Modular software architecture for flexible reservation mechanisms on heterogeneous resources

Michal Sojka[*,a], Pavel Píša[a], Dario Faggioli[b], Tommaso Cucinotta[b], Fabio Checconi[b], Zdeněk Hanzálek[a], Giuseppe Lipari[b]

[a]*Czech Technical University in Prague, Czech Republic*
[b]*Scuola Superiore Sant'Anna, Pisa, Italy*

Abstract

Management, allocation and scheduling of heterogeneous resources for complex distributed real-time applications is a challenging problem. Timing constraints of applications may be fulfilled by a proper use of real-time scheduling policies, admission control and enforcement of timing constraints. However, it is not easy to design basic infrastructure services that allow for an easy access to the allocation of multiple heterogeneous resources in a distributed environment.

In this paper, we present a middleware for providing distributed soft real-time applications with a uniform API for reserving heterogeneous resources with real-time scheduling capabilities in a distributed environment. The architecture relies on standard POSIX OS facilities, such as time management and standard TCP/IP networking services, and it is designed around CORBA, in order to facilitate modularity, flexibility and portability of the applications using it. However, real-time scheduling is supported by proper extensions at the kernel-level, plugged within the framework by means of dedicated resource managers. Our current implementation on Linux supports reservation of CPU, disk and network bandwidth. However, additional resource managers supporting alternative real-time schedulers for these resources, as well as additional types of resources, may be easily added.

We present experimental results gathered on both synthetic applications and a real multimedia video streaming case study, showing advantages deriving from the use of the proposed middleware. Finally, overhead figures are reported, showing sustainability of the approach for a wide class of complex, distributed, soft real-time applications.

Key words: Real-Time, Operating Systems, Embedded Systems, Distributed Systems, Middleware

1. Introduction

Soft real-time requirements are becoming pervasive in today complex distributed applications. The widespread diffusion of service oriented applications (SOA) and the advent of cloud computing have moved the core of computation from single PC to distributed systems. In this new panorama, in order for service providers to gain a competitive advantage, it is very important to provide a high level of Quality of Service (QoS) to users at minimum cost.

In contrast to hard real-time safety critical applications, in soft real-time ones timing constraints can occasionally be violated without causing a system failure. However, the quality of service provided by the system depends on the number of violated constraints and severity of such violations over a time interval of interest. Therefore, one of the prominent goals for a soft real-time system is to keep the violation of timing constraints under control.

There are many applications that require soft real-time support, both in the consumer electronics market and in the industrial application domain. Many of such applications involve video processing: for example, distributed video monitoring for video-surveillance and collection of sensible data (e.g., people counting, monitoring of parking lots, etc.). In industrial control, image recognition applications for identifying objects on conveyor belts, or to find defects in products is increasingly important. In the consumer electronics market, soft real-time support is needed for teleconferencing, video streaming distribution, and remote interactive applications of all types.

The use of classical hard real-time techniques for designing and developing such type of applications is rarely appropriate for many reasons: it requires a long and costly off-line analysis of the application requirements (execution times and similar); it makes the application design inflexible; it makes the application less robust to unexpected changes of external conditions; it ties the application to a specific hardware/software platform.

In this context, we advocate the use of real-time scheduling techniques for shared resources, and of a middleware for the management and allocation of them to the real-time applications in a distributed environment. In the proposed architecture, applications negotiate a *service contract* specifying the amount of resources that are needed for achieving the desired real-time performance. Contracts are negotiated with our framework, which is in charge of performing admission control in the distributed platform and, if the contract is accepted, of resource allocation. The access to the allocated resources is controlled

[*]Corresponding author
Email addresses: `sojkam1@fel.cvut.cz` (Michal Sojka), `pisa@fel.cvut.cz` (Pavel Píša), `d.faggioli@sssup.it` (Dario Faggioli), `t.cucinotta@sssup.it` (Tommaso Cucinotta), `f.checconi@sssup.it` (Fabio Checconi), `hanzalek@fel.cvut.cz` (Zdeněk Hanzálek), `g.lipari@sssup.it` (Giuseppe Lipari)

by run-time scheduling mechanisms.

Many integrated solutions to support soft real-time and QoS in distributed environments have been proposed in the academic literature until now (see Section 2 for a comprehensive state of the art). However, many problems still remain open that strongly limit the diffusion of such techniques.

One problem is related to the heterogeneity of hardware/-software platforms. In fact, clients are becoming more and more "embedded" and tailored to the user needs and preferences. As a consequence, a plethora of different hardware devices (smart phones, netbooks, ebooks, etc.) are used to access on-line services, each one supporting different operating systems. While it is possible to find commercial middleware solutions to support functional requirements that abstract from the specific implementation (e.g. SOAP-XML), non-functional requirements (such as soft real-time deadlines, bandwidth and throughput guarantees, power consumption, etc.) did not receive the same attention. Therefore, different APIs are provided to application developers, making it difficult to write portable real-time applications in a systematic way. Also, many applications exist that use simple heuristics to adapt themselves to changing external conditions, thus providing QoS in a best-effort way.

From the perspective of middleware developers, it is difficult to build and maintain families of products to cope with such heterogeneity of hardware, operating systems, and application techniques. This means an increased cost of development, testing and maintenance of the software.

Contributions of This Paper. To overcome these problems, in this paper we propose a modular middleware architecture to support real-time requirements and QoS in heterogeneous distributed environments. The architecture is based on a lightweight implementation of CORBA.

The architecture integrates *support for different types of hardware resources* – from CPU and disk to wired and wireless network. Support for new types of resources is provided as separated components and can be plugged in at run-time. Such components encompass the resource scheduling policies and bandwidth allocation strategies. Therefore, it is possible to *change the resource management policy* by just changing the underlying modules.

The system allows to specify real-time constraints with different levels of criticalness. The use of resource reservation techniques allows hard and soft real-time applications to coexist in the same system. The use of a Contract Broker coordinates the allocation on the different resources, thus *supporting QoS in a holistic way*.

The architecture provides distributed negotiation of new and existing applications, seamless reclaiming of spare bandwidth, and QoS management and control using custom policies.

Our architecture brings the following advantages:

- *real-time support*: the use of real-time resource allocation and scheduling strategies makes it possible to precisely specify real-time constraints;
- *uniformity*: an uniform API for the application developers, independent of the specific components;
- *modularity*: possibility to plug new resource management modules specific to a certain application needs or operating systems;
- *dynamism*: the ability to change real-time requirements dynamically as shown in e.g. Figure 14;
- *portability* of applications and middleware to different operating systems and hardware platforms.

The architecture has been implemented on the Linux Operating System, and is demonstrated on a complex case study inspired by a distributed video monitoring context. Therefore, an important contribution of this paper is constituted by the report on the experience gained in implementing this concrete application, particularly on the side of tuning the scheduling parameters for the various involved resources (CPU, network and disk).

1.1. Paper Structure

The paper is organized as follows. After reviewing related work in the research literature in Section 2, we describe in Section 3 our general software architecture for the integrated real-time scheduling of multiple heterogeneous resources. Then, in Section 4, we provide specific details about how the CPU, disk and network resource types have been plugged within the general architecture. The framework is evaluated on a real-word case study described in Section 5. Section 6 then presents the extensive experimental results gathered on both synthetic and real applications, along with the overhead figures associated to various critical parts of our framework. Finally, we draw conclusions in Section 7, and sketch out in Section 8 possible directions for future work on the topic.

2. Related Work

In this section, related work in the area of real-time support for general-purpose operating systems (GPOS) is presented. We will first give a brief overview of existing work on real-time scheduling for CPU, disk and network resources. Then, we will present existing integrated approaches to the problem and discuss the differences with the work presented in this paper.

Real-Time CPU Scheduling. For real-time scheduling of the CPU, hard real-time modifications to the Linux kernel have been proposed, like RT-Linux[1], proposed by Yodaiken et al. [2] and RTAI[2], proposed by Mantegazza et al. [13]. These approaches are more oriented to hard real-time control applications, where tight response time to interrupts and high priority activities is necessary. However, as discussed in the introduction, the use of a hard real-time scheduler cannot be regarded

[1]More information is available at `http://www.rtlinuxfree.com`.
[2]More information is available at `http://www.rtai.org`.

as an appropriate solution for supporting soft real-time applications in GPOSs.

One key feature which is usually not implemented, and that is fundamental in an *open system*, is the *temporal isolation* property [7], as provided for example by the Sporadic Server [17] scheduling policy. Without such a mechanism, a higher priority task runs undisturbed until it blocks, independently of the computation time that may have been considered at system analysis/design time. This results in the potential disruption of the guarantees offered to lower priority tasks.

In order to overcome these limitations, other approaches targeted explicitly soft real-time applications, by comprising a temporal isolation mechanism. Such an approach, which is also exploited by the work in this paper, allows for the coexistence of soft real-time and best-effort applications, all within a GPOS kernel with potentially long non-preemptive sections.

An overview of these approaches has been carried out by Gopalan in 2001 [22]. One remarkable example is the Linux/RK project [41], whose code has also been designed so as to be *portable* across multiple GPOS kernels [35], but it has been implemented on Linux only, to the best of authors' knowledge. An effort on portability of a real-time scheduler across various Operating Systems (Microsoft, Unix and Linux families), is constituted by the DSRT scheduler[3] by Nahrstedt et al. [55].

Also, soft real-time schedulers for Linux have been investigated and implemented in the context of various European Projects, like the CBS implementation on Linux developed during the OCERA Project[4], and its subsequent evolution, the AQuoSA [37] scheduler for Linux, developed during the FRESCOR Project[5]. More recently, the IRMOS Project[6] (which also supports the research results shown in this paper) is also investigating on the use of real-time scheduling for high-performance machines, with a strong focus on virtualized distributed real-time applications. To this purpose, a new multi-core scheduler for AQuoSA is being investigated [8].

The architecture presented in this paper integrates the real-time scheduling techniques for CPU, disk and network developed in the context of the FRESCOR project.

Real-Time Disk Scheduling. Various scheduling algorithms have been studied in the past for obtaining QoS guarantees and real-time performance for hard-disk access. Disk scheduling algorithms providing a predictable access to the resource can be classified in two main groups, the first one composed by real time disk schedulers, and the second one composed by proportional share disk schedulers.

Real time disk schedulers usually assume to know a deadline per each request, or assign them one, and use modified versions of Earliest Deadline First (EDF [32]) to schedule requests. As an example, Molano et al. in [33] propose JIT (Just In Time slack stealing), consisting in an EDF scheduler, modified in order to anticipate the service of requests near to the disk head position, postponing requests with smaller deadlines but with enough slack time to allow their completion even after being reordered. SCAN-EDF, introduced by Reddy et al. in [42], reorders requests with the same deadline, sorting them in SCAN order.

Proportional share disk schedulers try to allocate shares of the service (or disk service time) provided by the device to applications. As an example of this class of schedulers, we describe here YFQ (Yet another Fair Queueing), proposed by Bruno et al. in [6]. It serves requests in batches; the requests belonging to each batch are chosen using WFQ [4], and can be sorted using a SCAN or C-LOOK discipline in order to improve the disk throughput. The default Linux scheduler, CFQ (Completely Fair Queuing), belongs to this class, and uses a round robin policy to schedule the time slices it allocates to applications.

None of the proposals in the literature deals explicitly with the deceptive idleness problem, and most of them are not able to provide the guarantees they are designed for when the arrival pattern is not independent from the service they provide. We will see in Section 4.2 how BFQ, the disk scheduler used in our framework, is able to cope with these two common issues.

Real-Time Network Scheduling. There have been numerous attempts examining the possibilities for the Ethernet to enter the domain of field buses (see survey articles [18] and [12]) and there have been many protocols proposed. Some of them have been realized in industrial standards, such as DDS [36], Ethernet Powerlink [16, 47], or Profinet IO [40]. The suggested methods usually propose traffic smoothing [30, 3] or time-triggered approaches [38, 28].

Furthermore, Wireless Local Area Networks (WLANs) are very attractive for many applications since they enable fast installation with minimal maintenance costs. The IEEE 802.11 standard [26] defines the Distributed Coordination Function (DCF) which provides best-effort service at the Medium Access Control (MAC) layer. The recently accepted IEEE 802.11e standard [25] specifies the Hybrid Coordination Function (HCF) which enables prioritized and parametrized QoS at the MAC layer, on top of DCF. The HCF combines a distributed contention-based channel access mechanism, referred to as Enhanced Distributed Channel Access (EDCA) extending the DCF by multiple Access Categories (ACs), and an optional centralized polling-based channel access mechanism, referred to as HCF Controlled Channel Access (HCCA).

Widely deployed DCF and EDCA use Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) and slotted Binary Exponential Backoff (BEB) mechanism as the basic access method.

DCF and EDCA cannot provide parametrized QoS for real-time applications, unless the network is operating in non-saturated

[3]More information is available at `http://cairo.cs.uiuc.edu/software/DSRT-2/dsrt-2.html`.

[4]Open Components for Embedded Real-time Applications (OCERA), European Project n.IST2001-35102. More information is available at: `http://www.ocera.org`.

[5]Framework for Real-Time Embedded Systems based on Contracts (FRESCOR), European Project n.FP6/2005/IST/5-034026. More information is available at: `http://www.frescor.org`.

[6]Interactive Real-Time Multimedia Applications on Service-Oriented Infrastructures, European Project FP7-214777. More information is available at `http://www.irmosproject.eu`.

state [9]. The admission control algorithm is recommended but not specified in [25] in order to limit the network load. Inan et al. [27] considered the problem of multimedia capacity estimation and admission control for the EDCA function and Engelstad and Østerbø proposed [15] a probabilistic model used for delay estimation. Several authors (see [10, 54, 19]) studied how the QoS parameters of EDCA will affect the performance and proposed how to control these parameters.

Integrated approaches. Various authors focused on the design of a middleware architecture to support quality of service. García-Valls et al. proposed Hola-QoS [20], a modular and flexible software architecture to support multimedia consumer electronics applications. The goals are similar to the ones of the work presented in this paper, however they only consider control of CPU resources and do not address distributed soft-real time systems.

The Eclipse/BSD [5] Project integrates real-time scheduling of CPU, network and disk access, and exposes to applications a file-system based user-space interface. However, the project does not deal with distributed real-time applications, as we do in this paper.

Gopalan et al. [23] proposed MURALS, a distributed real-time architecture built upon TimeSys Linux[7], supporting real-time applications with end-to-end constraints making use of distributed heterogeneous resources, such as disks, CPUs and network links. Similarly, Nahrstedt et al. worked on QualMan [34], a distributed real-time resource allocation architecture supporting network, disk and memory allocation, with prototype implementation on the Solaris OS. However, in both cases the degree of modularity of the architecture is limited, and they would greatly benefit from a CORBA-oriented design.

Eide et al. [14] presented a CORBA-based middleware for the CPU management in distributed systems, however they do not consider other resources.

TAO [46] constitutes a C++ implementation of the Real-Time CORBA specification [53], which exposes the main functionality of distributed real-time applications via the CORBA paradigm. Later, TAO was integrated with QuO [29, 21], a framework that exploits the capabilities of CORBA to reduce the impact of QoS management on the application code. The result [45] is a middleware for adaptive QoS control. Recently, such an architecture has been used by Shankaran et al. within their HiDRA [48] project for hierarchical management of multiple resources in distributed real-time systems. However, these works are focused on issues related to the monitoring of the run-time application behavior, and the dynamic adaptation of resource allocations and/or application behavior to their continuously changing needs. Therefore, they consider only marginally issues related to the low-level scheduling mechanisms needed for guaranteeing the respect of timing constraints, needed by soft real-time applications. Instead, in this paper we build on real-time resource reservation scheduling strategies which allow precise allocation of the resources.

[7]More information at `http://www.timesys.com`.

It is worth to mention the architecture [11] developed in the context of the RI-MACS project for distributed real-time applications in the factory automation domain. The architecture relies on the capabilities of service-oriented infrastructures for providing discovery of resources and their real-time capabilities, self-configuration, fault-tolerance and scheduling parameters negotiation. The RI-MACS architecture also exploits AQuoSA [37] as the low-level CPU scheduler (as this work does), but it lacks both the support for multiple heterogeneous resources as the work in this paper has (wired and wireless network, CPU, disk), and the well-defined unified API enabling applications to exploit such real-time capabilities.

More recently, Rajkumar et al. [31] proposed Distributed Resource Kernels, an extension of Linux/RK adding support for distributed real-time applications. Unlike our proposal, in DistributedRK the architecture for the distributed management and allocation of resources is built inside the kernel, presumably for improved efficiency as claimed by the authors, but preventing portability. Also, in DistributedRK the architecture needs the presence within the kernel of various components that are traditionally present at a higher level, in user-space, in the system, like an HTTP server, a DNS-like server, and an NTP-like server. Implementation of such services in user-space is highly beneficial for security and robustness purposes.

3. Modular Architecture of FRSH/FORB

This section describes the internal software architecture of the FRSH/FORB framework. The basic principle of the architecture is its decomposition into four levels (see Fig. 1) — application programming interface, resource-independent level, resource-specific level, and operating system abstraction.

The *application programming interface* is used by the applications to interact with the framework. The FRSH API was developed in the context of the FRESCOR project as a portable and generic interface to provide resource reservation services to hard and soft real-time applications. The main service provided by the API is contract negotiation: the application specifies its resource requirements in the form of a contract, and submits it for negotiation to the framework. If the negotiation succeeds, the framework provides the application with a set of so called *virtual resources* (VRES), which is a generic name for resource reservations. The application then uses FRSH API services to *bind* its entities (threads, communication endpoints) to the VRESes in order to use the reserved resources. The detailed description of the API can be found in [24].

The *resource-independent level* is represented by the *contract broker* and is intended to implement algorithms for spare capacity distribution, multi-resource transactions and global QoS optimization. The contract broker interacts with the resource managers through abstract interfaces. The framework is implemented on top of a lightweight CORBA-like communication middleware called FORB [51]. This middleware hides the complexity and different nature of inter-process and inter-node communication and provides method-call semantics for remote application objects.

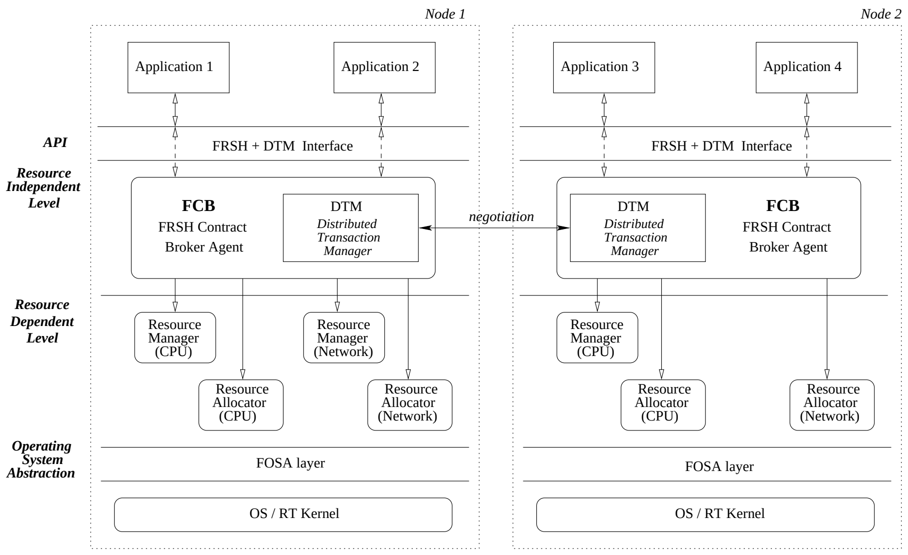


Figure 1: Logical block diagram of FRSH/FORB framework.

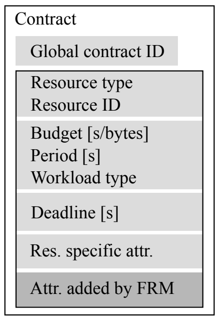


Figure 2: A contract and its attributes.

The *resource-specific level* consists of several modules called *resource managers* and *resource allocators*, which are in charge of managing the individual resources (e.g. CPU, network, disk). Their goal is to implement resource reservation scheduling algorithms supporting real-time execution and temporal isolation for tasks running on the associated resource. One key feature of our architecture is that the modules in this level can be easily plugged in and out, allowing the framework to be used on various platforms which exploit different resource reservation mechanisms.

The *operating system abstraction* facilitates portability across multiple hardware/software platforms. It consists of the FRSH Operating-System Abstraction (FOSA) layer, which implements the FOSA API. This is a cross-platform API designed within the FRESCOR project for the purpose of abstracting OS services related to the management of time, posting of timers, management of threads and synchronization primitives (e.g., signals and mutexes). The use of FOSA simplifies porting of the FRSH/FORB middleware on different Operating Systems. Indeed, FOSA has been implemented in a straightforward way on Linux by means of the POSIX API for the just mentioned services, with a few Linux-specific extensions which have been leveraged for performance reasons. Also, FOSA has been implemented on MarteOS [8] [43, 44], Partikle [9] [39] and Enea's OSE [10]. However, note that the implementation of FRSH/FORB requires usually extensions at the kernel resource scheduling level. Such extensions are forcibly OS specific and each has to be interfaced separately in the resource-specific level. However, the presence of FOSA allows for the straightforward porting of all the contract management part of the framework.

The following sections describe the individual modules in more detail as well as their interactions.

[8]`http://marte.unican.es/`
[9]`http://www.e-rtl.org/partikle/`
[10]`http://www.enea.com/Templates/Product____27035.aspx`

3.1. Resource Managers and Allocators

Let us start the description of the framework modules with FRSH *resource manager* (FRM). This module provides an admission test for the given resource. The test is usually based on some kind of schedulability analysis, and its objective is to check whether the new contract(s) can be accepted without violating the service guarantees negotiated so far. In case of mode-change (i.e. when an application, or a set of applications, change their operating mode, and need to renegotiate their contracts), the module also able to test the feasibility of the mode change. Based on the analysis, the resource manager may add a piece of information to the contract, which can be later utilized by the scheduler. An example might be a fixed-priority scheduler which schedules tasks according to the priority calculated by deadline-monotonic algorithm in the resource manager.

The second module is FRSH *resource allocator* (FRA) which always accompanies the corresponding resource manager. There can be multiple allocators for a single resource, e.g. in case of a network, there is one allocator for every network node. The purpose of the resource allocator is:

1. to interact with the resource scheduler, i.e. to create, change or cancel *virtual resources* according to "instructions" from the resource manager and contract broker;
2. to provide an API for binding application entities (threads, network endpoints, ...) to the virtual resources.

3.2. Contract Broker

The main objective of the FRSH *contract broker* (FCB) is to act as a mediator between applications and individual resources. Contract broker is a distributed application with an agent running in every node. Agents collaborate on distribution of information about resources and contracts in the whole distributed system. In the simplest case, the FCB agent only resends the contracts received in negotiation requests to the appropriate resource manager and then, if the admission test succeeds, to the resource allocator and back to the application. More details on the functionality of the contract broker are provided in Sections 3.6 and 3.7.

3.3. Examples

Figure 1 shows two nodes running the FRSH/FORB framework and connected by a network. Every node runs two (arbitrary) FRSH/FORB applications and a contract broker agent. Furthermore, node 1 runs two resource managers: one for the local CPU and one for the network. Node 2 runs only the resource manager for its local CPU. The network resource uses a centralized manager, which means that the manager runs only in one node. The figure also contains blocks representing the allocators. Note that the network resource has an allocator in every node even if the manager is located in a single node. The reason is that the virtual resource implementation must enforce the application not to use the network bandwidth beyond what was negotiated and for most networks this can be only implemented at sending side.

To illustrate the interaction of these components we present two example scenarios of the contract negotiation (a more detailed description is provided in section 3.5).

Example 1. Consider the case in which application 1 wants to use the local CPU for a periodic task, and requires a guarantee for meeting all deadlines. It prepares the contract with appropriate attributes (period, budget, deadline and resource). Then it sends the contract to the local contract broker agent. The agent finds out that the contract refers to the local CPU resource and resends the contract to the local CPU resource manager. The manager executes an admission test and returns the result (accepted/rejected) to the broker. If the contract is accepted the broker asks the resource allocator to create the virtual CPU resource according to the attributes specified in the contract.

Example 2. Application 3 wants to periodically communicate over the network with a guarantee of meeting all deadlines. Negotiation will be accomplished as follows: The application prepares a contract and sends it to the contract broker agent in node 2. The FCB agent issues a reservation request to the network resource manager running in node 1. If the contract is accepted, the FCB agent in node 2 requests the local network resource allocator to create the network virtual resource.

3.4. Representation of Contracts and Virtual Resources

In order for the framework to be modular enough to support different resources, a dynamic data structure is used to represent contracts. By dynamic we mean, that the number of attributes stored in the contract and their type can vary depending on the resource and the state of the negotiation process. The graphical representation of the contract is depicted in Figure 2. Every contract is identified by an ID which is unique in the whole distributed system.

The most common contract attributes are *budget*, *period*, *deadline* and *workload type*. The first three attributes are self explaining. The workload type describes the application workload model, which can be either *bounded* or *indeterminate*. Bounded workload means that the application has a bounded amount of work (called job) that to do during each virtual resource period, and it notifies the framework whenever the job is done. As a consequence, the framework can notify the application about overrunning its budget or about a deadline miss. Indeterminate workload model is used when there is no concept of jobs in the application.

The attributes of a contract can be set and modified not only by applications, but also by the contract broker and by resource managers. This makes the framework very flexible – for example, an application can specify only platform independent attributes in the contract. The contract broker may exploit knowledge of the underlying platform to add platform-dependent attributes, and finally the resource manager may add "instructions" for the allocator/scheduler.

Virtual resources are represented in applications by a data structure containing the negotiated contract together with any data needed by a particular resource allocator implementation to manipulate the reservation and communicate with the scheduler.

3.5. Contract Negotiation Process

In this section the negotiation process will be described briefly. For a detailed description with figures see [51].

The negotiation starts in an application by preparation of a contract and filling its attributes. Then the contract is sent to the contract broker agent by means of calling FRSH API function `frsh_contract_negotiate()`. The agent finds the resource manager and passes the contract to it. The resource manager executes an admission test to determine whether there is enough resource capacity to satisfy the application requirements. If the contract is accepted, then the resource manager can add attributes to the contract (e.g. priority which will be used to schedule an application task) and sends the modified contract back to the contract broker agent. After that, the contract broker agent sends the contract to the allocator to create a new VRES according to the contract attributes. Finally, the application is informed by the agent about the identity of the new VRES. This finishes the negotiation process, but the application usually continues with binding an entity (thread, network endpoint, ...) to the VRES through the *bind* service of the resource allocator library.

3.6. Distribution of Spare Capacity

An application can specify in the contract that it is able to make use of additional resource capacity if that is available. When the contract broker is requested to negotiate such a contract, it tries to reserve the maximum capacity requested. If that is not possible, the contract broker tries to find an optimal distribution of spare capacity among applications and reallocates the resources according to the result.

As the contracts are represented by the dynamic data structure, applications have great flexibility in specifying all possible uses of spare capacity. For example, an application can specify two different budgets in the contract and the contract broker ensures that the highest possible budget is reserved/allocated at all times. Note that resource managers and allocators always receive a simple contract, i.e. the one with only a single possible reservation. See [51] for more details on this topic.

3.7. Negotiation of Multi-Resource Transactions

Many applications operate on multiple resources. For such applications it is beneficial if a set of contracts for the different resources is negotiated as a unit – either all contracts or none of them. We call such a set of contracts *multi-resource transaction*. These transactions are negotiated similarly to what is described in section 3.5, except that resource reservations (calls to resource managers) are made for all resources in the transaction before any resource is allocated by its resource allocator. This assures that no resource is allocated before it is known that there is enough capacity on all resources participating on the transaction. Currently, only contracts without spare capacity can be negotiated in transactions. In future, we plan to remove this limitation by using global optimization techniques to find optimal distribution of spare capacity across multiple resources.

4. Supported Resources

4.1. CPU Bandwidth Management

Within the architecture described in this paper, real-time scheduling for the CPU has been supported by integrating the AQuoSA scheduler. In what follows, for the purpose of completeness, first we recall briefly basic concepts around AQuoSA, then we describe how it has been integrated within the FRSH/FORB framework.

The AQuoSA Architecture. The Adaptive Quality of Service Architecture for the Linux kernel (AQuoSA) is an open-source architecture enriching Linux with soft real-time capabilities, comprising: EDF-based scheduling, temporal encapsulation and enforcement of timing constraints, limited support for hierarchical scheduling, admission control, controllable and secure exposure of real-time capabilities to unprivileged processes, and feedback-based scheduling.

The components of AQuoSA which are relevant for this paper are the following (the reader is referred to [37] for a more comprehensive description):

- the Generic Scheduler Patch (GSP), a small patch to the kernel which allows to extend the Linux scheduler by intercepting scheduling events and executing external code in a kernel module;
- the AQuoSA real-time scheduler, a dynamically loadable kernel module which, exploiting the GSP patch, enhances the Linux CPU scheduling with an EDF-based scheduling policy, and precisely a hard-reservation version of the CBS algorithm [1].
- the AQuoSA Resource Reservation (RR) Library, which allows applications to request real-time scheduling services through a properly designed API, and forwards requests to the real-time scheduler via `ioctl()` system calls operated on a special virtual device.

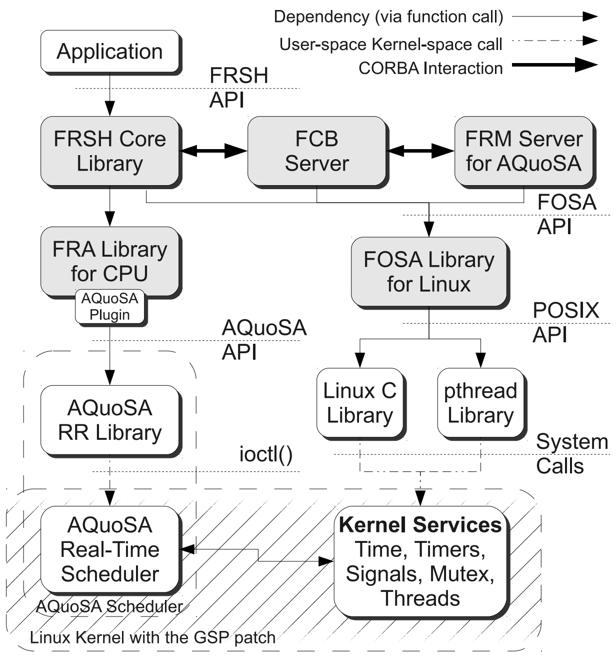


Figure 3: Integration of the AQuoSA scheduler within the FRSH/FORB architecture.

Integration of AQuoSA in FRSH/FORB. Figure 3 shows how the AQuoSA scheduler is plugged within the FRSH/FORB framework, where the grayed blocks identify software components implementing the CPU-related parts of the architecture presented in this paper. The application uses the FRSH Core API, available by linking a library. When an application negotiates a new contract, the library performs admission-control via the Contract Broker CORBA object (FCB Server), which in turn contacts the AQuoSA-specific Resource Manager. The latter performs the admission-test based on the currently admitted contracts, and the parameters provided by the application. This admission test may be potentially more complex then the simple utilization-one as currently implemented. Once the new contract has been admitted, the FRSH Core library performs the actual allocation via the FRSH Resource Allocator (FRA) library for the CPU, which has a proper plug-in for communicating with the AQuoSA scheduler via the AQuoSA user-space API. When the FRA allocates resources corresponding to a contract within AQuoSA, it sets the budget to the values indicated in the FRSH contract, or to ones scaled-up by the spare-capacity capability of the contract broker.

It must be noted that (see Figure 3), in the FRSH/FORB over AQuoSA scheme, the CORBA interactions occur only for those actions that do not have strict real-time requirements, and not for monitoring actions typically required during a real-time task activation. For example, a new contract set-up involves the FCB, FRM and FRA components, whilst reading the current budget (e.g., as needed for implementing *anytime* computing algorithms) or the server deadline are actions managed quickly through a set of function calls to the FRA library. On a related note, if configured properly, the FCB and FRM CORBA objects may be given precise scheduling guarantees within the framework, in order to provide minimum guarantees on the contract set-up time, if needed.

Note that, in the just described architecture, Linux tasks that do not use resource reservations via the FRSH API are still managed by the default Linux scheduler.

4.2. Disk Bandwidth Management

To provide individual applications with timing guarantees on disk access we have to consider that requests response time is something highly variable and dependant on physical disk parameters. Moreover, many applications (e.g., video streaming) only issue *synchronous* requests to the disk, i.e., they send the request and then block waiting for it to complete. Therefore, work conserving approaches tend to introduce a lot of seeks, as they see only one request per application, and delaying the dispatch of a request (which is done by some classes of disk schedulers) is of no help, since it actually prevents the application to issue its next ones.

The Budget Fair Queuing (BFQ [52]) algorithm is a timestamp-based proportional-share disk scheduler designed to provide strong guarantees on disk bandwidth distribution even in presence of synchronous workloads. Bandwidth distribution guarantees can be turned into soft timeliness guarantees, based on the mere knowledge of the aggregate throughput in the context of some workload scenario.

The algorithm maintains a per-application queue, and a B-WF^{2}Q+ (a slightly modified version of the Worst-case Fair Weighted Fair Queueing Plus algorithm) scheduler selects the queue to be dispatched to the disk device. Each application is also assigned a budget, representing the numbers of sectors to which it is entitled after being selected, and the scheduler involves some idling (usually referred to as *anticipation*) in case an application has no pending request but it still has some budget left.

The interested reader can find an overview of traditional elevator algorithms in [49], while BFQ is detailed in [52].

BFQ is able to provide applications with precise service guarantees. Let a_i^k be the time instant at which the i-th application issues its k-th request R_i^k, c_i^k be the time instant at which such request is completely served. We are interested in giving a worst-case upper bound to c_i^k, so we assume that all the applications are continuously backlogged in the time interval $[a_i^k, c_i^k]$, except for the i-th one, which can be subject to idling. Let also T_{agg} be the minimum aggregate throughput during $[a_i^k, c_i^k]$ under the above worst-case assumption. Let L_i^k be the size of the k^{th} request. If requests of at most Q_i sectors, with a period (or minimum inter-arrival time) of at least P_i seconds are issued, the following inequality holds:

$$c_i^k - a_i^k \leq \frac{Q_i(a_i^{k\,-}) + L_i^k}{\phi_i T_{agg}} + d(B_{max}, L_{max}, \phi_i, T_{agg}), \quad (1)$$

where $Q_i(a_i^{k\,-})$ is the sum of the sizes of the requests of the $i-th$ application not yet completed immediately before time a_i^k, B_{max} is the maximum budget size used by any application in the system, L_{max} is the maximum request size and ϕ_i is the fraction of the total disk service allocated to the application (i.e., its normalized weight). The first component represents the worst-case completion time of R_i^k in an optimal system guaranteeing no lagging behind the reserved service over any time interval, and the quantity d is the *worst-case delay* with respect to the ideal worst-case completion time.

Note that the quantity T_{agg} must be known to some extent, in order to assess the actual time guarantees provided by BFQ. The tricky aspect is that T_{agg} is in its turn a non-decreasing function of one of the components of the worst-case delay, namely B_{max}. To deal with this aspect, basing upon (1), the desired trade-off between worst-case delay and expected throughput boosting can be achieved by iteratively tuning the value of B_{max}.

Integration of BFQ in FRSH/FORB. Similarly to what happens for AQuoSA, BFQ has been integrated within FRSH/-FORB by implementing a BFQ Resource Manager and a BFQ Resource Allocator.

The BFQ Resource Manager (FRM) performs admission-test for disk contracts and lets a new one enter the system only if the service time over the period it is asking can be guaranteed. It is possible to ask for a *background contract*, which results in no service guarantees, i.e., the requests will be served when all the reserved applications are “idle”. This can be used for the applications that do not need specific disk access guarantees, so to avoid wasting some bandwidth for them.

The BFQ Resource Allocator (FRA) calculates the actual BFQ weight ϕ_i of a request associated to a contract by a bind operation, according to the following formula, derived from equations 1 above and 1 in [52]:

$$\phi_i = \frac{B_i}{P_i - \frac{2 \cdot B_{max} + L_{max}}{T_{agg}}} \quad (2)$$

where B_i and P_i are the budget and the period the contract is requesting, respectively. As said above, the worst-case aggregate throughput figures of the disk device is required, both in the admission-test (FRM) and allocation phases (FRA). For that reason, it can be either specified at FRM starting time (if known in advance), or it is automatically calculated by the system with a benchmarking procedure.

Again, new contract negotiation and first bind of an application to it require a CORBA interaction between all the framework components, and therefore should be done before starting the actual processing of the application. Yet, runtime usage of the disk — i.e., issuing read and write requests — is not affected by the overhead of contacting different components neither locally or on remote machines.

4.3. Wireless Bandwidth Management

FRSH/FORB framework supports communication over Wi-Fi networks. The part of the framework responsible for Wi-Fi resource is called *FRSH Wireless Protocol* (FWP) [51]. This protocol takes advantage of IEEE 802.11e standard [25]. More specifically it uses medium access technique called *Enhanced Distributed Channel Access* (EDCA) which provides differentiated access to medium by means of four access categories called (in decreasing “priorities”) *voice*, *video*, *best effort* and *background*. Within these categories, the classical *exponential* *back-off algorithm* is used to lower the probability of collision. Note that although EDCA improves communication capabilities for real-time applications, it still uses a probabilistic approach in the medium access algorithm and the guaranties are not "hard". FWP provides FRSH API for creating communication endpoints, binding them to virtual resources (VRES) and sending/receiving messages over them. Internally, FWP uses UDP protocol for sending the messages.

Integration of FWP in FRSH/FORB. As for any other resource, FWP implements resource manager and resource allocator components. FWP resource manager should provide schedulability analysis, however, since EDCA technique is not deterministic, the FWP resource manager cannot calculate an exact schedulability analysis. There exist probabilistic EDCA models such as [15], which could be used, but FWP uses a simpler approach which works reasonably well [50]. In short, FWP resource manager is responsible for two things:

1. It assigns the stream to the one of the four EDCA access categories according to the deadline specified by the application in the contract.
2. It checks that the overall bandwidth requested by all applications is lower than the bandwidth available. Currently, the available bandwidth is specified manually when the manager is started. For every contract, the exact time needed for transmission of the messages described by the contract is calculated. The time is divided by contract period and the results are summed up for all contracts. If the final sum (total bandwidth utilization) is less than 0.96 the contract is accepted. By keeping the bandwidth utilization under 96% the wireless channel is not fully saturated and the number of collisions is low. Therefore, we do not need complicated models to estimate EDCA back-off time and we use constant values for this delay, one for each access category.
   Currently, FWP works only when transmission bitrate is fixed. Since Wi-Fi network interface cards (NIC) normally change bitrate dynamically to cope with changing channel conditions, this constraint is quite limiting. In [51], section 3.6.2, we describe, how FRSH/FRSH framework could support dynamically changing bitrate.

FWP resource allocator creates FWP virtual resources and configures their internally used sockets in such a way that the messages are sent through the EDCA access category specified by the manager. Every FWP VRES employs a traffic limiter to ensure that applications do not send more data within a period than they requested in the contract. If the application exhausts its budget, it is either blocked until the next replenishment time (in case of synchronous send) or the message is queued and sent by VRES at the next replenishment time (asynchronous send).

5. Case Study

The proposed framework has been evaluated from the perspective of usability and achievable experimental results by realising a concrete case-study application. It is constituted by

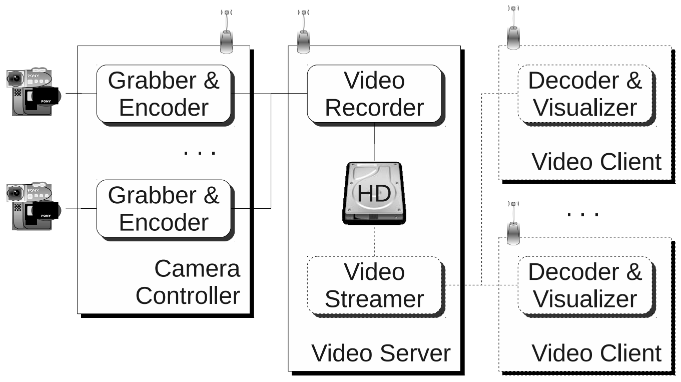


Figure 4: Case study block diagram.

a video-surveillance system with multiple cameras deployed in a building. Cameras are physically connected to the camera controller which communicates via Wi-Fi with the video server recording the video on a hard disk. The video is on-line and off-line surveyed by the operator, who dynamically decides upon the cameras to be recorded and the required quality of the video. Given the limited resources (CPU, WiFi and disk) the system presented in this paper allows the operator to dynamically (on-line) add/remove cameras and to change the video quality as long as the resource capacity is not exceeded (demonstrated in Figure 14).

The main components of the applications are the following (see Figure 4):

- the *Camera Controller* grabs videos from multiple connected video cameras, encodes them for transmission and sends them over the Wi-Fi network to the video server;
- the *Video Server* embeds two distinct components: the *Video Recorders* receive the video streams from the camera controller, re-encode them to an on-disk format and store them on a local hard drive; the *Video Streamer* reads back the stored videos and streams them over the network for being visualized by the video client(s);
- the *Video Clients* decode and visualize video streams, transmitted by the video streamer, on a local display.

In the following, we consider a concrete set-up of the general structure presented in Figure 4: one instance of the Camera Controller acquiring videos from up to three connected cameras and a single video client. This setup is depicted in Figure 5 together with resources involved in individual components.

The application has been realized by exploiting the open-source multimedia library FFMPEG[11], and the FRSH API described previously. The Video Client has been realized by using the VLC media player[12].

The video grabbing rate was selected to be 30 frames per second (FPS) and the size of one frame was 320×240 pixels.

[11]More information is available at: `http://ffmpeg.org/`.
[12]More information is available at: `http://www.videolan.org/vlc`.

Planned parameters	
Video rate	30 FPS
Video resolution	320x240
Maximal video bandwidth	1 Mbit/s
Measured parameters	
Average frame size	3192 B
Avg video bandwidth	3192*30*8 = 751 kbit/s
I-frame every	12 frames = 0.4 s
Avg (max) I-frame size	8377 (8825)
Avg (max) P-frame size	2697 (5990)
CPU load of video encoding	15 %
CPU load of video recording	6 %

Table 1: Application parameters.

The acquired video was encoded to an MPEG-4 stream with an `h263` codec and a bitrate of 1 Mbit/s. The stream was transmitted to the recording server using the real-time transport protocol (RTP)[13] which is based on the non-reliable UDP protocol. The recording server decoded each received stream, re-encoded it and stored it in MPEG-4 format onto the local disk. The video streamer is capable of streaming the recorded video either at full quality (same as used by the camera controller) or at lower quality 15 FPS, 160×120, 100 kbit/s. Due to the environmental set-up and the distance between the Camera Controller and the Video Server, the wireless link between the Camera Controller and the Video Server was operating at a fixed bitrate of 12 Mbit/s.

5.1. Parameter Tuning

The biggest difference between developing an application with and without FRSH/FORB framework is that the developers need to provide contract parameters to the framework. It should be easy for strictly periodic applications with constant workload but it is more difficult for an application involving video compression where the workload differers every period (every processed video frame). This section summarizes our experience with determining proper contract parameters.

To properly setup contract parameters for a video processing application, some knowledge of video encoding and processing is required: The video stream is composed of different types of frames (I-frame, P-frame) and each type requires different CPU processing time, network and disk bandwidth. I-frames represent the full video frames while P-frames contain only differences from the previous frame(s). In our experiments, the size of an encoded I-frame was, in average, three times bigger than the size of a P-frame.

A correct set-up of the contract parameters is obviously determined by the application parameters. The parameters affecting resources requirements have been identified and measured. They are summarised in Table 1.

A correct set-up of the contract parameters has been fine-tuned based on a benchmarking phase. It was sufficient to

[13]More information is available at: `ftp://ftp.isi.edu/in-notes/rfc3550.txt`.

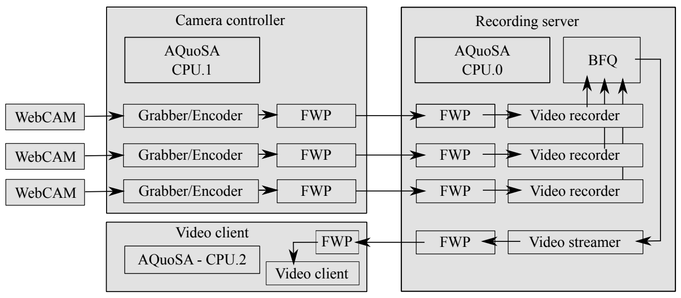


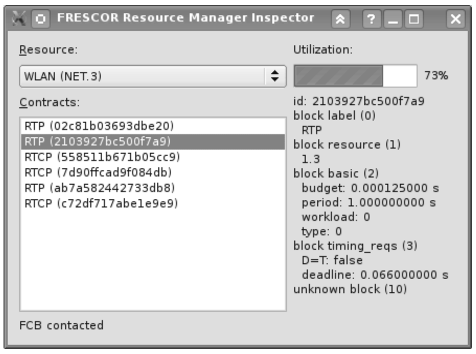
Figure 5: Detailed case study block diagram.

benchmark the individual components separately because, as can be seen from the results in Section 6.5, the framework guarantees that after integration the negotiated parameters are reserved for the components in the same way as when the components were benchmarked in isolation.

Wi-Fi contract. With the setting given in Table 1, the Wi-Fi network becomes the most limiting resource. It allows for transmission of approximately four streams, but the FWP manager admits only three streams. Although the maximal video bandwidth is 1 Mbit/s, the FWP manager needs to account for the real communication overhead (packet fragmentation, UDP and IP headers, MAC/LLC overhead – inter-frame spaces, contention window size etc.), which is in this case 47 %. Also note that every packet is transmitted two times – once from the source station to the access point (AP) and once from the AP to the destination station. Therefore we get the total used Wi-Fi bandwidth as 3 × 1 Mbit/s × 1.47 × 2 = 8.82 Mbit/s.

As a consequence of different sizes of I-frames and P-frames, if the contract period is set to match the video frame rate, and the budget is set to be big enough for processing every I-frame, then approximately 64% ($1-3192/8825$) of the reserved bandwidth would be wasted due to the low resource utilization by P-frames. Since the Wi-Fi network is the bottleneck in our scenario, it was decided to set the period in the Wi-Fi contracts to 1 second and the budget to 125 KB, which corresponds to the maximum stream bandwidth. Deadline was set to $1/30$ seconds so that the proper EDCA access category was used by FWP. The exact values of Wi-Fi contract attributes can be seen in the screen shot of a simple framework monitoring application in Figure 6. The list on the left side of the figure shows negotiated Wi-Fi contracts. For every video transmission there are two contracts: one for RTP protocol itself and one for accompanying RTCP protocol. The right side of the screen shot shows the attributes of the highlighted RTP contract.

CPU contract. The CPU capacity on both the camera controller and the recording server was sufficient (one stream needs on average 15% of CPU on the camera controller and 6% on the recording server). Given the maximum of three streams, we can waste some CPU bandwidth by reserving more CPU than is actually needed. The period was set to match the frame rate and the budget was set to 25% of the period on the sender side, and to 10% of the period on the receiver side. It was experimentally



```
Resource:                          Utilization:
WLAN (NET.3)                       73%
Contracts:                         id: 2103927bc500f7a9
RTP (02c81b03693dbe20)             block label (0)
RTP (2103927bc500f7a9)               RTP
RTCP (558511b671b05cc9)            block resource (1)
RTCP (7d90ffcad9f084db)              1.3
RTP (ab7a582442733db8)             block basic (2)
RTCP (c72df717abe1e9e9)              budget: 0.000125000 s
                                     period: 1.000000000 s
                                     workload: 0
                                     type: 0
                                   block timing_reqs (3)
                                     D=T: false
                                     deadline: 0.066000000 s
                                   unknown block (10)
FCB contacted
```

Figure 6: Screen shot of the graphical application for inspecting negotiated contracts in resource managers.

Camera Controller	
Grabber/encoder budget	9 ms
Grabber/encoder period = deadline	1/30 s
FWP Budget	125 kB
FWP Period	1 s
FWP Deadline	1/30 s
Recording Server	
Writer CPU budget	5 ms
Writer CPU period = deadline	1/30 s
Writer Disk budget	5 kB
Writer Disk period	1/30 s
Streamer Disk budget	5 kB
Streamer Disk period	1/30 s
Streamer FWP Budget	12 (125) kB
Streamer FWP Period	1 s
Streamer FWP Deadline	1/15 (1/30) s
Video Client	
CPU budget	5 ms
CPU period = deadline	1/30 s

Table 2: Parameter values set in the FRSH contracts. The two values for Streamer correspond to the low and full video quality.

checked that these values are sufficient even for processing the biggest I-frames.

Disk contract. The disk throughput was measured to be 22 MB/s. Therefore, storing 125 KB/s video streams represented very low load for the disk. However, disk performance depends not only on bandwidth but also on seek patterns and therefore it was very important to setup the contracts correctly. It can be seen in Figure 13 d) that the additional disk load has significant performance impact even on such low-bandwidth streams. It must be noted that in the current version of the framework, in order to get the benefit from using disk reservations, the applications must use "direct I/O" services when accessing the disk.

The disk contract period was chosen to match the frame rate and the budget was set to 5 kB.

Summary. Summarizing, the parameters for the various contracts in the FRSH API have been set-up as in Table 2. The results of experimental case study are presented in Section 6.5.

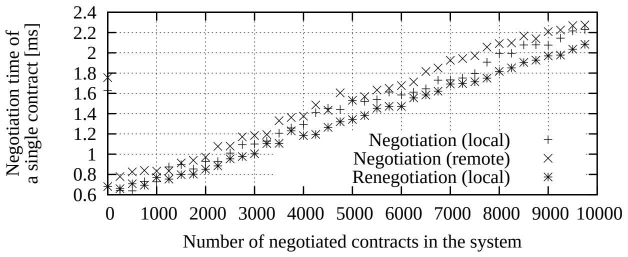


Figure 7: Contract negotiation time as a function of the number of negotiated contracts.

5.2. *Lessons Learned*

It was very helpful to have a *central view of the state of the framework*. We had a real-time monitoring application (see Figure 6) and the log of all framework operations (the excerpt is shown in Figure 14). It helped us to find quickly the reasons for reservation failures. We were able to generate the log because we setup the framework in a way that all contract negotiations went through the contract broker agent running in the recording server.

Resource reservation helped us in *discovering certain errors* earlier than during integration phase. It happened when the actually used video stream bandwidth was higher (by mistake) than was allowed by the negotiated network contract. This mistake was noticed due to jerky video on the video client. It would not be noticed without the framework because the available network bandwidth was sufficient for that single video stream.

Determining the contract parameters often requires a benchmarking phase. In our case study, this benchmarking was done manually, which is time consuming and error prone. It would be much easier if the framework provided *resource usage statistics* such as the minimum/maximum/average consumed budget, deadline miss and budget overrun counts etc. Therefore, we plan to add such functionality to the framework in the future.

6. Experiments

In this section we present experimental results for the validation of the proposed approach. The experimental validation aims to gather overhead figures for the contract negotiations in the proposed architecture, and to highlight its capabilities in the provisioning of guarantees to individual applications. The capability of the framework to temporally isolate applications from each other is shown first when applications are reserving contracts for only one type of resource. Then, we present the experimental results gathered on the integrated case-study presented in Section 5, where contracts for the three types of resources (CPU, network and disk) are all used at the same time.

All experimental results have been gathered on a Pentium 4 at 2.4 GHz with 2 GB of RAM, running a Linux OS with a 2.6.29.1 kernel patched with BFQ and AQuoSA.

6.1. *Negotiation Overhead Evaluation Experiments*

First, we measured the overhead of the negotiation procedure. To measure only the overhead of the framework and not the computation times of schedulability analysis and of VRES creation for a particular resource, we created a dummy resource, whose manager and allocator did nothing. In the experiment, we successively negotiated ten thousand contracts and measured the time of every single contract negotiation. The results are shown in Figure 7, with the lines labeled as “Negotiation”. In case of local negotiation, both contract broker, resource manager and allocator were running on the same node. For remote negotiation, the manager was running on the second computer connected by a 100 Mbps Ethernet. The result is that the remote negotiation has a slightly higher overhead (as expected) and that in both cases the negotiation time is almost linearly dependent on the number of contracts in the system.

Then, we evaluated the overhead involved in renegotiation of existing contracts. This evaluation was done similarly to the previous experiment: we had several contracts in the system and we measured the time needed to renegotiate a single contract. The result is depicted again in Figure 7, with the line labeled as “Renegotiation”. It can be seen that renegotiation takes, in average, slightly less time than the initial negotiation. The reason is that renegotiation involves less work to be done.

6.2. AQuoSA Experiments

To evaluate the behavior of the AQuoSA resource reservation component, and to validate its usage in the framework described in the paper, we used a synthetic periodic real-time application called `rt-app`[14]. Its purpose is emulating the behavior of a multimedia or control application, where computation phases (e.g., frame decoding or control action computation) and sleeping phases, waiting for the next activation instant, are regularly interleaved. The actual finishing time of each of these periodic jobs (relatively to its activation instant) is being measured in microseconds.

This section reports the results of many experiments with the following configurations:

- the number of `rt-app` instances simultaneously running in the system has been varied from 1 to 14;
- each `rt-app` instance had a random period uniformly chosen in the $[2ms, 200ms]$ range. These values have been selected since they are considered representative of typical multimedia and/or audio-video processing applications;
- each `rt-app` instance had a load approximately equal to 6%. This implies the overall system load varied from 6% to 84% during the various experiments.

Each experiment has been repeated under various scheduling policies:

- with the `SCHED_OTHER` best effort scheduling policy (labeled as *Linux be* in the figures);

[14]Available for download at: `http://aquosa.sf.net/rt-app.c`.

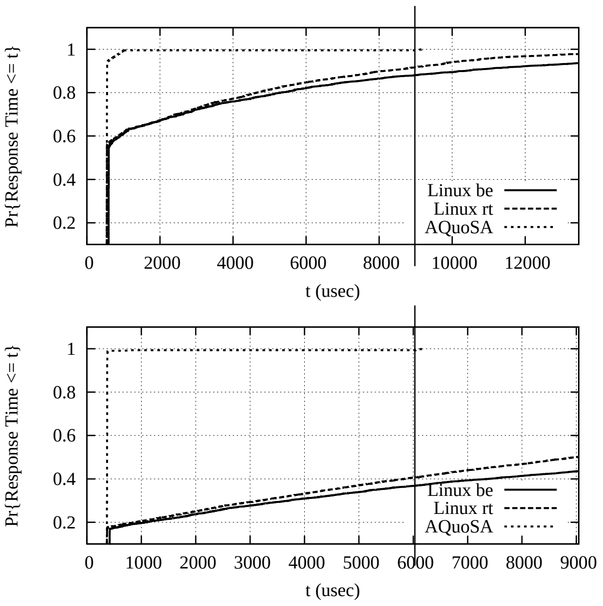


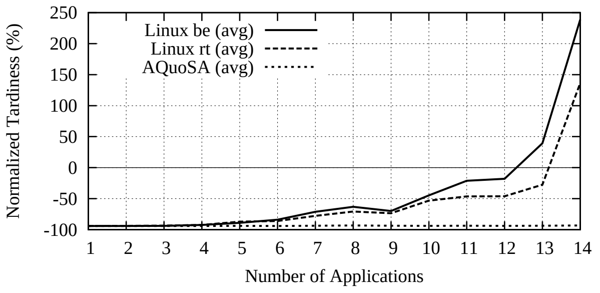
Figure 8: Cumulative distribution function of the response time for the real-time task with the shortest period under light (top figure) and heavy (bottom figure) system load, equal to $\sim 48\%$ and $\sim 84\%$, respectively.

- with the `SCHED_RR` policy at a fixed priority (*Linux rt* in the figures), thus achieving a Round-Robin scheduling policy; however, other tasks of the OS are forced in the background and cannot interfere with the `rt-app` instances, in this case;
- with the AQuoSA scheduler (*AQuoSA* in the figures), where each task has been attached a separate reservation with reservation period equal to the task period and budget tuned so to achieve a utilization of 6.6%, corresponding to a 10%, of over-provisioning with respect to the expected average application load.

In Figure 8 the cumulative distribution function of two runs of the previously described experiment are presented, in both cases for the task with the shortest period. This is because such task is the one that will suffer most, and that will more likely miss its deadline, because of the interference of other tasks. Vertical lines close to $9ms$ (top figure) and $6ms$ (bottom figure) indicate such period (equal to the relative deadline), therefore the values of the plotted functions at those instants are the probability of deadline hit for the given configuration.

It can be easily seen that, both under light and heavy system load conditions, running the task inside an AQuoSA reservation results in a deadline hit probability almost equal to one, with only few instances finishing too late. On the other hand, existing Linux scheduling policies are not able to make the task respect all its deadlines even in a lightly loaded system, and things get severely worse as long as more real-time tasks are added and the load increases.

Figure 9 depicts the normalized slack (or tardiness, in case of deadline miss) again for the `rt-app` with the smallest period, varying the total number of tasks, and consequently the system load. In formula $\frac{f_i - P_i}{P_i}$, where f_i and P_i are the fin-



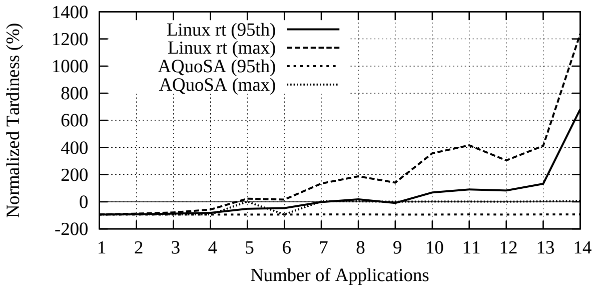


Figure 9: Average (top figure), maximum and 95th percentile (bottom figure) of the normalized slack/tardiness for the real-time task with the shortest period, varying the number of real-time tasks in the system.

ishing time and the period of the smallest period task for the $i-th$ run. Therefore, 0 means the task respected its deadline with no slack time and negative values means deadline hit with some slack time. Finally, positive values are deadline violation with some tardiness, e.g., 100% means the task finished one full period away from the deadline.

The top inset of Figure 9 shows how the slack time of the task –with standard Linux scheduling policies– decreases while the system load increases, and this seems to yield to systematic deadline miss when the number of tasks reaches 13, corresponding to an overall system load of $\sim 78\%$. However, the bottom inset shows that, even when Linux rt policies are used, the large part of the instances start missing their deadlines as the number of tasks in the system reaches 7, which is just a $\sim 42\%$ load. Moreover, the maximum experienced tardiness within the whole experiment is much worse than the average, and than the 95th percentile too, it increases with system load and reaches values higher than 1200% when the number of tasks is 14, i.e., a load of $\sim 84\%$. On the other hand, exploiting the resource reservation capabilities of AQuoSA, not only the average and the 95th percentile figures of the slack time are flattened to a constant value far below 0%, but also the maximum experienced tardiness is hardly greater than 0, showing that the task ability to meet its deadline is almost independent from the system load conditions.

6.3. BFQ Experiments

Evaluation of the BFQ performance has been done by two experiments. The first one shows that BFQ does not cause losses in the overall throughput as compared to the default Linux disk scheduler. The second one shows that BFQ improves the disk access times of delay-sensitive applications.

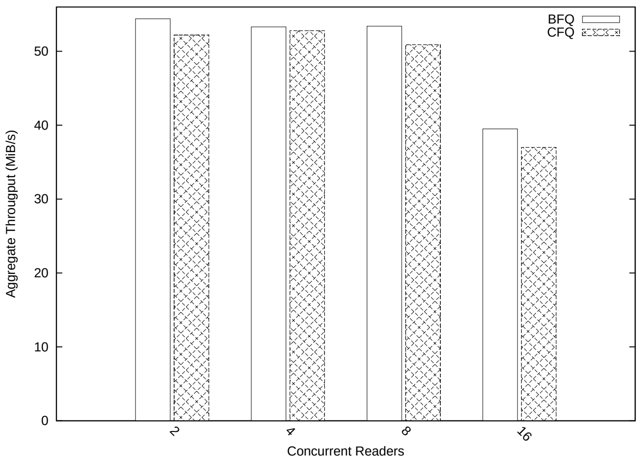


Figure 10: Comparison between the aggregate throughput of BFQ and CFQ.

Flows	BFQ	CFQ
32	23.92 ± 143.80	55.84 ± 250.90
24	12.08 ± 85.29	57.37 ± 249.13
20	6.53 ± 64.63	51.75 ± 226.76
16	4.05 ± 49.25	48.94 ± 215.46
8	2.14 ± 32.87	51.84 ± 220.11

Table 3: Streaming latency of BFQ and CFQ.

We used `fio`[15], a well known micro-benchmarking tool to collect our results. In the first experiment we set it up to execute, respectively, 2, 4, 8 and 16 parallel sequential readers, doing 32 KiB back to back reads, each one over a different file. The experiment was carried out over an `ext3` filesystem. We let each configuration run for two minutes, then we measured the aggregate throughput. As shown in Figure 10, there is no throughput loss with respect to the default Linux disk scheduler (CFQ). Actually, the experiments show that BFQ exhibits a little improvement over CFQ.

In the second experiment we configured `fio` to simulate a latency-sensitive application. We set up ten parallel greedy readers, simulating an interfering best effort load, and a varying number of parallel streamers, each one performing 32 KiB-sized reads, with a think-time of 40 ms between each iteration. The average latencies and the standard deviation on each measure are shown in Table 3. BFQ is able to serve each streamer in time, while under CFQ the streamers have to wait on the same round robin list with the background greedy readers. The big values for the standard deviation are due to the order of magnitude of the allocated slices. For example, CFQ allocates by default 100 ms time slices, so in the worst case a streamer may have to wait for ten times the slice of a greedy reader, plus any other streamer enqueued before it woke up. This kind of service is inherently subject to high variability, as the actual delay depends on the position which a newly activated flow gets in the round robin list. With BFQ, applications are served on the basis of the timestamps they get, so the effect is similar to a

[15]Available for download at `git://git.kernel.dk/fio.git`

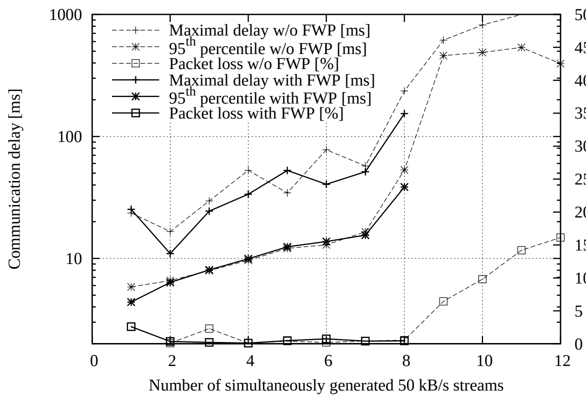


Figure 11: Illustration of how FWP resource manager maintains feasible bandwidth allocation.

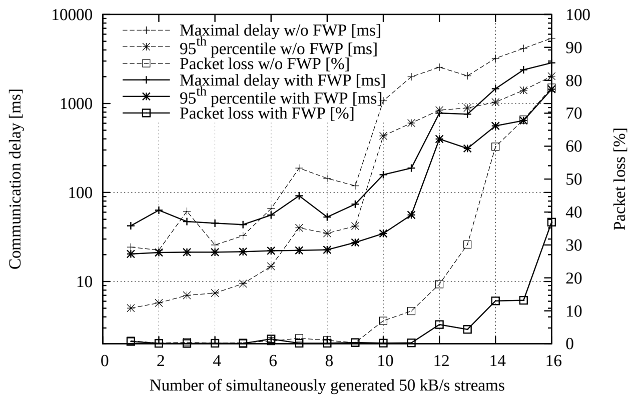


Figure 12: Demonstration of how traffic limiter in FWP VRES helps when Wi-Fi channel gets saturated.

round robin only among the latency-sensitive streamers (they all have the same timing constraints), and, as we can see from the results, only the number of streamers affects the perceived latency.

6.4. FWP Experiments

To evaluate the FWP protocol we mounted four Wi-Fi network interface cards (NICs) on our testbed PC, and an EDCA enabled Wi-Fi access point. The transmission bitrate was fixed to 12 Mbit/s. The Linux kernel was patched with *send-to-self* patch[16] which allows the messages addressed to the same computer to be sent over the external network. The messages were sent through one NIC and received through another NIC. Therefore, we did not need synchronized clocks on multiple computers to measure the communication delay.

Our testing application generated multiple data streams composed of messages with a 1024 bytes size, sent every 20 ms. The streams were received by the same application in different threads and the communication delays were measured. The messages of the i^{th} stream were sent from the $(i \mod 4)$-th NIC to the $((i+1) \mod 4)$-th NIC. Every test was run for 20 seconds so that every stream transmitted one thousand messages. We compared the results with FWP and without it.

The first experiment shows the consequence of limiting the total used bandwidth in the resource manager. The results can be seen in Figure 11. The horizontal axis shows the number of simultaneously generated streams and the vertical axis shows the maximal measured communication delay, its 95th percentile and the packet loss. From the figure, it can be seen that the communication delay increases when the utilization grows. The highest bandwidth allowed by the FWP resource manager corresponds to eight streams. When the same experiment is repeated without FWP (dashed lines), both communication delays and packet loss rise dramatically (note the logarithmic scale used for the delay axis) for nine simultaneous streams and beyond. By limiting the total bandwidth (here at eight streams), FWP is able to keep delays and packet loss low.

Also note that the maximal delay is strongly influenced by the non-determinism of the EDCA medium access algorithm and by external disturbances. This explains why the maximal delay curve relative to FWP was occasionally higher than the one without it (for five streams).

In the second experiment we highlight the influence of the traffic limiter in FWP virtual resources (see the last paragraph of Section 4.3). The previous experiment was modified so that the delay between sending of messages in one stream was not fixed to 20 ms, but was a random variable uniformly distributed between 0 and 40 ms. The results can be seen in Figure 12. In order to see the difference, we had to bypass the FWP resource manager in all experiments, because the differences showed up only when the medium was saturated which is what the manager tries to prevent (see the limit of 8 streams in Figure 11). However, such situation may happen even when the manager is in use with disturbances which lower the link quality and decrease the available bandwidth. The results show that the maximum experienced delay (lines labeled as $+$) is approximately the same with and without the traffic limiter. The difference can be found in the 95th percentile (lines labeled as $*$). For low utilization values, when the traffic limiter is active, the maximal delay is obviously close to the VRES period because some packets are delayed by the limiter. Without the limiter the delay is lower. However, the limiter helps when the medium is more saturated. For ten or more streams, the packet loss (lines labeled as $\square$) is lower with FWP than without it. Furthermore, for seven and more streams, the delay rises slower with the limiter than without it.

A careful reader may wonder why there is "non-zero" packet loss for nine and more streams in Figure 11 and in Figure 12 only for twelve and more streams (the non-dashed line in the latter figure should roughly correspond to the dashed line in the former figure). The reason is the difference in channel conditions caused by external disturbances. When the experiment was run during working hours (the first one), other Wi-Fi networks on close channels disturbed us, while the second experiment was run in the evening when other wireless traffic was lower.

[16]More information is available at `http://www.ssi.bg/~ja/#loop`.

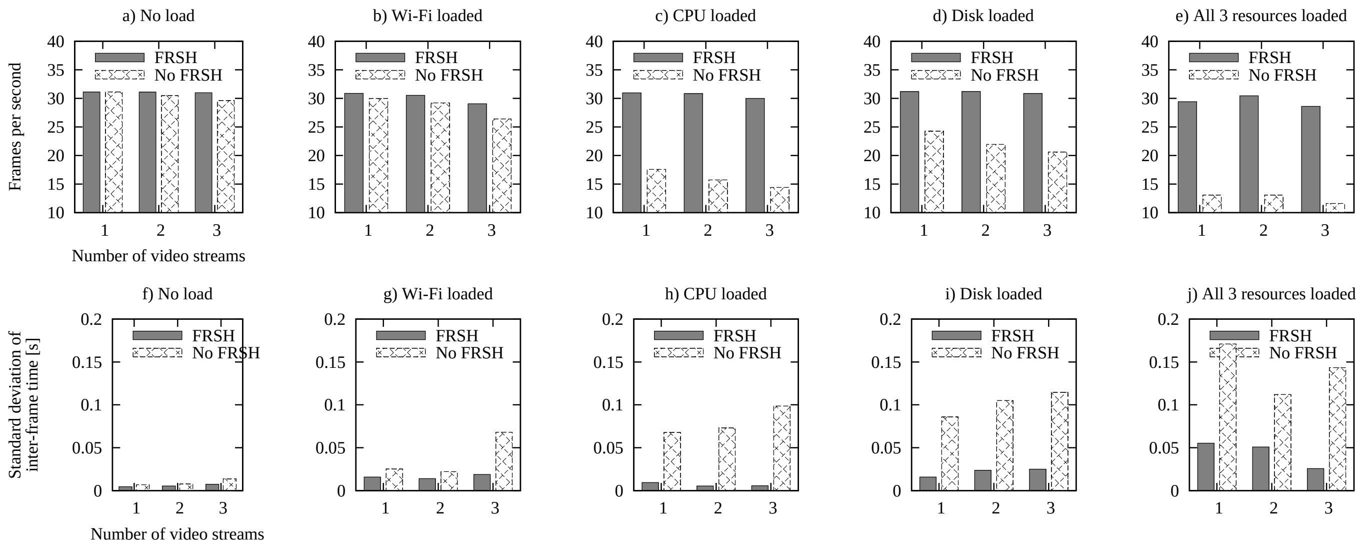


Figure 13: Results of the case study.

6.5. *Integrated Case Study*

In the case study, we ran the involved applications (described in Section 5) with and without the FRSH framework and under different loads. Every experiment lasted for 500 frames (cca 16 seconds). During those experiments several timing metrics were measured. The first metric was the average number of frames per second processed by the video recorder application. The second metric was the standard deviation of the time interval between the end of processing of two consecutive frames. The results can be seen on the graphs in Figure 13. Graphs a) and f) represent the case when all resources were loaded only by the applications of our case study. There are no significant differences in the measured frame rates, and the standard deviations show that the execution with FRSH is only slightly more regular than the one without FRSH. The reason why the measured frame rate is greater than 30 is that our cameras supplied approximately 31 frames per second even if we requested only 30 frames per second.

Graphs b) and g) show the metrics when the Wi-Fi network was loaded by a concurrently running communication. We connected two additional computers to the Wi-Fi network and let them interchange some data (all zeros) as fast as possible using the `netcat`[17] program. These communications were not under control of the FRSH framework (it can be considered and disturbances) and we setup two simultaneous streams running in opposite directions.

It can be seen that the load on the Wi-Fi channel influences the achieved frame rate. Clearly the impact increases with the number of transmitted streams but it is smaller when the FRSH framework is employed. The explanation of why the framework cannot guarantee a constant frame rate is that EDCA is not a deterministic medium access protocol and changing the EDCA access category can only increase the *probability* of

```
Time[s] Message
-------------------------------
 0.004: Waiting for requests
 0.111: Registering manager "AQuoSA" (0.0)
 0.115: Registering manager "AQuoSA" (0.1)
 0.121: Registering manager "AQuoSA" (0.2)
 0.125: Registering manager "WLAN" (1.3)
 5.219: Registering manager "Disk BFQ" (3.0)

 5.389: Negotiation request: NET.3 RTP
 5.391: Negotiation request: NET.3 RTCP
 5.396: Negotiation request: CPU.1 camera_ctrl
 5.402: Negotiation request: NET.3 RTP
 5.462: Negotiation request: NET.3 RTP
 5.463: Negotiation request: NET.3 RTCP
 5.465: Negotiation request: NET.3 RTCP
 5.468: Negotiation request: CPU.1 camera_ctrl
 5.469: Negotiation request: CPU.1 camera_ctrl

 9.259: Negotiation request: CPU.0 recorder
 9.261: Negotiation request: DISK.0 stream0.mp4
 9.565: Negotiation request: CPU.0 recorder
 9.606: Negotiation request: DISK.0 stream2.mp4
 9.622: Negotiation request: CPU.0 recorder
 9.663: Negotiation request: DISK.0 stream1.mp4

10.502: Negotiation request: CPU.2 client
10.519: Negotiation request: NET.3 RTP
10.521: Negotiation request: NET.3 RTCP
10.523: Negotiation request: CPU.0 client_streamer
10.559: Negotiation request: DISK.0 stream.mp4
13.931: Renegotiation request: CPU.0 client_streamer
13.933: Renegotiation request: NET.3 RTP
13.942: Contract(s) was/were rejected
17.235: Cancelation request: CPU.0 client_streamer
17.235: Cancelation request: DISK.0 stream.mp4
17.236: Cancelation request: NET.3 RTP
17.237: Cancelation request: NET.3 RTCP
17.240: Cancelation request: CPU.2 client

29.477: Cancelation request: CPU.0 recorder
29.477: Cancelation request: DISK.0 stream2.mp4
29.548: Cancelation request: CPU.0 recorder
29.548: Cancelation request: DISK.0 stream1.mp4
29.574: Cancelation request: CPU.0 recorder
29.575: Cancelation request: DISK.0 stream0.mp4
```

Figure 14: Log of the contract broker running in the video server.

[17] `http://netcat.sourceforge.net/`

faster medium access. On the other hand, one may wonder why the impact on the frame rate is not higher when running without FRSH. This can be explained by the netcat use of the TCP protocol, which automatically adapts its bandwidth according to the detected channel capacity. We tried to generate a more aggressive load (UDP floods) on the Wi-Fi link, but the camera controller started disconnecting from the network and the experiment could not be finished. We blame the used network adapter and/or its Linux driver for this problematic behavior.

Graphs c) and g) represent the case where the CPU on the video server was loaded by 20 additional CPU intensive non-FRSH applications. Here we can see that AQuoSA is highly successful in keeping the requested frame rate and regular execution (low variance of inter-frame times).

Similarly the disk scheduler (BFQ) achieves constant frame rate — see graphs d) and i)) — when the disk was loaded by two processes which read from two different places on the disk as fast as possible.

Finally, we ran all the three above mentioned loads simultaneously. The results are presented in graphs e) and j). The framework was able to keep the resources available for the applications in a way that no significant loss of quality was detected. The small decrease of quality can be attributed to the Wi-Fi network, which, in this case, constitutes the actual bottleneck. When the same experiment was run without the FRSH framework, the results are, as expected, very bad—only approximately 12 frames per seconds were successfully transported. Given the fact that in such a case it is very likely that the I-frames are lost, the recorded video is almost useless. With the FRSH framework, the recorded video is of good quality with only occasional small disturbances caused by dropped frames.

To highlight the dynamic nature of our framework, in Figure 14 we provide the timed log of important operations executed by the contract broker agent in the recording server, which has "connected" all resource managers needed for the case study. Shortly after the contract broker was started, five resource managers registered to it. According to Figure 4 there were three CPUs (CPU.0 – video server, CPU.1 – camera controller, and CPU.2 – video client), one disk and one Wi-Fi network. The disk resource manager probes for available disk throughput for five seconds after start and registers itself after the probe is finished. Then, at 5.38, three video steaming applications were started in the camera controller. Approximately four seconds later, three recording applications were started in the video server and they negotiated their CPU and disk contracts. A second later (10 seconds after start), the video client started on the 3rd computer to play back a formerly recorded stream. Initially, the stream was played back at low quality, but at time 13, the operator decided to increase the quality. The renegotiation happened while the old reservation was still in effect, so the video playback was not interrupted. Unfortunately, the Wi-Fi bandwidth was not available to satisfy that request so the quality remained the same until time 17 when the video client was terminated. Finally, approximately 25 seconds after the start, all the recorder applications were terminated and their reservations were canceled.

7. Conclusions

In this paper we presented a software architecture for the management of multiple heterogeneous resources shared across a set of distributed soft real-time applications. The architecture exposes to the application developers the FRSH API, which has been designed so as to allow access to real-time scheduling services of heterogeneous resources, such as CPU, disk and network, in a way that is as uniform as possible. This way, users do not need to deal with different APIs for reserving resources on the underlying OS, but they can declare the application requirements using natural attributes such as deadlines and required computation times or throughput figures, instead of priorities. This allows for an easier deployment of real-time applications over a distributed system, especially in those cases in which the system is dynamic, i.e., applications can be started dynamically by users, depending on the environmental conditions.

Our framework is effective in the provisioning of temporal isolation for distributed real-time applications that share multiple heterogeneous resources with real-time scheduling capabilities, as shown by the presented experimental results, gathered on a real implementation of the proposed framework on the Linux OS. We reported results from synthetic application workloads stressing each resource individually, in order to show the degree of achievable temporal isolation. More importantly, we reported results from a real case-study application, developed around the theme of video recording, showing the main benefits of adopting the proposed architecture. Also, we reported about our experience in how the proposed framework was used, and specifically how the resource allocation was carried on, in the context of the proposed case-study, constituting a valuable experience that can be leveraged by future researchers/developers who may want to make use of it.

Finally, we collected results about the overheads associated to our framework, comprising contract negotiation overheads. The obtained figures are sustainable for complex soft real-time embedded systems.

The framework and the case study have been released with an open-source license and they can be downloaded from project web site *http://frsh-forb.sourceforge.net*.

8. Future Work

The FRSH/FORB framework provides a bidirectional communication channel between applications and resource managers/allocators. Applications specify their requirements (e.g., in terms of possible budgets that they are capable of using) and the framework responds with precise information about what has been allocated. However, the allocation of the spare system capacity, if any, can change independently of application requests, e.g., as a consequence of another application (re)negotiating a contract or terminating. Therefore, every application that uses spare capacity needs to be able, by definition, to adapt to the changed (spare) allocation. The natural extension of this property would be that the applications adapt similarly to *changes of the resource capacity* (e.g., Wi-Fi bit-rate). For example, a video streaming application could be instructed to decrease its frame rate if the framework detects a lower Wi-Fi link quality.

Another advanced feature which was left for future work is the implementation of *multi-resource transactions* combined with spare capacity distribution. This will enable the optimal allocation of resources to applications with respect to criterions like perceived QoS or total power consumption. Continuing the example from the previous paragraph, if the Wi-Fi capacity decreases and there is enough CPU capacity, instead of lowering the frame rate, the streaming application could be instructed to use the available CPU capacity to encode the stream with a more powerful algorithm, which will also lead to the lower network utilization but with a better video quality.

Finally, there are many things which, when implemented, will make the framework more robust. One such thing is the automatic release of resources after an application crash.

Acknowledgements

Research leading to these results has been supported by the European Commission under grant agreement n.FP6/2005/IST/5-034026, in the context of the FRESCOR Project, and n.214777, in the context of the IRMOS Project. Also, it has been supported by the Ministry of Education of the Czech Republic under project number 1M0567 (CAK).

References

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

[2] Ayers, Yodaiken, B. V., 1997. Introducing real-time linux. Linux J., 5.

[3] Bello, L. L., Kaczynski, G. A., Mirabella, O., 2005. Improving the real-time behavior of Ethernet networks using traffic smoothing. IEEE Trans. Industrial Informatics 1 (3), 151–161.

[4] Bennett, J. C. R., Zhang, H., 1997. Hierarchical packet fair queueing algorithms. IEEE/ACM Transactions on Networking 5 (5), 675–689.

[5] Blanquer, J., Bruno, J., Gabber, E., Mcshea, M., Ozden, B., Silberschatz, A., Singh, A., 1999. Resource management for QoS in Eclipse/BSD. In: In Proceedings of the FreeBSD'99 Conference.

[6] Bruno, J., Brustoloni, J., Gabber, E., Ozden, B., aham Silberschatz, A., 1999. Disk scheduling with quality of service guarantees. In: ICMCS '99: Proceedings of the IEEE International Conference on Mul timedia Computing and Systems Volume II-Volume 2. IEEE Computer Society, Washington, DC, USA, p. 400.

[7] Buttazzo, G., Lipari, G., Abeni, L., Caccamo, M., 2005. Soft Real-Time Systems Predictability vs. Efficiency. No. 10.1007/0-387-28147-9-3 in Series in Computer Science. Springer.

[8] Checconi, F., Cucinotta, T., Faggioli, D., Lipari, G., June 2009. Hierarchical multiprocessor CPU reservations for the linux kernel. In: Proceedings of the 5^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2009). Dublin, Ireland.

[9] Chen, X., Zhai, H., Tian, X., Fang, Y., 2006. Supporting QoS in IEEE 802.11e wireless LANs. IEEE Transactions on Wireless Communications 5 (8), 2217–2227.

[10] Chou, C. T., Shin, K. G., Shankar N, S., Dec. 2006. Contention-based airtime usage control in multirate IEEE 802.11 wireless LANs. Networking, IEEE/ACM Transactions on 14 (6), 1179–1192.

[11] Cucinotta, T., Mancina, A., Anastasi, G. F., Lipari, G., Mangeruca, L., Checcozzo, R., Rusin'a, F., 2009. A real-time service-oriented architecture for industrial automation. (to appear on) IEEE Transactions on Industrial Informatics.

[12] Decotignie, J.-D., 2001. A perspective on Ethernet as a fieldbus. In: 4^{th} Int. Conference on Fieldbus Systems and Their Applications (FeT'01).

[13] Dozio, L., Mantegazza, P., May 2003. Real time distributed control systems using rtai. In: Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on. pp. 11–18.

[14] Eide, E., Stack, T., Regehr, J., Lepreau, J., May 2004. Dynamic CPU management for real-time, middleware-based systems. In: Proc. of 10th IEEE Real-Time and Embedded Technology and Applications Symposium. Toronto, Canada.

[15] Engelstad, P., Østerbø, O., June 2006. Analysis of the total delay of IEEE 802.11e EDCA and 802.11 DCF. In: Communications, 2006 IEEE International Conference on. Vol. 2. pp. 552–559.

[16] EPSG, October 2007. Ethernet Powerlink V2.0, Communication Profile Specification. Ethernet Powerlink Standardization Group.

[17] Faggioli, D., Mancina, A., Checconi, F., Lipari, G., October 2008. Design and implementation of a POSIX compliant sporadic server. In: Proceedings of the 10^{th} Real-Time Linux Workshop (RTLW). Mexico.

[18] Felser, M., 2005. Real-time Ethernet – industry prospective. Proceedings of the IEEE 93.

[19] Freitag, J., da Fonseca, N., de Rezende, J., Aug. 2006. Tuning of 802.11e network parameters. Communications Letters, IEEE 10 (8), 611–613.

[20] García-Valls, M., Alonso, A., Ruiz, J., Groba, A. M., 2002. An architecture of a quality of service resource manager middleware for flexible embedded multimedia systems. In: Coen-Porisini, A., van der Hoek, A. (Eds.), SEM. Vol. 2596 of Lecture Notes in Computer Science. Springer, pp. 36–55.

[21] Gill, C. D., Gossett, J. M., Corman, D., Loyall, J. P., Schantz, R. E., Atighetchi, M., Schmidt, D. C., march 2005. Integrated adaptive QoS management in middleware: A case study. Real-Time Systems 29 (2-3), 101–130.

[22] Gopalan, K., 2001. Real-time support in general purpose operating systems.

[23] Gopalan, K., Kang, K.-D., June 2007. Coordinated allocation and scheduling of multiple resources in real-time operating systems. In: Proc. of Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Pisa, Italy.

[24] Harbour, M. G., de Esteban, M. T., 2008. Architecture and contract model for integrated resources II. Deliverable of the FRESCOR project (D-AC2v2), `http://www.frescor.org/index.php?page=publications`.

[25] IEEE 802.11 — WG Reference number ISO/IEC 8802-11:1999(E), 2005. IEEE Std 802.11e.

[26] IEEE 802.11 — WG Reference number ISO/IEC 8802-11:1999(E) IEEE Std 802.11, 1999. International standard [for] information technology – telecommunications and information exchange between systems-local and metropolitan area networks-specific requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications.

[27] Inan, I., Keceli, F., Ayanoglu, E., Nov. 2009. A capacity analysis framework for the IEEE 802.11e contention-based infrastructure basic service set. Communications, IEEE Transactions on 57 (11), 3433–3445.

[28] Kopetz, H., Ademaj, A., Grillinger, P., Steinhammer, K., 2005. The time-triggered Ethernet (TTE) design. 8^{th} IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05) 00, 22–33.

[29] Krishnamurthy, Y., Kachroo, V., Karr, D. A., Rodrigues, C., Loyall, J. P., Schantz, R. E., Schmidt, D. C., 2001. Integration of QoS-enabled distributed object computing middleware for developing next-generation distributed application. In: LCTES/OM. pp. 230–237.

[30] Kweon, S.-K., Shin, K. G., Workman, G., 2000. Achieving real-time communication over Ethernet with adaptive traffic smoothing. 6^{th} IEEE Real Time Technology and Applications Symposium (RTAS'00) 00, 90.

[31] Lakshmanan, K., Rajkumar, R., 2008. Distributed resource kernels: OS support for end-to-end resource isolation. In: RTAS '08: Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium. IEEE Computer Society, Washington, DC, USA, pp. 195–204.

[32] Liu, C. L., Layland, J., 1973. Scheduling alghorithms for multiprogramming in a hard real-time environment. Journal of the ACM 20 (1).

[33] Molano, A., Juvva, K., Rajkumar, R., 2-5 Dec 1997. Real-time filesystems. guaranteeing timing constraints for disk accesses in rt-mach. Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE, 155–165.

[34] Nahrstedt, K., Chu, H.-h., Narayan, S., 1998. QoS-aware resource man-

agement for distributed multimedia applications. J. High Speed Netw. 7 (3-4), 229–257.

[35] Oikawa, S., Rajkumar, R., 1999. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In: RTAS '99: Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium. IEEE Computer Society, Washington, DC, USA, p. 111.

[36] OMG Data Distribution SIG (DDSIG), 2009. Data-distribution service for real-time systems (DDS). [on-line] `http://portals.omg.org/dds`.

[37] Palopoli, L., Cucinotta, T., Marzario, L., Lipari, G., 2009. AQuoSA — adaptive quality of service architecture. Software – Practice and Experience 39 (1), 1–31.

[38] Pedreiras, P., Almeida, L., 2002. The FTT-Ethernet protocol: Merging flexibility, timeliness and efficiency. In: Proceedings of the 14^{th} Euromicro Conference on Real-Time Systems (ECRTS'02).

[39] Peiro, S., Masmano, M., Ripoll, I., , Crespo, A., 20/11/2007 2007. PaRTiKle OS, a replacement for the core of RTLinux-GPL. Real-Time Systems Group, Polytechnic University of Valencia, Linz, Austria, p. 6. URL `http://www.e-rtl.org/partikle/fileshare/files/5/PaRTiKle-OS.pdf`

[40] Profibus International, October 2007. Application Layer protocol for decentralized periphery and distributed automation, Specification for PROFINET, IEC 61158-6-10/FDIS. Profibus International.

[41] Rajkumar, R., Juvva, K., Molano, A., Oikawa, S., 1998. Resource kernels: A resource-centric approach to real-time and multimedia systems. In: In Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking. pp. 150–164.

[42] Reddy, A. L. N., Wyllie, J., 1993. Disk scheduling in a multimedia I/O system. In: MULTIMEDIA '93: Proceedings of the first ACM international confere nce on Multimedia. ACM, New York, NY, USA, pp. 225–233.

[43] Rivas, M. A., Harbour, M. G., 2000. Early experience with an implementation of the POSIX.13 minimal real-time operating system for embedded applications. In: 25th IFAC Workshop on Real-Time Programming.

[44] Rivas, M. A., Harbour, M. G., May 2001. Marte os: An ada kernel for real-time embedded applications. In: Ada-Europe. Leuven, Belgium.

[45] Schantz, R. E., Loyall, J. P., Rodrigues, C., Schmidt, D. C., Krishnamurthy, Y., Pyarali, I., 2003. Flexible and adaptive QoS control for distributed real-time and embedded middleware. In: Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware. Springer-Verlag New York, Inc., New York, NY, USA, pp. 374–393.

[46] Schmidt, D. C., Levine, D. L., Mungee, S., 1997. The design of the TAO real-time object request broker. Computer Communications 21, 294–324.

[47] Seno, L., Vitturi, S., Zunino, C., May 2009. Analysis of Ethernet powerlink wireless extensions based on the IEEE 802.11 WLAN. IEEE Trans. on Industrial Informatics 5 (2), 86–98.

[48] Shankaran, N., Koutsoukos, X. D., Schmidt, D. C., Xue, Y., Lu, C., 2006. Hierarchical control of multiple resources in distributed real-time and embedded systems. In: ECRTS'06: Proceedings of the 18th Euromicro Conference on Real-Time Systems. IEEE Computer Society, Washington, DC, USA, pp. 151–160.

[49] Silberschatz, A., Galvin, P. B., Gagne, G., 2008. Operating System Concepts. Wiley Publishing.

[50] Sojka, M., Molnár, M., Hanzálek, Z., 2008. Experiments for real-time communication contracts in IEEE 802.11e EDCA networks. In: Factory Communication Systems. IEEE International Workshop on. pp. 89 – 92.

[51] Sojka, M., Molnár, M., Trdlička, J., Jurčík, P., Smolík, P., Hanzálek, Z., 2008. Wireless networks – documented protocols, demonstration. FRESCOR Deliverable D-ND3v2, Czech Technical University in Prague.

[52] Valente, P., Checconi, F., 2010. High throughput disk scheduling with fair bandwidth distribution. IEEE Transactions on Computers 99 (PrePrints).

[53] Wolfe, V. F., DiPippo, L. C., Ginis, R., Squadrito, M., Wohlever, S., Zykh, I., Johnston, R., 1997. Real-time CORBA. In: IEEE Real Time Technology and Applications Symposium. IEEE Computer Society, pp. 148–.

[54] Xiao, Y., Li, H., Nov. 2004. Voice and video transmissions with global data parameter control for the IEEE 802.11e enhance distributed channel access. Parallel and Distributed Systems, IEEE Transactions on 15 (11), 1041–1053.

[55] Yuan, W., Nahrstedt, K., 2003. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles. ACM, New York, NY, USA, pp. 149–163.