RC22676 (W0208-034) August 12, 2002
Computer Science

IBM Research Report

Performance Management for Web Services

Giovanni Pacifici, Mike Spreitzer, Asser Tantawi, Alaa Youssef

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Performance Management for Cluster Based Web Services

Giovanni Pacifici, Mike Spreitzer, Asser Tantawi, and Alaa Youssef
IBM TJ Watson Research Center
{giovanni,mspreitz,tantawi,ayoussef}@us.ibm.com

May 13, 2003

Abstract

We present an architecture and prototype implementation of a performance management system for cluster-based web services. The system supports multiple classes of web services traffic and allocates server resources dynamically so to maximize the expected value of a given cluster utility function in the face of fluctuating loads. The cluster utility is a function of the performance delivered to the various classes, and this leads to differentiated service. In this paper we will use the average response time as the performance metric. The management system is transparent: it requires no changes in the client code, the server code, or the network interface between them. The system performs three performance management tasks: resource allocation, load balancing, and server overload protection. We use two nested levels of management mechanism. The inner level centers on queuing and scheduling of request messages. The outer level is a feedback control loop that periodically adjusts the scheduling weights and server allocations of the inner level. The feedback controller is based on an approximate first-principles model of the system, with parameters derived from continuous monitoring. We focus on SOAP-based web services. We report experimental results that show the dynamic behavior of the system.

1 Introduction

Today we are seeing the emergence of a powerful distributed computing paradigm, broadly called web services [1]. Web services feature ubiquitous protocols, language-independence, and standardized messaging. Due to these technical advances and growing industrial support, many believe that web services will play a key role in dynamic e-business [2]. In such an environment, a web service provider may provide multiple web services, each in multiple grades, and each of those to multiple customers. The provider will thus have multiple classes of web service traffic, each with its own characteristics and requirements. Performance management becomes a key problem, particularly when service level agreements (SLA) are in place. Such service level agreements are included in service contracts between providers and customers and they specify both performance targets, known as performance objectives, and financial consequences for meeting or failing to meet those targets. A service level agreement may also depend on the level of load presented by the customer.

In this paper we present an architecture, and describe a prototype implementation, of a performance management system for web services that supports service level agreements. We have designed and implemented reactive control mechanisms to handle dynamic fluctuations in service demand while keeping service level agreements in mind. Our mechanisms dynamically allocate resources among the classes of traffic, balance the load across the servers, and protect the servers against overload - all in a way that maximizes a given cluster utility function. This produces differentiated service.

We introduce a *cluster utility function* that is a composition of two kinds of functions, both given by the service provider. First, for each traffic class, there is a *class-specific utility function* of performance. Second, there is a *combining function* that combines the class utility values into one cluster utility value. This parameterization by two kinds of utility function gives the service provider flexible control over the trade-offs made in the course of service differentiation. In general, a service provider is interested in profit (which includes cost as well as revenue) as well as other considerations (e.g., reputation, customer satisfaction).

We have organized our architecture in two levels: (i) a collection of in-line mechanisms that act on each connection and each request, and (ii) a feedback controller that tunes the parameters of the in-line mechanisms. The in-line mechanisms consist of connection load balancing, request queuing, request scheduling, and request load balancing. The feedback controller periodically sets the operating parameters of the in-line mechanisms so as to maximize the cluster utility function. The feedback controller uses a performance model of the cluster to solve an optimization problem. The feedback controller continuously adjusts the model parameters using measurements of actual operations. In this paper we report the results obtained using an approximate, first-principles model. We focus on SOAP-based web services and use statistical abstracts of SOAP response times as the characterization of performance. We allow ourselves no functional impact on the service customers or service implementation: we have a transparent management technique that does not require changes in the client code, the server code, or the network protocol between them.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the system architecture and prototype implementation. Performance modeling and optimization analysis are described in Section 4. Section 5 illustrates some experimental results, showing both transient responses and service differentiation. Section 6 presents conclusions and discusses future work.

2 Related Work

Several research groups have addressed the issue of QoS support for distributed systems [3]. In this section we summarized the current state of the art. The first class of research study deals with session-based admission control for overload protection of web servers. Chen et al. [4] proposed a dynamic weighted fair sharing scheduler to control overloads in web servers. The weights are dynamically adjusted, partially based on session transition probabilities from one stage to another, in order to avoid processing requests that belong to sessions likely to be aborted in the future. Similarly, Carlström et al. [5] proposed using generalized processor sharing for scheduling requests, which are classified into multiple session stages with transition probabilities, as opposed to regarding entire sessions as belonging to different classes of service, governed by their respective service level agreements.

Another area of research deals with performance control of web servers using classical feedback control theory. Abdelzaher et al. [6] used classical feedback control to limit utilization of a bottleneck resource in the presence of load unpredictability. They relied on scheduling in the service implementation to leverage the utilization limitation to meet differentiated response-time goals. They used simple priority-based schemes to control how service is degraded in overload and improved in under-load. In this paper we use a new technique that gives the service provider a finer grain control on how the control subsystem should tradeoff resource among different web services requests. Diao et al. [7] used feedback control based on a black-box model to maintain desired levels of memory and CPU utilization. In this paper we use a first-principles model and maximize a cluster objective function.

Web server overload control and service differentiation using OS kernel-level mechanisms, such as TCP SYN policing, has been studied in [8]. A common tendency across these approaches is tackling the problem at lower protocol layers, such as HTTP or TCP, and the need to modify the web server or the OS kernel in order to incorporate the control mechanisms. Our solution on the other hand operates at the SOAP protocol layer, which does not require changes to the server, and allows for finer granularity of content-based request classification.

Service differentiation in cluster-based network servers has also been studied in [9] and [10]. The approach taken here is to physically partition the server farm into clusters, each serving one of the traffic classes. This approach is limited in its ability to accommodate a large number of service classes, relative to the number of servers. Lack of responsiveness due to the nature of the server transfer operation from one cluster to another is typical in such systems. On the other hand, our approach uses statistical multiplexing, which makes fine-grained resource partitioning possible, and unused resource capacities can be instantaneously shared with other traffic classes.

Chase et al. [11] refine the above approach. They note that there are techniques (e.g., cluster reserves [12], and resource containers [13]) that can effectively partition server resources and quickly adjust the proportions. Like our work, Chase et al. also solve a cluster-wide optimization problem. They add terms for the cost (due, e.g., to power consumption) of utilizing a server, and use a more fragile solution technique. Also, they use a black-box model rather than first-principles one.

Zhao and Karamcheti [14] propose a distributed set of queuing intermediaries with non-classical feedback control that maximizes a global objective. Their technique does not decouple the global optimization cycle from the scheduling cycle.

In this paper we use the concept of utility function to encapsulate the business importance of meeting or failing to meet performance targets for each class of service. The notion of using a utility function and maximizing a sum [15] or a minimum [16] of utility functions for various classes of service has been used to support service level agreements in communication services. In such analyses, the utility function is defined in terms of bandwidth allocated (i.e. resources). In our work, we define a class utility function to express the business value of meeting the service level objective as well as deviating from it. Further, the effect of the amount of allocated resources on performance level is separated from the business value objectives.

3 Performance Management System Architecture and Implementation

In this section we present the system architecture and prototype implementation of a management system for web services. This system allows service providers to offer and manage service level agreements for web services. The service provider may offer each web service in different *grades*, with each grade defining a specific set of *performance objective parameters*. For example, the `StockUtility` service could be offered in either *premium* or *basic* grade, with each grade differentiated by performance objective and base price. A prototypical grade will say that the service customers will pay \$10 for each month in which they request less than $100,000$ transactions and the 95^{th} percentile of the response times is smaller than $5sec$, and \$5 for each month of slower service.

Using a configuration tool the service provider will define the number and parameters of each grade. Using a subscription interface users can register with the system and subscribe to services. At subscription time each user will select a specific offering and associated grade.

The service provider uses the configuration tool to also create a set of *traffic classes* and map a `<customer, service, operation, grade>` tuple into a specific traffic class (or simply class). The service provider assigns a specific response time target to each traffic class. Our management system allocates resources to traffic classes and assumes that each traffic class has a homogeneous service execution time.

We introduce the concept of class to separate operations with widely differing execution time characteristics. For example the `StockUtility` service may support the operations `getQuote()` and `buyShares()`. The fastest execution time for `getQuote()` could be $10ms$ while the `buyShares()` cannot execute faster that $1sec$. In such a case the service provider would map these operations into different classes with different set of response time goals. We also use the concept of class to isolate specific contracts to handle the requests from those customers in a specific way.

Figure 1 shows the system architecture. The main components are: a set of *gateways*, a *global resource manager* a management *console* and a set of *server nodes* on which we deploy the target web services. We use gateways to execute the logic that controls the request flow and we use the server nodes to execute the web services logic. Gateway and server nodes are software components. We usually have only one gateway per physical machine and in general we have server nodes and gateways on separate machines. The simplest configuration is one gateway and one server node running on the same physical machine.

In this paper we assume that all server nodes are homogeneous and that every web service is deployed on each server. We can deal with heterogeneous servers by partitioning them into disjoint pools, where all the servers in a given pool have the same subset of web services deployed, and where the traffic classes are also partitioned among the pools.

The servers, gateways, global resource manager, and console share monitoring and control information via a publish/subscribe network[17]. In coping with higher loads, the system scales by having multiple gateways. An L4 switch distributes the incoming load across the gateways.

3.1 Gateway

We use gateways to controls the amount of server resources allocated to each traffic class. By dynamically changing the amount of resources we can control the response time experienced by each traffic class.

We denote with $N_{g,s}$ the maximum number of concurrent requests that server s executes on the behalf of gateway g. We also use $w_{g,c}$ to describe the minimum number of class c requests that all servers will execute

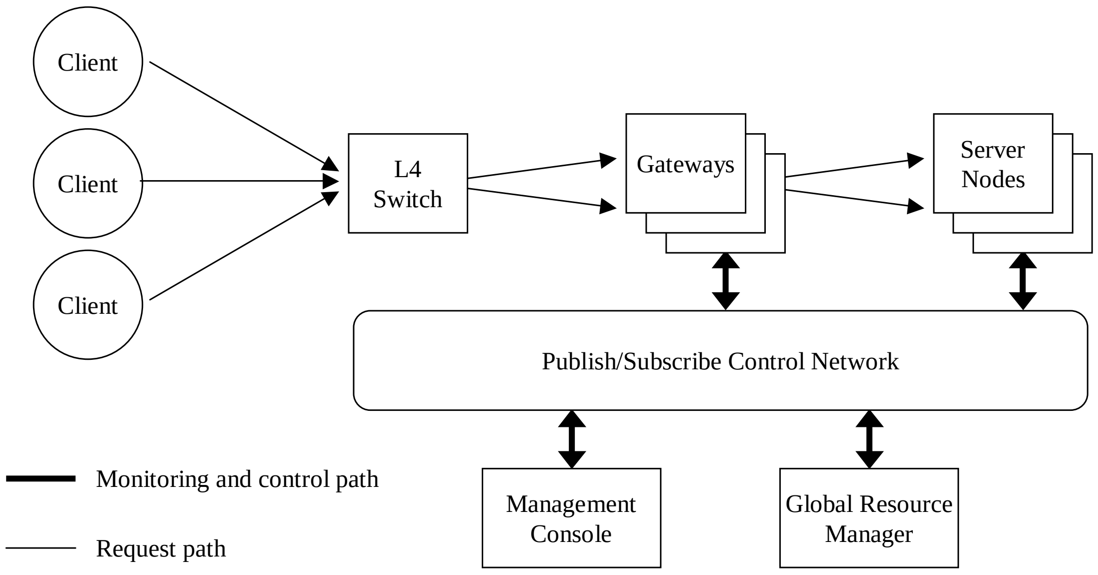


Figure 1: System overview

on the behalf of gateway g. We refer to $w_{g,c}$ as *server shares*. In Section 4 we will describe how we compute $w_{g,c}$ and $N_{g,s}$, while, in this section we describe how gateway g enforces the $w_{g,c}$ and $N_{g,s}$ constraints. For each gateway g, we use w_g and N_g to denote the following:

$$w_s = \sum_{c \in C} w_{g,c}, \qquad\qquad N_g = \sum_{s \in S} N_{g,s} \tag{1}$$

where C and S denote the set of all classes and servers respectively. Figure 2 illustrates the gateway components. We have used Axis [18] to implement all our gateway components and we have implemented some of the mechanisms using Axis handlers, which are generic interceptors in the stream of message processing. Axis handlers can modify the message, and can communicate out-of-band with each other via an Axis message context associated with each SOAP invocation (request and response) [18].

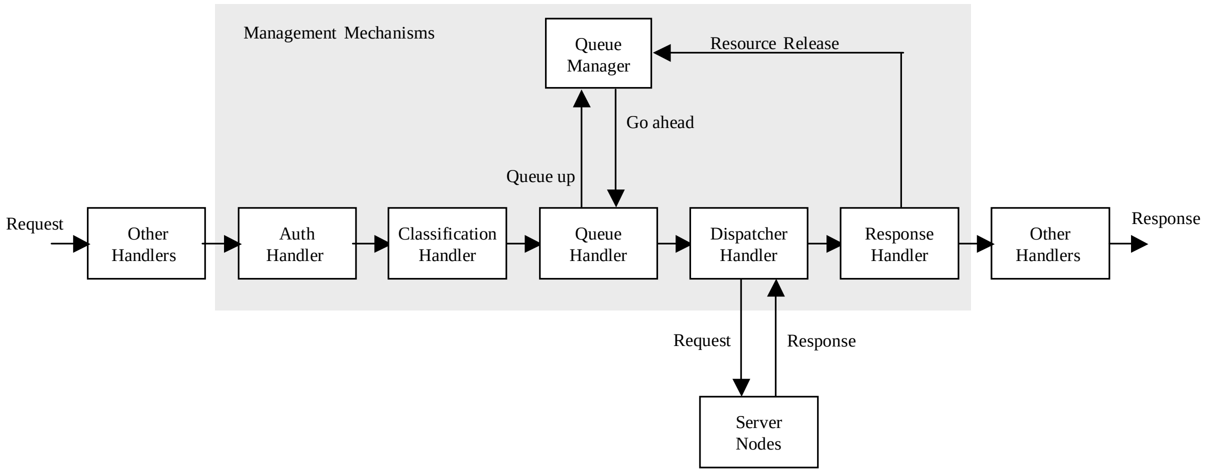


Figure 2: Gateway components

When a new request arrives a *classification handler* determines the traffic class of the request. The mapping functions use the request meta-data (user id, subscriber id, service name, etc.). In our implementation the classification handler uses the user and SOAP action fields in the HTTP headers as inputs, and reads the mappings from configuration files. We avoid parsing the incoming SOAP request to minimize the overhead.

After we classify the requests, we invoke the *queue handler*, which in turn contacts a *queue manager*. The queue manager implements a set of logical FIFO queues one for each class. When the queue handler invokes the queue manager the queue manager suspends the request and adds the request to the logical queue corresponding to the request's class.

The queue manager includes a *scheduler* that runs when a specific set of events occurs and selects the next request to execute. The queue manager on gateway g tracks the number of outstanding requests dispatched to each server makes sure that there are at most N_g requests concurrently executing on all the servers. When the number of concurrently outstanding requests from gateway g is smaller than N_g the scheduler selects a new requests for execution.

The scheduler uses a round robin scheme. The total length of the round robin cycle is w_g and the length of class c interval is $w_{g,c}$. We use a dynamic boundary and work conserving discipline that always selects a non-empty queue if there is at least one. The above discipline guarantees that during periods of resource contention the server nodes will concurrently execute at least $w_{g,c}$ requests of class c on the behalf of gateway g.

After the scheduler selects a request the queue manager resumes the execution of the request's corresponding queue handler. The queue manager collects statistics on arrival rates, execution rates, and queueing time and periodically broadcasts these data on the control network.

The *dispatch handler* selects a server and sends the request to the server, using a protocol defined by configuration parameter. Our implementation supports SOAP over HTTP and SOAP over JMS. The dispatch handler distributes the requests among the available servers using a simple load balancing discipline while enforcing the constrains that at most $N_{g,s}$ requests executes on server s concurrently on the behalf of gateway g.

When a request completes its execution the *response handler* reports to the queue manager the completion of the request's processing. The queue manager uses this information to both keep an accurate count of the number of requests currently executing and to measure performance data such as service time.

The gateway functions may be run on dedicated machines, or on each server machine. The second approach has the advantage that it does not require a sizing function to determine how many gateways are needed, and the disadvantage that the server machines are subjected to load beyond that explicitly managed by the gateways.

3.2 Global Resource Manager and Management Console

The *global resource manager* computes $N_{g,s}$, the maximum number of concurrent requests that each server s executes on the behalf of each gateway g, and it computes $w_{g,c}$, the minimum number of class c requests that all servers will execute on the behalf of each gateway g.

The global resource manager runs periodically and computes the resource allocation parameters every time interval Γ_i, which we define as the i^{th} control horizon. The global resource manager computes $N_{g,s}$ and $w_{g,c}$ that each gateway will use during the control horizon Γ_i using the resource allocation parameters computed in the control horizon Γ_{i-1} as well request and server utilization statistics measured in during Γ_{i-1}.

The size of the control horizon affects the ability of the global resource manager to respond to rapid changes in the traffic load or response time. On the one hand, when Γ is small, the resource allocation parameters are updated frequently which make the system more adaptive. On the other hand, a larger value of Γ increases the stability of the system.

Figure 3 shows the global resource manager inputs and outputs. In addition to real-time dynamic measurements, the global resource manager uses resource configuration information and the *cluster utility function*. The cluster utility function consists of as a set of *class utility functions* and a *combining function*. Each class utility function maps the performance for a particular traffic class into a scalar value that encapsulates the business importance of meeting, failing to meet or exceeding the class service level objective. A combining function combines the class utility function into one cluster utility function. In this paper we have implemented the combining functions as a sum of the utility functions, however, our work could be extended to study the impact of other combining function on the structure of the solution.

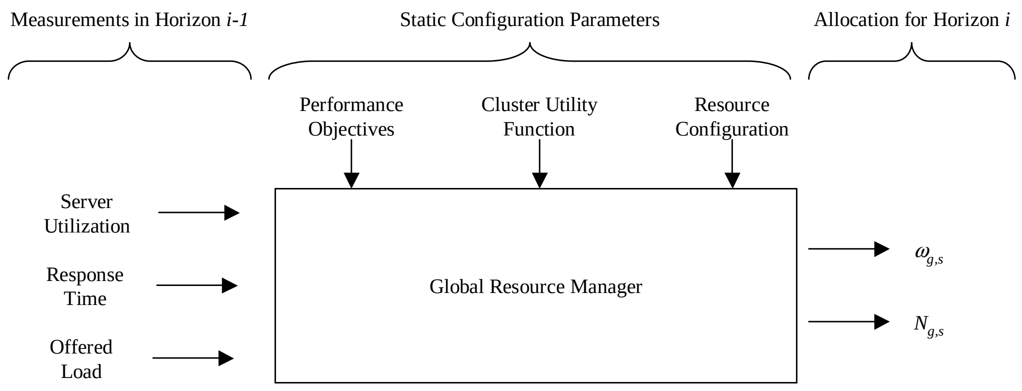


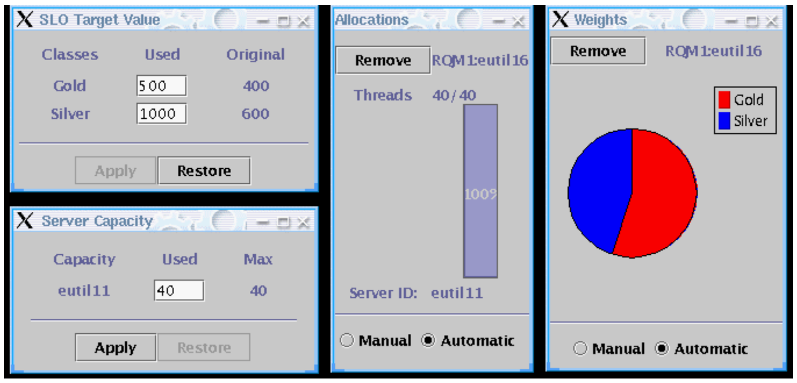
Figure 3: Global resource manager inputs and outputs

As shown in Figure 3 the global resource manager may assume the responsibility of computing the capacity N_s of each server s. N_s represents the maximum number of web services requests that server s can execute concurrently. The global resource manager should select N_s to be large enough to efficiently utilize the server's physical resources, but small enough to prevent overload and performance degradation. The global resource manager may use server utilization data to determine the value of N_s.

The global resource manager partitions N_s among all gateways and classes. The global resource manager use $w_{g,c}$ to describe the minimum number of class c requests that all servers will execute on the behalf of gateway g. The global resource manager uses a queuing model of the system to predict the performance that each class would experience for each given allocation $w_{g,c}$. The global resource manager implements a dynamic programming algorithm to find the $w_{g,c}$ that maximize the cluster utility function. After we compute $w_{g,c}$ we compute $N_{g,s}$ by partitioning N_s among all gateways. We describe the details on the model and the resource allocation algorithm in in Section 4.

After the global resource manager computes a new set of $w_{g,c}$ and $N_{g,s}$ values, it broadcasts them on the control network. Upon receiving the new resource allocation parameters each gateway switch to the new values of $w_{g,c}$ and $N_{g,s}$. We discuss the algorithm used to predict the class performance and maximize the cluster utility function in Section 4.

The *management console* offers a graphical user interface to the management system. Through this interface the service provider can view and override all the configuration parameters. We also use the console to display the measurements and internal statistics published on the control network. Finally we can use the console to manually override the control values computed by the global resource manager. Figure 4 shows a subset of views available from our management console.


SLO Target Value

Classes	Used	Original
Gold	500	400
Silver	1000	600

Server Capacity

Capacity	Used	Max
eutil11	40	40

Allocations

RQM1:eutil16

Threads 40/40

Server ID: eutil11

Manual [✓ Automatic]

Weights

RQM1:eutil16

🟥 Gold
🟦 Silver

Manual [✓ Automatic]

Figure 4: Management console: configuration and control values

4 Modeling and Optimization

In this section we describe how the global resource manager computes the resource allocation. First we give an abstract definition of the problem solved. Then we discuss the simplified queuing model used to predict the performance of each class for a given resource allocation. We also examine the class utility functions detail.

4.1 The Resource Allocation Problem

The global resource manager computes the $N_{g,s}$ and $w_{g,c}$ values to maximize the *cluster utility function* over the next control period. We decouple the $N_{g,s}$ and $w_{g,c}$ problems by solving for the $w_{g,c}$ first, and then deriving the $N_{g,s}$ from them.

To determine the $w_{g,c}$, we use dynamic programming to find the $w_{g,c}$ that maximizes the cluster utility function Ω which we defined as a combination of each class utility function U_c. In our work we have studied two different kind of combining functions. In particular we find the set of values of $w_{g,c}$ that:

$$\max_{w_{g,c}} \quad \Omega = \begin{cases} \displaystyle\sum_{c \in C} \sum_{g \in G} U_c(w_{g,c}) & (a) \\ \\ \displaystyle\min_{c \in C} \left(\sum_{g \in G} U_c(w_{g,c}) \right) & (b) \end{cases} \tag{2}$$

subject to

$$w_{g,c} \geq 1, \qquad \sum_{g \in G} \sum_{c \in C} w_{g,c} = N \tag{3}$$

where

$$N = \sum_{s \in S} N_s \tag{4}$$

and C, G and S denote the set of classes, gateways and servers respectively. The utility function $U_c(w_{g,c})$ defines the utility associated with allowing $w_{g,c}$ requests of class c traveling through gateway g to concurrently execute on any of the servers. When we use the objective function in (2a) we compute the cluster utility as the sum of each class utility function, thus we maximize the overall system utility. When we use (2b) to compute the cluster utility, the resource manager will find the allocation vector that maximizes the smallest utility function, which means it looks for a solution that equalizes the utility of all classes. In Section 4.2 we discuss the structure of the utility function and in Section 4.3 we show how we compute U_c as a function of $w_{g,c}$.

As we mentioned in the previous section, we enforce for each server s, a limit N_s on the maximum number of requests that may be concurrently active on that server. Once we have computed $w_{g,c}$. the value w_g derived from (1) represents the portion of server resources that have been allocated to gateway g. To computes $N_{g,s}$ for each gateway g we divide each server s available concurrency N_s among the gateways in proportion to w_g. In particular for each server s we select the point $[N_{1,s}, N_{2,s}, \ldots, N_{G,s}]$ (G being the number of gateways) with integer-valued coordinates constrained by

$$\sum_{g \in G} N_{g,s} = N_s \tag{5}$$

and near the point $\left[\hat{N}_{1,s}, \hat{N}_{2,s}, \ldots, \hat{N}_{G,s}\right]$ defined by

$$\hat{N}_{g,s} = \frac{w_g}{N} N_s \tag{6}$$

where N is the total number of resources across all servers as defined in (4).

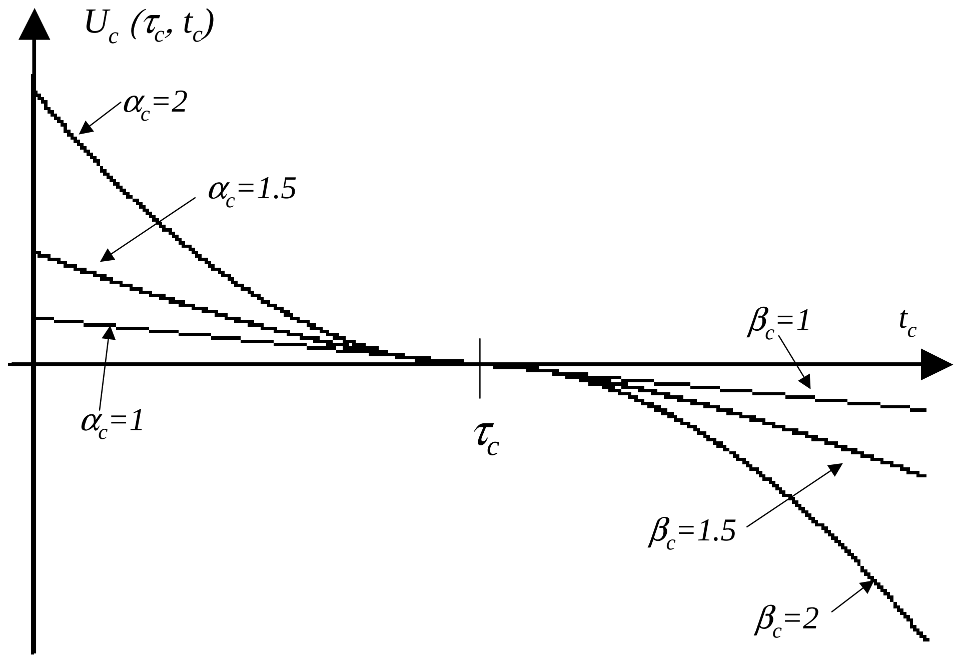


Figure 5: Utility function for class c

4.2 The Structure of Class Utility Functions

We use the a utility function U_c to encapsulate the business importance of meeting or failing to meet class c performance. The utility function maps the performance actually experienced by web services requests into a real number U_c. Since in (2) we use a combination of utility functions to construct the cluster objective function, by changing the size and shape of the utility function we can influence the amount of resources that we will allocate to each class. There is no single way to construct a utility function. In this paper we study a family of functions and we use experiments to determine the impact of different choices of utility function. When selecting the utility functions we have used the following guidelines:

- the value of U_c should be larger when the performance experienced by c requests is better than the target and smaller when the performance is worse;
- the value of U_c should increase as the performance experienced by c increases and decrease otherwise;
- the size and shape of the utility function should be controlled by one or two parameters that can be adjusted by the platform provider to reflect the importance of one class of traffic over another;

In this paper, we express each class performance objective as an *upper bound on the average response time* and therefore U_c will depend on the negotiated upper bound as well as the actual response time. We denote with t_c the average response time experienced by class c requests and with τ_c the negotiated upper bound on the average response time. We then use the following family of functions to describe class c utility:

$$U_c(\tau_c, t_c) = \begin{cases} \phi_c\,(\tau_c - t_c)^{\alpha_c} & \text{if } t_c \le \tau_c \\ \phi_c\,(t_c - \tau_c)^{\beta_c} & \text{if } t_c > \tau_c \end{cases} \tag{7}$$

The function in (7) and shown in Figure 5 compares average response time t_c to target response time τ_c for class c. When $t_c \le \tau_c$ the utility grows as the response time distance from the target to the power of α_c. When $t_c > \tau_c$ the utility decays as the response time distance from the target to the power of β_c. We also use ϕ_c as a scaling factor.

For the plot in Figure 5 we have used $\tau_c = 6$, $\phi_c = 1$, $\alpha_c = [1, 1.5, 2]$ and $\beta_c = [1, 1.5, 2]$. By increasing α_c we control the business importance of exceeding the target for class c, while by increasing β_c we can

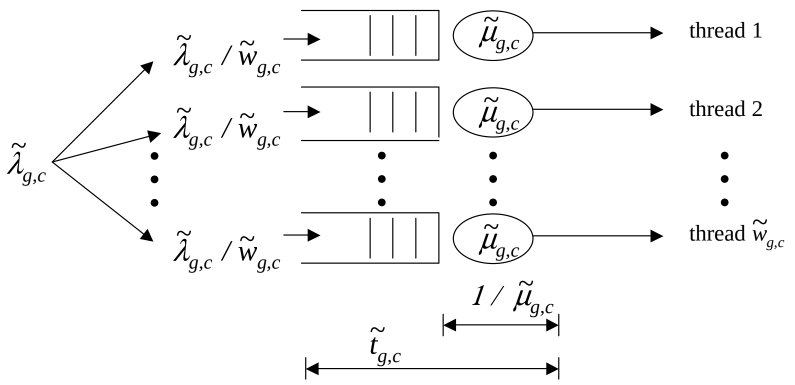


Figure 6: Modeling the response time behavior for class c requests handled by gateway g

control how fast the business utility degrades when class c experience a delay bigger than the objective. By chancing ϕ_c, α_c and β_c we can influence how resource are allocated to each class of traffic and in turn the class performance. In the next section we describe how we estimate the expected response time t_c for class c given a resource allocation $w_{g,c}$. where α_c is a factor that we use to weight utility functions. In the next section we describe how we estimate the expected response time t_c for class c given a scheduling weight of $w_{g,c}$.

4.3 System Modeling

To compute the class c utility U_c given an allocation of $w_{g,c}$ resources, we need to predict $t_{g,c}$, i.e., the average response time of class c requests handled by gateway g given a proposed allocation $w_{g,c}$ resources. To predict $t_{g,c}$ we use the observed arrival rate, response time, and the allocation values, from the previous control cycle denoted by $\tilde{\lambda}_{g,c}$, $\tilde{t}_{g,c}$, and $\tilde{w}_{g,c}$.

We use an M/M/1 queue to model the response time behavior of requests of class c traveling through gateway g, i.e., we assume that $\tilde{\lambda}_{g,c}$ was evenly divided among the server threads that have been concurrently executing all requests of class c traveling through gateway g during the previous control cycle. Using this assumption we compute the equivalent service rate of the M/M/1 queue that has been handling the fraction of requests served by one of the $w_{g,c}$ threads. The equivalent service rate is given by

$$\mu_{g,c} = \frac{1}{\tilde{t}_{g,c}} + \frac{\tilde{\lambda}_{g,c}}{\tilde{w}_{g,c}} \tag{8}$$

Figure 6 exemplifies the above modeling technique. We now use $\tilde{\mu}_{g,c}$ to predict the response time of all class c requests traveling through gateway g in the next control cycle under an allocation of $w_{g,c}$ threads, as follows

$$t_{g,c}\left(w_{g,c}\right) = \frac{1}{\frac{1}{\tilde{t}_{g,c}} + \tilde{\lambda}_{g,c}\left(\frac{1}{\tilde{w}_{g,c}} - \frac{1}{w_{g,c}}\right)} \tag{9}$$

In the previous calculation we have assumed that the request load in the new cycle is equal to the previous one.

Using (7) and (9) we can compute the utility function $U_c(\tau_c, t_c)$ as a function of the expected allocation $w_{g,c}$. Using dynamic programming we can then compute the set of $w_{g,c}$ that will maximize the cluster utility function Ω in (2) under the constraints in (3).

The resource allocation methodology described in this section will achieve an optimal resource allocation only under the assumptions mentioned above. For all other cases our methodology achieves a sub-optimal solution. Given the nature of our system an optimal allocation can be determined only by simulation and extensive search. More work is required to determine the difference between our approach and an optimal allocation of resources. In the next section we report the results of several experiments intended to study the effectiveness of this approach.

5 Experimental Results

In this section we describe a set of experiments that we have conducted to study the behavior of our system under different traffic load conditions.

We used two Intel-based machines for our experiments. We used the first machine to run a Web Services load generator. For the load generator we used a Java-based application that can simulates large numbers of Web Services clients each generating requests according to a defined stochastic models. We used the second Intel machine to run both the gateway and the Web Services server. We used Axis [18] running on Tomcat [19] to implement the server and gateway containers.

For the experiments described in this paper we used two different classes of clients, referred to as *Premium* and *Basic*. Both classes of clients generate requests using a closed-loop model. In such a model, a number of clients of each class generate requests independently. Each client generates one request and waits until the server responds. Then, the client goes to sleep, modeling the think time of an application or user. The sleep times are i.i.d. random variables with negative exponential distribution with a mean of $1sec$. After waking up, the client generates a new request. In our experiments we varied the number of clients of each class in the range of 5 to 20 clients as shown in Figure 7.

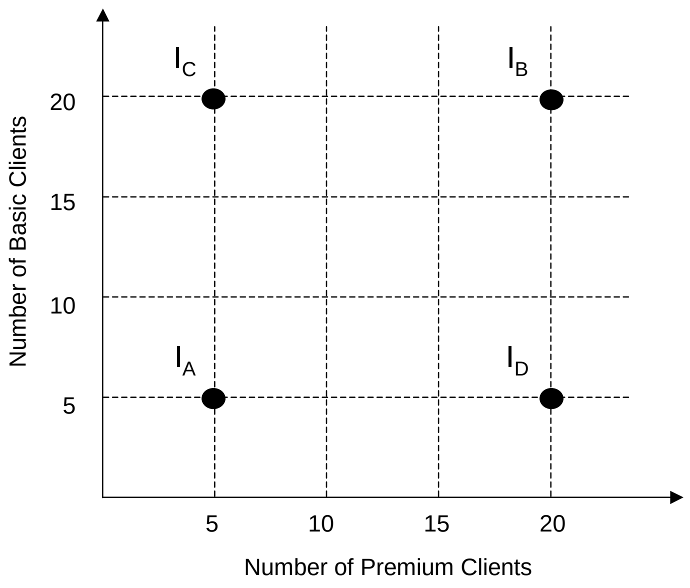


Figure 7: Traffic load combinations used in our experiments

On the server we deployed a synthetic Web Service. We chose a synthetic service to better control our experiments. We implemented the synthetic service using a Java class that alternates between CPU-bound processing and sleeping. We used the sleeping intervals to emulate periods in which a process waits response from a back-end server or database. The service times are i.i.d. random variables with negative exponential distribution and average of $1sec$.

5.1 Effect of Degree of Concurrency on Throughput

We used the first set of experiments to determine the optimal degree of concurrency for our set up.

In order to determine the optimal value of N_s for our server, we measured the system throughput for various settings of N_s. In these experiments, the load consisted of only one traffic class, and we always used a large enough number of clients to make sure that at any given time N_s requests were executing concurrently.

We started with a value of $N_s = 1$ for the first experiment. We run our the experiment for several minutes and we measured the average throughput of the system, i.e., the number of request that complete in a unit of time. We repeated the same experiment several times using larger values of N_s each time. Figure 8 shows the results of our experiments. When N_s is small the CPU is under utilized and the throughput increases by increasing N_s. When $N_s = 10$ the CPU reaches 100% utilization and the throughput remains constant even if we increase N_s further. When we used values of N_s much larger than 10 the throughput decreased because of context switching overheads.

Based on these results we selected the value of $N_s = 10$ as the concurrency setting in all other experiments describes in this section. In general the optimal value of N_s will change dynamically and will depend on the type of services being invoked, their parameters and the service mix.

For the experiments reported in this paper we did not use an automatic mechanism to compute the optimal value of N_s. However such a mechanism is a key component for a production system and will be the subject of future work.

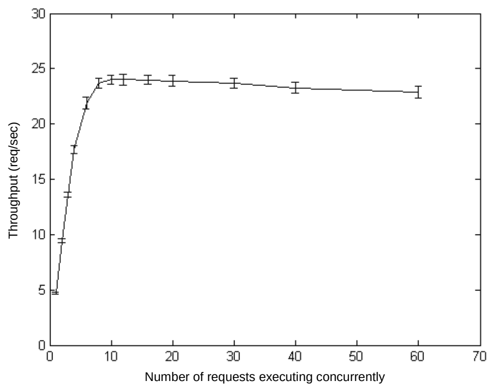


Figure 8: Throughput vs requested maximum number of concurrent executions

5.2 Service Level Differentiation and System Responsiveness

In this section we describe the results of a set of experiments aimed at studying the dynamic behavior of our system and its ability to react to changes in the traffic load and mix. In our experiments we configured our sensors to report traffic load and performance statistics every 5*sec*. We also configured these sensors to average traffic and performance statistics over a period of 30*sec*. We set the length of the global resource manager control cycle to 5*sec*, i.e., the global resource manager recomputes a new set of server shares $w_{g,c}$ every time the sensors publish a new value of traffic load and performance statistics.

We used the utility function in (7) with $\phi_c = \alpha_c = \beta_c = 1$ for both the *Premium* and *Basic* class. For the Premium class we set a target average response time of Premium requests $\tau_P = 2sec$ and we set the average response time for Basic to $\tau_B = 3sec$. We used the cluster utility function in (2)a, thus attempting to equalize the utilities of both classes.

We started from an idle server, and changed the load to the system in four phases, denoted by Φ_A, Φ_B, Φ_C, and Φ_D, respectively. During phase Φ_A, we set the number of clients to 5 for each of the classes, which corresponds to a light load situation. We denote this case as $(L_P, L_B)_{\Phi_A} = (5, 5)$, where L_P denotes the number of premium clients and L_B denotes the number of basic clients. The other three phases are as follows: $(L_P, L_B)_{\Phi_B} = (20, 20)$, $(L_P, L_B)_{\Phi_C} = (5, 20)$, and $(L_P, L_B)_{\Phi_D} = (20, 5)$. We use Φ_B to simulate a heavy load situation and both Φ_C and Φ_D to simulate moderate load conditions, with a different mix of Premium and Basic clients. Our experiment study starts with light load, then moves to heavy load, followed by moderate load with more Basic then more Premium respectively.

During the experiment, the global manager adjusted the values of $w_{g,c}$ for each class to respond to the changes in the traffic load and mix as shown in Figure 9. Since we use a work conserving scheduling discipline, during the light load phase Φ_A, the unused allocated capacity of one traffic is available to other traffic class. Therefore, the response time for both classes during phase Φ_A is not sensitive to the value of the allocation vector. During the heavy load phase Φ_B, the allocation remained at $(w_{g,P}, w_{g,B}) = (7, 3)$ to ensure good response for the Premium class. During phase Φ_C, the allocation changed to $(w_{g,P}, w_{g,B}) = (6, 4)$, giving more capacity to the Premium traffic which is about three times as large as the Premium. During phase

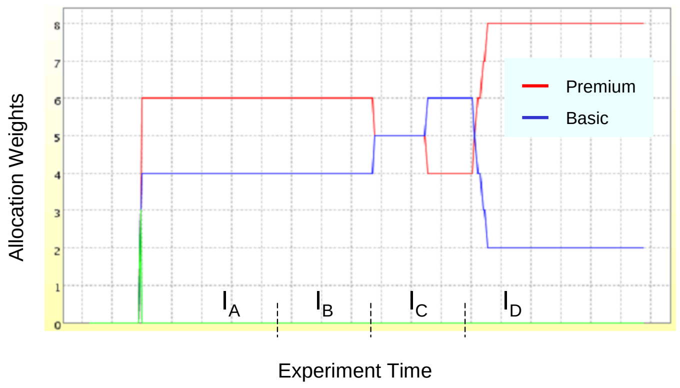


Figure 9: Weights allocated by the GRM

Φ_D, the global manager changed the allocation to $(w_{g,P}, w_{g,B}) = (8, 2)$ because of the higher load from the Premium clients.

Figure 10 show the average response time. In Figure 10 we marked the target values of $\tau_P = 2sec$ and $\tau_B = 3sec$ for the Premium and Basic classes. Due to the light traffic during phase Φ_A, the queueing time is negligible and the response time is simply due the service time which has an average of $1sec$. During the heavy-loaded phase Φ_B, the allocation $(w_{g,P}, w_{g,B}) = (7, 3)$ results in a average response time for the Premium class that is slightly above the target value, whereas the average response time for the Basic clients is about twice as large as the Premium one. Since we use the same utility function for both Premium and Basic traffic and we use the cluster objective function in (2a) the heavy load impacted both traffic classes in a way that is proportional to their target values.

During phase Φ_C, the response time decreased because the load decreased, but we still observe a difference in the response time for the different class of clients. The switch between phase Φ_C and phase Φ_D caused the Premium traffic to initially experience a increase in the response time until the global resource manager detected the new load conditions and corrected by adjusting the allocation vector $(w_{g,P}, w_{g,B})$. Similarly, the Basic clients experience a better response time at the edge of the transition between phases Φ_C and Φ_D. The response time for the Basic clients increases after the global resource manager changes the allocation vector.

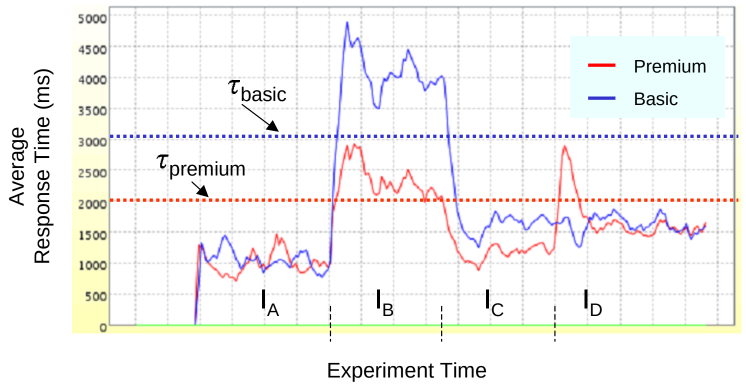


Figure 10: Average response time

Figure 11 shows the average queue length. During phase Φ_A there is no queueing. During the other three phases the number N_s of concurrent Web Services requests executed by the server is almost always equal to the maximum of 10. Therefore we have requests waiting in the queue manager buffers.

In the heavy load phase Φ_B $(L_P, L_B)_{\Phi_B} = (20, 20)$ the allocation vector is $(w_{g,P}, w_{g,B}) = w(6, 4)$, and since the average client think time equals the average request service time, the average number of clients in the think state to be $(6, 4)$. Thus, the remaining number of requests $(8, 12)$ must be queued or somewhere in transition in the network. During phase Φ_C the load was reduced to $(L_P, L_B)_{\Phi_C} = (5, 20)$ and the resulting allocation was $(w_{g,P}, w_{g,B}) = (4, 6)$. The queue length for Premium was negligible, and most of the requests in the queue manager buffers were from the Basic clients. During phase Φ_D the traffic load was switched to $(L_P, L_B)_{\Phi_D} = (20, 5)$ and the resulting allocation was $(w_{g,P}, w_{g,B}) = (8, 2)$, yielding an average queue length for Premium that is about twice as much as for Basic. The performance during phase Φ_D is not ideal because the Premium traffic has a lower target response time than Basic and we would desire a smaller queue.

The performance in this phase is due to the small number of maximum concurrent requests allowed to executed on the server. Since $N_s = 10$ the global manager can allocate server shares in coarse increments of 10% only. A decrease in the Basic allocation from 2 to 1 would have resulted in extremely poor performance for the requests associated with the Basic clients.

We could have achieved a better performance if we could have used a fractional allocations. An allocation vector of $(w_{g,P}, w_{g,B}) = (8.3, 1.7)$ would have would have increased the performance of the Premium requests without exploding the response time of the Basic clients. Based on this results we are improving our system to support fractional weights.

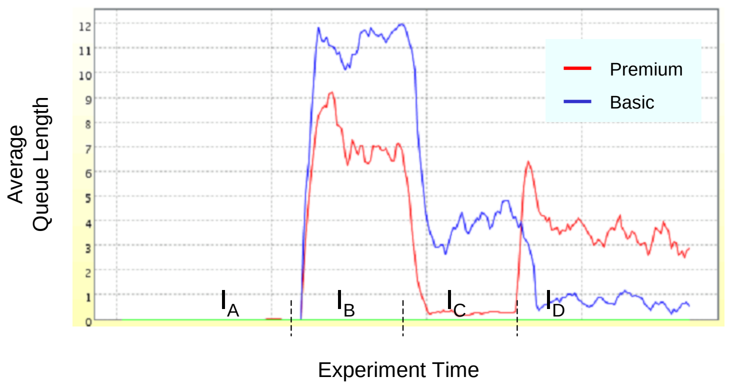


Figure 11: Average queue length

The throughput curves are shown in Figure 12. During phase Φ_A the total throughput may be evaluated as the ratio of the total number of clients, 10, and the total round trip time (think time plus service time), $2sec$, yielding $5req/sec$, or about $2.5req/sec$ for each class. During phases Φ_B, Φ_C, and Φ_D the server was busy most of the time executing the maximum number of concurrent requests $N_s = 10$ and therefore the total throughput was limited to $10req/sec$ (obtained by dividing 10 threads by the service time of $1sec$).

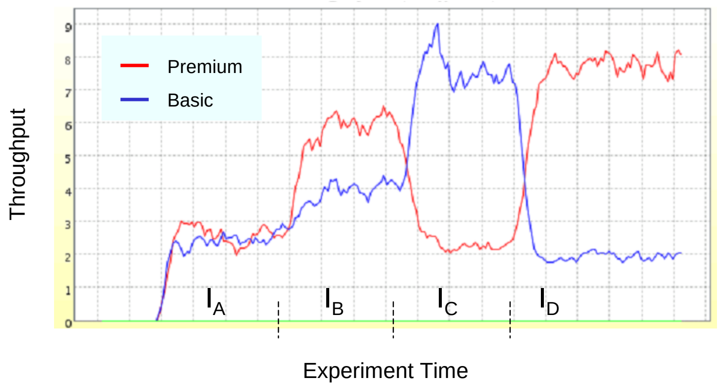


Figure 12: Throughput

Figure 13 illustrates the utility values. For the experiments reported in this section we used the optimization criterion that maximize the minimum of the utility values of Premium and Basic traffic. In other words, the optimization attempts to yield equal utility values for both traffic classes. A near perfect equalization is achieved during phases Φ_A, Φ_B, and Φ_C. As for phase Φ_D, there is a difference between the two utility values. This is due to the use of an integer allocation vector, rather than a fractional one.

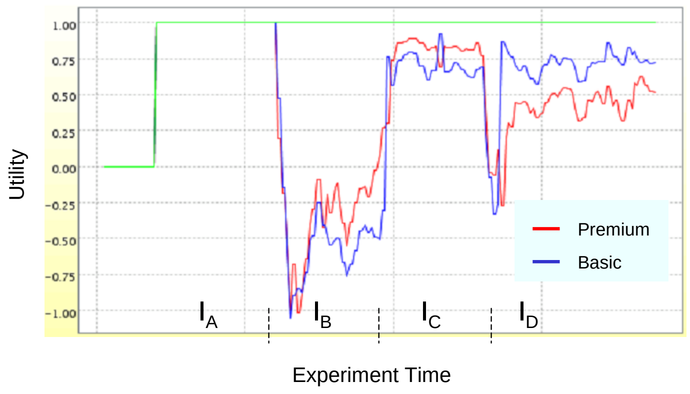


Figure 13: Utility functions

5.3 Optimality of Resource Allocation

In this section we compare the behavior of our system to two systems that use a FCFS (First-Come-First-Served) and SP (Static Priority) scheduling disciplines, respectively. In the FCFS system, requests are treated similarly, independent of their class. All requests queue up in a single FCFS queue and wait until one the server becomes available. In this section we still limited the maximum number of requests concurrently executing on the server to $N_s = 10$ to maximize the server performance and to study the impact of the allocation discipline in isolation.

When a request completes a corresponding response is generated and a server thread becomes available. In the SP system, we implemented two queues, one for each class of requests. When a thread becomes available, the request in the head of the highest priority queue uses it until the request completes. In both FCFS and SP systems, the target response time values are not used to decide which request will be served.

We consider the experimental setup described above, where there are two classes of requests: Premium and Basic. Instead of a single traffic point per phase, we consider a two-dimensional workload traffic space, given by the number of Premium clients and Basic clients, respectively. We ran experiments using 5, 10, 15, and 20 clients of each class, thus resulting in a 16-point space as depicted in Fig 7. At each point we measure the resulting cluster utility function, which in our case is the minimum of the utilities of Premium and Basic classes. We use the same traffic, service time and target time values as in the previous section.

Figure 14 illustrates the utility regions, as a surface plot, obtained in the uncontrolled case of FCFS. We note that the utility function decreases as the number of clients increases. The contour lines are diagonal in a way that exhibits the lack of differentiation between Premium and Basic requests. For example, achieving a non-negative utility function value (i.e. both classes meet or exceed their targets) puts a limit of about 29 as the total number of clients.

The cluster utility with static priority (SP) scheduling is shown in Figure 15. First, we note that the contour lines are more slanted due to the preferential treatment of Premium requests. For example a zero utility value is achieved with $(L_P, L_B)_\Phi = (20, 5)$ clients or $(L_P, L_B)_\Phi = (11, 20)$ clients. Achieving the target for Basic requests requires less number of Basic clients in the former and less number of Premium clients in the latter. Second, we note that the utility region (for non-negative utility values) is roughly smaller than the corresponding region in the FCFS system. The total number of clients on the zero contour line varies from 25 to about 31 clients. For smaller values of the utility function, the utility regions become remarkably smaller than the corresponding regions in the FCFS system.

Figure 16 shows the utility regions obtained with our optimized controlled system. The resulting utility regions are larger than both the FCFS and the SP systems. This means that we can accommodate more

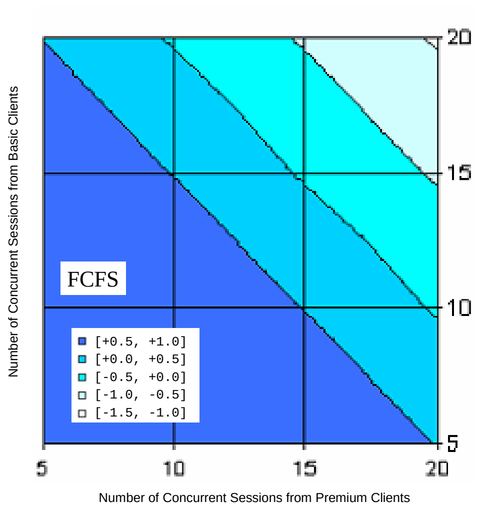


Figure 14: Utility regions with FCFS scheduling

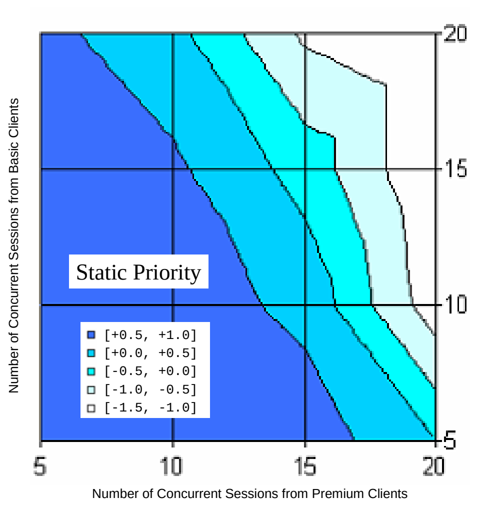


Figure 15: Utility regions with Priority scheduling

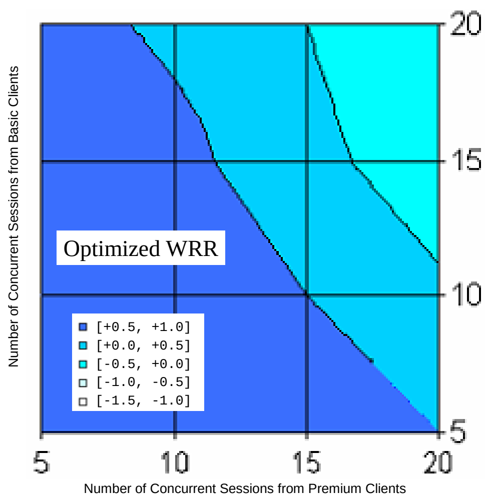


Figure 16: Utility regions with Optimized scheduling

workload using the same resources while achieving the target response times. The zero contour line passes by points where the total number of clients is somewhere between about 32 and 35 clients. We achieve this result because the global resource manager allocates server resources to optimize the cluster utility function.

6 Conclusions and Future Work

We have presented an architecture and a prototype implementation of a performance management system for cluster-based web services. The management system is transparent and allocates server resources dynamically so to maximize the expected value of a given cluster utility function. We use a cluster utility to encapsulate business value, in the face of service level agreements and fluctuating offered load. The architecture features gateways that implement local resource allocation mechanisms. A global resource manager solves an optimization problem and tunes the parameters of the gateway's mechanisms. In this study we have used a simple queuing model to predict the response time of request for different resource allocation values. Feedback controllers based on first-principles model of the system converge quickly and with fewer oscillations than controllers based on a black-box model.

Our work can be extended in several directions. Our platform could be enhanced with additional management functionality such as policing, admission control and fault management. We will need to develop more sophisticated models of web services and web services traffic loads to study and predict platform performance under different service and traffic conditions. The effect of control parameters, such as control cycle, on the performance of the feedback controller needs further study. We could refine our global resource manager by adding black box and hybrid control techniques. Finally, we will need to study the impact of using other scheduling algorithms on the end-to-end resource management problem, especially in the presence of multiple gateways.

7 Acknowledgment

We would like to thank Ron Levy and Jayakumar Nagarajao for their valuable contributions to this work.

References

[1] S. Vaughan-Nichols, "Web services: Beyond the hype," *IEEE Computer*, vol. 35, pp. 18–21, February 2002.

[2] S. Aissi, P. Malu, and K. Srinivasan, "E-business process modeling: The next big step," *IEEE Computer*, vol. 35, pp. 55–62, May 2002.

[3] D. Schmidt, "Middleware for real-time and embedded systems," *Communications of the ACM*, vol. 45, June 2002.

[4] H. Chen and P. Mohapatra, "Session-based overload control in QoS-aware web servers," in *Proceedings of the IEEE INFOCOM*, (New York, NY), June 2002.

[5] J. Carlström and R. Rom, "Application-aware admission control and scheduling in web servers," in *Proceedings of the IEEE INFOCOM*, (New York, NY), June 2002.

[6] T. Abdelzaher, K. Shin, and N. Bhatti, "Performance guarantees for web server end-systems: A control-theoretical approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, January 2002.

[7] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury, "Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache web server," in *Network Operation and Management Symposium*, (Florence, Italy), pp. 219–234, April 2002.

[8] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra, "Kernel mechanisms for service differentiation in overloaded web servers," in *In Proceedings of the USENIX Annual Technical Conference*, (Boston, MA), June 2001.

[9] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger, "Oceano SLA based management of a computing utility," in *Proceedings of the International Symposium on Integrated Network Management*, (Seattle, WA), pp. 14–18, May 2001.

[10] H. Zhu, H. Tang, and T. Yang, "Demand-driven service differentiation in cluster-based network servers," in *Proceedings of the IEEE INFOCOM*, (Anchorage, AL), April 2001.

[11] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle, "Managing energy and server resources in hosting centers," in *Proceedings of the ACM Symposium on Operating System Principles*, (Chateau Lake Louise, Banff, Canada), pp. 103–116, October 2001.

[12] M. Aron, P. Druschel, and W. Zwaenepoel, "Cluster reserves: A mechanism for resource management in cluster-based network servers," in *ACM Sigmetrics*, (Santa Clara, CA), June 2000.

[13] G. Banga, J. Mogul, and P. Druschel, "Resource containers: A new facility for resource management in server systems," in *Proceedings of the Symposium on Operating Systems Design and Implementation*, (New Orleans, LA), February 1999.

[14] T. Zhao and V. Karamcheti, "Enforcing resource sharing agreements among distributed server clusters," in *Proceedings International Parallel and Distributed Processing Symposium*, (Ft. Lauderdale, FL), pp. 501–510, April 2002.

[15] S. Low and D. Lapsley, "Optimization flow control I: basic algorithm and convergence," *IEEE/ACM Transactions on Networking*, vol. 7, December 1999.

[16] P. Marbach, "Priority service and max-min fairness," in *Proceedings of the IEEE INFOCOM*, (New York, NY), June 2002.

[17] S. Microsystems, *Java Messaging Service API*. `http://java.sun.com/products/jms/`.

[18] *Apache XML Project*. `http://xml.apache.org/axis/`.

[19] *The Apache Jakarta Project*. `http://jakarta.apache.org/tomcat/`.