Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 116 (2005) 253–266
www.elsevier.com/locate/entcs

A Hierarchical Framework for Component-based Real-time Systems

Giuseppe Lipari[1,2] Paolo Gai[3] Michael Trimarchi[5]
Giacomo Guidi[4] Paolo Ancilotti[6]

Scuola Superiore S. Anna
Pisa (Italy)

Abstract

In this paper, we describe a methodology for the design and the development of component-based real-time systems. In our model, a component consists of a set of concurrent real-time threads that communicate by means of synchronous and asynchronous operations. In addition, each component can specify its own *local scheduling algorithm*. We also discuss the support that must be provided at the operating system level, and present an implementation in the SHaRK operating system.

Keywords: Real-Time, Component-based design, Hierarchical scheduling

1 Introduction

Component-based design and development techniques are now being applied to real-time embedded systems. However, such techniques must be adapted to the particular needs of this domain. Until now, little work has been done on the

[1] This work has been partially supported by the Italian Ministry of University and Research within the COFIN 2001 project "Quack: a platform for the quality of new generation integrated embedded systems", and by the European Community within the IST project 34140 FIRST (Flexible Integrated Real-Time System Technology).
[2] Email: lipari@sssup.it
[3] Email: pj@gandalf.sssup.it
[4] Email: guidi@gandalf.sssup.it
[5] Email: trimarchi@gandalf.sssup.it
[6] Email: paolo@sssup.it

doi:10.1016/j.entcs.2004.02.081

characterization of the quality of service of a component from a temporal point of view. This characterization is especially useful in the real-time domain, where components consist of concurrent cyclic tasks with temporal constraints (e.g. deadlines). In fact, when we integrate all the components in the final system, we must be able to analyse the schedulability of the system (i.e. to check if the temporal constraints are respected).

Lipari, Bini and Fohler [10] presented a model for real-time concurrent components. A component consists of one or more concurrent real-time threads and it is characterized by a *demand function* that describes its temporal requirements. The methodology was later extended by Lipari and Bini [9]. However, in these papers, simplified model is considered in which components can communicate only asynchronously. In this paper, we refine the model of a real-time concurrent component by considering blocking primitives, like synchronized methods. We also present an implementation of these techniques in the real-time operating system SHaRK.

The paper is organized as follows. In Section 2 we describe the model of the system and motivate our work. In Section 4.1 we present our model of a component and we list the system requirements. Section 5 describes briefly the mechanisms that must be provided by a real-time operating system to support our model. Section 6 describes the implementation in the SHaRK operating system. Finally, Section 7 presents the conclusions and some future work.

2 System model

2.1 The thread model

The thread model of concurrent programming is very popular and it is supported by most operating systems. In this model, concurrency is supported at two levels: processes and threads. Each process has its own address space and processes communicate mainly exchanging messages by means of operating system primitives. Creating a process is an expensive operation because it involves a lot of steps like creating the address space, setting up the file descriptors, etc. Moreover, context switch among processes is an expensive operation.

A process can be multi-threaded, i.e. it can consist of several concurrent threads. Different threads belonging to the same process share address space, file descriptors, and other resources. Since threads belonging to the same process share the address space, the communication is often realized by means of shared data structures protected by mutexes. Creating a new thread is far less expensive than creating a new process. Also, context switch among

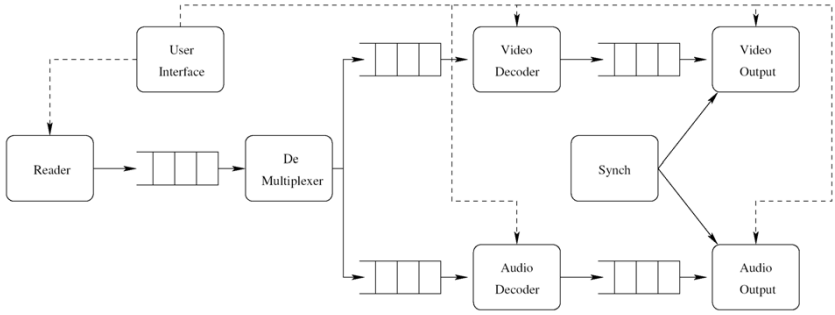

Fig. 1. Typical structure of an MPEG player.

threads of the same process is faster. The thread model is supported by all general purpose operating systems because it has a lot of advantages with respect to the pure process models. The designer of a concurrent application can design the application as a set of cooperating threads, simplifying the communication and reducing the overhead of the implementation.

As an example, consider an MPEG player that plays video and audio streams coming from the network. A typical design structure for this application is shown in Figure 1: it consists of several concurrent threads denoted by rounded boxes. A first "Reader" thread waits for new data from the network and writes them into a buffer. The data are then "de-multiplexed" into video and audio data and sent to separate threads. The "Video decoder" and the "Audio decoder" threads perform the decompression and decryption of the video frames and send the data to two the video and audio output, respectively. The "Synch" threads synchronizes the images with the sound. Finally, a "User interface" thread interacts with the user and sends asynchronous command to all the other threads. All the threads interact tightly: therefore, communication and scheduling must be fast and efficient.

2.2 Real-time applications

Classical hard real-time systems usually consist of periodic or sporadic tasks that tightly cooperate to fulfill the system goal. For efficiency reasons, they communicate mainly through shared memory, and appropriate synchronization mechanisms are used to regulate the access to shared data. Since all tasks in the system are designed to cooperate, a global schedulability analysis is done on the whole system to guarantee that the temporal constraints will be respected. There is no need to protect one subset of tasks from the others. Therefore, we can assimilate a hard real-time system to a single multi-threaded process where the real-time tasks are modeled by threads.

If we want to support some sort of real-time execution in general-purpose

operating systems we have to take into consideration multi-threaded programming. According to the multi-thread model, in this paper we assume that real-time tasks are implemented as threads, and a classical real-time application as one single multi-thread process. Therefore, a real-time application is a process that can be multi-threaded, that is, it can consists of many real-time tasks. In the remainder of the paper, we will use the terms *thread* and *task* as synonyms. The same for the terms *application* and *process*.

A user that wants to execute (soft) real-time applications in a general-purpose operating system would like to have the following nice features:

- It should be possible to assign each real-time application a *fraction* of the system resources, so that it executes as it were executing alone in a slower *virtual processor*;
- Each application should receive execution in a timely manner, depending on its real-time characteristics (e.g., the tasks' deadlines);
- A non real-time application should not be able to disrupt the allocation guaranteed to real-time applications.

2.3 Customized scheduler

Figure 2 illustrates an example of a multi-thread real-time operating system. An interesting feature would be the possibility of specifying a *local scheduler* for the application's threads. For example, Application A could specify a non-preemptive FIFO scheduler, Application B could specify a Round Robin scheduler, whereas Application C a fixed priority scheduler.

Therefore, in this model, we distinguish two levels of scheduling. The *global scheduler* selects which application is assigned the processor, and the *local scheduler* selects which task has to execute on the processor. This two-level scheduling has many obvious advantages:

- each application could use the scheduler that best fits its needs;
- legacy applications, designed for a particular scheduler, could be re-used by simply re-compiling, or at most, with some simple modification.
- support component-based design of concurrent real-time applications. Each component can be seen as a multi-threaded process with its own custom scheduler.

In addition, if different applications that have been developed assuming different scheduling paradigms must be executed on a single platform with one specific scheduler, some problem can arise.

A good example of such portability problems is the well-known problem with the thread support in the Java Virtual Machine (JVM) [15][13]. In fact,

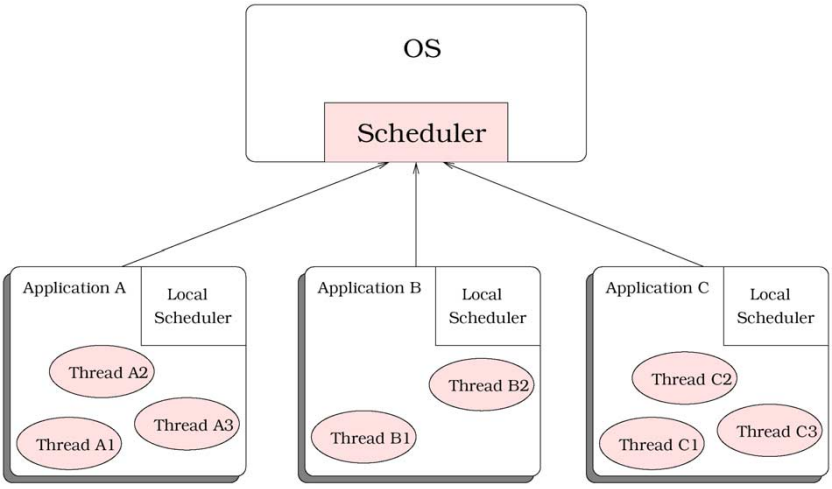

Fig. 2. Structure of a multi-thread system, where each application consists of one or more threads and a customized scheduler.

most JVMs rely on the operating system support for implementing threads. If a Java program has been developed under Windows NT (that assumes round robin scheduling for threads) and is run under Solaris with the so-called "green-thread" implementation (that assumes fixed priority scheduling), the program may unexpectedly go into a deadlock. In fact, the developer has implicitly made an assumption on the underlying scheduling mechanism, and it is not easy to correct such assumption.

If we want to integrate the two components in a new system and we cannot go back to the design phase (for example for cost reasons), we need a method for combining and analyzing the two components together, *without changing their local scheduling algorithms.*

3 Related work

Most of the research on component based real-time embedded system is related to the software design phase. Only recently non-functional constraints like deadline are being taken into consideration.

The OMG has proposed the UML-RT profile for schedulability, performance and time specification [16]. The profile allows the design of real-time applications with UML. However, the profile is not well suited for component based design.

Kopetz [6] proposes a methodology based on the Time Triggered paradigm to develop systems according to a component-based approach. His methodology is based on the concept of "temporal firewall". A temporal firewall is an

sequence of temporal intervals in time between which each components must execute. The approach is very interesting, but it is only applicable in the context of static off-line scheduled systems.

Isovic, Lindgren and Crnkovic [5] presented a similar idea in the context of the slot shifting scheduler [2]. However, a component can consists of one single thread.

Nielse and Agha [14] propose to further constraint a component in order to separate the functional specification from the timing constraints. For example, a component is not allowed to specify the scheduling policy, nor priorities or deadlines. The timing constrains are specified separately and verified after integration. In contrast, in our work paper we explicitly allow components to specify their own scheduling policy.

Stankovic [21] proposes a tool set called VEST that allows the construction and the analysis of component based real-time systems. Again, a component is not allowed to specify its own scheduling algorithm. Moreover, a failing component can influence the behaviors of all other components in the systems, since there is no temporal isolation between components.

A general methodology for temporal protection in real-time system is the resource reservation framework [18,11,12]. The basic idea is that each task is assigned a *server* that is reserved a fraction of the processor available bandwidth: if the task tries to use more than it has been assigned, it is *slowed down*.

This framework allows a task to execute in a system as it were executing on a dedicated virtual processor, whose speed is a fraction of the speed of the processor. Thus, by using a resource reservation mechanism, the problem of schedulability analysis is reduced to the problem of estimating the computation time of the task without considering the rest of the system.

Recently, many techniques have been proposed for extending the resource reservation framework to hierarchical scheduling. Baruah and Lipari in [8] propose the H-CBS algorithm, which permits to compose CBS schedulers at arbitrary levels of the hierarchy. The H-CBS is also able to *reclaim* the spare time due to tasks that execute less than expected according to the hierarchical structure. However, the only scheduling algorithm permitted at all levels is the EDF algorithm. A similar work has been done by Saewong et al. [19] in the context of the resource kernels. They propose a schedulability analysis based on the worst-case response time for a local fixed priority scheduler.

Lipari, Bini and Fohler [10,9] developed a methodology for analysing hierarchical schedulers based on a server algorithm like the CBS. In addition, they propose a methodology for deriving the server parameters from the component temporal characteristic.

4 Component-based real-time systems

4.1 Model of a real-time component

In our model, a real-time component $\mathcal{C}$ is defined by a t-uple $\{\mathcal{T}, \mathcal{S}, \mathcal{R}, \mathcal{P}, DBF\}$, where:

- $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is the set of concurrent threads;
- $\mathcal{S}$ is the *local scheduling algorithm* for the component's threads (see Section 2.3);
- $\mathcal{R} = \{\omega_1, \ldots, \omega_m\}$ is the set of *required synchronized operations*;
- $\mathcal{P} = \{\pi_1, \ldots, \pi_l\}$ is the set of *provided synchronized operations*;
- $DBF(t)$ is the demand bound function as defined in [10,9].

A thread $\tau_i = \{C_i, D_i, T_i\}$ can be periodic or sporadic. It is characterized by a worst-case execution time C_i, a relative deadline D_i and a period (or a minimum interarrival time) T_i. Such threads must be scheduled by the local scheduling algorithm $\mathcal{S}$. In our research, we considered Fixed Priority (FP), Earliest Deadline First (EDF), Round Robin (RR). In general, any scheduling algorithm can be used as local scheduling algorithm.

The component may offer some synchronized operation to be used by other components. These are called *provided synchronized operations*. Every provided operation π_j is characterized by a mutex μ and a worst case execution time $d(\pi_j)$. Of course, the provided operation can be also accessed by the threads belonging to the same component. The component may also offer non-synchronized operations, i.e. simple function calls. For the sake of simplicity, we do not consider these functions in this model since they cannot cause blocking time.

The component may need to execute synchronized operations provided by other components. These are called *required synchronized operations*. The thread that performs a call to a required synchronized operation ω_i can be blocked because the corresponding mutex is currently used by another thread. This blocking time has to be taken into account for checking the performance of the component. If the blocking time is too long, some deadline could be missed. In the next section, we will briefly describe a methodology for computing such blocking time.

The *demand bound function* $DBF(\Delta t)$ of a component is the maximum amount of time that the component requires in any interval of length δt otherwise some deadline could be missed. Therefore, in designing our system, we must ensure that the component is allocated at least $DBF(\Delta t)$ units of time in every interval of time Δt. The demand bound function can be computed

from the thread characteristics and from the local scheduler, as described in [10,9]. For example, assuming an EDF local scheduler, the demand bound function of a set of periodic threads can be computed as:

$$DBF(\Delta t) = \sum_{i=1}^{n} \left(\left\lfloor \frac{\Delta t - D_i}{T_i} \right\rfloor + 1 \right) C_i$$

where C_i, D_i and T_i are the worst case execution time, the relative deadline and the period of thread τ_i. For the component to be schedulable, the following condition must hold:

$$\forall \Delta t > 0 \qquad DBF(\Delta t) \leq Z(\Delta t) \tag{1}$$

where $Z(\Delta t)$ is the minimum amount of execution time that the component can receive in any interval of length Δt.

5 Operating system support

In this section we discuss the basic mechanisms to be provided by the operating system to support our methodology. These mechanisms have been implemented in the SHaRK OS [3], an open source real-time kernel. A brief description of the implementation is given in Section 6.

5.1 Temporal isolation

Our system allows the integration of different components. Some component may be very critical; its requirements mandate that each thread in the component must complete its job before the deadline, otherwise something catastrophic may happen. Other components can be less critical: if some deadline is missed nothing catastrophic happens, although the quality of service provided by the component may decrease. Finally, we may have components that do not possess temporal constraints (non real-time components).

To separate concerns, the operating system must support the "temporal isolation property": the temporal behavior of one component (i.e. its ability to meet its deadlines) should only depend on the amount of bandwidth assigned to it and not on the presence of other components in the system.

Temporal isolation can be provided by using the *resource reservation framework* [18]. In this framework, each component $\mathcal{C}_i$ is assigned a fraction of the processor bandwidth U_i. The net effect is that each component executes as it were executing alone on a dedicated processor of speed U_i. In the SHaRK OS, we use the Constant Bandwidth Server (CBS) [1], an algorithm of the class of the *resource reservation algorithms*, and the GRUB algorithm [4], an extension of the CBS.

In these algorithms, each "component" is assigned a "server". A server is an abstraction (i.e. a structure internal to the operating system) that is used by the global scheduler to store the scheduling parameters. Each server S_i is characterized by a *budget* Q_i and a period P_i. The scheduler guarantees that each server S_i (and in turn its associated component) will receive an execution time of Q units of time every period P_i. In practice, these scheduling algorithms are similar to a Round Robin Scheduler, but each server is assigned a different quantum Q_i. The only constraint is that the total bandwidth assigned to the servers cannot exceed 1:

$$\sum_{i=1}^{n} \frac{Q_i}{P_i} \leq 1 \tag{2}$$

5.2 Hierarchical scheduling

As anticipated in Section 2.3, we need to compose schedulers in a hierarchical way. Such hierarchical composition can be done in many different ways. We will follow the same approach as in our previous papers [10,9].

Each component is assigned a local scheduler $\mathcal{S}$. The global scheduler is the CBS algorithm, which is able to provide temporal isolation. The CBS algorithm selects the "server" to be executed. In turn, the local scheduler selects which one of the component's threads must be executed. The thread is allowed to execute until:

- the thread is preempted by another thread of the same component
- the budget of the component is exhausted; at this point, a new server (and hence, a new component) is selected by the global scheduler.

By using a hierarchical scheduling strategy together with the resource reservation framework, we guarantee that each component behaves approximately as it were executing alone on a dedicated slower processor. In this way we allow:

- independence among components; each component can be developed separately from the rest of the system. Also, non-functional constraints like deadline or precedence constraints can be verified independently from the rest of the system; in this respect, our work differs radically from other approaches [21,14,5]
- temporal protection; a malfunction in a component does not influence the behaviors of the other components.

5.3 Synchronized operations

When we consider synchronized operations, components can actually interfere among each other. For example, if a component $\mathcal{C}_1$ wants to invoke a synchronized operation π_{21} on another component $\mathcal{C}_2$, and the operation is currently locked, the first component experiences a *blocking time* delay. This delay depends on how long the operation remains locked by another component. If it remains locked for too long, some deadline in component $\mathcal{C}_1$ could be missed.

There is no way to solve this problem. If two components interact through synchronized operations, there is no way to separate the temporal behavior of the two. Therefore, we need to take this blocking time into account during the integration phase. Our analysis is detailed in the next section.

Another problem is how to reduce the blocking time. In fact, a particular problem, called "Priority Inversion" may arise. The problem was solved by Sha, Rajkumar and Lehoczky in their seminal paper [20], in the context of fixed priority scheduling.

Their work has been recently extended to resource reservation algorithms by Lamastra, Lipari and Abeni [7] that proposed the Bandwidth Inheritance Algorithm (BWI). When a thread tries to execute an operation (provided or required), it may get blocked. According to the BWI protocol, a thread that holds a lock on an operation and blocks another server from executing, may "inherit" the capacity of the blocked server. In this way, the blocking time is "transformed" into an interference time I_i for each component. In [7], the authors propose a methodology for computing the interference time of a server in the case of single-thread applications. We extended the methodology to multi-threaded applications: unfortunately, in the case of multi-threaded applications, the worst case interference time can be very high. In fact, in the case of single-thread applications, it is possible to make safe assumptions on the maximum of times threads can interfere in each period of the server, and this allows to reduce the pessimism in computing the interference. In the case of multi-threaded applications, such assumptions do not hold anymore, and a thread can be blocked by a large number of threads on every operation. We are currently studying a way of reducing this interference time by changing the protocol.

The interference must be taken into account during the integration phase to check is all components will respect their temporal constraints.

5.4 System integration

Our system is composed of many components interacting through synchronized operations. After each component has been developed individually, we

can analyze the temporal behavior of the entire system by using the following steps:

(i) First, we analyze each component in isolation. Given the characteristics of the threads, and the local scheduling algorithm, we can compute the demand bound function as described in [10]. However, if the thread uses some synchronized required operations ω_j or provided operation π_i, we cannot know yet how long the operation will take and how much blocking time the thread can experience in the worst case. Therefore, we start by considering these terms as unknown variables. The duration of operations ω_i and π_i are denoted with $d(\omega_i)$ and $d(\pi_i)$, respectively, whereas the total interference time for the component is denoted by I_i.

(ii) The unknown variables $d(\omega_j)$ and I_i are all initialized to 0.

(iii) We now apply the methodology described in [9] to compute the server budget Q_i and the server period P_i for each component. Actually, there are many possible values for Q_i and P_i. We chose one possible value according to some heuristics. For example, in general it is convenient to assign a period that it the largest possible, to reduce the overhead of context switch.

(iv) Then, we try to integrate all components in the final system. In doing this, we can compute the duration of the synchronized required operations $d(\omega_i)$ as the length of the corresponding synchronized provided operations $d(\pi_j)$. Morevore, we can compute the interference time by considering all threads of other applications that may interfere with our server. Of course, after the computation, the resulting interference time may be greater than the interference time previously assigned. If this is the case, we must go back to step 3 and compute a new pair (Q_i, P_i). The value of the interference time at each iteration can only increase, because at each iteration we increase the utilization factor Q_i/P_i of each server, and the interference time I_i is monotonically increasing with Q_i/P_i. The iterative process stops when the value of the interference time is equal to the value computed at the previous iteration step, or when no feasible solution can be found (i.e. $\sum_i Q_i/P_i > 1$).

The methodology described above is quite pessimistic, since the interference time computation must take into account the worst case pattern of requests of one operation, when many threads on different components try to execute a synchronized operation at the same time. We are currently investigating alternative solutions to the BWI protocol to reduce the interference time.

6 Implementation

SHaRK (Soft and Hard Real-time Kernel) is an open source real-time operating system [3] designed for modularity and extensibility. It has been developed over the years as a kernel for didactic purposes. The goal is to teach how to implement new scheduling algorithms on a real-time system, and to compare the performance of different scheduling algorithm. For these reasons, SHaRK allows programmers to define their own algorithm.

The structure of SHaRK is based on the concept of *module*. A module is a small component that provides an interface toward a generic scheduling mechanism and an interface to the threads. Every thread, upon creation, is assigned to a module. Modules are organized on a pile, and each module is assigned a unique order number.

When the generic scheduling mechanism has to take a scheduling decision, it invokes the appropriate function of the module with the lowest order number. In the case that the module has no ready thread to serve, the next module's function is asked, until a module is found that has one ready thread to scheduler.

The structure of the modules allows the coexistence of different scheduling algorithms in the same system. This characteristic has been extended to support hierarchical scheduling as described in Section 5.2. In this configuration,

- one module implements the global scheduling mechanisms (CBSSTAR). This module has the lowest order number (0). Therefore, the generic scheduler mechanism will ask this module for taking a scheduling decision.
- For each component (application), a dedicated module implements the local scheduling mechanism. Each module implements the local scheduling strategy. It communicates with the CBSSTAR scheduling module to select which thread has to be scheduled when a certain component is selected to execute.

For example, in the configuration of Figure 2, we have 4 modules: the CBSSTAR module, which implements the global scheduler; a FIFO scheduling module which implements the local scheduling policy for component "application A"; a Round Robin scheduling module for component "application B"; and a fixed priority scheduling module for component "application C". Moreover, the BWI algorithm [7] has been implemented to allow components to access synchronized operations.

SHaRK provides an interface to the user to define a component. This interface is the result of a joint work with the University of Cantabria and it was implemented with the financial support of the FIRST (Flexible Inte-

grated Real-Time System Technology) project (IST-2001-34140). We do not report here the details of the scheduling interface, and we remand to [17] for a complete description of the API.

7 Conclusions and future work

In this paper, we described a methodology for the design and the development of component-based real-time systems. Unlike previous proposals, we propose a model of a component consisting of a set of concurrent real-time threads plus their scheduling algorithm. Therefore, we can compose components with different temporal constraints and schedulers. After describing the model of the component, we described the support needed at the operating system level. Moreover, we present the implementation of this framework on the SHaRK operating system.

As a future work, we would like to further extend the model to consider other kinds of interactions, like the use of rendez-vous operations and synchronous message passing. Moreover, we are currently investigating alternative soluitions to the BWI protocol to reduce the pessimism in the computation of the interference time due to synchronized operation.

References

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

[2] Fohler, G., *Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems.*, in: *Proceedings of the 16th Real Time System Symposium, Pisa, Italy*, 1995.

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

[4] G.Lipari and S. Baruah, *Greedy reclaimation of unused bandwidth in constant bandwidth servers*, in: *IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stokholm, Sweden, 2000.

[5] Isovic, D., M. Lindgren and I. Crnkovic, *System development with real -time components*, in: *Proc. of ECOOP2000 Workshop 22 - Pervasive Component-based systems*, Sophia Antipolis and Cannes, France, 2000.

[6] Kopetz, H., *Component-based design of large distributed real-time systems*, in: *14-th IFAC workshop on distributed computer control systems (DCCS '97)*, 1997.

[7] Lamastra, G., G. Lipari and L. Abeni, *A bandwidth inheritance algorithm for real-time task synchronization in open systems*, in: *Proceedings of the IEEE Real-Time Systems Symposium*, London. UK, 2001.

[8] Lipari, G. and S. Baruah, *A hierarchical extension to the constant bandwidth server framework*, in: *Proceedings of the Real-Time Technology and Application Symposium*, 2001.

[9] Lipari, G. and E. Bini, *Resource partitioning among real-time applications.*

[10] Lipari, G., E. Bini and G. Fohler, *A framework for composing real-time schedulers*, in: *Proceeding of Test and Analysis of Component-based Systems (TACOS-03)*, Warsaw, 2003.

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

[12] Mercer, C. W., S. Savage and H. Tokuda, *Processor capacity reserves for multimedia operating systems*, Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburg (1993).

[13] Miyoshi, A., T. Kitayama and H. Tokuda, *Implementation and evaluation of real-time java threads*, in: *18th IEEE Real-Time Systems Symposium (RTSS '97)*, San Francisco, CA, 1997.

[14] Nielsen, B. and G. Agha, *Towards reusable real-time objects*, Annals of Software Engineering **7** (1999), pp. 257–282.

[15] Oaks, S. and H. Wong, "Java Threads, 2nd Edition," O'Reilly, 1999.

[16] object management group, T., "UML profile for schedulability, Performance and time," OMG (2003).

[17] project group, T. F., *Scheduler integration definition report - version 2*, Technical report, FIRST IST project, http://www.idt.mdh.se/salsart/FIRST/ (2003).

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

[19] Saewong, S., R. Rajkumar, J. P. Lehoczky and M. H. Klein, *Analysis of hierarchical fixed-priority scheduling*, in: *Proceedings of the* 14^{th} *IEEE Euromicro Conference on Real-Time Systems*, 2002.

[20] Sha, L., R. Rajkumar and J. P. Lehoczky, *Priority inheritance protocols: An approach to real-time synchronization*, IEEE Transaction on computers **39** (1990).

[21] Stankovic, J. A., *VEST — A toolset for constructing and analyzing component based embedded systems*, Lecture Notes in Computer Science **2211** (2001), pp. 390–400.