RC23942 (W0604-124) April 25, 2006
Computer Science

IBM Research Report

Managing the Response Time for Multi-tiered Web Applications

Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, Malgorzata Steinder, Asser Tantawi, Ian Whalley
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Managing the Response Time for Multi-tiered Web Applications

Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer,
Malgorzata Steinder, Asser Tantawi, and Ian Whalley
IBM T. J. Watson Research Center

Abstract

We introduce an edge-based technique for providing differentiated service and simple overload protection in the case of an inhomogeneous workload and a partially overlapping deployment, notably including a multi-tier structure. Previous work has shown how reverse proxies doing inexpensive operations, under guidance from a controller, can provide differentiated service and overload protection assuming single-tier homogenous or disjoint deployment and, in some techniques, homogenous workload. We use a model-based controller with an optimization objective. The added complexities of inhomogeneous workload and deployment are handled by two key ideas: (1) the proxies classify traffic and protect against overload at an appropriate granularity, and (2) the controller solves an optimization problem subject to bounds derived from the deployment and using coefficients that characterize the computational intensity of the traffic. The modeling in the controller includes quantitative reasoning about computational load and thus supports other functions besides control of the proxies. We present some experiments that demonstrate differential queuing to manage the response times of applications with different intensities and changing and partially overlapping deployment.

1 Introduction

We build on previous work [15, 18] that has shown how to do edge-based performance management of clustered web applications, using a controller that addresses an optimization problem based on performance models and goals, assuming homogeneous or disjoint deployment and, in some techniques, homogeneous workload. We describe how a technique that assumes homogeneity can make a large error when managing a heterogeneous system. We show that an inhomogeneous deployment of an inhomogeneous workload can be handled by suitable granularity in the proxies and a suitable way of bounding the controller's search. We elaborate this idea in the context of one particular system, and suspect the idea is more broadly applicable.

The challenges faced by a real-world data center include the following. There are multiple applications deployed. They are structured into multiple tiers, potentially with replicated server processes in any given tier. There is some overlap, in at least two ways: (a) two applications may overlap in one tier of machines and (b) two applications may use the very same server processes in another tier of machines. Within one tier of machines, there may be too many applications for each machine to run a server process for each application. The offered load for each application changes over time. In addition to the changing arrival rate, the computational load imposed by the average request of an application also changes over time. There is some dynamism in the configuration of the data center, at least due to actions by administrators and possibly also due to on-line management techniques. There are performance goals to be met; those goals may be stated in terms of latency (e.g., response time).

There are many different things that can be done to manage such a data center, and these things differ in scope and time scale. At the largest scope and time scale is the activity of adding new machines to the data center and retiring old ones. While working with a fixed set of machines, there may be coarse-grained decisions that dedicate each machine to use in one part of the data center; this partitioning might be based on the customer, the kind of processing done, the kind of finer-grained management used, and/or other things. At a finer grain there are decisions on the specific deployment issue we will call *placement*: which processes run on which machines. For example, one machine might run an application server dedicated to application A and an application server dedicated to B, while another machine runs an application server dedicated to B and an application server

dedicated to C. There are many technologies that involve special processing at the edge of the data center, for performance as well as other (e.g., security) functions. The performance functions at the edge include rejection and queuing of requests, as well as routing to servers. There may also be rejection, queuing, and routing in the internals of the data center. At the finest grain is performance management within each machine, by its operating system and/or other mechanisms. Note that finer grained mechanisms generally make smaller changes but do them faster and cheaper.

A study of a complete and integrated suite of management techniques that do all of the above things is well beyond the scope of this paper. We focus on a management technique that uses layer 7 reverse proxies at the data center's edge and optimization based on performance models to (a) do queuing and rejection at the edge and (b) support the coarser grained functions outlined above. For the sake of brevity we further focus on (a), giving only very brief mention of (b). In a commercial product [8] we have implemented both (a) and (b) using this technique; in companion papers [11, 10] we describe the associated server placement controller.

The rest of this paper is organized as follows. Section 2 presents the overall organization and characteristics of our technique, including a discussion of the granularity of traffic classification and overload protection. Section 3 fills in the key details on the controller. Section 4 presents some empirical studies. Section 5 compares the technique in this paper to key related work. Section 6 presents our conclusion and discusses some future work.

2 System Overview

Figure 1 gives an overview of the components and communication involved in our technique and a broader performance management context. The solid black lines and boxes carry and perform the useful work, the dashed lines and boxes outline our management technique, and the dotted lines and boxes are example parts of the broader management context. Our technique has wide applicability because it applies control only at the edge of the data center and requires relatively non-invasive monitoring. The useful work in the data center is done by a collection of running server processes. Work originates in clients, which send requests through the Internet to the data center. Our system begins with a set of reverse proxies at the edge of the data center. Between these proxies and the clients can be various routing and load balancing techniques such as DNS-based load balancing, layer 2-3-4 sprayers, etc. Our proxies forward requests to some of the server processes, which do some work and possibly make requests on other server processes. The example in the figure shows exactly two tiers of server processes, but the technique supports a general flow of work within the data center.

In addition to this flow of computational load, there are flows of management messages. The proxies and load balancers produce traffic statistics, which are used by the *resource controller* and possibly other components. The resource controller produces control parameters that affect the operation of the proxies. The resource controller also can produce information on the computational power requirements in the data center for use by other components in the broader context; an example of such a component is shown: a placement controller that starts and stops server processes to best utilize the available computational resources to meet the needs of fluctuating offered load. The resource controller needs to be given power consumption factors. These could be configured by administrators. Alternatively, to cope with changes in these factors over time, a broader performance management context could involve continuous (re)estimation of the power consumption factors by an on-line work profiling component, using CPU utilization readings from the server machines and/or processes as well as the aforementioned traffic statistics.

Edge-based techniques for response time management, overload protection, and a host of other functions are not new. Layer 7 intermediation (either in separate processes or as an early stage in server processes) is often used for purposes besides the management that is the focus of this paper. Indeed, the general topology of server processes, server process clusters, and machines allowed here — and often used in real-word data centers — requires layer 7 considerations for simply routing the requests.

Figure 4 (in section 4) illustrates the relationships between the key concepts of (a) server processes, (b) clusters, and (c) machines. The server processes are organized into clusters; each cluster is a collection of server processes, and each server belongs to exactly one cluster. In general, a cluster consists of a set of functionally equivalent replicas of some easily replicable service; all the servers in a cluster can handle the same traffic (although affinity, due to things like HTTP session state, may cause a particular request to prefer a particular server). An example would be a collection of J2EE application servers in which the same J2EE modules have been deployed and equivalently configured. For a service that is non-replicable, a cluster that always contains exactly one server is used. Our technique can cope with startups and shutdowns of server processes; indeed, it can be used to drive a controller that initiates such startups and shutdowns.

The incoming traffic to each cluster is load balanced by some technique that equalizes the response times among the servers in a cluster. In the figure there is a

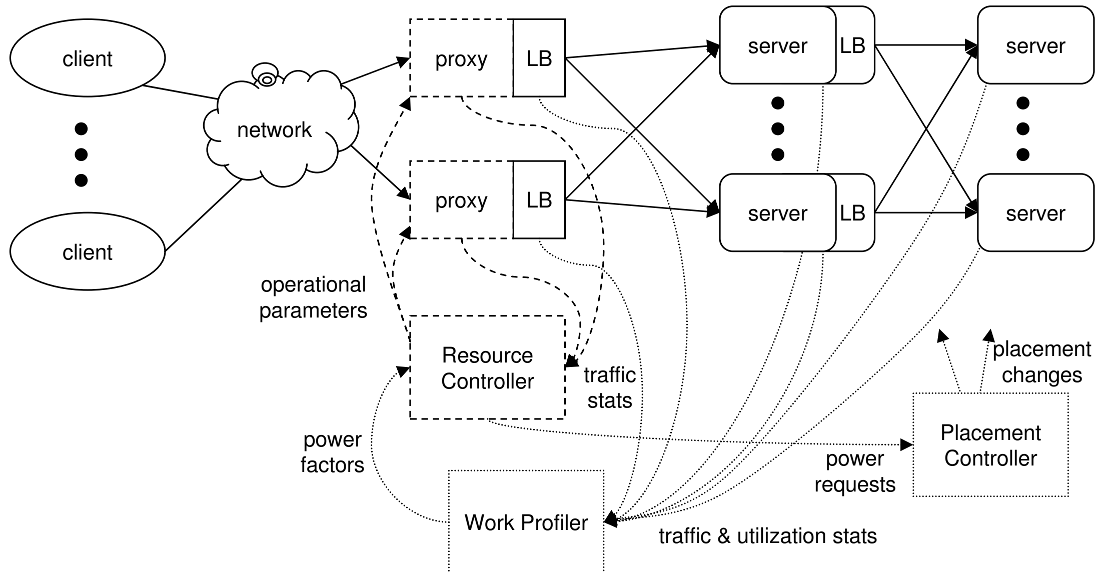


Figure 1: System Overview (including other performance management components)

load balancing component in each client of each cluster, but our technique can utilize any organization of components and communication that gets the load balancing job done.

In the course of doing its work, the software in one cluster may invoke services of one or more other clusters. For example, a cluster of simple HTTP servers may invoke a cluster of J2EE application servers, which may in turn invoke both (1) a singleton cluster of a relational database server and (2) a cluster of LDAP servers; the HTTP servers could also invoke the LDAP cluster. It is this flow of computational load among clusters that captures the multi-tier structure of a data center.

In a data center with multiple applications deployed, some applications may share the services of some clusters. The relationships among the clusters capture the relationships among the applications, at an appropriate level of granularity for management. In the remainder of this paper we will focus on the clusters rather than the applications.

Each server process runs on a machine, and a machine may host several running server processes at a time. For simplicity, we assume here that there is no point in running multiple server processes for a given cluster on one machine; thus, the servers running on a machine are all in different clusters. Notably, we do not require that each machine has, at a given time, a running server process for every cluster; in some real word data centers there are so many clusters that no machine has enough memory to run that many server processes at once. Rather, our technique accepts an arbitrary *server placement matrix*. We use the symbol I for such a matrix. For a given machine m and cluster d, $I_{m,d}$ holds a Boolean value indicating whether that machine currently hosts a running server process for that cluster. We suppose it is possible to automatically sense the current value of this matrix. The resource controller respects this matrix by using it to impose bounds on the resource allocations considered.

We allow a fairly arbitrary relationship between clusters and machines. Multiple clusters can overlap (i.e., have running servers) on a given machine; those clusters can be directly related or unrelated by the flow of computational work. Among the machines hosting a cluster, the set of overlapping clusters can vary from machine to machine.

Figure 2 shows the internal structure of one (“X”) of three reverse proxies (the others are “Y” and “Z”) in an example system. Each incoming request is first classified, to determine (a) the cluster that will serve it and (b) the service class to apply. In the figure, “A”, “B”, and “C” are clusters, while “Gold”, “Silver”, and “Bronze” are service classes. Administrators define service classes, indicating for each (i) the requests to which it applies, (ii) the performance target of the class, and (iii) the relative importance of the class. We consider two kinds of response-time based performance target: averages and percentiles.

A proxy has, for each cluster to which it directly forwards traffic, a pipeline of (a) a set of FIFO queues, (b) a scheduler that draws requests from these queues, (c) a concurrency limiter, and (d) a load balancer for the cluster. We use the term *gateway* for such a pipeline, and the variable g to index over them. For a given gateway g, there is a queue for each service class c that applies

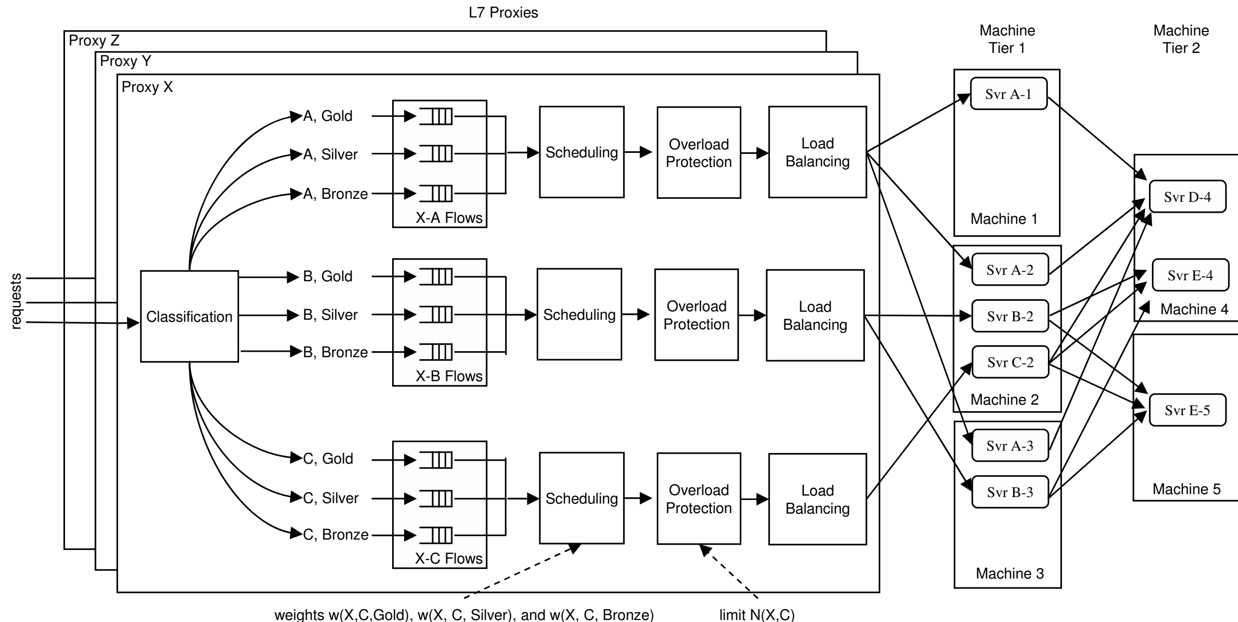


Figure 2: Internals of a proxy

to some traffic through that gateway. When asked, the scheduler picks a queue from which to draw the next request, using a simple weighted round-robin scheme. That is parameterized by a weight $w_{g,c}$ for each queue (we can identify a queue by the $\langle g, c \rangle$ to which it applies). The concurrency limiter provides a simple form of overload protection. It notes the flow of both requests and replies, and allows at most some given number N_g of requests to be outstanding at a time. We say N_g is the number of *seats* allocated to gateway g. In a given gateway, a new request is released from a queue toward a server process when both (1) the number of outstanding requests is below its limit and (2) some queue holds a request. The concurrency limit and the scheduler's round robin weights together comprise the control parameters of a gateway. The figure calls out those parameters for the gateway for cluster "C" through proxy "X". The resource controller periodically adjusts the control parameters of all the gateways.

Note that because a gateway's concurrency limiter enforces only an aggregate concurrency limit, one flow may temporarily borrow a seat from another when the latter is not using all its allocated seats. Thus, each gateway is work-conserving. This comes at the risk that, for example, a low importance request may block a high importance request that arrives soon after. But that blockage will continue only until the next completion of some low importance request — which is generally fairly soon with realistic traffic.

While an individual gateway conserves work, the whole collection of them does not. If there is a sudden large shift in load between the gateways, the management might significantly under-load the data center until the next time the resource controller adjusts the control parameters of the gateways. In practice the frequency of that adjustment is on the order of once per minute, so the duration of the under-load will be limited. One could reduce the risk of such under-loads by using fewer gateways (i.e., being less discriminating in defining gateways), but that raises a complementary problem: too much borrowing.

One could imagine various ways of diving a proxy's incoming traffic into flows and collecting them into groups for overload protection. A very simple way — which is used in earlier work [14, 18] — divides only by service class, and uses just one overload protection unit for all of a proxy's traffic. Alternatively, the traffic could be classified very finely — by service class, directly-

loaded cluster, and perhaps other things — and each flow could be subjected to independent overload protection. The more finely a proxy's traffic is divided into overload protection units, the less borrowing is allowed — and this decrease is generally a bad thing. However, it is possible to allow too much borrowing. Specifically, if flow A can borrow from flow B, but A imposes a different pattern of computational load on the data center than does B, then that borrowing can lead to over-loading one part of the data center while under-loading another part. With a sufficiently large difference in load pattern, and a sufficiently large amount of borrowing, an arbitrarily large load imbalance can result. However, as in the case of too little borrowing, this imbalance will persist only until the resource controller adjusts the control parameters of the gateways — assuming the controller is sensitive to the current placement. If the controller is placement-insensitive, it will not restore balance.

In the context of the product containing our implementation we have chosen to divide each proxy's traffic into a flow for every relevant ⟨directly-loaded cluster, service class⟩ pair and use in that proxy an overload protection unit for every directly-loaded cluster. We chose to divide based on the directly-loaded cluster because it is both easy to determine from inexpensive request inspection and generally a pretty helpful discriminator in this regard. In other contexts the best answer might be different; indeed, we may add more discrimination in the future, based on a cost/benefit trade-off.

The focus of our work is on the response time management; we consider only a simple scheme for overload protection. This scheme, outlined above, uses the queues to shape traffic over short transients. To handle long-term overload we simply impose a limit on the length of each queue, dropping requests that would exceed this limit. A more sophisticated treatment of overload is a topic for future work.

3 The Resource Controller

Our response time management technique involves a feedback control loop, in which gateways take inexpensive actions to manage the request flow guided by control parameters that are periodically adjusted by a resource controller based on traffic statistics and configured service classes. Figure 3 outlines how the resource controller works. The key idea is this: to derive the control parameters for the gateways, the resource controller solves a resource allocation problem with bounds derived from the current server placement. In a broader performance management context, this controller can also produce information on the ideal allocation of a data center's compute power by solving a similar resource allocation problem but with bounds derived from the *potential*

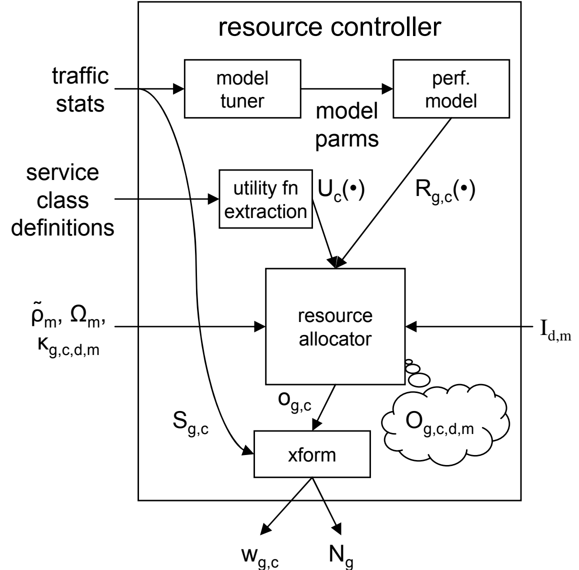


Figure 3: The resource controller

placement of the servers. In this section we outline how the resource controller works; we have a technical report [14] that provides more details (avoiding the concept of clusters in an attempt at brevity).

The resource allocation problem is posed as an optimization problem with a fairness goal. Specifically, the goal is to equalize the utilities of the expected performance results of the various traffic flows. The proxies divide the traffic into flows, one for every combination of gateway g and service class c for which traffic of that class actually flows through that gateway. Usually only a subset of the possible combinations actually occur, and we write $\mathcal{F} \subseteq \mathcal{G} \times \mathcal{C}$ for the set of actual flows; here $\mathcal{G}$ the set of gateways, and $\mathcal{C}$ the set of service classes. The decision variables in the optimization problem are allocations of concurrency to flows: there is a decision variable $o_{g,c}$ for every $\langle g, c\rangle \in \mathcal{F}$. The decision for a flow is the number requests it may have concurrently outstanding at its gateway; we say this is the number of seats allocated to the flow. The resource controller derives its various outputs from the chosen values of the decision variables.

The expected performance results are estimated by a model whose parameters are extracted on-line from monitoring of the data center. The exact choice of model is not our primary concern here. We favor a model whose parameters can be tuned on-line based only on relatively non-invasive instrumentation. In our implementation we use a model that is tuned using only the traffic statistics from the gateways. Other models [18, 17] could be used. Because the model parameters are tuned on-line, it is generally easy to make the model fairly accurate in the neighborhood of the current operating state. The primary

model quality issue is what happens when the model is asked to predict the results for a very different state. As used here in the controller that picks the operating state, the big picture issue is what do the transients look like. While we have not mounted a rigorous study of transients, section 4 shows some. Note that they are fairly quick and decisive.

Given particular values for its parameters, the model for a flow $\langle g, c\rangle$ produces a function $R_{g,c}(\cdot)$ that maps a hypothetical resource allocation $o_{g,c}$ to the expected performance result. That result is either the expected average response time or (in the case of a percentile goal) the expected percentage of requests whose response time would exceed the goal's threshold. Thus, these functions are non-increasing. Note also that a flow's function depends only on the allocation for that flow. This is only an approximation of actual behavior; we make this approximation because it, along with the convexity of $U_c(R_{g,c}(\cdot))$, makes the optimization problem separable — which admits efficient solution techniques. Note that the overload protection limits the degree to which actual behavior will deviate from this approximation.

The fairness objective is based on utility functions that map expected performance to utility; it is the utility values that are equalized. There is a utility function $U_c(\cdot)$ for each service class c. This function is derived from the performance goal and relative importance of the service class. The utility function is non-increasing, which when composed with the non-increasing performance prediction function gives us the desired convexity.

The resource allocation problem is primarily to pick an integer value for each $o_{g,c}$ in a way that maximizes

$$\min_{g,c} U_c(R_{g,c}(o_{g,c})) \tag{1}$$

subject to certain bounds. Those bounds say that there exists a feasible *seat placement*. A seat placement O assigns a non-negative real number $O_{g,c,d,m}$ to every combination of flow $\langle g, c\rangle$, cluster d that (possibly indirectly) serves that flow, and machine m that hosts a running server of d. That number is the number of gateway seats through which flow $\langle g, c\rangle$ may load cluster d's server on machine m. For every such cluster d and flow $\langle g, c\rangle$ that loads it, the seat placements across the machines that host the cluster d must add up to the allocation $o_{g,c}$ (remember that seats are always counted in terms of requests outstanding at the gateways).

The bounds are captured by two sets of inequalities. One set says that the current server placement must be respected — if there is no running server for cluster d on machine m, then the relevant seat placements must all be zero:

$$\forall d, m : (\neg I_{m,d}) \Rightarrow \forall g, c : (O_{g,c,d,m} = 0)\ . \tag{2}$$

The other set of inequalities says that no machine may be overloaded:

$$\forall m : \sum_{d \in \mathcal{D}(m), \langle g,c\rangle \in \mathcal{F}(d)} O_{g,c,d,m}\kappa_{g,c,d,m} \leq \tilde{\rho}_m \Omega_m \tag{3}$$

where $\mathcal{D}(m)$ is the set of clusters that have running servers on machine m and $\mathcal{F}(d)$ is the set of flows that load (possibly indirectly) cluster d. We write Ω_m for the computational power of machine m. We allow the administrators to configure a utilization limit $\tilde{\rho}_m$ for each machine.

The no-overload constraint uses a power consumption factor $\kappa_{g,c,d,m}$ for each actual flow $\langle g, c\rangle$, cluster d it loads (possibly indirectly), and machine m hosting a server in d. A power consumption factor quantifies the average compute power that would be utilized on machine m, due to a server of cluster d, by a single request of flow $\langle g, c\rangle$ if all its work in d were done on m; this is averaged over the whole time from the request's entry to its exit at the server to which its gateway directly sends it. For example, consider a flow of HTTP GET requests that take an average of 100 milliseconds to run (from entry to the HTTP server to reply from it). Suppose such a request causes, on average, 1.5 database queries, and these DB queries consume an average of 100 standard megacycles each. The power consumption factor for that flow, the database cluster, and its machine is 1.5 $\times$ 100 standard MC / 0.1 seconds, which is 1,500 standard MHz. We quantify compute work and power in machine-independent units; think of them as megacycles and MHz on a standard kind of machine. These could refer to computational resources besides CPUs; what is important is that we quantify the usage of the resource that is the bottleneck. As described in this paper, our technique can handle any situation where the bottleneck on a given cluster is always the same kind of resource; it is the power of that kind of resource that should be used here. It is straightforward to extend the resource controller to consider multiple kinds of resource per machine (we have done so in our actual implementation, but omit it from this paper for the sake of brevity).

We solve this optimization problem by an optimal greedy algorithm for allocations to flows, augmented with a heuristic technique for constructing a feasible seat placement. The seat placement bounds make the problem NP hard, and in total our solution is heuristic. While we have not yet mounted a rigorous study of the optimality of our solution, we have noticed few suboptimal answers in practice.

The full resource allocation problem goes beyond maximizing the minimum of all the utility values. If only the global minimum were maximized, any flow that "gets stuck" before the others will hold down that maximum

minimum, allowing arbitrary allocations (within a potentially wide range) among all the other flows. A flow can get stuck either by reaching a point of diminishing returns in its composite utility-of-allocation function (i.e., $U_c(R_{g,c}(\cdot))$) or by lack of available power due to that flow having an exceptionally large power consumption factor. To get good results for all flows, we ask for maximization of the minimum among successively smaller sets of flows. This is easily handled by the greedy optimization algorithm — it simply drops flows from further consideration when they get stuck.

The control parameters for the gateways are easily derived from the seat allocations. The concurrency limit N_g is simply the sum of the allocations $o_{g,c}$ for the relevant service classes c. When fully loaded, the scheduler divides its total throughput among its service classes in proportion to their round-robin weights. Inspired by Little's Result, which says that the average concurrency in a box is the product of the throughput and the average time an item spends in the box, we set the round-robin weight $w_{g,c}$ to be the quotient $o_{g,c}/S_{g,c}$ where $S_{g,c}$ is the average service time (time from (a) entry to the first server to (b) completion) for the flow.

Since the product of seat allocation and power consumption factor is a power allocation, this controller can support additional functions that require reasoning about compute power. For example: the sum of $O_{g,c,d,m}\kappa_{g,c,d,m}$, over the flows $\langle g, c\rangle$ that load cluster d and the machines m that host servers in that cluster, is an estimate of the compute power that will be used by cluster d if the current server placement remains in effect. This quantitative compute power modeling can be used to answer other hypothetical questions such as: (a) what would happen if the server placement were changed in some specific way, (b) how much power would be needed in each cluster to achieve the highest possible utility supposing the server placement could be changed within certain bounds (our implementation actually computes this, for use by the associated placement controller), (c) what would happen if the service class goals were changed in some specific way, and (d) we result could be achieved with more or fewer machines of a given kind? These are useful in working with provisioning controllers that operate on a coarser scale, and in capacity planning.

4 Experimental Results

In this section we investigate four issues over the course of two experiments. The first experiment: (i) demonstrates how the proposed system protects against server overload situations by basing resource allocation decisions on power consumption factors and machine power, (ii) shows how the system achieves service level differentiation, and (iii) illustrates how the system respects placement constraints while making resource allocation decisions. All those is done for a pair of applications with different power consumption factors. The second experiment studies the system resilience to application server placement changes. In both experiments, we compare results obtained with and without our control mechanisms.

We study the behavior of our platform using a pair of benchmark applications and a synthetic load. We used the setup described in Fig. 4 to run our experiments. The setup used two applications, TradeA and TradeB, deployed on J2EE application servers and a database server.

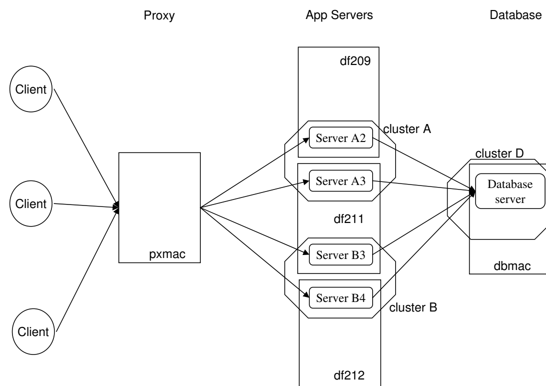


Figure 4: Experimental topology

For our applications we used two different configurations of Trade6, an IBM WebSphere end-to-end benchmark and performance sample application. This benchmark models an online stock brokerage application and it provides a real world workload driving J2EE 1.3 modules and Web Services. TradeA consists of the Trade6 application configured to use direct JDBC connections. TradeB consists of Trade6 configured to access the database through a layer of Enterprise JavaBeans (EJBs). We used these two different configurations to study the effects of web requests that bring different resource demands to the platform.

We also configured the system with two service classes: *gold* and *bronze* and we set an average response time goal of 350 ms for the gold requests and 1.2 seconds for the bronze requests. We also configured the gold request with the highest importance level and the bronze requests with the lowest level. We mapped all the URIs associated with Trade A onto the gold class and all the URIs associated with TradeB onto the bronze class.

Service Class	Requests	Response Time Target	Importance
Gold	TradeA	350 ms	1
Bronze	TradeB	1,200 ms	99

Table 1: Service classes

flow	cluster	machine	factor
TradeA	A	df209	125 MHz
TradeA	A	df211	125 MHz
TradeA	D	dbmac	50 MHz
TradeB	B	df211	300 MHz
TradeB	B	df212	300 MHz
TradeB	D	dbmac	75 MHz

Table 2: Power consumption factors

	df209, df211, df212	dbmac
Ω	1,490 MHz	8,000 MHz

Table 3: Machine power

For our experiments we used three machines running the WebSphere J2EE application servers and one machine running DB2. We used three machines to run the two partially overlapping clusters of application servers. We deployed both TradeA and Trade B on df211, while deploying only TradeA on df209 and only TradeB on df212. We used the machine named dbmac to run the database server. Table 3 shows that the machines running the application servers all have the same CPU power. The machine running the database server has greater power. Table 2 shows the power consumption factors for the two flows. We used a machine with power similar to df212 to run the proxy that fronts the application tiers. Finally we used a set of machines to run the client session emulators.

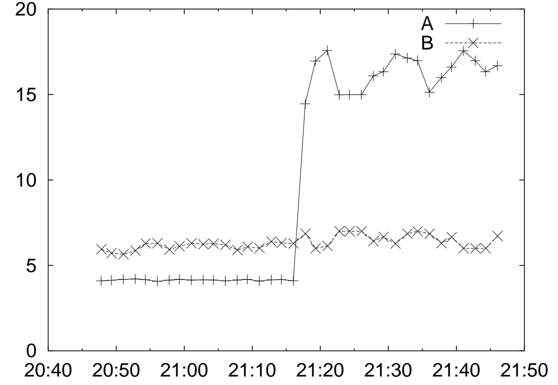


Figure 5: Experiment 1, number of requests concurrently executing

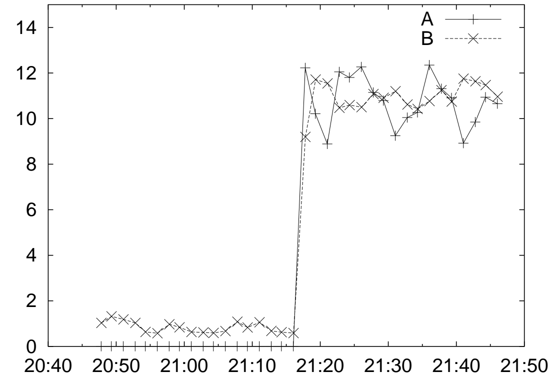


Figure 6: Experiment 1 - Average queue length

4.1 Handling inhomogeneity

In the first experiment the generated load goes through two phases; each phase is about 20 minutes long. In the first phase there are 10 clients for TradeA and 10 for TradeB. In the second phase there are 40 clients for TradeA and 20 for TradeB.

Figure 5 illustrates the number of requests permitted by the proxy to execute concurrently. During the first phase, where the number of emulated sessions of TradeA and TradeB are equal, the concurrency of TradeB is slightly higher than that of TradeA. During the second phase, where the number of emulated sessions of TradeA is double that of TradeB, the concurrency of TradeB remained the same at less than half the concurrency of TradeA. The rest of the TradeB sessions are made to wait in the proxy queue, as Figure 6 shows. This is due to the fact that $\kappa_{\mathrm{gw}_A,\mathrm{Gold},A,m}$ is about half of $\kappa_{\mathrm{gw}_B,\mathrm{Bronze},B,m}$ (regardless of m). This is more evident when we observe that the CPU utilization levels in the two phases are almost equal, as shown in Figure 7. Thus, taking power consumption factors into account while making resource allocation decisions enabled us to prevent machine overload in phase two of the experiment.

In phase two of the experiment, where the total offered load is above the system capacity, queuing takes place. Figures 8 shows the average response time. Since the performance target for TradeA traffic is tighter than that for TradeB traffic, and its importance is much higher than TradeB, the system differentiates accordingly, favoring TradeA over TradeB traffic, as expected.

Figure 9 illustrates the concurrent requests executing on each machine, for each of the two applications. As can be easily observed from the figure, TradeA requests always executed on df211 and df209, while TradeB requests always executed on df211 and df212. df211 was the only shared machine, and its power was divided among the two traffic classes by the resource controller. At all times, the placement constraints were respected.

As more of the CPU power of df211 is shifted from TradeB to TradeA, in the second phase of the experiment, we can see that the concurrency of TradeB on df211 decreases by about three sessions, while that of TradeA

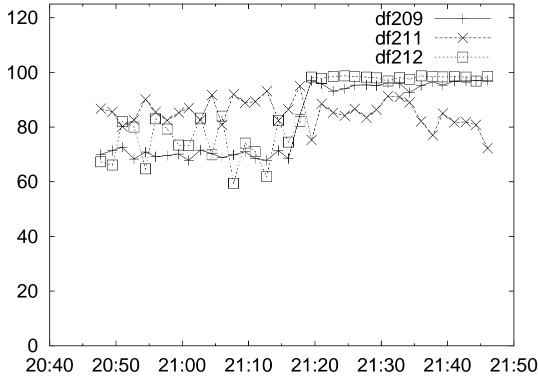


Figure 7: Experiment 1 - CPU Utilization

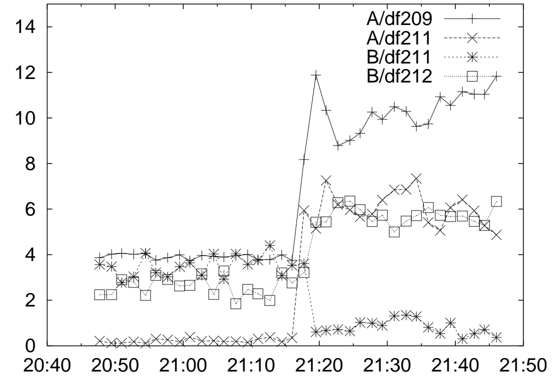


Figure 9: Experiment 1 - Number of requests concurrently executing at each machine

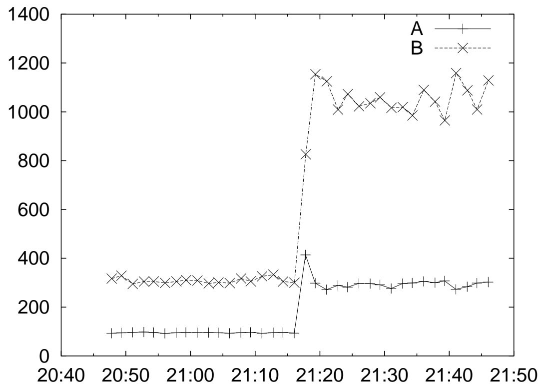


Figure 8: Experiment 1 - Average response time

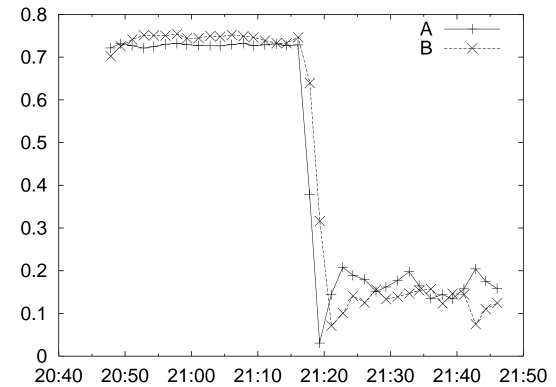


Figure 10: Utility per flow, with control

rises by about six sessions. This is because the power consumption factor of TradeB requests is about double that of TradeA requests, as previously mentioned. This emphasizes that the heterogeneity in power consumption factors of different request types is respected while making resource allocation decisions, and shifting resources from one traffic class to another.

The goal of our controller is to equalize the utility values of the various traffic classes, independent of offered load, server placement, and available power. This is apparent from Figure 10 where we see that both TradeA and TradeB had similar high utility values during phase one, while they both had similar lower utility values during phase two. Without a controller, the utility values would have been as illustrated in Figure 11, where the utility values of TradeA goes negative (i.e. missed target) while TradeB receives high utility values. The average response time for TradeA and TradeB in the uncontrolled case is illustrated in Figure 12. Note that TradeA missed its target of 350 msec during phase two, while TradeB was well below its target of 1,200 msec. In contrast, using our controller and as shown in Figure 8, the average response time for both TradeA and TradeB were slightly below target during the second phase of the experiment.

4.2 Resilience to placement changes

In this experiment, we start with the same placement of application servers as in the first experiment. However, after 15 minutes from the beginning of the experiment we emulate a machine failure scenario by bringing one machine (df209) down, and we observe how the system reacts to this bottleneck. After another 15 minutes, we bring df209back up, with only TradeA placed on it. We repeat the same scenario for an uncontrolled system (no queuing in our proxies) and compare the results of the 2 cases.

In Figures 13 (a)–(f), we compare the performance of the managed flows in the controlled and uncontrolled systems side by side. Figures 13 (a) and (b) show throughput of TradeA and TradeB. We see that, in the uncontrolled case, the throughput of the high importance traffic (TradeA) drops by as much as 70% when df209is

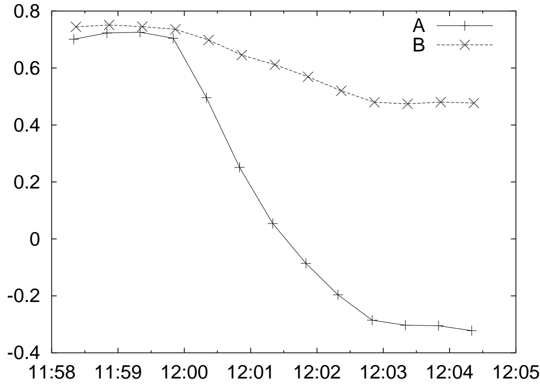


Figure 11: Utility per flow, without control

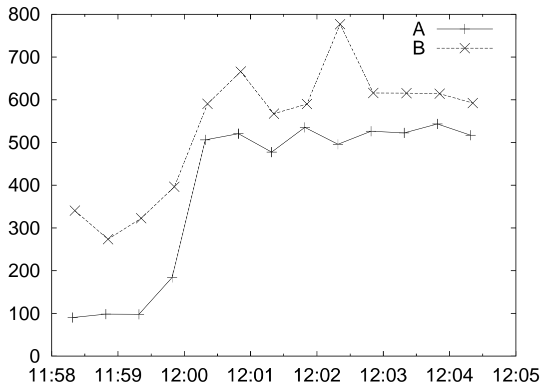


Figure 12: Experiment 1 - Average response time, without control

brought down at around 8:02. In the controlled case, the impact of server failure at 23:58 is reduced to about 50%. At the same time, the throughput of TradeB class is only slightly lower in the controlled case. Furthermore, in the uncontrolled case (Figure 13 (d)), TradeA experiences response time which is much above its goal of 350 ms, while in the controlled case, it almost always stays below this target (Figure 13 (c)). While the machine is down, the performance of TradeB is worse in the controlled case compared to the uncontrolled case, as the response time goal of TradeB is much higher. Nevertheless, even TradeB always stays below or near its response time goal.

Recall that the objective of our system is to equalize utilities among flows. Figures 13 (e) and (f) show that in the controlled case the management indeed kept the values of the objective functions for TradeA and TradeB closer to each other than happened in the uncontrolled system. In the controlled case, the values of both utility functions are above 0, indicating both flows meet their performance goals. In the uncontrolled case, the lower importance class receives better performance at the expense of the higher importance class.

5 Related Work

During the last couple of years, we have seen a proliferation of research work in the area of QoS management for web applications. The work may be generally divided into a few categories: admission control and overload protection, service differentiation, low-level mechanisms, feedback control-based mechanisms, and utility-based mechanisms. Due to space limitations, we will mention some examples of prior work in the various categories and contrast our work to the prior art.

Admission Control and Overload Protection. Urgaonkar and Shenoy [18] developed the *Cataclysm* system for handling the case of extreme overloads in a single tier of web application servers. They studied an admission control system that runs on a "sentry" tier and decides in real-time and low overhead which flow to admit when the resources in the application tier are overloaded. They consider the placement control of applications on nodes in a disjoint fashion. Assuming off-line profiling of applications and invasive instrumentation, they implement a non-work conserving priority discipline in the sentry and provide admission control and overload protection mechanisms. Welsh and Culler [21, 20] presented a multi-stage approach to overload control based on adaptive per stage admission control. In this approach, the system actively observes application performance and tunes the admission rate of each processing stage to attempt to meet a 90th-percentile response time target. This approach is based on the SEDA architecture [22], and was extended to perform class-based service differentiation. In such multi-staged admission control approaches, a request may be rejected late in the processing pipeline, after it has consumed significant resources in upstream stages. Blanquer et al. [4] described *Quorum*, a non-invasive approach to scalable quality-of-service provisioning that uses traffic shaping, admission control, and response monitoring at the border of an Internet site to ensure throughput and response time guarantees. The load controller uses a sliding window mechanism similar to TCP. They demonstrated that their system is capable of dealing with sudden surges in the traffic pattern.

Our focus is not a solution to the admission control problem. Though concerned with overload protection, our goal is to manage the response time QoS for service classes in a multi-tiered server farm environment. Our objective is to manage the end-to-end performance of request flows. We address common deployment scenarios in which a web application may be replicated on different but overlapping subset of machines, at each tier.

Service Differentiation. Service differentiation in cluster-based network servers has been studied in [2] and [24]. There, the approach 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. Chase et al. [5] refined the above approach by partitioning server resources and quickly adjusting the proportions. Using a black-box model, they solved a cluster-wide optimization problem, where the utilization of a server is represented by a cost term. Zhao and Karamcheti [23] proposed a distributed set of queuing intermediaries with non-classical feedback control that maximizes a global objective. Their technique seems to suffer from the coupling of the global optimization cycle and the scheduling cycle.

Our approach is to control the flows at the edge of the data center and aim for a fairness result constrained by the current server placement. We use statistical multiplexing, which makes fine-grained resource partitioning possible, and unused resource capacities can be instantaneously shared with other traffic classes.

Low-Level Mechanisms. The approach in this category of work is to tackle the problem at lower protocol layers, such as HTTP or TCP. As a consequence, modifications to the web server or the OS kernel are necessary in order to incorporate the control mechanisms. Web server overload control and service differentiation using OS kernel-level mechanisms, such as TCP SYN policing, has been studied by Voigt et al. [19]. Harchol-Balter et al. [7] and Schroeder and Harchol-Balter [16] studied socket-level prioritization of packets based on a shortest remaining processing time policy.

Basically, our approach is non-invasive, in the sense that it does not require changes to the kernel, and applies control only at the entrance to the system.

Feedback Control-Based Mechanisms. Another area of research deals with performance management through classical feedback control theory. Abdelzaher et al. [1] used a feedback controller 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. Kamra et al. [9] addressed the problem of overload protection and meeting QoS requirements in a multi-tiered web environment by employing a self-tuning PI (Proportional Integral) controller in a proxy called *Yaksha*. The gain parameters of the controller are calculated dynamically using a M/G/1 processor sharing queue as a model of the multi-tiered system. Their approach is non-invasive and collects external performance measurements such as traffic density and response time. Diao et al. [6] used feedback control based on a black-box model to maintain desired levels of memory and CPU utilization.

Rather than employing feedback controllers, which may work well in the neighborhood of an operating point, we use queuing network models to predict performance as a function of load and amount of allocated resources. The models are dynamically tuned based on measured quantities. We believe that models provide an educated guess of (1) the performance when there is a significant change in the load and (2) the impact of allocating a given amount of resources to a service class.

Utility-Based Mechanisms. The notion of using a utility function and maximizing a sum [12] or a minimum [13] of utility functions for various classes of service has been used extensively to support service level agreements in communication services. Recently, the same concept has been applied to Web servers. Ardagna and Zhang [3] proposed a controller for multi-tier web data centers, which maximizes the profits associated with multi-class service level agreements. The cost model consists of a class of linear utility functions which include revenues and penalties incurred depending on the achieved average response time and the cost associated with running servers. The overall optimization problem considers the set of servers to be turned on, the allocation of applications to servers, and routing and scheduling at servers as joint control variables. This problem is NP-hard. The authors ended up dividing the problem into smaller problems, and developed heuristics based on a local search algorithm.

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. Through the use queuing network models, our system maps the performance to a utility function, and constantly adapts the resource allocation, yielding fairness among the various service classes. Our approach is to address each of the performance management problems by a separate autonomic controller, thus decomposing a large combinatorial problem into smaller and more manageable problems.

6 Conclusions and Future Work

We have shown how to do edge-based response time management and simple overload protection for an inhomogeneous placement and inhomogeneous workload. The key ideas are (1) to classify traffic and apply overload protection at an appropriate granularity and (2) to bound the resource allocation space according to the placement and the workload characteristics.

Several interesting issues remain to be investigated, both in terms of studying the existing technique and improving it. For the former, the transient behavior and the optimality of the resource controller's search algorithm have already been called out. Another interesting topic to explore is further failure scenarios. For example, we are working on enhancements in how the management reacts to various subsets of the requests getting stuck in an idle state inside the data center. There remain many other failure modes to investigate.

It is also interesting to investigate the joint behavior of a collection of management techniques (as suggested in the introduction).

There are data centers containing thousands of machines, and we are interesting in developing techniques that scale to such a size.

One (hopeful) improvement that we are pursuing is to quantify computing power requirements by the product of throughput and a *work factor* (i.e., average work per request) rather than the product of occupancy and power consumption factor. This has the advantage that work factors are (a) not machine-dependent and (b) easier to estimate from on-line observations.

The layer 7 proxies of our technique may not be the outermost edge of the data center. We have seen data centers with other edge components even farther out, and some of those can tolerate only a limited number of outstanding requests. We are working on rejecting requests to keep the occupancy within the scope of our management technique below a given limit.

The optimization objective should be affected by request rejections.

Many workloads include requests that are grouped into sessions, and the utility of serving or rejecting a request may depend on its role in its session (if it has one).

The overload protection is currently based on occupancy, specifically counting occupied seats. It would surely be an improvement to reckon with power rather than seats. We are also interested in the alternative of limiting throughput rather than occupancy. That is intriguing because it may provide a more robust approach to handling stuck-idle requests. Making it conserve work is an interesting challenge.

References

[1] ABDELZAHER, T., SHIN, K., AND BHATTI, N. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems 13*, 1 (January 2002).

[2] APPLEBY, K., FAKHOURI, S., FONG, L., GOLDSZMIDT, G., KALANTAR, M., KRISHNAKUMAR, S., PAZEL, D., PERSHING, J., AND ROCHWERGER, B. Oceano SLA based management of a computing utility. In *Proceedings of the International Symposium on Integrated Network Management* (Seattle, WA, May 2001), pp. 14–18.

[3] ARDAGNA, D., AND ZHANG, L. SLA based profit optimization in autonomic computing systems. In *International Conference On Service Oriented Computing* (New York, NY, November 2004), pp. 173 – 182.

[4] BLANQUER, J., BATCHELLI, A., SCHAUSER, K., AND WOLSKI, R. Quorum: Flexible quality of service for internet services. In *Second Symposium on Networked Systems Design and Implementation (NSDI)* (Boston, MA, May 2005).

[5] CHASE, J., ANDERSON, D., THAKAR, P., VAHDAT, A., AND DOYLE, R. Managing energy and server resources in hosting centers. In *Proceedings of the ACM Symposium on Operating System Principles* (Chateau Lake Louise, Banff, Canada, October 2001), pp. 103–116.

[6] DIAO, Y., GANDHI, N., HELLERSTEIN, J. L., PAREKH, S., AND TILBURY, D. M. 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, April 2002), pp. 219–234.

[7] HARCHOL-BALTER, M., SCHROEDER, B., BANSAL, N., AND AGRAWAL, M. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems 21*, 2 (May 2003), 207–233.

[8] IBM. Websphere extended deployment. http://www.ibm.com/software/webservers/appserv/extend/ (2005).

[9] KAMRA, A., MISRA, V., AND NAHUM, E. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. In *Twelfth IEEE International Workshop on Quality of Service (IWQoS)* (Montreal, Canada, June 2004).

[10] KARVE, A., KIMBREL, T., PACIFICI, G., SPREITZER, M., STEINDER, M., SVIRIDENKO, M., AND TANTAWI, A. Dynamic placement for clustered web applications. In *Proceedings of the 2006 World Wide Web Conference* (Edinburgh, Scotland, May 2006), International World Wide Web Conference Committee.

[11] KIMBREL, T., STEINDER, M., SVIRIDENKO, M., AND TANTAWI, A. Dynamic application placement under service and memory constraints. In *4th International Workshop on Efficient and Experimental Algorithms* (Santorini Island, Greece, May 2005).

[12] LOW, S., AND LAPSLEY, D. Optimization flow control I: basic algorithm and convergence. *IEEE/ACM Transactions on Networking 7*, 6 (December 1999).

[13] MARBACH, P. Priority service and max-min fairness. In *Proceedings of the IEEE INFOCOM* (New York, NY, June 2002).

[14] PACIFICI, G., SEGMULLER, W., SPREITZER, M., STEINDER, M., TANTAWI, A., AND YOUSSEF, A. Managing the response time for multi-tiered web applications. Tech. Rep. 23651, IBM, 2005.

[15] PACIFICI, G., SPREITZER, M., TANTAWI, A., AND YOUSSEF, A. Performance management for cluster-based web services. *IEEE Journal on Selected Areas in Communications 23*, 12 (December 2005), 2333–2343.

[16] SCHROEDER, B., AND HARCHOL-BALTER, M. Web servers under overload: How scheduling can help. In *Proceedings of 18th International Teletraffic Congress* (Berlin, Germany, September 2003).

[17] URGAONKAR, B., PACIFICI, G., SHENOY, P., SPREITZER, M., AND TANTAWI, A. An analytical model for multi-tier internet services and its applications. In *ACM Sigmetrics* (Banff, Canada, June 2005), pp. 291 – 302.

[18] URGAONKAR, B., AND SHENOY, P. Cataclysm: Policing extreme overloads in internet applications. In *Proceedings of the International World Wide Web Conference* (Chiba, Japan, May 2005), pp. 740 – 749.

[19] VOIGT, T., TEWARI, R., FREIMUTH, D., AND MEHRA, A. Kernel mechanisms for service differentiation in overloaded web servers. In *In Proceedings of the USENIX Annual Technical Conference* (Boston, MA, June 2001).

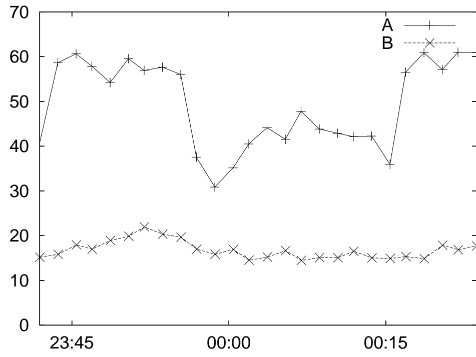
[20] WELSH, M., AND CULLER, D. Overload management as a fundamental service design primitive. In *Proceedings of the 10th ACM SIGOPS European Workshop* (Saint-Emilion, France, September 2002).

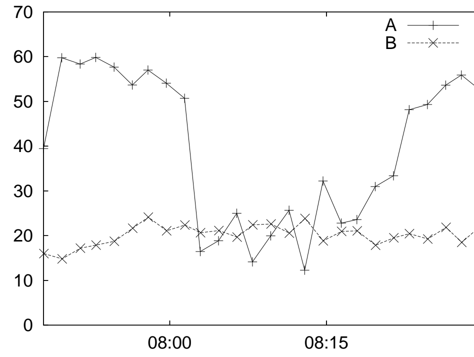
[21] WELSH, M., AND CULLER, D. Adaptive overload control for busy internet servers. In *Proceedings of the 4th USENIX Conference on Internet Technologies and Systems (USITS03)* (March 2003).

[22] WELSH, M., CULLER, D., AND BREWER, E. Seda: An arcitecture for well-contained, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Banff, Canada, October 2001).

[23] ZHAO, T., AND KARAMCHETI, V. Enforcing resource sharing agreements among distributed server clusters. In *Proceedings International Parallel and Distributed Processing Symposium* (Ft. Lauderdale, FL, April 2002), pp. 501–510.

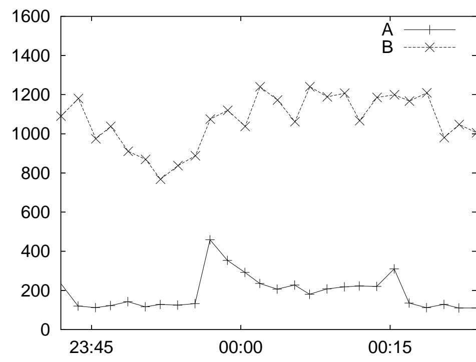
[24] ZHU, H., TANG, H., AND YANG, T. Demand-driven service differentiation in cluster-based network servers. In *Proceedings of the IEEE INFOCOM* (Anchorage, AL, April 2001).

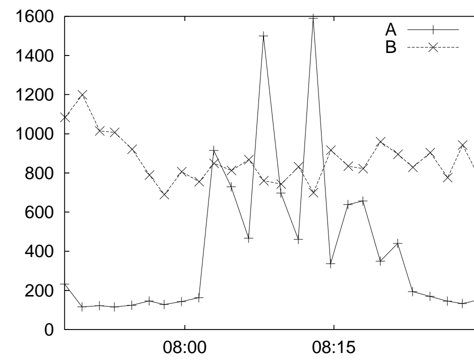




Throughput

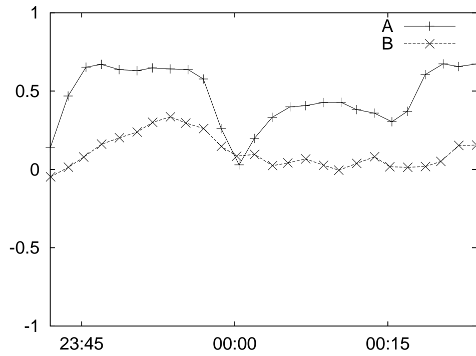
(a) Controlled (b) Uncontrolled

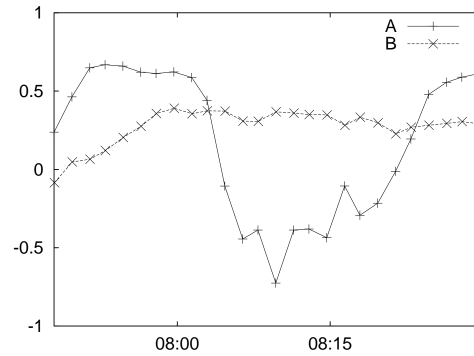




Response time

(c) Controlled (d) Uncontrolled





Utility

(e) Controlled (f) Uncontrolled

Figure 13: Experiment 2 - Comparing a controlled system to an uncontrolled one