Crossbow: A Vertically Integrated QoS Stack

Sunay Tripathi
sunay.tripathi@sun.com

Nicolas Droux
nicolas.droux@sun.com

Thirumalai Srinivasan
thirumalai.srinivasan@sun.com

Kais Belgaied
kais.belgaied@sun.com

Venu Iyer
venu.iyer@sun.com

Solaris Kernel Networking
Sun Microsystems, Inc.
17 Network Circle, Menlo Park, CA 94025, USA

ABSTRACT

This paper describes a new architecture which addresses Quality of Service (QoS) by creating unique flows for applications, services, or subnets. A flow is a dedicated and independent path from the NIC hardware to the socket layer in which the QoS layer is integrated into the protocol stack instead of being implemented as a separate layer. Each flow has dedicated hardware and software resources allowing applications to meet their specified quality of service within the host.

The architecture efficiently copes with Distributed Denial of Service (DDoS) attacks by creating zero or limited bandwidth flows for the attacking traffic. The unwanted packets can be dropped by the NIC hardware itself at no cost.

A collection of flows on more than one host can be assigned the same Differentiated Services Code Point (DSCP) label which forms a path dedicated to a service across the enterprise network and enables end-to-end QoS within the data center.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Network communication; C.2.4 [**Computer-Communication Networks**]: Network operating systems

General Terms

Design, Performance, Security, Experimentation

Keywords

Networking, Performance, Classification, Crossbow, QoS, Flows, DDoS

1. INTRODUCTION

Recent technological advances have resulted in the convergence of voice, video, multimedia, e-Commerce, and traditional data traffic on the Internet. However each type of traffic has different characteristics and requirements in terms of delay, jitter and bandwidth. In addition, Internet Service Providers (ISPs) have an obligation to support a level of service that customers paid for according to their service class. Thus there is a need to support QoS in a scalable and manageable way, with low performance overhead.

Many QoS models have been proposed. For example, Integrated Services (Intserv) [7] offers end-to-end guarantees about service levels. Other models, such as Differentiated Services (Diffserv) [6], specify service differentiation behaviors locally on a per-hop basis. Offering true end-to-end QoS requires support from all entities including the source host, routers and the destination. In this paper we propose a QoS model for the end hosts that is superior in performance and can be used in conjunction with Diffserv.

Crossbow is the code name of the new OpenSolaris networking stack which vertically integrates QoS functionality from the NIC hardware all the way up to the transport and socket layers. It uses NIC hardware features aggressively for performance, security isolation, and for meeting the QoS requirements of applications.

This paper first takes a look at the problems with existing QoS solutions. It then describes an architecture that addresses some of these problems and shows how the architecture can be used to mitigate DDoS attacks. It then proceeds to show how the architecture can also be used to build an end-to-end QoS solution that spans the enterprise data center. Finally, it explores other work happening in this area and describes future direction.

2. ISSUES IN EXISTING ARCHITECTURES

Performance overheads, complexity, scalability, deployment and manageability issues, are some of the issues facing various QoS solutions.

Intserv suffers from complexity and scalability issues [11] attributable to the complicated signaling mechanisms and the need to maintain flow related state in intermediate routers. Intserv was consequently never widely deployed. On the other hand, Diffserv, although simpler, does not specify end-to-end guarantees by itself. In [12] the author points out that though deployment of QoS mechanisms in the Internet remains sparse, diffserv represents a good start to address the real-world QoS needs.

On the host side, QoS has been traditionally implemented as a separate layer between the Data link and the Network (IP) layers. The QoS layer does the Diffserv processing that is needed. However this model creates a significant performance and scalability bottleneck on high bandwidth 10 Gigabit Ethernet networks. In addition recent CPU architectures [19] are moving towards a massively multi-threaded

WREN'09, August 21, 2009, Barcelona, Spain.
ACM 978-1-60558-443-0/09/08.

multi-core model rather than higher clock speeds. The cost of bringing the packet into the host and inspecting the packet headers has become increasingly prohibitive on such architectures and makes it almost impossible to honor the Service Level Agreement (SLA). Thus, there is a need to integrate the QoS functionality vertically with the network stack in order to amortize the various per packet costs and reduce the QoS overheads.

Intrusion Detection Systems constantly contend with DDoS attempts that exhaust CPU or network bandwidth resources [22]. Traditional QoS models do not solve this problem because they are structured high up in the stack and don't have a way to turn off the incoming attacking stream at the lowest level and to relieve system resources.

The essence of what QoS should be is lucidly brought out in [2] where the author points out the following QoS requirements: it must be bottom up, it needs to be supported at the lowest layer, below IP, and it needs to be extremely efficient and simple.

3. CROSSBOW ARCHITECTURE

Project Crossbow in OpenSolaris implements a new networking stack that has QoS functionality integrated in the stack itself instead of an add on layer. The approach uses NIC hardware classification and partitioning capability to allow the use of some of the most commonly used QoS features without any performance overheads. The project also attempts to make the concepts and configuration easy for users to understand and deploy.

The Crossbow Virtualization functionality [28] also takes advantage of the NIC hardware capabilities. A Crossbow virtualization lane consists of dedicated hardware and software resources for a particular type of traffic and defines a vertical path from the NIC hardware to the socket layer. NIC receive and transmit rings, interrupts etc. are examples of hardware resources, while kernel queues, threads, CPU bindings of kernel threads etc. are examples of software resources. One or more virtualization lanes may be assigned to a virtual machine.

Crossbow flows discussed in Section 3.1 are analogous to virtualization lanes. Flows may however be used even without any virtualization. Early packet classification and use of hardware rings achieve traffic separation and partitioning in both cases. Dynamic polling [28] on individual NIC receive rings, smooth transition between interrupts and polling, and support for multi-core CPUs contribute to performance in both cases. In the virtualization case, bandwidth limits are used to set the link speed of a virtual NIC.

The major components of the architecture discussed in this paper are already available for download as part of OpenSolaris kernel. Some of the features like hardware based Flows and enhancement to DDoS defense are works in progress.

3.1 Flows

Bernet et al.[5] outline an informal management model for the Diffserv architecture including meters, markers, queueing disciplines and shapers. Crossbow, on the other hand allows the creation of flows which implement a mostly queueless scheduling engine for packet processing. On the receive side, packets are allowed to come in the system when they are scheduled to be processed. Similarly, on the transmit side, the applications generating traffic are flow controlled as needed thus largely eliminating the need for queuing.

A flow essentially creates a data path from the hardware to the transport layer. The path is created by configuring classification rules in the NIC, which result in steering packets at the hardware level to an assigned receive ring. The stack can identify and schedule the packets for a flow even before it brings them into the system memory or look at any headers. Dynamic Polling on a per receive ring basis ensures that packets for a flow are allowed in the system based on their assigned service levels. To summarize, the following components make up the flow:

Classifying parameters (attributes) – These can be attributes from Layer 3 or Layer 4 headers and can include host and subnet IP addresses (local and remote with variable length netmasks), transport protocol (TCP, UDP, SCTP, ICMPv6, etc), ports (local and remote), DSCP bits, etc.

Properties – A property of a flow determines how packets for that flow will be treated. Properties can be bandwidth limits and guarantees, processing thread priorities, and processing CPUs.

System resources – Consist of the following hardware and software resources:

NIC resources – Hardware receive and transmit ring (groups) and classification rule. Most modern NICs [15] [25] [20] support multiple Receive and Transmit rings and hardware classification features.

MAC resources – The key MAC resource is the construct called *Softring Set* (SRS). The SRS is a FIFO based with an attached poll and worker thread. It also implements the packet scheduling based on backlog, and specified bandwidth limits or guarantees.

On the receive side, a one to one mapping exists between a SRS and a NIC hardware receive ring. Thus, the SRS can switch the hardware ring between interrupt and poll mode without impacting any other flow or traffic. In addition, a SRS can also have a collection of *softrings* (hence the name softring set) which emulates the hardware ring group. Softrings are also queues with a worker thread running on unique CPUs (where possible) that are assigned to the softring. The purpose of softrings is to offer software based fanout to spread the incoming packet processing across multiple CPUs.

On the transmit side, the SRS can have a direct relation with a hardware transmit ring. The SRS schedules the packet transmission, manages the driver transmit buffers and flow controls the application when transmit ring is running out of buffers.

IP and transport layer resources – TCP and SCTP have vertical perimeter (squeues) [27], which includes squeue poll and worker threads. Typically, a unique squeue is assigned to the SRS (in the absence of rings) or to each soft ring within the SRS when software based fanout is enabled for the flow.

3.2 Receive-Side Processing

Packets for a flow are classified by the NIC hardware into the Receive ring for the flow. They enter the MAC layer either through the interrupt path or as a result of being polled by the MAC SRS's poll thread.

Bandwidth control is implemented by a simple average rate meter with the average computed every fixed period (currently 10 milli second) and enforced by the received side SRS. When the bandwidth limit is reached for the specified period, the SRS switches the receive ring into poll mode and packets are left in the hardware receive ring to be picked up by the poll thread according to the bandwidth constraints. The interrupt mode is enabled when there is no backlog (i.e. queued packets) and the arrival rate is within specified bandwidth limits for that particular period.

In the case of TCP, if the squeue cannot keep up with the processing, it sets the underlying SRS or soft ring in poll mode. In turn, the SRS switches the receive ring to poll mode. In poll mode, the SRS attempts to continue polling periodically based on a low and high water mark to keep traffic flowing. However, if there is no backlog or new packet arrival, the SRS can set the receive ring back in interrupt mode.

It is worth noting that the entire system of SRS (any soft rings) and squeue provides a contention free path without the need for any fine grained locks. Furthermore, the packets are normally not queued in the system at all because as soon as any backlog starts to build up, the hardware receive ring is switched to the poll mode and acts as the only queue in the path.

Figure 1 illustrates the TCP receive path for a flow. When the flow is added (e.g. flow for TCP packets), Crossbow programs the NIC's classifier to steer all TCP packets to a ring. Packets get either immediately delivered to SRS through interrupt mode or are picked later by means of an SRS poll thread.

The Crossbow MAC layer has a full featured software classification engine. It is used when the NIC is not capable of classifying based on L3/L4 headers, or is out of hardware receive rings. Software classification is performed very early to assign a packet to a flow and steer the packet to the SRS associated with the flow. The system can work in hybrid mode where hardware based flows can be combined with an unlimited number of software based flows. Traffic arriving through the default receive ring is software classified and delivered to the software based flows.

3.3 Transmit-Side Processing

On the transmit side, the application thread sends data directly to the flow's SRS. The packets are sent directly to the NIC's transmit routine provided that transmit buffers are available and the bandwidth limit for that period is not being exceeded. As a host, Crossbow uses application flow control on outbound data traffic rather than dropping packets. When a NIC runs out of descriptors, or if the bandwidth is being exceeded, the SRS exerts back pressure and the client is blocked from sending further data down. When the NIC has transmit descriptors available, it sends a notification to MAC to remove the blocked condition on the SRS. In turn, the IP layer is notified by the SRS to enable the client to resume sending data.

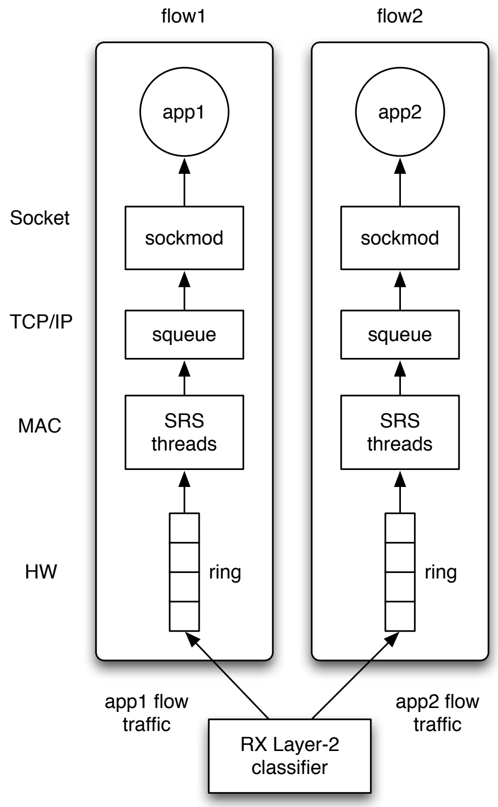

Figure 1: Crossbow hardware flows

3.4 Types of flows

As mentioned in Section 3.1, layer 3 or 4 attributes may be used to specify a flow over an interface or data link. The following types of flows are fairly common and are supported efficiently.

Service-based flow – Services are typically defined as a combination of a particular transport and well-known port. For example an HTTP service that uses TCP protocol over port 80 can be assigned its own resources by creating a TCP flow for the above protocol and local port combination.

IP address-based flow – Traffic that uses a particular local or remote IP address may be given its own resources by creating a flow that specifies the desired local or remote IP address(es).

IP subnet-based flow – Specifying a combination of IP address and subnet mask creates an IP subnet-based flow that can be given its own resources.

DSCP label-based flows – This is done by specifying the DSCP bits and the DSCP mask that is to be applied on incoming packets.

While it is possible to design a very generic classifier to handle any arbitrary combination of attributes, the challenge is to achieve it with minimal performance impact and also use NIC hardware classification support.

3.5 TCP and UDP flows

Given the extensive use of TCP and UDP protocols, we discuss some more details about TCP and UDP service based flows in this section. TCP is very sensitive to packet drops. It interprets packet loss as a measure of link congestion and self-paces its throughput accordingly. It is well known [16] that the bandwidth delay product of a TCP connection in steady state corresponds to the capacity of the channel between sender and receiver. Efficient bandwidth control is achieved by controlling that delay and minimizing packet drop.

In the case of a hardware based TCP flow, the dynamic polling mechanism of NIC receive rings implements bandwidth control on the inbound side. As mentioned in Section 3.2 the SRS polling thread picks up only as many packets from the NIC receive ring as allowed by the configured bandwidth limit. Bandwidth control for outbound packets is implemented similarly by introducing a suitable delay corresponding to the bandwidth limit. As mentioned in Section 3.3 the application is flow controlled, and thereby prevented from sending more data without packet drops. Inbound packets are not dropped as long as the NIC receive ring has sufficient receive descriptors and buffers to hold the entire TCP window's worth of data.

The bandwidth control mechanisms for UDP are similar to TCP on both inbound and outbound directions. However the UDP protocol does not have a built-in flow control mechanism. If the arrival rate of incoming packets for a UDP flow is greater than the bandwidth limit, the NIC receive ring will eventually fill up, and incoming packets will be dropped by the NIC. As it is the case for TCP traffic, the outbound bandwidth usage is regulated by flow controlling the application.

In the case of software based flows, a software queue is used in place of the NIC receive ring, and the bandwidth usage of inbound packets is regulated using a simple tail-drop mechanism. Hardware based flows inherit the drop model implemented by the NIC hardware.

While a single TCP connection can use at most 1 ring to avoid costly reordering, multiple connections of a high bandwidth flow can be spread among multiple hardware rings. Outbound traffic is spread across multiple transmit rings by the Crossbow stack, and inbound traffic is spread between multiple receive rings by the NIC hardware. When a bandwidth controlled flow is spread across multiple rings, the bandwidth counter for that flow is shared among the rings. This sharing introduces fairness issues in polling mode, which is an area for future work.

3.6 Flow Performance

Most of the QoS solutions function well on transmit side. However, bringing the packet into the system and looking at headers to identify either the packet destination or the QoS class constitute a major cost. Thus, QoS solutions are less efficient when processing received network traffic.

Figure 2, shows the advantage of Crossbow flows. The Software based flow performs some of the same operations as the Solaris 10 IPQoS [26]. However, by being integrated in the stack, Crossbow flows perform better than a traditional layered IPQoS implementation. The hardware based flows perform best with almost no overheads. The classification occurs in the hardware and an independent path exists through the stack. In addition, sizable advantages can be obtained by using Dynamic Polling to schedule packet processing on the receive side.

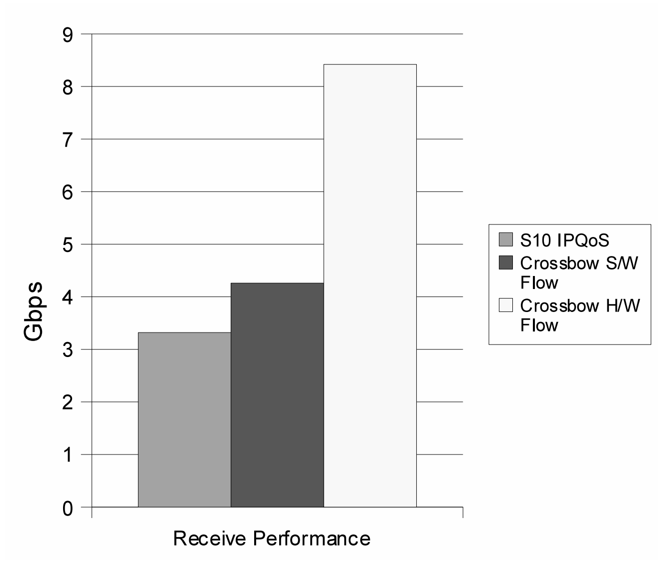

Figure 2: Receive Performance with a simple TCP based rule

Configured BW	Fedora 2.6 TC	Crossbow HW Flow
1Gbps	0.4Mbps	0.95Gbps
2Gbps	0.7Mbps	1.87Gbps
3Gbps	1.13Mbps	2.86Gbps
4Gbps	1.26Mbps	3.81Gbps
5Gbps	1.24Mbps	4.40Gbps

Table 1: Measured TCP bandwidth with Fedora 2.6 TC and Crossbow hardware flow vs configured bandwidth

In Table 1, we compare Crossbow hardware based flows with Fedora 2.6 TC[1] under bandwidth limits to see how well the integrated hardware and software approach worked. As it is evident from the graph, Crossbow hardware based flows allowed TCP to get very close to the bandwidth limit while Fedora 2.6 TC does not allow the throughput beyond a few megabits even when the configured limits were in gigabit range on a 10 Gb/s network. The issue was that TC does not schedule receive side processing and any packets exceeding the buffer limit are dropped hurting TCP performance. Crossbow Hardware based flow on the other hand employs the NIC receive ring and helps to adjust TCP RTT instead of dropping the packets. It would be worthwhile to note that on transmit experiments, both Crossbow and TC were able to get close to the configured limits. The commands necessary to configure both Crossbow and TC are shown in Section 5.

For all experiments, the test setup consisted of four identical machines. One machine acted as a system under test (SUT) and the other three machines acted as clients. Each machine was dual socket, quad core Intel machine with each core operating at 2.8 GHz. All four machines had an Intel 10 Gigabit Ethernet NIC and were connected to a dedicated Foundry 10 Gigabit Ethernet switch. Each client was sending 10 TCP streams to the SUT. The wire MTU was 1500 bytes, and the application write buffer size was 8 Kbytes.

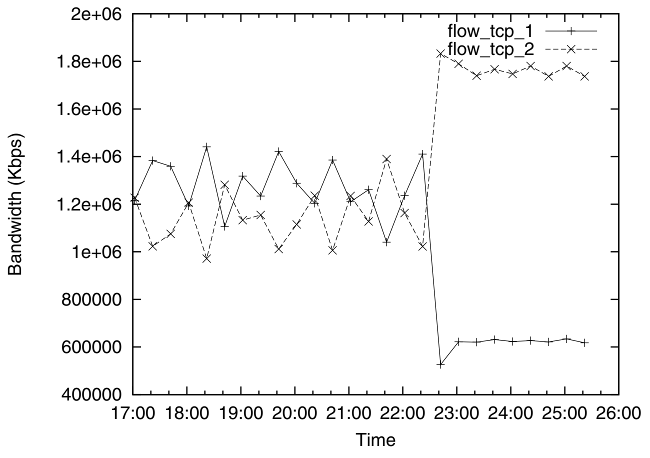

Figure 3: Throughput of competing TCP flows with different maximum bandwidth property

We used an application called "uperf" [4], which is similar to other micro benchmarks but allows us to deal with a multi client case more effectively thus better simulating a real world performance.

It is possible that TC may show much better results with sufficient tuning or different queue implementation which forces TCP to rate limit itself more effectively instead of dropping packets.

3.7 Bandwidth Sharing Between Flows

Unless resource constraints are specified on flows defined over a datalink, the flows share the link's available bandwidth. The sum of the bandwidths used by the flows will never exceed the link bandwidth. Flows may be created for observation only (Section 5), in which case they will use as much bandwidth as generated by the client applications. The scheduler assigns resources to the flows on a best-effort basis. Figure 3 shows an example of historical bandwidth usage collected from two competing TCP flows. In this case, two identical instances of the same application (netperf) were used to generate traffic through both flows simultaneously. In the first part of the curve, we see the two flows consume comparable amounts of bandwidth, one occasionally exceeding the other's share, but with no particular preference. In the middle of the experiment, at around the time point 22:50 (min:sec), the `maxbw` property was set on `flow_tcp1` flow to cap its bandwidth consumption at 800 Mb/s. After an initial short dip in bandwidth usage, `flow_tcp1` reaches a steady speed. TCP interprets a sudden loss of packets as encountering congestion conditions, and backs off rather aggressively in response, which explains the dip [16]. It is interesting to observe how the second flow, faced with less competition, absorbed the remaining available bandwidth.

3.8 CPUs Bindings and Priority Assignment for Flows

As mentioned in Section 3, one of the key differentiators of Crossbow is that Crossbow does not rely on queuing then sorting packets according their matching flow properties. Instead, it relies on an opportunistic and efficient classification to steer packets to flows and letting the SRS schedule the packet processing.

Flows	Priority	Min BW	Max BW	Avg BW
flow1	high	365	415	390
flow2	low	291	360	338

Table 2: Observed traffic bandwidth in Mb/s for different flow priorities

Configured flow priorities (Low, Medium, and High) are translated to priorities of the kernel threads that are associated with the flows. The scheduler elects threads to move to the runnable queues based on the priority. The higher the priority, the sooner a thread will execute on the CPU. Unlike user threads, kernel threads do execute until they are interrupted or they voluntarily relinquish the CPU [17] when they are blocked.

On the transmit side, threads associated with flows are blocked for pacing or bandwidth control purposes, until a subsequent notification to resume sending data is received. Over a long period of time, the scheduler will pick threads from higher priority flows ahead of the lower priority ones.

On the receive side, the flows polling threads are blocked during interrupt mode, or whenever the flow has consumed its allotted amount of packets for the time period. The priority is applied when the polling thread is reactivated to pickup more packets for the flow. The polling threads with higher priority tend to be selected more often by the kernel scheduler. Higher priority flows relieve the hardware rings first. Consequently, the risk that these flows might drop packets because of hardware overruns is greatly reduced.

Our experiments show that different priorities do not always translate to an observed difference in the bandwidth awarded to competing flows, as shown by Table 2. In fact, it takes at least twenty kernel threads in the runnable queue to saturate a CPU, after which we start to see a marked difference in the bandwidth usage between the threads.

Under lower load, the CPUs were fast enough to process most of packets for all priorities, and the ranges of measured bandwidths consumption from competing flows were overlapping.

3.9 Bandwidth Control Challenges

In order to keep the bandwidth limit mechanism simple and without causing high CPU overheads, the Crossbow architecture imposes the following constraints:

Bandwidth control resolution – To minimize the CPU utilization overheads, the Crossbow architecture utilizes the system tick mechanism which is by default updated every 10 millisecond. Using a high resolution timer causes higher per-packet processing cost under lower bandwidth limits.

No carryover of unused bandwidth across ticks – The bandwidth limits are enforced at the granularity of the ticks, and if the per-tick limit is not consumed, it does not carryover to the next interval.

No splitting of packets – Again, in order to minimize the overheads, packets are not split on the receive side. A packet is processed in its entirety, or not at all.

Together, these constraints cause at least one packet to be processed per interval. For example, assuming 1500 bytes packets, the minimum enforceable bandwidth limit is 1.2 Mb/s, or 1500 bytes every 10 millisecond. Lower bandwidth limits are often desirable, for example when using DSL or wireless links. In addition, because a bandwidth limit is not a hard maximum, a whole packet is processed as soon as at least one byte can be allowed during an interval. In this case, the receive side can process as much as 1499 more bytes per tick than expected, resulting in giving about 1.19 Mb/s extra bandwidth to any configured bandwidth limit. For limits of 100 Mb/s or more, these limitations are not significant, but they can be non-negligible for lower limits.

Work is in progress to allow carrying over unused bandwidth between consecutive intervals. In this case, the imposed bandwidth limits can become hard limits, where a packet is processed only if it's allowed in its entirety. Otherwise, any unused bandwidth is carried over to the next interval. This will allow for a more accurate enforcement of low bandwidth limits.

4. DDOS ATTACKS MITIGATION

DDoS (Distributed Denial of Service) attacks are among the most common threats on IP networks today [21]. In a DDoS scenario, a collection of hosts attack a particular set of servers and causes these servers to become unresponsive. The attacking hosts are typically hosts that have been compromised by various malware such as internet worms, and they work in concert when attacking targets.

Although several types of DDoS attacks exist, the most popular is the SYN flooding attack [24]. In this type of attack, target systems are bombarded with TCP connection requests (TCP SYN messages) to keep the systems' inbound connection queues full and cause legitimate requests to be dropped.

Several mechanisms have been proposed to detect [30] and defend against SYN flooding attacks [23]. Some mechanisms define methods which allow a network under attack to build a list of attacking system addresses [10]. That list is then given to a firewall. The firewall drops traffic from attacking hosts while letting legitimate traffic go through to the servers.

Relying on a firewall however has drawbacks. New hardware dedicated to building and maintaining a list of attacking systems must be deployed and maintained and can add to cost. Constant passing of tables between hosts and the firewall also introduces overhead and complexity.

With Crossbow, the list of addresses for the attacking systems can be programmed into the NIC in the form of hardware flows. These hardware flows can be then either associated with a special rule to drop the packets in hardware without passing them to the host. Or, the stack can program the hardware classifier to steer the suspect traffic to a dedicated hardware ring with very low bandwidth with that ring. This latter option would allow the host to capture information about the attacking traffic if desired. In each case, the processing associated with filtering the packets is done by the NIC hardware itself, and allows the host to be shielded from the attack and continue to service legitimate traffic. This approach is similar to firewalls black list filtering techniques.

Another method to mitigate threats of DDoS attacks is to use a combination of a dynamically populated white list of flows together with client puzzles [3]. Puzzles require the client to commit computational resources to solve a cryptographic puzzle before it is authenticated with the server, and before more server resources are allocated to that client. Servers can choose to enable puzzles when they suspect that they are under attack.

When Crossbow is applied to that approach, all traffic by default share a common hardware ring with an associated low to medium bandwidth limit and priority. When the session with a client is authenticated, the server can then create flows for the TCP connections associated with that session, effectively adding it to a white list of permitted flows. These flows are given a set of hardware rings with a higher priority and bandwidth. Both the application process and the clients are running on the same host. Therefore, the flows can be very efficiently passed from the application to the hardware classifier on the NIC.

Some NIC hardware implementations [20] also provide the ability to classify TCP SYN packets apart from the rest of the traffic. In that case, the SYN packets could be sent to a dedicated hardware ring which allow new connections to be throttled before they are delivered to the application.

5. EASE OF MANAGEMENT

Configuring policies by using IPQoS on the Solaris 10 release or TC on Linux require defining filters, classes, queuing disciplines, and a mapping among these components. This implementation provides QoS flexibility but also adds complexity. Crossbow provides a simple mechanism for configuring and managing flows. For example, a policy for applying a bandwidth limit of 100 Mbps for TCP packets on the interface bge0 can be configured by using the `flowadm(1m)` command-line utility, as follows:

```
# flowadm add-flow -l bge0 -a transport=tcp \
-p maxbw=100 tcp-flow
```

This command creates a flow called `tcp-flow` that applies a limit of 100 Mbps to all the TCP packets on the link bge0. The bandwidth limit can be modified by using the `set-flowprop` subcommand to set the `maxbw` property. Flows can be deleted using the `remove-flow` subcommand of `flowadm(1M)`.

Flow configuration can be displayed by using the `show-flow` subcommand. In addition to managing flows, Crossbow provides simple mechanisms with the `flowstat(1M)` command to monitor the statistics for the various flows that are configured. For example, to obtain running statistics for `tcp-flow` on the system, use the following command:

```
# flowstat show -i 5 tcp-flow
```

The statistics for the `tcp-flow` (input and output bytes and packets, drops, errors) will be printed every 5 seconds. If desired, the information can be limited to either the receive or transmit side statistics. Statistics aggregated over a certain period of time can also be obtained. The `flowstat(1M)` command supports several options to display statistics for flows.

Apart from the running statistics, Crossbow also uses `acctadm(1M)` to log statistics that can then be retrieved to get historical usage information. `acctadm(1M)` is used to enable the logging of networking information to a specific file. `flowstat(1M)` can then read the log file and display output in the required format, including statistics based on date, time, and so on. For example, to query the usage for `tcp-flow` on date `D1` from starting time `shh:smm:sss` to ending time `ehh:emm:ess`, use the following syntax:

```
# flowstat show-history -f logfile \
-s D1,shh:smm:sss -e D1,ehh:emm:ess tcp-flow
```

To obtain a graphical presentation of the use of `tcp-flow`, the output can be generated as a plot file that is fed directly to a plotting utility such as `gnuplot`.

In contrast, setting a similar receive and transmit side policy with Fedora 2.6 TC requires following set of commands:

```
# tc qdisc add dev eth4 handle ffff: ingress

# tc filter add dev eth4 parent ffff: \
protocol ip prio 20 \
u32 match ip protocol 6 0xff \
police rate 1Gbit buffer 1M drop flowid :1

# tc qdisc add dev eth4 root handle 1:0 \
cbq bandwidth 10Gbit avpkt 1000 cell 8

# tc class add dev eth4 parent 1:0 \
classid 1:1 cbq bandwidth 10Gbit \
rate 4Gbit weight 0.4Gbit prio 8 \
allot 1514 cell 8 maxburst 20 \
avpkt 1000 bounded

# tc class add dev eth4 parent 1:1 \
classid 1:3 cbq bandwidth 10Gbit \
rate 1Gbit weight 0.4Gbit prio 5 \
allot 1514 cell 8 maxburst 20 \
avpkt 1000

# tc class add dev eth4 parent 1:1
classid 1:4 cbq bandwidth 10Gbit \
rate 5Gbit weight 0.3Gbit prio 5 \
allot 1514 cell 8 maxburst 20 \
avpkt 1000

# tc qdisc add dev eth4 parent 1:3 \
handle 30: sfq
# tc qdisc add dev eth4 parent 1:4 \
handle 40: sfq

# tc filter add dev eth4 parent 1:0 \
protocol ip prio 1 u32 match ip \
protocol 6 0xff flowid 1:3
```

6. END-TO-END QOS

This section shows how to extend the features of the Crossbow architecture to deploy an end-to-end QoS solution across a network.

Building a true end-to-end QoS solution that is easily deployable over existing IP networks has been a challenge. Diffserv by itself only provides an assured per-hop behavior, rather than end-to-end QoS. In this section we show how the Crossbow architecture in conjunction with Diffserv can be deployed in an enterprise data center environment to achieve service guarantees that closely resemble a true end?to?end QoS solution. With Crossbow, this configuration is attainable with minimum impact on performance, scalability, and management complexity.

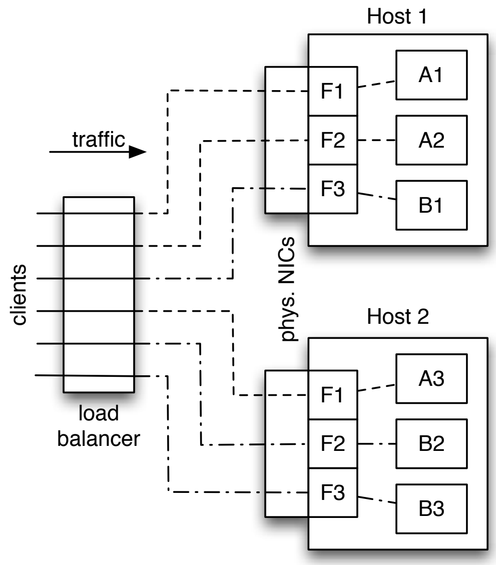

Figure 4: End-to-end QoS deployment example

Horizontally scaled applications are deployed in a multi-tier architecture consisting of front-end load balancers, mid-tier web and application servers, and back-end database servers. The load balancer hosts the well known IP address (or Virtual IP address) of the service and directs incoming connections to a set of back-end servers. The load balancer spreads requests to the back-end servers using either pre-existing TCP connections or dynamically created TCP connections. The load balancer can also embed client Service Level Agreement (SLA) related information in the application level headers.

Diffserv uses the 6-bit DSCP field in the IP header to segregate traffic into distinct classes which can then be treated differently. In Crossbow, packets can be classified based on the value in the DSCP field, and associate properties such as bandwidth limit, bandwidth guarantee, priority, and so on, to these classified packets. Similarly, on the outbound side we set the DSCP fields of the packets based on the flow properties.

Figure 4 shows a Crossbow based load balancer at the left that is located at the ingress point of the Data Center. The load balancer classifies incoming packets into flows based on their headers, and marks them with an appropriate DSCP label. The packets then traverse the network to the nodes. Every node running the Crossbow architecture can classify the packets based on their packet headers and DSCP label into the right flows (one of F1, F2 or F3 in our example). Each flow is in turn associated with its dedicated share of hardware and software resources.

Thus, with hardware supported flow classification, dynamic polling, and bandwidth guarantees, our architecture supports end-to-end QoS guarantees across an enterprise network provided that the intermediate routers are also Diffserv-aware.

7. RELATED WORK

In [14], the authors propose a QoS architecture which targets high-bandwidth applications such as visualization applications. That architecture relies on the network components and APIs to support the requirements of these applications. On the hosts, the authors rely on a real-time CPU scheduler to reserve CPU resources according to the need of the applications. In [9], the authors focus on scheduling applications based on system and network status. These proposals do not address the issues of the host network stack performance and efficiency, which we have shown can have a direct impact on the bandwidth available to applications.

W.-Y. Cai et al. in [8] present a network stack based on a cross-layer design for QoS of wireless sensor networks. With that approach, different layers of the stack, whether adjacent or not, are coupled through feedback modules. The lack of results presented by the authors do not allow us to evaluate the overhead required to implement these cross-layer exchanges, and it is not clear if it can be applied to other network stacks and applications in general.

Exokernel [13], Nemesis [29] and Piglet [18] are examples of vertical operating systems. The vertical structure involves moving most of the functionality of the operating system into shared libraries that become part of the application. The privilege boundary is moved up, and the shared kernel is small, consisting of just the scheduler and interrupt and trap handlers. Resource management is moved up from the kernel to the applications. The key issue of these architectures is that they give untrusted applications direct access to the underlying hardware resources.

Nemesis uses a single virtual address space which enables it to efficiently and easily share data with applications. Nemesis also uses bandwidth reservation to achieve QoS guarantees. However a single address space is limiting, in particular considering the trend of consolidating more services and applications on the same hosts.

Piglet is a vertical operating system that is based on the principle of dedicating a system CPU to executing an instantiation of the OS. This is somewhat similar to a Crossbow flow which extends vertically from the NIC to the socket layer. However unlike Piglet, Crossbow flows don't extend into the application since Crossbow is based on a conventional OS.

8. CONCLUSION AND FUTURE WORK

The Crossbow architecture presented in this paper is a novel way to implement QoS on a server host. We achieve minimal performance impact, by integrating the QoS functionality vertically with the networking stack, and by leveraging hardware support whenever possible. The model uses independent queue-less traffic paths with dedicated hardware and software resources for different flows, all the way from the NIC to the socket layer. This feature, together with dynamic polling of the NIC on the receive side, and application level flow control on the transmit side, ensure that the host QoS requirements are met. The architecture handles application misbehavior and DDoS attacks to ensure that CPU or bandwidth will not be exhausted. Our architecture also provides a simple management interface to enable and configure QoS.

In the future, DSCP marking on the outgoing packets will be implemented, as well as full hardware classification, including DSCP labels, on incoming traffic. Likewise, bandwidth guarantees per flow as well as memory pools will be supported, in addition to the flow priorities and bandwidth limits currently being offered. These new features, along with the existing Crossbow framework, will enable a network of Crossbow nodes and Diffserv aware routers to support end-to-end QoS. We plan to support lower bandwidth limits and more accurate bandwidth enforcement.

9. ACKNOWLEDGEMENTS

Acknowledgements are duly given to the Crossbow project team for bringing the architecture to life. Special thanks to Shrikrishna Khare for helping with the Fedora 2.6 TC experiments.

10. REFERENCES

[1] W. Almesberger. Linux network tracffic control.

[2] G. J. Armitage. Revisiting ip QoS: why do we care, what have learned? In *RIPQOS'03: Proceedings of the ACM SIGCOMM Workshop on Revisiting IP QoS*, 2003.

[3] T. Aura, P. Nikander, and J. Leiwo. DOS-resistant authentication with client puzzles. In *Lecture Notes in Computer Science*, pages 170–177. Springer-Verlag, 2000.

[4] A. Banerjee. Introducing uperf - an open source network performance measurement tool, 2008.

[5] Y. Bernet, S. Blake, D. Grossman, and A. Smith. RFC 3290, an informal management model for Diffserv routers, 2002.

[6] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC 2475, an architecture for differentiates service, 1998.

[7] R. Braden, D. Clark, and S. Shenker. RFC 1633, integrated services in the internet architecture: an overview, 1994.

[8] W.-Y. Cai and H.-B. Yang. Cross-layer QoS optimization design for wireless sensor networks. In *Wireless, Mobile and Sensor Networks, 2007. (CCWMSN07)*, 2007.

[9] A. Caminero, O. Rana, B. Caminero, and C. Carrion. An autonomic network-aware scheduling architecture for grid computing. In *ACM/IFIP/USENIX 8th International Middleware Conference*, 2007.

[10] S. Chen, Y. Tang, and W. Du. Stateful DDoS attacks and targeted filtering. *J. Network and Computer Applications*, 30:823–840, 2007.

[11] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. QoS's downfall: at the bottom, or not at all! In *RIPQOS'03: Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS*, 2003.

[12] B. Davie. Deployment experience with differentiated services. In *ACM SIGCOMM 2003 Workshops, Aug 2003, Karlsruhe, Germany*, pages 131–136, 2003.

[13] D. Engler, M. Kaashoek, and O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM Symposium on Operating systems principles*, 1995.

[14] I. Foster, M. Fidler, A. Roy, V. Sander, and L. Winkler. End-to-end quality of service for high-end applications. *Computer Communications*, 27, September 2004.

[15] Intel. Intel 82598 10GbE Ethernet Controller Open Source Datasheet, 2008.

[16] V. Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM '88*, 1988.

[17] R. McDougall and J. Mauro. Solaris internals(tm): Solaris 10 and OpenSolaris kernel architecture (2nd edition) (solaris series) (hardcover), 2006.

[18] S. Muir and J. Smith. Piglet: A low-intrusion vertical operating system. Technical report, Department of Computer and Information Science, University of Pennsylvania, 2000.

[19] Nawathe, Hassan, Yen, Kumar, Ramachandran, and Greenhill. Implementation of an 8-core, 64-thread, power-efficient SPARC server on a chip. *IEEE Journal of Solid-State Circuits*, 43:6–20, January 2008.

[20] Neterion. Neterion Xframe II 10 Gigabit Ethernet.

[21] S. Northcutt and J. Novak. *Network Intrusion Detection (3rd Edition)*. New Riders Publishing, ISBN, 2002.

[22] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. white paper, 1998.

[23] L. Ricciulli, P. Lincoln, and P. Kakkar. TCP SYN flooding defense. In *Proceedings of CNDS*, 1999.

[24] 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 *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 208–223. IEEE Computer Society Press, 1997.

[25] Sun Microsystems. Sun Multithreaded 10GbE (Gigabit Ethernet) Networking Cards, 2007.

[26] Sun Microsystems. Solaris 10 system administration guide: IP services (network resource management using IPQoS), 2008.

[27] S. Tripathi. FireEngine - a new network architecture for Solaris OS, 2004.

[28] S. Tripathi, N. Droux, T. Srinivasan, and K. Belgaied. Crossbow: From hardware virtualized NICs to virtualized networks. In *Proceedings of the ACM SIGCOMM workshop VISA'09 (To appear)*, 2009.

[29] T. Voigt and B. Ahlgren. Scheduling TCP in the Nemesis operating system. In *IFIP WG 6.1/WG 6.4 International Workshop on Protocols for High-Speed Networks*, 1999.

[30] H. Wang, D. Zhang, and K. G. Shin. Detecting SYN flooding attacks. In *Proceedings of the IEEE Infocom*, pages 1530–1539, 2002.