Architectures for Service Differentiation in Overloaded Internet Servers

Thiemo Voigt

A Dissertation submitted
for the Degree of Doctor of Philosophy
Department of Information Technology
Uppsala University

May 2002

Dept. of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

DoCS 02/119
ISSN 0283-0574

Swedish Institute of Computer Science
Box 1263
SE-164 29 Kista
Sweden

SICS Dissertation Series 30
ISSN 1101-1335
ISRN SICS-D--30--SE

Dissertation for the Degree of Doctor of Philosophy in Computer Systems presented at Uppsala University in 2002.

ABSTRACT

Voigt, T. 2002: Architectures for Service Differentiation in Overloaded Internet Servers. *SICS Dissertation Series* 30. Also as *DoCS* 02/119. 153 pp. Uppsala. ISBN 91-506-1559-9.

Web servers become overloaded when one or several server resources such as network interface, CPU and disk become overutilized. Server overload leads to low server throughput and long response times experienced by the clients.

Traditional server design includes only marginal or no support for overload protection. This thesis presents the design, implementation and evaluation of architectures that provide overload protection and service differentiation in web servers. During server overload not all requests can be processed in a timely manner. Therefore, it is desirable to perform service differentiation, i.e., to service requests that are regarded as more important than others. Since requests that are eventually discarded also consume resources, admission control should be performed as early as possible in the lifetime of a web transaction. Depending on the workload, some server resources can be overutilized while the demand on other resources is low because certain types of requests utilize one resource more than others.

The implementation of admission control in the kernel of the operating system shows that this approach is more efficient and scalable than implementing the same scheme in user space. We also present an admission control architecture that performs admission control based on the current server resource utilization combined with knowledge about resource consumption of requests. Experiments demonstrate more than 40% higher throughput during overload compared to a standard server and several magnitudes lower response times.

This thesis also presents novel architectures and implementations of operating system support for predictable service guarantees. The Nemesis operating system provides applications with a guaranteed communication service using the developed TCP/IP implementation and the scheduling of server resources. SILK (Scout in the Linux kernel) is a new networking stack for the Linux operating system that is based on the Scout operating system. Experiments show that SILK enables prioritizing and other forms of service differentiation between network connections while running unmodified Linux applications.

Thiemo Voigt, Department of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden, E-mail: Thiemo.Voigt@sics.se

ISSN 0283-0574
ISSN 1101-1335
ISRN SICS-D--30--SE

Printed in Sweden by Elanders Gotab, Stockholm 2002.
Distributor: SICS, Box 1263, SE-164 29 Kista, Sweden.

To K&K

Acknowledgments

First of all I want to thank my supervisor Per Gunningberg. Already as a thesis student in Uppsala many years ago, I decided that if I do a PhD, I will do it for Per. During dark hours in the last years I sometimes regretted having started the PhD studies, but I do not remember (m)any instances where I regretted being Per's student. I have greatly benefitted from Per's technical knowledge, experience in writing and judging the values of papers and his broad contact net.

I am very grateful to Bengt Ahlgren, manager of the Computer and Network Architectures Laboratory (CNA) at SICS, who has also been my secondary advisor and co-authored one of the papers presented in this thesis. During the Pegasus II project, Bengt taught me a lot about writing papers. Probably even more important, Bengt has made this thesis possible by assigning me to projects suitable to pursue my PhD studies.

I am also grateful for the help I have got in the CNA lab. In particular, I am thankful to Laura Feeney and Ian Marsh, who have proofread many of my papers, as well as to Assar Westerlund, Björn Grönvall, Adam Dunkels and Lars Albertsson who answered many questions on UNIX and C programming. Thanks to all my colleagues both in the CNA lab and SICS for making SICS an exciting place to conduct research at. Thanks to my fellow PhD students and the rest of the people at DoCS, in particular the members of the Communication Research group.

One of the greatest experiences during my PhD studies was the internship at the IBM TJ Watson Research Center in Hawthorne, NY. It was an honour and pleasure to work with Renu Tewari, Ashish Mehra and Douglas Freimuth who are also co-authors of one of the papers in my thesis. Thanks also to Erich Nahum and Anees Shaikh for helping me to get the internship.

I also had the pleasure to work with Andy Bavier. Even a buggy TCP could not prevent us from having a great time while working on SILK. Andy provided also valuable comments on this thesis. Thanks also to Larry Peterson and Mike Wawrzoniak who co-authored the SILK paper.

I am grateful for all the help I got from the rest of the Nemesis crowds in Cambridge and Glasgow while working on Nemesis. In particular, without Austin Donnelly's help I might still try to boot Nemesis. Thanks to Steven Hand for coming to Sweden and acting as opponent for my licentiate thesis defense.

Thanks to Jakob Carlström and Jakob Engblom who proofread parts of my licentiate thesis as well as to Ingela Nyström and Arnold Pears for excellent feedback on parts of both my licentiate and PhD thesis.

Hans Hansson and Per Stenström, the directors of ARTES and PAMP, made my financial support for the last two years of my work possible for which I am grateful. I also want to thank Lars Björnfot, my industrial contact person in that project.

Finally, I want to thank my parents and the rest of my family. My deepest thanks go to my wife Kajsa whose love and support has made it possible for me to both accomplish this thesis and have a wonderful time with our son Kalle.

The work is supported in part by the CEC DG III Esprit LTR project 21917 Pegasus II with additional support from Telia. This work is also partially funded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for Strategic Research.

The Swedish Institute of Computer Science is sponsored by Telia, Ericsson, SaabTech Systems, FMV (Defence Materiel Administration), Green Cargo (Swedish freight railway operator), IBM, Hewlett-Packard and ABB.

This thesis is composed of the following papers. In the summary, the papers will be referred to as papers A through E.

[A] Thiemo Voigt and Bengt Ahlgren. Scheduling TCP in the Nemesis Operating System. *IFIP WG 6.1/WG 6.4 International Workshop on Protocols for High-Speed Networks*, Salem, MA, USA, August, 1999.

[B] Thiemo Voigt, Renu Tewari, Douglas Freimuth and Ashish Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. *Proceedings of Usenix Annual Technical Conference*, pages 189 – 202, Boston, MA, USA, June 2001.

[C] Thiemo Voigt and Per Gunningberg. Kernel-based Control of Persistent Web Server Connections. *ACM Performance Evaluation Review*, 29(2):20–25, September 2001.

[D] Thiemo Voigt and Per Gunningberg. Handling Multiple Bottlenecks in Web Servers Using Adaptive Inbound Controls. *Seventh International Workshop on Protocols for High-Speed Networks (PfHSN 2002)*, Berlin, Germany, April 2002.

[E] Andy Bavier, Thiemo Voigt, Mike Wawrzoniak, Larry Peterson and Per Gunningberg. SILK: Scout Paths in the Linux Kernel. Technical Report 2002-009, Department of Information Technology, Uppsala University, Uppsala, Sweden, February 2002.

Other Papers and Reports:
These papers constitute part of my thesis work but are not included in the thesis.

[1] Bengt Ahlgren and Thiemo Voigt. IP over ATM. *Pegasus II Project Deliverable*, Technical Report, September 1999.

[2] Bengt Ahlgren, Lars Albertsson and Thiemo Voigt. IPv4 and IP Multicast Functionality. *Pegasus II Project Deliverable*, Technical Report, September 1999.

[3] Bengt Ahlgren and Thiemo Voigt. IP QoS for Nemesis. *Pegasus II Project Deliverable*, Technical Report, September 1999.

[4] Thiemo Voigt. Providing Quality of Service to Networked Applications Using the Nemesis Operating System. *Ph. Lic. thesis*, Technical Report DoCS 99/113, Uppsala University, Sweden, October 1999.

[5] Thiemo Voigt, Renu Tewari and Ashish Mehra. In-Kernel Mechanisms for Adaptive Control of Overloaded Web Servers. *Eunice Open European Summer School*, Twente, The Netherlands, September 2000.

[6] Thiemo Voigt and Per Gunningberg. Dealing with Memory-intensive Web Requests. Technical Report 2001-010, Department of Information Technology, Uppsala University, Sweden, May 2001.

[7] Thiemo Voigt and Per Gunningberg. Handling Persistent Connections in Overloaded Web Servers. *Real-Time in Sweden 2001*, Halmstad, Sweden, August 2001.

[8] Thiemo Voigt. Overload Behaviour and Protection of Event-driven Web Servers. *International Workshop on Web Engineering, Networking 2002*, Pisa, Italy, May 2002.

[9] Thiemo Voigt and Per Gunningberg. Adaptive Resource-based Web Server Admission Control. *7th IEEE Symposium on Computers and Communication 2002*, Taormina/Giardini Naxos, Italy, July 2002.

Contents

1 Introduction

With the advent of the World Wide Web (WWW) the number of Internet users has grown steadily. In some countries more than half of the population uses the Internet regularly. In October 2001, the country with the highest rate of Internet penetration was Sweden with more than 63% of the population having Internet access according to Nua Internet Surveys [44]. At about the same time, the number of people all over the world having Internet access reached 500 million [44].

This growth in popularity, together with the appearance of new services such as e-commerce and on-line banking, has increased the demand made on the capacity of both the Internet infrastructure and web servers. Web servers consist of one or several computers that provide services such as searching and retrieval of documents and information, online banking and electronic commerce. To cope with this increasing demand, huge amounts of bandwidth have been added to the core of the Internet. Technical advances have been made in many areas that have enabled continuous operation of the Internet despite the increasing demands.

However, many users still experience the WWW more as a World Wide Wait than a satisfying medium for business and entertainment. The web response times are not always caused by a congested network, but often by overloaded and non-responding web servers. A web server is overloaded when the demand exceeds the capacity of the server, i.e., when a server receives more user requests than it can handle. In particular, during exciting, often unforeseeable events such as terror attacks or stock market panics, there is a dramatic increase of user requests to the affected web servers which makes it hard if not impossible to retrieve the requested information from these servers. This can be very annoying in case of a stock market panic. It can also lead to serious financial losses if shareholders cannot fulfill their intended financial affairs due to non-responding web servers.

Under normal load conditions when the rate of incoming requests is below the capacity of a server, the server can service all requests without introducing large delays. During high load, however, one or several of the critical server resources – network interface, disk, physical memory and CPU (Central Processing Unit) – become scarce, which may lead to low server throughput and customers experiencing long delays. In such situations, the server does not have sufficient resources to provide good service to all clients. Instead, the server should perform service differentiation which aims at providing better service, i.e., lower response time and higher throughput to preferred clients as opposed to regular clients. The latter may receive degraded or no service during overload. This thesis deals with architectures that enable service differentiation in overloaded Internet servers, in particular web servers.

Even if one is willing to pay an extra fee to receive guaranteed fast service also during unforeseeable events, banks do not offer such services. The follow on question is why not? One possible explanation is that it is not worth the effort

Figure 1: Interaction between client and web server

for the banks, but there are also technical reasons that do not make it feasible to provide such a service. Most traditional server operating systems, such as UNIX, are designed as time-sharing systems. The aim of such systems is to maximize system utilization, while simultaneously providing users, or processes, with a fair share of the CPU. In an overload situation each process gets a small but fair share of the CPU which means that everyone receives poor service. This is in contrast to the notion of service differentiation.

Service differentiation is also important in other situations, for example, when a service provider wants to give better service to requests associated with electronic purchases or other transactions that provide financial gain as opposed to data requests associated with users merely browsing the provider's site. Furthermore, there is a trend of co-hosting multiple customers' web sites on the same server (in this case a customer can be thought of as a company or an organization). Customers paying a higher fee expect better service for requests to their site than customers paying less for the hosting service.

1.1 Background

This section presents how web transactions in general are handled. Readers who are familiar with web transactions and web server operation may wish to proceed directly to Section 1.2.

1.1.1 Web Transactions and Web Servers

In the World Wide Web, users request content from web servers. Usually, a browser application such as Internet Explorer or Netscape Navigator running on the user's host[1] (the client), sends a request over the Internet to the web server. The web server responds by sending the requested object back to the client. The client's browser handles the response by displaying the requested web page in the browser's window. Figure 1 shows this interaction between the client and the server called a web transaction.

A web server is typically an application program running on the server host (Figure 2). It receives requests from clients over the network. Before the request is transmitted, a TCP (Transmission Control Protocol) connection is set up (see Figure 3, step 1–4) which requires three messages. The web server application is

[1] A host is a computer that is connected to the network.

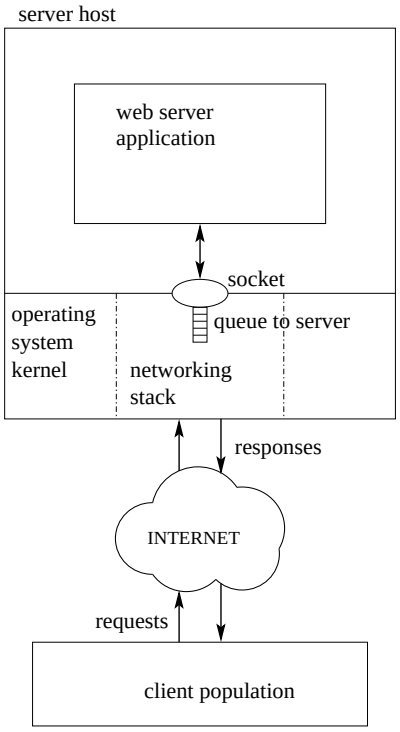

Figure 2: Simplified web server architecture

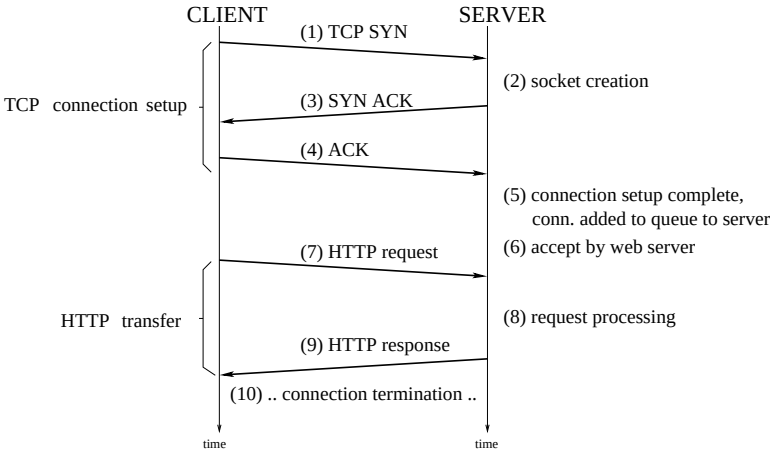

Figure 3: Simple web transaction

not involved in this setup, it is handled by the underlying network stack. After the connection is established (Figure 3, step 5), it is enqueued in the listen queue from which the web server application accepts it (step 6). When the web server application has accepted a connection, it awaits a request from the client, typically a Hypertext Transfer Protocol (HTTP) [11, 26] request (step 7). After processing the request (step 8), the web server returns the requested page to the client (step 9). After the server has transmitted the requested data the connection is closed (step 10).

HTTP is a stateless protocol, i.e., HTTP does not require web servers to keep any information about clients and their requests. Cookies are used to keep state in HTTP. A cookie is a small amount of state sent by the server to the client. The server expects the client to include the cookie in subsequent requests. This way, the server is able to maintain information about the client and its state within a session and across different sessions.

The operating system and the web server application use the socket interface to communicate with each other. The content of a message is copied between the operating system and the application, and vice versa. This data movement through the socket interface is a fairly time consuming operation.

Handling and processing requests utilizes web server resources. For example, the processing of messages and parsing of the URL (Uniform Resource Locator) that identifies a web page requires CPU time. Reading the file from the disk consumes disk bandwidth and transmitting the response requires bandwidth on the network interface.

1.1.2 The Internet and the World Wide Web

The Internet is the underlying infrastructure of the World Wide Web. The Internet can be regarded as a hierarchy of ISPs (Internet Service Providers) or networks, each having their own administration. As shown in Figure 4, web clients are usually connected to a local network, for example, a university network or a local ISP. Web servers can be placed anywhere in the network. Web clients communicate with servers over the Internet.

Not all requests that clients issue travel all the way to the web server. Some servers have mirrors or replicas, i.e., servers with a duplicate of the content of the original server. The ideal situation is to find an optimal replica for clients where optimal is based on proximity or on other criteria such as load [21]. The simplest scheme, by which a replica can be selected, is to embed replica identities in the URLs of the web pages. The HTTP protocol itself can also redirect a request to a replica by returning a particular response code.

Some requests also get redirected to web caches. A web cache is an intermediary host with storage facilities. A cache stores responses in order to reduce the response time and network bandwidth consumption for equivalent future requests [21]. That means that if another client subsequently requests the same page, the local copy can be provided immediately to satisfy the request. Since web caches primarily aim at reducing the response time, they are often placed

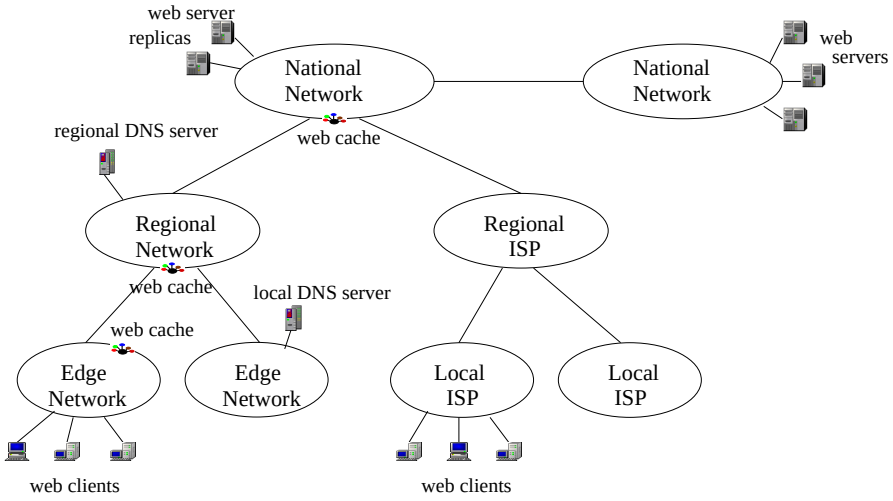

Figure 4: Internet and WWW infrastructure

close to clients. Another placement possibility for web caches are the access points between two different networks to avoid that the packets need to travel through the network [48].

The Domain Name System (DNS) plays an important part in the web infrastructure. Its task is the translation between hostnames, e.g., www.website.com, and IP (Internet Protocol) addresses. IP addresses are the identifiers for hosts in the Internet. The DNS also provides facilities to select server replicas for clients in a user-transparent manner. Instead of having only one possible IP address for each hostname, DNS can select among several servers depending on estimated proximity, load or other properties.

1.2 Problem Areas

Servers become overloaded when one or several critical resources become scarce. Server overload affects both the server throughput and the response time experienced by the clients. Figure 5 schematically illustrates the response time and total server throughput as functions of the request rate. The left part of the figure demonstrates how the response time increases with the server load. The response time is low as long as no server resource is overutilized. However, when the server resource bottleneck becomes overutilized, i.e., the bottleneck resource cannot keep up with the arrival rate of requests, the queue length to the resource bottleneck and thus the response time theoretically increases to infinity. This is depicted by the sudden increase of the response time.

The right part of Figure 5 depicts how the server throughput increases with the request rate until the request rate exceeds the capacity of the web server.

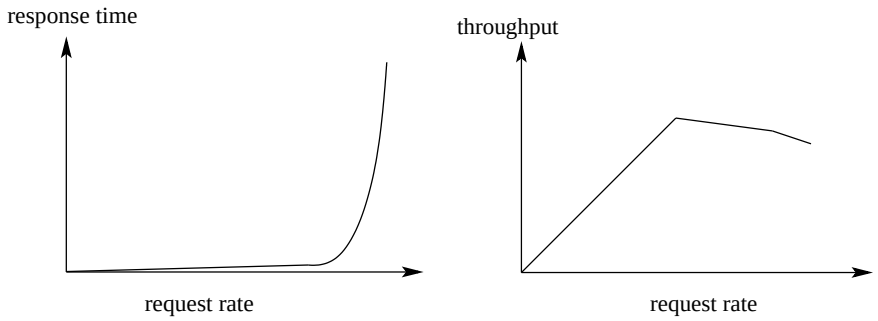

Figure 5: Impact of server load on response time and throughput

At this point, the throughput decreases due to the additional and unproductive time the CPU spends on processing incoming connection requests that are dropped when the listen queue is full. Moreover, the high rate of network interrupts prevents the web server application from making fast progress which contributes to the lower throughput. Lower server throughput leads to loss of revenue, while long delays cause user frustration and decrease task success and efficiency [17]. Users' tolerance for delay is application dependent, but often a threshold of 10 seconds for web interaction is mentioned in the literature [12].

In this thesis I will use the term architecture to define a set of mechanisms and their interaction, designed for a specific task, for example, service differentiation in overloaded web servers. An architecture describes how the mechanisms interact with each other and the environment, i.e., the operating system and the web server.

One way of reducing the load on individual servers is to utilize distributed web server architectures. These architectures distribute client requests to server replicas or caches as described in the previous section. Whereas traditional web cache proxies are placed close to clients to reduce client-perceived latency, reverse proxies are placed close to servers to reduce the load on them [34]. However, not all web data is cacheable, in particular dynamic and personalized data. During recent years, there has been an overall reduction in the fraction of traffic that is cacheable [34].

A current trend is to organize a number of servers in a cluster with front-ends or dispatchers that distribute the incoming requests among the servers [27]. If the capacity of the cluster is not sufficient, more machines can be added to the cluster. However, it is not unusual that the peak demand for web services is 100 times greater than the average demand [31]. Hence, provisioning for peak demand is not economically feasible, since this would imply that most servers are idle most of the time. Neither does it help to improve the efficiency of the web server or the underlying operating system, because also faster servers will become overloaded when the demand is sufficiently high.

Note that it is generally not possible to buffer requests until overload situations have ceased, since the duration of overload situations is unpredictable. Moreover, buffering requests increases the delay experienced by the clients.

In summary, while the approaches above contribute to reduce the load on an individual server, individual servers may still be confronted with a demand exceeding their capacity. In other words, individual servers may always experience overload situations where they do not have enough resources to process all requests in a timely manner. Thus, in an overload situation, we must perform service differentiation and determine which requests to serve and which ones to reject. The main focus of this thesis is to study architectures that enable service differentiation in overloaded individual and clustered web servers.

The content of this thesis is divided in two research areas, described in the next two sections. The first research area deals with mechanisms and architectures that perform overload protection and service differentiation by regulating the access to the server itself. The second area deals with the problem of providing predictable service by regulating the access to the critical server resources.

1.2.1 Mechanisms and Architectures for Server Overload Protection and Service Differentiation

Most web servers deploy rather simple schemes for admission control. For example, when the number of new requests enqueued exceeds a predefined threshold, additional incoming requests are dropped. Chen *et al.* state that such a tail-dropping admission control scheme requires careful system capacity planning and works well only in steady workload situations since the approach has problems coping with the highly non-steady and variable demands that web servers experience [19]. Furthermore, tail dropping without considering the identity of the requestor cannot provide any service differentiation. Hence, to provide differentiated access to a web server under high load, we need enhanced mechanisms for request classification and admission control. Request classification is needed to identify and classify the incoming requests to decide which service a request should receive, i.e., to which service class a request belongs to. Admission control mechanisms have to decide about the acceptance of the request, which can be based on several factors such as the request's service class, the expected resource requirements and the current server load.

Given the scarceness of available server resources during high load, the efficiency of these mechanisms is of major importance. Note that even requests that are eventually discarded consume resources. Abdelzaher and Bhatti noticed that under very high load, about 50% of the end-system utilization is wasted on connections that are eventually rejected [1]. Bhoj *et al.* have experienced that under certain conditions classifications can become the server bottleneck [14]. The earlier in the lifetime of a web transaction admission control is performed, the less resources are wasted in case of a rejection. However, the earlier admission control is performed, the less information about both the client and the requested object, including its potential resource consumption, is known.

Paper B presents mechanisms for performing efficient admission control and service differentiation in overloaded web servers.

Mechanisms for classification and admission control of individual requests are important parts of overload protection and service differentiation architectures for web servers. Depending on the admission control strategy all requests, or all requests belonging to a certain service class, may be rejected under high load. Admission control of requests should be triggered when the server starts to experience high load. Different architectures use different indicators of high load, e.g., the length of queues [13], CPU utilization [20] or a variety of other load indicators [31].

Many architectures try to avoid server overload by limiting the number of requests that are allowed to enter the system during a certain time unit [14, 31] or that are in the system concurrently [2]. Many of them use static thresholds as indicators of high load and limit the number of accepted requests when the threshold is exceeded. However, web server workloads change frequently, for example with the popularity of documents or services. If one chooses low thresholds, it is possible to guarantee low response times since no server resources are fully utilized. On the other hand, choosing low thresholds also leads to lower server throughput and thus loss of potential revenue. If one chooses high thresholds, it is possible to achieve higher utilization and throughput, but there is a risk of overload and high response times. Hence, in order to maximize throughput while keeping response times low, adaptation of the threshold values to the current workload can be advantageous.

Depending on the current workload, some server resources can be overutilized, while the demand on other resources is not very high because certain types of requests utilize one resource more than others. Paper D describes an architecture that sets the maximum number of requests admitted per time unit dynamically based on the current server resource utilization in combination with acquired knowledge about resource consumption of requests.

An overload protection architecture also needs to deal with persistent connections. Persistent connections allow clients to send several requests on the same TCP connection to reduce client latency and server overhead [41]. Persistent connections represent a challenging problem for web server admission control, since the HTTP header of the first request does not reveal any information about the resource consumption of the requests that may follow on the same connection. This problem is addressed in Paper C.

1.2.2 Operating System Support for Predictable Service

The goal of Quality of Service (QoS) is to provide *predictable* service to users or applications independent of the demand of other applications competing for the same resources. In order to provide QoS guarantees to applications, it is necessary that all resources used by, or on behalf of, an application are accounted for correctly. Without proper resource accounting, resources cannot be provided

to applications in a predictable way because one application may exceed its share which prevents other applications from receiving their shares. Traditional operating systems have problems with correct accounting of resources. For example, in UNIX systems the CPU time spent in the context of a network device interrupt, triggered by an arriving packet, is accounted to the interrupted application instead of the application the packet is destined for [23]. In microkernel environments work performed by shared servers is often not accounted to the right application [37]. This coarse-grained resource control is inappropriate for multimedia applications that are sensitive to variations of the delay. These applications need more fine-grained resource controls to, for example, avoid flicker in video displays.

One approach to providing fine-grained QoS guarantees is to design and build operating systems from scratch with the goal of fine-grained QoS in mind. Nemesis is such an operating system [37]. In Nemesis, applications can reserve CPU time and bandwidth on network interfaces. Paper A shows how the Nemesis operating system can provide applications a guaranteed communication service by scheduling CPU time and transmit bandwidth. A guaranteed communication service enables the transmission of data at a specified rate, provided that the bandwidth is not limited by the network.

One of the problems with operating systems developed from scratch is that their distribution is often limited. People do not want to invest a lot of time to get acquainted with and to learn a new system. To be able to fully exploit the features of new operating systems, applications must often be modified which people are hesitant to do. An alternative approach is to change the internals of an existing operating system while maintaining the user API (application programming interface). Paper E presents SILK (Scout in the Linux Kernel) which is a port of the Scout operating system [43] to run as a kernel module in the popular Linux operating system. SILK is a modular, configurable, communication-oriented operating system developed from scratch for small network appliances. By running in the Linux kernel, SILK can take advantage of existing Linux applications with small or no modifications at all.

1.2.3 Combining QoS and Admission Control Architectures

The QoS architectures described in the previous section are able to provide fine-grained QoS guarantees even during server overload by controlling access to server resources. However, low priority requests that have entered such a system and consumed resources, might be starved or must be preempted when high priority requests enter the system and consume the available resources. Combining the QoS architectures with the admission control architectures described in Section 1.2.1 avoids this problem by not admitting such low priority requests when the server is becoming overloaded. On the other hand, the admission control architectures are much more lightweight, meaning they only require small changes or rather additions to existing operating systems. Hence, when the architectural goal is merely to protect important customers from the

consequences of server overload, these admission control architectures are more appropriate. Comparing these two types of architectures with the service classes for quality of service in IP internetworks, the admission control architectures are comparable to the Controlled-Load Service [53], while the QoS architectures are comparable to the Guaranteed Quality of Service [50].

1.3 Method

The research method for the work presented in this thesis is mainly experimental. Experimental research often starts with either a potential or concrete problem.

The first step towards solving the problem is to find and formulate a hypothesis, i.e., an idea or statement that can be validated or invalidated. As an example, a hypothesis in Paper B is formulated as: "Kernel mechanisms for overload protection of web servers are more efficient than mechanisms implemented in user space".

The next step is to design experiments that validate or invalidate the hypothesis. In the case of my work, this phase also includes the design and implementation of a prototype such as admission control mechanisms in an existing operating system. Having a prototype that is complete enough, experiments need to be designed and conducted. Here, the experiments have been conducted in isolated, controlled environments. The advantage of conducting experiments in an isolated network is the possibility to obtain consistent and repeatable results. The disadvantage is that disturbances that may occur in real-world scenarios or in a complete system may not appear in a controlled testbed and, thus, not be taken into account properly.

When not working with real users generating requests, the choice of the workload and the workload generator is very important. The workload should both be realistic, i.e., it should conform with empirical measurements or use representative values such as typical requested file sizes from web servers, and at the same time give the desired effects. For example, many request generators use simple methods that cannot generate requests at a rate that exceeds the capacity of the web server and, thus, fail to evaluate web server behaviour during overload [8].

In the third step, the results of the experiments need to be collected and analyzed to see if they are conclusive and whether they validate the hypothesis or not. If the hypothesis can neither be validated nor invalidated, the experiment has to be redesigned.

The described process is iterative in the sense that the validation of a hypothesis often leads to a more fine-grained or completely new hypothesis. In particular, unexpected behaviour discovered in the experiments often leads to new insights and the formulation of new hypotheses.

1.4 Results and Scientific Contributions

The scientific contributions presented in this thesis are:

- Design of, and evaluation of, efficient in-kernel mechanisms for service differentiation and overload protection of web servers, and demonstrating the improved efficiency and scalability of the in-kernel mechanisms compared to the same mechanisms implemented in user space.
- A demonstration of the problem persistent connections cause for web server admission control and a kernel-based architecture that solves the problem. The architecture provides service differentiation judging the importance of persistent connections based on cookies.
- An adaptive admission control architecture that supervises multiple resource bottlenecks in server systems. The architecture uses TCP SYN policing and HTTP header-based connection control in a combined way to perform efficient and yet informed web server admission control.
- Demonstrating that the TCP/IP implementation in Nemesis can utilize the scheduling of CPU time and transmit bandwidth to provide applications with a guaranteed communication service.
- A new networking subsystem for Linux based on the Scout path architecture that is QoS-capable. The idea and evaluation of the concept of extended paths, which enables coscheduling of application and network processing.

Additional results in the form of prototypes that have been an outcome during the course of my work are:

- An IP version 4 implementation for Nemesis, including the transport protocols UDP and TCP as well as end host support for RSVP.
- An implementation of IP version 4 on top of ATM, running under Nemesis.
- Traffic control schemes to provide IP QoS to Nemesis applications.
- Prototype implementations of the proposed architectures including enhanced versions of the *sclient* [8] traffic generator.
- The IBM Linux Technology Center has ported one of the proposed mechanisms for service differentiation in web servers to Linux and distributes it as a Linux patch.

2 Summary of the Papers

2.1 Paper A

Scheduling TCP in the Nemesis Operating System

This paper was written within the context of the EU Esprit Pegasus II project. The aim of the project was to explore an operating system design that provides guaranteed quality of service to applications, in particular to multimedia applications. High-quality multimedia applications not only demand a specific amount of resources but also timely, usually periodic, access to resources.

In the Nemesis operating system [37], designed and implemented during the project, applications use shared library code to perform functionality usually provided by the operating system kernel. This feature enables correct accounting of all resources, which is a necessary prerequisite for enforcing and providing guaranteed access to resources. In Nemesis, CPU time, memory, disk I/O bandwidth as well as transmit bandwidth on network interfaces are resources that can be reserved.

The paper reports on the TCP/IP implementation for Nemesis. We study CPU scheduling of TCP/IP, the scheduling of network interface transmit bandwidth, and their interdependence in the context of the Nemesis operating system. We present a set of experiments which demonstrate the ability of Nemesis to provide appropriate end-system communication guarantees for the application. First, we show that the scheduling of transmit bandwidth can both be used as a rate limiter and to provide guaranteed transmit bandwidth. We measure the amount of CPU time an application needs to be able to run the TCP/IP protocol stack and to send data at a particular speed. Our experiments show that the CPU time needed to run the protocol stack increases linearly with the amount of data sent for a given packet size. We also show that the measured values hold, even when several applications strive for CPU time and transmit bandwidth.

The schedulers in Nemesis are primarily designed for networked multimedia applications and periodic access to resources. It is not obvious that TCP, which is designed for reliable data transfer, should work well in such an environment. Nevertheless, we were able to demonstrate that TCP/IP can utilize CPU and transmit bandwidth scheduling of this type to provide applications with a guaranteed communication service, provided that the bandwidth is not limited by the network.

Comments

I presented this paper at the sixth International Workshop on Protocols for High-Speed Networks, held in Salem, MA, USA in August 1999.

Most of the architecture design work was done by me, partly with help of Bengt Ahlgren. I designed and conducted the experiments. The paper was

written by me except for parts of the introduction which were written by Bengt Ahlgren.

2.2 Paper B

Kernel Mechanisms for Service Differentiation in Overloaded Web Servers

Web servers need to be protected from overload since overload can lead to high response times, low throughput and even loss of service. Also, it is highly desirable that web servers provide continuous service during overload, at least to preferred customers.

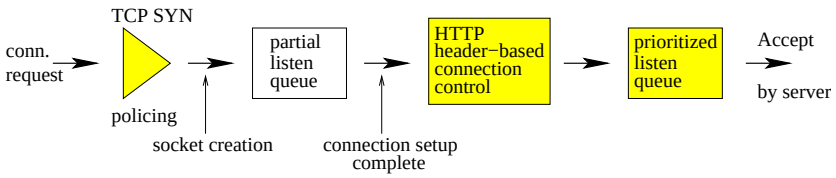

Figure 6: Kernel mechanisms

Most existing web server admission control architectures are implemented in user space. However, performing admission control in user space implies that requests that are later discarded consume a non-negligible amount of resources. Mogul and Ramakrishan have demonstrated the benefit of dropping packets early [42]. Following the principle of "early discard", we have designed and implemented kernel mechanisms that protect web servers against overload by providing admission control and service differentiation based on customer site, client and application layer information. Figure 6 shows the placement of the mechanisms:

- The first mechanism, *TCP SYN policing*, is located at the bottom of the protocol stack. It limits the acceptance of new SYN packets based on compliance with a token bucket policer. A token bucket policer is a token bucket used for admission control. It has a rate, denoting the average number of requests accepted per second, and a burst, denoting the maximum number of requests admitted at one time.

- The next mechanism, *HTTP header-based connection control* is located higher up in the stack. It is activated when the HTTP header is received. It enables admission control and priority based on application-layer information contained in the header, for example, URLs and cookies.

- The third mechanism, *prioritized listen queue*, is located at the end of the TCP 3-way handshake, i.e., when the connection is established. This

mechanism supports different priority levels among established connections by inserting connections into the listen queue according to their priority.

The first mechanism is the least costly but the most coarse-grained. The third is the most costly but provides the most fine-grained admission control.

We have implemented these controls in the AIX 5.0 operating system kernel as a loadable module. We present experimental results to demonstrate that these mechanisms effectively provide selective connection discard and service differentiation in an overloaded server. We also compare the performance of our mechanisms against the same application layer mechanisms added in the Apache 1.3.12 server.

The contribution of this paper is the design and evaluation of the kernel-based mechanisms. In particular, we show that the implementation of the mechanisms in the kernel is much more efficient and scalable compared to user space implementations.

Comments

Most of the work described in this paper was done during my internship at IBM TJ Watson Research Center in Spring/Summer 2000. I presented the paper at the Usenix 2001 Annual Technical Conference held in Boston, MA, USA in June 2001.

The prioritized listen queue mechanism presented in the paper has been ported to Linux at the IBM Linux Technology Center and is commercially distributed as a kernel patch. Discussions about integrating the mechanism into the standard Linux kernel are underway.

Most of the architecture design work was done by me together with Ashish Mehra and Renu Tewari. I implemented the mechanisms and designed and conducted the experiments. Renu Tewari and I wrote the paper together. Renu Tewari focused more on the introduction while I did most of the work on the experimental sections.

2.3 Paper C

Kernel-based Control of Persistent Web Server Connections

This paper builds on the work described in Paper B. Paper C extends that work by presenting a solution for handling persistent web server connections. This problem is ignored by most of the web server architectures described in the literature.

Web servers use admission control for overload protection. Some web servers base their admission decision on information found in the HTTP header. Persistent connections allow HTTP clients to send several requests on the same TCP connection to reduce client latency and server overhead [41]. Using the

same TCP connection for several requests makes admission control more difficult, since the admission control decision should be performed when the first request is received. However, the HTTP header of the first request does not reveal any information about the resource consumption of the following requests on the same connection. Thus, persistent connections make admission control a trade-off. If one is too conservative, and sets low acceptance rates, potential customers might be rejected unnecessarily, resulting in loss of revenue. If one is too optimistic the server may become overloaded, with long response times and low throughput as a possible consequence. Our solution avoids uncontrollable overload while maximizing access.

If there is an overload situation caused by resource consumption of persistent connections we abort persistent connections. But we do not abort connections blindly. Instead, we preserve connections regarded as important and abort connections considered less important. For example, a connection can be regarded as important when the client has placed some items in a shopping bag.

The admission control mechanism judges the importance of persistent connections based on cookies in the HTTP header. The web application decides when a cookie denoting the importance of the connection should be sent to the client. Using cookies has several advantages: Cookies are a widely used technique; they can contain long-lasting information such as customer identification; and they are easy to remove or update. We present experiments demonstrating that our approach prevents server overload and provides service differentiation between important and less important connections under high load.

The key contribution of this paper is the kernel-based architecture that prevents overload in web servers caused by persistent connections. To our knowledge, this paper is the first to provide a solution for the challenges of persistent connections.

Comments

I presented an extended version of this paper at the workshop PAWS 2001, Performance and Architecture of Web Servers. This workshop was held in conjunction with the ACM SIGMETRICS conference in Boston, MA, June 2001. The version presented in this thesis appeared in the ACM Performance Evaluation Review.

The work described in the paper was done by me with discussions with Per Gunningberg.

2.4 Paper D

Handling Multiple Bottlenecks in Web Servers Using Adaptive Inbound Controls

This paper describes an adaptive admission control architecture that uses mechanisms presented in Paper B. Servers become overloaded when one or several

critical resources, such as network interface, CPU or disk, are overutilized and become the bottleneck of the server system. The key idea of Paper D is to avoid server overload by preventing overutilization of specific server resources using adaptive inbound controls.

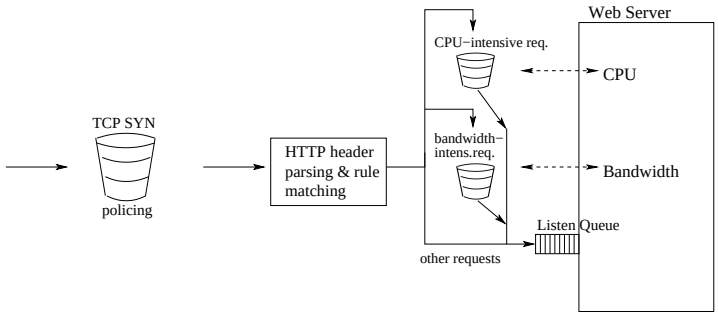

Figure 7: Admission Control Architecture

Our idea is to collect all web objects that, when requested, are the main consumers of the same server resource into one directory. Thus, we have one directory for each supervised resource. We associate a filter rule with each directory in that maps resource-intensive requests to the corresponding token bucket policer. Hence, we can use HTTP header-based connection control to avoid overutilization of specific resources. For example, CPU-intensive scripts can reside in the web server's `/cgi-bin` directory and a filter rule specifying the URL prefix `/cgi-bin` can be associated with it. For each of the critical resources, we use a feedback control loop that adapts the token rate at which we accept requests in order to avoid overutilization of the resource. We call our approach, illustrated in Figure 7, resource-based admission control.

We do not perform resource-based admission control on all requests. Requests such as those for small static files do not put significant load on one resource. However, if requested at a sufficiently high rate, these requests can still cause server overload. When such situations arise we use *TCP SYN policing* applied to all requests, independent of resource requirements. Using SYN policing the admission of connection requests is based on network-level attributes, i.e. on IP addresses and port numbers, and not on fine-grained HTTP header attributes.

We have implemented our admission control architecture in the Linux operating system. Our experiments show that the combination of resource-based admission control and TCP SYN policing works and adapts the rates as expected for our load scenarios. When resource-based admission control alone cannot prevent server overload, TCP SYN policing becomes active and high throughput and low response times can be sustained even when the request demand is high. We achieve more than 40% higher throughput and several magnitudes lower

response times during overload compared to a standard Apache on Linux configuration. We also show that the adaptation mechanisms can cope with bursty request arrival distributions.

The architecture is targeted towards single node servers or to back-end servers in a web server cluster. We believe that the architecture can easily be extended to web server clusters and enhance sophisticated request distribution schemes such as the Harvard Array of Clustered Computers (HACC) [55] and Locality-aware Request Distribution (LARD) [46]. In an extended architecture the front-end performs resource-based admission control. The back-end servers monitor the utilization of each critical resource and propagate the values to the front-end. Based on these values, the front-end updates the rates for the different token bucket policers. After the original distribution scheme has selected the node that is to handle the request, compliance with the corresponding token bucket ensures that critical resources on the back-ends are not overutilized.

The main contribution of this paper is the adaptive admission control architecture that handles multiple resource bottlenecks in server systems.

Comments

This paper has been accepted for the seventh International Workshop on Protocols for High-Speed Networks, to be held in Berlin, Germany, in April 2002, where I will present the paper.

The work described in the paper was done by me with discussions with Per Gunningberg.

2.5 Paper E

SILK: Scout Paths in the Linux Kernel

A lot of research effort has been invested into operating system architectures for providing QoS to applications. Some efforts have focused on new QoS features and abstractions to existing operating systems, while others have built new operating systems from scratch. However, the results of these efforts have hardly been put to general use, so far. New QoS mechanisms for mainstream operating systems are often only available for specific versions of the operating system. Due to feature interaction problems between different kernel patches, it is often impossible to combine several of these mechanisms into one system.

Scout [43] is a modular, configurable, communication-oriented operating system tailored for small network appliances. Scout combines features such as early demultiplexing, early dropping, resource accounting, explicit scheduling and extensibility into a single abstraction called a *path*.

SILK is a port of the Scout operating system to run as a downloadable kernel module in a standard Linux 2.4 kernel. SILK can replace the Linux networking subsystem. Regular Linux applications can use SILK which is demonstrated using the popular Apache web server. Introducing the path concept into the Linux

networking subsystem enables prioritizing and other forms of service differentiation between different network connections. Additionally, SILK coordinates the scheduling of applications and paths in the networking stack by *extending paths* into the application.

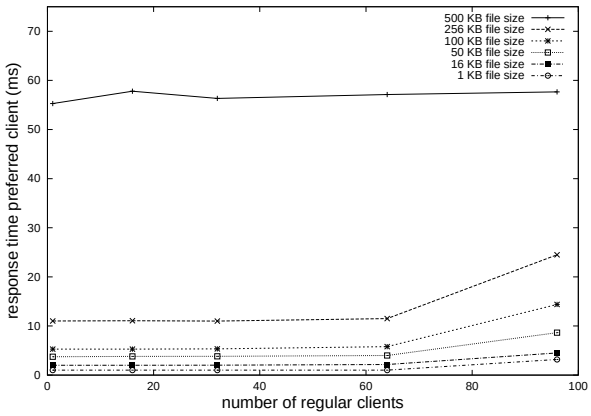

Figure 8: Response time preferred client

Our results show that SILK's performance is still comparable to the native Linux networking stack, i.e., using paths does not lead to performance loss. We also compare latency and throughput for preferred and regular clients using a fixed priority scheduler in SILK. Figure 8 illustrates that SILK provides almost constant response time for preferred clients independent of the number of regular clients accessing the server simultaneously. Without priorities, the response time would increase linearly with the number of clients, since each client receives about $1/n$ of the resources when n clients are active simultaneously.

The contributions of this paper are a networking subsystem for Linux based on the Scout path architecture and the concept of extending paths into the application.

Comments

This paper has been published as Technical Report 2002-009, Department of Information Technology, Uppsala University, Uppsala, Sweden, February 2002.

Andy Bavier implemented most of SILK. I assisted in debugging the system (in particular its TCP implementation) and conducted most of the evaluation experiments. Andy and I wrote the paper together, my focus was on the experimental section.

3 Related Work

This section presents related work. The discussion is divided into the two problem areas presented in Section 1.2.

3.1 Overload Protection and Service Differentiation for Web Servers

One of the main design objectives addressed by our admission control architectures is to employ efficient early connection discard mechanisms that provide overload protection and service differentiation for web servers. Many architectures ignore the importance of efficient admission control and presumably reject requests after passing them to user space [2, 19, 20, 33, 38]. Other admission control and service differentiation architectures such as WebQoS [13] and Web2k [14] are deployed in user space. In these architectures admission control is less efficient than in our kernel-based architecture.

A few web server admission control architectures adhere to the principle of early discard. Kant *et al.* moved overload protection into intelligent network cards [30]. While more efficient by off-loading the host, their approach is less flexible since it relies on special hardware. The performance gains are unknown. Jamjoom *et al.* use a mechanism similar to TCP SYN policing to avoid server overload [31]. Their mechanism bases the admission decision on network-level information such as IP addresses and port numbers. Hence, they cannot discriminate between different resource bottlenecks, but have to reduce the acceptance rate for all requests when only one resource is overutilized.

Service differentiation can also be realized by scheduling of server processes and by dynamically partitioning server nodes in a web server cluster. Almeida *et al.* assign different priorities to the processes handling requests [3]. In their approach the application, i.e., the web server, classifies the requests and assigns scheduling priorities. In a similar approach, Eggert and Heidemann propose to lower the priority of processes serving less important requests [24]. They also propose limiting the available bandwidth and the number of server processes for less important requests. While we perform service differentiation before a request is accepted by the web server, the approaches above perform service differentiation when scheduling or processing requests. Combining these approaches would further decrease the impact of low priority requests on the service high priority requests receive.

One approach to provide service differentiation in server clusters is by dynamically partitioning server nodes and forwarding different classes of requests to different partitions [56, 16]. The aim is to dynamically adjust the server partitions, not to perform efficient admission control. A similar approach designed for single server nodes is to reallocate the number of server processes for each service class. Abdelzaher *et al.* implement such an approach [39]. They enforce relative delays among service classes using a feedback control loop to reallocate the number of server processes for each service class. In their work, none of the

critical resources is overutilized. Instead, a peculiar bottleneck introduced by persistent connections causes large delays.

There are also other approaches to deal with server overload such as adapting the delivered content [1].

3.2 Operating System Support for Predictable Service

Both Scout and Nemesis are operating systems built from scratch to provide QoS to applications. Designing and implementing operating systems from scratch is a major effort, which is one reason why a lot of novel mechanisms and abstractions for QoS have been implemented in mainstream operating systems instead.

Among these new abstractions are resource containers, virtual services and processor reserves. Resource containers [9] present an abstraction that encompasses all system resources that a server uses to perform an independent activity, such as serving one client connection. All user and kernel processing time and other resource consumption such as memory is accounted to a resource container. Resource containers are used in conjunction with Lazy Receiver Processing (LRP), a network subsystem architecture that includes early demultiplexing and protocol processing at the priority of the receiving application [23]. Cluster reserves extend resource containers to server clusters [6] to provide differentiated and predictable quality of service in clustered web server systems. On one hand, these abstractions require more changes to the operating system than, for example, the admission control architecture presented in Paper D. On the other hand, the abstractions are implemented in an existing operating system and not in an operating system designed from scratch such as Nemesis or Scout.

Reumann *et al.* have presented *virtual services*, an abstraction that provides resource partitioning and management [47]. Virtual services can enhance web server overload protection architectures by dynamically controlling the number of processes a web server is allowed to fork.

Processor reserves [40] are used to provide QoS for multimedia applications in microkernel environments such as the Mach microkernel. Applications can make CPU reservations, which are guaranteed by the system, even in the presence of shared servers. Nemesis avoids this potential problem by using shared libraries instead of shared servers.

Several other researchers have explored ways to provide QoS guarantees to networked applications by controlling bandwidth and CPU. Examples include Yau and Lam's migrating sockets [54], Lakshman *et al.*'s *Adaptive Quality of service Architecture* [35] and Gopalakrishnan and Parulkar's real-time upcalls [28]. QLinux provides fair queuing mechanisms for CPU and network packets as well as an LRP networking subsystem and in addition an advanced disk scheduling algorithm [51]. None of these QoS provisioning methods is able to completely solve the problem of "QoS crosstalk", or in other words, to provide perfect performance isolation.

Rialto [32] is designed and built from scratch to support coexisting independent real-time and non-real-time programs. An abstraction called an *activity* is

the entity to which resources are allocated and charged. Another QoS operating system of this kind is Eclipse. Eclipse's proportional share schedulers and the corresponding API have also been ported to FreeBSD [15]. *Resource kernels* provide applications with explicit guarantees to system resources through abstractions such as CPU Reserves [45]. The portable implementation of a resource kernel implemented in Linux shares some goals with SILK, for example modularity and minimal changes to Linux. In order to increase performance, flexibility and functionality of applications, Exokernels [25] provide applications with a large degree of control over the physical resources, similar to the Nemesis operating system.

4 Conclusions and Future Work

In this thesis I present mechanisms and architectures for service differentiation and overload protection in web servers. In order to reduce the resources spent on requests that are eventually discarded, admission control should be performed as early as possible in the lifetime of web transactions. However, the earlier admission control is performed, the less information about the request, its resource requirements and its originator, is available. Deferred admission control enables a more informed control decision. Therefore, many web server admission control schemes are implemented in the web server application. Our research demonstrates that it is both desirable and possible to perform early but yet informed web server admission control.

This research has been conducted under the assumption of a single node web server. Modern web servers are often built as clusters with one or several front-ends. In Paper D we discuss how to extend the presented architecture towards web clusters. The implementation and evaluation of the extended architecture would highlight both the performance gains and potential problems, such as the scalability of the cluster. This is left for future work.

The admission control architectures in this thesis have been implemented as prototypes. They include most, but not all, details required for a production system. It would also be interesting to study the impact of the proposed mechanisms on user perception during overload.

My other contribution to providing service differentiation is the work on SILK and Nemesis. These novel operating system architectures can provide applications with fine-grained QoS guarantees. SILK suggests that it is feasible to adopt main abstractions from an operating system developed from scratch to provide fine-grained QoS guarantees in mainstream operating systems.

The work on SILK is still ongoing and will include work with more advanced schedulers such as Weighted Fair Queueing to provide web server QoS. We also plan to port our adaptive admission control architecture to SILK. Then we could study how to regulate both access to the web server itself and access to the critical server resources.

References

[1] T. Abdelzaher and N. Bhatti. Web content adaptation to improve server overload behavior. In *8th International World Wide Web Conference*, Toronto, Canada, May 1999.

[2] T. Abdelzaher and C. Lu. Modeling and performance control of Internet servers. In *IEEE Conference on Decision and Control*, Sydney, Australia, December 2000.

[3] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated levels of service in web content hosting. In *Internet Server Performance Workshop*, Madison, WI, USA, March 1999.

[4] M. Arlitt and C. Williamson. Web server workload characterization: The search for invariants. In *Proc. of ACM SIGMETRICS*, pages 126–137, Philadelphia, PA, USA, April 1996.

[5] M. Aron. *Differentiated and Predictable Quality of Service in Web Server Systems*. PhD thesis, Rice University, Houston, TX, USA, October 2000.

[6] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Proc. of ACM SIGMETRICS*, pages 90–101, Santa Clara, CA, USA, June 2000.

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

[8] G. Banga and P. Druschel. Measuring the capacity of a web server. In *USENIX Symposium on Internet Technologies and Systems*, pages 61–71, Monterey, CA, USA, December 1997.

[9] G. Banga, P. Druschel, and J. Mogul. Resource containers: a new facility for resource management in server systems. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, USA, February 1999.

[10] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proc. of ACM SIGMETRICS*, pages 151–160, Madison, WI, USA, June 1998.

[11] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol - http/1.0. Internet RFC 1945, May 1996.

[12] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. In *9th International World Wide Web Conference*, Amsterdam, The Netherlands, May 2000.

[13] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, 7(4):36–43, September 1999.

[14] P. Bhoj, S. Ramanathan, and S. Singhal. Web2k: Bringing QoS to web servers. Technical Report HPL-2000-61, HP, May 2000.

[15] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Retrofitting quality of service into a time-sharing operating system. In *Proc. of Usenix Annual Technical Conference*, pages 15–26, Monterey, CA, USA, June 1999.

[16] V. Cardellini, E. Casalicchio, M. Colajanni, and M. Mambelli. Web switch support for differentiated services. *ACM Performance Evaluation Review*, 29(2):20–25, September 2001.

[17] J. Carlström and R. Rom. Application-aware admission control and scheduling in web servers. In *IEEE Infocom 2002*, New York, NY, USA, June 2002.

[18] E. Casalicchio and M. Colajanni. A client-aware dispatching algorithm for web clusters providing multiple services. In *10th International World Wide Web Conference*, Hong Kong, China, May 2001.

[19] X. Chen, H. Chen, and P. Mohapatra. An admission control scheme for predictable server response time for web accesses. In *10th International World Wide Web Conference*, Hong Kong, China, May 2001.

[20] L. Cherkasova and P. Phaal. Session based admission control: a mechanism for improving the performance of an overloaded web server. Technical Report HPL-98-119, HP, 1999.

[21] I. Cooper, I. Melve, and G. Tomlinson. Internet web replication and caching taxonomy. Internet RFC 3040, January 2001.

[22] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *USENIX Symposium on Internet Technologies and Systems*, pages 243–245, Boulder, CO, USA, October 1999.

[23] P. Druschel and G. Banga. Lazy receiver processing (LRP): a network subsystem architecture for server systems. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 91–105, Seattle, WA, USA, October 1996.

[24] L. Eggert and J. Heidemann. Application-level differentiated services for web servers. *World Wide Web Journal*, 3(2):133–142, September 1999.

[25] D. Engler, F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *15th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review*, pages 251–266, Copper Mountain Resort, CO, USA, 1995.

[26] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol - http/1.1. Internet RFC 2616, June 1999.

[27] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier. Cluster-based scalable network services. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 78–91, Saint-Malo, France, October 1997.

[28] R. Gopalakrishnan and G. M. Parulkar. Efficient user space protocol implementations with QoS guarantees using real-time upcalls. *IEEE/ACM Transactions on Networking*, 6(4):374–388, August 1998.

[29] S. Hand. Self-paging in the Nemesis operating system. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 73–86, New Orleans, LA, USA, February 1999.

[30] R. Iyer, V. Tewari, and K. Kant. Overload control mechansims for web servers. In *Performance and QoS of Next Generation Networks*, Nagoya, Japan, November 2000.

[31] H. Jamjoom and J. Reumann. Qguard: Protecting internet servers from overload. Technical Report CSE-TR-427-00, University of Michigan, 2000.

[32] 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*, pages 249–256, Connemara, Ireland, September 1996.

[33] V. Kanodia and E. Knightly. Multi-class latency-bounded web servers. In *International Workshop on Quality of Service*, pages 231–239, Pittsburgh, USA, June 2000.

[34] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*. Addison-Wesley, 2001.

[35] K. Lakshman, R. Yavatkar, and R. Finkel. Integrated CPU and network-I/O QoS management in an endsystem. In *7th International Workshop on Quality of Service*, pages 167–178, New York, NY, USA, April 1997.

[36] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol processing in real-time Mach. In *Real-Time Technology and Application Symposium*, pages 115–123, Boston, MA, USA, June 1996.

[37] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, Sep. 1996.

[38] K. Li and S. Jamin. A measurement-based admission controlled web server. In *IEEE Infocom 2000*, Tel Aviv, Isreal, March 2000.

[39] C. Lu, T. Abdelzaher, J. Stankovic, and S. Son. A feedback control approach for guaranteeing relative delays in web servers. In *Real-Time Technology and Application Symposium*, TaiPei, Taiwan, June 2001.

[40] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Boston, MA, USA, May 1994.

[41] J. C. Mogul. The case for persistent-connection HTTP. In *SIGCOMM '95 Conference Proceedings*, pages 299–313, Cambridge, MA, USA, August 1995. ACM SIGCOMM Computer Communication Review, 25(4).

[42] J. C. Mogul and K. K. Ramakrishan. Eliminating receive livelock in an interrupt-driven kernel. In *Proc. of USENIX Annual Technical Conference*, San Diego, CA, USA, January 1996.

[43] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 153–168, Seattle, WA, USA, October 1996.

[44] Nua. Nua Internet Surveys. http://www.nua.ie/surveys/analysis/weekly_editorial/archives/issue1no197.html, October 2001.

[45] S. Oikawa and R. Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Real-Time Technology and Application Symposium*, Vancouver, Canada, June 1999.

[46] V. Pai, M. Aron, G. Banga, M. Svendsen, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, USA, October 1998.

[47] J. Reumann, A. Mehra, K. Shin, and D. Kandlur. Virtual services: A new abstraction for server consolidation. In *Proc. of USENIX Annual Technical Conference*, San Diego, CA, USA, June 2000.

[48] P. Rodriguez, C. Spanner, and E. Biersack. Web caching architectures: hierarchical and distributed caching. In *4th International Caching Workshop*, San Diego, CA, USA, April 1999.

[49] J. Schiller and P. Gunningberg. Feasibility of a software-based ATM cell-level scheduler with advanced shaping. In *Broadband Communications'98*, Stuttgart, Germany, April 1998.

[50] S. Shenker, C. Partridge, and R. Guerin. Specification of guaranteed quality of service. RFC 2212, September 1997.

[51] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. Vin. Application performance in the QLinux multimedia operating system. In *Eighth ACM Conference on Multimedia*, pages 127–136, Los Angeles, CA, USA, November 2000.

[52] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *ACM Symposium on Operating Systems Principles*, pages 230–243, Chateau Lake Louise, Banff, Canada, October 2001.

[53] J. Wroclawski. Specification of the controlled load quality of service. RFC 2211, September 1997.

[54] D. K.Y. Yau and S. S. Lam. Migrating sockets - end system support for networking with Quality of Service guarantees. *IEEE/ACM Transactions on Networking*, 6(6):700–716, Dec. 1998.

[55] X. Zhang, M. Barrientos, J. Chen, and M. Seltzer. HACC: An architecture for cluster-based web servers. In *Third Usenix Windows NT Symposium*, pages 155–164, Seattle, WA, July 1999.

[56] H. Zhu, H. Tang, and T. Yang. Demand-driven service differentiation in cluster-based network servers. In *IEEE Infocom 2001*, Anchorage, AK, USA, April 2001.

Paper A

Thiemo Voigt and Bengt Ahlgren. Scheduling TCP in the Nemesis Operating System. *IFIP WG 6.1/WG 6.4 International Workshop on Protocols for High-Speed Networks*, Salem, MA, USA, August, 1999.

Scheduling TCP in the Nemesis Operating System[◇]

Thiemo Voigt
(thiemo@sics.se)

Bengt Ahlgren
(bengta@sics.se)

Swedish Institute of Computer Science
Box 1263, SE-16429 Kista, Sweden

Abstract

The Nemesis operating system is designed to provide Quality of Service to applications. Nemesis also allows applications to reserve CPU time and transmit bandwidth on network interfaces. We have implemented a TCP for Nemesis that makes use of these guarantees.

We show that the Nemesis transmit scheduler rate-controls TCP traffic and thus leads to predictable traffic behavior when applications choose not to utilize non-allocated bandwidth. Applications that want to make use of the non-allocated transmit bandwidth receive the guaranteed bandwidth plus a share of the non-allocated bandwidth.

We also study the impact of the guaranteed fraction of CPU time on the throughput that networked applications achieve. We measure the amount of CPU time applications have to reserve in order to run the TCP protocol stack and send data at a particular speed. We show that these values hold even when several applications strive for CPU time and transmit bandwidth.

1 Introduction

Providing a deterministic service quality from a distributed application involves ensuring service quality in a whole chain of systems. Both the communication network and the computer platform on which the application is implemented have to provide service guarantees. In this paper we assume that the network provides the necessary service quality and focus on the mechanisms needed in the end-system computer platform to make it deliver the network quality of service to the application. These mechanisms include providing service guarantees to the software implementing the communication protocols, i.e., guaranteed CPU time, and the allocation of transmit bandwidth on the network interface. In this paper we study CPU scheduling of a TCP/IP implementation, the scheduling of network interface transmit bandwidth and their interdependence in the context of the Nemesis operating system.

The Nemesis operating system [6] is designed to provide guaranteed quality of service (QoS) to applications. In order to provide guarantees it is necessary

[◇] This work is supported in part by the CEC DG III Esprit LTR project 21917 Pegasus II.

that all resources used by or on behalf of an application are accounted for correctly. In this respect, shared servers are a problem since they make it hard to charge the correct application for the resources used. In Nemesis the use of shared servers is instead reduced to a minimum. This leads to the *vertical structure* of Nemesis. Besides CPU time and disk I/O bandwidth, Nemesis regards transmit bandwidth on network interfaces as a resource that can be reserved.

We present a set of experiments which demonstrate the ability of Nemesis to provide the appropriate end-system communication guarantees for the application. First we show that the scheduling of transmit bandwidth can both be used as a rate limiter and to provide guaranteed transmit bandwidth. We measure the amount of CPU time an application needs in order to be able run the TCP/IP protocol stack and send data at a particular speed. Our experiments show that the CPU time to run the protocol stack increases linearly with the amount of data sent for a given packet size. We also show that the measured values hold even when several applications strive for CPU time and transmit bandwidth.

A possible usage of our scheme is an Internet service provider running one Nemesis host with several web servers for a number of customers with different performance requirements. Also other applications that need reliable data transfer can be run concurrently with guaranteed progress according to their reservations.

The contribution made in this paper is showing that the TCP/IP implementation in Nemesis can utilize the scheduling of CPU time and transmit bandwidth to provide the application with a guaranteed communication service. The scheduling of transmit bandwidth allows to rate-control TCP which can be useful for not exceeding a given traffic contract. We believe that this is difficult to achieve with traditional operating systems.

In the next section we briefly describe the features of Nemesis relevant for our work. In section three we give an overview on the design and implementation of the Nemesis TCP. Section four presents results and performance figures. In section five we discuss related work while section six concludes the paper.

2 Nemesis

The *vertical structure* of the Nemesis operating system is shown in Figure 1. Applications use shared library code to perform functionality usually associated with the operating system.

The Nemesis kernel is very small, consisting of the scheduler and activations of domains, the interrupt and trap handlers, support for inter-domain communication as well as some processor control. There are no kernel threads. A *domain* is similar to a Unix process. Trusted domains, such as device drivers, can register interrupt handlers and affect the processor mode. Despite having extra rights, trusted domains are scheduled like all other domains.

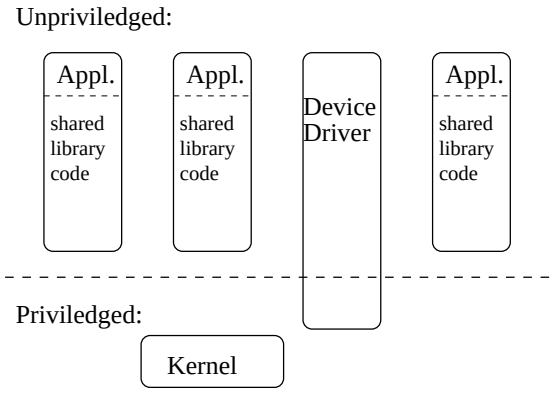

Figure 1: The Nemesis structure.

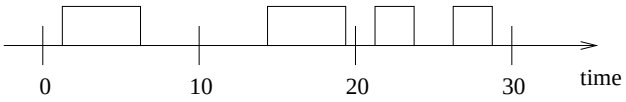

Figure 2: Scheduling of an application over three periods

Nemesis has a single virtual address space which makes it easy to share data. Each domain still has its own memory protection on this address space, making lightweight inter-domain communication possible. Shared memory is used to transport the marshalled arguments and event counts synchronize the shared buffers.

Shared libraries export one or more strongly-typed *interfaces* which are specified in an interface description language called MIDDL. Interface descriptions comprise types, exceptions and methods. Having acquired a so-called binding or *interface reference* [6], domains can call methods of interfaces exported by other domains.

2.1 QoS Guarantees and Scheduling

Applications can request a share of the CPU time. Application requests consist of a tuple containing a period p, a time slice s and a flag x denoting whether the application wants to use *extra time*, i.e., a fair share of the non-allocated CPU time. If the reservation succeeds, the application is guaranteed a portion of s time units in each period of length p. The application is not guaranteed an atomic slice. The slice may be split up into several smaller parts.

An example is shown in Figure 2. An application has specified a period p of 10 time units, a slice time s of 5 units and the extra time flag x is set to `false`. In the first two periods the application receives one slice of 5 time units with the slice starting at different times. In the third period the application receives

its slice time split up into two parts.

The scheduling is done using the Atropos scheduler [6] which internally uses an Earliest Deadline First algorithm. The deadlines are not specified by the applications but computed from the applications' specifications.

A similar scheme is used for the reservation of transmit bandwidth. Applications specify their reservations using the same tuple as described above, i.e., the reservations are expressed as transmission time and not as bandwidth. As for CPU time, the deadlines are not specified by the applications but computed from the applications' specifications. Setting the extra flag x to `true` means that the application wants a fair share of the non-allocated part of the transmit bandwidth.

2.2 Rbufs and I/0 Channels

Nemesis deploys a mechanism called *Rbufs* [3] for the inter-domain transport of bulk I/O through so-called I/O channels. Packets are formed using a data structure called I/O Record or *iorec*. An `iorec` consists of a header and a sequence of base pointer and length pairs similar to the `iovec` structure in Unix, with the header denoting how many pairs belong to the `iorec`. The base pointer points into a contiguous region of virtual address space called *Rbuf Data Area*, which is always backed by physical memory.

3 TCP Design and Implementation

In this section we briefly describe the design and implementation of the Nemesis TCP, including some implementation problems that occurred due to the buffer handling scheme.

3.1 Overview

The Nemesis TCP implementation is partitioned between a shared library (*InetMod*) executed in the application domain and a trusted domain called the *flow manager*. The latter is responsible for both the synchronization of global resources, such as TCP port numbers, and for various control tasks like installing and removing packet filters. The main task of the packet filters is to demultiplex incoming packets to the right application at the lowest possible level. The packet filter is part of the device driver domain (see Figure 3).

Our implementation is based on the TCP/IP implementation of BSD [15]. As is done in BSD Unix, applications call TCP functions indirectly via a socket interface. For native Nemesis networking we provide two socket abstractions, namely `Socket` and `ServerSocket`.

As other TCP user-space implementations, such as Thekkat *et al.* [13], our implementation uses threads, three for each connection. The *receiver thread* receives and processes incoming packets. Packets that should be delivered to

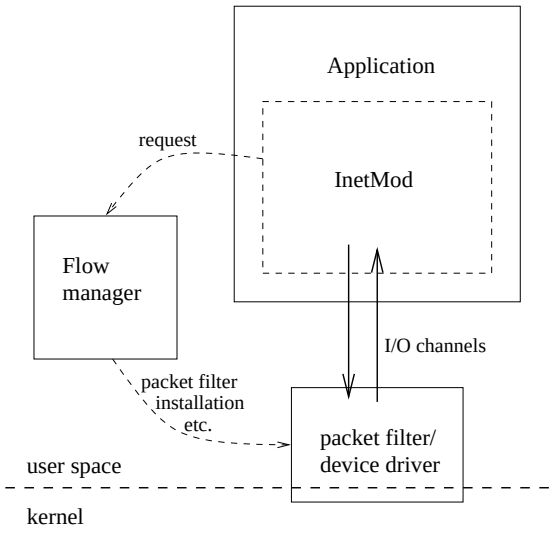

Figure 3: Architectural overview

the application are queued in a FIFO queue (*to_app*) containing pointers to `iorecs`. The *timer thread*'s task is to trigger delayed acknowledgements, to force retransmissions, window and keepalive probes and to drop connections when the peer does not respond. The timer thread also prevents that the connection stays in the `FIN_WAIT_2` state forever and handles the deletion of the control block in the `TIME_WAIT` state.

The third thread is the application's thread. When the application wants to transmit data its thread also places the data into the send-queue and executes `tcp_output()` as well as `ip_output()` and the link-level output function. When the application asks to receive packets, the first packet(s) from the *to_app* queue are returned. If there are no packets, the thread blocks if so specified.

3.2 Buffer Handling

The buffer handling scheme in Nemesis is different from kernel-space TCP implementations. Applications need to supply the device drivers with empty buffers to receive data in. They must also reclaim buffers from the device driver when the corresponding packet has been sent.

3.2.1 Supplying the Device Driver with Empty Receive Buffers

On the arrival of a packet on a network interface, the packet filter determines the receiving application and the device driver copies the packet into receive buffers provided by the application. Thus, applications must *prime* the device driver, i.e., applications have to supply the device driver with empty receive buffers.

In our TCP implementation, the receiver thread allocates receive buffers for header and payload of incoming packets and sends them to the device driver. The receive buffers are allocated from shared memory between the application and the device driver.

The application returns receive buffers to `InetMod` via an *extended* I/O channel. We call this I/O channel extended because it is an intra-domain extension of the I/O channel between the device driver and `InetMod` to the application program. `InetMod` then supplies the device driver with these receive buffers.

The number of empty receive buffers at the device driver limits the number of packets that a TCP connection can receive immediately. Thus, if the application or `InetMod` return the receive buffers slowly the device driver will run out of empty receive buffers and has to drop incoming packets. In our TCP implementation this situation is avoided by basing the calculation of the window advertisement on the number of empty receive buffers at the device driver and the number of empty slots in the *to_app* queue between `InetMod` and the receiving application.

3.2.2 Reclaiming Used Transmit Buffers

After a packet has been transmitted, Nemesis applications must reclaim the corresponding transmit buffers from the network interface. When transmitting data, applications only allocate the memory for the payload. The memory for the header is allocated by `InetMod`. Thus, when applications reclaim used transmit buffers, `InetMod` only returns the payload buffer. Of course, `InetMod` cannot return a transmit buffer to the application before the corresponding packet has been acknowledged by the peer, in case the packet gets lost and has to be retransmitted. When an application reclaims transmit buffers and there are no transmit buffers available, the application's thread is blocked if so specified.

`InetMod` itself needs to reclaim the transmit buffers from the device driver. To achieve good performance it is important to do that at the right time. A naive solution is that `tcp_output()` reclaims the transmit buffers from the device driver after having called `ip_output()`. But this leads to bad performance since the device driver cannot return the transmit buffers before the packet has been put on the wire. Thus, it is advantageous to postpone this task until the device driver can return the transmit buffers without blocking.

3.3 Transmit Scheduling

The transmit scheduling is not part of `InetMod` but part of the device driver domain. It is implemented by a thread, the *TX thread*. The scheduled entities are the I/O channels to the device driver. The reservations are made by applications for each of their I/O channels by a method call.

The TX thread runs in an endless loop. It calls the Atropos scheduler to determine the next I/O channel to be served. The scheduler chooses an I/O

channel based on the reservations. If this I/O channel has nothing to send, the scheduler searches for another suitable I/O channel. Thus, the call to the Atropos scheduler will return an I/O channel if there is an I/O channel that has pending packets and that has either a non-zero slice left in its current period or the extra flag set to `true`.

Thereafter the TX thread takes one packet out of the scheduled I/O channel, transmits the packet and charges the I/O channel for the transmission.

4 Experiments

The experiments in the first part of this section deal with scheduling of interface transmission bandwidth only. In these experiments the applications have sufficient CPU time to produce data and process outgoing and incoming packets. Under heavier CPU load, however, solely reserving transmit bandwidth is not enough. If an application needs to send at a particular bandwidth, a sufficient part of the CPU time has to be reserved as well. Otherwise the application might not be able to produce data and run the protocol stacks fast enough. This is addressed in the experiments in the next part of this section that deals with the reservation of transmit bandwidth and CPU time. In the final part of this section we present performance results that show that the Nemesis TCP achieves good throughput.

4.1 Reserving Transmit Bandwidth

In this section we present two experiments. In the first experiment, we show how the choice of the period and slice time impacts TCP performance. The second experiment includes several transmitters that strive for transmit bandwidth. In this experiment we can also see the effect of the extra flag on the throughput that connections receive.

We have implemented a TCP sender application that sends MTU-sized packets as fast as possible to the peer. We have varied the QoS parameters for the transmit network bandwidth. The transmitter was a Pentium 200 running Nemesis with a 10 Mb/s 3c509 Ethernet card. The transmission of one MTU-sized packet (1500 bytes including TCP/IP headers) on a 10 Mb/s Ethernet takes about 1.2 ms. Setting the slice time to, e.g., twice this time allows the application to send two packets in every period.

4.1.1 One Transmitter

In our first experiment, the Nemesis box sends to a SPARCstation running SunOS. Figure 4 shows a section of a TCP connection with the slice time set to 2.4 ms (which allows to send up to two MTU-sized packets in one period)[2] and

[2]When we set the slice time to e.g. 3 ms (which allows to send "2.5" MTU-sized packets in every period), we will get periods with two mixed with periods with three packets sent. The value 2.4 ms is thus chosen to get a very regular traffic pattern.

Figure 4: QoS TCP transmitting (tcpdump plot)

the period set to 50 ms. The extra time flag x is set to `false` in order to rate control TCP. The small lines with arrows represent the packets that are sent. The dotted line under the packets is the highest sequence number that has been acknowledged by the peer. The solid line above the packets is the window offered by the receiver. We can see that the sender transmits two packets, pauses almost 50 ms until the next two packets are sent and so on. Note that the transmission of packets is not triggered by the arrival of the acknowledgements but by the start of a new period.

From the period, slice time and the number of bytes to be sent per period we can compute the throughput for a connection. We have measured the throughput for a period of 200 ms and various slice times and compared the results with the theoretical values. For slice times up to 9.6 ms (maximum 8 MTU-sized packets per period) there are almost no discrepancies. However, for slice times that are larger than 9.6 ms, we do get very varying results. The explanation is that the receiver's window fills when the transmitter sends many packets during some periods and then the window offered by the peer varies depending on how fast the receiver acknowledges data. This prevents the transmitter from utilizing the guaranteed transmit bandwidth. However, the aim of rate-control is to set upper limits.

When transmitting over long distances or low-bandwidth connections, TCP flow control determines the overall traffic pattern. However, when the peer acknowledges several TCP segments, the Nemesis TCP does not reply by transmitting immediately a bulk of data until the receiver's window is filled again, but spreads out the packets depending on the chosen period and slice time.

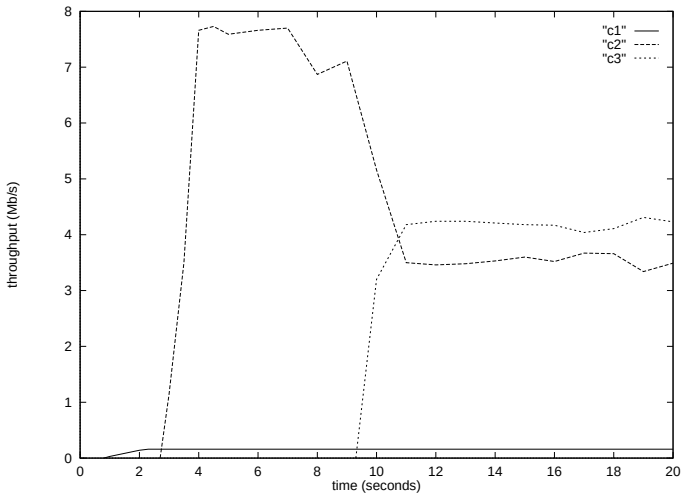

Figure 5: Time-throughput graph for 3 connections

4.1.2 Several Transmitters

In our second experiment, the Nemesis machine sends to a SPARCstation called *kay* and another SPARCstation (called *garuda*) running NetBSD. *Garuda* is equipped with a 10 Mb/s Ethernet card and sits on the same LAN as the Nemesis machine. The result is shown in Figure 5. First we start a TCP connection (*c1*) to *kay* with a period of 200 ms, a slice time of 3.6 ms and the extra flag set to `false`. This connection consistently receives the same bandwidth (about 0.17 Mb/s) which conforms to its guarantees.

About three seconds later, we start a second TCP transmitter (*c2*) to *garuda* with a period of 10 ms, a slice time of 1.2 ms and the extra time flag set to `true`. This means that *c2* wants to make use of the non-allocated part of the transmit bandwidth but also has some guaranteed transmit bandwidth. Since *c1* only consumes a small part of the available bandwidth, *c2* initially receives a throughput of about 7.5 Mb/s, with some variation due to how fast the receiver acknowledges the data. Some seconds later the third connection (*c3*), also to *garuda*, is started. It specifies a period of 10 ms, a slice time of 6 ms and the extra flag is set to `false`. The large slice time is chosen to reserve a large fraction of the available transmit bandwidth. As soon as *c3* has started, the non-allocated part of the transmit bandwidth decreases, and *c2* therefore receives much less transmit bandwidth (about 3.5 Mb/s).

Unfortunately, *c3* does not receive the guaranteed bandwidth of 5.6 Mb/s but only 4.2 Mb/s. The reason for this lower than expected throughput is that, quite often, the sender is not allowed to transmit since the receiver's window is closed. This happens because the acknowledgements that the receiver sends

are often delayed and arrive in bursts instead of arriving regularly (we saw a similar behavior when we ran Linux on both the sender and a receiver on the same LAN and transmitted as fast as possible). The delay is most likely caused by contention for the Ethernet media. We use a 10 Mb/s hub and have two fast connections. Under these conditions the shared Ethernet becomes a bottleneck. On the other hand, since *c2* has the extra flag set, the overall throughput does not decrease because *c2* makes use of the bandwidth that *c3* cannot utilize due to the closed window.

4.2 Reserving Transmit Bandwidth and CPU Time

In this section we present experiments that show the impact of CPU reservations on the achieved bandwidth of TCP applications. All connections are on a LAN with both transmitter and receiver running de4x5 100 Mb/s Ethernet cards. The transmitter is a Nemesis machine while the receiver runs Linux. In the following experiments we run a TCP application called *tcp_send* that transmits MTU-sized packets as fast as possible. We also run an application called *carnage*. This application has the property that it makes use of all its reservations. If it is guaranteed a certain fraction of the CPU time, it will make use of this fraction.

4.2.1 Impact of CPU Time on Achieved Throughput

In this experiment *tcp_send* reserves a large fraction of the network transmit bandwidth and has the extra time flag x set to `true`. Since *tcp_send* is the only transmitting application it will receive almost all the available transmit bandwidth. The CPU reservation request of *tcp_send* specifies a period of 10 ms, a slice time of 2 ms and the extra time flag is set to `true`. Thus *tcp_send* will receive a fraction of about 20 % of the CPU time and a fair share of the CPU time not used by other applications. *carnage* has specified a period of 20 ms, the extra flag is set to `false` and we vary the slice time to control the fraction of the CPU time that *carnage* receives. The more CPU time *carnage* receives the less CPU time is left for *tcp_send*. *tcp_send* will receive its slice time every period but the non-allocated part of the CPU time will decrease when *carnage's* slice time increases.

Figure 6 shows how the decreasing fraction of the CPU impacts the throughput that *tcp_send* achieves. Since *tcp_send* reserves 20 % of the CPU time, we set the maximum reservation for *carnage* to 72 %. We do not make a reservation of 80 % for *carnage* since there are also other domains running such as the flow manager and several device drivers. In particular the Ethernet driver consumes a non-negligible part of the CPU time in the experiments in this section.

When only *tcp_send* is running, it achieves a throughput of about 53 Mb/s. The figure shows that the throughput decreases proportionally to the fraction of CPU time used by *carnage*.

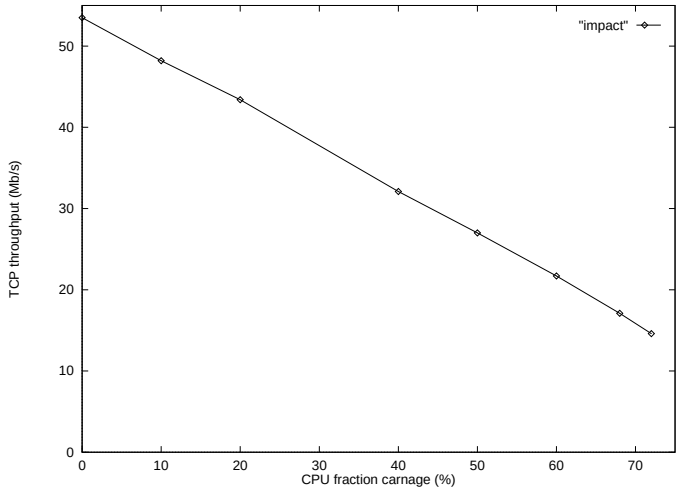

Figure 6: Impact of *carnage* application on TCP bandwidth

4.2.2 Reserving CPU Time to Achieve Throughput

In this experiment we assume that we have an application that wants to transmit data with a desired throughput. The goal is to determine the fraction of the CPU time an application needs in order to produce data and run the protocol stack fast enough to be able to transmit the desired amount of data. This can be important when you have reserved network bandwidth and you want to make sure that your application really achieves the bandwidth you pay for but you still want to have your text editor and other applications running.

The application *tcp_send* reserves the desired transmit bandwidth on the network interface. When the desired throughput is e.g. 10 Mb/s, *tcp_send* specifies a period of 1 ms, a slice time of 110 μs and sets the extra flag to `false`. The transmission of one MTU-sized packet (1500 bytes including TCP/IP headers) takes about 120 μs on a 100 Mb/s Ethernet. Since we need to send slightly less than one packet per millisecond to achieve a throughput of 10 Mb/s, setting the slice time to 110 μs seems to be reasonable. This has been confirmed by measurements.

In Figure 7 we show the fraction of the CPU time the application *tcp_send* needs to reserve to be able to transmit the desired amount of data. Since *tcp_send* does not do any data processing and mainly sends and retrieves buffers this can be seen as a minimum CPU fraction necessary to run the TCP protocol stack. However, the exact fraction of the CPU time needed depends on the packet length, the CPU, the network adapter and other system components.

Figure 7 shows that the CPU fraction needed increases linearly with the amount of data sent.

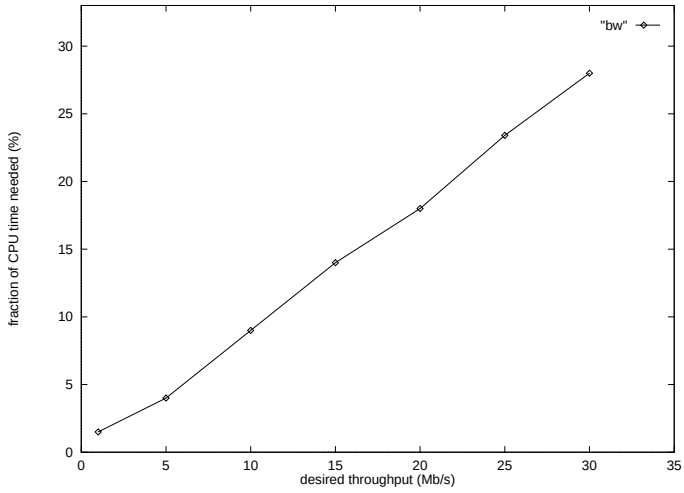

Figure 7: Fraction of CPU necessary to send particular amount of data

app.	aim	TX QoS	CPU QoS	Throughput
s1	10 Mb/s	1 ms, 110 μs, F	10 ms, 1 ms, F	10.1 Mb/s
s2	10 Mb/s	2 ms, 220 μs, F	20 ms, 2 ms, F	10.1 Mb/s
s3	5 Mb/s	2 ms, 110 μs, F	20 ms, 1 ms, F	5.1 Mb/s

Table 1: Experiment with three transmitting applications

4.2.3 Several Applications Reserving Transmit Bandwidth and CPU Time

In the next experiment we run three *tcp_send* applications (*s1*, *s2* and *s3*). *s1* and *s2* both want to transmit data at a speed of 10 Mb/s and *s3* at a speed of 5 Mb/s. The applications set the corresponding QoS-parameters (see Table 1) using the tuple (period, slice time, extra flag).

The fractions of the CPU share are the ones we determined in the previous experiment (see Figure 7), i.e., 10 % for *s1* and *s2* (*s1* and *s2* have different period and slice times but reserve the same fraction of CPU time and transmit bandwidth) and 5 % for *s3*. We also run a *carnage* application that has the extra flag set to `true` and thus consumes all the non-allocated CPU time.

Table 1 also shows that all applications receive their desired throughput.

4.3 Performance

Nemesis and its network architecture are primarily designed to support QoS. Raw throughput performance is secondary. We nevertheless think that the throughput we achieved with the TCP is reasonable. We have run *tcp_send*

between a Pentium 200 Nemesis machine and another Pentium 200 machine running Linux. Both machines are equipped with 100 Mb/s de4x5 Ethernet cards. We received a throughput of more than 58 Mb/s. This is lower than the 81 Mb/s we received when running Linux on both machines but we still think this is reasonable considering the primary goal with Nemesis. We believe that it is possible to optimize the Nemesis TCP a lot more. A problem in our architecture is that the transmitting application and the device driver run as different domains and are thus scheduled separately. The deployment of a *path* mechanism as in Scout [10] would probably increase throughput performance.

5 Related Work

Satyavolu *et al.* [12] have investigated in methods for controlling the rate of TCP applications. Our work deals with the end system, whereas their techniques can be applied in ATM edge devices and Internet routers. Crowcroft and Oechslin [4] do not rate control TCP traffic but limit the throughput of TCP connections by decreasing the size of the receive buffer in order to achieve weighted proportional fairness. Black *et al.* [2] have used Nemesis and the Atropos scheduler for UDP. In contrast to TCP, UDP does not have schemes for, e.g., congestion and flow control which can prevent a transmitter from sending although there are packets ready for transmission. Stride scheduling [14] has been used for both CPU and device driver scheduling but the authors do not report on any results of an integrated use.

Other architectures than Nemesis for providing QoS in an end system have been proposed. Yau and Lam [16] propose an end system architecture that supports networking with quality of service guarantees. They also use scheduling for both CPU and network interface access by deploying an adaptive rate-controlled (ARC) scheduler. The AQUA framework of Lakshman and Yavatkar [8] is used to manage CPU and network I/O resources in an integrated fashion. Their scheme is adaptive while we use reservations to provide QoS. Both Yau's and Lakshman's architectures have been implemented in Solaris whereas we use the Nemesis operating system which is designed from scratch to support quality of service. Gopalakrishnan and Parulkar [5] have implemented an efficient user-space protocol that supports QoS using real-time upcalls (RTU's). They achieve impressive performance running TCP/IP over ATM. Their mechanisms also allow to minimize delay or maximize throughput. The architecture the authors present in their paper deals with protocol processing only. In our scheme, the binding between an application and the underlying protocol processing unit is more explicit because protocol processing is performed by the application itself using shared libraries. Rajkumar *et al.* [9] aim for predictable communication protocol processing in Real-Time Mach by using *processor reserves*. As in Nemesis, library code in the application implements protocol processing. In contrast to Nemesis, their scheme relies on processor reservations only. Thus, it is possible that an application that is able to produce data faster can achieve

higher throughput than an application that has higher processor reserves but needs more time to produce the data that it transmits. Another difference is that processor reserves are made per thread and the real-time socket library comprises at least four threads.

The Rialto operating system [7] supports coexisting independent real-time and non-real-time programs by sharing the limited physical resources available to them. In one of their experiments, the authors study the influence of CPU reservations over video rendering fidelity. Due to the interdependency between frames there is no linear relationship between the CPU reservation and the frames not rendered.

A different way of providing quality of service in an end-system are feedback-based approaches such as G. Beaton's [1]. One of the goals of this approach is to avoid the complexity of other architectures like Nahrstedt's and Smith's QoS Broker [11].

6 Conclusions

We have designed and implemented a TCP for Nemesis, a vertically integrated operating system that can guarantee resources such as CPU time, disk I/O bandwidth and transmit network bandwidth to applications. Our experiments show that the Nemesis transmit scheduler rate-controls TCP traffic when applications choose not to utilize non-allocated bandwidth. Applications that want to make use of the non-allocated transmit bandwidth receive the guaranteed bandwidth plus a share of the non-allocated bandwidth. We show that the CPU time needed to run the protocol stack increases linearly with the amount of data sent for a given packet size. When networked applications reserve enough CPU time and transmit bandwidth they receive sufficient resources even when several applications strive for both CPU time and transmit bandwidth.

Thus, Nemesis allows us to run several applications, which require both CPU time and transmit bandwidth, concurrently, making sure each of them receives the guaranteed resources. Possible applications include concurrently running web servers with different performance requirements as well as other applications requiring reliable data transfer.

7 Future Work

After having optimized the Nemesis TCP we plan to investigate how Nemesis can be extended to perform as an end-host in a differentiated services platform. We believe that we might benefit from the Nemesis architecture here. An obvious advantage is that applications that have reserved enough resources can produce their data in time, independent of other applications.

8 Acknowledgements

Many people have contributed to Nemesis during the last years. Without their work we could not have written this paper. In particular Austin Donnelly's work has been of importance to us.

The authors would also like to thank Per Gunningberg for valuable comments on an earlier draft of this paper.

References

[1] Gordon Beaton. A feedback-based quality of service management scheme. In *Third International Workshop on High Performance Protocol Architectures (HIPPARCH '97)*, Uppsala, Sweden, June 12–13 1997.

[2] Richard Black, Paul Barham, Austin Donnelly, and Neil Stratford. Protocol implementation in a vertically structured operating system. In *IEEE 22nd Annual Conference on Computer Networks (LCN)*, pages 179–188, November 2–5 1997.

[3] R.J. Black. Explicit network scheduling. Technical Report 361, University of Cambridge Computer Laboratory, December 1994. Ph.D. Dissertation.

[4] J. Crowcroft and P. Oechslin. Differentiated end-to-end internet services using a weighted proportional fair sharing tcp. *ACM SIGCOMM Computer Communication Review*, 28(3):53–69, July 1998.

[5] R. Gopalakrishnan and G. M. Parulkar. Efficient user space protocol implementations with qos guarantees using real-time upcalls. *IEEE/ACM Transactions on Networking*, 6(4):374–388, August 1998.

[6] 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*, 14(7):1280–1297, September 1996.

[7] 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 *Seventh ACM SIGOPS European Workshop*, pages 249–256, Connemara, Ireland, September 1996.

[8] K. Lakshman, R. Yavatkar, and R. Finkel. Integrated cpu and network-i/o qos management in an endsystem. In *Int. Workshop on Quality of Service (IWQoS)*, pages 167–178, 1997.

[9] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol processing in real-time mach. In *Proceedings of the Real-Time Technology and Application Symposium*, pages 115–123, June 1996.

[10] 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)*, pages 153–167, October 1996.

[11] K. Nahrstedt and J. Smith. The qos broker. *IEEE Multimedia*, 2(1):53–67, 1995.

[12] Ramakrishna Satyavolu, Ketan Duvedi, and Shivkumar Kalyanaraman. Explicit rate control of tcp applications. ATM Forum Document, February 1999. ATM Forum Document Number: ATM_Forum/98-0152R1.

[13] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, October 1993.

[14] C.A. Waldspurger and W.E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical report, MIT Laboratory for Computer Science, 1995.

[15] G.R. Wright and W.R. Stevens. *TCP/IP Illustrated, Volume 2*. Addison-Wesley Publishing Company, 1995. ISBN 0-201-63354-X.

[16] David K.Y. Yau and Simon S. Lam. Migrating sockets - end system support for networking with quality of service guarantees. *IEEE/ACM Transactions on Networking*, 6(6):700–716, December 1998.

Paper B

Thiemo Voigt, Renu Tewari, Douglas Freimuth and Ashish Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. *Proceedings of Usenix Annual Technical Conference*, pages 189 – 202, Boston, MA, USA, June 2001.

Kernel Mechanisms for Service Differentiation in Overloaded Web Servers

Thiemo Voigt*
(thiemo@sics.se)
Swedish Institute of Computer Science

Renu Tewari
(tewarir@us.ibm.com)
IBM T.J. Watson Research Center

Douglas Freimuth
(dmfreim@us.ibm.com)
IBM T.J. Watson Research Center

Ashish Mehra
(asish@iscale.net)
iScale Networks

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 three kernel-based mechanisms that protect Web servers against overload by providing 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 based on the priorities of the incoming connections. Third, we present *HTTP header-based connection control* that uses application-level information such as URLs and cookies to set priorities and rate control policies.

We have implemented these mechanisms in AIX 5.0. Through numerous experiments we demonstrate their effectiveness 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.

1 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

* This work was partially funded by the national Swedish Real-time Systems Research Initiative (ARTES). This work was done when the author was visiting the IBM T.J. Watson Research Center.

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 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 [4]) to delay or control overload is not adequate as it does not distinguish between the individual QoS requirements. For example, it would be desirable that requests of non-preferred customer sites 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 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 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 is to be discarded it should be done as early as possible, before it has consumed a lot of system resources [24]. Since a web server's workload is generated by incoming network connections we place our control mechanisms in the network subsystem of the server OS at 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 that enable content-based admission control.

Our second principle was to introduce minimal changes to the core networking subsystem in commercial operating systems that typically implement a BSD-style stack. There have been prior research efforts that modify the architecture of the networking stack to enable stable overload behavior [18]. Other researchers have developed new operating system architectures to protect against overload and denial of service attacks [28]. 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 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 [1]. Such switches have some form of overload protection mechanisms that typically consists of dropping a new connection packet (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 [3]. 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 [9]. 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 [18] 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.

We have implemented these controls in the AIX 5.0 kernel as a loadable module using the framework of an existing QoS-architecture [14]. The existing QoS architecture on AIX supports policy-based outbound bandwidth management [23]. These techniques are easily portable to any OS running a BSD style network stack[3].

We present experimental results to demonstrate that these mechanisms effectively provide 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 Section 2 we give a brief overview on input packet processing. Our architecture and the kernel mechanisms are presented in Section 3. In Section 4 we present and discuss experimental results. We compare the performance of kernel based mechanisms and application level controls in Section 5. We describe related work in Section 6 and finally, the conclusions and future work in Section 7.

2 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. When the device interface receives a packet it triggers a hardware interrupt that is serviced by the corresponding device driver [31]. The device driver copies the received packet into an `mbuf` and de-multiplexes 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. The IP input routine dequeues the packet from the IP input queue 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.

[3] A port to Linux is underway.

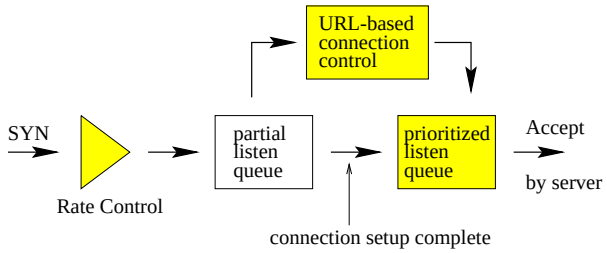

Figure 1: Proposed kernel mechanisms.

3 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 1 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. Each of these mechanisms can be activated at varying degrees of overload where the earliest and simplest control is triggered at the highest load level.

3.1 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 concurrent new requests. 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 2). 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.

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[4]. An alternate option, which we do not consider, is to send a TCP RST to reset the connection indicating an abort from the server. This approach, however, incurs unnecessary extra overhead. Secondly, some clients

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

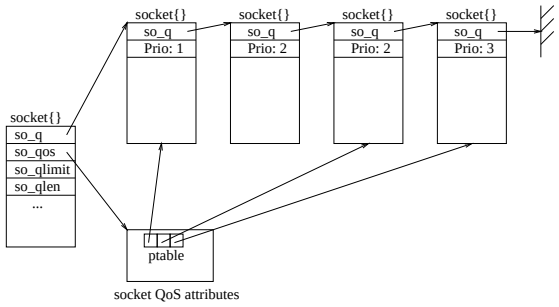

Figure 2: Implementation of the prioritized listen queue

send a new SYN immediately after a TCP RST is received instead of aborting the connection. Note that we drop non-compliant SYNs even *before* a socket is created for the new connection thereby investing only a small amount of overhead on requests that are dropped.

To provide service differentiation, connection requests are aggregated based on filters and each aggregate has a separate token bucket profile. Filtering based on client IP addresses is useful since a few domains account for a significant portion of a web server's requests [10]. The rate and burst values are enforced only when overload is detected and can be dynamically controlled by an adaptation agent, the details of which are beyond the scope of this paper.

3.2 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. The priorities are associated with filters (see Table 2) and connections are classified into different *priority classes*. When a TCP connection is established, 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.

Figure 2 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.

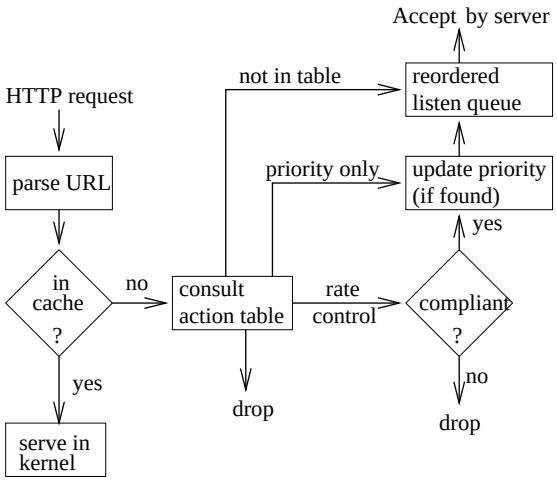

Figure 3: The HTTP header-based connection control mechanism.

3.3 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. 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, 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.

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) [26], an in-kernel web cache on AIX. For Linux, an in-kernel web engine called KHTTPD is available [6]. As opposed to the normal operation, where the sleeping process is woken up after a connection is established, AFPA responds to cached HTTP requests directly without waking up the server

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 >

Table 1: URL action table

(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)	*

Table 2: Example Network-level Policies

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.

The HTTP header-based connection control mechanism comes into play at this juncture, as illustrated in Figure 3, before the socket is moved out of the partial listen queue. The URL action table (Table 1) 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.

3.4 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 1. Table 2 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 filter rules can contain range of IP addresses, wildcards, etc.

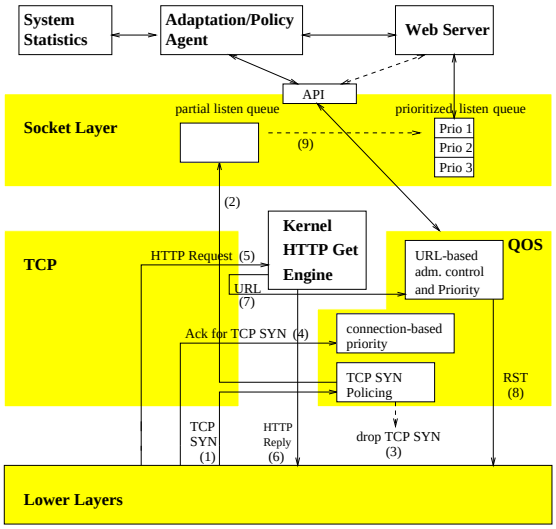

Figure 4: Enhanced protocol stack architecture.

3.5 Protocol Stack Architecture

We have developed architectural enhancements for Unix-based servers to provide these mechanisms. Figure 4 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 [23]; an adaptation/policy agent installs policies into the kernel via a special API. The policy agent interacts with the kernel via an enhanced socket interface by sending (receiving) messages to (from) special control sockets. The policies specify filters to select the traffic to be controlled, and actions to perform on the selected traffic. The figure shows the flow of an incoming request through the various control mechanisms.

3.6 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 4, 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.

All experiments were conducted on a testbed comprising an IBM HTTP Server running on a 375 MHz RS/6000 machine with 512 MB memory, several 550 MHz Pentium III clients running Linux, and one 166 MHz Pentium Pro

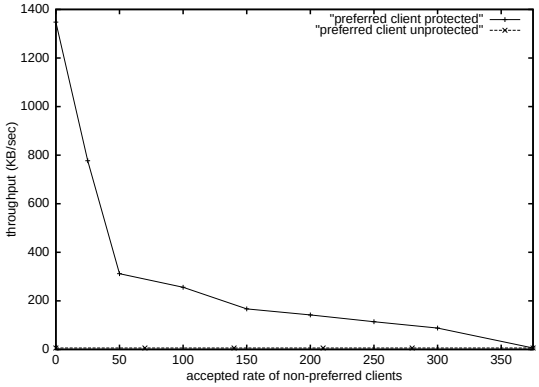

Figure 5: Throughput of the preferred e-tailer's client with and without TCP SYN policing. On the X-axis is the SYN policing rate of the non-preferred Webstone clients that are continuously generating requests. The Y-axis shows the corresponding throughput received by the e-tailer's client when there was no SYN control and when SYN control was enforced.

client running FreeBSD. The server and clients are connected via a 100 BaseT Ethernet switch. For client load generators we use Webstone 2.5 [7] and a slightly modified version of sclient [12]. 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 [2] 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.

4 Experimental Results

4.1 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 a large e-tailer'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 [10].

Without SYN policing, the e-tailer'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 e-tailer's client to increase. Figure 5 shows that

the throughput for e-tailer'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. For effective overload prevention the policing rate must be dynamically adapted to the resource consumption of accepted requests.

4.2 Impact of Burst Size

In the previous 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. The burst size also controls the responsiveness of rate control. 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 6 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 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/sec (585.6 KB/sec) to 47.7 conns/sec (752 KB/sec), while the throughput received by the preferred client decreases from about 140 conns/sec (1117 KB/sec) to 79 conns/sec.

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.

4.3 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. We demonstrate next that

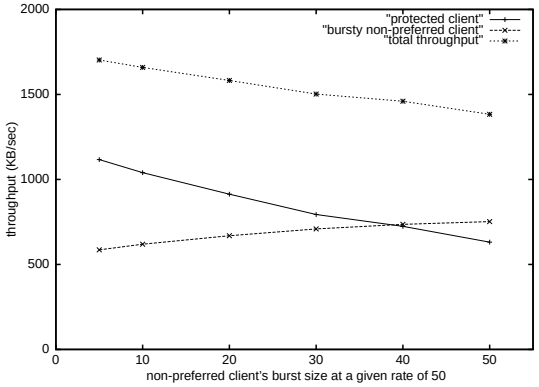

Figure 6: Impact of burst size on preferred client throughput. The burst size for policing non-preferred client is varied from 5 to 50 while the connection acceptance rate is fixed at 50 conn/sec. The plot shows the throughput achieved by the preferred and non-preferred clients along with the total throughput.

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 7 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 7 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 8 illustrates the effect on response times observed by clients of the three priority classes. It can be 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

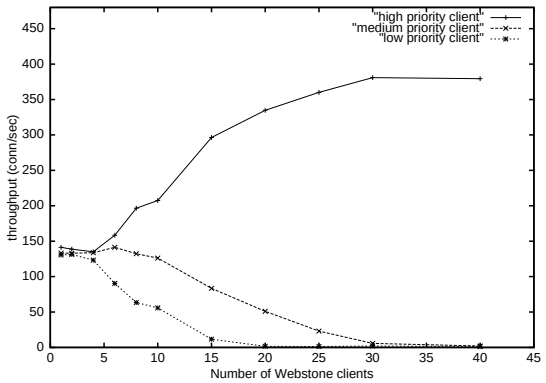

Figure 7: Throughput with the prioritized listen queue and 3 priority classes with 50 Apache processes. The number of clients in each class remains equal.

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.

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.

4.4 Combining 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 a full listen queue, which would in turn cause it to be aborted after its 3-way handshake is completed. 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 3 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 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

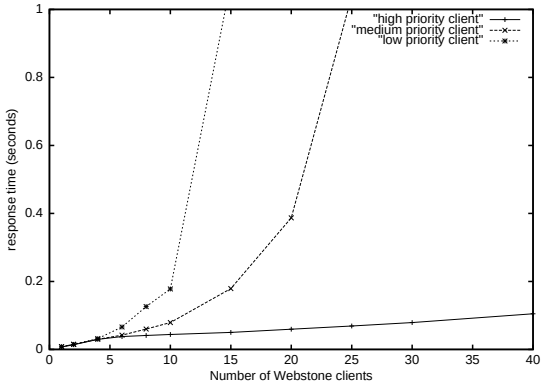

Figure 8: Response time with the prioritized listen queue and 3 priority classes with 50 Apache processes. The number of clients in each class remains equal.

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

Table 3: TCP SYN policing of a high-priority client to avoid starvation of other clients.

starved. Table 3 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.

4.5 HTTP Header-based Connection Control

In this section we illustrate the performance and effectiveness of admission control and service differentiation based on information in the HTTP headers i.e., URL name and type, cookie fields etc.

Rate control using URLs: In our experimental scenario the preferred client replaying the e-tailer'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 9 shows that without any protection, the preferred e-tailer'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 e-tailer's customer improves from 1 KB/sec to 650 KB/sec. In

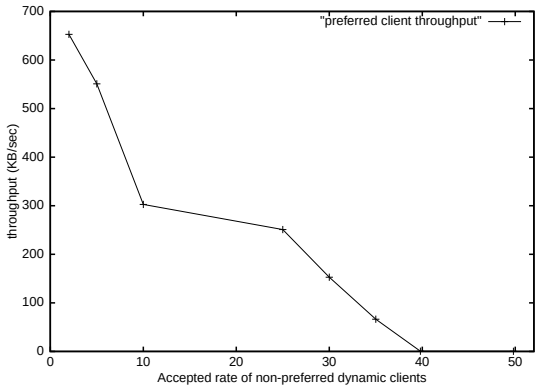

Figure 9: URL-based policing to protect preferred e-tailer's customers. The graph shows the resulting throughput of the preferred e-tailer's client as a specific high overhead CGI requests is limited to a given number of conn/sec

contrast to TCP SYN policing (Figure 5), URL rate control targets a specific URL causing overload instead of a client pool.

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 10 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 7). 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 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 4 shows that starvation can be avoided by rate-limiting the high-priority URL requests.

4.5.1 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

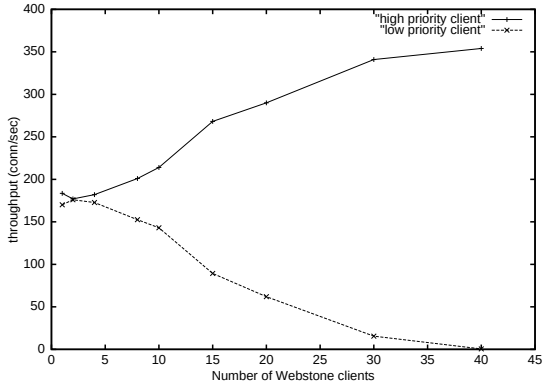

Figure 10: Throughput with 2 URL-based priorities and 50 Apache server processes. The number of clients in each class is equal

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

Table 4: URL-based policing of a high-priority client to avoid starvation of other clients.

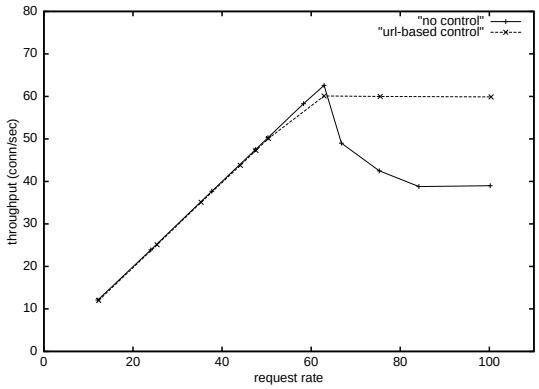

Figure 11: Overload protection from high overhead requests using URL-based connection control. The graph shows the throughput of web server with no controls servicing CPU intensive CGI requests and the corresponding throughput when the CGI requests are limited to 60 reqs/sec.

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 11 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 up to a critical point of about 63 connections/sec. For any further increase in the request rate the throughput falls exponentially 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 11 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

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 5: Performance of AFPA and matching a URL to a rule for a 8 KB file with different URL lengths.

subsequent SYN request from that same client for a specified time interval. The time interval selected is the timeout value used for a lost SYN.

4.5.2 Discussion

The HTTP header-based rate control relies on sending TCP RST to terminate non-preferred connections as and when necessary. In a more user-friendly implementation we could directly return an HTTP error message (e.g., server busy) back to the client and close the connection.

Our current implementation of URL-based control handles only HTTP/1.0 connections. We are currently exploring different mechanisms for HTTP/1.1 with keep-alive connections to limit the number and types of requests that can be serviced on the same 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.

4.6 Overhead of the Kernel Mechanisms

We quantify the overhead of matching URLs in the kernel for varying URL lengths. Table 5 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 serve any object from the cache. When caching is enabled Webstone received a throughput of over 800 connections per second on a cache hit.

Table 6 summarizes the additional overhead of the implemented kernel mechanisms. The overhead of compliance check and filter matching for TCP SYN

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

Table 6: Overhead of kernel mechanisms

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.

5 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. User level controls although limited in their capabilities, have easy access to application layer information. However, kernel mechanisms are more scalable and provide much better performance. In general, placing mechanisms in the kernel is beneficial if it leads to considerable performance gains and increases the robustness of the server without relying on the application layer to prevent overload.

To enable a fair comparison we have extended the Apache 1.3.12 server with additional modules [29] that police requests based on the client IP address and requested URL. The implemented rate control schemes use exactly the same

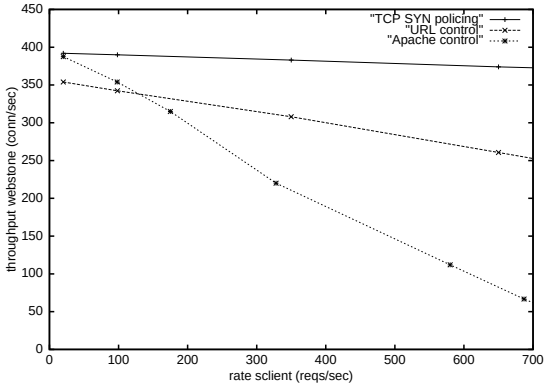

Figure 12: Throughput of kernel-based TCP SYN policing, kernel based URL rate control and Apache module based connection rate control. The throughput achieved by Webstone clients is measured against an increasing request load generated by sclient. The sclient requests are rate controlled to 10.0 req/sec with a burst of 2.

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.

The experimental setup consists of a Webstone traffic generator with 100 clients requesting a file of size 8 KB along with an sclient program requesting a file of size 16 KB. 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.

Figure 12 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 4.6. 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 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 corresponding results for response times are shown in Figure 13.

The experiment demonstrates that the kernel mechanisms are more efficient and scalable than the user space mechanisms. There are two main rea-

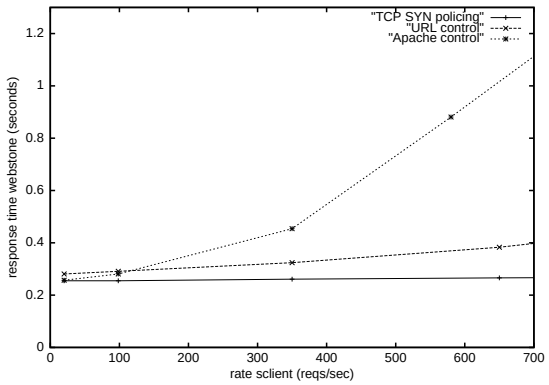

Figure 13: Response times using kernel-based TCP SYN policing, kernel based URL rate control and Apache module based connection rate control. The response time achieved by Webstone clients is measured against an increasing request load generated by sclient. The sclient requests are rate controlled to 10 req/sec with a burst of 2.

sons for the higher efficiency and scalability: First, non-compliant connection requests are discarded earlier reducing the queuing time of the compliant requests, in particular less CPU is consumed 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.

6 Related Work

Several research efforts have focused on admission control and service differentiation in web servers [8], [16], [21], [22], [9] and [15]. Almeida *et al.* [9] 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., the web server, determines request priorities, our mechanisms reside in the kernel and can be applied without context-switching to user level. *WebQoS* [15] 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.* [16] present an enhanced web server that provides session-based admission control to ensure that longer sessions are completed. Crovella *et al.* [17] 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.* [27] have presented virtual services, a new operating sys-

tem 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. In [19] Reumann *et al.* have described an adaptive mechanism to setup rate controls for overload protection. The receiver livelock study [24] showed that network interrupt handling could cause server livelocks and should be taken into consideration when designing process scheduling mechanisms. Banga and Druschel's [13] *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. In the same paper they also describe a multi listen socket approach for priorities in which a filter splits a single listen queue into multiple queues from which connections are accepted separately and accounted to different principals. Our approach is similar, however, connections are accepted from the same single listen queue but inserted in the queue based on priority. Kanodia *et al.* [21] present a simulation study of queuing-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.* [11] describe a scalable request distribution architecture for clusters and also present resource management techniques for clusters.

Scout [25], Rialto [20] 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. Our focus, however, was not to build a new operating system or networking architecture but to introduce simple controls in the existing architecture of commercial operating systems that could be just as effective.

7 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 and prevents overload by enforcing a maximum acceptance rate of non-preferred clients. The prioritized listen queue provides 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 priorities 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. We compared the kernel mechanisms to similar application layer controls added in the Apache server and 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. 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 4) 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 and monitors the interaction between various control options on the overall performance during overload.

Our current implementation does not address security issues of fake IP addresses and client identities. We plan to integrate a variety of overload prevention policies with traditional firewall rules to provide an integrated solution.

References

[1] Alteon web systems. http://www.alteonwebsystems.com.

[2] apache. http://www.apache.org.

[3] Cisco arrowpoint web network services. http://www.arrowpoint.com.

[4] Cisco TCP intercept. http://www.cisco.com.

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

[6] Khttpd - linux http accelerator. http://www.fenrus.demon.nl/.

[7] webstone. http://www.mindcraft.com.

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

[9] 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*, March 1999.

[10] Martin F. Arlitt and Carey l. Williamson. Web server workload characterization: The search for invariants. In *Proc. of ACM Sigmetrics*, April 1996.

[11] 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.

[12] G. Banga and P. Druschel. Measuring the capacity of a web server. In *Proc. of USITS*, December 1997.

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

[14] 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*, 16(3):397–413, April 1998.

[15] Nina Bhatti and Rich Friedrich. Web server support for tiered services. *IEEE Network*, September 1999.

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

[17] M. E. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *Proc. of USITS*, October 1999.

[18] P. Druschel and G. Banga. Lazy receiver processing (LRP): a network subsystem architecture for server systems. In *Proc. of OSDI*, pages 91–105, October 1996.

[19] H. Jamjoom and J. Reumann. Qguard:protecting internet servers from overload. Technical report, University of Michigan, CSE-TR-427-00, 2000.

[20] 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*, pages 249–256, September 1996.

[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*, March 2000.

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

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

[25] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *Proc. of OSDI*, pages 153–167, October 1996.

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

[27] 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.

[28] O. Spatscheck and L. Peterson. Defending against denial of service attacks in scout. In *Proc. of OSDI*, February 1999.

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

[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*, August 1999.

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

Paper C

Thiemo Voigt and Per Gunningberg. Kernel-based Control of Persistent Web Server Connections. *ACM Performance Evaluation Review*, 29(2):20–25, September 2001.

Kernel-based Control of Persistent Web Server Connections

Thiemo Voigt
(thiemo@sics.se)
Swedish Institute of Computer Science

Per Gunningberg
(Per.Gunningberg@it.uu.se)
Uppsala University

Abstract

Several overload admission control architectures have been developed to protect web servers from overload. Some of these architectures base their admission decision on information found in the HTTP header. In this context, persistent connections represent a challenging problem since the HTTP header of the first request does not reveal any information about the resource consumption of the requests that might follow on the same connection. In this paper, we present an architecture that prevents uncontrollable server overload caused by persistent connections. We evaluate our approach by various experiments.

1 Introduction

Web servers have to be protected from overload since overload can lead to high response times, low throughput and even loss of service. To protect servers from overload several admission control architectures have been developed [6, 5, 9]. Many of these architectures can reject or accept connection requests based on information provided in the HTTP header [5, 9]. In this context, persistent connections represent a challenging problem. Persistent connections allow HTTP clients to send several requests on the same TCP connection to reduce client latency and server overhead [8]. When the same TCP connection is used for several requests, the HTTP header of the first request does not reveal any information about the resource consumption of the following requests on the same connection.

In [9] we presented kernel-based mechanisms that provide admission control and service differentiation based on filter rules associated with connection and application level information. We have shown that kernel-based mechanisms are more efficient and scalable than controls implemented in user space. In this paper, we extend this architecture to provide kernel-based control of persistent connections. The goal is to allow clients to complete the sessions (by session, we mean a sequence of individual requests on the same TCP connection) that are regarded as important as well as sessions initiated by clients that are regarded as important even when the server becomes overloaded.

We have decided to judge the importance of persistent connections based on the cookies found in the HTTP header. The major advantage of using cookies

is that all the information needed to determine the current importance of a connection is found in the HTTP and lower layer protocol headers. Cookie-based connection control is also flexible: When the application is changed, only the filter rules associated with the affected cookies need to be updated. Furthermore, cookies do not only contain information about the current session, but also longer lasting information such as customer identification.

Based around the notion of cookie-based connection control, we extend the architecture presented in [9]. We present experiments that show that our approach can prevent server overload and provide service differentiation under high load.

The contribution made in this paper is a kernel-based architecture that prevents overload in web servers caused by persistent connections. The architecture is able to provide service differentiation during overload. We demonstrate the effectiveness of an approach that denotes the relative importance of persistent connections using cookies.

2 Related Work

Several research efforts have focused on admission control and service differentiation in web servers, e.g. [6, 5, 9]. Most of them ignore the challenges of persistent connections.

Cherkasova and Phaal [6] present an admission control scheme whose aim is to allow sessions to run to completion even under overload. In their approach they deny all access when the server load exceeds a predefined threshold. We do not reject all clients, but abort connections that are considered less important.

WebQoS [5] is another architecture that aims at providing web server QoS. The architecture contains a session management entity. The authors denote cookies as one way to identify sessions, but do not discuss the problem of unknown resource demands of persistent connections.

Luo and Yang [7] augment server clusters with mechanisms that enable important sessions to be smoothly migrated and recovered on another node in the case of node failures. Their work is similar in the respect that they also identify important connections. Aron *et al.* [3] study mechanisms for supporting persistent connections in cluster-based web servers that employ content-based request distribution. Their aim is to improve throughput while we aim at preventing server overload.

3 Problem Description

To prevent server overload, the amount of work entering the web server must be controlled. For example, HP's *WebQoS* [5] triggers rejection of requests when queue lengths exceed certain thresholds. Our kernel-based architecture [9] uses token bucket based policers to limit the acceptance rate of new requests. In

both architectures, persistent connections are a challenging problem since the resource consumption of the requests is unknown at the time the admission control decision has to be made, i.e. when the web server receives the first request on a persistent connection.

For example, a user visiting the home page of a company might either leave the company's web pages after visiting a single page or might initiate a long session. Obviously, a long session demands much more resources. Hence, accepting the first request does not reveal much about the amount of work that enters the system. Setting maximum queue lengths or acceptance rates is thus a trade-off. If one is too conservative, i.e. maximum queue sizes are short or acceptance rates are low, potential customers might be rejected unnecessarily resulting in loss of revenue. If one is too optimistic, the server may become overloaded with long response times and low throughput as a possible consequence. Even well-engineered adaptive overload prevention schemes suffer from this problem. For example the onset of overload may not be predicted with sufficient accuracy due to workload fluctuations. This defines the goal of our work: To avoid uncontrollable overload while maximizing access.

Once overload has set in, there is an urgent need to restore server performance. In general, the extent of performance degradation due to overload is unpredictable and can persist for a long time. When a high percentage of the active connections are persistent connections, the situation can become worse since their future requests and the resource demands are unknown. Hence, in an overload situation, it might be inevitable to abort some persistent connections.

When aborting persistent connections, one should not abort connections blindly. Instead, it is desirable to preserve connections regarded as important, and abort connections that are less important. For example, a session can be regarded as important when the client has placed some items in a shopping bag. As the shopping bag example suggests, the importance of a persistent connection changes over its lifetime. It is thus important to design mechanisms that keep track of the current importance of persistent connections and that allow the determination of the importance of the connections easily when this information is needed.

Approach

We propose to use cookies to code the importance of a session. When the server sends the HTTP reply back to a client, a cookie denoting the importance of the session is sent along. In the next request, the cookie is automatically inserted into the HTTP header. The admission control mechanism then uses the cookies to determine the importance of the connection. This approach is attractive since:

- All the information needed to determine the current importance of a connection is embedded in the cookie.
- Cookies are an established, widely used technique.

- It is easy to remove or update cookies and the associated filter rules.
- Cookies can also contain long-lasting information, for example a cookie can identify a preferred customer.
- No changes to the web server are necessary.

There are other possible approaches. Many applications encode the state of sessions in the chosen URL, e.g. URLs with a common prefix denote common state. Our approach can be modified to use URL-based control in the same way as we use cookies here. This is only an implementation issue. A completely different approach is to add an additional field in the socket data structure. This field denotes the importance of the connection. On behalf of the web application, the web server updates this field when the importance of the connection changes. During server overload, control of persistent connection is based on this field. However, this approach requires modifications to both the web server, the kernel and the application.

4 System Architecture

The Original Architecture

The architecture described in this paper is an extension to the architecture presented in [9]. This kernel-based architecture protects web servers against overload by controlling the amount and rate of work entering the system. For this work, the relevant mechanisms are located in the network stack of the operating system: *TCP SYN policing* is located at the start of protocol stack processing of the first TCP SYN packet of a new connection and limits acceptance of a new SYN packet based on compliance with a token bucket based policer. This enables service differentiation based on information in the TCP and IP headers of an incoming connection (i.e, the source and destination address and port number). *HTTP header-based connection control* is activated when the HTTP header is received (which could be after multiple data packets) and enables admission control based on application-layer information contained in the header such as cookie, URL name or type (e.g., CGI request). Using this mechanism a more informed control is possible. For example a majority of the load is caused by a few CGI requests. This suggests that we want to be able to specify lower access rates for CGI requests than other requests causing less load. This is done using filter rules.

Filter rules specify 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 rule is a four-tuple consisting of local IP address, local port, remote IP address, and remote port combined with the action to be taken: Rate-control with a specified token bucket based policer, drop or assign a priority. Application-level filter rules specify three types of actions for each URL or set of URLs (identified by e.g. a common prefix): drop,

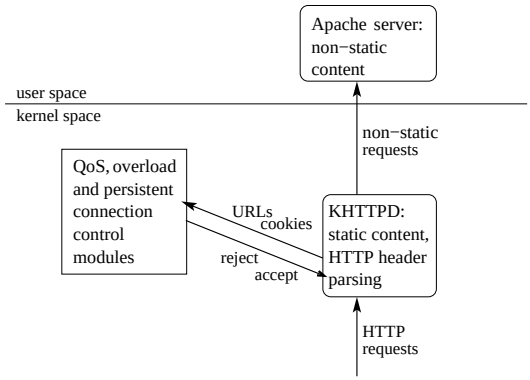

Figure 1: Architecture

rate-control or assign a priority. Application-level filter rules are associated with network-level filter rules. In order to provide service differentiation, each network-level filter rule may have a different set of application-level filter rules. This way, we can give clients with different IP addresses different access rates for the CGI requests that cause a majority of the server load.

The kernel mechanisms have proven to be much more efficient and scalable than application level controls implemented in the web server.

The Extended Architecture

We have implemented the architecture presented in Figure 1 in the Linux 2.4 operating system. The implementation consists of an unmodified Apache web server, modules for QoS, overload and persistent connection control as well as a khttpd in-kernel HTTP Get engine [1].

For HTTP header parsing we use a slightly modified version of khttpd (we have modified khttpd to interact with the QoS module). Khttpd is an in-kernel web server that efficiently serves static requests. Khttpd needs to be used in combination with a user-space web server because khttpd cannot serve non-static content such as dynamically built web pages. When receiving such a request, khttpd does a hand-over of the connection to the user-space web server.

After parsing an HTTP header, khttpd calls the QoS module passing information found in the HTTP header such as URL and cookies. The QoS module searches for matching filter rules and based on the information found returns the admission decision to khttpd. Depending on the admission decision khttpd resets the connection or processes the request.

We have extended the original QoS module to also do cookie-based persistent connection control. The QoS module regularly measures the CPU utilization and enforces cookie-based overload control only when the CPU utilization is

higher than a predefined threshold. When cookie-based control is active, all requests on persistent connections that do not carry a cookie matching one of the rules are aborted by sending a TCP RST to the client. A nicer option is to send a "server temporarily unavailable" back to the client. Or in other words, when cookie-based control is active only persistent connections that carry cookies denoting that the connection is important are allowed to proceed. By aborting the connection in the kernel, we save the context-switch to user space. Note that in this model cookies have two functions. HTTP header-based connection control uses cookies (in the same way as URLs or request types) to decide on admission of the *initial* request. Cookie-based connection control uses cookie to decide on abortion or continuation of persistent connections. In our prototype the same filter rules are used for both functions.

As in the original architecture, application-level rules are tied to network-level rules, i.e. we can have different application level-rules for clients with different IP addresses. This allows us to handle the same cookies belonging to clients with different IP addresses differently. Note that cookie-based connection control itself can be used to provide service differentiation by assigning, for example, important customers cookies that allow them to always complete their sessions, even when they appear with different IP addresses.

Cookie-based overload control can also be implemented in a middleware layer or in the web server itself, e.g., as an Apache module.

5 Experiments

Our testbed consists of a server and some traffic generators connected via a 100 Mb/sec Ethernet switch. The server machine is a 300 MHz Pentium II with 128 MBytes of memory running Linux 2.4. The server is an unmodified Apache web server, v.1.3.9., with the default configuration, i.e. a maximum of 150 worker processes.

For client load generation we use the sclient traffic generator [4]. Sclient is able to generate client request rates that exceed the capacity of the web server. We have modified sclient in a way that all requests on persistent connections carry a keep-alive option to signal to the web server that there are more requests to follow on the same connection. The last request does not carry a keep-alive option to signal the web server that it can tear down the connection after sending the response. On the persistent connection, sclient sends request number *N* after the response for request number *N-1* is received. Aron *et al.* have extended the sclient program in a similar way to handle persistent connections [3].

In our experiments each sclient program represents a group of clients requesting the same set of files. For simplicity, we will use the term sclient to denote the group of clients represented by one sclient program. The experimental workload consists of static and dynamic requests. The dynamic files are minor modifications of standard Webstone [2] CGI files.

5.1 Persistent Connections Can Cause Server Overload

In the first experiment we illustrate how persistent connections can cause a server to become overloaded. As discussed previously, the difficulty with persistent connections is that the first request does not reveal anything about the resource consumption of the requests that might follow on a persistent connection. Hence, the admission control schemes may admit too many connection requests which can cause server overload.

We run two sclient programs. The first sclient program (called sclient_1) requests a static file of size 12 KB. The second sclient program (sclient_p) requests the same file. Thus, the admission control will treat both files in the same way. However, sclient_p's first request is followed by an additional three requests on the same persistent connection. The second and third requests consist of static files with a size of 29 KB and 16 KB respectively followed by a CGI file of size 8 KB. In this experiment, sclient_1 represents a client population visiting the index page of a company only, while sclient_p may represent the client population that browses several pages and then puts an order on one of the company's products.

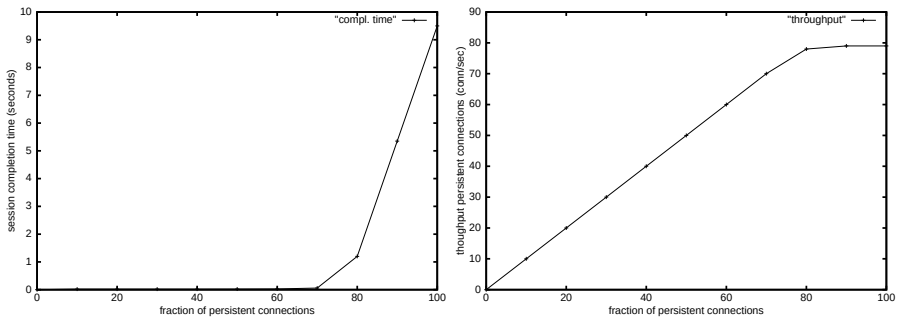

Figure 2: Persistent connections can cause overload

The sum of the request rates of the two sclients is 100 reqs/sec. In the experiment we gradually increase the share of the persistent requests. Figure 2 shows the session completion time for sclient_p and the achieved throughput. The percentage of persistent connections does not impact sclient_p's response time until 70% of the requests are for persistent connections, i.e. sclient_p has a request rate of 70 reqs/sec while sclient_1 has a request rate of 30 reqs/sec. As the percentage of persistent connections increases beyond 70%, the session completion times increase fast. The throughput increases with the request rate until it reaches a plateau of about 80 conn/sec. Also, sclient_1's response time increases from some milliseconds when we have a low fraction of persistent connections to more than one second, when 90% of the requests are for persistent connections.

Measuring the CPU utilization shows that if more than 75% of the requests are from sclient_p, the CPU idle time shrinks to zero. Hence, the high session

cookie-onset	throughput	completion time
-	40.3 conn/sec	9.9 sec
98% CPU-util.	37.8 conn/sec	2.1 sec
90% CPU-util.	36.4 conn/sec	1.85 sec

Table 1: Overload protection using cookie-based control

completion times are caused by server overload due to high CPU utilization.

5.2 Overload Control

The aim of this experiment is to demonstrate that cookie-based connection control can prevent uncontrollable server overload caused by the resource consumption of persistent connections.

In this experiment we have again two sclient programs. Sclient_2 requests a session consisting of a sequence of six objects, first five static requests for files of size 12 KB, 10 KB, 16 KB, 24 KB and 29 KB respectively followed by a dynamic request of size 16 KB. The increasing length of the requested static files emulates more demanding CGI requests. As discussed above, using khttpd we can only make the last request dynamic. The last four requests are "protected" by cookies. Thus sclient_2's sessions are never aborted by the server after the second request has been completed. An example where this is useful is an e-commerce site where one does not want to abort clients that are about to purchase something, but clients that have not yet added something to their shopping basket might be aborted when an unexpected overload situation occurs. The request rate of sclient_2 is 50 reqs/sec which results in a CPU utilization of about 80-85 %, i.e. sclient_2's requests do not overload the server.

Sclient_b requests a static file of size 12 KB (the same request as sclient_2's first request) and then a large dynamic request of size 32 KB. In order to emulate short-time overloads, sclient_b runs for nine seconds (this is an arbitrary choice), then is silent for eight seconds, runs again for nine seconds etc. Sclient_b's request rate is 50 reqs/sec. This is enough to overload the server when both sclient_2 and sclient_b are active at the same time.

We expect that cookie-based connection control is able to prevent server overload. When sclient_b is active, the load on the server will increase beyond the threshold, and cookie-based connection control will abort connections and that way restore server performance.

The results in Table 1 show that cookie-based control can reduce server overload. Sclient_2's average session completion time is more than eight seconds lower when using cookie-based controls. The throughput decreases slightly since many sessions are aborted when both sclients are active at the same time.

From our logs files, we can see that as expected cookie-based control sets in when sclient_b becomes active. The threshold determines how long cookie-based control is on while sclient_b is active. Table 1 also shows that a lower

Throughput (conn/sec)		
throughput	sclient$_{pr}$	sclient$_2$
overall	25.3 conn/sec	19.4 conn/sec
during bursts	21.1 conn/sec	9.4 conn/sec
during non-bursts	28.0 conn/sec	24.8 conn/sec

Table 2: Service differentiation using cookie-based control

threshold means that cookie-based control is enabled more often than with a higher threshold. In this experiment we saw 10% more aborted connections with the lower threshold than with the higher threshold. As expected, fewer aborted connections lead to higher throughput, but slightly higher session completion time.

5.3 Service Differentiation

In this experiment we demonstrate how cookie-based connection control can be used to provide service differentiation during server overload. When we need to abort persistent connections, we want to be able to treat clients differently e.g. in order to not violate service agreements.

We have three sclient programs, sclient$_b$, sclient$_2$ and sclient$_{pr}$. The latter two request the same sequence of six requests as sclient$_2$ in the previous experiment. Both sclients have a request rate of about 26 reqs/second. Sclient$_{pr}$ represents a preferred client whose HTTP headers contain cookies that prevent connection abortion during the whole session, while sclient$_2$'s HTTP headers prevent connection abortion from the third until the final request. As in the previous experiment, sclient$_b$ causes short-term server overload.

As in the previous experiment, we expect that when sclient$_b$ is active and the load on the server increases beyond the threshold, cookie-based connection control will abort connections to restore server performance. Since we prevent sclient$_{pr}$'s connections from being aborted, sclient$_{pr}$ should receive a higher throughput than sclient$_2$.

Table 2 shows sclient$_{pr}$'s and sclient$_2$'s throughput. In this scenario the overall throughput received by sclient$_{pr}$ is about 25.3 conn/sec while sclient$_2$ achieves a throughput of about 19.4 conn/sec. During the periods where sclient$_b$ is active (we call these periods bursts), sclient$_2$ receives a very low throughput, while sclient$_{pr}$'s throughput is higher but lower than its overall throughput. When sclient$_b$ is not active, sclient$_{pr}$'s throughput is higher than its overall throughput. The reason for this is that due to the high load during bursts, many sessions initiated during a burst are completed after the duration of the burst causing higher "non-burst" throughput.

In this experiment, sclient$_2$ has a slightly lower average session completion time than sclient$_{pr}$. The reason for this is that during bursts a large part of sclient$_2$'s sessions are aborted while sclient$_{pr}$'s sessions are still active. Due

Throughput (conn/sec)		
no controls	first cookie matches	third cookie matches
55.3 conn/sec	55.1 conn/sec	54.6 conn/sec

Table 3: Performance of cookie-based control

number of filter rules	Cookie size (chars)	Cost (μsec)
one	12	2.37
three	12	3.62
one	50	3.89
three	50	4.88
one	100	5.62
three	100	6.61

Table 4: Overhead of cookie matching

to the higher server load sessions take longer to complete during bursts which increases sclient$_{pr}$'s average session completion time.

5.4 Overhead

In this experiment, we quantify the overhead of cookie-based connection control. First, we measure the throughput when the same sequence of six objects as in Section 5.2 and in Section 5.3 is requested. We run until 1000 sessions have been completed. The request rate is 100 reqs/sec. We measure the throughput when no controls are applied, when the cookie in the client's request's HTTP header is matching the first filter rule and when the third filter rule is matching. The results presented in Table 3 demonstrate that the throughput decreases only marginally when cookie-based connection control is active.

We have also measured the overhead of the cookie mechanism for cookies consisting of 12, 50 or 100 characters. From the results shown in Table 4 we conclude that the overhead is not very high in spite of using standard string comparison (*strcmp*) for cookie matching without optimizations. On the other hand, the increasing cost for larger cookies suggests that a production system should deploy better matching techniques.

6 Limitations and Future Work

The presented architecture is implemented in Linux and makes use of the khttpd kernel HTTP GET engine to parse HTTP headers in the operating system kernel. While khttpd is efficient it can only handle static files and not dynamic files which are dominant in applications that have an inherent need for longer sessions such as electronic commerce applications. To be able to perform HTTP header parsing in the kernel, only the final request in a session can be a dynamic

request in our experiments. This drawback could be overcome by implementing our own HTTP kernel parser or using a more advanced HTTP GET engine.

Under overload, sessions regarded as unimportant are aborted by the kernel, i.e. without the knowledge of the application. Since this is the same as clients disappearing during sessions, we assume that applications can handle such a situation. Some users have disabled cookie usage in their web browsers. During overload, persistent connections of those users are aborted.

In our experiments, we send request number *N* directly after receiving the response for request number *N-1*, i.e. our emulated clients have a think time of zero seconds. Real users, however, have longer think times and thus persistent connections can time out at the server. This means, that we must ensure that the new connection request is not rejected by the initial admission control mechanisms. Since TCP SYN policing is based on e.g. the client's IP address, it is possible to dynamically insert filter rules that do not reject the initial TCP SYN. The cookies in the HTTP header can be used to avoid rejection by the HTTP header-based connection control. Consider a client that is about to pay but due to long think time his persistent connection has timed out at the server. The cookie(s) in the HTTP header now ensures, that the new connection request is admitted by the HTTP header-based connection control mechanism.

Our simple strategy of turning cookie-based connection control on and off could potentially lead to oscillations. We are currently studying when they appear and how to dampen them. In overload situations, our current implementation relies on the existence of "unimportant" persistent connections that can be aborted. If all active persistent connections carry cookies that prevent connections from being aborted, our system will not abort any of them and fail to recover.

7 Conclusions

We have presented an architecture that protects web servers from overload caused by persistent connections. The resource consumption of the requests on a persistent connection is unknown at the time the admission control decision has to be made. In an overload situation caused by resource demands of persistent connections our architecture aborts persistent connections regarded as less important. This approach can prevent uncontrollable server overload, provide service differentiation and has low overhead.

Acknowledgements

This work builds on the architecture described in [9] that has been developed at IBM TJ Watson together with Renu Tewari, Ashish Mehra and Douglas Freimuth. Thanks to Carl Erickson for helpful comments on content and grammar. This work is partially funded by the national Swedish Real-Time Systems

research initiative ARTES.

References

[1] Khttpd. http://www.fenrus.demon.nl/.

[2] Webstone. http://www.mindcraft.com.

[3] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient support for p-http in cluster-based web servers. In *Usenix Annual Technical Conference*, June 1999.

[4] G. Banga and P. Druschel. Measuring the capacity of a web server. In *Proc. of USITS*, December 1997.

[5] Nina Bhatti and Rich Friedrich. Web server support for tiered services. *IEEE Network*, September 1999.

[6] L. Cherkasova and P. Phaal. Session based admission control: a mechanism for improving the performance of an overloaded web server. Technical report, HP, 1999.

[7] M. Y. Luo and C. S. Yang. Constructing zero-loss web services. In *Proc. of INFOCOMM*, April 2001.

[8] J. C. Mogul. The case for persistent-connection http. In *Proc. of SIGCOMM*, 1995.

[9] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. Usenix Annual Technical Conference 2001.

Paper D

Thiemo Voigt and Per Gunningberg. Handling Multiple Bottlenecks in Web Servers Using Adaptive Inbound Controls. *Seventh International Workshop on Protocols for High-Speed Networks (PfHSN 2002)*, Berlin, Germany, April 2002.[5]

[5]The version in this thesis has been extended by adding the sections on service differentiation.

Handling Multiple Bottlenecks in Web Servers Using Adaptive Inbound Controls

Thiemo Voigt
(thiemo@sics.se)
Swedish Institute of Computer Science

Per Gunningberg
(Per.Gunningberg@it.uu.se)
Uppsala University

Abstract

Web servers become overloaded when one or several server resources are overutilized. In this paper we present an adaptive architecture that prevents resource overutilization in web servers by performing admission control based on application-level information found in HTTP headers and knowledge about resource consumption of requests. In addition, we use an efficient early discard mechanism that consumes only a small amount of resources when rejecting requests. This mechanism first comes into play when the request rate is very high in order to avoid making uninformed request rejections that might abort ongoing sessions. We present our dual admission control architecture and various experiments that show that it can sustain high throughput and low response times even during high load.

1 Introduction

Web servers need to be protected from overload because web server overload can lead to high response times, low throughput and potentially loss of service. Therefore, there is a need for efficient admission control to maintain high throughput and low response time during periods of peak server load. Servers become overloaded when one or several critical resources are overutilized and become the bottleneck of the server system. The main server resources are the network interface, CPU and disk [13]. Any of these may become the server's bottleneck, depending on the kind of workload the server is experiencing [17]. For example, the majority of CPU load is caused by a few CGI requests [5]. The network interface typically becomes overutilized when the server concurrently transmits several large files.

In this paper, we report on an adaptive admission control architecture that utilizes the information found in the HTTP header of incoming requests. Combining this information with knowledge about the resource consumption we can avoid resource overutilization and server overload. We call our approach resource-based admission control.

The current version of our admission control architecture is targeted at overloaded single node web servers or the back-end servers in a web server cluster. The load on web servers can be reduced by distributed web server architectures

that distribute client requests among multiple geographically dispersed server nodes in a user-transparent way [12]. Another approach is to redirect requests to web caches. For example, in the distributed cache system Cachemesh [32] each cache server becomes the designated cache for several web sites. Requests for objects not in the cache are forwarded to the responsible cache. However, not all web data is cacheable, in particular dynamic and personalized data. Also, load balancing mechanisms on the front-ends of web server clusters can help to avoid overload situations on the back-end servers. However, sophisticated load-balancing cannot replace proper overload control [22]. Another existing solution to alleviate the load on web servers based on a multi-process architecture such as Apache is to limit the maximum number of server processes. However, this approach limits the number of requests that the server can handle concurrently and can lead to performance degradation.

The main contribution of this work is an adaptive admission control architecture that handles multiple bottlenecks in server systems. Furthermore, we show how we can use TCP SYN policing and HTTP header-based control in a combined way to perform efficient and yet informed web server admission control.

Resource-based admission control uses a kernel-based mechanism for overload protection and service differentiation called *HTTP header-based connection control* [31]. HTTP header-based connection control allows us to perform admission control based on application-level information such as URL, sets of URLs (identified by, for example, a common prefix), type of request (static or dynamic) and cookies. HTTP header-based control uses token bucket policers for admission control. HTTP header-based connection control is used in conjunction with filter rules that specify application-level attributes and the parameters for the associated control mechanism, i.e. the rate and bucket size of the policer.

Our idea to avoid overutilization of server resources is the following: we collect all objects that when requested are the main consumers of the same server resource into one directory. Thus, we have one directory for each critical resource. Each of these directories is then moved into a separate part of the web server's directory tree. We associate a filter rule with each of these directories. Hence, we can use HTTP header-based control to protect each of the critical resources from becoming overutilized. For example, CPU-intensive requests reside in the `cgi-bin` directory and a filter rule specifying the application-level information (URL prefix `/cgi-bin`) is associated with the content at this location.

When the request rate reaches above a certain level, resource-based admission control cannot prevent overload, for example during flash crowds. When such situations arise, we use *TCP SYN policing* [31]. This mechanism is efficient in terms of resource consumption of rejected requests because it provides "early discard". The admission of connection requests is based on network-level attributes such as IP addresses and a token bucket policer.

This paper also presents novel mechanisms that dynamically set the rate of the token bucket policers based on the utilization of the critical resources. Since

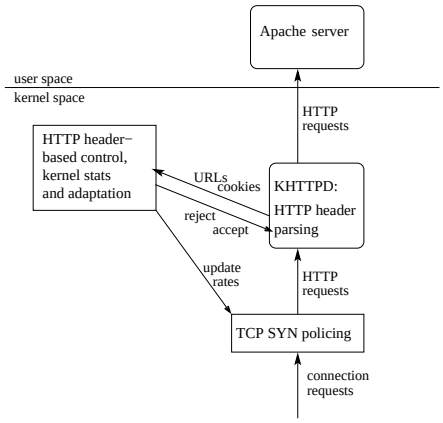

Figure 1: Kernel-based architecture

the web server workload frequently changes, for example when the popularity of documents or services changes, assigning static rates that work under these changing conditions may either lead to underutilization of the system when the rates are too low or there is a risk for overload when the rates are too high. The adaptation of the rates is done using feedback control loops. Techniques from control theory have been used successfully in server systems before [2, 24, 18, 11].

In our adaptive architecture accepted requests can be processed quickly since server resources are not overutilized. Therefore, the major goal of service differentiation in our architecture is to provide high throughput to premium customers. This is done by splitting the token buckets used for admission control into logical partitions [26]. Each logical partition corresponds to one service class with larger partitions for more important service classes.

We have implemented this admission control architecture in the Linux OS and using an unmodified Apache web server, we conducted experiments in a controlled network. Our experiments show that overload protection, service differentiation and adaptation of the rates works as expected. Our results show higher throughput and much lower response times during overload compared to a standard Apache on Linux configuration.

The rest of the paper is structured as follows. Section 2 presents the system architecture including the controllers. Section 3 presents experiments that evaluate various aspects of our system. Section 4 discusses architectural extensions. Before concluding, we present related work in Section 5.

2 Architecture

2.1 Basic Architecture

Our basic architecture deploys mechanisms for overload protection and service differentiation in web servers that have been presented earlier [31]. These mechanisms control the amount and rate of work entering the system (see Figure 1).

TCP SYN policing limits acceptance of new SYN packets based on compliance with a token bucket policer. Token buckets have a token rate, which denotes the average number of requests accepted per second and a bucket size which denotes the maximum number of requests accepted at one time. TCP SYN policing enables service differentiation based on information in the TCP and IP headers of the incoming connection request (i.e, the source and destination addresses and port numbers).

HTTP header-based connection control is activated when the HTTP header is received. Using this mechanism a more informed control is possible which provides the ability to, for example, specify lower access rates for CGI requests than other requests that are less CPU-intensive. This is done using filter rules, e.g. checking URL, name and type.

The architecture consists of an unmodified Apache server, the TCP SYN policer, the in-kernel HTTP GET engine khttpd [29] and a control module that implements HTTP header-based control, monitors critical resources and adapts the acceptance rates. Khttpd is used for header parsing only. After parsing the request header it passes the URLs and cookies to the control module that performs HTTP header-based control. If the request is rejected, khttpd resets the connection. In our current implementation, this is done by sending a TCP RST back to the client. If the request is accepted it is added to the Apache web server's listen queue. TCP SYN policing drops non-compliant TCP SYNs. This implies that the client will time-out waiting for the SYN ACK and retry with an exponentially increasing time-out value. For a more detailed discussion, see [31].

Both mechanisms are located in the kernel which avoids the context switch to user space for rejected requests. The kernel mechanisms have proven to be much more efficient and scalable than the same mechanisms implemented in the web server [31].

2.2 The Dual Admission Control Architecture

Our dual admission control architecture is depicted in Figure 2. In the right part of the figure we see the web server and some of its critical resources. With each of these resources, a filter rule and a token bucket policer is associated to avoid overutilization of the resource, i.e. we use the HTTP header-based connection control mechanism. For example, a filter rule `/cgi-bin` and an associated token bucket policer restrict the acceptance of CPU-intensive requests. On receipt of a request, the HTTP header is parsed and matched against the filter rules. If

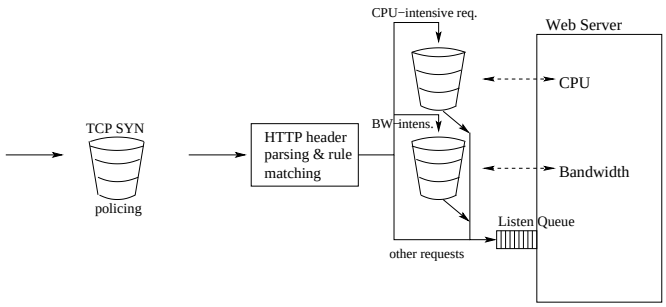

Figure 2: Admission control architecture

there is a match, the corresponding token bucket is checked for compliance. Compliant requests are inserted into the listen queue. We call this part of our admission control architecture resource-based admission control.

For each of the critical resources, we use a feedback control loop to adapt the token rate at which we accept requests in order to avoid overutilization of the request. We do not adapt the bucket size but assume it to be fixed. Note, that the choice of the policer's bucket size is a trade-off [31]. When we have a large bucket size, malicious clients can send a burst to put high load on the machine, whereas when the bucket size is small, clients must come at regular intervals to make full use of the token rate. Furthermore, when the bucket size is smaller than the number of parallel connections in a HTTP 1.0 browser, a client might not be able to retrieve all the embedded objects in a HTML-page since requests for these objects usually come in a burst after the client has received the initial HTML page.

Note that we do not perform resource-based admission control on all requests. Requests such as those for small static files do not put significant load on one resource. However, if requested at a sufficiently high rate, these requests can still cause server overload. Hence, admission control for these requests is needed. We could insert a default rule and use another token bucket for these requests. Instead, we have decided to use TCP SYN policing and police all incoming requests. The main reason for this is TCP SYN policing's early discard capability. Also for TCP SYN policing, we adapt the token rate and keep the bucket size fixed.

One of our design goals for the adaptation mechanisms is to keep TCP SYN policing inactive while resource-based admission control can protect resources from being overutilized. When performing resource-based admission control, the whole HTTP header has been received and can be checked not only for URLs but also for other application-level information, such as cookies. This gives us the ability to identify ongoing sessions or premium customers. TCP SYN policing's admission control is based on network-level information only and cannot assess such application-level information. Note that this does not

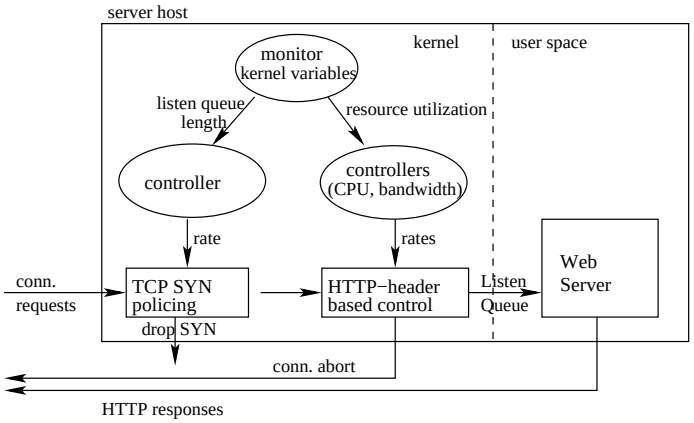

Figure 3: The control architecture

mean that TCP SYN policing is not worthwhile [31]. Firstly, TCP SYN policing can provide service differentiation based on network-level attributes. Secondly, TCP SYN policing is more efficient than HTTP header-based control in the sense that less resources are consumed when a request is discarded.

Our architecture uses several control loops to adapt the rate of the token bucket policers: One for each critical resource and one to adapt the SYN policing rate. A consequence of this approach is that the interaction between the different control loops might cause oscillations. Fortunately, requests to large static files do not consume much CPU while CPU-intensive requests usually do not consume much network bandwidth. Thus, we believe that the control loops for these resources will not experience any significant interaction effects. Adapting the TCP SYN policing rate affects the number of CPU-intensive and bandwidth-intensive requests, which may cause interactions between the control loops. To avoid this effect, we increase rates quite conservatively. Furthermore, in none of our experiments we have seen an indication that such an interaction might actually occur. One of the reasons for this is that TCP SYN policing becomes active when the acceptance rate of CPU-intensive requests is very low, i.e. most of the CPU-intensive requests are discarded.

2.3 The Control Architecture

Our control architecture is depicted in Figure 3. The monitor's task is to monitor the utilization of each critical resource and pass it to the controller. The controllers adapt the rates for the admission control mechanisms. We use one controller for the CPU utilization and one for the bandwidth on the outgoing network interface. We call the former CPU controller and the latter bandwidth controller. Both high CPU utilization and dropped packets on the networking interface can lead to long delays and low throughput. Other resources that could

be controlled are disk I/O bandwidth and memory. In addition, we use a third controller that is not responsible for a specific resource but performs admission control on all requests, including those that are not associated with a specific resource. The latter controller, called SYN controller, controls the rate of the TCP SYN policer.

Since different resources have different properties, we cannot use the same controller for each resource. The simplest resource to control is the CPU. The CPU utilization changes directly with the rate of CPU-intensive requests. This makes it possible to use a proportional (P) controller. The equation that computes the new rates is called the *control law*. For our P-controller the control law is:

$$rate_{\text{cgi}}(t+1) = rate_{\text{cgi}}(t) + K_{\text{P\_CPU}} * e(t) \tag{1}$$

where $e(t) = CPU\_util_{\text{ref}} - CPU\_util(t)$, i.e. the difference between the *reference* or desired CPU utilization and the current, measured CPU utilization. $rate_{\text{cgi}}(t)$ is the acceptance rate for CGI-scripts at time t. $K_{\text{P\_CPU}}$ is called the *proportional gain*. It determines how fast the system will adapt to changes in the workload. For higher $K_{\text{P\_CPU}}$ the adaptation is faster but the system is less stable and may experience oscillations [19].

The other two controllers in our architecture base their control laws on the length of queues: The SYN controller on the length of the listen queue and the bandwidth controller on the length of the queue to the network interface. The significant aspect here is actually the change in the queue length. This derivative reacts faster than a proportional factor. This fast reaction is more crucial for these controllers since the delay between the acceptance decision and the actual occurrence of high resource utilization is higher than when controlling CPU utilization. For example, the delay between accepting too many large requests and overflow of the queue to the network interface is non-negligible. One reason for this is that it takes several round-trip times until the TCP congestion window is sufficiently large to contribute to overflow of the queue to the network interface.

We decided therefore to use a proportional derivative (PD) controller for these two controllers. The derivative is approximated by the difference between the current queue length and the previous one, divided by the number of samples. The control law for our PD-controllers is:

$$rate(t+1) = rate(t) + K_{\text{P\_Q}} * e(t) + K_{\text{D\_Q}} * (queue\_len(t) - queue\_len(t-1)) \tag{2}$$

where $e(t) = queue\_len_{\text{ref}} - queue\_len(t)$. The division is embedded in $K_{\text{D\_Q}}$. $K_{\text{D\_Q}}$ is the *derivative gain*.

We imposed some conditions on the equations above. Naive application of Equation 1 results in an increase in the acceptance rate when the measured value is below the reference value, i.e. the resource could be utilized more. However,

it is not meaningful to increase the acceptance rate, when the filter rule has less hits than the specified token rate. For example, if we allow 50 CGI requests/sec, the current CPU utilization is 60%, the reference value for CPU utilization is 90% and the server has received 30 CGI requests during the last second, it does not make sense to increase the rate to more than 50 CGI requests/sec. On the contrary, if we increase the rate in such a situation, we would end up with a very high acceptance rate after a period of low server load. Hence, when the measured CPU utilization is lower than the reference value, we have decided to update the acceptance rate only when the number of hits was at least 90% of the acceptance rate during the previous sampling period.

Thus, Equation 1 rewrites as:

$$rate_{\text{cgi}}(t+1) = \begin{cases} rate_{\text{cgi}}(t) & \text{if } (\#\text{ hits }) < 0.9 * rate_{\text{cgi}}(t) \wedge \\ & CPU\_util(t) < CPU\_util_{\text{ref}} \\ rate_{\text{cgi}}(t) + K_{\text{P\_CPU}} * e(t) & \text{otherwise} \end{cases}$$

We impose a similar condition on the SYN controller and the bandwidth controller. Both adapt the rates based the queue lengths. The listen queue and the queue to the network interface usually have a length of zero. This means, that the length of the queue is below the reference value. For the same reason as in the discussion above, if the queue length is below the reference value, we update the acceptance rate only when the length of the queue has changed.

When performing resource-based admission control, we do not police all requests, even if all requests consume resources at least some CPU. Thus, if the CPU utilization is already high, i.e. it is above the reference value, we do not want to increase the amount of work that enters the system since this might cause server overload. Thus, we increase the TCP SYN policing rate only when the CPU utilization is below the reference value.

An important decision is the choice of the sampling rate. For ease of implementation, we started with a sampling rate of one second. To obtain the current CPU utilization, we can use the so-called *jiffies* that the Linux kernel provides. Jiffies measure the time the CPU has spent in user, kernel and idle mode. Since 100 jiffies are equivalent to one second, it is trivial to compute the CPU utilization during the last second. Since even slow web servers can process several hundred requests per second, a sampling rate of one second might be considered long. The question is if we do not miss important events such as the listen queue filling up. This would be the case given that the requests entering the server during one second was not limited. However, TCP SYN policing limits the number of requests entering the system. This bounds the system state changes between sampling points and allows us to use a sampling rate of one second. This is an acceptable solution since the experiment in Section 3.3 shows that the control mechanisms still adapt quickly when we expose the server to sudden high load.

The number of packets queued on an outgoing interface can change quite rapidly. When the queue is full, packets have to be dropped. The TCP connec-

tions that experience drops back off and thus less packets are inserted into the queue. We have observed that the queue length has changed from maximum length to zero and back to maximum length within 20 milliseconds. To avoid incorporating such an effect when computing new rates, we sample the queue length to that interface more frequently and compute an average every second. Using this average, the controller updates the rates every second.

2.4 Service Differentiation

One of the design goals of our architecture is to provide better service to premium clients, for example, clients having a service contract with the web operator. In an adaptive architecture the length of the listen queue is almost always zero. This means that traditional mechanisms to provide service differentiation in web servers, such as having different queues for each service class [10] or reordering of the listen queue [31], will have little effect, at least when not used in combination with other schemes. As the experiments in the next section show, using our adaptive architecture the average response time of accepted requests is low. In the experiments without service differentiation, the average response time is almost always below 150 milliseconds, even when the offered load is very high. Even the 90-percentile of the response time is not high, mostly below 200 milliseconds. This means, that the important task of service differentiation is to make sure that the acceptance rate of premium requests is higher than the acceptance rate of standard requests.

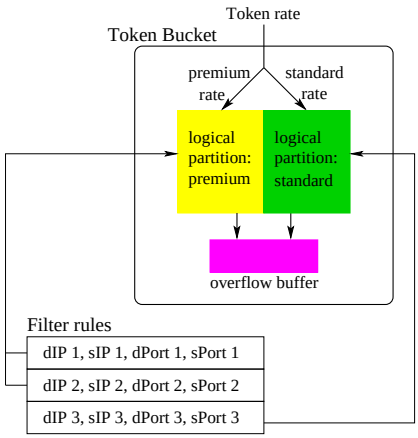

Figure 4: Token bucket with logical partitions

To achieve higher throughput for more important service classes, we divide token buckets into logical partitions or logical buckets [26], one for each service class. Generated tokens are divided between the logical buckets with a specified proportion. When a logical bucket is full, the newly generated tokens for this

partition are put into the overflow buffer. Requests can consume these extra tokens when there is no token in their logical partition. This enables requests belonging to one service class to use unused tokens from the other service class.

Figure 4 shows an example with two service classes which are distinguished based on network-level information. TCP SYN policing would use such a scheme. For example, we could choose to reserve two thirds of the tokens for the "premium partition", i.e., for premium clients, and one third for the "standard partition". This should lead to twice the acceptance rate for premium requests compared to standard requests when the offered load is sufficiently high.

Requests are matched against filter rules to determine the logical bucket corresponding to their service class. For example, the first two filter rules in Figure 4 could contain the IP addresses of premium clients while the last filter rule could be a default rule matching all other requests.

The same scheme can also be used to provide service differentiation for resource-based admission control. For example, the token bucket associated with CPU-intensive requests could be partitioned in the same way as described above.

Note that also the size of the logical buckets plays an important role. As we will show in Section 3.5, larger bucket sizes lead to higher throughput when the request arrival distribution is bursty. Therefore, it is meaningful to use a larger bucket size for the premium partition than for the standard partition.

3 Experiments

Our testbed consists of a server and two traffic generators connected via a 100 Mb/sec Ethernet switch. The server machine is a 600 MHz Athlon with 128 MBytes of memory running Linux 2.4. The traffic generators run on a 450 MHz Athlon and a 200 MHz Pentium Pro. The server is an unmodified Apache web server, v.1.3.9., with the default configuration, i.e. a maximum of 150 worker processes. We have extended the length of the web server's listen queue from 128 to 1024. Banga and Druschel [8] argue that a short listen queue can limit the throughput of web servers. A longer queue will not have an effect on the major results.

Parameter Settings

We have used the following values for the control algorithms: The reference values for the queues are set to 100 for the listen queue and 35 for the queue to the network interface which has a length of 100. These values are chosen arbitrarily but they can be chosen lower without significant impact on the stability since the queue lengths are mostly zero. Repeating for example the experiment in Section 3.3 with reference values larger than 20 for the listen queue length leads to the same results. The reference value for CPU utilization is 90%. We chose this value since it allows us to be quite close to the maximum utilization

while higher values would more often lead to 100% CPU utilization during one sampling period.

The proportional gain for the CPU-controller is set to 1/5. We have obtained this value by experimentation. In our experiments we saw that for gains larger than 1/4, the CGI acceptance rate oscillates between high and low values, while it is stable for smaller values. The proportional gain for the SYN and bandwidth controller is set to 1/16, the derivative gain to 1/4. These values were also obtained by experimentation. When both gains are larger than 1/2, the system is not stable, i.e. the change in both the length of the listen queue and the rates is high when the server experiences high load. When the derivative gain is higher than the proportional gain, the system reacts fast to changes in the queue lengths and the queues rarely grow large when starting to grow. We consider these values to be specific to the server machine we are using[6]. However, we expect them to hold for all kinds of web workloads for this server since we use a realistic workload as described in the next section. The bucket size of the token bucket used for TCP SYN policing is set to 20 unless explicitly mentioned. The token buckets for HTTP header-based controls have a bucket size of five.

3.1 Workload

For client load generation we use the sclient traffic generator [8]. Sclient is able to generate client request rates that exceed the capacity of the web server. This is done by aborting requests that do not establish a connection to the server in a specified amount of time. Sclient in its unmodified version requests a single file. For most of our experiments we have modified sclient to request files according to a workload that is derived from the surge traffic generator [9]:

1. The size of the files stored on the server follows a heavy tailed distribution.
2. The request size distribution is heavy tailed.
3. The distribution of popularity of files follows Zipf's Law. Zipf's Law states that if the files are ordered from most popular to least popular, then the number of references to a file tends to be inversely proportional to its rank.

Determining the total number of requests for each file on the server is also done using surge. We separated the files in two directories on the server. The files larger than 50 KBytes were put into one directory (`/islarge`), the smaller files into another directory. Harchol-Balter et al. [20] divide static files into priority classes according to size and assign files larger than 50 KBytes into the group of largest files. We made 20% of the requests for small files dynamic. The dynamic files used in our experiments are minor modifications of standard Webstone [27] CGI files and return a file containing randomly generated characters of the specified size. The fraction of dynamic requests varies from site to site with some

[6]The values also worked well on another machine we tested, but we do not assume this is the general case.

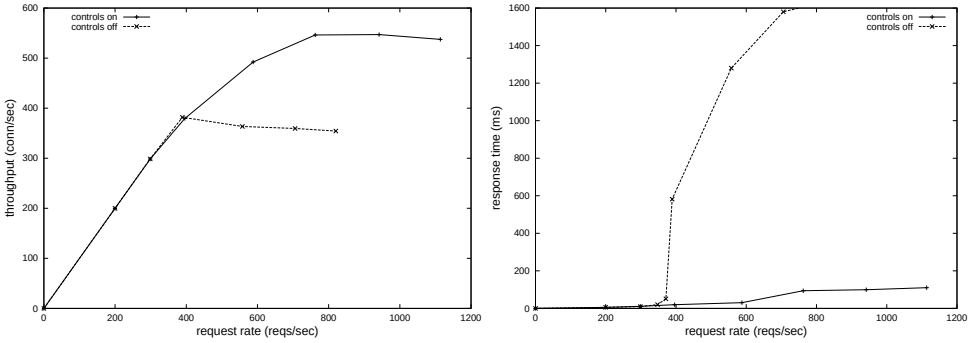

Figure 5: Comparison standard system (no controls) and system with adaptive overload control

sites experiencing more than 25% dynamic requests [25, 14]. For the acceptance rate of both CGI-scripts and large files, minimum rates can be specified. The reason for this is that the processing of CGI-scripts or large files should not be completely prevented even under heavy load. This minimum rate is set to 10 reqs/sec in our experiments.

In the next sections we report on the following experiments: In the first experiment we show that the combination of resource-based admission control and TCP SYN policing works and adapts the rates as expected. In this experiment the CPU is the major bottleneck. In the following experiment, we expose the system to a sudden high load and study the behaviour of the adaptation mechanisms under such circumstances. In the experiment in Section 3.4, we make the bandwidth on the interface a bottleneck and show how resource-based admission control can prevent high response times and low throughput. In the next experiment, we show that the adaptation mechanisms can cope with more bursty request arrival distributions. The last experiment demonstrates that the proposed scheme for service differentiation works as expected.

3.2 CPU Time and Listen Queue Length

In this experiment, we use two controllers: the CPU controller that adapts the acceptance rate of CGI-scripts and the SYN controller. As mentioned earlier, the reference for adapting the rate of CGI-scripts is the CPU utilization and the reference for the TCP SYN policing rate is the listen queue length. About 20% of the requests are for dynamic files (CGI-scripts). In the experiment, we vary the request rate across runs. The goals of the experiment are: (i) show that the control algorithms and in particular resource-based admission control prevent overload and sustain high throughput and low response time even during high load; (ii) show that TCP SYN policing becomes active when resource-based admission control alone cannot prevent server overload; (iii) show that the system achieves high throughput and low response times over a broad

fraction dynamic reqs. (%)	req rate for onset of SYN policing (reqs/sec)
20	675
10	640
0	610

Table 1: Request rate for which SYN policing becomes active for different fractions of dynamic requests in the workload

range of possible request rates.

For low rates, we expect that no requests should be discarded. When the request rate increases, we expect that the CPU becomes overutilized mostly due to the CPU-intensive CGI-scripts. Hence, for some medium request rates, policing of CGI-scripts is sufficient and TCP SYN policing will not be active. However, when the offered load increases beyond a certain level, the processing capacity of the server will not be able keep up with the request rate even when discarding most of the CPU-intensive requests. At that point, the listen queue will build up and thus TCP SYN policing will become active.

Figure 5 compares the throughput and response times for different request rates. When the request rate is about 375 reqs/sec, the average response time increases and the throughput decreases when no controls are applied[7]. Since our workload contains CPU-intensive CGI-scripts, the CPU becomes overutilized and cannot process requests with the same rate as they arrive. Hence, the listen queue builds up which contributes additionally to the increase of the response time.

Using resource-based admission control, the acceptance rate of CGI-scripts is decreased which prevents the CPU from becoming a bottleneck and hence keeps the response time low. Decreasing the acceptance rate of CGI-scripts is sufficient until the request rate is about 675 reqs/sec. At this point the CGI acceptance rate reaches the predefined minimum and cannot be decreased anymore despite the CPU utilization being greater than the reference value. As the server's processing rate is lower than the request rate, the listen queue starts building up. Due to the increase of the listen queue, the controller computes a lower TCP SYN policing rate which limits the number of accepted requests. This can be seen in the left-hand graph where the throughput does not increase anymore for request rates higher than 800 reqs/sec. The right-hand graph shows that the average response time increases slightly when TCP SYN policing is active. Part of this increase is due to the additional waiting time in the listen queue.

We have repeated this experiment with workloads containing 10% dynamic requests and only static requests. If more requests are discarded using HTTP header-based control, the onset of TCP SYN policing should happen with higher request rates. The results in Table 1 show that this is indeed the case. When the fraction of dynamic requests is 20%, TCP SYN policing sets in at about

[7]For higher request rates than those shown in the graph the traffic generator runs out of socket buffers when no controls are applied.

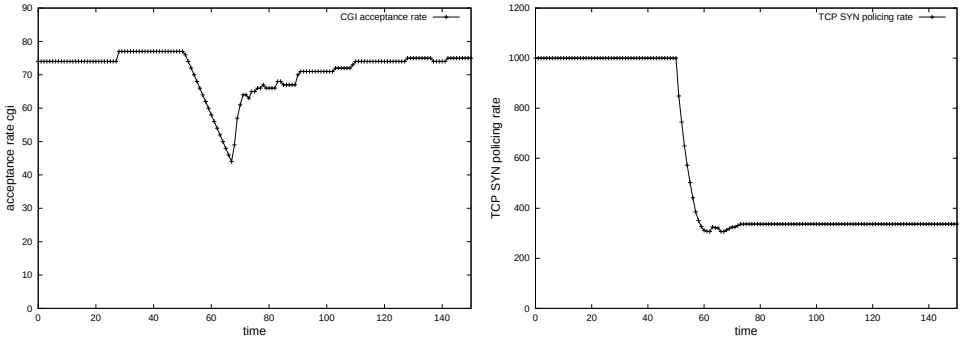

Figure 6: Adaptation under high load (left CGI acceptance rate, right SYN policing rate)

675 reqs/sec while the onset for SYN policing is at about 610 reqs/sec when all requests are for static files.

In summary, for low request rates, we prevent server overload using resource-based admission control that avoids over-utilization of the resource bottleneck, in this case CPU. For high request rates, when resource-based admission control is not sufficient, TCP SYN policing reduces the overall acceptance rate in order to keep the throughput high and response times low.

3.3 Exposure to a Very High Load

In this experiment we expose the server to a sudden high load and study the behaviour of the control algorithms. Such a load exposure could occur during a flash crowd or a Denial-of-Service (DoS) attack. We start with a relatively low request rate of 300 reqs/sec. After 50 seconds we increase the offered load to 850 reqs/sec and sustain this high request rate for 20 seconds before we decrease it to 300 reqs/sec again. We set the initial TCP SYN policing rate to 1000 reqs/sec.

Figure 6 shows that the TCP SYN policing rate decreases very quickly when the request rate is increased at time 50. This rapid decrease is caused by both parts of the control algorithms in Equation 2. First, since the length of the listen queue increases quickly, the contribution of the derivative part is high. Second, the absolute length of the listen queue is at that time higher than the reference value. Thus, the contribution of the proportional part of Equation 2 is high as well. The TCP SYN policing rate does not increase to 1000 again after the period of high load. However, we can see that around time 70, the policing rate increases to around 340 which is sufficiently high so that no requests need to be discarded by the SYN policer when the request rate is 300 reqs/sec. For higher request rates after the period of high load, the SYN policing rate settles at higher rates.

req rate large workload	metric	workload			
		large workload		surge workload	
		no controls	controls	no controls	controls
50 reqs/s	tput (reqs/sec)	46.8	41.5	270.7	289.2
50 reqs/s	resp. time (ms)	2144	80.5	1394.8	26.9
80 reqs/s	tput (reqs/sec)	55.5	45.8	205.2	285.1
80 reqs/s	resp. time (ms)	5400	94	3454.5	29.3

Table 2: Outgoing bandwidth

As expected, the CGI acceptance rate does not decrease as fast. With $K_{\text{P\_CPU}}$ being 1/5, the decrease of the rate is at most two per sampling point during the period of high load. Figure 6 also shows that the CGI-acceptance rate is restored fast after the period of high load. At a request rate of 300 reqs/sec, the CPU utilization is between 70 and 80%. Thus, the absolute difference to the reference value is larger than during the period of high load which enables faster increase than decrease of the CGI acceptance rate. At time 30 in the left-hand graph, we can see the CGI acceptance rate jump from 74 to 77. The reason for this jump is that during the last sampling period, the number of hits for the corresponding filter rule was above 90%, while it was otherwise below 90% until time 50.

3.4 Outgoing Bandwidth

Despite the fact that the workload used in the previous section contains some very large files, there were very few packet drops on the outgoing network interface. In the experiments in this section we make the bandwidth of the outgoing interface a bottleneck by requesting a large static file of size 142 KBytes from another host. The original host still requests the surge-like workload at a rate of 300 reqs/sec. From Figure 5 in Section 3.2, we can see that the server can cope with the workload from this particular host requested at this rate. The request of the large static file will cause overutilization of the interface and a proportional drop of packets to the original host.

Without admission control, we expect that packet drops on the outgoing interface will cause lower throughput and in particular higher average response times by causing TCP to back off due to the dropped packets. We therefore insert a rule that controls the rate at which large files are accepted. Large files are identified by a common prefix (`/islarge`). The aim of the experiment is to show that by adapting the rate with that requests for large files are accepted, we can avoid packets drops on the outgoing interface.

We generate requests to the large file with a rate of 50 and 80 reqs/sec. The results are shown in Table 2. As expected the response times for both workloads become very high when no controls are applied. In our experiments, we observed that the length of the queue to the interface was always around

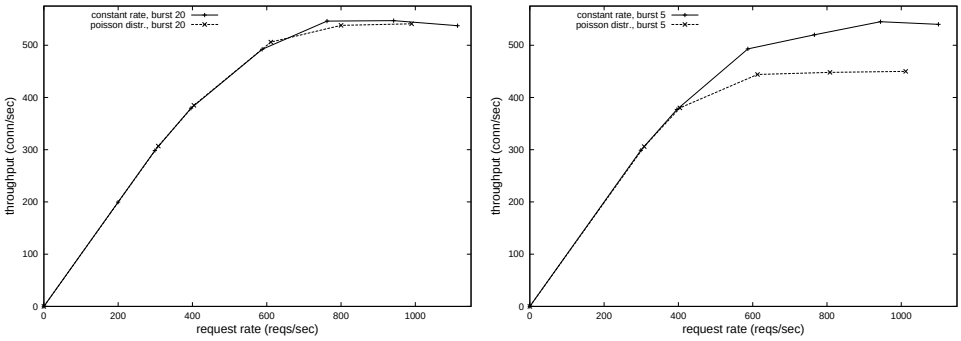

Figure 7: Comparison between constant request rate and traffic generated according to Poisson distribution for different bucket sizes (left 20, right 5)

the maximum value which indicates a lot of packet drops. By discarding a fraction of the requests for large files our controls keep the response time low by avoiding drops in the queue to the network interface. Although the throughput for the large workload is higher when no controls are applied, the sum of the throughput for both workloads is higher using the controls. Note, that when controls are applied the sum of the throughput for both workloads is the same for both request rates (about 331 reqs/sec).

3.5 Burstier Arrival Requests

The sclient program generates web server requests at a constant rate. The resulting requests also arrive at a constant rate to the web server. We have modified the sclient program to generate requests following a Poisson distribution with a given mean. To verify that our controllers can cope with burstier traffic we have repeated the experiment in Section 3.2 with a Poisson distribution. In Figure 7 we show the throughput at constant rate and at a Poisson distribution. The x-axis denotes the mean of the Poisson distribution or the constant request rate of the standard sclient program while the y-axis denotes the throughput. The difference between the two graphs is that the bucket size of the policers is 20 in the left-hand graph and five in the right-hand graph. Traffic generated at a constant rate should be almost independent of the bucket size since it arrives regularly at the server and a new token should always be available given the token rate is sufficient.

The left-hand graph shows that we achieve about the same throughput independent of the distribution of the requests when the policer's bucket size is 20. If the policer's bucket size is small as in the right-hand graph, more requests than necessary are rejected when the distribution of the requests' arrival times is burstier. This experiment shows that our adaptation mechanisms should be able to cope with bursty traffic provided we make a sensible choice of the bucket

service class	throughput (reqs/sec)
premium	331.1
standard	222.1
sum	553.2

Table 3: Throughput for each service class (equal request rates)

service class	throughput (reqs/sec)
premium	429.5
standard	120.9
sum	550.4

Table 4: Premium class using unused tokens from standard class

size.

3.6 Service Differentiation

The aim of this experiment is to show that our architecture can provide higher throughput to premium clients, using the scheme discussed in Section 2.4. We have two service classes, premium and standard. We reserve 3/5 of the tokens generated for the token bucket associated with TCP SYN policing for premium requests, while reserving the remaining tokens for standard requests. The size of the logical bucket for premium requests is 20, while the other logical bucket has a size of five. The size of the overflow buffer is also five. For simplicity, we have fixed the acceptance rate of CGI-scripts to 10 reqs/sec. Sclient requests the surge-workload from two machines. We have two filter rules. One rule matches the IP address of the host emulating premium clients, the other one is a default rule matching the requests from the standard host.

In the first experiment we request the same workload from both hosts at a rate of 450 reqs/sec each. We expect that TCP SYN policing is active at this request rate. Of the accepted requests about 60% should be premium and 40% should be standard requests. The results are shown in Table 3. The overall throughput is 553.2 reqs/sec. The throughput of the premium requests is 331.1 reqs/sec which is about 60% of the total throughput.

In the next experiment we want to show that the unused tokens of one service class can be utilized by the other service class using the overflow buffer. The request rate for premium requests is 760 reqs/sec while the request rate for standard requests is about 150 reqs/sec. Again, the acceptance rate of CGI-scripts is fixed to 10 reqs/sec. The token rate for the standard partition of the token bucket will be higher than 150 tokens per second. Thus, this logical bucket will become full and the exceeding tokens are put into the overflow buffer where they can be consumed by premium requests. The results are shown in Table 4. The throughput for standard requests is about 121 reqs/sec, i.e. the re-

quested rate minus most of the dynamic requests. The throughput for premium requests is 429.5 requests/sec, i.e. more than 60% of the overall acceptance rate. TCP SYN policing accepts slightly below 700 reqs/sec. However, most of the dynamic requests are then discarded by the resource-based admission control. These results demonstrate that the service differentiation mechanism works as expected.

4 Architectural Extensions

Our current implementation is targeted towards single node servers or the back-end servers in a web server cluster. We believe that the architecture can easily be extended to LAN-based web server clusters and enhance sophisticated request distribution schemes such as HACC [33] and LARD [28]. In LARD and HACC, the front-end distributes requests based on locality of reference to improve cache hit rates and thus increase performance. Aron et al. [7] increase the scalability of this approach by performing request distribution in the back-ends. In our extended architecture, the front-end performs resource-based admission control. The back-end servers monitor the utilization of each critical resource and propagate the values to the front-end. Based on these values, the front-end updates the rates for the token bucket based policers using the algorithms presented in Section 2.3. After the original distribution scheme has selected the node that is to handle the request, compliance with the corresponding token bucket ensures that critical resources on the back-ends are not overutilized. This way, we consider the utilization of individual resources as a distribution criteria which neither HACC nor LARD do. HACC explicitly combines these performance metrics into a single load indicator.

The front-end also computes the rate of the SYN policers for each back-end based on the listen queue lengths reported by the back-ends. Using these values, the front-end itself performs SYN policing, using the sum of the acceptance rates of the back-ends as acceptance rate to the whole cluster. There are two potential problems: First, the need to propagate the values from the back-ends to the front-end causes some additional delay. If the evaluation of the system shows that this is indeed a problem, we should be able to overcome it by setting more conservative reference values or by increasing the sampling rate. Second, there is a potential scalability problem caused by the need for $n * c$ token buckets on the front-end, where n is the number of back-ends and c the number of critical resources. However, we believe that this is not a significant problem since a token bucket can be implemented by reading the clock (which in kernel space is equal to reading a global variable) and performing some arithmetical operations.

For a geographically distributed web cluster resource utilization of the servers and the expected resource utilization of the requests can be taken into account when deciding on where to forward requests.

Our architecture is implemented as a kernel module, but could be deployed in user space or in a middleware layer. Since our basic architecture is implemented

as a kernel module, we have decided to put the control loops in the kernel module as well. An advantage of having the control mechanisms in the kernel is that they are actually executed at the correct sampling rate. But the same mechanisms could be deployed in user space or in a middleware layer.

Our kernel module is not part of the TCP/IP stack which makes it easy to port the mechanisms. The only requirements are availability of timing facilities to ensure correct sampling rates and facilities to monitor resource utilization.

It is also straightforward to extend the architecture to handle persistent connections. Persistent connections represent a challenging problem for web server admission control since the HTTP header of the first request does not reveal any information about the resource consumption of the requests that might follow on the same connection. A solution to this problem is proposed by Voigt and Gunningberg [30] where under server overload persistent connections that are not regarded as important are aborted. The importance is determined by the cookies. This solution can easily be adapted to fit our architecture. If a resource is overutilized, we abort non-important persistent connections with a request matching the filter rule associated with that resource.

It would be interesting to perform studies on user perception. Since the TCP connection between server and client is already set up when HTTP header-based control decides on accepting a request, we can inform the client (in this case the user) by sending a "server busy" notification. TCP SYN policing, on the other hand, just drops TCP SYNs which with current browsers does not provide timely feedback to the client. This is another reason for keeping TCP SYN policing inactive as long as resource-based admission control can prevent server overload.

The netfilter framework which is part of the Linux kernel contains functionality similar to TCP SYN policing. We plan to invest if TCP SYN policing can be reimplemented using netfilter functionality. Other operating systems such as FreeBSD contain firewall facilities that could be used to limit the bandwidth to a web server. It is possible to use such facilities instead of SYN policing. However, since there is no one-to-one mapping between bandwidth and requests, it is harder to control the actual amount of requests entering the web server.

The proposed solution of grouping the objects according to resource demand in the web server's directory tree, is not intuitive and awkward for the system administrator. We assume that this process can be automated using scripts.

5 Related Work

Casalicchio and Colajanni [13] have developed a dispatching algorithm for web clusters that classifies client requests based on their impact on server resources. By dispatching requests appropriately they ensure that the utilization of the individual resources is spread evenly among the server back-ends. Our and their approach have in common that they utilize the expected resource consumption of web requests, however, for different purposes.

Several others have adopted approaches from control theory for server systems. Abdelzaher and Lu [2] use a control loop to avoid server overload and meet individual deadlines for all served requests. They express server utilization as a function of the served rate and the delivered bandwidth [1]. Their control task is to keep the server utilization at $ln2$ in order to guarantee that all deadlines can be met. In our approach we aim for higher utilization and throughput. Furthermore, our approach also handles dynamic requests. In another paper, Lu et al. [24] use a feedback control approach for guaranteeing relative delays in web servers. Parekh et al. [18] use a control-theoretic approach to regulate the maximum number of users accessing a Lotus Notes server. While a focus of these papers is to use control theory to avoid the absence of oscillations, Bhoj et al. [11] in a similar way as we, use a simple controller to ensure that the occupancy of the priority queue of a web server stays at or below a pre-specified target value. Reumann et al. [23] use a mechanism similar to TCP SYN policing to avoid server overload.

Several research efforts have focused on overload control and service differentiation in web servers [3, 10, 22, 15]. *WebQoS* [10] is a middleware layer that provides service differentiation and admission control. Since it is deployed in middleware, it is less efficient compared to kernel-based mechanisms. Cherkasova et al. [15] present an enhanced web server that provides session-based admission control to ensure that longer sessions are completed. Their scheme is not adaptive and rejects new requests when the CPU utilization of the server exceeds a certain threshold. The focus of cluster reserves [6] is to provide performance isolation in cluster-based web servers by managing resources, in their work CPU. Their resource management and distribution strategies do not consider multiple resources.

There are some commercial approaches that deserve mention. Cisco's *LocalDirector* [16] enables load balancing across multiple servers with per-flow rate limits. Inktomi's *Traffic Server C-Class* [21] provides system server overload detection and throttling from traffic spikes and DoS attacks by redistributing requests to caches. Alteon's *Web OS Traffic Control Software* [4] parses HTTP headers to perform URL-based load balancing and redirect requests based on content type to servers.

6 Conclusions

We have presented an adaptive server overload protection architecture for web servers. Using the application-level information in the HTTP header of the requests combined with knowledge about resource consumption of resource-intensive requests, the system adapts the rates at which requests are accepted. The architecture combines the use of such resource-based admission control with TCP SYN policing. TCP SYN policing first comes into play when the load on the server is very high since it wastes less resources when rejecting requests. Our experiments have shown that the acceptance rates are adapted as expected. Our

system sustains high throughput and low response times even under high load.

7 Acknowledgements

This work builds on the architecture that has been developed at IBM TJ Watson together with Renu Tewari, Ashish Mehra and Douglas Freimuth [31]. The authors also want to thank Martin Sanfridson and Jakob Carlström for discussions on the control algorithms and Andy Bavier, Ian Marsh, Arnold Pears and Bengt Ahlgren for valuable comments on earlier drafts of this paper.

This work is partially funded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for Strategic Research.

References

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

[2] T. Abdelzaher and C. Lu. Modeling and performance control of internet servers. In *Invited paper, IEEE Conference on Decision and Control*, December 2000.

[3] 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*, March 1999.

[4] Alteon. Alteon Web OS Traffic Control Software. http://www.nortelsnetworks.com/products/01/webos.

[5] M. Arlitt and C. Williamson. Web server workload characterization: The search for invariants. In *Proc. of ACM Sigmetrics*, April 1996.

[6] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Proc. of ACM SIGMETRICS*, June 2000.

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

[8] G. Banga and P. Druschel. Measuring the capacity of a web server. In *Proc. of USITS*, December 1997.

[9] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proc. of SIGMETRICS*, 1998.

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

[11] P. Bhoj, S. Ramanathan, and S. Singhal. Web2k: Bringing qos to web servers. Technical report, HP, May 2000.

[12] V. Cardellini, M. Colajanni, and P. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, May 1999.

[13] E. Casalicchio and M. Colajanni. A client-aware dispatching algorithm for web clusters providing multiple services. In *Proc. of 10th Int'l World Wide Web Conference*, May 2001.

[14] J. Challenger, P. Dantzig, and A. Iyengar. A scalable and highly available system for serving dynamic data at requently accessed web sites. In *Proc. of ACM/IEEE SC 98*, November 1998.

[15] L. Cherkasova and P. Phaal. Session based admission control: a mechanism for improving the performance of an overloaded web server. Technical report, HP, 1999.

[16] Cisco. Cisco localdirector. http://www.ciso.com.

[17] L. Eggert and J. Heidemann. Application-level differentiated services for web servers. *World Wide Web Journal*, 3(2):133–142, September 1999.

[18] S. Parekh *et al.* Using control theory to achieve service level objectives in performance management. In *IFIP/IEEE International Symposium on Integrated Network Management*, May 2001.

[19] T. Glad and L. Ljung. *Reglerteknik: Grundläggande teori (in Swedish).* Studentlitteratur, 1989.

[20] M. Harchol-Balter, B. Schroeder, M. Agrawal, and N. Bansal. Size-based scheduling to improve web performance. submitted for publication, http://www-2.cs.cmu.edu/ harchol/Papers/papers.html.

[21] Inktomi. Inktomi traffic server c-class. http://www.inktomi.com/products/cns/products/tscclass_works.html.

[22] R. Iyer, V. Tewari, and K. Kant. Overload control mechansims for web servers. In *Performance and QoS of Next Generation Networks*, November 2000.

[23] H. Jamjoom and J. Reumann. Qguard: Protecting internet servers from overload. Technical report, University of Michigan, 2000.

[24] C. Lu, T. Abdelzaher, J. Stankovic, and S. Son. A feedback control approach for guaranteeing relative delays in web servers. In *Real-Time Technology and Application Symposium*, June 2001.

[25] S. Manley and M. Seltzer. Web facts and fantasy. In *Proc. of USITS*, December 1997.

[26] A. Mehra, R. Tewari, and D. Kandlur. Design considerations for rate control of aggregated TCP connections. In *Proc. of NOSSDAV*, June 1999.

[27] Mindcraft. Webstone. http://www.mindcraft.com.

[28] V. Pai, M. Aron, G. Banga, M. Svendsen, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, USA, October 1998.

[29] A. van de Ven. Khttpd. http://www.fenrus.demon.nl/.

[30] T. Voigt and P. Gunningberg. Kernel-based control of persistent web server connections. *ACM Performance Evaluation Review*, 29(2):20–25, September 2001.

[31] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proc. of Usenix Annual Technical Conference*, June 2001.

[32] Z. Wang. Cachemesh: A distributed cache system for the world wide web. In *2nd NLANR Web Caching Workshop*, 1997.

[33] X. Zhang, M. Barrientos, J. Chen, and M. Seltzer. HACC: An architecture for cluster-based web servers. In *Third Usenix Windows NT Symposium*, pages 155–164, Seattle, WA, July 1999.

Paper E

Andy Bavier, Thiemo Voigt, Mike Wawrzoniak, Larry Peterson, Per Gunningberg. SILK: Scout Paths in the Linux Kernel. Technical Report 2002-009, Department of Information Technology, Uppsala University, Uppsala, Sweden, February 2002.

SILK: Scout Paths in the Linux Kernel

Andy Bavier
(acb@cs.princeton.edu)
Princeton University

Thiemo Voigt
(thiemo@sics.se)
Swedish Institute of Computer Science

Mike Wawrzoniak
(mhw@cs.princeton.edu)
Princeton University

Larry Peterson
(llp@cs.princeton.edu)
Princeton University

Per Gunningberg
(Per.Gunningberg@it.uu.se)
Uppsala University

Abstract

SILK stands for Scout In the Linux Kernel, and is a port of the Scout operating system to run as a Linux kernel module. SILK forms a replacement networking subsystem for standard Linux 2.4 kernels. Linux applications create and use Scout paths via the Linux socket interface with virtually no modifications to the applications themselves. SILK provides Linux applications with the benefits of Scout paths, including early packet demultiplexing, per-flow accounting of resources, and explicit scheduling of network processing. SILK also introduces the concept of an *extended path* to provide a framework for application QoS. We demonstrate the utility of SILK by showing how it can provide QoS for the Apache Web server.

1 Introduction

In recent years, many research efforts have focused on improving operating system architectures. Architectural features have been advanced to help systems avoid receive livelock and overload, fight denial of service attacks, account for kernel resources used on behalf of applications, and provide application QoS. Despite the quality of many of these efforts, their underlying ideas have not spread to the average desktop. For example, Linux is a popular operating system with freely available source. Yet the standard Linux networking stack still cannot prioritize among incoming network packets, and so a Linux application cannot take full advantage of QoS provided by the network. Nor can an MPEG player running in Linux inform the system that it requires a certain rate on the CPU to play its video. Researchers have solved these problems, but adoption of the mechanisms they have proposed, and the applications to exploit them, has been slow. The question is, how can research more quickly have an impact on the systems and programs that average people use every day?

We have identified several factors that inhibit the quick distribution and widespread evaluation of systems research ideas. They are:

- Steep learning curves. Several research systems have been built from scratch, usually around better abstractions and architectures, and are freely available. These systems often possess real advantages. However, most people are reluctant to invest the time to master a new system.
- Lack of applications. This problem plagues research systems that are built from the ground up, but also affects any research idea that requires heavy changes to existing applications. The issue is one of chicken-or-egg: new mechanisms will not be quickly adopted if few applications can use them effectively, yet people will not spend effort rewriting applications to take advantage of mechanisms that are unavailable.
- Kernel patches. Most good research efforts are targeted at specific problems. However, a sysadmin may want to combine a number of these solutions in her system. She faces unpredictable results if she downloads six kernel patches from six different research projects and applies them all at once—there could be feature interaction, or the patches themselves might conflict. Also, a patch may not be available for the kernel version she is using.

This paper presents SILK, which stands for Scout In the Linux Kernel, in response to these problems. SILK makes three contributions. First, SILK is a replacement networking subsystem for Linux based on the Scout path architecture [18]. Scout paths combine a number of widely advocated research ideas into a simple, clean system abstraction. Applications interact with SILK through the Linux socket interface. Second, SILK provides a QoS framework through the idea of an *extended path*. SILK conceptually extends the path from the network to the application to coschedule application and network processing. We believe that many current applications can take advantage of this framework with minimal modifications, providing them with immediate benefits. Third, SILK is packaged as a kernel module that can be loaded into a standard Linux 2.4 kernel. SILK allows people to experiment with advanced research ideas with very little effort and risk, and serves as a vehicle for widely distributing these ideas and evaluating them in real contexts.

We demonstrate the capabilities of SILK using the Apache Web server, a popular and complex application. Our results show that SILK's performance is competitive with the native Linux network stack, and that Apache with SILK can provide nearly constant response time for preferred requests independent of the total number of clients accessing the server.

The rest of the paper is organized as follows. Section 2 presents an overview of Scout paths, which form the heart of SILK, and shows how SILK fits into Linux. Section 3 describes important parts of the SILK design. Section 4 presents a set of experiments done with Apache, to show that SILK can provide

benefits to a widely-used application. Section 5 discusses further issues and future work, and Section 6 discusses other work related to SILK.

2 Overview

2.1 Scout

Scout [18] is a modular, configurable, communication-oriented operating system developed for small network appliances. Scout was designed around the needs of data-centric applications with particular attention given to networking. It incorporates a number of ideas found in other network architectures as well. They are:

Early demultiplexing of incoming packets to flow queues. This allows the system to isolate flows as early as possible, in order to prioritize packet processing and accurately account for resources.

Early dropping when flow queues are full. The server can avoid overload by dropping packets before investing many resources in them.

Accounting of the resources used by each data flow, including CPU, memory, and bandwidth. Knowledge of the resources used by a flow is necessary in order to provide overall fairness or to place resource limits on individual flows.

Explicit scheduling of flow processing, including network processing. Scheduling and accounting are combined to provide resource guarantees to flows; for example, CPU or bandwidth reservations.

Extensibility through Scout's modular design. This makes it easy to add new protocols and construct new network services. Different protocol versions can exist side-by-side. A new service can be deployed by specifying a sequence of modules for a data flow.

Scout's main contribution is to combine all of the features listed above into a single, clean abstraction: the *path*. A path is a structured system activity. Each Scout path encapsulates a flow of data, for example, a single TCP connection. A path consists of a string of code modules that process and perhaps transform the data as it flows through the system, and all resources consumed on the flow's behalf are charged to the path. Previous research has demonstrated the usefulness of Scout paths for distributing multimedia processing across configurable network nodes [20], scheduling packet processing in a software router [23], and for protecting against denial of service (DoS) attacks [25].

Figure 1 shows a picture of a Scout TCP path. The path corresponds to a single TCP connection. It consists of a chain of protocol modules that process packets belonging to the connection, with input and output queues at each end.

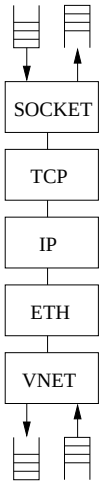

Figure 1: A Scout path

When a packet arrives on the network device VNET, it is demultiplexed based on its header information to locate its corresponding path. If the packet belongs to the TCP connection of this path, it is placed in the input queue at the bottom of the path; if the queue is full the packet is dropped. A path is considered *ready* to run once it has data in its input queue.

The Scout scheduler chooses a path to run from among those that are ready. Scout provides a number of CPU schedulers for scheduling paths; fixed priority, Earliest Deadline First (EDF), Weighted Fair Queueing (WFQ), and Best Effort Real Time (BERT) schedulers can be configured in Scout. When a path is run, a thread belonging to the path dequeues a piece of data from its input queue, runs the code modules in sequence, and deposits the result in the output queue at the opposite end of the path. Scout can prioritize among different data flows using a fixed priority scheduler, or give each a CPU share with a WFQ scheduler. The combination of paths and configurable scheduling allows Scout to produce a rich variety of system behaviors.

2.2 SILK

SILK is an encapsulation of Scout in a Linux kernel module. SILK provides a drop-in, extensible networking subsystem for Linux based on Scout paths. SILK also provides a framework for building application-level QoS solutions through the concept of an *extended path*. In this section we give a high-level view of SILK.

Figure 2 shows the SILK kernel module within the Linux kernel. In the left portion, the SILK module exchanges data with the network device drivers, the

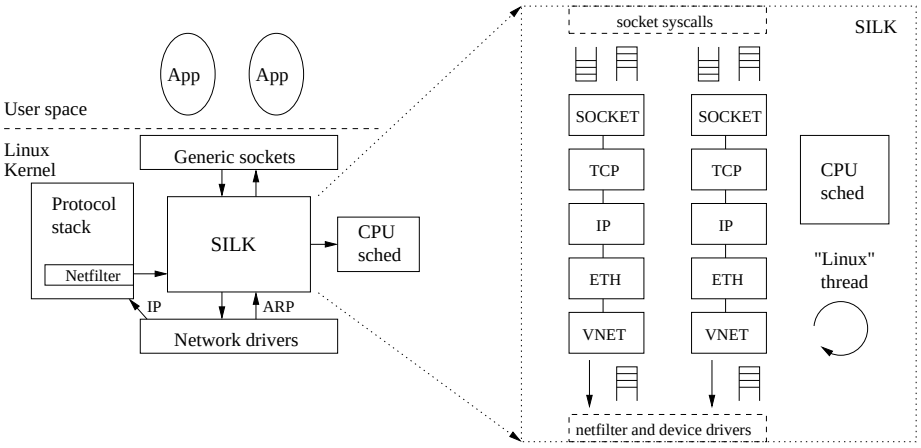

Figure 2: The SILK module in Linux

socket interface, and the packet filter interface `netfilter` in standard ways. SILK also modifies the scheduling parameters of Linux tasks to influence the decisions made by the Linux CPU scheduler. The right part of the figure shows two Scout paths within SILK, corresponding to two TCP connections. (In the remainder of this paper, when we refer to a "path" we mean a Scout path in SILK.) Each path has two input queues and one output queue; the lower output queue is unnecessary since the path sends outgoing packets straight to the device. [8] The SILK module also contains its own CPU scheduler that cooperates with the one in Linux. This cooperation is represented by the "Linux thread"; by executing this thread, SILK transfers control to the Linux scheduler. The next section will discuss each piece in more detail and describe how they fit together.

3 Design of SILK

We have three main design goals for SILK:

1. Minimal changes to Linux. Since we want SILK to be widely used it is important that it is able to run in an unmodified Linux kernel. Only minimal changes should be required to Linux applications to enable them to use paths in SILK.

2. SILK has control of the CPU. Our aim is to take advantage of the different CPU schedulers implemented in Scout. In particular, we want to be able to prioritize among paths, and to provide them with CPU guarantees.

[8] In other words, the output queue is the packet queue maintained by the network device driver.

3. Coscheduling of paths and applications. SILK should be able to schedule Linux tasks as well as paths to provide application-specific QoS.

This section describes how our design and implementation of SILK meets each of these goals.

3.1 SILK Sockets

Linux applications create and use paths in SILK through the Linux socket API. New applications can use the new PF_SCOUT protocol family to construct paths with experimental or non-standard protocols. SILK can also intercept socket calls on the PF_INET family, allowing unmodified legacy applications to access TCP and UDP paths. [9] In both cases, SILK implements socket operations and intercepts network packets using interfaces exported to kernel modules by Linux, and hence does not require any kernel patches. The rest of this section focuses on the TCP path shown in Figure 1, and describes how this path exchanges data with Linux.

The VNET module at the bottom of the path connects to the `netfilter` interface, the packet filtering interface provided by recent Linux kernels. Incoming packets belonging to a TCP path must be processed only by SILK and not by Linux. To accomplish this, SILK inserts a netfilter hook at the earliest possible location, before Linux has performed any IP processing; all incoming IP packets are diverted to this hook. SILK demultiplexes each packet to see if it matches a path. If it matches, then the packet is enqueued at the bottom of the path, the path is scheduled as described in Section 2.1, and Linux is instructed to drop the packet. If demultiplexing does not match the packet to a path, SILK lets Linux continue with network processing.

SILK uses another method to receive ARP packets (netfilter only handles IP packets). SILK snoops ARP packets directly from the network device using a lower-level interface than netfilter. This interface is not as powerful as netfilter since it is not possible to use it to filter packets. However, it is sufficient for implementing new network-layer protocols within SILK, because by default the Linux networking stack will drop packets it receives for unknown protocols.

At the top of the path, the SOCKET module connects to Linux using the generic socket interface in the kernel. SOCKET supplies routines that map the familiar socket calls into operations on paths. For example, `connect()` creates a path for a new TCP connection and then starts the three-way handshake; `recv()` reads data from the output queue of the path and passes it to user space; `send()` reads data from user space and enqueues it on the path's input queue; and `close()` tears down the connection and destroys the path. These hooks are straightforward.

An interesting point is that the SOCKET and VNET modules are actually specializations of a single Scout module called GenericNet. This module pro-

[9] This behavior is configurable. If on, SILK takes over all networking functions; if off, SILK and Linux networking exist side-by-side.

vides generic versions of the operations that always occur at path endpoints (e.g., demultiplexing and enqueuing messages), while allowing the module to be specialized for different contexts. For example, when the application invokes `send()`, the data is passed to SILK in a user-space buffer. The specialized code in SOCKET reads the data from user space and converts it to a Message (SILK's internal packet abstraction). Likewise, VNET converts the outgoing packet from a Message to an `sk_buff` (Linux's internal representation). GenericNet is also used to implement hardware device drivers in Scout.

3.2 CPU Scheduling

SILK contains its own CPU scheduler and thread package that coexists with the Linux scheduler. In this discussion, SILK runs *threads* within paths, while the Linux scheduler runs *tasks*. SILK implements thread scheduling on top of a Linux kernel task created when the SILK module initializes. As far as we are aware, we are the first to implement a self-contained thread package and scheduler in a Linux kernel module; we realize that this may be regarded as an abuse of the kernel module concept. This section describes how the SILK and Linux schedulers interact, i.e., how SILK controls the CPU.

SILK creates a Linux kernel task at startup and sets its priority to maximum realtime priority, the highest priority in the system. Therefore, the SILK kernel task will run almost immediately whenever it is runnable [10]. Kernel tasks run nonpreemptively in Linux, hence another task can be scheduled to run only when the SILK kernel task yields or sleeps. SILK multiplexes all of its threads onto this high priority kernel task.

SILK temporarily transfers control back to Linux through the "Linux" thread shown in Figure 2. This "Linux" thread is an actual thread in SILK and SILK can schedule it like any other thread. When SILK executes this thread, it causes the SILK kernel task to yield and thus transfers control to the Linux scheduler. Note that a Linux task that yields is not considered runnable again until another task has run. Therefore, when SILK executes the "Linux" thread, the Linux scheduler chooses one other task to run and then transfers control back to SILK. Through the mechanism of the "Linux" thread, SILK gains the ability to allow Linux to run one task.

The scheduling parameters assigned to the "Linux" thread determine SILK's policy for transferring control to Linux. This policy will depend on which scheduler SILK is running—for example, with the WFQ scheduler the thread can be given a CPU rate of 50%, and this will cause SILK and Linux to evenly share the CPU. In this manner SILK controls how often Linux "as a whole" is allowed to run.

[10] It may have to wait for another kernel task to yield.

3.3 Extending the Path

One of our goals is to provide application-specific QoS to Linux programs that use SILK. A scheduling-aware application should be able to specify how it and its paths are scheduled by the system. On the other hand, since SILK is modular and configurable, intelligence can be built directly into SILK so that it can provide QoS to existing "dumb" applications without the application's participation or even its knowledge. To this end, SILK introduces the concept of an *extended path* to encompass coscheduling of applications and paths.

The idea is that, by giving the SILK scheduler the ability to control Linux's scheduling decisions, SILK can coordinate processing between the network stack and the application. SILK must be able to do two things to "extend the path" in this sense. First, it must identify the task associated with each SILK path. This is simply the task that calls `connect()` or `accept()`. Second, SILK must cause the Linux scheduler (running Linux tasks) to mirror the decisions made by the SILK scheduler (running paths). Exactly what form this cooperation takes depends on which scheduler SILK is using. We have implemented path extensions for SILK's fixed priority scheduler as follows.

We had two objectives when implementing an extended path for the priority scheduler. The first was that networked Linux tasks using SILK's networking stack should be scheduled at the same relative priorities as their corresponding paths. The second was that, since we want to provide these tasks with QoS, they should run at an absolute priority higher than other tasks in the system to avoid interference from these tasks. Note that the Linux scheduler usually provides some form of fairness to tasks. When choosing a task to run, Linux takes its priority (i.e., as set by `nice`) into account but does not strictly schedule by task priority alone. However, Linux can be made to perform strict priority scheduling by using the realtime priorities. A task with realtime priority p runs at a higher priority than a realtime task of priority less than p as well as any non-realtime task. Realtime priorities meet both of our objectives, and hence we map SILK priorities onto Linux realtime priorities.

SILK forms an extended path by mirroring the network path's priority in the realtime priority of the Linux task that uses it. For example, if a path has priority 2 (in SILK) then a Linux task reading from it would inherit a realtime priority of 2 (in Linux). Furthermore, the priority inherited by a Linux task from a path can change over time. A task that blocks on `accept()` first receives the priority of the SILK listen socket's path. Then it adopts the path priority of the socket returned by `accept()` and finally it returns to its original priority when closing the socket. SILK does not change the priorities of Linux tasks, except for the ones that use SILK paths.

Priority inversion will result if SILK chooses to schedule a path when there is a runnable Linux task with a higher priority. To avoid this, the SILK kernel task yields to Linux when a runnable task has a higher priority than any ready path. For each priority p starting with the highest, if there are no ready paths with that priority, then the SILK priority scheduler checks a list of Linux tasks

having realtime priority p to see if one is runnable. If so, then SILK runs the Linux thread described in Section3.2. This causes the SILK kernel task to yield, and the Linux scheduler then runs one of the tasks with realtime priority p. If SILK finds nothing runnable at any priority, it runs the Linux thread by default; this allows Linux to schedule a task unrelated to SILK. In this way, SILK socket priorities are inherited by Linux tasks and SILK controls the scheduling of Linux tasks as well as paths.

4 Evaluation

SILK is an entire networking subsystem and not just an architectural feature. As such, it introduces new implementations of network protocols and scheduling algorithms into Linux. Because SILK itself encompasses so much, and because the benefits of Scout paths have been shown elsewhere, our experiments focus on demonstrating SILK's high-level behavior with a real application: the popular Apache web server. Our evaluation focuses on three areas: performance, prioritizing network processing, and providing service differentiation through extended paths.

Our testbed consists of a server and two traffic generators. The server machine is a 1.4 GHz Athlon with 256 MB of memory running SILK on Linux 2.4 and Apache version 1.3.20. We are running the standard Apache configuration unless explicitly mentioned. The traffic generator machines are both Pentium IIIs, running at 733 MHz and 600 MHz. All three machines have Netgear GA622T Gigabit Ethernet cards and are connected via a 1 Gb/sec Ethernet switch. Our desire is to stretch the limits of SILK by using muscular machines and fast networks.

As a traffic generator we use sclient [4]. Sclient uses a single process to manage a large number of concurrent connections to the server. In our version, sclient makes a request to the server and waits for the response. Immediately after receiving the response, sclient makes a new request to the server. By increasing the number of concurrent connections (referred to as clients), the load on the server can be increased. We use up to 97 clients in our experiments.

4.1 Performance

In the first set of experiments we compare the performance of Apache on SILK against Apache on Linux. Our goal is to show that, for a real application, the overall performance of Linux and SILK are comparable. This would provide evidence that SILK can be a viable network subsystem replacement for Linux.

Two metrics are important when measuring a Web server's performance [7]: overall throughput and response time. The throughput measured in requests served per second provides a good indication of the efficiency of the system. The response time or latency is the time measured by the client from when it opens a connection until the last byte of the response arrives. Response time is

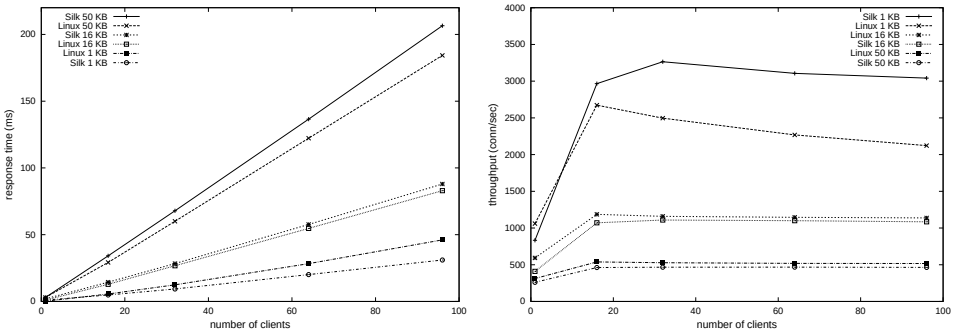

Figure 3: Comparison between standard Linux and SILK: response time and throughput for small files

a measurement of the usability of the system, since people perceive long delays as unacceptable [8]. Each data point in our experiments represents an average of the observed throughput or response time over a run of 2 minutes.

Our experiments measure the throughput and latency for a group of clients making requests to the server. For each run of the experiment we configure sclient with a static number of clients, all of which request the same file. We vary the number of clients and the size of the requested file across runs. Note that we do not use a more realistic request pattern because our goal is to stress different aspects of the networking stack, and constant file sizes are more useful for this purpose. Shorter file sizes place more emphasis on per-connection overheads [6], for example, SILK path creation and destruction. On the other hand, inefficiencies in the SILK protocol stack, such as excessive data copying, should be more visible using large files.

For this experiment we configure all SILK paths with equal priorities. Since the SILK kernel task runs at a higher priority than any Linux task, this means that all paths are prioritized equally within SILK but have higher priority than Linux tasks. In Linux this is also the case because protocol processing occurs in a high-priority interrupt context. Therefore, we expect that SILK and Linux will behave similarly. In particular we expect that the overall throughput for both Linux and SILK remains roughly the same regardless of the number of clients. The response time should increase linearly with the number of clients, since we would expect each client to receive about $1/n$ of the resources when n clients are active simultaneously.

Figures 3 and 4 present the results for SILK and Linux. Figure 3 shows results for three small files of sizes between 1 KB and 50 KB. The bottom graph shows the throughput and the top graph shows the response time. The figure shows that the throughput of SILK is slightly lower and the response time slightly higher than for Linux, except for 1 KB files, where for some reason SILK is faster than Linux. The same measurements for three large files between

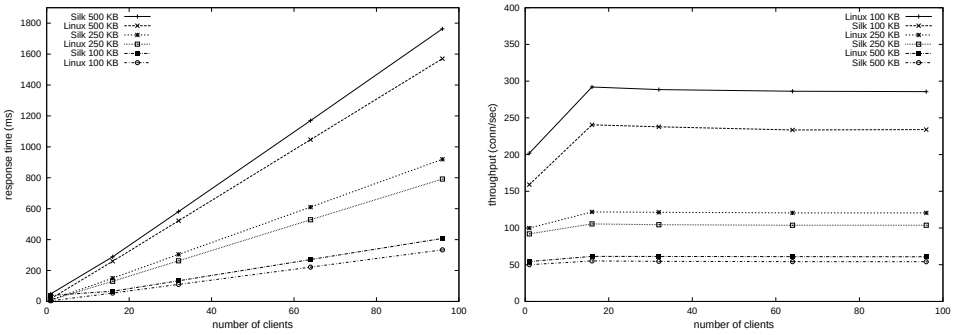

Figure 4: Comparison between standard Linux and SILK: response time and throughput for large files

Description	**Avg Time in μs**
Path create	41.5
Path destroy	12.3
Packet demultiplexing	1.8
Copy from Msg to sk_buff	4.3
Copy from sk_buff to Msg	1.1

Table 1: Microbenchmarks for SILK

100 KB and 500 KB are shown in Figure 4. Ignoring the 1 KB file results which we feel to be anomalous, these graphs show that SILK's networking performance is marginally slower than Linux.

We are aware of inefficiencies in SILK that can affect performance. For example, SILK uses Scout Messages as its packet abstraction. This requires copying data between a Linux `sk_buff` and a Message when moving a packet between SILK and Linux. Byte copying is one of the most expensive operations for Web servers [19]. The copy could be avoided by integrating Messages and `sk_buff`s; this optimization is in progress. Also, path creation and destruction are fairly heavyweight operations. Table 1 shows some microbenchmarks for SILK averaged across runs for different file sizes. We note that caching SILK paths for reuse could save considerable overhead, especially when the server is handling thousands of requests per second. Packet demultiplexing is already fairly cheap, but since demultiplexing occurs in an interrupt context, it should be further optimized in order to decrease the vulnerability of the system to receive livelock [17]. Finally, it is much more expensive to copy data from a Message (which can be fragmented across multiple buffers) than from an `sk_buff`. SILK performs the more expensive copy when sending an outgoing packet; this overhead represents a significant performance hit for a Web server. In summary, there is room for optimization in SILK, but its overall performance

appears to be respectable.

4.2 Prioritizing Paths

SILK sockets provide the ability to schedule network processing on a per-connection basis. For instance, if the underlying network provides some quality of service (traffic priorities or bandwidth reservations), SILK can extend this QoS to the application itself. One question is how much current non-QoS-aware applications can benefit from this capability. In this section we investigate allowing SILK to schedule only paths and not Apache.

For the experiment there are two classes of requests: preferred and standard. We differentiate between these based on source IP address. This determination could also be based on information gathered from the network (e.g., the Type of Service field in the IP header), or by the URL requested or an embedded cookie.

We assign the following priorities to SILK paths: paths handling preferred requests have priority 2, the listen socket's path has priority 1, and standard request paths have priority 0. [11] These priorities affect network processing as follows. All incoming SYN packets are delivered to the listen path. When the listen path runs, the source IP address of the SYN is inspected and a new path of the appropriate priority is created to handle the request. We chose priority 1 for the listen path because SYNs belonging to both request classes arrive on this path. We wanted to give preferred SYN packets a higher priority than standard connections, yet ensure that standard SYN packets had a lower priority than preferred connections. Note that only the network processing done in SILK is prioritized in this way. Linux runs the Apache server processes as usual, and the "Linux" thread in SILK runs at priority 0.

We repeat the experiment in Section 4.1 while adding a client generating preferred requests. We run sclient on two host. The first runs a static number of standard clients requesting the same file as before. The second runs one client also requesting the same file but these requests are preferred. Again, we vary the number of standard clients and the requested file size across runs. We also increase the number of Apache server processes to 100; using the low standard number of Apache server processes, the time requests spend in the listen queue dominates the response time. We would expect to see some improvement in the response time of the preferred requests.

The results in Figure 5 compare the response times of preferred requests in this experiment to the response times we observed in Experiment 4.1 for four representative file sizes. The x-axis denotes the number of standard clients competing with the preferred client. Our initial assumption was that prioritizing paths in SILK would show a greater improvement for preferred requests of large files, since they require more network processing. However, the results clearly show a benefit for small files and none for large files. The second column in

[11] Higher values mean higher priority.

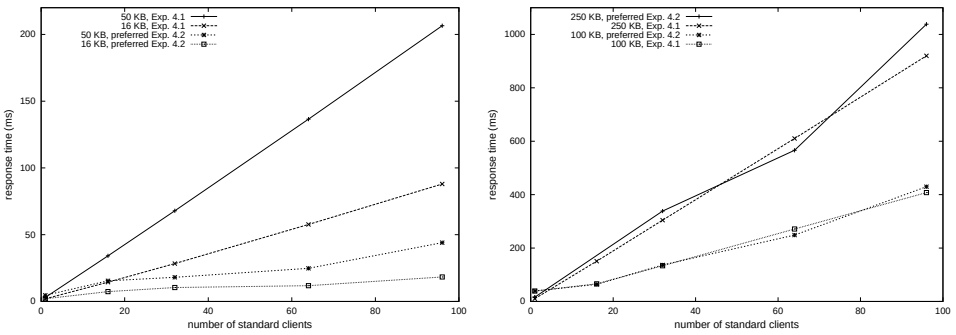

Figure 5: Comparison between response time for small (left) and larger file requests (right) in the first two experiments

Number of standard clients	**Avg Time no path ext.**	**Avg Time with path ext.**
1	387.7 μsec	39 μsec
16	18.0 msec	46.6 μsec
32	32.6 msec	47 μsec
64	50.1 msec	53 μsec
96	57.5 msec	58 μsec

Table 2: Average time elapsed between when a server task unblocks and when it runs, for preferred requests and large files only

Table 2 shows the reason why. An Apache server task that is sending a large file will repeatedly fill up the socket send buffer and block. When the send buffer opens and the task is unblocked, some time elapses before it is scheduled to run again. Since Linux tries to schedule all processes fairly, this time increases proportionally to the total number of processes in the system. In this case the Linux scheduler dominates the response time for preferred requests. For small files, this delay was zero in all experiments because the entire file fit in the send buffer.

4.3 Extending the Path

An *extended path* is SILK's abstraction for coscheduling applications and paths. In this section, we redo the experiment of the previous section while enabling extended paths. This time, SILK changes the realtime priority of a Linux task to reflect the (SILK) priority of the path that the Linux task "extends". In the current setup, this means that a server task servicing a preferred request runs at realtime priority 2, a process waiting on the listen queue has realtime priority 1, and a process servicing a standard request is given realtime priority 0. We

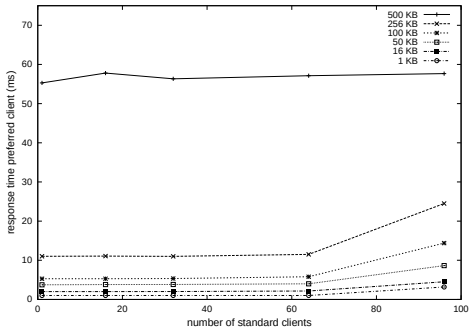

Figure 6: Response time with path extensions

expect that this will remove the Linux scheduler as the bottleneck for providing better service to large preferred requests.

The results are shown in Figure 6. The response times are very low even for the larger files. For example, with 96 standard clients the response time for a 250 KB preferred request has decreased from about one second in the previous experiment to less than 25 milliseconds. In fact, though the numbers in the figures are averages, the maximum response time seen by a preferred client is quite close to the average response time. For 96 clients and the 500KB file the average response time was 58 ms while the maximum was 98 ms. Also, the response times are almost independent of the number of competing standard clients. There is a small increase in the response time with 96 competing standard clients, and the main reason for this is that the Linux interrupt handler has higher priority than SILK. [12] The third column of Table 2 shows the average elapsed time that an unblocked "preferred" server task [13] waits to run with extended paths. Note that there is a difference of three orders magnitude compared to not using path extension with 96 standard clients. This is the reason for the improvements shown in Figure 6.

Table 3 shows the overall system throughput of SILK and Apache, with 96 standard clients for each of the three experiments detailed in this section. Recall that the first experiment had only standard clients, while the other two had 96 standard and one preferred client. We can see from the table that introducing priorities and path extension into SILK has a minor impact on the system performance. The throughput was slightly worse in the second experiment than in the first, and this is probably because in the second experiment we "penalized" the standard clients by lowering their priority from 1 (in the first experiment) to 0. The third experiment with extended paths shows overall higher throughput

[12] Another noticeable fact is that, based on the results for other file sizes, we would expect the response time for the 500KB file to be lower. From our detailed timing log we could see that this result was due to an interaction between the server's TCP implementation and the client's Linux TCP, and was not caused by any scheduling overhead.

[13] In other words, a server task servicing a preferred request.

File size (KB)	Exp 1	Exp 2	Exp 3
1	3041.1	2752.8	2880.5
16	1084.4	974.5	992.7
50	463.2	434.7	440.35
100	234	215.2	221
250	103.8	103.5	103.9
500	54.1	53.2	54.2

Table 3: Throughput in conn/sec with 96 standard clients in the three experiments

compared to the second, and for large file sizes the throughput is the same as in the first experiment.

In summary, we have shown that SILK can handle a high request load, give low response times to preferred requests, and all without rewriting Apache.

5 Discussion and Future Work

5.1 Collisions with Linux

Scout has its origins as a standalone operating system. For this reason, both Scout and Linux independently manage resources that are actually shared between them. In order to realize SILK's goal of requiring no changes to Linux, it is necessary for the system administrator to ensure that SILK and Linux do not collide in two areas: CPU scheduling and TCP port space.

The relationship between the SILK and Linux schedulers described in Sections 3.2 and 3.3 assumes that there are no other realtime Linux tasks. If this is not the case, careful thought must be given to the task's interaction with SILK when choosing priorities for it, the SILK kernel task, and tasks scheduled by path extensions. Such a discussion is beyond the scope of this paper.

SILK and Linux cannot automatically coordinate which TCP port numbers each is using without any Linux changes. If SILK chooses to use the same port as Linux (for instance, two Apache processes listening on port 80 with one using SILK sockets) then SILK will grab all of the packets matching that port and Linux will never see them. One solution is to manually partition the port space between SILK and Linux. If a minor change to Linux is permissible, then another solution could be to export the port management functions from the Linux kernel, and to modify SILK so that it could use them too.

5.2 System Priorities

Realtime resource kernels minimize the work done in interrupts, for the reason that interrupts run asynchronously and at a higher priority than any task. This

can disrupt the system's ability to offer firm guarantees of timing behavior. Since Linux manages hardware devices for SILK, SILK socket processing can be preempted by Linux interrupts. Therefore, SILK can provide only soft realtime guarantees.

Many of the key ideas of Scout and other recent network architectures stem from the observation that today's operating systems cannot assign different priorities to network processing. The standard Linux kernel is no exception—all network processing occurs in a bottom-half handler that runs at interrupt priority. Packet demultiplexing to SILK sockets occurs in this Linux handler, but packet processing itself runs at the priority of the SILK kernel task. This means that, strictly speaking, Linux network processing runs at a higher priority than SILK sockets which can lead to receive live-lock [17]. However, our expectation is that systems running SILK will either use Scout networking exclusively, or at least minimize the amount of network traffic handled by Linux. A solution that may allow both SILK and Linux networking to coexist is to use polling of the network device; this capability already exists in Scout and we intend to port it to SILK.

5.3 Extending the Path

The extended path mechanism described in Section 3.3 assumes a one-to-one correspondence between Linux processes and SILK sockets. In contrast, recent Web servers such as Flash [22] and Apache 2.0 use a single process to manage multiple connections. Single-process servers with multiple connections must make heavy use of the `select()` system call. We believe that the extended path concept could be used in this environment by limiting the set of paths returned via `select()`. We have not yet implemented and evaluated this mechanism and leave it as future work.

Extended paths provide a framework for implementing application QoS. One interesting question is whether QoS policies reside in the application or in SILK itself. To allow applications to take advantage of special features in SILK, we are defining an API for configuring paths using the `setsockopt()` interface. For example, QoS-aware applications can specify path priorities and request extended paths through this interface. On the other hand, since SILK is a modular and configurable system, it can include application-specific policy modules for providing QoS to "dumb" legacy applications. We are also building a SILK-based QoS kernel module for Apache as a demonstration of this capability. We note that SILK provides flexibility as to where in the system we implement policy intelligence.

6 Related Work

SILK provides a network architecture for Linux based on Scout paths. Lazy Receiver Processing [11] incorporates many of the same ideas as Scout, includ-

ing early demultiplexing and protocol processing at the priority of the receiving application. A difference is that LRP waits to process the packet until the receiver requests it, which in turn depends on when the system runs the receiving process; in contrast, through path extension SILK can implement high-level QoS by scheduling the receiver itself to run. In conjunction with LRP, Resource Containers [5] allow considerable flexibility in accounting for all system resources, including kernel processing, used on behalf of an activity. Resource Containers can be associated with multiple processes or network connections based on the structure of the application. The Scout path abstraction encapsulates a single data flow and is less general than a Resource Container. However, resource containers themselves are just an accounting mechanism, while Scout paths combine a number of ideas embracing aspects of both Resource Containers and LRP. Additionally, unlike SILK, LRP (implemented in Sun OS) and Resource Containers (originally implemented in FreeBSD and, more recently, Linux [1]) require significant changes to the OS kernel.

The coordination of application and kernel scheduling underlying the path extension concept has a long history, mainly with a focus on multimedia. Processor reserves [16] can be used to provide QoS for multimedia applications in a microkernel environment. Client applications make CPU reservations which are then guaranteed by the system, and work done by a server on a client's behalf is accounted to the client. In [14], Jeffay *et al.* propose early packet demultiplexing along with coordinated proportional share scheduling of both packet processing and user tasks to avoid receive livelock in overload. We believe that SILK configured with path extensions and WFQ scheduling very closely resembles their scheme.

Several research efforts have focused on building new operating systems around abstractions that support QoS for multimedia applications. Operating systems such as Nemesis [13] and Rialto [15] can provide finer-grained QoS guarantees to applications. However, systems that are built from the ground up often suffer from a lack of real applications which prevents their wider adoption.

Another approach is to modify existing operating systems to provide QoS to applications. Bruno *et al.* have implemented the Eclipse operating system into FreeBSD and support hierarchical proportional-share CPU, disk and link schedulers [10]. QLinux [12] incorporates hierarchical schedulers for CPU and network, LRP and an advanced disk scheduling framework. Both systems depend on particular versions of the hosting operating system.

Linux/RK [21] is an adaptation of the resource kernel concept, developed in RT-Mach, to Linux. A resource kernel provides applications with explicit guarantees to system resources through abstractions such as CPU Reserves. Linux/RK shares some goals with SILK, including modularity and minimal changes to Linux. Linux/RK also maps its own scheduling policies to Linux task priorities like SILK does; their experience may be relevant to SILK as we create path extension mechanisms for schedulers other than fixed priority. However, SILK is not a resource kernel, but rather a networking subsystem that supports coordinated network and application processing.

Scheduler Activations [3] address problems with multiplexing user-level threads onto kernel threads. Most of these problems stem from poor coordination of thread scheduling between the user and kernel domains. SILK sidesteps these issues by multiplexing its threads onto a Linux kernel task and then controlling how this task itself is scheduled. Since the task runs nonpreemptively, Linux cannot choose to take control away from SILK; since the task has the highest priority, Linux must always run this task when it is runnable. The extended path concept avoids priority inversions by further coordinating scheduling across the SILK and Linux schedulers.

A few papers in the area of Web server QoS deserve mention. *WebQoS* [9] is a middleware that provides service differentiation and admission control. Reumann *et al.* [24] have presented virtual services, a new operating system abstraction that provides resource partitioning and management. This approach relies on some kernel modifications. Almeida *et al.* [2] use priority-based schemes to provide differentiated levels of service to clients depending on the Web pages accessed. While in their approach the Web server determines request priorities, Voigt *et al.* [26] provide mechanisms for QoS and overload protection that reside in the kernel and can be applied without context-switching to user level. Our approach goes even further than the architectures described above since we can apply scheduling policies already during the TCP connection setup.

7 Conclusions

SILK is a replacement networking subsystem for Linux based on Scout paths. "Extended paths" provide a QoS framework through coscheduling applications and paths. We have demonstrated that the performance of SILK is comparable with Linux for the Apache Web server, and shown how extended paths can be used to provide differentiated service for Web requests. SILK can serve as a platform for research in network protocols, resource management, CPU scheduling, and QoS policies. Finally, it works with an unmodified Linux 2.4 kernel and existing applications, and hence provides a vehicle for distributing research solutions so that they can be widely used and evaluated. The SILK code will be freely available and is scheduled for release in early 2002.

References

[1] M. Alicherry and K. Gopinath. Predictable management of system resources for linux. In *Proc. of Usenix Annual Technical Conference*, Boston, MA, USA, June 2001.

[2] 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*, March 1999.

[3] Tom Anderson, Brian Bershad, Edward Lazowska, and Henry Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[4] G. Banga and P. Druschel. Measuring the capacity of a web server. In *Proc. of USITS*, December 1997.

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

[6] G. Banga and J. Mogul. Scalable kernel performance for internet servers under realistic loads. In *Proc. of Usenix Annual Technical Conference*, New Orleans, LA, USA, June 1998.

[7] Paul Barford. Web server performance analysis. Tutorial at ACM SIGMETRICS, May 1999.

[8] N. Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. In *9th International World Wide Web Conference*, Amsterdam, May 2000.

[9] Nina Bhatti and Rich Friedrich. Web server support for tiered services. *IEEE Network*, September 1999.

[10] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Retrofitting quality of service into a time-sharing operating system. In *Proc. of Usenix Annual Technical Conference*, Monterey, CA, USA, June 1999.

[11] P. Druschel and G. Banga. Lazy receiver processing (LRP): a network subsystem architecture for server systems. In *Proc. of OSDI*, pages 91–105, October 1996.

[12] P. Goyal, J. Sahni, P. Shenoy, R. Srinivasan, H. Vin, and T. Vishwanath. Qlinux. http://www.cs.umass.edu/ lass/software/qlinux/.

[13] 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*, 14(7):1280–1297, September 1996.

[14] K. Jeffay, F. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *Real-Time Technology and Application Symposium*, pages 480–491, December 2–4, 1998.

[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*, pages 249–256, September 1996.

[16] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE Int. Conference on Multimedia Computing and Systems*, May 1994.

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

[18] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proc. of OSDI*, pages 153–167, October 1996.

[19] E. Nahum, T. Barzilai, and D. Kandlur. Performance issues in WWW servers. In *Proc. of ACM SIGMETRICS*, May 1999.

[20] A. Nakao, A. Bavier, and L. Peterson. Constructing end-to-end paths for playing media objects. In *The Fourth IEEE Conference on Open Architectures and Network Programming*, April 2001.

[21] S. Oikawa and R. Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Real-Time Technology and Application Symposium*, June 1999.

[22] V. Paj, P. Druschel, and W. Zwaenepoel. Flash: an efficient and portable web server. In *Proc. of Usenix Annual Technical Conference*, June 1999.

[23] X. Qie, A. Bavier, L. Peterson, and S. Karlin. Scheduling computations on a software-based router. In *Proc. of ACM Sigmetrics*, June 2001.

[24] 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.

[25] O. Spatscheck and L. Peterson. Defending against denial of service attacks in scout. In *Proc. of OSDI*, February 1999.

[26] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proc. of Usenix Annual Technical Conference*, Boston, MA, USA, June 2001.

Department of Computer Systems

Dissertation Series

85/03 Joachim Parrow, *Fairness Properties in Process Algebra*
87/09 Bengt Jonsson, *Compositional Verification of Distributed Systems*
90/21 Parosh A. Abdulla, *Decision Problems in Systolic Circuit Verification*
90/22 Ivan Christoff, *Testing Equivalences for Probabilistic Processes*
91/27 Hans A. Hansson, *Time and Probability in Formal Design of Distributed Systems*
91/31 Peter Sjödin, *From LOTOS Specifications to Distributed Implementations*
93/37 Linda Christoff, *Specification and Verification Methods for Probabilistic Processes*
93/40 Mats Björkman, *Architectures for High Performance Communication*
94/46 Fredrik Orava, *On the Formal Analysis of Telecommunication Protocols*
96/70 Lars Björnfot, *Specification and Implementation of Distributed Real-Time Systems for Embedded Applications*
97/80 Bengt Ahlgren, *Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption*
98/98 Björn Victor, *The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes*
98/100 Ernst Nordström, *Markov Decision Problems in ATM Traffic Control*
99/101 Paul Pettersson, *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice*
99/110 Mats Kindahl, *Verification of Infinite-State Systems: Decision Control and Efficient Algorithms*
00/114 Kristina Lundqvist, *Distributed Computing and Safety Critical Systems in Ada*
00/115 Jan Gustafsson, *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*
00/116 Jakob Carlström, *Reinforcement Learning for Admission Control and Routing*
00/117 Mikael Sjödin, *Predictable High-Speed Communications for Distributed Real-Time Systems*
01/118 Björn Knutsson, *Architectures for Application Transparent Proxies: A Study of Network Enhancing Software*
02/119 Thiemo Voigt, *Architectures for Service Differentiation in Overloaded Internet Servers*

Swedish Institute of Computer Science

SICS Dissertation Series

01. Bogumil Hausman, *Pruning and Speculative Work in OR-Parallel PROLOG*, 1990
02. Mats Carlsson, *Design and Implementation of an OR Parallel Prolog Engine*, 1990
03. Nabiel A. Elshiewy, *Robust Coordinated Reactive Computing in SANDRA*, 1990
04. Dan Sahlin, *An Automatic Partial Evaluator for Full Prolog*, 1991
05. Hans A. Hansson, *Time and Probability in Formal Design of Distributed Systems*, 1991
06. Peter Sjödin, *From LOTOS Specifications to Distributed Implementations*, 1991
07. Roland Karlsson, *A High Performance OR-parallel PROLOG System*, 1992
08. Erik Hagersten, *Towards Scalable Cache Only Memory Architectures*, 1992
09. Lars-Henrik Eriksson, *Finitary Partial Inductive Definitions and General Logic*, 1993
10. Mats Björkman, *Architectures for High Performance Communication*, 1993
11. Stephen Pink, *Measurement, Implementation and Optimization of Internet Protocols*, 1993
12. Martin Aronsson, *GCLA: The Design, Use, and Implementation of a Program Development System*, 1993
13. Christer Samuelsson, *Fast Natural-Language Parsing Using Explanation-Based Learning*, 1994
14. Sverker Jansson, *AKL—A Multiparadigm Programming Language*, 1994
15. Fredrik Orava, *On the Formal Analysis of Telecommunication Protocols*, 1994
16. Torbjörn Keisu, *Tree Constraints*, 1994
17. Olof Hagsand, *Computer and Communication Support for Interactive Distributed Applications*, 1995
18. Björn Carlsson, *Compiling and Executing Finite Domain Constraints*, 1995
19. Per Kreuger, *Computational Issues in Calculi of Partial Inductive Definitions*, 1995
20. Annika Wærn, *Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction*, 1996
22. Klas Orsvärn, *Knowledge Modelling with Libraries of Task Decomposition Methods*, 1996
23. Kristina Höök, *A Glass Box Approach to Adaptive Hypermedia*, 1996
24. Bengt Ahlgren, *Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption*, 1997
25. Johan Montelius, *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*, 1997
26. Jussi Karlgren, *Stylistic Experiments in Information Retrieval*, 2000
27. Ashley Saulsbury, *Attacking Latency Bottlenecks in Distributed Shared Memory Systems*, 1999

28. Kristian Simsarian, *Toward Human-Robot Collaboration*, 2000
29. Lars-Åke Fredlund, *A Framework for Reasoning about Erlang Code*, 2001
30. Thiemo Voigt, *Architectures for Service Differentiation in Overloaded Internet Servers*, 2002