MITIGATING DENIAL OF SERVICE USING QOS REGULATION

A Dissertation

by

AMAN GARG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

June 2001

Major Subject: Electrical Engineering

MITIGATING DENIAL OF SERVICE USING QOS REGULATION

A Dissertation

by

AMAN GARG

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

A. L. Narasimha Reddy (Chair of Committee)	Riccardo Bettati (Member)
Nitin H. Vaidya (Member)	Chanan Singh (Head of Department)

August 2001

Major Subject: Electrical Engineering

MITIGATING DENIAL OF SERVICE USING QOS REGULATION

A Dissertation

by

AMAN GARG

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

A. L. Narasimha Reddy (Chair of Committee)	Riccardo Bettati (Member)
Nitin H. Vaidya (Member)	Chanan Singh (Head of Department)

August 2001

Major Subject: Electrical Engineering

MITIGATING DENIAL OF SERVICE USING QOS REGULATION

A Dissertation

by

AMAN GARG

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

A. L. Narasimha Reddy (Chair of Committee)	Riccardo Bettati (Member)
Nitin H. Vaidya (Member)	Chanan Singh (Head of Department)

August 2001

Major Subject: Electrical Engineering

ABSTRACT

Mitigating Denial of Service using QoS Regulation. (June 2001)
Aman Garg, B.Tech, Indian Institute of Technology Kanpur, India
Chair of Advisory Committee: Dr. A. L. Narasimha Reddy

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 and fooling the underlying system into believing as if these malicious packets were legitimate. An excess of these legitimate packets can cause overload conditions in the network and on the end servers. The purpose of this research 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 is basically the marriage of two seemingly different paradigms – Denial of Service, with Quality of Service. The solution has two parts. The first part is to maintain sufficient state information *per resource* in the network layer, 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 flush the state, recover excess resources (if they are believed to be held by malicious flows), and regulate the movement of traffic. A *Window Regulation* based scheme is proposed and implemented to do the regulation of traffic. It is shown that this scheme works better than conventional rate-based QoS regulation.

To my parents

ACKNOWLEDGMENTS

I am highly thankful to my advisor, Dr. A. L. Narasimha Reddy, for guiding me through my research. His patience in explaining things out to me and putting things in context was highly helpful.

I am also grateful to my labmates, Ravi Wijayaratne, Xiaonan Ma, and Ikjun Yeom for numerous discussions I have had with them. Xiaonan helped me understand a lot of finer details of the Linux kernel networking stack, and Ravi was the pioneer Linux kernel hacker of our group.

I'm grateful to my parents for having faith in me and providing me the background motivation all through my life. Their sacrifice in sending me so far away from home in pursuit of my career is really commendable.

I would also like to thank all my friends at College Station, in particular my roommate Vivek, my other KGP friends, my Austin friends, and my dear Wingmates (aka *ftp-iitk*) – all of whom provided me the cheer and enthusiasm through my tenure as a grad student.

TABLE OF CONTENTS

CHAPTER Page

I INTRODUCTION 1

1. Problem Statement 2
A. Background on Denial of Service Attacks 2
B. Resource Control and QoS 5
C. Related Work 6
D. Motivation 8
E. Methodology 9
F. Organization 11

II IMPLEMENTATION BACKGROUND 13

A. Design of the Linux IPv4 Network Stack 13
1. Socket Buffers *sk_buffs* 14
2. Interrupt and Bottom Half Handling 15
B. Linux Traffic Control 16
1. Queuing Disciplines and Schedulers 17
2. Classes 17
3. Filters 17
C. Design Space 18

III QOS REGULATION WITH RATE CONTROL 19

A. What is QoS Regulation? 19
1. Controlling traffic with Bastion Host 20
B. Traffic Control Mechanisms 22
C. Rate Control to do QoS Regulation 23
D. Our Implementation and Testbed 24
E. Measurements and Results 26
1. Benchmarking 26
2. UDP Goodput versus QoS Rate limit 27
3. CPU Load versus QoS Rate limit 29
4. Preferred flow in presence of UDP flood 30
5. Preferred flow in presence of ICMP flood 32
6. Rate limit on CPU Intensive Packets 33
7. Rate limit on TCP SYN packets 34

CHAPTER Page

F. Discussion on Results 35
1. Byte Counting versus Packet Counting 36
2. Interrupt Processing Overhead 36
3. Why simple rate control is not sufficient? 37

IV RESOURCE ALLOCATION AND WINDOW REGULATION . 38

A. What are Resources? 38
B. The Big Picture 38
C. QoS with Window Regulation 40
1. When is Window Regulation Appropriate? 40
2. Window Control 40
D. Implementation Details 41
1. Maintaining State in Kernel 43
2. Resetting System State 44
E. Results 45
1. Window Control on TCP SYN packets 45
2. Window Control for CPU Intensive Packets 46
3. Performance Measurements 48
F. Conclusions 50

V CONCLUSIONS AND FUTURE WORK 51

REFERENCES 53

VITA 59

LIST OF TABLES

TABLE Page

I Configuration of Machines Used 25

II Configuration of Test Network 26

LIST OF FIGURES

FIGURE		Page
1	ICMP Directed Broadcast (smurf) Attack	4
2	Linux Network Stack Implementation	14
3	Linux Traffic Control Location	16
4	Queuing Disciplines, Classes, Filters	17
5	Resource re-distribution for desired QoS	20
6	QoS Regulator or Bastion Host	21
7	QoS Regulation by Rate Limit	23
8	Topology of Test Network	25
9	Basic Benchmarking of Machines with Dhrystones	27
10	UDP Goodput as a function of QoS ratelimit	28
11	UDP Goodput measured in Packets per second	28
12	CPU Load on End Server as a function of QoS ratelimit	30
13	Good flow in presence of UDP flood (MTU 500 bytes)	31
14	Good flow in presence of UDP flood (MTU 1000 bytes)	31
15	Good flow in presence of UDP flood (MTU 1500 bytes)	32
16	Good flow in presence of ICMP flood	33
17	Rate control on CGI requests (1 request/sec)	34
18	Rate Control with TCP SYN packets	35
19	Aggregate View of Resource Consumption	39

FIGURE Page

20 Window Control Overview 42
21 State Maintenance for SYN connections 43
22 State Maintenance for CGI Packets 43
23 Window Control with Fast SYN Attack 45
24 Window Control with Slow SYN Attack 46
25 Window Control with CPU Intensive loads 47
26 Advantage Gained with Window Control 47
27 Per Packet Processing Cost 48
28 Processing Cost versus Number of Rules 49

CHAPTER I

INTRODUCTION

Recent times have seen a plethora of 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, but which work because the attack packets are in no way differentiable from normal traffic. The problem in the latter 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.

Quality of Service (QoS) is an overused term which could be used to mean various things in different contexts. For example, in networks, the term usually implies certain guarantees on the kind of service a flow or stream of packets receives. These guarantees could be statistical or deterministic, to imply soft or hard guarantees, respectively. Alternatively the specifications could be made from a user's perspective rather than the network's perspective - for example, a user may be interested in the 'smoothness' of a video transmission, rather than system parameters like packet latency, throughput, or end to end jitter which actually affect that smoothness.

The link between QoS and DoS is sometimes not so obvious and we attempt to explore this relation in this research. While Quality of Service is a measurement of the perceived quality, be it from a user's perspective or from the network's, Denial

---

The journal model is *IEEE Transactions on Automatic Control.*

of Service is a means of disrupting that quality. In other words, DoS is an acid test to find out if the underlying system can provide the requested QoS even when some malicious activity is going on. This brings us to face new challenges which the Quality of Service infrastructure on a system must address, and this is the problem explored in the current work.

1. Problem Statement

More succinctly, the problem can be specified thus – Denial of Service is a malicious way to consume resources in the network, the end host or the server network, and the system as a whole, thereby denying service to other legitimate users. This renders the system unusable because it cannot do productive work. The challenge is to design a scheme that tries to identify and successfully curb a variety of these attacks by following a common general strategy applicable to various scenarios. Quality of Service is a kind of negotiation process that allows clients to request resources and provides certain kind of end to end guarantees. The underlying system is able to do this by means of policing and admission control. However, most implementations of QoS are not geared to effectively handle malicious users. Our research identifies this problem and suggests a way to keep track of server and network resources in an attempt to mitigate the effect of such attacks. By keeping a snapshot of available system resources right at the network layer, our scheme is geared to effectively regulate resource consumption in the face of such attacks.

A. Background on Denial of Service Attacks

Denial of service (DoS) attack is a malicious attempt to prevent a server from providing service to legitimate users, by depleting one or more of the resources that are

at the disposal of the server. Unlike other attacks, this attack is not meant to alter, steal, or damage the data. The intent is to bring down services, effectively denying service to legitimate users. In general, is is harder to prevent against such attacks because there is no concrete way to distinguish between a genuine user and a malicious user. This section lists the common types of denial of service attacks.

TCP SYN Flood : SYN flood is by far the most popular of all DoS attacks. This attack is TCP specific. During a TCP 3-way handshake, the client sends a SYN packet with an initial sequence number. The server sets up data structures in anticipation for a new tcp connection, sends an acknowledgment for this SYN back to the client, and also a SYN from its side, containing the sequence number that the server wishes to use. Until this stage, the connection is in half-established or SYN_RECVD state. If the client does not send back an acknowledgment for whatever reason, the data structures will be held up in the server until they time out – typically after 75 to 450 seconds. A malicious client may thus force a server to waste several megabytes of protocol state buffers and kernel memory. SYN attack is a form of an asymmetric attack where the client does not need to spend much CPU power, or memory, or even network bandwidth, but can still bring down services on a relatively powerful server.

UDP Flood : UDP is a connection-less protocol that has no notion of flow control. Hence it is, in principle, possible for a misbehaving client to send arbitrarily large amount of traffic to a victim host. The receiving machine would be forced to receive this traffic, perhaps only to throw it away later. But by the time the server decides that the packets are junk, it would already have spent substantial system resources in receiving those packets. Unlike a TCP SYN attack, the idea of a UDP flood is to eat up both CPU and available network bandwidth.

ICMP Directed Broadcast : Also called the *smurf* attack, this attack is meant to consume network bandwidth. The two main components of this attack are the use of forged ICMP echo request packets and the direction of packets to IP broadcast addresses. Intermediate routers are also affected in this attack.

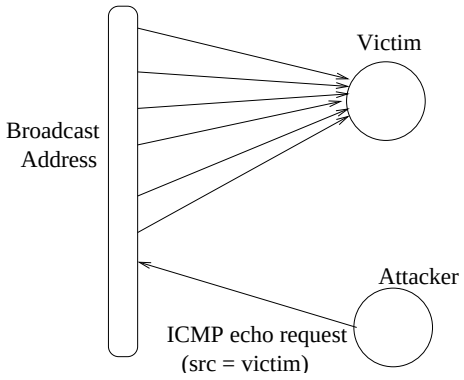

Fig. 1. ICMP Directed Broadcast (smurf) Attack

ICMP Flood Ping : Known by various names like 'Ping of Death', etc, this attack involves sending a flood of ICMP echo request packets. The aim is almost similar to a UDP flood – to eat up network bandwidth and CPU. In a distributed attack, a master could instruct a large number of daemons to synchronize the sending of pings to a victim. A flood ping could be pretty heavy on a network and especially so in the case of large size ping packets which subsequently get fragmented by the network.

DNS Flood : A zone transfer 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. This is yet another way to generate a large amount of data rather easily and at little expense to the attacker. The network and the targeted servers have to bear the consequences.

Fragmentation Attack : This could be used with any of the attacks mentioned above. The idea here is to send very large number of packets as very small fragments. When the receiving machine tries to put together these packets in the IP stack, it will eat 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 eat 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.

B. Resource Control and QoS

Resource Control and QoS are closely tied to each other. By managing one, we can automatically manage the other . The problem of resource control is much older than networks and occurs as a classical operating systems problem. We restrict ourselves to the domain of managing server and network resources in overload situations as when a flood based attack is under way. We do not talk about managing resources, for example, to to do deadlock resolution or handle priority inversion in the system. In our context, resource control means that we want to do what we can at the *network layer* to ensure that applications, flows, and server threads keep getting the QoS they need to get a 'good' response from the server. The same problem can be addressed in the context of process scheduling in an operating system as well. Traditionally, a multilevel priority feedback scheduler which does priority aging has been seen to be effective to make sure that a runaway process does not consume all the CPU while

other processes are starved. However there is no context of *priority class* or *best effort class* in the context of general operating systems.

Our focus is to *extend resource control to the network subsystem.* The network layer is really the place to implement resource control since by taking corrective action here, we can affect how the other resources are consumed in the system. Resources could include network bandwidth, protocol state memory buffers, kernel interrupt processing, memory, or CPU cycles. Our solution really is to maintain a rough snapshot of available system resources in the network layer. As flows or packets keep consuming these resources, appropriate state entries are updated. For example, each time a request fro an expensive CGI request comes in, a record is made that the amount of processing capacity remaining int he end server has gone down by some unit. When a traffic class has exceeded its allocated quota, future packets belonging to this class are dropped.

C. 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 [1, 2, 3] 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. This is similar to the support an OS lends to do user-space process accounting. There is no proper support to do accounting for OS time spent in doing system functions – like system calls and

interrupt processing.

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 [4]. The QGuard solution [5] also relies heavily on OS support to do monitoring and load-control.

When realtime network guarantees are required, OS based solutions may be the only option. An extreme solution, when total partitioning of resources is required, would be to use a realtime operating system with separate budgets for each traffic class and hence ensuring total isolation in terms of consumed resources.

Middleware Solutions [6],[7] 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 are as restrictive as OS based solutions since changes to the OS are required.

Network Based solutions [8],[9] are more powerful and flexible since the network subsystem is already separate from the way the operating system allocates resources. In [8], 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 [9], 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. Since the network subsystem also consumes operating system resources (hi-priority interrupt processing for example), it may be better to implement this solution on a *Bastion Host* outside the server cluster, and route all traffic through this bastion host. This is our approach.

D. Motivation

The motivation for this work comes from the open question that what is it that allows resources in the network or on the end host to be so easily consumed to make Denial of Service attacks effective. People with background in Operating Systems would immediately think of solutions akin to processor scheduling in a typical operating system. Then there is a generalized area of research that deals with resource containers [4] and load balancing issues[10, 11, 12, 13]. Schemes have been proposed that do exhaustive resource accounting along a process's entire path through the system, an example of which is the SCOUT operating system [1, 2, 3].

A network is much more susceptible to misuse of its resources by malicious users because of the inherent distributed nature of a network. An analogy can be drawn to processor scheduling in operating systems. In a processor, the scheduler maintains multilevel hierarchical priority queues with feedback for all its processes, and there is process accounting to see that there is no runaway process on the system that is consuming all the cpu or other resources on the machine. Resource control is relatively simple in this case because of the centralized nature of the resources.

In a distributed scenario, the problem of resource allocation shows up, for exam-

ple, in designing a front-end for a load balanced server cluster. Abstract notions of resource containers [4] are sometimes employed to keep track of resource availability and consumption.

While OS-based approaches are effective, a number of attacks such as TCP SYN attacks succeed by consuming resources such as network stack data structures which are not monitored or regulated by the operating system. Also, attacks such as UDP and ICMP attacks succeed by utilizing the processor resources in high-priority interrupt mode. To counter such threats, *a network-centric approach is necessary.*

Currently most of these network-centric approaches have been attack-specific. For example, TCP SYN attacks are handled by TCP SYN cookies [14] or TCP SYN regulation and termination [8]. Similarly ICMP flood attacks are handled by turning off ICMP echo reply [15]. Turning off ICMP echo reply however still does not save the servers from wasting time in serving high-priority interrupts. It is possible to design a general approach that monitors multiple attacks and regulates multiple resources in an aggregate fashion. This is the problem we approach in this thesis.

E. Methodology

Recent Linux kernels (beginning with the 2.2 kernel) have included a Quality of Service architecture in the network stack [16, 17, 18, 19, 20, 21, 22]. This QoS architecture allows for provisioning of network QoS by implementing a variety of traffic control functions which can be combined in a modular way. The system is powerful enough to provide QoS support both in a differentiated services and integrated services setting. The primary focus of this architecture is to provide QoS in the network.

We used this architecture as a basis for our implementation and experiments, since this platform is highly flexible and lends itself to modification fairly easily. It

may be noted that this architecture is still under development and has not been fully incorporated into official releases of the Linux kernel. It may be noted that we just use the packet classification and user interface part of this architecture. We have added functionality to do State Maintenance, QoS regulation, and Window Control.

We first ran experiments on a real network test-bed in the lab to find out the shortcomings of this architecture for controlling flood based network attacks. Our test-bed consisted of a router, also referred here as a *Bastion Host* or *QoS regulator*, through which all traffic to the end host was routed. The end host may well be a corporate server network or cluster. This QoS regulator may be likened to a corporate firewall, an ingress router, or just a proxy sitting in front of a bunch of machines.

Simple rate-based control for limiting flows seemed to be ineffective. In addition there was no notion of feedback mechanism from the end host (or server cluster) to inform the QoS regulator that rate based schemes are not working on a certain flow.

To overcome these shortcomings, we proposed a *Windowing Scheme* and a notion of *implicit server feedback* at the Qos regulator. The purpose of these schemes is to dynamically handle the incoming flows based on implicit information of resources available in the end host. This is managed by maintaining resource allocation tables at the regulator and doing admission control and window regulation.

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, if protocol state buffers (like outstanding SYN buffers) is a resource on the end host that the regulator wants to control, we maintain a table hashed on the basis of TCP initial sequence number. This resource allocation table is

[1]QoS Regulation can be done on a per-flow basis also but that requires apriori information about client flows – like which domains are trusted and which are untrusted.

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. The QoS regulator is used to control 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 then maintains state information per resource and admits packets 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) indicating that more flows or packets can be admitted. 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 rules in the regulator.

F. Organization

The rest of the thesis is organized as follows. In Chapter 2, we take an in-depth look at our implementation background and Linux network QoS architecture which has been part of kernel versions 2.2 and higher since April 1999. In Chapter 3, we show how simple rate based control built on top of Linux network QoS is able to control some

of the load based attacks, while failing miserably in other cases. Chapter 4 introduces our extensions to Linux QoS architecture in the form of *windowing protocol* and other resource management techniques in the network stack. Complex resource allocation for SYN buffers and load based indices are possible using this mechanism. Results of window regulation for CPU intensive packets and TCP SYN packets are shown. Chapter 5 concludes with remarks about the contribution of this work and direction of future work.

CHAPTER II

IMPLEMENTATION BACKGROUND

A. Design of the Linux IPv4 Network Stack

The linux IPv4 networking stack is implemented in a layered fashion, modeled after the freeBSD implementation [23, 24, 25, 26]. The different layers comprising the stack are shown in Figure 2. Packets arrive through the medium and the Network Interface Card (NIC) takes care of Medium Access Control (MAC) and physical carier sense. 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 soft interrupt (or bottom half) runs whenever the kernel returns from a system call, or after a process switch. It is then that the next layer if the stack – usually IP or Internet Protocol – runs. This layer does all the work related to the IP protocol but without doing transport specific tasks like TCP and UDP. So defragmentation, handling ICMP messages, doing port de-multiplexing and delivering the packets to the transport layer is included in this layer.

The *transport layer* does specific transport level control. This includes handling flow control and end-to-end reliability semantics for TCP, and very little work for UDP. After this layer, the packets are queued in socket queues and the *socket layer* takes over. The socket layer provides a generic Application Programmer Interface (API) for the user in that it provides a lot of functions for the user to interact with the socket layer. These functions are actually system calls that cross the user-kernel

space boundary to deliver data and buffers in either direction. Copying of buffers is involved here.

On the downward path, the data is handed over from user space to kernel space though the use of socket API system calls. Once the data is in the kernel, it is handed down, layer by layer, through kernel function calls.

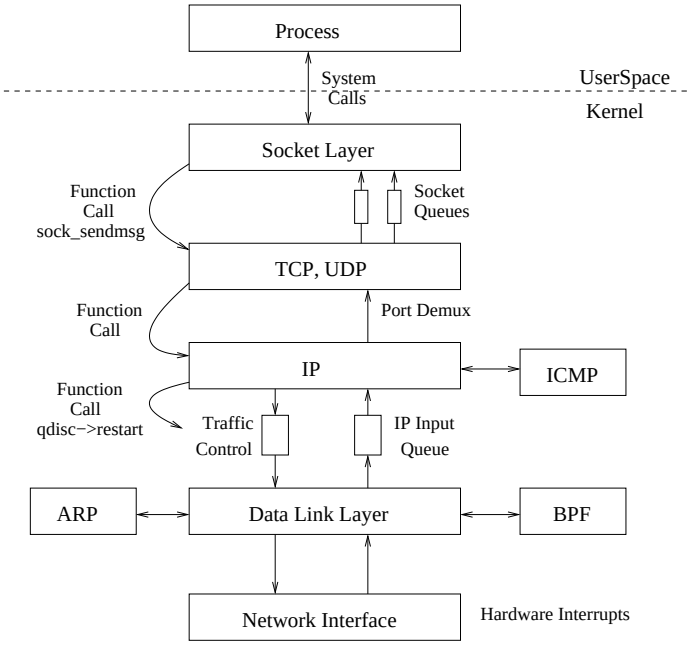

Fig. 2. Linux Network Stack Implementation

Figure 2 shows a schematic diagram of the linux IPv4 implementation.

1. Socket Buffers *sk_buffs*

One of the problems of having many layers of network protocols, each one using the services of another, is that each protocol needs to add protocol headers and tails to data as it is transmitted and to remove them as it processes received data. This make passing data buffers between the protocols difficult as each layer needs to find where

its particular protocol headers and tails are. Linux uses socket buffers or *sk_buffs* to pass data between the protocol layers and the network device drivers. *sk_buffs* contain pointer and length fields that allow each protocol layer to manipulate the application data via standard functions or "methods".

2. Interrupt and Bottom Half Handling

Interrupts are used to allow the hardware to communicate with the operating system. Here, a brief explanation of the principles governing the execution of an interrupt is given. There are two types of interrupts in Linux: fast and slow. Slow interrupts are the usual kind. Other interrupts are legal when they are being dealt with. After slow interrupt has been processed, additional activities requiring regular attention are carried out by the system - for example, scheduler is called as and when required. A typical example is the timer interrupt. Fast interrupts are used for the short, less complex tasks. While they are handled, other interrupts are blocked.

The variable "intr_count" (*kernel/softirq.c* of the source) is used by the kernel to keep track of the level of the interrupt nesting.

At certain times, the kernel wants to postpone work to a later time. An example of this is during interrupt processing. When the interrupt was asserted, the processor stopped what it was doing and the operating system delivered the interrupt to the appropriate device driver. Device drivers should not spend too much time handling interrupts as, during this time, nothing else in the system can run. There is often some work that could just as well be done later on. Linux's bottom half handlers were invented so that device drivers and other parts of the Linux kernel could queue work to be done later on [23].

B. Linux Traffic Control

Starting kernel version 2.2.0 [1], linux started to support a network QoS architecture in the network layer. The functionality provided is similar to that found on a dedicated network processor box, although the development effort is in its early stages. Articles [16, 17, 18, 19] describe this architecture in more detail. The traffic control is implemented on the outgoing interface.

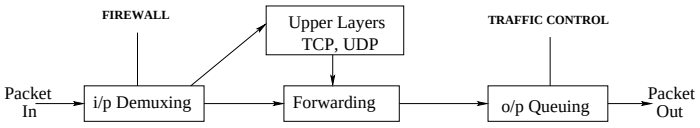

Fig. 3. Linux Traffic Control Location

The main components of this architecture are or queuing disciplines, classes, filters, and meters. These are discussed one by one below. 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 class. A class is a traffic class – we would be using the notion of a class to denote a group of flows that consumes the same bulk resource. 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 when it feels like. The ejection may be based on rate disciplines like token bucket filters, priority filters, round robin, etc.

[1] released June 1999

1. Queuing Disciplines and Schedulers

Queuing discipline (qdisc) is basically a queue that has a scheduler attached at the end. Example schedulers that linux implements are FIFO, Random Early Detection (RED), Token-Bucket Filter (TBF), Generalized RED with n-drop probability (GRED), Class Based Queuing (CBQ), n-Band Priority Queuing, Stochastic Fair Queuing (SFQ), DSCP Re-marker (DsMark),and CSZ (Clark Scott Zhang). Figure 4 shows the layout of filters, classes, and queuing disciplines. A queuing discipline can further have filters and classes to build another level of hierarchy.

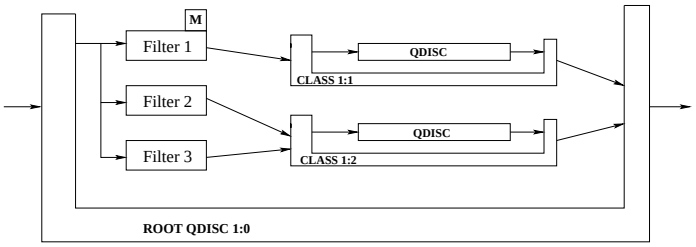

Fig. 4. Queuing Disciplines, Classes, Filters

2. Classes

Traffic classes are nothing but packets denoted by the same filter. When a filter matches an incoming packet, the packet is assigned to a class. Each class owns a queue (or qdisc), and it uses this qdisc to actually hold the packet before dispatching it over the network.

3. Filters

Filters are packet classifiers. Linux QoS provides various kinds of filters such as those based on u32 headers, rsvp messages, route decisions, or even firewall rules

and marking. By writing complex filters, any amount of parsing can be done on the incoming packets. These filters allow us, for example, to match all TCP packets containing the ack bit.

C. Design Space

Our design space consists of changes made in two parts – the *IProute2/tc* user space tool, and the in the kernel traffic control functions. Iproute2/tc is a user space tool to configure the Linux traffic control disciplines in the kernel. It communicates with the kernel through Netlinks socket API. We changed the sematics so that extra variables to take care of window control can be passed from user space to the kernel.

The kernel part of our code was implemented such that it runs whenever the filter matching code of Linux QoS runs. At that point, we are able to see whether the filter matched or it failed. Accordingly, we update the per-resource statistics that we maintain in the kernel. In the next Chapter, we see how rate control is partially effective in controlling some denial of service attacks. In Chapter 4 we will explain more on how this state information is kept after packets match one of the filters.

CHAPTER III

QOS REGULATION WITH RATE CONTROL

Perceived Quality of Service is dependent on how much overload the given system can absorb and still provide acceptable service. Most schemes that attempt to provide some kind of Quality of Service (QoS) for the end user, tend to do admission-control so that the underlying system does not go into overload situation.

A. What is QoS Regulation?

In order to provide predictable service, any infrastructure requires that it operate in a certain *operating range* because it is only within this range that the system can provide a good response time, good service and an acceptable *perceived QoS* to the end user. For example, edge routers usually enforce a limit on the amount of traffic a corporation injects into the core network. Sometimes this limit appears as a hard limit where all packets exceeding the limit are dropped, but more frequently, the edge routers just mark packets that are above the limit as *OUT* packets [27], [28]. This allows core routers to drop the *OUT* packets whenever they are under overload. This solution is more or less static since the edge router enforces the limit at which the corporation can inject load into the core and this limit does not change very often.

Static QoS allocation is also the concept used in *Virtual Private Networks* and *Class Based Queuing* schemes. Here QoS requirements are more or less fixed or vary only over long periods of time (see Figure 5).

In situations where (static) aggregation 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

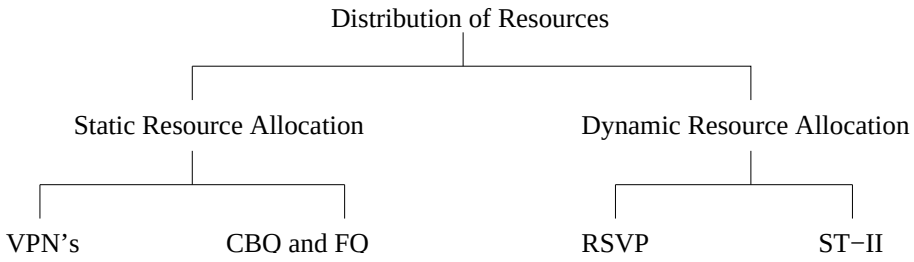

Fig. 5. Resource re-distribution for desired QoS

on an end to end basis. Intermediate routers commit resources and disallow the flow if they cannot meet the requirements of the flow. So there is some control on how many flows are admitted.

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

Rate Control as a means to do QoS regulation appears in other contexts too other than networks. For example, in a real time systems environment, the rate at which new processes enter a system has to be controlled to make sure that currently scheduled tasks do not start to miss their deadlines.

1. Controlling traffic with Bastion Host

The concept 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 would be 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 all the more 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 6 shows a typical topology in which a bastion host would be deployed.

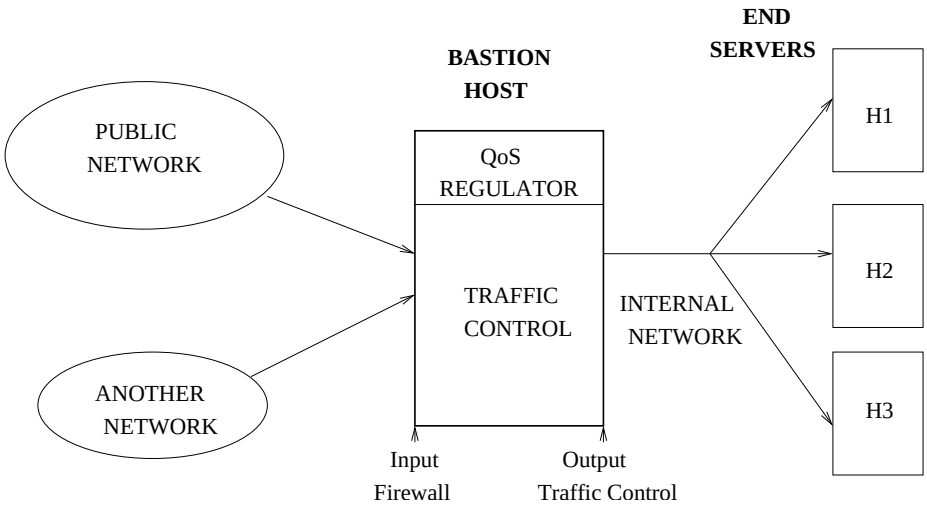

Fig. 6. QoS Regulator or Bastion Host

Since all traffic now comes 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 can give their feedback to the bastion host which can then implement new traffic control rules.

B. Traffic Control Mechanisms

Traffic regulation is a means to attain 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. For the system, these high level primitives translate into various possible control mechanisms at different levels of sophistication. 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 [29],[30].

- Throttling : This is just enforcing a flow rate, for example in bytes/sec, as does *apache*'s *mod_throttle* [31]. Excess packets are simply discarded and no attempts are made to do any kind of traffic shaping.

- Pacing of Acknowledgments: This is a more elegant solution but it works only on protocols like TCP which are flow based and which respond to congestion. By holding back or delaying *ack*'s flowing in the reverse direction, the flow can be forced to back-off to a rate imposed by the administrator [9].

- Simple Rate Control versus 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 also 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 [32],[33]. These include priority

based scheduling, fair scheduling, class based queuing, in addition to simple (σ, ρ) token bucket shaping. The linux TC/IPROUTE solution implements this.

- Window Control : There are situations when simple rate control is not enough and some kind of feedback from end server is required as to amount of pending workload. An example is TCP which uses the sliding window protocol to probe the capacity of the network pipe. We implement a similar windowing protocol but with the idea of regulating load on the end server. More details about our window regulation scheme are discussed in Chapter 4.

C. 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* is not sufficient – a finer grain control is required. The incoming packets are classified into *traffic classes* depending upon the resource they consume. For example, Figure 7 shows three classes of traffic and the corresponding rate limits imposed by the administrator.

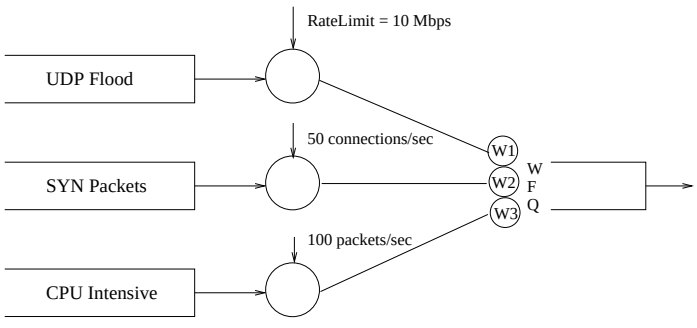

Fig. 7. QoS Regulation by Rate Limit

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 rules into our QoS regulator to implement filters and classify the packets into 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. While queuing the packet, Linux TC infrastructure (discussed in Chapter 2) allows us to choose one of several queuing disciplines. Figure 7 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 an even finer level of control on different traffic classes.

D. Our Implementation and Testbed

To test *Rate Control* with Linux TC, a private network with the following topology 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.

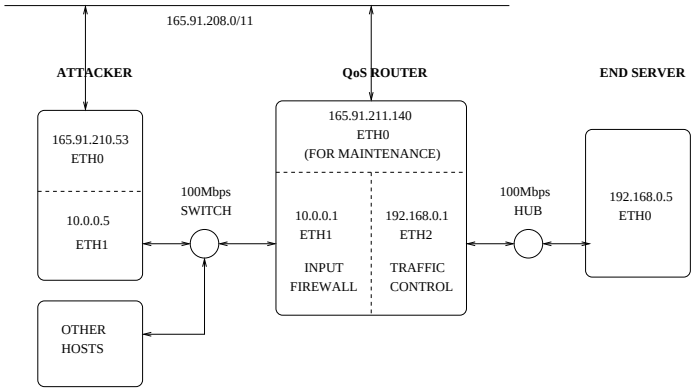

Fig. 8. Topology of Test Network

The machines used in the testbed had the configuration shown in Table I. A fast machine 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 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.

	CPU	**RAM**	**Swap Space**	**Kernel**
Attacker	500Mhz	128MB	176MB	2.2.16−22
QoS Router	233Mhz	64MB	72MB	2.2.14
End Server	166Mhz	48MB	130MB	2.2.12−20

Table I. Configuration of Machines Used

All machines had multiple interfaces that had different roles to play. The interfaces, the configuration of the network cards, and their speeds are shown in Table II. The QoS Router had three interfaces: one to connect it to the world of attackers and 'normal' hosts; second to connect it to the server farms which were our intended

victims; and a third interface to do maintenance work. This last interface was used for trace collection and downloading code onto the test machines.

	Interface	Card	Driver	Kernel Module	Speed
Attacker	eth0	Winbond 89C940	PCI NE2000	ne2k−pci	10Mbps
	eth1	SMC 1211 TX EZCard	RealTek RTL8139	rtl8139	100Mbps
QoS Router	eth0	RealTek RTL−8029	PCI NE2000	ne2k−pci	10Mbps
	eth1	Lite−On PNIC−II rev37	tulip	tulip	100Mbps
	eth2	Lite−On PNIC−II rev37	tulip	tulip	100Mbps
End Server	eth0	Lite−On PNIC−II rev37	tulip	tulip	100Mbps

Table II. Configuration of Test Network

E. Measurements and Results

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. This section presents some results on how well the QoS Rate Control is able to control the load on the end servers in some cases, while it fails in other cases.

1. Benchmarking

First, our test machines were benchmarked for CPU and network performance. For measuring CPU performance, the popular Dhrystone benchmark [34] test suite was used. Figure 9 shows the results of the basic CPU performance as measured with the Dhrystone benchmarking program. This sets the upper bound for the processor power available on each machine.

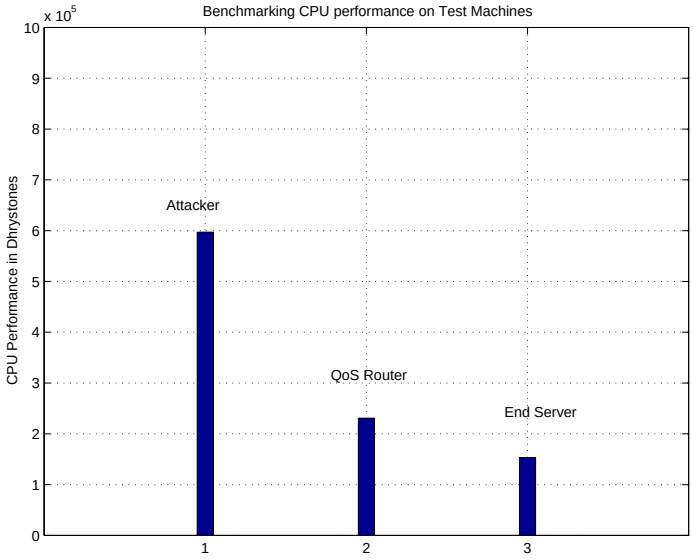

Fig. 9. Basic Benchmarking of Machines with Dhrystones

2. UDP Goodput versus QoS Rate limit

Figure 10 shows the UDP goodput received on the end server when packet size is varied. In this experiment, we determine the limits for peak UDP traffic that can be processed by the end server. This will help the administrator set hard limits on the QoS regulator for UDP class of traffic. The attacker sends a full flood of 100Mbps UDP to the the end host. The QoS regulator regulates this flood to a rate which is plotted on the x-axis in Figure 10. The y-axis shows the goodput received at the end server. There was no background load in these experiments.

It is seen that for MTU size packets, the end server is able to perform best – receiving the entire UDP load of 100 Mbps – almost up to the line-speed of an OC-3 connection. As 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

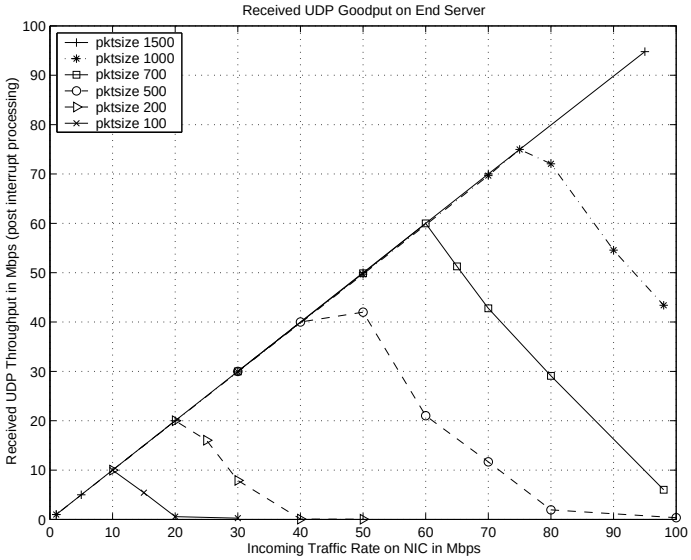

Fig. 10. UDP Goodput as a function of QoS ratelimit

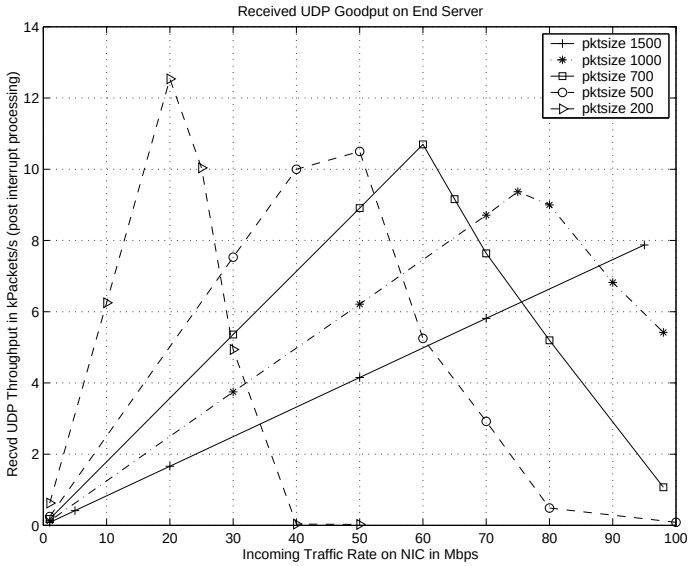

Fig. 11. UDP Goodput measured in Packets per second

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. Hence received UDP goodput suffers. Therefore the regulator should make sure that UDP class of traffic should not send so much traffic that the end server goes into overload. Perhaps separate classes could be maintained within the UDP class, to exercise stricter control if packet sizes are smaller.

Figure 11 shows the same results but instead of plotting UDP goodput in bytes/sec, we have plotted it in packets/sec. This is a fairer comparison in some way since it shows the per-packet processing cost more clearly. It is seen that the per-packet processing cost increases slightly as packet size increases.

3. CPU Load versus QoS Rate limit

Figure 12 shows the load on the CPU (as a percentage of CPU still available for useful work) as a function of QoS Rate limit on router. Results for varying packet sizes are plotted. The x-axis shows the QoS rate limit on the regulator while the attacker is still sending a 100Mbps UDP flood. As it may be seen from the earlier packets-per-second measure (Figure 11), the load on the end server should be a function not only of the raw bitrate but also of the packets-per-second rate. As was also evident from Figure 11, the per-packet processing cost decreases slightly with decreasing packet sizes. The load however goes up because of increased frequency of packet arrivals. At almost ethernet MTU size packets, the peak packet processing rate is limited to about 9000 packets per second, while at 200 byte packets, the end server can handle as high as 13000 packets per second before getting overloaded.

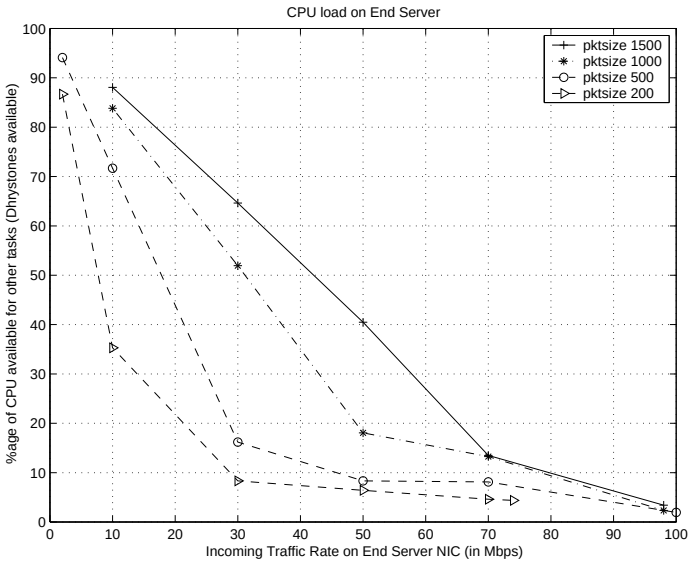

Fig. 12. CPU Load on End Server as a function of QoS ratelimit

4. 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 kept sending a constant UDP flood at network capacity of 100Mbps. The QoS rate limit for the UDP class was varied on the regulator. The packet size of UDP flood packets was also varied over 500, 1000, and 1500 bytes and the results are shown in Figures 13, 14, and 15

In all three cases, it is seen that the preferred flow is able to keep up 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 fail in the competition 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. This is

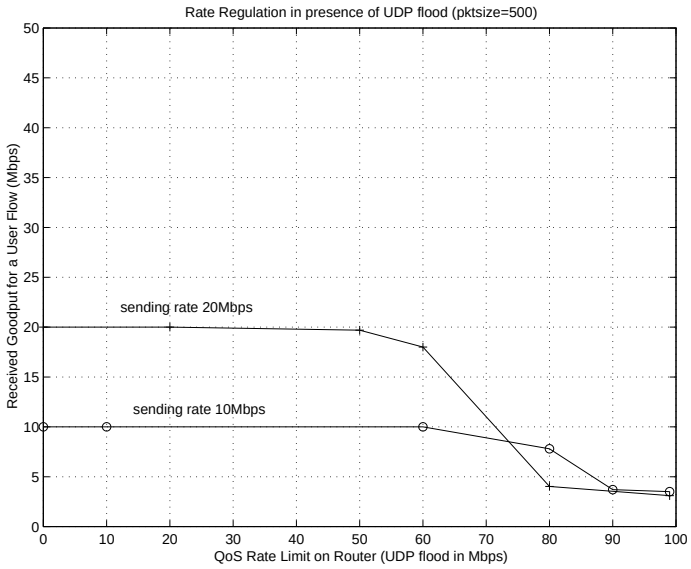

Fig. 13. Good flow in presence of UDP flood (MTU 500 bytes)

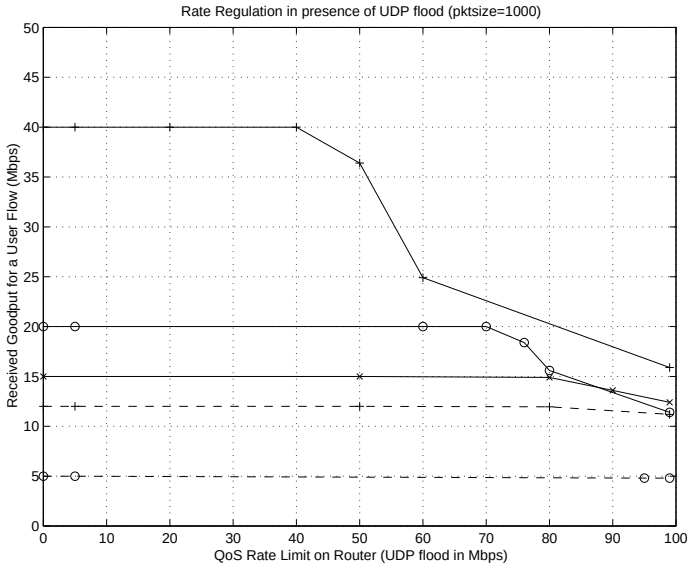

Fig. 14. Good flow in presence of UDP flood (MTU 1000 bytes)

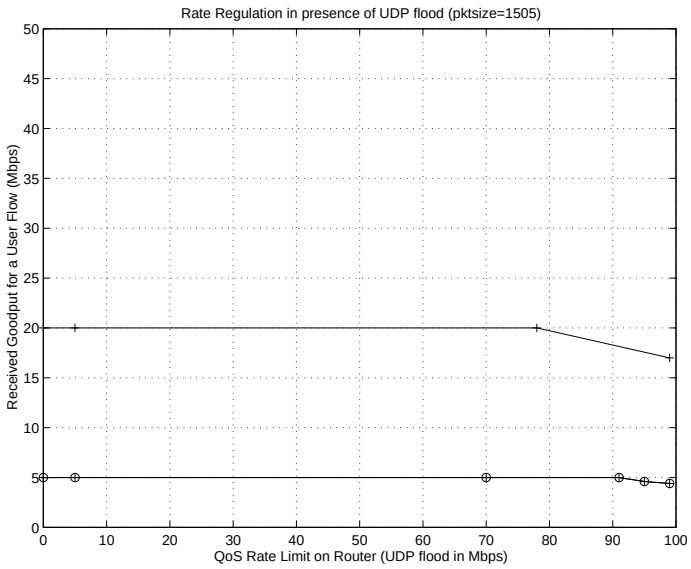

Fig. 15. Good flow in presence of UDP flood (MTU 1500 bytes)

an important parameter for the QoS regulator to do effective flow policing.

5. 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 16. It is seen that if there were no rate control, the preferred flow would only get a bandwidth of 25Mbps instead of the requested 40Mbps. The preferred flow is able to get full bandwidth provided the ICMP load is less than 20Mbps. This result again helps us to enforce policing rules on the QoS regulator.

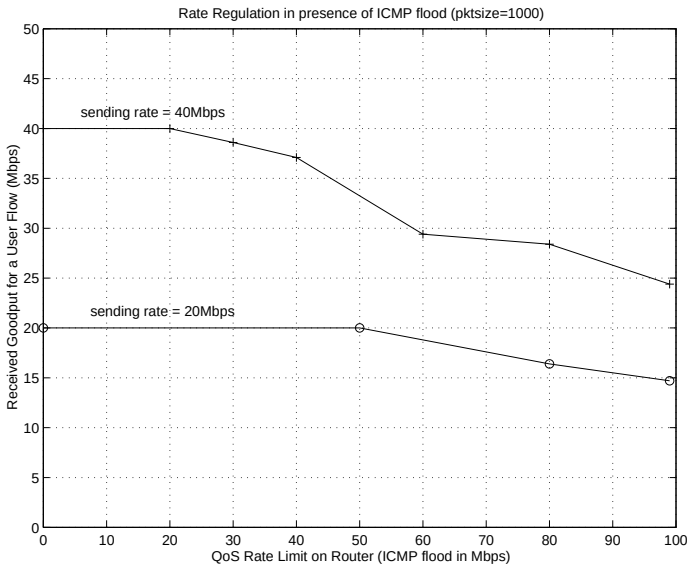

Fig. 16. Good flow in presence of ICMP flood

6. Rate limit on CPU Intensive Packets

Computation intensive CGI scripts were chosen to conduct this experiment. CGI requests to execute these scripts were sent to a web server running on the end server and the available CPU power remaining was measured with the Dhrystone benchmark. There was some other background load on the end server besides the CGI load. The CGI scripts themselves executed the Dhrystone benchmark program at normal priority to mimic a CPU intensive workload. A rate limit of one CGI request per second was imposed on the QoS regulator. As more and more CGI scripts entered the system, the CPU power got divided among all competing processes – the background load processes, as well as CGI processes. As we kept on increasing the CGI load, the end server is virtually crippled and did not have any free CPU cycles available to do useful work. At about 15 computation intensive CGI scripts, only 5% of the CPU reamins for useful work.

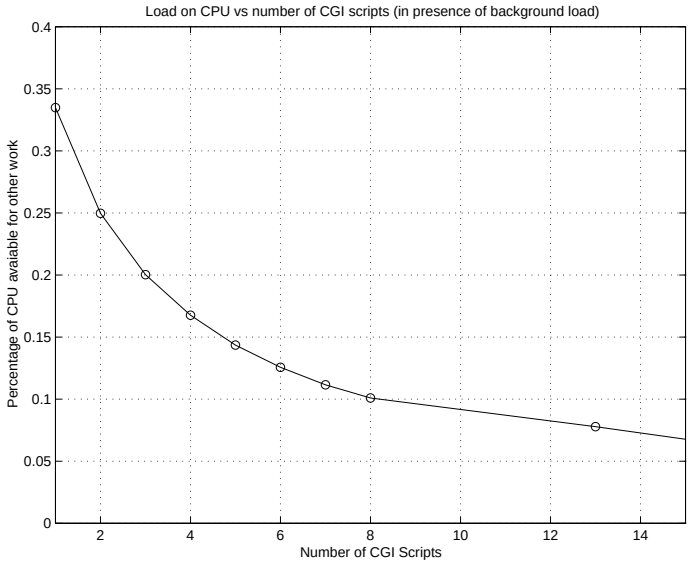

Fig. 17. Rate control on CGI requests (1 request/sec)

A solution to this could be to make changes to the operating system so that lower priority is given to certain CGI scripts. however this solution is very specific and requires changes to the end servers. Our Window Control solution limits the number of CGI scripts to some number N and does not allow more requests to come in until previous requests have exited the system.

7. Rate limit on TCP SYN packets

In this experiment, a rate limit of two SYN packets per second was imposed on the QoS regulator. As expected, simple rate control fails for SYN packets. Normally, the size of the backlog queue per tcp port is 128. The SYN timeout period on the other hand is usually a large number – like 75 or even 450 seconds. Hence, it is very easy to flood the port with fake SYN packets even with a slow enough attack rate. As a side-note, it may be harmful to set a limit as low as 2 SYN packets per second because

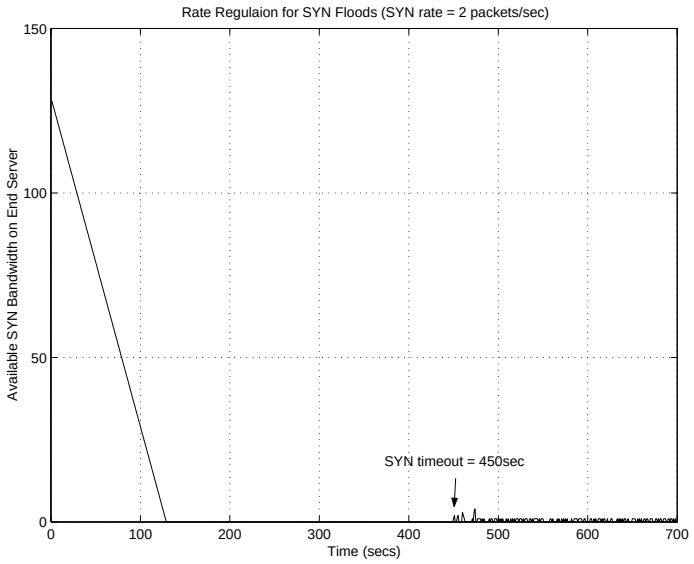

Fig. 18. Rate Control with TCP SYN packets

in that case, the service to genuine clients will be limited to 2 new connections per second.

F. Discussion on 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. For example, if the end server has a peak packet processing capability of 13000 packets per second at 200 byte packets, a good strategy, for example, might be to bound the incoming traffic to 80% of that limit. For this, the QoS router needs to maintain an hierarchical or per class structure to track statistics and average sending rates. Our experiments also show that rate limit works for UDP and ICMP based floods and fails for CGI and SYN floods.

Some of the issues encountered in our experiments are discussed below:

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 get 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. Finally if an acknowledgment never really comes, these structures have to be purged and resources freed. 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.

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 [35],[36],[37]. 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 Figures 10 and 11. The server spends all its time handling network interrupts whereas it could have done some useful work delivering packets to the ap-

plication. This motivates the need for a "Bastion Host" to regulate QoS to end server cluster.

3. Why simple rate control is not sufficient?

Traffic Control by means of enforcing rate limits – either in form of bitrates, or packet rates – may not be sufficient to enforce proper utilization of resources in the end server. For example, in the event that per packet processing is high, as in a computation intensive CGI packet, multiple invocations of the script over the network could overload the processor leading to loss of service to other clients and services. In case of TCP SYN packets too, enforcing a rate limit like 2 SYN packets per second does not work because the size of per-port backlog queue is usually a small number like 128. Even if the number had been larger, it is possibly to very quickly consume all available ports by fake SYN requests. In the next chapter we propose a scheme called *Window Regulation* in the network stack. This scheme keeps a state entry per resource, for a variety of resources in the end servers. If the resource under consideration is the available CPU for example, a counter would be maintained which would be incremented every time a CGI script enters the system and decremented when it exits. This allows us to introduce a new and powerful way of policing traffic when simple rate control does not suffice. We show how this Window Control can be used to control CGI loads and SYN attacks in the next chapter.

CHAPTER IV

RESOURCE ALLOCATION AND WINDOW REGULATION

We saw in the previous chapter that conventional rate based schemes are not enough for load management and denial of service control. The reason for the failure of rate-based schemes is that at the network level it is hard to know the conditions at the server. If the firewall or the bastion host, could somehow get feedback from the end-server about load conditions, then they could change the way incoming and outgoing traffic is policed.

A. What are Resources?

A network plus server system consists of several resources – bandwidth, memory, CPU processing, interrupt handling, buffers, file handles, network sockets, etc. Some of these resources can be partitioned by rate based schemes – like traffic shaping or rate control as described in the previous chapter. Other resources are capacity based resources which are consumed and released alternately. Figure 19 shows different resources consumed by a packet as it moves through the system.

B. The Big Picture

Our approach is to have an aggregate view of all resources in the system and to let the bastion host know of the limit of each kind of resource. The bastion host can then enforce that limit by regulating the window for that resource. Also, this approach allows us to partition by resource as well as by flow aggregate – as an example, an administrative policy might be, to not let the packet-per-second rate to exceed a certain number because of the overhead it would cause in high priority interrupt

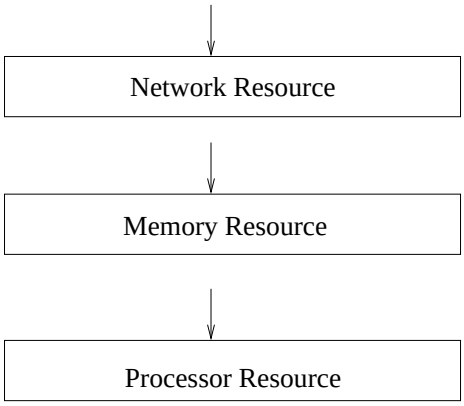

Fig. 19. Aggregate View of Resource Consumption

processing at the end server. Out of this limit, we can further partition this resource and say, for example, that all traffic coming from interface A at the ingress router (or bastion host) be given p percentage of the peak *packets per second* rate.

The advantage of our approach is that we have a picture of all available resources at a central place – viz. at the network layer. This is really significant since we can regulate QoS and control denial of service by a single and simple solution. There is no need to implement different solutions at different levels in the system.

Unlike the QGuard [5] solution which is implemented on the end server itself and which gathers its statistics from the end server, our solution is totally transparent to the end servers. By implementing the QoS regulation and window control on the bastion host, all state maintenance (see next section), and wasteful interrupt processing is relegated to the bastion host. The bastion host is supposed to have higher capacity than the end servers since it is also supposed to act a router for a server farm.

The concept of implicit feedback makes it unnecessary to actually poll the end server about load statistics. Whenever a unit of resource is consumed by a flow,

the corresponding state counters for that resource are decremented. We call this a reduction in the available window. When the resource is freed, and the server gives some indication to this effect by an acknowledgment traversing the network, the window is incremented to allow more such requests to be serviced.

C. QoS with Window Regulation

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).

2. Window Control

Our scheme is designed using the a Window Regulation concept, such as used by TCP for flow control [38]. The windowing algorithm in TCP is basically a means to probe the currently available capacity of a network pipe. It tries to keep the pipe properly utilized by adjusting the sending rate based on the acknowledgments received by the sender. The window limit is also altered if the protocol senses a change in available capacity.

In our model, we propose a window limit *per resource* or *per 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 some resource has been freed. When this happens, the window is relaxed and more flows or requests can be admitted. Different window limits quantify the amount of resource available. The same resource can be split up into multiple portions, and a portion could be allocated to a different set of users. 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 transaction or preferred flows are not starved in presence of overload, or denial of service scenarios.

We have seen in Chapter 3 how some of these QoS guarantees can be provided with rate control. For example, we saw that low priority UDP traffic could be limited to a small percentage of the total link bandwidth. 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 will be preserved.

Figure 20 shows a simple implementation of our window control as applied to CPU intensive CGI packets.

D. Implementation Details

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. These classifiers run through the *sk_buff* headers of the outgoing packet to determine if the current packet should

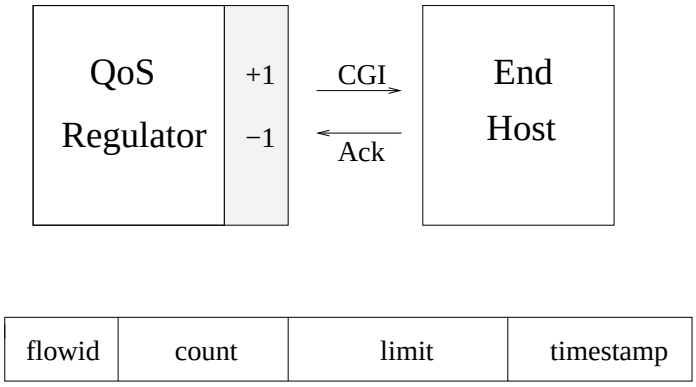

Fig. 20. Window Control Overview

affect the state table for some resource. The number of resources being monitored is decided administratively.

Most of the modifications were made in the `src/net/sched/` directory structure. Policing and window control function was implemented in the `tcf_police()` function in `police.c`. The expanded data structures were populated in `src/include/linux/pkt_cls.h`, `~/include/net/pkt_sched.h` and in `~/include/net/ip.h` The `tcf_police()` function is called each time a filter on the outgoing interface matches a rule. We create a rule for each resource that we want to police. For example, a resource could be the SYN protocol buffers for port 80 on end server #1. A filter is installed (via `cls_u32.c`) that will check all packets containing a SYN for server #1 port 80. Each time that happens, an entry is added in our state table. If an entry already existed, then only the time-stamp is updated. If the entry table is full (that is, the window limit for the current resource has been reached), this SYN packet would be dropped.

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. These variables would then be written into kernel data structures. After that point, the main policing functions – `tcf_police()` and `cls_u32` take control of which traffic classes are to be monitored.

1. Maintaining State in Kernel

To enforce window control, varying amount of state is needed in the network – preferably at the bastion host (Figure 6). We keep this state information in kernel variables and data structures on the bastion host. For each TCP SYN packet, an entry of the type shown in Figure 21 is maintained. The entries are hashed based on the initial sequence number of the connections. The 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.

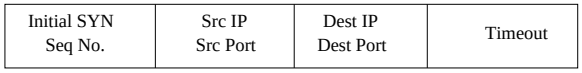

Fig. 21. State Maintenance for SYN connections

Fig. 22. State Maintenance for CGI Packets

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 the 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.

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.

To reset system state, a system call was implemented in the kernel. This call constructs a raw socket [39] 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. A `sock_sendmsg()` function is called to send this packet to the correct end server. This has the effect of clearing the protocol state buffers that were held at the end server.

E. Results

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).

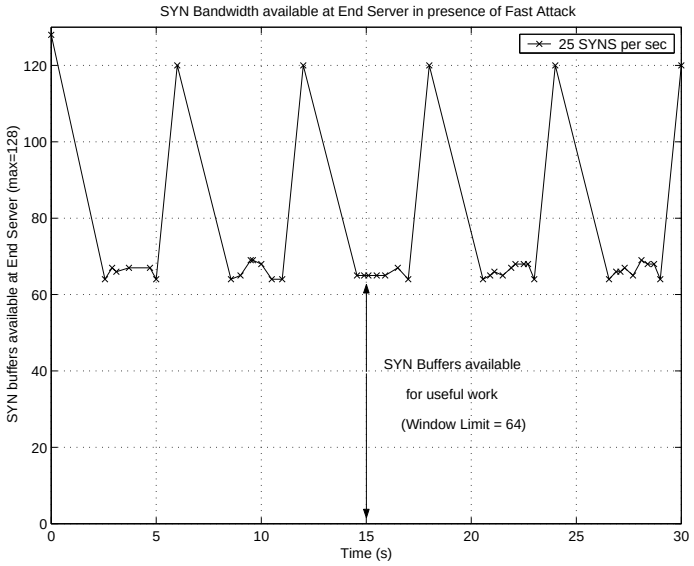

Fig. 23. Window Control with Fast SYN Attack

Two attack scenarios were constructed – a fast attack (Figure 23) and a slow attack (Figure 24). 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

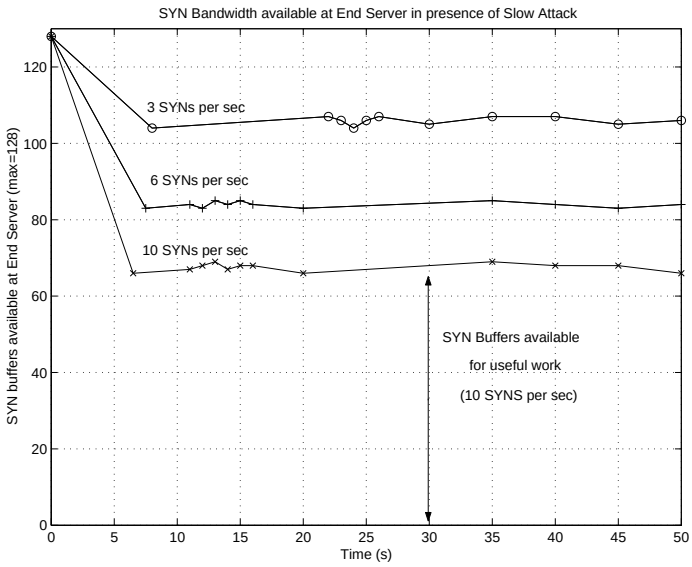

Fig. 24. Window Control with Slow SYN Attack

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.

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.

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. Figure 25 shows

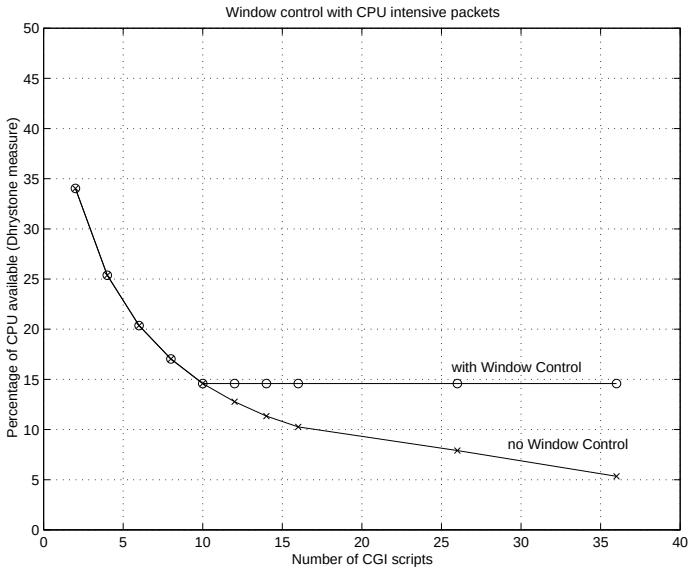

Fig. 25. Window Control with CPU Intensive loads

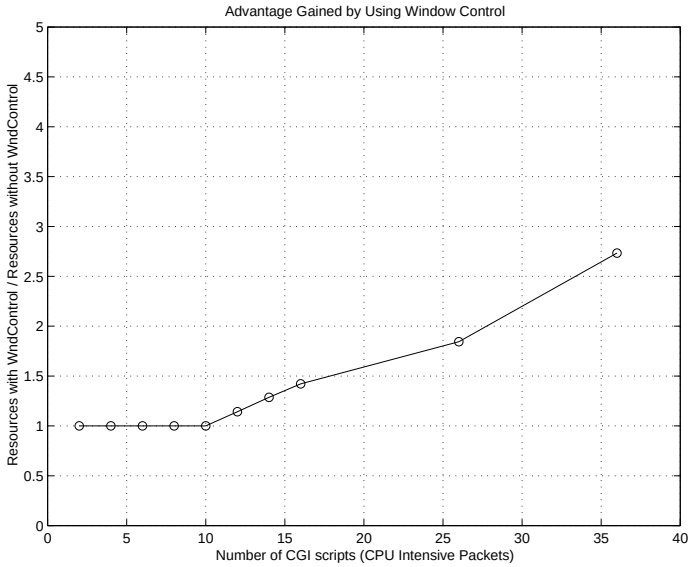

Fig. 26. Advantage Gained with Window Control

the percentage of CPU available as more and more CGI requests are sent to the system. The advantage gained in terms of available server CPU resources is plotted in Figure 26.

3. Performance Measurements

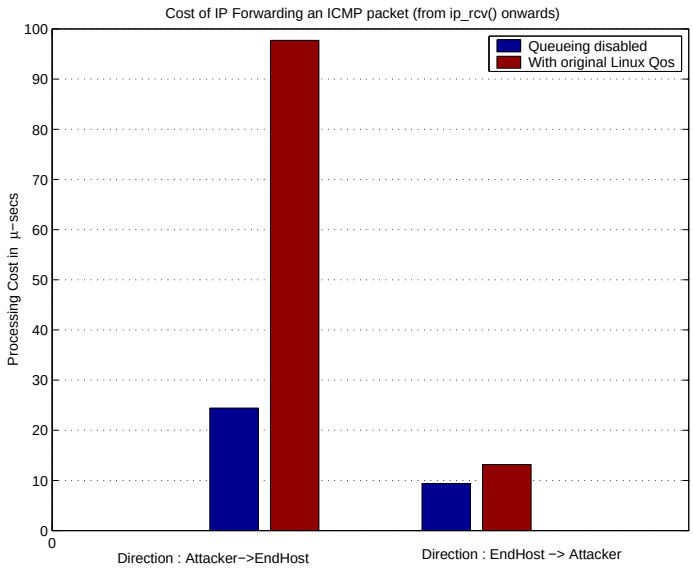

Fig. 27. Per Packet Processing Cost

This section shows some preliminary performance measurement results. Figure 27 shows the processing cost per packet before including our code for window control and state maintenance. The left set of bars shows the prcessing cost incurred at the Bastion Host for an ICMP packet travelling from the attacker to the end server. The right two bars show the cost incurred by replies as they come back from the end server to the attacking machine.

In the first set, the first black bar represents the cost incurred by a vanilla router that does not have any of the smart QoS queueing (as discussed in Chapter 2) built

into it. When the 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 does not need to do any classification, queueing, etc – it can simply queue the packets on the outgoing device when IP passes the packets after processing.

The second set of two bars just shows that even after enabling the queueing disciplines, the incoming path is not changed. THe additional cost is added only on the outgoing path on which queues are enabled.

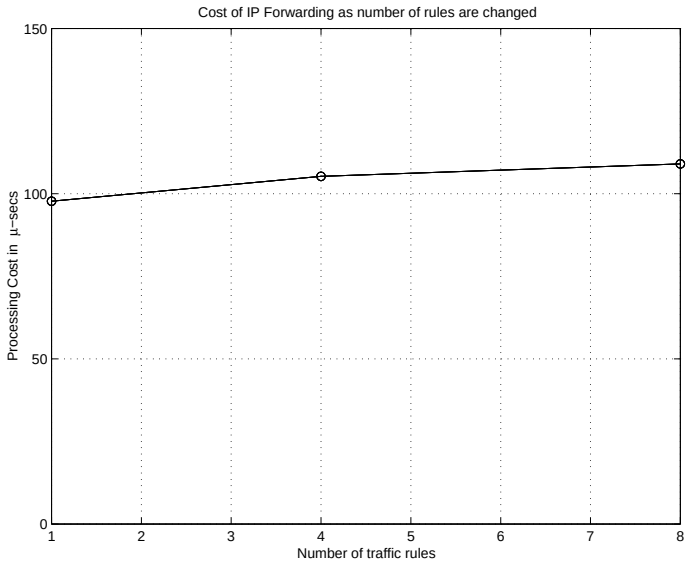

Fig. 28. Processing Cost versus Number of Rules

Figure 28 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 that we insert and because of the system call that we make to flush the state sometimes. The cost of a single system call was

measured to be about 2000 clock cyles or about 8.5 μ-secs. Figure 28 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.

F. Conclusions

It was seen from preliminary experiments that window regulation is effective in regulating resource consumption when an aggregate limit on resources is imposed. A finer level heirarchical flow-level control would be the focus of our future work. Our experiments show the effectiveness in bounding the resources consumed by SYN attacks and CPU intensive packets like CGI requests. Our architecture allows us to have a picture of all available resource at a central place (viz. the network layer), rather than at several places in the operating system.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

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 firewall. Previous work has focussed on maintaining this state at several places inside the operating system on the end servers.

We implemented the scheme using a linux box as a router. The available resources were divided into separate classes – like classes of traffic which consume protocol state buffers, class of CPU intensive packets, UDP packets, ICMP packets, etc. Then a per-resource policing was enforced on the bastion host so that each class of traffic stayed within its resource limit. This solution provides some resisteance against Distributed Denial of Service attacks in which a per flow policing is not effective. Since resources within a class are policed, the attack cannot take away resources from another class.

Also, if a single attacker is consuming resources within a class, we could revert to flow based policing and enforce stricter limits for this one class of traffic. In other words, if a class of traffic (like SYN buffers) is facing scarce resources, we may conclude that an attack is going on and enforce a limit of 3 outstanding SYNs per IP address.

However if an attack consumes resources in multiple classes – like UDP, ICMP, SYN, CPU intensive network packets, etc – then our scheme will not be able to provide protection in all classes simultaneously.

Other applications of our platform include *loadbalancing*. 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 ligtly 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. There needs to be a simple way to match traffic classes to resources consumed.

REFERENCES

[1] 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`.

[2] Allen Brady Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, Todd A. Proebsting, and John H. Hartman, "Scout: A Communications-Oriented Operating System," in *Operating Systems Design and Implementation*, 1994, p. 200.

[3] David Mosberger and Larry L. Peterson, "Making Paths Explicit in the Scout Operating System," in *Operating Systems Design and Implementation*, 1996, pp. 153–167.

[4] 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.

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

[6] T. F. Abelzaher and Kang G. Shin, "QoS Provisioning with qContracts in Web and Multimedia Servers," Pheonix, AZ, December 1999.

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

[8] 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.

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

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

[11] "Cisco Inc. Local Director (White Paper)," 2000, `http://www.cisco.com/warp/public/cc/pd/cxsr/400`.

[12] 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.

[13] 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.

[14] "SYN Cookies," `http://cr.yp.to/syncookies.html`.

[15] "CERT Coordination Center," `www.cert.org`.

[16] Almesberger, Werner and Kuznetsov, Alexey and Salim, Jamal Hadi, "Differentiated services on linux," in *Proceedings of Globecom'99*, Rio de Janeiro, December 1999, pp. 831–836.

[17] Werner Almesberger, "Linux network traffic control – implementation overview," in *Proceedings of 5th Annual Linux Expo*, Raleigh, NC, May 1999, pp. 153–164.

[18] W. Almesberger, "Linux Network Traffic Control," .

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

[20] 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.

[21] "Linux Packet Shaping HOWTO," 1999, http://www.guides.sk/Packet-Shaping-HOWTO.html.

[22] T. Dawson, "Linux NET-3-HOWTO," 1999.

[23] David A. Rusling, "The Linux Kernel," *The Linux Documentation Project*, 1996.

[24] Richard Stevens and Gary Wright, *TCP/IP Illustrated*, Addison Wesley, 1999.

[25] Glen Herrin, "Linux IP Networking – A Guide to the Implementation and Modification of the Linux Protocol Stack," May 2000, http://kernelnewbies.org/documents/ipnetworking/.

[26] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*, OŔeilly Publishing Company, October 2000.

[27] "Diffserv Home Page," `diffserv.lcs.mit.edu`.

[28] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," *Work in progress. Internet Draft draft-ietf-diffserv-arch-02.txt*, 1998.

[29] P. Russell, "ipchains-HOWTO," `http://www.linuxdoc.org/HOWTO/IPCHAINS-HOWTO.html`.

[30] Rusty Russell, "iptables-HOWTO," `http://www.telematik.informatik.uni-karlsruhe.de/lehre/seminare/LinuxSem/downloads/netfilter/iptables-HOWTO.html`.

[31] "Apache mod_throttle," `http://www.snert.com/Software/mod_throttle/`.

[32] Abhay K. Parekh and Robert G. Gallagher, "A generalized processor sharing approach to flow control in integrated services networks: the multiple node case," *IEEE/ACM Transactions on Networking*, vol. 2, no. 2, pp. 137–150, 1994.

[33] S. Golestani, "A Self-Clocked Fair Queueing Scheme for Broadband Applications," 1994.

[34] R. Weicker, "Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules," *SIGPLAN Notices, 23, 8*, pp. 49–62, August 1988.

[35] 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.

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

[37] Mohit Aron and Peter Druschel, "Soft timers: efficient microsecond software timer support for network processing," in *17th ACM Symposium on Operating Systems Principles*, December 1999, pp. 232–246.

[38] J. Postel, "RFC 793: Transmission Control Protocol," 1981.

[39] Thamer Al-Herbish, "Raw ip networking faq," `http://www.whitefang.com/rin/`.

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

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

[42] S. M. Bellovin, "Security Problems in the TCP/IP Protocol Suite," *Computer Communication Review*, vol. 19, no. 2, pp. 32–48, 1989.

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

[44] 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.

[45] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar, "PATHFINDER: A Pattern-Based Packet Classifier," in *Proceedings of the First Symposium on Operating Systems Design and Implementation*, 1994, pp. 115–123.

[46] Steven McCanne and Van Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *USENIX Winter Conference*, 1993, pp. 259–270.

[47] 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.

[48] Rik Farrow, "TCP SYN Flodding Attacks and Remedies," `http://www.networkcomputing.com/unixworld/security/004/004.txt.html`.

[49] S. Keshav, "An Engineering Approach to Computer Networking," 1997.

[50] David K. Hess David R. Safford and Douglas Lee Schales, "The TAMU Security Package: An Ongoing Response to Internet Intruders in an Academic Environment," Tech. Rep., 1993.

VITA

Aman Garg received his B.Tech degree in Electrical Engineering from the Indian Institute of Technology, Kanpur in 1998. He joined the Electrical Engineering department at Texas A&M University in fall 1999, where he is pursuing his MS degree. He is working with Professor A L Narasimha Reddy and his main interests are in systems, networks and QoS.

The typist for this thesis was Aman Garg.