Performance Guarantees for Cluster-Based Internet Services

Chang Li   Gang Peng   Kartik Gopalan   Tzi-cker Chiueh
Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
Email: {changli, gpeng, kartik, chiueh}@cs.sunysb.edu

Abstract

As web-based transactions become an essential element of everyday corporate and commerce activities, it becomes increasingly important that the performance of web-based services be predictable and guaranteed even in the presence of wildly fluctuating input loads. In this paper, 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. We describe the design, implementation, and evaluation of a web request distribution system called Gage*, which can provide every subscriber with distinct guarantee on the number of* generic *web requests that are serviced per second regardless of the total input loads at run time.* Gage *is one of the first systems that can support QoS guarantee involving multiple system resources, i.e., CPU, disk, and network. The front-end request distribution server of* Gage *distributes incoming requests among a cluster of back-end web server nodes so as to maintain per-subscriber QoS guarantee and load balance among the back-end servers. Each back-end web server node includes a* Gage *module, which performs distributed TCP splicing and detailed resource usage accounting. Performance evaluation of the fully operational* Gage *prototype demonstrates that the proposed architecture can indeed provide the guaranteed request throughput for different classes of web accesses, even in the presence of excessive input loads. The additional performance overhead associated with QoS support in* Gage *is merely 3.06%.*

1 Introduction

First-generation Internet service infrastructure research focused on the development of scalable hardware/software architecture to support the growing demands on web-based services. As these services evolve from a novelty to an essential building block for enterprise operations and 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, commercial off-the-shelf clusters have become the de facto computing platform for scalable Internet services. As a consequence, service providers need a management and provisioning system to guarantee QoS on a per-customer basis on large-scale clusters. In this paper, we propose a general implementation framework for supporting guaranteed QoS on cluster-based Internet servers and demonstrate its feasibility with a web server cluster called *Gage*. *Gage* can guarantee each service subscriber with a distinct number of URL requests serviced per second, even in the presence of excessive input loads.

Consider a web hosting service provider having 100 customers, each with a different requirement on the size of web site and on the web access processing rate. 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 performance characteristic, on a single physical web server cluster. Here each logical web server is is guaranteed a pre-specified performance. More generally, the service provider needs a *virtualization* technology that can partition a given physical resource into multiple logical entities each having a distinct performance guarantee that is independent of others' performance and input loads. The resource being virtualized could potentially be a storage system, a compute cluster, or a network link. Our focus is 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 on the shared cluster and demand a guaranteed QoS according to a well-defined performance metrics. The web-based service in question can range from simple URL page accesses to E-commerce transactions, instant messaging sessions, and public key certificate authorization. As we show later, *Gage* is sufficiently general and flexible that it can be easily tailored to a new Internet service or platform with only minuscule modifications.

There are three components in the proposed cluster virtualization architecture: *request classification*, *request scheduling*, and *resource usage accounting*. Each customer subscribes to a pre-specified level of QoS, and is allocated a per-subscriber request queue. When an input request arrives, the classification module determines the per-subscriber queue in which the request is queued. Requests within a queue are serviced in a FIFO order. However, the request scheduling module determines which queue to service next to meet the QoS requirement of each subscriber. Different input requests, even for the same Internet service, may consume different amounts of 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, which in turn dynamically adjusts request scheduling according to both QoS requirements and run-time resource consumption. These three components together physically partition a given cluster into multiple subclusters each of which meets a particular subscriber's QoS requirement without interfering with one another. 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 resource.

The rest of this paper is organized as follows. Section 2 reviews related work on cluster-based Internet servers and real-time resource allocation and scheduling. Section 3 describes the system architecture and detailed design. Section 4 presents the effectiveness of *Gage*'s virtualization capability, based on empirical measurements on the first *Gage* prototype. Section 5 summarizes the main research results and ongoing work.

2 Related Work

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, we make a conscious effort to minimize the part of the proposed architecture that is service-specific or platform-dependent. In particular, we assume that the kernel of the OS running on individual cluster nodes should not be modified. In this section, we will discuss some earlier work on providing differentiated services on web server clusters.

In order to dispatch web service requests based on URL contents, we need to decouple request dispatching mechanism from the TCP connection setup. Previous systems used either TCP connection splicing [7, 17] or TCP hand-off [15, 3, 4]. TCP splicing requires the front-end to process TCP traffic in both directions and thus tends to become the system bottleneck. TCP hand-off uses connection state migration to move the TCP connection state either from the front-end node 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 for fully distributed implementation.

Cluster reserve [2] is a web server cluster abstraction that comes the closest to *Gage* in the goal of providing a predictable QoS to each web site sharing the physical cluster. As it chooses a two-level scheduling (inter-node and intra-node) for accurate resource allocation, cluster reserve makes 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* performs a cluster-wide global resource scheduling. As a result, *Gage*'s implementation is completely confined to a thin layer between the Ethernet driver and the IP layer and self-contained as a kernel module. 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. Also as a result of this simplicity, *Gage* can be readily ported to other Internet services without any changes to the service programs.

Most other efforts at providing quality of service in web hosting clusters are priority-based i.e, they do not provide *guaranteed* QoS. In other words, these approaches allow one service class to receive qualitatively better service than the other, but do not provide a quantitative bound on the service quality received. The systems in [1, 8] use user-level control to support two or more levels of web service priorities. Pandey et al. [11] use 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' current resource consumption. In their system, requests are either admitted or rejected but never re-ordered. Bhatti and Friedrich [5] 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. Similarly Lu et al. [20] modified the Apache web server to provide differentiated web services to provide relative delay guarantees. All the above approaches are based on user-level implementations that cannot have an accurate system resource usage information, and consequently the QoS support is mostly qualitative rather than quantitative.

Several research efforts have focused on QoS guarantee for specific type of resource, such as CPU [14, 10], disk [6, 16], and shared network link [13, 9]. However, different underlying resources that are under contention may require different accounting mechanisms, and may pose different trade-offs between QoS-oriented request scheduling and utilization efficiency-conscious request scheduling (such as disk scheduling or multiprocessor scheduling).

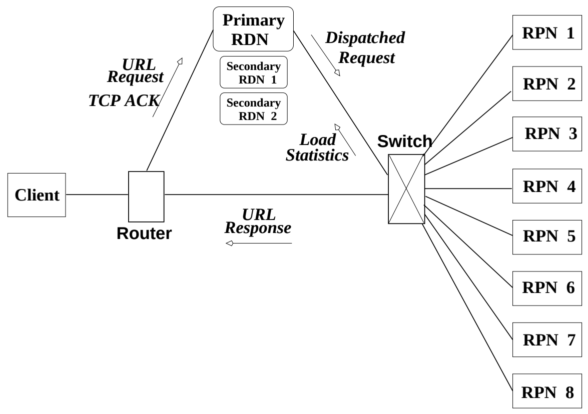

Figure 1: *Gage* consists of a front-end request distribution node (RDN) back-end request processing nodes (RPN).

3 The Gage System

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. RDN allocates a request queue for each service subscriber (and hence the corresponding web site). All the incoming requests are first routed through the RDN and buffered in the queue associated with the request's target web site. 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 request, it lets the subscriber's application service the request and eventually returns the requested page back to the requesting client *without going through the RDN*.

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 the time-consuming task in front-end processing such as TCP three-way hand-shaking. We now describe the nature of QoS guarantees and the individual service components of *Gage*.

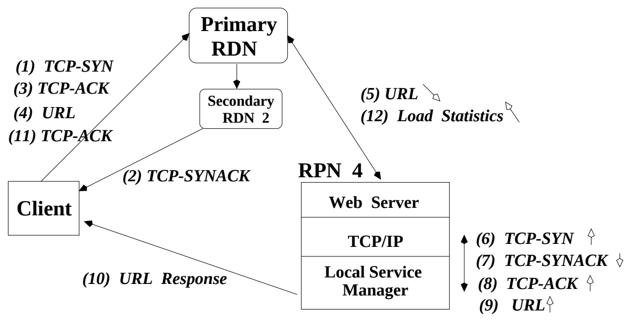

Figure 2: The the sequence of message exchanges involved between the time that a client sends a URL request and receives back a response.

3.1 High-level QoS Guarantees

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. The QoS is in terms of 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 this QoS metric, there are other possibilities such as response time, delay jitter and reliability. Generalization of the proposed implementation framework 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. Nevertheless, *Gage* serves as an essential building block for any QoS-guaranteed Internet service.

3.2 Distributed TCP Connection Splicing

URL requests are delivered via the HTTP protocol, which in turn is built on top of connection oriented TCP protocol. Hence, before the first payload packet can arrive at the RDN. a three-way hand-shake procedure is required to establish a new TCP connection. The RDN needs to select a back-end server to which a web access request can be dispatched and this decision is tied to the contents of the URL. Herein lies the dilemma: In order to service a web access request the selected back-end RPN needs to have a TCP connection with the client; however, to select an RPN in the first place, the RDN needs to examine the contents of URL that arrives only in the first payload packet carried by the TCP connection. *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*> . 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 needs 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 needs to be modified to fool the RPN's TCP/IP stack into thinking the packet is directed to that RPN. Figure 2 illustrates the set of network packets and the steps involved in processing a URL request from a client.

In summary, for every URL request, TCP connection splicing involves two TCP connection setups 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. At every RPN, the sequence number-address re-mapping of an incoming or outgoing packet is performed by a software module called the *local service manager* that resides above the Ethernet driver but below the IP layer. In addition, the setup of the second TCP connection (one between a client and its servicing RPN) is also performed by the RPN's local service manager. Finally, *Gage* can employ an asymmetric RDN cluster to collectively shoulder the processing load of setting up the first TCP connection for incoming URL requests. This ensures that more resources can be readily incorporated into the system architecture to alleviate the performance problem when it arises.

3.3 Request Classification

The primary RDN classifies an incoming packet into the 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. The front-end RDN itself handles packets of the first type 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 setup for TCP splicing is minimized. Upon receiving a packet of the second type, the primary RDN determines the subscriber web site to which the access request belongs according to the host-name part of the URL, and then places the request packet in the queue associated with the service subscriber.

For all other packets, the primary RDN simply acts 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, which includes source IP address, source port number, destination IP, and destination port number. After a URL request is dispatched to an 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 client are routed to the corresponding RPN.

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 made according to each subscriber's static resource reservation and dynamic resource usage. In making the "which RPN" decision, *Gage* attempts to maximize the system utilization efficiency by balancing the load on the RPNs, in other words, dispatching a request to the RPN with the least load. 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 periodically fed back to the RDN from the RPNs once every *accounting cycle*. *Gage*'s request scheduler executes a weighted round-robin algorithm and visits each subscriber's queue in a cyclic fashion. In each such visit, the scheduler 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 not empty according to their resource reservations.

Because different URL requests may consume different amount of system resources, the total resource consumption of the requests from a queue is not necessarily proportional to the number of requests that have been dispatched and have 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 have 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 assumes that the resource consumption of each dispatched request is equal to a weighted average resource consumption of the past requests that belong to the same queue.

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. RPN runs on the Linux kernel, which already keeps track of the CPU usage of each active thread. To collect the disk usage time of each thread, the disk driver records the amount of time that each physical disk I/O takes and charges it to the thread that issues the disk I/O request. Network bandwidth consumption is proportional to the size of the page and does not require extra measurements.

Gage's resource usage accounting model assumes that a set of dedicated processes are associated with each charging entity, i.e., a virtual web site in the case of web 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. This period is the accounting cycle. The model also allows the number of processes for a service to be varied dynamically. In general, by maintaining per-process resource usage accounting and associating a separate set of processes with each distinct subscriber, the above model can accurately account for each service subscriber's resource usage. *This is performed without making any assumption on the request/response formats of the Internet services* and thus matches the design goal of *Gage* perfectly. In particular, *Gage*'s resource accounting model automatically works for CGI programs without any additional mechanisms.

To support QoS, for each subscriber, the RDN maintains its current *balance*, and an *estimated resource usage array*. Each element of the array records the sum of predicted resource usage of all pending requests dispatched from a subscriber to a particular RPN. In addition, to support load balancing, the RDN maintains each RPN's current *capacity* and *estimated outstanding load*. The latter 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. Each accounting message from RPN includes the total and per-subscriber resource usage on that RPN in the previous accounting cycle. Every time an accounting message from an RPN arrives, the RDN subtracts the RPN's total resource usage from its capacity and estimated outstanding load. Furthermore, it reduces each subscriber's balance and estimated resource usage by the corresponding usage indicated in the accouting message.

3.6 Service-Specific Component

Gage is an instance of a general implementation framework for supporting QoS guarantee for Internet service. *Gage* makes the following assumptions that are specific to web access service. First, packet classification is tied to the host name part of URL requests. Secondly, 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. Finally, 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 I/Os.

4 Performance Evaluation

In the prototype implementation of *Gage* , both the RDN and RPN components are built as a thin software layer be-

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.

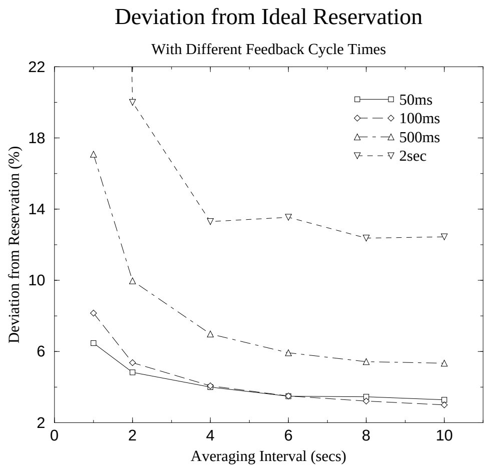

Figure 3: Deviation of actual resource usage from the ideal reservation under different accounting cycle times.

tween the Ethernet driver and the IP module under Linux. RDN performs connection table lookup, request classification, enqueuing, scheduling and packet forwarding. RPN performs local TCP connection setup, per-node and per-subscriber resource usage accounting, and sequence number-address re-mapping. The testbed is similar to Figure 1 except that there is no secondary RDN. There are eight back-end RPNs, each of which is an industrial 1U server consisting of 600-MHz Celeron CPU, 64-MByte memory, 10-GByte disk, and two Fast Ethernet interfaces. The primary RDN and the clients are all 450-MHz Pentium-III machines with 64 MBytes of main memory. The RDN has two Fast Ethernet interfaces. Clients, RDN, 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 negligible. Two types of workload were used in the following experiments - synthetic and realistic. We obtained the realistic workload by collecting the trace generated by SPECWeb99[18]. The method of load generation by the client is similar to that used in [19]. The clients load the trace from a file and issue requests to *Gage* at a *constant* rate. Unless otherwise mentioned, the workloads used in the following experiments are synthetic.

4.1 QoS under Excessive Input Loads

Performance Isolation: Since *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, 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 QoS requirement of each subscriber.

Spare Resource Allocation: *Gage*'s spare resource allocation policy is that "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 whom 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.

Deviation from Ideal Reservation: In *Gage*, the main source of instability in QoS guarantee is the inaccuracy of predicting the per-request resource usage. This inaccuracy is attributed to two factors. One is that the RDN obtains the resource usage information for each subscriber only periodically from the RPNs. This can result in RDN relying on information that may be old by a time lag of one period and thus the resulting resource usage predictions may not be 100% accurate. The other is that the resource consumption for 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 usage history.

We now demonstrate how these two factors impact the stability of *Gage*'s QoS guarantee. In each experiment, we measure the deviation of resource usage by each subscriber from its reservation over different time intervals, and then compute an overall average among all subscribers. The input for the first experiment is a constant synthetic workload with each request accessing a file of the size of 6 KBytes. The input to the second 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. At the data point corresponding to a 2-second accounting cycle and 1-second averaging interval, the deviation is above 100% because the accounting cycle is twice as long as the averaging interval and the resource usage that RDN observes is either *0* or around *twice the reservation*. 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 with a periodicity no larger than 500 msec. To evaluate the stability of *Gage*'s QoS guarantee, we also ran an experiment using a representative web workload trace derived from SPECWeb99. The result shows that under realistic web access workloads, the QoS deviation from reservation is less than 5% with the averaging interval being 4 sec or higher.

4.2 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 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 and is shown in the first three columns in Table 3. The latter is due to forwarding and address-sequence re-mapping for each packet and is shown in the last three columns in Table 3. Compared to the time a web server takes to service a HTTP request, which is in the range of several to tens of miliseconds, the additional latency that *Gage* introduces is relatively small and negligible. It takes 56.7 μsec for connection setup and address-sequence number remapping, assuming each request consists of 5 data-ACK packet pairs. Under a load of 540 GRPS that one RPN can sustain (shown in next section), the total overhead imposed on a RPN is less than 56.7 x 540 = 30,618 μsec, or only under 3.06% of a RPN's CPU capacity.

4.3 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 is increased from 1 to 8. The throughput grows linearly from about 540 requests/sec to around 4800 requests/sec 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%. To project the maximal throughput that one RDN (PIII 450MHz) could support, we further measured how the CPU utilization of the RDN varies as the throughput increases. The result 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 increases exponentially as the throughput advances to around 4800 requests/sec. The utilization leap is due to the overloaded network subsystem, which results in an increase in the interrupt handling time. 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 14,000 to 15,000 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. Hence it greatly simplifies 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 guarantees that *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. The overhead it introduces, in terms of its impacts on the per-request latency and overall system throughput, is 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

Connection setup (μsec)		Packet classification (μsec)	Packet forwarding (μsec)	Remapping (μsec)	
RDN	RPN			incoming	outgoing
29.3	27.2	3.0	7.0	1.3	4.6

Table 3: Per-connection overhead (column 1-3) and per-packet overhead (column 4-6) in *Gage*.

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.

Acknowledgement

We would like to thank the anonymous refrees for their helpful comments. This research is supported by NSF awards ANI-9814934, ACI-9907485, and ACI-0083497, USENIX student research grants, as well as fundings from National Institute of Standards and Technologies, Siemens, and Rether Networks Inc.

References

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

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

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

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

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

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

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

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

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] 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 Proc. of the 8th Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA.

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

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

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

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

[20] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. "A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers" In Proc. of IEEE Real-Time Technology and Applications Symposium, Taipei, Taiwan, Jun. 2001.