Performance Guarantee for Cluster-Based Internet Services

Chang Li Gang Peng Kartik Gopalan Tzi-cker Chiueh

February 23, 2003

Abstract

As Web-based transactions become an essential element of everyday corporate and commerce activity, it becomes increasingly important for the performance of Web application services to be predictable and adequate even in the presence of wildly fluctuation input loads. In this work we propose a general implementation framework to provide quality of service (QoS) guarantee for cluster-based Internet services, such as E-commerce or directory service, that is *largely independent* of the Web application and the hardware/software platform used in the cluster. This paper describes the design, implementation, and evaluation of a Web request distribution system called *Gage*, which is able to guarantee a service subscriber a pre-defined number of *generic* Web requests serviced per second regardless of the total input loads at run time. *Gage* is one of the first, if not the first systems that can support QoS guarantee which involves multiple system resources, i.e., CPU, disk, and network. The front-end request distribution server of *Gage* distributes incoming requests among a cluster of Web server nodes in a way that takes into account load balancing and quality of service (QoS) guarantee. Each Web server node includes a back-end *Gage* module, which performs distributed TCP splicing and detailed resource usage accounting. The fully operational *Gage* prototype shows that the proposed architecture can indeed provide a guaranteed level of service for specific classes of Web accesses according to their QoS requirements, even in the presence of excessive input loads. In addition, empirical measurement on the *Gage* prototype demonstrates that the additional performance overhead associated with *Gage*'s QoS support for guaranteed Internet service is merely 3.06%.

1 Introduction

First-generation Internet service infrastructure research focused on the development of scalable hardware/software architecture that is capable of supporting the exponentially growing demand on Web-based services. As these services evolve from a novelty to an essential building block for enterprise operations and/or commercial transactions, it is increasingly important for service providers to ensure that the performance of these Web-based services remain predictable and adequate across a wide range of input loads. At the same time, PC cluster has become the de facto computing platform for scalable Internet services. As a consequence, service providers need a management/provision system that can provide guaranteed QoS on a per-customer basis for large-scale PC clusters. This paper proposes a general implementation framework for supporting guaranteed QoS on cluster-based Internet servers, and demonstrates the feasibility of this framework with a Web server cluster called *Gage*, which can guarantee each service subscriber a subscriber-specific number of URL requests per second, even in the presence of excessive input loads.

Consider a Web hosting service provider that has 100 customers, each of which has a different requirement on the size of its Web site and on the Web access service rate required to achieve its business goal. In this case, what this Web hosting service provider needs is a capability to multiplex 100 logical web servers, each with a potentially distinct capacity and/or performance characteristic, on a single physical web server cluster in such a way that each of these logical Web servers is indeed capable of delivering the pre-specified performance at all time. More generally, what this Web hosting service provider needs is a *virtualization* technology that can partition a given physical resource, be it a storage system, a compute cluster, or a network link, into multiple logical entities each of which exhibits a performance characteristic that is independent of others' performance characteristics and input loads. In this paper, we focus on the virtualization technology for compute clusters as a whole.

In the most general form of the compute cluster virtualization problem, multiple paying customers each deploy a Web-based service that runs on a shared PC cluster and demand a guaranteed QoS according to a well-defined performance metric. The Web-based service in question can range from simple URL page accesses, E-commerce transactions, and instant messaging sessions, to public key certificate authorization. As is shown later, the proposed implementation framework for cluster virtualization is sufficiently general and flexible that it can be easily tailored to a new Internet service and a new cluster hardware/software platform with only minuscule modification.

There are three components in the proposed cluster virtualization architecture: *request classification*, *request scheduling*, and *resource usage accounting*. Each Internet service subscriber subscribes to a pre-specified level of QoS, and is allocated a per-subscriber request queue. When an input request arrives, the request classification module determines to which per-subscriber request queue this request should go. Requests within a request queue are serviced in a FIFO order. However, the request scheduling module determines which request from which per-subscriber queue should be serviced next to meet the QoS requirement of each service subscriber. Different input requests, even for the same Internet service, may consume different amounts of system resource. The resource usage accounting module captures detailed resource usage associated with each subscriber's service requests, and feeds them back to the request scheduler so that it can dynamically allocate the system resource according to both subscribed QoS requirement and run-time resource consumption. These three components together physically partitions a given cluster into multiple sub-clusters each of which meets a particular subscriber's QoS requirement without interfering with one another and in a way that is independent of the underlying Internet service. To demonstrate the generality and flexibility of this cluster virtualization architecture, we have designed and implemented a virtualizing Web server cluster that can guarantee each Web hosting service subscriber a specific Web access rate, e.g., 100 generic-URL-requests/sec, where a generic URL request is one that consumes a fixed amount of system resource.

The rest of this paper is organized as follows. Section 2 reviews previous research efforts on cluster-based Internet servers and real-time resource allocation and scheduling that are related to this work. Section 3 describes the system architecture and detailed design of the virtualizing Web server cluster called *Gage*, including its support for load-sensitive request dispatching policy. Section 4 presents the results and analysis of a detailed performance study on the effectiveness and efficiency of *Gage*'s virtualization capability, based on empirical measurements on the first *Gage* prototype. Section 5 concludes this paper with a summary of major research results from this work and a brief outline of the on-going development.

2 Related Work

QoS guarantee can be enforced against different types of resources, from CPU, disk, and network link to an entire compute cluster. The *Gage* project aims to develop a scalable QoS-aware resource scheduler for cluster-based Internet service that is largely independent of the type of the Internet service and the hardware/software platform of the underlying cluster. As a result, *Gage* makes a conscious effort to minimize the part of the proposed architecture that is service-specific and/or platform-dependent. In particular, it assumes that the kernel of the OS running on individual cluster nodes cannot be modified. In this section, we will focus mainly on previous work that provided differentiated services on Web server clusters.

To exploit the information in URL for content-aware request distribution, one needs to decouple request dispatching from TCP connection set-up. Previous systems used either TCP connection splicing [11, 30], which requires the front-end to carry the splicing processing load in both incoming/outgoing traffic and thus tends to become the system bottleneck, or TCP hand-off [26, 4, 5], which uses a connection state migration mechanism to move the TCP connection state either from the front-end one to a back-end node or from one back-end node to another. *Gage* enjoys the scalability benefit of TCP hand-off and the implementation simplicity of TCP splicing. *Gage*'s TCP splicing is scalable because it allows fully distributed implementation.

Cluster reserve [3] is a Web server cluster abstraction that comes the closest to *Gage* in the goal of providing a predictable QoS to each virtual Web site sharing the physical cluster. However, the approach is quite different. Unlike *Gage*, cluster reserve focuses specifically on virtualization of Web page access service. As a result, it takes an approach that requires significant changes to the operating system kernel at the front-end and back-end nodes for customized CPU and disk scheduling, as well as to the Web server for resource principal binding. In contrast, *Gage*'s implementation is completely refined to a thin layer between Ethernet driver and the IP layer and self-contained as a kernel module. This simplicity results from *Gage*'s decision to perform a cluster-wide global resource scheduling, rather than two-level scheduling (inter-node and intra-node) as in cluster reserve. We will show later on that this simplicity does not lead to any compromise in the QoS guarantee that *Gage* can provide under various workloads. As a result of this simplicity, *Gage* can be readily ported to other Internet services without any changes to the service programs themselves.

There are several other projects that also attempted to provide differentiated qualities of Web hosting service. Most of these are priority-based in that they do not provide *guaranteed* QoS. In other words, these approaches allow one service class to receive qualitatively better service than the other, but does not provide a quantitative bound on "by how much." Almeida et al. [1] used a special scheduler process that determines the number of concurrent Web server threads allowed for each service class, and further maps these threads to different priority levels of the kernel's process scheduler. Pandey et al. [22] used a dedicated QoS daemon that sits *behind* the Web server cluster to determine the placement of each incoming request according to the load on each Web server node and each service class's current resource consumption. In this system, requests are never re-ordered. They are either admitted or rejected. Eggert and Heidemann [12] proposed a set of user-level control techniques to support two levels of Web service priorities: limiting the number of concurrent processes for the low-priority class to 5, decrease the process scheduling priority value for the processes dedicated to the low-priority class, and finally throttling the network transmission rate of the processes for the low-priority class to slow them down. Bhatti and Friedrich [7] modified the Apache Web server to build a tiered Web service that performs all request classification, admission control, and request/resource scheduling completely at the user level. Virtually all aspects of this system are configurable through a policy management interface, including a network fabric API. All the above approaches are based on user-level implementations that cannot have an accurate system resource usage information of each service class, and consequently the QoS support is mostly qualitative rather than quantitative. Li and Jamin [18] uses a measurement-based approach to estimate the total network bandwidth consumption of each service class, and then dynamically schedule the order of incoming requests based solely on the network resource usage information. This approach is incomplete because it does not take into account the server's CPU and disk resources. In contrast, *Gage*'s request scheduling considers CPU, disk, and network resource usage[1] Ensim [13] is a commercial effort that supports outsourcing services with guaranteed QoS based on cluster virtualizing technology. Commercial Layer-5/6/7 switches such as those from Alteon [2], Cisco [10], Foundry [15] at most support only content-aware and load-balancing request dispatching to a Web server cluster, but not Web server QoS.

Several research efforts focused on QoS guarantee for specific type of resource, such as CPU [25, 6, 16, 29, 8, 19, 17], disk [20, 9, 28], and on shared network link [21, 24, 14]. Almost all of these systems can be abstracted into the same implementation framework as described in this paper. In particular, one can show the fluid fair queuing model [27] provides the theoretical basis for all these resource schedulers. However, different underlying resources that are under contention may require different detailed accounting mechanisms, and may pose different tradeoffs between QoS-oriented request scheduling and utilization efficiency-conscious request scheduling (such as disk scheduling or multiprocessor scheduling).

[1]The current *Gage* implementation for Web server cluster doesn't consider network resource usage in scheduling because typically the network bandwidth within the Web server cluster is not the system bottleneck . However, the *Gage* architecture allows the network resource usage to be incorporated into the scheduling easily if needed for other cluster-based services.

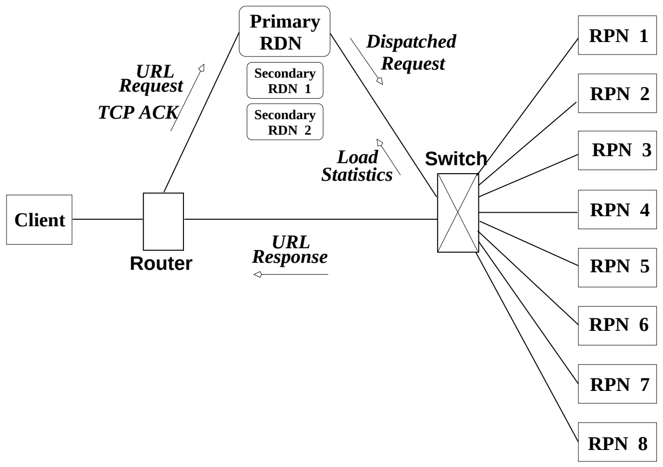

Figure 1: The system architecture of *Gage*. It consists of a front-end request distribution node (RDN) and a set of back-end request processing nodes (RPN) (in this case 8). To prevent front-end processing from becoming a system bottleneck, the front-end could be itself a asymmetric cluster consisting of multiple secondary RDNs (in this case 2), with the primary RDN interacting with the rest of the world on behalf of the entire RDN/RPN cluster.

3 The Gage System

3.1 System Architecture

As shown in Figure 1, *Gage* is a virtualizing Web server cluster that consists of a front-end request distribution node (RDN), and a set of back-end request processing nodes (RPN) that service incoming Web access requests in an order determined by the RDN. To the rest of the Internet, the entire Web server cluster has one single IP address. All the incoming Web access requests are first routed through the RDN, which allocates a request queue for each Web hosting service subscriber, and buffers each incoming access request in the queue associated with the request's target Web site (and thus subscriber). The RDN's scheduler determines the service order of the requests buffered in these queues based on the corresponding subscribers' static performance requirements and dynamic resource consumption rates. Once an RPN receives a dispatched request, it instantiates a separate thread to service it and eventually returns the requested result back to the requesting client *without going through the RDN*. To allow maximum dispatch flexibility, the RPNs are assumed to share a network file storage through a shared-disk storage system.

As the number of back-end RPNs in *Gage* increases, the total Web request processing throughput initially scales up linearly because the processing of distinct Web accesses is largely independent of one another. However, as the number of RPNs further increases, the front-end RDN may become the system bottleneck that eventually renders the end-to-end performance a plateau. One possible solution to this problem is to use an asymmetric RDN cluster to alleviate the performance bottleneck associated with front-end processing and thus scale up the total system throughput. This RDN cluster consists of a *primary RDN*, which receives all the incoming packets and makes all the queuing and scheduling decisions, and a set of *secondary RDNs*, which are dedicated to performing

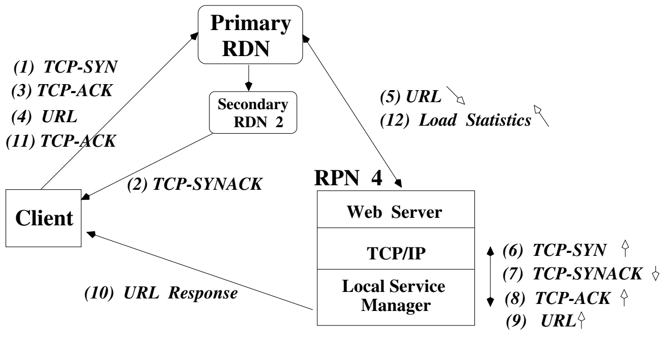

Figure 2: The set of network packets and the sequence of steps involved between a client sending a URL request and it receiving a response. In this case, the secondary RDN 2 performs the first TCP connection set-up, while RPN-4 performs the second TCP connection set-up and sequence number/address re-mapping, as well as processes the request. Note that the second TCP connection set-up is completely confined within a single RPN, between its local service manager and TCP/IP stack.

the time-consuming task in front-end processing such as TCP three-way hand-shaking.

In contrast to low-level and single-resource QoS metrics that other systems support, *Gage* guarantees service subscribers a high-level and multiple-resource QoS: a fixed number of generic URL requests per second (GRPS), where a generic URL request represents an average Web site access and is assumed to take 10 msec of CPU time, 10 msec of disk channel usage time, and 2000 bytes of network bandwidth. If a subscriber's QoS requirement is 50 GRPS, this means that all the requests targeting at this subscriber's Web site are entitled to 500 msec of CPU time, 500 msec of disk access time from the back-end RPN cluster and 100 KBytes of network bandwidth on the outgoing link within every second.

Besides the above QoS metric, there are other possibilities. For example, some subscribers may be more interested in response time or latency; some in delay jitter or reliability. We choose resource usage as the QoS metric because it relates directly to resource allocation. Generalization of the proposed implementation framework for QoS guarantee to other QoS metrics remains an open problem. On a related note, *Gage* does not provide end-to-end QoS guarantee because it does not support network QoS over the Internet. Despite this, *Gage* serves as an essential building block for any end-to-end QoS-guaranteed Internet service.

3.2 Distributed TCP Connection Splicing

URL requests are transported via the HTTP protocol, which in turn is built on top of TCP. Because TCP is a connection-oriented protocol, before the first payload packet can be transported, a three-way hand-shake procedure is required to establish a new TCP connection. Because *Gage* ties the dispatch decision of a Web access request to its URL, here lies the dilemma: To associate a back-end RPN with an incoming Web request, *Gage* needs to examine the request's URL; but to receive the packet that contains the URL associated with a request, *Gage* needs to identify a proper RPN that can set up a TCP connection with the requesting client. *Gage* solves this problem with *TCP connection splicing.* When a requesting client sends in the first TCP packet (*SYN* packet), the front-end RDN first establishes a TCP connection with the requesting client to receive the packet that contains the URL, then establishes another TCP connection with the RPN chosen to process this request, and finally *splices* these two TCP connections into one that is between the RPN and the requesting client.

As an example, the first TCP connection established between the requesting client and the front-end RDN is characterized by

<*Client_IP, Client_PortNo, Client_SequenceNo, RDN_IP, WebServer_PortNo, RDN_SequenceNo*>

whereas the second TCP connection between the front-end RDN and the chosen RPN is characterized by

<*Client_IP, Client_PortNo, Client_SequenceNo, RPN_IP, WebServer_PortNo, RPN_SequenceNo*>

Here we assume *RDN_IP* is the unique IP address for the entire Web server cluster. The result of splicing is a TCP connection between the requesting client and the chosen RPN, in which the requesting client thinks it is communicating with the primary RDN when in fact it is communicating directly with the RPN. To support such an illusion, the source address and the sender sequence number of every outgoing packet from an RPN need to be modified so that they appear to originate from the front-end primary RDN to the outside world; similarly the destination address and the ACK sequence number of every incoming packet to an RPN need to modified to fool the RPN's TCP/IP stack into thinking the packet is directed to that RPN.

In summary, TCP connection splicing involves, for every URL request, two TCP connection set-ups in the beginning, and a sequence number/address re-mapping for every subsequent packet between the requesting client and the servicing RPN. A truly scalable TCP connection splicing implementation requires that these steps be executed in a fully distributed fashion. In *Gage*, sequence number/address re-mapping of an incoming or outgoing packet is performed at the RPN that is the packet's source or destination, by a software module called the *local service manager* that resides above the Ethernet driver but below the IP layer. In addition, the set-up of the second TCP connection, one between a client and its servicing RPN, is also by the RPN's local service manager. Finally, the fact that *Gage* can employ an asymmetric RDN cluster to collectively shoulder the processing load of setting up the first TCP connection for incoming URL requests ensures that more resources can be readily incorporated into the system architecture to alleviate the performance problem when it arises. Figure 2 illustrates the set of network packets and the sequence of steps involved in sending a URL request from a client to a *Gage* cluster and receiving a response.

In contrast, most existing TCP connection splicing implementations in commercial Layer-4/5 switches perform both three-way hand-shakes for each URL request, as well as sequence number/address re-mapping for all subsequent packets. As a result, these implementations tend to suffer from the scalability problem despite the use of custom ASIC. The only fully distributed TCP connection splicing implementation [5] that we are aware of relies on the support of TCP connection state migration so that the RPN that performs initial TCP connection establishment may be different from the RPN that actually services the incoming request. However, connection state migration requires substantial modification to the TCP/IP protocol stack implementation in the RPN's kernel.

3.3 Request Classification

The primary RDN classifies an incoming packet into the following three categories:

1. *SYN* or *ACK* packets that are involved in TCP's three-way hand-shake procedure,
2. Packets that contain a URL-based Web access request, and
3. All other packets.

Upon receiving the first type of packets, the front-end RDN handles them itself by emulating the three-way hand-shake procedure instead of sending them to the TCP/IP stack of the kernel. In this way, the overhead of the first-leg TCP set-up for TCP splicing is minimized.

After receiving a packet of the second type, the primary RDN determines to which subscriber web site the packet belongs according to the host-name part of the URL, and then puts the request packet to the queue associated with the service subscriber. There are a few request classification criterion, but *Gage* chooses to use the host name (such as www.ibm.com) as the classification criterion because it targets at virtual web hosting service.

For all other packets, the primary RDN simply serves as a Layer-2 bridge that forwards each incoming packet to its corresponding back-end RPN. This routing is based on a connection table that is indexed on the quadruple of the packet header, source IP address, source port number, destination IP, and destination port number. After a URL request is dispatched to a RPN, the packet's quadruple and the MAC address of the RPN is inserted into this connection table, so that all the subsequent packets from the clients are routed to the corresponding RPNs.

3.4 Request Scheduling

There are two decisions that *Gage*'s request scheduler needs to make: which request should be serviced next and which RPN should service the request. The "which request" decision is related to QoS guarantee and is made according to each subscriber's static resource reservation and dynamic resource usage. In the process of making the "which RPN" decision, *Gage* attempts to maximize the system utilization efficiency by balancing the load on the RPNs. Accordingly, *Gage*'s request scheduling mechanism is implemented by a *request scheduler* and a *node scheduler*, both running on the primary RDN.

Gage's request scheduler is invoked periodically in a polling loop, with the *scheduling cycle* set to be 10 msec for responsiveness. Three pieces of information enter into the request scheduling decision: the static resource reservation of each service subscriber, the instantaneous measured resource usage attributed to each subscriber's requests, and the resource availability of each RPN. The last two pieces of information are fed back to the RDN from the RPNs periodically, once every *accounting cycle*. *Gage*'s request scheduler performs a weighted round-robin algorithm, which visits each subscriber's queue in a cyclic fashion, and in each such visit, first adds a credit to the queue's balance according to its associated reservation and dispatches as many requests as possible until the balance becomes negative or when the queue is empty. Whatever spare resource remains after the first round of scheduling is then distributed in a weighted fashion among those queues that are still non-empty according to their resource reservations.

Because different URL requests may consume different amounts of system resource, the total resource consumption of the requests from a queue is not necessarily proportional to the number of requests that are dispatched and not yet completed. Since it is not possible to determine a URL request's resource usage at the time of its dispatch, *Gage* cannot update the associated queue's resource usage balance accurately at that time. However, for those URL requests that are already completed, the request scheduler does know how much system resource they consume, based on the feedback provided by the RPNs, and thus can deduct them from the corresponding queue's balance properly. The current *Gage* request scheduler implementation assumes that the resource consumption of each dispatched request is equal to the (dynamically computed) average resource consumption of the past requests that belong to the same queue. More concretely, suppose the measured average per-request resource usage for requests from Subscriber i during the kth accounting cycle is P_k^i, and the measured average per-request resource usage for requests from Subscriber i before the kth cycle is M_{k-1}^i. Then M_k^i is set to $\alpha * M_{k-1}^i + (1-\alpha) * P_k^i$, where α is a smoothing parameter, and the resource usage prediction for requests from Subscriber i that are dispatched during the $(k+1)$th accounting cycle is M_k^i.

In each accounting cycle, the resource usage feedback message from an RPN to the RDN includes per-subscriber resource usage and the total resource usage at that RPN during that accounting cycle. The accounting cycle time is typically higher than the request scheduling cycle to reduce the traffic volume of feedback messages. As a result, there are multiple scheduling cycles per accounting cycle and some of the request scheduling decisions are made based on stale per-RPN and per-subscriber resource usage information. After receiving resource usage feedback messages from all RPNs, *Gage* sums up available resources at the RPNs and divides the total available resource evenly among the scheduling cycles associated with the coming accounting cycle. Within each scheduling cycle *Gage* first performs one iteration of weighted round-robin resource allocation, then computes the spare resource available based on the predicted resource usage of those requests that have just been dispatched, and finally performs another iteration of weighted round-robin resource allocation to distribute the available spare resource among non-empty queues. The second iteration is the same as the first except that each queue's balance is incremented with a percentage of the spare resource that is proportional to its reservation, rather than a fixed credit. This spare resource allocation algorithm ensures that the amount of additional service that each subscriber receives beyond its reservation is proportional to its reservation, when there is enough system capacity to do so.

Every time the request scheduler decides to dispatch a request, *Gage* calls the node scheduler to determine to which RPN this request should be sent for service. The node scheduler makes this decision based on the load on each RPN, and/or which RPN requests for similar pages have been dispatched to in the past. For example, if a dispatched request is sent to the RPN with the least load, the node scheduler corresponds to a load balancer. On the other hand, if a dispatched request targeting the same web site is always sent to the same RPN, the node scheduler is exploiting file cache affinity. The node scheduler in the current *Gage* prototype makes this decision based only on the load on each RPN, i.e a dispatched request sent to the RPN with the least load.

Once the primary RDN dispatches a request to an RPN, it sends a message to the chosen RPN asking its local service manager to establish the second TCP connection with its TCP/IP stack. The three-way handshake of the second TCP connection does *not* involve the RDN. Once the second TCP connection is set up, the local service manager establishes a sequence number/address map and performs sequence number/address re-mapping for all incoming and outgoing packets of this TCP connection. After this TCP connection is closed, the local service manager sends a request completion message to the primary RDN to delete the corresponding entry in the quadruple-based connection table.

3.5 Resource Usage Accounting

To accurately account for the system resource that a URL request consumes on an RPN, the RPN kernel needs to measure the CPU time, the disk access time, and the network bandwidth that a URL request consumes. When a request is completed, the local service manager at the RPN feeds this information back to the primary RDN, which then adjusts the balance of the subscriber queue to which this request belongs. As there are three resources involved, when the request scheduler visits a queue, it continues to dispatch requests and subtracts their estimated disk/CPU/network resource usage from the queue's disk/CPU/network balance until one of them becomes negative.

Gage's RPN runs the Linux kernel, which already keeps track of the CPU usage of each active thread. To collect the disk usage time of each thread, we added a timestamp to the disk driver code to record the amount of time that each physical disk I/O takes and to charge it to the thread that issues the disk I/O request. Network bandwidth consumption is equal to the size of the returned page, and therefore does not require extra measurements.

Gage's resource usage accounting model assumes that a set of dedicated processes are associated with each charging entity, a virtual web site in the case of web page access service. When a charging entity is launched, *Gage* records the first process or processes associated with the entity. At run time, periodically *Gage* traverses the kernel data structure that keeps track of parent-child relationships among processes and sums up the resource usage of all the processes that are associated with each charging entity. The period is the accounting cycle. The model also allows the the number of processes for a service to be dynamically changing. In general, by maintaining per-process resource usage accounting and associating a separate set of processes with each distinct Internet service subscriber, the above model can accurately account for each service subscriber's resource usage *without making any assumption on the request/response formats of the Internet services*, and thus matches the design goal of *Gage* perfectly. In particular, this resource accounting model automatically works for CGI programs without any additional mechanisms.

However, there are two potential limitations with this resource accounting model. First, each Internet service needs to be uniquely associated with a set of "root" processes. Such association needs to be input to the kernel when an Internet service is instantiated in the beginning or when the root processes are killed and restarted. In practice, this additional input imposes only minor bookkeeping problem. Second, if a child process is spawned and terminated within an accounting cycle, the system may not be able to account for its resource usage. To solve this problem, when a process is terminated, *Gage* adds the process's resource usage since the beginning of the current accounting cycle to its parent process's resource usage data structure. This way *Gage* can still correctly account for the resource usage for such processes.

To support QoS, for each subscriber, the RDN maintains its current *balance*, and an *estimated resource usage array*, each element of which records the sum of predicted resource usage of all pending requests dispatched from this subscriber to a particular RPN. In addition, to support load balancing, for each RPN, the RDN maintains its current *capacity*, and an *estimated outstanding load* field, which is the sum of predicted resource usage of all pending requests dispatched to this RPN. Every time a subscriber's request is dispatched to a RPN, the request's predicted resource usage is added to the RPN's estimated outstanding load, as well as the subscriber's estimated resource usage array element associated with the RPN. Every time a resource usage accounting message from an RPN arrives, which includes the total and per-subscriber resource usage on that RPN in the previous accounting cycle, the RDN subtracts the RPN's total resource usage from its capacity and estimated outstanding load, and subtracts each per-subscriber resource usage from the corresponding subscriber's balance and estimated resource usage array element associated with that RPN.

3.6 Service-Specific Component

Gage is an instance of a general implementation framework for supporting QoS guarantee for Internet service. In this subsection, we enumerate the assumptions of *Gage* that are specific to Web page access service as follows:

- Packet classification is tied to the host name part of URL requests.
- The QoS metric assumes that a generic Web page access costs 10 msec of CPU time, 10 msec of disk channel usage time, and 2,000 bytes of network bandwidth on the average.
- Content-aware request dispatching is based on the assumption that URL pages in the same proximity should be serviced by the same RPN to exploit access locality.

The fact that the *service-specific* component of *Gage* is relatively small demonstrates that the proposed framework is indeed quite general and thus can be easily adapted to other Internet services. For a different Internet service, the packet classification criterion may be completely different. For example, it could be based on user IDs embedded in the application-layer protocol header. Similarly, what constitutes a "generic request " may also be very different for a different Internet service. For example, a request in another Internet service may involve different amounts of CPU/disk/network usage. Finally, different Internet services may use different optimization techniques to improve the system's utilization efficiency, just like content-aware request dispatching can improve the effective processing capacity of a Web server cluster by avoiding unnecessary disk I/Os.

4 Performance Evaluation

4.1 Prototype Implementation and Test-bed Set-up

The *Gage* prototype implementation includes a RDN and a RPN component. Both are built as a thin software layer between the Ethernet driver and the IP module under Linux 6.0. RDN performs connection table look-up, classification of URL requests according to their host name field, en-queuing and scheduling of URL requests, and packet forwarding. RPN performs local TCP

Subscriber	Reservation	Input Load	Served	Dropped
site1	250	259.4	259.4	0.0
site2	150	161.1	161.1	0.0
site3	50	390.3	365.4	24.9

Table 1: *The QoS guarantee provided by* Gage *in terms of numbers of generic requests per second.*

Subscriber	Reservation	Input Load	Served	Spare Resource
site1	250	424.6	422.2	172.2
site2	200	364.5	342.4	142.1

Table 2: *Spare resource allocation in* Gage. *All values are in terms of generic requests per second.*

connection set-up, per-node and per-subscriber resource usage statistics collection and reporting, and sequence number/address re-mapping for incoming and outgoing packets.

The test-bed is similar to Figure 1 except that there is no secondary RDNs. There are eight back-end RPNs, each of which is an industrial PC consisting of 600-MHz Celeron CPU, 64-MByte memory, 10-GByte disk, and two Fast Ethernet interfaces. The primary RDNs and the clients are all 450-MHz Pentium-III machines with 64 MBytes of main memory. Each RDN has two Fast Ethernet interfaces. Clients, RDNs, and RPNs are connected through a 16-port Fast Ethernet switch that features a 3-Gbit/sec cross-section bandwidth in its switch fabric. Therefore, network contention effect is reduced to the minimum in the following experiments.

There were two types of workload used in the following experiments - synthetic and realistic. We obtained the realistic workload by collecting the trace generated by SPECWeb99[31]. The method of load generation by the client is similar to that used in [32]. The clients load in the trace from a file and issue requests to *Gage* in a *constant* rate. Unless otherwise mentioned, the workloads used in the following experiments are synthetic.

4.2 QoS under Excessive Input Loads

4.2.1 Performance Isolation

Because *Gage* is designed to provide QoS guarantee to individual Web service subscribers, we first show the effectiveness of *Gage*'s performance isolation mechanism in the presence of excessive input loads. Table 1 shows the Web request throughput *Gage* provides to three subscribers (site1, site2 and site3), where the input load for site1 and site2 are almost equal to their resource reservations and for site3 is much higher than its reservation. The result shows that *Gage* satisfies the QoS requirements of each subscriber and then allocates residual resources to site3 whose input load is much higher than its reservation. Although most spare resource is allocated to site3 because other subscribers did not exceed their reservation much, some of the requests to site3 still have to be dropped because there is not enough residual resource left to service them all after satisfying the

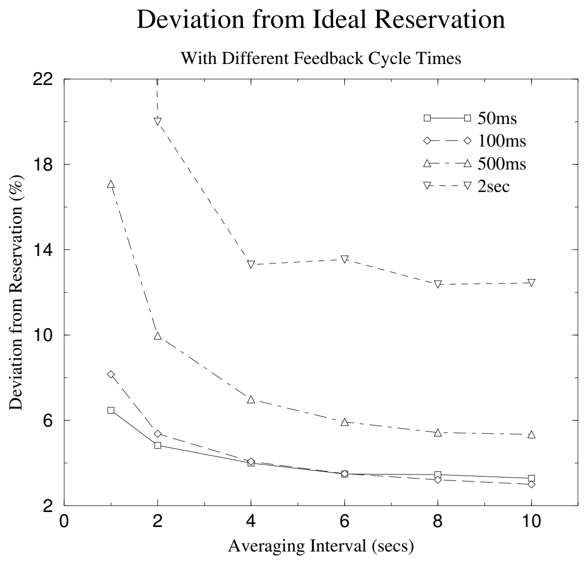

Figure 3: Deviation of actual resource usage from the ideal reservation under different accounting cycle times. On the curve associated with the accounting cycle time of 2 seconds, the deviation corresponding to the averaging interval of 1 second is 114.3% and is not shown on the curve.

QoS requirement of each subscriber.

4.2.2 Spare Resource Allocation

Gage's spare resource allocation policy is "higher reservation gets larger share of spare resource" as opposed to "higher input load gets larger share of spare resource." We believe that this policy is more fair to the subscribers with higher reservation since it apportions the spare resource in proportion to subscribers' reservations. Table 2 shows the Web request throughput that *Gage* provides to two subscribers, both of which have input loads higher than their reservations. The result shows that the residual resource allocated between site1 and site2 is roughly proportional to their reservations.

4.2.3 Deviation from Ideal Reservation

A key architectural decision of *Gage*'s QoS guarantee mechanism is its strive to be independent of the type of Internet service and the platform of back-end RPNs. As a result, the resource usage information from the RPNs may not be exact and timely, and the QoS guarantee provided by *Gage* exhibits a certain amount of jittery. For example, *Gage* may be able to guarantee 100 requests per sec, but may not be able to guarantee 10 requests per 100 msec. In this subsection, we examine the extent of jittery under variations of input workloads and system parameters.

One desirable and significant characteristic of a system providing QoS guarantee is *stability*, i.e. the QoS guarantee should remain stable across input workload fluctuation and the resource allocated to each subscriber does not vary much from time to time given a fixed input load. The difference

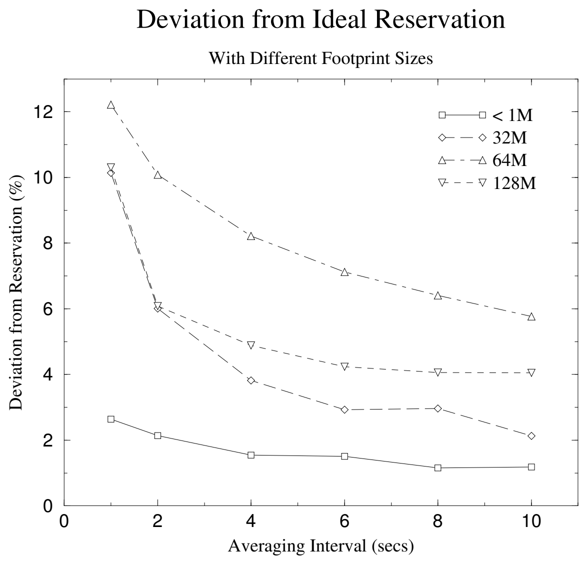

Figure 4: Deviation of actual resource usage from the ideal reservation under different footprint sizes.

between the resource allocated to a subscriber and its original reservation, and its evolution across time, offer critical insight into the stability of a system that provides guaranteed QoS like *Gage*. The smaller the deviation from the original reservation, the more stable QoS guarantee the system can provide.

In *Gage*, the major source of instability in QoS guarantee is the inaccuracy of predicting the per-request resource usage. Based on the resource usage history for requests from a subscriber, the RDN projects the per-request resource consumption in the near future for that subscriber using a dynamically computed running average. If the predicted per-request resource usage is not close to the actual resource usage, the request scheduling algorithm that enforces QoS guarantee may operate on erroneous accounting information and QoS instability may result.

The inaccuracy in per-request resource usage prediction in turn is attributed to two factors in *Gage*. One is that the RDN is unable to obtain the resource usage statistics for each subscriber in time, because RPNs only feed resource usage information back to the RDN periodically. This forces the RDN to rely on *stale* accounting information to project per-request resource usage, and thus the resulting prediction may not be 100% accurate. The other is that the resource consumption by requests from a subscriber may vary greatly from request to request. In this case, it is inherently difficult to make accurate per-request resource usage prediction purely from resource usage information history.

The experiments in this subsection demonstrate how these two factors impact the stability of *Gage*'s QoS guarantee. In each experiment, we measure the resource usage deviation of each subscriber from its reservation over different time intervals, and then compute an overall average among all subscribers. The input for the first two experiments is a constant synthetic workload with each request accessing a file of the size of 6 KBytes. The input to the third experiment is a synthetic workload that varies the request size according to a fixed mean and standard deviation.

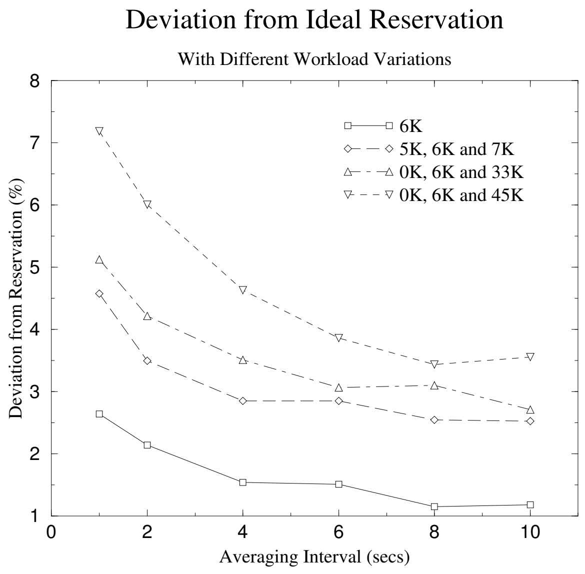

Figure 5: Deviation of actual resource usage from the ideal reservation under different access request sizes. The variations in average file size used in the experiments corresponding to the curves from the uppermost to the lower-most are 0%, 13.6%, 110.4% and 117.4%, respectively.

The input to the last experiment is a request trace extracted from SPECWeb99.

Figure 3 shows the deviation of actual resource usage from the ideal reservation under different accounting cycle times and averaging intervals. The result shows that as the accounting cycle time increases, the deviation from the ideal reservation also increases for the same averaging interval. This is because the resource accounting information on the RDN is updated less frequently with increasing accounting cycle time and hence the resource usage prediction is less accurate. In particular, at the data point corresponding to a 2-second accounting cycle and 1-second averaging interval, the deviation is above 100%. At this point, the accounting cycle time is twice as long as the averaging interval, the resource usage for subscribers that RDN observes within an averaging interval is either *0* or around *twice the reservation*. Thereby, the deviation from the ideal reservation associated with that data point is around 100%. As the averaging interval increases, the deviation decreases because the effect of short-term jitters become less noticeable. Overall, the deviation computed over an averaging interval of 4 seconds or more can be kept under 8% if the resource usage information from RPNs is sent at a period no larger than 500 msec.

Figure 4 shows the deviation of resource usage from the ideal reservation with different footprint[2] size. The result shows that the footprint with the size less than 1 MBytes incurs the lowest deviation, and 64 MBytes the highest, and 32 MBytes and 128 MBytes in the middle. The < 1 MBytes footprint can be cached *completed* in memory at RPNs. Hence, there is almost no variation of the per-request resource consumption in this case and the predicted per-request resource usage is quite accurate. On the other hand, in the case of 64 MBytes footprint, around *half* of the footprint is cached and so around *half* of the requests will require I/O operation. As a result, the

[2]The set of files on RPNs that the clients access.

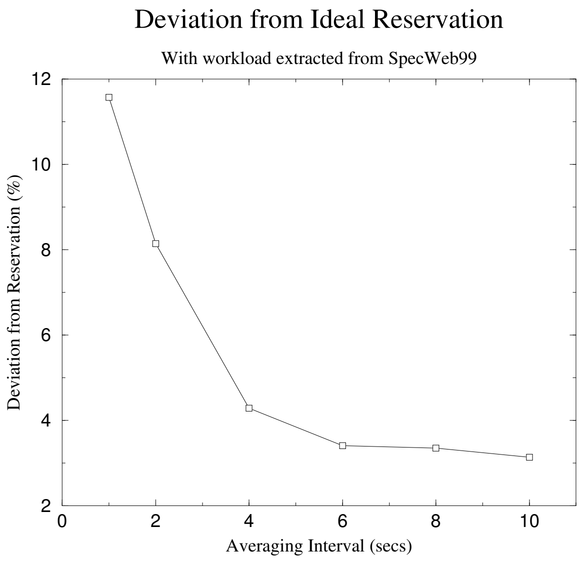

Figure 6: Deviation of actual resource usage from the ideal reservation with respect to workloads based on SPECWeb99.

connection setup(μsec)		classification
on RDN	on RPN	(μsec)
29.3	27.2	3.0

Table 3: *Per-connection overhead in* Gage. *Classification is done by RDN.*

variation in per-request resource usage is the highest and the prediction is the least accurate. For the 32 MBytes and 128 MBytes cases, either *most* of the footprint is cached or not cached and hence *most* of the requests in either case will consume similar amount of resource, either CPU or I/O. Therefore, the per-request resource usage in these two cases does not vary as much as the 64 MBytes case and the accuracy of prediction is somewhere in between.

The deviation of resource usage from the ideal reservation under different access request sizes is shown in Figure 5. The figure confirms that greater variation in access request size generally leads to more substantial deviation. This is because large request size variation makes per-request resource usage prediction less accurate and thereby results in more serious QoS deviation.

To further evaluate the stability of *Gage*'s QoS guarantee, we ran an experiment using a representative Web workload trace derived from SPECWeb99. The result in Figure 6 shows that under realistic Web access workloads, the QoS deviation from reservation is less than 5% if the averaging interval is 4 sec or higher.

4.3 Overhead Analysis

An important concern in the design of *Gage* is the additional performance overhead it incurs because of the support of QoS guarantee. The per-request overhead in *Gage* can be classified

packet forwarding (μsec)	address/sequence number remapping (μsec)	
	incoming packet	outgoing packet
7.0	1.3	4.6

Table 4: *Per-packet overhead in* Gage. *Packet forwarding is at RDN and address/sequence number remapping is performed by RPN.*

into two categories: per-connection overhead and per-packet overhead. The former is due to the extra work needed for connection setup in TCP splicing and request classification. It is shown in Table 3. The latter is due to forwarding and address/sequence remapping for each packet. Table 4 summarizes this overhead.

From Table 3, without considering time for handling individual packets, the total time required to set up one connection in *Gage* is 56.5 μsec. The overhead of processing a single ACK from clients is 8.3 μsec and sending a data packet to clients is 4.6 μsec according to Table 4. For a request whose targeted file size is 6 KBytes, which is the typical HTTP request size, the total measured overhead for handling such a request in *Gage* is around 124 μsec. Compared to the time a Web server takes to service a HTTP request, which is in the range of several to tens of mini-seconds, the additional latency that *Gage* introduces is relatively small and negligible.

According to Figure 7, the number of requests, each accessing a 6 KBytes file, that one RPN can sustain per second is about 540. Each request takes the local service manager on RPN a total of 56.7 μsec for connection set-up and address/sequence number remapping, assuming each request consists of 5 data-ACK packet pairs. Therefore, the total additional overhead imposed by *Gage* to a RPN is less than 56.7 x 540 = 30,618 μsec, or only under 3.06% of a RPN's CPU cycles.

4.4 Scalability Study

To evaluate the scalability of *Gage*, we measured the throughput that *Gage* supports in terms of request service rate as the number of RPNs was increased from 1 to 8. The result of this experiment presented in Figure 7 shows that the throughput grows linearly with the number of RPNs increased from 1 to 8. We also measured the throughput each RPN can support *without Gage*. It was 550.5 requests/sec, compared to 540 requests/sec when *Gage* is in place. This shows that the throughput penalty because of *Gage*'s QoS guarantee mechanism is about 1.8%.

Because of the limitation in hardware availability, especially RPNs, we were not able to extend the above experiment so as to demonstrate the maximal throughput that one RDN (PIII 450MHz) could support. However, we can project that value by measuring how the CPU utilization of the RDN varies with the throughput increase. All the processing in the *Gage* module on RDN is done in the bottom half of the Linux kernel. Hence we can calculate the RDN's CPU utilization by measuring the CPU time for handling interrupts and bottom half processing. The result in Figure 8 shows that the CPU utilization on the RDN increases close to linearly as the throughput grows from around 500 requests/sec to 4400 requests/sec and then it makes a big jump when the throughput advances to around 4800 requests/sec. To identify the cause of this utilization leap, we re-ran the experiment under the two highest throughput and measured the interrupt handling

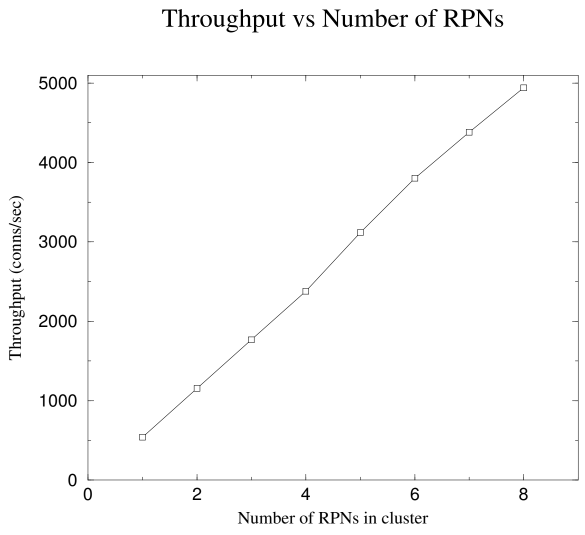

Figure 7: *Gage*'s throughput in requests/sec under different number of RPNs.

time and bottom half handling time *separately*. The result reveals that the utilization leap is attributed mainly to the increasing in interrupt handling time, which grew from 9.77% of the CPU time to 28.46%. This is because when the throughput is very high, the network subsystem may be overloaded and the per-interrupt latency increases. This problem can be alleviated with an intelligent network interface that has its own processor. With such intelligent interfaces in place, conservatively with one PIII 450MHz RDN the throughput *Gage* can support is around 1,4000 to 1,5000 requests/sec; alternatively it can support up to 24 RPNs without being a performance bottleneck.

5 Conclusion

This paper presents the design and implementation of *Gage*, a scalable QoS-aware resource scheduler for cluster-based Web application services. The architecture of *Gage* allows it to be implemented as a self-contained layer between IP and network device driver, and thus greatly simplifying the task of porting it to other operating system platforms. Furthermore, the *Gage* layer is completely transparent to the Web service applications that process incoming requests. *Gage*'s resource accounting model is based on per-process/thread resource usage information extracted from the operating system, and is thus mostly independent of the OS platform and hardware architecture of the back-end servers. Finally, the QoS guarantee *Gage* is able to support requires careful allocation of multiple system resources, and reflects the emerging performance requirements of commercial enterprises that critically depend on Web-based services.

The experimental results on the *Gage* prototype demonstrate that the system can indeed provide guaranteed QoS in terms of number of generic requests serviced per second, and the overhead it introduces, in terms of its impacts on the per-request latency and overall system throughput, is

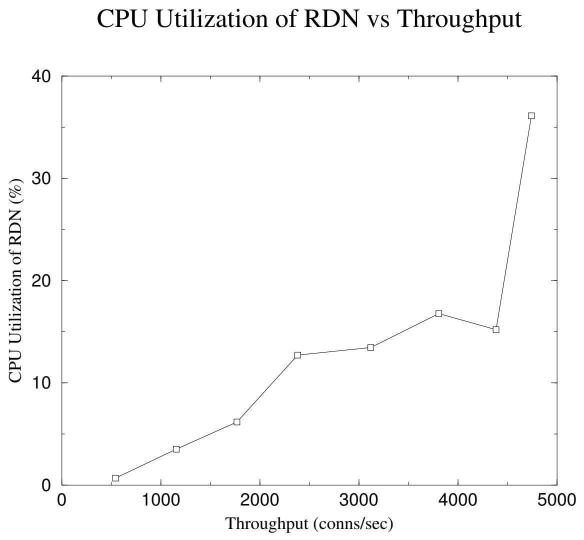

Figure 8: The CPU utilization of RDN with the increase in the system throughput. The CPU utilization of RDN is calculated by summing up the CPU time required to process interrupts and dealing with bottom half work.

negligible (less than 4%). It also demonstrates that its overall performance scales linearly with respect to the number of back-end nodes.

Armed with the success in Web server QoS, we are currently applying the QoS implementation framework described in this paper to the other components of the standard three-tier Web-based service architecture: application servers and database servers. In particular, we plan to develop a virtualizing database server cluster that supports multiple logical database servers each of which is guaranteed a pre-defined number of "generic" SQL transactions per second regardless of the total input loads.

References

[1] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. "Providing Differentiated Quality-of-Service in Web Hosting Services." In Proceedings of the Workshop on Internet Server Performance, Madison, WI, June 1998.

[2] Alteon WebSystems. ACEdirector. http://www.alteon w ebsystems.com.

[3] M. Aron. *Differentiated and Predictable Quality of service in Web Server Systems.* Ph.D. Thesis, Computer Science Department, Rice University, October 2000.

[4] M. Aron, P. Druschel, and W. Zwaenepoel. "Efficient Support for P-HTTP in Cluster-based Web Servers." In Proceedings of the USENIX 1999 Annual Technical Conference, Monterey, CA, June 1999.

[5] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. "Scalable Content-aware Request Distribution in Cluster-based Network Servers." In Proceedings of the USENIX 2000 Annual Technical Conference, San Diego, CA, June 2000.

[6] G. Banga, P. Druschel, and J.C. Mogul. "Resource containers: A new facility for resource management in server systems." In Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation , Feb. 1999.

[7] N. Bhatti and R. Friedrich. "Web Server Support for Tiered Services." IEEE Network, 13(5):64-71, Sept. 1999.

[8] J. Blanquer, J. Bruno, E. Gabber, M. Mcshea, B. Ozden, and A. Silberschatz. "Resource Management for QoS in Eclipse/BSD." In Proceedings of the FreeBSD 1999 Conference, Berkeley, California, October 1999.

[9] J. Bruno, J. Brustoloni, E. Gabber, M. McShea, B. Ozden, and A. Silberschatz. "Disk Scheduling with Quality of Service Guarantees." In Proceedings of ICMCS99, June 1999.

[10] Cisco Systems Inc. LocalDirector. http://www.cisco.com.

[11] A. Cohen, S. Rangaraan, and H. Slye. "On the Performance of TCP Splicing for URL-Aware Redirection." In Proceedings of the 2nd Usenix Symposium on Internet Technologies and Systems, Boulder, CO, Oct. 1999.

[12] L. Eggert and J. Heidemann. "Application-Level Differentiated Services for Web Servers." World Wide Journal, 2(3):133-142. August 1999.

[13] Ensim. http://www.ensim.com.

[14] S. Floyd and V. Jacobson. "Link-sharing and resource management models for packet networks." IEEE/ACM Transactions on Networking, vol.3, no.4, p. 365-86, Aug. 1995.

[15] Foundry Networks. ServerIron. http://www.foundrynet.com.

[16] M. Jones, J. Barrera III, A. Forin, P. Leach, D. Rosu, and M. Rosu. "An overview of the Rialto Real-Time Architecture." In Proc. of the Seventh ACM SIGOPS European Workshop, 249-256, Sept 1996.

[17] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barharn, D. Evers, R. Fairbairns, and E. Hyden. "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications." IEEE Journal on Selected Areas In Communications, Vol. 14, No. 7, pp. 1280-1297, September 1996.

[18] K. Li and S. Jamin. "A Measurement-Based Admission Controlled Web Server." In Proceedings of the IEEE Infocom Conference, Tel-Aviv, Israel, Mar. 2000.

[19] C. W. Mercer. "Operating System Resource Reservation for Real-Time and Multimedia Applications." Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, June 1997

[20] A. Neogi, A. Raniwala, T.Chiueh. "Phoenix: a low-power fault-tolerant real-time network-attached storage device." In Proceedings of ACM Multimedia 1999, Orlando, FL, October 1999.

[21] Packeteer. PacketShaper. http://www.packeteer.com.

[22] R. Pandey, J.F. Barnes, and R. Olsson. "Supporting Quality of service in HTTP Servers." In Proceedings of the 17th SIGACT-SIGOPS Symposium on Principles of Distributed Computing, p 247-256, June 1998.

[23] A. L. N. Reddy and J. Wyllie. "Disk Scheduling in Multimedia I/O System." In Proceedings of ACM Multimedia'93, Anaheim, CA, 225-234, August 1993.

[24] Rether Networks Inc. Internet Service Management Device. http://www.rether.com.

[25] J. Neih and M. S. Lam. "The design, implementation and evaluation of SMART: A scheduler for multimedia applications." In Proc. ACM Symposium on Operating Systems Principles, St.Malo, France, Oct. 1997.

[26] V.S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. "Locality-Aware Request Distribution in Cluster-based Net- work Servers." In Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA.

[27] A. Parekh. "A generalized processor sharing approach to flow control in integrated services networks." Ph.D dissertation, Massachusetts Institute of Technology, February 1992.

[28] P . Shenoy and H.M. Vin. "Cello: A Disk Scheduling Framework for Next-generation Operating Systems." In Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Madison, WI, June 1998.

[29] B. Verghese, A. Gupta, and M. Rosenblum. "Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors." In Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, Oct. 1998.

[30] C.S. Yang and M.Y. Luo. "Efficient Support for Content-Based Routing in Web Server Clusters." In Proceedings of the 2nd Usenix Symposium on Internet Technologies and Systems, Boulder, CO, Oct. 1999.

[31] Standard Performance Evaluation Corporation (SPEC). SPECWeb99 Benchmark. http://www.spec.org/osg/web99/.

[32] G.Banga and P.Druschel. "Measuring the Capacity of a Web Server." In Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems, Monterey, CA, Dec. 1997.