RC 21925 (98589) 5 January 2001 **Computer Science**

IBM Research Report

Kernal Mechanisms for Service Differentiation in Overloaded Web Servers

Thiemo Voigt, Renu Tiwari, Douglas Freimuth
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Ashish Mehra
iScale Networks

Kernel Mechanisms for Service Differentiation in Overloaded Web Servers

Thiemo Voigt, Renu Tewari, Douglas Freimuth, Ashish Mehra

Abstract— **The increasing number of Internet users and innovative new services such as e-commerce are placing new demands on Web servers. It is becoming essential for Web servers to provide performance isolation, have fast recovery times, and provide continuous service during overload at least to preferred customers. In this paper, we present the design and implementation of a kernel-based architecture that protects Web servers against overload by controlling the amount and rate of work entering the system. We present three mechanisms that provide admission control and service differentiation based on connection and application level information. Our basic admission control mechanism, *TCP SYN policing*, limits the acceptance rate of new requests based on the connection attributes. The second mechanism, *prioritized listen queue*, supports different service classes by reordering the listen queue of a server socket based on the priorities of the incoming connection requests. Third, we present *URL-based connection control* that uses application-level information such as URLs and cookies to define priorities and rate control policies.**

We have implemented these mechanisms in AIX 5.0. Through numerous experiments we demonstrate the effectiveness of these mechanisms in achieving the desired degree of service differentiation during overload. We also show that the kernel mechanisms are more efficient and scalable than application level controls implemented in the Web server.

I. Introduction

Application service providers and Web hosting services that co-host multiple customer sites on the same server cluster or large SMP are becoming increasingly common in the current Internet infrastructure. The increasing growth of e-commerce on the web means that any server down time that affects the clients being serviced will result in a corresponding loss of revenue. Additionally, the unpredictability of flash crowds can overwhelm a hosting server and bring down multiple customer sites simultaneously, affecting the performance of a large number of clients. It becomes essential, therefore, for hosting services to provide performance isolation and continuous operation of the hosted customer sites under overload conditions.

Each of the co-hosted customers sites or applications may have different quality-of-service (QoS) goals based on the price of the service and the application requirements. Furthermore, each customer site may require different services during overload based on the client's identity (preferred gold client) and the application or content they access (e.g., a client with a buy order vs. a browsing request). A simple threshold based request discard policy (e.g., a TCP SYN drop mode in commercial switches/routers discards the incoming, oldest or any random connection [1]) to delay or control overload is not adequate as it does not distinguish between the QoS requirements of the customer sites. For example, it would be desirable that requests of non-preferred customer sites/clients be discarded first. Such QoS specifications are typically negotiated in a service level agreement (SLA) between the hosting service provider and its customers. Based on this governing SLA, the hosting service providers need to support service differentiation of the incoming request based on client attributes (IP address, session id, port etc.), server attributes (IP address, type), and application information (URL accessed, CGI request, cookies etc.).

In this paper, we present the design and implementation of kernel mechanisms in the network subsystem for Web servers that provide admission control and service differentiation during overload based on the customer site, the client, and the application layer information.

One of the underlying principles of our design was that it should enable "early discard", i.e., if a connection needs to be discarded it should be done as early as possible in its lifetime, before it has consumed a lot of system resources [2]. Since a web server workload is generated by incoming network connections we place our control mechanisms in the network subsystem of the server OS in different stages of the protocol stack processing. To balance the need for early discard with that of an informed discard, where the decision is made with full knowledge of the content being accessed, we provide mechanisms in the networking stack to enable content-based admission control.

Thiemo Voigt is with the Swedish Institute of Computer Science (SICS) and Uppsala University, Sweden. E-mail: thiemo@sics.se. This work was done when the author was an intern at IBM T.J. Watson

Renu Tewari and Douglas Freimuth are with IBM T.J. Watson Research Center, Hawthorne, NY, USA. E-mail: {tewarir,freimuth}@watson.ibm.com

Ashish Mehra is currently at iScale Networks. E-mail ashish@iscale.net

Our second principle was to introduce minimal changes to the basic operation of the networking subsystem in commercial operating systems which typically implement a BSD stack. There have been prior research efforts that modify the architecture of the networking stack to enable stable overload behavior [3]. Other researchers have developed new operating system architectures to protect against overload and denial of service attacks [4]. Some "virtual server" implementations try to sandbox all resources (CPU, memory, network bandwidth) according to administrative policies and enable complete performance isolation [5]. Our aim in this design, however, was not to build a new networking architecture but to introduce simple controls in the existing architecture that could be just as effective.

The third design principle was to implement mechanisms that can be deployed both on the server as well as outside the server in layer 4 or 7 switches that perform load balancing and content based routing for a server farm or large cluster [6]. Such switches have some form of overload protection mechanisms that typically consists of dropping a new connection packet (or the oldest packet in the queues, or some random new connection packet) when a load threshold is exceeded. For content-based routing the layer 7 switch functionality consists of terminating the incoming TCP connection to determine the destination server based on the content being accessed, creating a new connection to the server in the cluster, and splicing the two connections together [7]. Such a switch has access to the application headers along with the IP and TCP headers. The mechanisms we built in the network subsystem can easily be moved to the front-end switch to provide service differentiation based on the client attributes or the content being accessed.

There have been proposals to modify the process scheduling policies in the OS to enable preferred web requests to execute as higher priority processes [8]. These mechanisms, however, can only change the relative performance of higher priority requests; they do not limit the requests accepted. Since the hardware device interrupt on a packet receive and the software interrupt for packet protocol processing can preempt any of the other user processes [3] such scheduling policies cannot prevent or delay overload. Secondly, the incoming requests already have numerous system resources consumed before any scheduling policy comes into effect. Such priority scheduling schemes can co-exist with our controls in the network subsystem.

An alternate approach is to enable the applications to provide their individual admission control mechanisms. Although this achieves application level control it requires modifications to existing legacy applications or specialized wrappers. Application controls are useful in differentiating between different clients of an application but are less useful in preventing or delaying overload across customer sites. More importantly, various server resources have already been allocated to a request before the application control comes into effect, violating the early discard policy. However, the kernel mechanisms can easily work in conjunction with application specific controls.

Since most web servers receive requests over HTTP/TCP connections, our controls are located in three different stages in the lifetime of a TCP connection.

- The first control mechanism, *TCP SYN policing*, is located at the start of protocol stack processing of the first SYN packet of a new connection and limits acceptance of a new TCP SYN packet based on compliance with a token bucket based policer.
- The next control, *prioritized listen queue*, is located at the end of a TCP 3-way handshake, i.e., when the connection is accepted and supports different priority levels among accepted connections.
- Third, *HTTP header-based connection control*, is located after the HTTP header is received (which could be after multiple data packets) and enables admission control and priority values to be based on application-layer information contained in the header e.g., URLs, cookies etc.

These three mechanisms provide a rich set of options for system administrators or application agents to control and adapt the behavior of the server during overload.

We have implemented these controls in the AIX 4.3.3 and 5.0 kernels as a loadable kernel module using the framework of an existing QoS-architecture [9]. The existing QoS architecture on AIX supports policy-based outbound bandwidth management [10]. These techniques are easily portable to any OS running a BSD style network stack[1].

We present experimental results to demonstrate that these mechanisms effectively provide admission control, selective connection discard, and service differentiation in an overloaded server. We also compare against application layer controls that we added in the Apache 1.3.12 server and show that the kernel controls are much more efficient and scalable.

The remainder of this paper is organized as follows: In the next section we introduce the problem. In Section 3 we give a brief overview on input packet processing. Our architecture and the kernel mechanisms are presented in Section 4. In Section 5 we present and discuss experimental results. We compare the performance of kernel based mechanisms and application level controls in Section 6.

[1] A port to Linux is underway.

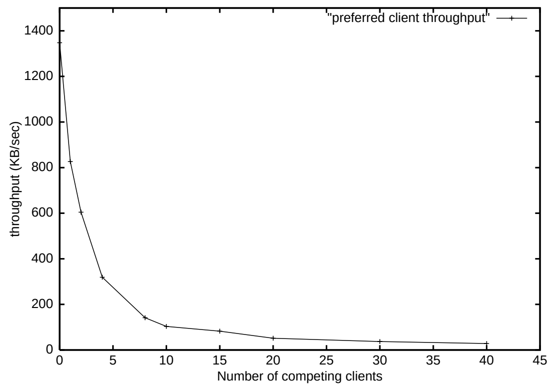

Fig. 1. Effect of Overload: Throughput of the preferred Macy's clients as a function of the number of competing clients.

We describe related work in Section 7 and finally, the conclusions and future work in Section 8.

II. Problem Overview

A traditional Internet (e.g., Web) server has no controls for service differentiation and overload protection. A sudden flash crowd or a set of greedy clients can easily overwhelm the server and drive it into overload. In an overload condition the throughput decreases and the delay experienced by all clients, including the well-behaved ones, increases. To illustrate this we ran a simple experiment in which one preferred client accesses premium content while other clients are making competing requests. To emulate the preferred client, we replay a server access log from the Macy's shopping site; the flash crowd is simulated by connections running the Webstone benchmark. The details on the experiment testbed are described in Section 4-6. Figure 1 shows the throughput achieved by the preferred Macy's client as a function of the number of competing client connections. As the load generated by the competing clients increases, the preferred client's throughput falls exponentially.

In general, once driven into overload, the server could spend most of its processing cycles thrashing, thereby, interrupting service without having any built-in capability to restore service in a time-bound fashion. It is critical to equip servers with capabilities for service differentiation and overload protection without necessarily requiring application modifications and/or involvement.

III. Input Packet Processing: Background

In this section we briefly describe the protocol processing steps executed when a new connection request is processed by a web server. The packet is first handled by the device interface and the device driver. When the device interface receives a packet it triggers a hardware interrupt that is serviced by the corresponding device driver [11].

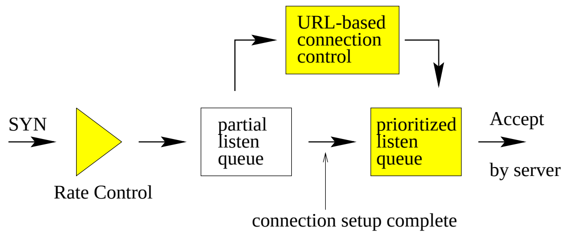

Fig. 2. Proposed kernel mechanisms.

The device driver copies the received packet into an `mbuf`, performs error checking, and demultiplexes it to determine the queue to insert the packet. For example, an IP packet is added to the input queue, `ipintrq`. The device driver then triggers the IP software interrupt which is serviced by its interrupt service routine. This starts the IP part of protocol processing of the packet. The IP input routine verifies the incoming packet headers, dequeues the packet from the IP input queue, determines if it has reached the destination, does packet reassembly if needed and does the next layer demultiplexing to invoke the transport layer input routine. For example, for a TCP packet this will result in a call to a `tcp_input` routine for further processing. The call to the transport layer input routine happens within the realm of the IP input call, i.e., there is no queuing between the IP and TCP layer. The TCP input processing verifies the packet and locates the protocol control block (PCB). If the incoming packet is a SYN request for a listen socket, a new data socket is created and placed in the partial listen queue and an ACK is sent back to the client. When the ACK for the SYN-ACK is received the TCP 3-way handshake is complete, the connection moves to an established state and the data socket is moved to the listen queue. The sleeping process, e.g., the web server, waiting on the `accept` call is woken up. The connection is ready to receive data. When a data packet is received it is placed in the socket receive buffer. The data is copied into the application address space and processed.

IV. Architecture Design

The network subsystem architecture adds three control mechanisms that are placed at the different stages of a TCP connection's life time. Figure 2 shows the various phases in the connection setup and the corresponding control mechanisms: (i) when a SYN packet is processed it triggers the SYN rate control and selective drop (ii) when the 3-way handshake is completed the prioritized listen queue selectively changes the ordering of accepted connections in the listen queue (iii) when the HTTP header is received the HTTP header controls decide on dropping or re- prioritizing the requests based on application layer information.

A. SYN Policer

TCP SYN policing controls the rate and burst at which new connections are accepted. Arriving TCP SYN packets are policed using a token bucket profile defined by the pair $< rate, \ burst >$, where $rate$ is the average number of new requests admitted per second and $burst$ is the maximum number of new requests accepted concurrently. Incoming connections are aggregated using specified filter rules that are based on the connection end points (source and destination addresses and ports as shown in Table II). The corresponding token bucket parameters are set per aggregate. The specification of the filter rules is discussed in a later section. On arrival at the server, the SYN packet is classified using the IP/TCP header information to determine the matching rule. A compliance check is performed against the token bucket profile of the rule. If compliant, a new data socket is created and inserted in the partial listen queue otherwise the SYN packet is silently discarded. TCP SYN policing basically limits the average connection acceptance rate and the maximum burst of concurrent connections admitted.

When the SYN packet is silently dropped, the requesting client will time-out waiting for a SYN ACK and retry again with an exponentially increasing time-out value[2]. An alternate option, which we do not consider, is to send a TCP RST to reset the connection indicating an abort from the server. Some clients, however, send another SYN immediately after a TCP RST is received instead of aborting the connection. Note that we drop non-compliant SYNs *before* a connection is established, in fact, even before a socket is created for the new connection. Thus, our approach invests only a small amount of overhead into requests that are dropped.

To provide service differentiation, connection requests are aggregated based on filters and each aggregate has a separate token bucket profile. To avoid dropping TCP SYNs blindly using thresholds, we check if SYN packets match a rule and comply with its rate and burst. This enables service differentiation as well as overload protection. Filtering based on client IP addresses is useful since a few domains account for a significant portion of a web server's requests [12]. The rate and burst values can be dynamically controlled by an adaptation agent, the details of which are beyond the scope of this paper.

B. Prioritized Listen Queue

The prioritized listen queue reorders the listen queue of a server process based on pre-defined connection priorities such that the highest priority connection is located at the head of the queue. This allows higher priority connections to get faster service from the server process. The priorities are associated with filters (see Table II) and connections are classified into different *priority classes*. When a TCP connection is established (i.e., on completion of the three-way handshake [11]), it is 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.

[2]The timeout values are typically set to 6, 24, 48, upto 75 seconds.

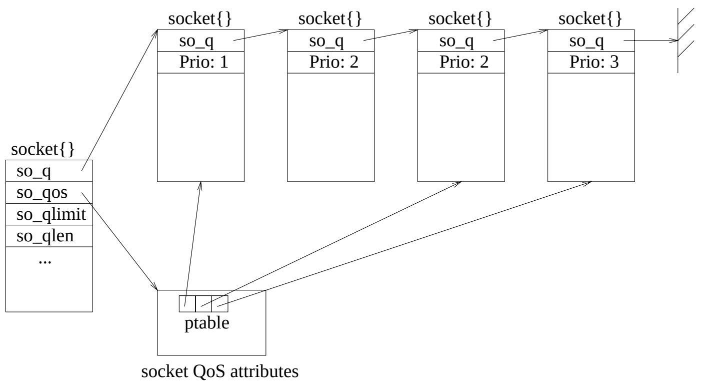

Fig. 3. Implementation of the prioritized listen queue

Figure 3 shows the implementation of a prioritized listen queue. A special data structure used for maintaining socket QoS attributes stores an array of *priority pointers*. Each priority pointer points to the *last* socket of the corresponding priority class. This allows efficient socket insertion — a new socket is always inserted behind the one pointed to by the corresponding priority pointer.

C. HTTP Header-based Controls

The SYN policer and prioritized listen queue have limited knowledge about the type and nature of a connection request, since they are based on the information available in the TCP and IP headers. For web servers with the majority of the traffic being HTTP over TCP, a more informed control is possible by examining the HTTP headers. For example, a majority of the load is caused by a few CGI requests and most of the bytes transferred belong to a small set of large files. It has been shown in earlier studies that 90% of the web requests are for 10% of the pages at a site [12]. This suggests that targeting specific URLs, types of URLs, or cookie information for service differentiation can have a wide impact during overload.

Our third mechanism, *HTTP header-based connection control*, enables content-based connection control by examining application layer information in the HTTP header,

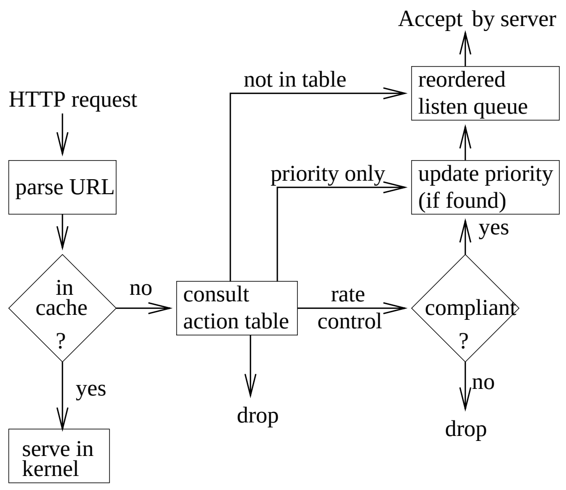

Fig. 4. The HTTP header-based connection control mechanism.

TABLE I
URL ACTION TABLE

URL	ACTION
noaccess	<drop >
/shop.html	<priority=1 >
/index.html	< rate=15 conn./sec, burst=5 conn.>, <priority=1 >
/cgi-bin/*	< rate=10 , burst=2 >

such as the URL name or type (e.g., CGI requests) and other application-specific information available in cookies. The control is applied in the form of rate policing and priorities based on URL names and types and cookie attributes. Thus, it is possible to reorder the listen queue based on the application layer information included in the HTTP header along with the connection information.

This mechanism involves parsing the HTTP header in the kernel and waking the sleeping web server process only after a decision to service the connection is made. If a connection is discarded, a TCP RST is sent to the client and the socket receive buffer contents are flushed.

For URL parsing, our implementation relies upon Advanced Fast Path Architecture(AFPA) [13], an in-kernel web cache on AIX. For Linux, an in-kernel web engine called KHTTPD is available [14]. As opposed to the normal operation, where the sleeping process is woken up after a connection is established and the socket is moved from the partial to the listen queue, AFPA responds to cached HTTP requests directly without waking up the server process. With AFPA, a connection is *not* moved out of the partial listen queue even after the 3-way handshake is over. The normal data flow of TCP continues with the data being stored in the socket receive buffer. When the HTTP header is received (that is when the AFPA parser finds two CR control characters in the data stream), AFPA checks for the object in its cache. On a cache miss, the socket is moved to the listen queue and the web server process is woken up to service the request.

TABLE II
EXAMPLE NETWORK-LEVEL POLICIES

(dst IP,dst port,src IP,src port)	(r,b)	priority
(*, 80, *, *)	(300,5)	3
(*, 80, 10.1.1.1, *)	(100,5)	2
(12.1.1.1, 80, *, *)	(10,1)	*

The HTTP header-based connection control mechanism comes into play at this juncture, as illustrated in Figure 4, before the socket is moved out of the partial listen queue. The URL action table (Table I) specifies three types of actions/controls for each URL or set of URLs. A drop action implies that a TCP RST is sent before discarding the connection from the partial listen queue and flushing the socket receive buffer. If a priority value is set it determines the location of the corresponding socket in the ordered listen queue. Finally, rate control specifies a token bucket profile of a <rate, burst> pair which drops out-of-profile connections similar to the SYN policer.

D. Filter Specification

A filter rule specifies the network-level and/or application-level attributes that define an aggregate and the parameters for the control mechanism that is associated with it. A network-level filter is a four-tuple consisting of local IP address, local port, remote IP address, and remote port; application-level filters were shown in Table I. Table II lists some network-level filter examples. The first rule applies to the web server process listening at local port 80 on all network interfaces; it specifies that all connections to the server are rate-controlled at a rate of 300 conns/sec, a burst of 5, and a priority of 3 (the default lowest priority). The second rule assigns all connections from the client IP address 10.1.1.1 a priority level of 2, and a lower incoming rate of 100 conns/sec. Finally, all clients connecting to the host at 12.1.1.1 are policed at the rate of 10 conns/sec. The filter rules can contain range of IP addresses, wildcards, lists of addresses etc.

E. Protocol Stack Architecture

We have developed architectural enhancements for Unix-based servers to provide these mechanisms. Figure 5 shows the basic components of the enhanced protocol stack architecture, with the new capabilities utilized either by user-space agents or applications themselves. This architecture permits control over an application's inbound network traffic via policy-based traffic management [10]; an adaptation/policy agent installs policies into the kernel

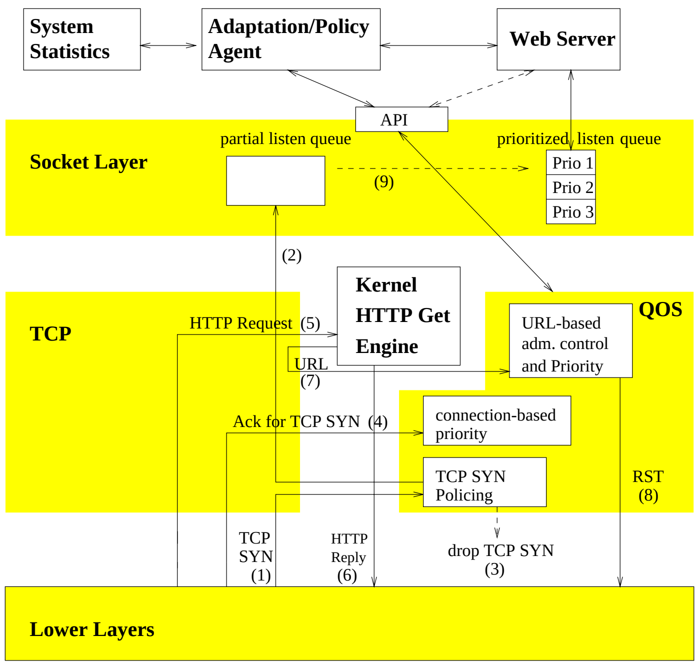

Fig. 5. Enhanced protocol stack architecture.

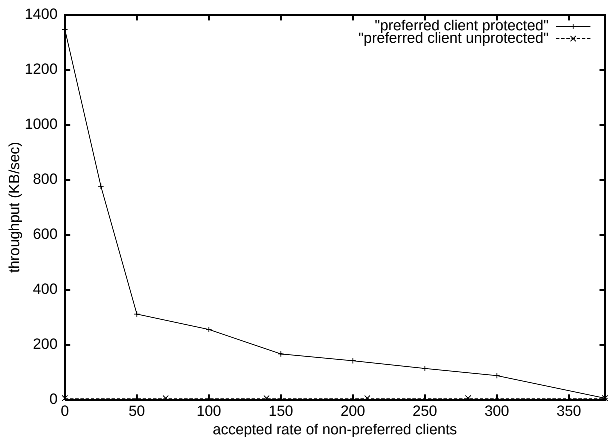

Fig. 6. Throughput of the preferred Macy's client with and without TCP SYN policing. On the X-axis is the SYN policing rate of the non-preferred Webstone clients that are continuously generating requests. The Y-axis shows the corresponding throughput received by the Macy's client when there was no SYN control and when SYN control was enforced.

via a special API. These policies specify filters to select the traffic to be controlled, and actions to perform on the selected traffic. The actions include limits on the acceptable rate and/or priority assignment for incoming connections. The figure shows the flow of an incoming request through the various control mechanisms.

F. *Implementation Methodology and Testbed*

We have implemented the proposed kernel mechanisms in AIX 5.0, and evaluated them on the testbed described below. As shown in Figure 5, the QoS module contains the TCP SYN policer, a priority assignment function for new connections, and the entity that performs URL-based admission control and priority assignment. On arrival of certain segments, the TCP layer invokes the corresponding functions in the QoS module. The in-kernel HTTP get engine is called to determine the URL of incoming HTTP requests. The Web server receives HTTP requests from the socket layer via the prioritized listen queue.

All experiments were conducted on a testbed comprising an IBM HTTP Server running on a 375 MHz RS/6000 machine, several 550 MHz Pentium III clients running Linux, and one 166 MHz Pentium Pro client running FreeBSD. The server and clients are connected via a 100 BaseT Ethernet switch. For client load generators we use Webstone 2.5 [15] and a slightly modified version of sclient [16]. Both programs measure client throughput in connections per second. The experimental workload consists of static and dynamic requests. The dynamic files are minor modifications of standard Webstone CGI files that simulate memory consumption of real-world CGIs.

The IBM HTTP Server is a modified Apache [17] 1.3.12 web server that utilizes an in-kernel HTTP get engine called the Advanced Fast Path Architecture (AFPA). We use AFPA in our architecture only to perform the URL parsing and have disabled any caching when measuring throughput results. Unless stated otherwise, we configured Apache to use a maximum of 150 server processes.

V. Experimental Results

A. *Efficacy of SYN Policing*

In this section we show how TCP SYN policing protects a preferred client against flash crowds or high request rates from other clients. In our setup, one client replays the Macy's trace file representing a preferred customer. For the competing load we use five machines running Webstone, each with 50 clients. All clients request an 8 KB file, which is reasonable since a typical HTTP transfer is between 5 and 13 KB [12]. We need a large number of Webstone clients because when a non-compliant TCP SYN packet is dropped, TCP's exponential backoff algorithm causes the Webstone client to refrain from making new requests until the backoff timer expires (which could be as high as 75 secs.). Note that this problem is due to the closed-loop nature of Webstone, and does not occur in more natural client populations.

Without SYN policing, the Macy's client receives a low throughput of about 6 KB/sec. Using policing to lower the acceptance rate of Webstone clients, we expect the throughput for the Macy's client to increase. Figure 6 shows that the throughput for Macy's client increases from 100 KB/sec to 800 KB/sec as the acceptance rate for Webstone clients is lowered from 300 reqs/sec to 25 reqs/sec. The experiment demonstrates that a preferred client can be successfully protected by rate-controlling connection requests of other greedy clients.

TCP SYN policing works well when client identities and request patterns are known. In general, however, it is difficult to correctly identify a misbehaving group of clients. Moreover, as discussed below, it is hard to predict the rate control parameters that enable service differentiation for preferred clients without under-utilizing the server. TCP SYN policing can protect servers against overload (e.g., due to dynamic requests) and flash crowds by limiting the total number of accepted requests regardless of client identity or requests. For effective overload prevention the policing rate must be dynamically adapted to the resource consumption of accepted requests. Furthermore, since the non-preferred clients are rate limited, they cannot overrun the server's partial listen queue which could have resulted in a preferred client being dropped.

B. *Impact of Burst Size*

In the pervious experiment we did not analyze the effect of the burst size on the effective throughput. The burst size is the maximum number of new connections accepted concurrently for a given aggregate. With a large burst size, greedy clients can overload the server, whereas with a small burst, clients may be rejected unnecessarily and must connect at regular intervals to utilize the rate optimally. Furthermore, if 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. The burst size also controls the responsiveness of rate control. With a large burst size, a change in the rate control values will not be reflected until the burst of accepted requests are serviced. A small burst size, on the other hand, makes the system more responsive. There is a tradeoff, however, between responsiveness and the achieved throughput.

We next show the effect of the burst size on the throughput of a preferred client. In our experiment, the non-preferred client is a modified sclient program that makes 50 to 80 back-to-back connection requests about twice a second, in addition to the specified request rate. Both the length of the incoming request burst and its timing are randomized. Figure 7 shows the throughput of preferred and non-preferred client with the SYN policing rate of the non-preferred client set to 50 conn/sec and the burst size varying from 5 to 50. The non-preferred sclient program

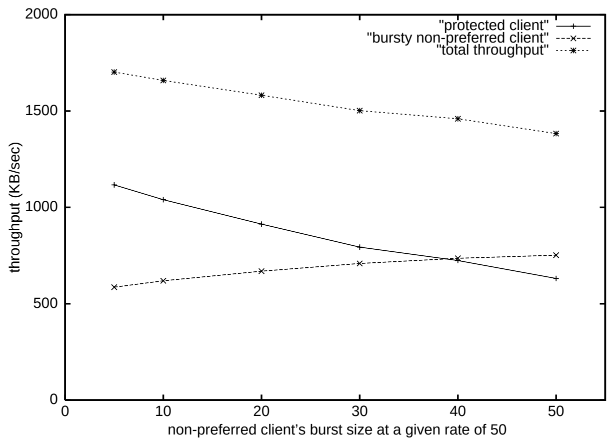

Fig. 7. Impact of burst size on preferred client throughput. The burst size for policing non-preferred client is varied from 5 to 50 while the connection acceptance rate is fixed at 50 conn/sec. The plot shows the throughput achieved by the preferred and non-preferred clients along with the total throughput.

requests a 16 KB dynamically generated cgi file. The preferred client is a Webstone program with 40 clients, requesting a static 8 KB file. As the burst size is increased from 5 to 50, the sclient's throughput increases from 36.6 conns per second (585.6 KB/sec) to 47.7 conns per second (752 KB/sec), while the throughput received by the preferred client decreases from about 140 conns per second (1117 KB/sec) to 79 conns per second.

Intuitively the overall throughput should have increased, however, the observed decrease in total throughput is due to the fact that we accept more CPU consuming CGI requests from sclient, thereby, incurring a higher overhead per byte transferred.

C. *Prioritized Listen Queue: Simple Priority*

With TCP SYN policing, one must limit the greedy non-preferred clients to a meaningful rate during overload. In most cases it is relatively simpler to just give the preferred clients a higher absolute priority. By reordering a server's listen queue based on connection priorities the highest priority connection remains at the head of the queue and gets serviced first. We demonstrate next that the prioritized listen queue provides service differentiation, especially with a large listen queue length.

In our experiments we classify clients into three priority levels. Clients belonging to a common priority level are all created by a Webstone benchmark that requests an 8 KB file. A separate Webstone instance is used for each priority level. We measure client throughput for each priority level while varying the total number of clients in each class. Each priority class uses the same number of clients.

In the first experiment, the Apache server is configured to spawn a maximum of 50 server processes. The results in Figure 8 show that when the total number of clients is small, all priority levels achieve similar throughput. With fewer clients, server processes are always free to handle incoming requests. Thus, the listen queue remains short and almost no reordering occurs. As the number of clients increases, the listen queue builds up since there are fewer Apache processes than concurrent client requests. Consequently, with re-ordering the throughput received by the high priority client increases, while that of the two lower priority clients decreases. Figure 8 shows that with more than 30 Webstone clients per class only the high-priority clients are served while the lower-priority clients receive almost no service.

Figure 9 illustrates the effect on response times observed by clients of the three priority classes. It can seen that as the number of clients increases across all priority classes the response time for the lower priority classes increases exponentially. The response time of the high priority class, on the other hand, only increases sub-linearly. When the number of high priority requests increases, the lower priority ones are shifted back in the listen queue, thereby, increasing their response times. Also as more high priority requests get serviced by the different server processes running in parallel and competing for the CPU their response times increase.

We also observed that when the number of high priority requests was fixed and the lower priority request rate was steadily increased, the response time of the high priority requests remained unaffected.

To understand the impact of the number of server processes on the throughput of the low priority class, we decreased the number of Apache server processes to 20. When there are fewer processes to service connections, the length of the listen queue increases. As expected Figure 10 shows that the low-priority clients are starved much earlier (at 10 clients per class) with 20 server processes than they did with 50 processes.

The priority-based approach enables us to give low delay and high throughput to preferred clients independent of the requests or request patterns of other clients. However, one may need many priority classes for different levels of service. The main drawback of a simple priority ordering is that it provides no protection against starvation of low-priority requests.

D. Combining TCP SYN Policing and Priority

To prevent starvation, low priority requests need to have some minimum number reserved slots in the listen queue so that they are not always preempted by a high priority request. However, reserving slots in the listen queue arbitrarily could cause a high priority request to find the listen queue to be full, which would in turn cause it to be aborted after its 3-way handshake is completed. If reservations are desired they have to be done in the partial listen queue before the SYN is acknowledged. To avoid starvation with fixed priorities, we combine the listen queue priorities with SYN policing to give preferred clients higher priority, but limiting their maximum rate and burst, thereby, implicitly reserving some slots in the queue for the lower priority requests.

Table III shows the results for experiments with three sets of Webstone clients with different priorities and rate control of the high priority class. The lower priority class has 30 Webstone clients while the high priority class has

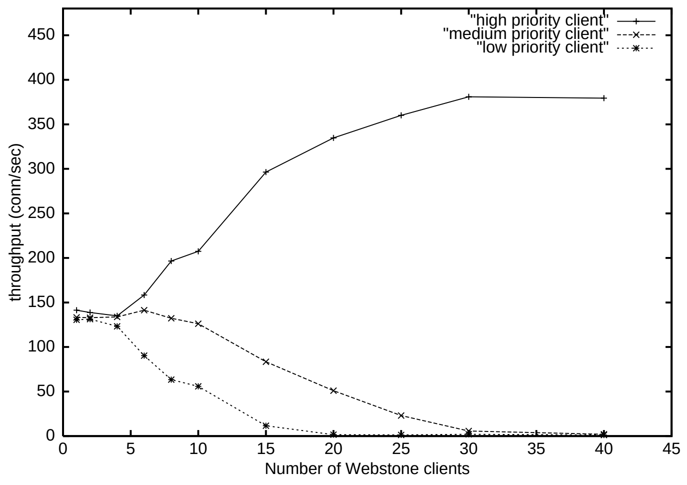

Fig. 8. Throughput with the prioritized listen queue and 3 priority classes with 50 Apache processes. The number of clients in each class remains equal.

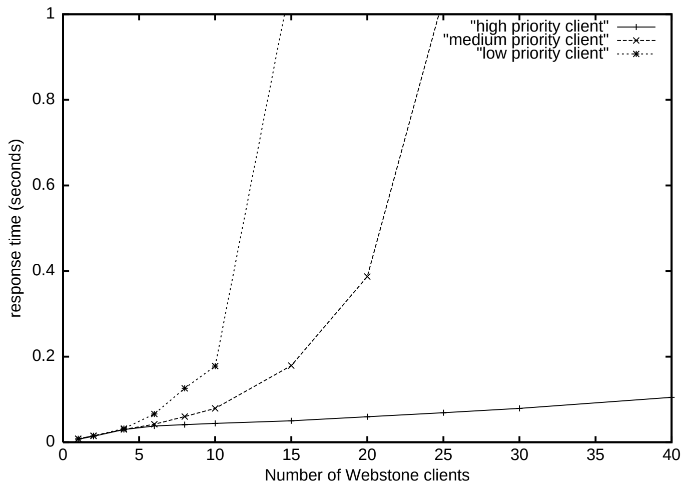

Fig. 9. Response time with the prioritized listen queue and 3 priority classes with 50 Apache processes. The number of clients in each class remains equal.

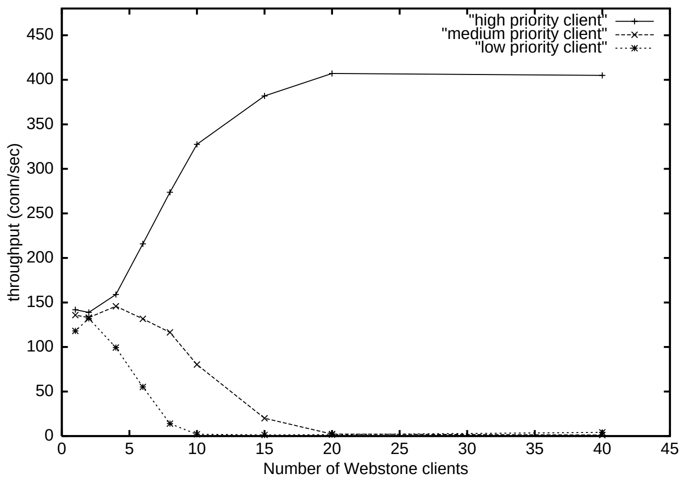

Fig. 10. Throughput with the prioritized listen queue and 3 priority classes with 20 Apache processes. The number of clients in each class remains equal.

TABLE III

TCP SYN POLICING OF A HIGH-PRIORITY CLIENT TO AVOID STARVATION OF OTHER CLIENTS.

Throughput (conn/sec) of each priority class			
client priority	(rate, burst) limit of high priority		
	none	(300,300)	(200,200)
high	381	306	196
medium	0	78.6	180
low	0	4.1	13

150 Webstone clients spread over three different hosts. With no SYN policing of the clients in the high priority class, the two low-priority clients are completely starved. Table III shows that rate limiting the clients in the high priority class to 300 conn/sec prevents starvation; the medium and low priority clients achieve a throughput of 78.6 and 4.1 conn/sec respectively.

E. HTTP Header-based Connection Control

The SYN policer and prioritized listen queue described earlier enable service differentiation to be based entirely on information available in the TCP and IP headers of an incoming connection (i.e, the source and destination address and port number). For web servers, with the majority of the traffic being HTTP over TCP, a more informed control is possible by examining the HTTP headers to obtain application specific information.

In this section we illustrate the performance and effectiveness of admission control and service differentiation based on the URL name and type.

Rate control using URLs: In our experimental scenario the preferred client replaying the Macy's trace needs to be protected from overload due to a large number of high overhead CGI requests from non-preferred clients. The client issuing CGI requests is an sclient program requesting a dynamic file of length 5 KB at a very high rate. Figure 11 shows that without any protection, the preferred Macy's customer receives a low throughput of under 1 KB/sec. By rate-limiting the dynamic requests from 40 reqs/sec to 2 reqs/sec the throughput of the preferred Macy's customer improves from 1 KB/sec to 650 KB/sec. In contrast to TCP SYN policing (Figure 6), URL rate control targets a specific URL causing overload instead of a client pool. Knowledge of the load caused by a particular request (or type of request) can be used to target specific requests for rate control in order to achieve the desired throughput for the preferred client.

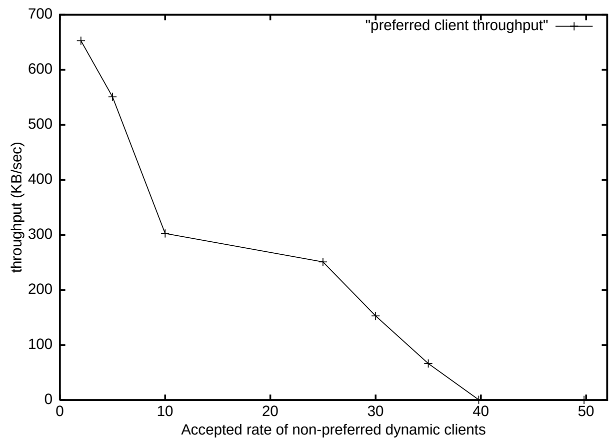

Fig. 11. URL-based policing to protect preferred Macy's customers. The graph shows the resulting throughput of the preferred Macy's client as a specific high overhead CGI requests is limited to a given number of conn/sec

URL priorities: In this section we present the results of priority assignments in the listen queue based on the URL name or type being requested. The clients are Webstone benchmarks requesting two different URLs, both corresponding to files of size 8 KB. There are two priority classes in the listen queue based on the two requested URLs. Figure 12 shows that the lower priority clients (accessing the low priority URL) receive lower throughput and are almost starved when the number of clients requesting the high priority URL exceeds 40. These results correspond to the results shown earlier with priorities based on the connection attributes (see Figure 8). The average total throughput, however, is slightly lower with URL-based priorities due to the additional overhead of URL parsing.

Combined URL-based rate control and priorities: To avoid starvation of requests for the low-priority URL, we rate limit the requests for the high-priority URL. In this

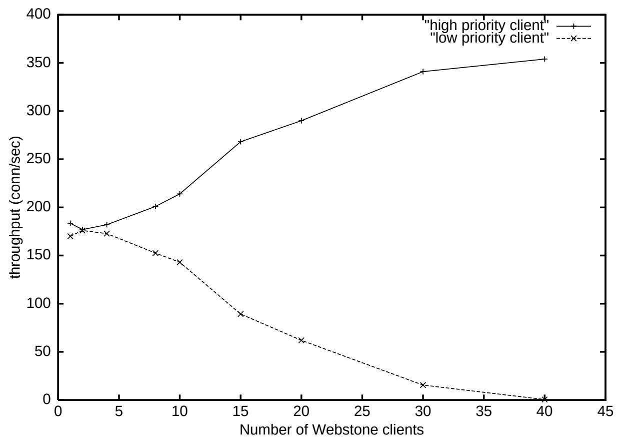

Fig. 12. Throughput with 2 URL-based priorities and 50 Apache server processes. The number of clients in each class is equal

TABLE IV

URL-BASED POLICING OF A HIGH-PRIORITY CLIENT TO AVOID STARVATION OF OTHER CLIENTS.

Throughput (conn/sec)			
client priority	(rate, burst) limit of high priority		
	none	(30,10)	(10,10)
high	61.7	29.0	10.1
low	0	10.2	117

experiment, we assign a higher priority to requests for a dynamic CGI request of size 5 KB (requested by an sclient program), and lower priority to requests for a static 8 KB file (requested by the Webstone program). Table IV shows that without rate-control of high-priority requests, the lower-priority requests are completely starved; this can be avoided by rate-limiting the high-priority URL requests.

E.1 Overload Protection from High Overhead Requests

So far we have used the URL-based controls for providing service differentiation based on URL names and types. In the next experiment, we investigate if URL-based connection control can be used to protect a web server from overload by a targeted control of high overhead requests (e.g., CGI requests that require large computation or database access).

We use the sclient load generator to request a given high overhead URL and control the request rate, steadily increasing it and measuring the throughput. Figure 13 shows the client's throughput with varying request rates for a dynamic CGI request that generates a file size of 29 KB. The throughput increases linearly with the request rate upto a critical point of about 63 connections/sec. For any further increase in the request rate the throughput falls exponen-

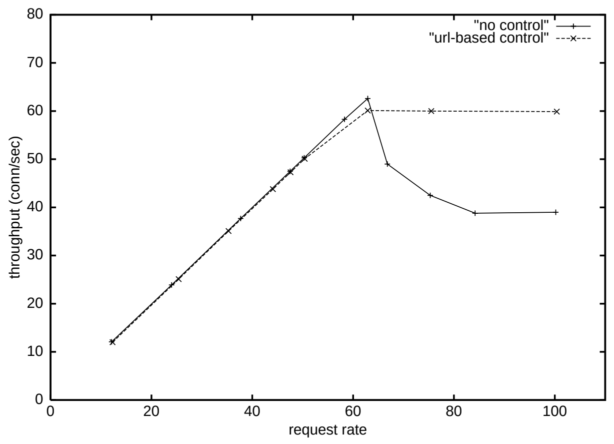

Fig. 13. Overload protection from high overhead requests using URL-based connection control. The graph shows the throughput of web server with no controls servicing CPU intensive CGI requests and the corresponding throughput when the CGI requests are limited to 60 reqs/sec.

tially and later plateaus to around 40 connections/sec. To understand this behavior we used `vmstat` to capture the paging statistics. Since the dynamic requests are memory-intensive, the available free memory rapidly declines. For some combinations of the request rate and the number of active processes, the available free memory falls to zero. Eventually the system starts thrashing as the CPU spends most of the time waiting for pending local disk I/O. In the above experiment with 150 server processes and a request rate of 63 reqs/sec the wait time starts increasing as indicated by the `wait` field of the output from vmstat.

To prevent overload we use URL-based connection control to limit the number of accepted dynamic CGI requests to a rate of 60 reqs/sec and a burst of 10. The dashed line in Figure 13 shows that with URL-based control the throughput stabilizes to 60 reqs/sec and the server never thrashes. In the above experiment, the URL-based connection control can handle request rates of up to 150 requests per second. However, for request rates beyond that thrashing starts as the kernel overhead of setting up connections, parsing the URL and sending the RSTs, becomes substantial.

To further delay the onset of thrashing we augment the URL-based control with the TCP SYN policer. For every TCP RST that is sent we drop any subsequent SYN request from that same client for a specified time interval. This reduces the excess overhead of sending back-to-back RSTs and parsing URLs for the same client's repeat requests, consequently, increasing the sustainable request rate.

TABLE V
PERFORMANCE OF AFPA AND MATCHING A URL TO A RULE FOR A 8 KB FILE WITH DIFFERENT URL LENGTHS.

Throughput (conn/sec)			
URL off length	AFPA (no cache)	AFPA on, (no cache) no rule	AFPA on, matching rule
11 char.	370.1	340.5	338.3
80 char.	361.5	321.9	319.4
160 char.	355.1	321.1	303.7

TABLE VI
OVERHEAD OF KERNEL MECHANISMS

Operation		Cost(μsec)
TCP SYN policing	1 filter rule	7.9
	3 filter rules	9.6
classification and priority	1 rule	4.4
	3 rules	5.0
AFPA including URL parsing		19
URL-based rate control including URL matching	1 rule	5.0
	2 rules	5.8
	3 rules	6.5
URL-based priority including URL matching	1 rule	3.8
	2 rules	4.1
	3 rules	4.3

E.2 Discussion

The HTTP header-based rate control relies on sending TCP RST to terminate non-preferred connections as and when necessary. However, different client TCP implementations behave differently on receiving a RST. Some clients respond to a RST by immediately trying again for some fixed number of times. This adds additional load on the server. It would be preferable to have clients backoff instead. In our implementation we could directly return an HTTP error message (e.g., server busy) back to the client and close the connection instead of sending a TCP RST.

Our implementation of URL-based control handles only HTTP/1.0 connections. We have designed a solution for HTTP/1.1 with keep-alive connections that service multiple requests per open connection. The rate control is applied only at the start of a connection; subsequent requests are not controlled. We are currently exploring different mechanisms to limit the number and types of requests that can be serviced on an accepted persistent connection.

The experiments in the previous section have only presented results on URL based controls. Similar controls can be set based on the information in cookies that can capture transaction information and client identities. We have not yet evaluated the gain from cookie-based controls and the potential overhead of parsing large number of cookies.

F. Overhead of the Kernel Mechanisms

We quantify the overhead of matching URLs in the kernel for varying URL lengths.

Table V shows that the overhead of matching a URL to a rule is moderate (under 6 % for a 160 character URL). The throughput numbers are for 20 Webstone clients requesting an 8 KB file. Rules are matched using the standard string comparison (`strcmp`) with no optimizations; better matching techniques can reduce this overhead significantly. On a cache miss, the in-kernel AFPA cache introduces an overhead of about 10% for an 8 KB file. However, the AFPA cache under normal conditions increases performance significantly for cache hits. In our experiments we have the cache size set to 0 so that AFPA cannot server any object from the cache. When caching is enabled Webstone received a throughput of over 800 connections per second on a cache hit.

Table VI summarizes the additional overhead of the implemented kernel mechanisms. The overhead of compliance check and filter matching for TCP SYN policing with 1 filter rule is 7.9 μsecs. Simply matching the filter, allocating space to store QoS state, and setting the priority adds an overhead of around 4.4 μsecs for 1 filter rule. The policing controls are more expensive as they include accessing the clock for the current time. Surprisingly the URL matching and rate control has a low overhead of 5.0 μsecs for a URL of 11 chars. This happens to be lower than SYN policing as the `strcmp` matching is cheaper for one short URL compared to matching multiple IP addresses and port numbers. The overhead of URL matching and setting priorities for a single rule is around 3.8 μsecs. The most expensive operation is the call to AFPA to parse the URL. AFPA not only parses the URL, but also does logging, checks if the requested object is in the network buffer cache, and pre-computes the HTTP response header.

VI. COMPARISON OF USER SPACE AND KERNEL MECHANISMS

In this section we compare the effectiveness of our kernel mechanisms with overload protection and service differentiation mechanisms implemented in user space. One might argue that kernel-based mechanisms are less flexible and more difficult to implement than mechanisms implemented in user space. However, kernel mechanisms are robust and provide much better performance. User level controls although limited in their capabilities, have easy access to application layer information. In general, placing mechanisms in the kernel is beneficial if it leads to considerable performance gains and increases the robustness of the server.

To enable a fair comparison we have extended the Apache 1.3.12 server with additional modules [18] that police requests based on the client IP address and requested URL. The implemented rate control schemes use exactly the same algorithms as our kernel based mechanisms. If a request is not compliant we send a "server temporarily unavailable" (503 response code) back to the client and close the connection. Note that this involves more overhead than aborting the TCP connection by sending a TCP RST as we do with URL based connection control in the kernel. It is, however, not possible to abort a connection from user space. The connection could have been closed abruptly from user space, however, handling the close has the similar overheads as sending 503 response code.

The experimental setup consists of a Webstone traffic generator with 100 clients requesting a file of size 8 KBytes along with an sclient program requesting a file of size 16 KBytes. The sclient's requests are rate controlled with a rate of 10 requests per second and a burst of 2; there are no controls set for the Webstone clients. During our experiments, we steadily increased the sclient's request rate. When the sclient's request rate increases, the server load approaches saturation and the rate control mechanisms need to discard more requests. The overhead of the control mechanisms adds to the server load, thereby, reducing the throughput achieved by the Webstone client. The more efficient the controls are, the higher the throughput the Webstone program can sustain when the request rate from the sclient program increases.

When Webstone runs standalone, it receives a throughput of about 395 conn/sec with the AFPA kernel module loaded (with no caching); the throughput is about 372 conn/sec when AFPA is not loaded. The corresponding average response times are 0.265 seconds and 0.248 seconds respectively.

Figure 14 illustrates that when the request load of the sclient program is low (20 reqs/sec), the Webstone throughput is 392 conn/sec and 387.3 conn/sec for TCP SYN policing and Apache user level controls respectively. These controls limit the sclient acceptance rate to 10.0 conn/sec. With in-kernel URL-based rate control the throughput is lower (354 conn/sec). This low throughput is caused by the additional 10% overhead added by AFPA (with no caching) as discussed in Section 5-6. As discussed earlier, with the cache size set to zero, we add more overhead than necessary for URL parsing, without the corresponding gains from AFPA caching.

As the sclient's request load increases further, TCP SYN policing is able to achieve a sustained throughput for the Webstone clients, while the Apache based controls shows a marked decline in throughput. The graph shows that

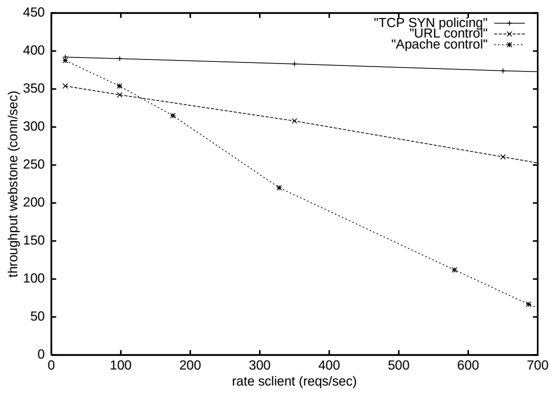

Fig. 14. Throughput of kernel-based TCP SYN policing, kernel based URL rate control and Apache module based connection rate control. The throughput achieved by Webstone clients is measured against an increasing request load generated by sclient. The sclient requests are rate controlled to 10.0 req/sec with a burst of 2

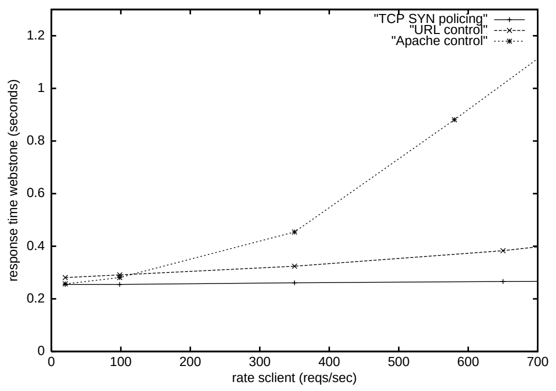

Fig. 15. Response times using kernel-based TCP SYN policing, kernel based URL rate control and Apache module based connection rate control. The response time achieved by Webstone clients is measured against an increasing request load generated by sclient. The sclient requests are rate controlled to 10 req/sec with a burst of 2

for a sclient load of 650 reqs/sec the Webstone throughput for TCP SYN policing is 374 conn/sec; for in-kernel URL-based connection control it is 260.7 conn/sec; for Apache user level controls the throughput sinks to about 75 conn/sec.

The experiment demonstrates that the kernel mechanisms are more efficient and scalable than the user space mechanisms. There are two main reasons for the higher efficiency and scalability: First, non-compliant connection requests are discarded earlier, in particular less CPU is consumed by the request and the context switch to user space is avoided. Second, when implementing rate control at user space, the synchronization mechanisms for sharing state among all the Apache server processes decrease performance.

The corresponding results for response times are shown in Figure 15. Using the Apache controls, non-compliant requests from sclient are enqueued in the listen queue and delay the processing of all other requests, thereby, increasing Webstone's response time.

VII. Related Work

Several research efforts have focused on admission control and service differentiation in web servers [19], [20], [21], [22], [8] and [23]. Almeida *et al.* [8] use priority-based schemes to provide differentiated levels of service to clients depending on the web pages accessed. While in their approach the application, i.e. web server, determines request priorities, our mechanisms reside in the kernel and can be applied without context-switching to user level. *WebQoS* [23] is a middleware layer that provides service differentiation and admission control. Since it is deployed in user space, it is less efficient compared to kernel-based mechanisms. While WebQoS also provides URL-based classification, the authors do not present any experiments or performance considerations. Cherkasova *et al.* [20] present an enhanced web server that provides session-based admission control to ensure that longer sessions are completed. Crovella *et al.* [24] show that client response time improves when web servers serving static files serve shorter connections before handling longer connections. Our mechanisms are general and can easily realize such a policy.

Reumann *et al.* [25] have presented virtual services, a new operating system abstraction that provides resource partitioning and management. Virtual services can enhance our scheme by, for example, dynamically controlling the number of processes a web server is allowed to fork. The receiver livelock study [2] showed that network interrupt handling could cause server livelocks and should be taken into consideration when designing process scheduling mechanisms. Banga and Druschel's [26] *resource containers* enable the operating system to account for and control the consumption of resources. To shield preferred clients from malicious or greedy clients one can assign them to different containers. When using resource containers to prevent overload caused by CGI scripts, one creates a container for CGIs and restricts its CPU and memory usage. Kanodia *et al.* [21] present a simulation study of queueing-based algorithms for admission control and service differentiation at the front-end. They focus on guaranteeing latency bounds to classes by controlling the admission rate per class. Aron *et al.* [27] describe a scalable request distribution architecture for clusters and also present resource management techniques for clusters.

Scout [28], Rialto [29] and Nemesis[30] are operating systems that track per-application resource consumption and restrict the resources granted to each application. These operating systems can thus provide isolation between applications as well as service differentiation between clients. However, there is a significant amount of work involved to port applications to these specialized operating systems.

VIII. Conclusions and Future Work

In this paper, we have presented three in-kernel mechanisms that provide service differentiation and admission control for overloaded web servers. TCP SYN policing limits the number of incoming connection requests using a token bucket policer. We demonstrated that this mechanism can protect preferred clients against denial-of-service caused by other requests by enforcing a maximum acceptance rate on the latter. The prioritized listen queue gives good service, i.e. low delay and high throughput, to clients with high priority, but can starve low priority clients. We show that starvation can be avoided by combining this mechanism with TCP SYN policing. Finally, URL-based connection control provides in-kernel admission control and priority based on application-level information such as URLs and cookies. This mechanism is very powerful and can, for example, prevent overload caused by dynamic requests. URL-based connection control involves additional overhead since it is effective only after numerous resources have been invested in a connection. We propose to augment URL-based control with TCP SYN policing, and are also exploring other performance optimizations.

We have compared the kernel mechanisms against application layer controls that we added in the Apache server. Our experiments have demonstrated that the kernel mechanisms are much more efficient and scalable than the Apache user level controls.

The kernel mechanisms that we presented rely on the existence of accurate policies that control the operating range of the server. Although they provide a rich set of controls for system tuning, the underlying assumption is that the system administrator has complete understanding of server behavior under varying load conditions. In a production system it is unrealistic to assume knowledge of the optimal operating region of the server. We are currently implementing a policy adaptation agent (Figure 5) that dynamically adapts the rate control policies to the changing workload conditions. This adaptation agent uses available kernel statistics and past history to select appropriate values for the various policies. We also define an API to export the kernel mechanisms to the user level. Along with the adaptation agent, this API enables trusted applications to directly influence kernel behavior.

Our current implementation does not address security issues of fake IP addresses and client identities. However, the adaptation agent flags sudden load changes and could possibly set up policies to block such requests. We plan to integrate a variety of overload prevention policies with traditional firewall rules to provide an integrated solution.

REFERENCES

[1] "Cisco TCP intercept," http://www.cisco.com.

[2] J. C. Mogul and K. K. Ramakrishan, "Eliminating receive livelock in an interrupt-driven kernel," in *Proc. of USENIX Annual Technical Conference*, Jan. 1996.

[3] P. Druschel and G. Banga, "Lazy receiver processing (LRP): a network subsystem architecture for server systems," in *Proc. of OSDI*, Oct. 1996, pp. 91–105.

[4] O. Spatscheck and L. Peterson, "Defending against denial of service attacks in scout," in *Proc. of OSDI*, Feb. 1999.

[5] "Ensim corporation: virtual servers," http://www.ensim.com.

[6] "Alteon web systems," http://www.alteonwebsystems.com.

[7] "Cisco arrowpoint web network services," http://www.arrowpoint.com.

[8] 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.

[9] 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, no. 3, pp. 397–413, Apr. 1998.

[10] A. Mehra, R. Tewari, and D. Kandlur, "Design considerations for rate control of aggregated tcp connections," in *Proc. of NOSSDAV*, June 1999.

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

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

[13] P.Joubert, R. King, R.Neves, M.Russinovich, and J.Tracey, "High performance memory based web caches: Kernel and user space performance," in preparation.

[14] "Khttpd - linux http accelerator," http://www.fenrus.demon.nl/.

[15] "webstone," http://www.mindcraft.com.

[16] G. Banga and P. Druschel, "Measuring the capacity of a web server," in *Proc. of USITS*, Dec. 1997.

[17] "apache," http://www.apache.org.

[18] L. Stein and D. MacEachern, *Writing Apache modules with Perl and C*, O'Reilly, 1999.

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

[20] L. Cherkasova and P. Phaal, "Session based admission control: a mechanism for improving the performance of an overloaded web server," Tech. Rep., Hewlett Packard, 1999.

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

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

[23] Nina Bhatti and Rich Friedrich, "Web server support for tiered services," *IEEE Network*, Sept. 1999.

[24] M. E. Crovella, R. Frangioso, and M. Harchol-Balter, "Connection scheduling in web servers," in *Proc. of USITS*, Oct. 1999.

[25] J. Reumann, A. Mehra, K. Shin, and D. Kandlur, "Virtual services: A new abstraction for server consolidation," in *Proc. of USENIX Annual Technical Conference*, June 2000.

[26] G. Banga, P. Druschel, and J. Mogul, "Resource containers: a new facility for resource management in server systems," in *Proc. of OSDI*, Feb. 1999.

[27] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel, "Scalable content-aware request distribution in cluster-based network servers," in *Proc. of USENIX Annual Technical Conference*, June 2000.

[28] D. Mosberger and L. L. Peterson, "Making paths explicit in the scout operating system," in *Proc. of OSDI*, Oct. 1996, pp. 153–167.

[29] 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*, Sept. 1996, pp. 249–256.

[30] Thiemo Voigt and Bengt 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.