QGuard: Protecting Internet Servers from Overload

Hani Jamjoom* John Reumann †
and Kang G. Shin
Department of Electrical Engineering and Computer Science,
The University of Michigan
Ann Arbor, MI 48109
{*jamjoom, reumann*}*@eecs.umich.edu*

Abstract

Current operating systems are not well-equipped to handle sudden load surges that are commonly experienced by Internet servers. This means that service providers and customers may not be able to count on servers being available once their content becomes very popular. Recent Denial-of-Service attacks on major e-commerce sites have capitalized on this weakness.

Remedies that were proposed to improve server behavior under overload require substantial changes to the operating system or applications, which is unacceptable to businesses that only want to use the tried and true. This paper presents *QGuard*, a much less radical solution to providing differential QoS, protection from overload and some DoS attacks. QGuard is an adaptive mechanism that exploits rate controls for inbound traffic in order to fend off overload and provide QoS differentiation between competing traffic classes.

Our Linux-2.2.14-based QGuard prototype provides freely configurable QoS differentiation (preferred customer treatment and service differentiation) and effectively counteracts SYN and ICMP-flood attacks. Since QGuard is a purely network-centric mechanism, it does not require any changes to server applications and can be implemented as a simple add-on module for any OS. Our measurements indicate no performance degradation on lightly loaded servers and only a small reduction of aggregated server throughput (less than 2%) under overload. Well-behaved "preferred customers" remain virtually unaffected by server overload.

*Hani Jamjoom is supported by the Saudi Arabian Ministry of Higher Education

†John Reumann is supported by IBM's Research Fellowship Program

1 Introduction

Recent blackouts of major web sites, such as Yahoo, eBay, and E*Trade, demonstrated how susceptible e-business is to simple Denial-of-Service (DoS) attacks [9, 11]. Using publicly available software, amateur hackers can chose from a variety of attacks such as SYN or ping-floods to lock out paying customers. These attacks either flood the network pipe with traffic or pound the server with requests, thus exhausting precious server resources. In both attack scenarios, the server will appear dead to its paying (or otherwise important) customers.

This problem has been known since the early 1980's [5]. Since then, various fixes have been proposed [4, 17, 23]. Nevertheless, these fixes are only an insufficient answer to the challenges faced by service providers today. What makes things more difficult today is that service providers want to differentiate between their important and less important clients at all times, even while drawing fire from a DoS attack.

The recent DoS attacks are only one instance of poorly managed overload scenarios. A sudden load surge, too, can lead to a significant deterioration of service quality

(QoS) — sometimes coming close to the denial of service. Under such circumstances, important clients' response time may increase drastically. More severe consequences may follow if the amount of work-in-progress causes hard OS resource limits to be violated. If such failures were not considered in the design of the service, the service may crash, thus potentially leading to data loss.

These problems are particularly troubling for sites that offer price-based service differentiation. Depending on how much customers pay for the service, they have different QoS requirements. First of all, paying customers want the system to remain available even when it is heavily loaded. Secondly, higher-paying customers wish to see their work requests take priority over lower-paying customers when resources are scarce. For example, a Web site may offer its content to paying customers as well as free-riders. A natural response to overload is not to serve content to the free-riders. However, this behavior cannot be configured in current server OSs.

Although pure middleware solutions for QoS differentiation [1, 14] exist, they fail when the overload occurs before incoming requests are picked up and managed by the middleware. Moreover, middleware solutions fail when applications bypass the middleware's control mechanisms, e.g., by using their own service-specific communication primitives or simply by binding communication libraries statically. Therefore, much attention has been focused on providing strong performance management mechanisms in the OS and network subsystem [4, 6, 7, 12, 15, 19, 20, 23, 25]. However, these solutions introduce more controls than necessary to manage QoS differentiation and defend the server from overload.

We propose a novel combination of kernel-level and middleware overload protection mechanisms called *QGuard*. QGuard learns the server's request-handling capacity independently and divides this capacity among clients and services according to administrator-specified rules. QGuard's differential treatment of incoming traffic protects servers from overload and immunizes the server against SYN-floods and the so-called "ping-of-death." This allows service providers to increase their capacities gradually as demand grows since their preferred customers' QoS is not at risk. Consequently, there is no need to build up excessive over-capacities in anticipation of transient request spikes. Furthermore, studies on the load patterns observed on Internet servers show that over-capacities can hardly protect servers from overload.

This paper is organized as follows. We present our design rationale in Section 2 and discuss its implementation in Section 3. Section 4 studies QGuard's behavior in a number of typical server overload and attack scenarios. Section 5 places QGugard in the context of related work. The paper ends with concluding remarks in Section 6.

2 What is QGuard?

Internet servers suffer from overload because of the uncontrolled influx of requests from network clients. Since these requests for service are received over the network, controlling the rate at which network packets may enter the server is a powerful means for server load management. QGuard exploits the power of traffic shaping to provide overload protection and differential service for Internet servers. By monitoring server load, QGuard can adapt its traffic shaping policies without any *a priori* capacity analysis or static resource reservation. This is achieved by the cooperation of the four QGuard components: *traffic shaper*, *monitor*, *load-controller*, and *policy-manager* (see Figure 1).

2.1 The Traffic Shaper

QGuard relies on shaping the incoming traffic as its only means of server control. Since QGuard promises QoS differentiation, differential treatment must begin in the traffic shaper, i.e., simply controlling aggregate flow rates is not good enough.

To provide differentiation, the QGuard traffic shaper associates incoming packets with their traffic classes. Traffic classes may represent specific server-side applications (IP destinations or TCP and UDP target ports), client populations (i.e., a set of IP addresses with a common prefix), DiffServ bits, or a combination thereof. Traffic classes should be defined to represent business or outsourcing needs. For example, if one wants to control the request rate to the HTTP service, a traffic class that aggregates

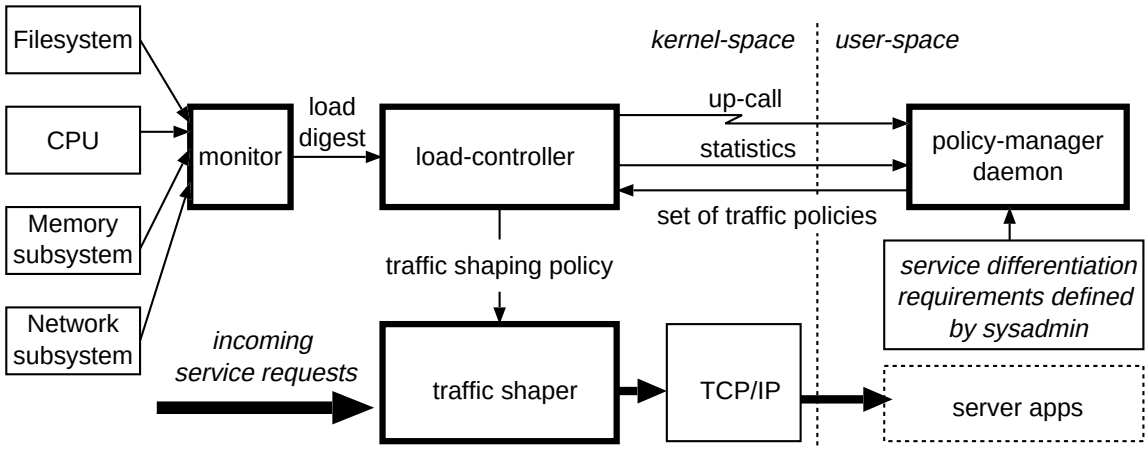

Figure 1: **The QGuard architecture**

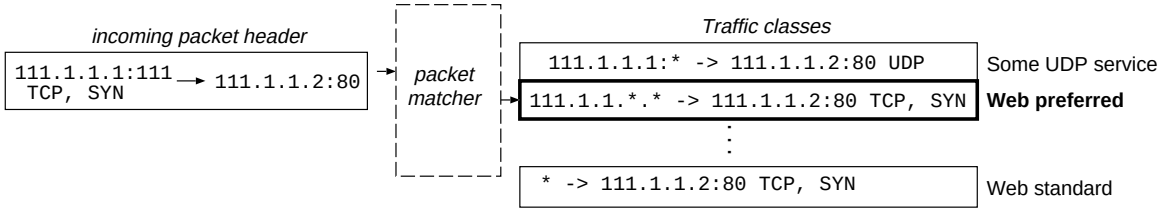

Figure 2: Classifying incoming traffic

all TCP-SYN packets sent to port 80 on the server should be introduced. This notion of traffic classes is commonly used in policy specifications for firewalls and was proposed initially by Mogul *et al.* [18]. Figure 2 displays a sample classification process. Once the traffic class is defined, it may be policed.

For effective traffic management, traffic classification and policing are combined into *rules*. Each rule specifies whether a traffic class' packets should be accepted or dropped. Thus, it is possible to restrict certain IP domains from accessing certain (or all) services on the server while granting access to others without affecting applications or the OS. As far as the client and servers OS's are concerned, certain packets simply get lost. Such all-or-nothing scheme are used for server security (firewalls). However, for load-control more fine-grained traffic control is necessary. Instead of tuning out a traffic source completely, QGuard allows the administrator to limit its packet rate. Thus, preferred clients can be allowed to submit requests at a higher rate than non-preferred ones. Moreover, QGuard also associates a weight representing traffic class priority with each rule. We refer to these prioritized, rate-based rules as *QGuard rules*. QGuard rules accept a specific traffic class' packets as long as their rate does not exceed the maximal rate specified in the rule. Otherwise, a QGuard rule will cause the incoming packets to be dropped.

QGuard rules can be combined to provide differential QoS. For example, the maximal acceptance rate of one traffic class can be set to twice that of another, thus delivering a higher QoS to the clients belonging to the traffic class identified by the rule with the higher acceptance rate. The combination of several QGuard rules — the building block of QoS differentiation — is called a *QGuard filter* (henceforth filter). They may consist of an arbitrary number of rules. Filters are the inbound equivalent of CBQ polices [10].

2.2 The Monitor

Since QGuard does not assume to know the ideal shaping rate for incoming traffic, it must monitor server load to determine it. Online monitoring takes the place of offline system capacity analysis.

The monitor is loaded as an independent kernel-module to sample system statistics. At this time the administrator may indicate the importance of different load-indicators for the assessment of server overload. The monitoring module itself assesses server capacity based on its observations of different load indicators. Accounting for both the importance of all load indicators and the system capacity, the monitor computes the *server load-index*. Other kernel modules may register with the monitor to receive a notification if the load-index falls into a certain range.

Since the monitor drives QGuard's adaptation to overload, it must be executed frequently. Only frequent execution can ensure that it will not miss any sudden load surges. However, it is difficult to say exactly how often it should sample the server's load indicators because the server is subject to many unforeseeable influences [13], e.g., changes in server popularity or content. Therefore, all relevant load indicators should be oversampled significantly. This requires a monitor with very low runtime overheads. The important role of the monitor also requires that it must be impossible to cause the monitor to fail under overload. As a result of these stringent performance requirements we decided that the logical place for the monitor is inside the OS.

2.3 The Load-Controller

The load-controller is an independent kernel module, for similar reasons as the monitor, that registers its overload and underload handlers with the monitor when it is loaded into the kernel. Once loaded, it specifies to the monitor when it wishes to receive an overload or underload notification in terms of the server load-index. Whenever it receives a notification from the monitor it decides whether it is time to react to the observed condition or whether it should wait a little longer until it becomes clear whether the overload or underload condition is persistent.

The load-controller is the core component of QGuard's overload management. This is due to the fact that one does not know in advance to which incoming rate the packets of individual traffic classes should be shaped. Since one filter is not enough to manage server overload, we introduce the concept of a *filter-hierarchy* (FH). A FH is a set of filters ordered by filter restrictiveness (shown in Figure 3). These filter-hierarchies can be loaded into the load-controller on demand. Once loaded, the load-controller will use monitoring input to determine the least restrictive filter that avoids server overload.

The load-controller strictly enforces the filters of the FH, and any QoS differentiation that are coded into the FH in the form of relative traffic class rates will be implemented. This means that QoS-differentiation will be preserved in spite of the load-controllers dynamic filter selection.

Assuming an overloaded server and properly set up FH, i.e.,

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

the load-controller will eventually begin to oscillate between two adjacent filters. This is due the fact that the rate limits specified in one filter are too restrictive and not restrictive enough in the other.

Oscillations between filters are a natural consequence of the load-controller's design. However, switching between filters causes some additional OS overhead. Therefore, it is advantageous to dampen the load-controller's oscillations as it reaches the point where the incoming traffic rate matches the server's request handling capacity. Should the load-controller begin to oscillate between filters of vastly different acceptance rates, the FH is too coarse-grained an should be refined. This is the policy-manager's job. To allow the policy-manager to deal with this problem, the load-controller keeps statistics about its own behavior.

Another anomaly resulting from ineffective filter-hierarchies occurs when the load-controller repeatedly switches to the most restrictive filter. This means that no filter of the FH can contain server load. This can either be the result of a completely misconfigured FH or due

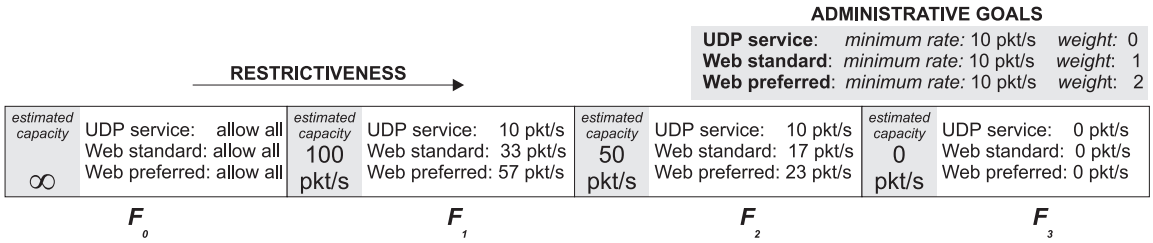

Figure 3: A sample filter-hierarchy

to an attack. Since switching to the most restrictive policy results in a loss of service for all clients, this condition should be reported immediately. For this reason the the load-controller implements an up-call to the policy-manager (see Figure 1). This notification is implemented as a signal.

2.4 The Policy-Manager

The policy-manager fine-tunes filter-hierarchies based on the effectiveness of the current FH. A FH is effective if the load-controller is stable, i.e., the load-controller does not cause additional traffic burstiness. If the load-controller is stable, the policy-manager does not alter the current FH. However, whenever the load-controller becomes unstable, either because system load increases beyond bounds or because the current FH is too coarse-grained, the policy-manager attempts to determine the server's operating point from the oscillations of the load-controller, and reconfigures the load-controller's FH accordingly.

Since the policy-manager focuses the FH with respect to the server's operating point, it is the crucial component to maximizing throughput during times of sustained overload. It creates a new FH with fine-granularity around the operating point, thus reducing the impact of the load-controller's oscillations and adaptation operations.

The policy-manager creates filter-hierarchies in the following manner. The range of all possible acceptance rates that the FH should cover — an approximate range given by the system administrator — is quantized into a fixed number of bins, each of which is represented by a filter. While the initial quantization may be too coarse to provide accurate overload protection, the policy-manager successively zooms into smaller quantization intervals around the operating point. We call the policy-manager's estimate of the operating point the *focal point*. By using non-linear quantization functions around this focal point, accurate, fine-grained control becomes possible. The policy-manager dynamically adjusts its estimate of the focal point as system load or request arrival rates change.

The policy-manager creates filter-hierarchies that are fair in the sense of max-min fair-share resource allocation [16]. This algorithm executes in two stages. In the first stage, it allocates the minimum bandwidth to each rule. It then allocates the remaining bandwidth based on a weighted fair share algorithm. This allocation scheme has two valuable features. First, it guarantees a minimum bandwidth allocation for each traffic class (specified by the administrator). Second, excess bandwidth is shared among traffic classes based on their relative importance (also specified by the administrator). Figure 3 shows an example FH that was created in this manner. This figure shows that the policy-manager makes two exceptions from the max-min fair-share rule. The leftmost filter admits all incoming traffic to eliminate the penalty for the use of traffic shaping on lightly-loaded servers. Furthermore, the rightmost filter drops all incoming traffic to allow the load-controller to drain residual load if too many requests have already been accepted.

There are some situations that cannot be handled using the outlined successive FH refinement mechanism. Such situations often result from DoS attacks. In such cases, the policy-manager attempts to identify ill-behaved traffic classes in the hope that blocking them will end the overload. To identify the ill-behaved traffic class, the policy-manager first denies all incoming requests and admits traffic classes one-by-one on a probational basis (see

Figure 8) in order of their priority. All traffic classes that do not trigger another overload are admitted to the server. Other ill-behaved traffic classes are tuned out for a configurable period of time (typically a very long time).

Since the policy-manager uses floating point arithmetic and reads configurations from the user, it is implemented as a user-space daemon. This also avoids kernel-bloating. This is not a problem because the load controller already ensures that the system will not get locked-up. Hence, the policy-manager will always get a chance to run.

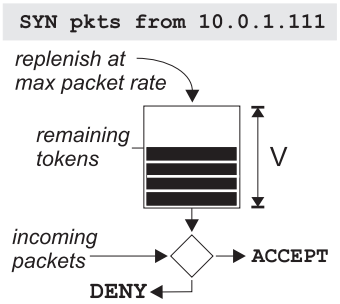

Figure 4: A QGuard Firewall Entry

3 Implementation

3.1 The Traffic Shaper

Linux provides sophisticated traffic management for outbound traffic inside its traffic shaper modules [8]. Among other strategies, these modules implement hierarchical link-sharing [10]. Unfortunately, there is nothing comparable for inbound traffic. The only mechanism offered by Linux for the management of inbound traffic is *IP-Chains* [21] — a firewalling module. To our advantage, the firewalling code is quite efficient and can be modified easily. Furthermore, the concept of matching packet headers to find an applicable rule for the handling of each incoming packet is highly compatible with the notion of a QGuard rule. The only difference between a QGuard's and IP-Chains' rules is the definition of a rate for traffic shaping. Under a rate-limit a packet is considered to be admissible only if the arrival rate of packets that match the same header pattern is lower than the maximal arrival rate.

QGuard rules are fully compatible with conventional firewalling policies. All firewalling policies are enforced before the system checks QGuard rules. This means that the system with QGuard will never admit any packets that are to be rejected for security reasons.

Our traffic shaping implementation follows the well-known *token bucket* [16] rate-control scheme. Each rule is equipped with a counter (`remaining_tokens`), a per-second packet `quota`, and a `timestamp` to record the last token replenishment time. The `remaining_tokens` counter will never exceed $V \times$ `quota` with V representing the bucket's volume. We modified the Linux-based IP-Chains firewalling code as follows. The matching of an incoming packet against a number of packet header patterns for classification purposes (see Figure 2) remains unchanged. At the same time, QGuard looks up the traffic class' `quota`, `timestamp`, and `remaining_tokens` and executes the token bucket algorithm to shape incoming traffic. For instance, it is possible to configure the rate at which incoming TCP-SYN packets from a specific client should be accepted. The following command:

```
qgchains -A qguard --protocol TCP --syn
--destination-port --source 10.0.0.1 -j
RATE 2
```

allows the host `10.0.0.1` to connect to the Web server at a rate of two requests per second. The syntax of this rule matches the syntax of Linux IP-Chains, which we use for traffic classification. We chose packets as our unit of control because we are ultimately interested in controlling the influx of requests. Usually, requests are small and, therefore, sent in a single packet. Moreover, long-lived streams (e.g., FTP) are served well by the packet-rate abstraction, too, because such sessions generally send packets of maximal size. Hence, it is relatively simple to map byte-rates to packet-rates.

3.2 The Monitor

The Linux OS collects numerous statistics about the system state, some of which are good indicators of overload conditions. We have implemented a lightweight monitor-

ing module that links itself into the periodic timer interrupt run queue and processes a subset of Linux's statistics (Table 1). Snapshots of the system are taken at a default rate of 33 Hz. While taking snapshots the monitor updates moving averages for all monitored system variables.

When loading the monitoring module into the kernel, the superuser specifies overload and underload conditions in terms of thresholds on the monitored variables, the moving averages, and their rate of change. Moreover, each monitored system variable, x_i, may be given its own weight, w_i. The monitor uses overload and underload thresholds in conjunction with the specified weights to compute the amalgamated *server load index* — akin to Steere's "progress pressure" [24]. To define the server load index formally we introduce the overload indicator function, $I_i(X_i)$, which operates on the values of monitored variables and moving averages X_i:

$$I_i(X_i) = \begin{cases} 1 & \text{if } X_i \text{ indicates an overload condition} \\ -1 & \text{if } X_i \text{ indicates an underload condition} \\ 0 & \text{otherwise} \end{cases}$$

For n monitored system variables the monitor computes the server load index as $\sum_{i=1}^{n} I_i(X_i)$. Once this value has been determined, the monitor checks whether this value falls into a range that triggers a notification to other modules (see Figure 5). Modules can simply register for such notifications by registering a notification range $[a, b]$ and a callback function of the form

```
void (* callback) ( int load_index )
```

with the monitor. In particular, the load-controller — to be described in the following section — uses this monitoring feature to receive overload and underload notifications.

Since the server's true capacity is not known before the server is actually deployed, it is difficult to define overload and underload conditions in terms of thresholds on the monitored variables. For instance, the highest possible file-system access rate is unknown. If the administrator picks an arbitrary threshold, the monitor may either fail to report overload or indicate a constant overload. Therefore, we implemented the system to dynamically learn the maximal and minimal possible values for the monitored variables, rates of change, and moving averages. Hence, thresholds are not expressed in absolute terms but in percent of each variable's maximal value. Replacing absolute values with percentage-based conditions improved the robustness of our implementation and simplified administration significantly.

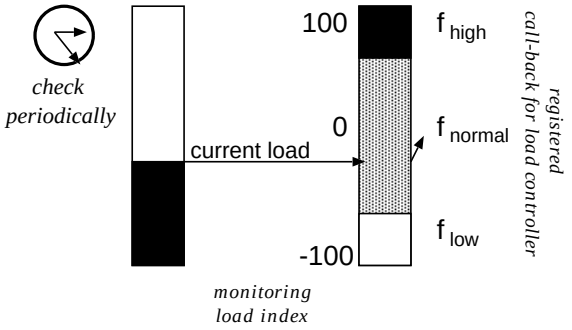

Figure 5: The monitor's notification mechanism

3.3 The Load-Controller

QGuard's sensitivity to load-statistics is a crucial design parameter. If QGuard is too sensitive, it will never settle into a stable state. On the other hand, if QGuard is too insensitive to server load, it will fail to protect it from overload. For good control of QGuard's sensitivity we introduce three different control parameters:

1. The minimal sojourn time, s, is the minimal time between filter switches. Obviously, it limits the switching frequency.
2. The length of the load observation history, h, determines how many load samples are used to determine the load average. The fraction $\frac{1}{h}$ is the grain of all load-measurement. For example, a history of length 10 allows load measurements with 10% accuracy.
3. A moderator value, m, is used to dampen oscillations when the shaped incoming packet rate matches the server's capacity. To switch to a more restrictive filter, at least m times more overloaded than underloaded time intervals have to be observed. This means that the system's oscillations die down as the target rate is reached, assuming stable offered load.

Small values for m (3–6) serve this purpose reasonably well. Since both s and m slow down oscillations, relatively short histories ($h \in [5, 15]$) can be used in determining system load. This is due to the fact that accurate load assessment is necessary only if the server operates

Indicator	Meaning
High paging rate	Incoming requests cause high memory consumption, thus severely limiting system performance through paging.
High disk access rate	Incoming requests operate on a dataset that is too large to fit into the file cache.
Little idle time	Incoming requests exhaust the CPU.
High outbound traffic	Incoming requests demand too much outgoing bandwidth, thus leading to buffer overflows and stalled server applications.
Large inbound packet backlog	Requests arrive faster than they can be handled, e.g., flood-type attacks.
Rate of timeouts for TCP connection requests	SYN-attack or network failure.

Table 1: Load indicators used in the Linux implementation

close to its operating point. Otherwise, overload and underload are obvious even when using less accurate load measurements. Since the moderator stretches out the averaging interval as the system stabilizes, measurement accuracy is improved implicitly. Thus, QGuard maintains responsiveness to sudden load-shifts and achieves accurate load-control under sustained load.

For statistical purposes and to allow refinement of filter hierarchies, the load-controller records how long each filter was applied against the incoming load. Higher-level software (Section 3.4) can query these values directly using the new QUERY_QGUARD socket option. In response to this query, the load-controller will also indicate the most recent load condition (e.g., CPU_OVERLOAD) and the currently deployed filter (Figure 6).

The load-controller signals an emergency to the load-controller whenever it has to switch into the most restrictive filter (drop all incoming traffic) repeatedly to avoid overload. Uncontrollable overload can be a result of:

1. ICMP floods
2. CPU intensive workloads
3. SYN attacks
4. Congested inbound queues due to high arrival rate
5. Congested outbound queues as a result of large replies
6. The onset of paging and swapping
7. File system request overload

To avoid signaling a false uncontrollable overload, which happens when the effects of a previous overload are still present, the system learns the time, t, that it takes for the system to experience its first underload after the onset of an overload. The time t indicates how much system load indicators lag behind control actions. If $2t > s$ (sojourn time, s), then $\frac{t}{2}$ is used in place of the minimal sojourn time. Thus, in systems where the effects of control actions are delayed significantly, the load-controller waits for a longer time before increasing the restrictiveness of inbound filters. Without the adaptation of minimal sojourn times, such a system would tend to oversteer, i.e., drop more incoming traffic than necessary. This problem occurs whenever server applications queue up large amounts of work internally. Server applications that decouple workload processing from connection management are a good example (e.g., the Apache Web server). However, if per-request work is highly variant, QGuard fails to stabilize. In such cases, a more radical solution like LRP [4] becomes necessary.

3.4 The Policy-Manager

The policy-manager implements three different features. First, it performs statistical analysis to dynamically adjust the granularity of the FH and estimates the best point of operation. Second, it identifies and reacts to sustained overload situations and tunes out traffic from malicious sources. Finally, it creates a FH that conforms to the service differentiation requirements.

The policy-manager views a FH as a set of n filters $\{F_0, F_1, ..., F_n\}$. As described in Section 2.1, filter F_i consists of a set of QGuard rules $\{r_{i,0}, r_{i,1}, ..., r_{i,m}\}$. For convenience we introduce some notation to represent different attributes of a filter.

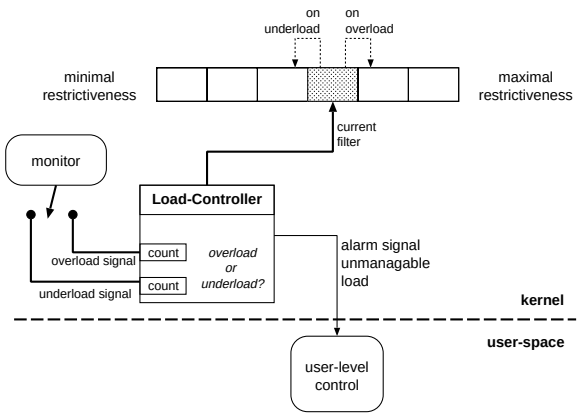

Figure 6: The Load-Controller

$\mathrm{TIME}(F_i)$ is the amount of time for which the load controller used F_i to contain system load. This attribute can be directly read from the statistics of the load-controller.

$\mathrm{RATE}(F_i)$ is the rate at which F_i accepts incoming packets. This is the sum of the rates given for all QGuard rules, j, that belong to the filter, $\mathrm{RATE}(F_i, j)$.

Since QGuard provides fair-share-style resource allocation, the policy-manager must create filter hierarchies where adjacent filters F_i and F_{i+1} satisfy the following: if a packet is admissible according to QGuard rule $r_{i+1,j}$, then it is also admissible according to rule $r_{i,j}$. However, the converse is not necessarily true. First, this implies that corresponding rules from different filters within a FH always specify the same traffic class. Second, $\mathrm{RATE}(F_{i+1}, j) < \mathrm{RATE}(F_i, j)$ for all j. Furthermore, F_0 always admits all and F_n drops all incoming traffic. The monotonicity of the rates in a filter-hierarchy is a result of our commitment to fair-share resource allocation.

The FH defined above guarantees that there is at least one filter, F_n, that can suppress any overload. Moreover, if there is no overload, no packet will be dropped by the load-controller because F_0 admits all packets. Depending on the amount of work that it takes to process each request and the arrival rate of requests, the load-controller will oscillate around some filter near the operating point of the system, i.e., the highest incoming rate that does not generate an overload. Since the rate difference between

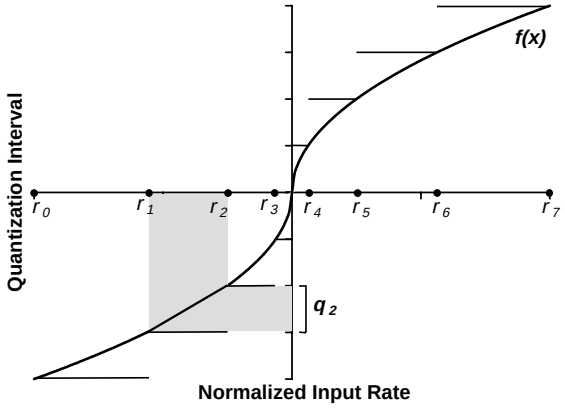

Figure 7: The compressor function for $q = 1/2$

filters is discrete, it is unlikely that there is one particular filter that shapes incoming traffic exactly to the optimal incoming rate. Therefore, it is necessary to refine the FH. To construct the ideal filter F^* that would shape incoming traffic to the maximal request arrival rate of the server, the policy-manager computes the *focal point (FP)* of the load-controller's oscillations:

$$\mathrm{FP} := \frac{\sum_{i=1}^{n} \mathrm{TIME}(F_i) * \mathrm{RATE}(F_i)}{\sum_{i=1}^{N} \mathrm{TIME}(f_i)}$$

Whether or not the policy-manager uses a finer quantization around the focal point depends on the load-controller's stability (absence of oscillations covering many filters). To switch between different quantization grains, the policy-manager uses a family of *compressor functions* [22] that have the following form:

$$f_q(x - FP) = \begin{cases} (x - FP)^q & \text{for } x \geq \mathrm{FP} \\ -(FP - x)^q & \text{for } x < \mathrm{FP} \end{cases}$$

Our experimental configuration only used $f_q(x)$ for $q = \{1, 1/2, 1/3\}$; Figure 7 shows $f_{1/2}(x)$. The horizontal lines reflects the quantization of the same function based on 8 quantization levels (the dashes on the y-axis). The ranges for each interval, marked on the x-axis, illustrate how their widths become smaller as they approach the focal point. Therefore, we only need to decrease q to

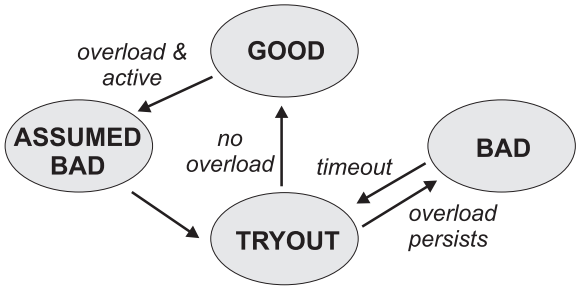

Figure 8: State transition diagram for the identification of misbehaving traffic classes

achieve higher resolution around the focal point. To compute the range values of each quantization interval, we apply the inverse function (a polynomial). This is illustrated by the shaded area in Figure 7.

Under the assumption that the future will resemble the past, compressor functions should be picked to minimize the filtering loss that results from the load controller's oscillations. However, this requires keeping long-term statistics, which in turn requires a large amount of bookkeeping. Instead of bookkeeping, we choose a fast heuristic that selects the appropriate quantization, q, based on the load-controller's statistics. Simply put, if the load-controller only applies a small number of filters over a long time, a finer resolution used. More specifically, if the load-controller is observed to oscillate between two filters, it is obvious that the filtering-grain is too coarse and a smaller q is used. We found that it is good to switch to a smaller q as soon as the load-controller is found oscillating over a range of roughly 4 filters.

When a new FH is installed, the load-controller has no indication as to which filter it should apply against incoming traffic. Therefore, the policy-manager advances the load-controller to the filter in the new FH that shapes incoming traffic to the same rate as the most recently used filter from the previous FH. The policy manager does not submit a new FH to the load-controller if the new hierarchy does not differ significantly from the old one. A change is significant if the new FP differs more than 5% from the previous one. This reduces the overheads created by the policy-manager, which includes context switches and the copying of an entire FH.

The above computations lead to improved server throughput under controllable overload. However, if the load-controller signals a sustained (uncontrollable) overload, the policy-manager identifies misbehaving sources as follows (see also Figure 8).

Assumed Bad: Right after the policy-manager recognizes that the load-controller is unable to contain the overload, each traffic class is labeled as potentially bad. In this state the traffic class is temporarily blocked.

Tryout: Traffic classes are admitted one-by-one and in priority order. A "tryout-admission" is probational and used to identify whether a given traffic class is causing the overload.

Good: A traffic class that passed the "tryout" state without triggering an overload is considered to be "good." It is admitted unconditionally to the system. This is the normal state for all well-behaved traffic classes.

Bad: A traffic class that triggered another overload while being tried out is considered to be a "bad" traffic class. Bad traffic classes remain completely blocked for a configurable amount of time.

To avoid putting traffic classes on trial that are inactive, the policy-manager immediately advances such traffic classes from state "tryout" to "good." All other traffic classes must undergo the standard procedure. Unfortunately, it is impossible to start the procedure immediately because the server may suffer from residual load as a result of the attack. Therefore, the policy-manager waits until the load-controller settles down and indicates that the overload has passed.

The problem of delayed overload effects became evident in the context of SYN-flood attacks. If Linux 2.2.14 is used as the server OS, SYN packets that the attacker places in the pending connection backlog queue of the attacked server take 75 s to time out. Hence, the policy-manager must wait at least 75 s after entering the recovery procedure for a SYN-attack. Another wait is may become necessary during the recovery period after one of the traffic classes revealed itself as the malicious source because the malicious source had a second chance to fill the server's pending connection backlog.

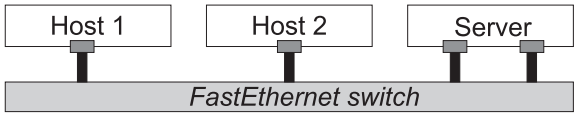

Figure 9: Testbed

4 Evaluation

To study QGuard's performance under various workloads, we implemented a load-generating server application. This server load generator can be configured to generate different types of load depending on the UDP or TCP port on which it received the request. The server application is fully parallel and uses threads from a thread-pool to serve incoming requests. The generated load may consist of four configurable components: CPU activity, memory usage, file accesses, and the generation of large response messages. We also implemented a client-side request generator which could simulate an arbitrary number of parallel clients. Each client can be configured to submit requests at a specific rate with random (Poisson-distributed) or constant inter-arrival times to the load-generating server.

The load generating server was run on its own Intel Pentium-based PC (450 MHz, 210 MB memory). Up to 400 simulated clients located on two other PCs request service at an average rate of 1 req/s. Client and server were connected through FastEthernet (see Figure 9).

For each test run, we established a baseline by comparing the QGuard-controlled server's performance against the server without QGuard. We found that QGuard fully achieved the goals for which it was designed: differential QoS and defense from overload attacks. We further found that QGuard degrades maximal server throughput only minimally — 2-3% (see Figure 10). This degradation results from the limitation of the input rate and the fact that we configured QGuard to keep the server's resource utilization at and below 99%.

4.1 Providing Differential QoS

The main objective in QGuard's design was graceful QoS degradation under overload. To study QGuard's behavior, we split our experiments into two series: one that studies the differential treatment of homogenous services — a typical preferred vs. standard services scenario — and of heterogenous services. In both cases, 200 clients were configured to request the preferred service while number of clients requesting the standard service increased from 0 to 200. Each measurement point represents the average response time or throughput over a 12 minute interval.

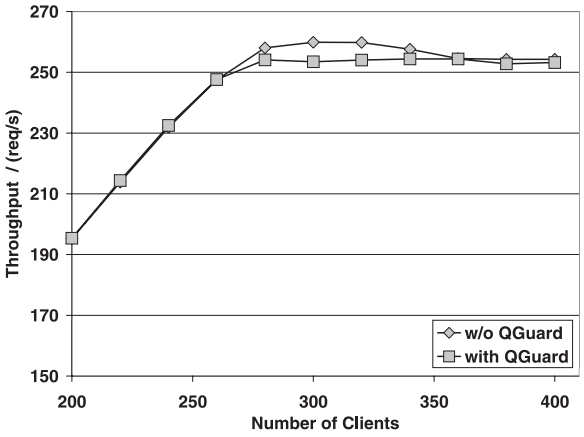

Figure 10: Performance loss due to QGuard

In the first experiment, we set the relative share of preferred vs. standard service to 4:1. As Figure 11 shows, QGuard clearly distinguishes the preferred from the standard service since the throughput of the preferred service remains stable as we increase the number of clients for standard service. This compares favorable with the approximately 40% performance drop that clients for preferred service would have experienced without QGuard protection. The results are even more dramatic in terms of clients' response time (Figure 12).

These results remain valid if one differentiates across clients instead of services. The only difference in the setup would be to configure traffic classes based on source addresses rather than destination services.

In a second series of experiments with services of heterogeneous workloads, we configured the preferred service to be CPU intensive and the standard service to be memory intensive. We set the priority of the preferred service to 10 and that of standard service to 1. This large ratio is very critical for maximizing system throughput because the maximal throughput of the CPU intensive service was an order of magnitude higher than that

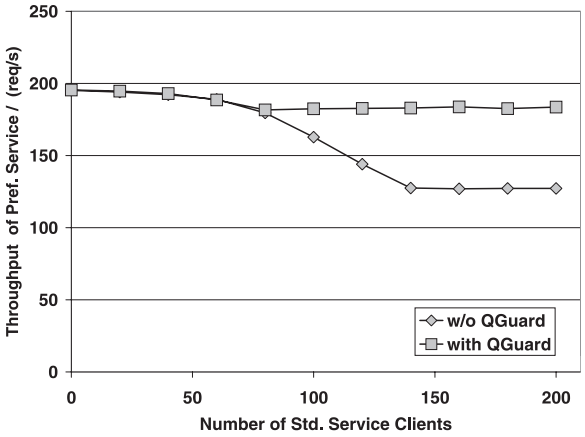

Figure 11: Throughput of preferred customers as the load from standard clients grows

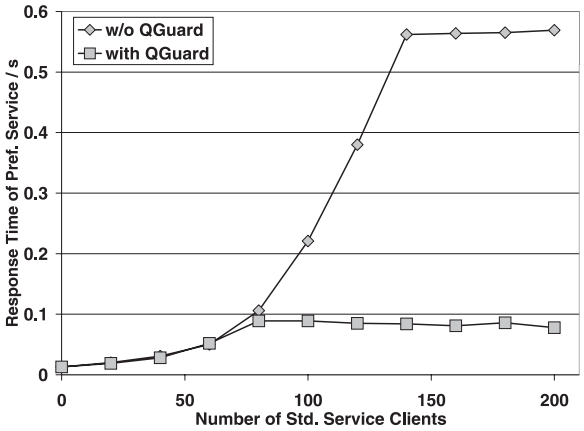

Figure 12: Response time of seen by preferred customers

of memory-intensive service. If the weights are not chosen appropriately (approximately reflecting the different maximal throughput values), then rounding errors in favor of the resource-heavy service's shaping rate can lead to significant performance degradation for the preferred service.

Aside from the previously mentioned limitation, Figures 13 and 14 show that QGuard performs well even if the workload is very heterogeneous. Without QGuard, the performance of clients requesting preferred service drops severely as demand for the standard service increases. With QGuard, the performance of both services matches the QoS differentiation requirements exactly, i.e., clients of preferred service are served at 10 $\times$ the rate of client of standard service.

The extreme increase in the clients' response time (around 40 clients) is a result of Linux's memory management. The likelihood of swapping increases as more non-preferred clients compete for service until swapping is inevitable (40 simultaneous non-preferred customers). Beyond this point there is always a number of requests from non-preferred clients swapped out so that the preferred customers' requests receive a larger share of the CPU, thus improving their response time. However, response times with QGuard are still 3 times better than without.

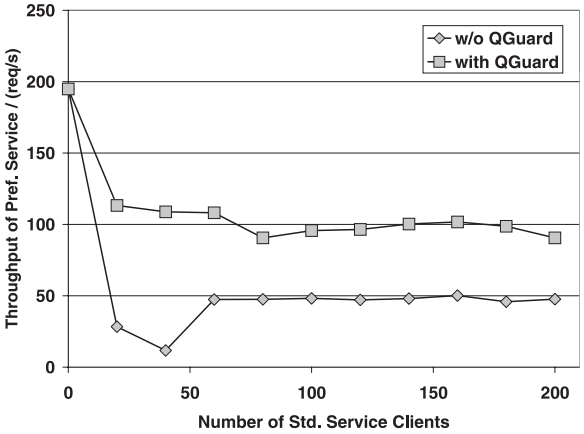

Figure 13: Throughput for preferred customers when standard customers request memory intensive services (e.g. database joins)

4.2 Effective SYN-Flood Defense

One of the main motivations behind our research on inbound traffic controls for overload defense mechanisms was the recent surge in the number of DoS attacks experienced by major Internet servers. To put QGuard to the test, we configured the server to provide service to three different client populations: preferred customers from host 1, standard service customers from host 2, and best-effort service for the rest of the world. 200 seconds into the measurement, we launched a SYN-flood attack on the server from a spoofed (unreachable) address. Service was quickly disrupted (point [b] in Figure 15). However, after a short time [c], QGuard detects the DoS attack

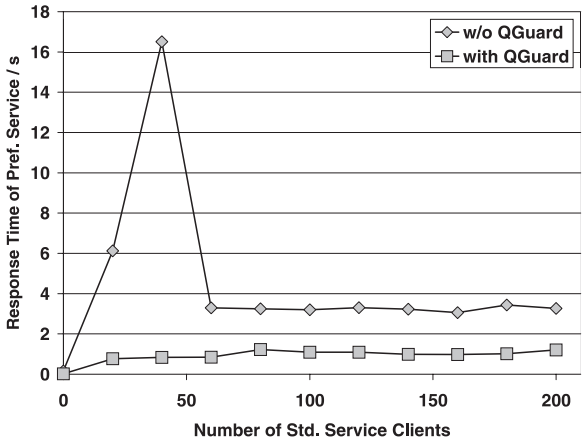

Figure 14: Response time seen by preferred customers

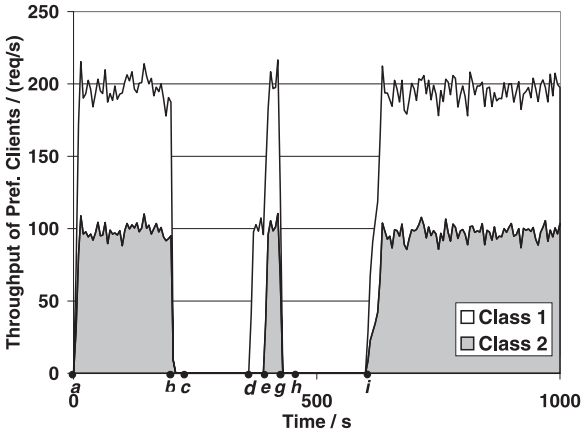

Figure 15: Restoring service under SYN-flood attacks

and disallows all incoming traffic until all SYN packets that are currently present in the server's connection backlog time out (point [d]). Then it enables client accesses in priority order ([d] and [e]). Since neither standard nor preferred clients cause the SYN-flood, they are labeled as *good* traffic classes. Once QGuard admits all other clients [g] — including the attacker — the service experiences another disruption which is detected by QGuard at point [h]. Upon detection, best-effort clients are denied access to the server and service resumes [j] after all false SYN packets that the attacker placed on the server during its temporary admission time out. The graph shown in Figure 15 represents a typical test run (not the best case).

As we studied the behavior of QGuard under SYN-floods, we found that it is difficult to distinguish a SYN-flood from a surge in legitimate requests until spoofed SYN packets begin to time out. Since this timeout is very large in the regular Linux kernel (75 s) the recovery phase takes quite long. Almost all of the recovery time can be attributed to this generous timeout. One may argue that we should simply wipe out all SYN packets in the server's backlog once a SYN attack has been discovered to speed up recovery. However, this is not possible without violating the TCP protocol. Such a protocol alteration could break some client/server applications.

4.3 Tuning out the "Ping-of-Death"

The "ping-flood" attack exploits the fact that the processing of ICMP requests is kernel-based, thus generally preempting all other system activities. In this scenario an attacker either directly or through so called zombies sends a never ending stream of ICMP ping packets to a server. Although the per-request processing overhead of ping is quite low, the large number of packets leads to a complete lock-up of the server. To create a lock-up situation on the experimental server, we flooded it on both of its incoming interfaces at the maximal packet rate of 100Mbps Ethernet.

At 100 s in Figure 16, the start of the ping-flood, the server's throughput plummets in response to the high workload placed on the system due to ICMP request processing. QGuard responds to the excessive load immediately and reduces the acceptance rate for ICMP packets until service throughput almost reaches pre-attack levels (after 175 s). The reason why the maximal throughput is not quite reached is that QGuard still admits a small, manageable amount of `ping` requests. QGuard's reaction is caused by three events: almost all busy cycles are executed on behalf of the system, a large backlog of incoming packets, and high CPU utilization.

Since QGuard successfully defends the system from this kind of attack, it is quite safe to connect a QGuard protected server directly to the Internet even through high-bandwidth links. However, QGuard can only mitigate the effect that such a ping-flood has on the incoming link's available bandwidth. The sources of the attack may still saturate incoming bandwidth by flooding the link. However, a QGuard protected system does not aggravate the

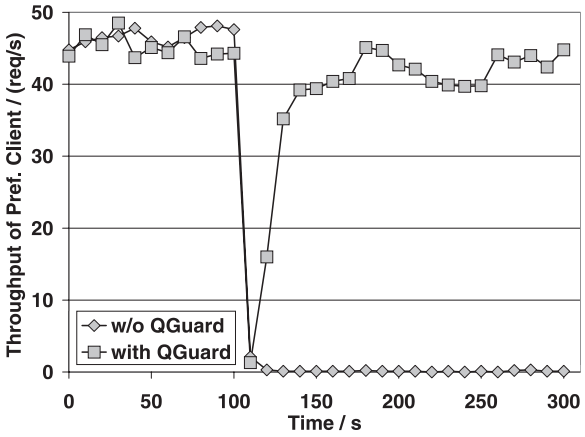

Figure 16: QGuard's response to an ICMP flood

problem by sending replies over the same congested link.

5 Related Work

A number of commercial and research projects address the problem of server overload containment and differential QoS. Ongoing research in this field can be grouped into three major categories: adaptive middleware [2, 3, 14], OS [4, 6, 12, 15, 17, 20, 23] and network-centric solutions [7, 19].

5.1 Middleware for QoS Differentiation

Middleware solutions coordinate graceful degradation across multiple resource-sharing applications under overload. Since the middleware itself has only little control over the load of the system, they rely on monitoring feedback from the OS and application cooperation to make their adaptation choices. Middleware solutions work only if the managed applications are cooperative (e.g., by binding to special communication libraries).

IBM's workload manager (WLM) [3] is the most comprehensive middleware QoS management solution. WLM provides insulation for competing applications and capacity management. It also provides response time management, thus allowing the administrator to simply specify target response times for each application. WLM will manage resources in such a way that these target response times are achieved. However, WLM relies heavily on strong kernel-based resource reservation primitives, such as I/O priorities and CPU shares to accomplish its goals. Such rich resource management support is only found in resource rich mainframe environments. Therefore, its design is not generally applicable to small or mid-sized servers. Moreover, WLM requires server applications to be WLM-aware. WebQoS [14] models itself after WLM but requires fewer application changes and weaker OS support. Nevertheless, it depends on applications binding to the system's dynamic communication libraries. WebQoS is less efficient since it manages requests at a later processing stage (after they reach user-space).

5.2 Operating System Mechanisms for Overload Defense and Differential QoS

Due to the inefficiencies of user-space software and the lack of cooperation from legacy applications, various OS-based solutions for the QoS management problem have been suggested. OS-level QoS management solutions do not require application cooperation, and they strictly enforce the configured QoS.

The Scout OS [23] provides a *path* abstraction, which allows all OS activity to be charged to the resource budget of the application that triggered it. When network packets are received, for example, they are associated with a path as soon as their path affiliation is recognized by the OS; they are then handled using to the resources that are available to that path. Unfortunately, to be effective Scout's novel path abstraction must be used directly by the applications. Moreover, Scout and the other OS-based QoS management solutions [4, 6, 12, 15, 20] must be configured in terms of raw resource reservations, i.e., they do not manage Internet services on the more natural per request-level. However, these solutions provide very fine-grained resource controls but require significant changes to current OS designs.

Mogul's and Ramakrishnan's work [17] on the receive livelock problem has been a great inspiration to the design of QGuard. Servers may suffer from the receive livelock problem if their CPU and interrupt handling mechanisms are too slow to keep up with the interrupt stream caused by incoming packets. They solve the problem by making

the OS slow down the interrupt stream (by polling or NIC-based interrupt mitigation), thus reducing the number of context switches and unnecessary work. They also show that a monitoring-based solution that uses interrupt mitigation only under perceived overload maximizes throughput. However, their paper only targets receive-livelock avoidance and does not consider the problem of providing QoS differentiation — an important feature for today's Internet servers.

5.3 Network-Centric QoS Differentiation

Network-centric solutions for QoS differentiation is becoming the solution of choice. This is due to the fact that they are even less intrusive than OS-based solutions. They are completely transparent to the server applications and server OSs. This eases the integration of QoS management solutions into standing server setups. Some network centric-solutions are designed as their own independent network devices [7], whereas others are kernel-modules that piggy-back to the server's NIC driver [19].

Among the network-centric solutions is NetGuard's Guardian [19], which is QGuard's closest relative. Guardian, which implements the firewalling solution on the MAC-layer, offers user-level tools that allow real-time monitoring of incoming traffic. Guardian policies can be configured to completely block misbehaving sources. Unlike QGuard, Guadian's solution is not only static but also lacks the QoS differntiation since it only implements an all-or-none admission policy.

6 The Future of QGuard

Since the QGuard prototype still requires the addition of kernel modules to the Internet server's OS, some potential users may shy away from deploying it. We quoted the same issue earlier as a reason for the popularity of network-centric solution for the QoS-management problem. Therefore, we believe that QGuard should follow the trend. It would ideally be built into a separate firewalling/QoS-management device. Such a device would be placed in between the commercial server and the Internet, thus protecting the server from overload. Such a setup necessitates changes in the QGuard monitoring architecture. Further research is necessary to determine whether an SNMP-based monitor can deliver sufficiently up-to-date server performance digests so that QGuard's load-controller can still protect the server from overload without adversely affecting server performance.

Another future direction for the QGuard architecture would be to embed it entirely on server NICs. This would provide the ease of plug-and-play, avoid an additional network hop (required for a special QGuard frontend), and reduce the interrupt load placed on the server's OS by dropping packets before an interrupt is triggered. Another advantage of the NIC-based design over our current prototype is that it would be a completely OS-independent solution.

In this paper we have proven that it is possible to achieve both protection from various forms of overload attacks and differential QoS using a simple monitoring control feedback loop. Neither the core networking code of the OS nor applications need to be changed to benefit from QGuard's overload protection and differential QoS. QGuard delivers surprisingly good performance even though it uses only inbound rate controls. QGuard's simple design allows decoupling QoS issues from the underlying communication protocols and the OS, and frees applications from the QoS-management burden. In the light of these great benefits, we believe that inbound traffic controls will gain popularity as a means of server management. The next step for future firewall solutions is to consider the results of this study and add traffic shaping policies and a simple overload control-loop similar to QGuard's load-controller. As we have shown in this paper, these two mechanisms may suffice for the design of sophisticated QoS management solutions such as QGuard's policy-manager.

7 Acknowledgements

We gracefully acknowledge helpful discussions with, useful comments from, and the support of Kang Shin, Brian Noble, and Padmanabhan Pillai.

References

[1] ABDELZAHER, T., AND BHATTI, N. Web Content Adaptation to Improve Server Overload Behavior. In *International World Wide Web Conference* (May 1999).

[2] ABELZAHER, T. F., AND SHIN, K. G. QoS Provisioning with qContracts in Web and Multimedia Servers. In *IEEE Real-Time Systems Symposium* (Pheonix, AZ, December 1999).

[3] AMAN, J., EILERT, C. K., EMMES, D., YOCOM, P., AND DILLENBERGER, D. Adaptive Algorithms for Managing Distributed Data Processing Workload. *IBM Systems Journal 36*, 2 (1997), 242–283.

[4] BANGA, G., AND DRUSCHEL, P. Lazy Receiver Processing LRP: A Network Subsystem Architecture for Server Systems. In *Second Symposium on Operating Systems Design and Implemenation* (October 1996).

[5] BELLOVIN, S. M. Security Problems in the TCP/IP Protocol Suite. *Computer Communication Review 19*, 2 (April 1989), 32–48.

[6] BRUNO, J., BRUSTOLONI, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. Retrofitting Quality of Service into a Time-Sharing Operating System. In *USENIX Annual Technical Conference* (June 1999).

[7] CISCO INC. Local Director (White Paper) http://cisco.com/warp/public/cc/ci sco/mkt/scale/locald/ tech/lobal_wp.htm. 2000.

[8] DAWSON, T. Linux NET-3-HOWTO. 1999.

[9] ELLIOTT, J. Distributed Denial of Service Attacks and the Zombie Ant Effect. *IT Pro* (March 2000).

[10] FLOYD, S., AND JACOBSEN, V. Link-Sharing and Resource Management Models for Packet Networks. *Transactions on Networking 3*, 4 (1995), 365–386.

[11] GARBER, L. Denial-of-Service Attacks Rip the Internet. *Computer* (2000).

[12] HAND, S. M. Self-Paging in the Nemesis Operating System. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation* (New Oreleans, Lousiana, February 1999), USENIX, pp. 73–86.

[13] HEWLETT PACKARD CORP. Ensuring Customer E-Loyalty: How to Capture and Retain Customers with Responsive Web Site Performance. http://www.internetsolutions.enterprise.hp.com/ webqos/products/overview/e-loyalty_white_paper.pdf.

[14] HEWLETT PACKARD CORP. WebQoS Technical White Paper. http://www.internetsolutions.enterprise.hp.com/ webqos/products/overview/wp.html, 2000.

[15] JEFFAY, K., SMITH, F., MOORTHY, A., AND ANDERSON, J. Proportional Share Scheduling of Operating System Services for Real-Time Applications. In *Proceedings of the 19th IEEE Real-Time Systems Symposium* (Madrid, December 1998).

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

[17] MOGUL, J. C., AND RAMAKRISHNAN, K. K. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *Transactions on Computer Systems 15*, 3 (August 1997), 217–252.

[18] MOGUL, J. C., RASHID, R. F., AND J. ACCETTA, M. The Packet Filter: An Efficient Mechanism for User-Level Network Code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles and Design* (November 1987), ACM.

[19] NETGUARD, INC. Guardian Real-time Performance Monitoring (RPM). http://www.netguard.com/support_doc.html.

[20] REUMANN, J., MEHRA, A., SHIN, K., AND KANDLUR, D. Virtual Services: A New Abstraction for Server Consolidation. In *Procedings of the 2000 USENIX Annual Technical Conference* (June 2000), USENIX.

[21] RUSSELL, P. IPCHAINS-HOWTO. http://www.rustcorp.com/linux/ipchains/HOWTO.html.

[22] SAYOOD, K. *Introduction to Data Compression*. Morgan Kaufmann Publishers, Inc., 1996.

[23] SPATSCHECK, O., AND PETERSON, L. L. Defending Against Denial of Service Attacks in Scout. In *Third Symposium on Operating Systems Design and Implemenation* (February 1999), pp. 59–72.

[24] STEERE, D. C., GOEL, A., GRUENBERG, J., MCNAMEE, D., PU, C., AND WALPOLE, J. A feedback-driven proportion allocator for real-rate scheduling. In *Third Symposium on Operating Systems Design and Implementation* (New Orleans, LA, USA, Feb 1999), pp. 145–58.

[25] STÖCKLE, O. Overload Protection and QoS Differentiation for Co-Hosted Web Sites. Master's thesis, ETH Zürich, June 1999.