IST-2001 34140

Scheduler integration definition report

Deliverable D-SI.1v2

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

3rd April 2003

Contents

1 Introduction

In this document we present the software architecture of the systems that we intend to develop and support 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 is reported. The complete list of requirements is listed in the /AF/requirementslist.pdf document, available for download 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, and a first draft proposal for an API. 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 will be supported by the operating system mechanisms. Finally, in Section 8 we presents our conclusions.

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 Abbreviated List of Application Requirements

This section provides an abbreviated list of the application requirements that FIRST will address. The full numbered list of supported application requirements is listed in the document /AF/requirementslist.pdf on the FIRST web site, and will be included in the FIRST deliverable D-AF.1-v2.

1 Multiple Applications/Components. This section describes the ability of the framework to support multiple applications at the same time.

- **1.1** Composability, the ability to compose together separate applications with their own timing requirements and schedulers.
- **1.2** Shared Resources, to be able to deal effectively with resources shared between applications.
- **1.3** Distribution, to support distributed applications.

2 Contract Support. 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 a 'stream of jobs' model. 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.

3 Application Requirements. The application requirements which form the contract are described in 5 sections: periodicity, resource usage, guarantees, change management and robustness. Note that the characteristics in each section are neither mutually exclusive, nor required to be specified in all circumstances.

- **3.1** Periodicity. This concerns how often Jobs 'arrive' for processing. The framework will support the following types of periodicity: periodic, sporadic, bursty, continuous scale, discrete scale, unbounded.
- **3.2** 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: minimum required execution time per job invocation, continuous scale, discrete scale, data/time dependent execution time, execution time profiling, and will support a high variability of execution times.
- **3.3** Performance and Guarantees. This section relates to the general area of supporting deadlines, quality of service and application specific control of resource usage. Specifically: the job can be informed of how much execution time it has, and the framework will directly support hard deadlines, and an importance/weight hierarchy for distributing spare capacity. In addition, the project will explore the feasibility of supporting data/time dependent deadlines based on contract renegotiation, and the possibility of supporting soft deadlines through an off-line analysis.

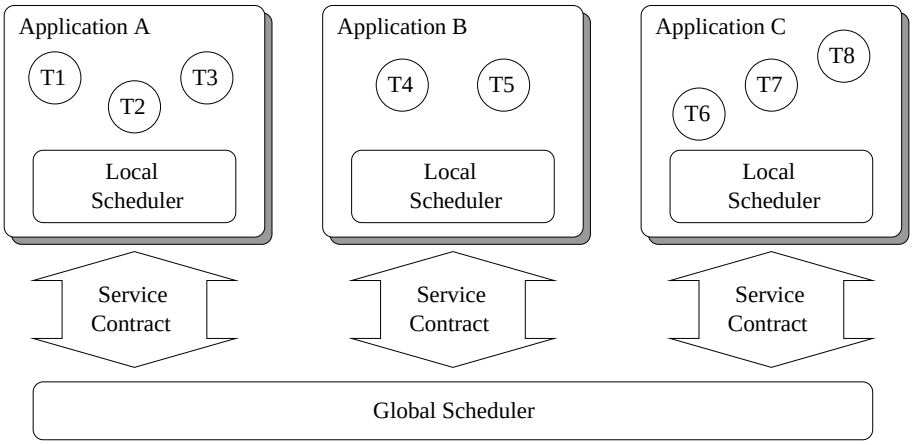

Figure 1: Hierarchy of schedulers.

3.4 Change Management. This concerns the ability of an application to change its requirements at run-time or join a system. The following shall be supported: contract renegotiation, bounded negotiation time, open systems, closed systems.

3.5 Robustness and Isolation. This section considers the temporal semantics in the case of contracts being broken. This applies particularly to applications breaking their contract.

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

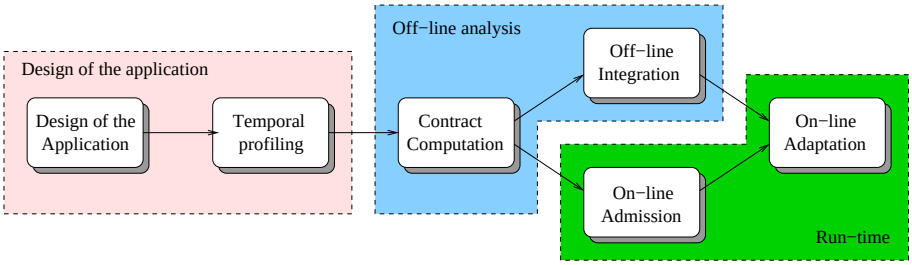

Figure 2: Design flow of an application.

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 protection 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 deciding which resources are *private* to this application and which might be *shared* with other ap-

plications. 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 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 application requires a certain level of service to the system by issuing a service contract. This contract specifies a number, n, of scheduling servers required by the application, each server with the parameters described in Table 1. If the system accepts the contract, then it must guarantee that the terms of the contract are respected throughout the life of the application, or until the contract is renegotiated at the request of the application, and accepted by the system.

Parameter	Values	Observation
Budget	$C_{i,min}$, $C_{i,max}$	mandatory
Period	$T_{i,min}$, $T_{i,max}$	mandatory
D=T	Yes/No	optional
Deadline	D_i	optional
Granularity	Continuous or discrete	default = continuous
Utilisation set	($C_{i,1}$,$T_{i,1}$,. . .,$C_{i,n}$,$T_{i,n}$)	optional
Preemption level	P_i	mandatory
Critical sections	($C_{s,1}$,$P_{s,1}$,. . .,$C_{s,n}$,$P_{s,n}$)	optional
Quality and importance	Q_i	optional
Workload	Bounded/Indeterminate	default = indeterminate
Budget Overrun Notification	none or signal number	default=none
Deadline Miss Notification	none or signal number	default=none

Table 1: Contract parameters

The system does not directly guarantee any temporal requirement of the application. It is upon the system designer to *translate* the application requirements into appropriate contract parameters. In this way, if the system respects the contract the application requirements are fulfilled. It is also possible to specify more than one contract for the same application.

The contract parameters are listed in Table 1. They are discussed in the following.

The **Budget** and **Period** are the basic parameters of every service contract. They correspond the the equivalent parameters of any periodic server algorithm. The application will be guaranteed a minimum of $C_{i,min}$ units of execution time out of $T_{i,max}$ units of time. If the application requires less than this amount (because all of its tasks are idle or blocked for some reason and the application knows that it cannot make any further progress until its next server period) the spare execution time can be assigned to other applications. If the application requires more than that, the system can allow more execution time depending on the amount of available spare resources. The budget and the period are expressed as a minimum and maximum amount. $C_{i,min}$ and $T_{i,max}$ are very important, because they set the minimum amount of service that the application will receive in the worst case situation. The $C_{i,max}$ and $T_{i,min}$ are used for informing the system that the application will never require more than $C_{i,max}$ out of every successive interval of $T_{i,min}$ units of time. This information can help the system to manage the spare bandwidth more efficiently, but it does not affect the minimum guarantees.

The ratio C_i/T_i is also called *bandwidth* of the server. It is important to notice that, from the application's point of view, the system can be seen approximately as a slower dedicated processor of speed sC_i/T_i, where s is the speed of the processor. The level of approximation depends on the server algorithm and on the T_{min} parameters. The approximation will be discussed in more details in Deliverable D-SI.4v2.

The **Deadline** parameter is the deadline of the server that will manage the application. It is an optional parameter; if it is not specified, the system may assume a server's deadline equal to the server's period (although no notification will be made if this artificial deadline is not met). If on-line admission testing is enabled, the system will guarantee that the application receives its guaranteed $C_{i,min}$ before this deadline, relative to the start of the server's period. If on-line admission test is not enabled, the system will notify the application if the deadline is not met. If the **deadline=period** parameter is set to "yes", then the deadline parameter must not be present and the deadline of each server's job will be set equal to the current period

assigned by the system to that particular job.

The **Granularity** type is used to specify an adaptive contract. An adaptive contract specifies the parameters by which an application can make use of any spare or reclaimed capacity that the system may be able to assign to it. If the budget or the period can be dynamically changed by using an on-line adaptive mechanism, then this parameter indicates how this change can take effect. If the granularity is continuous, then the budget and the period can be changed arbitrarily between the minimum and maximum values specified by the budget and period parameters of the contract. If the granularity is discrete, the budget and the period can only take specific values among a set of different possibilities, described in the **Utilisation set** as a list of possible (budget:period) pairs ordered by increasing utilisation. These values must be within the maximum and minimum values specified by the budget and period parameters in the contract. The first pair must be $(C_{i,min} : T_{i,max})$ and the last one must be $(C_{i,max} : T_{i,min})$.

The **Preemption level** is an integer number that represents the preemption level of the server. It is important when dealing with critical sections guarded by mutex semaphores with the Stack Resource Policy (SRP) [2]. This policy has been chosen because it is compatible both with underlying EDF or FP schedulers. It should be chosen such that the shared resources and associated mutex semaphores used by the tasks allocated to the server always have a preemption level that is higher than or equal to the server's preemption level. As a simple rule of thumb, if we assign fixed priorities to the servers, the preemption level of the server is its priority, and the preemption level of the resource is the priority ceiling. If, instead, we assign deadlines to the servers, the server's preemption level can be an integer proportional to the inverse of its deadline, and the resource preemption level would be the highest of the preemption levels of all the servers using that resource. For a more detailed discussion of the SRP algorithm and of the use of the preemption level, please see [2, 11, 6].

The set of **Critical sections** is used when applications share resources using mutex semaphores with the SRP policy [2]. For each different preemption level among the resources used by the tasks allocated to this server, the contract must specify the maximum lengths of the critical sections, i.e., the worst-case execution time spent with the associated mutex semaphore locked. Therefore, the list of critical sections is a list of pairs of execution time and preemption level, only one value per preemption level, and ordered by decreasing values of preemption level.

The **Quality** and **Importance** parameters specify how the spare capacity can be distributed among the different servers executing in the system. The system will give the spare capacity to the servers of higher importance, up to the maximum that each server can accept. Only if there is additional spare capacity left, it will be assigned to the next important servers, in an iterative way. The importance is a small integer value between 1 and 5, the higher the value the higher the importance. To distribute capacity among the servers of equal importance, the system uses the quality parameter as a measure of the relative share that each server should get. The value is an integer value whose unit is conceived as a percentage of the available bandwidth. The spare capacity that a particular server will get is proportional to its quality divided by the sum of qualities of equal-importance servers. The specific value may range between 0 and 1000.

The **Workload** parameter specifies whether each job running under a particular server has a bounded workload, or whether the amount of work that the job can consume is indeterminate. For example, a periodic task running under a server is typically bounded, in this case the task period coincides with the server period. An aperiodic task for which no inter-arrival time can be specified is usually indeterminate, because it is not possible to predict how much work it can request during a server period. The concept of workload type is important to be able to do reclamation of unused capacity for bounded tasks. The system will schedule the start of bounded tasks, and will expect them to declare when they finish the execution of a particular job, so that reclamation of unused capacity can be made. Reclamation of indeterminate workloads is not always possible, because the system must assume that the server will have to provide its guaranteed minimum

capacity at any time.

The **Budget overrun notification** parameter tells the system whether the application or upper-level scheduler wants to be notified (using a signal mechanism) about budget overruns or not, and in case notification is requested, which signal number is to be used. Only bounded workload servers can notify about budget overruns.

The **Deadline miss notification** parameter tells the system whether the application or upper-level scheduler wants to be notified (using a signal mechanism) about deadline misses or not, and in case notification is requested, which signal number is to be used. Only bounded workload servers with a deadline specified (either a fixed value or with the deadline=period parameter equal to "yes") can notify about deadline misses.

4.1 A first draft of the Service Contract API

The API between the application or upper-level scheduler and the underlying scheduler will include at least the operations listed below. The contract parameters are specified as an opaque or private type, depending on the programming language facilities. Values of this type are only set through the interfaces defined by the following abstract operations, or additional ones that may be defined in the future:

- Initialize

  Input Data none

  Output Data Contract parameters object

  Description The operation initializes a contract parameters object setting it to the default values, and returns this object

- Set_Basic_Parameters

  Input Data Budget, Period, Workload (Bounded or Indeterminate), Contract parameters object

  Output Data Updated contract parameters object

  Description The operation updates the specified contract parameters object by setting its budget, period, and workload to the specified input parameters. (Note: the workload is a basic parameter because bounded tasks are triggered by the scheduler (see the Timed_Schedule_Next_Job operation, later), while indeterminate tasks are not; therefore, their programming model is quite different).

- Set_Timing_Requirements

  Input Data D=T (boolean), Deadline (must be null if D=T true), Budget Overrun Notification (may be null), Deadline Miss Notification (may be null), Contract parameters object

  Output Data Updated contract parameters object

  Description The operation updates the specified contract parameters object by setting its D=T and deadline parameters to the specified input parameters.

- Set_Reclamation_Parameters

  Input Data Granularity (continuous or discrete), Utilization set (null if continuous), Quality and Importance, Contract parameters object

  Output Data Updated contract parameters object

Description The operation updates the specified contract parameters object by setting its granularity, utilization set, quality, and importance to the specified input parameters.

- Set_Synchronization_Parameters

  Input Data Preemption Level, Critical sections, Contract parameters object

  Output Data Updated contract parameters object

  Description The operation updates the specified contract parameters object by setting its preemption level and critical sections to the specified input parameters.

An abstract synchronization object is defined by the application. This object can be used by an application to wait for an event to arrive by invoking the Event_Triggered_Schedule_Next_Job operation. It can also be used to signal the event either causing a waiting server to wake up, or the event to be queued if no server is waiting for it. It has the following operations:

- Init

  Input Data None

  Output Data Initialized synchronization object

  Description This operation initializes a synchronization object managed by the scheduler.

- Signal

  Input Data Synchronization object

  Output Data None

  Description If one or more servers are waiting upon the specified synchronization object one of them is awakened; if not, the event is queued at the synchronization object.

- Negotiate_Contract

  Input data Contract parameters

  Output data Accepted (boolean), Server Id (Number)

  Description The operation negotiates a contract for a new server. If the on-line admission test is enabled it determines whether the contract can be admitted or not based on the current contracts established in the system. Then it creates the server and recalculates all necessary parameters for the contracts already present in the system. This is a potentially blocking operation; it returns when the system has either rejected the contract, or admitted it and made it effective.

- Cancel_Contract

  Input data Server_Id

  Output data None

  Description The operation eliminates the specified server and recalculates all necessary parameters for the contracts remaining in the system. This is a potentially blocking operation; it returns when the system has made the changes effective.

- Renegotiate_Contract

Input data Server Id, New Contract Parameters

Output data Accepted (boolean)

Description The operation renegotiates a contract for an existing server. If the on-line admission test is enabled it determines whether the contract can be admitted or not based on the current contracts established in the system. If it cannot be admitted, the old contract remains in effect. If it can be admitted, it recalculates all necessary parameters for the contracts already present in the system. This is a potentially blocking operation; it returns when the system has either rejected the new contract, or admitted it and made it effective.

- Request_Contract_Renegotiation

  Input data Server Id, New Contract Parameters, Notification (None, or signal number)

  Output data None

  Description The operation enqueues a renegotiate operation for an existing server, and returns immediately. The renegotiate operation is performed asynchronously, as soon as it is practical; meanwhile the system operation will continue normally. When the renegotiation is made, if the on-line admission test is enabled it determines whether the contract can be admitted or not based on the current contracts established in the system. If it cannot be admitted, the old contract remains in effect. If it can be admitted, it recalculates all necessary parameters for the contracts already present in the system. When the operation is completed, notification is made to the caller, if requested, via a signal. The status of the operation (in progress, admitted, rejected) can be checked with the `Renegotiation_Request_Status` operation.

- Change_Quality_And_Importance

  Input data New Quality, New Importance, Server Id

  Output data None

  Description The operation enqueues a request to change the quality and importance parameters of the specified server, and returns immediately. The change operation is performed as soon as it is practical; meanwhile the system operation will continue normally.

- Timed_Schedule_Next_Job

  Input data Server_Id, Absolute_time

  Output data Current Budget, Current Period, Deadline Missed (boolean), Budget Overran (boolean)

  Description This operation is invoked for bounded workload servers to indicate that a job has been completed (and that the scheduler may reassign the unused capacity of the current job to other servers), and also when the first job requires to be scheduled. The system will activate the job at the specified absolute time, and will then use the scheduling rules to determine when the job can run, at which time the call returns. Upon return, the system reports the current period and budget for the current job, whether the deadline of the previous job was missed or not, and whether the budget of the previous job was overrun or not.

- Event_Triggered_Schedule_Next_Job

  Input data Server_Id, Synchronization object,

Output data Current Budget, Current Period, Deadline Missed (boolean), Budget Overran (boolean),

Description This operation is invoked for bounded workload servers to indicate that a job has been completed (and that the scheduler may reassign the unused capacity of the current job to other servers), and also when the first job requires to be scheduled. If the specified synchronization object has events queued, one of them is dequeued; otherwise the server will wait upon the specified synchronization object until it is signalled. Then, the system will use the scheduling rules to determine when the job can run and the call will return at that time. Upon return, the system reports the current period and budget for the current job, whether the deadline of the previous job was missed or not, and whether the budget of the previous job was overrun or not.

- Available_Capacity

  Input data Server Id

  Output data Capacity, in percentage of utilization

  Description This operation returns the current spare capacity (in percentage of processor or network utilization), currently assigned to the importance level of the specified server.

- Total_Quality

  Input data Server Id

  Output data Quality

  Description This operation returns the sum of the quality parameters for all servers in the system of importance level equal to that of the specified server.

- Renegotiation_Request_Status

  Input data Server Id

  Output data In-progress, rejected, admitted

  Description The operation reports on the status of the last renegotiation operation enqueued for the specified server. It is callable even after notification of the completion of such operation, if requested.

- Is_Admission_Test_Enabled

  Input data None

  Output data Test Enabled Status (boolean)

  Description Returns true if the system is configured with the on-line admission test enabled, and false otherwise.

Operations to negotiate, cancel, and renegotiate multiple-server contracts will be available in the future (next phase of the project). For now, a multiple-server contract can be negotiated as a sequence of single-server contracts.

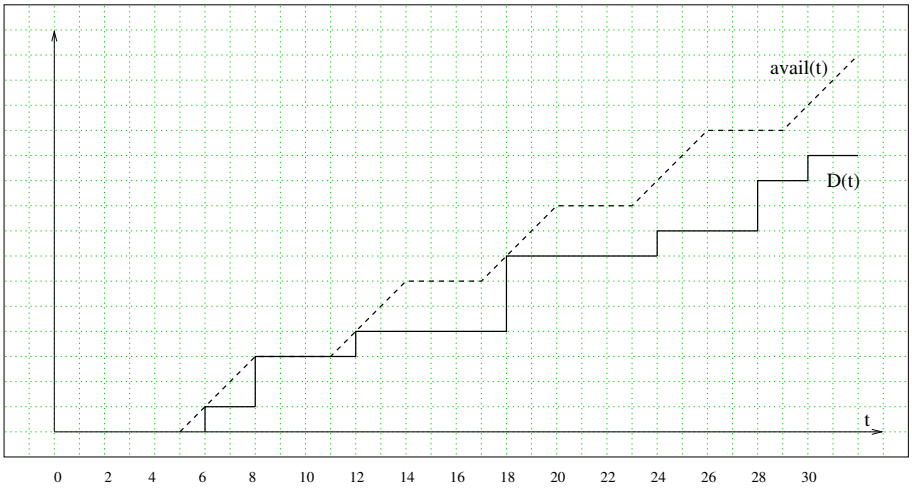

Figure 3: Demand bound function and server's available time for the example's application.

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 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 3 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 [12].

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 [17], and the Processor Capacity Reserve [13, 15]. Examples of servers for earliest deadline first are the Dynamic Sporadic Server, the Total Bandwidth Server [18] and the Constant Bandwidth Server [1]. The Total Bandwidth server has also been combined with the Slot Shifting algorithm by Isovic and Fohler [9] 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* [15] 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 [18] to serve an application with its own local scheduler. Saesong et. al. [16] 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 [12]), 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 algorithms, nor to any specific global scheduling algorithm. Two different implementation 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 1, that fulfill the application requirements. In the second phase of this project, we will investigate the problem of computing the contract parameters, by extending the approach presented in [12] and by off-line analysis for complex constraints. The resulting techniques will be presented in D-SI.4v2.

7 Integration and implementation

This section presents in more detail how the framework described in the previous sections will be implemented in the two operating systems used in the FIRST project, MaRTE OS and Shark OS. These mecha-

nisms will then be implemented in the second phase of Workpackage 3. The APIs for accessing the proposed service contract, both in Ada and in C, will be defined in Workpackage 3 and will be the same for both operating systems.

In Sections 7.1 and 7.2, we describe the basic scheduling mechanisms that will be used in Shark OS and in MaRTE OS, respectively. Section 7.3 discusses how to support interacting applications, i.e. applications that need to exchange data. In Section 7.4 presents possible implementations of the admission test. Support for distributed systems is discussed in Section 7.5.

7.1 Service based on EDF and CBS

In the Shark OS, the framework will be implemented using the earliest deadline first (EDF) scheduler and the Constant Bandwidth Server (CBS) algorithm [1]. We will use a slightly modified version of the CBS that is more suited to hierarchical scheduling. In the original CBS [1], when the server budget is exhausted the server is not suspended. The free bandwidth is automatically reclaimed by postponing the server deadline and inserting the server again in the ready queue. Although this can be considered an advantage when scheduling aperiodic tasks, this rule creates some problems when using the CBS for hierarchal scheduling.

To explain why, consider an application consisting of two periodic tasks, τ_1 with computation time equal to $C_1 = 10$ and period $T_1 = 100$, another one is sporadic with computation time and period equal to $C_2 = 1$, $T_2 = 10$. This application is scheduled by EDF and is assigned a CBS with $C_s = 2$, $T_s = 10$. Suppose that τ_1 is the only active task at time 0, and that the server is the one with the earliest deadline. According to the CBS algorithm (see [1]) after two units of time the budget is replenished and the server deadline is postponed to $d_s = 20$. If the server is still the one with the earliest deadline, it continues to execute and again depletes its budget. In the worst possible situation, when τ_1 has finished at time $t = 10$, the server deadline is equal to 100. Now, if τ_2 arrives, the server deadline is set to $d_s = 110$. In the worst case, the first time for τ_2 to be executed is $t = 108$, because of other servers with deadline less than 110. As a consequence, τ_2 will probably miss its deadline.

To avoid such problem, a new rule is added to the CBS algorithm. Also, the CBS algorithm is extended by allowing a relative deadline for each server that can be different from the server period.

We report the CBS algorithm along with the new rules. Also, we relate the server algorithm to the service contract parameters described in Section 4.

The modified CBS Algorithm. A CBS S_i is described by: *the server budget*, C_i; the *server period*, T_i; and the *server relative deadline*, D_i. When supporting an adaptive contract, these values can vary dynamically: however, $C_{i,min} \leq C_i \leq C_{i,max}$ and $T_{i,min} \leq C_i \leq T_{i,max}$. The *server bandwidth*, $U_i = \frac{C_i}{T_i}$, is the fraction of the CPU bandwidth reserved to S_i. To avoid inconsistencies and overload situations, the following condition must hold at all times:

$$\sum_{\forall i} U_i \leq 1.$$

Therefore, a variation in the parameters is allowed only at replenishment time, and only if the above condition is respected.

Dynamically, each server updates two variables (q_i, d_i). Variable q_i is the *current budget* and keeps track of the consumed bandwidth. Variable d_i is the server's *scheduling deadline*. Initially, q_i is set to the maximum budget C_i and d_i is set to 0. A server is *active* if any of the application's tasks has a pending instance and the current budget is greater than 0. If there is pending instance and the current budget is 0, the server is *suspended* until the current budget is recharged.

The system consists of n servers and a global scheduler based on the Earliest Deadline First (EDF) priority assignment. At each instant, the active server with the earliest scheduling deadline d_i is selected and the corresponding task is dispatched to execute.

1. Initially, $q_i = 0$, $d_i = 0$ and the server is *inactive*.
2. When a task is activated at time t, if the server is *inactive*, then $q_i = C_i$ and $d_i = t + D_i$, and the server becomes *active*. If the server is already active, then q_i and d_i remain unchanged.
3. At any time t, the global scheduling algorithm selects the active server with the earliest deadline d_i. When the server is selected, it executes the first task in its ready queue (which is ordered by the local scheduling policy).
4. While some application task is executing, the current budget q_i is decremented accordingly.
5. The global scheduler can *preempt* the server to execute another server: in this case, the current budget q_i is no longer decremented.
6. If $q_i = 0$ and some task has not yet finished, then the server is *suspended* until time $r_i = d_i - D_i + T_i$; at time r_i, q_i is recharged to C_i, d_i is set to $d_i + T_i$ and the server can execute again.
7. When, at time t, the last task has finished executing and there is no other pending task in the server, the server yields to another server. Moreover, if $t \geq r_i - q_i \frac{T_i}{C_i}$, the server becomes *inactive*; otherwise it remains *active*, and it will become *inactive* at time $r_i - q_i \frac{T_i}{C_i}$, unless another task is activated before.

7.2 Service based on FPS and SS

In MaRTE OS, the framework will be implemented using the underlying fixed priority scheduler and a modified version of the sporadic server algorithm [17]. This server is characterised by two parameters: the execution capacity, and the replenishment period. Initially the execution capacity is set to the maximum. When a portion of execution capacity is spent at the desired priority level, a replenishment operation is scheduled to occur at the time of the activation of that portion of execution plus the replenishment period. When the capacity is zero, the server is not allowed to continue executing (or is set to a background priority level). When the capacity is higher than zero, the server is allowed to execute with its normal priority level.

The above algorithm can guarantee a bandwidth equal to the the maximum execution capacity every replenishment period. Moreover, the effects of a sporadic server on lower priority tasks are bounded to those of an equivalent periodic task with execution time equal to the maximum execution capacity, and a period equal to the replenishment period.

As a result, the sporadic server provides a bandwidth guarantee while having bounded effects on lower priority tasks. It makes the analysis of systems with unbounded aperiodic activities feasible, while giving those aperiodic tasks a level of service that depends on the assigned priority level, and which on average can be rather fast compared to other methods, such as polling.

In our scheduling framework we will use a modified sporadic server in which the total capacity and the replenishment period can be modified to make use of any spare capacity. The appropriate way of handling these changes needs to be investigated.

7.3 Task synchronization

Tasks may exchange data during their execution. If two tasks belonging to the same application and served by the same server exchange data, the synchronization mechanism that is used must be specified in the local scheduling algorithm.

As an example, consider two tasks belonging to the same application that access a shared memory protected by mutex semaphore. It is then possible to use a priority ceiling protocol as local scheduling algorithm. Other applications can use different protocols.

However, when two tasks belonging to different applications interact by exchanging data, a global synchronisation protocol is needed. In this project we will consider the following interaction mechanisms.

1. Asynchronous communication. The data are exchanged through a set of shared memory buffers to avoid blocking and de-couple the temporal behaviour of the two applications. An example of such mechanism is the Cyclic Asynchronous Buffer [5], which allows one writer task to send data to many readers. With such a mechanism the writer always overwrites the content of the buffer, and the readers never consume the data.

2. Shared memory protected by mutex semaphores. To avoid priority inversion and unbounded blocking time, it is necessary to define a global synchronisation mechanism between servers. Also, we must take into account the maximum blocking time for each application in the temporal profiling of the application and in the admission control phase. We choose the Stack Resource Policy [2] as global synchronisation mechanism. In fact the same algorithm can be used with both fixed priority and EDF scheduling mechanisms. When using the SRP together with a fixed priority scheduler, it behaves exactly like the Immediate Priority Ceiling protocol [14] as long as the *preemption level* parameter is mapped directly onto the priority of the server. The SRP has been recently extended to work together with the CBS [6].

3. Communication through signals. A task of one application, can send a signal to a task of another application. This mechanism needs to be controlled at the receiver's end to ensure that excessive work is not included (for example through a leaky-bucket mechanism).

The use of any of the above interaction mechanisms has an impact on the temporal profile of an application. In all the three case, we have to extend the temporal profile by adding information on the data to be exchanged. For example, in the case of asynchronous communication (1), we may want to compute how old the data is when it is read. Thus, we have to specify how often the data is provided by the writer and the maximum delay with which it is provided. In the case of shared memory protected by mutex semaphore (2), we have to specify the worst case execution time of any application's job on each critical section. In the latter case, we have to specify how often the signal is sent. Mechanisms (2) and (3) also require support at the operating system level.

7.4 Utilisation based admission

In *open systems* application can be dynamically activated at run-time requiring a certain service contract. In this setting, it is necessary to provide an *on-line admission test* to see if a new contract can be accommodated given the currently available system resources.

Since the test is done on-line, it must be fast and efficient. Therefore, it is not possible to use complex necessary and sufficient tests like response time analysis [20, 10, 19] or demand bound analysis [3], but

we must restrict to utilisation based tests, which are only sufficient. The drawback is that we may waste resource.

Recently, some new schedulability tests [4] have been presented that can be tuned in order to trade complexity against accuracy. One possibility would be to reserve some processor time (for example through a dedicated server) to perform a complex schedulability test. Using such approach, the result of a request for a new service contract cannot be returned immediately but after some (bounded) delay.

In this phase of the project, we will provide utilisation-based schedulability tests. We will also evaluate the use of more complex mechanisms through a dedicated server mechanism.

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 [21, 20] and EDF [19] 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 somehow difficult because the scheduling decisions themselves cannot be made by the network, but 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 each processor computes all the scheduling decisions for the network in parallel. If all processors are based on the same data they will all make the same scheduling decisions, which will therefore be consistent. Although this implies replicating the scheduling computations, under the above assumption it is better than sending synchronization and scheduling messages around.

To implement the network scheduler it is necessary to make sure that the scheduling information can be distributed at the same time to all the nodes. This can easily be accomplished by broadcasting this information, a service that most modern networks are able to provide.

In summary, the same scheduling framework will be used on the network and in all 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 schedulers in each node will work independently of the others, and independently of the network scheduler. The computations required for the network scheduler are replicated by all the nodes to minimise scheduling traffic on the network. To accomplish this, the network scheduling information must be broadcasted to all the nodes.

7.6 Contract computation and constraint transformation

A key issue in the FIRST framework is the computation of service contracts derived from application requirements. It is straightforward to compute service contract parameters for simple period, deadline constrained applications. More general constraints require novel algorithms to translate constraints into service contracts. The computation of parameters based on the approach presented in [12] will be investigated.

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. We will investigate off-

line transformation methods to derive simple constraints suitable for service contracts, such that when the new simple constraints are fulfilled, the original complex constraints are met as well. Naturally, this comes at the price of loosing optimality.

The resulting techniques will be presented in D-SI.4v2.

8 Summary and conclusions

This document has 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. As explained in Section 7, the proposed mechanisms and API will be implemented in both Shark and MaRTE OS, using different underlying mechanisms.

Schedulability analyses and techniques for the proposed methodologies will be presented in Deliverable D-SI.4v2.

References

[1] 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.

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

[3] S.K. Baruah, L.E. Rosier, and R.R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor. *The Journal of Real-Time Systems*, 2, 1990.

[4] Enrico Bini and Giorgio C. Buttazzo. The space of rate monotonic algorithm. In *Proceedings of the* 23^{rd} *IEEE Real-Time Systems Symposium*, December 2002.

[5] G. C. Buttazzo. Hartik: A real-time kernel for robotics applications. In *IEEE Real-Time Systems Symposium*, December 1993.

[6] M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proceedings of the IEEE Real-Time Systems Symposium*, London. UK, December 2001.

[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] D. Isovic and G. Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, Orlando, Florida, USA, Nov. 2000.

[10] J.P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *IEEE Real-Time Systems Symposium*, 1990.

[11] G. Lipari and G. Buttazzo. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal of Systems Architecture*, 46:327–338, 2000.

[12] 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.

[13] 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.

[14] R. Rajkumar, L. Sha, and J. P. Lehoczky. An experimental investigation of synchronisation protocols. In *Proceedings 6th IEEE Workshop on Real-Time Operating Systems and Software*, pages 11–17, 1989.

[15] 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.

[16] 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.

[17] 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.

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

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

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

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