Policy Based End Server Resource Regulation

Aman Garg and A.L.Narasimha Reddy

Abstract—

As more and more critical services are provided over the Internet, the risk to these services from malicious users is also increasing. Several networks have witnessed problems like Denial of Service attacks over the recent past. Denial of Service attacks work by flooding a network, router or an end server with malicious packets. An excess of these packets can cause overload conditions in the network and on the end servers. The purpose of this paper is to provide an efficient way to keep a track of server and network resources in an attempt to mitigate the effect of such attacks. Our scheme provides a general, and not attack specific, mechanism to provide graceful server degradation in the face of such an attack.

Our solution has two parts. The first part is to maintain sufficient state information *per resource* at the network level, so as to be able to get a high-level picture of current network behavior and resource consumption status. The second part is a policing or enforcing mechanism which can regulate resource consumption, and recover excess resources if they are believed to be held by malicious flows. A *Window Regulation* based scheme is proposed and implemented to do the regulation of traffic. It is shown that this scheme, when combined with conventional rate-based QoS regulation, offers enhanced control over resources and traffic.

Index Terms—**Denial of Service, QoS, Bastion Host, Policy Based Networking, SYN Attack, CGI Attack, Resource Regulation, Server Resources.**

I. Introduction

With the ever increasing popularity of web, information retrieval over wide area networks has gained significant importance. Protecting network and endhost infrastructure has become essential. Several recent Denial of Service (DoS) attacks on popular servers have brought this to prominence. DoS is a malicious way to consume resources in a network, a server cluster or in an end host, thereby denying service to other legitimate users. This renders the system unusable because it cannot do productive work. The challenge is to design general schemes that identify and successfully curb a variety of these attacks by following a common general strategy. In this paper, we address the DoS problem from a policy based resource regulation perspective. Existing Quality of Service (QoS) mechanisms are geared to provide different levels of service to users. The underlying system achieves this by means of policing and admission control. This paper utilizes QoS mechanisms in order to regulate resource consumption at the end servers. It is expected that such a regulation would mitigate the impact of DoS attacks. This paper identifies a method to keep track of server and network resources at the network level, before much work is carried out on the received packets at the end servers, in an attempt to mitigate the effect of DoS attacks.

A. Garg is with the Department of Electrical Engineering, Texas A&M University, College Station, TX 77843–3128. Phone: +1 979 845–9578, e-mail: aman@ee.tamu.edu

A. L. Narasimha Reddy is with the Department of Electrical Engineering, Texas A&M University, College Station, TX 77843–3128. Phone: +1 979 845–7598, e-mail: reddy@ee.tamu.edu

A. Our Approach

Resource Control and QoS are closely tied to each other. By managing one, we can automatically manage the other. In our context, resource control or regulation is employed to reduce the opportunities for DoS attacks to succeed. Our basic idea is to *extend resource control to the network subsystem*. By taking corrective action at the network layer, we can affect how the resources are consumed in a networked server.

Resources could include network bandwidth, protocol state memory buffers, kernel interrupt processing, memory, or CPU cycles. Our solution is, to maintain at the network level, a rough snapshot of available system resources. We do this by reflecting available resources at a central place in the network. We call this central location as the *Bastion Host*, which could as well be the ingress router or corporate firewall. As flows or packets keep consuming network or system resources, appropriate state entries are updated. For example, each time a request for an expensive CGI script comes in, a record is made that the amount of processing capacity remaining in the end server has gone down by some units. When a traffic class has exceeded its allocated quota, future packets belonging to this class are dropped. Such accounting is expected to identify potential attacks before they have consumed valuable resources at the end servers.

B. Motivation

Currently most of the network-centric approaches to mitigate the effect of Denial of Service, have been attack-specific. For example, TCP SYN attacks are handled by TCP SYN cookies or TCP SYN regulation and termination [1]. Similarly ICMP flood attacks are handled by turning off ICMP echo reply. Turning off ICMP echo reply however still does not save the servers from wasting time in serving high-priority interrupts. Also, currently available solutions tend to fail in the case of distributed attacks because attack packets seem to come from all directions. The motivation for this work was to investigate the feasibility of a general approach that monitors multiple attacks and regulates multiple resources in an aggregate fashion.

C. Background on Denial of Service Attacks

Recent times have seen several kind of attacks against computer networks. These attacks can usually be divided into (i) intrusion based attacks which work after a computer system or network has been compromised, and (ii) *Denial of Service (DoS)* based attacks, which do not require privileged access to a system. DoS attacks succeed because the attack packets are in no way differentiable from

normal traffic. The problem in this case is that the attacker tries to flood the server or network with legitimate traffic which has an undue effect on the system because of load constraints, buffer overflows, and less robust implementations on the server side. Denial of Service (or DoS) as these attacks are commonly called, result in rapidly degrading the service, thus denying service to other legitimate users. We list here several common types of denial of service attacks and the *resources consumed* in each case.

- *TCP SYN Flood* : This attack is tcp specific. During a tcp 3-way handshake, the client sends a SYN packet with an initial sequence number to the server. The server sets up data structures in anticipation for a new tcp connection, and sends a SYN from its side. If the client does not send back an acknowledgment, the data structures will be held up in the server until they time out – typically after 75 to 450 seconds. In most operating system implementations, there is a limit on the number of connections in the SYN_RECVD state. The resources consumed by this attack are protocol state buffers, backlog queues, and precious kernel memory.
- *UDP Flood* : UDP is a connection-less protocol that has no notion of flow control. Hence, anyone can send arbitrarily large amount of traffic to a victim host and monopolize its resources. The idea of a UDP flood is to eat up both CPU and available network bandwidth.
- *ICMP Directed Broadcast* : The attacker sends ICMP echo request packets to broadcast addresses, with the source address spoofed to that of the victim. In response, the broadcast addresses send out a flood of ICMP echo replies to the victim. Intermediate routers are also affected in this attack. Resource consumed is primarily network bandwidth and end host CPU cycles.
- *ICMP Flood Ping* : These 'ping' packets carry a large payload and sometimes this payload is fragmented to increase the processing cost on the destination machine. The resources targeted in this attack are similar to those consumed in a UDP flood – network bandwidth and CPU.
- *DNS Flood* : A DNS zone transfer request allows a remote user to copy all of the contents out of a domain name server zone file. This will give a complete listing of all the hosts listed with the DNS. Such a request is yet another way to generate a large amount of data which could be directed to a victim. Resources consumed are identical to that in a UDP flood.
- *Fragmentation Attack* : This could be used with any of the attacks mentioned above. The idea here is to send a very large number of packets as very small fragments. When the receiving machine tries to put together these packets in the IP stack, it eats up CPU and memory resources.
- *CGI Flood* : CGI packets are examples of expensive packets that require more CPU cycles to process. A small request might execute an expensive script on the server side. By repeatedly asking a server to execute a lot of CGI scripts, an attacker could tie up CPU resources on the server.

All the above attacks can be combined to make a distributed denial of service attack (DDoS). This attack uses a large number of compromised machines or agents in a synchronized fashion. It is harder to trace a distributed attack. Part of our motivation for doing aggregate resource regulation, comes from the threat due to distributed attacks.

D. Organization

The rest of the paper is organized as follows. In Section 2, we take an in-depth look at our design rationale and discuss the key ideas of our contribution. Section 3 presents details about our prototype implementation. Results obtained for rate control as well as for window control are discussed in Section 4. Related work is cited in Section 5 while Section 6 concludes the paper.

II. Design Details

A. Resources and Regulation

A network and server system consists of several resources – network bandwidth, memory, CPU processing, interrupt handling capacity, buffers, file handles, network sockets, etc. Some of these resources can be protected by rate based schemes – like traffic shaping or rate control. Other resources are fixed, or capacity based resources, which are consumed and released alternately.

QoS regulation is required so that a flow or traffic class achieves the service guarantee that an end-user expects from it. QoS requirements could be static or dynamic (Figure 1). Examples of static QoS allocation are *Virtual Private Networks* and *Class Based Queuing* schemes. Here the requirements are largely fixed and vary only over long periods of time. In situations where (static) aggregation of traffic is not sufficient, a case by case admission control to do QoS regulation may be required. The *Resource Reservation Protocol (RSVP)* and *Internet Streaming Protocol (ST-II)* are two examples of dynamic QoS allocation protocols. These protocols allow a client or a flow to negotiate resources on an end to end basis. Intermediate routers commit resources and disallow the flow if they cannot meet the requirements of the flow.

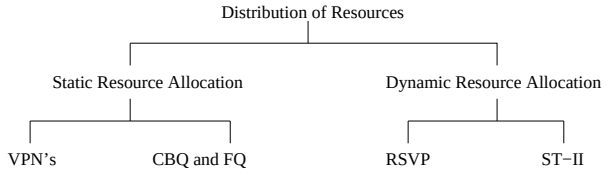

Fig. 1. Resource re-distribution for desired QoS

Once admitted, a flow or a class of flows, has to be policed so that it abides by its requested rate. This is attained by maintaining statistics on the long term and short term sending rates of the flows. Usually a traffic bounding function like a *token bucket filter* is used for this purpose.

B. Traffic Control Mechanisms

Traffic regulation is a means to achieve the required QoS from a user or application's point of view. These goals may have to do with load, bandwidth, burstiness, delay, security, etc. Some common schemes used to control traffic are described here :

- Firewalling : This is simply an accept or drop policy. The primary uses are more oriented toward security and admission policies.
- Throttling : This is to enforce a flow rate, for example in bytes/sec, as does *apache*'s *mod_throttle* module. Excess packets are simply discarded and no attempts are made to do any kind of traffic shaping.
- Pacing of Acknowledgments : This solution works only on protocols like TCP which are flow based and which respond to congestion. By holding back or delaying *acks* flowing in the reverse direction, the flow can be forced to backoff to a rate imposed by the administrator [2].
- Traffic Shaping : Simple rate control does not ensure any leniency in flow rates. If a flow exceeds its rate, albeit for a short duration, excess packets are discarded. This behavior can be rectified by having a token-bucket kind of filter that does long term rate control but at the same time allows short term burstiness.
- Complex Scheduling : Combining nearly all the schemes above, and also taking cues from switch scheduling algorithms implemented on routers, we can come up with more complex scheduling algorithms. These include priority based scheduling, fair scheduling, class based queuing, in addition to simple (σ, ρ) token bucket shaping.
- Window Regulation : There are situations when simple rate control is not enough. This is true for fixed resources like memory which do not get renewed with time. Fixed resources are like a fixed number of tokens. These tokens get regenerated only when the server sends back an acknowledgment indicating that it has freed some units of that resource. These mechanisms can be applied on a per-flow basis or on an aggregate basis.

C. Concept of a Bastion Host

The idea of a *Bastion Host* originated from security considerations when it was important to maintain a strict control over the kind of traffic entering a corporation's network. A bastion host is nothing but a hardened ingress router that combines the function of a firewall and admission control engine. All incoming traffic would be directed to the corporation or server farm through this bastion host. Hence, it is important for the bastion host to operate efficiently and not become a bottleneck itself. Usually routers are expected to handle higher traffic volumes than end servers, so this should not pose a problem. Figure 2 shows a typical topology in which a bastion host would be deployed.

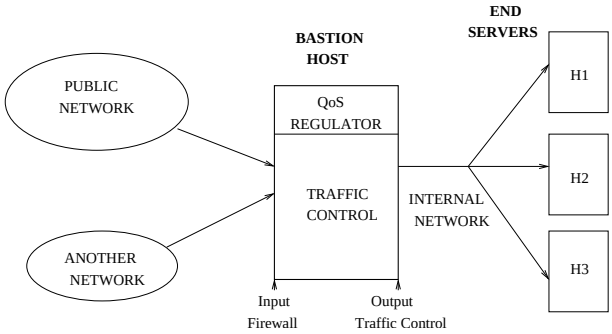

Fig. 2. QoS Regulator or Bastion Host

Since all traffic now comes in through the bastion host, we can enforce all admission control, traffic management, and traffic shaping on the outgoing interfaces of this ingress router. The advantages of this approach is that it does not require any modification on the server farm or the corporation that is being protected. Policies can be deployed and enforced without the end servers knowing that new policies have come into existence. Monitors can be installed on the bastion host itself or as independent machines sniffing network traffic. These monitors give feedback to the bastion host which can then implement new traffic control policies.

D. Reflecting Resources into the Bastion Host

In order for the bastion host to function as desired, it needs to have a quantitative idea of the resources available in the end servers. As mentioned above, this can be achieved by monitors. However, we introduce the concept of *implicit feedback*. This works by reflecting into the bastion host, the resources available in the end servers and in the network. Whenever a new flow or packet comes into the system, it is mapped to a traffic class. Each traffic class has a quota associated with it. Whenever a unit of resource is consumed by a flow, the corresponding state counters for that resource are decremented. The advantage of reflecting available resources to the bastion host allows the bastion host to make intelligent decisions while admitting traffic. It also eliminates the need for polling the end server about load statistics.

Our approach is to have an aggregate view of all the resources in the system. This gives us a coarse level control over each kind of resource present in the system. Such an approach proves useful in distributed attacks because all attack packets (consuming a specific kind of resource), will map to the same traffic class and consume quotas out of that traffic class only. Traffic in other classes would stay unaffected.

E. The Big Picture

Figure 3 shows a high level overview of how output traffic control is implemented on the bastion host. The core part of the design is to use a packet classifier to map different flows or packets to traffic classes, depending on the resources they consume. No assumptions are made about

the identity of the packets. The primary mapping criteria is to isolate resource consumption. Traffic classes are then policed by means of either rate-control or window control. Rate control is useful for controlling certain resources like bandwidth, etc. Other traffic classes that require fixed resources are better policed by our window control scheme, as described in later sections. The feedback coming from the server is utilized to make the window limit self-regulated. Figure 3 shows two traffic classes (1 and 2) being regulated by a rate control and one traffic class (3) being regulated by a window control.

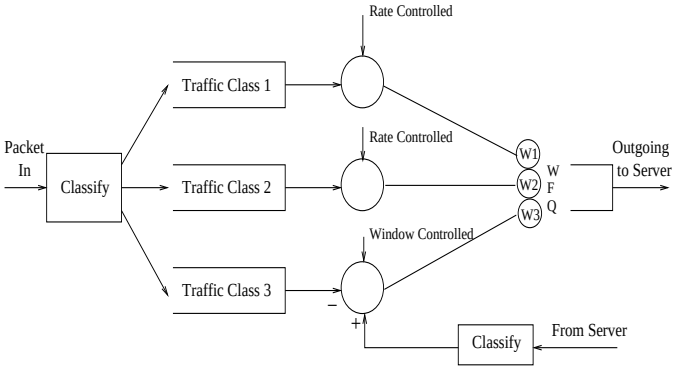

Fig. 3. QoS Regulation : Big Picture

F. Rate Control to do QoS Regulation

The idea of doing rate control is to identify the per-packet processing cost for different types of packets, and limiting the flow rates such that the end server does not go into overload situations. Simply controlling the *aggregate flow rate*, such as total network bandwidth, is not sufficient – a finer grain control is required. This requires incoming packets to be classified into *traffic classes* depending upon the resources they consume. For example, different rate-limiting policies could be applied to flows in Fig. 3 in terms of bitrate, connections per second, or packets per second.

The administrator has a high degree of flexibility in assigning incoming traffic to various classes. In our implementation, we first identify the resource that we need to police. Examples are UDP bandwidth, TCP bandwidth, TCP SYN rate per service port, and CGI packets. We then build policies into our QoS regulator to implement filters and classify the packets into traffic classes. Policing is done to ensure that the individual classes stay within the rates set by the administrative policies. A token bucket filter is used to allow for short term bursts.

Whenever a new packet comes into the system, it is matched against all available filters and an attempt is made to classify it into one of the traffic classes. If the packet matches a class, the corresponding state variables for that class are updated and a test is run. This test determines if admitting the packet will not violate the ratelimit set for this class of traffic. If all goes well, the packet is queued on the outgoing interface. Figure 3 shows a case when in addition to the rate limits imposed on the different classes, weights are assigned at the time of queuing. This allows excess resources to be shared by other traffic classes. It is also noted that traffic can be controlled at each input link independently from other links.

G. Window Control for Aggregate Resource Regulation

G.1 When is Window Regulation Appropriate?

Window Control is needed when simple rate control is not sufficient to police the traffic. Fixed server or network resources which are based on capacity, as opposed to rates, are an example where window control is applicable. By enforcing a separate window for each resource, we can ensure that the traffic stays within the administrator-decided policies, and does not consume too much of a fixed resource. Examples of such resources are CPU cycles on end servers, memory, network buffers (like *sk_buffs*), and protocol state buffers (like SYN backlog queues).

G.2 Window Control Rationale

Our scheme is designed using a Window Regulation concept, such as used by TCP for flow control. Windowing allows a resource to be self-regulated, as new requests cannot enter the system until the earlier requests have left the system.

In our model, we propose a window limit *per resource* or *per traffic aggregate*. In other words, we impose a limit on how much of a certain resource can be consumed at any given time. After this limit is reached, incoming requests or packets seeking this resource are dropped or delayed at the router until the server sends some kind of indication that an earlier request has freed its resources. When this happens, the window is relaxed and more flows or requests can be admitted. The window limit quantifies the resource availability. The same resource can be split up into multiple portions, and a portion could be allocated to a different traffic class. For critical content, for example, when a transaction is going on at a web server, a portion could be allocated such that all transaction requests are guaranteed some percentage of resources even during overload. This ensures that critical transactions or preferred flows are not starved in presence of overload, or denial of service scenarios.

Window control assumes significance when we consider resources that do not depend on rate but rather on the current state of the server. As a simple example, consider the CPU cycles of the end server as a resource. Some requests or flows consume more of the CPU than others. By regulating these CPU intensive requests, the server load can be controlled and quality of service for critical applications can be preserved.

G.3 Window Control Design

There are two steps involved in realizing the above scheme :

First, state information is maintained in the regulator on a per-aggregate, or per-resource basis[1]. For example,

[1] QoS Regulation can be done on a per-flow basis also but that

if protocol state buffers (like outstanding SYN buffers) is a resource on the end server that the regulator wants to control, we maintain a table hashed on the basis of TCP initial sequence number. This resource allocation table is maintained dynamically based on statistics gathered from the packets as they move through the regulator. In other words, there is no need to modify the end servers to be able to provide this information.

Secondly, some kind of policing or enforcement mechanism has to be implemented to make sure that flows are within the server administrator's policies. A firewall is capable of simple dropping mechanisms. The Linux network QoS architecture (released with kernel versions 2.2 onwards) can handle rate-based policing. We extend these schemes further to send packets based on a Window Control algorithm which is used to control the server load.

We could use any metric to measure load. For example the limit could be represented as "100 concurrent cgi scripts", or "at most 64 outstanding SYNs per tcp port". The QoS regulator admits packets only if there is room in the current window. If the current window is full, the regulator drops or delays packets temporarily until the server sends an indication (in terms of acknowledgments flowing in the opposite direction) showing that more flows or packets can be admitted. These acknowledgments need not be TCP acks – all we need is some kind of *completion status* signal which is likely to be different for each traffic class. For the CGI case, the completion signal could be an HTTP 1.1 OK message showing that that the CGI job has exited the system.

The QoS regulator may take action to free up resources when resource-specific guidelines are violated. In a more complex scenario, the state information in the regulator could be used to feed a daemon process analyzing the beginning of an attack. This daemon could then make administrative decisions and install new policies in the regulator. Figure 4 shows a simple implementation of our window control as applied to CPU intensive CGI packets.

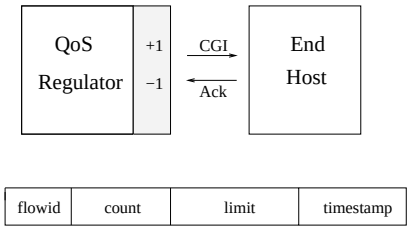

Fig. 4. Window Control Overview

H. Flow control vs. Resource Aggregate control

Usually when policing traffic is concerned, people think of the notion of per-flow policing. So traffic is divided into flows depending on their identities, in terms of the source or intended destination. Limits are then imposed based on administrative policies.

In our case, the focus is on resources and not on flow identities. So it makes more sense to do policing based on resource aggregates instead of individual flows. The classification engine that parses packets as they come in on the network interface, however, does not have a notion of how expensive the packet is in terms of resources it would consume on the end server. Therefore, there is a need to *map* or reflect incoming packets to resource aggregates, and controlling resources at the aggregate level. This leads us to the concept of creating *traffic classes*, where each traffic class is supposed to consume a certain kind of resource. By regulating aggregate resource consumption, distributed attacks (like Distributed Denial of Service), can be effectively contained.

requires apriori information about client flows – like which domains are trusted and which are untrusted.

III. Implementation Details

Recent Linux kernels (beginning with the 2.2 kernel) have included a Quality of Service architecture in the network stack [3], [4]. This QoS architecture allows for provisioning of network bandwidth by implementing a variety of traffic control functions which can be combined in a modular way. We used this architecture as a basis for our implementation and experiments, since this platform is highly flexible and lends itself to modification and experimentation. We added functionality to do State Maintenance, QoS regulation, and Window Control.

A. Background on Linux QoS

The linux IPv4 networking stack is implemented in a layered fashion. Different layers comprising the stack and the interactions between them, are shown in Figure 5. Packets arrive through the medium and the Network Interface Card (NIC) takes care of Medium Access Control (MAC) and physical carrier sensing. After the NIC has received a packet successfully without errors, it signals this event via a hardware interrupt to the Operating System (OS). The OS allocates buffers for this newly arrived packet, copies them from the NIC to the kernel memory, and signals a soft interrupt (or kernel bottom half) – this portion is generally the *data-link layer* of the stack. The traffic control is implemented on the outgoing interface, as shown by the shaded box in Figure 5.

The main components of Linux QoS architecture are *queuing disciplines (qdiscs)*, *classes*, *filters*, and *meters*. When IP queues a packet on the outgoing interface, the packet is matched against the available filters, or classifiers. Filters have priority associated, so a filter with higher priority is matched first. The purpose of filters is to *classify* the packet into some traffic class. Each class owns a physical queue – to actually hold packets after the filter has assigned the packet to a class. After being queued in this qdisc, it is the qdisc's responsibility to eject the packet. The ejection may be based on rate disciplines like token bucket filters, priority filters, round robin, or our window control.

The kernel part of our code was implemented such that

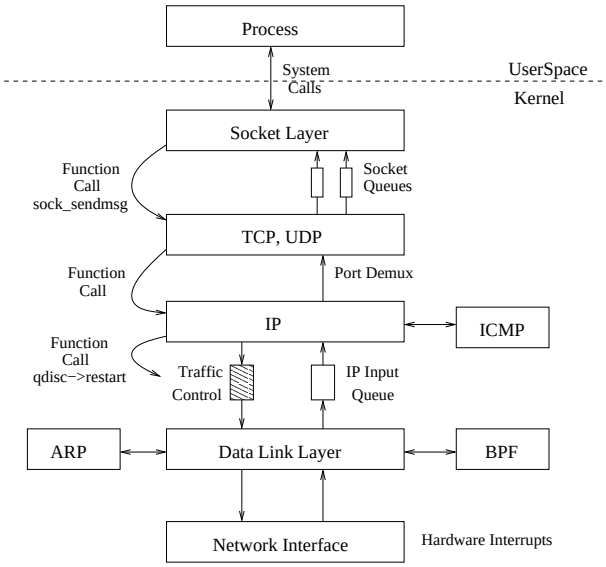

Fig. 5. Linux Network Stack Implementation

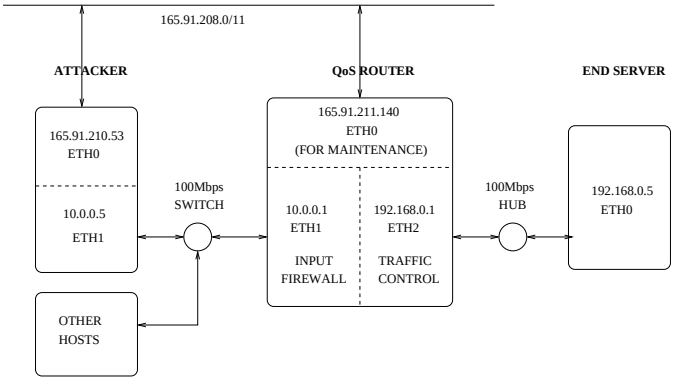

Fig. 6. Topology of Test Network

it ran whenever the filter matching code of Linux QoS executed. At that point, we are able to see whether the filter matched or failed. Accordingly, we updated the per-resource statistics that we maintain in the kernel.

B. Our Implementation and Testbed

Our test-bed consisted of a router, to take the role of a *Bastion Host* or *QoS regulator*, through which all traffic to the end server was routed (Figure 6). The end server may well be a corporate server network or cluster. The role of the QoS regulator may be played by a corporate firewall, an ingress router, or just a proxy sitting in front of a bunch of machines.

To test *Rate Control* with Linux QoS, a private network was set up. Machines were assigned to take the role of an Attacker, a QoS Regulator, and an End Server. The end server was intended to be used as a victim for various kinds of attacks, while at the same time providing some useful service to good clients. The test network was interconnected with 100 Mbps ethernet hardware. Maintenance ports were just a means to download code onto the test machines and collect statistics.

A fast machine (500 MHz Pentium) was chosen for the attacker to enable it to carry out an attack faster and more efficiently without getting slowed down by a slow CPU. A slow machine (133 MHz Pentium) was deliberately chosen for the victim because it allowed us to see the effects of overload more clearly. All machines were equipped with the Linux 2.2 kernels.

C. Window Implementation

The Window Control was implemented on top of the Linux 2.2.14 kernel. Packets going out on the outgoing network interface, are passed through classifiers just before they join the output queue on the outgoing interface. The number of resources being monitored is decided administratively. If the entry table is full (that is, the window limit for the current resource has been reached), the current packet would not be forwarded to the server (dropped at the bastion host).

The user space Linux traffic control module – *iproute2/tc* – was also modified to include an interface for our window control. Syntax for window control was added so that the administrator could specify the window limit without recompiling the kernel each time.

C.1 Maintaining State in Kernel

To enforce window control, varying amount of state is needed in the network – preferably at the bastion host (Figure 2). We keep this state information in kernel variables and data structures on the bastion host. The state maintained depends on the kind of resource being regulated and the kind of regulation (rate or window control).

For the TCP SYN regulation, the state entries are hashed based on the initial sequence number of the connections. A timeout value specifies the time when the entry should be purged. We set this limit to a small limit of 5 seconds. In the default TCP implementation, this timeout is anywhere between 75 and 450 seconds. Information is also kept about the source IP, source port, destination IP, and destination port as this required to flush the state and reset stale connections.

For CPU intensive CGI packets, we also maintain a similar table and a per connection entry keeping track of source IP, source port, destination IP, and destination port. However, in the case of CGI, so much state is not required. We could simply keep a single counter counting the number of CGI scripts on the system at any given time. This is akin to the *window_size* variable that TCP protocol keeps. Whenever a CGI script enters the system, the state counter is incremented by one. When a corresponding ack comes back from the server indicating that the process is complete, the counter is decremented by one. If the window limit is reached, no more CGI scripts are allowed into the system until the server sends more acknowledgments.

If there are several of CGI scripts, all of varying cost, we could maintain different traffic classes for different CGI scripts and do window control on each of them. There is a

trade-off between the amount of state maintained and the granularity of control.

C.2 Resetting System State

Each state entry has a timer associated with it. In case of TCP SYN requests, the timer is a small number based on the approximate RTT expected. Usually it should not take more than a few seconds for the TCP handshake to complete. We choose the timeout period to be 5 seconds. After timeout, the system recovers by injecting a TCP reset packet (RST flag set) to the end server. This has the desired effect of freeing uselessly held resources – in this case, protocol state buffers. The system also flushes its local entry from the state table. In normal cases, an acknowledgment coming back from the client cleans this state as this indicates that the TCP SYN handshake is completed.

To reset system state, a system call is made from inside the kernel. This call constructs a raw socket with source and destination set from the state table entry. The raw packet is of the correct type (TCP) but has the RST flag set. This has the effect of clearing the protocol state buffers that were held at the end server.

IV. Results

A. Results with Rate Control

Several experiments were conducted on the testbed to test out Linux rate control strategies and their effectiveness in controlling denial of service type attacks. Various parameters like achieved throughput, goodput, and CPU load were measured under normal conditions and under overload. We now present some results on how well the QoS Rate Control is able to control the load on the end servers.

A.1 Received packets/sec versus QoS Rate limit in Mbps

Fig. 7 shows the received UDP throughput in packets-per-second as a function of rate control applied on the QoS regulator. The x-axis shows increasing limits for the UDP class of traffic. The bastion host itself is always receiving 100Mbps. From the figure, it is seen that for MTU size (1500 byte) packets, the end server is able to perform best – receiving the entire UDP load of 100 Mbps. As the packet size decreases, peak receive rate also saturates. After hitting a peak, the peak received rate actually starts dropping. This is because at the peak received rate, the server has already hit a bottleneck in terms of processing power available to service the network load. Beyond this point, it starts spending more time at just servicing network interrupts and the received packets never get serviced by the kernel. In other words, a lot of resources are wasted in doing hi-priority network interrupt processing while less and less resources are available to deliver the successfully received packets to the application. It is also seen that the per-packet processing cost increases slightly as packet size increases. Roughly, about 10K packets can be processed per second. If the end server wants at most, say, 20% of the processing power to be consumed, then it could limit UDP packets to 2K packets per second.

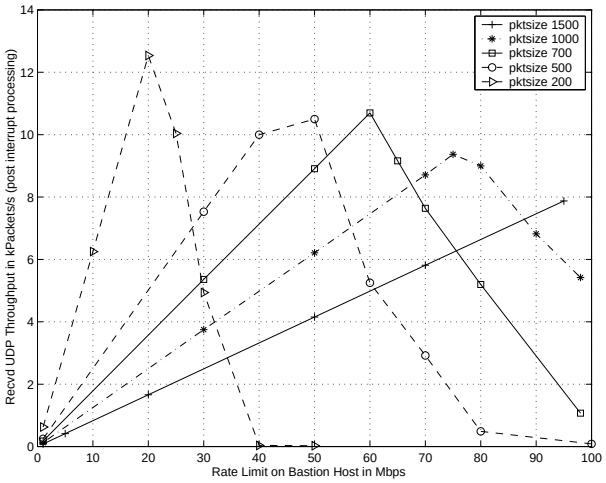

Fig. 7. UDP Goodput measured in Packets per second

A.2 Preferred flow in presence of UDP flood

In this experiment, we constructed two traffic classes – a preferred flow and a bulk UDP traffic class. The preferred flow maintained a constant sending rate at MTU size (1500 byte) packets. The attacking machine, on a different input link of the bastion host, kept sending a constant UDP flood at network capacity of 100Mbps. The QoS rate limit for the UDP class (on the attacker line), was varied on the regulator. The packet size of UDP flood packets was also varied over 500, 1000, and 1500 bytes. Sustained rate of the good flow with 500 byte packets is shown in Figure 8.

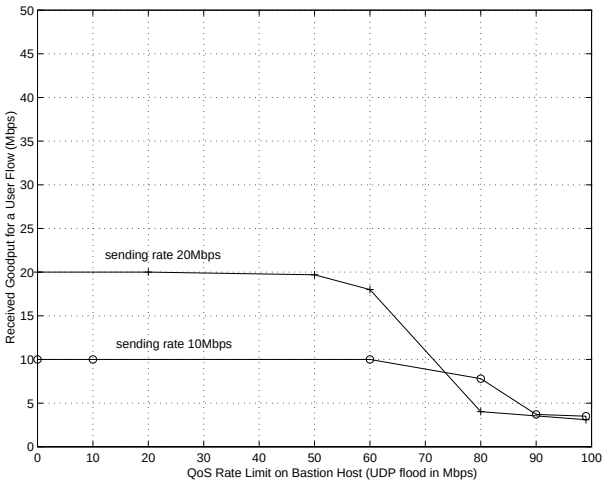

Fig. 8. Good flow in presence of UDP flood (MTU 500 bytes)

It is seen that the preferred flow is able to sustain its sending rate if the UDP flood class is limited to a certain point. After that the UDP flood starts consuming too many network and server resources, that the goodput of

the preferred flow starts falling. In other words, the preferred flow starts to get less service against the UDP flood. If we maintain a rate limit of, say 40Mbps for the UDP traffic, the preferred flow would always get its full requested rate. Fig. 7 and 8validate the utility of rate controlling the UDP traffic.

A.3 Preferred flow in presence of ICMP flood

A similar experiment was performed with a preferred flow in competition with an ICMP flood. The preferred flow kept sending packets at a constant rate. The attacking machine kept sending a 100Mbps flood of ICMP packets of packet size 1000 bytes to the end server. The rate limit on the QoS regulator was varied and the results plotted in Figure 9. It is seen that if there were no rate control, the preferred flow would only get a bandwidth of 25Mbps when the sending rate is 40Mbps. The preferred flow is able to get full bandwidth provided the ICMP load is less than 20Mbps. This result again helps in formulating policies on the QoS regulator.

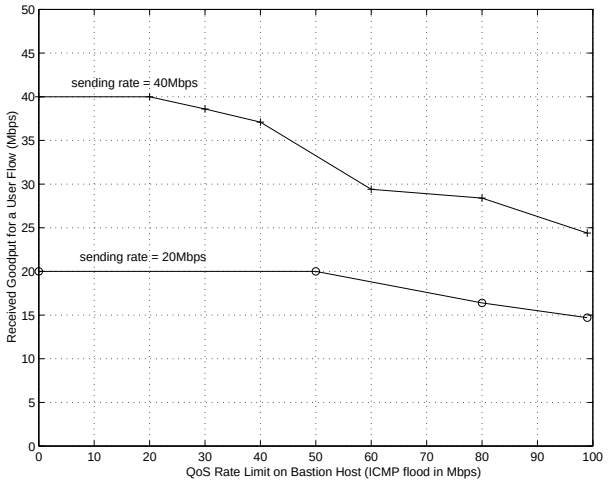

Fig. 9. Good flow in presence of ICMP flood

B. Discussion on Rate Control Results

These results give us an important idea of *how* to implement rate control at the QoS router. They give us an important understanding of the load characteristics of the end server. Our experiments also show that rate limit works for UDP and ICMP based floods. Some of the issues encountered in our experiments are discussed below:

B.1 Byte Counting versus Packet Counting

An important issue when enforcing rate control is to decide when to use byte counting and when to enforce limits in terms of packet counting. As we have seen, this may be important when the per packet processing overhead is not deterministic or the cost varies widely over different types of packets. As an example, the per packet overhead of a 40 byte SYN packet is much more than an equivalent 40 byte UDP or a 40 byte ICMP packet. In the case of a SYN, a lot of work has to be done in the protocol stack to save protocol state buffers, timers, and other bookkeeping related tasks. When flows are mapped to classes, it is important to consider the per-packet processing-cost in order to do a fair allocation of end server resources across flows. Our experiments have shown that it may be better to set a packets/sec rate limit than a bytes/sec rate limit. The packet rate limit, however, may depend on the class of traffic (ICMP vs. UDP) being regulated.

B.2 Interrupt Processing Overhead

Network packet processing can be classified into the initial interrupt processing overhead (which cannot be avoided), and a lower priority handling of received buffers by the kernel. The bottom half routines examine the packets received over the network and assign them to queues to be later handled by IP, TCP, UDP or other processes that constitute the network stack. If the end server is overloaded, it may start spending a higher percentage of time in processing network interrupts since interrupts have a higher priority [5], [6], [7]. If other resources like memory or CPU on the server are running low, these buffers may be dropped before they get a chance to be serviced by the kernel's bottom half handler and the network stack. This phenomenon can be observed in Figure 7. Hence, there is need for a "Bastion Host" to regulate QoS to end server cluster in order to protect the end servers from getting swamped by interrupt processing of arriving packets.

C. Results from Window Regulation

We now present results from using Window Control for fixed resources. We chose the TCP SYN flood and CGI flood as two test cases to experiment with our window control implementation as these attacks could not be contained by the rate control mechanisms.

C.1 Window Control on TCP SYN packets

Experiments were conducted on our testbed to see how the system behaved in presence of real attacks. The first attack to be studied was the TCP SYN attack. The end server configuration was set to a maximum backlog queue of 128 which is what most servers are set to. The maximum SYN timeout on the end server was set to the default value (75 or 450 seconds).

Two attack scenarios were constructed – a fast attack (Figure 10) and a slow attack (Figure 11). The number of available SYN buffers available on the end server were monitored as the attack progressed. On our system, a window limit of 64 was specified for this class of traffic (SYN packets going to port 80 of end server). In addition, a timeout value of 5 seconds was set for stale SYN entries.

It was seen that the window control was able to correctly enforce the limit of 64 packets which is half the total SYN bandwidth available on the end server. This is good since the other half can be used by for example, good flows coming on another interface of the ingress router. Also the timeout feature ensures that unnecessary buffers are recovered after 5 seconds.

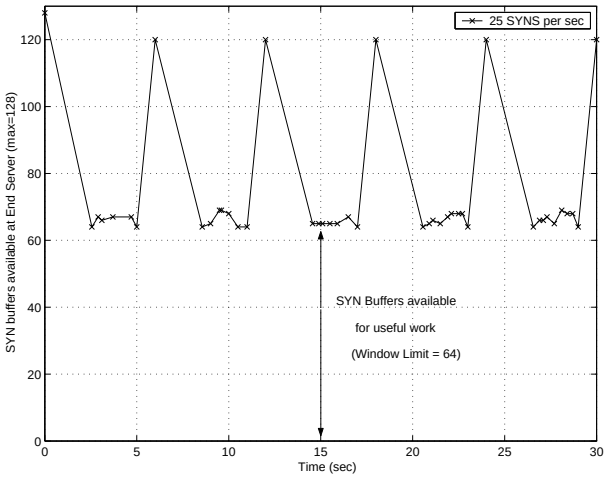

Fig. 10. Window Control with Fast SYN Attack

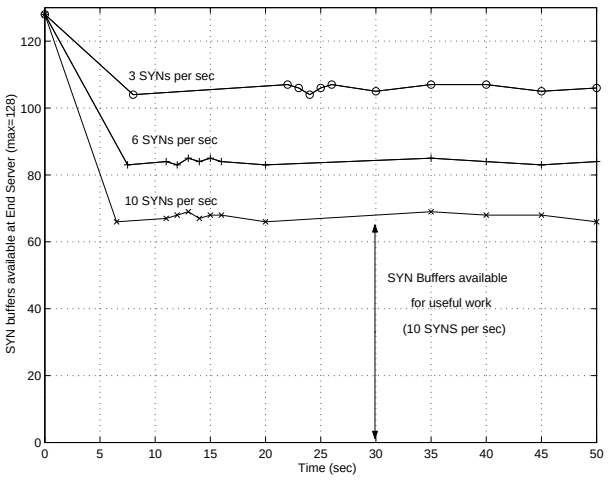

Fig. 11. Window Control with Slow SYN Attack

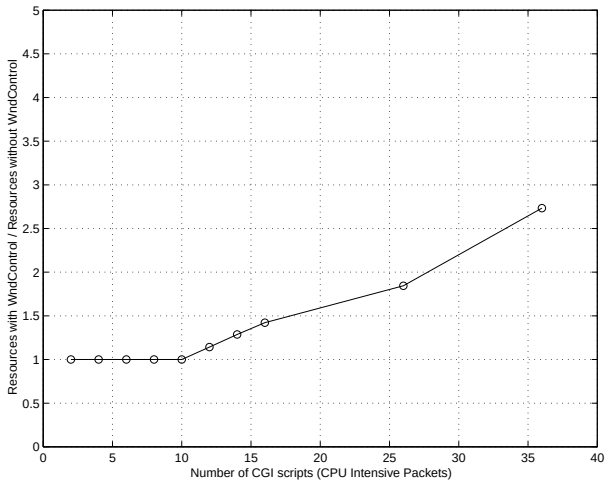

Fig. 12. Advantage Gained with Window Control

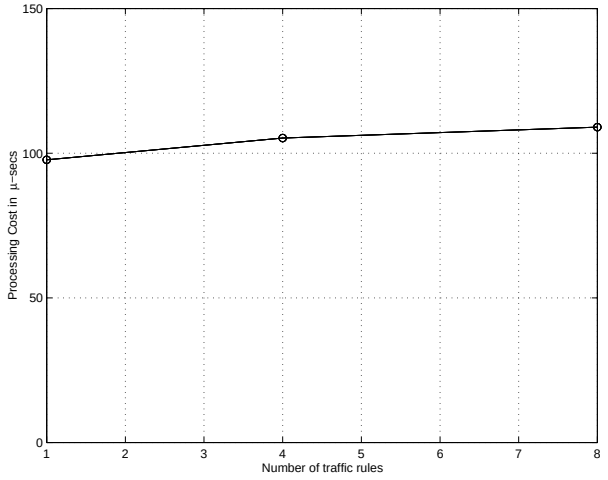

Fig. 13. Processing Cost of IP Forwarding versus Number of Rules

In case of a slow attack, the timeout kicked in before the attack SYN packets had an opportunity to deplete their complete window. Hence the number of available buffers for good flows is more than the 64 which was reserved for them.

C.2 Window Control for CPU Intensive Packets

To see the effect of CPU intensive packets, experiments with expensive CGI packets were performed. These CGI requests executed a mock Dhrystone benchmark running at low priority on the end server. If there was no window control, any number of these expensive processes could be fired by a malicious user. The advantage gained in terms of available server CPU resources is plotted in Figure 12. As higher number of CGI scripts arrive, our window control does not forward then to the end server and hence higher CPU power remains available for other applications.

D. Performance Measurements

We present some performance measurement results of the overhead of our implementation.

We first measured the cost incurred by a forwarded packet as it passes through a vanilla router that does not have any of the QoS queuing disciplines built into it. This cost was about 26 μ-secs. When Linux QoS in enabled, the cost goes up to about 95 μ-secs. This cost is not really high considering the router used in our experiment was a small 233 MHz general purpose Pentium CPU. Even at 95 μ-secs, the machine is able to handle about 10,000 packets per second. The vanilla router is fast because it does not need to do any classification, queuing, etc – it can simply queue the packets on the outgoing device when IP passes the packets after processing. There are also no scheduling delays involved.

It was noted that even after enabling the queuing disciplines, the incoming path was not changed. The additional cost is added only on the outgoing path on which queues

are enabled.

Figure 13 shows the additional cost incurred by our implementation. The cost only marginally increases from 95 μ-secs to about 105 μ-secs. The additional cost comes from the extra state maintenance code and the occasional system call to flush the state. The cost of a single system call was measured to be about 2000 clock cycles or about 8.5 μ-secs. Figure 13 plots the processing cost per packet as number of traffic rules is increased. It is seen that the marginal increase in cost is not significant.

V. Related Work

Past work on resource control can roughly be divided into three categories – operating systems based solutions, network centric solutions, and middleware solutions. Our solution roughly falls in the network centric solution category although it is fully transparent to the end server.

Operating System based solutions are usually invasive and require a lot of support from the operating system. The SCOUT operating system [8] is an example. Such OS level support includes, but is not limited to, keeping an account of network packets as they move through various layers of the OS. The Scout OS introduces a path abstraction to demarcate clearly the resources consumed by an I/O path. As soon as a network packet enters the OS, it is assigned to a resource principal. All resources accessed by the packet – be it system calls, interrupt handlers, or user space processing – are charged to this principal. Not very different from this concept is the general idea of *resource containers* introduced by Banga et. al. in [9]. The QGuard solution [10] is implemented on the end server also relies heavily on OS support to do monitoring and load-control. Unlike QGuard however, our solution is totally transparent to the end servers. Also wasteful interrupt processing is relegated to the bastion host.

Middleware Solutions [11] allow a common interface to be exported to all applications. They provide some level of isolation between applications and broker resources for the applications. In order to provide middleware support, the operating system (or network subsystem) has to be capable of understanding the middleware primitives. In other words, middleware solutions and OS based solutions require changes to the OS.

Network Based solutions [1],[2] are more flexible since the network subsystem is already separate from the way the operating system allocates resources. In [1], for example, the network is monitored by a sniffer machine to identify fake TCP SYN packets. The destination is noted and an RST packet is constructed and dispatched for the victimized host, thereby releasing the held up buffers immediately. In the Packeteer approach [2], TCP acknowledgments going back to the client are paced to regular intervals, in order to let the original flow back-up and stick to the rate defined for its traffic class. This approach would not work for protocols like UDP which do not respond to congestion.

Network centric solutions may lack total control over resources and hence may be unsuitable for hard realtime applications when absolute guarantees are required from the operating system.

VI. Conclusions

This work was a preliminary study to see the effectiveness of keeping an aggregate account of resources available on the network. This state was maintained in a central place in the network on a bastion host which could also be the ingress router or the firewall.

Traffic was divided into traffic classes based on the resources it consumed – like classes of traffic which consume protocol state buffers, class of CPU intensive packets, UDP packets, ICMP packets, etc. Then per-resource policing was enforced on the bastion host so that each class of traffic stayed within its resource limit. This solution provides some resistance against Distributed Denial of Service attacks in which per-flow policing is not effective. Since resources within a class are policed, the attack cannot take away resources from another class. If an attack consumes resources in multiple classes – then our scheme might not be able to provide protection in all classes simultaneously.

An important application of our platform could be *load-balancing*. Since we maintain a picture of current resource availability on end servers, we can use this information to direct traffic based on load criteria. The per-resource state-information maintained on the bastion host can be used to find out lightly loaded end servers. More traffic may then be directed to these lightly loaded servers out of a server farm.

Future work would concentrate on doing better resource management *within* a class. So, if an attack is going on within a class A of traffic, and a resource R is being consumed as a result, we want to study how we can partition resource R such that the attacker does not eat up all of resource R, and good clients can still get some service. Also, future work would look at how to set more accurate limits on aggregate resources taking into account overload characteristics of the network and of the servers.

References

[1] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. Spafford, Aurobindo Sundaram, and Diego Zamboni, "Analysis of a denial of service attack on TCP," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 1997, pp. 208–223.

[2] "Packeteer Inc. White Papers," `www.packeteer.com/solutions/resources`.

[3] Werner Almesberger, Jamal Hadi Salim, and Alexey Kuznetsov, "Differentiated Services on Linux – Internet draft," 1999, `draft-almesberger-wajhak-diffserv-linux-01.txt`.

[4] Bob hubert, Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, and Paul B Schroeder, "Linux 2.4 Advanced Routing HOWTO," 2000, http://linuxdoc.org/HOWTO/Adv-Routing-HOWTO.html.

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

[6] J. Mogul and K. Ramakrishnan, "Eliminating Receiver Livelock in an Interrupt-Driven Kernel," in *USENIX Technical Conference*, San Diego, CA, 1996.

[7] Mohit Aron and Peter Druschel, "Soft timers: efficient microsecond software timer support for netw ork processing," in *17th*

ACM Symposium on Operating Systems Principles, December 1999, pp. 232–246.

[8] Oliver Spatscheck and Larry L. Petersen, "Defending Against Denial of Service Attacks in Scout," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, Louisiana, Feb. 1999, Available online at `http://www.cs.arizona.edu/scout/Papers/osdi99.ps`.

[9] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Operating Systems Design and Implementation*, 1999, pp. 45–58.

[10] Hani Jamjoom, John Reumann, and Kang G. Shin, "QGuard: Protecting Internet Servers from Overload," Tech. Rep., University of Michigan.

[11] "Hewlett packard corp. webqos technical white paper," `www.hp.com/products1/webqos`.

[12] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "TCP Congestion Control with a Misbehaving Receiver," *ACM Computer Communications Review*, October 1999.

[13] E. Anderson, "The Magicrouter, an Application of Fast Packet Interposing," `http://www.cs.berkeley.edu/eanders/projects/magicrouter`, May 1996.

[14] Gaurav Banga and Peter Druschel, "Measuring the Capacity of a Web Server," in *USENIX Symposium on Internet Technologies and Systems*, December 1997.

[15] J. Mogul, R. Rashid, and M. Accetta, "The Packet Filter: An Efficient Mechanism for User-Level Network Code," in *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, November 1987, vol. 21, pp. 39–51.

[16] Ariel Cohen, Sampath Rangarajan, and J. Hamilton Slye, "On the Performance of TCP Splicing for URL-Aware Redirection," in *USENIX Symposium on Internet Technologies and Systems*, 1999.

[17] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich M. Nahum, "Locality-Aware Request Distribution in Cluster-based Network Servers," in *Architectural Support for Programming Languages and Operating Systems*, October 1998, pp. 205–216.

[18] Oliver Spatscheck, Jorgen S. Hansen, John H. Hartman, and Larry L. Peterson, "Optimizing TCP forwarder performance," *IEEE/ACM Transactions on Networking*, vol. 8, no. 2, pp. 146–157, 2000.