A QoS Registry for Adaptive Real-Time Service-Oriented Applications

Gaetano F. Anastasi, Tommaso Cucinotta, Giuseppe Lipari
ReTiS Lab.
Scuola Superiore Sant'Anna
Pisa (Italy)
{g.anastasi, t.cucinotta, g.lipari}@sssup.it

Marisol García-Valls,
Distributed Real-Time Systems Lab.
Universidad Carlos III de Madrid
Leganés, Madrid (Spain)
mvalls@it.uc3m.es

***Abstract*—Real-time service-oriented applications are characterized by Quality of Service (QoS) requirements that cannot be properly managed by using classical real-time systems methodologies. In dynamic environments the QoS management can effectively leverage adaptive techniques, that provide flexibility and do not require a complex offline analysis. In turn, such techniques make a massive use of on-line collected data regarding the application performance and the resource requirements. Moreover, a common issue for adaptive systems is the one of deciding the initial configuration of the application and/or the run-time environment at the time of service instantiation.**

In this paper, we propose a QoS registry for coping with these issues and supporting the configuration of proper scheduling parameters in real-time Service Oriented Architectures (SOAs). The registry permits to gather QoS data related to different functional behaviors of applications, to predict the future trend based on data already collected and to permanently store such data for an effective reuse at the time of future re-instantiations. We have also built an implementation of such registry, computed its overhead costs and performed some experiments for showing the effectiveness in auto-tuning resource allocations for providing QoS guarantees in a real-time SOA.

I. Introduction

Soft real-time applications are nowadays widely used in many domains and typical examples can be image recognition applications for finding defects in industrial products or video processing for on-line conferencing systems. All these domains have recognized the benefits coming from the adoption of the service-oriented design paradigm, that permits to achieve autonomicity and interoperability ranging from the device perspective [1] to large-scale pervasive environments [2], up to reach the interactive Web [3]. Thus, soft real-time applications are often made available as services and are commonly executed in open systems, where they can be activated and terminated in any moment, generating a time-varying workload.

For these reasons, classical real-time systems design methodologies are rarely used in this context. Instead, the use of adaptive techniques is more suitable for managing the Quality of Service (QoS) of service-oriented real-time applications, given that such techniques have the advantage of not requiring an offline analysis of the application workload and provide the flexibility needed in these cases. The most common adaptive techniques can be classified in three groups: *application-level* adaptation, in which the application operating modes are adapted to the availability of resources; *resource-level* adaptation, in which the resource shares granted to the applications are adapted to the dynamic workload requirements; and *power-level* adaptation, in which the resource speed, and thus the corresponding power consumption, is adapted to the requirements of the system.

Though adaptation techniques belonging to each group can be reasonably used by themselves, a better approach for QoS control counts to use these techniques in conjunction [4] and as part of an integrated QoS framework that contains the set of required mechanisms for QoS management [5]. Moreover, a particular care must be done when providing QoS guarantees for services, as concurrent activations can easily disrupt the response time of a service [6].

Adaptive techniques for QoS management achieve self-configuration capabilities by relying on previously collected information about the application configuration, its achieved performance and the corresponding resource requirements. For example, data from previous executions may be used in a control loop for adjusting the resource allocation for future executions. However, in Service Oriented Architectures (SOAs), the application might be instantiated on a request-by-request basis by a web server, and with potentially different parameters or operating modes, making it difficult to build such an on-line control loop. Also, in presence of a multitude of operation modes and environmental conditions, it may be cumbersome to build a historical data set which is comprehensive of all the possible cases for future instantiations of the application. For these reasons, we believe that the use of a well-structured framework for handling QoS data is of paramount importance in such a context.

In this paper we present a QoS registry, called `QoSDB`, for supporting QoS management in SOAs. It can be exploited for gathering persistently QoS data related to different functional behaviors of the application (application operating modes) and for predicting the future performance based on historical data. Furthermore, a modular architecture allows for defining various models for the prediction of the resource requirements under a set of conditions which has not been observed yet. This allows for achieving a nearly correct resource allocation (self-configuration) for the application with a great reduction

of the needed observation/benchmarking points, especially in those contexts in which the space of possible configuration parameters is big (e.g., multimedia applications supporting arbitrary resolutions). In order to show viability of the proposed approach, we provide overhead measurements gathered on an implementation of the QoSDB on Linux. Moreover, through some experiments we highlight the benefits of using such registry in a real SOA scenario with QoS provisioning capabilities. By leveraging the QoSDB, the system is capable of auto-tuning for a better exploitation of internal resources while guaranteeing the QoS required by users.

In the remainder of the paper, related work is analyzed in Section II, whilst Section III describes the architecture of the proposed QoS registry. Section IV focuses on the interface exposed to the application, and the typical usage pattern. Section V shows experimental results gathered with an implementation of the proposed registry. Finally, Section VI draws conclusions.

II. Related Work

In SOA applications the importance of historical data and statistics for supporting the QoS is commonly recognized and leveraged [7], [8], [9]. However existing approaches rarely deal with real-time services in dynamic environments, where respecting time requirements imposes a particular care. For example, the use of historical data for SOAs has been also exploited by Yu and Lin [10], that propose a QoS-capable Web service architecture by deploying a QoS broker between Web service clients and providers. This broker uses QoS information collected from each server for choosing the best provider that can satisfy client requests. However, this information is mainly static and it is not used to make application-level or resource-level adaptation. Instead, our experimental work focuses on adapting the resource shares for the enforcement of certain QoS guarantees.

In the QoS management of adaptive SOAs, the use of QoS prediction mechanisms is of paramount importance for understanding the trend of QoS data and reacting to changes in the application and/or in the execution environment. In our work such aspects has been considered in the design phase and analyzed in the experimental section, however proposing novel prediction techniques is not in the scope of this paper. For the sake of simplicity, a linear regression model has been used in our experiments but, in principle, any QoS prediction technique can be embedded in the proposed work, like the approach proposed by Li et al. [11], based on forecast combination.

As a note, the QoS registry proposed in this work has been conceived for supporting service providers in the QoS management and thus its information is not produced with the intent to be published for discovery and/or integration processes. Some work in this direction has been done by Lee [12] for representing QoS information of services in UDDI (Universal Description, Discovery and Integration) registries.

In the realm of real-time systems other well-structured framework for management of historical data exist. Among them, it is worth to mention the BACC [13] (Budget ACCountant) module inside the HOLA-QoS framework [14]. That module provides the basic means for enforcing and accounting for resource usage, by notifying task overruns and by keeping statistical information on the used task budgets. It stands at the Operating System (OS) level and the information provided by it is directed to monitoring tasks for checking how a task is behaving. Instead, our framework stands at a higher level than the OS, supposing the existence of a real-time enhanced OS for the QoS enforcement. In this way, it can be used for adapting the application performance by exploiting high-level information affecting the QoS, rather than for simply checking a possible misbehavior.

For the QoS management of soft real-time applications, the use of adaptive techniques is not new and some approaches have been recognized as effective, especially for adaptive scheduling. For example, in a work by Eile et al. [15], feedback scheduling is used for the design and implementation of a CPU Broker, that adjusts allocations over time to ensure that high application-level QoS is maintained. Further, in the context of feedback-based real-time scheduling, Cucinotta et al. [16] introduced a clear separation between the prediction algorithm, responsible for estimating the workload of the subsequent task activation(s), and the control algorithm itself, leveraging the output of the predictor and the knowledge about the current task delay, for deciding the next allocation. Instead, our work does not propose any new feedback strategy but focuses on the collection and management of historical data for adaptive systems. Moreover, our proposal can be used for supporting feedback scheduling by clearly separating the feedback algorithm from the data management.

The work proposed in this paper has been inspired by the FQDB component [17], introduced in the FRESCOR[1] Application-Level Contracts (FALC) architecture [4] with the purpose of supporting feedback-based QoS control and admission policies for embedded real-time systems.

Instead, this work has been conceived for SOAs, as showed by the proposed experiments that have been carried out by integrating an implementation of QoSDB into a service-oriented QoS architecture for industrial automation [6]. Other software architectures based on modifications of the Linux kernel for supporting distributed real-time applications exist in the literature [18], [19].

III. QoSDB Description

This section describes QoSDB, a QoS registry for supporting the QoS management in adaptive SOAs. In the most simple case, QoSDB operates at the application level, shared between all the tasks of an application. In more complex architectures, a middleware acting as QoS manager could be interposed between the applications and the OS. In that case the QoSDB would stand in the middle layer and its functionalities would be better exploited by the QoS manager.

The main features of QoSDB can be summarized as follows:

[1]More information is available at: http://www.frescor.org.

- it supports adaptation techniques by counting that each application can be characterized by different modes and can run at different resource speeds;
- it allows applications to predict QoS parameters for operating modes that have not yet been experienced by the application.
- it permits to store in memory, save in a database and recover statistics related to QoS parameters;

The QoSDB has been designed pursuing the following goals.

Modularity: QoSDB is characterized by different plugins, for permitting a rapid change of its functionalities. Moreover, even the core has been designed by keeping in mind modularity for a clean competence separation. This will eventually allow programmers to modify even the internal data structures and algorithms with a minimum effort.

Flexibility: QoSDB has been designed for exploiting the three different levels of adaptation in conjunction. However, it is flexible enough for being used in many contexts, even if only a subset of the provided functionalities is required.

Efficiency: the overhead introduced by the QoSDB should be negligible. For pursuing this goal, it has been developed in the C language. This also widens the possibilities of usage in the context of resource-constrained SOAs (e.g. SOAs for industrial automation), without precluding the possibility to build gateways towards different programming languages.

A. Model and Notation

In this paper we focus on a generic application, software component or service α, which is capable of switching among a set of operating modes M. The behavior of α is affected by a set of parameters $\{\mathsf{in}_j : j = 1, 2, \ldots\}$. For example, for an image processing application, these can be the image resolution and color depth in bits; for an interactive application they can be represented by the current workload in terms of connected users. However, for the sake of simplicity, we assume that the set of parameters may be mapped to a scalar quantity $v = f(\{in_j\})$, so that the actual impact on the application behavior (in terms of resource requirements and performance) depends only on v. For example, for an application that operates on images, v could be computed as the image size ($\mathsf{width} * \mathsf{height} * \mathsf{depth}$). Thus, each operating mode $m \in M$ is characterized by a scalar quantity v_m.

Also, the historical behavior of α in each mode m is characterized in terms of a vector of observed samples sp_m with maximum dimension N_S, which constitutes a moving window over the past history of the system. Also, a vector of statistics $\mathsf{st}_m = (\mathsf{st}_{m,1}, \ldots, \mathsf{st}_{m,N_T})$ is associated with these observations, where each element of st_m is basically computed by performing certain operations on the elements of sp_m (e.g. moving average). The number of samples N_S and the number of statistic N_T can be defined directly by α.

The resources used by α can be in a different power consumption mode pm. Thus, each resource has associated a set of power modes $PM = \{\mathsf{pm}_1, \mathsf{pm}_2, \ldots, \mathsf{pm}_L\}$. For example, considering the CPU, L could be equal to the different CPU speed levels of the system. For the sake of simplicity, we consider a relationship of orthogonality between m and pm, i.e. data collected for a particular m can be reused $\forall\, \mathsf{pm} \in PM$. Thus, each element of sp_m is stored after an operation of normalization with respect to the current power mode pm_{set} and is reused after an operation of unnormalization with respect to the future power mode pm_{get}.

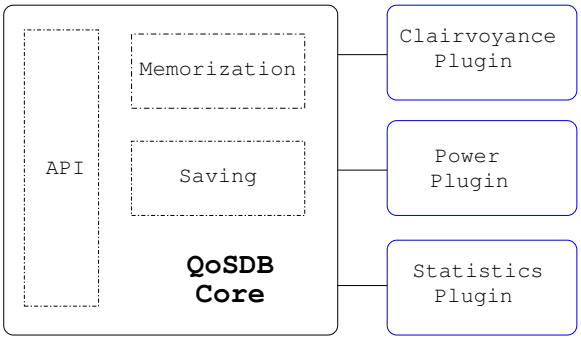

Figure 1. The QoSDB Architecture

B. Architecture

The QoSDB architecture has been designed with a particular emphasis on modularity. Its main components, depicted in Figure 1, are detailed below.

1) QoSDB Core: The core of QoSDB provides the main functionalities of the framework. It can be divided into the following subcomponents.

The *API* constitutes the glue between internal modules and plugins, by coordinating operations and information flows. Moreover, this module allows applications to interact with the QoSDB as detailed in Section IV.

The *Memorization* module provides the functionalities for storing/recalling data to/from the QoSDB data structure, that contains sampling and statistic data for each operating mode. The number of modes could be potentially high, as applications characterized by various input parameters must collapse them in a unique value (see Section III-A). For this reason, the used data structure is a Binary Search Tree (BST), with each node label corresponding to an operating mode. From a theoretical point of view, the BST would guarantee a good performance in searching and inserting nodes in the tree, that are the operations mostly performed in the QoSDB.

The *Saving* module has the task of saving in permanent storage the statistics contained in the internal data structures. This module is also responsible of restoring such statistics when a new instance is started. Data are intended to be managed by the QoSDB methods only and thus the use of a relational database has been avoided: it would add overhead and dependencies without adding much benefits. Instead, an ad-hoc database has been used.

2) QoSDB Statistics Plugin: The QoSDB Statistics Plugin allows applications to specify the statistics they are interested into. Applications define the maximum number of statistics N_T to be stored and saved for each $m \in M$, and they define the proper computation function for each statistic. The

index k used for the insertion in the vector st_m represents the statistic type. Denoting $\hat{m}$ as the current operating mode, each computation function gets as input the vector of samples $\mathsf{sp}_{\hat{m}}$ and returns the computed statistic $\mathsf{st}_{\hat{m},k}$. Actually the QoSDB comes with the average and maximum computation functions already built-in.

3) QoSDB Clairvoyance Plugin: The QoSDB Clairvoyance Plugin permits to predict a QoS statistic related to a given operating mode $\tilde{m}$ for which no statistic has been observed so far. The plugin allows applications to define the chosen algorithm according to the preferred model. Actually, the QoSDB comes with a linear regression prediction algorithm based on the Ordinary Least Square (OLS) method. A new algorithm can be inserted by simply redefining the `qosdb_clair_stat_prediction()` function, that can predict the value of a particular statistic whose type is identified by the index k. Such function computes the value $\mathsf{st}_{\tilde{m},k}$ by receiving as input the mode $\tilde{m}$ for which we predict the statistic, the remaining set of modes in the QoSDB $X = M \setminus \{\tilde{m}\}$, and the statistics of interest for the given modes $Y = \{\mathsf{st}_{m,k}\, \forall m \in X, k = \mathsf{statistic\ of\ interest}\}$.

4) QoSDB Power Plugin: This plugin allows for setting the preferred model for correlating statistics and samples to different resource speeds. It provides the `qosdb_power_normalize` method, used for store samples whose values are independent of the resource speed at which they have been taken (denoted by the pm_{set} value). It also provides the `qosdb_power_unnormalize` method, used for returning values related to the actual power mode pm_{get}, potentially different from pm_{set}. These two methods are internally called in the QoSDB API (see Section IV), respectively in the `qosdb_set_sample` and `qosdb_get` methods.

IV. QoSDB Interface

In this section a description of the available Application Program Interface (API) is given, in order to highlight the capabilities of QoSDB.

The QoSDB library allows applications to exploit its feature through a well-defined API, as described in Listing 1. The definition of these methods follows the common C programming practice of returning an exit status, represented by a `qosdb_rv` type. The functionality of each method is detailed below.

- `qosdb_init` This method initializes the QoSDB library. It is responsible of creating internal data structures and of loading historical statistics from permanent storage.
- `qosdb_cleanup` This method cleans-up the internal data structures and resources (e.g. file streams, log handler) associated to the QoSDB library.
- `qosdb_lookup_app_mode` This method looks-up an operating mode identifier. Such identifier will be used for subsequent calling exploiting the QoSDB features.
- `qosdb_set_sample` This method sets a sample for a particular mode m. It operates on the internal data structures and does not save data in permanent storage. For each observed mode m, samples are stored in a vector sp_m, implemented as a circular buffer whose dimension N_S can be set by the application.
- `qosdb_get` This method gets the vector st_m for a particular mode m. It operates on the internal data structures and does not get data from permanent storage. If a merge has been performed, the result will keep in count fresh values, otherwise it will return historical statistics.
- `qosdb_merge` This method merges new statistics with historical ones, for a particular mode m. The historical statistics are those stored in the vector st_m when the method is called; instead, the new statistics are computed on-the-fly from fresh samples contained in sp_m. As an outcome, the vector st_m is updated with the merged information (the importance of the fresh information with respect to the historical one can be weighted). Note that this method operates in the internal data structures only.
- `qosdb_save` This method saves in permanent storage the statistical data contained in the internal data structures. In particular, it permanently stores the vector of statistics st_m for each $m \in M$.

The structure of a task using the QoSDB can vary according to the application purposes. An example of the typical usage can be found in Algorithm 1, described in Section V-B, where the proposed registry is leveraged for performing QoS management in SOAs.

V. Experiments

This section describes some experiments that have been performed with the QoSDB described in the previous sections. First, we show the overheads associated with the use of the QoSDB highlighting their sustainability in a large class of target applications. Subsequently, the functionalities of the QoSDB are shown by reporting its performances in service provisioning and the effectiveness in the adaptive prediction of resource requirements.

A. Overhead Measurements

The QoSDB library has been developed pursuing efficiency, as the overhead introduced should be negligible for a proper integration in QoS architectures.

For this reason, the execution times of the QoSDB API methods has been measured by using a test program that reflects the typical usage. The test has been performed on a $64bit$ GNU/Linux system featured by an Intel CPU running at 1.2 Ghz. The average execution times, as perceived by the application, are reported in Table I as a function of the operating mode number (the t_{10} row is related to a QoSDB featured by 10 operating modes, whilst the t_{100} is related to 100 operating modes). Each test case has been repeated 50 times and reported results in the first row have the 90% confidence interval always below 6.3%, whilst 90% confidence interval in the second row is always below 5.9%.

It can be seen that the overhead is almost always negligible and in the order of microseconds. Only the `qosdb_save` method has a significant value, as data have to be saved on the hard disk. Other experiments show the same behavior when

```
qosdb_rv qosdb_init (qosdb_context **c, Database name);
qosdb_rv qosdb_cleanup (qosdb_context *c);
qosdb_rv qosdb_save (qosdb_context *c);
qosdb_rv qosdb_merge (qosdb_context *c, const qosdb_app_mode_id app_mode_id);
qosdb_rv qosdb_lookup_app_mode (qosdb_context *c, const qosdb_app_mode app_mode,
                                qosdb_app_mode_id *app_mode_id);
qosdb_rv qosdb_set_sample (qosdb_context *c, const qosdb_app_mode_id app_mode_id,
                           const qosdb_mode_sample y, const qosdb_rspeed s);
qosdb_rv qosdb_get (const qosdb_context *c, const qosdb_app_mode_id app_mode_id,
                    qosdb_mode_stat **y, const qosdb_rspeed s);
```

Listing 1. QoSDB API

Table I
EXECUTION OVERHEAD OF QoSDB API

	init	lookup	set_sample	get	merge	save
t_{10} (ms)	0.259	0.001	0.001	0.001	0.003	2.289
t_{100} (ms)	0.264	0.001	0.001	0.001	0.003	2.336

Table II
MEMORY OVERHEAD OF QoSDB

	Code	Data	
		avg	max
m_{10} (KiB)	20	159.4	172
m_{100} (KiB)	20	340.6	448

the qosdb_init method loads data from the database (in the reported experiments the database is deleted before each repetition).

As a further overhead measurement, the memory usage of a program using the QoSDB library has been measured. We did not use the information that can be gathered by the common *ps* tool, as it reports only a coarse grain information (the total amount of memory allocated for that process). Instead, we made use of the *smaps* interface present in the procfs of Linux since the 2.6.14 version. This interface permits to gather information about the actual memory reserved to the process, being also able to distinguish between the private memory and the shared memory, as due to dynamically linked libraries.

Our experiment consists in monitoring for 20 seconds (with a grain of 1 second) the output of the *smaps* interface while our program was running. The only shared libraries used by our program were libc-2.7.so and ld-2.7.so. The memory reserved for such libraries was always between 572 and 600 KiB.

Table II reports instead the amount of private memory used by the program as a function of the operating modes number (the m_{10} row is related to a QoSDB featured by 10 operating modes, whilst the m_{100} is related to 100 modes). In the first column of such table is reported the memory usage for the code section, equal to 20 KiB in the two cases. The second and the third columns correspondingly report the average and the maximum usage related to the data section. This value is variable because it also counts the heap usage, that could vary during the program execution as a consequence, for example, of malloc calls. The 90% confidence intervals are 1.4% in the case of 10 operating modes and 5.8% in the other case.

In our opinion, such values are acceptable and adequate for a wide range of applications working on Linux-capable platforms.

B. Service Provisioning Scenario

Some experiments have been performed in order to show the effectiveness of QoSDB in tuning the resource allocation for providing QoS guarantees in SOAs. In particular, the QoSDB has been plugged into a real-time QoS architecture [6] featured by the mod_reserve[2], a resource reservation module for the Apache2 web server. The mod_reserve is capable of guaranteeing the QoS required by clients in the provisioning of CPU-intensive services. For the QoS enforcement, it uses the user-space library made available through the AQuoSA framework [20], that enhances the Linux kernel with a real-time scheduling policy based on earliest deadline first (EDF). In particular, mod_reserve exploits the uniprocessor[3] AQuoSA scheduler for allocating CPU "shares" managed through the Resource Reservation (RR) [22] approach. In the RR framework, a resource allocation is specified in terms of a budget Q and a period P, with the meaning that the resource is granted for a minimum of Q time units every time-frame of duration. The ratio $B = Q/P$ represents the share of the resource that has been reserved.

The RR approach provides the fundamental property of temporal isolation [23] in allocating a shared resource to a set of tasks that need to concurrently use it. This means that each task is reserved a fraction of the resource utilization, so that its ability to meet timing constraints is not influenced by the presence of other tasks in the system. Focusing on the CPU, this property ensures that each task can be thought of as running on a virtual CPU, whose speed is a fraction B of the real CPU speed.

The experiments have been performed on a system with a 1.2 Ghz Intel CPU, $3GiB$ RAM, running a $64bit$ GNU/Linux

[2]More information on the mod_reserve, former known as RtModule, is available at the URL: http://freecode.com/projects/mod_reserve.

[3]The AQuoSA framework has been recently extended to use the IRMOS real-time multi-processor scheduler, even though this version was not used in this paper. The interested reader can find additional details in [21].

OS with a 2.6.28 kernel patched with AQuoSA. In the proposed scenario, the service provider makes available an image rotation service, whose QoS parameters can be specified through a contract negotiation scheme, for example by using Service Level Agreements (SLAs). The service consumer must specify the image width w, the image height h and the desired response time D.

In particular, for satisfying the latter QoS requirement, the `mod_reserve` challenge consists in "guessing" the proper pair B and P for scheduling the serving task. In this context, a good choice for the time granularity of the reserve is $P = 100ms$ (as the period is also representative of the maximum activation delay, lower values could be more appropriate in different contexts, like virtualization [24]). In this case the period P is automatically set for each request by the service provider, based on the information provided by the service engineer. However, our QoS architecture also allows for the negotiation of such parameter with the service consumer.

Instead, the bandwidth B is computed request-by-request by leveraging the capabilities provided by the `QoSDB` library. In this way, it is possible to allocate only the resource share needed for guaranteeing the required QoS, permitting to use resources for completing other tasks or eventually use strategies for energy saving (e.g. switching off a core processor or lowering the resource speed).

The operations performed by `mod_reserve` for providing the service with QoS capabilities can be described by Algorithm 1, which follows the notation introduced in Section III-A and reports the `QoSDB` methods without any C artifact. In particular, such operations can be detailed as follows:

1) Retrieving the operating mode id v_m with the `qosdb_lookup` (line 1). In the service considered in this scenario, each mode m is represented by the product $w * h$ of the image to rotate.
2) Getting the average statistic for the corresponding mode with the `qosdb_get` (lines 2-4).
3) Computing the value B as the ratio between the average statistic and the service deadline d (lines 5-9). In case the statistic is null, an arbitrary initial bandwidth value B^0 is assigned. The deadline d is assigned by the system, based on the response time D specified by the service consumer. It could be set equal to D, however a safer practice consists in setting d equal to a bit lower target value $\hat{D}$, for considering the various source of indeterminism e.g., cache and software interrupts, that can still affect the real-time behavior of AQuoSA in implementing the RR mechanism on Linux.
4) Serving the request by assigning a fraction B of CPU to the serving task (lines 10-12). The metafunction *allocate_CPU_share* wraps the AQuoSA calls necessary for performing the resource allocation, whilst the *execute_service* wraps operations performed by the web server. We also assume the availability of a system call for getting the service execution time.
5) Storing in memory the service execution time with the `qosdb_set_sample` and updating the statistics with the `qosdb_merge` (lines 13-15).

Algorithm 1 QoS_PROVISIONING(m, d)

1: $v_m \leftarrow \mathsf{qosdb\_lookup\_app\_mode}(m)$
2: $\mathsf{pm}_{get} \leftarrow get\_current\_CPU\_speed()$
3: $\mathsf{st}_m \leftarrow \mathsf{qosdb\_get}(v_m, \mathsf{pm}_{get})$
4: $avg \leftarrow \mathsf{st}_m[AVG]$
5: **if** $avg = NULL$ **then**
6: $\quad B \leftarrow B^0$
7: **else**
8: $\quad B \leftarrow avg/d$
9: **end if**
10: $allocate\_CPU\_share(B, getpid())$
11: $execute\_service()$
12: $t \leftarrow get\_service\_execution\_time()$
13: $\mathsf{pm}_{set} \leftarrow \mathsf{pm}_{get} * B$
14: $\mathsf{qosdb\_set\_sample}(v_m, t, \mathsf{pm}_{set})$;
15: $\mathsf{qosdb\_merge}(v_m)$;
16: **return**

Please note that Algorithm 1 is general and does not rely on the particular service used in this scenario. Of course, it requires that a mapping function exist for each different service, in order to determine the operating mode m from the service parameters in_j (see Section III-A). We also stress the fact that Algorithm 1 is performed by the `mod_reserve` component, that does not execute itself the services. It "intercepts" request calls performed to the web server and enhances its normal behavior by providing requested QoS guarantees. For the sake of simplicity, operations performed in different phases of the web server loop have been presented in a sequential manner.

A first experiment, called Experiment I, has been conducted for showing the auto-tuning capability the system acquires by leveraging the proposed QoS registry. Service consumers perform 50 subsequent requests with parameters $w = 1000pixel, h = 1000pixel, D = 660ms$. The desired response time D is not considered as the target deadline of the system; instead we consider, as discussed, a lower internal target $\hat{D} = 640ms$. Denoting $\bar{r}_{\mathsf{full}}$ as the average service response time when requests are processed by using the CPU at full speed, an off-line analysis on such service reveals that $\bar{r}_{\mathsf{full}}$ is equal to $50.18ms$ (the 90% confidence interval is about 0.1% of the average value). Thus, an optimal assignment B^* in the sense of minimizing the resource allocation for sustaining the desired deadline would be $B^* = \bar{r}_{\mathsf{full}}/\hat{D} \simeq 0.0784$. This measure is used for benchmarking the performance of the system. Instead, for highlighting the auto-tuning capabilities of our architecture, an arbitrary value $B^0 = 0.09$ has been chosen for the initial CPU bandwidth assignment.

At the beginning, requests are performed by 1 service consumer. For each request the assigned bandwidth B and the response time r have been measured and results are plotted in Figure 2. In Figure 2(a), it can be seen that the first computations of B are clearly overestimated but the system rapidly evolves thanks to the use of `QoSDB`, assigning for

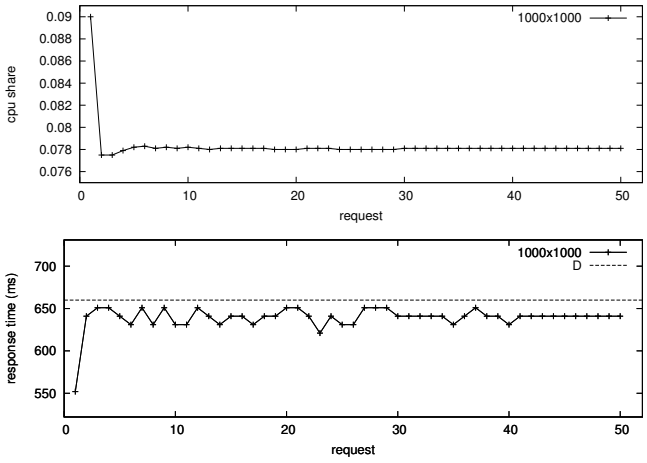

Figure 2. Adaptive Resource Allocation in Service Provisioning

Table III
PERFORMANCES OF ADAPTIVE RESOURCE ALLOCATION BASED ON THE AVERAGE QOS STATISTIC

	MAPE (%)	DMR (%)
1 client	0.71	0
5 clients	1.76	2.0
10 clients	1.85	5.8

most of the requests a bandwidth equal to B^*. Figure 2(b) reports the actual response times and shows how the serving task scheduled with the computed bandwidth reservation is capable of guaranteeing the required QoS (values are always under the dotted line denoting the desired response time D).

Then, the experiment has been repeated when requests are performed by a different number of concurrent service consumers c and the same behavior of Figure 2 has been observed. For doing a comparison of the collected results we introduce two different metrics. By denoting $r_{c,i}$ and $B_{c,i}$ respectively the response time and the bandwidth allocated for the i-th requests performed by client c, such metrics can be defined as follows:

- the Deadline Miss Ratio $\mathsf{DMR} = \frac{\mathsf{dmn}}{N}$, where the number of deadline misses dmn is the cardinality of the set $\{r_{c,i} | r_{c,i} > D\}$ and N is the total number of requests received by the provider;
- the Mean Absolute Percentage Error defined as $\mathsf{MAPE} = \frac{1}{N} \sum_{i,c} \frac{|B^* - B_{c,i}|}{B^*}$

Table III reports the value obtained for the introduced metrics when requests are performed by 1, 5 and 10 concurrent clients. The results reported for the MAPE metric are very low and show that the system performs very well in allocating the right resource share. Instead, values for the DMR metric grows proportionally with the number of concurrent clients, reaching a quite significant value in the case of 10 clients. However, the average service response times, calculated throughout the whole experiment, are quite similar in all the three cases (respectively equal to $639.02ms$, $639.22ms$ and $641.79ms$ with the 90% confidence intervals lower than 0.2%) and do not present the same proportional difference with respect to the DMR values. This can suggest that when strong guarantees must be provided in terms of respecting deadlines for real-time service-oriented applications, QoS management techniques based on the analysis of the maximum values could be more appropriate, rather than referring to the average QoS statistics, as done in this experiment.

Table IV
PERFORMANCES OF ADAPTIVE RESOURCE ALLOCATION BASED ON THE MAX QOS STATISTIC

	MAPE (%)	DMR (%)
1 client	1.17	0
5 clients	2.89	0.8
10 clients	8.22	0.4

Following this reasoning, the Experiment II has been performed with the same setup of Experiment I, except for considering the max statistic instead of the average in line 4 of Algorithm 1. In the results, reported in Table IV, it could be seen that DMR values has been drastically reduced with respect to values of Table III, meaning that a minor number of deadline misses occur (both in the case of 5 and 10 concurrent clients, only 2 deadline are missed). Of course, this is achieved at the cost of overestimating the resource allocation for service execution, as reflected by MAPE values that are increased with respect to those of Table III.

Finally, an experiment is performed in which the server receives 50 subsequent requests by 1 client but the image resolution changes every 10 requests, forcing a change of the operating mode. This experiment has been conceived for highlighting the advantage of using the Prediction Plugin for guessing the behavior of the application when changing from one mode to another. Thus, for each operating mode the error in computing the bandwidth B has been measured and plotted in Figure 3. It can be seen that the allocation error for the mode 2000x2000 is significant, as the database is not populated yet and a $B^0 = 0.15$ allocation value is given. Subsequently, the system evolves towards a plateau, similarly to the first experiment reported in Figure 2(a). At request number 11, when the operating mode changes to 1500x1500, the database is not populated for that operating mode but the `QoSDB` exploits the statistics already collected for performing a guess that is very close to the plateau value. The same behavior can be observed for the other mode changes, that happen at requests number 21, 31 and 41. The prediction algorithm used in this experiment is based on the OLS method and is already built in the `QoSDB`, as described in Section III-B.

In traditional feedback-based scheduling, an application continuously running adapts dynamically the scheduler parameters based on recently observed resource requirements. Instead, in the presented experiment, the individual requests are served by independent activations of the `cgi-bin` service, which can occur at great or small distances in time. The service is thus instantiated each time along with the creation

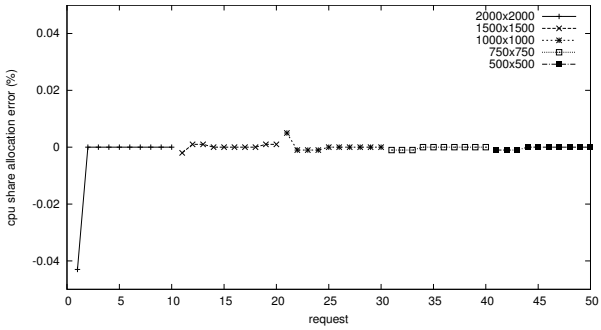

Figure 3. QoS Prediction on Mode Changes

of the associated resource reservation into the scheduler (as handled by `mod_reserve`). Thus, in the proposed work, the "feedback control loop" is closed in an off-line fashion, by recurring to the registry.

VI. Conclusion

In this paper a QoS registry (`QoSDB`) has been presented for adaptive real-time service-oriented applications. It has been conceived for supporting QoS management by permitting to predict, store and save QoS data and statistics related to different functional behaviors of an application. It is characterized by a modular architecture that permits the insertion of new algorithms for an easy customization. The proposed registry has also been developed and integrated into a real-time SOA and some experiments conducted in this context have shown the efficiency of the proposed solution, along with its effectiveness in supporting adaptive techniques for providing services with QoS guarantees.

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme under grant agreements n.214777 and n.248465, in the context of the IRMOS and S(o)OS Projects.

References

[1] F. Jammes and H. Smit, "Service-oriented paradigms in industrial automation," *Industrial Informatics, IEEE Transactions on*, vol. 1, no. 1, pp. 62–70, Feb. 2005.

[2] R. Zender, U. Lucke, and D. Tavangarian, "SOA Interoperability for Large-Scale Pervasive Environments," *Proceeding of the 5th International Workshop on Service Oriented Architectures in Converging Networked Environments*, vol. 0, pp. 545–550, 2010.

[3] T. Cucinotta, F. Checconi, Z. Zlatev, J. Papay, M. J. Boniface, G. Kousiouris, D. Kyriazis, T. A. Varvarigou, S. Berger, D. Lamp, A. Mazzetti, T. Voith, and M. Stein, "Virtualised e-learning with real-time guarantees on the irmos platform," in *IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2010*, December 2010, pp. 1–8.

[4] T. Cucinotta, L. Palopoli, L. Abeni, D. Faggioli, and G. Lipari, "On the integration of application level and resource level qos control for real-time applications," *Industrial Informatics, IEEE Transactions on*, vol. 6, no. 4, pp. 479–491, Nov. 2010.

[5] M. García-Valls, I. Estévez-Ayres, and P. Basanta-Val, "Dynamic priority assignment scheme for contract-based resource management," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, 7 2010, pp. 1987 –1994.

[6] T. Cucinotta, A. Mancina, G. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo, and F. Rusina, "A real-time service-oriented architecture for industrial automation," *Industrial Informatics, IEEE Transactions on*, vol. 5, no. 3, pp. 267–277, Aug. 2009.

[7] A. Sheth, J. Cardoso, J. Miller, and K. Kochut, "Qos for service-oriented middleware," in *The 6th World Multiconference on Systemics, Cybernetics and Informatics, Proceedings Vol. 8*, Orlando, FL, 7 2002, pp. 528 – 534.

[8] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *Software Engineering, IEEE Transactions on*, vol. 30, no. 5, pp. 311 – 327, may 2004.

[9] I.-L. Yen, H. Ma, F. Bastani, and H. Mei, "Qos-reconfigurable web services and compositions for high-assurance systems," *Computer*, vol. 41, no. 8, pp. 48 –55, aug. 2008.

[10] T. Yu and K.-J. Lin, "Qcws: an implementation of qos-capable multimedia web services," *Multimedia Tools Appl.*, vol. 30, pp. 165–187, August 2006.

[11] J. Li, Y. Zhao, J. Ren, and D. Ma, "Towards adaptive web services qos prediction," in *Service-Oriented Computing and Applications (SOCA), 2010 IEEE International Conference on*, dec. 2010, pp. 1 –8.

[12] Y. Lee, "Quality-context based soa registry classification for quality of services," in *Advanced Communication Technology, 2009. ICACT 2009. 11th International Conference on*, vol. 03, feb. 2009, pp. 2251 –2255.

[13] A. Alonso, E. Salazar, and J. López, "Resource management for enhancing predictability in systems with limited processing capabilities," in *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, 9 2010, pp. 1–7.

[14] M. García-Valls, A. Alonso, J. Ruiz, and A. Groba, "An architecture of a quality of service resource manager middleware for flexible embedded multimedia systems," in *Software Engineering and Middleware*, ser. Lecture Notes in Computer Science, A. Coen-Porisini and A. van der Hoek, Eds. Springer Berlin / Heidelberg, 2003, vol. 2596, pp. 36–55.

[15] E. Hide, T. Stack, J. Regehr, and J. Lepreau, "Dynamic cpu management for real-time, middleware-based systems," in *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, May 2004, pp. 286 – 295.

[16] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli, "Qos management through adaptive reservations," *Real-Time Systems*, vol. 29, pp. 131–155, 2005, 10.1007/s11241-005-6882-0. [Online]. Available: http://dx.doi.org/10.1007/s11241-005-6882-0

[17] A. Donadoni, "Progetto e realizzazione di un'architettura software modulare ed estendibile per il monitoraggio e la gestione dei parametri di esecuzione di applicazioni soft real-time," 10 2008. [Online]. Available: http://etd.adm.unipi.it/theses/available/etd-09012008-160756/

[18] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "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*, 1998, pp. 150–164.

[19] K. Lakshmanan and R. Rajkumar, "Distributed resource kernels: Os support for end-to-end resource isolation," in *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, april 2008, pp. 195 –204.

[20] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "Aquosa—adaptive quality of service architecture," *Software: Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2009. [Online]. Available: http://dx.doi.org/10.1002/spe.883

[21] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari, "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, June 2009.

[22] C. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: operating system support for multimedia applications," in *Multimedia Computing and Systems, 1994., Proceedings of the International Conference on*, May 1994, pp. 90 –99.

[23] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo, *Soft Real-Time Systems: Predictability vs. Efficiency (Series in Computer Science)*. Plenum Publishing Co., 2005.

[24] T. Cucinotta, G. Anastasi, and L. Abeni, "Respecting temporal constraints in virtualised services," in *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, vol. 2, July 2009, pp. 73–78.