# Autonomic QoS control in enterprise Grid environments using online simulation

Ramon Nou [a,*] Samuel Kounev [b] Ferran Julià [a] Jordi Torres [a]

[a] *Barcelona Supercomputing Center (BSC), Technical University of Catalonia (UPC), Barcelona, Spain*

[b] *Computer Laboratory, University of Cambridge, UK*

**Abstract**

As Grid Computing increasingly enters the commercial domain, performance and Quality of Service (QoS) issues are becoming a major concern. The inherent complexity, heterogeneity and dynamics of Grid computing environments pose some challenges in managing their capacity to ensure that QoS requirements are continuously met. In this paper, a comprehensive framework for autonomic QoS control in enterprise Grid environments using online simulation is proposed. The paper presents a novel methodology for designing autonomic QoS-aware resource managers that have the capability to predict the performance of the Grid components they manage and allocate resources in such a way that service level agreements are honored. Support for advanced features such as autonomic workload characterization on-the-fly, dynamic deployment of Grid servers on demand, as well as dynamic system reconfiguration after a server failure is provided. The goal is to make the Grid middleware self-configurable and adaptable to changes in the system environment and workload. The approach is subjected to an extensive experimental evaluation in the context of a real-world Grid environment and its effectiveness, practicality and performance are demonstrated.

## 1 Introduction

Grid Computing emerged in the second half of the 1990s as a new computing paradigm for advanced science and engineering. It not only managed to establish itself as the major computing paradigm for high-end scientific applications, but it is

* *Email addresses:* rnou@ac.upc.edu (Ramon Nou), skounev@acm.org (Samuel Kounev), ferran.julia@bsc.es (Ferran Julià), torres@ac.upc.edu (Jordi Torres).

now becoming a mainstream paradigm for enterprise applications and distributed system integration [1, 2]. By enabling flexible, secure and coordinated sharing of resources and services among dynamic collections of disparate organizations and parties, Grid computing promises a number of advantages to businesses, for example faster response to changing business needs, better utilization and service level performance, and lower IT operating costs [2]. However, as Grid computing enters the commercial domain, performance and QoS (Quality of Service) aspects, such as customer observed response times and throughput, are becoming a major concern. The inherent complexity, heterogeneity and dynamics of Grid computing environments pose some challenges in managing their capacity to ensure that QoS requirements are continuously met.

Large scale grids are typically composed of large number of heterogeneous components deployed in disjoint administrative domains, in highly distributed and dynamic environments. Managing QoS in such environments is a challenge because Service Level Agreements (SLA) must be established and enforced both globally and locally at the components involved in the execution of tasks [3]. Grid components are assumed to be autonomous and they may join and leave the Grid dynamically. At the same time, enterprise and e-business workloads are often characterized by rapid growth and unpredictable variations in load. These aspects of enterprise Grid environments make it hard to manage their capacity and ensure that enough resources are available to provide adequate QoS to both customers and enterprise users. The resource allocation and job scheduling mechanisms used at the global and local level play a critical role for the performance and availability of Grid applications. To guarantee that QoS requirements are satisfied, the Grid middleware must be capable of predicting the application performance when deciding how to distribute the workload among the available resources. Prediction capabilities make it possible to implement intelligent QoS-aware resource allocation and admission control mechanisms.

Performance prediction in the context of traditional enterprise systems is typically done by means of performance models that capture the major aspects of system behavior under load [4]. Numerous performance prediction and capacity planning techniques for conventional distributed systems, most of them based on analytic or simulation models, have been developed and used in the industry. However, these techniques generally assume that the system is static and that dedicated resources are used. Furthermore, the system is normally exposed to a fixed set of quantifiable workloads. Therefore, such performance prediction techniques are not adequate for Grid environments which use non-dedicated resources and are subject to dynamic changes in both the system configuration and workload. To address the need for performance prediction in Grid environments, new techniques are needed that use performance models generated on-the-fly to reflect changes in the environment. The term *online performance models* was recently coined for this type of models [5]. The *online* use of performance models defers from their traditional use in capacity planning in that configurations and workloads are analyzed that reflect the real

system over relatively short periods of time. Since performance analysis is carried out on-the-fly, it is essential that the process of generating and analyzing the models is completely automated.

We have developed a method for building online performance models of Grid middleware with fine-grained load-balancing [6]. The Grid middleware is augmented with an online performance prediction component that can be called at any time during operation to predict the Grid performance for a given resource allocation and load-balancing strategy. Our performance prediction mechanism is based on hierarchical queueing Petri net models [7] that are dynamically composed to reflect the system configuration and workload. In [8], we showed how this approach can be used for autonomic QoS-aware resource management. We presented a novel methodology for designing autonomic QoS-aware resource managers that have the capability to predict the performance of the Grid components they manage and allocate resources in such a way that SLAs are honored. The goal is to make the Grid middleware self-configurable and adaptable to changes in the system environment and workload. QoS-aware resource reservation and admission control mechanisms are employed to ensure that resources are only allocated if enough capacity is available to provide adequate performance. Resource managers engage in QoS negotiations with clients making sure that they can provide the requested QoS before making a commitment.

Our approach is the first one to combine QoS control with fine-grained load-balancing making it possible to distribute the workload among the available Grid resources in a dynamic way that improves resource utilization and efficiency. The latter is crucially important for enterprise and commercial Grid environments. Another novel aspect of our approach is that it is the first one that uses queueing Petri nets as online performance models for autonomic QoS control. The use of queueing Petri nets is essential since it enables us to accurately model the behavior of our resource allocation and load balancing mechanism which combines hardware and software aspects of system behavior. Moreover, queueing Petri nets have been shown to lend themselves very well to modeling distributed component-based systems [9] which are commonly used as building blocks of Grid infrastructures [10]. Finally, although our methodology is targeted at Grid computing environments, it is not in any way limited to such environments and can be readily used to build more general QoS-aware Service-Oriented Architectures (SOA).

Building on our previous work, in this paper we propose a comprehensive framework for autonomic QoS control in enterprise Grid environments using online simulation. We first present in detail our Grid QoS-aware resource manager architecture focusing on the performance prediction mechanism and the resource allocation algorithm. We then introduce a method for autonomic workload characterization on-the-fly based on monitoring data. This makes it possible to apply our approach to services for which no workload model is available. Following this, we further extend our resource allocation algorithm to support adding Grid servers on demand as

well as dynamically reconfiguring the system after a server failure. We present an extensive performance evaluation of our framework in two different experimental environments using the Globus Toolkit [11]. We consider five different scenarios each focusing on selected aspects of our framework. The results demonstrate the effectiveness and practicality of our approach. The paper complements and extends our previous work [6, 8] along the following dimensions:

- A method for characterizing the workload of a server on-the-fly in an autonomic fashion based on monitoring data is proposed.
- The resource manager architecture is extended to support adding servers on demand as well as reconfiguring the system dynamically after a server failure.
- The framework is evaluated in a larger more realistic environment with up to 9 Grid servers (our previous work only considered 2 servers) and with more complex and representative workloads. The evaluation includes experiments in dynamic environments not considered previously.
- The servers are deployed in a virtualized environment which is typical for modern state-of-the-art Grid and SOA systems.
- An extensive performance evaluation of the individual features of our extended architecture is presented including the method for autonomic workload characterization, the method for adding servers on demand and the method for dynamic reconfiguration after a server failure.
- The online performance prediction mechanism is extended to support a general purpose simulation system OMNeT++ [12], in addition to QPN models. With OMNeT++ an arbitrary complex system can be modeled since it allows component behavior to be described using a general purpose programming language.

The rest of the paper is organized as follows. In Section 2, we introduce our QoS-aware resource manager architecture with the extensions mentioned above. In Section 3, we describe the experimental environment in which we evaluate the performance of our framework. Section 4 presents the detailed results of our performance evaluation. Section 5 summarizes related work in the area of QoS management in Grid computing. Finally, in Section 6, we present some concluding remarks and discuss future work.

## 2 Grid QoS-Aware Resource Management

The allocation and scheduling mechanism used at the global and local level play a critical role for the performance and availability of Grid applications. To prevent resource congestion and unavailability, it is essential that admission control mechanisms are employed by local resource managers. Furthermore, to achieve maximum performance, the Grid middleware must be smart enough to schedule tasks in such a way that the workload is load-balanced among the available resources and they are all roughly equally utilized. However, this is not enough to guaran-

tee that the performance and QoS requirements are satisfied. To prevent inadequate QoS, resource managers must be able to predict the system performance for a given resource allocation and workload distribution. In this section, we present a methodology for designing QoS-aware resource managers as part of the Grid middleware. Our prototype is built for a scenario where no security issues arise due to Grid server providers being unwilling to share workload information about the services they offer. Building the same system with limitations on the workload information that can be shared between providers would require some extensions to our framework.

### *2.1 Resource Manager Architecture*

The resource manager architecture we propose is composed of four main components: QoS Broker, QoS Predictor, Client Registry and Service Registry. Figure 1 shows a high-level view of the architecture. A resource manager is responsible for managing access to a set of Grid servers each one offering some Grid services. Grid servers can be heterogeneous in terms of the hardware and software platforms they are based on and the services they offer. Services can be offered by different number of servers depending on the user demand. The resource manager keeps track of the Grid servers currently available and mediates between clients and servers to make sure that SLAs are continuously met. For a client to be able to use a service, it must first send a *session request* to the resource manager. The session request specifies the type of service requested, the frequency with which the client will send requests for the service [1] (*service request* arrival rate) and the required average response time (SLA). The resource manager tries to find a distribution of the incoming requests among the available servers that would provide the requested QoS. If this is not possible, the session request is rejected or a counter offer with lower throughput or higher average response time is made. Figure 2 shows a more detailed view of the resource manager architecture and its internal components. We now take an inside look at each component in turn.

The *service registry* keeps track of the Grid servers the resource manager is responsible for and the services they offer. Before a Grid server can be used, it must register with a resource manager providing information on the services it offers, their resource requirements and the server capacity made available to the Grid. For maximum interoperability, it is expected that standard Web Services mechanisms, such as WSDL [13], are used to describe services. In addition to sending a description of the services, the Grid server must provide a workload model that captures the service behavior and resource requirements. Depending on the type of services, the workload model could vary in complexity ranging from a simple specification

[1] Note that we distinguish between *session requests* and the individual *service requests* sent as part of a session.

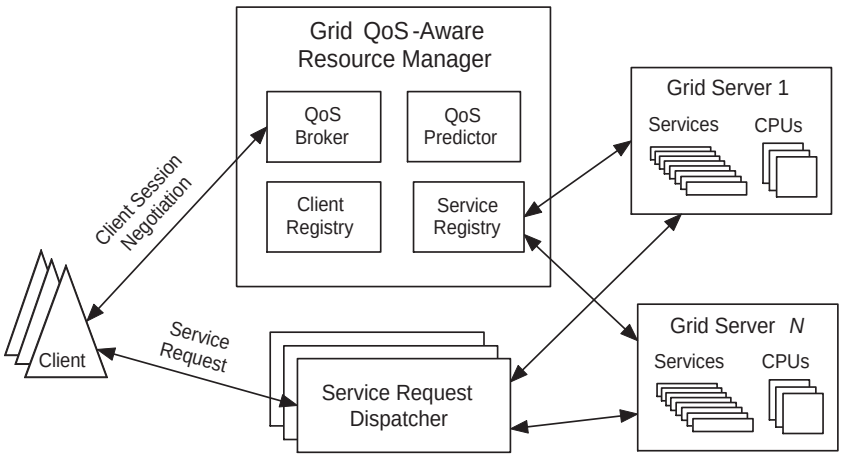


Figure 1. Resource manager architecture.

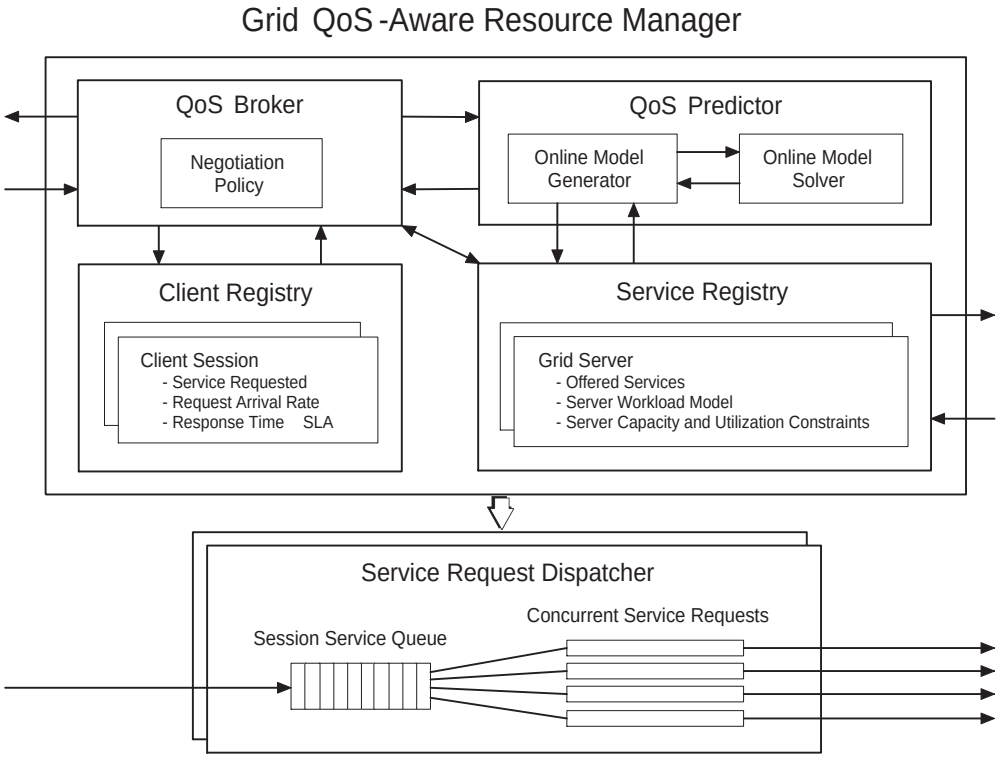


Figure 2. Inside view of the resource manager architecture.

of the service demands at the server resources to a detailed mathematical model capturing the flow of control during service execution. The Job Submission Description Language (JSDL) [14] could be used to describe service resource and data requirements in a standarized way. Finally, the server administrator might want to impose a limit on its usage by the Grid middleware in order to avoid overloading it. Some Grid servers have been shown to exhibit unstable behavior when their utilization approaches 99% [15]. For each server, the service registry keeps track of its maximum allowed utilization by the Grid.

The *QoS broker* receives session requests from clients, allocates server resources and negotiates SLAs. When a client sends a session request, the QoS broker tries to find an optimal distribution of the workload among the available Grid servers that satisfies the QoS requirements. It is assumed that for each client session, a given number of threads (from 0 to unlimited) is allocated on each Grid server offering the respective service. Incoming service requests are then load-balanced across the servers according to thread availability. Threads serve to limit the concurrent requests executed on each server, so that different load-balancing schemes can be enforced. The QoS broker tries to distribute the workload uniformly among the available servers to achieve maximum efficiency. In doing so it considers different configurations in terms of thread allocation and for each of them uses the *QoS predictor* to predict the performance of the system. The goal is to find a configuration that satisfies both the SLAs of active sessions and the constraints on the maximum utilization of the Grid servers. If no configuration can be found, the QoS broker must either reject the session request or send a counter offer to the client. While this is beyond the scope of the current paper, the QoS negotiation protocol we use here can easily be extended to support standard service negotiation protocols such as the Web Services Agreement Specification (WS-Agreement) [16]. At each point in time, the *client registry* keeps track of the currently active client sessions. For each session, information on the service requested, the request arrival rate and the required average response time (SLA) is stored.

If a session request is accepted, the resource manager sets up a *service request dispatcher* for the new session, which is a standalone software component responsible for scheduling arriving service requests (as part of the session) for processing at the Grid servers. It is ensured that the number of concurrent requests scheduled on a Grid server does not exceed the number of threads allocated to the session for the respective server. The service request dispatcher queues incoming requests and forwards them to Grid servers as threads become available. Note that threads are used here as a *logical* entity to enforce the desired concurrency level on each server. Thread management is done entirely by the service request dispatcher and there is no need for Grid servers to know anything about the client sessions and how many threads are allocated to each of them. In fact the only requirement is that the request dispatcher sends no more concurrent requests to a Grid server than the maximum allowed by the active configuration. While the request dispatcher might use a separate physical thread for each logical thread allocated to a session, this is not required by the architecture and there are many ways to avoid doing this in the interest of performance. Service request dispatchers are not required to run on the same machine as the resource manager and they can be distributed across multiple machines if needed. To summarize, service request dispatchers serve as light-weight session load-balancers enforcing the selected workload distribution.

Service request dispatchers play an essential role in our framework since they completely decouple the Grid clients from the Grid servers. This decoupling provides some important advantages that are especially relevant to modern Grid applications.

First of all, the decoupling enables us to introduce fine-grained load-balancing at the service request level, as opposed to the session level. Second, service request dispatchers make it possible to load-balance requests across heterogeneous server resources without relying on any platform-specific scheduling or load-balancing mechanisms. Finally, since clients do not interact with the servers directly, it is possible to adjust the resource allocation and load-balancing strategies dynamically. Thus, our framework is geared towards making the Grid middleware self-configurable and adaptable to changes in the system environment and workload. Taken together the above mentioned benefits provide extreme flexibility in managing Grid resources which is essential for enterprise and commercial Grid environments.

The resource manager architecture we propose is mainly targeted at scenarios where the resources managed by a single resource manager belong to the same business entity. In scenarios where a resource manager is used to manage resources spanning multiple business domains, it is expected that providers of Grid resources are willing to share information about the behavior and resource requirements of the services they offer. Note that only high-level information on the service resource consumption is expected to be shared, i.e., no information about the internal service implementation is required. Although it is outside of the scope of the current paper, our approach can be easily extended to support managing Grid resources using a hierarchy of resource managers. This would eliminate any confidentiality issues in a scenario with multiple business domains, since each domain would use their own local resource managers. As part of our future work, we plan to develop algorithms for load balancing across multiple resource managers.

### *2.2 QoS Predictor*

The QoS Predictor is a critical component of the resource manager architecture since it is the basis for ensuring that the QoS requirements are continuously met. The QoS Predictor is made of two subcomponents - model generator and model solver. The model generator automatically constructs a performance model based on the active client sessions and the server workload models retrieved from the service registry. The model solver is used to analyze the model either analytically or through simulation. Different types of performance models can be used to implement the QoS Predictor. We propose the use of Queueing Petri Nets (QPNs) which provide greater modeling power and expressiveness than conventional modeling formalisms like queueing networks, extended queueing networks and generalized stochastic Petri nets [17]. In [9], it was shown that QPN models lend themselves very well to modeling distributed component-based systems and provide a number of important benefits such as improved modeling accuracy and representativeness. The expressiveness that QPNs models offer makes it possible to model the logical threads used in our load-balancing mechanism accurately. Depending on the size of

QPN models, different methods can be used for their analysis, from product-form analytical solution methods [18] to highly optimized simulation techniques [19].

Figure 3 shows a high-level QPN model of a set of Grid servers under the control of a QoS-aware resource manager. The Grid servers are modeled with nested QPNs represented as subnet places. The *Client* place contains a $G/G/\infty/IS$ queue which models the arrival of service requests sent by clients. Service requests are modeled using tokens of different colors, each color representing a client session. For each active session, there is always one token in the Client place. When the token leaves the Client queue, transition $t_1$ fires moving the token to place *Service Queue* (representing the arrival of a service request) and depositing a new copy of it in the Client queue. This new token represents the next service request which is delayed in the Client queue for the request interarrival time. An arbitrary request interarrival time distribution can be used. For each Grid server the resource manager has a *Server Thread Pool* place containing tokens representing the logical threads on this server allocated to the different sessions (using colors to distinguish between them). An arriving service request is queued at place Service Queue and waits until a thread for its session becomes available. When this happens, the request is sent to the subnet place representing the respective Grid server. After the request is processed, the logical service thread is returned back to the thread pool from where it was taken. By encapsulating the internal details of Grid servers in separate nested QPNs, we decouple them from the high-level performance model. Different servers can be modeled at different level of detail depending on the complexity of the services they offer. It is assumed that when registering with the resource manager, each Grid server provides all the information needed for constructing its nested QPN model. This information is basically what constitutes the server workload model discussed earlier. In the most general case, Grid servers could send their complete QPN models to be integrated into the high-level model.

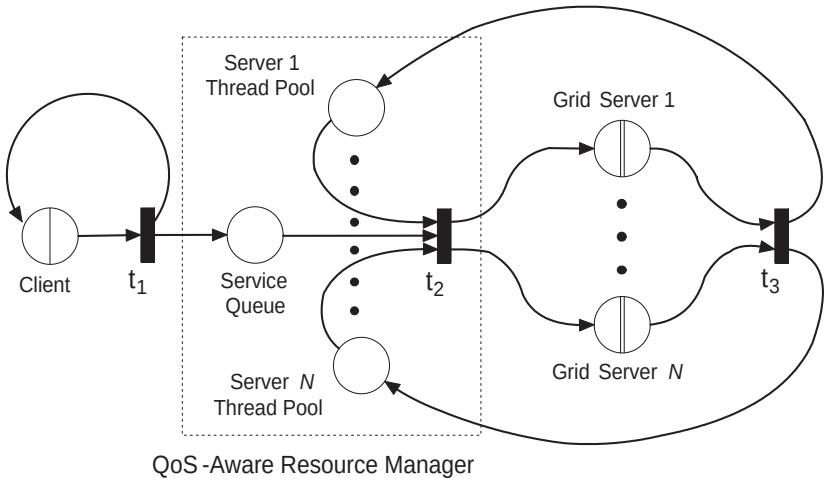


Figure 3. Generic performance model.

### 2.3 Resource Allocation Algorithm

The presented resource manager architecture is reliant on an efficient algorithm for allocating resources in such a way that both the QoS requirements and the Grid server utilization constraints are honored. In this section, we present such an algorithm that can be used to implement the QoS Broker component described in Section 2.1. Formally, the Grid environment under the control of a QoS-aware resource manager can be represented as a 4-tuple $G = (S, V, F, C)$ where:

**S** $= \{s_1, s_2, ..., s_m\}$ is the set of Grid servers,
**V** $= \{v_1, v_2, ..., v_n\}$ is the overall set of services offered by the Grid servers,
**F** $\in [S \longrightarrow 2^V]$ [2] is a function assigning a set of services to each Grid server. Since Grids are typically heterogeneous in nature, we assume that, depending on the platform they are running on, Grid servers might offer different subsets of the overall set of services,
**C** $= \{c_1, c_2, ..., c_l\}$ is the set of currently active client sessions. Each session $c \in C$ is a triple $(v, \lambda, \rho)$ where $v \in V$ is the service used, $\lambda$ is the rate at which requests for the service arrive and $\rho$ is the client requested average response time (SLA).

We denote the number of processors (CPUs) of server $s \in S$ as $P(s)$ and the server's maximum allowed average utilization as $\overline{U}(s)$. It is assumed that for each client session, a given number of threads (from 0 to unlimited) is allocated on every Grid server offering the respective service. Recall that threads are used as a *logical* entity to limit the concurrency level on each server and they should not be confused with physical threads allocated on the machines. The goal of the resource allocation algorithm is to find a configuration, in terms of allocated threads, that satisfies both the SLAs of active sessions and the constraints on the maximum utilization of the Grid servers. A configuration is represented by a function $T \in [C \times S \longrightarrow \mathbb{N}_0 \cup \{\infty\}]$ which will be referred to as *thread allocation function*. Hereafter, a superscript $T$ will be used to denote functions or quantities that depend on the thread allocation function, e.g. $X^T(c)$.

As discussed in Section 2.1, the QoS Broker examines a set of possible configurations using the QoS Predictor to determine if they meet the requirements. For each considered configuration, the QoS Predictor takes as input the thread allocation function $T$ and provides the following predicted metrics as output:

$X^T(c)$ for $c \in C$ is the total number of completed service requests from client session $c$ per unit of time (the overall throughput),
$U^T(s)$ for $s \in S$ is the average utilization of server $s$,
$R^T(c)$ for $c \in C$ is the average response time of an arriving service request from client session $c$.

[2] $2^V$ denotes the set of all possible subsets of $V$, i.e. the power set.

We define the following predicates:

$P_X^T(c)$ for $c \in C$ is defined as $(X^T(c) = c[\lambda])$
$P_R^T(c)$ for $c \in C$ is defined as $(R^T(c) <= c[\rho])$
$P_U^T(s)$ for $s \in S$ is defined as $(U^T(s) <= \overline{U}(s))$

For a configuration represented by a thread allocation function $T$ to be acceptable, the following condition must hold $(\forall c \in C : P_X^T(c) \wedge P_R^T(c)) \wedge (\forall s \in S : P_U^T(s))$. We define the following functions:

$A^T(s) \stackrel{def}{=} (\overline{U}(s) - U^T(s))P(s)$

is the amount of unused server CPU time for a given configuration taking into account the maximum allowed server utilization,

$I^T(v, \epsilon) \stackrel{def}{=} \{s \in S : (v \in F(s)) \wedge (A^T(s) \geq \epsilon)\}$ is the set of servers offering service $v$ that have at least $\epsilon$ amount of unused CPU time. We now present a simple heuristic resource allocation algorithm in mathematical style pseudocode. It is outside the scope of this paper to present complete analysis of possible heuristics and their efficiency. Let $\tilde{c} = (v, \lambda, \rho)$ be a newly arrived client session request. The algorithm proceeds as follows:

---

```
1  C := C ∪ {c̃}
2  for each s ∈ S do
3     if s ∈ I^T(v, ε) then T(c̃, s) := ∞
4     else T(c̃, s) := 0
5  if (∃ c ∈ C : ¬P_X^T(c)) then reject c̃
6  while (∃ ŝ ∈ S : ¬P_U^T(ŝ)) do
7  begin
8     T(c̃, ŝ) := 1
9     while P_U^T(ŝ) do T(c̃, ŝ) := T(c̃, ŝ) + 1
10    T(c̃, ŝ) := T(c̃, ŝ) − 1
11 end
12 if (∃ c ∈ C \ {c̃} : ¬P_X^T(c) ∨ ¬P_R^T(c)) then reject c̃
13 else if (¬P_X^T(c̃) ∨ ¬P_R^T(c̃)) then
14    send counter offer o = (v, X^T(c̃), R^T(c̃))
15 else accept c̃
```

---

The algorithm first adds the new session to the list of active sessions and assigns it an unlimited number of threads on every server that has a given minimum amount of CPU time available. If this leads to the system not being able to sustain the required throughput of an active session, the request is rejected. Otherwise, it is

checked if there are servers whose maximum utilization requirement is broken. For every such server, the number of threads for the new session is set to the highest number that does not result in breaking the maximum utilization requirement. It is then checked if the response time or throughput requirement of one of the original sessions is violated and if that is the case the new session request is rejected. Otherwise, if the throughput or response time requirement of the new session is broken, a counter offer with the predicted throughput and response time is sent to the client. If none of the above holds, i.e., all requirements are satisfied, the session request is accepted. In general, the larger the resource demands of a request, the higher the likelihood of rejection would be if the system is loaded. A larger workload request would require more resources and therefore the likelihood of interfering with the already running sessions (breaking their SLAs) is higher leading to higher probability of rejection.

Given that for each overutilized server, threads are allocated one by one until the maximum allowed utilization is reached, in the worst case, the number of candidate configurations considered would be upper bounded by the number of server CPUs available. The algorithm presented above could be improved in a number of different ways, for example, by aggregating sessions that execute the same service, allocating server resources bottom up instead of top down, parallelizing the simulation to utilize multi-core processors, caching and reusing analyzed configurations, simulating configurations proactively and so forth. The overall overhead of the QoS Broker and possible optimizations are discussed in more detail in Section 4.6. As mentioned above, it is outside the scope of this paper to present complete analysis of the possible heuristics and we intend to consider this as part of our future work.

### *2.4 Workload Characterization On-The-Fly*

So far we have assumed that when a Grid server is registered with the resource manager, a workload model is provided that captures the service behavior and its resource requirements. The workload model is used to generate a performance model as described in Section 2.2. Depending on the type of services, the workload model could vary in complexity ranging from a simple specification of the service demands at the server resources to a detailed mathematical model capturing the flow of control during service execution. In this section, we present a method for generating workload models on-the-fly in an autonomic fashion based on monitoring data. We assume that services use the Grid to execute some business logic requiring a given amount of CPU time. The method we propose is applicable for services with no internal parallelism. The service business logic, however, might include calls to external (third-party) service providers which are not part of the Grid environment. Thus, in order to generate a service workload model, we need to estimate the *CPU service demands* of the services (the total time they spend using the server CPUs) and the total time during service execution spent waiting for external service

providers.

We have developed an algorithm for estimating the CPU service demands iteratively during system operation. We assume that services are added and removed dynamically and no information on their behavior and resource consumption is available. The service demands are estimated for each client session based on monitoring data. In the beginning of the session, the service demand at each server is set to the session response time SLA ($\tilde{c}[\rho]$ in the notation of the previous section). The latter serves as a conservative upper bound on the real service demand. Whenever a request is processed at a Grid server, the server response time is compared with the current estimate of the service demand and if it is lower the estimate is updated. Thus, as the session progresses, the service demand is iteratively set to the lowest observed response time on the respective server. It is expected that during periods of lower workload intensity, the observed response time will be close to the service demand. This approach would work provided that the time spent waiting for external service providers is insignificant compared to the time spent executing the service business logic. To accommodate the more general case, we further monitor the server CPU utilization during service execution and use it to break down the measured response time into time spent using the CPU and time spent waiting for external calls. This allows us to estimate the external provider time. More formally, the algorithm proceeds as follows:

---

```
1 Upon arrival of new session request ĉ do
2  for each s ∈ S : (ĉ[v] ∈ F(s)) do
3  begin
4       Γ[ĉ, s]  :=  ĉ[ρ]
5       Ψ[ĉ, s]  :=  0
6       Θ[ĉ, s]  :=  ĉ[ρ]
7  end
8
9 Upon completion of request x of session ĉ at server ŝ do
10 if (M_R(x) < Θ[ĉ, ŝ]) then
11 begin
12      Θ[ĉ, ŝ]  :=  M_R(x)
13      Γ[ĉ, ŝ]  :=  M_U(ŝ)Θ[ĉ, ŝ]
14      Ψ[ĉ, ŝ]  :=  Θ[ĉ, ŝ]  −  Γ[ĉ, ŝ]
15 end
```

---

where

$\Gamma[c, s]$ for $c \in C$ and $s \in S$ denotes the CPU service demand (total CPU service time) of requests of session $c$ at server $s$.

$\Psi[c, s]$ for $c \in C$ and $s \in S$ denotes the total time spent waiting for external service providers when serving a request of session $c$ at server $s$.

$\Theta[c, s]$ for $c \in C$ and $s \in S$ denotes the minimum observed response time of a request of session $c$ at server $s$.

$M_U(s)$ for $s \in S$ denotes the measured CPU utilization of server $s$ during a service execution.

$M_R(x)$ where $x$ is a service request denotes the measured response time of the request excluding any queueing time at the service request dispatcher.

We shall distinguish between the *basic* version of the algorithm and the *enhanced* version which includes the additional step to break down the measured response time into time spent using the CPU and time spent waiting for external calls. Our algorithm is conservative in that it starts with conservative estimates of the CPU service demands and refines them iteratively in the course of the session. As a result of this, in the beginning of the session, its resource requirements would be overestimated possibly leading to rejecting some client requests even though they could have been accepted without breaking SLAs. The algorithm as presented above can be used for sessions that run services for which no previous information is available in the service registry. Once the service demands of a new service has been estimated, the estimates can be registered in the service registry and used as a starting point in future sessions. The estimates can be further refined as new sessions are executed.

### *2.5 Dynamic Reconfiguration*

The resource allocation algorithm we presented in Section 2.3 assumes that at each point in time there is a fixed set of servers available and tries to distribute the workload among them in such a way that client SLAs are honored. With the increasing proliferation of virtualization solutions and server consolidation, launching servers on demand is becoming more typical. In line with this trend, we have extended our resource allocation algorithm to support adding Grid servers on demand as well as dynamically reconfiguring the system after a server failure. Whenever the QoS requested by a client cannot be provided using the currently available server resources, the extended algorithm considers to launch an additional server to accommodate the new session. At the same time, each time a server failure is detected, the resource manager reconfigures all sessions that had threads allocated on the failed server. The extended algorithm proceeds by considering the affected sessions as new client requests. Existing sessions might have to be cancelled in case there are not enough resources available to provide adequate QoS. Economic factors such as revenue obtained from customer sessions should be considered when deciding which sessions to cancel.

We now present the version of our algorithm that supports adding servers on de-

mand. We denote with $B$ the set of off-line servers available that can be started on demand. Let $\tilde{c} = (v, \lambda, \rho)$ be a newly arrived client session request. The algorithm proceeds as follows:

---

```
1  C := C ∪ {c̃}
2  Ī := I^T
3  B̄ := B
4  for each ŝ ∈ S do T(c̃, ŝ) := 0
5  repeat
6   begin
7     for each ŝ ∈ Ī(v, ε) : T(c̃, ŝ) = 0 do
8     begin
9        T(c̃, ŝ) := ∞
10       if (∃ c ∈ C \ {c̃} : ¬P_X^T(c) ∨ ¬P_R^T(c)) then
11       begin
12          T(c̃, ŝ) := 0
13          Ī(v, ε) := Ī(v, ε) \ {ŝ}
14       end
15       else break
16    end
17    if ((∀ s ∈ S : T(c̃, s) = 0) ∨ ¬P_X^T(c̃) ∨ ¬P_R^T(c̃)) then
18    begin
19      if (∃ n ∈ B̄) then
20      begin
21         B̄ := B̄ \ {n}
22         Ī(v, ε) = Ī(v, ε) ∪ {n}
23      end
24      else break
25    end
26  end
27 until ((P_X^T(c̃) ∧ P_R^T(c̃)) ∨ ¬(∃ ŝ ∈ Ī(v, ε) : T(c̃, ŝ) = 0))
28 if (∀ s ∈ S : T(c̃, s) = 0) then reject c̃
29 else if (¬P_X^T(c̃) ∨ ¬P_R^T(c̃)) then
30    send counter offer o = (v, X^T(c̃), R^T(c̃))
31 else accept c̃
```

---

The algorithm tries to minimize the number of servers on which threads are assigned for the new session. In case the requested QoS cannot be provided with the current set of servers, a new server is started.

Finally, we present our algorithm for dynamic reconfiguration after a server failure. We will use $Alg(\hat{c}, I^T, S)$ to denote a call to our original resource allocation algorithm from Section 2.3 with the respective parameters. Let $E(s)$ be the number of failures detected on server $s$ (initially 0).

```
1  Ī := I^T
2  while (∃ ŝ ∈ S: E(ŝ) > 0) do
3  begin
4     D := ∅
5     for each c ∈ C : T(c, ŝ) > 0 do
6     begin
7        D := D ∪ {c}
8        for each s ∈ S do T(c, s) = 0
9     end
10    S := S \ {ŝ}
11    Ī := Ī \ {ŝ}
12    for each ĉ ∈ D do Alg(ĉ, Ī, S)
13 end
```

The algorithms presented here can be easily extended to take into account additional factors such as the costs associated with adding new servers, the revenue gained from new customer sessions as well as the costs incurred when breaking customer SLAs. Utility functions can be used to take into account the influence of these factors when making decisions [20].

## 3 Experimental Environment

In this section, we introduce the experimental environment in which we evaluate the performance of our QoS-aware resource management framework. We use two different experimental setups, the first one with only two Grid servers, the second one with up to nine servers running in a virtualized environment.

### *3.1 Experimental Setup 1*

The first setup consists of two heterogeneous Grid servers, one 2-way Pentium Xeon at 2.4 GHz with 2 GB of memory and one 4-way Pentium Xeon at 1.4 GHz with 4 GB of memory. Both servers run Globus Toolkit 4.0.3 [11] (with the latest patches) on a Sun 1.5.0_06 JVM. Access to the Grid servers is controlled by our

QoS-aware resource manager, running on a separate machine with identical hardware as the first Grid server. This machine is also used for emulating the clients that send requests to the Grid. The machines communicate over a Gigabit network. The focus of our analysis is on the QoS negotiation and resource allocation algorithms and not on the way individual Grid servers are modeled.

As a basis for our experiments, we use three sample services each with different behavior and service demands. The services use the Grid to execute some business logic requiring a given amount of CPU time. The business logic might include calls to external (third-party) service providers which are not part of the Grid environment. The time spent waiting for external service providers is emulated by introducing some sleep time during the processing of service requests. Table 1 shows the CPU service times of the three services at the two Grid servers and the total time emulated waiting for external service providers. The third service does not use any external service providers.

Table 1
Workload service demands.

| | Service 1 | Service 2 | Service 3 |
|---|---|---|---|
| CPU service time on 2-way server (sec) | 6.89 | 4.79 | 5.84 |
| CPU service time on 4-way server (sec) | 7.72 | 5.68 | 6.49 |
| External service provider time (sec) | 2.00 | 3.00 | *na* |

Both of the Grid servers offer all of the three services and they provide the data on Table 1 as part of their server workload model when registering with the resource manager. The resource manager uses this data to construct a performance model of the Grid servers as discussed in Section 2.2.

### *3.2 Experimental Setup 2*

The second setup consists of nine Grid servers deployed in a virtualized environment based on the Xen virtual machine [21]. Using virtualization makes it easy to run multiple Grid servers [22] on the same physical hardware and to experiment with different system topologies. The nine virtualized servers are hosted on two 64 bit machines, one 8-way Pentium Xeon at 2.60 GHz with 9 GB of memory and one 4-way Pentium Xeon at 3.16 GHz with 10 GB of memory. The first machine is hosting seven 1-way servers, whereas the second machine is hosting one 1-way server and one 2-way server. One CPU on each physical machine is assigned to *Domain-0* of Xen and the rest of the CPUs are each assigned exclusively to one virtual server. Every server is running Globus Toolkit 4.0.5 with the latest patches on a Sun 1.5.0_12 JVM with 1 GB of memory available. Finally, the machines are connected using a Gigabit Ethernet. The environment is depicted in Figure 4.

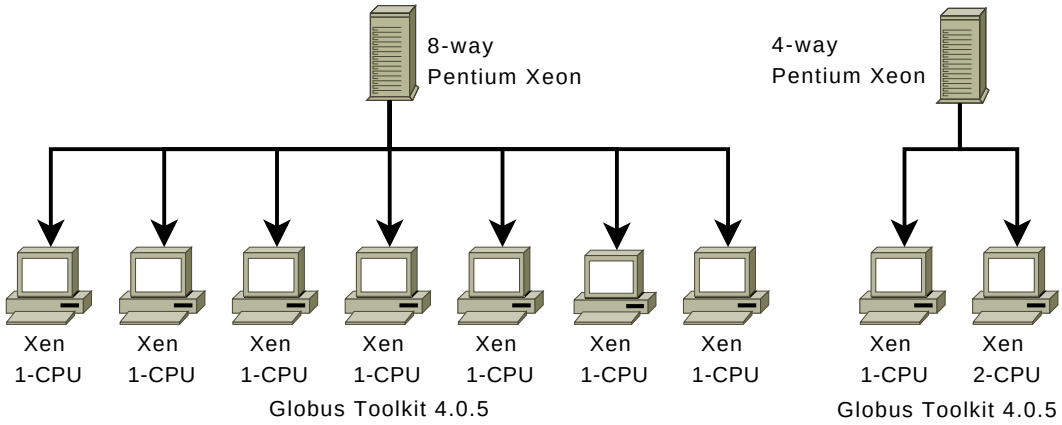


Figure 4. Experimental setup 2.

The QoS-aware resource manager is running on a separate 2-way machine also used for emulating the Grid clients. The same services as in the first setup are used. Table 2 shows the CPU service times of the three services at the virtualized servers and the total time emulated waiting for external service providers.

Table 2
Workload service demands in the virtualized environment.

| | Service 1 | Service 2 | Service 3 |
|---|---|---|---|
| CPU service time on 1-way server (8-way machine) | 7.48 | 5.28 | 6.05 |
| CPU service time on 1-way server (4-way machine) | 7.17 | 5.19 | 6.22 |
| CPU service time on 2-way server (4-way machine) | 7.04 | 5.07 | 6.04 |
| External service provider time (sec) | 2.00 | 3.00 | *na* |

### *3.3 Modeling the Grid Servers*

Each Grid server is modeled using a nested QPN as shown in Figure 5. The nested QPNs are then integrated into the subnet places of the high-level system model in Figure 3. Service requests arriving at a Grid server circulate between queueing place *Server CPUs* and queueing place *Service Providers*, which model the time spent using the server CPUs and the time spent waiting for external service providers, respectively. Place *Server CPUs* contains a $G/M/m/PS$ queue where $m$ is the number of CPUs, whereas place *Service Providers* contains a $G/M/\infty/IS$ queue. For simplicity, it is assumed that the service times at the server CPUs, the request interarrival times and the times spent waiting for external service providers are all exponentially distributed. In the general case this is not required. The firing weights of transition $t_2$ are set in such a way that place *Service Providers* is visited one time for Services 1 and 2 and it is not visited for Service 3. The QPN models of

the two Grid servers were validated and shown to provide accurate predictions of performance metrics (with error below 10%). The model solver component of the QoS Predictor was implemented using SimQPN - our highly optimized simulation engine for QPNs [19].

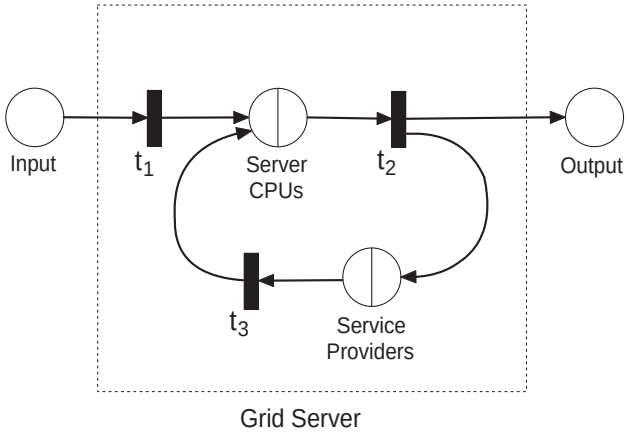


Figure 5. Grid server model.

An alternative approach to model the Grid servers is to use a general purpose simulation system such as OMNeT++ [12] which is based on message passing. We have used OMNeT++ successfully to model a Tomcat Web server [23] with software admission control. With OMNeT++ an arbitrary complex system can be modeled since it allows component behavior to be described using a general purpose programming language like C++. The increase in modeling power, however, comes at the price of the extra effort needed to build the models. In this paper, we use both QPN models (solved using SimQPN) and OMNeT++ models. Figure 6 compares the predicted response times of a 1-way server against measurements on the real system. A number of different configurations under different session mixes, thread allocations and request arrival rates were analyzed and in each case the model predictions were compared against measurements of the real system. Several concrete scenarios that we studied were presented in [6]. The results showed that both QPN and OMNeT++ models provided very consistent and accurate predictions of performance metrics. In all cases, the modeling error was below 15%.

Figure 7 compares the precision of interval estimates provided by SimQPN and OMNeT++ when running the simulation for a fixed amount of CPU time. The precision is measured in terms of the maximum width of 95% confidence intervals for request response times. For run times below 1 second, SimQPN provides slightly wider confidence intervals than OMNeT++, however, there is hardly any difference for run times greater than 1 second. At the same time, while OMNeT++ results are limited to request response times, SimQPN results are more comprehensive and include estimates of request throughputs, server utilization, queue lengths, etc. Moreover, QPN models have the advantage that they are much easier to build and, as discussed in Section 2.2, they can be hierarchically composed which facilitates

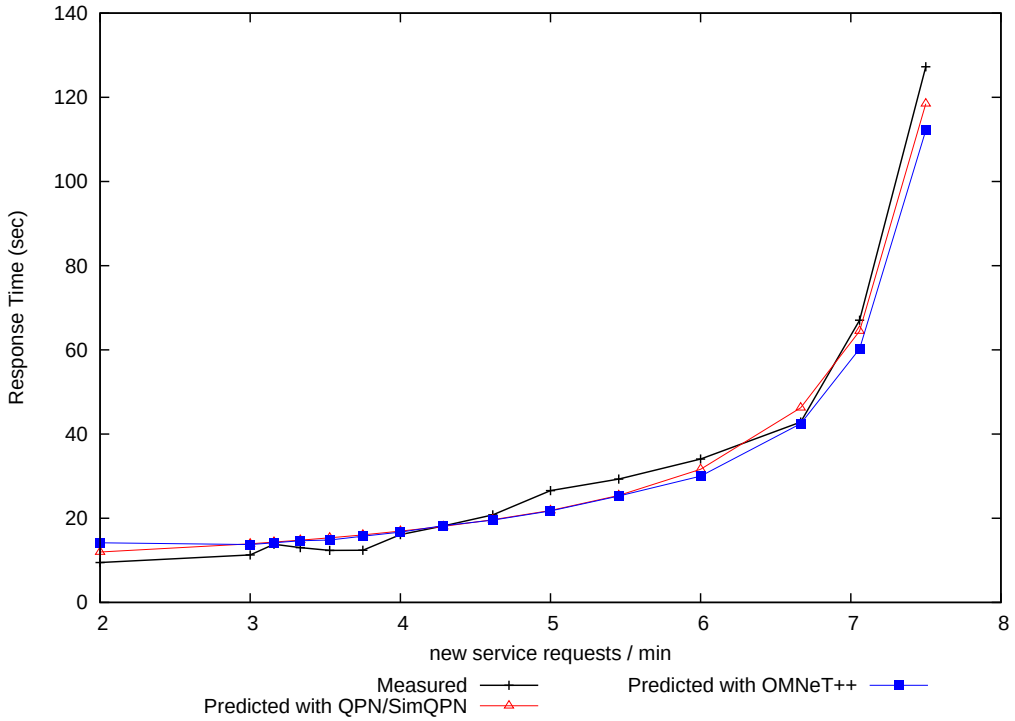


Figure 6. Predicted vs. measured session response times (sec) on a 1-way server

the dynamic model construction. The hierarchical composition is essential since it introduces a clear separation of the high-level system model from the individual Grid server models. The latter can be developed independently without knowing in which environment they will be used.

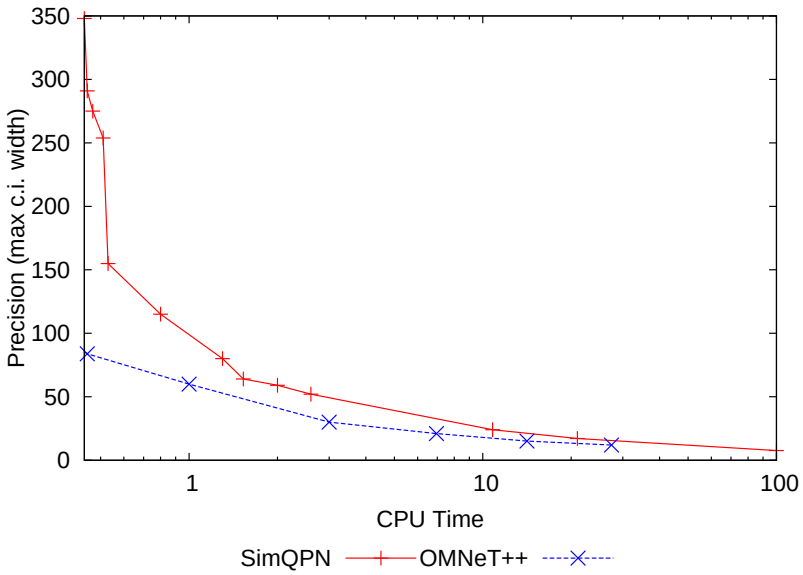


Figure 7. Precision of interval estimates provided by SimQPN and OMNeT++ for a given simulation run time (sec).

### *3.4 Client Emulation*

We have developed a client emulator framework that emulates client sessions sending requests to the Grid environment. Each experiment runs a set of client sessions with given SLAs on the response time of service requests. The user can configure the target session mix specifying for every session the service used and the time between successive service requests (the interarrival time). Client requests are received by the service request dispatcher and forwarded to the Grid servers according to the configured scheduling strategy. Figure 8 illustrates the flow of control when processing service requests.

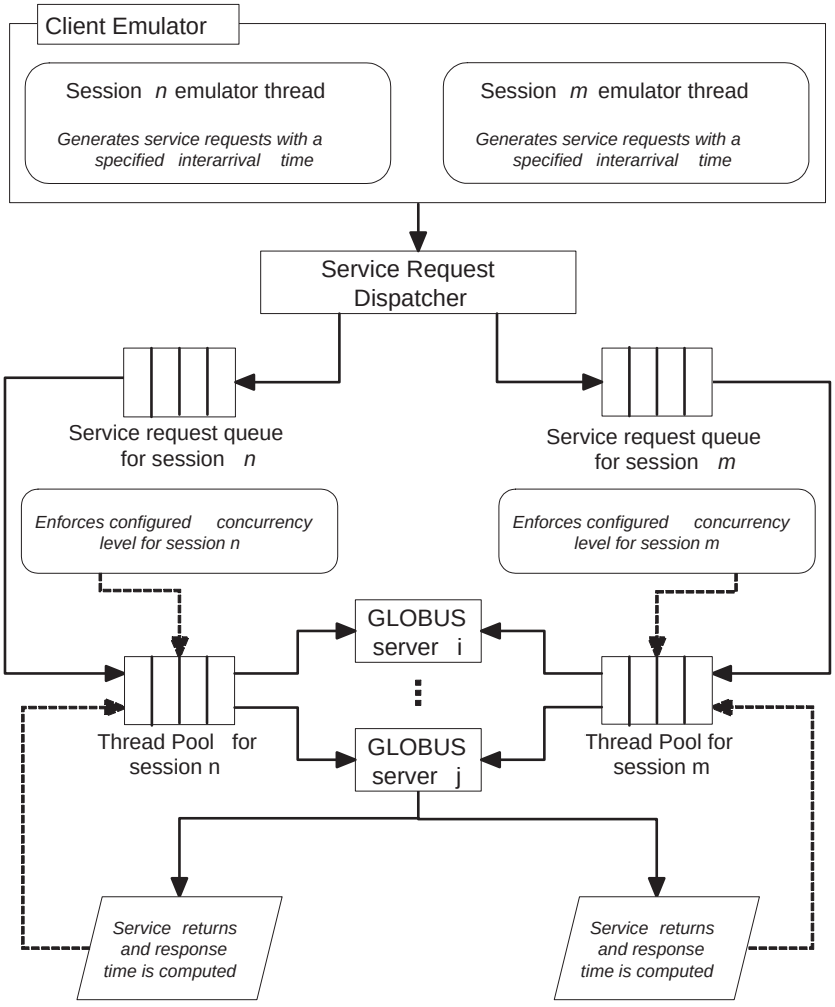


Figure 8. Flow of control when processing service requests.

## 4 Evaluation

We now evaluate the performance of our QoS-aware resource management framework in several scenarios varying the system configuration and workload. The first two scenarios were run in the first experimental setup (see Section 3.1), while the rest of the scenarios were run in the virtualized setup (see Section 3.2).

### *4.1 Scenario 1*

In the first scenario, 16 session requests are sent to the resource manager each with a given throughput and response time SLA. The experiment is run until all accepted sessions complete. The session length, in terms of the number of service requests sent before closing a session, varies between 20 and 120 with an average of 65. The response time SLA ranges between 16 and 30 seconds. We compare the behavior of the system in two different configurations - "with QoS Control" vs. "without QoS Control". In the first configuration, the resource manager applies admission control using our resource allocation framework to ensure that SLAs are honored. In the second configuration, the resource manager simply load-balances the incoming requests over the two servers without considering QoS requirements. For both Grid servers, we assume that there is a 90% maximum server utilization constraint. The experiment was repeated 10 times for each of the two configurations to evaluate the variability of measured data.

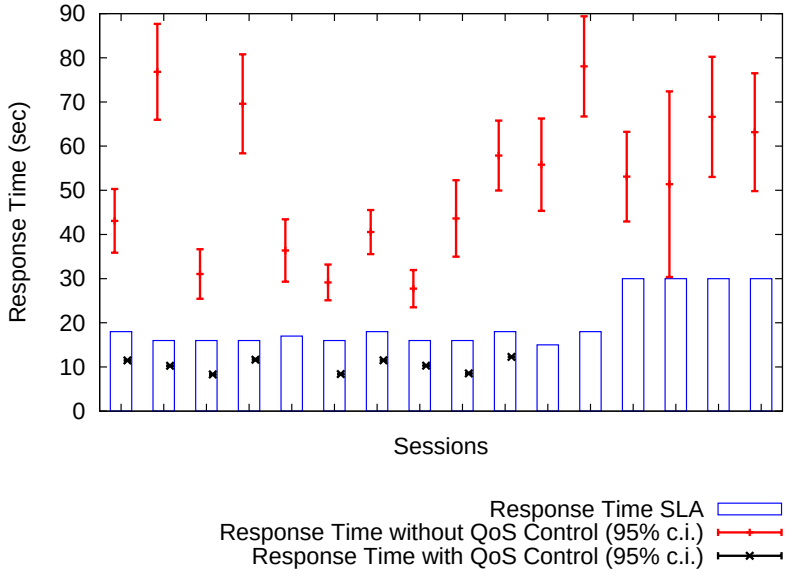


Figure 9. Response time obtained with QoS Control vs. without QoS Control.

Without QoS Control both servers are overloaded during the first half of the experiment exceeding their targeted maximum utilization. In contrast, when running with

QoS Control enabled, some session requests are rejected and the server utilization does not exceed its target upper bound of 90%. Figure 9 shows the measured average response times of sessions with and without QoS Control (95% confidence intervals are given) [3] . As seen from the results, when QoS Control is enabled, response times are very stable and all SLAs are fulfilled. The system exhibits very stable behavior from one iteration of the experiment to the next and the confidence intervals are very narrow given that they are computed for the mean of a quantity which is itself an average value (i.e. average request response time). In contrast, without QoS Control, due to the fact that the Grid servers are overloaded, the system exhibits very variable response times and the client requested SLAs are broken. The confidence intervals are by far much wider in this case. Further details on the above experiment including throughput and utilization data, can be found in [8].

### *4.2 Scenario 2*

In this scenario, a longer experiment was run in which 99 sessions were executed over a period of 2 hours. Figure 10 shows the response time results. The average session duration was 18 minutes in which 92 service requests were sent on average. When running without QoS control, the system was configured to automatically reject session requests during periods in which both Grid servers were completely saturated (overload control). While this improved the average response times of accepted sessions, the response times were still too high when running without QoS control and the SLAs were violated. In contrast, with QoS control, the response times of accepted sessions were much lower and all SLAs were fulfilled. The experiment was repeated for a number of different workload configurations varying the transaction mix, the average session length, the Grid server utilization, etc. The results were of similar quality as the ones presented above and they confirmed the effectiveness of our resource manager architecture in ensuring that QoS requirements are continuously met.

### *4.3 Scenario 3: Workload Characterization On-The-Fly*

In this scenario, we evaluate the effectiveness of our framework when no information on the service behavior and resource consumption is available. Workload characterization is performed on-the-fly using our technique presented in Section 2.4. The experiment was conducted in the virtualized setup with 9 Grid servers. A total of 85 sessions were run over a period of 2 hours. The experiment was repeated for several different configurations:

[3] Note that in the case with QoS Control, the response time is only shown for sessions that were accepted by the resource manager. The sessions are ordered by the time they were started.

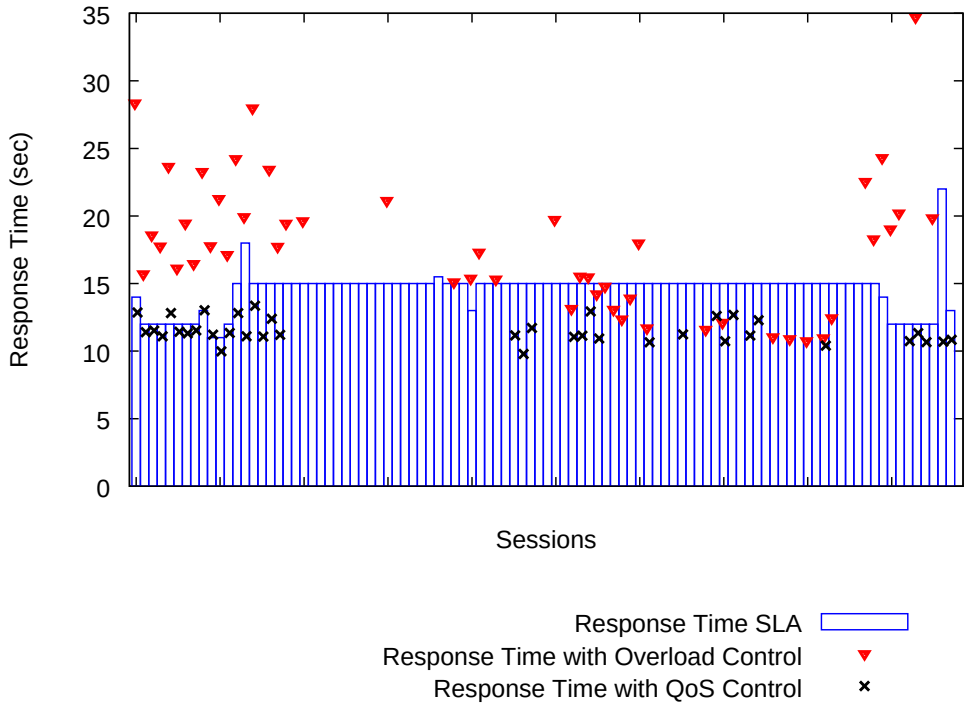


Figure 10. Response time obtained with QoS Control vs. without QoS Control.

**Config 1** Overload control: reject new session requests when server utilization exceeds a specified threshold.

**Config 2** QoS control with workload model available in advance.

**Config 3** QoS control with workload characterization done on-the-fly using the basic algorithm.

**Config 4** QoS control with workload characterization done on-the-fly using the enhanced algorithm.

In the first configuration, server resources were allocated on a round-robin basis without QoS control and the resource manager was configured to reject new session requests when the utilization of all servers exceeds 70%. In the second configuration, QoS Control was enabled and the resource manager was configured to use predefined workload parameters obtained through offline workload characterization. In the third and fourth configurations, workload characterization was performed on-the-fly using the basic and enhanced algorithm, respectively. Figure 11 shows the measured mean response times of accepted sessions in the four configurations. The sessions are ordered by the time the respective session request is sent, however, since most sessions are long their execution overlaps in time. As a result, session requests sent in the middle are rejected since the Grid is saturated at this point. As sessions start to finish, resources are freed and new sessions start being accepted again. The estimated service workload parameters (CPU service demands and time spent waiting for external service providers) for six arbitrarily selected servers in the third and fourth configurations are shown in Table 3. As we can see,

the accuracy of the estimated workload parameters varies from server to server depending on the average server load and the amount of measurement data available for the different services. As time progresses, the estimates become more and more accurate since more data becomes available.

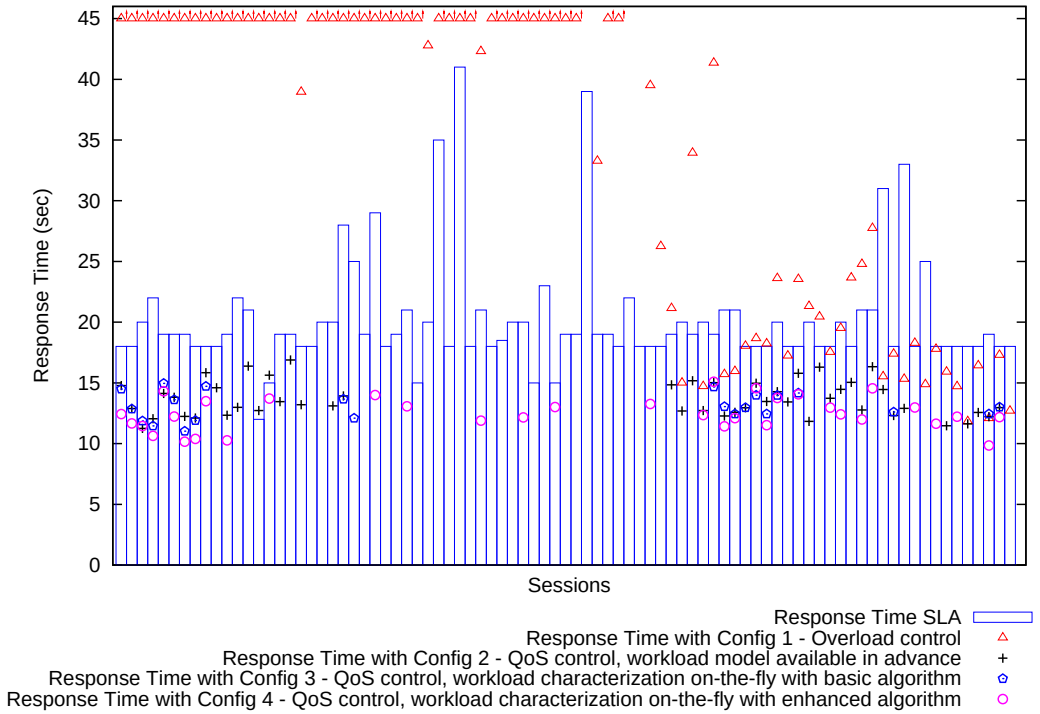


Figure 11. Response times obtained with workload characterization on-the-fly.

Table 4 presents a break down of the client sessions into i) sessions for which the client SLA was observed, ii) sessions for which the client SLA was violated and iii) sessions that were rejected by the resource manager. Without QoS Control, 96% of the requested sessions were admitted, however, the client SLAs were observed in only 22% of them. In contrast to this, in all configurations with QoS Control, the SLAs were observed for nearly 100% of the accepted sessions. Indeed, only 2 sessions had their SLAs violated and the violation was by a tiny margin, more specifically, the measured response times were only 1% higher than agreed. As expected, the price for performing workload characterization on-the-fly is that more sessions are rejected since conservative estimates of the service demands are used. In the third configuration, with the basic workload characterization algorithm, only 22 sessions (26%) were accepted compared to 48 (54%) in the case with offline workload characterization (second configuration). In the fourth configuration, however, with the enhanced algorithm, the penalty for not knowing the workload parameters in advance was much less significant and only 14 sessions (16%) more compared to the case with with offline workload characterization were rejected. As we can see, despite of the fact that the service demand estimates were very rough (Table 3), the resource allocation algorithm still performed quite well in distributing the load among the available servers and ensuring that the SLAs of accepted

Table 3
CPU service demands (Γ) and time spent waiting for external service providers (Ψ) in milliseconds obtained in the 2nd, 3rd and 4th configuration for six randomly selected servers.

| **Server** | **Service** | **Configuration** | | | | | |
|---|---|---|---|---|---|---|---|
| | | **2** | | **3** | | **4** | |
| | | Γ | Ψ | Γ | Ψ | Γ | Ψ |
| 1-way | 0 | 7480 | 2000 | 9423 | 0 | 6973 | 2450 |
| | 1 | 5280 | 3000 | 8570 | 0 | 4799 | 3771 |
| | 2 | 6050 | 0 | 6940 | 0 | 6662 | 278 |
| 1-way | 0 | 7480 | 2000 | 9513 | 0 | 7040 | 2473 |
| | 1 | 5280 | 3000 | 8570 | 0 | 3599 | 4971 |
| | 2 | 6050 | 0 | 6720 | 0 | 6384 | 336 |
| 1-way | 0 | 7480 | 2000 | 9423 | 0 | 4052 | 5371 |
| | 1 | 5280 | 3000 | 8570 | 0 | 4199 | 4371 |
| | 2 | 6050 | 0 | 6720 | 0 | 5309 | 1411 |
| 1-way | 0 | 7480 | 2000 | 9423 | 0 | 6502 | 2921 |
| | 1 | 5280 | 3000 | 8570 | 0 | 4799 | 3771 |
| | 2 | 6050 | 0 | 6720 | 0 | 4701 | 1719 |
| 1-way | 0 | 7040 | 2000 | 9423 | 0 | 6973 | 2450 |
| | 1 | 5070 | 3000 | 8570 | 0 | 3685 | 4885 |
| | 2 | 6040 | 0 | 6720 | 0 | 4838 | 1882 |
| 2-way | 0 | 7170 | 2000 | 9423 | 0 | 6973 | 2450 |
| | 1 | 5190 | 3000 | 8570 | 0 | 4799 | 3771 |
| | 2 | 6220 | 0 | 6732 | 0 | 6395 | 337 |

sessions are satisfied. Our experience showed that for scenarios where the Grid servers are not continuously saturated, workload characterization on-the-fly is a viable option.

### *4.4 Scenario 4: Grid Servers Added On Demand*

In this scenario, we evaluate the effectiveness of our framework when servers are added on demand as explained in Section 2.5. Again, the experiment was conducted in the virtualized setup with 9 Grid servers. A total of 85 sessions were run over a period of 2 hours. This time the experiment was repeated for the following four

Table 4
Summary of session SLA compliance in Scenario 3.

| Configuration | SLA fulfilled | SLA violated | Sessions rejected |
|---|---|---|---|
| 1 | 19 | 63 | 3 |
| 2 | 46 | 2 | 37 |
| 3 | 22 | 0 | 63 |
| 4 | 34 | 0 | 51 |

configurations:

**Config 1** Overload control with all nine servers available from the beginning.
**Config 2** QoS control with all nine servers available from the beginning.
**Config 3** Overload control with one server available in the beginning and servers added on demand.
**Config 4** QoS control with one server available in the beginning and servers added on demand.

The first two configurations are the same as in the previous scenario and they assume that all nine servers are available from the beginning of the experiment. The third and fourth configurations assume that only one server is available in the beginning and servers are added on demand. In the former, a new server is added to the Grid whenever the utilization of all available servers is over 70% and a new session request arrives. In the latter, a new server is added whenever the QoS requested by a client cannot be provided using the currently available server resources. The main difference in the case of adding servers on demand is that resources are allocated incrementally and thus the space of possible configurations searched when allocating threads is reduced. Therefore, a deployment with all servers available from the beginning allows for better load balancing since the space of possible workload distributions is larger.

Figure 12 shows the measured mean response times of accepted sessions in the four configurations. Table 5 presents a summary of the session SLA compliance. The results show that adding servers on demand does not have a significant impact on the performance of the resource manager even though as mentioned above there is less flexibility in distributing the workload. The algorithm can be easily extended to take into account additional factors such as the costs associated with adding new servers, the revenue gained from new customer sessions as well as the costs incurred when breaking customer SLAs. Utility functions can be used to take into account the influence of these factors when making decisions [20].

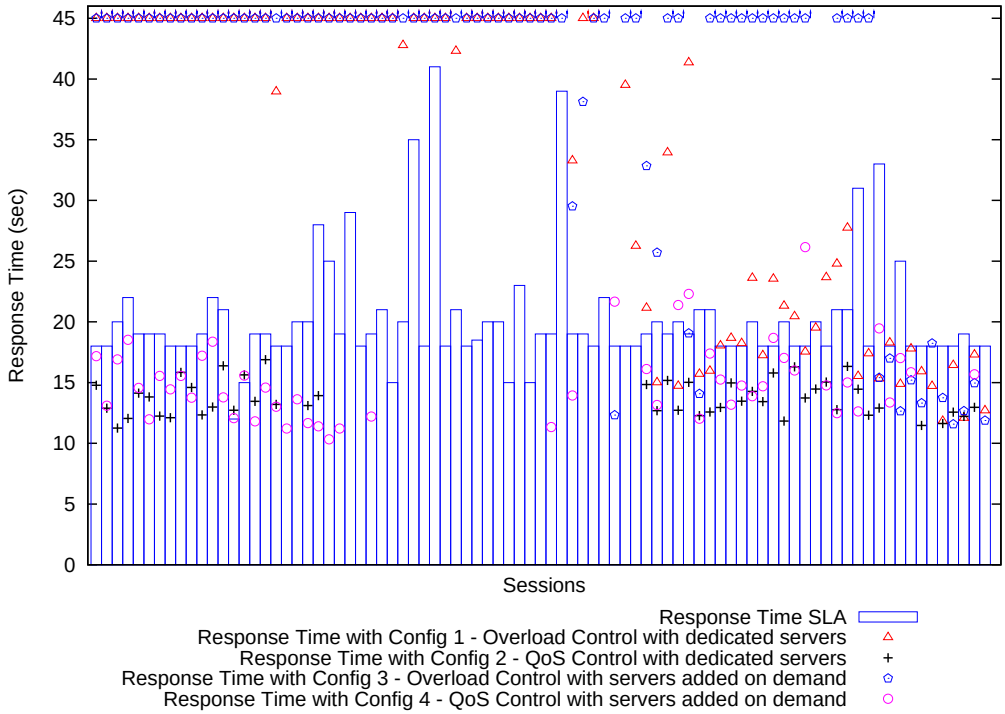


Figure 12. Response times obtained with servers added on demand.

Table 5
Summary of session SLA compliance in Scenario 4.

| Configuration | SLA fulfilled | SLA violated | Sessions rejected |
|---|---|---|---|
| 1 | 19 | 63 | 3 |
| 2 | 46 | 2 | 37 |
| 3 | 15 | 61 | 9 |
| 4 | 45 | 7 | 33 |

### *4.5 Scenario 5: Dynamic Reconfiguration After Server Failure*

In this last scenario, we consider dynamic system reconfiguration after a Grid server failure. As previously, the experiment was conducted in the virtualized setup with 9 Grid servers and a total of 85 sessions were run over a period of 2 hours. The experiment was repeated in five settings each with different number of emulated server failures, from 1 to 5. The points at which server failures were emulated were chosen randomly during the 2 hours. Each time a server failure is detected the resource manager reconfigures all sessions that had threads allocated on the failed server. This is done using the resource allocation algorithm considering the affected sessions as new client requests. Existing sessions might have to be cancelled in case there are not enough resources available to provide adequate QoS.

Table 6
Summary of session SLA compliance in Scenario 5.

| Failures emulated | Without QoS Control | | | With QoS Control | | |
|---|---|---|---|---|---|---|
| | SLA fulfilled | SLA violated | Sessions rejected | SLA fulfilled | SLA violated | Sessions rejected |
| 1 | 14 | 62 | 9 | 37 | 1 | 47 (0) |
| 2 | 16 | 57 | 12 | 39 | 3 | 43 (1) |
| 3 | 10 | 58 | 17 | 40 | 3 | 42 (2) |
| 4 | 3 | 56 | 26 | 38 | 1 | 46 (6) |
| 5 | 4 | 45 | 36 | 31 | 4 | 50 (13) |

Figure 13 shows the measured mean response times of accepted sessions for the five iterations of the experiment. Table 6 presents a summary of the session SLA compliance when running with vs. without QoS Control. In the case without QoS Control, the proportion of cancelled sessions after a system failure is shown in parentheses in the last column. As we can see, when run in a reactive manner the resource manager still does a good job in distributing the workload among the available resources and ensuring that client SLAs are fulfilled.

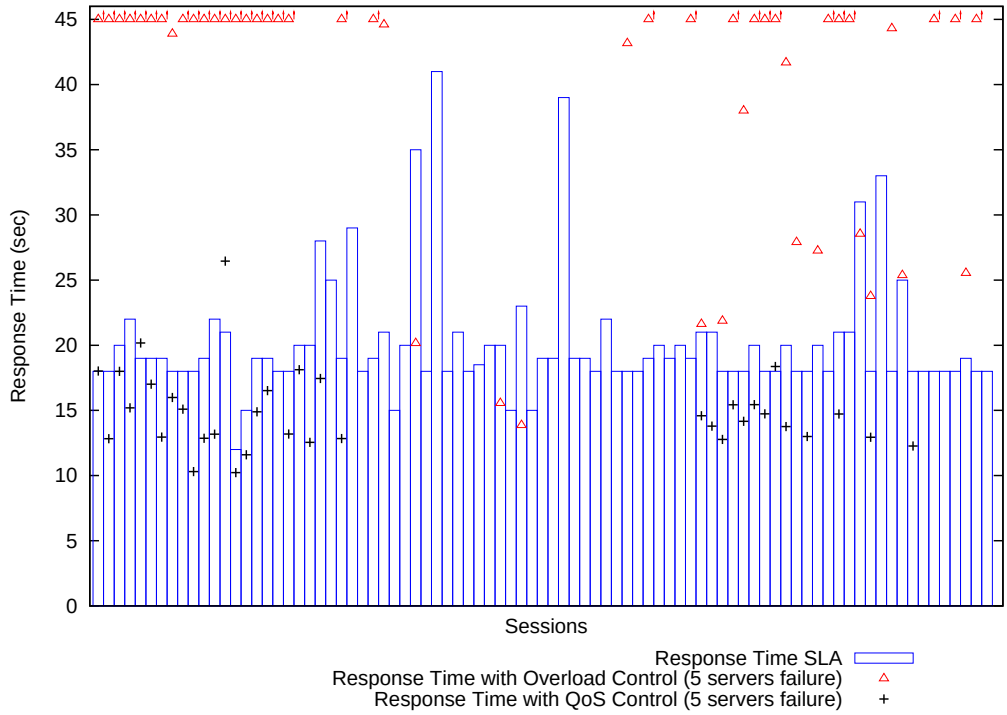


Figure 13. Response times obtained in an experiment with 5 server failures.

An important goal of our resource management framework is to minimize the overhead for evaluating alternative configurations using the resource allocation algorithm and the QoS Predictor when making scheduling decisions. The more detailed the workload models, the higher the overhead for QoS Control. Thus, there is a trade-off between the quality of the resource allocation decisions and the efficiency of the resource manager. To limit the time needed to reach a decision in the scenarios we considered, the simulation was configured to run for a fixed length of model time. Even though the results were approximate, our experiments showed that they were accurate enough to guarantee acceptable QoS for admitted sessions. In scenarios 3, 4 and 5, the average time to reach a decision was 14.81 seconds with a maximum of 37.35 seconds.

There are several approaches to speed up the resource allocation algorithm. For example, we could significantly cut the simulation time for a candidate configuration if we simplify the models by aggregating sessions that execute the same service. Currently, in our QPN models every session is modeled separately using its own token type. If two sessions execute the same service and their request arrivals can be modeled as Poisson processes, the two sessions can be combined into one aggregate session whose arrival stream is the composition of the two Possion processes. This has the potential to significantly reduce the size of the model at the price of less flexibility in load-balancing the workload given that sessions of the same type would be indistinguishable. Using this approach, we were able to simulate a scenario with 40 Grid servers and 80 concurrent sessions in less than 60 sec. Given that QoS Control is done only in the beginning of a session, we believe this is acceptable for a large class of applications.

Another approach we are currently working on is to enhance the SimQPN simulation engine to take advantage of parallelism so that multi-core processors can be utilized for faster simulation. Currently, parallelism can only be exploited by analyzing multiple candidate configurations simultaneously, however, every individual configuration is simulated sequentially. For larger and more complex models, some significant gains can be made by parallelising the simulation. Further, we intend to investigate under what conditions analytical product form solution techniques for QPNs or approximation techniques can be exploited.

Finally, as part of our future work, we intend to use efficient combinatorial search techniques to minimize the number of candidate configurations examined when searching for an acceptable resource allocation. In particular, we plan to investigate the trade-off between bottom up and top down resource allocation methods and consider caching and reusing analyzed configurations as well as simulating configurations proactively.

## 5 Related Work

There is a large body of work on resource management and QoS in Grid computing environments and service-oriented architectures in general.

The Globus Resource Management Architecture (GRMA) [10] addresses the relatively narrow QoS problem of providing dedicated access to collections of computers in heterogeneous distributed systems. The General-purpose Architecture for Reservation and Allocation (GARA) [24] generalizes and extends GRMA to support advance reservation and co-allocation of *heterogeneous* collections of resources (e.g., computers, networks or storage systems) for end-to-end QoS. The GARA system comprises a number of resource managers that each implement reservation, control, and monitoring operations for a specific resource. In [25], the authors extend their work combining features of reservation and adaptation. The above mechanisms provide a basic framework for managing QoS in Grid applications based on simple ad hoc procedures used to map application QoS requirements into resource requirements. As such these mechanisms do not possess any sophisticated performance prediction capabilities that are required to guarantee that application SLAs are honored. Furthermore, being targeted at high-end applications, they do not provide support for *fine-grained* QoS-aware load-balancing which is essential for commercial workloads.

In [3] a framework for resource allocation in Grid computing is presented. The authors consider the general case in which applications are decomposed into tasks that exhibit precedence relationships. The problem consists in finding the optimal resource allocation that minimizes total cost while preserving execution time service level agreements. A framework for building heuristic solutions for this NP-hard problem is developed. In [26] the authors show how analytic queueing network models combined with combinatorial search techniques can be used to develop methods for optimal resource allocation in autonomic data centers. The above works however do not deal with the problem of QoS negotiation and enforcement.

In [27], a framework for designing QoS-aware software components is proposed. The authors introduce so-called Q-components that negotiate soft QoS requirements with clients (i.e., average response time and throughput) and use online analytic performance models (more specifically closed multiclass queueing networks) to ensure that client requests are accepted only if the requested QoS can be provided. The same approach was applied in [5] to the design of QoS-aware service-oriented architectures. While these works provide some basic support for negotiating and enforcing QoS requirements in loosely-coupled SOA environments, they do not completely decouple service users from service providers and therefore suffer from several significant drawbacks. For example, *fine-grained* load-balancing at the service request level is not provided. Moreover, the resource allocation and load-balancing strategy of a client session cannot be dynamically reconfigured. Fi-

nally, these methods being based on product-form queueing networks are rather limited in terms of modeling accuracy and expressiveness.

An alternative approach to autonomic resource allocation in multi-application data centers based on reinforcement learning is proposed in [28]. Instead of using explicit performance models, this approach uses a knowledge-free trial-and-error methodology to learn resource valuation estimates and construct decision-theoretic optimal policies. In [29] the authors extend their approach to support offline training on data collected while an externally supplied initial policy (based on an explicit performance model) controls the system. An active learning approach to resource allocation for simple batch workloads is proposed in [30]. This approach uses performance histories gathered through noninvasive instrumentation to build predictive models of frequently used applications. The approach however is focused on compute batch tasks that run to completion at machine speed. Request arrivals and concurrency related behavior is not considered.

In [31] a QoS guided task scheduling algorithm for Grid computing is proposed. The algorithm uses a long-term, application-level prediction model to estimate the task completion time in a non-dedicated environment. Based on the same model a performance prediction and task scheduling system called Grid Harvest Service was developed [32]. The focus of this work is on long-term (long-running) applications. In [33] a performance management system for cluster-based web services is presented. The system supports multiple classes of web services traffic and allocates server resources dynamically with the goal to maximize the expected value of a given cluster utility function in the face of fluctuating loads. Simple queueing models are used for performance prediction. This framework currently does not support QoS negotiation and admission control.

Several approaches to autonomic workload characterization have recently been proposed, for example in [34] based on fuzzy logic, in [35] based on genetic algorithms and finally in [36] based on traces used to generate behavior patterns. The latter technique is however not applicable in our environment since it assumes a homogeneous workload.

Two industrial standards related to our QoS negotiation and resource allocation mechanisms are the Job Submission Description Language (JSDL) [14] and the Web Services Agreement Specification (WS-Agreement) [16]. JSDL is used to describe the requirements of computational jobs for submission to resources in Grid environments. JDSL contains a vocabulary and normative XML Schema that facilitate the expression of job identification, resource and data requirements. The WS-Agreement specification, on the other hand, is a Web Services protocol for establishing agreement between two parties using an extensible XML language for specifying the nature of the agreement, and agreement templates to facilitate discovery of compatible agreement parties. The specification consists of three parts: a schema for specifying an agreement, a schema for specifying an agreement tem-

plate, and a set of port types and operations for managing agreement life-cycle, including creation, expiration, and monitoring of agreement states. Although WS-Agreement goes beyond the scope of the current paper, our QoS negotiation protocol can easily be adapted and extended to comply with the WS-Agreement specification. Similarly, the resource manager architecture can be extended to support the specification of service resource requirements using JSDL.

Further related work in the area of resource management and QoS in Grid computing and SOA environments can be found in [37], [38], [39], [40], [41], [42], [43] and [44]. The resource manager architecture we proposed can be extended to take into account the trade-offs between performance and power consumption (and energy efficiency) similar to the way this is done in [45].

## 6 Conclusions

In this paper, we proposed a comprehensive framework for autonomic QoS control in enterprise Grid environments using online simulation. We first presented a novel methodology for designing autonomic QoS-aware resource managers that have the capability to predict the performance of the Grid components they manage and allocate resources in such a way that SLAs are honored. We then proposed a method for autonomic workload characterization on-the-fly based on monitoring data. This makes it possible to apply our approach to services for which no workload model is available. Following this, we further extended our resource allocation algorithm to support adding Grid servers on demand as well as dynamically reconfiguring the system after a server failure.

We presented an extensive performance evaluation of our framework in two different experimental environments using the Globus Toolkit. Five different scenarios each focusing on selected aspects of the framework were studied. In all scenarios with QoS Control enabled, the measured response times were stable and nearly 100% of the client SLAs were fulfilled. The results confirmed the effectiveness of our resource manager architecture in ensuring that QoS requirements are continuously met. The proposed method for workload characterization on-the-fly proved to be quite effective in providing estimates of the service resource demands based on online monitoring data. We saw that adding servers on demand did not have a significant impact on the performance of our resource manager even though as expected in this case there is less flexibility in distributing the workload. Finally, we demonstrated the effectiveness of our scheduling algorithms for dynamic reconfiguration after a server failure.

The evaluation presented in this paper is the first comprehensive validation of our methodology in a dynamic environment. The results were very encouraging and demonstrated the effectiveness, practicality and performance of the approach. The

area considered here has many different facets that will be subject of future work. First, we intend to pursue ways to further optimize our resource allocation algorithm and performance prediction mechanisms. The trade-offs in using different model analysis techniques and combinatorial search methods to minimize the number of candidate configurations will be investigated. We also intend to consider caching and reusing analyzed configurations as well as simulating configurations proactively. Second, the overhead of the QoS Predictor as the size and complexity of the modeled Grid servers and their workload increase will be evaluated. We intend to consider larger more realistic Grid environments to evaluate the scalability of our approach. While, currently only soft QoS requirements (average values) are guaranteed, we intend to enhance the architecture to support hard QoS requirements. This would make it possible for clients to specify and negotiate SLAs in terms of more detailed performance metrics such as for example the 90% percentiles of response times. Finally, we intend to study how our framework can be extended to take into account the economic aspects involved in using the Grid [46]. The goal will be to extend the framework presented in this paper to take into account economic factors such as the costs associated with using the Grid resources, the revenue gained from new customer sessions, as well as the costs incurred and penalties imposed when breaking customer SLAs.

## Acknowledgment

This work is supported by the Ministry of Science and Technology of Spain and the European Union under contract TIN2004-07739-C02-01 and TIN2007-60625, and the German Research Foundation under grant KO 3445/1-1. Thanks to Iñigo Goiri for his help on virtualization issues.

## References

[1] I. Foster, C. Kesselman, J. M. Nick, S. Tuecke, Grid Services for Distributed System Integration, Computer 35 (6) (2002) 37–46.

[2] OGF, Open Grid Forum, www.ogf.org (2008).

[3] D. Menascé, E. Casalicchio, A Framework for Resource Allocation in Grid Computing, in: Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.

[4] D. Menascé, V. Almeida, L. Dowdy, Performance by Design, Prentice Hall, 2004.

[5] D. Menascé, M. Bennani, H. Ruan, Self-Star Properties in Complex Information Systems, Vol. 3460 of LNCS, Springer Verlag, 2005, Ch. On the Use of Online

Analytic Performance Models in Self-Managing and Self-Organizing Computer Systems.

[6] R. Nou, S. Kounev, J. Torres, Building Online Performance Models of Grid Middleware with Fine-Grained Load-Balancing: A Globus Toolkit Case Study, in: 4th European Performance Engineering Workshop on Formal Methods and Stochastic Models for Performance Evaluation, EPEW, Vol. 4748 of Lecture Notes in Computer Science, 2007, pp. 125–140.

[7] F. Bause, P. Buchholz, P. Kemper, Hierarchically Combined Queueing Petri Nets, in: Proceedings of the 11th International Conference on Analysis and Optimization of Systems, Discrete Event Systems, 1994.

[8] S. Kounev, R. Nou, J. Torres, Autonomic QoS-Aware Resource Management in Grid Computing using Online Performance Models, Second International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS-2007), Nantes, France.

[9] S. Kounev, Performance Modeling and Evaluation of Distributed Component-Based Systems using Queueing Petri Nets, IEEE Transactions on Software Engineering 32 (7) (2006) 486–502.

[10] I. Foster, C. Kesselman, The Grid 2: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, 2003.

[11] I. Foster, Globus Toolkit Version 4: Software for Service-Oriented Systems, in: Proceedings of the 2005 IFIP International Conference on Network and Parallel Computing, 2005, pp. 2–13.

[12] A. Varga, The OMNeT++ Discrete Event Simulation System, in: Proceedings of the European Simulation Multiconference (ESM'2001), The Society for Modeling and Simulation International (SCS), Prague, Czech Republic, 2001.

[13] R. Chinnici, J.-J. Moreau, A. Ryman, S. Weerawarana, Web Services Description Language (WSDL) Version 2.0, Tech. rep., W3C, http://www.w3.org/TR/wsdl20 (mar 2006).

[14] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, A. Savva, Job Submission Description Language (JSDL) Specification, `http://www.gridforum.org/documents/GFD.56.pdf` (2005).

[15] R. Nou, F. Julià, J. Torres, Should the grid middleware look to self-managing capabilities?, in: Proceedings of the 8th International Symposium on Autonomous Decentralized Systems, 2007.

[16] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, M. Xu, Grid Resource Allocation Agreement Protocol WG (GRAAP-WG), `https://forge.gridforum.org/projects/graap-wg/` (2007).

[17] F. Bause, "QN + PN = QPN" - Combining Queueing Networks and Petri Nets, Technical report no.461, Department of CS, University of Dortmund, Germany (1993).

[18] F. Bause, P. Buchholz, Queueing Petri Nets with Product Form Solution, Performance Evaluation 32 (4) (1998) 265–299.

[19] S. Kounev, A. Buchmann, SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation, Performance Evaluation 63 (4-5) (2006) 364–394.

[20] W. E. Walsh, G. Tesauro, J. O. Kephart, R. Das, Utility Functions in Autonomic Systems, Proceedings of the IEEE International Conference on Autonomic Computing (ICAC) (2004) 70–77.

[21] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, in: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, ACM, New York, NY, USA, 2003, pp. 164–177.

[22] R. Figueiredo, P. Dinda, J. Fortes, A case for grid computing on virtual machines, 23rd International Conference on Distributed Computing Systems (2003) 550–559.

[23] R. Nou, J. Guitart, J. Torres, Simulating and Modeling Secure Web Applications., in: 6th International Conference on Computational Science- ICCS 2006, Lecture Notes in Computer Science, 2006, pp. 84–91.

[24] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, A. Roy, A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation, in: Proceedings of the International Workshop on Quality of Service, 1999.

[25] I. Foster, A. Roy, V. Sander, A quality of service architecture that combines resource reservation and application adaptation, in: Proceedings of the 8th International Workshop on Quality of Service, 2000, pp. 181–188.

[26] M. Bennani, D. Menascé, Resource Allocation for Autonomic Data Centers using Analytic Performance Models, in: Proceedings of the Second International Conference on Automatic Computing, 2005.

[27] D. Menascé, H. Ruan, H. Gomaa, A Framework for QoS-Aware Software Components, in: Proceedings of the 4th International Workshop on Software and Performance, 2004.

[28] G. Tesauro, R. Das, W. E. Walsh, J. O. Kephart, Utility-Function-Driven Resource Allocation in Autonomic Systems, in: Proceedings of the Second International Conference on Autonomic Computing, 2005.

[29] G. Tesauro, N. Jong, R. Das, M. Bennani, A hybrid reinforcement learning approach to autonomic resource allocation, in: Proceedings of the 3rd International Conference on Autonomic Computing, 2006.

[30] P. Shivam, S. Babu, J. Chase, Learning Application Models for Utility Resource Planning, in: Proceedings of the 3rd International Conference on Autonomic Computing, 2006.

[31] X. He, X. Sun, G. Laszewski, A QoS Guided Scheduling Algorithm for Grid Computing, in: Proceedings of the International Workshop on Grid and Cooperative Computing, 2002.

[32] X.-H. Sun, M. Wu, Grid Harvest Service: A System for Long-Term, Application-Level Task Scheduling, in: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, 2003.

[33] G. Pacifici, M. Spreitzer, A. N. Tantawi, A. Youssef, Performance management for cluster-based web services, IEEE Journal on Selected Areas in Communications 23 (12) (2005) 2333–2343.

[34] J. Xu, M. Zhao, J. Fortes, R. Carpenter, M. Yousif, On the Use of Fuzzy Modeling in Virtualized Data Center Management, in: Proceedings of the Fourth International Conference on Autonomic Computing (ICAC'07), 2007, p. 25.

[35] A. Andrzejak, S. Graupner, S. Plantikow, Predicting Resource Demand in Dynamic Utility Computing Environments, in: International Conference on Autonomic and Autonomous Systems (ICAS), 2006.

[36] D. Gmach, J. Rolia, L. Cherkasova, A. Kemper, Workload Analysis and Demand Prediction of Enterprise Data Center Applications, IEEE 10th International Symposium on Workload Characterization (IISWC) (2007) 171–180.

[37] N. H. Kapadia, J. Fortes, C. E. Brodley, Predictive Application-Performance Modeling in a Computational Grid Environment, in: 8th IEEE International Symposium on High Performance Distributed Computing, 1999.

[38] R. Al-Ali, K. Amin, G. von Laszewski, O. Rana, D. Walker, M. Hategan, N. Zaluzec, Analysis and Provision of QoS for Distributed Grid Applications, Journal of Grid Computing 2 (2).

[39] C. Adam, R. Stadler, A Middleware Design for Large-scale Clusters Offering Multiple Services, IEEE electronic Transactions on Network and Service Management 3 (1).

[40] R. Al-Ali, O. Rana, G. von Laszewski, A. Hafid, K. Amin, D. Walker, A Model for Quality-of-Service Provision in Service Oriented Architectures, Journal of Grid and Utility Computing.

[41] D. Xu, K. Nahrstedt, A. Viswanathan, D. Wichadakul, QoS and Contention-Aware Multi-Resource Reservation, in: Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing, 2000.

[42] A. Othman, P. Dew, K. Djemamem, I. Gourlay, Adaptive Grid Resource Brokering, in: Proceedings of the 2003 IEEE International Conference on Cluster Computing, 2003, pp. 172–179.

[43] Y. Diao, J. L. Hellerstein, S. Parekh, H. Shaikh, M. Surendra, A. Tantawi, Modeling Differentiated Services of Multi-Tier Web Applications, in: 14th IEEE International Symposium on Modeling, Analysis, and Simulation, 2006.

[44] D. Gmach, S. Krompass, A. Scholz, M. Wimmer, A. Kemper, Adaptive quality of service management for enterprise services, ACM Transactions Web 2 (1) (2008) 1–46.

[45] J. Slegers, I. Mitrani, N. Thomas, Optimal Dynamic Server Allocation in Systems with On/Off Sources, in: 4th European Performance Engineering Workshop on Formal Methods and Stochastic Models for Performance Evaluation, EPEW, Vol. 4748 of Lecture Notes in Computer Science, 2007, pp. 186–199.

[46] R. Buyya, D. Abramson, S. Venugopal, The Grid Economy, Proceedings of the IEEE 93 (3) (2005) 698–714.