Quorum: Providing Flexible Quality of Service for Large-Scale Internet Services

Josep M. Blanquer, Antoni Batchelli, Klaus Schauser and Rich Wolski
Department of Computer Science, University of California Santa Barbara
{*blanquer,tbatchel,schauser,rich*}*@cs.ucsb.edu*

Abstract

In this paper we describe Quorum: a new non-invasive software approach to scalable quality-of-service provisioning that uses traffic shaping, admission control, and response monitoring at the borders of the Internet hosting site to ensure throughput and response time guarantees.

We compare Quorum both to over-provisioning and to Neptune – a research and now commercially successful middleware system that implements QoS for Internet services. Our results show that Quorum can enforce the same QoS guarantees as Neptune for Neptune-enabled services, but without requiring the additional development overhead required by Neptune. We also detail Quorum's flexibility by using it to provide QoS guarantees for an Internet service benchmark that is not ready to be used with Neptune or any other invasive software approach. In this case, Quorum achieves the same results a over-provisioning (which is the only other feasible solution) using one-half of the site resources.

1 Introduction

With the current importance of and dependence upon Internet services, it is imperative for companies relying on web-based technology to offer (and potentially guarantee) predictable, consistent, as well as differentiated quality of service to their consumers. For example, a search engine such as Google may want to guarantee a different service quality for the results served to America On-Line (AOL) than the quality it can guarantee to Stanford University searches. Internet services are commonly hosted using clustered architectures where a number of machines, rather than a single server, work together in a distributed and parallel manner to serve requests to all interested clients. Implementing service quality guarantees scalably in such a distributed setting is a difficult challenge.

Both research and commercial Internet service communities have explored hardware-based and software-based approaches to meeting this QoS challenge. A simple and effective hardware-based approach is to rely on an over-provisioning and physical partitioning of cluster nodes, each partition dedicated to a different class of service. Unfortunately, the necessity to handle large and unpredictable fluctuations in load cause these techniques to incur in potentially high cost (enough resource must be available in each partition to handle load spikes) and low resource utilization (the extra resources are idle between spikes). Moreover, such a static approach does not offer much flexibility in the event services are added or upgraded, or more problematically, QoS guarantees are changed. A change in conditions frequently requires hardware reconfiguration, which may be expensive and/or error prone.

As a result, software-based approaches have been suggested that embed the QoS logic into the internals of the operating system [7, 10, 30, 5], distributed middleware [24, 25, 33], or application code [2, 8, 32] running on the cluster. Operating System techniques have been shown to provide a tight control on the utilization of resources (e.g., disk bandwidth or processor usage) while techniques that are closer to the application layer are able to satisfy QoS requirements that are more important to the clients. However the majority of existing software approaches offer guarantees within the scope of a single machine or for an individual application, and fail to provide global service quality throughout the site. Most current Internet sites are composed of a myriad of different hardware and software platforms which are constantly evolving and changing. The largest drawback to software-based approaches is the high cost and complexity of reprogramming, maintaining, and extending the entirety of the complex software system such that it can

provide QoS guarantees for all hosted services.

In addition, the source code for many service components hosted at a site may not be available for proprietary reasons, making software reprogramming by the entity maintaining the site impossible. Since it is the site operator and not the author of the service software that must make and honor QoS guarantees, invasive software-based QoS approaches may not be feasible unless the site has access to the source code of all services it supports.

In this paper, we propose *Quorum*: a new non-invasive software approach that uses traffic shaping and admission control at the entrance to an Internet site, and monitors service at the exit, to ensure reliable QoS guarantees. We describe the Quorum architecture and the way in which our realization of it ensures both throughput and response time guarantees together for multiple classes of service. Our approach treats the cluster and the services it is hosting as an unknown "black-box" system and uses feedback-driven techniques to dynamically control how and when each of the requests from the clients must be forwarded into the cluster. Because the system uses only the observed request and response streams in its control algorithms, new services can be added, old ones upgraded, and resources reconfigured without engineering or re-engineering the necessary QoS mechanisms into the services themselves.

We compare Quorum both to over-provisioning and to Neptune [24, 25] – a research and now commercially successful middleware system that implements QoS for Internet services, but which requires the services themselves to be re-written to use Neptune primitives. Using the *Teoma* [26] search engine, which is built on top of Neptune, we show that Quorum can enforce the same QoS guarantees as Neptune for Neptune-enabled services, but without requiring the additional engineering overhead associated with modifying the services that it supports. We also detail Quorum's flexibility by using it to provide QoS guarantees for an Internet service benchmark that cannot be modified and, thus, cannot be used with Neptune.

As such, we demonstrate Quorum's ability to achieve the same level of performance as one of the best and most successful software approaches, while providing the ability to support a wider range of Internet services with a greater degree of flexibility. At the same time, because it is a software-only approach, Quorum does not require hardware reconfiguration when the QoS guarantees made by the site operator change. Quorum improves the flexibility with which scalable Internet services can be hosted while avoiding the resource waste associated with over-provisioning.

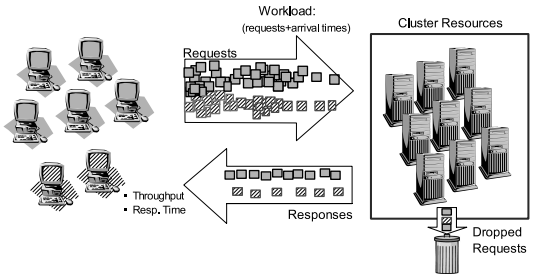

Figure 1: System Model for Internet Services

1.1 Contributions

This paper makes four main contributions:

- We present Quorum as a novel approach to QoS provisioning for large-scale Internet services that uses only observed input request and output response streams to control the load within the site so that quality guarantees are met.
- We build a prototype implementation and demonstrate its viability using a large cluster system hosting commercial and community benchmark services.
- We compare the efficiency Quorum with the best state-of-the-practice approaches in the cases where comparisons are feasible.
- We demonstrate that the flexibility provided by Quorum enables more efficient Internet service deployments than can currently be supported by existing approaches.

The remainder of this paper is organized as follows. Section 2 briefly describes the QoS model we use to illustrate Quorum and evaluate it with respect to competitive approaches. In Section 3 we detail the Quorum architecture. Section 4 demonstrates, using a contrived example, the features and capabilities of Quorum that are relevant to the performance evaluation presented in Section 5. In Section 6 we discuss related work, and we conclude in Section 7.

2 The QoS Model and Evaluation Criteria

We model Internet services (Figure 1) as a stream of requests coming from clients that are received at the entrance of the site, processed by the internal resources, and returned back to the clients upon completion. In the case of system overload or internal error condition, requests can be dropped before completion and thus may not be returned to the client. Requests can be classified

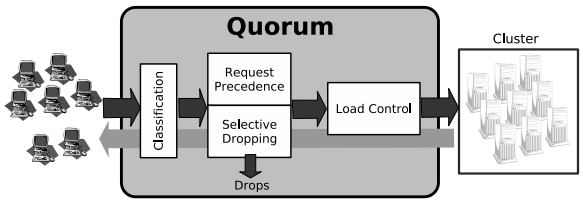

Figure 2: The architecture of Quorum

or grouped into different *service classes* according to a combination of service (e.g., URL, protocol, port, etc.) and client identity (e.g., source IP address, cookies, etc.)

We view the QoS challenge as the ability to guarantee, at all times, a predefined quality for each service class as measured at the output of the cluster. A particular quality for a service class is guaranteed only under certain restrictions on its incoming workload. Thus, we define a *QoS class* as a set of output guarantees (throughput *and* response times) in terms of the input workload restrictions (compute requirements) of a particular service class. For example, a QoS class for a typical e-commerce site could specify a minimum throughput guarantee of 200 requests per second (req/sec) and a maximum response time of 500 ms, given that the resource consumption for calculating a single incoming request requires an average of 10 ms. Notice that the guarantee includes both throughput and service time requirements. Often it is possible to meet one QoS demand at the expense of the other. A good solution can accommodate both.

In addition to ensuring quality guarantees, any solution to the QoS problem should also have other desirable properties. A solution should achieve good resource utilization (i.e. be efficient), support QoS for a a broad range of hosted services, tolerate varying workloads and resource changes, and degrade gracefully when overloaded. These properties, while not strictly necessary to implement QoS, make give a solution possessing them greater utility, particularly in a commercial setting. Flexibility with respect to service hosting, for example, allows e-commerce sites to change, upgrade, and extend their services rapidly in response to market and/or customer demands. While difficult to quantify, ease of deployment and maintenance certainly adds practical utility. Thus we consider both the quantitative capabilities of the QoS methodologies we investigate as well as the qualitative properties that make them practically viable.

3 The Quorum Architecture

Our approach to implementing QoS for Internet services uses a single policy-enforcement engine to intercept and control in-bound traffic at the entrance of the site (typically a cluster of machines) hosting the services. By tracking the responses to requests that are served within the site, our system – called Quorum – automatically determines when new requests can be allowed entry such that a specified set of QoS guarantees will be enforced. The desired QoS policy is specified to the Quorum engine in the form of definitions for different QoS classes of service. Each definition must

- specify how the engine can identify the QoS class associated with each request, and
- the capacity and response time guarantees that must be maintained.

Rather than relying on instrumentation of the services and resources within the site, Quorum continually observes the relative request and response rates for each QoS class and reacts adaptively to changes in input request frequency, internal service compute times, etc. As such, it permits new levels of flexibility. New services can be added, existing services can be upgraded, cluster resources can be changed, added, decomissioned, etc. and Quorum automatically adjusts its behavior to the new conditions so that the specified QoS guarantees can be met.

Figure 2 depicts the architecture of Quorum, consisting of four different modules each of which implements a functionality that is necessary to support QoS. The *Classification* module categorizes the intercepted requests from the clients into one of the service classes defined in the QoS class. The *Load Control* module determines the pace (for the entire system and all client request streams) in which Quorum releases requests into the cluster. The *Request Precedence* module dictates the proportions in which requests of different classes are released to the cluster. The *Selective Dropping* module drops requests of a service class to avoid work accumulation that would cause a QoS violation and maintains responsiveness when the incoming service demands for that class exceed the processing capacity that has guaranteed. In the next sections we detail further the implementation of the Quorum modules. We explicitly exclude the details associated with Classification since it is a well understood problem that has already been studied in the literature [18] and the implementation we use does not extend the state-of-the-art.

3.1 Load Control

The functionality of the Load Control module is twofold. First, it maintains the resources within the cluster at a high level of utilization to achieve good system

performance. Second, it prevents large amounts of incoming traffic from overloading the internal resources of the cluster. The goal of the Load Control module is to have the cluster operate at maximum capacity so that the largest possible capacity guarantees can be met, but to prevent overload conditions that would cause response time guarantees to be violated.

Our implementation exploits the direct correlation between the amount of work accumulation and the time it takes for requests to be computed in any general purpose operating system. In general, higher parallelism corresponds to longer compute times for each service given a fixed amount of limited resources. With this in mind, the Load Control module externally controls the load in the cluster by appropriately forwarding or holding the traffic received from the clients, according to current performance metrics measured at the output of the cluster. By controlling how much traffic is accumulated in the cluster at any time, this module can directly affect the amount of time that requests take to be computed inside the cluster (i.e. compute time).

Similar to TCP, our implementation uses a sliding window scheme that defines the maximum number of requests that can be outstanding at any time. The basic operation of the Quorum engine consists of successively incrementing the size of the window until the compute times of the QoS class with the most restrictive response times approaches the limits defined by its guarantees. In our implementation, the algorithm increments the window until it observes a maximum compute time that is half the most restrictive of all the guarantees. The choice of 'half' is a compromise between maintaining the cluster occupied and running out of queuing space to absorb peaks of traffic. We are currently working on an optimized version that can dynamically adapt the threshold to allow longer queuing space without hurting the overall system performance.

3.2 Request Precedence

The function of Request Precedence is to partition virtually the cluster resources among each of the service classes. Capacity isolation is a necessary functionality that allows each service class to enjoy a minimum guaranteed capacity, independent of potential overload or misbehavior of others. This module is able to partition externally the service delivered by the cluster, by controlling the proportions in which the input traffic for each class is forwarded to the internal resources.

In addition to allocating a guaranteed capacity for each class, the Request Precedence module also reassigns unclaimed capacity to other QoS classes that have input demands exceeding their output guarantees. This functionality allows the QoS engine to take full advantage of the available cluster resources when the input load from one or more QoS classes can be serviced with fewer than the maximum number of resources necessary to meet the associated guarantees. In this way Quorum differs from an approach that relies on physical partitioning of the resources where temporary reassignment cannot be implemented dynamically.

Under Quorum, Request Precedence is implemented through the use of modified Weighted Fair Queuing [14, 23, 11] techniques that function at the request level. By factoring the input restrictions of each class into the fair queuing weights, Quorum transforms throughput guarantees into capacity guarantees. Capacity is a fungible metric that links output throughput and input restrictions such that an increment of one results in a decrement of the other. For example a capacity of 4000 ms/s corresponds to 400 req/s at a compute cost of 10ms/req, but also to 800req/s if the compute cost is only 5ms/req. Using this fungible capacity, the request precedence safely protects the service classes even when one or more of them violate the input restrictions specified in the QoS class. At the same time, the module automatically reassigns all of the surplus capacity in a way that is proportional to the capacities of the needed QoS classes.

3.3 Selective Dropping

The function of Selective Dropping is to discard the excessive traffic received for a QoS class in the situations where there is not enough available capacity to fulfill its incoming demands. A dropping module is necessary to prevent large delays from occurring in overloaded situations for the requests that are to be served. That is, the QoS guarantees will be observed for all requests that are serviced, but if input load exceeds the maximum level that can be supported for the given guarantee, some requests will be dropped. Our Selective Dropping implementation ensures that the guarantees will be met for all requests that can be serviced. It does so by independently observing each of the QoS queues of the engine and discarding the requests that have been sitting in the queue for too long. In our implementation, a request has been sitting in the queue for too long if the time left for completion once it gets at the head of the queue is less than the expected time of computation of its class. In other words, a request will be dropped if ($queue\_time + observed\_compute\_time > max\_response\_time\_allowed$).

In Quorum, Selective Dropping works closely with the Load Control module by signaling ahead of time when

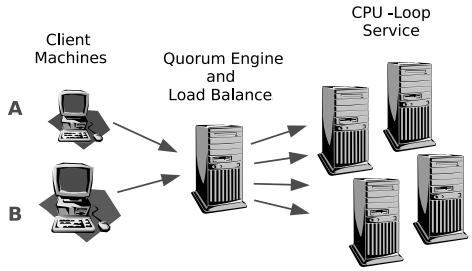

Figure 3: Setup and QoS requeriments for the controlled experiments.

a service class is likely to become overloaded. This module leverages the queuing inside Quorum to absorb safely peaks of traffic during transient overload conditions without violating the response time guarantees. For efficiency reasons, the module delays the dropping of requests to prevent discarding traffic in transient situations only to realize a moment later that the requests could have been served within the allowed response time limits. The implementation of independent dropping techniques, coupled with strong capacity guarantees given by the Request Precedence, allow this module to isolate response times of one class against misbehavior of others.

Combined, the functions of all four Quorum modules (Classification, Load Control, Request Precedence and Selective Dropping) enable cluster responsiveness, efficient resource utilization, capacity isolation and delay differentiation, thus guaranteeing capacity and response times for each independent service class.

4 Understanding How Quorum Works

Before evaluating how Quorum performs under realistic, large-scale systems, we first present results from a small-scale, specially-crafted experiment that helps us illustrate how Quorum and its different modules behave and interact with each other. This illustration is intended to help describe in greater detail the different specific properties of Quorum as well as to introduce the terminology and concepts that will be used when analyzing the large-scale experiments of Section 5.

The testbed used for the experiments in this section consists of 2 client machines, accessing 4 server nodes through a gateway machine implementing the Quorum engine (Figure 3). The gateway machine includes a load-balancer module that balances the requests being released by Quorum into the 4 server machines in a Round-Robin fashion. Each of the server nodes runs the Tomcat application server [27], providing a "CPU-loop service" consisting of a servlet that loops a number of times so that it utilizes a certain amount of CPU (as specified in

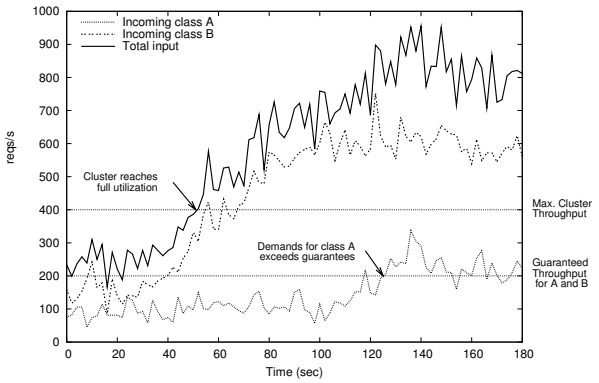

Figure 4: Traffic characteristics of the experiment.

Service Class	QoS Guarantees	
	Max. Response Time (ms)	Min. Throughput (req/s)
A	400	200
B	700	200

Table 1: QoS policy for the controlled experiment

the HTTP parameters of each incoming request). Requests are classified into two different QoS classes (i.e. A and B) according to the host field name found in the HTTP header of the request (i.e. host: A or host: B). Details on the QoS policy can be found in Table 1.

Figure 4 shows the incoming traffic characteristics chosen for the experiment so that we can illustrate the behavior of Quorum under different operating regimes of the cluster. Throughputs are presented in average request per second over 2 second intervals. As depicted in the figure, we deliberately increase the demands for class B such that the combined traffic is able to make the cluster 100% busy by second 50. Between seconds 50 and 120 the cluster remains fully utilized but the demands of class A are still below its guarantees (i.e. 200 req/sec). Finally, after second 120, both class A and class B generate input demands that exceed the guaranteed service specified in their respective QoS classes.

In the rest of this section we analyze the results from the experiment as a way of illustrating the precision with which each of the Quorum modules operates to enforce both capacity and response time guarantees.

4.1 Limiting the Load in the Cluster

Figure 5 shows how the Load Control module of Quorum manages the amount of outstanding requests during the experiment, and the direct effect that this control has on the time it takes the cluster to compute them (i.e. compute times). At the bottom of the figure we include the total input and serviced traffic to show how the cluster becomes fully utilized at second 50 of the experiment.

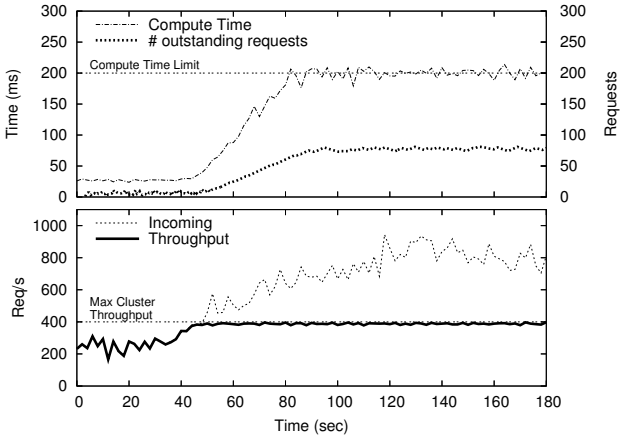

Figure 5: Evolution of computing times and window size during the experiment (top) and Input vs. Throughput (bottom)

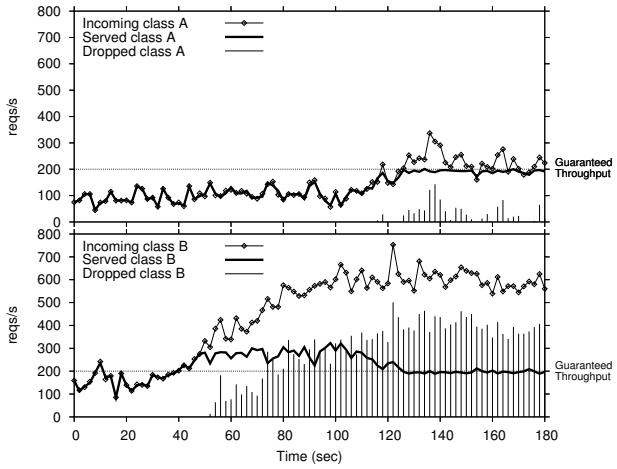

Figure 6: Achieved throughput and request drops for each class in the experiment.

In the figure, as soon as the input demand surpasses the total capacity, the load control module allows the number of outstanding requests to start increasing in an attempt to exploit the parallelism in the cluster and achieve a better overall throughput. However, as soon as the additional parallelism causes requests to take 200 ms to be computed in the cluster (which is one-half of the maximum allowable service time of 400 ms) the Load Control module stops increasing the window of outstanding requests.

It is worth noting that in this example, both QoS classes have the same observed compute time given that they use the same underlying service (i.e. CPU-loop service). In the cases where QoS classes correspond to services with different cluster requirements (e.g. with light vs. heavy resource consumption per request), the load control module would independently track the computation times of each class and will stop increasing the window of outstanding requests as soon as the compute time of one of them reaches half of its service time guarantee.

4.2 Ensuring Capacity Isolation

In this section we illustrate how Quorum virtually partitions the resources of the cluster among the different QoS classes in such a way that their guaranteed capacities are fully isolated when the request load for one or more of the classes exceeds its allowable level.

To illustrate this property, we observe the evolution of the service given to QoS class A during the experiment. Figure 6 depicts the incoming, served and dropped traffic for classes A and B during the run. There are two interesing observations to extract from the figure. The first one is that requests for class A are only dropped when the incoming demand exceeds the guarantee (i.e. there are no drops before second 120). The second observation is that while incoming demands of class A are below its guarantee, all the incoming request are immediately forwarded into the cluster avoiding any queuing time in the engine. This phenomenon can be seen by observing how the incoming and served traffic closely follow each other during the first 120 seconds of the run. Note that even though the cluster reached full utilization in second 50, there are no detrimental effects on how fast the requests for QoS class A are handled showing that capacity agreements are independently guaranteed for each QoS regardless of resource utilization or incoming traffic demands.

4.3 Reassiging Excess Capacity

In this section we show how Quorum reassigns the unutilized capacity from one class to other QoS classes that can use it. To illustrate its behavior we focus on the throughput achieved by class B during the experiment. Consider the incoming and served traffic for both class A and class B in Figure 6. From the figure one can see that the guaranteed resources for class A that are not used are committed to class B when the input traffic of the latter input exceeds its guarantee (seconds 50 to 120). Note the symmetry between the throughputs of class A and B during this interval. However, this surplus capacity received by customers of class B does not cause class A to experience a deterioration of service. Class A's input and throughput values closely track each other indicating that all input request are immediately serviced. As soon as demands for class A exceed their guarantee (i.e. after second 120) there is no spare capacity to reassign to class B. Quorum quickly reassigns the excess resources

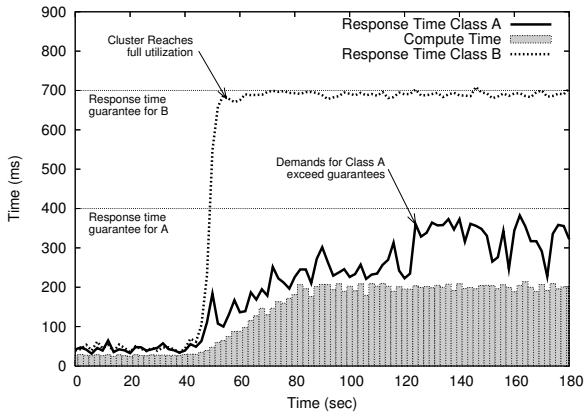

Figure 7: Response time isolation is achieved through independent queuing and selective dropping techniques.

that were temporarily committed to class B back to class A so that both services are maintained at exactly their capacity guarantee of 200 req/sec.

4.4 Response Time Isolation

We now show how Quorum effectively isolates the response times of each QoS class by selectively queuing and potentially dropping requests that cannot meet the guarantees. To illustrate this case, we show the evolution of response times for both A and B QoS classes in Figure 7. We also specify the measured compute time during the run (the shaded area) in order to better observe which portion of the displayed response time corresponds to the time the requests spend in the cluster versus portion that is spent queued inside Quorum. In the figure, we can see that while the cluster is underloaded, all requests are released as fast as possible from the engine and promptly computed by the cluster (seconds from 0 to 50). Queuing or dropping is not necessary during this time interval. As soon as the cluster becomes overloaded at second 50 the engine starts queuing the excess of incoming requests that cannot be immediately computed by the cluster in an attempt to absorb transient peaks of traffic and avoid unnecessary "early" drops. Moreover, class B is the only class that sees an increase in queuing times – it is the only class that has incoming demands exceeding the guarantees. At the same time, class A is isolated from such effects given that its input demands are still below the guarantees. We should note that the increase in response times for class A from seconds 30 to 80, is due uniquely to the underlying increase of compute times triggered by the Load Control module. Before the guarantees are exceeded, Quorum is gradually introducing work in an effort to drive up utilization, as shown in Section 4.1. As soon as both QoS classes have input demands that exeed their guarantee, the system begins enqueuing the incoming requests causing delays up to the maximum of what their respective guarantees allow. For example, the demands for class A exceed the feasible guaranteed capacity at second 123 (denoted with an arrow in the figure). The sudden widening of the space between the response time and the compute time for class A that occurs after that point indicates the magnitude of the queuing delay deliberatly introduced by Quorum. Notice, though, that the sum of both delays never exceed the response time guarantee for the class.

5 Performance Evaluation

In this section we demonstrate that Quorum, even though it treats the cluster resources and Internet services as a "black-box," can provide QoS guarantees under realistic conditions. In addition, we compare Quorum to *Neptune* [24, 25] – a successful research and now commercially deployed QoS system for Internet service provisioning that relies upon a tight integration with the implementation of each service it supports. The goal of this comparison is to determine what effect on performance and capability Quorum's black-box approach introduces. Our investigation is empirical and is based on the deployment of an Internet search service used by *Teoma* [26] using a 68-CPU cluster. We analyze how five different techniques offer differentiated quality to distinct groups of customers by generating message traffic based on web-search traces and then observing the quality of service delivered to each group.

We also examine how well Quorum's flexibility supports multiple hosted services with distinct QoS requirements. In this experiment, we add the TPC-W [28, 29] e-commerce benchmark to the cluster and detail how well the different QoS methodologies support both services simultaneously.

5.1 Experimental Testbed

Our experimental setup consists of several client machines accessing a cluster system through an intermediate gateway/load-balancer machine. Accessing the services through a load balancer machine is the most commonly used architecture in current Internet services. For example, Google [17] funnels traffic through several Netscaler [20] load-balancing systems to balance the search load presented to each of its internal web servers. For our experiments, we use a cluster of 2.6 MHz Intel Xeon processors each with 3 gigabytes of main memory organized into nodes with either two or four four proces-

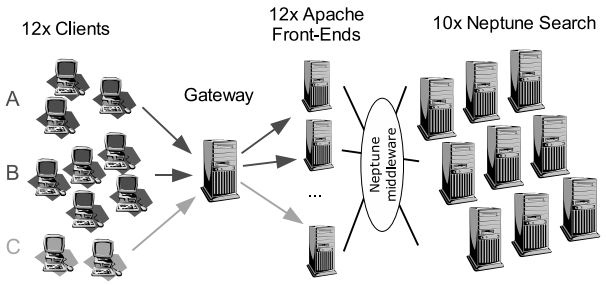

Figure 8: Experimental Testbed.

sors per node. The network interconnect between processors is switched gigabit ethernet and the host operating system is RedHat Linux/ Fedora Core release 1, using kernel version 2.4.24.

As depicted in Figure 8, we use a pool of 12 "client" machines within the cluster to generate requests emulating customer traffic. The client software we use is a custom-made application that can accurately replay server traces by issuing HTTP 1.1 requests using original inter-arrival frequency as specified in the trace.

Our gateway node is a 4-CPU dedicated machine that can function in two different modes: as a load-balancer or as the Quorum engine. When running in load-balancer mode, the machine is configured to implement the typical (Weighted) Round Robin, least connections, and maximum connections options available in most commercial hardware [16, 21, 20]. When running as Quorum engine, the gateway is configured to enforce the QoS policy defined for the experiment. Both the load-balancer and Quorum engine are entirely implemented in user-level software. The load-balancer is implemented as a multi-threaded java application which makes extensive use of the new NIO libraries for improved I/O performance. We use Sun's 1.5 java virtual machine with low-latency garbage collection settings. Our performance tests show that our implementation can handle slightly more than 2000 requests/second with approximately 50% CPU utilization. Thus the performance of our base-level system is comparable to that of a commercial solution for the levels of load presented in this paper. Note that these levels of traffic are close to the service rates of some of the most popular sites (e.g. Google reports around 2500 req/sec [19], Ask Jeeves around 1000 req/sec [6]). Both our implementation of a load-balancer and the Quorum engine are based on the same core software for fielding and forwarding HTTP requests.

5.2 Differentiating Quality of Service by Client

In our first experiment, we compare Quorum to other QoS methodologies in terms of its ability to deliver specified service qualities to separate and potentially competing groups of clients.

5.2.1 Internet Search Benchmark

We use the main index search component of the Teoma commercial Internet search service [26] as a benchmark Internet service. The index search component consists of traversing an index database and retrieving the list of URLs that contain the set of words specified in the search query. The total size of the index database used is 12GB and is fully replicated at each node. The index search application from Teoma is specifically built for the Neptune middleware [25], a cluster-based software infrastructure that provides replication, aggregation and load balancing for network-based services. The version of Neptune we use also provides QoS mechanisms allowing the specification of proportional throughput guarantees and response times constraints through the definition of yield functions [24]. As it is the case with commercial search engines, our system accesses the service through a set of front-end machines which transforms the received URL into an internal query that can be sent to the Neptune middleware servicing the search database for processing. To mimic the setup used by Teoma, we implement the front-end with an Apache web server [4] and a custom-built Apache module which can interface with the Neptune infrastructure. This module is able to utilize the middleware functionality to locate other Neptune-enabled nodes and appropriately balance the requests based on the current load of the available servers.

In order to benchmark Quorum and the other considered QoS methodologies, client requests are replayed from a request trace supplied by Teoma that spans 3 different days of commercial operation. We also use Temoa-supplied traces of word sequences to generate real search queries. The levels of incoming traffic are designed so that the input demands of the different clients are far below (for class A) far above (for class B) and coinciding with (class C) the capacity constraints specified in their respective QoS classes. Clients for each QoS class use different interarrival times, corresponding to one of the three different days of the original traces. Table 2 further depicts the details of each experiment including the capacity and response time guarantees for each QoS class.

Service Class	QoS Guarantees		Experimental Workload	
	Max. Response Time (ms)	Min. Throughput (req/s)	Avg. Input Traffic (req/s)	Service Status
A	200	375	185	Not overloaded
B	600	937	1718	Overloaded
C	300	562	557	Slightly Overloaded

Table 2: QoS guarantees and traffic workload of the Teoma search engine benchmark.

5.2.2 Experimental Methodology

For this experiment our methodology consists of using the previously described testbed to recreate search traffic and to explore the effectiveness with which five different approaches can enforce a particular QoS policy for a single service with multiple client groups. The five compared approaches are:

Load Balancer The gateway machine is configured as a load balancer and tuned to match common high performance settings of Internet sites. Specifically, we configure it to use the least connections load-balancing algorithm and limit the maximum number of open connections for each front-end to match their configured maximum (i.e. 250 processes for Apache server and 150 for the Tomcat servlet engine).

Physical Partitioning A separate group of machines are dedicated for each of the existing QoS classes. We configure the load-balancer to forward requests of a particular class only to its restricted set of reserved nodes.

Overprovisioning The size of each physical partition is increased such that the resulting capacity and response time guarantees can be achieved as specified by the QoS policy (possibly at the expense of under utilized resources).

Neptune QoS The gateway is configured as a load balancer and the QoS mechanisms of Neptune are enabled to implement the QoS policy under study.

Quorum QoS The gateway runs the Quorum engine which implements QoS and the internal cluster resources implement only the Internet service. (i.e. QoS functionality in Neptune is disabled)

We evaluate the effectiveness of each technique in terms of response time and capacity guarantees as measured at the client.

Throughput (req/s)							
Class	Input	Guarantee	Load Balancer	Physical Partitioning	Overprovi-sioning	Neptune	Quorum
A	185	375	148	185	185	185	185
B	1718	937	1333	986	1718	1133	1059
C	557	562	443	558	557	548	557

Response Times (ms)							
Class	Input	Guarantee	Load Balancer	Physical Partitioning	Overprovi-sioning	Neptune	Quorum
A	-	200	13335	40	40	45	121
B	-	600	13395	18937	75	256	600
C	-	300	13487	305	19	62	134

Table 3: Experimental results for Teoma search engine.

5.2.3 QoS Results

Figure 9 presents the results in terms of achieved throughput and average response times of the five QoS methodologies using the same input request streams. The upper portion of the figure shows how the totality of incoming traffic for a class (represented by the height of a bar) has been divided into traffic that is served and traffic that is dropped. Horizontal marks delimit the minimum amount of traffic that has to be served if the QoS guarantees are met. The lower part of Figure 9 presents the results in terms of response times. For response times, we use horizontal marks to denote the maximum response times allowed by the QoS policy and denote with a darker color the classes that do not meet the guarantees. We present these response time results using a logarithmic scale for better visual comparison since the delays differ substantially. Table 3 summarizes these results in tabular form to further detail their comparison.

We begin by analyzing the quality of the service achieved by a load-balancer technique. Throughput results show that amount of traffic served in this case are directly dependent on the levels of incoming traffic rather than driven by the specified QoS policy. In this case we see that the dominance of class B traffic provokes drops in A and C, even though the demands for these classes are always below (in the case of class A) or never exceed (for class C) the input constraints associated with each class. At the same time, observing response time results in the lower figure, we can see that simple connection limiting techniques employed by the load-balancer are not enough to prevent large delays in response times (e.g. up to 14 seconds per request), rendering this technique inadequate to provide timing guarantees.

When resources are physically dedicated through Physical Partitioning, the system is able to serve the expected amount of traffic for each of the classes and drop requests only in the cases when the incoming demand exceeds the input constraint. Throughput guarantees are met, however, if we observe the results in terms of re-

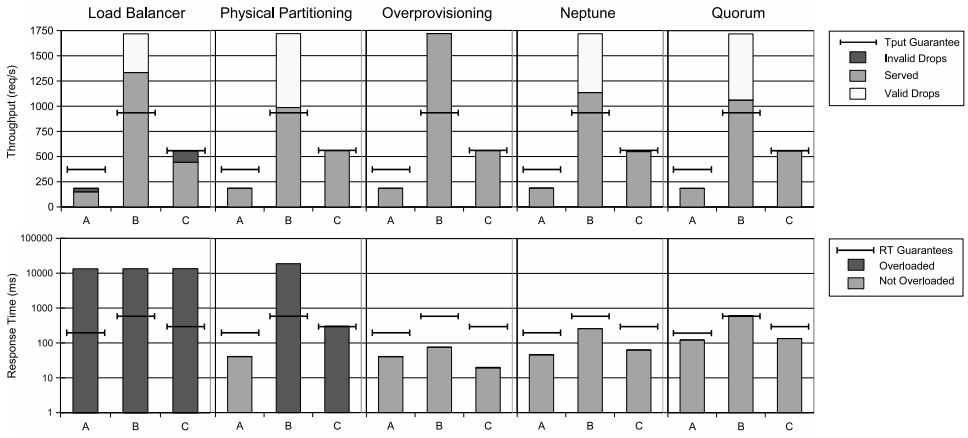

Figure 9: Benchmark results using Teoma's search engine

sponse time, we can see that the overloaded partition B experiences a delay more that 30 times higher than the maximum allowed by the QoS policy. Thus while physically partitioning resources is able to provide capacity guarantees, it fails to ensure response times constraints for arbitrary incoming demands. It is worth noting that the reason for partition B serving more throughput than its guarantee is that the raw performance of the cluster is slightly higher than the combined QoS guarantees defined in the policy.

When each of the partitions is augmented with enough resources (i.e. Overprovisioning) all requests are successfully served. The response times are reduced below the maximum allowed delay. In this case, class B and class C required an additional 5 and 1 nodes respectively in order to meet the specified response time guarantees. Thus Overprovisioning is the first of the techniques that can successfully provide both throughput and response time guarantees. However the cost of such over provisioning can be very high given the extreme load fluctuations that most Internet services have shown to exhibit. In particular, between load "spikes" the extra resources needed to serve these surges in load lay idle.

Neptune and Quorum both meet both the specified throughput and response time guarantees. Both techniques serve at least the necessary amount of traffic and are able to keep response time below the maximum delays. Furthermore, both techniques are able to successfully reassign the capacity not utilized by class A to the greedy clients of class B. We observe that the direct control the resources and services in the cluster allows Neptune to achieve a slightly better throughput than Quorum (i.e. 3%). Also, the resulting response times from Neptune are somewhat lower than Quorum. We expected these slight performance advantages of Neptune over Quorum for two reasons. First, as a purely external implementation, Quorum should incur some performance penalty. Secondly, the current implementation of Quorum is only designed to enforce maximum delay constraints and it is not concerned about minimizing the overall delay of service times. However, given the purely user-space java implementation of the current Quorum prototype, we were surprised by how closely our system matched the performance achieved by the integrated and commercially developed Neptune system. Encouraged by these results, we are currently working on a new prototype that can both ensure response time constraints and lower response delays when possible.

5.3 Differentiating Service by Service Type

Another important type of commercial scenario requires differentiated QoS based on service type rather than customer identity. To explore this case, we deploy an e-commerce service benchmark in addition to the Teoma search service within the test cluster and the apply the QoS methodologies described in Section 5.2.2.

5.3.1 e-Commerce Service Benchmark

The e-commerce service benchmark we use is a freely available implementation [28] of TPC-W – a transactional Web-based benchmark specifically designed for evaluating e-commerce systems by the Transaction Processing Council [29] (TPC). It models a typical e-Commerce site in the form of an online bookstore. Web-based e-Commerce transactions following the TPC-W specification are required to exercise 14 different interactions that are designed to capture the typical tasks of an online bookstore customer. Each interaction requires access to a database that contains information such as book titles, customers, orders and credit card informa-

tion. TPC defines the benchmark specification for TPC-W but provides no reference implementation or source code. For our investigation, we have chosen the publically available implementation from [28] in which the application logic is implemented using servlets. We use the Tomcat servlet engine as the front-end to our online bookstore and deploy a MySql database on an additional 4-CPU machine within the cluster. Our test database contains 100,000 books and 288,000 customers with a total size of 495MB.

5.3.2 Experimental Methodology

We follow a methodology that is similar to the one we used for the previous search experiment. In this case, we deploy both the Teoma search and the e-Commerce application on top of the same 10 machines and configure 12 client machines to generate requests competing for both services. For the search traffic we use the same traces from the previous experiment. To generate requests for the e-Commerce service we use client software from [28] which emulates the behavior of an e-Commerce customers as specified by the TPC-W standard. A particularly noteworthy behavior of this client application is that it stops generating load if any of the requests fails. We speculate that this design feature has been included to implement fail-stop in hardware-partitioned settings when a partition is not sufficiently provisioned. For Quorum and other software-based approaches, however, it introduces the requirement that the QoS methodology guarantee enough capacity at all times to prevent requests from being dropped. Thus it forms an especially stringent test of QoS provisioning by service type since the TPC-W service will not function if the QoS it requires cannot be guaranteed.

We also note that as a standard and comparative reference implementation of the TPC-W benchmark, we were not free to modify the source code. Doing so introduces the possibility that such modifications would skew the results in favor of one methodology or another, and would also eliminate the possibility of comparison to previously published results [13, 3, 15]. As such, the benchmark is an example of an Internet service for which the site operator must provide QoS, but for which it is not possible or desirable to modify the service to meet this need.

As a result, the techniques that we compare in this experiment do not include load-balancer or physical partitioning since these cannot enforce both capacity and response time guarantees without dropping requests as demonstrated in the previous experiment. Drops cause the the TPC-W benchmark to fail. In addition, we cannot present results using the QoS facilities of Neptune given

Service Class	QoS Guarantees		Experimental Workload	
	Max. Response Time (ms)	Min. Throughput (req/s)	Avg. Input Traffic (req/s)	Service Status
Search	500	1300	2460	Overloaded
e-Commerce	750	500	390	Not overloaded

Table 4: QoS guarantees and traffic workload of for the experiment with Search and e-Commerce services.

that benchmark implementation is not programmed to use the middleware QoS primitives. As Neptune gains in popularity, we expect a Neptune-enabled version of TPC-W will become available, but at present the need to modify the source code illustrates a drawback of invasive QoS approaches. Indeed, we considered altering the TPC-W benchmark to use Neptune as an aid to further comparison but doing so requires modification to the benchmark code itself, the Tomcat servlet engine it uses, and the MySql back-end database, all of which must observe the internal Neptune protocols for QoS to be ensured. The possibility for error in these modifications combined with the difficulty in ensuring that their inclusion did not inadvertently alter the performance and functionality of each component deterred us from this effort.

Thus in this experiment, we are only able to compare Overprovisioning with Quorum. We show the details of the complete QoS policy in Table 4. In the experiment, the levels of incoming traffic are designed such that the amount of Teoma search traffic always exceeds the maximum input that can be supported by the output guarantees. That is, the input for the search service attempts to overload the capacity available for searching. The amount of TPC-W traffic is always below its minimum necessary such that the QoS guarantees can be met. That is, there is always sufficient capacity for the TPC-W requests to be serviced without requests being dropped.

5.3.3 QoS Results

Figure 10 presents the results of this experiment using the same format as the previous one. As discussed above, three of the techniques (Load-Balancer, Physical Partitioning and Neptune QoS) cannot run the TPC-W benchmark successfully, thus are not shown. As Figure 10 shows, Overprovisioning with enough resources allows the cluster to serve enough traffic such that both throughput and response time guarantees are within the limits defined by our QoS policy. In this case, the Teoma search partition was augmented to 15 nodes and the e-Commerce service to 4 Tomcats and one additional database, using 100% more resources than the size of the cluster used by Quorum. Thus the efficiency with which Quorum reassigns unused capacity allows our system to

meet the QoS demands while using only half of the hardware resources used by the over-provisioning approach.

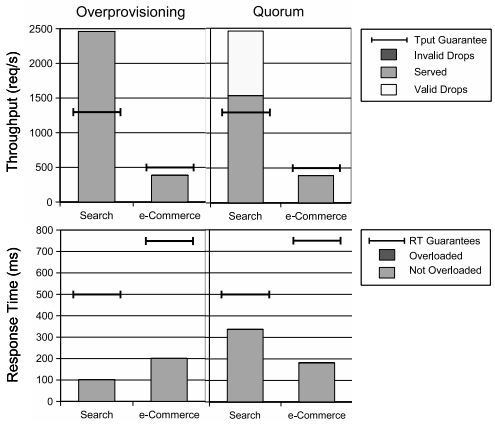

Figure 10: Benchmark results using Search and e-Commerce.

5.4 Discussion

The results from these experiments illustrate several important points. First, static Load-Balancer methods and simple Physical Partitioning without over provisioning are not adequate techniques to provide both throughput and response time guarantees. Secondly, Overprovisioning can provide QoS guarantees, but at the expense of wasted resources. Perhaps unsurprisingly (given the commercial success of Teoma) an integrated approach that requires modification of the Internet services themselves, such as Neptune, both meets the QoS targets and makes efficient use of the resources at hand, but only when a considerable engineering effort has been made to perform the integration. Without this effort, it is not possible to extend or change the services that are supported by the cluster. Our prototype implementation of Quorum, however, is able to achieve nearly the same level of performance as the integrated system for the services it can support, but Quorum can also support new services and extensions transparently, without modification to the service implementation or the cluster resources. At the same time, the efficiency of our Quorum prototype makes it possible to redirect unused capacity automatically thereby limiting resource waste and under utilization.

6 Related work

There are many approaches that address different aspects of the QoS problem, yet very few exist that have focused on providing a complete solution for large-scale Internet services. In this section we briefly introduce some of the most relevant work done in each area and show how each addresses a different aspect of the problem.

In the network community the study of QoS has mainly been focused on ensuring reliable and predefined communication between two endpoints. Protocols such as [9, 12] leverage the existing routing infrastructure to provide bandwidth allocation and packet delay guarantees over the Internet. Other approaches such as Content Distribution Networks [1] provide similar features by appropriately managing their own overlay network and pushing content closer to the end-user. These approaches only tackle the network component and do not address the computational part involved in servicing requests. On the other hand, Quorum is an approach that provides QoS guarantees only at the boundaries of an Internet site. Thus we see the existing network approaches as a necessary complement to Quorum such that end-to-end QoS in Internet services can be realized.

Load balancers [16, 21, 20] can also enhance the quality of the service offered by a cluster system. Basic techniques can successfully balance the load across a number of internal nodes to achieve higher resource utilization. Also, most load balancing routers allow to implement physical partitioning by defining the groups of nodes where requests of a particular type should be forwarded. Similarly, products such as [22] offer the traffic shaping functionality such that minimum bandwidth guarantees can be allocated to distinct clients or applications. More sophisticated techniques such as [20] apply intelligent connection management that shield the internal nodes from becoming overloaded in front of large bursts of incoming traffic. Quorum differs from these approaches in that it is a non-invasive and self-adapting QoS solution that does not need to be configured or tuned for the specifics of the hardware or software of the site.

In the operating systems area, the QoS challenge is usually approached as a resource management problem. Many research operating systems [7, 10, 30] have shown to achieve a tight control on the utilization of resources such that capacity isolation can be ensured between service classes. Although these techniques have proven to be effective in terms of capacity isolation, they are not designed for providing response time guarantees. Furthermore, these techniques are designed for managing the resources of a single machine and fail to enforce the QoS policy in the clustered environments of current Internet services. Cluster reserves [5] is an approach that extends single-node resource management techniques to achieve cluster-wide QoS. Although this technique is shown to provide resource isolation at the cluster level, it

still fails to provide any response time guarantees.

Middleware systems such Neptune [25, 24] include QoS functionality in their distributed infrastructure. By programming the applications to use these primitives it is possible to easily construct distributed services that offer cluster-wide QoS guarantees. However in order for these frameworks to be effective each of the constituents of a service must be integrated with the middleware infrastructure. In a general way, this constitutes a very restrictive constraint given the heterogeneity and proliferation of current Internet services.

Approaches that address the QoS problem at the application level have also been proposed. For example, the approach presented in SEDA [32] advocates the use of a specific framework for constructing well-conditioned scalable services and [31] shows the effectiveness of this framework when explicit QoS mechanisms are used to prevent overload in busy Internet servers. Rather than building an application with QoS support, other work has modified existing applications to include QoS capabilities [2]. For example, the work done in [2] shows how it is possible to modify the popular apache web served to provide differentiated services without the use of resource management primitives at the operating system level. However, as it is the case with middleware approaches, the large cost of modifying the application code to include QoS mechanisms is only effective if the entirety of the software deployment is able to function in a concerted way towards providing QoS.

7 Conclusions

Commercial Internet service provisioning depends increasingly on the ability to offer differentiated classes of service to groups of potentially competing clients. In addition, the services themselves may impose minimum QoS requirements for correct functionality. However, providing reliable QoS guarantees in large-scale Internet settings is a daunting task. Simple over-provisioning and physical partitioning of resources is effective but expensive as enough resource must be available to handle the "worst-case" load conditions for any class. In steady-state, these additional resources are idle and the expense associated with them potentially wasted.

To address this problem, software-based approaches have been developed to control the resource usage of each hosted service. These approaches, such as is Neptune [25, 24] or SEDA [32], require the hosted Internet service to observe an explicit protocol when requesting and using the resources within the site hosting it. As such, they are invasive in that they require a reprogramming and/or re-engineering of the services within a site to implement QoS functionality.

In this paper we present an alternative, non-invasive software approach called Quorum that uses traffic shaping, admission control, and response feedback to treat an Internet site as a "black-box" control system. Quorum intercepts the request and response streams entering and leaving a site to gauge how and when new requests should be forwarded to the hosted services. As a result, new services can be added, existing ones upgraded, decommissioned, etc. without a coordinated re-engineering of the hosted service infrastructure.

We demonstrate the capabilities of a prototype Quorum implementation by comparing it to Neptune using a commercial Internet search service from Teoma [26]. The search service has been explicitly programmed to use Neptune for QoS management by its authors. Despite its non-invasive approach, Quorum is able to implement the same QoS guarantees that Neptune does in the our experimental environment. Quorum is also able to provide QoS to multiple services without their modification, which is not possible with Neptune. We demonstrate this capability using the TPC-W [28] Web transaction benchmark and the Teoma search service simultaneous in the same hosting environment. In this case, Quorum is able to meet the necessary QoS requirements using one-half of the resources required by a successful over-provisioning solution.

Thus by treating the hosting site as a black-box, controlling input traffic, and monitoring output traffic, Quorum is able to achieve the same results as a leading invasive approach while offering additional flexibility and extensibility not previously possible. Encouraged by the performance of our results we are currently working on both enhancing the performance snd scalability of the Quorum engine as well as studying its robustness in front of highly dynamic scenarios including node failures and hardware reconfigurations. Also we are currently exploring its performance using a wider array of Internet services and cluster architectures.

References

[1] Akamai Technologies, Inc. Content Delivery Network. http://www.akamai.com.

[2] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated quality-of-service in Web hosting services. In *Proceedings of the First Workshop on Internet Server Performance*, June 1998.

[3] C. Amza, A. Chanda, E. Cecchet, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks, Austin, Texas, USA November 2002.

[4] Apache HTTP server. http://httpd.apache.org .

[5] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Santa Clara, California, June 2000.

[6] Ask Jeeves Search Engine. http://www.ask.com .

[7] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[8] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, September 1999.

[9] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. RFC 2475, Dec. 1998.

[10] J. Blanquer, J. Bruno, E. Gabber, M. Mcshea, B. Özden, and A. Silberschatz. Resource Management for QoS in Eclipse/BSD. In *Proceedings of the First FreeBSD Conference*, Berkeley, California, Oct. 1999.

[11] J. Blanquer and B. Özden. Fair Queuing for Aggregated Multiple Links. In *Proceedings of the ACM SIGCOMM*, San Diego, CA, August 2001.

[12] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: An overview. RFC 1633, July 1994.

[13] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Nov 2002.

[14] A. Demers, S. Keshav, and S. Shenker. Design and Simulation of a Fair Queuing Algorithm. In *Proceedings of the ACM SIGCOMM*, Austin, Texas, September 1989.

[15] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th International World Wide Web Conference*, New York City, NY, USA 2004.

[16] Foundry Networks, Inc. Server Iron Internet Traffic Management. http://www.foundrynet.com.

[17] Google. Internet search engine. http://www.google.com.

[18] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network Special Issue*, march 2001.

[19] P. Hochmuth. Speedy returns are google's goal. *Network World Fusion*, Jan 2003. http://www.nwfusion.com/news/2003/0901google.html.

[20] Netscaler, Inc. Request Switching Technology. http://www.netscaler.com.

[21] F. Networks. Big/IP Load Balancer. http://www.f5.com.

[22] Packeteer, Inc. Application Traffic Management System. http://www.packeteer.com/.

[23] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks-the Single Node Case. *IEEE/ACM Transactions on Networking*, June 1993.

[24] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[25] K. Shen, T. Yang, L. Chu, J. Holliday, D. Kuschner, and H. Zhu. Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, CA, Mar 2001.

[26] Teoma. Internet search engine. http://www.teoma.com.

[27] Apache Tomcat server. http://jakarta.apache.org/tomcat.

[28] TPC-W Servlet implementation. Tpc-w benchmark java implementation http://mitglied.lycos.de/jankiefer/tpcw/.

[29] Transaction Processing Performance Council. http://www.tpc.org.

[30] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded Web servers. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, Massachusetts, June 2001.

[31] M. Welsh and D. Culler. Overload management as a fundamental service design primitive, Saint-Emilion, France Sept 2002.

[32] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, Banff, Canada, Oct 2001.

[33] H. Zhu, H. Tang, and T. Yang. Demand-driven service differentiation in cluster-based network servers. In *Proceedings of the IEEE INFOCOM*, Anchorage, Alaska, April 2001.