# Autonomic QoS-Aware Resource Management in Grid Computing using Online Performance Models

Samuel Kounev
University of Cambridge
Computer Laboratory
Cambridge, CB3 0FD, UK
skounev@acm.org

Ramon Nou
Technical University of Catalonia
Computer Architecture Dept.
Jordi Girona 1-3
E08034 Barcelona, Spain
rnou@ac.upc.edu

Jordi Torres
Barcelona Supercomputing Center
Jordi Girona, 29
E08034 Barcelona, Spain
jordi.torres@bsc.es

## 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, an approach to autonomic QoS-aware resource management in Grid computing based on online performance models 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. 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.

## Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Modeling and prediction*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; C.4 [**Performance of Systems**]: Modeling techniques

## Keywords

QoS control, resource allocation, online performance models, Grid computing, SOA

## 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 now becoming a mainstream paradigm for enterprise applications and distributed system integration [1,12]. 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 [1]. 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.

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 [22]. 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 [21]. 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

*Valuetools* '07, October 23-25, 2007, Nantes, France
Copyright 2007 ICST 978-963-9799-00-4.

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 [20]. 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.

This paper proposes an approach to autonomic QoS-aware resource management in Grid computing based on predictive online performance models. A novel methodology is presented 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 methodology 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 [18] which are commonly used as building blocks of Grid infrastructures [10]. As demonstrated in [5], queueing Petri nets provide greater modeling power and expressiveness than conventional queueing networks and stochastic Petri nets. Thus, being based on queueing Petri nets, our methodology provides flexibility in choosing the level of detail and accuracy at which Grid components are modeled. Finally, although the methodology we propose 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).

To validate our approach, we implemented a prototype of a QoS-aware resource manager and deployed it in the context of a real-world Grid environment based on the Globus Toolkit [15]. We subjected the system to an extensive experimental evaluation comparing its behavior in two different configurations - "with QoS Control" vs. "without QoS Control". The results demonstrate the effectiveness of our approach and its applicability to QoS-aware resource management in Grid environments.

The rest of the paper is organized as follows. Section 2 introduces the Globus Toolkit discussing the protocols and mechanisms it provides for resource management and QoS control. Section 3 presents our methodology for autonomic QoS-aware resource management in Grid environments. In Section 4, the results of our experimental evaluation of the proposed approach are presented. Finally, an overview of some related work is given in Section 5 and the paper is wrapped up in Section 6.

## 2. BACKGROUND

The main problem Grid Computing aims to address was defined in [13] as the "coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations". The goal of these dynamically created virtual environments, referred to as *virtual organizations*, is to enable disparate groups of mutually distrustful participants (organizations and/or individuals) to share resources in a controlled fashion and benefit from non-trivial qualities of service such as improved performance, availability, coordinated fail-over, optimal resource management and utilization, common security semantics, etc.

In [13] the authors present an extensible and open multi-layered Grid architecture comprising five layers: fabric, connectivity, resource, collective and application. The *fabric* layer provides the resources to which shared access is mediated by Grid protocols, for example, computational resources, data storage resources, network resources or sensors. A resource may also be a logical entity, such as high-performance cluster or distributed storage system. The *connectivity* layer provides the core communication and security protocols (authentication and access control) required for Grid-specific transactions. The *resource* layer defines protocols, APIs, and SDKs for secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operations on individual resources. The *collective* layer provides protocols and services that coordinate interactions across collections of resources. Finally, the *application* layer comprises the user applications that operate in the Grid environment.

An implementation of this multi-layered architecture is found in the Globus Toolkit (GT) [15]. The latter is a community-based, open-architecture, open-source set of services and software libraries that support Grids and Grid applications. The toolkit addresses issues of security, information discovery, resource management, data management, communication, fault detection, and portability. Globus Toolkit mechanisms are in use at hundreds of sites and by dozens of major Grid projects worldwide. The latest Web Services-based GT 4 release provides significant improvements over previous releases in terms of usability, robustness, performance, standards compliance and functionality. Below we look at some of the protocols and mechanisms the Globus Toolkit provides for resource management (resource reservation and allocation) and QoS control.

The Globus Resource Management Architecture [10] addresses the relatively narrow QoS problem of providing dedicated access to collections of computers in heterogeneous distributed systems. The architecture consists of three main components: an information service, local resource managers, and various types of co-allocation agents, which implement strategies used to discover and allocate the resources required to meet application QoS requirements. An application that wishes to create a computation passes a description of that computation to a co-allocation agent. This agent uses some combination of information service queries, general heuristics, and application-specific knowledge to map application QoS requirements into resource requirements, to discover resources with those requirements, and to allocate those resources.

The General-purpose Architecture for Reservation and Allocation (GARA) [11] generalizes and extends the Globus Resource Management Architecture 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. Uniform interfaces allow applications to express QoS requirements for different types of resources in similar ways, hence simplifying the development of end-to-end QoS management strategies. In [14] an approach to managing QoS in Grid environments is proposed that combines features of reservations and adaptation. In this approach, a combination of online control interfaces for resource management, a sensor allowing online monitoring, and decision procedures embedded in resources enable a rich variety of dynamic feedback interactions between applications and resources.

The mechanisms discussed above provide a basic framework for managing QoS in Grid applications which is 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.

# 3. 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 guarantee 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. The methodology is meant to be applied at the resource layer of the general Grid architecture outlined in Section 2.

## 3.1 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.

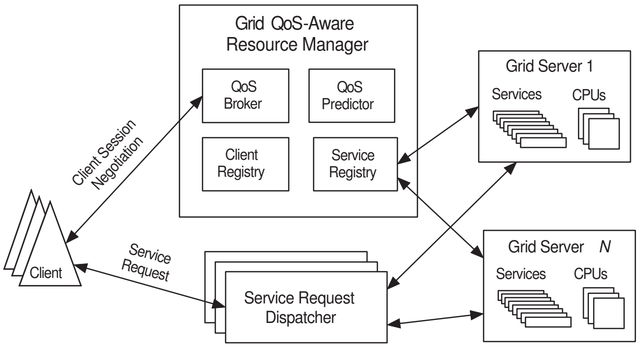

**Figure 1: Resource manager architecture.**

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 [8], 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 of the service demands at the server resources to a detailed mathematical model capturing the flow of control during service execution. 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% [24]. For each server, the service registry keeps track of its maximum allowed utilization by the Grid.

The *QoS broker* receives session requests from clients, al-

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

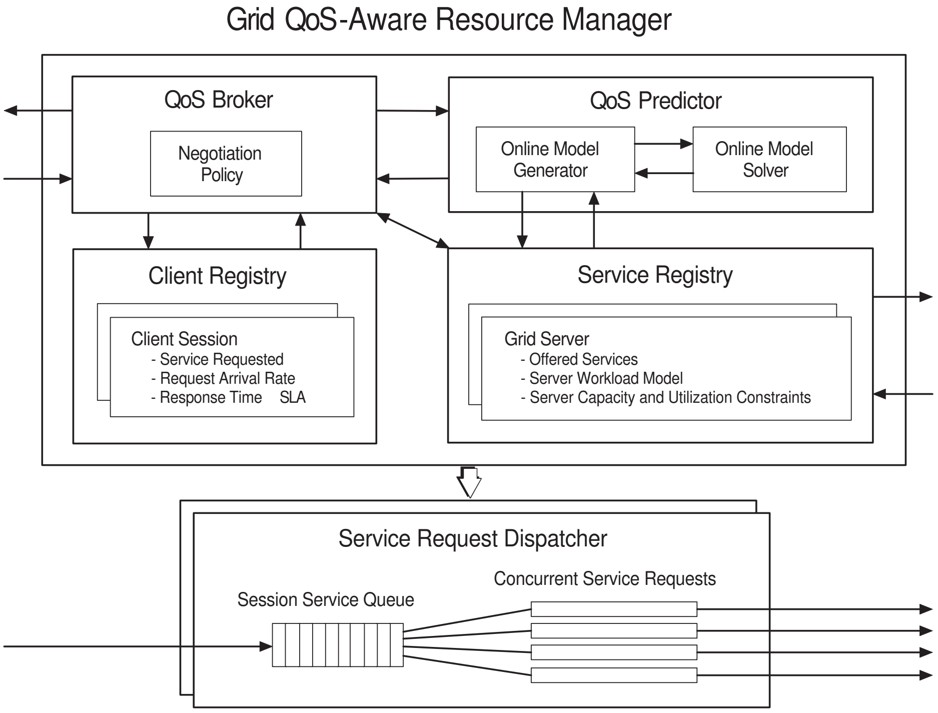

**Figure 2: Inside view of the resource manager architecture.**

locates 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. 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.

### 3.2 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 [5]. In [18], 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 [6] 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.

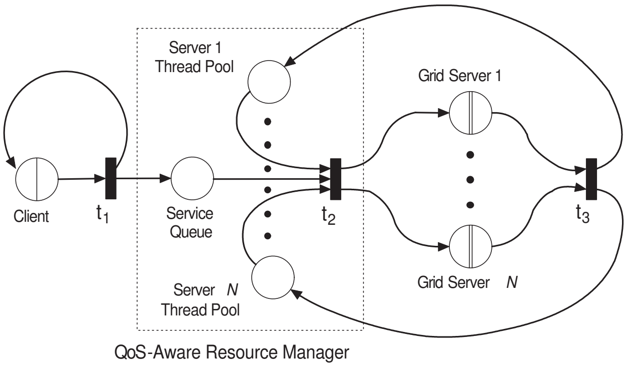

**Figure 3: Generic performance model.**

### 3.3 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 3.1. 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. $X^T(c)$.

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

As discussed in Section 3.1, 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 $c$.

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:

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

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

$I^T(v, \epsilon) \overset{def}{=} \{s \in S : (v \in F(s)) \wedge (A^T(s) \geq \epsilon)\}$

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:

```
$C := C \cup \{\tilde{c}\}$
for each $s \in I^T(v,\epsilon)$ do $T(\tilde{c}, s) := \infty$
if $\big(\exists\ c \in C : \neg P_X^T(c)\big)$ then reject $\tilde{c}$
while $\big(\exists\ \hat{s} \in S : \neg P_U^T(\hat{s})\big)$ do
begin
    $T(\tilde{c}, \hat{s}) := 1$
    while $P_U^T(\hat{s})$ do $T(\tilde{c}, \hat{s}) := T(\tilde{c}, \hat{s}) + 1$
    $T(\tilde{c}, \hat{s}) := T(\tilde{c}, \hat{s}) - 1$
end
if $\big(\exists\ c \in C \setminus \{\tilde{c}\} : \neg P_X^T(c) \vee \neg P_R^T(c)\big)$ then reject $\tilde{c}$
if $\big(\neg P_X^T(\tilde{c}) \vee \neg P_R^T(\tilde{c})\big)$ then
    send counter offer $o = \big(v, X^T(\tilde{c}), R^T(\tilde{c})\big)$
else accept $\tilde{c}$
```

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.

## 4. EXPERIMENTAL ANALYSIS

The autonomic QoS-aware resource manager architecture described in the previous sections was implemented in C++ and subjected to an extensive experimental analysis to evaluate its effectiveness. This section presents the results of our study. For lack of space, we cannot include all configurations considered and what we do instead is present the results from one representative scenario which illustrates our findings.

Our test environment consists of two heterogeneous Grid servers, the first one 2-way Pentium Xeon at 2.4 GHz with 2 GB of memory and the second one 4-way Pentium Xeon at 1.4 GHz with 4 GB of memory. Both servers run Globus Toolkit 4.0.3 (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 services.**

| | 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 3.2. Each Grid server is modeled using a nested QPN (see Figure 4). 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].

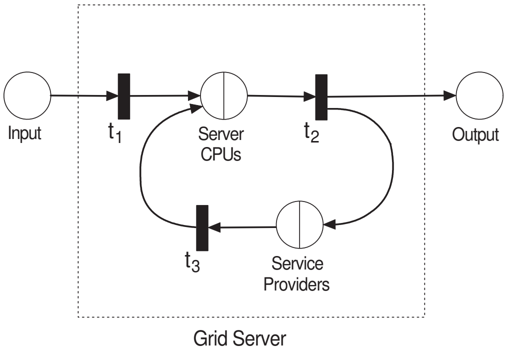

**Figure 4: Grid server model.**

## 4.1 Analysis Results

As a first step in validating our approach we conducted an extensive evaluation of the accuracy of our model-based QoS Predictor in predicting the performance of the Grid environment for a given configuration. 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. The results showed that the QoS Predictor provided very consistent and accurate predictions of performance metrics. In all cases, the modeling error was below 15%.

We now present the results from an experiment in which 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.

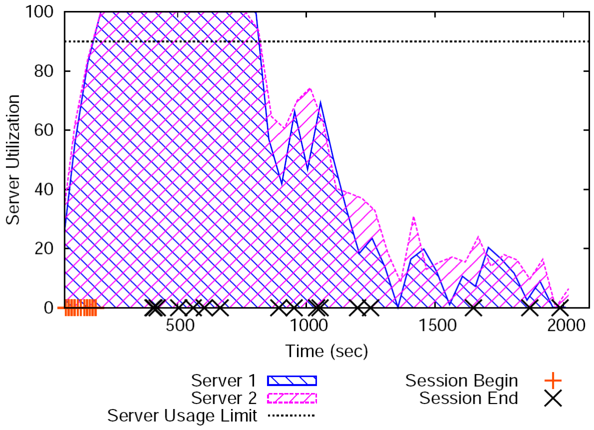

**Figure 5: Server utilization without QoS Control.**

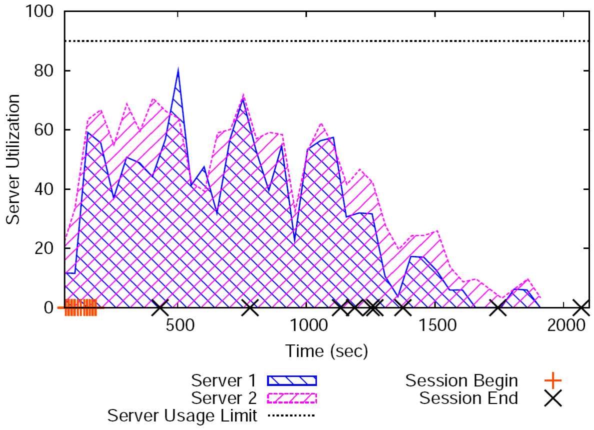

**Figure 6: Server utilization with QoS Control.**

Figures 5 and 6 show the measured CPU utilization of the Grid servers during the experiment in the two configurations. The points at which sessions begin and end are shown on the x-axis. As we can see, 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%. The throughput achieved for each session in the two configurations is depicted in Figure 7 (95% confidence intervals are given)[3]. In

[3] Note that in the case with QoS Control, the throughput is only shown for sessions that were accepted by the resource manager.

the case without QoS Control, the expected throughput of session 3 and sessions 11-15 is not reached because the Grid servers are overloaded. When QoS Control is enabled, all accepted sessions achieve their target throughput. Note that the expected throughput is interpreted as an average value over a longer period of time and therefore, due to the limited session length, the measured throughput is sometimes slightly higher or lower than the expected average.

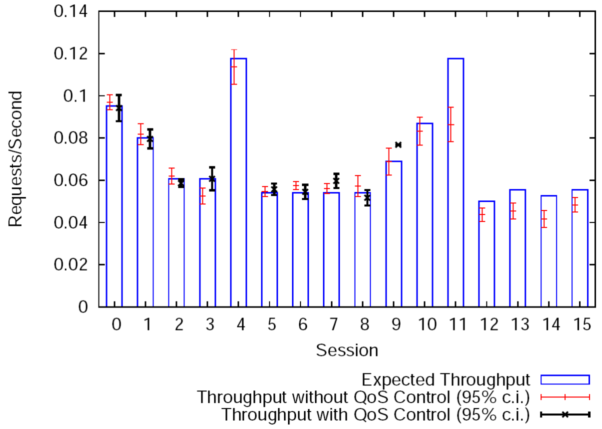

**Figure 7: Throughput obtained with QoS Control vs. without QoS Control.**

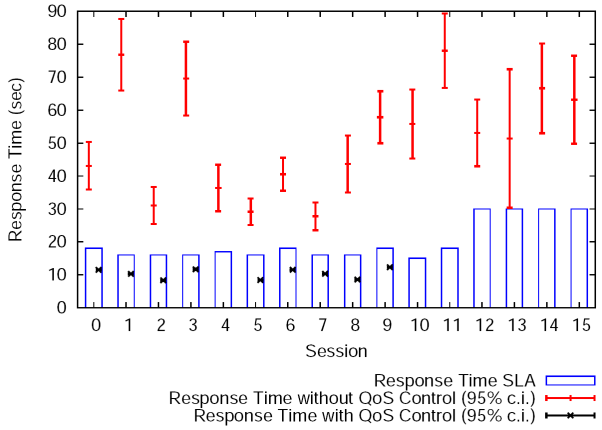

**Figure 8: Response time obtained with QoS Control vs. without QoS Control.**

Figure 8 shows the measured average response times of sessions with and without QoS Control (95% confidence intervals are given). 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.

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 deciding whether to accept or reject a new session request. A 95% confidence interval for the time required to reach a decision was estimated to be $10.82 \pm 0.14$ seconds in the above experiment. 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.

The above 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. Figure 9 shows the response time results from a longer experiment in which 99 sessions were executed over a period of 2 hours. The average session duration was 18 minutes in which 92 service requests were sent on average. In this experiment, when running without QoS control, the system was configured to automatically reject session requests during periods in which both Grid servers were completely saturated. 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.

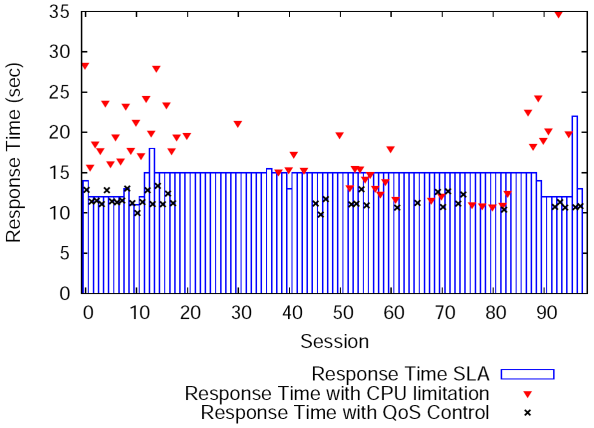

**Figure 9: Response time obtained with QoS Control vs. without QoS Control.**

We are currently conducting an in-depth evaluation of the performance of our framework as the size and complexity of the modeled Grid servers and their workload increase. In our preliminary experiments the overhead of the QoS Predictor was measured to be less than 60 sec for a scenario with 40 Grid servers and 80 sessions. Given that QoS Control is done only in the beginning of a session, we believe this is acceptable for a large class of applications. We will report the detailed results of our evaluation in a follow up paper.

## 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.

In [22] 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 [7] 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 [23], 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 [20] 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. Finally, 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 [29]. 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 [30] 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 [27]. 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 [16] 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 [28]. The focus of this work is on long-term (long-running) applications. In [26] 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.

Further related work in the area of resource management and QoS in Grid computing and SOA environments can be found in [17], [3], [2], [4], [31], [25], [9].

## 6. CONCLUSIONS AND FUTURE WORK

This paper presented a novel methodology for designing QoS-aware Grid 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 continuously met. 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. Moreover, by completely decoupling the Grid clients from the Grid servers, session resources can be reallocated on the fly to reflect changes in the system environment and workload. We exploited queueing Petri nets to accurately model the resource allocation and load balancing mechanism which combines hardware and software aspects of system behavior. A major advantage of our approach is that, being based on queueing Petri nets, it provides great flexibility in choosing the level of detail and accuracy at which system components are modeled. To the best of our knowledge, this is the first application of queueing Petri nets as online performance models for autonomic QoS control. Although the methodology we propose 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.

A prototype of the proposed resource management framework was implemented in C++ and subjected to an extensive experimental evaluation in the context of a real-world Grid environment based on the Globus Toolkit, the world's leading open-source framework for building Grid infrastructures. The results demonstrated the effectiveness of our approach and its applicability to QoS-aware resource management in Grid environments.

The area considered in this paper has many different aspects that will be subject of future work. We plan to extend our framework along several dimensions. First, we intend to evaluate the overhead of the QoS Predictor as the size and complexity of the modeled Grid servers and their workload increase. We envision a number of different ways in which the resource allocation algorithm can be optimized and plan to evaluate different approaches experimentally. While, currently only soft QoS requirements (average values) are guaranteed, we intend to enhance the architecture to support hard QoS requirements (e.g. guaranteeing 90% percentiles of performance metrics). Another aspect we intend to investigate is how our framework can be extended to take into account the costs associated with using the Grid resources when negotiating QoS targets.

## 7. ACKNOWLEDGEMENTS

This work was supported by the Spanish Ministry of Science and Technology, the European Union under contract TIN2004-07739-C02-01, and the German Research Foundation under grant KO 3445/1-1. We acknowledge the support of our colleague Ferran Julià from the Technical University of Catalonia in resolving many technical issues.

## 8. REFERENCES

[1] Enterprise Grid Alliance. www.gridalliance.org.

[2] C. Adam and R. Stadler. A Middleware Design for Large-scale Clusters Offering Multiple Services. *IEEE electronic Transactions on Network and Service Management*, 3(1), 2006.

[3] R. Al-Ali, K. Amin, G. von Laszewski, O. Rana, D. Walker, M. Hategan, and N. Zaluzec. Analysis and Provision of QoS for Distributed Grid Applications. *Journal of Grid Computing*, 2(2), 2004.

[4] R. Al-Ali, O. Rana, G. von Laszewski, A. Hafid, K. Amin, and D. Walker. A Model for Quality-of-Service Provision in Service Oriented Architectures. *Journal of Grid and Utility Computing*, 2005.

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

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

[7] M. N. Bennani and D. A. Menascé. Resource Allocation for Autonomic Data Centers using Analytic Performance Models. In *Proc. of the 2nd Intl. Conference on Automatic Computing*, 2005.

[8] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0. Technical report, W3C, Mar. 2006. http://www.w3.org/TR/wsdl20.

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

[10] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, second edition, Nov. 2003. ISBN: 1558609334.

[11] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *Proc. of the Intl. Workshop on Quality of Service*, 1999.

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

[13] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Intl. J. High Perform. Comput. Appl.*, 15(3), 2001.

[14] I. Foster, A. Roy, and V. Sander. A quality of service architecture that combines resource reservation and application adaptation. In *Proc. of the 8th Intl. Workshop on Quality of Service*, pages 181–188, 2000.

[15] I. T. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *Proc. of the 2005 IFIP Intl. Conference on Network and Parallel Computing*, pages 2–13, 2005.

[16] X. He, X. Sun, and G. Laszewski. A QoS Guided Scheduling Algorithm for Grid Computing. In *Proc. of the Int'l Workshop on Grid and Cooperative Computing*, 2002.

[17] N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive Application-Performance Modeling in a Computational Grid Environment. In *8th IEEE Intl. Symposium on High Perform. Distr. Comput.*, 1999.

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

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

[20] D. Menascé, M. Bennani, and H. Ruan. *Self-Star Properties in Complex Information Systems*, volume 3460 of *LNCS*, chapter On the Use of Online Analytic Performance Models in Self-Managing and Self-Organizing Computer Systems. Springer, 2005.

[21] D. A. Menascé, V. A. Almeida, and L. W. Dowdy. *Performance by Design*. Prentice Hall, 2004.

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

[23] D. A. Menascé, H. Ruan, and H. Gomaa. A Framework for QoS-Aware Software Components. In *Proc. of the 4th Intl. Workshop on Software and Performance*, 2004.

[24] R. Nou, F. Julià, and J. Torres. Should the grid middleware look to self-managing capabilities? In *Proc. of the 8th Intl. Symposium on Autonomous Decentralized Systems*, 2007.

[25] A. Othman, P. Dew, K. Djemamem, and I. Gourlay. Adaptive Grid Resource Brokering. In *Proc. of the 2003 IEEE Intl. Conference on Cluster Computing*, pages 172–179, 2003.

[26] G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance Management of Cluster-Based Web Services. *IEEE Journal on Selected Areas in Communications*, 23(12):2333–2343, Dec. 2005.

[27] P. Shivam, S. Babu, and J. Chase. Learning Application Models for Utility Resource Planning. In *Proc. of the 3rd Intl. Conference on Autonomic Computing*, 2006.

[28] X.-H. Sun and M. Wu. Grid Harvest Service: A System for Long-Term, Application-Level Task Scheduling. In *Proc. of the 17th Intl. Symposium on Parallel and Distributed Processing*, 2003.

[29] G. Tesauro, R. Das, W. Walsh, and J. Kephart. Utility-Function-Driven Resource Allocation in Autonomic Systems. In *Proc. of the 2nd Intl. Conference on Autonomic Computing*, 2005.

[30] G. Tesauro, N. Jong, R. Das, and M. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *Proc. of the 3rd Intl. Conference on Autonomic Computing*, 2006.

[31] D. Xu, K. Nahrstedt, A. Viswanathan, and D. Wichadakul. QoS and Contention-Aware Multi-Resource Reservation. In *Proc. of 9th IEEE Intl. Symposium on High Perform. Distr. Comput.*, 2000.