Adaptive Packet Filters

John Reumann Hani Jamjoom Kang Shin
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122

Abstract— Adaptive Packet Filters (APFs) are motivated by the proliferation of distributed servers and the lack of Quality-of-Service (QoS) management solutions for them. APFs merge packet-filtering and server load monitoring into a novel load-sensitive packet-filtering abstraction for overload protection and QoS differentiation. They integrate well into network protocol stacks and firewalls, scale to large server farms while remaining completely transparent to the applications. Experimental results of our prototype implementation demonstrate APFs' efficacy in providing QoS differentiation and overload protection with minimal overheads.

Keywords—Quality of Service, Packet Filters, Overload Protection.

I. Introduction

SUDDEN and drastic changes in demand, content, and service offerings are characteristic of the Internet environment. For many web-based Internet services, peak demands can be 100 times greater than average load [1]. The Quality-of-Service (QoS) of servers that are not able respond to extreme load seriously suffers under overload — many collapse. Such failure is unacceptable to clients who are increasingly dependent on the availability of Internet servers. Over-design alone does not adequately address this problem.

Realizing the need for higher server availability and greater capacity at a reasonable price, modern Internet servers are implemented as server clusters. Nevertheless, the fundamental problem of performance failure under heavy loads has not disappeared, mainly due to the fast growth of demand for services. What makes matters even worse is that previous, end-host-based QoS architectures [2–4] are not applicable to server farm deployments for three reasons. First, they were designed for single-server scenarios. Second, since QoS is enforced via strict, static resource bindings for applications, these solutions need constant reconfiguration to accommodate changing workload characteristics. Finally, the large number of OS changes required for their implementation seriously hampers their deployment in multi-OS server farms.

Adaptive Packet Filters (APFs) provide overload protection and QoS differentiation for today's Internet servers. They require neither OS changes nor difficult offline capacity analysis. Furthermore, APFs are easily integrated into networking protocols, firewalls, or Layer-3+ load-balancers that connect servers to the Internet. APFs reduce the need for heavily over-designed servers since the request flow to the server will be matched to its capacity. Thus, network servers can be upgraded gradually.

APFs can be seen as an enhancement of the classic network-based QoS-management approaches [5, 6] by allowing arbitrary load-inputs — not just local queue lengths — to affect traffic policing. Alternatively, one may view APFs as an extension to firewalling [7, 8] since they, too, can be configured to enforce arbitrary packet-filtering policies. What distinguishes APFs from traditional firewalls is that they respond to configurable load inputs by dynamically enforcing more or less restrictive packet filters.

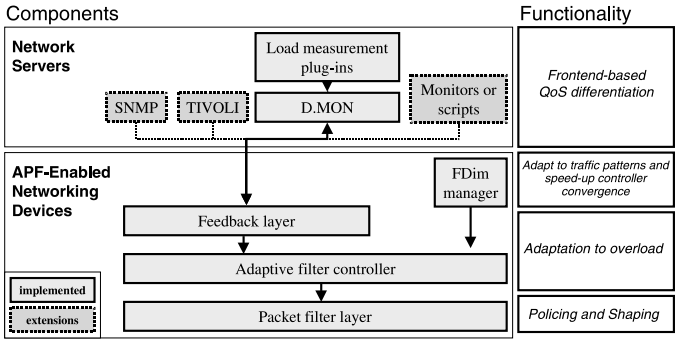

Fig. 1. **Layers of the APF Architecture**

This paper is organized as follows: Section II describes the basic APF abstractions. Section III discusses our APF prototype. Section IV evaluates its response to overload. We discuss related work in Section V. The paper ends with concluding remarks in Section VI.

II. Adaptive Packet Filtering

Overload in network servers is usually caused by an uncontrolled influx of requests from network clients. Since these requests for service are received as network packets, server load can be managed by controlling the influx of certain network packets, e.g., connection requests. APFs take advantage of this insight.

An APF allows alternative packet filters to be loaded into a *filter controller* and accepts feedback from load monitors. Depending on the configured filters and load indicators, an APF automatically enforces filters that avoid overload. APFs do not prescribe the use of any specific monitoring mechanism and can be integrated with various sources of monitoring information, e.g., SNMP, Tivoli, etc. (see Fig. 1).

A. Rules, Filters, and Filtering Dimensions

Network-based QoS control and differentiation requires basic packet classification where each packet, once classified, will be treated according to a policy associated with its traffic class. Traffic classes are defined by information contained in the packets' IP headers.

- Server-side applications via IP destination information.
- Client populations via IP source address prefixes.
- Packet types, such as, TCP, TCP-SYN, UDP, IPX, and ICMP.

- DiffServ code points.
- A combination of the above.

The APF notion of traffic classes is used in firewall configuration and was proposed by Mogul *et al.* [7]. A *rule* combines traffic class and policy. The policy specifies whether packets of the traffic class should be accepted, dropped, shaped, or policed to a particular rate. If multiple rules apply to an incoming packet, the most restrictive policy will be applied to the packet. This definition of rules is consistent with that of most modern firewall implementations.

Each individual rule defines QoS requirements for exactly one traffic class. Thus, QoS differentiation between competing traffic classes can only be achieved by installing rule combinations. Rule combinations are expressed in *filters*. For example, a filter of two rules could be set up to admit twice the packet rate from IP address X as from IP address Y. To guarantee QoS differentiation, the rules of a filter are considered a unit, i.e., all or none of its rules are enforced.

To make filters applicable to environments of unknown or variable server processing capacities, it is necessary to implement *filter adaptation*. Fixed filters with inadequate shaping rates could easily cause underutilization of network servers. For example, if one restricts incoming traffic to a moderately-loaded Internet server, one risks dropping requests, which the server could have handled easily. However, installing only permissive filters to accommodate the common (lightly-loaded) case would be a false conclusion. Permissive filters fail to defend the server from load surges.

Filter adaptation is achieved in *filtering dimensions* (FDims). Each FDim is a linear list of filters, $f_1, f_2, \ldots, f_n$, only one of which is enforced at any given time. A switch from one filter to another is an *atomic* operation, meaning that the old filter is uninstalled and the new filter is installed without processing any network packets in between. Fig. 2 shows an expanded view of a FDim. The filter selection process is driven by a FDim-specific load index (load var in Figure 1), which is fed by external load-monitoring software.

To provide effective overload protection, each FDim configuration must satisfy the following constraints:

- filters are ordered by increasing restrictiveness,
- the least restrictive filter does not police incoming traffic, and
- the most restrictive filter drops all incoming traffic.

Multiple FDims may be installed simultaneously, each tied to its own load variable. This pays tribute to the fact that different types of overload may be caused by different network services, which must be policed separately. Support for multiple FDims is particularly useful when an APF-enabled device controls access to different services, each of which is located on its own separate backend server.

APFs are conceptually similar to CBQ [5]. There are, however, three key differences between the two. First, APFs are inbound controls. Therefore, they remain effective QoS controls even if they are only installed on the network servers. Second, APFs are not tied to any particular link scheduling or packet scheduling architecture. Finally, APFs may police incoming packets based on load measurements other than just link utilization. CBQ cannot adapt incoming packet rates to match the network servers' capacities.

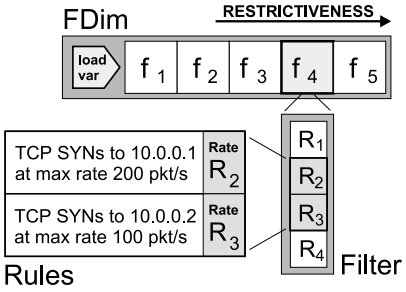

Fig. 2. **Relationship between rules, filters, and FDim**

B. Adaptation to Overload

In the APF framework, QoS differentiation and overload defense depend on the controller's ability to determine the best filter for incoming traffic. The following control mechanism automatically determines the least restrictive filter in each FDim that prevents their respective monitoring variables from reporting overload. This optimizes overall server throughput subject to the "no-overload" constraint.

1. Load indices are integral numbers between 0 and 100, with 0 representing no load and 100 representing highest load.
2. APF associates user-defined thresholds with each load index as to what constitutes an overload and an underload.
3. When an overload is detected on some load index, v_i, the next most restrictive filter for its corresponding FDim, F_i, is applied. If this change triggers the application of the most restrictive filter for that FDim, the FDim is flagged as severely overloaded potentially requiring network administrator intervention.
4. When an underload is detected on load variable, v_i, the next less restrictive filter for its corresponding FDim, F_i, is applied.
5. APFs are configured with a configurable inter-switching delay, s_i, for each FDim, F_i to stabilize its adaptation behavior.
6. APFs keep track of how long each filter is applied against incoming load. This is an indication of a filter's effectiveness, and therefore, important for FDim reconfiguration.

APF eventually begins to oscillate between the application of a few adjacent filters because the policies of one filter are too restrictive, and too permissive in the other. This behavior is a natural consequence of the controller's design and is non-problematic unless the controller oscillates between vastly different filters. In this case, the FDim is configured too coarse-grained and needs to be refined. We tackle this problem using an automated FDims generation and reconfiguration tool [9], which is beyond the scope of this paper.

III. Prototype Implementation

The APF prototype is implemented as an extension module, called *QGuard*, for the Linux 2.2.14 kernel's firewalling layer `ipchains`. `Ipchains` provides efficient packet header matching functionality and simple packet rejection policies. We implemented additional support for traffic shaping. The recently released Linux 2.4.x firewalling support, *netfilters*, already im-

plements this policy. Our prototype implements the well-known *token bucket* [10, 11] rate-control scheme by associating each traffic class with a counter of remaining tokens, a bucket volume, a timestamp for the last token replenishment time, and a per-second token rate. These variables are part of the firewalling *rule* data structure. We refer to work by Mehra *et al*. [11] for a detailed analysis of different configurations of traffic shaping and smoothing solutions and their impact on client-perceived server performance.

As soon as a FDim is created inside the kernel, user-space scripts may start reporting load for it. This, too, is done via a special system call. Less invasive configuration mechanisms like a configuration file, the so-called `/proc` file system, or SNMP MIBs could have been used instead. However, this detail does not discount our prototype as a proof-of-concept.

To avoid any conflicts between adaptive filters and firewalling rules that warrant network security, dynamic QGuard filters are always the last to be enforced. Thus, a QGuard-enabled system will never admit any packets that are to be rejected for security reasons. Our particular implementation achieves this goal by linking the QGuard firewalling chain as the last rule chain into `ipchain`'s input chain list.

The QGuard implementation relies on a distributed user-space monitor to obtain its load information (see D.MON in Fig. 1). It consists of monitors on the backend servers and a collector component, which resides on the AFP-enabled device. Whenever the load on the backend servers changes, the collector receives load updates, which it feeds through to the QGuard kernel module. A configuration file specifies the association between load indices and FDims. This flexible monitoring architecture allows for simple reconfiguration of the basic adaptation mechanism. Hence, we were able to explore a variety of different load indicators (e.g. paging rate, disk access rate, idle time, bandwidth consumption, packet arrival rate, and TCP timeout rate), their relationship to overload, and their usability in QGuard-based overload defense.

As mentioned in our discussion of the APF design, the filter switching delay, s, is an important design parameter. We found that s can be derived directly if load indices are computed as averages over a base interval. The value of s should simply be set to the duration of the base interval.

IV. Evaluation

A. Experimental Setup

We studied the performance of APF using the QGuard prototype in two key deployment scenarios: APF-enabled (standalone) servers and APF-enabled frontends. An Intel Pentium-based PC (450 MHz, 192 MB memory) acts as a server hosting a synthetic load generator with the following configurable load components:

CPU: Executes a busy cycle of uniformly-distributed length.
File system: Access random blocks in a large file.
Network: Returns response messages of configurable size.
Memory: Allocates a configurable amount of memory and touch one byte on every page.

TABLE I
Basic overhead: aggregated performance drop vs. the number of rules per filter (FDim size 20 filters).

APF Deployment		Number of Rules					
		0	2	16	64	256	512
APF-enabled servers	% Drop	0%	6%	11%	11%	14%	27%
APF-enabled frontend	% Drop	0%	0%	0%	0%	0%	0%

Load generated in response to a received request can be configured on a per-port basis to include different amounts of CPU, file system, network, and memory components.

The client machines, two other Pentium-class machines, connect to the server through a Fast-Ethernet switch. They execute a request generator that simulates an arbitrary number of concurrent clients to the load generating server. Conceptually, our request generator follows the design of SURGE [12] and HTTPerf [13]:

- It maintains a given request rate regardless of the server's processing capabilities. This allows the simulation of severe overload conditions.
- The clients are hosted on multiple hosts, allowing us to study QGuard's ability to differentiate between clients who access the server from different subnets.
- Clients' arrival rates can be configured on a per-server-port basis. This, for example, allows setting the arrival rate for memory-intensive requests to a lower rate than CPU-intensive ones.

We configured client requests' inter-arrival times to follow a Poisson-distribution. Unless stated otherwise, QGuard was configured with one FDim of 20 filters. This FDim's load index was tied to the server's CPU load. A baseline comparing the QGuard-controlled server's performance against the server without QGuard was established in all our tests. Because of space considerations, we present a limited evaluation. A more complete evaluation can be found in [9].

B. Basic Overhead

APF overhead depends on how often filters are switched and on the number of rules per filter. We determined that the delay between filter switches has only little impact on the aggregated throughput of a QGuard-protected server. However, the number of rules per filter can affect performance, since the APF-enabled device must check each incoming packet against its rule-base. Table I shows that QGuard in standalone configuration suffers a performance drop when the number of rules becomes large. However, this penalty disappears when QGuard is configured on a frontend device.

C. Overload Protection and QoS Differentiation

QGuard is designed to protect servers from overload and degrade QoS gracefully during overload. Overload protection was tested by subjecting our server to various overload conditions while monitoring the overall health of the server. In extreme cases, such as heavy memory swapping, the unprotected server

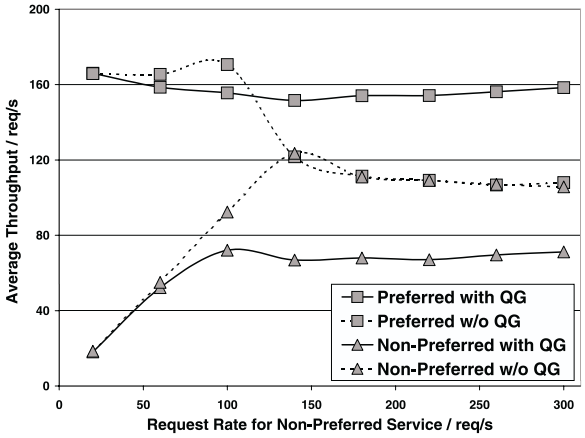

Fig. 3. **Throughput differentiation**

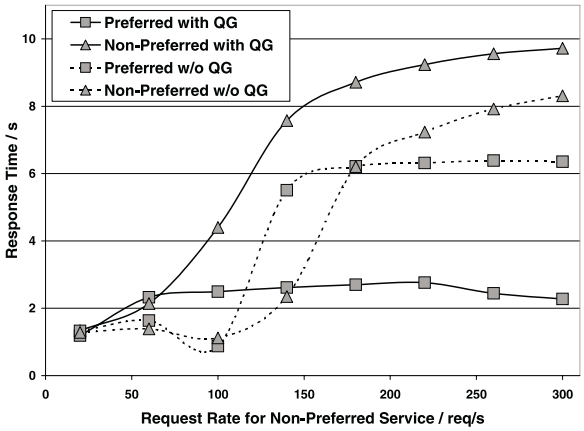

Fig. 4. **Response time differentiation**

crashed, whereas QGuard avoided the crash.

We validated QoS differentiation in two different scenarios: (a) the differential treatment of co-hosted services by performing destination port-based differentiation and (b) the differentiation based on clients' IP addresses. Each filter was configured to deliver up to three times more throughput to the preferred service than to the non-preferred service. As Figures 3 and 4 show, QGuard provides QoS differentiation between the preferred and the non-preferred services. The throughput and response time of the preferred service remain stable, regardless of the increase in the number of clients for the non-preferred service. This compares favorably with a 50% drop in throughput and a twofold increase in response time for clients of the preferred service without QGuard. Similar QoS differentiation can be achieved when traffic classes represent client groups instead of services.

D. Responsiveness to Overload

Not only is it important to show that APF is capable of providing QoS differentiation, but one also wants to show how quickly it responds to overload. The following experiment subjected the QGuard-protected server to a sudden request surge and studied how quickly it restores preferred clients' throughput to 160 $\mathrm{reqs/s}$. The experiment highlights the importance of the size of the averaging history window that is used to compute server load averages.

Initially, only one client class, the preferred clients, request service from the server. Because the server can handle all requests easily, QGuard does not throttle incoming traffic at all until non-preferred clients cause a heavy load surge — more than twice the server's capacity — after time 60s.

Fig. 5 shows that QGuard's reaction to this overload depends on the length of the sliding history window over which the load index is computed — longer histories imply slower adaptation. As expected, short histories produce fast response to overload. However, if histories are reduced too much, load indices become unstable and cause an unpredictable response (not shown in the graph).

E. Limitations of Adaptive Packet Filters

The most crucial limitation of APFs relates to disparate per-request work requirements of the controlled traffic classes (see Fig. 6). We observed that QGuard's ability to distinguish the preferred service from the non-preferred service is degraded when the amount of work required to serve one request of the non-preferred service is orders of magnitude larger than that required to serve one request of the preferred service. Although QGuard does the best it can to provide QoS differentiation, controlling workload on a per-packet basis is too coarse-grained when each packet imposes large amounts of work. We conclude that APF-based solutions should not be used to police long-lived, resource-intensive service requests. In this context, fine-grained OS resource reservation mechanisms like Resource Containers, Scout, or Virtual Services [4, 14, 15] are needed. Fortunately, most network-based services — FTP, HTTP, SMTP, etc. — handle requests at high rates, thus permitting APF-based QoS control in most cases.

V. Related Work

The research presented in this paper extends previous ideas of packet-filtering [7] and links them to QoS-management. The QoS-management concepts related to our work fall into three main categories: network-based QoS differentiation [5, 6, 16], adaptive middleware [17], and OS-level resource reservation [2–4, 15, 18]. Because of space limitation, we only discuss network-based concepts.

Network-based QoS management solutions for Internet servers, receive a fair amount of commercial attention since they are completely transparent to both applications and server OSs, inherently scalable, and easily added to existing server infrastructure. These solutions allow static rate definitions for different traffic classes. Extreme's *ExtremeWare* [16], for example, allows the definition of rate limits for different flows (based on packet source and/or destination). These approaches do not allow the integration of arbitrary monitoring inputs with packet-filtering policies. They only provide static rate control. In contrast, APFs allow administrators to define a dynamic overload response for their systems.

APFs may also be viewed as a descendant of traffic-flow

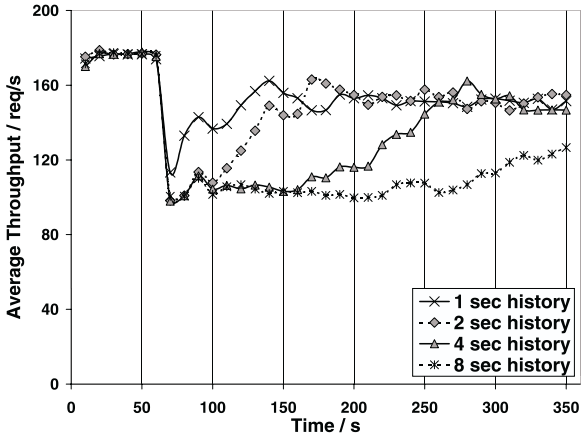

Fig. 5. **History length vs. rate of convergence**

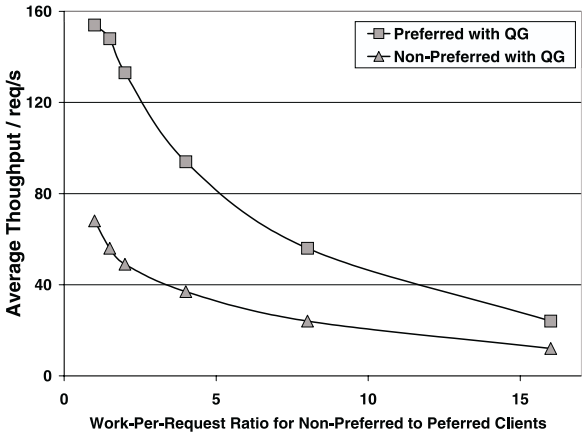

Fig. 6. **Achieved QoS differentiation under increasing per-request work for the non-preferred service**

policing approaches, such as CBQ [5], Network Hoses [6], and Adaptive Packet Marking [19]. These approaches target QoS differentiation between competing traffic classes in the Internet. They, like more recent work on DiffServ [20], share the assumption that network bandwidth is the main service bottleneck and that policy-enabled queuing at the routers will provide end-to-end QoS differentiation. We do not question this argument's validity with respect to long-lived, session-based services, e.g., digital media streaming. However, in many Internet service scenarios, server load is the most important QoS factor. To manage QoS effectively, server load must be taken into account, as is done in our APF abstraction.

VI. Concluding Remarks

The generic nature of the APF abstraction is the result of three major design choices. First, the dynamic adaptation mechanism allows APF deployment in clusters of unknown request processing capacity. Second, the configurable monitoring-feedback mechanism makes its use against a variety of adverse server conditions possible. Third, network-orientation make application and OS modifications unnecessary, thus facilitating APF use in arbitrary service environments. These benefits clearly justify further research on APFs and should encourage its use in OSs and network infrastructure devices.

Since most current OS kernels implement IP-based packet-filtering, APFs can be added to them easily. Furthermore, firewalling devices, edge-routers, and IP-layer switches, and anything else connecting servers to the Internet are primary candidates for APF integration. As we have shown in our experiments, an APF-enabled frontend does not affect server performance, yet it provides QoS differentiation.

References

[1] Keynote Systems, "Web Robustness Measurement: The Future May be Now," http://www.keynotesytemes.com/services/assets/applets/ZMR37/Web_Robustnes.pdf, 2000.

[2] Gaurav Banga and Peter Druschel, "Lazy Receiver Processing LRP: A Network Subsystem Architecture for Server Systems," in *Second Symposium on Operating Systems Design and Implemenation*, October 1996.

[3] K. Jeffay, F.D. Smith, A. Moorthy, and J.H. Anderson, "Proportional Share Scheduling of Operating System Services for Real-Time Applications," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, December 1998.

[4] Oliver Spatscheck and Larry L. Peterson, "Defending Against Denial of Service Attacks in Scout," in *Third Symposium on Operating Systems Design and Implemenation*, February 1999, pp. 59–72.

[5] S. Floyd and V. Jacobsen, "Link-Sharing and Resource Management Models for Packet Networks," *Transactions on Networking*, vol. 3, no. 4, pp. 365–386, 1995.

[6] N. G. Duffield, Pawan Goyal, Albert Greenberg, Parho Mishra, K. K. Ramakrishnan, and Jacobus E. van der Merive, "A Flexible Model for Resource Management in Virtual Private Networks," in *Proceedings of ACM SIGCOMM '99*, 1999.

[7] Jeffrey C. Mogul, Richard F. Rashid, and Michael j. Accetta, "The Packet Filter: An Efficient Mechanism for User-Level Network Code," in *Proceedings of the 11th ACM Symposium on Operating Systems Principles and Design*. ACM, November 1987.

[8] Paul Russell, "IPCHAINS-HOWTO," http://www.rustcorp.com/linux/ipchains/HOWTO.html.

[9] Hani Jamjoom, John Reumann, and Kang Shin, "QGuard: Protecting Internet Servers from Overload," Tech. Rep. CSE-TR-427-00, University of Michigan Technical Report, 2000.

[10] S. Keshav, *An Engineering Approach to Computer Networking*, Addison-Wesley Publishing Company, 1997.

[11] Ashish Mehra, Renu Tewari, and Dilip Kandlur, "Design Considerations for Rate Control of Aggregated TCP Connections," in *NOSSDAV*, June 1999.

[12] P. Barford and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," in *In Proceedings of Performance'98/ACM Sigmetrics'98*, May 1998, pp. 151–160.

[13] David Mosberger and Tai Jin, "httperf - A Tool for Measuring Web Server Performance," Tech. Rep. HPL-98-61, HP Labs, 1998.

[14] Gaurav Banga, Peter Druschel, and Jeffrey Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Third Symposium on Operating Systems Design and Implemenation*, February 1999, pp. 45–58.

[15] John Reumann, Ashish Mehra, Kang Shin, and Dilip Kandlur, "Virtual Services: A New Abstraction for Server Consolidation," in *Procedings of the 2000 USENIX Annual Technical Conference*. USENIX, June 2000.

[16] Extreme Networks, Inc., "Policy-Based QoS," http://www.extremenetworks.com/technology/ whitepapers/Policy-basedV5.asp, 2000.

[17] Hewlett Packard, "WebQoS Technical White Paper," http://www.internetsolutions.enterprise.hp.com/ webqos/products/overview/wp.html, 2000.

[18] Jeffrey C. Mogul and K. K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-Driven Kernel," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 217–252, August 1997.

[19] W. Feng, D. Kandlur, D. Saha, and K. Shin, "Adaptive Packet Marking for Providing Differentiated Services in the Internet," Tech. Rep. CSE-TR-347-97, University of Michigan, October 1997.

[20] S. Blake, D. Black, M. Carlson, E. Davis, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services," Tech. Rep. RFC 2475, IETF, December 1998.