FSF: A Real-Time Scheduling Architecture Framework

M. Aldea[*], G. Bernat[+], I. Broster[+], A. Burns[+], R. Dobrin[&], J. M. Drake[*], G. Fohler[&], P. Gai[#], M. González Harbour[*], G. Guidi[#], J.J. Gutiérrez[*], T. Lennvall[&], G. Lipari[#], J.M. Martínez[*], J.L. Medina[*], J.C. Palencia[*], M. Trimarchi[#]

[*]*Dpto. de Electrónica y Computadores, Universidad de Cantabria, Santander, Spain (mgh@unican.es)*
[+]*Department of Computer Science, University of York, UK (burns@cs.york.ac.uk)*
[&]*Computer Engineering Department, Malardalen University, Vasteras, Sweden (gerhard.fohler@mdh.se)*
[#]*ReTiS Lab, Scuola Superiore Sant'Anna, Pisa, Italy (lipari@sssup.it)*

Abstract[1]

Scheduling theory generally assumes that real-time systems are mostly composed of activities with hard real-time requirements. Many systems are built today by composing different applications or components in the same system, leading to a mixture of many different kinds of requirements with small parts of the system having hard real-time requirements and other larger parts with requirements for more flexible scheduling and for quality of service. Hard real-time scheduling techniques are extremely pessimistic for the latter part of the application, and consequently it is necessary to use techniques that let the system resources be fully utilized to achieve the highest possible quality. This paper presents a framework for a scheduling architecture that provides the ability to compose several applications or components into the system, and to flexibly schedule the available resources while guaranteeing hard real-time requirements. The framework (called FSF) is independent of the underlying implementation, and can run on different underlying scheduling strategies. It is based on establishing service contracts that represent the complex and flexible requirements of the applications, and which are managed by the underlying system to provide the required level of service.

1. Introduction

Many previous real-time system design approaches and scheduling schemes have focused on safety-critical control applications characterized by stringent timing constraints. In these applications, missing a single deadline can jeopardize the entire system behavior and even cause catastrophic consequences. Compared to traditional control environments, new applications in areas such as industrial automation and consumer multimedia systems are characterized by less stringent timing constraints. In these systems, missing some deadlines is acceptable and can be tolerated up to a certain degree. Nevertheless, control on the system's performance and resource usage is still required, although with more relaxed constraints, and some of the timing requirements of the system remain hard.

Consequently, these new applications demand various types of tasks and constraints within the same system. There are requirements for flexibly sharing the available resources, and in many cases these requirements change dynamically. While off-line guarantees are still essential for meeting a minimum performance, different types of requirements and runtime changes have to be included in the system analysis, such as demands on quality of service (QoS) or acceptance probabilities. These systems often demand the coexistence and co-operation of diverse scheduling algorithms to combine properties. Algorithms might even change during system's runtime to better adapt to environment variations.

Another problem that real-time scheduling has to face is the difficulty in persuading engineers to use the most advanced techniques in their applications. Pressure of time to market makes it difficult to spend the resources needed to understand and implement scheduling theory, to the extent that many practitioners regard real-time theory as "the solution to the wrong problem". In our view, the difficulty is not really with the "solution", but rather in the way this solution is presented to the application developer.

In this paper, we present the architecture framework of the FIRST (Flexible Integrated Real-time Scheduling Technologies) project with the objective of providing engineers with a scheduling framework that represents a high-level abstraction that lets them concentrate on the specification of the application requirements, while

1. This work has been funded by the *Commission of the European Communities* under contract IST-2001-34140 (FIRST project)

the system will transparently use advanced real-time scheduling techniques to meet those requirements.

The framework aims at achieving a trade-off between predictability in performance and efficiency in resource utilization, even during overload situations. The FIRST scheduling framework (FSF) provides for the co-operation and coexistence of standard scheduling algorithms such as table driven (TD), fixed priority (FP), and earliest deadline first (EDF). It enables users to select the best-suited service for individual activities, rather than the prevalent monolithic approaches enforcing a single regime for the entire system.

The key abstraction in FSF is the use of a hierarchical scheduling architecture based on servers. Careful use of servers allows different parts of the system (whether they are processes, applications, components, or schedulers) to use budgeting schemes. Not only can servers be used to help enforce temporal independence, but a process can interact with a server to query its resource usage and hence support the kinds of algorithms where execution paths depend on the available resources.

Server algorithms are defined for particular scheduling schemes, such as fixed priority or EDF. In order to keep our framework independent of specific scheduling schemes, we introduce an interface between applications and the global scheduler, called the service contract. So instead of using parameters of a specific server algorithm, the application defines its needs in the form of service contracts, which are independent of the actual server used. Thus, diverse server algorithms and implementations, based on a variety of scheduling schemes can then meet the service contracts. Should the application be run on a system with a different scheduling scheme, the service contracts remain the same, only their realization in terms of the specific server algorithms used is different. Application requirements are mapped to a contract, which is then mapped to the server parameters by the scheduler. Contracts can be verified at design time by providing off-line guarantees, or can be negotiated at runtime, when they may or may not be admitted.

1.1. Related work

A general methodology for temporal protection in real-time systems is the resource reservation framework [15][23][24][28]. The basic idea, which was formalized by Rajkumar [27], is that each task is assigned a server that is reserved a fraction of the processor available bandwidth: if the task tries to use more than it has been assigned, it is slowed down. This framework allows a task to execute in a system as if it is executing on a dedicated virtual processor, whose speed is a fraction of the speed of the processor. Thus, by using a resource reservation mechanism, the problem of schedulability analysis reduces to the problem of estimating the computation time of the task without considering the rest of the system.

Jones et al present in [12] a model of CPU Reservations and Time Constraints in the Rialto OS, which was initially designed as a resource manager with distributed and real-time capabilities. The system has similarities with the framework presented in this paper because it defines an abstraction similar to the server (the activity) that can be used to schedule several threads. This work has evolved to Rialto/NT which is the implementation of the Rialto scheduling scheme in a high priority level of a Windows NT system [13].

Nieh and Lam [26] present SMART, an API for a fairness-based processor scheduler that fully supports applications with soft and non real-time, overload management, any dynamic adaptation.

Recently, many techniques have been proposed for extending the resource reservation framework to hierarchical scheduling. Baruah and Lipari in [19] propose the hierarchical constant bandwidth server (HCBS) algorithm, which permits composition of CBS schedulers at arbitrary levels of the hierarchy. A similar work has been done by Saewong et al [29] in the context of the resource kernels. Liu and Deng in [7][8] proposed a two-level hierarchical architecture, that uses the EDF as global scheduler and uses a dedicated total bandwidth server (TBS) for each application. This work has been later extended by Kuo et al [14] for using rate monotonic as global scheduling algorithm.

Lipari and Baruah in [16][17] presented the bandwidth sharing server (BSS) scheduling algorithm that uses EDF as global scheduling algorithm, and permits selecting any scheduling algorithm as an application-level scheduler.

Feng and Mok [25] presented a general methodology for hierarchical partitioning of a computational resource. It is possible to compose schedulers at arbitrary levels of the hierarchy. They also propose a simple sufficient schedulability test for any scheduler at any level of the hierarchy.

There have been several efforts in the introduction of QoS in distributed systems. Wang et al. [32] presented techniques for dynamically adapting the CORBA Components Model (CCM) middleware by introducing alternative configurations via reflective information in the containers operation. The Quality Objects project (QuO) [5] also proposes abstractions and mechanisms that complement CORBA with concepts [30] similar in spirit to those presented in this paper.

1.2. Specific contributions

Compared to these approaches, FSF provides application developers a generalized architecture framework that combines different kinds of requirements:

- co-operation and coexistence of standard real-time scheduling schemes, time-triggered and event-triggered, dynamic and fixed priority based, as well as off-line based through a common architecture that is independent on the underlying scheduling mechanism
- integration of different timing requirements such as hard and soft, and more flexible notions allowing the distribution of spare capacity according to the specified application requirements and capabilities
- temporal encapsulation of subsystems in order to support the composability and reusability of available components including legacy subsystems
- integration of the shared resource management
- integrated view of the processor and network resources for distributed applications

Support for FSF contracts has been implemented in two POSIX compliant real-time operating systems, MaRTE [2] and SHARK [9], which are based on FP and EDF scheduling schemes, respectively, thus illustrating the platform independence of the presented approach.

The rest of the paper is organized as follows. Section 2 summarizes the results of a survey on industrial needs for real-time applications, leading to a set of requirements that FSF should address, and to the definition of the system model. Section 3 describes the service contracts that allow the applications to specify their flexible timing requirements independently of the underlying scheduler. Section 4 briefly discusses the implementations of FSF that have been developed in two different operating systems. Finally, section 5 gives our conclusions.

2. Application Requirements and Scheduler Flexibility

Key in FIRST is the concept of integration of complex applications with a variety of real-time requirements, including hard real-time behavior as well as soft requirements. It is noted that in many modern real-time systems, only a small fraction of the system could be considered to have hard deadlines, yet most of the rest of the system might have important timing or performance requirements.

The coexistence and cooperation of different scheduling schemes is encouraged in the FIRST framework; different applications with their own scheduling requirements and algorithms need to be integrated together on one platform.

2.1. Application Characteristics

The application characteristics and requirements used to develop the framework were obtained by performing a survey of industrial needs. The range of products that are called “real-time” is very diverse: from small embedded control systems, digital multimedia devices, consumer electronics and image processing chips in cameras to large systems like guidance and control systems for avionics. As systems grow in size, the amount of critical or hard real-time code does not grow in proportion, because much of the software is concerned with less time-critical functionality.

There is a clear trend towards multi-processor solutions, even for smaller systems. In the long term, the availability of inexpensive single chip computers and the ability to put multiple custom processing cores on FPGA integrated circuits indicates the future will become distributed on more and more processors. Many industrial control systems are based on conventional industrial processors linked through field busses and ethernet networks.

Real-time theory is ineffective in many systems. For example in larger systems where there is a dynamic number of processes, perhaps up to 1000 at a time, forms of worst case analysis are not appropriate because the worst case is unlikely to occur; to ensure that all deadlines are met in the worst case leads to very low utilization. Also, the dynamic nature of such real-time systems means that it is difficult to use much real-time systems theory because it is based on assumptions typically found in static systems, where much is known about the future demands on the systems. More fundamentally, notions such as ‘deadline’ fail to specify requirements adequately, particularly in a softer system where scheduling is a battle to maintain a high quality of service. In particular scenarios, it may be important to ensure low latency, in others it is more important to exhibit low jitter. Further, in many cases industry does not know how to describe or measure quality of service.

The periodicity requirements assumed in most real-time scheduling techniques are not always simple. For example, it is common to find systems with variable period, either on a *continuous or discrete scale*. The continuous scale, variable within an upper and a lower limit, follows the elastic task model [6]. The discrete model occurs when the period can change to any of a set of discrete values, and is useful in control systems with different modes of operation or levels of quality.

A similar situation occurs with the resource usage, i.e., how much processing time each job requires. In addition to the classic model with a minimum execution time guaranteed by the system, we find systems in which the execution time may vary on a *continuous scale*, such as in any-time algorithms, or in a *discrete scale*, such as in N-version or imprecise computation schemes.

2.2. System Level Integration

Systems consisting of applications which change their timing behavior over their lifetime are common. The timing behaviors change either because of software changes, mode changes, data-dependencies, input/output traffic levels, and other causes. These place a requirement on the scheduling system to be able to respond to these changes.

A key concept in the FIRST framework is a 'contract' between the application and the scheduler. Contracts can be negotiated at initialization, when requirements change, or when new software is added to the system, regardless of whether or not the applications are previously known to the system. The time to complete a negotiation must be adjustable by assigning specific system resources to the part of the system making them. Integration of different applications with different local scheduling schemes is important for supporting component-based methodologies as well as for integrating legacy code into systems based on the FIRST framework.

To protect existing contracts and ensure that applications are sensitive to time-domain failures, it is clear that the operating system must be capable of enforcing the timing behavior of applications, even if the application attempts (by accident or maliciously) to exceed its contracted behavior.

2.3. The application model

As a result of the application requirements that we have discussed, the FIRST framework is able to give support to systems running one or more applications, each consisting of a number of schedulable components that encapsulate a given functionality with timing and quality of service requirements; components may have requirements for sharing resources. Each component may have one or more threads of control, and may even be distributed among different processing nodes and networks. The application component negotiates a set of service contracts, which specify the component requirements, and then binds each thread (and each message stream in distributed components) to a contract. Multiple threads may be bound to the same contract if hierarchical scheduling is supported. Although there may be multiple instances of the same component, each used by the same or by different applications, each instance will have its own threads and service contracts. Key to understanding the specific timing requirements supported by the framework to provide flexible scheduling is the contract model that will be described next.

3. Service contracts

The service contract is the mechanism that we have chosen for the application to dynamically specify its own set of complex and flexible execution requirements. From the application's perspective, the requirements of an application or application component are written as a set of service contracts, which are negotiated with the underlying implementation. To accept a set of contracts, the system has to check as part of the negotiation if it has enough resources to guarantee all the minimum requirements specified, while keeping guarantees on all the previously accepted contracts negotiated by other application components. If as a result of this negotiation the set of contracts is accepted, the system will reserve enough capacity to guarantee the minimum requested resources, and will adapt any spare capacity available to share it among the different contracts that have specified their desire or ability for using additional capacity.

As a result of the negotiation process initiated by the application, if a contract is accepted, a server is created for it. The server is a software object that is the run-time representation of the contract; it stores all the information related to the resources currently reserved for that contract, the resources already consumed, and the resources required to handle the budget consumption and replenishment events in the particular operating system being used. Figure 1 shows the relationship between the service contract in the application side, and the server in the underlying implementation

The system may be configured to perform an on-line schedulability analysis test at negotiation time. If the test is enabled, a new contract set is accepted only if the new system situation passes the test. However, because on-line tests may be suboptimal, for static systems it is also possible to perform a more exact off-line schedulability analysis test, and disable the on-line analysis. In that case, a contract set will always be accepted.

Because there are various application requirements specified in the contract, they are divided into several groups, also allowing the underlying implementation to give different levels of support trading them against implementation complexity. This gives way to a modular implementation of the framework, with each module addressing specific application requirements. The mini-

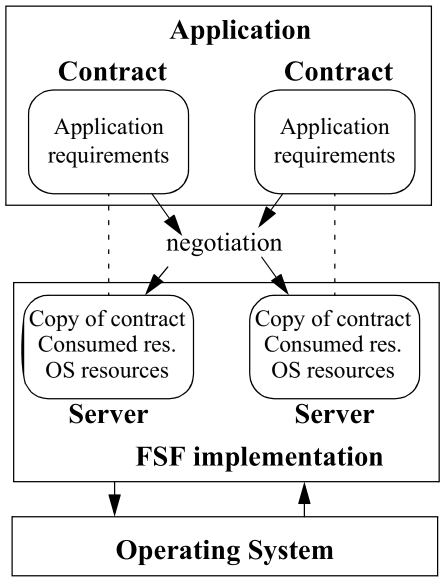

Figure 1. Contract negotiation process

mum resources required by the application to be reserved by the system are specified in the *core* module. The requirements for mutual exclusive synchronization among parts of the application being scheduled by different servers or among different applications are specified in the *shared objects* module. Flexible resource usage is associated with the *spare capacity* and *dynamic reclamation* modules. The ability to compose applications or application components with several threads of control, thus requiring hierarchical scheduling of several threads inside the same server are supported by the *hierarchical scheduling* module. Finally, the requirements of distributed applications are supported by the *distributed* and the *distributed spare capacity* modules. We will now explain these modules together with their associated application requirements.

3.1. Core

The core module contains the service contract information related to the application minimum resource requirements, the operations required to create contracts and negotiate them, to bind a thread to a server, and the underlying implementation of the servers with a resource reservation mechanism that allows the system to guarantee the resources granted to each server. The application requirements specified in the core module are shown in Table 1.

The basic application requirements are the *minimum budget* and *maximum period* of the server. The server will guarantee that every period, the part of the application running on it will get, if requested, at least the minimum budget. Passing a schedulability test only guarantees that the minimum requirements specified in the contract will be satisfied. In particular, for components with hard deadlines it is the responsibility of the application developer to ensure that the execution times required in each server period by the threads bound to a contract are within the minimum budget.

Table 1. Core attributes

Name	Description
minimum budget	Minimum execution capacity per server period
maximum period	Maximum server period
workload	Whether the workload running on the server is bounded or indeterminate
deadline	The deadline of the server
D=T	Whether the server's deadline is equal to the period or not
budget overrun signal	The mechanism to get notification of a possible budget overrun for bounded workloads
deadline miss signal	The mechanism to get notification of a possible deadline miss for bounded workloads

Another important timing requirement is the server's *deadline*. The server guarantees that a piece of work of size less than or equal to the minimum budget and requested for a server with full capacity will be completed by the server's deadline. Since the period may be adjustable, it is possible to specify that the deadline is equal to the period.

The *workload* attribute describes two fundamentally different models of the work that the server has to manage. The first model is the *bounded* workload, in which the application can bound the amount of work that it requests during an interval equal to the server's period. We call this work a *job*. In this model, it is possible for an application to tell the system that it has completed the current job (thus allowing the system to make its current available budget equal to zero), and that it should be awakened by the system at the beginning of the next job. If requested, the system can notify the application about a job overrunning its budget, or missing its deadline. This is the preferred approach for periodic of sporadic tasks running on top of an FSF server. The notification can be addressed to a thread under the server, or to a special-purpose handler thread, created by the application.

The framework provides the ability to awaken a bounded workload job in two different ways: timed, or event driven. The timed wake up is achieved with the use of a timer or some OS timing mechanism, but the event driven mechanism requires a synchronization object that is managed by the FSF. Consequently, a mechanism exists to create such synchronization objects, and to signal them to awaken a bounded workload waiting upon it. In both cases, timed or event-driven, the new job will only be started after the server's budget has been replenished at the next server's period.

The second workload model is called *indeterminate*, and represents the case in which the application cannot make any guarantees over the amount of work requested to be executed inside a server period. In this case, the server guarantees the minimum budget, and defers further execution to a next period, if required to guarantee the execution of the other servers in the system. There is no budget overrun or deadline miss notification in this case, because there is no concept of a *job*.

Some application parts may not have any real-time requirements but may just want to run when the system is not busy running real-time activities. For this purpose we provide the ability to run activities in the *background*, in a round-robin fashion, and with no time guarantees. It is possible to specify it by creating a special background contract.

With the described services, the core module provides support for the basic timing requirements of real-time applications, including the ability to reserve and get guarantees on execution time budgets, the ability to specify arbitrary deadlines, and to detect budget overruns or deadline misses. This is done independently of the underlying scheduler; for instance, we could have fixed priorities with sporadic servers [31], or EDF with constant bandwidth servers underneath [1].

3.2. Shared objects

It is common for applications to have to share data or other resources in a mutually exclusive way with other applications or concurrent parts of the same application. Most real-time synchronization protocols are able to bound the delay that an application may experience due to the use of shared objects [4], but nevertheless this delay exists and must be taken into account by the schedulability test.

The shared objects module of FSF allows the application to create shared objects and to specify in the contract attributes all the information required to do the schedulability analysis. Table 2 shows the attribute related to shared objects that can be specified as part of a service contract.

Table 2. Shared object attributes

Name	**Description**
list of critical sections	Each critical section has a reference to the shared object and its worst case execution time

The set of shared objects present in the system together with the lists of critical sections specified for each contract are used for schedulability analysis purposes only. A run-time mechanism for mutual exclusion is not provided in FSF for two important reasons. One of them is upward compatibility of previous code using regular primitives such as mutexes or protected objects (in Ada); this is a key issue if we want to persuade application developers to switch their systems to the FSF environment. The second reason is that enforcing worst case execution time for critical sections is expensive. The number of critical sections in real pieces of code may be very high, in the tens or in the hundreds per task, and monitoring all of them would require a large amount of system resources.

The FSF application does not depend on any particular synchronization protocol, but there is a requirement that a budget expiration cannot occur inside a critical section, because otherwise the blocking delays could be extremely large. This implies that the application is allowed to overrun its budget for the duration, at most, of the critical section, and this extra budget is taken into account in the schedulability analysis.

3.3. Spare capacity

Many applications have requirements for flexibility in the amount of resources that can be used. The spare capacity module allows the system to share the spare capacity that may be left over from the negotiation of the service contracts, in a static way. During the negotiation, the minimum requested resources are granted to each server, if possible. Then, if there is any extra capacity left, it is distributed among those applications that have expressed their ability to take advantage of it.

Table 3 shows the service contract attributes related to the spare capacity. There are two ways of making use of spare capacity, described with the *granularity* attribute. In the *continuous* granularity, the application is able to make useful work for any value of budget between the *minimum* and the *maximum budget*, and for any period between the *maximum* and the *minimum period*. The case of continuous budget, for instance, corresponds to anytime algorithms, while the continuous period corresponds to an iterative algorithm, for instance a video display process, in which the quality increases with the frequency of execution.

The *discrete* granularity is designed for n-version algorithms that can run different versions with different quality levels, each with a different value of budget per

Table 3. Spare capacity attributes

Name	Description
granularity	indicates how we can make use of extra capacity: continuous or discrete utilization values
maximum budget	maximum usable budget
minimum period	minimum useful period
utilization set	set of pairs {budget,period}, used for discrete granularity
importance	a fixed priority used to distribute extra capacity
quality	a relative number used to distribute extra capacity among servers of the same importance

period. The possible values are described in the contract through the *utilization set* attribute.

The method to distribute the spare capacity is based on two numeric values called the *importance* and *quality*. The importance is a small integer like a fixed priority intended for non-cooperative servers: a higher importance server will get all the available spare capacity before any lower importance server. If there are servers of the same importance level, they share the extra capacity proportionally to their quality value, which is intended for cooperative servers: the share that they get is proportional to their value divided by the total quality for their importance level.

The distribution of spare capacity is made every time there is a negotiation, a renegotiation, or just a change of quality and importance. The values assigned to each server are reported to them, so that they can use the information to know how to run. The assigned capacity is guaranteed until the next negotiation or change.

3.4. Dynamic reclamation

This module is used to dynamically reclaim any execution capacity that is not used by the different servers, so that it can be assigned to other servers that can make use of it. The application requirements are similar to spare capacity sharing, except that the application only knows that it has some additional budget later during its server period. Therefore, this reclamation is appropriate for anytime algorithms (i.e., continuous granularity servers) and is inappropriate for n-version algorithms which must know the version to run from the beginning of the current instance.

Dynamic reclamation is a difficult scheduling problem that is not completely solved. If new dynamic reclamation techniques become available in the future, the FSF can immediately take advantage of them because all the information on how the application can make use of it is already in the service contract.

Table 4. Hierarchical attributes

Name	Description
scheduling policy	This is an identifier for the top-level scheduling policy
scheduler init info	Scheduling-policy-dependent information that is supplied at server initialization time

Because this module shares its application requirements with the spare capacity module, it has no contract server attributes of its own.

3.5. Hierarchical scheduling

One of the application requirements that FSF addresses is the ability to compose different applications, possibly using different scheduling policies, into the same system. This can be addressed with support in the system for two-level hierarchical scheduling. The lower level is the scheduler that takes care of the service contracts, using an unspecified scheduling policy (for instance, a CBS on top of EDF, or a sporadic server on top of fixed priorities). The top level is a scheduler running inside one particular FSF server, and scheduling the application threads with whatever scheduling policy they were designed. Of course the top level scheduler is just local to a particular server; the only global scheduler is the lower level server scheduler. With this approach, it is possible to have in the same system one application with, for example, fixed priorities, and another one running concurrently with an EDF scheduler.

We have chosen to provide the top-level schedulers inside the FSF implementation because it is simpler than having a specific API for the application to develop its own scheduler. We are currently providing three top-level schedulers: fixed priorities, EDF, and table-driven. The service contract attributes associated with the hierarchical scheduling module are shown in Table 4.

For the *scheduling policy* attribute the allowed values are fixed priorities, EDF, table driven, and *none*. The latter case corresponds to a server with no top-level scheduler, that only allows one thread to be bound to it.

The *scheduler init info* attribute is scheduler-dependent information. For fixed priorities or EDF it is empty, and for table-driven scheduling it contains the table with the schedule.

In addition to the server's attributes each thread that is bound to the server has its own *scheduling parameters*, that depend on the particular scheduling policy.

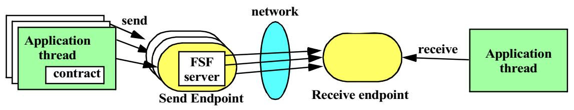

Figure 2. Communication elements in FSF

For instance, they have a priority for fixed priorities, or a relative deadline for EDF.

3.6. Distribution (core)

FSF is designed to support applications with requirements for distribution. Although there are few networks that are capable of guaranteeing hard real-time requirements, there are some field buses, like the CAN Bus, and some network protocols on standard networks that provide priority-based hard real-time behavior, and which can be used to implement the distributed FSF module.

The first step towards distribution is the ability to support service contracts for the network or networks used to interconnect the different processing nodes in the system. Similar to the core FSF 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 use the same contract that is used for processing nodes, and thus the core attributes for distribution are the same as for the core FSF, described in Table 1, with the addition of the *network id* attribute (see Table 5), that identifies the contract as a network contract for the specified network. The default value for the *network id* is *null*, which means that the contract applies to the processing node where the contact is negotiated.

For the FSF implementation to keep track of consumed network resources, i.e., the amount of packets sent in a server period, and to enforce the budget guarantees, it is necessary that the information is sent and received through specific FSF services. To provide communication in this context we need to create objects similar to the sockets used in most operating systems to provide message communication services. We call these objects *communication endpoints*, and we distinguish send and receive endpoints (see Figure 2).

A *send endpoint* contains information about the network to use, the destination node, and the port that identifies a reception endpoint. It is bound to a network server 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. It provides message buffering for storing messages that need to be sent.

Table 5. Distributed core attributes

Name	Description
network id	Identifies the network for which the contract is negotiated; if null, the contract is negotiated on a processing node

Table 6. Distributed spare capacity attribute

Name	Description
granted capacity flag	Once the negotiation is finished, the first values for the budget and period given to the server must not change automatically

A *receive endpoint* contains information about the network and port number to use. It provides message buffering for storing the received messages until they are retrieved by the application. A receive endpoint may get messages sent from different send endpoints, possibly located in different processing nodes.

3.7. Distribution (spare capacity)

In the distributed FSF 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. For example, a distributed transaction may have several activities executing in different processing nodes. One of them is periodic, and the others are activated by the arrival of a message from the preceding activity. Therefore, the latter activities inherit the period of the first activity (with the additional jitter introduced by the processing and message transmission). If the transaction allows a continuous scale of periods between some minimum and maximum values, separate negotiations in the network and in the different processing nodes will most probably result in different periods because the spare capacity is different in each node. Since the transaction cannot run with different periods, there needs to be some renegotiation to change the period to the maximum obtained (representing the minimum resource consumption). During this renegotiation things might have changed, requiring further renegotiation rounds.

We do not want to embed all this complexity into the FSF implementation. Therefore, we have chosen to give a minimum support for spare capacity distribution inside FSF, and leave the consensus problem to some higher-level manager that would make the negotiations for the application. For this minimum support there is a new attribute in the service contract called the *granted capacity flag* (see Table 6), which has the implication that the period or budget of the server can only change if a renegotiation or a change of quality and importance is requested for it; it may not change automatically, for instance because of negotiations for other servers. This provides a stable framework while performing the distributed negotiation.

For a server with the *granted capacity flag* set, there is an operation that is used to return spare capacity that has been assigned by the system to that server, but that cannot be used due to restrictions in other servers of a distributed transaction.

4. Support on operating systems

The FIRST project has defined a clear API for using the FSF either from the application or from some middleware agent that manages the quality of service requirements for a system. The API allows the application to be completely independent of the underlying FSF implementation. Figure 3 shows the main elements of the API, decomposed into the different modules described in Section 3. The API is provided for both C and Ada.

The FSF services and associated API are designed to be implementable inside any real-time operating system. It is also possible to implement them on top of an operating system that provides the ability to install application-level schedulers. There is currently no standard way of providing this kind of functionality, and therefore each FSF implementation would have to be tailored to a specific OS. To overcome this difficulty in the future, an API has been defined to specify services that allow an OS to provide application-level scheduling support in a uniform way [3], and we have started the process to request inclusion of this API into the real-time POSIX standard [10].

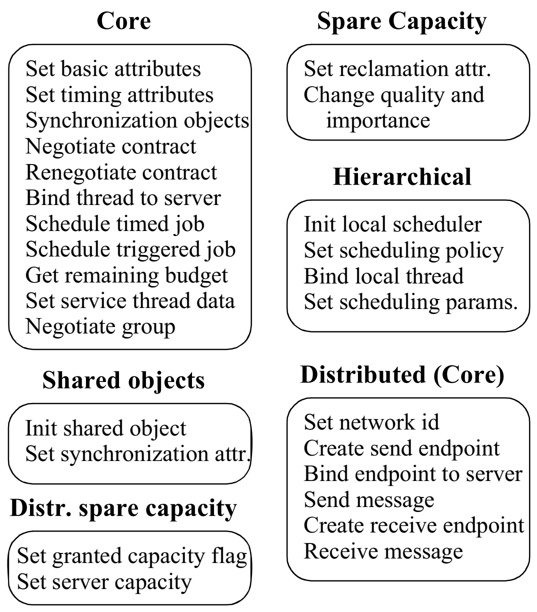

Figure 3. Main elements of the FSF API

As a proof of concepts, the FSF services have been implemented in two real-time kernels, MaRTE OS[1] [2] and Shark[2] [9], both of which follow the POSIX minimum real-time profile [11]. These implementations show that even with very different scheduling strategies it is possible to provide a portable FSF API.

In MaRTE OS the FSF services were implemented using the application-defined scheduling API proposed for the POSIX standard. The underlying operating system is based on the traditional real-time POSIX fixed priority scheduling, and we install a secondary application defined scheduler that contains the FSF servers and manages the negotiated contracts. The modules that are currently implemented in MaRTE OS are the core (using fixed priorities and sporadic servers [31]), shared objects (with the SRP [4]), spare capacity sharing, and the distributed module.

Shark does not implement the application-defined scheduling API, but instead its kernel was designed with a modular structure to allow the coexistence and interplay of different scheduling algorithms. The underlying scheduling algorithm is EDF. The server algorithm, used for implementing the service contract, is the GRUB algorithm (Greedy Reclamation of Unused Bandwidth [18]). This algorithm automatically adds dynamic reclamation to the CBS (Constant Bandwidth Server [1]).The modules that are implemented in Shark are the core (using EDF and constant bandwidth servers), spare capacity sharing (restricted to deadlines equal to periods), dynamic reclamation, hierarchical scheduling, and the shared objects module (using Bandwidth Inheritance (BWI) [20], which extends the priority inheritance protocol to server based scheduling).

In both implementations, MaRTE OS and Shark, the admission control algorithm and the spare capacity calculation are implemented via a special thread, called the service thread, which consumes part of the system resources in a controlled way. The budget and period of the service thread can be adjusted by the user, to trade between timeliness of negotiations and overhead. In any case, when the budget of the service thread is exhausted, it can run in the background.

The admission control algorithm used to negotiate a contract and create its corresponding server if accepted, is based on utilization tests in both implementations,

1. MaRTE OS is distributed under the GPL and can be found at: `http://marte.unican.es/`

2. Shark is distributed under the GPL and is downloadable from: `http://shark.sssup.it/`

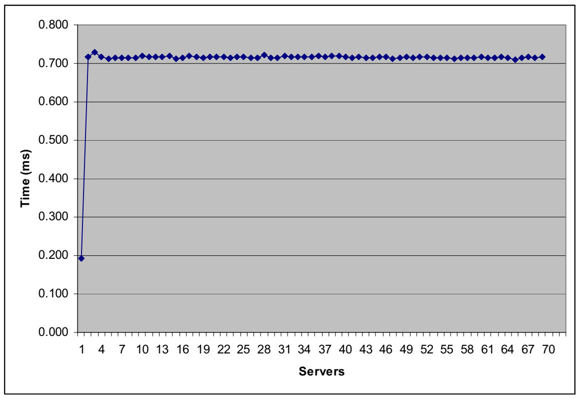

Figure 4. Network negotiation times

although more elaborate algorithms could be used in the future, for example based on response time analysis. Both take into account the blocking effects of shared object synchronization. The utilization test in Shark is simpler but is restricted to deadlines equal to periods for the servers, while the utilization test in MaRTE OS allows deadlines smaller than or equal to the periods.

We have made extensive tests to evaluate the approach and the associated implementations. It is significant that although very different underlying schedulers and synchronization protocols were used, the test applications were portable between both systems.

As an example of the experimental data obtained, Table 7 shows the context switch times measured in Shark for different hierarchical scheduling strategies. These experiments were executed on a (rather slow) Pentium Celeron running at 300 Mhz. The test application consisted of 3 servers, each one with a different local scheduler, one with EDF, the second one with Fixed Priority (FP) and the third one with Round Robin (RR). We measured the maximum context switch time between tasks belonging to the same server and between tasks belonging to different servers. The overheads for the hierarchical scheduling structure in Shark, and consequently also for the simpler case in which no hierarchical scheduling is supported, are comparable to the overhead for a normal context switch.

Figure 4 shows the time it takes to make a negotiation for the network (a 100Mb/s real-time ethernet), under the distributed FSF in MaRTE OS, in a system with four processing nodes, in which all the negotiations are made by a single node, with delays between negotiations. The contracts specify a network utilization of 1%, and have deadlines equal to periods. The admission test in this case is a plain rate monotonic utilization test, which gives constant time in relation to the number of servers, and has a utilization bound of 69%.

Figure 5 shows the results of reachable utilization in Shark. In this experiment the minimum and maximum utilization obtainable through the spare capacity sharing mechanism were measured for systems with a increasing number of servers. The total utilization of the servers was bounded to 70%. We can see how the spare capacity is completely distributed, with no difference between the minimum and maximum utilizations obtained in the different runs of the experiment.

The scalability of FSF has to be analyzed from two points of view. From the execution runtime overhead, the scheduling algorithms used to manage the server information and schedule the threads have $O(\log n)$ complexity, where n is the number of servers currently instanced in the system. Adding the server management scheme on top of the EDF or fixed priority scheduler does not add further complexity to the algorithm, so the framework is very scalable. From the point of view of negotiation times, the scalability depends on the particular admission test implemented. For example, in Shark we implemented an utilization based test that had the restriction of deadlines equal to periods, and the advantage of a linear dependency, $O(n)$, on the number of servers. In MaRTE OS we used a more elaborate utilization test that can be used for deadlines less than or equal to the periods, which has a time dependency of $O(n^2)$, making it a bit less scalable. Of course, off-line admission test may be carried out for a fully scalable approach.

Table 7. Maximum context switch times in case of hierarchical schedulers. Times in μs.

	EDF	FP	RR
EDF	14.787	11.497	11.503
FP	-	11.793	15.749
RR	-	-	25.787

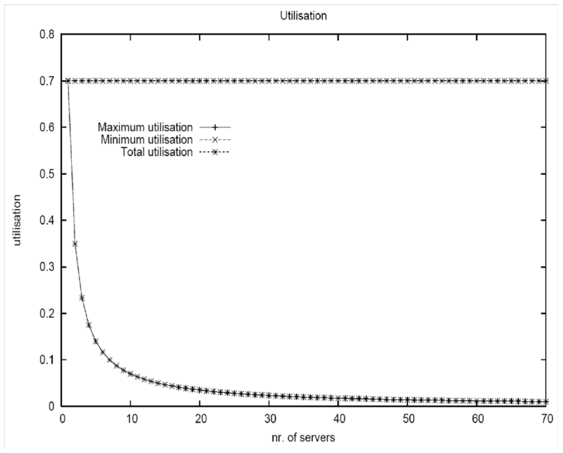

Figure 5. Reachable Utilization in Shark

The framework has been successfully used in three different case studies: a multimedia application, an artificial intelligence application for soccer robots, and a distributed controller for an industrial robot. In two of the case studies, the applications were already built, and they were migrated to FSF, taking advantage of the new real-time features built into FSF, such as the ability to guarantee minimum processing resources and network bandwidth. For instance, it was possible to combine in the same network hard real-time traffic for robot control operations (with 10-50 millisecond hard deadlines) with image transmission over the network. Migration to the FSF was straightforward as it did not require changes to the software architecture. The other case study had software specifically developed with the FSF API in mind, and it showed the ease of use, even to novice users.

5. Conclusions and future work

In this paper, we have presented the scheduling architecture proposed in the FIRST (Flexible Integrated Real-Time Scheduling Technologies) project. It provides for the coexistence and cooperation of diverse real-time scheduling schemes, in particular table driven, fixed priority, and earliest deadline, and integrating of different task types such as hard and soft, or more flexible notions. The FIRST scheduling framework, FSF, proposes a hierarchical scheduling structure based on servers. Application requirements are defined as service contracts which provide an interface layer to the underlying servers provided by the operating system. Contracts can be verified at design time by providing off-line guarantees, or can be negotiated at runtime, when they may or may not be admitted.

FSF provides for the temporal encapsulation of subsystems to support the composability and reusability of available components including legacy subsystems. FSF also represents a higher level of abstraction relative to the operating system primitives. It provides an abstract API that makes time a first class citizen, making it easier to build real-time applications. Instead of concentrating on condition variables, timers, or other low-level RTOS mechanisms, the application developer is able to think in terms of deadlines, execution time budgets, or high-level synchronization primitives.

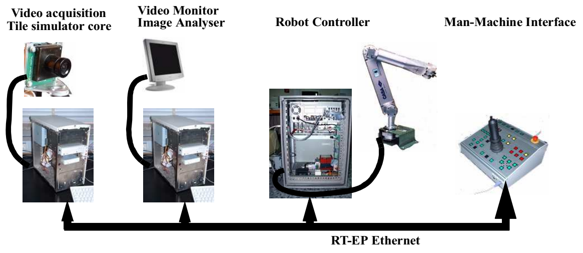

Figure 6. Industrial robot controller implemented with distributed FSF

FSF has been implemented in two contrasting POSIX compliant real-time operating systems, MaRTE and Shark, which are based on fixed priority and EDF scheduling schemes, respectively, thus illustrating the platform independence of the presented approach. The implementation has proved to be feasible, efficient, and easy to use. It has been successfully used to implement a distributed controller for an industrial robot, a multimedia application, and an artificial intelligence application controlling a team of soccer robots. These implementations and case studies have shown that FSF can be implemented efficiently with moderate effort, is effective and easy to use from the application, and can be used in distributed systems both in the processors and in the network. Both implementations are available as free software from their respective operating system web pages.

In summary, the FSF defines the basis of a new flexible scheduling strategy, and opens a new line of future research in design methodologies that help in translating application requirements into the best possible set of contracts to achieve the highest level of resource utilization and quality. Future work also includes enhanced on-line schedulability analysis and derivation methods for server parameters, as well as the development of high level transaction management middleware.

References

[1] L. Abeni and G. Buttazzo. “Integrating Multimedia Applications in Hard Real-Time Systems”. Proceedings of the IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998

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

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

[4] T.P. Baker. “Stack-Based Scheduling of Realtime Processes”, Journal of Real-Time Systems, Volume 3, Issue 1 (March 1991), pp. 67–99.

[5] BBN Technologies. Quality Objects (QuO). `http://quo.bbn.com/`

[6] G. Buttazzo, G. Lipari, L. Abeni. “Elastic Task Model For Adaptive Rate Control”, The 19th IEEE Systems Symposium (RTSS98), Madrid, Spain, December 2-4, 1998, pp 286-295.

[7] Z. Deng, J.W.S. Liu, and J. Sun. “A scheme for scheduling hard real-time applications in open system environment”. In Proceedings of the Ninth Euromicro Workshop on Real-Time Systems, 1997.

[8] Z. Deng and J.W.S. Liu. “Scheduling real-time applications in open environment”. In Proceedings of the IEEE Real-Time Systems Symposium, December 1997.

[9] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo, “A New Kernel Approach for Modular Real-Time systems Development”, Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems, Delft, The Netherlands, June 2001.

[10]ISO/IEC 9945-1:2003. Standard for Information Technology -Portable Operating System Interface (POSIX).

[11]IEEE Std. 1003.13-2003. Information Technology - Standardized Application Environment Profile- POSIX Realtime and Embedded Application Support (AEP). The Institute of Electrical and Electronics Engineers.

[12]M.B. Jones, D. Rosu, and M.C. Rosu. “CPU reservations and Time Constraints: efficient, Predictable scheduling of Independent Activities”. Proceedings of the 16th ACM Symposium on Operating systems Principles (SOSP), Saint.Malo, France, pp. 198-211, 1997.

[13]M.B. Jones, J. Regehr. “CPU Reservations and Time Constraints: Implementation Experience on Windows NT”. Proceedings of the Third USENIX Windows NT Symposium, Seattle, Washington, pp. 93-102, 1999.

[14]T.W. Kuo and C.H. Li. “Fixed-priority-driven open environment for real-time applications”. In Proceedings of the IEEE Real Systems Symposium, December 1999.

[15]C. Lee, R. Rajkumar, and C. Mercer. “Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach”. In Proceedings of Multimedia Japan, March 1996.

[16]G. Lipari. “Resource Reservation in Real-Time Systems”. PhD thesis, Scuola Superiore S.Anna, Pisa, Italy, 2000.

[17]G. Lipari and S. Baruah. “Efficient scheduling of multi-task applications in open systems”. In IEEE Proceedings of the 6th Real-Time Systems and Applications Symposium, June 2000.

[18]G. Lipari and S.K. Baruah “Greedy reclamation of unused bandwidth in constant bandwidth servers” IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems, Stockholm, Sweden, June 2000.

[19]G. Lipari and S. Baruah. “A hierarchical extension to the constant bandwidth server framework”. In Proceedings of the Real-Time Technology and Application Symposium, June 2001.

[20]G. Lipari, G. Lamastra and L. Abeni, “Task Synchronization in Reservation-Based Real-Time Systems”, IEEE Transactions on Computers, December 2004.

[21]J.W.S Liu, K.J. Lin, W.K. Shih, J.Y. Chung, A. Yu, and W. Zhao, “Algorithms for scheduling imprecise computations”, IEEE Computer, May 1999.

[22]J.W.S. Liu, K.J. Lin, W.K. Shih, R. Bettati, and J.Y. Chung, “Imprecise Computations”, IEEE Proceedings, vol. 82, p. 1-12, January 1994.

[23]C.W. Mercer, R. Rajkumar, and H. Tokuda. “Applying hard real-time technology to multimedia systems”. In Workshop on the Role of Real-Time in Multimedia/ Interactive Computing System, 1993.

[24]C.W. Mercer, S. Savage, and H. Tokuda. “Processor Capacity Reserves: Operating System Support for Multimedia Applications”. IEEE International Conference on Multimedia Computing and Systems, pp. 90-99, May 1994.

[25]A.K. Mok and X.A. Feng. “Towards compositionality in real-time resource partitioning based on regularity bounds”. In IEEE Proceeding of the Real-Time Systems Symposium, 2001.

[26]J. Nieh and M.S. Lam. “The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications”. Proceedings of the 16 ACM Symposium on Operating Systems Principles, St. Malo, France, October, 1997.

[27]R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. “Resource kernels: A resource-centric approach to real-time and multimedia systems”. In Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking, January 1998.

[28]D. Reed and R. F. (eds.). “Nemesis, the kernel overview”, May 1997. `http://sherry.ifi.unizh.ch/reed97nemesis.html`

[29]S. Saewong, R. Rajkumar, J. Lehoczky, and M.H. Klein. “Analysis of hierarchical fixed priority scheduling”. In Euromicro Conference on Real-Time Systems, June 2002.

[30]R. Schantz, J. Loyall, M. Atighetchi, and P. Pal. “Packaging Quality of Service Control Behaviors for Reuse”. ISORC 2002, The 5th IEEE International Symposium on Object-Oriented Real-time distributed Computing, April 29 - May 1, 2002, Washington, DC.

[31]B. Sprunt, L. Sha and J.P. Lehoczky. “Aperiodic Task Scheduling for Hard-Real-Time Systems”. The Journal of Real-Time Systems, Kluwer Academic Publishers, 1, pp. 27-60, 1989.

[32]N. Wang, D.C. Schmidt, M. Kircher, and K. Parameswaran. “Towards an Adaptive and Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications”. IEEE Distributed Systems On Line (Vol. 2, No. 5) May 2001.