IST-2001 34140

Scheduler integration definition report

Deliverable D-SI.1v3

Responsible: MDH

Gerhard Fohler, Tomas Lenvall, Radu Dobrin,
Alan Burns, Guillem Bernat, Ian Broster,
Michael Gonzalez Harbour, J. Javier Gutiérrez Garcia, Julio L. Medina Pasaje
Giuseppe Lipari, Paolo Gai

12th April 2005

Contents

1 Introduction **2**
1.1 Objectives of the project . . . 2

2 Application Requirements and Scheduler Flexibility **3**
2.1 Application Characteristics . . . 3
2.1.1 Scale . . . 3
2.1.2 Concurrency . . . 3
2.1.3 Processors . . . 3
2.1.4 Operating Systems and Scheduling . . . 4
2.1.5 Real-time Behaviour . . . 4
2.2 Application Requirements . . . 4
2.3 System Level Integration . . . 7

3 Overview of the software framework **7**

4 Service contract **10**
4.1 Core . . . 11
4.2 Shared objects . . . 12
4.3 Spare capacity . . . 13
4.4 Dynamic reclamation . . . 14
4.5 Hierarchical scheduling . . . 14
4.6 Distribution (core) . . . 15
4.7 Distribution (spare capacity) . . . 16
4.8 The Service Contract API . . . 16

5 Temporal profile of an application **16**

6 Global scheduling **18**

7 Integration and implementation **19**
7.1 Service based on EDF and GRUB . . . 21
7.2 Service based on FPS and SS . . . 22
7.3 Task synchronization . . . 24
7.4 Utilisation based admission . . . 24
7.4.1 Acceptance test and reclamation algorithm in MaRTE OS . . . 24
7.4.2 Acceptance test and reclamation algorithm in S.Ha.R.K OS . . . 25
7.5 Distribution . . . 25
7.6 Table-driven scheduling . . . 26

8 Summary and conclusions **27**

1 Introduction

In this document we present the software architecture of the systems that we support and have developed in the FIRST project.

After summarising the objectives of the project, Section 2 gives an abbreviated short list of application requirements addressed by the project. The complete list of requirements is listed in the requirementslist.pdf document, available for download in the AF directory on the FIRST web site. A more detailed discussion on the requirements will be presented in Deliverable D-AF1.v2.

We will then give an overview of the software framework in Section 3. In Section 4 we present the *service contract* specification, a key concept in our framework. In Section 5, we describe how to analyse the temporal behaviour of an application through the *temporal profile*. In Section 6, the underlying scheduling model is discussed. In Section 7 we describe in more details how the framework is supported by the operating system mechanisms. Section 8 presents our conclusions and Annex 1 describes the API reference manual.

1.1 Objectives of the project

We report here the objectives of the project for easy reference. They are discussed in more details in Deliverable D-EPrv.

1. To be able to compose different applications, each one with its own scheduler.
2. To be able to analyse an application/component/subsystem independently from the rest of the system.
3. To add robustness by providing protection from timing faults in a subsystem.
4. To be able to support and analyse diverse timing requirements.
5. To be able to support explicitly adaptive applications.
6. To be able to do the above things on distributed systems.
7. To provide schedulability analyses for the proposed algorithms.
8. Demonstrate the viability of the proposed solutions in real-case studies and operating systems.
9. To influence the relevant standards (POSIX, Ada) to support the primitives needed to implement the proposed solutions.

2 Application Requirements and Scheduler Flexibility

Key in FIRST is the concept of integration of complex applications with a variety of real-time requirements. In particular, we cover a broad range of realistic application requirements and real-time behaviour, including well-understood hard real-time behaviour as well as less well defined concepts of 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 following characteristics and requirements were obtained by performing a survey of industrial needs, and summarise the application characteristics FIRST is attempting to support:

2.1.1 Scale

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 guidance and control systems for avionics. The platform for current systems ranges from small 16MHz microcontroller systems to complex multiprocessor and heterogeneous operating systems. The software size varies considerably, from below 1,000 lines of code to over 1,000,000. Production times vary from as little as one man-month to tens of man-years.

2.1.2 Concurrency

Industrial experience reveals that small, hard real-time systems tend to have a constant number of processes, typically less than 10, of which about half may be considered ’hard’.

Larger systems may have thousands of concurrent processes, but the number of processes is not constant. For example, in a telecommunications system, when new connections are made, new processes are spawned as necessary. These processes still have real-time requirements.

It appears that as systems grow in size, the amount of critical or hard real-time code does not grow in proportion. The reason for this may be related to software growing at a faster rate than the hardware that it is running on; and it is the hardware that has a large influence of the timing requirements for the system.

Large systems with a high proportion of real-time constraints are not as common as it seems (much of the software is concerned with less time-critical functionality) although large control systems may consist of a large number of fairly independent real-time activities.

2.1.3 Processors

According to industrial experts, 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.

Current multiprocessor systems include a variety of heterogeneous cores including MIPS and TriMedia cores with dedicated function blocks and weakly programmable function blocks. Many industrial control systems are based on conventional industrial processors linked through field busses and ethernet networks.

2.1.4 Operating Systems and Scheduling

Fixed priority scheduling is common in industry. Also, various forms of deadline driven scheduling are currently being used in industrial projects. A few other flexible scheduling schemes are being used in novel situations, for example feedback based scheduling and schemes to manage overloads. Table-driven scheduling, is often used, although usually in the form of a cyclic executive. Table-driven scheduling in FIRST encompasses more than a simple cyclic executive; we regard table driven scheduling as an opportunity to use better, but more costly, scheduling algorithms to prepare better schedules than can be performed with simple algorithms like FP and EDF.

2.1.5 Real-time Behaviour

Industry has taken account of much real-time systems research: particularly, hard real-time theory is well used. Notions such as deadlines and periods are common and worst case response time analysis is used for analysis. Deadline-monotonic priority assignment for fixed priority scheduling is used extensively.

However, theory is ineffective in many systems. For example in larger systems where there are 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 utilisation. 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.

A bounded and known worst case execution time is taken for granted in almost all theoretic work. However, determining a suitable value for this is extremely difficult; previous approaches tend to be very pessimistic; worst case assumptions are made at every stage of analysis, resulting in a figure that although 'safe', is much higher than any worst case scenario achieved at run-time. Execution-time profiling, where a probability distribution of execution time is derived is a promising technology for FIRST, allowing an application to budget for (say) 95% of its expected executions (hence achieving timeliness in at least 95% of cases) and hoping to reclaim additional capacity at run-time.

2.2 Application Requirements

We will now provide a list of application requirements that have been derived from the application characteristics and were taken into account in the design of the FIRST scheduling framework.

These requirements are high-level concepts and the reader should be aware that inclusion in this list does not imply that a particular characteristic is supported directly by on-line mechanisms. Instead, particular application requirements are managed by a combination of off-line analysis and configuration, on-line acceptance testing and on-line scheduling. For example worst case execution time analysis would generally be performed off-line, worst case response time analysis may be performed on-line or off-line depending on the application.

1 Multiple Applications/Components. The ability of the system to support running multiple applications or components at the same time.

- **1.1** Composability. The ability to compose together separate applications or application components with their own timing requirements and schedulers.
- **1.2** Shared Resources. To be able to deal effectively with resources shared between applications or components. Specifically with mutually exclusive critical sections.
- **1.3** Distribution. To support distributed applications, specifically with static allocation of processes to nodes (process migration introduces complex timing effects).

2 Contract Support. Regardless of the underlying scheduling mechanism, the operating system provides a consistent API to the applications. The overall scheme is 'contract based' where the application and operating system agree on various real-time parameters such as required processing time, etc. Then, once accepted, the contract forms the basis of the application's access to the CPU.

- **2.1** Online acceptance test. It is a parameter of the system (not the contract) whether or not on-line acceptance tests are done. It is expected that suitable analysis is done either off-line (in which case, an on-line acceptance test may not be needed), on-line in the case of open systems or adaptive systems, or some combination of off-line analysis and on-line acceptance test.
- **2.2** Job Model. The contract model is based on two kinds of workloads: some behave as a 'stream of jobs', and others as indeterminate workloads. A stream is a (possible infinite) sequence of related jobs. A job is an instance of execution. Parameters such as how often jobs need doing, how long they take and guarantees of completion are part of the contract. For indeterminate workloads it may not possible to distinguish separate jobs, but a certain amount of system processing resources must be allocated to them anyway.

3 Periodicity. This concerns how often jobs 'arrive' for processing. The framework will support the following types of periodicity:

- **3.1** Periodic. Every period T, execution is required.
- **3.2** Sporadic. Execution is required with a minimum inter-arrival time.
- **3.3** Bursty. Execution is required in bursts; some bound can be placed on the nature of the bursts.
- **3.4** Continuous scale. It is useful for the period to change over a continuous scale, between two limits. It follows the elastic task model [5].
- **3.5** Discrete scale. It is useful for the period to change to any of a set of discrete values.
- **3.6** Unbounded. There is no bound on the arrival time of jobs, however it is useful and expected that there will be some other measure of how often jobs are required, such as a random distribution with known parameters.

4 Resource usage. This concerns how much processing time each job requires. The framework will support applications which may specify their requirements for resource usage in the following ways:

- **4.1** Minimum execution time required per job invocation. This is to be guaranteed by the system.
- **4.2** Continuous scale. It is useful for the budget of CPU time allocated to vary over a continuous scale between two limits. This follows the elastic task model.

4.3 Discrete scale. It is useful for the budget of CPU time allocated to change to any of a set of discrete values.

4.4 Execution time variability. The execution time of most processes has a wide natural variation. The framework recognises this and provides support. In particular, a combination of off-line analysis and slack reclamation may be used.

4.5 Unlimited. The process will happily use any available CPU time available. Weights and Importance metrics are used to schedule competing processes.

5 Performance and Guarantees. This section relates to the general area of supporting deadlines, quality of service and application-specific control of resource usage.

5.1 Resources for stream-based workloads. The job is informed of how much time it has been allocated in this instance. Once informed, this is a guaranteed minimum that the job will receive. The job may indicate at run-time that no further useful processing may be done, so that any unused capacity can be dynamically reassigned.

5.2 Resources for indeterminate workloads. The processes in this workload may usefully use any extra time that it has been allocated up to a stated maximum. It should be possible to know how much processing time there is left (at least) in the current server period.

5.3 Importance/weight. A two tier hierarchy is supported for allocation of resources. This is to accommodate a structured best-effort approach to maintaining an application level quality of service.

5.4 Importance. This is the highest layer on the hierarchy. A job of high importance is more important than an infinite number of jobs of lower importance.

5.5 Weight. Within an importance level, jobs may compete for spare resources (above any minimum guarantee) by specifying a weight. Weight may be adjusted.

5.6 Deadline Guarantees. The framework will support deadline-based guarantees. Note that supporting deadlines may involve off-line work, rather than on-line acceptance tests. Although the current framework only supports hard deadlines, it is also possible to make analysis on probabilistic guarantees and N-in-M guarantees, by performing off-line analysis based on the service contract model.

6 Change Management. This concerns the ability of an application to change its requirements at runtime or join a system.

6.1 Renegotiation. An application may renegotiate its contract. Note that in some systems, it may not be possible for the operating system and the application to agree on a new contract.

6.2 Negotiation time. The time taken to negotiate a contract must be adjustable and acceptably small.

6.3 Open System. A system may be 'open' with respect to applications. This means that unknown applications may attempt to join the system and negotiate a contract.

6.4 Closed System. A system may be 'closed' with respect to applications. This means that only known applications may join the system. A system is still closed if not all characteristics are known about the application.

7 Dependability. This section considers the temporal semantics in the case of contracts being broken. This applies particularly to applications breaking their contract.

7.1 Robustness and Isolation. The operating system will prevent applications from suffering due to other applications overrunning for example. A system of budgets is used to enforce this.

2.3 System Level Integration

Systems consisting of applications which change their timing behaviour over their lifetime are common. The timing behaviours 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 behaviour of applications, even if the application attempts (by accident or maliciously) to exceed its contracted behaviour.

3 Overview of the software framework

The main objective of this project is to be able to compose different applications, each one with its own scheduler, in the same system (Objective 1). Thus, the basic unit of composable object in our framework is the *application*. An application is defined as a set of tasks with a scheduler. Since this definition is quite general, and since our framework can eventually be used for component-based design of real-time systems, we sometimes will use the terms *component* and *subsystem* instead of *application*, which are also more appropriate for distributed systems. An application can also consist of only one task: in that case the application scheduler can be very light. An application is viewed as a stream of essentially repetitive work. Some applications will be strictly periodic but others will have more elastic parameters.

Since different applications in the same system can have different schedulers, we will adopt a hierarchical scheduling structure. A *global scheduler* selects which application is executed at each time, and the *local application's scheduler* will select which task is executed. In principle, it is possible to compose schedulers at any level of the hierarchy. Thus, an application can be composed of many sub-applications, each one with its own scheduler. However, in this document we will address only two-level hierarchies. The two-level hierarchy appears to be enough for most application domains. In Figure 1, an example of a hierarchical system that consists of three applications is shown.

As discussed in the previous section, the application's tasks can have diverse timing requirements. Therefore, by using this hierarchical structure, each application can be developed using the most appropriate scheduling strategy that best meets the application requirements (Objectives 4 and 5).

Although in general it is possible to use any scheduling algorithm as the global scheduler, we will build our hierarchical framework on server-based algorithms. All servers have the general property of protecting the processing resource so that applications do not execute for more than has been agreed. Server capacity is replenished following a process defined for that server algorithm - one of the main differences between servers is their replenishment algorithm. Note that servers can be built on fixed priority or EDF schedulers.

Each application is assigned one or more servers, and each server is assigned a fraction of the processor bandwidth. These algorithms, also referred to as *resource reservation algorithms*, provide temporal pro-

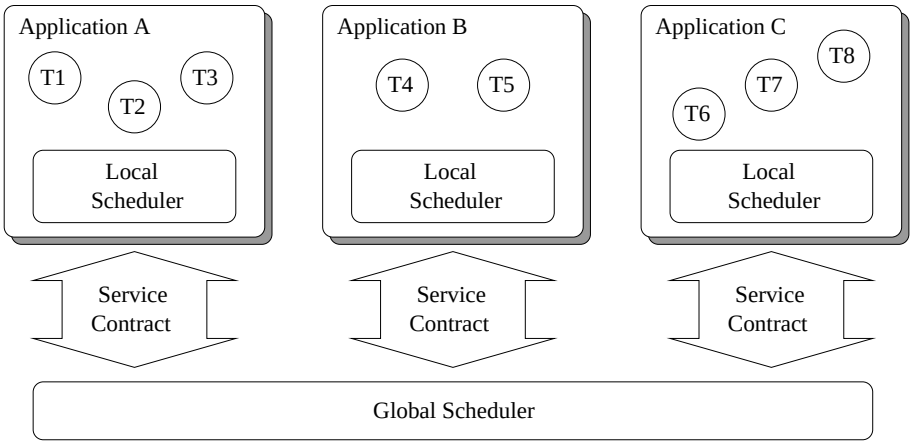

Figure 1: Hierarchy of schedulers.

tection between applications. Each application executes as if it were on a *slower virtual processor*, and therefore may be analysed independently from the other applications in the system. This approach is compliant with Objective 2 (independent analysis) and Objective 3 (temporal protection). Where an application uses two or more servers then it is devolving some of its local scheduling behaviour to the global scheduler. In the extreme each task of an application could execute on its own server on the global scheduler.

However, we do not want to restrict our analysis to a specific server algorithm, nor to any specific global scheduling algorithm. Therefore, we identified a class of properties that can be provided by most of the server algorithms presented in the literature. In our framework, the application and the global scheduler communicate by means of the *service contract*. Each application *proposes* a contract to the system with certain parameters. If the contract can be fulfilled by the global scheduling strategy, then the application is admitted into the system.

Many applications are adaptive, i.e. they change their requirements and their behaviour depending on the amount of available resources. Many other applications change their requirements at run-time, depending on their input data or on some state variable. In order to support these kinds of applications (Objective 4 and 5), the service contract is flexible and the server parameters can be changed on-line depending on the applications requirements. Also, the application will be informed on the minimum amount of resources available, so that it can adjust its own internal behaviour. The service contract will be discussed in more details in Section 4.

Our framework explicitly addresses two kinds of system. In an *open system*, applications can dynamically arrive in the system and ask for execution. In this case, the application goes through an *on-line admission* phase. In a *static system* all applications and their arrival times are known during design phase. Therefore, the design of the system includes an *integration* or *configuration phase*, where a global schedulability analysis is performed to check if all applications can coexist in the system. In an open system applications will complete and leave the system thereby freeing up resources for future applications.

The design flow of an application is shown in Figure 2. It consists of five steps:

Application Design In this phase, the application is designed and a local scheduling algorithm is chosen for it. Any scheduling algorithm can be used, so the application designer can choose the algorithm that best fulfils the application requirements. The user should design the application without considering the presence of other applications in the system. However, some care should be taken in

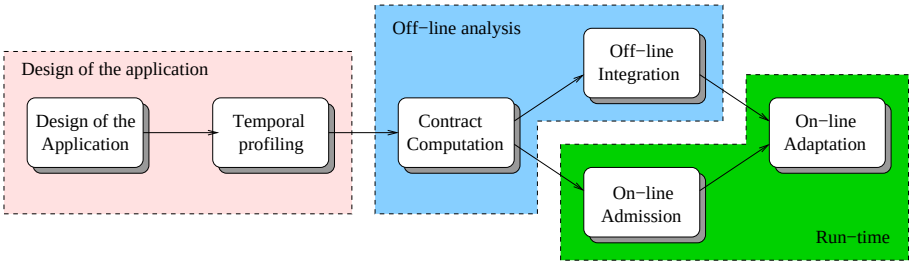

Figure 2: Design flow of an application.

deciding which resources are *private* to this application and which might be *shared* with other applications. The interaction between applications is only considered during the integration phase (or on-line admission).

Temporal Profiling The temporal behaviour of the application is analysed and described using a precise mathematical formalism. This phase is used to simplify the integration process and should be performed with the help of an appropriate analysis tool. In particular, the output of this phase is a temporal profile, i.e. a mathematical description of the temporal characteristics of the application, including its "interface" with the other applications. The temporal profile is independent of the underlying server mechanism that will be used for executing the application. It can be used as a synthetic description of the temporal characteristics of the application, and it is very useful for re-use. For relatively straightforward applications temporal profiling will not be needed as it will be possible to move directly from Application Design to Contract Computation.

Contract Computation This phase is used to compute the characteristics of the contract that the application should be assigned in order to fulfil its requirements. Should the application involve complex constraints, which cannot be handled efficiently at run-time directly, complexity reduction methods can be applied in that phase which translate these constraints for efficient runtime use, however, in a suboptimal way.

The contract is only negotiated (for an open system) when the application arrives (not on each invocation of its stream of work). However renegotiation will be possible when application characteristics change.

Off-line Integration / On-line admission Once the contracts for all applications have been computed, we can integrate all applications in the same system. This integration phase can be done off-line or on-line and it consists of a schedulability analysis of the servers that implement the service contracts. If it is done off-line, this schedulability analysis can be complex, in order to optimise the system resources. If the integration phase is not successful, the system designer is informed and it is necessary to go back and modify some of the design choices. In the on-line case, integration is done by an admission test. Since this is done on-line, it must be simple and fast. Therefore, it will initially be based on an utilisation test. If the admission test is not successful, again the system is informed and can take some relevant action.

On-line Adaptation The server parameters can be modified at run-time, depending on many factors as the amount of free resources, the actual requirements of the application, etc. However, the amount of allowed modifications should never compromise the service contract. In order to support adaptive

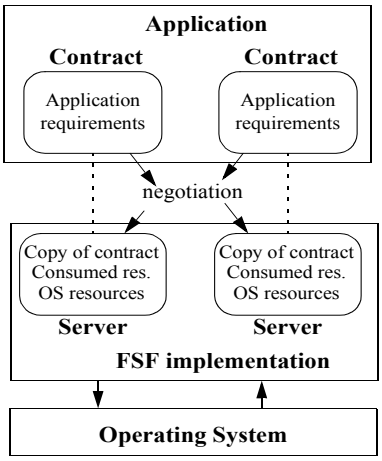

Figure 3: Contract negotiation process

applications, every application is informed of the amount of available resources for the next instance. It can then use this information to adapt its requirements to the available resource. This is particularly useful for anytime algorithms and imprecise computation.

4 Service contract

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, 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 3 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 mod-

ule addressing specific application requirements. The minimum 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.

4.1 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, 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.

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 servers 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

Table 1: Core attributes

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.

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 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. Background contracts are defined by specifying a $budget\_min$ and period max of zero

With the described services, the core module provides support for the basic timing requirements of real-time applications, including the ability to reserve and have 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 [25], or EDF with constant bandwidth servers underneath [1].

4.2 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, 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 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.

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

Table 2: Shared object attributes

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.

4.3 Spare capacity

Many applications have requirements for flexibility regarding 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.

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

Table 3: Spare capacity attributes

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 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: 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: 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.

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

4.5 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 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. In this way, 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. For instance, they have a priority for fixed

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

Table 4: Hierarchical 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 5: Distributed core attributes

priorities, or a relative deadline for EDF.

4.6 Distribution (core)

FSF is designed to support applications with requirements for distribution. 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 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.

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.

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.

4.7 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 purpose, 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 to return spare capacity that cannot be used due to restrictions in other servers of a distributed transaction.

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

Table 6: Distributed spare capacity

4.8 The Service Contract API

We refer to Appendix 1 for a complete description of the Service Contract API.

5 Temporal profile of an application

Describing the temporal characteristics of an application can be difficult, especially if the application shows complex timing constraints and custom scheduling algorithms. We are seeking a uniform method for describing the temporal requirements of any application that abstracts most of the internal details. We call this abstract description the *application's temporal profile*.

Temporal profiles are translated into service contracts, which are then subjected to global scheduling servers. Temporal profiles and contracts are system independent, while the servers are system specific. As

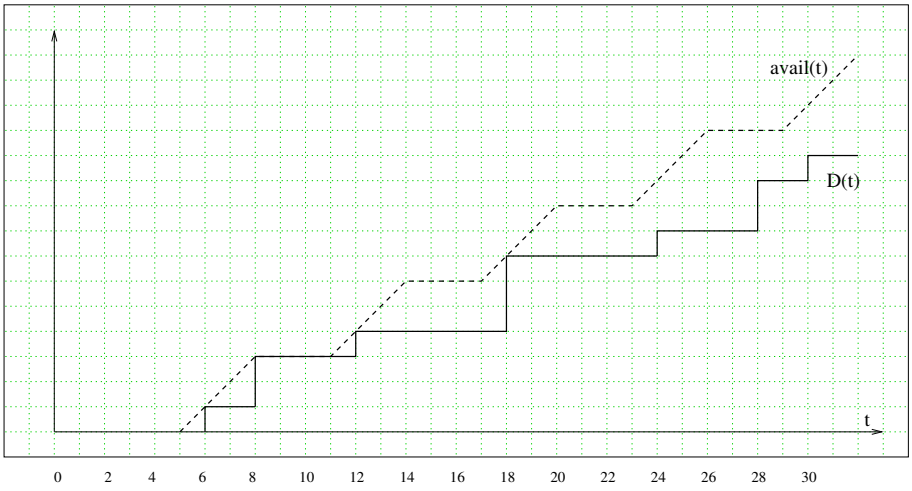

Figure 4: Demand bound function and server's available time for the example's application.

such, the profiles define the boundary between system independent application description and their system specific execution. Via temporal profiles, applications can be reused or executed on different platforms — with the same temporal behaviour — only by providing appropriate global scheduling servers.

Temporal profiles are important because there is not a unique service contract that fulfills the application requirements. Consider the following example.

Example. Consider an application $\mathcal{A}_1$ consisting of two periodic tasks: τ_1 with $C_1 = 2\,\mathrm{ms}$, $D_1 = 8\,\mathrm{ms}$ and $T_1 = 10\,\mathrm{ms}$; and τ_2 with $C_2\,\mathrm{ms}$ and $D_2 = T_2 = 6$, with an EDF local scheduler. A necessary and sufficient condition for the application to be schedulable is that in every interval $[t_1, t_1 + \Delta t]$ the amount of available execution time is not less than the demand of the application:

$$\forall t_1, \Delta t \;\; \mathrm{avail}(\mathrm{t_1}, \mathrm{t_1} + \Delta \mathrm{t}) \geq \mathrm{D}(\mathrm{t_1}, \mathrm{t_1} + \Delta \mathrm{t}) = \sum_{\mathrm{a_{i,j}} \geq \mathrm{t_1},\ \mathrm{d_{i,j}} \leq \mathrm{t_1} + \Delta \mathrm{t}} \mathrm{c_{i,j}}$$

where $a_{i,j}$, $d_{i,j}$ and $c_{i,j}$ are the arrival times, the deadlines and the computation times of all the application's jobs.

The demand of the application is shown in Figure 4 as a thick line. In the same figure, is plotted the available time provided by a periodic server with $C_s = 3$, $D_s = 5$ $T_s = 6$ with a dashed line. Many other servers can successfully support the above application. For example, the same application can be supported by a periodic server with $C_s = 4$, $D_s = 4$ and $T_s = 9$. Note that all that is necessary to analyse the application and compute the server parameters is the demand function.

When considering applications with other local schedulers, the profile function depends also on the adopted scheduling algorithm, but has a similar form. Preliminary results on the temporal profile of an application with a local fixed priority scheduler can be found in [16].

Therefore, we propose to use a similar formalism for describing the application. The temporal profile consists of one or more demand bound functions that describe the requirements of the application.

Temporal profiles can be used in a *reactive* manner, i.e., by analysing the temporal behaviour of application and scheduler, as well as *proactive*, i.e., by taking a profile as input and ensuring that the local scheduler will keep executions within the profile.

The temporal behaviour of the application is analysed and described using a precise mathematical formalism. Applications with constraints following a basic period, deadline model will not require special temporal profiling, as service contracts can be derived directly from the application parameters. More complex applications will need more elaborate analysis methods, which will be carried out by special tools.

6 Global scheduling

In standard single processor systems with standard scheduling schemes, a single scheduler decides which task to execute at what time, from of a set of those ready: all tasks compete for the CPU and are arbitrated by a single scheduler, who has complete control over the CPU. Our framework enables the coexistence of a number of applications and their schedulers, while maintaining for each application the view that it is executing alone on the CPU, with a certain approximation. The approximation comes from the fact that the global scheduler may introduce some (bounded) delay, which depends on the contract parameters.

Thus, each application can choose the scheduler best suited for its timing requirements. The presence of other applications and schedulers is reflected as the CPU appearing slower, and having additional predictable delays.

To maintain the view of exclusive CPU control to application, the following has to be ensured:

(i) enough CPU resources are available to each application;

(ii) applications are protected from each other.

This is provided by the global scheduler which (i) provides sufficient access to the CPU for the assigned servers to meet the temporal profile of an application and (ii) enforces temporal profiles, e.g., protecting from overruns.

The global schedulers used in our framework will be based on a class of scheduling algorithms called *servers*. These algorithms were initially proposed for minimising the response time of aperiodic tasks in hard real-time periodic system. In these models, one server in the system is in charge of executing one or more aperiodic tasks. The server is characterised by a *budget* (or capacity), C, and by a *period*, T. Intuitively, the server guarantees that the served tasks are allowed to execute C units of time every interval of T units of time.

Server algorithms exist both for fixed priority and dynamic priority algorithms, and can in theory even be implemented on time triggered systems. Examples of servers for fixed priority scheduling are the Polling Server, the Deferrable Server and the Sporadic Server [20], and the Processor Capacity Reserve [17, 18]. Examples of servers for earliest deadline first are the Dynamic Sporadic Server, the Total Bandwidth Server [21] and the Constant Bandwidth Server [3]. The Total Bandwidth server has also been combined with the Slot Shifting algorithm by Isovic and Fohler [12] for servicing aperiodic tasks in Table Driven Schedules.

All these servers have the general property of protecting the processing resource so that tasks do not execute for more than has been agreed. This property, referred as *temporal protection*, is considered very important so that a misbehaving task does not affect the guarantees on the other tasks in the system. Therefore the server approach has been extended to the entire system. In the *resource reservation framework* [18] each task is assigned a server. Hard real-time tasks are assigned a server with capacity not lower than their worst case execution time and period not greater than their minimum inter-arrival time. Soft real-time tasks are assigned a server with capacity based on some other measure such as a probabilistic execution time profile.

Recently, many server algorithms have been extended to hierarchical scheduling systems. Deng and Liu [8, 7] proposed to use the total bandwidth server of Spuri [21] to serve an application with its own local

scheduler. Saesong et. al. [19] extended the resource reservation framework so that each server can schedule applications with a local fixed priority scheduler, and propose a schedulability test for the application based on response time analysis. In the first phase of this project (see deliverable SI.4v1 and [16]), Lipari and Bini proposed a schedulability analysis for applications with a local fixed priority scheduler that is independent of the server algorithm. This analysis can also be used for computing the server parameters that fulfill the application requirements.

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. A set of properties which are supported by most current server algorithms has been identified, detailed in section 4. So instead of using parameters of a specific server algorithm, the application defines its need 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.

In this project, we will not restrict our analysis to a specific server algorithm, nor to any specific global scheduling algorithm. Two different implementations of the project framework will be provided. In the MaRTE operating system, a global scheduling algorithm based on fixed priorities and a modified Sporadic Server algorithm is used, whereas in the Shark operating system the global scheduler will be based on earliest deadline first scheduling and on a modified Constant Bandwidth Server. Both algorithms will provide the same programming interface, based on the service contract (see Section 4).

One issue with servers is that it cannot be determined exactly *when* the application will receive execution, because it depends also on the presence of other servers in the system. Therefore, it may be possible that an application receives all the needed computation time at the beginning of the server interval, or it may happen that it receives all computation time at the end of the interval, or that the execution is scattered along the interval. Some scheduling schemes, including table driven approaches, require task executions within specific, short intervals. We introduce server deadlines to enable more precise specification of when the actual executions will take place.

It is also important to be able to compute the service contract parameters, listed in Table 4, that fulfill the application requirements. In the third phase of this project, we have investigated the problem of computing the contract parameters, by extending the approach presented in [16] and by off-line analysis for complex constraints. The resulting techniques will be presented in D-SI.4v3.

7 Integration and implementation

This section presents in more detail how the FSF framework, described in the previous sections, is implemented in the two operating systems used in the FIRST project, MaRTE OS and Shark OS.

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 5 shows the main elements of the API, decomposed into the different modules described in Section 4. The API is provided for both C and Ada and is available in Appendix 1.

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

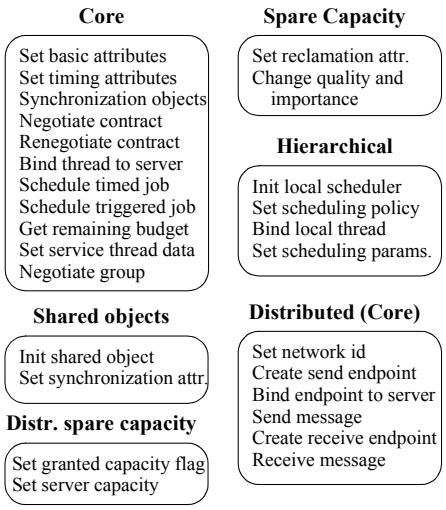

Figure 5: Main elements of the FSF API.

therefore each FSF implementation would have to be tailored to a specific OS. To overcome this difficulty in the future, an API will be defined to specify services that allow an OS to provide application-level scheduling support in a uniform way [5], and we will start the process to request inclusion of this API into the real-time POSIX standard [2].

As a proof of concepts, the FSF services will be implemented in two real-time kernels, MaRTE OS [4] and S.Ha.R.K [11], both of which follow the POSIX minimum real-time profile [1]. These implementations will show that even with very different scheduling strategies it is possible to provide a portable FSF API.

In MaRTE OS the FSF services will be 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 will install a secondary application defined scheduler that contains the FSF servers and manages the negotiated contracts. The modules that will be implemented in MaRTE OS are the core (using fixed priorities and sporadic servers [20]), shared objects (with the SRP [6]), spare capacity sharing, and the core distributed module.

Shark will not implement the application-defined scheduling API, but instead its kernel will be designed with a modular structure to allow the coexistence and interplay of different scheduling algorithms. The underlying scheduling algorithm will be EDF. The server algorithm, used for implementing the service contract, will be the GRUB algorithm (Greedy Reclamation of Unused Bandwidth [14]). This algorithm automatically adds dynamic reclamation to the CBS (Constant Bandwidth Server [3]). The modules that will be implemented in Shark are the core (using EDF and constant bandwidth servers), spare capacity sharing (restricted to deadlines equal to periods), dynamic reclamation, and hierarchical scheduling; we will implement the shared objects module using Bandwidth Inheritance (BWI) [15], which extends the priority inheritance protocol to server based scheduling.

In both MaRTE OS and Shark the admission control algorithm and the spare capacity calculation will be implemented through a service thread that consumes part of the system resources in a controlled way. The budget and period of the service thread will be able to 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.

7.1 Service based on EDF and GRUB

As global scheduling algorithm, we selected the EDF togheter with a resource reservation algorithm, the GRUB algorithm [5]. This algorithm is very similar to the CBS algorithm of Abeni and Buttazzo [1], and in addition it automatically performs the dynamic reclaimation of the bandwidth. In S.Ha.R.K. it is possible to configure the total amount of available bandwidth for the framework. In most of the experiments, this bandwidth has been limited to 80% of the total system bandwidth. However, it is possible to change this fraction.

S.Ha.R.K. permits to easily modify the scheduler and to mix different scheduling policies. The basic mechanisms to implement a new scheduling policy is the scheduling module. A scheduling module resemble an object in a object oriented language: it has internal data structures, a set of ”private” functions and a set of ”public” functions that implement the interface with the S.Ha.R.K. generic scheduling mechanism.

Therefore, to implement the basic structure of the FSF in S.Ha.R.K. we will implement a set of scheduling modules. Particular attention will be devoted to the hierarchical scheduling structure designed in the FSF.

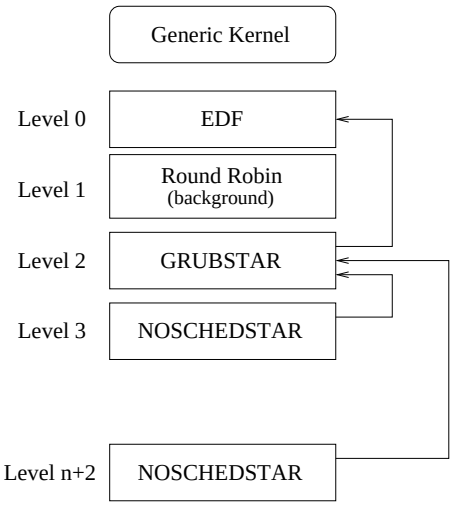

Figure 6: Organisation of the modules in S.Ha.R.K.

In Figure 6 we show the basic structure of the S.Ha.R.K. scheduling modules used in FIRST. The Generic Kernel performs generic operations like the dispatching and suspension of a task. It also implements the interface for all system calls. The actual scheduling is done in the modules that are organized in levels. Modules in lower levels have higher priority.

Tasks are assigned to scheduling modules. When a system call is invoked by a task, the Generic Kernel identifies which module the task belongs to, and invokes the appropriate operation on that module.

In the structure we designed for the FSF, the module in the lower level is an EDF scheduler. Only if the EDF has no task to schedule, the module in level 1 (a simple Round Robin scheduler) is asked for something to be executed. This module contains the dummy task and the main() function. Thus the main and the dummy run in background.

Modules can ”insert” tasks in other modules. This is the mechanism used to implement the server algorithm. Module GRUBSTAR handles the descriptors of all servers in the system, and the corresponding tasks. In Figure 1, the arrow from the GRUBSTAR module to the EDF module means that the GRUBSTAR

inserts one task per each server in the EDF module using the deadline of the server.

Finally, each task is assigned a different module. In Figure 6, we present the structure when no hierarchical server is involved: therefore, each task is assigned different NOSCHEDSTAR module (the name comes from the fact that this module does nothing becuase it has no specific local scheduler and can hanlde only one task).

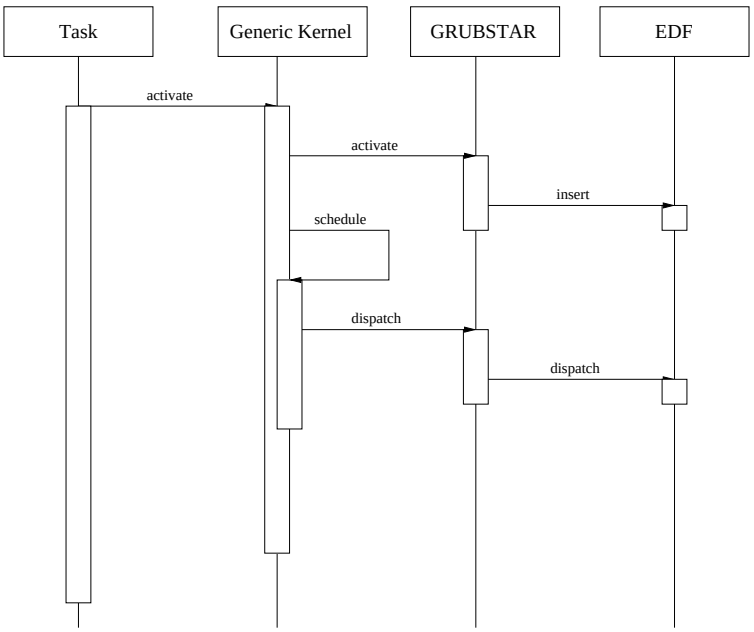

Figure 7: Sequences of messages exchanged between the S.Ha.R.K scheduling modules when a task is activated.

Now we describe the actual sequence of function calls that is performed when a task becomes active. The sequence of messages is shown in Figure 7 as a UML sequence diagram.

When a task corresponding to a server becomes active, it invokes the activate() function of the Generic Kernel. This, in turns, calls the insert function of the NOSCHEDSTAR module, which in turns simply redirects it to the insert function of GRUBSTAR module. The module computes the budget and the deadline for the task, and ”inserts” the task in the EDF module. If the task is the earliest deadline task, this insertion triggers a ”re-schedule” in the Generic Kernel, which invokes a dispatch on the involved modules. As a consequence, module GRUBSTAR activates a timer to expire at the budget expiration of the task. The situation described above is depicted in Figure 7.

Similar situations happen for other scheduling decisions, like preemption or task suspension. For brevity, we do not report here the complete description of the S.Ha.R.K.s internal scheduling mechanism for FSF.

Refer to the OS2.v3 deliverable for more detail of the S.Ha.R.K implementation.

7.2 Service based on FPS and SS

MaRTE OS Implementation of FSF The main architecture of the solution implemented is shown in Figure 8 by means of a class diagram. In this diagram two separate memory areas are considered, one for the application threads, labelled in 8 as User Memory Space and the other one for the application scheduler related

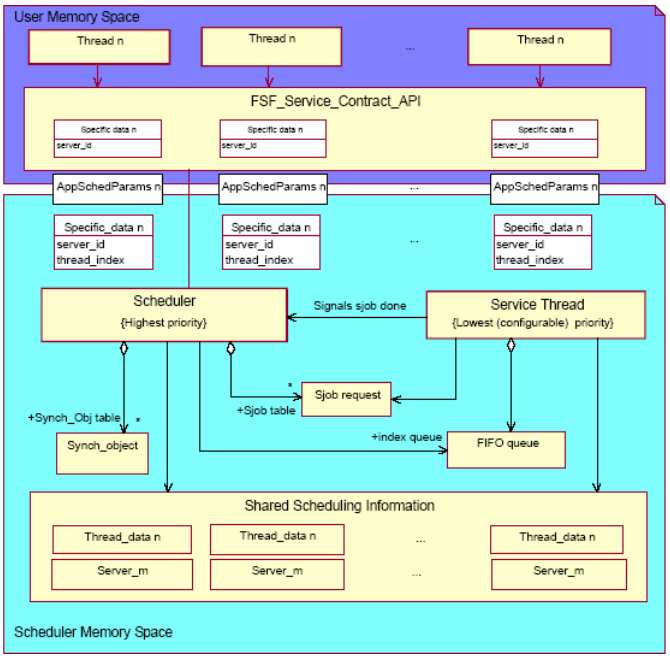

Figure 8: Main architecture of the implementation in MaRTE OS

objects, called Scheduler Memory Space. This separation is caused by the implementation-independent nature of the application scheduling interface, which must allow for the application scheduler to be eventually implemented in the kernel memory space. The link between the user threads and the underlying scheduler is realized in the implementation of the $FSF\_Service\_Contract\_API$. This API is common to both target platforms of the project: S.Ha.R.K OS and MaRTE OS. As it is shown in 8, all the scheduling responsibilities are solved by two internal threads, the scheduler and the service thread. The scheduler runs at a priority higher than any FSF scheduled task and consequently its effect over the schedulability of the application must be considered as an additional preemption term. The service thread instead is scheduled by the underlying operating system as a normal sporadic server with a budget and period assigned.

In the MaRTE OS implementation, servers behave basically as the POSIX sporadic servers. Depending on whether their workload is indeterminate or bounded they may go to the background or not, respectively. Figure 9 shows the state chart of an FSF server; it is only when the server is in the active state that its budget is consumed. If the server has a thread bound to it, it may only execute if the server is in the active state or if there is an available background time slice.

The detailed architecture of this scheduling mechanism, as well as the scheduling information data structures used to implement it, the detailed design of the methods provided by the $FSF\_Service\_Contract\_API$, the way in which the reclamation algorithm and the acceptance test are implemented, the implementation of the shared objects management, the organization of the code and the way of using it in an end-user application, and the detailed description of the Distributed services module and its implementation are described in document D-OS1v3.

7.3 Task synchronization

Shared objects??

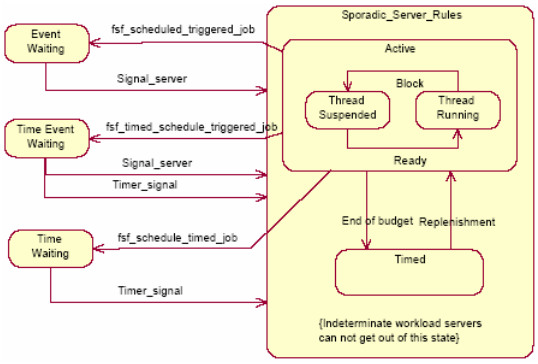

Figure 9: State chart of an FSF server in MaRTE OS

7.4 Utilisation based admission

7.4.1 Acceptance test and reclamation algorithm in MaRTE OS

In order to calculate the schedulability of the admitted servers, a ”pessimistic” though simple utilization bounds test algorithm has been implemented, that can handle tasks with synchronization and deadlines smaller than or equal to the task periods [13]. The test is applied each time a contract is negotiated and maintains or recalculates as necessary a number of accumulative values, which are updated whenever the test is made and the contract admitted, after a successful renegotiation or when a server is to be cancelled. Among these values we calculate the ”utilization slack” of each server; the minimum of these slacks is the utilization that will be safely shared by the reclamation algorithm. The terms that are used in the description of the algorithms that follows, that are also held in a data structure associated to each server, are:

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

Other terms used:

- $Servers\_set$ - the set of servers already admitted, it can be empty
- *i* - indicates one server in the set if it is not empty
- $Critical\_sections\_set$ - is the set of critical sections of a certain server
- *s* - a critical section in a set

- C - the worst case execution time in a certain critical section
- $U\_available$ - the total available utilization (the minimum of all the slacks)

The formulation of the modified utilization bound test used is the next:
For the analysis of the server i consider all the servers with priority equal to or higher than P_i and group them into the following two sets:

- *H1,* singly preemptive tasks: servers with periods ¿= T_i
- *Hn,* multiply preemptive tasks: servers with periods ¡ T_i

The effective utilization for server i is:

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

The utilization bound for the server i is:

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

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

$$U_{bound}(n, \Delta_i) = \begin{cases} n((2\Delta_i)^{1/n} - 1) + 1 - \Delta_i & , 0.5 \leq \Delta_i \leq 1 \\ 2 & , 0 \leq \Delta_i \leq 0.5 \end{cases}$$

And the schedulability test is:

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

The dynamic reclamation module is not implemented in the MaRTE OS version of FSF.

7.4.2 Acceptance test and reclamation algorithm in S.Ha.R.K OS

description from SSSA

7.5 Distribution

In distributed systems it is possible to perform a global schedulability analysis; for example, response time analysis techniques exist both for fixed priority [24, 23] and EDF [22] scheduling. However, we do not want to impose a global scheduling strategy at the underlying schedulers which would be too complex to implement. Rather, the application will be partitioned, with different parts executing in each node, and artificially assign timing requirements to ensure that the global timing requirements can be met. This allows the analyses to be performed independently for each node.

In this context the scheduling framework described in this document will be used for each of the processing nodes of the system, and also for scheduling the network. The implementation of the framework in the network is difficult because the scheduling decisions themselves cannot be made by the network, and must be implemented by the network communication drivers and executed by the processor nodes. Based

on the assumption that in distributed systems the network is usually a scarce resource, while the processors have more capacity, we will design a network scheduler in which the processors compute the scheduling decisions for the network, using shared information regarding the network contracts, and minimizing the amount of information sent through the network. To avoid the possibility of two processors inconsistently negotiating in parallel, a negotiation token will be created and circulated through the different processors, until one of them has to perform a negotiation, in which case it will hold the token throughout the negotiation process. Once the negotiation is completed the new state of the network will be propagated to all the other nodes to update their shared network server information, and then the token will be released.

To implement the network scheduler it is necessary to make sure that the scheduling information can be distributed to all the nodes. This can be accomplished by boadcasting the changes, or in the case of a token passing network, by sending the information with the token.

In summary, the same scheduling framework will be used in the network and in the processing nodes. The overall system and timing requirements will be distributed among specific contracts for each application part executing in each node and in the network. The minimum guarantees in these contracts must ensure that the requirements are met. The underlying scheduler in each node will work independently of the others, and independently of the network scheduler. The computations required for the network scheduler will be executed in the node requesting negotiation, in a mutually exclusive way, and will be synchronized with all the other nodes in the network.

7.6 Table-driven scheduling

Complex constraints, such as stemming from distribution, precedence, control or media applications, cannot be handled directly by online algorithms, as many pose NP hard problems. However, these constraints are easily solved in table-driven scheduling by using existing off-line schedulers to create a scheduling table that specifies the task executions such that the constraints are fulfilled. The table is usually constructed for an application that has exclusive access to all resources in the system. However, if several applications coexist simultaneously, constructing the table by ignoring the utilization demanded by the rest of the applications will introduce the risk that the table-driven application will monopolize the resources such that the rest of the applications will starve.

Therefore, in order for the table-driven application to coexist with the rest of the applications, the table construction must take into account the demands of the rest of the applications. However, in the general case, it is difficult to know and include the requirements of the other applications in the table construction.

In FIRST the application demands are specified by their service contracts. Thus, a modified offline scheduler constructs a table based on the contracts, rather than considering the table-driven application constraints alone. The table is a list of tasks. For each task we need a list of non overlapping target windows, i.e., temporal windows in which the tasks have to execute and complete. The local scheduler in FSF (table-driven) decides which task to execute based on the table. The API for specifying the table is defined in Appendix 1, the API reference manual.

In addition, possibilities exist to transform off-line tables to attributes suitable for FPS scheduling [9]. In that case, an off-line schedule first is constructed for a set of tasks to fulfill their complex constraints . Then, by analyzing the off-line schedule, we derive FPS attributes, i.e., priorities, offsets, deadlines, such that the tasks, when scheduled by FPS, will execute flexibly, while fulfilling the same original complex constraints. The method generates optimal solutions with an ILP-based algorithm. It does so by deriving priority inequalities, which are then resolved by integer linear programming.

Another possibility is to transform off-line schedules to attributes for EDF scheduling [10]. In a first step, the offline scheduler resolves complex constraints and reduces their complexity. The constructed schedule

is translated into independent tasks on single nodes with start times and deadline constraints only. These are then executed using earliest deadline first scheduling algorithm at runtime.

8 Summary and conclusions

This document presented the software architecture that will be investigated and supported in the FIRST project. Specifically, it has described the mechanisms and API that will be provided to the programmer to support the application requirements listed in Section 2.

This document will be an input for Workpackage 3, Operating System Support.

Schedulability analyses and techniques for the proposed methodologies will be presented in Deliverable D-SI.4v3.

References

[1] IEEE Std. 1003.13-2003. Information Technology - Standardised Application Environment Profile - POSIX Realtime and Embedded Applications Support (AEP).

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

[3] L. Abeni and G.Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, december 1998. IEEE.

[4] M. Aldea and M. Gonzlez. Marte os: An ada kernel for real-time embedded applications. In *Proceedings of the International Conference on Reliable Software Technologies*, Leuven, Belgium, May 2001.

[5] M. Aldea and M. Gonzlez. Posix-compatible application-defined scheduling in marte os. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, Vienna, Austria, June 2002.

[6] T.P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3, 1991.

[7] Z. Deng and J. W. S. Liu. Scheduling real-time applications in open envirovment. In *Proceedings of the IEEE Real-Time Systems Symposium*, San Francisco, December 1997.

[8] 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* 9^{th} *Euromicro Workshop on Real-Time Systems*, 1997.

[9] R. Dobrin, P. Puschner, and G. Fohler. Translating Off-line Schedules into Task Attributes for Fixed Priority Scheduling. In *Proceedings of Real-Time Systems Symposium*, December 2001.

[10] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proc. 16th Real-time Systems Symposium*, Pisa, Italy, 1995.

[11] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, Delft, Netherlands, June 2001.

[12] D. Isovic and G. Fohler. Resource aware mpeg-2 using real-time scheduling. In *Technical report, Malardalen Real-Time Research Centre*, Vasteras, Sweden, 2003.

[13] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Gonzlez Harbour. *A practitioners Handbook for Real-Time Analysis*. Kluwer Academic Publishers, 1993.

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

[15] G. Lipari, G. Lamastra, and L. Abeni. Task synchronisation in reservation-based real-time systems. *IEEE Transactions on Computers*, December 2004.

[16] Giuseppe Lipari and Enrico Bini. Resource partitioning among real-time applications. In *to appear on Proceedings of Euromicro conference on Real-Time Systems*, Porto (PT), July 2003.

[17] Clifford W. Mercer, Raguanathan Rajkumar, and Hideyuki Tokuda. Applying hard real-time technology to multimedia systems. In *Workshop on the Role of Real-Time in Multimedia/Interactive Computing System*, 1993.

[18] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the 4th Real-Time Computing Systems and Application Workshop*. IEEE, November 1997.

[19] Saowanee Saewong, Ragunathan Rajkumar, John P. Lehoczky, and Mark H. Klein. Analysis of hierarchical fixed-priority scheduling. In *Proceedings of the* 14^{th} *IEEE Euromicro Conference on Real-Time Systems*, June 2002.

[20] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *The Journal of Real-Time Systems*, 1:27–60, 1989.

[21] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1994.

[22] Marco Spuri. Holistic analysis of deadline scheduled real-time distributed systems. Technical report, INRIA, 1996.

[23] K. Tindell. An extendible approach for analysing fixed priority hard real-time tasks. *Journal of Real-Time Systems*, 6(2), March 1994.

[24] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing & Microprogramming*, 50(2-3):117–134, April 1994.