Mitigating Denial of Service Attacks Using QoS Regulation *

Aman Garg[1] and A L Narasimha Reddy[1]

Texas A&M University, College Station TX 77840, USA,
{aman, reddy}@ee.tamu.edu

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 Denial of Service attacks over the recent past. This paper reports on our experience in building a Linux-based prototype to mitigate the effect of such attacks. Our prototype provides an efficient way to keep track of server and network resources at the network layer and allows aggregate resource regulation. Our scheme provides a general, and not attack specific, mechanism to provide graceful server degradation in the face of such an attack. We report on the rationale of our approach, the experience in building the prototype, and the results from real experiments. We show that traditional rate-based regulation combined with proposed window-based regulation of resources at the aggregate level at the network layer is a feasible vehicle for mitigating the impact of DOS attacks on end servers.

1 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. Much work has been done in designing solutions for such attacks. Most of these solutions take an attack-specific approach whereby solutions cannot be designed until an attack is designed or staged. The challenge is to design general schemes that identify and successfully counter a variety of these attacks. In this paper, we address the DoS problem from a resource regulation perspective. Our approach, instead of identifying the attacks, identifies the resources that need to be protected. Our approach then relies on the regulation of consumption of these resources. It is expected that successful regulation of resource consumption will lead to mitigation of DOS attacks. In this paper, we are primarily considering attacks on end servers.

The common types of DoS attacks include attacks on particular kinds of protocol behavior as in the case of TCP SYN flood, ICMP directed broadcast, DNS

* This research was funded in part by an NSF Career Award and NSF Grant ANI-9909229

flood, etc. TCP SYN attack consumes protocol data structures in the server side OS. ICMP directed broadcast involves spuriously directing a broadcast address to send a flood of icmp replies to a target host thereby increasing the load on the target machine. DNS flood attack uses specific weaknesses in DNS protocols to generate a lot of traffic which can be directed to a target. UDP being a connectionless protocol with no congestion control built into it, turns out to be a good vehicle for many of these attacks. Another attack specific to web servers, is to repeatedly ask expensive CGI requests to be executed, thereby driving up the CPU load on the server. Typical resources that get consumed in such attacks are network bandwidth, CPU cycles on server, interrupt processing on the server, and sometimes specific protocol data structures (like fragmentation buffers, TCP SYN buffers, etc). A distributed denial of service attack (DDoS) uses a large number of compromised machines or agents to attack a target in a synchronized fashion. It is harder to trace distributed attacks.

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.

Resource regulation at end servers can be done at the flow level such that each flow is ensured to claim only a fair share of its resources. Such flow-based approaches are feasible at end hosts [2]. However, distributed attacks when staged from a large number of machines will still succeed since each flow seems to be consuming only a limited amount of end server's resource. To mitigate such distributed attacks, we adopt a resource-centric regulation approach. Even when a coordinated attack is staged from many machines to consume a resource at a server, resource-based regulation is expected to identify such an attack.

Resource Control and QoS are closely tied to each other. In our context, resource control 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 maintains at the network level, a rough snapshot of available system resources. Traffic directed towards a server is classified based on its likely resource consumption. By regulating the resource consumption of each such traffic class, we expect to limit an attack to consume only certain level of server resources such that other classes can continue receiving service. Resource regulation is controlled through

QoS mechanisms such as rate control, Weighted Fair Queuing and our proposal for window control.

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.

2 Design Details

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.

Traffic regulation is a means to achieve the required QoS goals. These goals may have to do with load, bandwidth, burstiness, delay, security, etc. In our approach we use a combination of Rate Control and Window Control to have a better control over resources. Other mechanisms for regulating traffic include firewalling, flow throttling, ack pacing [2], traffic shaping, and scheduling [3], [4]. In our approach, we employ a *bastion host* to do both rate-based and window-based control of server resources at network layer.

The idea of a bastion host originated from security considerations when it is 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 1 shows a typical topology in which a bastion host would be deployed. The advantages of this approach is that it does not require any modification on the server farm being protected. Policies to do traffic control, management and shaping, can be deployed on the bastion host without the end servers knowing that new policies have come into existence. As we will show later, bastion host also relieves the end hosts of expensive interrupt processing of attack packets.

In our approach, we deploy aggregate resource regulators at the bastion host to reduce the impact of attacks on the end hosts and the network. 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 and the level of resource consumption by different packets, flows or traffic classes. In our case, the focus is on resources and not on flow identities. Policing, therefore, is 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

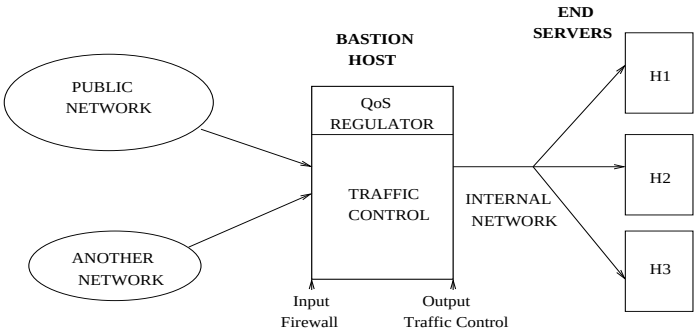

Fig. 1. QoS Regulator or Bastion Host

controlling resources at the aggregate level. This leads us to the creation of *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. To facilitate the regulation at the bastion host, we reflect the end server resources at the network layer of the bastion host.

Figure 2 shows a high level overview of how output traffic control is implemented on the bastion host. The core part of our design is to use a packet classifier to map different flows or packets to traffic classes, depending on the resources they consume. 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. Figure 2 shows two traffic classes (1 and 2) being regulated by a rate control and one traffic class (3) being regulated by a window control.

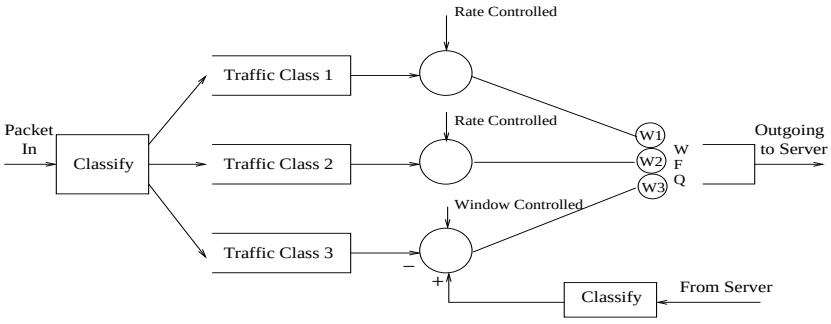

Fig. 2. QoS Regulation : Big Picture

In order to employ such a resource regulation, we maintain some state information for each class at the bastion host. This state information may contain

the level of resource consumption of this class, timestamps etc.. Whenever a unit of resource is consumed by a flow, the corresponding state counters for that resource are decremented. Reflecting available resources to the bastion host allows the bastion host to make intelligent decisions while admitting traffic. We call this technique *implicit feedback*. Implicit feedback avoids the need for polling the end server about load statistics. In the rest of this section, we discuss the design rationale behind using rate and window control.

Rate Control: 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. Different rate-limiting policies could be applied to different classes of traffic based on the resources they consume, in Fig. 2, in terms of bitrate, connections per second, or packets per second.

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 could be 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. If admitting the packet will not violate the rate limit set for this class, the packet is queued on the outgoing interface. Weights can be assigned at the time of queuing (Fig. 2) allowing 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.

Window Control: Window Control is needed when simple rate control is not sufficient to police the traffic. Fixed resources which are based on capacity, as opposed to rates, are examples 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). 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*. This allows us to control how a certain resource can be consumed by a traffic class at any given time. After this limit is reached, incoming requests or packets seeking this resource are dropped or delayed at the bastion host until the server sends some kind of indication that an earlier request from this traffic class has freed its resources. When this happens, 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.

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 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. Second, we need to enforce the window constraint set by the administrator for this resource. This requires that packets/flows be delayed once the window limit is reached and that the acks going in the reverse direction update the window variables appropriately (at the bastion host). 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 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 3 shows a simple implementation of our window control as applied to CPU intensive CGI packets.

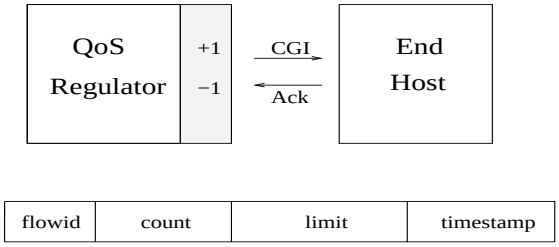

Fig. 3. Window Control Overview

3 Implementation Details

In this section, we highlight how the ideas presented earlier are implemented in our prototype based on the Linux kernel. Recent Linux kernels (beginning with the 2.2 kernel) have included a Quality of Service architecture in the network

[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.

stack [3], [4]. This QoS architecture allows for provisioning of network bandwidth by implementing a variety of traffic control functions. We used this architecture as a basis for our implementation and experiments, since this platform is highly flexible and lends itself to modification. The features are very similar to those offered by current state-of-the-art network processors, the difference being that this architecture is implemented at the OS level (in software). We added functionality to do State Maintenance, QoS regulation, and Window Control.

Different layers comprising the networking stack and the interactions between them, are shown in Figure 4. After the Network Interface Card (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 4.

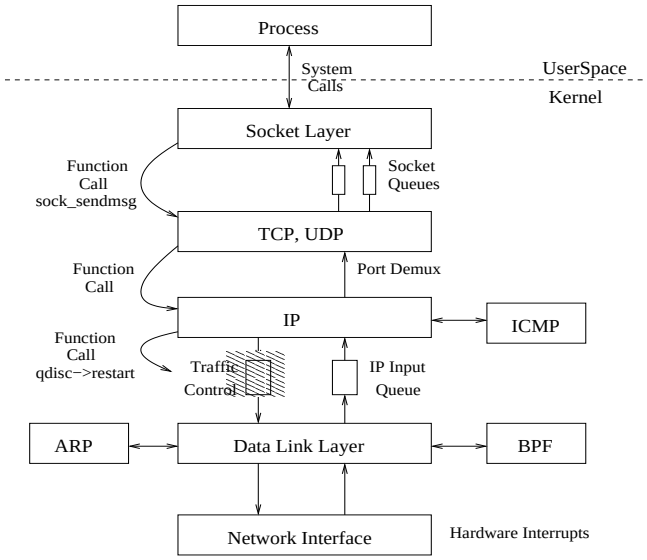

Fig. 4. Linux Network Stack Implementation

The main components of Linux QoS architecture are queuing disciplines *(qdiscs)*, *classes*, classifiers or *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 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.

Our Implementation and Testbed: Our test-bed consisted of a router, to take the role of a bastion host, through which all traffic to the end server was routed (Figure 5). The end server may be a corporate server network or a clustered server.

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.

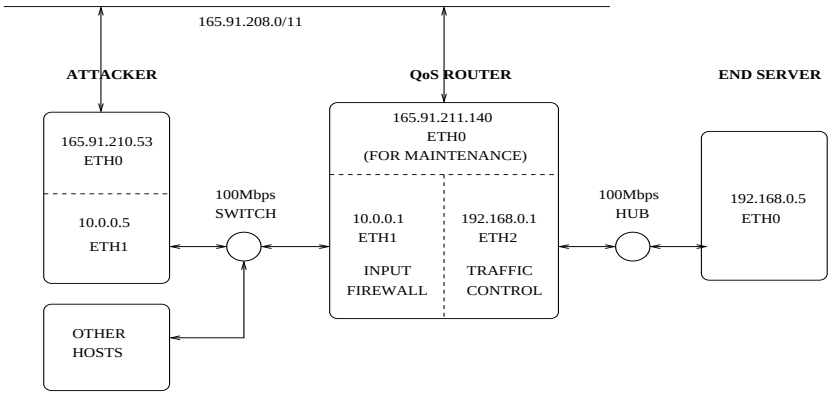

Fig. 5. Topology of Test Network

A fast machine (500 MHz Pentium) was chosen for the attacker to enable it to carry out an attack 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.

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.

Maintaining State in Kernel: To enforce window control, varying amount of state is needed at the bastion host (Figure 1). 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 this entry would be purged and resources freed. 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.

Resetting System State: In general, the state information can be monitored to detect attacks. When resources are held at the server for long periods of time (for example, during an attack), corresponding state does not change at the bastion host. This information can be used for detection of attacks and for freeing up held resources at the server. Timers can be used for this purpose. When a timer expires indicating that resources are being held (uselessly) over long periods of time, a corrective action can be taken by the bastion host to free up those resources. For example, we could send a TCP reset packet to free up TCP protocol state buffers on the end server during a TCP SYN attack. The corrective actions will depend on the held resource. 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 based on the held resource. (for example, a TCP packet with RST flag for TCP SYNs). This has the effect of clearing the resources that were held at the end server.

4 Results

4.1 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. This serves to benchmark end server capabilities.

Received packets/sec versus QoS Rate limit in Mbps: Fig. 6 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.

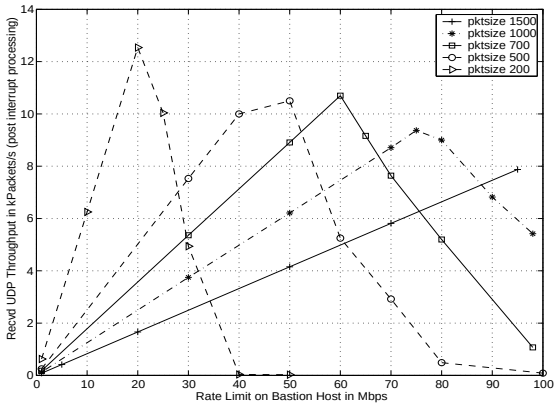

Fig. 6. UDP Goodput measured in Packets per second

Preferred flow in the 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 7. The x-axis shows the UDP rate limit on the input link where UDP flood originated and the y-axis shows the realized bandwidth of preferred flow arriving on a separate input link.

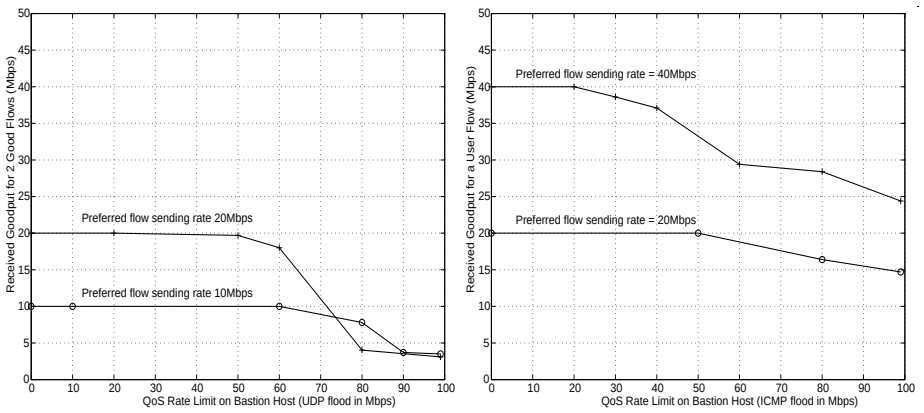

Fig. 7. 2 Good flows in presence of (i) UDP flood (MTU 500 bytes) (ii) ICMP flood

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. 6 and 7validate the utility of rate controlling the UDP traffic.

Preferred flow in the 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 7. 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.

4.2 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:

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.

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 6. These results underscore our earlier motivation for employing a bastion host to regulate QoS of end server cluster in order to protect the end servers from getting swamped by interrupt processing of arriving packets.

4.3 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.

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 and a slow attack (Figure 8). 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. Shorter timeout periods (smaller than 5 seconds) enable survival against faster TCP SYN attacks.

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.

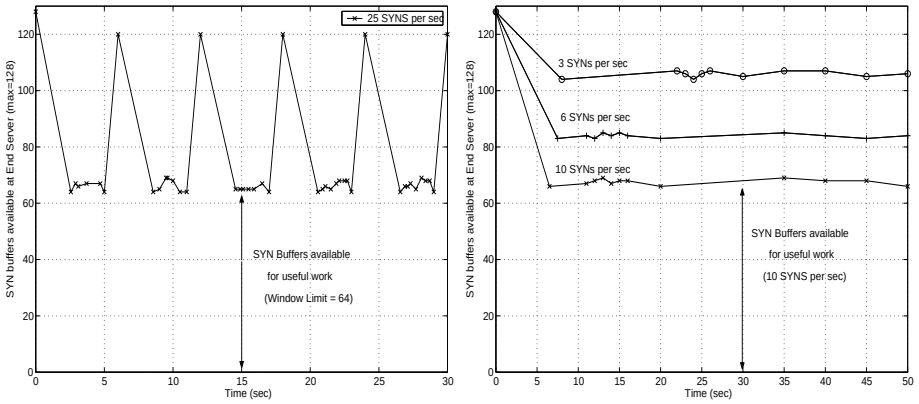

Fig. 8. Window Control with (i) Fast SYN Attack (ii) Slow SYN Attack

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 9. As higher number of CGI scripts arrive, our window control does not forward them to the end server and hence higher CPU power remains available for other applications. This demonstrates the efficacy of window control for CGI queries.

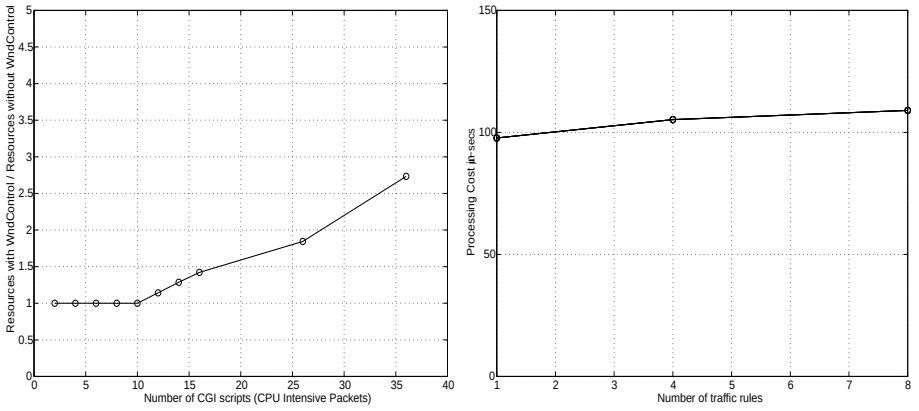

Fig. 9. (i) Advantage Gained with Window Control (ii) Cost of IP Forwarding

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. These classification costs are much smaller than the end server costs in serving the application packets and hence this is not a scalability issue.

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 9 plots the processing cost per packet incurred by our implementation as the number of traffic rules is increased. 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. It is seen that the marginal increase in cost is not significant.

5 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 require a lot of support from the operating system [8]. 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 [8] 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. The general idea of *resource containers* introduced by Banga et. al. in [9] somewhat similar. 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 of attack packets 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 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.

6 Conclusions and Future Work

In this paper, we have proposed to utilize QOS regulation techniques to mitigate the impact of DOS attacks on end hosts and networks. Our solution was based on maintaining a state of the server resource usage at the network layer of a bastion host. Traffic regulation policies are enforced across traffic classes based on the resource usage of packets or flows. We showed that such an approach could be effective in mitigating the impact of DOS attacks through real experiments conducted a Linux based prototype. Our prototype testbed has shown that the proposed approach does not incur significant per-packet overhead. Our solution is effective against distributed DOS attacks as well since regulation is done at an aggregate level and not individual flow level.

We are extending our current framework and implementation in multiple directions. 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. We are trying to build a hierarchical framework where aggregate resource regulators could be coupled with flow-based regulators. This is expected to provide a finer level of resource control while providing DDOS detection capabilities. Such an architecture is also expected to provide protection against an attack that consumes multiple resources simultaneously.

An important application of our platform could be *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 lightly loaded end servers. More traffic may then be directed to these lightly loaded servers out of a server farm.

References

1. C. L. Schuba, I. V. Krsul, M. G. Kuhn, E. H. Spafford, A. Sundaram, and D. Zamboni, "Analysis of a denial of service attack on TCP," in *Proc. of IEEE Symposium on Security and Privacy*, May 1997, pp. 208–223.
2. "Packeteer Inc. White Papers," `www.packeteer.com/solutions/resources`.
3. W. Almesberger, J. Hadi Salim, and A. Kuznetsov, "Differentiated Services on Linux – Internet draft," 1999, `draft-almesberger-wajhak-diffserv-linux-01.txt`.
4. B. Hubert, G. Maxwell, R. van Mook, M. van Oosterhout, and P. 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. M. Aron and P. 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. O. Spatscheck and L. Petersen, "Defending Against Denial of Service Attacks in Scout," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, Feb. 1999.
9. G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Operating Systems Design and Implementation*, 1999, pp. 45–58.
10. H. Jamjoom, J. Reumann, and K. 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. G. Banga and P. 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. A. Cohen, S. 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. V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum, "Locality-Aware Request Distribution in Cluster-based Network Servers," in *ASPLOS*, October 1998, pp. 205–216.
18. O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. Peterson, "Optimizing TCP forwarder performance," *IEEE/ACM Transactions on Networking*, vol. 8, no. 2, pp. 146–157, 2000.