PERFORMANCE MANAGEMENT FOR CLUSTER BASED WEB SERVICES

R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef
IBM T.J. Watson Research Center, Yorktown Heights, NY 10598

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 [17]. 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

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35674-7_66

G. Goldszmidt et al. (eds.), *Integrated Network Management VIII*
© IFIP International Federation for Information Processing 2003

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 [15]. In this section we summarize the current state of the art.

The first class of research studies deals with session-based admission control for overload protection of web servers. Chen et al. [9] 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. [7] 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 SLAs.

Another area of research deals with performance control of web servers using classical feedback control theory. Abdelzaher et al. [1] 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 resources among different web services requests. Diao et al. [10] 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 [18]. 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 [4] and [20]. 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. [8] refine the above approach. They note that there are techniques (e.g., cluster reserves [5], and resource containers [6]) 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 [19] 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 [13] or a

minimum [14] 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 the class utility function in terms of the experienced performance relative to the guaranteed service objective. Thus, it is possible 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* of service level, with each *grade* defining a specific set of performance objective parameters. For example, the `StockUtility` service could be offered in either *gold*, *silver*, or *bronze* 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 95th percentile of the response times is smaller than 5 seconds, 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 homogenous 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 10 ms 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 the target web services are deployed. 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. Refer to [12] for details on how to use server pools.

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

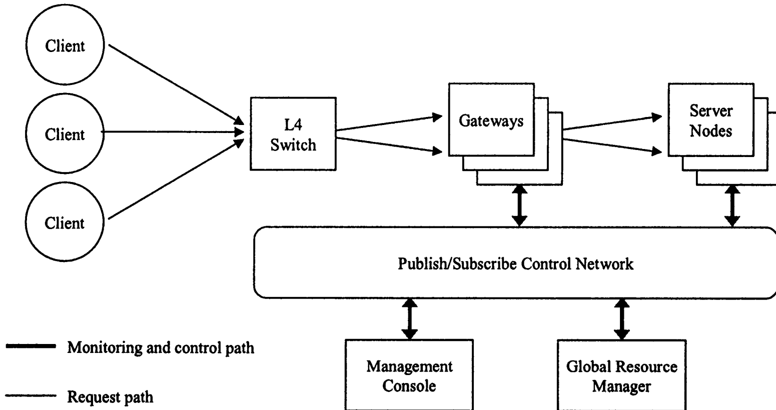

Figure 1. System Overview

3.1 Gateway

We use gateways to control 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.

Gateways dispatch requests to servers. We denote with N_s the capacity of server s. N_s represents the maximum number of web services requests that server s can execute *concurrently*. We select N_s to be large enough to efficiently utilize the server's physical resources, but small enough to prevent overload and performance degradation. In the remainder of this paper we assume that N_s is given.

We partition N_s among all gateways and we denote with $N_{g,s}$ the maximum number of concurrent requests that server s executes on behalf of gateway g. We also use $w_{g,c}$ to describe the minimum number of class c requests that all servers will execute on behalf of gateway g. Each request executes in a separate initial thread. Thus, we refer to $w_{g,c}$ as server *threads*. 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:

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

where C and S denote the set of all classes and servers, respectively. Figure 2 illustrates the gateway components. We have used Axis [3] 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) [3].

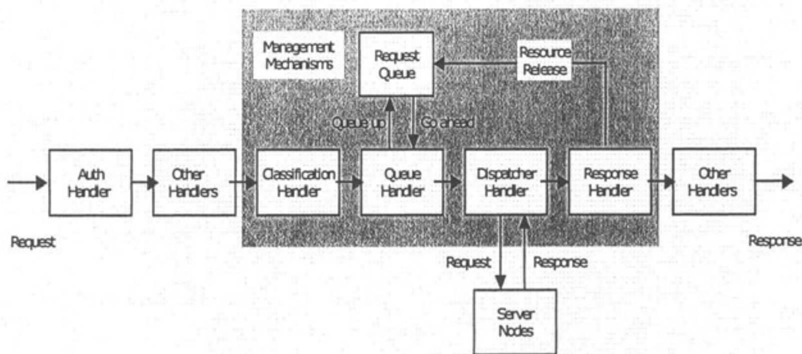

Figure 2. Gateway components

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 and 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 weighted 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 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 parameters. Our implementation supports SOAP over HTTP and SOAP over JMS [16]. The dispatch handler distributes the requests among the available servers using a simple load balancing discipline while enforcing the constrain that at most $N_{g,s}$ requests execute on server s concurrently on 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

The *global resource manager* runs periodically and computes $N_{g,s}$ and $w_{g,c}$ using the request load statistics and performance measurements from each gateway. 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 a set of class *utility functions* and a *combining function*. Each class *utility function* maps the performance of 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 function as a sum of the utility functions, however, our work could be extended to study the impact of other combining functions on the structure of the solution.

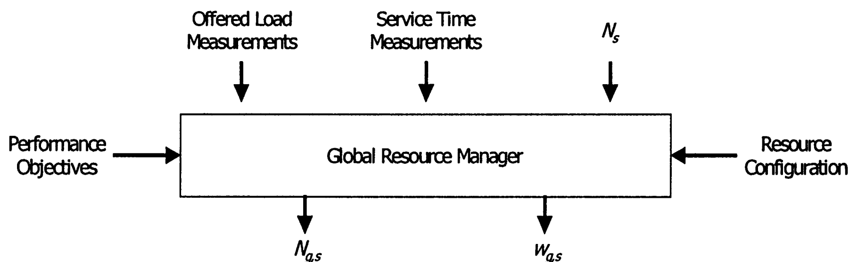

Figure 3. Global resource manager inputs and outputs

The global resource manager uses a queuing model of the system to predict the performance that each class would experience for a given allocation $w_{g,c}$ and the corresponding $N_{g,s}$. The global resource manager implements a dynamic programming algorithm to find the $w_{g,c}$ and $N_{g,s}$ that maximize the cluster utility function. 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 switches 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.

3.3 Management Console

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 the views available from our management console.

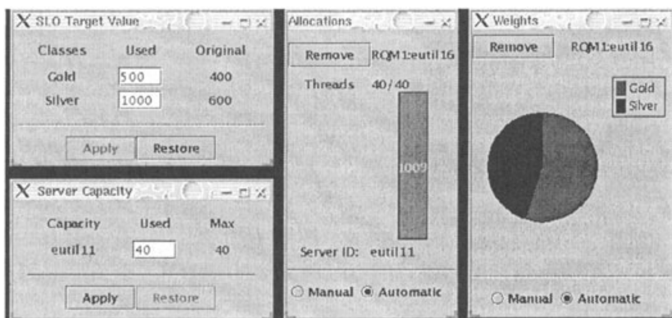

Figure 4. Management console: configuration and control values

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. Finally, we 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 define as the sum of each class utility function U_c. In particular Ω is given by:

$$\Omega = \sum_{c\in C}\sum_{g\in G} U_c(w_{g,c}) \tag{2}$$

subject to:

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

$$\text{where } N \equiv \sum_{s\in S} N_s, \tag{4}$$

and where 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. In the following section 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 [12]. Once we have computed $w_{g,c}$. the value w_g derived from equation 1 represents the portion of server resources that have been allocated to gateway g. To compute $N_{g,s}$ 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

$$[N_{1,s}, \text{K } N_{n_G,s}]$$

where n_G is 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 $[\hat{N}_{1,s}, \ldots \hat{N}_{n_G,s}]$ 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 equation 4.

4.2 The Structure of Class Utility Functions

We use U_c to encapsulate the business importance of meeting or failing to meet class c performance. 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 predicted response time given an allocation of $w_{g,c}$ resources. In the studies reported in this paper, we use a prototypical function to express the utility of class c when its requests experience a performance t_c under a contracted performance objective τ_c. An example for such a function is given below.

$$U_c(\tau_c, t_c) = \begin{cases} \alpha_c & if\, 0 \le t_c < 1/\mu_c \\ a_c\left(\dfrac{\tau_c - t_c}{\tau_c - 1/\mu_c}\right) & 1/\mu_c \le t_c < \tau_c \\ \dfrac{-\alpha_c (t_c - \tau_c)^{\beta_c}}{\beta_c (\tau_c - 1/\mu_c)} & t_c \ge \tau_c \end{cases} \tag{7}$$

The function in equation 7 and shown in Figure 5 compares average response time t_c to target response time τ_c for class c as follows. The best possible long-term average is $1/\mu_c$ where μ_c is the mean service rate for class c. When $t_c = 1/\mu_c$ $U_c(\tau_c, t_c)$ is constant. Between that point and $t_c = \tau_c$, we simply follow a straight line. For $t_c > \tau_c$ we use a negative polynomial function to map response times bigger than the objective into a negative value of $U_c(\tau_c, t_c)$. For the plot in Figure 5 we have used $\mu_c = 1$, $\tau_c = 6$, $\alpha_c = [1,2,3]$ and $\beta_c = [1,3,5]$. By increasing α_c we control the business importance of exceeding the target for class c, while by increasing β_c we can control how fast the business utility degrades when class c experiences a delay bigger than the objective.

By changing the size and shape of the utility function we can influence how resource are allocated to each class of traffic and in turn the class performance. A more detailed description of the concept of the utility function and its impact on the overall system is given in [12]. In the next section we describe how we estimate the expected response time t_c for class c given a scheduling weight of $w_{g,c}$.

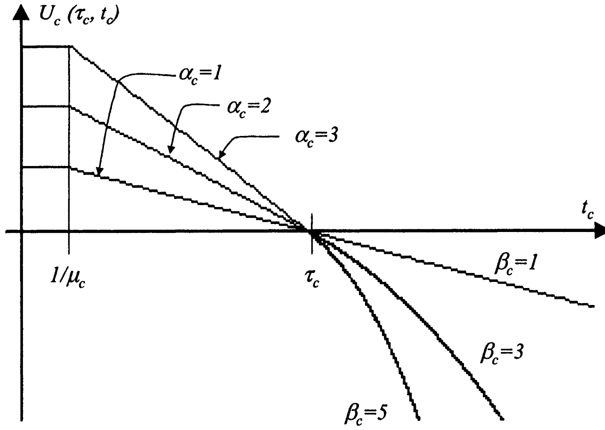

Figure 5. Utility function

4.3 System Modeling

To predict the average response time $t_{g,c}$ given a proposed allocation $w_{g,c}$ we use the observed arrival rate, response time, and the previous allocation values, denoted by $\tilde{\lambda}_{g,c}$, $\tilde{t}_{g,c}$, and $\tilde{w}_{g,c}$, respectively.

We use an M/M/1 queue [11] to model the response time behavior of requests of class c traveling through gateway g, i.e., we assume that $\tilde{\lambda}_{g,c}$ is evenly divided among the $\tilde{w}_{g,c}$ 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:

$$\tilde{\mu}_{g,c} = 1/\tilde{t}_{g,c} + \tilde{\lambda}_{g,c}/\tilde{w}_{g,c} \qquad (8)$$

Figure 6 exemplifies the above assumption. 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

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

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

Using equation 9 and 7 we can express the utility $U_c\ (\tau_c, t_c)$ 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 equation 2 under the constraints in equation 3.

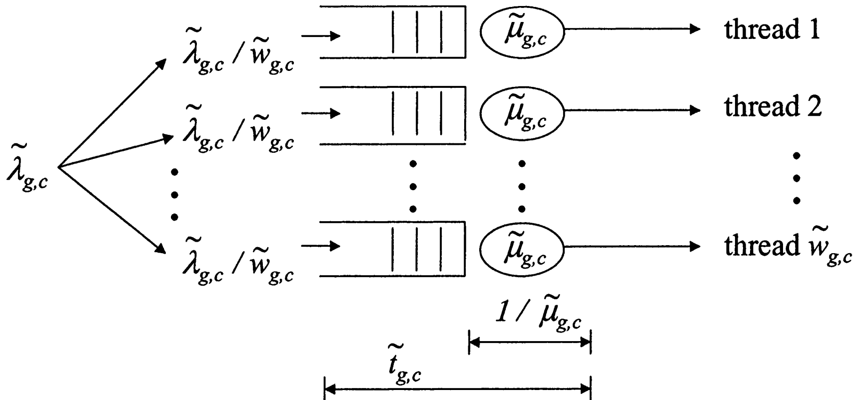

Figure 6. Modeling the response time behavior for class *c* requests handled by gateway *g*

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 [12] we report the results of several experiments indented to study the effectiveness of this approach. In the next section we report a subset of these experiments.

5. EXPERIMENTAL RESULTS

In order to illustrate the fundamental behavior of the system, the following experiments were conducted using a combined gateway and server machine, while another machine was used to generate the traffic load.

During the experiment, clients connect to the gateway and send requests to a synthetic service, with exponentially distributed service time. The service alternates between CPU-bound processing and sleeping. The sleeping intervals are intended to emulate periods in which a process awaits response from a back-end server or database.

In order to determine the desired N_S for the one server, we examined the system throughput for various settings of N_S. In these experiments, the load consisted of only one traffic class, and we ensured that the request queue was always non-empty. As shown in Figure 7, a maximum throughput of 23.5 requests/sec is achieved at an N_S of 10. For larger values of N_S the CPU reaches saturation and the overhead begins to degrade the server throughput. In the experiments reported below N_S was always set to 10.

In the following experiment, clients are classified into two types: *gold* and *silver*. The gold clients' service level agreements specify a performance objective of *1 sec* average response time, while the silver clients' service level agreements specify a *2 sec* average response time target.

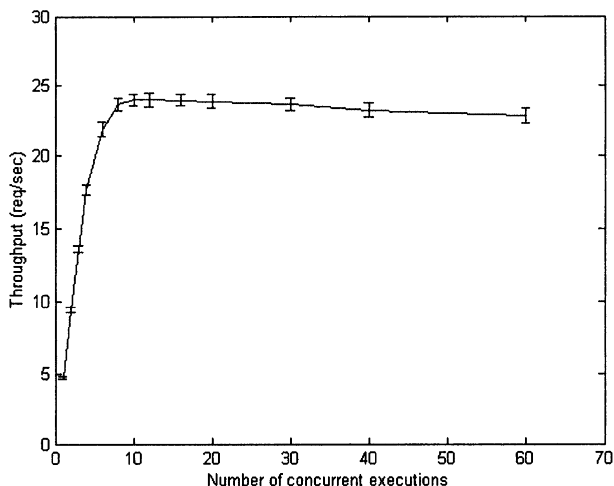

Figure 7. Throughput vs requested maximum number of concurrent executions

This experiment emulated an infinite client population, where initially the server was subjected to *1req/sec* from gold clients and *11.5req/sec* from silver clients. After *100sec*, the gold traffic rate was increased to *11.5req/sec*, which brought the total load close to the system capacity. We show the effect of this change in traffic on response time in Figures 8. Since the invoked service spends a good portion of its time performing CPU-bound processing, the service time increases as the degree of concurrency of executing requests increases. This experiment demonstrates that the control mechanism immediately started to react to the load changes in order to maximize the cluster utility. In this experiment, the control cycle for the global resource manager was set to *10sec*. We smoothed the load and response time statistics used by the global resource manager over a *30sec* intervals using a sliding window. The plot in Figures 8 shows values smoothed by that *30sec*-sliding window. We report more details on these experiments as well as additional experiments, with specific settings, in [12].

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.

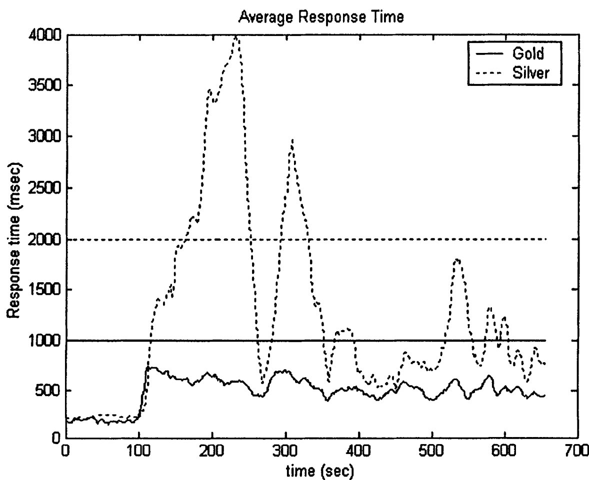

Figure 8. Response time for infinite client population experiment

REFERENCES

[1] 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, No. 1, Jan 2002.

[2] S. Aissi, P. Malu, and K. Srinivasan, "E-Business Process Modeling: The Next Big Step", *IEEE Computer* 35(5), pp 55-62, May 2002.

[3] Apache XML Project, http://xml.apache.org/axis/

[4] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, DP. Pazel, J. Pershing, and B. Rochwerger, "Oceano SLA based management of a computing utility", *Proceedings of 2001 International Symposium on Integrated Network Management,* Page 14-18. May 2001.

[5] M. Aron, P. Druschel, and W. Zwaenepoel, "Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers", *ACM Sigmetrics 2000*, Santa Clara, CA, Jun 2000.

[6] G. Banga, J. Mogul, and P. Druschel, "Resource containers: A new facility for resource management in server systems", *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (OSDI'99), New Orleans, LA, Feb 1999.

[7] J. Carlström, and R. Rom, "Application-aware Admission Control and Scheduling in Web Servers", *IEEE INFOCOM 2002, New York, NY*, Jun 2002.

[8] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle, "Managing Energy and Server Resources in Hosting Centers", *Proceedings of 18th ACM Symposium on Operating System Principles*, pages 103-116, Oct 2001.

[9] H. Chen and P. Mohapatra, "Session-Based Overload Control in QoS-Aware Web Servers", *IEEE INFOCOM 2002, New York, NY*, Jun 2002.

[10] 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 ApacheWeb Server", Proc. NOMS 2002, 219-234, Apr 15-19, 2002, Florence, Italy.

[11] L. Kleinrock, Queueing Systems – *Volume 1: Theory*, John Wiley, 1975.

[12] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef, "Performance Management For Cluster Based Web Services", *IBM Research Technical Report*, RC22676, Dec 2002.

[13] S. H. Low and D. E. Lapsley, "Optimization Flow Control I: basic Algorithm and Convergence", *IEEE/ACM Transactions on Networking*, Vol. 7, No. 6, Dec 1999.

[14] P. Marbach, "Priority Service and Max-Min Fairness", *IEEE INFOCOM 2002, New York, NY*, Jun 2002.

[15] D. Schmidt, "Middleware for Real-Time and Embedded Systems", *Communications of the ACM*, Vol. 45, No. 6, Jun 2002.

[16] Sun Microsystems, *Java Messaging Service API*, http://java.sun.com/products/jms/

[17] S.J. Vaughan-Nichols, "Web Services: Beyond the Hype", *IEEE Computer*, Feb 2002.

[18] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra, "Kernel Mechanisms for Service Differentiation in Overloaded Web Servers", *In Proceedings of the 2001 USENIX Annual Technical Conference, Boston, MA*, Jun 2001.

[19] T. Zhao and V. Karamcheti, "Enforcing Resource Sharing Agreements among Distributed Server Clusters", *Proceedings International Parallel and Distributed Processing Symposium, IPDPS 2002*, Ft. Lauderdale, FL, Apr 2002, pp. 501-510.

[20] H. Zhu, H. Tang, and T. Yang, "Demand-driven Service Differentiation in Cluster-based Network Servers", *IEEE INFOCOM 2001, Anchorage, Alaska*, Apr 2001.