Waiting state, where they await the information with the negotiation result.

Although the previous implementation of DFSF was quite successful the architecture had the several problems:

- It was too specific to the protocol, because it was embedded in it, and severe difficulties in its maintenability and extendability.
- There was a 29 bytes overhead in each packet even when no negotiation was in process.
- DFSF was implemented in Ada. As the FRSH framework is supposed to be written in C, we need to offer a C interface.
- Also, negotiation times were not bounded in case of contention.

Therefore, a new architecture in which the negotiation process takes place only on top of the real-time protocol and negotiation times can be bounded is needed.

2. FNA (FRESCOR Network Adaptation) layer

The architecture of FRESCOR in a distributed system is divided into independent processor modules that handle negotiations in each processor, and independent network modules that handle negotiations for each network in the system.

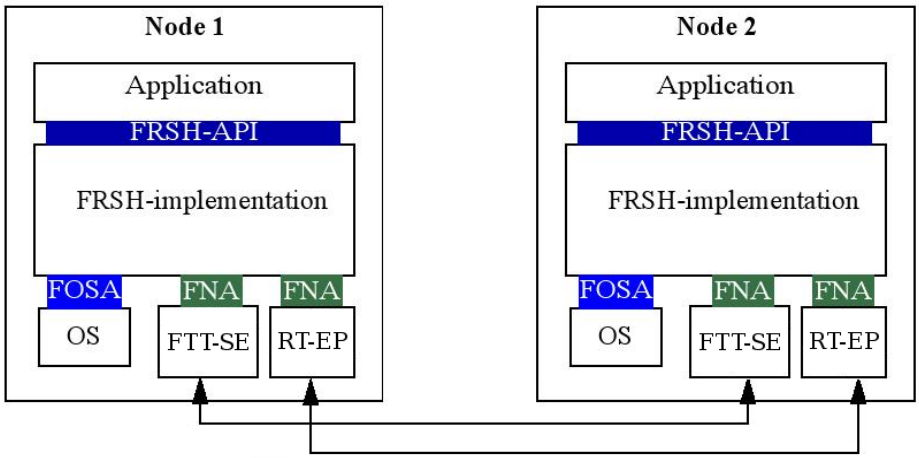


Fig 13.- FNA architecture

A uniform interface called FNA (FRSH Network Adaptation layer) has been designed to allow easily plugging in network modules for the same or different networks. Figure 13 shows a high-level view of this modular architecture, for an example of a system with two nodes and two different networks.

The full description of FNA is provided in the Annex A of this deliverable and its most important functionality is to handle negotiations of contracts, send-receive operations and mantain virtual network resources. FNA functions are called internally from the FRSH implementation. We need to register them as hooks and associate them to a specific value of resource id. This can be done with a static configuration file or by using a register function. In the figure 14, we can see the main functions from the FRSH API that concerns to distribution and the corresponding FNA hooks.

Similar to the FOSA operating system adaptation layer, there are some functions in the FNA module that are completely dependent on the protocol and therefore must be released as public functions from the application point of view. These set of functions are prefixed with 'frsh' instead of 'fna' to make clear that they are public, and are placed in a different header file.

In order to be integrated into the framework, each network must implement this FNA layer. To be able to negotiate contracts and handle virtual network resources, some services must be provided by the underlying protocol:

- A way to control and analize the network traffic
- A way to reliably spread the results of a negotiation
- A way to negotiate in mutual exclusion

How to provide these services to support FNA negotiations depends on the protocol. For example, in a centralized approach using the master-slave FTT-SE protocol [4] the control and analisis of the traffic is done by the master using a table and mutual exclusion can be provided through ordinary

mutexes. As the master is the only node performing negotiations, the results of the negotiation could be mantained in its own memory, but for performance reasons a copy should be propagated to the slaves.

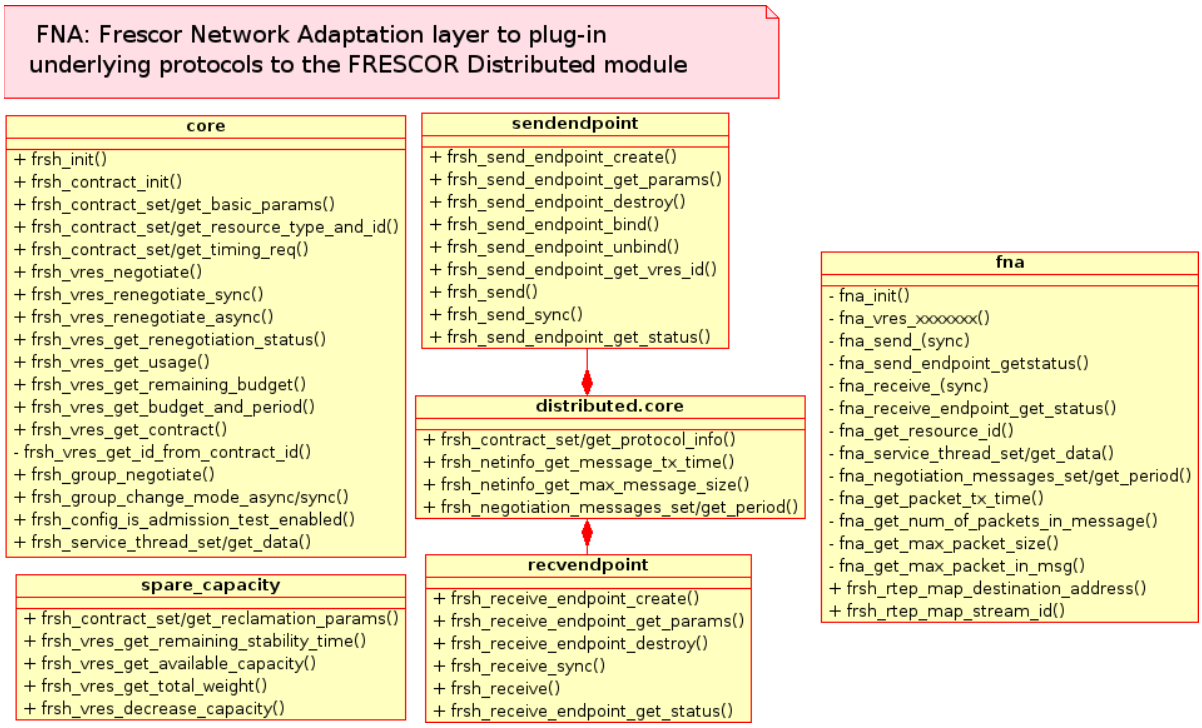


Fig 14.- FNA hooks

In the case of RT-EP we propose a decentralized approach, where FNA implementation will be a distributed module that has parts in all the processors connected through the network. Compared to the DFSF approach we will use a multi-layered architecture in order to facilitate maintainability. RT-EP will be enhanced to support the previous requirements, as we will see in the next section, by adding network scheduling servers, a reliable multicast mechanism and distributed mutexes to the protocol. After that we will see how to build a new layer on top of those requirements to support the FNA negotiation services.

3. RT-EP Enhancements

In order to clarify ideas, the RT-EP implementation has been cleaned up, and restructured. Figure 15 depictes the new directory structure for the RT-EP code:

- RTEP: this is the root directory, where most of the functionality happens
- POSIX: the RT-EP implementation uses some posix functions (signals, timers and mutexes). We have created this directory to store the Ada files that import these functions from an OS that is written in C.
- LIB: here we store several data types used widely in the RT-EP implementation. We have written them as generics so they can be reusable.
- RTEP_FNA: this is the place for the new FNA layer will be stored.
- TESTS: a directory with a battery of tests and scripts to try them on a free x86 emulator
- DOC: a directory with a lot of new documentation about the protocol

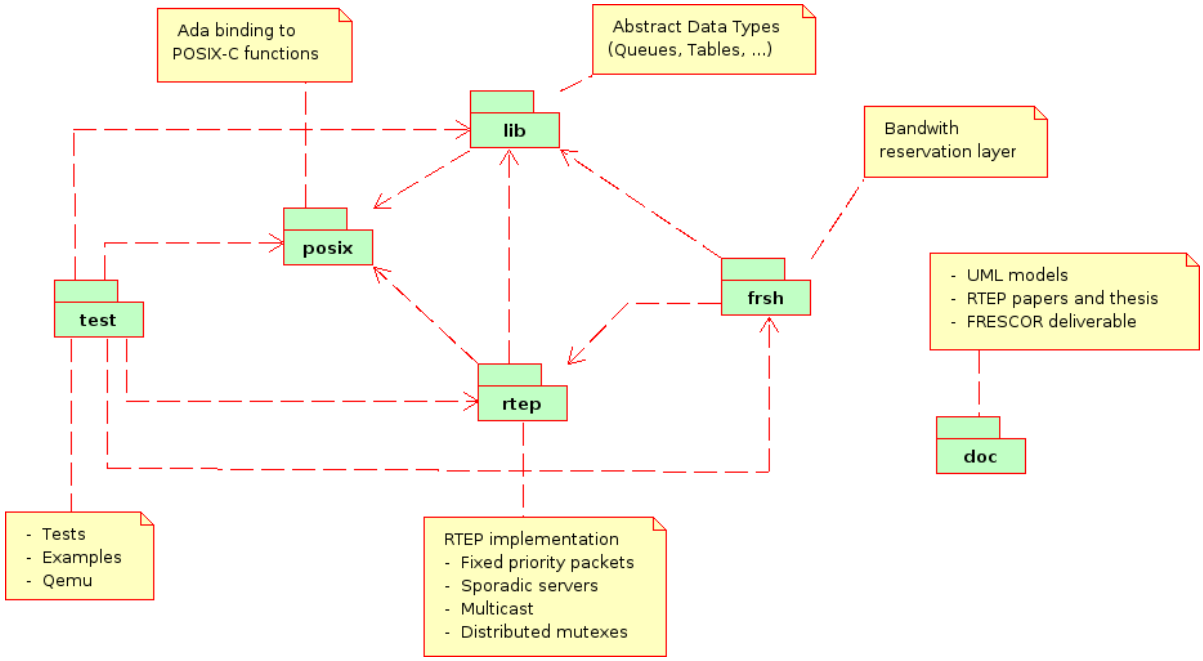


Fig 15.- RT-EP directories

As we see, there are no circular dependencies among the different packages. For example, the RTEP_FNA layer depends on the RT-EP services and the generic data types, but RT-EP does not depend on RTEP_FNA.

3.1 Sporadic servers

Server-based scheduling techniques have been used for a long time, typically associated with processor time scheduling, to limit the bandwidth assigned to a particular computation or set of computations while also guaranteeing some minimum level of service. Servers such as the periodic server [6], the sporadic server [7], or the constant bandwidth server are examples of such scheduling policies. The concept of server is also applicable to other resources and in particular to networks. For example, the leaky bucket concept used in network trac shaping [8] is similar to the sporadic

server; EDF-based servers [9] have been successfully applied to the CAN bus.
In RT-EP, packets are scheduled using fixed-priorities so a natural choice to implement servers for RT-EP is to base them on the sporadic server policy [7]. In a processor, the sporadic server policy works by reserving a given amount of execution-time capacity (called budget) for executing a process or a set of processes. In the beginning, the budget is set to a value called the initial capacity. When one of these processes executes it consumes budget. When it finishes executing, the consumed budget is stored in a queue for it to be replenished at some interval after the process started executing. This interval is called the replenishment period. When the budget is exhausted, the processes under the control of the sporadic server are not allowed to run, or at run at a background priority level below the priorities of any other real-time processes. When a server is used to schedule a network the concept of execution time must be turned into transmission time. In most networks, information is sent in units called packets which are usually non preemptible and therefore constitute the minimum eective budget. Therefore, in a network the most natural unit for measuring budget is a number of packets. In RT-EP the server's budget will be consumed one packet at a time. The non preemptability property of a single packet must be taken into account as a bounded blocking effect on higher priority servers.

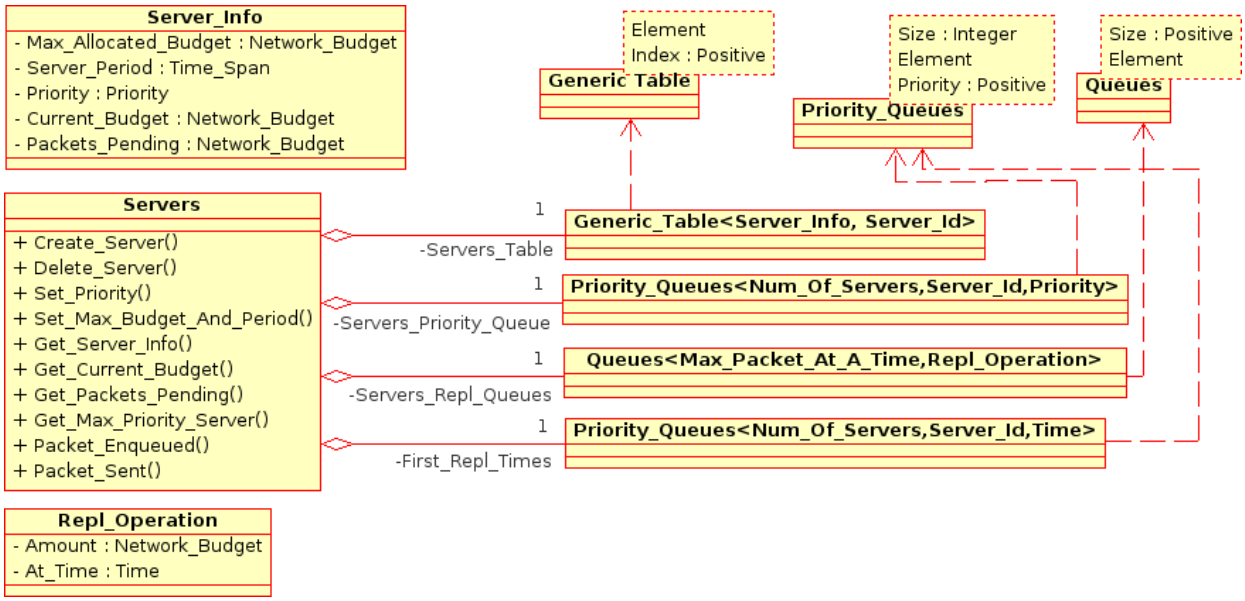


Fig 16.- RT-EP Sporadic Servers

Fig 16 shows the properties and architecture of the implementation of servers in RT-EP. Each server is assigned an initial transmission capacity (measured in network packets), a replenishment period (which is a time interval) and a priority. Internally, servers keep track of the current available budget and the count of packets pending to be sent. All this information is kept in a table at the sending node. The extension that we have made to RT-EP allows messages to be sent through a sporadic server, or with a plain fixed priority. In this way, the introduction of servers in RT-EP is still compatible with the use of fixed priority packets. For the plain fixed priority messages, there is a priority queue, called the FP transmission queue, that stores the packets pending to be sent. Packets sent through a sporadic server are stored in a FIFO queue associated with each server, called the server transmission queue. This architecture allows engineers to design several priority bands by using fixed priorities and servers at the same time. It also eases the maintenability of the protocol by eliminating the need of branching the RTEP implementation in two different versions.

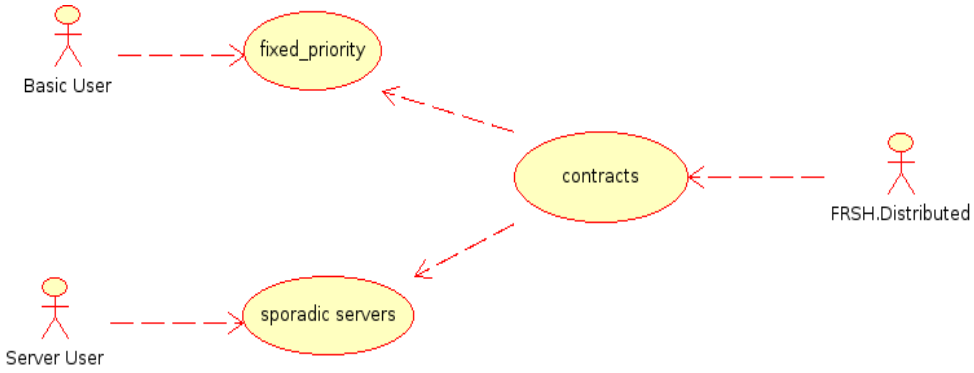


Fig 17.- RT-EP Cases of use

When several servers in the same node have packets to be sent, they compete among themselves using a priority queue, called the servers priority queue, that contains pairs of server id and current priority. In this queue, if the current server budget is zero the priority of the server is set to a background level, and if it is larger than zero the server's priority level is used. When the RT-EP main communication task needs to find out which is the packet with the highest priority, it executes the possible pending replenishment operations and then it checks both priority queues, the servers priority queue and the FP transmission queue, to find out which is the pending packet with the highest priority. If the packet is from a server, it finds it in the corresponding server transmission queue.

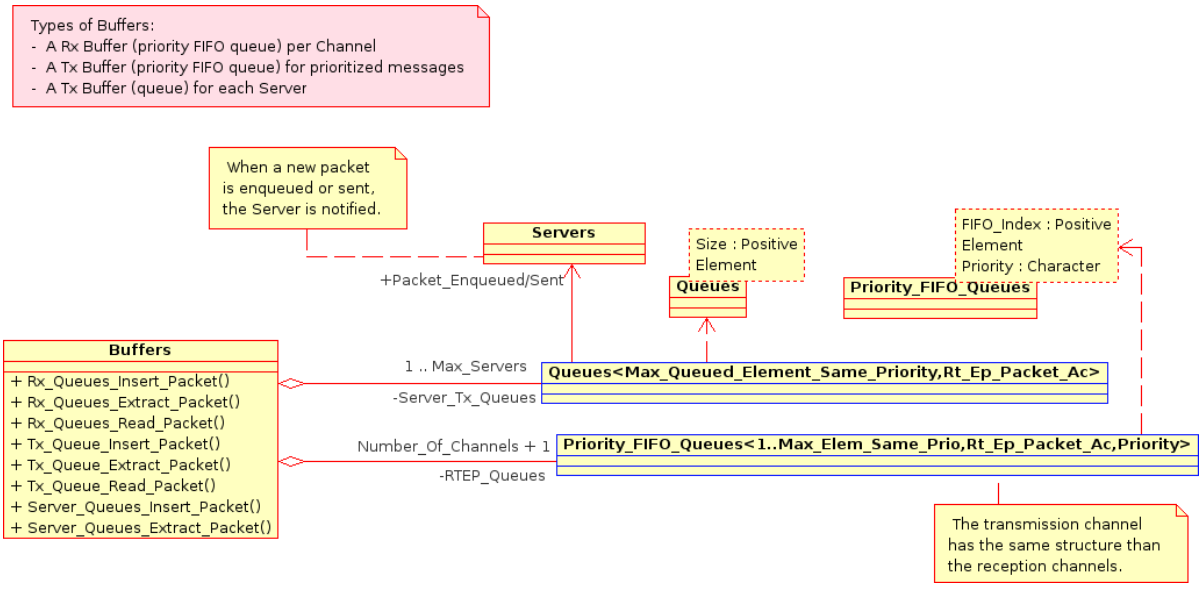


Fig 18.- RT-EP Buffers architecture

Figures 18 and 19 shows the queues involved in the integration of the sporadic servers inside RT-EP. For each packet sent from a server the associated budget is decreased by one unit and a replenishment operation is enqueued to occur one replenishment period later. The replenishment operations are stored inside a time-ordered priority queue, for performance reasons.

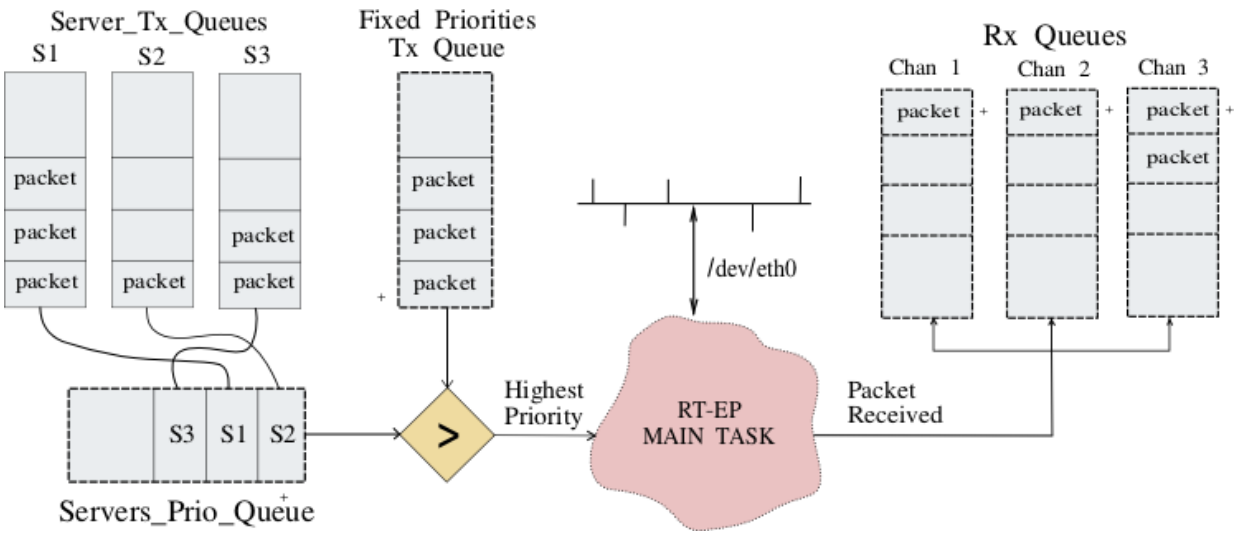


Fig 19.- RT-EP Buffers and the Main Task

When the current budget of a server is zero and a replenishment operation takes place, the priority of the server is raised to the normal priority level in the servers priority queue. In the receiving end, RT-EP stores the received messages in reception queues each associated with an application-specied channel number, to implement logical channels for the messages. For each channel, the messages are stored in a priority queue, so that they can be retrieved in priority order.

3.2 Reliable Multicast

A reliable multicast service must ensure that packets are delivered to receivers from the multicast group in a bounded amount of time, free of errors and in the order they were sent by the source. The most typical approach to reliable multicast is the sender-initiated approach, where the sender maintains the state of all the receivers from whom it has to receive acknowledgments (acks). But this technique has a scalability drawback, commonly known as the acknowledgement implosion problem [10]. A receiver-initiated approach, where receivers request the sender the retransmission of packets that are missing could be more scalable but it requires infinite buffers to prevent deadlocks [10].
In [10] two other solutions to the acknowledgment implosion problem that operate correctly with finite buffers are described. Tree-based protocols organize the receivers in a tree and send acks along the tree. And ring-based protocols where the basic premise is to have only one token site responsible for acknowledging packets back to the source. These solutions apply to generic protocols rather than to specific implementations so a direct implementation on RT-EP would be quite inefficient compared to the approach presented in this paper.

In RT-EP the acknowledgment implosion effect of the sender-initiated approach can be reduced by using the RT-EP token mechanism. A multicast message can be sent like the rest of messages but with a multicast address and then the next token round can be started. While finding out the priority of a new message to be sent, receivers could use a negative acknowledgment (NACK) mechanism to indicate that the previous multicast message received was incorrect. Although in the average case the performance of this multicast approach looks good, in the worst case, when there is an error in the transmission, the worst end-to-end transmission time of messages would be greater than for

normal messages because we would have to wait another token round. Also, new functionality should be added to receivers so that they could discard multicast retransmissions that were already received. These requirements would make the protocol more complex.

8	*6*	*6*	*2*	*2*	*1*	*1*	*2*	*2*	*2*	*2*	*1*	*33*	*4*
Pre	Ether Hdr			RTEP Token Hdr								Spare	FCS

Fig 20.- Ethernet Frame with a Token and Spare bytes

The approach presented in this paper takes the same premise as in the ring-based protocols proposed in [10] where there is only one token site responsible for acknowledging packets back to the source. But we go one step further by distributing the responsibility of retransmissions among all the nodes in the ring. The key of our approach consists on sending multicast information in the spare bytes of the token packets which, due to Ethernet minimum frame size, see Fig. 20, have some spare transmission capacity. In addition, we can take advantage of the built-in fault handling mechanisms present in the protocol to ensure automatic retransmission of faulty packets.

1	*2*	*2*	*1*	*1*	*26*
MType	MAddr	Chan	Prio	Length	Info

Fig 21.- Multicast Frame

Fig. 21 shows the fields of a multicast frame:

- **MType**: Multicast Type selects the multicast operation (an ordinary multicast message, a mutex lock or a mutex unlock operation)
- **MAddr**: Multicast Address (a static table, similar to the ring conguration table, states which nodes belong to a multicast group)
- **Chan**: the destination channel
- **Prio**: the priority of the packet
- **Length**: the length of Info (in bytes)
- **Info**: the information itself
- Also, the source address is implicit in the Token Master field of the RT-EP header

One of the main drawbacks of this approach is that the room for information in each token is small, 26 bytes (although congurable). If greater capacity for multicast were needed the previously mentioned approach with explicit NACKs would be more efficient.

Summing up the multicast procedure, when a node wants to send a multicast packet it enqueues the message as a normal message with the desired priority, but specifying as the destination address a multicast address. Then it competes in the arbitration phase of the protocol like a normal packet. When it is granted with the right to transmit, instead of sending a normal message, a new arbitration phase is initiated and the multicast message is sent through the token. Thanks to the reliability of the token transmission, which used implicit acknowledgements and a timeout-based fault handling mechanism, the protocol ensures that the multicast message is received by all the nodes and free of errors, up to the same reliability level as the normal packets have.

3.3 Distributed Mutexes

There are several algorithms in the literature [11] [12] describing ways of providing mutual exclusion in a distributed system without shared memory. Most of these algorithms can be divided in two groups: token-based and permission-based algorithms.

In [3], a token-based algorithm for RT-EP was implemented taking advantage of the protocol's token. Some extra fields were introduced in the header of every packet of the protocol to inform about the status of a single distributed mutex. In this approach, when a node receives a packet and wants to lock the mutex, it checks the mutex status field. If the mutex is free, it is locked and a notification is circulated in the ring. The unlock operation is similar.

One of the nice points of this architecture is its ability to notify other nodes about status changes in the mutex. For example, when negotiating, nodes that are notified of a mutex lock operation can execute a change of state to wait for the results of the negotiation. Despite this interesting capacity this architecture has not been reused because it also has some drawbacks:

- It lacks scalability and maintenability because if we want to have another mutex we have to modify the protocol and add new mutex fields to the token header
- In case of contention, when several nodes want to lock the mutex, the locking time is not bounded so higher level synchronization messages would be needed.

1	*1*	*1*	*2*
MType	Mutex	Locked	Holder

Fig 22.- Distributed Mutex Frame

In our new architecture a permission-based algorithm is proposed. A mutex operation is, like multicast messages, sent through the spare bytes of the RT-EP token with the format shown in the Fig. 22:

- **MType**: a mutex lock or a mutex unlock operation
- **Mutex**: the mutex id
- **Locked**: is the mutex locked?
- **Holder**: if locked, the current holder
- Also, the multicast address is implicit in the mutex id

This multicast frame visits each node during an arbitration phase, to find out if the mutex is free or locked. Nodes check if they own the mutex or not, and recirculate the token in the ring. When the token returns to the sender, variable Locked indicates if some station owns the mutex or not. If the mutex is not locked the node where the multicast message was generated becomes the owner. To unlock the mutex a new multicast message is sent to each node in the ring. With this behavior it is easy to implement the operations provided by the network protocol to the application, which allow the calling thread to block until the mutex is locked, and to unlock the mutex.

This approach solves the problem of scalability because a high number of mutexes can be used. It also helps in bounding the mutex locking time because locking operations can be priorized. In the

case in which two nodes compete for the mutex with the same priority, the order is not defined. This has been addressed in several distributed mutual exclusion algorithms to solve fairness, typically by using logical clocks to timestamp the requests. In our case, RT-EP sequence numbers could be used instead. In any case, for hard real-time analysis only the worst case scenario is taken into account so this timestamping has not been implemented in the protocol. As with the reliable multicast, fault-handling is also provided automatically by the protocol's token transmission procedures. In the event that a node holding a mutex fails, the RT-EP built-in fault handling mechanism will notify the other nodes about its exclusion from the ring and tasks waiting for that mutex will be awaken.

3.4 Other enhancements

Other minor enhancements include the improvement of the protocol's initialization, the creation of a new interface in C and optimizations in buffer handling by changing copies of data to copies of pointers where possible. Regarding the former enhancement, there was a case in which the protocol could go into a deadlock at ininitialization time. At initialization time, the first token Master sends requests to the rest of the nodes who must answer with an ACK packet and go to the Idle state. If those ACK packets got lost, the protocol entered to a deadlock. Now, it has been added the possibility of acknowledging this ACK packets even from the Idle State.

4. RT-EP FNA

Once we have an enhanced RT-EP that supports the requirements shown in section 2, all we need to do is to use them correctly to implement the RT-EP FNA layer. The most important functions are the ones regarding to distributed negotiations, renegotiations (synchronous or asynchronous) and cancel contracts. We base our architecture in two tasks: the *negotiation task* and the *receive negotiation results task*.

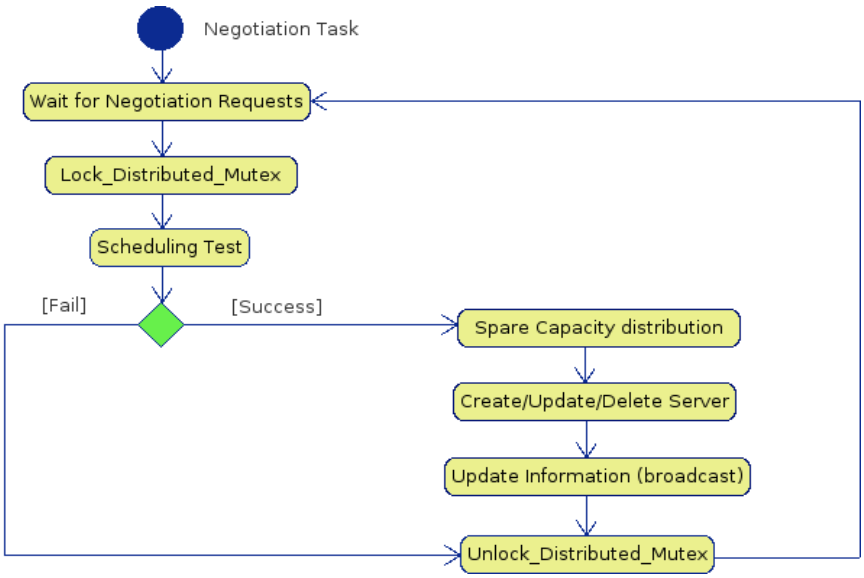


Fig 23.- Negotiation task

The *Negotiation Task* is in charge of serving requests from the application. Figure 23 depictes the behavior of this task through an activity diagram. The task consists on a loop where it waits for the application to send a request (for instance, to negotiate, renegotiate or cancel a certain contract). When a request arrives the task locks the distributed mutex to avoid inconsistences and executes a

scheduling test to check if the contract can be accepted or not. If the contract is accepted it starts the spare capacity distribution (this part has not been implemented yet. We plan to integrate an algorithm that is being developed by the University of York). After that we have to spread out the results of the negotiation. This includes our contract information and possibly other contracts that have been affected by the new spare capacity distribution. When distributing spare capacity we have to take into account the Stability Time parameter. If this time has not expired we can not change its virtual resource associated spare capacity. In order to avoid using global clocks we add the worst case time of spreading the results to the parameter. After all the nodes have received their results the negotiation mutex is unlocked so other nodes can perform negotiations.

Fig 24.- Negotiation task

The *receive negotiation results task* is a very simple task in charge of waiting negotiation results coming from other nodes. Its priority must be higher than the negotiation task in order to avoid possible inconsistencies (for example, because the results are still in the RT-EP buffer).

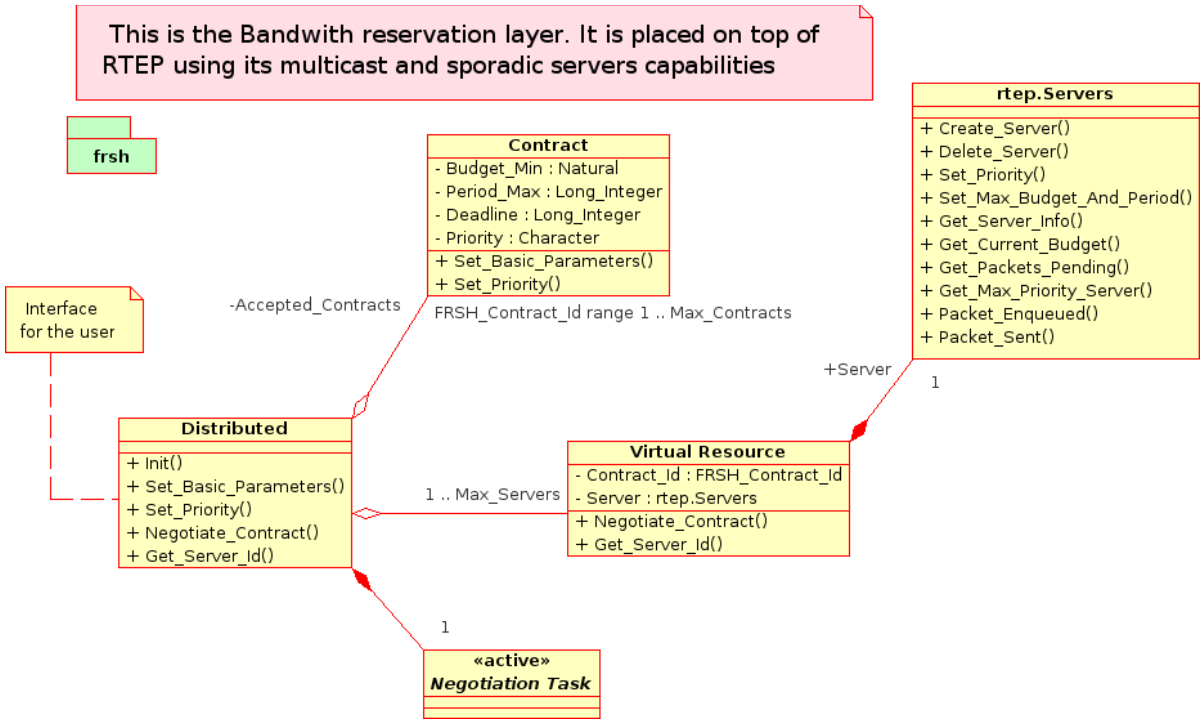


Fig 25.- RT-EP FNA Structures

Other functionality that RT-EP FNA must implement is described in the figure 25. One is to keep

the accepted_contracts table where the accepted contracts of all nodes is stored. Finally we need to keep a table with the virtual resources created as a result of local negotiations. A virtual resource in RT-EP FNA is composed of a copy of the contract parameters and the RT-EP Sporadic Server in charge of enforcing its values.

5. Testing, debugging and measurements

A lot of effort has been done in order to be able to look into the protocol as deep as possible in order to find possible bugs. A new module has been added to the protocol with several debugging functions. Some of the functions are aimed at printing warning, errors or debugging information depending on the value configurable booleans. Thank to these booleans we can ask the protocol to show only debugging information regarding to distributed mutexes, for example, and so on.

```
procedure DEBUG (Str     : in String;  Enabled : in Boolean);
```

Other functions are aimed at activating different types of time measurements throughout the protocol. A very simple interface is provided to the protocol:

```
procedure BEGIN_TIME_MEASURE (Id : in Time_Measure.Measure_ID);
procedure END_TIME_MEASURE (Id : in Time_Measure.Measure_ID);
```

Internally, there is a task (it can be disabled) in charge of collecting all these measures, calculate stadistics of worst, best and average case and send them to a Linux node connected to the network as a passive receiver. We provide also a program for Linux, that waits in a socket and receives this information that is then friendly printed in a file so the user can read it. For example, the following is extracted from one of this measurement reports:

```
Operation(
        Type                        => Simple,
        Name                        => Recv_Info_master,
        Worst_Case_Execution_Time => 1.5562e-05,
        Best_Case_Execution_Time  => 1.1988e-05,
        Avg_Case_Execution_Time   => 1.2417e-05,
        Shared_Resources_List       => (RT-EP_Core_master));
Operation(
        Type                        => Simple,
        Name                        => Token_Passing_Time_master,
        Worst_Case_Execution_Time => 0.000417039,
        Best_Case_Execution_Time  => 0.000405476,
        Avg_Case_Execution_Time   => 0.000409804,
        Shared_Resources_List       => (RT-EP_Core_master));
Operation(
        Type                        => Simple,
        Name                        => Go_Return_Time_master,
        Worst_Case_Execution_Time => 0.00118243,
        Best_Case_Execution_Time  => 0.000778442,
        Avg_Case_Execution_Time   => 0.00086987,
        Shared_Resources_List       => (Test_Server_Msgs_master));
```

We have used this library to measure the new RT-EP services timings and the their performance metrics, see figure 26, show that the transmission times are very ecient as compared to the regular message transmission times in the RT-EP protocol. The measurement platform is a ring configured with two AMD Duron 800Mhz stations connected through a 100 Mbps Ethernet switch. Since we lack a global time basis for measuring the amount of time to send a packet we measure the time required to send a packet, execute a handler in the other node and receive an answer.

Operation	Worst	Avg	Best
Send Server	1.2	0.9	0.8
Send Multicast	2	1.8	1.6
Lock a mutex	1	0.8	0.5
Unlock a mutex	0.5	0.4	0.2

Note: all measures are in milliseconds

Fig 26.- RT-EP service measurements

In the case of RT-EP FNA services the worst execution times are also expressed in terms of few milliseconds (2 to 5 ms in our tests).

Another effort in order to debug the protocol has been the development of a plug-in for a powerful Ethernet sniffer tool called Wireshark (formerly known as Ethereal) http://www.wireshark.org/

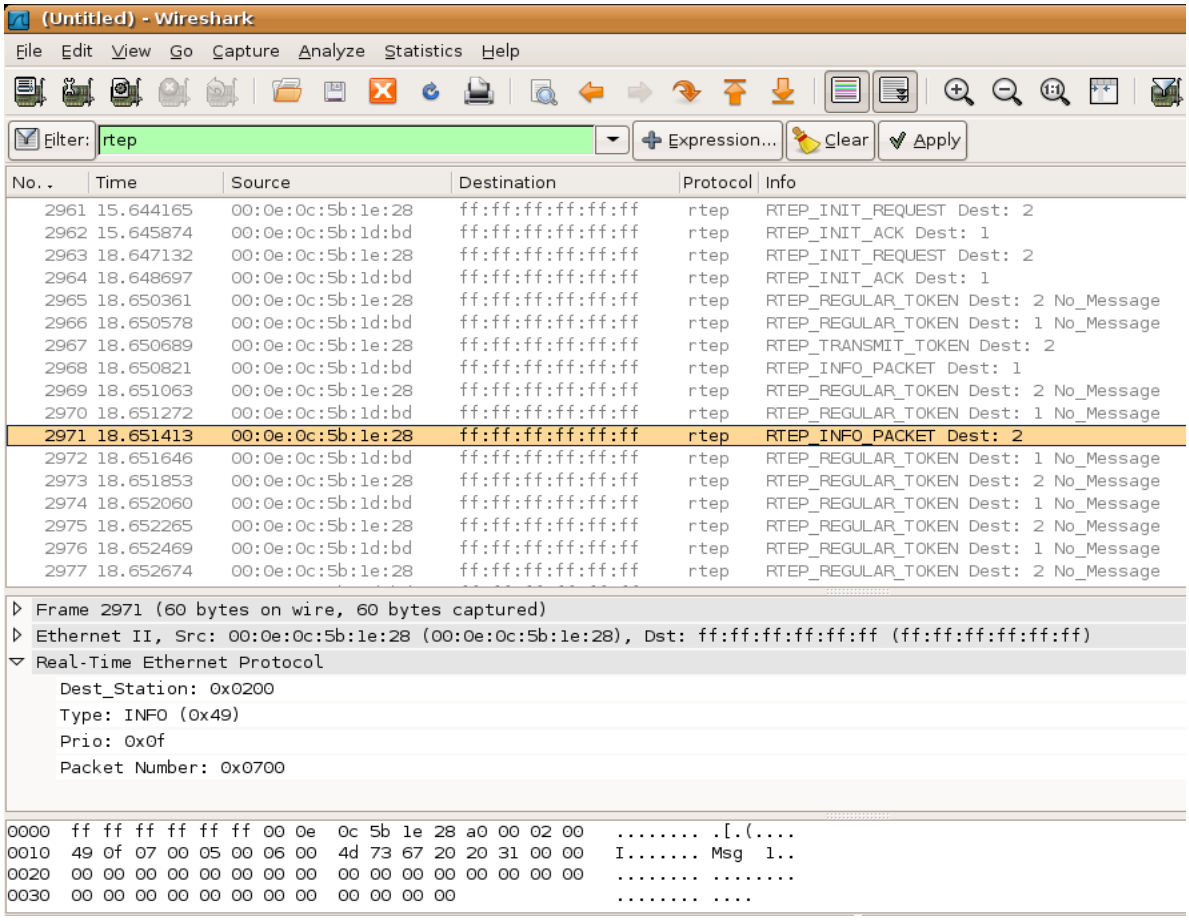


Fig 27.- Wireshark plug-in for RT-EP

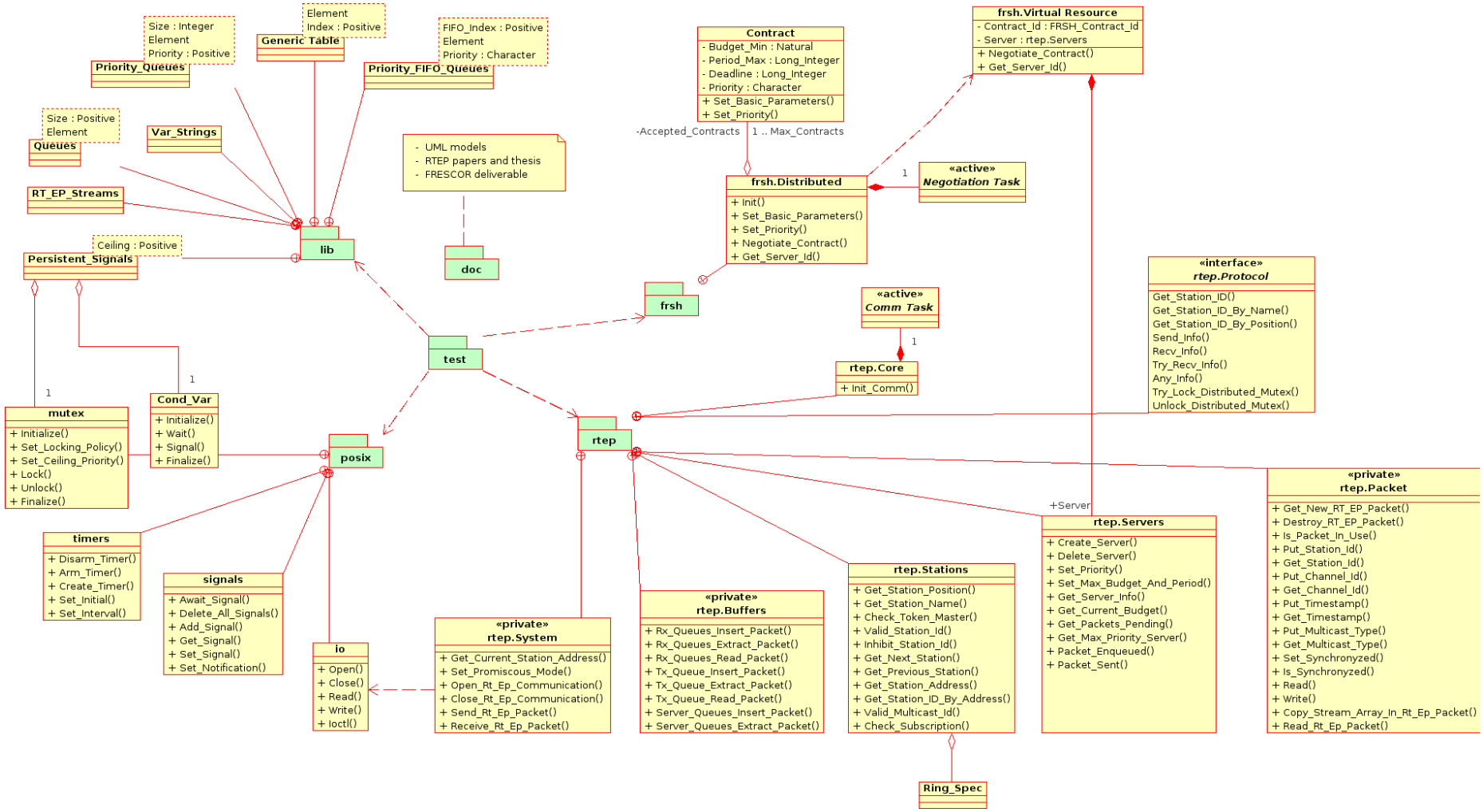
In the figure 28, we can see a sequence of the RT-EP protocol. First we see the initalization packets and how the new improvement to the initialization proccess (see 3.4) allows the protocol to overcome an ACK lost. After the protocol is initializated we see the arbitration phase, a permission token, a transmission message and then an answer. Each frame can be analyzed field by field in the window bellow, or byte by byte in the bottom window.

6. Conclusions

The fixed-priority real-time ethernet protocol called RT-EP has been extended with the addition of three new services: a server-based scheduling policy, reliable multicast and distributed mutexes. These services operate together with the previous fixed-priority message transmission that was already available in RT-EP. The new services have been implemented and tested, and their performance metrics show that the transmission times are very efficcient as compared to the regular message transmission times in the RT-EP protocol.

Over these services we have implemented a new layer, called RT-EP FNA, that allow us to support bandwidth reservations and integrate it into the FRESCOR distributed contract-based scheduling framework that is under development (when a FRSH implementation is available we will just need to configure FNA hooks so they are called when the user wants to negotiate in our network). This layer in charge of negotiating contracts and managing virtual network resources that keep track of the network resources consumed, and provide the necessary quality of service guarantees for supporting both hard and soft real-time requirements. The new RT-EP protocol will be one of the implementations of the FNA layer in FRESCOR.

In the following page we can see a BIG Picture of the protocol and the different packages involved in the development of the RT-EP FNA layer.


Size : Integer
Element
Priority : Positive
Priority_Queues
Element
Index : Positive
Generic Table
FIFO_Index : Positive
Element
Priority : Character
Priority_FIFO_Queues
Size : Positive
Element
Queues
Var_Strings
RT_EP_Streams
Persistent_Signals
Ceiling : Positive
lib
- UML models
- RTEP papers and thesis
- FRESCOR deliverable
doc
Contract
- Budget_Min : Natural
- Period_Max : Long_Integer
- Deadline : Long_Integer
- Priority : Character
+ Set_Basic_Parameters()
+ Set_Priority()
-Accepted_Contracts
1 .. Max_Contracts
frsh.Virtual Resource
- Contract_Id : FRSH_Contract_Id
- Server : rtep.Servers
+ Negotiate_Contract()
+ Get_Server_Id()
frsh.Distributed
+ Init()
+ Set_Basic_Parameters()
+ Set_Priority()
+ Negotiate_Contract()
+ Get_Server_Id()
1
«active»
Negotiation Task
frsh
«active»
Comm Task
1
rtep.Core
+ Init_Comm()
«interface»
rtep.Protocol
Get_Station_ID()
Get_Station_ID_By_Name()
Get_Station_ID_By_Position()
Send_Info()
Recv_Info()
Try_Recv_Info()
Any_Info()
Try_Lock_Distributed_Mutex()
Unlock_Distributed_Mutex()
test
1
1
mutex
+ Initialize()
+ Set_Locking_Policy()
+ Set_Ceiling_Priority()
+ Lock()
+ Unlock()
+ Finalize()
Cond_Var
+ Initialize()
+ Wait()
+ Signal()
+ Finalize()
posix
rtep
timers
+ Disarm_Timer()
+ Arm_Timer()
+ Create_Timer()
+ Set_Initial()
+ Set_Interval()
signals
+ Await_Signal()
+ Delete_All_Signals()
+ Add_Signal()
+ Get_Signal()
+ Set_Signal()
+ Set_Notification()
io
+ Open()
+ Close()
+ Read()
+ Write()
+ Ioctl()
«private»
rtep.System
+ Get_Current_Station_Address()
+ Set_Promiscous_Mode()
+ Open_Rt_Ep_Communication()
+ Close_Rt_Ep_Communication()
+ Send_Rt_Ep_Packet()
+ Receive_Rt_Ep_Packet()
«private»
rtep.Buffers
+ Rx_Queues_Insert_Packet()
+ Rx_Queues_Extract_Packet()
+ Rx_Queues_Read_Packet()
+ Tx_Queue_Insert_Packet()
+ Tx_Queue_Extract_Packet()
+ Tx_Queue_Read_Packet()
+ Server_Queues_Insert_Packet()
+ Server_Queues_Extract_Packet()
rtep.Stations
+ Get_Station_Position()
+ Get_Station_Name()
+ Check_Token_Master()
+ Valid_Station_Id()
+ Inhibit_Station_Id()
+ Get_Next_Station()
+ Get_Previous_Station()
+ Get_Station_Address()
+ Get_Station_ID_By_Address()
+ Valid_Multicast_Id()
+ Check_Subscription()
Ring_Spec
+Server
rtep.Servers
+ Create_Server()
+ Delete_Server()
+ Set_Priority()
+ Set_Max_Budget_And_Period()
+ Get_Server_Info()
+ Get_Current_Budget()
+ Get_Packets_Pending()
+ Get_Max_Priority_Server()
+ Packet_Enqueued()
+ Packet_Sent()
«private»
rtep.Packet
+ Get_New_RT_EP_Packet()
+ Destroy_RT_EP_Packet()
+ Is_Packet_In_Use()
+ Put_Station_Id()
+ Get_Station_Id()
+ Put_Channel_Id()
+ Get_Channel_Id()
+ Put_Timestamp()
+ Get_Timestamp()
+ Put_Multicast_Type()
+ Get_Multicast_Type()
+ Set_Synchronyzed()
+ Is_Synchronyzed()
+ Read()
+ Write()
+ Copy_Stream_Array_In_Rt_Ep_Packet()
+ Read_Rt_Ep_Packet()

Annex A - FNA Reference Manual

REFERENCES

[1] Frescor project home page. http://www.frescor.org/.
[2] M. Aldea and others others. FSF: A Real-Time Scheduling Architecture Framework. In 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06), pages 113124, San Jose (CA, USA), April 2006. IEEE.
[3] Josá María Martínez, M. González Harbour, Juán López Campos, J.Javier Gutierrez, and Julio L. Medina. Adding contract-based reservation services to a Hard Real-Time Ethernet Protocol. In 4th International Workshop on Real-Time Networks (RTN'05), Palma de Mallorca (Spain), July 2005.
[4] Ricardo Marau, Luis Almeida, Paulo Pedreiras, M. González Harbour, Daniel Sangorrín, and Julio M. Medina. Integration of a exible network in a resource contracting framework. In In Proc. of the WiP session of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'07). IEEE, April 2007.
[5] José María Martínez and M. González Harbour. RT-EP: A Fixed-Priority Real Time Communication Protocol over Standard Ethernet. In 10th International Conference on Reliable Software Technologies, Ada-Europe, pages 180195. Springer, June 2005.
[6] Giorgio C. Buttazzo. Hard Real-Time Computing Systems. Kluwer Academic Publishers, 2002.
[7] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic task scheduling for hard-realtime systems. In The Journal of Real-Time Systems, pages 2760, Palma de Mallorca (Spain), 1989. Kluwer Academic Publishers.
[8] Clarence Filsls John Evans. Deploying ip and mpls qos for multiservice networks: Theory and practice. Morgan Kaufmann Publishers, 2007.
[9] Thomas Nolte, Mikael Nolin, and Hans Hansson. Server-based scheduling of the can bus. In Proceedings of the 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'03), pages 169176, Calouste Gulbenkian Foundation, Lisbon, Portugal, September 2003. IEEE Industrial Electronics Society.
[10] Brian Neil Levine and J. J. Garcia-Luna-Aceves. A comparison of reliable multicast protocols. Multimedia Systems, 6(5):334348, 1998.
[11] M. Velazquez. A survey of distributed mutual exclusion algorithms. In Technical Report CS-93-116, Colorado State University, 1993.
[12] P. C. Saxena and J. Rai. A survey of permission-based distributed mutual exclusion algorithms. In Computer Standards and Interfaces Volume 25 Issue 2, pages 159181, 2003.