A unified framework for managing different resources with QoS guarantees

Luigi Palopoli, Tommaso Cucinotta,
Luca Marzario, Antonio Mancina,
Paolo Valente

Abstract

This paper describes ongoing research activities aimed at developing a unified framework by which a general purpose operative system is able to support applications with Quality of Service requirements. The work is focused on a feedback based dynamic management of the resources required by an application, where each resource is handled through the use of a Resource Reservation paradigm. This allows applications to share the access to a resource by specifying the fraction of usage, where such fractions are dynamically adapted by the system on the basis of observations made on the hosted activities.

Research in this area comprises development of both a theoretical framework for modelling applications and analysing the impact of control theoretic feedback strategies to the QoS experienced by applications, and a prototype implementation of an architecture with the ability of providing the needed functionality on a Linux Operative System.

1. Introduction

In the past few years, the use of computer based technologies has become pervasive in a wide class of applications ranging from communication (e.g., Voice-over-IP) to entertainment (e.g., video games). For many of these applications, the Quality of Service perceived by users is tightly linked with the temporal behaviour. For instance, for a video-on-demand streaming application, the framerate plays a role of paramount importance, whereas for a video-conference application the delay and the jitter are also crucial parameters. Recent research activities in this field have covered the following topics: 1) scheduling mechanisms inside the operating system such that it is possible to allocate specified fractions of resource to the competing tasks [11, 9, 7], 2) feedback based management strategies to cope with scarcely known or time-varying execution requirements of the tasks [5, 3, 10], 3) architectural solutions for the Operative Systems and middleware to support the technologies described above.

The results cited above do not offer a conclusive response to the problem of QoS management. Indeed, they miss a fundamental point: soft real-time tasks typically use resources of different types at the same time. Therefore we need a holistic approach, where scheduling requests of heterogeneous types, such as CPU, disk and/or network requests, are streamlined and managed in a holistic and coordinated way. As an example of such an application, consider a MPEG decoder. Its main routine could be the following:

```
for(;;) {
    read_frame();
    decode_frame();
}
```

From this sketch, it is possible to see that it makes little sense to provide the correct amount of CPU for the decoding part, if the application is delayed in its access to the disk or network. In general, the need for different types of resources may generate undesired effects of unexpected harshness, unless the interaction of different allocation mechanisms is not adequately accounted for. The problem becomes even more complex if adaptivity is required (i.e., feedback based allocation of resource in response to temporal variations of the workload).

Moreover, an evident shortcoming of reservation based scheduling mechanisms proposed so far is that they are strongly biased toward a particular class of applications, i.e., soft real-time systems and particularly continuous media (CM). From a wider perspective it is possible to identify three different types of applications that populate general purpose systems:

Interactive: applications like console shells, text editors and GUI-based applications;

Soft Real Time: flows of data like audio and video streams;

Batch: applications which ask for an intensive use of a resource but which do not have strict timing requirements (they are much less sensitive to delays than soft real-time ones), i.e. file-transfer applications.

Notably, the above is not a clear-cut division: indeed, applications may even change their nature, according to the inputs provided by the user and by the environment. As an example, consider a graphic application waiting for a user choice, and then starting a particular operation such as a spell-checker or a compiler. In this case an application switches from an interactive to a batch behaviour.

In our view, the operating system should be able to guarantee a specified QoS level - albeit of a different type - to all of these applications. For instance, interactive applications are sporadic in the true meaning of the adjective (i.e., they do not occur with specified frequencies) and have typically little requirements; still when they do occur the QoS experienced by the user is tightly connected with the shortness of the response time (i.e., the user expects to see a character on the screen right away, when he/she presses a key). Standard heuristics to solve this problem are commonplace in modern operating systems, but they are not able to offer any guarantee whatsoever when the system is heavily loaded. We will discuss in this paper a possible strategy to combine temporal guarantees with responsiveness for interactive applications, which is also sensitive to applications changing their behaviour as in the example above.

As far as soft real-time (e.g., Continuous Media) applications are concerned, in our prior work [5] we have shown a QoS management system built on the top of a resource reservation mechanism. In that case, we used a fluid model of the system's evolution (i.e., of the dynamic equations of the "plant" to be controlled). This model can be approximated to a desired precision by a resource reservation scheduler, paying the price of a potentially considerable overhead. In this paper, we take a more pragmatic view by building a model of the system that is closer to the implementation. The granularity used by the QoS manager is necessarily coarser but most of the attractive properties of the control schemes shown in [5] can be recovered to a satisfactory extent. Another contribution of this paper is the presentation of a middleware where these techniques are implemented. Although based on the Linux kernel, the architecture of the middleware relies on a kernel abstraction layer, which allows for easy porting to other operating systems (e.g., MAC-OSx and FreeBSD). Finally, we will show the lines of an ongoing research aimed at extending the approach to the case of applications modelled as pipelines of stages using different resources. This extension can be done both for the theoretical aspects (i.e., control design) and for the middleware.

2. Overview

Our framework is described in Figure 1 and it comprised of the following elements:

1. Resource Reservation (RR) mechanisms

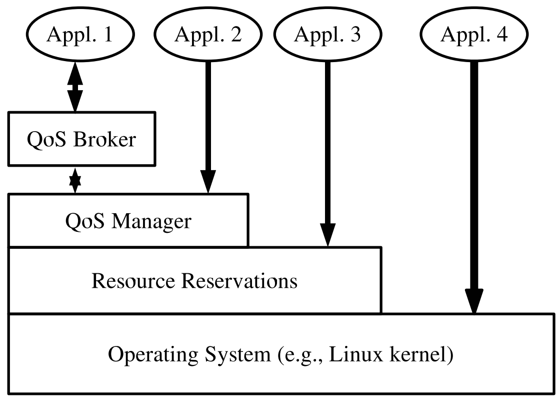

Figure 1. Main conceptual blocks

2. QoS manager
3. QoS Contract Broker

The Resource Reservation layer is the fundamental building block enabling us to control the allocation of CPU and other resources to the applications. In our idea, it should not require any kernel-specific functionality but, rather, be conceived as a sort of "plug-in" installable into any kernel by means of minimally invasive modifications. Indeed, referring to the Linux implementation, the RR is implemented in a small patch and in a pluggable module. Applications scheduled according to the resource reservation mechanism can coexist with other applications scheduled according to the standard policy used in the "host" kernel, although the latter do not receive any type of temporal guarantee. With respect to the standard view of resource reservations, our approach allows us to dynamically change the resource bandwidth alloted to each task, thus acting as a sort of "actuation" mechanism inside the kernel. Moreover, we explicitly address the problem of guaranteeing responsiveness to the interactive applications.

The QoS manager purpose is twofold. On the one hand, the QoS Manager performs an admission control: when an application requires admission, a negotiation phase with the broker determines the QoS level, if any, which can be guaranteed. The second purpose of the QoS manager is to dynamically tune the bandwidth reserved to each task for every resource it uses (in the following sections we will analyse this aspect thoroughly). The purpose of this activity is to maintain as much as possible the previously negotiated QoS levels. In case an overload occurs, the manager can decide to degrade the QoS level of any task by a call-back notification, urging the application to reduce its bandwidth utilisation.

At the QoS management level the way for measuring QoS is quite rough: the QoS is simply related to the delay with which a task accomplishes its function related to the expected termination. Moreover, the "currency" used for negotiating the admission of the application into the system is the resource utilisation required on the average by the application. This information is hardly available at the application level (a programmer typically does not make any assumption on the platform that will be used for the application), unless prior executions on the same platform have been profiled and logged. Indeed, the QoS level required by the application is associated to a set of parameters (e.g., resolution, latency, delays), whose mapping to the resource utilisation is not obvious. For this reason, we envision the use of a QoS contract broker, which is able to translate the application level requirements into kernel level parameters governing resource allocations. For example, a video player could ask for a 25 fps frame rate with a given resolution (level 1), or for the same frame rate with a lower resolution (level 2). The QoS contract broker is in charge of translating these application level parameters into resource demands (understood by the QoS Manager) and, likewise, to supervise the callback mechanism so that a bandwidth reduction required by the QoS manager is translated into the appropriate application level working mode. To the current state this component is still in the early design phase and, for this reason, it will not be referred to in this paper.

3. Resource Reservations

The problem coped with in this paper is how to effectively schedule time-shared resource in soft real-time systems. This task is to be regarded as a challenging one. Indeed, a difficult match has to be found between two different requirements: 1) ensuring a correct timing behaviour to the applications, 2) making an efficient use of the shared resource. Our strategy is based on the concept of *adaptive reservations*, which combine state-of-the-art scheduling solutions for soft real-time systems (resource reservations) and a feedback-based adaptation strategy.

In this section, we will formally introduce the problem of scheduling time-shared resources in a real-time system and present our proposed scheduling solution. Then, we will state the feedback control problem.

3.1. The real-time tasking model

A real-time task $\tau^{(i)}$ is a stream of jobs, or task instances. Each job J^i_j is characterised by an arrival time $r_j^{(i)}$, a resource utilisation time $c_j^{(i)}$, and a deadline $d_j^{(i)}$.

When a job arrives at time $r_j^{(i)}$, the task is eligible for the allotment of temporal units of the resource (e.g. CPU) from the scheduler. After the task receives $c_j^{(i)}$ time units the job finishes at a *finishing time* labelled as $f_j^{(i)}$. We will consider *preemptive* scheduling algorithms. In our model job $J_j^{(i)}$ cannot receive resource units before $f_{j-1}^{(i)}$, i.e. the activation of a job is deferred until the previous ones from the same task have been completed.

Furthermore, we will restrict to *periodic* tasks: task $\tau^{(i)}$ generates a job at integer multiples of a fixed period $T^{(i)}$ and the deadline of a job is equal to the next periodic activation: $r_j^{(i)} = d_{j-1}^{(i)} = kT^{(i)}$.

For example, a typical CPU-intensive task code structure is the following:

```
task () {
    initialisation;
    while(condition) {
        computation
        wait_for_next_instance();
    }
}
```

After initialisation, the task enters a loop where it performs the computation and then waits for the next activation. We assume that the task is activated by a periodic timer that, upon expiration, send a signal or a message that wakes up the task. Each instance of the loop is a job; the finishing time of the job is the time it invokes the `wait_for_next_instance()`.

As said, we assume that the relative deadlines of the tasks are equal to the tasks' periods. Therefore, if the task is soft real-time, a job is allowed to complete after the next activation. In this case, activations are buffered.

3.2. Temporal protection and Resource Reservations

When dealing with a multiprogrammed environment a major emphasis is usually put on the issue of *protection*. Roughly speaking, protection means that the effects of a problem in the execution of a task are confined to the task itself. The most important feature of real-time applications is that their correctness is related to their ability of meeting real-time constraints. Therefore, to cope with execution time variations and unpredictability in this type of systems, the notion of *temporal protection* plays a role of paramount importance alongside of memory protection. Restricting for the sake of simplicity to the case of independent tasks $\tau_1, \ldots, \tau_n$, a formal definition of this concept can be as follows: *a scheduling algorithm is said to guarantee temporal protection if the ability for each task* $\tau^{(i)}$ *to meet its timing constraints only depends on the evolution of the re-*

source utilisation time and inter-arrival times, and not on the other tasks' workload.

There are many scheduling algorithms known in literature that exhibit this property; examples are Proportional Share schedulers, and Resource Reservations.

In this paper, we focus on Resource Reservations, originally proposed in [9]. Reservations have been successfully applied to a variety of different resources (including CPU, disk bandwidth, network bandwidth and so forth) [9]. Therefore, even though we focus on CPU scheduling in the following, the presented approach may be applied on different kind of resources as well.

A *resource reservation* RSV_i for a task $\tau^{(i)}$ is described by a pair $(Q^{(i)}, P^{(i)})$, with the meaning that the task is *reserved* the resource for a maximum time $Q^{(i)}$ every $P^{(i)}$ units of time. $Q^{(i)}$ is the reservation *maximum budget* and $P^{(i)}$ is the reservation *period*. In general, the task $\tau^{(i)}$ needs not to be periodic; also, in case of periodic tasks, $P^{(i)}$ can be different from the task's period $T^{(i)}$.

The basic idea behind resource reservations is that every task has a limitation $Q^{(i)}$ on the amount of computation it can perform every period $P^{(i)}$. For example, if we want to keep under control a task $\tau^{(i)}$ with variable computation time, we can assign it a reservation $(Q^{(i)}, P^{(i)})$ with $P^{(i)} = T^{(i)}$ and $Q^{(i)}$ equal to some value above the average computation time of the task. The resource reservation *guarantees* that the task will never request more than $Q^{(i)}$ units of time every $P^{(i)}$. If a job $J_j^{(i)}$ needs to execute for more than $Q^{(i)}$ time units, some action must be taken to *enforce* the limitation on the execution time. There are many different algorithms that implement the resource reservation approach. Our framework is based on a derivation of the well known Constant Bandwidth Server (CBS) [1]. The interested reader is referred to the cited paper for details. For our purposes, it is sufficient to say that the CBS manipulates the deadlines to be used in an Earliest Deadline First Scheduler. The idea is very simple: at the beginning of a server period, a task is given a scheduling deadline equal to the end of the period. When a task becomes eligible for execution it starts to execute consuming its budget. This algorithm can be proved to be a possible implementation scheme for resource reservations.

As an important remark, with this scheme a reservation $RSV_i = (Q_i, P_i)$ behaves like a periodic task with worst-case computation time equal to Q_i and period equal to P_i. Therefore, to ensure the schedulability according to the selected scheduling algorithm, the following inequality has to be respected:

$$\sum^{(i)} \frac{Q^{(i)}}{P^{(i)}} \leq U^{lub} \tag{1}$$

with $U^{lub} = 1$ for the CPU. If (1) holds, every task $\tau^{(i)}$ attached to a reservation $(Q^{(i)}, P^{(i)})$ is guaranteed to receive its reserved amount of time (i.e., $Q^{(i)}$ time units over $P^{(i)}$). The ratio $B^{(i)} = \frac{Q^{(i)}}{P^{(i)}}$ is said *reserved bandwidth* and it can intuitively be thought of as the fraction of the CPU time reserved to the task.

3.3. Dealing with CM: Adaptive reservations

Resource reservations cannot be regarded as a conclusive solution to the problem of guaranteeing a correct temporal behaviour to a set of soft real-time tasks. In particular, a non-trivial problem still holds: how should the $(Q^{(i)}, P^{(i)})$ pair be dimensioned? In presence of scarcely known or and/or time-varying execution times a static partitioning of the CPU time (based on fixed choice for $(Q^{(i)}, P^{(i)})$) may lead to infeasible or inflexible choices. To address this problem, a feedback based mechanism can be used to self-tune the scheduling parameters and to dynamically reconfigure them in presence of changes in the workload.

More specifically, such a feedback mechanism can be constructed based on:

- an *actuator*, which permits to apply the feedback action to change the system behaviour;
- an *observed value*, used as input to the feedback mechanism;
- a *feedback function*, used to compute the new actuator's value, based on the observed value.

Since the feedback strategy is applied to a reservation based scheduler, the actuator is the amount of reserved time $Q^{(i)}$. In this case $Q^{(i)}$ and $B^{(i)}$ are not constant, but can change for each job: therefore, they will be indicated as $Q_j^{(i)}$ and $B_j^{(i)}$. We call the resulting abstraction an *Adaptive Reservation* [2]. To respect the consistency of the resource reservation algorithm, such changes cannot be done abruptly and attention must be paid to the current state of the servers that change their bandwidth. However important, we will not deal with this issue in this paper for evident reasons of space.

Since we aim at applying feedback control to a CPU scheduler, the observed value can be some QoS metric related to the tasks. The ideal goal of an adaptive reservation could be to schedule $\tau^{(i)}$ so that $\forall_j, f_j^{(i)} = d_j^{(i)}$. Indeed, in this way, not only is it guaranteed that the task progresses respecting its timing constraint but also the quantity of CPU alloted to the task is exactly the one the task needs. Therefore, the deviation, $f_j^{(i)} - d_j^{(i)}$ seems to be a reasonable choice as an observed value, to be used as a QoS metric. When this quantity is positive (and this is allowed to occur in a soft real-time system), we need to increase the amount of CPU reserved to the task. When it is negative, then it means that the task received CPU time in excess and we may want to decrease it. Unfortunately, due to the definition of a resource reservation given above, *it is impossible to control the exact time instant when a job finishes*. However, it is possible to control the reservation period inside which the job will finish. Assuming that $(h^i_j - 1)P_i \leq f^{(i)}_j \leq h^i_j P_i$ (i.e., the job terminates within the $h_{i,j} - th$ execution of reservation RSV_i and that the task period $T_i = n_i P_i$, we can define the scheduling error as the difference $(h^i_j - jn_i)P_i$

The *feedback function* $f()$ that is used to compute the new actuator value based on the observed value determines the closed-loop system behaviour, and must be carefully designed to attain the desired design goals. We will talk in some detail of this issue in the next section.

The System Dynamic Model A considerable advantage of the choice of the observed value as provided above is that it is possible to construct an accurate mathematical model of the system dynamic evolution. This model can be leveraged in the design of a feedback function $f()$.

Thanks to the temporal isolation property it is possible to describe the evolution of the scheduling error by the following equation:

$$\varepsilon_{k+1} = \max\{0, \varepsilon_k\} + \left\lceil \frac{c_k}{Q_k} \right\rceil P - T \tag{2}$$

where ε_k denotes the scheduling error experienced by the $k - th$ job, c_k denotes the computation time experienced by the $k - th$ job and Q_k is the budget allocated to the job. In the equation above dropped the (i) superscript referring to the task for notational convenience. The above is an approximate and simplified model that describes the evolution of the scheduling error. For practical purposes, the approximation can be regarded as a satisfactory one.

The case of multiple resources The model described above can easily be generalised to the case of *flows* managed by a pipeline of tasks. Each stage performs a partial processing and stores the results into an intermediate buffer (for instance, a stage could do the fetching of a packet, a stage could decode it and a stage could visualise the movie on the screen). Clearly, at each stage the task is periodically activated but its computation might be delayed if the data processed in the previous stage is not yet available. As far as the actuation variable is concerned, in this case we have a vector of budgets to work with (one for each stage of the pipeline). The observed variables can still be the scheduling errors experienced on each stage (assuming for each stage a deadline equal to the period). The extension of the model in Equation 2 is the following:

$$\varepsilon^r_{k+1} = \max\left\{0, \varepsilon^r_k, \varepsilon^{r-1}_{k+1}\right\} + \left\lceil \frac{c^r_k}{Q^r_k} \right\rceil P^r - T, \tag{3}$$

where the superscript r denotes the stage of the pipeline. This is a nonlinear model with a diagonal structure. This model is developed assuming that at each stage of the pipeline the task makes a prevalent utilisation of one type of resource, although it is not required that the resource used at each stage be of the same type (for instance Q^1_k could refer to the disk and Q^2_k to the CPU). Moreover we assume the presence of infinite buffers between the different stages.

3.4. Dealing with interactive applications

This mechanism has been shown to perform very well on continuous media application. However, as we said above, this is not necessarily the case for interactive applications. Consider as an example a text editor waiting for the user input; assume also that it is running in a lightly loaded system. If the application then starts a spell checker, a classic CBS strategy would grant the whole bandwidth to the reclaiming task and this would eventually end up with a deadline postponed far away, thus strongly reducing the application's priority. If the application becomes interactive once again (waiting for some other inputs), the user may experience a very low responsiveness due to low priority of the task.

The indefinite postponement of the deadline in response to a set of events like the one suggested earlier breaches the correctness of the implementation of a resource reservation scheme by the CBS and it has been solved in a fine tuning of the algorithm known as IRIS [6]. Unfortunately, the problem of lack of responsiveness still holds, depending on the choice of period lengths. Indeed,:

1. if we choose a long period P_i tasks' interference is reduced, but responsiveness can be very low (the Q_i budget can be potentially given only at the end of the period still respecting the resource reservation idea);
2. if we choose a short P_i, we get a better responsiveness paying the cost of a greater overhead

To cope with this problem, we propose and adaptive mechanism that operates on P_i. When the bandwidth required by the application grows, P_i is lowered thus allowing a fast response if the task really behaves in an "interactive" fashion. However, in the subsequent server activations P_i is increased in order to reduce the overhead if the task mutates into a batch application. The possibility of using also P_i to trade-off responsiveness against overhead discloses interesting opportunities, which are largely unexplored.

4. QoS Manager

In this section, we show a conceptual architecture for a feedback based management of the QoS experienced by the different tasks. In our model, the information on the QoS can be measured using the scheduling error $\varepsilon^{(i)}$. As shown earlier, when task $\tau^{(i)}$ is served by a reservation pair

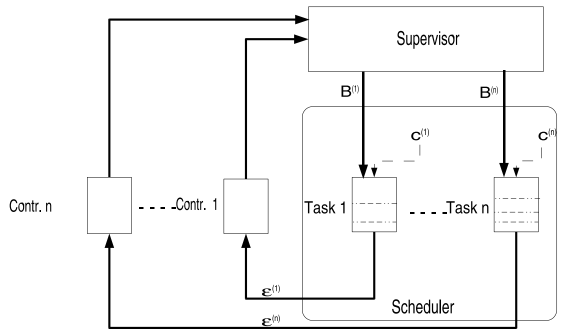

Figure 2. Decentralised control scheme

$(Q_k^{(i)}, P^{(i)})$, it generates a discrete event subsystem. The dynamic evolution of the subsystem can be succinctly described by Equation 2. In this model, $\varepsilon_k^{(i)}$ plays the role of state variable, $c^{(i)}$ is an exogenous disturbance and $q_k^{(i)}$ can be used as a command variable to steer the evolution of the system.

The subsystems related to the different tasks are very loosely coupled. Indeed, owing to the temporal isolation property, the evolution of task $\tau^{(i)}$ depends only on its own parameters and it is asynchronous from the one of the other tasks. Moreover, since we take measurements upon the termination of each job, a system-wide notion of "sampling" is entirely missing. The only global constraint is the one expressed by the consistency relation in Equation (1), which the system must never violate lest the temporal properties of the RSV mechanism be severely jeopardised.

The considerations above almost naturally dictate the control scheme in Figure 2, which is referred to as "decentralised" in the control literature. Each task is attached a dedicated controller, called "task controller", that uses only the information collected from the task. Roughly speaking, a task controller is responsible for maintaining the QoS provided by the task in specified bounds with the minimum impact on CPU utilisation. Still, the total bandwidth request from the different task controllers is allowed to exceed the bound in Equation (1). This situation is defined *overload* and it is the indirect outcome of concurrently large requests of computation from different tasks. In presence of an overload, a component called supervisor is used to reset the bandwidth allocated to the different tasks within appropriate levels respecting Equation (1). While an overload occurs, the task controllers are not allowed to work properly, since there may be alterations and/or delays in the application of the budget that they dynamically decide. For this reason, we will assume that overloads are episodic occurrences. This working condition can be attained if: 1) the number of tasks admitted into the system is kept within reasonable bounds, 2) the algorithms used for task controllers are designed to avoid excessive bandwidth requests.

4.1. Task controllers

A task controller is comprised of two components: a feedback controller and a predictor (as shown in Figure 3). An appropriate sensor located inside the RSV scheduler returns at the termination of each job the computation time of the job just concluded and the experienced scheduling error. The former information is used by the predictor to produce an estimation regarding the next computation time c_k.

We have developed both deterministic [8] and stochastic [4] approaches for control design. In the case of deterministic design, the control goal is to keep the evolution of the scheduling error confined in a small set. To attain this goal, the controller "plays" a game against the uncertainty introduced by the variability in computation time. In other words, the controller tries and counteract the variability of the computation times based on an assessment of its worst case effects. For this reason, the predictor produces an interval in which the next computation time is likely to fall, while the feedback controller decides the new budget based on the measurement of the current scheduling error and on the interval provided by the predictor. Intuitively speaking, the larger the size of the predicted range, the larger the set where it is possible to control the scheduling error.

The same conceptual structure is applicable also to stochastic approaches. In this case, the controller considers c_k as a stochastic process and tries to choose the control action to optimise some probabilistic performance metrics (conditioned to the current measurements of the scheduling error and to its past history). For instance, one possibility is to choose the control value so as to have an expected value of the scheduling error in the next step equal to 0. We called this approach stochastic dead beat (SDB) since it tries to reduce the scheduling error to 0 (in the average sense) in one step. The predictor in its turn works in a stochastic framework. For instance in the case of the SDB controller, the predictor produces a guess of the expected value of the next value of c_k conditioned to the past story of the process [4].

4.2. Supervisor

The supervisor component has the following design goals:

1. if all task controllers have bandwidth requests that do not exceed the bound in Equation 1, and are compatible with security limits imposed by the system administrator, then the supervisor has to be "transparent" (i.e.

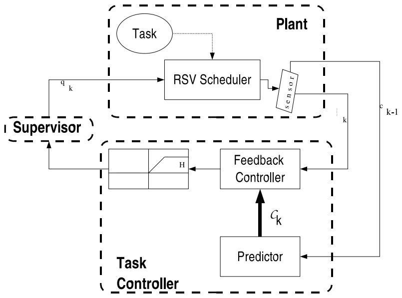

Figure 3. Block diagram of a task controller

it simply forwards to the scheduler the bandwidth requests coming from the task controllers)

2. in case of overload, the supervisor has to ensure to every task that the deviation from its required resource utilisation will not be permanent. For instance, consider the case of a deterministic control policy aiming at confining the system evolution into a set $\mathcal{I}$. In presence of overload the scheduling error may be taken afar form $\mathcal{I}$. This requirement amounts to saying that *eventually* it will return into $\mathcal{I}$.

The second design goal can be translated into guaranteeing a minimum bandwidth $b_{min}^{(i)}$ for each task, which is at least equal to its average computation requirements. Taking a pragmatic approach, we will require the enforcement of this requirement, only to the tasks for which such an estimation is available (choosing $b_{min}^{(i)} = 0$ for the remaining ones). The selection of a policy for managing in presence of overloads the bandwidth exceeding the minimum guaranteed to each task is a free design parameter for the supervisor. We have studied different alternatives (based on a prioritisation of tasks, on a weighted compression of the required bandwidths or on both), but we will not detail them here for reasons of space. The reader is referred to [5] for further details.

4.3. Possible extensions to multiple resources

The scheme proposed above can be extended to multimedia streams managed by a pipeline of tasks. Indeed, as discussed in Section (3.3), we can introduce a scheduling error ε^r for the $r-th$ stage of the pipeline and the evolution of the pipeline is describe by Equation (3). An important feature of this model is that the scheduling error at one stage only depends on the scheduling error experienced by the stage immediately before. Thereby, it is possible to operate with a dedicated controller at each stage of the pipeline. This controller takes its decision based on: 1) the scheduling error experienced by the previous job at the current stage of the pipeline ε_k^r, 2) the scheduling error ε_k^{r-1} experienced at the current job by the pipeline stage immediately before the one in consideration, 3) the prediction of the resource workload c_k^r required for the current job. In this framework, the extension of the deterministic control approaches cited earlier is straightforward. But, we are currently evaluating the possibility of different control strategies, which, for each stage, take their decision based on a global information.

5. Implementation Architecture

In this section we describe a software architecture aimed at proving the technological feasibility of the theoretical approaches described so far. The architecture is built on the top of the GNU/Linux OS (Kernel Version 2.4.27). The choice of Linux was motivated by its growing popularity and by the availability of the source code and of a large documentation on the kernel. A considerable advantage of Linux, is also its modular structure that allows one to extend the kernel by inserting a module. However, this choice is also to be considered as incidental: most of the architecture proposed here can be ported with a limited effort to other Operative Systems. More generally, our purpose was to prove that the theoretical machinery described in the previous sections can be introduced into a general purpose operating system, with a limited impact on the kernel structure and on the introduced overheads.

5.1. Design Goals and Architecture overview

The design of the system was carried out pursuing the following goals:

Portability: the link between the proposed architecture and the adoption of a specific kernel platform (Linux kernel 2.4.27) is shallow. To achieve this goal, we designed a layered structure where kernel dependent code is confined inside the lowermost level. Moreover, the changes made on the kernel are minimal and the communication between the different components of the architecture (which run partly at user and partly at kernel level) uses virtual devices, which are commonplace in operating systems of Posix class.

Backward compatibility: we did not change the API of the Linux kernel. Therefore, preexisting applications can run without modifications. Moreover, tasks that do not require QoS services are taken care of by the Linux scheduler.

Flexibility: our architecture allows one to easily introduce new control and prediction algorithms. These algorithms can be run either in user or in kernel space. In the former case, it is possible to use floating point mathematics and third parties math libraries (which is very useful during the prototyping phase). Another

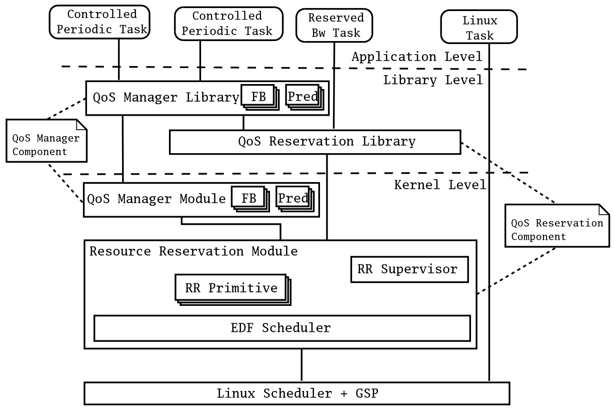

Figure 4. System Architecture

possibility potentially offered by user space implementation is to provide specific predictors along with the different applications (whose introduction into the kernel might be untrusted). On the other hand, kernel space implementation is certainly more efficient and preferable when the algorithms are well-tested and reliable (as is the case of the algorithms proposed in this paper).

Efficiency: the overhead introduced by QoS management mechanisms is acceptable. Moreover, applications that do not use QoS management functionalities experiment a negligible overhead.

Security: the possibility of changing the bandwidth reserved to the different applications makes for denial of service attacks; therefore the bandwidth allocation mechanism has to be compatible with a “maximum” CPU bandwidth defined on a per-user basis (in the same way as disk quotas are).

The proposed architecture is depicted in Figure 4, and it is composed of the following main components:

- the Generic Scheduler Patch (GSP), a small patch to the kernel (276 lines) which allows to extend the Linux scheduler functionality by intercepting scheduling events and executing external code;
- the Resource Reservation Component, composed of a kernel module and of an application library communicating through a Linux virtual device:
  - the Resource Reservation module implements the resource reservation mechanism and the RR supervisor; a set of compile-time configuration options allows one to use different Resource Reservation (RR) primitives, and to customise their exact semantics (e.g. soft or hard reservations);
  - the Resource Reservation library provides an API allowing an application to use resource reservation functions;
- the QoS Manager Component, composed of a kernel module, an application library, and a set of predictor and feedback subcomponents which may be configured to be compiled either within the library or within the kernel module:
  - the QoS Manager module offers kernel space implementations of some feedback control and prediction algorithms (including the ones shown in this paper);
  - the QoS Manager library provides an API allowing an application to use QoS management functionalities; as far as the control computation is concerned, the library either implements the control loop (if the controller and predictor algorithms are in user-space) or redirects all requests to the QoS Manager kernel module module (in case a kernel-space implementation is required). In the former case, the library communicates with the resource reservation module to take measurements of the scheduling error or to require bandwidth changes (such requests are “filtered” by the RR supervisor).

6. Conclusions and future work

In this paper we stated the general problem of adaptive management of resources for soft real-time applications. We described extensions to our prior model, which focused only on CPU allocation, in the context of applications using multiple resources. We also introduced the architecture that is being developed for supporting these mechanisms on a general purpose OS like Linux, based on an extension of our prior architecture allowing adaptation of the CPU bandwidth for QoS control.

We plan to investigate on the impact of the introduced decentralised control technique on the QoS experienced by soft real-time applications. We also plan to develop centralised control strategies, where decisions are based on the state of the entire pipeline of activities by which a multi-resource application is composed. We expect this approach to be particularly effective whenever applications possess the ability to adapt their fraction of use of a resource depending on quality parameters, as is commonly the case for multimedia applications.

Finally, even though development is primarily focused on the Linux OS, we plan to maintain portability across multiple systems (e.g. FreeBSD, on which we already have a resource reservation based management of the disk) by means of the kernel abstraction layer.

References

[1] L. Abeni. Server mechanisms for multimedia applications. Technical Report RETIS TR98-01, Scuola Superiore S. Anna, 1998.

[2] L. Abeni, L. Palopoli, and G. Buttazzo. On adaptive control techniques in real-time resource allocation. In *Proceedings of the Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, June 2000.

[3] G. T. C. Lu, J. Stankovic and S. Son. Feedback control real-time scheduling: Framework, modeling and algorithms. *Ppecial issue of RT Systems Journal on Control-Theoretic Approaches to Real-Time Computing*, 23(1/2), September 2002.

[4] T. Cucinotta, L. Palopoli, and L. Marzario. Stochastic feedback-based control of qos in soft real-time systems. In *Proc. of the IEEE 2004 conference on decision and control (CDC04)*, Paradise Island,Bahamas, December 2004.

[5] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, and L.Abeni. Adptive reservations in a linux environment. In *Proc. of the 10t10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTSA04)*, Toronto, Canada, May 2004.

[6] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. Iris: a new reclaiming algorithm for server-based real-time systems. In *Proc. of Real-Time Application Symposium (RTAS 04)*, May 2004.

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

[8] L. Palopoli, T. Cucinotta, and A. Bicchi. Quality of service control in soft real-time applications. In *Proc. of the IEEE 2003 conference on decision and control (CDC02)*, Maui, Hawai, USA, December 2003.

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

[10] D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third usenix-osdi*. pub-usenix, feb 1999.

[11] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.