RC23651 (W0507-029) July 5, 2005
Computer Science

IBM Research Report

Managing the Response Time for Multi-tiered Web Applications

Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, Malgorzata Steinder, Asser Tantawi, Alaa Youssef
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

Managing the Response Time for Multi-tiered Web Applications

Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, Malgorzata Steinder, Asser Tantawi, and Alaa Youssef
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
{giovanni,werewolf,mspreitz,steinder,tantawi,ayoussef}@us.ibm.com

***Abstract*— We present a system for managing the response time of web applications. This system allows service providers to group web application requests into different classes and assign response time goals to these classes of requests. We manage response times by using mechanisms that control the amount of server resources allocated to each class of web requests. We show how our platform can manage complex multi-tiered applications where each request uses multiple resources distributed over multiple tiers. We consider the case of a data center that supports multiple web applications, with each web application deployed and replicated on different but overlapping subsets of machines. We also show how our system can produce resource request signals that can guide another system that dynamically adjusts the application deployments. We also show how our system can manage heterogeneous requests by taking into account the amount of resources that each request consumes at each tier. To manage the response times of web requests we introduce an additional tier, the *Proxy tier*. The proxy adds layer-7 mechanisms that divide and control the flow of requests into the application tiers. We use a feedback control loop that periodically adjusts the resources allocated to each class. The feedback controller uses an approximate first-principles model of the system, with parameters derived from continuous monitoring. We discuss our prototype implementation and report experimental results that show the dynamic behavior of the system.**

I. Introduction

Many organizations rely on web applications to deliver critical services to their customers and partners. These service providers want to be able to specify response time goals for their web applications and be able to differentiate the performance delivered to preferred customers and business partners from other ordinary customers and partners. To manage the response time of web applications we must build platforms that provide resource virtualization and continuously monitor, in real-time and without human intervention, the performance and resource usage of user requests and dynamically decide how to allocate resources to meet the response time goals.

Several papers have reported studies that make important advances towards this goal. We categorize these studies into session-based admission control [1], [2], [3], server farm partitioning [4], [5], and request scheduling and routing techniques [6]. Some of these systems use classical feedback control [7], [8], some use reactive non-classical feedback [9], while others rely on optimizing a system-wide utility function [10], [11]. We have also addressed this problem in our prior work [6], [12], where we used a gateway tier, which performed multi-class queueing and weighted round robin scheduling of individual requests, in order to enforce statistical multiplexing of computing resources.

In this paper we describe a key progress towards the goal of managing response times for web applications. We describe a prototype platform and show how this platform delivers response time management for web application under realistic usage and deployment scenarios. In particular, we show how to manage multi-tiered web applications where each request uses multiple resources distributed over multiple tiers. We consider the case of a data center that supports multiple web applications, with each web application deployed and replicated on different but overlapping subsets of machines. In such a data center there may be another controller that dynamically adjusts the placement of applications on server machines; we show how our system can produce resource requests to guide such a placement controller.

We also show how our system can manage heterogeneous requests that use computing resources with heterogeneous capacities. We classify requests according to configured policy and use estimates of the resource demand by each kind of request for each machine and consider the capacity of each machine in each tier.

We describe how we implemented these response time management mechanisms by extending a popular middleware platform for Java 2 Enterprise Edition (J2EE) web applications. We have embedded our mechanisms in the middleware without relying on any special performance management functionality from the underlying operating system. In this way our system can support web applications that run on the many operating systems supported by the J2EE platform we used. Our management technique requires no help from the application code hosted by the middleware.

Our system uses a tier of layer-7 request proxies that classify incoming requests according to configured policy

and monitors their response times and resource usage. The gateways slectively queue and release requests to manage response time goals and protect computing resources from overload. We group requests into service classes and we use a per class utility function to express the value of any given performance result. A resource controller decides on: (i) the maximum number of outstanding concurrent requests allowed from each gateway, and (ii) a weighted round robin scheduling weight per queue; these are enforced by the layer-7 gateways. Those values are derived from the results of a resource allocation problem. We formulate the resource allocation problem as an optimization problem with an objective function derived from the class utility functions.

The rest of this paper is organized as follows. Section II discusses related work and what our study adds to the existing literature. Section III presents the system architecture and implementation. The resource allocation problem is formulated in Section IV. Section V describes the performance model used to derive response times from resource allocations. Section VI presents experimental results, which demonstrate the behavior of the system. In Finally, in Section VII we discuss our conclusions.

II. Related Work

Several research groups have addressed the issue of QoS support for middleware systems [13]. 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. Urgaonkar et al. [14] developed a system for handling the case of extreme overloads in web application servers. They study 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. Chen et al. [1] 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. [2] proposed using generalized processor sharing for scheduling requests, which are classified into multiple session stages with transition probabilities. Welsh et al. [3], [15] 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 [16], and was extended to perform class-based service differentiation. The downside of these multi-staged admission control approaches is that a request may be rejected late in the processing pipeline, after it has consumed significant resources in upstream stages.

Web server overload control and service differentiation using OS kernel-level mechanisms, such as TCP SYN policing, has been studied in [17]. Socket-level prioritization of packets, based on a shortest remaining processing time policy, has been also studied in [18] and [19]. 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 middleware layer, which does not require changes to the kernel, and allows for finer granularity of content-based request classification.

Another area of research deals with performance control of web servers using classical feedback control theory. Abdelzaher et al. [7] 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. Diao et al. [8] used feedback control based on a black-box model to maintain desired levels of memory and CPU utilization. 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 requests. We use a first-principles model and maximize a system-wide objective function.

All the abovementioned approaches focus on enhancing the performance of a server singleton, which is useful. However, having such enhanced servers does not solve issues related to optimizing load distribution in server farms, which is our focus in this paper.

In [9], a collection of middleware services, for monitoring, configuration, and load balancing, is used to construct a QoS-aware clustering service. The system relies on warning and breaching points that are derived from the SLA. Warning and breaching points for throughput, response time, etc, are combined into one point for the resource. Once the resource usage reaches the warning point, configuration adaptation must start. If the breaching point is reached, then adaptation has failed. In contrast, our system maps the observed performance to a continuous utility function, and constantly adapts the resource allocation to optimize the overall system utility.

Service differentiation in cluster-based network servers has also been studied in [4] and [5]. 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. [10] refine the above approach. They note that there are techniques (e.g., cluster reserves [20], and resource containers [21]) 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. They use a black-box model rather than first-principles one.

Zhao and Karamcheti [22] 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.

The approaches mentioned above for managing clustered web servers lack support for multi-tiered web applications. They do not provide a comprehensive management framework that manages the request performance from entry to exit from the system. In addition, these solutions do not address important realistic deployment scenarios in which each web application is replicated on different but overlapping subset of machines, at each tier.

The notion of using a utility function and maximizing a sum [23] or a minimum [24] 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. In [11], the authors 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. 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. Our approach is to address each of the above problems by a separate autonomic controller, optimized specifically for the problem it solves. In this paper, we focus only on the techniques used for optimizing the server allocations and scheduling weights, given a certain set of application placement constraints. The separation makes the controllers' problems simpler and easier to solve in real-time. Additionally, we believe that our architecture provides superior overload protection, since our gateway tier shields the servers from load surges. Our technique applies control only at the entrance to the system, while their technique requires control over both routing to, and scheduling within, all machines on all tiers. Moreover, we implemented our design and validated it using experimental data.

III. System Architecture and Implementation

In this section, we present the architecture and prototype implementation of our platform for managing the response times of web applications. Our platform allows a service provider to specify a policy that divides all possible requests for web applications into service classes and assigns a performance goal to each service class. A performance goal describes a target response time (either average or percentile), and includes an importance level that specifies the relative importance of meeting, exceeding, or failing to meet the target. We use the performance targets and importance levels to decide how to allocate resources to requests.

Using a management console, the service provider configures the goal of each service class and associates web requests with service classes. The prototype described in this paper uses URI patterns to map requests to service classes. Our prototype supports goals defined as average response times or percentile response times. We also support importance levels ranging from 1 to 99 (where 1 is the highest importance level). We could extend our platform to support classification based on a larger set of request parameters (for example user identities).

Figure 1 illustrates the overall system architecture using a prototypical J2EE application scenario. We show three applications A, B and C deployed across two tiers (Application Tier 1 and Application Tier 2). For example, in the case of a J2EE application the first tier hosts servlets and EJBs while the second tier hosts the database. In this example, the first tier uses three different machines while the second uses two machines. In this paper, we use the term *Node* when referring to a machine. In a given tier, a particular application may be available on multiple nodes. We refer to the presence of an application on a node as an application *instance*. In Fig. 1 application A has three instances on the first tier and one instance on the second tier. The first tier has a total of six application instances: three for application A, two for application B, and one for application C. In particular, the three instances of application A on Tier 1 are three application server processes (one on each machine of the tier), and those three application instances are grouped together in a cluster. However, we do not really care about the details of server processes and clusters; what is important is which application is available on which node. We presume some load balancing technique is used to spread the load among the instances of a given application in a given tier. In our experience, a large enterprise typically has several web applications deployed in an environment with a few tens to several hundred application instances spread across a few to a hundred nodes. Typically customers consider isolation, availability, and demand to decide the placement of application instances. Therefore, like in our example, we need

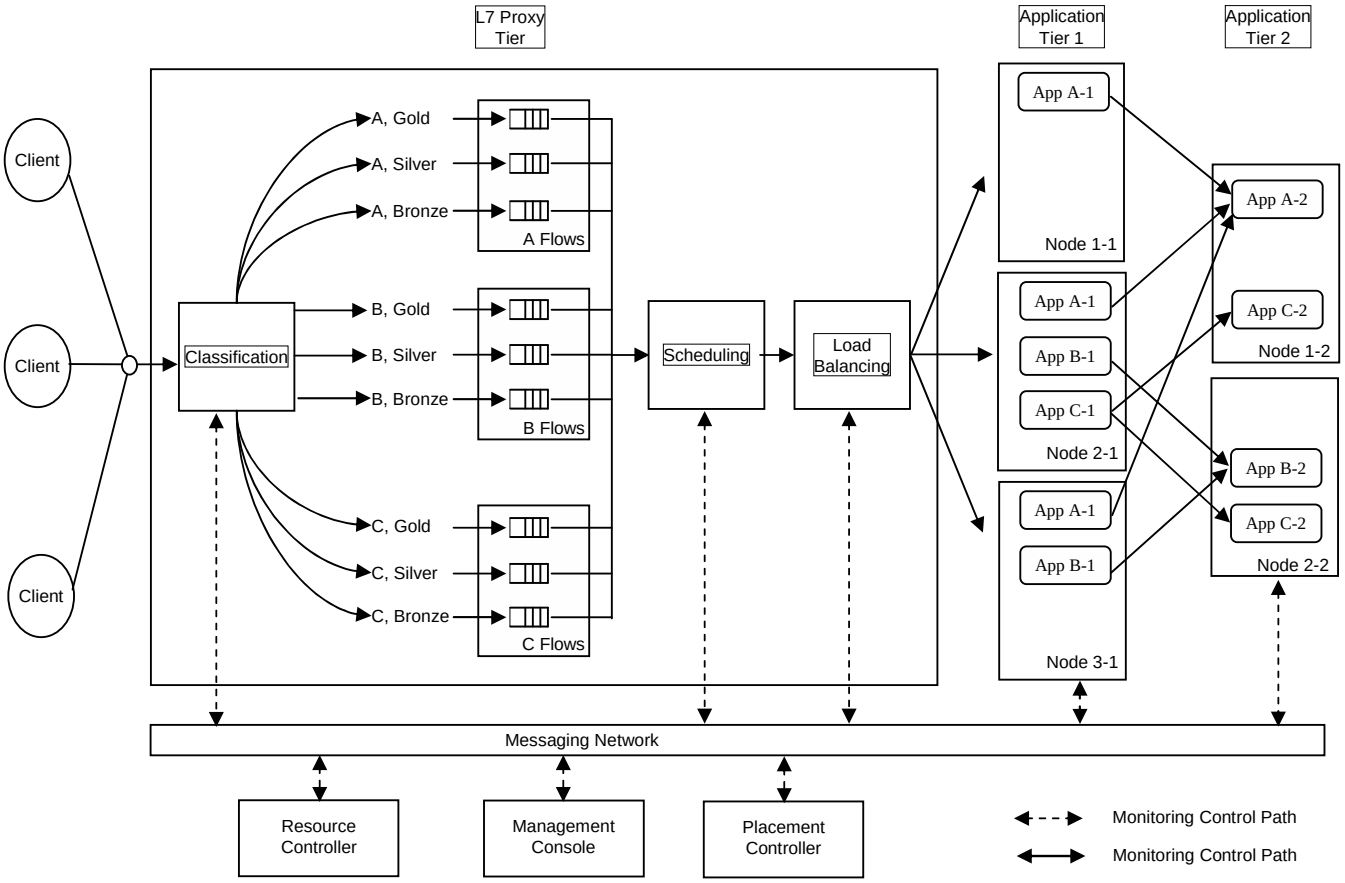


Fig. 1. System overview

to support a scenario where customers: replicate different applications to different degrees, use nodes with different computing capacities, and map URI patterns from different applications onto the same service class.

Our system primarily reacts to the application placement currently in effect, but also computes signals, called *resource requests*, that can guide the dynamic adjustment of the application placement. By application placement we mean the decision of how many replicas to run for each application, and which server machine should run each replica. We do not suppose that a single machine can run a replica for each application at the same time; that would require more memory, and perhaps more CPU switching, than is desirable. We do not even suppose that each application could potentially run on each machine; for example, an application may have hardware requirements that are met by only some machines. Our system takes as given a set of allowed placements, described simply in terms of which application could run on which machine. Our system computes a resource request for each application. A resource request is an amount of computing capacity. The computed set of resource requests can be met by an allowed placement, and would produce a nearly optimal performance result compared with other allowed placements (it may fall short of optimal because we use a heuristic search technique).

To manage the response times of web requests we introduce an additional tier, the *Proxy tier*. The proxy adds layer-7 mechanisms that divide and control the flow of requests into the application tiers. The proxy is given values for its resource allocation parameters and controls the request flows. Our platform has a *resource controller* that monitors the request response times and other performance metrics, and periodically recomputes the resource allocation parameter values that the proxy uses. The components of the proposed platform share monitoring and control information via a messaging network, which uses a publish/subscribe paradigm [25].

A. Proxy layer: Controlling the request flow

The proxy implements three main resource control mechanisms: *classification*, *queueing and scheduling*, and *load balancing*. When a request arrives at the proxy the classification mechanism examines the request attributes and classifies the request according to the user defined policy. The classification process produces a tuple $\langle d, c \rangle$ that defines the entry application and service class associated with the

request and is referred to as a *traffic class*. The proxy has a request queue for each traffic class. A classified request is placed at the end of the queue associated with its traffic class and its execution is suspended. As an example, in Fig. 1 we show a proxy configured with three service classes ($Gold$, $Silver$, and $Bronze$) and with three applications (A, B, and C). This makes nine request queues.

Note that we have designed a separate queue for each traffic class even though multiple traffic classes may belong to the same service class and thus share a performance goal. We have chosen to split a service class into traffic classes as separate management units, because the maximum amount of resources available for a request of one service class may differ depending on the application involved. For example, in Fig 1 application A has three instances in the first tier while application C has only one instance deployed in the same tier. Therefore, the maximum resource capacity available to application A is bigger than that available to application C. In addition, resource requirements of requests belonging to different applications may vary. Thus, to prevent the system overload, in addition to controlling the overall resource consumption in each tier, we must control the amount of resources consumed in each tier per application.

To implement the overload control, we logically divide the queues into *gateways*, where a gateway is a set of queues corresponding to a given application. Thus, in the example shown in Fig. 1, three gateways exist. We use the same symbol, d, to represent both a gateway and the corrresponding application. With each gateway, we associate two kinds of resource allocation parameter: (i) the maximum number of *concurrent* requests (counted at the proxy, not at deeper places where the number may be different) that the application tiers may execute on behalf of this gateway (for gateway d we denote this parameter with N_d) and (ii) the round robin scheduling weight for each traffic class (for gateway d and class c we denote this parameter by $\omega_{d,c}$).

Each gateway has a scheduling mechanism that decides when to remove a request from a queue and forward the request for execution on the application tiers. The scheduler divides the application's resources among its service classes. By dynamically changing the amount of resources allocated to each class we can control the response time experienced by the web requests in each class. A gateway's scheduler controls the request flows of all the service classes belonging to the gateway's application. A gateway's scheduler tracks the number of outstanding requests for its application and makes sure that there are at most N_d requests executing concurrently. When the number of concurrently outstanding requests from gateway d is smaller than N_d, and some requests are queued in d, the scheduler for d selects a new request for execution. That scheduler uses a weighted round robin scheme.

After selecting a request for execution, the scheduler passes the request to a load balancer mechanism that selects a node and sends the request to it. The load balancer spreads the load across the multiple deployed instances of an application. The load balancer tries to equalize response time across all the instances of a given application while respecting request affinities.

When a request completes its execution and the response returns from the application tiers, the relevant scheduler 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.

Each scheduler collects statistics on arrival rates, service rates, queueing times, and execution times. Each scheduler periodically broadcasts this data on the messaging network.

B. Resource Controller

The resource controller implements the logic that computes (a) the amount of resources to allocate to each pair of service class and application, and (b) the resource requests that may be used to drive placement changes.

Fig. 2 shows the internal structure of the resource controller and its inputs and outputs. The controller computes N_d values, the maximum number of concurrent requests that can be simultaneously executing for each application d on the application tiers. It also computes $\omega_{d,c}$ values, the round robin scheduling weight of the queue that is associated with each traffic class $\langle d, c \rangle$. The controller runs periodically and computes the resource allocation parameters, the N_d and $\omega_{d,c}$, that each gateway will use during the control interval $i + 1$, using the request execution and server utilization statistics measured during interval i.

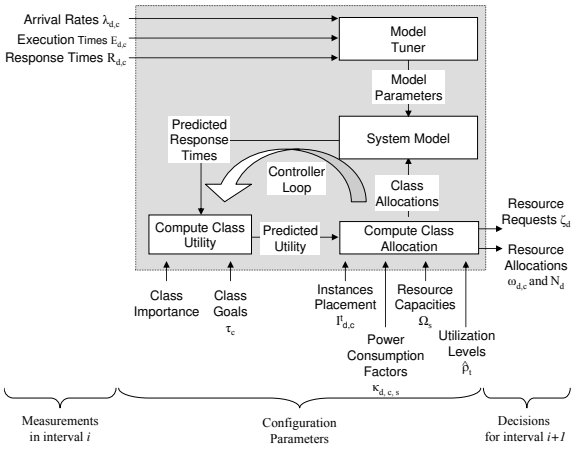


Fig. 2. Resource controller inputs and outputs

The size of the control interval affects the ability of the controller to respond to rapid changes in traffic load or response time. On one hand, with a small interval, the resource allocation parameters are updated frequently which makes the system more adaptive. On the other hand, a larger interval increases the stability of the system.

The resource controller uses an independent queuing-network model for each traffic class; the model is described in Section V. Using the queuing model for traffic class $\langle d, c\rangle$, the resource controller predicts the response time for this traffic class, $R_{d,c}$, under any given resource allocation represented by N_d and $\omega_{d,c}$. The resource controller also uses a per-class utility function of response time, $U_c(R)$ that expresses the business value of achieving any given performance for this class. During control interval i, the controller receives for each traffic class $\langle d, c\rangle$ the measured request arrival rate $\lambda_{d,c}$, response time $R_{d,c}$, and execution time $E_{d,c}$. The controller first uses these parameters, along with the allocation decision applied to interval i, to tune a queuing model of the trafic class $\langle d, c\rangle$. Then, using the tuned models the resource controller computes vectors of allocation parameters N_d and $\omega_{d,c}$ that equalize the utility of all traffic classes.

The controler also has a parameter $\hat{\rho}_t$ through which the service provider specifies a limit on the utilization of the bottleneck resource in tier t.

When exploring the space of possible allocations the controller uses a set of constraints to avoid overloading the resources in the application tiers. In this paper we assume that (a) there is one bottleneck resource in each application tier, and (b) that resource has a well-defined capacity that does not vary from application to application. Let t be a tier that includes node s. We use Ω_s to denote the capacity of the bottleneck resource on node s. We define a bottleneck power consumption factor, $\kappa_{d,c,s}$, that represents the amount of the bottleneck resource that is consumed on node s by a system-entering request of traffic class $\langle d, c\rangle$, averaged over the time that request spends among the application tiers, while the average bottleneck resource utilization on each relevant node is at the limit ($\hat{\rho}$) for its tier. For example, if Ω_s represents the CPU capacity of node s, in cycles per second, then $\kappa_{d,c,s}$ represents the amount of CPU, in cycles per second, that one request from traffic class $\langle d, c\rangle$ consumes on node s.

The controller must also know on which nodes in each tier the applications are deployed. In tier t, the relationship between the applications and nodes is given by the application *placement matrix* I^t. For a given tier, the rows of the matrix represent applications and the columns represent nodes in that tier. Thus for tier t, $I^t_{d,s} = 1$ only if an instance of application d is deployed on node s, otherwise $I^t_{d,s} = 0$. When exploring the space of possible allocation vectors the controller must observe per-application resource-capacity limits resulting from application placement I^t in each tier t.

After the controller computes a new set of N_d and $\omega_{d,c}$ values, it broadcasts them on the messaging network. Upon receiving the new resource allocation parameters, each gateway switches to use the new values.

Our system works under the assumption that the execution time of requests does not change greatly from one control cycle to the next. To help make this true, our system attempts to limit the utilization of the computational resources to a range over which the execution times vary relatively little. For example, execution times vary much less for CPU utilizations from 0% to 90% than for CPU utilizations that range up to 100%.

We use the approach of queueing the work before it enters the application tiers. We use power consumption factors ($\kappa_{d,c,s}$) and resource capacities (Ω_s) to decide how much work of each class the application tier can execute at any given time. We use a scheduler to decide how to split the application tier resources among the traffic classes. During period of low traffic the application tiers resources will be under utilized and there will be no queueing at each gateway. As the traffic demand increases some applications will reach their concurrency limits (N_d). At that time we will start queueing requests at the gateways and use the scheduling weights to decide which queued request should be forwarded to the application tiers when capacity becomes available.

For this system to work we must be able to compute the portion of resource s that each request of each traffic class consumes on average ($\kappa_{d,c,s}$). For the experiments reported in Sec VI we used an on-line profiling technique to compute $\kappa_{d,c,s}$. In the next subsection we show how we can define these parameters.

C. Power Consumption Factor

Let's focus on one application tier and let's assume that the node CPU is the bottleneck resource tier.

We want to describe the amount of CPU consumed by requests of a given type, i.e., the *power consumption factor* at which a given type of request uses the CPU cycles, measured in clock cycles per second, on a node in this tier, when the utilization there, and at every other node that has a performance interation with that one, is at the configured limit. If disk access rather than CPU is the bottleneck, we would express the *power consumption factor* and the resource capacity in bytes per second. Thus, request packing can be accomplished by the resource controller in the same manner, for all tiers, independent of the nature of the bottleneck resource.

Figure 3 shows how to define the power consumption factor for tier t, under the assumption that there is no parallelism in the processing of any one particular request. Consider a request of traffic class $\langle d, c\rangle$; that means it (1) has service class c and (2) is for application d. In the course of processing this request, application d may itself make requests on deeper tiers, and so on. Consider all the processing done on behalf of that original $\langle d, c\rangle$ request, in relation to some given node s in some tier t. The total average time spent among the application tiers can be divided into four segments: (a) the average time spent in upstream application tiers ($\theta^{(0)}_{d,c,s}$); (b) the average time spent using the

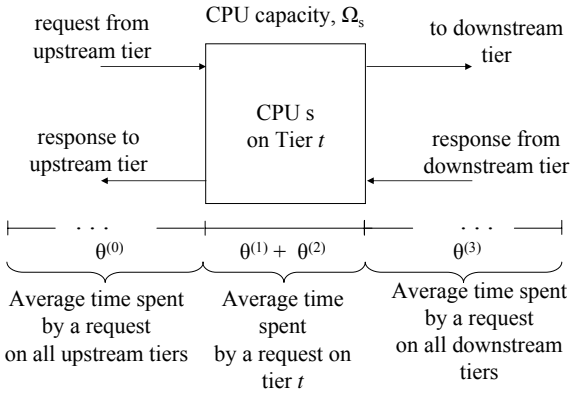


Fig. 3. Multi-tiered request processing

CPU on tier t ($\theta^{(1)}_{d,c,s}$); (c) average time spent waiting for the CPU on tier t because of resource contention ($\theta^{(2)}_{d,c,s}$); (d) the average time the request spends waiting for responses from downstream tiers ($\theta^{(3)}_{d,c,s}$). In particular, $\theta^{(0)}_{d,c,s}$, $\theta^{(1)}_{d,c,s}$, $\theta^{(2)}_{d,c,s}$ and $\theta^{(3)}_{d,c,s}$ are the values when the CPU utilization on node s at tier t is equal to $\hat{\rho}_s$, i.e., our target utilization for this resource; we similarly stipulate the CPU utilization of every other node on whose CPU utilization the response time in question (directly or indirectly) depends.

Here $\theta^{(1)}_{d,c,s}$ is the average time spent using the CPU on tier t for processing a single request admitted to the entry tier of the system. A single request admitted to the entry tier may result in multiple requests submitted to tier t. Therefore, $\theta^{(1)}_{d,c,s}$ can be expressed as the product $v\hat{\theta}^{(1)}_{d,c,s}$, where v is the number of visits to tier t induced by an entry tier request, and $\hat{\theta}^{(1)}_{d,c,s}$ is the average time spent using the CPU on tier t per visit to t.

We define the power consumption factor for node s at t for requests belonging to traffic class $\langle d, c\rangle$ as follows:

$$\kappa_{d,c,s} \quad = \quad \frac{\theta^{(1)}_{d,c,s}\Omega_s}{\theta^{(0)}_{d,c,s} + \theta^{(1)}_{d,c,s} + \theta^{(2)}_{d,c,s} + \theta^{(3)}_{d,c,s}} \qquad (1)$$

In other words, a request of this type consumes an average of $\theta^{(1)}_{d,c,s}\Omega_s$ CPU cycles on node s during an average period of time equal to $\theta^{(0)}_{d,c,s} + \theta^{(1)}_{d,c,s} + \theta^{(2)}_{d,c,s} + \theta^{(3)}_{d,c,s}$. This is easily extended to encompass parallelism in the processing of a request at the other tiers: the power consumption factor remains the quotient of the CPU cycles consumed on s divided by the time from entry to exit at the first application tier, provided the tier t processing is all done on s. As always, we must use the times that obtain when the utilization of every relevant node is at its configured limit.

The above approach assumes that the average power consumption factors do not vary over time. This assumption may be too restrictive when the power consumption factor depends heavily on the data associated with the request. In our experiments we have used an on-line estimation technique that recomputes $\kappa_{d,c,s}$ by periodically estimating $\theta^{(1)}_{d,c,s}$, $\theta^{(2)}_{d,c,s}$ and $\theta^{(3)}_{d,c,s}$ for the one tier of interest (where $\theta^{(0)}_{d,c,s}$ is necessarily 0).

In this paper we use $\kappa_{d,c,s}$ to denote the power consumption factor of requests belonging to traffic class $\langle d, c\rangle$ on server node s, when every relevant node is loaded to the desired utilization point. We also assume that each node is dedicated to a single tier.

D. Estimating Power Consumption Factor

A power consumption factor $\kappa_{d,c,s}$ characterizes its traffic class and the way it is served. When these do not change, off-line profiling can be used to experimentally determine the power consumption factors. To track changes, an on-line technique is needed. We have designed and implemented such a technique, and used it in the experimental work reported in this paper. The full details of on-line estimation of power consumption factors are beyond the scope of this paper. Following is a brief outline.

We estimate power consumption factors by two stages of computation. First we estimate *work factors*, which describe the total amount of computational work involved in serving requests, and then we estimate how that work will be spread out over time. A work factor $\alpha_{d,c,t}$ is the total amount of computational work on tier t required to serve a request of traffic class $\langle d, c\rangle$. We equate computational work with CPU usage. In the simplified situation discussed earlier, $\alpha_{d,c,t}$ is $\theta^{(1)}_{d,c,s}\Omega_s$ (we presume this product is the same for all servers s in tier t). We estimate work factors by fitting observations of request throughput and CPU utilization to a simple linear model. The model is

$$\rho_s\Omega_s = \sum_{d,c} \alpha_{d,c,t(s)}\, \lambda_{d,c,s} \qquad (2)$$

where ρ_s is the observed CPU utilization fraction of server s, Ω_s is the CPU power of server s, and $t(s)$ is the tier of server s. Observations of different servers within a tier may be combined to give more data against which to fit. Even so, there are challenges concerning noisiness and lack of throughput diversity in the data.

The second stage of estimating a power consumption factor is to (a) estimate what the entry-level service time would be if the CPU utilization of each relevant node were at the configured limit, and (b) divide the relevant work factor by that service time (as in equation (1)). The entry-level service time for a traffic class $\langle d, c\rangle$ is the time from (a) entry to the first application tier (application d) to (b) the response from that tier. We can observe what those entry-level service times are, but the CPU utilizations are not necessarily at the configured limits. We have a technique for estimating the desired entry-level service times, under the assumption that only one node's CPU utilization is relevant. It is based on a

simple queuing model that relates CPU utilization to service times by describing competition for the server's CPU(s) and the (presumedly constant) time spent waiting for work to be done by deeper tiers. We fit observed performance metrics to the model to extract the values of the model's parameters. Then we use the parameterized model to extrapolate the service times that would obtain if the server's load were high enough to cause the prescribed level of CPU utilization.

IV. Computing Optimal Server Allocation

In this section we describe the computation that the resource controller performs in order to find an optimal set of request concurrency values and scheduling weights. In Section IV-A we formulate a constrained resource allocation problem to solve for $N_{d,c,s}$, the allocated number of concurrent requests on server s for requests targeting application d with service class c. The solution technique is described in Section IV-B, where we use an incremental algorithm with heuristic bin packing. The utility function, which expresses the business importance of achieving the target performance goals, is given in Section IV-C. From the results of the resource allocation problem (the $N_{d,c,s}$ values) we then derive, as shown in Section IV-D, both (a) the gateway resource allocation parameter values and (b) the resource requests to drive placement.

A. Problem Formulation

We formulate a constrained resource allocation problem. The objective of this optimization problem is to equalize the utility of all traffic classes, as opposed to the more common objective of maximizing the sum of the utilities. Equalizing utilities results in fair discrimination among service classes, and avoids potential starvation of some of the classes, especially in situations of scarce resources. A utility function U_c is defined per service class c, since we define service goals and importance at the service class level. However, the utility of each traffic class $< d, c >$ appears as an independent term in the objective function of the optimization problem. Details of the utility function are described in Section IV-C.

We have two sets of constraints: server capacity and application placement. For each tier t, let $S(t)$ denote the set of servers belonging to that tier. We place a constraint that each of the servers in the set $S(t)$ is not loaded beyond a desired utilization level $\hat{\rho}_t$, for tier t servers. In other words, $\hat{\rho}_t$ is the maximum allowed utilization of the bottleneck resource on tier t, and it is a configurable parameter.

The current placement of application instances in tier t is represented by a placement matrix I^t defined in Section III-B.

To solve for the set of $N_{d,c}$ variables $\forall d, c$, we introduce the set of supplementary variables, $N_{d,c,s}$, $\forall d, c, s$, which represents the concurrency of traffic class $< d, c >$ on server s. Thus, we formulate the constrained resource allocation problem as maximizing

$$\min_{d,c} \quad U_c(R_{d,c}(N_{d,c})) \tag{3}$$

subject to

$$\forall t, s \epsilon S(t) : \sum_{d,c} \kappa_{d,c,s} N_{d,c,s} \quad \leq \quad \hat{\rho}_t \Omega_s \tag{4}$$

$$\forall t, d, c, s \in S(t) : \quad (I^t_{d,s} = 0) \quad \Rightarrow \quad (N_{d,c,s} = 0) \tag{5}$$

$$\forall t, d, c : \quad \sum_{s \epsilon S(t)} N_{d,c,s} \quad = \quad N_{d,c} \tag{6}$$

$$\forall d, c, s : \quad N_{d,c,s} \quad \geq \quad 0 \tag{7}$$

The function $R_{d,c}(N_{d,c})$, or simply $R_{d,c}$, gives the predicted response time of traffic class $< d, c >$ when allocated a concurrency value $N_{d,c}$. $R_{d,c}$ depends on the definition of the service goal for class c. Our system allows for average response time goals, as well as percentile response time goals. $R_{d,c}$ is computed by solving a single class queueing model, as will be detailed in Section V. Ideally, a multi-class model should be used in which $R_{d,c}$ is computed as a function of all the allocations given to all the classes. However, in order to simplify the problem and make it separable, we assume that $R_{d,c}$ depends only on the allocation $N_{d,c}$.

The first constraint ensures that the outcome of the resource allocation problem respects the capacity and desired utilization limit of individual servers. The second constraint is the placement constraint. It enforces the condition of not allocating any fraction of the resources of a server to traffic classes the target application of which is not deployed on that server. $N_{d,c}$ is the concurrency limit allocated to traffic class $< d, c >$. The third constraint limits it by the minimum allocated concurrency on the bottleneck tier for that traffic class. The last constraint bounds the solution to be non-negative.

B. Solution Technique

The optimization problem is a resource allocation problem with bin packing constraints. We use an incremental algorithm for the resource allocation problem [26] and a heuristic approach to satisfy the bin packing constraints. The incremental algorithm requires the objective function to be separable and convex. In our case, we treat each traffic class $< d, c >$ independently, i.e. the response time for a class is not a function of the allocation of other classes. Though in practice there are interdependencies, we capture such interactions among classes indirectly through the values of service times as described in V. Further, the response time function $R_{d,c}$ is nonincreasing and the utility function U_c is decreasing, hence we satisfy the convexity requirement. Though more efficient algorithms for the resource allocation problem exist, we find the incremental algorithm attractive due to the simplicity of its implementation and the fact that it lends itself to employing heuristics to satisfy the bin packing

constraints. More precisely, let traffic class $< d, c >$ be the one to receive an additional allocation to $N_{d,c}$ at an incremental step. We need to choose a server s for the assignment of this additional allocation. Since the optimization problem is solved repeatedly at every control cycle, the computation complexity is a prime concern. Hence, we do not attempt to find a global optimal solution. Rather we employ a simple heuristic approach where we use two heuristics in order, and without backtracking. The first heuristic chooses an available, non-shared server where class $N_{d,c}$ is the only class assigned to it. This postpones sharing to a later point in the incremental algorithm. The second heuristic chooses an available server with the largest minimum available allocations to other classes after assigning class $< d, c >$ to it. This reduces the possibility of having to switch two or more classes among servers in order to allow for a future allocation. The results of using our incremental and heuristic solution technique will be described in section VI.

Variations to the above optimization problem are possible [26]. Lower and upper constraints on the allocations $N_{d,c}$ are handled using standard techniques. Also, a more fair allocation may be obtained by solving both the minmax and maxmin problems, then by identifying a solution that lies between the two solutions obtained.

C. The Class Utility Function

We use a utility function U_c to encapsulate the business importance of meeting or failing to meet the performance goals of class c. The utility function maps the performance actually experienced by web requests into a real number. The utility function for service class c is defined using two parameters: (1) the response time threshold τ_c, and (2) the importance z_c of that service class. We allow for average as well as percentile response time thresholds.

If τ_c represents an average response time goal, the instantaneous utility of a traffic class $< d, c >$ experiencing average response time $R_{d,c}$ is given by

$$U_c(R_{d,c}) = \begin{cases} \frac{\tau_c - R_{d,c}}{\tau_c} & \text{if} \quad R_{d,c} \leq \tau_c \\ \left(\frac{\tau_c - R_{d,c}}{\tau_c}\right) \cdot \left(\frac{100 - z_c}{99}\right) & \text{if} \quad R_{d,c} \geq \tau_c. \end{cases} \tag{8}$$

The utility value is bounded by one, and is always greater than or equal to zero as long as the system meets the response time goal of the traffic class. If the goal is violated, the utility function yields negative values. In the negative region, z_c scales the magnitude of loss in utility according to the importance of the service class. A service class with $z_c = 1$ has the highest importance, while a service class with $z_c = 99$ has the lowest importance, as the utility value never goes below zero even under extreme violation of the response time goal of the class.

D. Deriving Controller Outputs from Resource Allocations

After solving for the set of $N_{d,c,s}$ variables $\forall d, c, s$, the resource controller maps them to (a) the required gateway resource allocation parameter values and (b) the resource requests that may be used to drive dynamic placement. The required gateway resource allocation parameter values are the N_d and $\omega_{d,c}$ values used by the gateway responsible for managing the traffic for application d. N_d is the maximum number of concurrent requests that the server nodes may execute on behalf of this gateway, and $\omega_{d,c}$ is the weighted round robin scheduling weight for traffic class $< d, c >$. The parameter values are computed as follows.

$$\omega_{d,c} = \frac{N_{d,c}}{E_{d,c}} \tag{9}$$

$$N_d = \sum_c N_{d,c} \tag{10}$$

where $E_{d,c}$ is the average execution time for requests targeting traffic class $< d, c >$.

In other words, the total concurrency limit allocated for a gateway is the sum of allocated concurrency limits for all traffic classes belonging to the application served by this gateway. The scheduling weights are chosen in proportion to the anticipated request flow rates based on the given allocations, since $\lambda_{d,c} = \frac{N_{d,c}}{E_{d,c}}$, from Little's law.

We chose to enforce a total concurrency limit for each gateway, and have the traffic classes served by the gateway proportionally share this limit, as opposed to enforcing a concurrency limit per traffic class. This is in order to have a work conserving scheduler that allows for instantaneous sharing of unused allocated resources, in response to rapid traffic fluctuations, which may occur within a control cycle of the resource controller.

The resource request for application d is an amount ζ_d of computing capacity to allocate to that application. It is computed in two steps. The first step simply takes the product of the concurrency and the power consumption factor for the application:

$$\hat{\zeta}_d = N_d \, \kappa_d \tag{11}$$

where the power consumption factor for application d is a weighted average, weighted by throughput, of the power consumption factors for the various request flows of that application:

$$\kappa_d = \frac{\sum_{c,s} \lambda_{d,c,s} \, \kappa_{d,c,s}}{\sum_{c,s} \lambda_{d,c,s}} \ . \tag{12}$$

The second step does smoothing. For each application d, the controller produces a series of $\hat{\zeta}_d$ values, one per control cycle. At each control cycle the current series is reduced to a weighted average and approximate standard deviation, using a simple incremental technique. The weights decrease geometrically with age, which means the new average and approximate standard deviation can be computed in constant space (regardless of the length of the series). The resource

request ζ_d is the sum of the average and twice the approximate standard deviation. The resource requests may be used by a controller that runs on a significantly slower time scale, and the resource requests are inflated by an estimate of the variability to provide some "headroom" over the course of one of those longer control cycles.

V. Performance Modeling

Figure 4 illustrates the closed queueing network used to model the traffic arriving at a gateway for traffic class $< d, c >$. The $M_{d,c}$ clients represent the sources of requests. Each client goes through a cycle of sending a request, waiting for the response, and then generating the next request. The time to generate a request is referred to as the think time, which is assumed to be generally distributed with mean $Z_{d,c}$. Upon arrival to the system, a request is queued in a FCFS queue in the gateway, identified for traffic class $< d, c >$. Once dispatched, a request is served by one of the server nodes. The execution time of a request is assumed to be exponentially distributed with mean $E_{d,c}$. The number of servers in the model represents the concurrency limit $N_{d,c}$, hence allowing no more than $N_{d,c}$ requests to be served simulateously. The response time includes both the queueing time in the gateway and the execution time in the server nodes. The mean response time for traffic class $< d, c >$ is denoted by $R_{d,c}$.

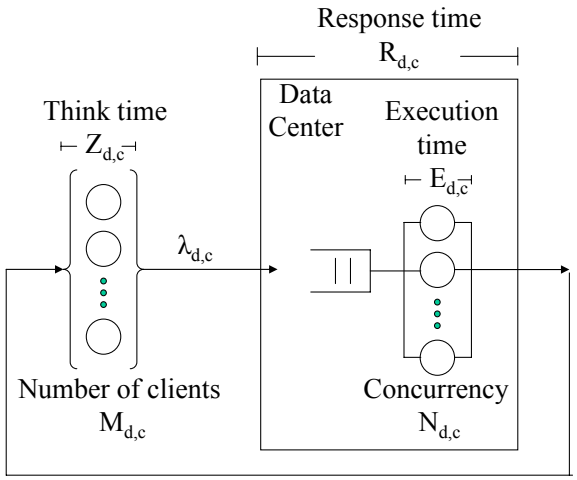


Fig. 4. Closed queueing network model for traffic class $< d, c >$

This closed queueing network model is also known as the machine repairmen model, where a set of $M_{d,c}$ machines breakdown, with mean up time $Z_{d,c}$, then get repaired by $N_{d,c}$ repairmen, with mean repair time $E_{d,c}$. A repairman works on one broken machine at a time. If all repairmen are busy, broken machines wait to be repaired by the first available repairman.

We model each traffic class independently using this closed queueing network model. The interaction among the various traffic classes, mainly due to competition on resources on the server nodes, manifests itself in the values of the execution times $E_{d,c}$. Thus, our independence approximation would work well in steady state, when changes in traffic and allocation are minimal. Otherwise, the values of $E_{d,c}$ will change over a few control cycles, until they reflect the interaction among the traffic classes.

We use the traffic class queueing model in two ways. First, we train the model using performance measurements to estimate the parameters of the model. Second, in the optimization, we analyze the model to obtain the mean response time $R_{d,c}$ as we change the concurrency $N_{d,c}$.

A. Model Parameter Estimation

The parameters of each traffic class closed queueing network model are estimated based on some performance measurements over a period of time. At a given concurrency, $N_{d,c}$, we measure the average response time, $R_{d,c}$, throughput, $\lambda_{d,c}$, and average execution time, $E_{d,c}$. Then, we find estimates for the number of clients, $M_{d,c}$, and the average think time between requests, $Z_{d,c}$, that would yield a average response time close to the measured value, $R_{d,c}$. One may formulate this problem as an optimization problem, where the objective function is the absolute error between the average response time obtained by the model and the measured average response time, and the variables are (1) the integer variable $M_{d,c}$ and (2) the continuous variable $Z_{d,c}$. Given that the average response time is nondecreasing in the first variable, $M_{d,c}$, and decreasing in the second variable, $Z_{d,c}$, there may be many values of ($M_{d,c}$, $Z_{d,c}$) which would result in the same measured average response time, $R_{d,c}$. We take the solution with the minimum $M_{d,c}$ that has an error in an acceptable range. The reason being that it is more efficient to solve the queueing network with a small number of clients. Hence, we decrease the solution time of the resource allocation optimization problem, where we need to analyze the queueing network model repeatedly.

B. Model Analysis

Given the above modeling assumptions, the closed queueing network has a product form solution, and therefore, it is straightforward to use the Mean Value Analysis (MVA) technique to compute the mean response time $R_{d,c}$ [27]. The mutiserver queue is treated as a single server queue with state-dependent service rate. Thus, the standard MVA technique has to be enhanced to recursively compute the queue length distribution.

Once the ($M_{d,c}$, $Z_{d,c}$) parameters are estimated, they are used, together with the measured $E_{d,c}$, to predict the response time corresponding to a given allocation, $N_{d,c}$. The optimizer repeatedly uses this prediction function, in the incremental algorithm [26], until an optimal choice of $N_{d,c}$ $\forall d, c$ is found, subject to placement and request packing constraints.

VI. Experimental Results

In this section, we study the behavior of our platform using a benchmark application and a synthetic load. We used the set up describe in Fig. 5 to run our experiments. The set up uses two applications deployed on a J2EE tier with three nodes and a database tier with one node.

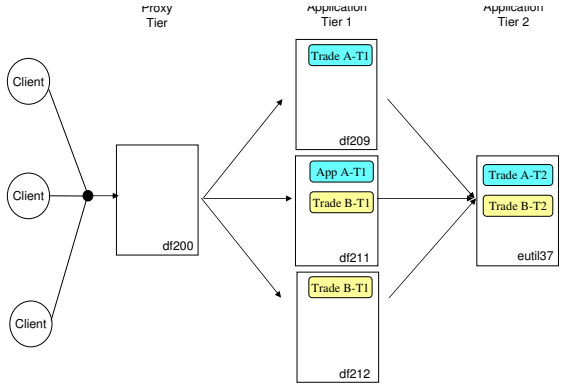


Fig. 5. Experimental topology

For our application we use 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 server implementation of J2EE 1.3 and Web Services. We deployed two different flavor of the Trade6 application on our experiment platform: TradeA and TradeB. 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 hieghest 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	Response Time Threshold	Importance	Application
Gold	350 ms	1	TradeA
Bronze	1,200 ms	99	TradeB

TABLE I
Service classes

For our experiments we used three nodes running the WebSphere J2EE platform and one node running DB2. Table II shows the resource configuration and the resource demand of the different classes. We used three machines with multiple CPUs in the J2EE tier. One of the machines, df211, has sufficient memory to run two application servers, while the other two have only enough memory for one application server. We therefore deployed both TradeA and Trade B on df211, while deploying only TradeA on df209 and only TradeB on df212. We used the node named eutil37 to run the database server. Table II shows that the nodes in the J2EE tier have similar CPU capacities. The node running the database server has a larger capacity. The table in Table II shows also the power consumption factors for the two traffic classes we used an for each resource (the only bottleneck resource in our experiments was the node's CPU). We used a node with capacity similar to df212 to run the proxy server that fronts the application tiers. Finally we used a set of machines to run the cient session emulators.

	df209	df211	df212	eutil37
κ_A	125 MHz	125 MHz	125 MHz	50 MHz
κ_B	300 MHz	300 MHz	300 MHz	75 MHz
Ω	1,490 MHz	1,490 MHz	1,490 MHz	8,000 MHz
Memory	0.75 GB	1 GB	0.75 GB	2 GB

TABLE II
Resource parameters

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 server capacities, (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 demands (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.

A. Overload protection

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 6 illustrates the number of requests permitted by the proxy to execute concurrently. We note that 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. While 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

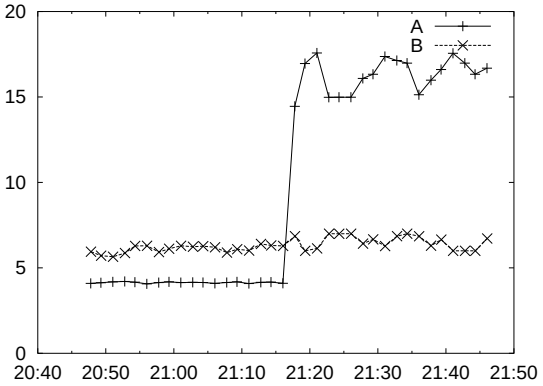


Fig. 6. Experiment 1, number of requests concurrently executing

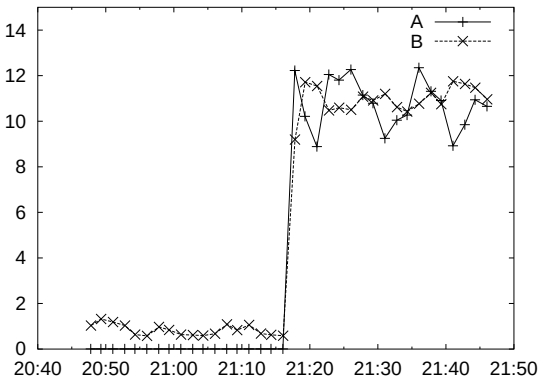


Fig. 7. Experiment 1 - Average queue length

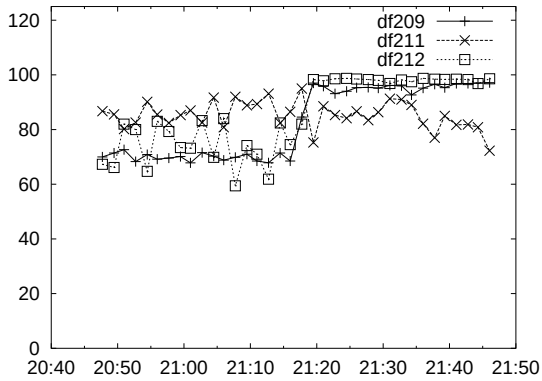


Fig. 8. Experiment 1 - CPU Utilization

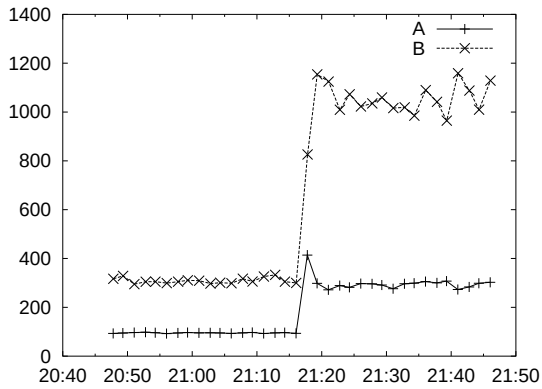


Fig. 9. Experiment 1 - Average response time

Figure 7 shows. This is due to the fact that κ_A is about half of κ_B. This is more evident when we observe that the CPU utilization levels in the two phases are almost equal, as shown in Figure 8. Thus, taking power consumption factors into account while making resource allocation decisions enabled us to prevent server overload in phase two of the experiment.

In phase two of the experiment, where the total offered load is above the system capacity, queueing takes place. Figures 9 and 10 show the average response time and throughput, respectively. Since the service goal 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 11 illustrates the concurrent requests executing on each server, for each of the two applications. As can be easily observed from the figure, TradeA requests always executed on servers df211 and df209, while TradeB requests always executed on df211 and df212. df211 was the only shared node, and its capacity 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 server 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 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, application placement, and available capacity. This is apparent from Figure 12 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 13, 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 14. 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

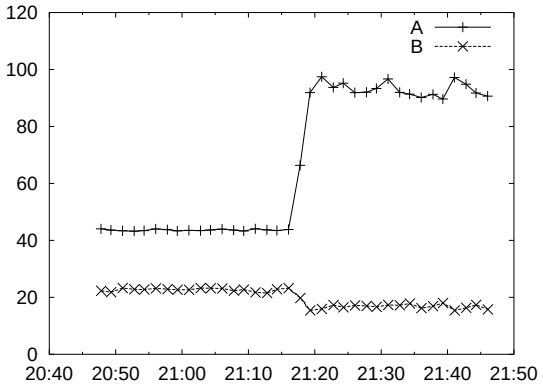


Fig. 10. Experiment 1 - Throughput

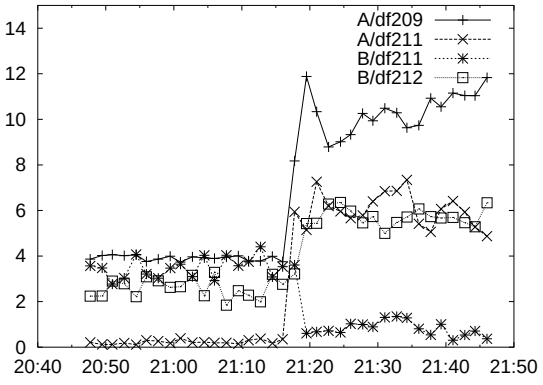


Fig. 11. Experiment 1 - Number of requests concurrently executing at each server node

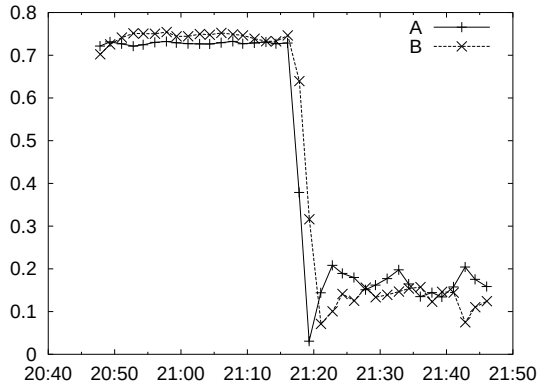


Fig. 12. Utility per flow, with control

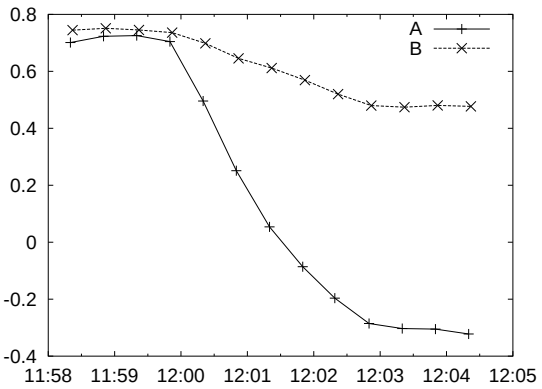


Fig. 13. Utility per flow, without control

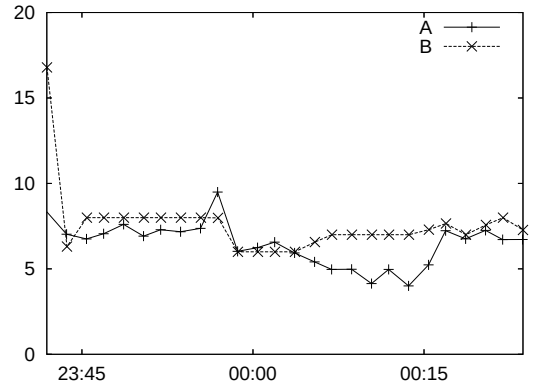
in Figure 9, the average response time for both TradeA and TradeB were slightly below target during the second phase of the experiment.

B. Resilience to placement changes

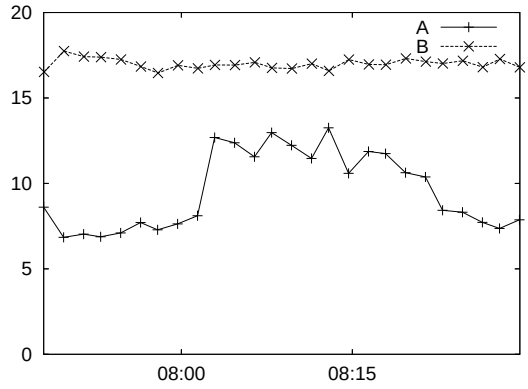
In this experiment, we start with the same placement of applications nodes as in the first experiment. However, after 15 minutes from the beginning of the experiment we emulate a server failure scenario by bringing one node (df209) down, and we observe how the system reacts to this bottleneck. After another 15 minutes, we bring node df209 back up, with only TradeA placed on it. We repeat the same scenario for an uncontrolled system (resource controller disabled) and compare the results of the 2 cases.

Figures 15 (a) and (b) show the number of requests that concurrently execute on backend servers in the case of the controlled and uncontrolled system, respectively. In the uncontrolled case, the number of executing requests is a function of service time, and increases as service time does. The controlled system reduces the number of executing requests by queuing them as shown in Figure 16. At 23:58 and 8:02 in the controlled and uncontrolled cases, respectively, we stop server on node df209. The controlled system reacts to the decreased capacity by increasing the rate of queuing for both flows. In the uncontrolled system, service time significantly increases due to system overload, hence the number of requests in the system also increases. At 00:13 and 8:17 in the controlled and uncontrolled cases, respectively, we restart df209, which causes the concurrency and queue length to return to the values seen in the first phase of the experiment.

In Figures 17 (a)–(f), we compare the performance of the managed flows in the controlled and uncontrolled systems side by side. Figures 17 (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 the server is brought down. In the controlled case, the impact of server failure 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 17 (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



(a) Controlled



(b) Uncontrolled

Fig. 15. Experiment 2 - Number of concurrently executing requests

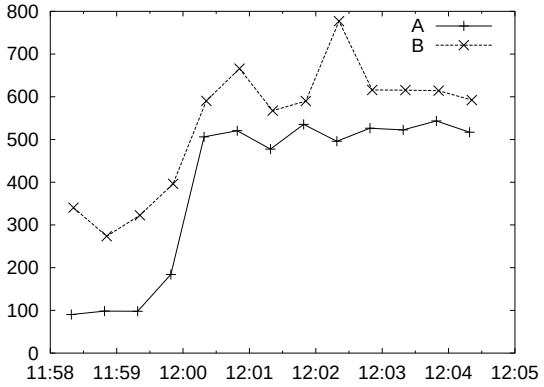


Fig. 14. Experiment 1 - Average response time, without control

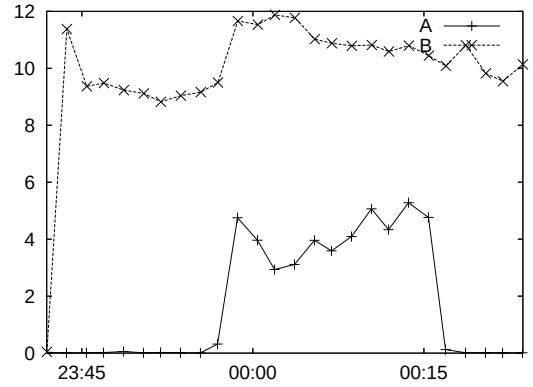


Fig. 16. Experiment 2 - Number of requests queued in the controlled system case

(Figure 17 (c)). While the server 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.

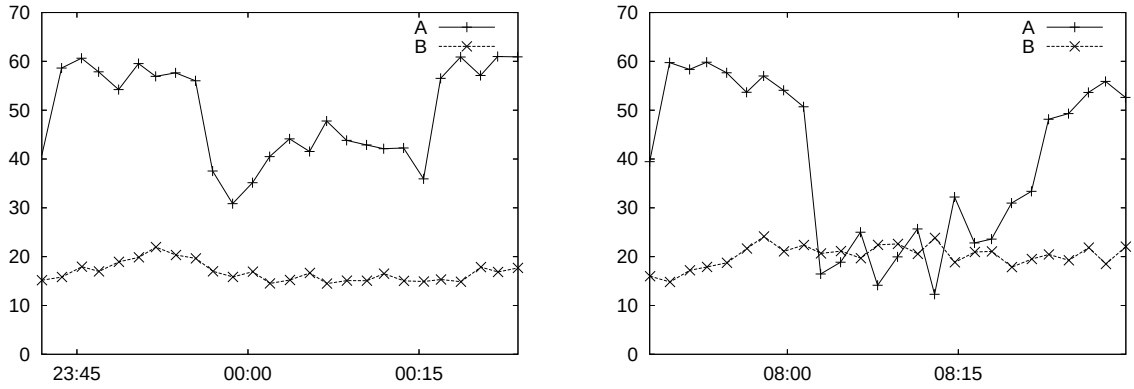
Recall, that the objective of our system is to equalize utilities among flows. In Figures 17 (e) and (f), we show that in the controlled case, we indeed manage to keep the values of the objective functions for TradeA and TradeB closer to each other as is the case in the uncontrolled system. Observe, that 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.

VII. Conclusions

We have presented the architecture and underlying model of a performance management system for multi-tiered web applications deployed on clustered web servers. The management system is transparent and allocates server resources dynamically in order to optimize the expected value of a system-wide utility function. The resource allocation problem is constrained by application placement constraints, and accounts for heterogeneity in server processing powers. The performance management system also computes resource requests that may be used to drive a dynamic placement management system.

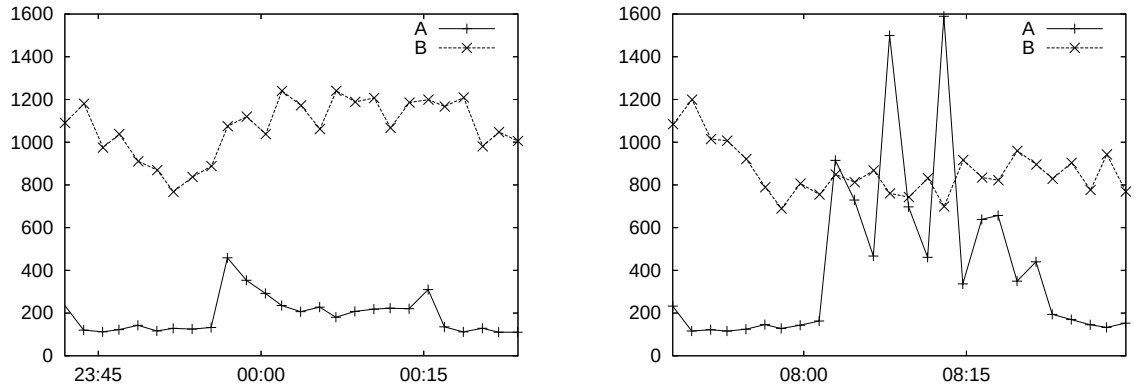
The architecture features gateways that implement local request queueing and scheduling mechanisms. A resource controller solves the optimization problem and tunes the parameters of the scheduling mechanisms. In this study, we have used a closed queuing network model to predict the response time of requests for different resource allocations.

We use the notion of power consumption factors per tier in order to allocate resources in a way that does not overload any server, beyond a desired utilization level. Server capacities and power consumption factors are expressed in the same units. Thus, the resource controller can address server capacity constraints while packing requests in a uniform way, which is independent of the nature of the bottleneck resource



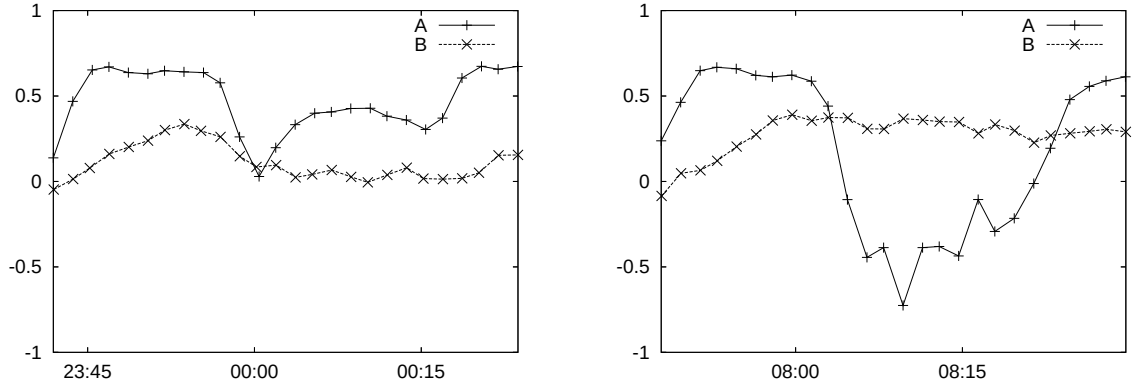
Throughput

(a) Controlled (b) Uncontrolled



Response time

(c) Controlled (d) Uncontrolled



Utility

(e) Controlled (f) Uncontrolled

Fig. 17. Experiment 2 - Comparing a controlled system to an uncontrolled one

at each tier.

Experimental results, using a full implementation of the proposed management system, showed that the control system converges quickly without significant oscillations, yielding service level differentiation and server overload protection based on set service goals and fluctuating offered loads.

REFERENCES

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

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

[3] M. Welsh and D. Culler, "Adaptive overload control for busy internet servers," in *Proceedings of the 4th USENIX Conference on Internet Technologies and Systems (USITS03)*, March 2003.

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

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

[6] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef, "Performance management for cluster based web services," in *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, (Colorado Springs, Colorado), March 2003.

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

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

[9] G. Lodi and F. Panzieri, "Qos-aware application server: preliminary design and implementation report," CTR technical report, University of Bologna, Italy, September 2004.

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

[11] D. Ardagna and L. Zhang, "SLA based profit optimization in autonomic computing systems," in *International Conference On Service Oriented Computing*, (New York, NY), pp. 173 – 182, November 2004.

[12] G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef, "Performance management for web services," IBM Research Technical Report RC22676, IBM T.J. Watson Research Center, Yorktown Heights, NY, August 2002.

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

[14] B. Urgaonkar and P. Shenoy, "Cataclysm: Policing extreme overloads in internet applications," in *Proceedings of the International World Wide Web Conference*, (Chiba, Japan), pp. 740 – 749, May 2005.

[15] M. Welsh and D. Culler, "Overload management as a fundamental service design primitive," in *Proceedings of the 10th ACM SIGOPS European Workshop*, (Saint-Emilion, France), September 2002.

[16] M. Welsh, D. Culler, and E. Brewer, "Seda: An arcitecture for well-contained, scalable internet services," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, (Banff, Canada), October 2001.

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

[18] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal, "Size-based scheduling to improve web performance," *ACM Transactions on Computer Systems*, vol. 21, pp. 207–233, May 2003.

[19] B. Schroeder and M. Harchol-Balter, "Web servers under overload: How scheduling can help," in *Proceedings of 18th International Teletraffic Congress*, (Berlin, Germany), September 2003.

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

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

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

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

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

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

[26] T. Ibraki and N. Katoh, *Resource Allocation Problems*. Massachusetts Institute of Technology, 1988.

[27] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance*. Prentice–Hall, 1984.