IN-KERNEL MECHANISMS FOR ADAPTIVE CONTROL OF OVERLOADED WEB SERVERS

*Thiemo Voigt** *Renu Tewari* *Ashish Mehra*

Network Architecture and Services
IBM T.J. Watson Research Center
Hawthorne, NY 10532
{thiemo, tewarir, mehraa}@watson.ibm.com

ABSTRACT

The increasing number of Internet users and innovative new services such as e-commerce are placing new demands on Internet servers, for example web servers. It is becoming essential for Internet servers to be highly available, have fast response times, and provide continuous service during overload at least to preferred customers. It is necessary, therefore, to protect Internet servers from overload since during server overload clients experience increased response times and service failures.

In this paper we present a kernel-based architecture that protects Internet servers against overload by controlling the amount and rate of work entering the system. Our basic control algorithm limits the incoming TCP SYN requests based on connection attributes. By dropping non-compliant connection requests, the server can provide continuous service to preferred clients even under overload. We present a second mechanism that reorders the listen queue of a server socket based on the priorities of the incoming connection requests. Our experiments show that both mechanisms enable service differentiation during overload. We describe an adaptive architecture that uses these mechanisms to dynamically protect web servers from overload.

1 INTRODUCTION

The current Internet architecture is moving towards application service providers and other Web hosting services that co-host multiple customer applications on the same server farm or large servers. In these settings a large increase in demand can bring down multiple sites simultaneously and affect the performance and access of a large number of users. Combined with the fact that these sites are increasingly geared towards e-commerce, the need for performance isolation and continuous operation under overload becomes critical. However, a simple "one size fits all" solution is not adequate. Since the co-hosted sites or the clients belonging to a site could be governed by different service level agreements (SLA), it is essential to provide service differentiation based on the attributes, for example client identity or service identity, of the incoming request.

Traditionally, commercial operating systems have provided only marginal protection from overload. During overload conditions the operating system may experience thrashing where little or no useful work is done. This results in the clients observing higher response times, connection timeouts, failures and eventually loss of service.

To avoid overload, the server needs to deploy some form of admission control that limits the amount of work entering the system. This control can be applied at (i) the application level (e.g., in the web server), (ii) in the kernel, or (iii) in an external appliance/router in front of a server cluster. A front-end appliance/router-based overload protection provides coarse-grained control of the load seen by a server or a cluster. One limitation of such an approach is that the front-end box may hold insufficient or stale information about the state of the server. Consequently, it does not provide rapid response to overload. Secondly, the front-end box is not in any adaptation loop governed by the applications running on the server machine.

An alternative approach is to enable the applications to do their individual admission control. Although this achieves application level adapta-

*The author is a graduate student at Uppsala University, Sweden. He is also affiliated with the Swedish Institute of Computer Science. He can be reached at thiemo@sics.se

tion it requires modifications to existing applications. Secondly, various system resources have already been allocated to the requests before the application control comes into play.

The need, therefore, is to: deploy mechanisms that benefit any server application (e.g. WWW, ftp, mail); enable controls to be effective as early as possible; have the ability to utilize the latest and complete information about the resource usage; and adapt to the nature and demands of the applications and any governing service agreements. A kernel-based mechanism for admission control and service differentiation at the server supports the above features and can co-exist with the controls at the front-end or in applications. A useful side-effect of server-based kernel controls is that in a typical web-site architecture the machine running the web server is more aware of the load on the back-end servers than a front-end box. The kernel controls can either be made completely transparent to the applications or one can allow applications to adapt them as and when required.

In this paper, we present a kernel-based architecture that controls the amount and rate of work entering the system. We have implemented the architecture on AIX using the framework of an existing QoS-architecture [1]. Since most web servers receive their requests over HTTP/TCP connections, the mechanisms are located in the network subsystem of the base kernel. The basic control mechanism [2] limits incoming TCP SYN requests using a token bucket based policer. The incoming requests can be aggregated based on the connection attributes (for example protocol number, source and destination IP address and port), and policed as a group to a given rate. In order to provide service differentiation we provide two new mechanisms: (i) TCP SYN policing to adapt the policing profile for individual aggregates and (ii) prioritized listen queue to prioritize requests between aggregates based on the connection attributes. We present experiments to show that these mechanisms effectively limit the number of accepted connections and provide better service to preferred clients under overload conditions.

Our paper is organized as follows: In the next section we introduce the problem and present an overview of the architecture. In Section 3 we describe the mechanism that polices incoming TCP SYN requests and show experimental results. In the following section we present the prioritized socket queue and some results. Section 5 discusses further issues and presents a more complete architecture to solve the problem discussed in this paper. Before concluding, we discuss related work in Section 6.

2 ARCHITECTURE

The main goal of our kernel mechanisms is to control the number and rate of accepted connections and provide better service to preferred clients. In the absence of such controls all connections are adversely affected during overload. The greedy behavior of a single client or groups of clients reduces the throughput observed by all clients. Figure 1 shows the achieved throughput of different client connections. One set of clients are accessing a preferred customer site (e.g., in the experiment we replay a server trace from the department store Macy) and other clients are making continuous requests to other contents. The throughput of the preferred Macy's clients falls as the load generated by the other clients increases.

In order to shield the preferred clients from losing their throughput we provide kernel mechanisms that are placed in the networking stack. Our basic architecture consists of two kernel mechanisms shown in Figure 2. The first mechanism called *TCP SYN policing* limits the number of connections to the server by policing incoming TCP SYN packets using a token bucket. The token bucket is defined by the parameters $< rate,\ burst >$, where *rate* is the average number of SYN requests admitted per second and *burst* is the maximum number of SYNs accepted at one time. The incoming connections are aggregated based on specified rules/filters and the token bucket parameters are assigned to each aggregate. Before a socket for a new connection request is created, we check the TCP SYN packet for compliance against its token bucket profile. If the SYN packet is compliant, a new socket is created and inserted in the partial listen queue, otherwise, the SYN packet is silently dropped.

Our second mechanism *prioritized listen queue* reorders the listen queue of a server process based on pre-defined priorities of the incoming connections. When TCP connections are established with the completion of the three way handshake [3], they are moved from the partial listen queue to the listen queue. We insert the socket at the position corresponding to its priority in the listen queue. Since the server process always removes the head of the listen queue when calling *accept()*, this approach provides better service, i.e. lower delay and higher throughput, to connections with higher priority. In order to associate connections with a token bucket and a priority we have filters/rules. Each rule is a four-tuple consisting of

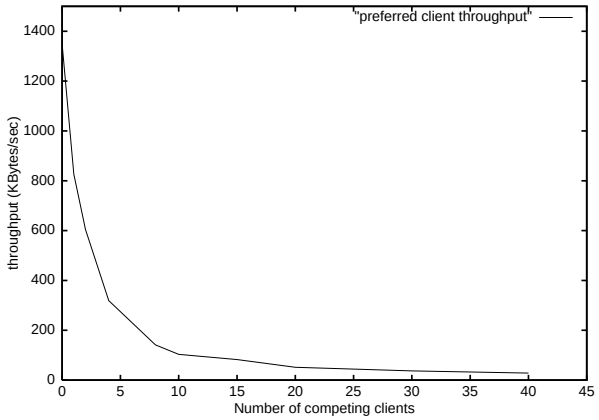

Figure 1: Macy-trace. The x-axis shows the number of competing clients. The y-axis shows the throughput of the Macy's client.

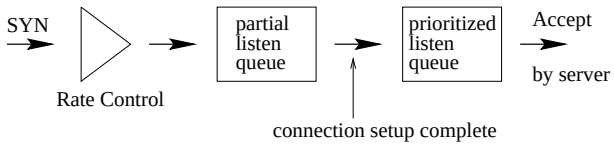

Figure 2: Architecture with the two kernel mechanisms

(dst IP,dst port,src IP,src port)	(r,b)	prio
(*, 80, *, *)	(300,5)	3
(*, 80, 10.1.1.1, *)	(100,5)	2
(*, 80, 10.1.1.1, 654)	(100,5)	1
(*, 80, 10.1.1.5, *)	(10,1)	3

Table 1: Filters for TCP SYN policing and priority

local IP address, local port, remote IP address, and remote port. An example is shown in Table 1. Here we have four rules: The first rule is the most general of the rules and applies to the process listening at the local port 80 on all network interfaces (usually this is a Web server). It specifies that all connections to the server are rate-controlled with a token bucket with rate 300 requests per second and a burst of 5. The priority is three, that is set to be the default lowest priority. The second rule gives the client IP address 10.1.1.1 a higher priority (two), but a lower rate. Clients from host 10.1.1.1 and port 654 get priority one, the highest priority in our scheme. Clients from host 10.1.1.5 receive lower priority and are only accepted at a small rate.

3 TCP SYN POLICING

The basic functionality of TCP SYN policing is to limit the rate at which connection requests from clients are accepted. Incoming SYNs are policed using a token bucket based traffic profile defined by an average rate and a burst. TCP SYN packets that are not compliant are silently dropped. The requesting client will time-out waiting for a SYN ACK and retry again with an exponentially increasing time-out value. We do not send a TCP RST to reset the connection since that would indicate an abort from the server. Also some clients try to send another SYN immediately after a RST instead of aborting the connection.

Note that we drop non-compliant SYNs *before* a connection is established; in fact, even before a socket for the new connection is created. Thus, our approach invests only a small amount of overhead into requests that are dropped. This is very important to avoid server overload.

To provide service differentiation, the connection requests are aggregated based on filters (or rules) and each aggregate has a separate token bucket-based traffic profile. To avoid dropping TCP SYNs blindly, we check if the SYN packets match a rule and then apply the rate and burst of the rule. By doing this we can provide both service differentiation and overload protection.

3.1 Experiment

We conducted experiments using Webstone 2.5 [4] clients and server traces to verify that TCP SYN policing works as desired. Our testbed consists of

an Apache web server [5] running on a 375 MHz IBM RS/6000 machine that runs AIX; the client machines are three 550 MHz IBM Pentium IIIs running Linux. The machines are connected together using a 100 BaseT Ethernet switch.

In this experiment we show how we can use TCP SYN policing to protect a preferred client against denial-of-service attacks caused by a high request rate from other clients. As in the experiment in the previous section, one host replays the Macy's trace file representing preferred customers. For the competing clients we use five machines running Webstone, each with 50 clients. All clients request the same file of size of 8 KBytes. We chose 8 KBytes since the size of a typical HTTP transfer is between five and 13 KBytes [6]. We need a large number of Webstone clients because when a TCP SYN packet is dropped (since it was non-compliant), TCP's exponential backoff algorithm causes the Webstone client to refrain from making a new request until the backoff timer expires. In order to have enough clients with outstanding requests, we need a large number of Webstone clients. Note that this problem is caused by the closed-loop architecture of Webstone. If we had a more natural client population, we would not see this problem.

In the experiments we do not vary the bucket size (burst). The burst is set to 100. For a lower burst, too many Webstone clients back off and do not get any service.

Without any policing mechanisms the Macy client receives a low throughput of about 6 KBytes/sec. When we use TCP SYN policing to lower the acceptance rate of the Webstone clients we expect that the throughput for the Macy's client will increase. The results are shown in Figure 3.

As expected, the throughput for the Macy customers increases from 100 KBytes/sec to almost 800 KBytes/sec when we lower the acceptance rate for the Webstone clients from 300 requests/sec to 25 requests/sec. The experiment demonstrates that we can protect a preferred client by rate-controlling the TCP SYN requests for all other clients.

The scenario above uses TCP SYN control to protect preferred clients against overload caused by too many connection requests from misbehaving or greedy clients. When dynamic objects are requested, web servers can become overloaded due to the extra CPU load per CGI process that is spawned for each dynamic request. TCP SYN policing can also be used to address this problem. In a real system we need to adapt the policing rate

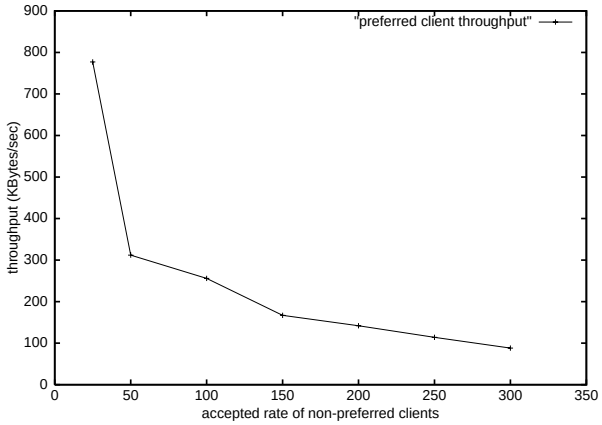

Figure 3: TCP SYN policing to protect Macy's customers. The x-axis shows the rate at which we limit other clients. The y-axis shows the throughput that Macy's client receives.

to the resource consumption of the current workload.

TCP SYN policing works very well when we know all the clients and their request patterns. However, it is hard to determine the right parameters that allow us to give good service to preferred clients without underutilizing the server. In particular, it is difficult to determine accurately who the misbehaving group of clients are. Secondly, finding the right parameters is tricky. For example, if the burst is too small, we might reject connection requests unnecessarily. Furthermore, when the burst size is smaller than the number of parallel connections opened by an HTTP/1.0 browser, a client might not be able to retrieve all the embedded objects in a HTML page since the requests for these objects usually arrive in a burst after the client has received the initial page. With a large burst size, however, a malicious client can use the burst to create a short-term overload at the web server.

TCP SYN policing can also be used to protect servers against overload by limiting the total number of accepted requests regardless of the client's identity. Another usage is to defend against certain denial-of-service attacks on the web server itself. As we do not admit every connection request, it is not possible for a potential attacker to fill up the partial listen queue of our web server.

4 PRIORITIZED LISTEN QUEUE

A problem with using only TCP SYN policing is to identify the greedy or misbehaving clients and assign them a proper rate. Instead if only the pre-

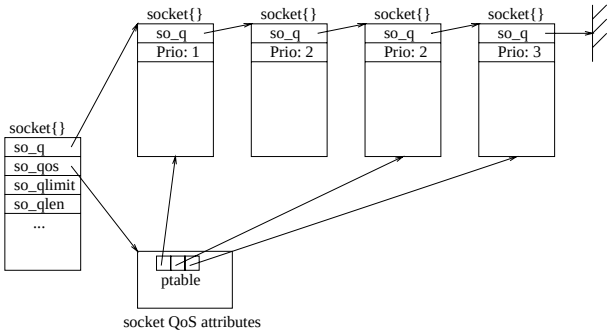

Figure 4: Implementation of the priority listen queue

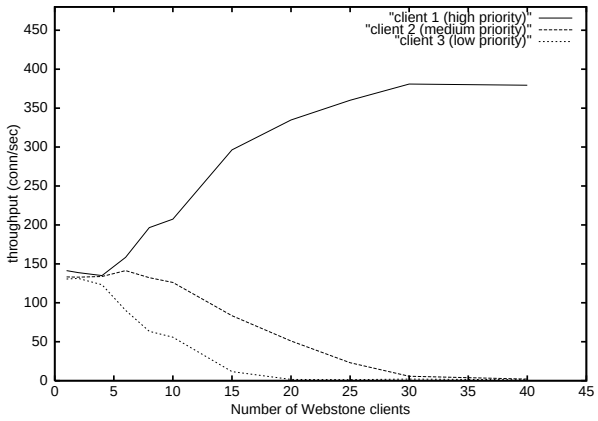

Figure 5: Priority with 50 Apache server processes. The x-axis shows the number of Webstone clients for each priority class. The y-axis shows the throughput that each priority class receives.

ferred clients are known it is simpler to give them a higher absolute priority. Our second mechanism prioritizes connections by reordering a server's listen queue. When calling *accept*(), the server process always removes the head of the listen queue. In order to allow the web server to process important requests before unimportant ones, priorities are associated with filters as shown in Table 1. This way, connections can be classified into different *priority classes*. We sort the listen queue after priority with the connections of the highest priority class at the head of the listen queue.

4.1 Implementation

Figure 4 shows the implementation of the prioritized listen queue. In a special data structure that maintains socket attributes related to QoS, we store an array of *priority pointers*. Each priority pointer points to the *last* element of the corresponding priority class. This makes insertion of new sockets efficient. We only need to insert the socket behind the one pointed to by the corresponding priority pointer and update the priority pointers.

4.2 Experiments

To evaluate the priority scheme, we use the same testbed as in Section 3.1.

A test program running on the web server makes one reservation for each client. Client 1 gets priority 1 (the highest priority), while client 2 and client 3 get the priorities 2 and 3. In our experiments, we run a separate Webstone 2.5 program on every client machine. All Webstone programs request the same file of size 8 KBytes. We vary the number of Webstone clients. We expect client 1 to achieve the highest throughput, followed by client 2 and client 3.

In the first experiment, the Apache server is configured to spawn a maximum of 50 server processes. The results are shown in Figure 5. For a small number of Webstone clients, all clients achieve about the same throughput. When we have a small number of Webstone clients, there are always free processes to handle incoming requests. Thus, the listen queue is kept short and almost no reordering of the listen queue occurs.

The higher the number of Webstone clients, the higher the throughput seen by client 1, while the throughput that clients 2 and 3 receive decreases. With an increasing number of Webstone clients the length of the listen queue increases. Later arriving connections from high-priority clients are inserted in the listen queue on the positions in front of connections from low-priority clients. Hence, low-priority clients will not be served until there are no high-priority connections in the listen queue. Figure 5 shows that if we have more than 30 Webstone clients in each machine only the high-priority clients are served and the lower-priority clients are starved.

In the next experiment we decrease the number of Apache server processes to 20. If we have fewer processes, the length of the listen queue will increase. Hence, we expect that the low-priority clients will be starved with fewer Webstone clients than in the previous experiment. The results shown in Figure 6 show that this is the case and that the mechanism works as expected.

The priority-based approach enables us to give low delay and high throughput to preferred clients independent of the requests or request patterns of

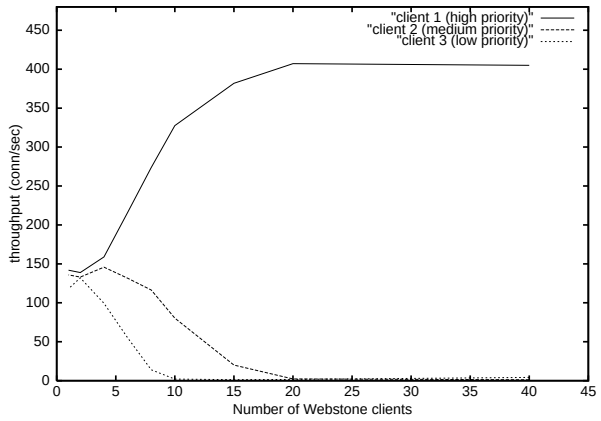

Figure 6: Priority with 20 Apache server processes. The x-axis shows the number of Webstone clients for each priority class. The y-axis shows the throughput that each priority class receives. The low-priority clients are starved with fewer clients than when we have 50 server processes.

other clients. However, when we have too many clients whom we will grant good service, we need many priority classes. In general, this approach is useful when there are a small number of priority classes.

5 DISCUSSION AND FUTURE WORK

In the previous sections we have shown that both TCP SYN policing and the reordering of the socket listen queue are useful mechanisms for overload protection and service differentiation. The mechanisms work best under certain circumstances: Due to its ability to enforce a maximum request rate, TCP SYN policing works very well when we know the misbehaving or greedy clients as we can limit the requests accepted from them.

The prioritized listen queue is useful when we know the preferred clients. It enables better service for clients with high priority but can lead to starvation of clients with low priority. A useful technique is to combine the two approaches: we give preferred clients high priority, but at the same time we use TCP SYN policing to limit their maximum rate and thus prevent starvation of clients with lower priority.

We have conducted experiments to show that the combination of these two mechanism works as expected. Table 2 shows the results for experiments with three clients, client 1 to client 3 having priorities 1 to 3. All clients are Webstone programs. Client 2 and client 3 have 30 Webstone clients while client 1 has 150 Webstone clients spread over three different hosts. We need this large number to compensate TCP's exponential backoff algorithm as discussed earlier. When we do not police client 1's TCP SYNs, client 2 and client 3 are totally starved. When we rate-control client 1 with a rate of 300 requests/second and a burst of 300, client 2 achieves a throughput of 78.6 connections per second, while client 3 is no longer starved. A further limitation of client 1's rate and burst increases the lower priority clients' throughput additionally.

Throughput (conn/sec)			
client	(rate, burst) of client 1		
	none	(300,300)	(200,200)
client 1	381	306	196
client 2	0	78.6	180
client 3	0	4.1	13

Table 2: Use of TCP SYN policing of a high-priority client to avoid starvation of other clients. The table shows the throughput (connections per second) for each client depending on client 1's rate and burst.

So far, we have only discussed the kernel mechanisms. As highlighted in the previous section, a production system needs an adaptation facility, both to adapt to load changes and changes of the request patterns of the clients. The system architecture of a production system is shown in Figure 7. In addition to the web server and the kernel policer, this architecture contains an adaptation agent and a kernel statistics module as well as an API. The API exports the mechanisms that we have deployed in the kernel to the user level. The API can be used in two different ways: (i) by an adaptation agent with the application being unaware of the existence of the kernel mechanisms or (ii) by the application itself (depicted by the dashed line between the web server and the API in Figure 7).

Based on statistics obtained from the kernel statistics module and the web server, the adaptation agent modifies the rates and bursts of the current rules.

We are currently developing such an adaptive agent. Its task is to modify the rules to detect and prevent overload based on the system usage parameters such as load/utilization and request patterns obtained from the statistics module and the web server. The adaptive agent maintains statistics for the previous n time intervals of:

- the observed CPU utilization,

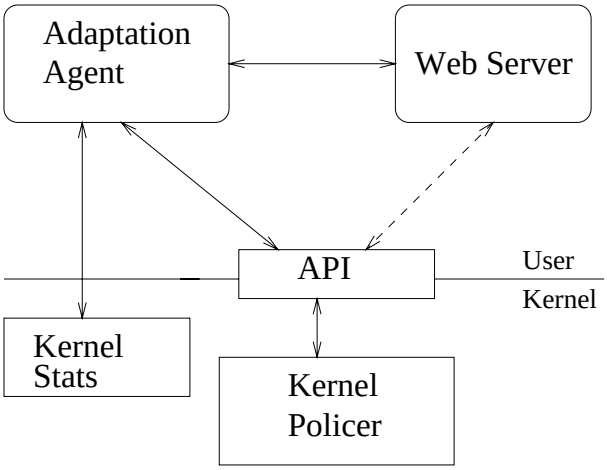

Figure 7: Architecture with adaptation agent

- the number of accepted connections,
- the request mix obtained from the web server module,
- the SLAs.

The expected load for the next time interval is computed based on an exponential average of the previous n time intervals. Given the new load, the agent computes new values for the TCP SYN policer's burst size and rate.

6 RELATED WORK

Several research efforts have focused on admission control and service differentiation in web servers [7], [8], [9], [10], [11] and [12].

Almeida *et. al.* [11] provide differentiated levels of service to clients depending on which customers web pages they access. While in their approach the application, i.e. web server, determines the priority of the request by parsing the request to find out to which customer the requested web page belongs, our mechanisms reside in the kernel. However, our mechanisms are not based on the requested page, but on the client issuing the request. Our mechanisms could extend Almeida's approach to provide overload protection and protection against greedy or misbehaving clients.

Bhatti and Friedrich [12] have developed an architecture called *WebQoS* to provide differentiated services. The key component of their architecture is a middleware located between the kernel and the web server. In their approach, the web server receives requests from the middleware and not directly from the kernel. The middleware provides service differentiation by classifying requests and placing them in different queues. In contrast to our scheme, where we prioritize the listen queue, the requests share the listen queue which is a FIFO queue. This way, low priority requests can delay requests with higher priority. The middleware also deploys admission control based on the number of requests queued. However, in their work admission control is done after a significant amount of work such as the establishment of a TCP connection has been invested in a connection.

Banga and Druschel [13] provide differentiated services using an operating system abstraction called *resource containers*. Resource containers enable the operating system to account for and control the consumption of resources for the processing of a request. They can also shield preferred clients from malicious or greedy clients by assigning them to different containers.

Scout [14], Rialto [15] and Nemesis [16] are operating systems whose design enables them to account applications for all the resources they use and restrict the resources granted to each application. This way, these operating systems can provide isolation between applications as well as service differentiation between clients. Nemesis is also able to distribute the available transmit bandwidth among different connections [17]. However, there is a significant amount of work involved to port applications to these operating systems.

7 CONCLUSIONS

We have presented two in-kernel mechanisms for adaptive control of overloaded Internet servers. Both mechanisms work with any TCP server application.

TCP SYN policing limits the number of incoming TCP SYN requests using a token bucket policer. Incoming TCP SYNs are aggregated based on specified filters and the corresponding policer drops non-conforming TCP SYNs. Our experiments show that this mechanism can protect preferred clients against denial-of-service caused by requests of well-known sites by enforcing a maximum acceptance rate on the latter.

Our second mechanism is a prioritized reordering of a socket's listen queue. Sockets receive a priority based on the connection's aggregate's attributes. This mechanism gives good service, i.e. low delay and high throughput, to clients with high priority. Under high load with a large number of requests from high priority clients, low-priority clients can be starved. To avoid starving of lower-priority requests, the two mechanisms can be com-

bined. We can enforce maximum request rates on clients with higher priority using TCP SYN policing. Our experiments show that the combination of the two mechanisms leads to the desired effect.

The presented mechanisms can be used to provide adaptive control of overloaded web servers. Since the mechanisms are deployed in the operating systems kernel, they are efficient and can provide fast response to overload.

REFERENCES

[1] T. Barzilai, D. Kandlur, A. Mehra, and D. Saha, "Design and implementation of an rsvp based quality of service architecture for an integrated services internet," *IEEE Journal on Selected Areas in Communications*, vol. 16, pp. 397–413, Apr. 1998.

[2] O. Stöckle, "Overload protection and qos differentiation for co-hosted web sites." Diploma Thesis, ETH Zürich, July 1999.

[3] G. Wright and W. Stevens, *TCP/IP Illustrated, Volume 2*. Addison-Wesley Publishing Company, 1995.

[4] "webstone." http://www.mindcraft.com.

[5] "apache." http://www.apache.org.

[6] M. F. Arlitt and C. l. Williamson, "Web server workload characterization: The search for invariants," in *Proc. of ACM Sigmetrics*, Apr. 1996.

[7] T. Abdelzaher and N. Bhatti, "Web server qos management by adaptive content delivery," in *Int. Workshop on Quality of Service*, June 1999.

[8] L. Ckerkasova and P. Phaal, "Session based admission control: a mechanism for improving the performance of an overloaded web server," tech. rep., Hewlett Packard, 1999.

[9] V. Kanodia and E. Knightly, "Multi-class latency-bounded web servers," in *Intl. Workshop on Quality of Service*, June 2000.

[10] K. Li and S. Jamin, "A measurement-based admission controlled web server," in *Proc. of INFOCOMM*, Mar. 2000.

[11] J. Almeida, M. Dabu, A. Manikutty, and P. Cao, "Providing differentiated levels of service in web content hosting," in *Proc. of Internet Server Performance Workshop*, Mar. 1999.

[12] N. Bhatti and R. Friedrich, "Web server support for tiered services," *IEEE Network*, Sept. 1999.

[13] G. Banga, P. Druschel, and J. 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 (OSDI)*, (New Orleans, USA), Feb. 1999.

[14] D. Mosberger and L. L. Peterson, "Making paths explicit in the scout operating system," in *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 153–167, Oct. 1996.

[15] M. B. Jones, J. S. Barrera III, A. Forin, P. J. Leach, D. Rosu, and M. Rosu, "An overview of the Rialto real-time architecture," in *ACM SIGOPS European Workshop*, pp. 249–256, Sept. 1996.

[16] I.M.Leslie, D.McAuley, R.Black, T.Roscoe, P.Barham, D.Evers, R.Fairbanks, 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, pp. 1280–1297, Sept. 1996.

[17] T. Voigt and B. Ahlgren, "Scheduling TCP in the Nemesis operating system," in *IFIP WG 6.1/WG 6.4 International Workshop on Protocols for High-Speed Networks*, Aug. 1999.