SMI: SNT / NTO / Sun Labs Internship Progress Report: NEON Summer 2003

Author: Ioan Raicu

Document Created:
6/2/2003

Last Date Modified:
9/1/2004

Table of Contents

This document has been edited for proprietary information that is protected by a non-disclosure agreement between Sun Microsystems Inc. and Ioan Raicu.

1.0 Introduction

I will be spending 1/5 of my time working on the Neon project. I will attempt to write up a detailed report on how the Neon architecture could be validated. I will address the construction of the traffic generator, which will be critical component in validating Neon's functionality. I will investigate the various options for how we can automatically / semi-automatic evaluate the Neon prototype. We are not concerned with validating the performance of Neon at this moment, but a general enough architecture that could potentially be used to validate the performance as well would be a plus.

After talking to Jason Goldschmidt, Valerie Bubb, Bruce Curtis, Michael Speer, and Paul Wernau in regards to their work dealing with validating network systems, we came to the conclusion that a traffic generator that would be flexible enough for evaluating such a complex and general architecture as Neon would most likely have to be made inhouse. The suggestions we had was to look into a utility called wget (rather simplistic), another called ISIC (written by a SunScreen intern from Purdue), and SpecWeb; there are some other as well, such as NetPerf, ttcp, etc… (see section 3.1.1 for more details), but these are great for performance testing, but lack the flexibility to be used as a functionality validation tool. The traffic generator that I will be specifying will be great for functionality testing, but it will probably not be appropriate for performance testing, except for simple connectionless protocols that do not require complex protocol interaction (ex. TCP). Therefore, for connection oriented performance testing, some off-the-shelf traffic generator will probably be better suited.

I have identified at least two different tests that must be performed to validate Neon. The first needs to validate the rule crunching algorithm in the FM (Flow Manager) while the second needs to validate the functionality of the particular implementation of the network service in the FED. A potential third test could be to test the performance of Neon in comparison to other architectures.

The biggest challenge will be to investigate the various methods for generating the necessary traffic and the means of collecting the necessary statistics (in order to be able to build a meaningful automated report). Due to the small amount of time I could devote (and there is left in my internship) to this project, the deliverables are not very firm or even realistic, but if I had more time, the deliverables would be:

- FED Modifications
    - Suggestions on how the TAG functionality could be implemented on the FED
    - Details on what kind of TAG information is necessary in the FED
- Traffic Generator (TG)
    - Detailed description of the traffic generator functionality in order to support a variety of service applications

  - A bare bone prototype of the traffic generator that is extensible so new protocols could be added with ease, and the user would have full control of all the field values at all layers

- Traffic Evaluator (TE)
  - Detailed description on what the role of the TE and a barebone prototype implementation
    - Evaluate traffic based on header fields; no modifications necessary at the FED
    - Evaluate traffic based on TAG information; TAG information is added at the FED
  - Detailed description of the report that would be generated by the TE

Given the fact that time is so limited, I hope to address most of the issues in the above points at least on a theoretical level, but I doubt I will have any time to actually implement anything within this internship.

2.0 Validate NEON Architecture

The two main components of the NEON architecture are: the flow manager (FM) and the flow enforcement device (FED). Figure 1, Figure 2, and Figure 3 are all taken from [1], and are the basics of the Neon architecture.

Figure 1: Neon architecture: general overview [1]

Figure 2: Neon architecture: data plane [1]

Figure 3: Neon architecture: control plane [1]

The Flow Enforcer Device (FED) will most likely be a network processor due its great attributes, ease of programmability, yet can handle line rate (faster than OC-48) while performing full header processing and even deep content inspection. In order to place everything into perspective, a quick description on how a network processor (NP) works is done below.

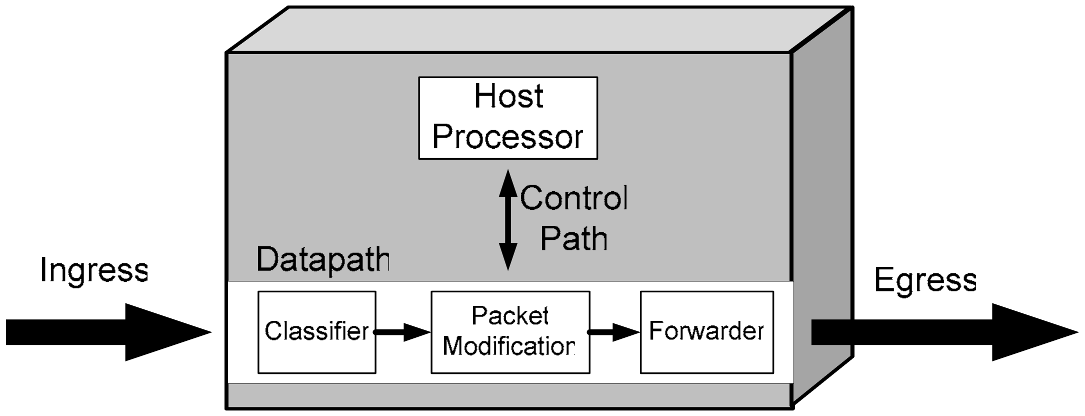

Figure 4: Network Processor logical overview of the flow of packets

Packets come in the ingress port and follow the datapath all the way to the egress port. Under normal circumstances, the NP will be able to make all the decisions in the fast path and keep line rate. In the event at some point, the NP does not know what to do with a particular packet, then that packet is sent up to the host processor for further and more complex processing. If a packet is taken out of the data path and sent to the host processor, then line rate cannot be maintained, and thus this exception handling is not a desirable action for the NP to take. Most NPs have some memory where several look-up tables are stored and are used throughout a packet's lifetime. These lookup tables can be used for classifying a particular packet, and as a result telling the later stages to perform a different action, such as forward to some specific egress port, drop the packet, or perhaps modify the packet in some way. These lookup tables are generally readable from the data path, but it is only writable from the host processor. This limitation usually means that for these tables to be updated from the flow of traffic in the datapath (dynamic updates), packets that the datapath does not know what to do with are sent up to the host processor, the appropriate tables are updated, and then the packets are injected back into the ingress port and the entire process starts all over again. Another way these tables can be updated is by getting new or updated tables from the Flow Manager (FM), which would be fed into the host processor, and finally the host processor would update the look-up tables.

2.1 Validate Flow Manager

Figure 5 shows the validating scenario for the FM. The overview of the configuration is that all the rules from the various network services are loaded into the FM, the rule cruncher algorithm performs its task, and the final condensed rule set is given as an input to the FED. The alternate scenario is where each network service is given a specific FM and FED, and therefore there should be no conflicts between the various rules. The FM's function is therefore simplified and the rules are merely fed into the FED. The TG (Traffic Generator) would generate the appropriate and identical traffic in both scenarios, while the TE (Traffic Evaluator) would analyze the traffic that came through. The TE would have the advantage that it is a general purpose traffic analyzer (supporting a specific set of protocols), that can be used to give statistics about the network segment it is listening on without modifying any of the traffic source in the network; this can be done by analyzing the packet headers of all the packets that reach the TE. The specifications of the TG and TE will be discussed in a later section; for the sake of explaining the overview, let us treat them as black boxes for now. The TE would automatically generate a report that contains statistics on the traffic that arrived at the TE. The reports in both scenarios need to be identical (within some small margin of error) in order to prove that the FM is really performing the correct functionality of resolving conflicting rules from various network services.

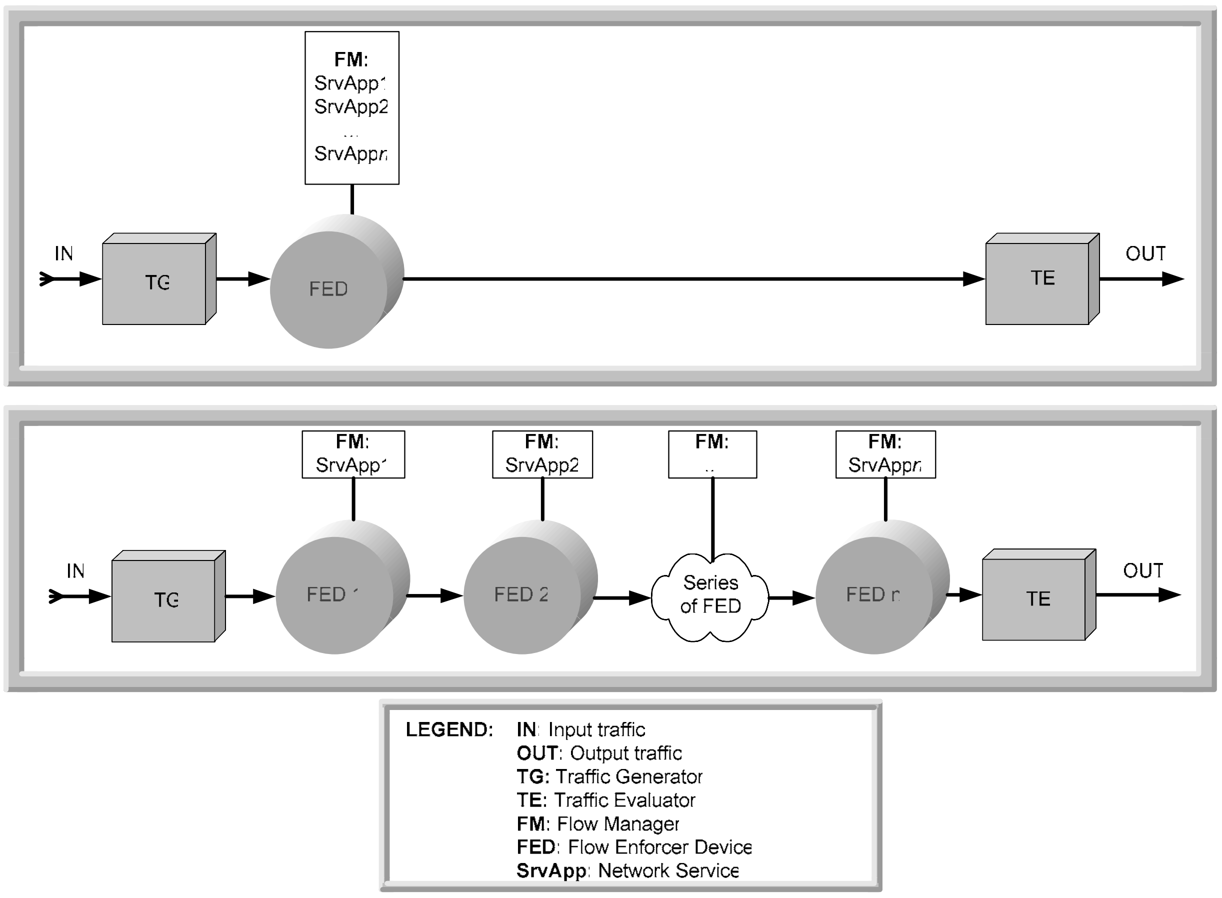

Figure 5: FM validating scenario

2.2 Validate FED Functionality

The previous setup in Figure 5 only validated the FM and did not tell us anything about the FED functionality. Figure 6 will capture the essence of how the FED functionality will be validated. The original FED would now become a tFED, which in addition to all the functionalities of the FED, it also has the ability to generate packets that contain TAG information regarding all packets that traverse the FED with a particular code based on the classification and / or action that was performed on the particular packet. The biggest difference between the TE and the tTE is that the new tTE would build the report based on the TAG information rather than the headers of the packets. Note that the out traffic is separate from the traffic that goes to the tTE since extra packets of TAG information are generated and forwarded to a different physical port and therefore remains separate from the actual egress data. Under normal circumstances (relatively large packet sizes, > a few hundred bytes), the tFED should have no problem keeping up with generating the TAG packets (which will be very small, ~64 bytes), but as we decrease the size of the packets that come in on the ingress port of the tFED, it the TAG functionality will incur an overwhelming burden on the tFED from a performance point of view in regards to keeping up with line rate (OC-48, etc…).

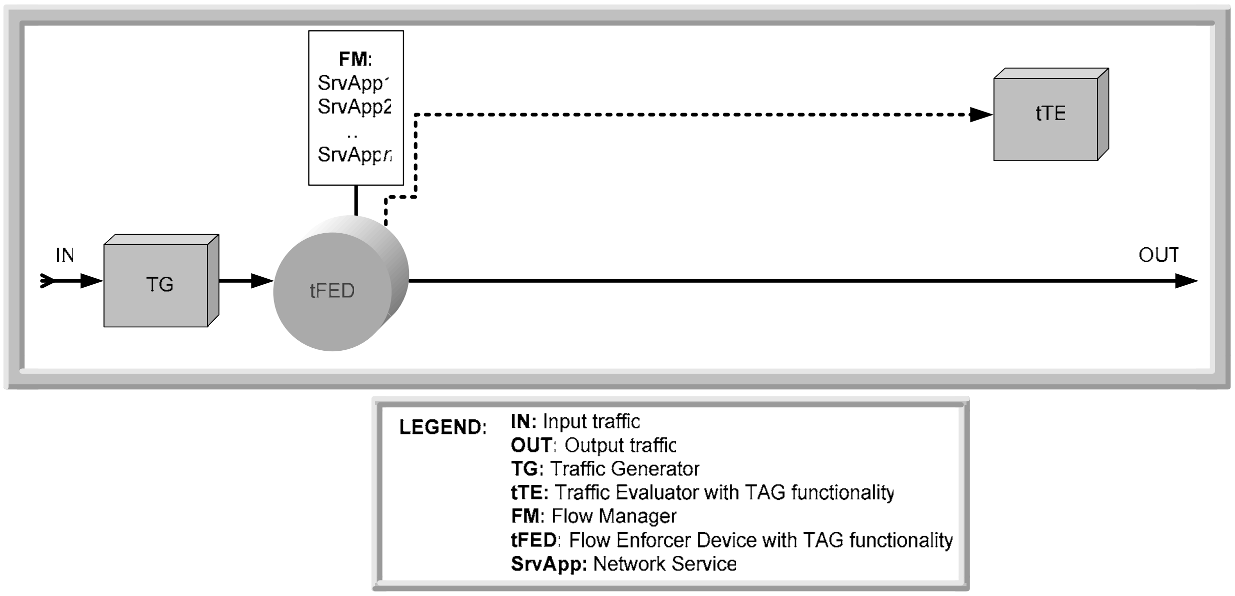

Figure 6: FED validating scenario

The biggest disadvantage of this approach is that the FED would have to be modified, which means that the tTE could not be used in a general scenario to generate a report because it is tightly couple with the tFED which is doing processing on the packets. Furthermore, the TAG functionality might impede performance since generating these packets and modifying their payload could be a very expensive task for most network processors (the target platform for the FED) if the FED is pushed to its limits with minimum size packets. A more valuable solution would be one that could avoid modifying the FED so the solution to this problem could be used as a generic traffic analyzer, performance meter, and functionality verification. This idea might not be realizable within my short internship, but I will look into at least some possible more generic solutions.

2.3 Performance of Neon

Figure 7 above depicts the scenario in which the performance of Neon could be easily compared to other competing architectures. In the example, Neon would do all the processing and decisions within a single FM and FED, while the rack of boxes would do the appropriate processing and pass the packet to the next box for further processing. This test scenario would actually show off all the benefits of Neon in a real world scenario, but is most likely prohibitively expensive to realize. A more realistic scenario which might help prove that it is better to have all the processing and decision making in one place (the FM and FED) is identical to the scenario depicted in Figure 5. This would show that the time saved to do the classification on each packet at each network service is significant enough that it is worth having an architecture such as Neon as long as the hardware on which the FED is built upon can keep up with the processing for the intended data rates.

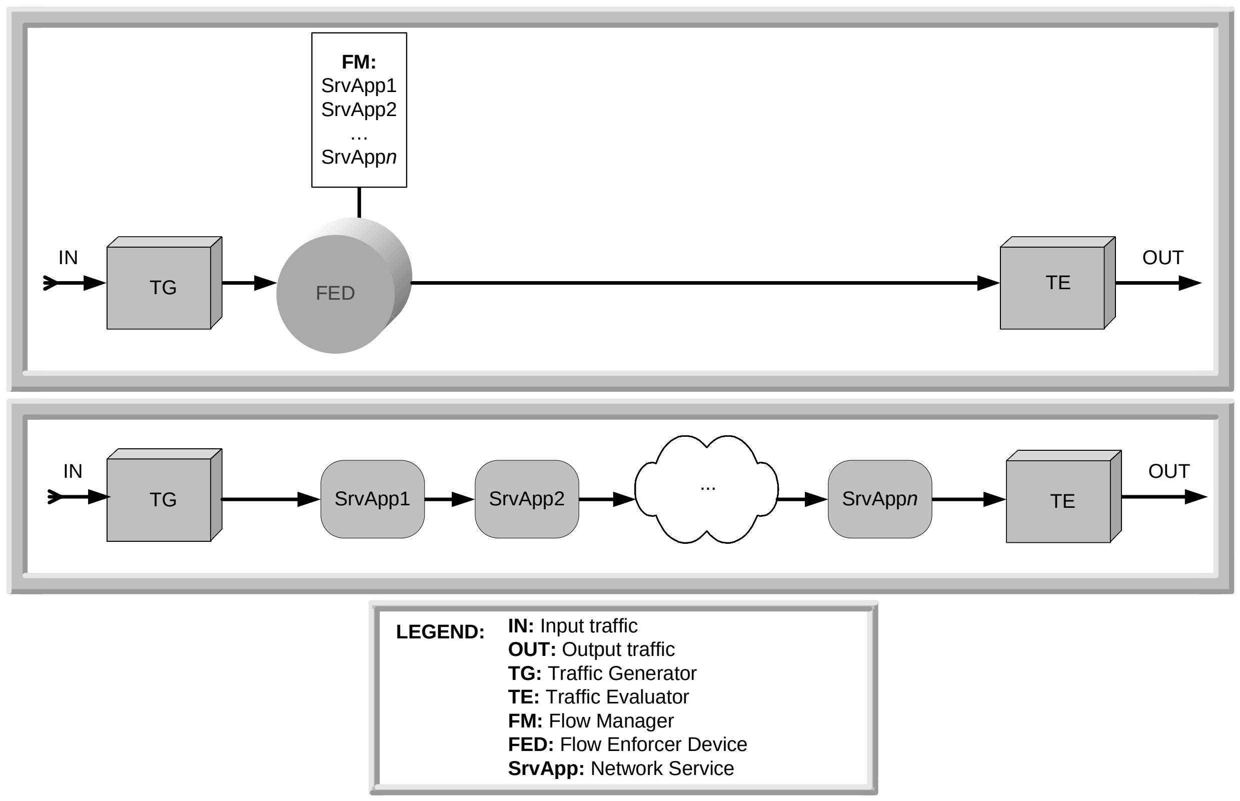

Figure 7: Performance of Neon vs. a rack of boxes.

3.0 Component Specifications

3.1 Traffic Generator (TG)

3.1.1 Existing Traffic Generators

Table 1 below describes existing traffic generators, their main features, protocols they support, and for which platforms they are targeted. This list is by no means exhaustive, but should cover most of the better known / more reputable ones.

Name	**Features – description**	**Protocols**	**OS**
Bprobe / Cprobe	bprobe provides an estimate of the un-congested bandwidth of a path. cprobe gives an estimate of the current congestion along a path. [2]		IRIX
ttcp	ttcp is a tool for measuring TCP and UDP throughput using a client / server model. The measurement can be configured with a couple of parameters such as number of packets, packet size, etc. A couple of variants exist. Newer variants like nttcp have more features like inetd support, checksums, multicast packets etc. [3]	UDP, TCP	Linux, Win 95/98, Win NT/2000, FreeBSD, Solaris, SunOS, AIX, HPUX, IRIX
NetPerf	Netperf is a benchmark that can be used to measure the performance of many different types of networking. It provides tests for both unidirectional throughput, and end-to-end latency. The environments currently measurable by netperf include: TCP and UDP via BSD Sockets, DLPI, Unix Domain Sockets, Fore ATM API and HP HiPPI Link Level Access. [4]	TCP, UDP	Linux, FreeBSD, Solaris
netpipe	NetPIPE is a protocol independent performance tool that encapsulates the best of ttcp and netperf and visually represents the network performance under a variety of conditions. By taking the end-to-end application view of a network, NetPIPE clearly shows the overhead associated with different protocol layers. [5]		Linux, FreeBSD, Solaris
DBS	DBS (Distributed Benchmark System) is a TCP performance measurement tool which is aiming to give performance index with multi-point configuration and also in order to measure changes of throughput. It measure the performance of entire TCP functions in various operational environments. [6]	UDP, TCP	Linux, FreeBSD, Solaris, Irix, HP-UX

TReno	Bandwidth Performance, UDP with limited TTL, ICMP error response or ECHO reply. TReno (Traceroute RENO) is a network testing tool designed to test network performance under load similar to that of TCP, the most commonly used Transport Protocol in the Internet today. Treno uses the same technique as traceroute to probe the network. By sending out UDP packets with low TTL, hosts and routers along the path to the final destination will send back ICMP TTL Exceeded messages which have similar characteristics to TCP ACK packets. [7]	UDP, TCP, ICMP	
TG	One of the public domain tools from SRI. Can generate constant, uniform, exponential on/off UDP or TCP traffic etc. [8]	UDP, TCP	Linux, FreeBSD, Solaris SunOS
NetSpec	NetSpec is a tool designed to provide sophisticated support for experiments testing the function and performance of networks. It can be used for TCP, UDP, WWW, FTP, MPEG, VBR, and CBR Traffic. NetSpec uses a scripting language that allows the user to define multiple traffic flows from/to multiple computers. Netspec can emulate a couple of different traffic types, has inetd support, and allows to build measurement daemons. [9]	TCP, UDP, WWW, FTP, MPEG, VBR, CBR	Linux, FreeBSD, Solaris, IRIX
Packet Shell	Packet Shell is a traffic generator for IP, IPv6, ICMP, ICMPv6, TCP, Ethernet, and TLI. This software is an extensible Tcl/Tk based software toolset for protocol development and testing. The Packet Shell creates Tcl commands that allow you to create, modify, send, and receive packets on networks. The operations available for each protocol are supplied by a dynamic linked library called a protocol library. These libraries are silently linked in from a special directory when the Packet Shell begins execution. [10]	IP, IPv6, ICMP, ICMPv6, TCP, Ethernet, TLI	SunOS 5.X or later
Rude / Crude	Traffic Generation and Measurement of UDP. RUDE stands for Real-time UDP Data Emitter and CRUDE for Collector for RUDE. RUDE is a small and flexible program that generates traffic to the network, which can be received and logged on the other side of the network with the CRUDE. Currently these programs can generate and measure only UDP traffic. The operation and configuration might look similar to the other available traffic generator tool called MGEN but these programs do not share any code. Actually	UDP	Linux, Solaris, FreeBSD

	these tools were designed and coded because of the accuracy limitations in the MGEN program. MGEN operates with system timers and e.g. in the Linux kernel on PC-platforms the timer resolution is only 10ms. That is pretty poor, so this implementation implemented its own timers with much higher resolution. [11]		
MGEN	Traffic Generation and Measurement of unicast/multicast UDP, support for ISI's RSVPd. MGEN provides programs for sourcing/sinking real-time multicast/unicast UDP/IP traffic flows with optional support for operation with ISI's "rsvpd". It now also includes support for scripted generation of packet flows with the IP TOS field set. The MGEN tools transmit and receive (and log) time-stamped, sequence numbered packets. Post-test analyses of the log files can be performed to assess network or network component ability to support the given traffic load in terms of packet loss, delay, delay jitter, etc. MGEN has been used to evaluate the capability of networks and devices to properly provide IP Multicast and RSVP support. [12]	UDP, RSVP, Multicast	Linux, FreeBSD, NetBSD, Solaris, SGI, DEC
UDPgen	UDPgen is a tool for generating UDP traffic. It aims on maximizing the packet throughput especially for Gigabit Ethernet. To maximize the throughput the traffic generator runs completely in the Linux kernel. This allows to send at much higher rates than with a userspace program. The toolset also includes a tool which counts UDP packets at the receiver and calculates the packet inter-arrival times. This tool also runs in kernel space to minimize CPU time needed and therefore be able to count all packets. [13]	UDP	Linux 2.4
Iperf	Iperf measures TCP bandwidth, datagram delay, jitter, loss. While tools to measure network performance, such as ttcp, exist, most are very old and have confusing options. Iperf was developed as a modern alternative for measuring TCP and UDP bandwidth performance. Iperf is a tool to measure maximum TCP bandwidth, allowing the tuning of various parameters and UDP characteristics. [14]	TCP, UDP	Linux, SGI IRIX, HP-UX, Solaris, AIX

Table 1: Existing Traffic Generators overview

3.1.2 Statistics

The TG should have all the following statistics.

Performance

- Duration of Test: measured in seconds and accurate to the microsecond granularity
- Packet Rate: The packets per second rate that the TG is generating traffic at.
- Number of Packets Transmitted: The total number of generated packets transmitted on the network.
- Total Number of Bytes Transmitted
- Throughput: Mbit/s data rates including header information; Mbit/s data rates of just the payload.
- Bandwidth: Capacity of the network.
- Round Trip Time (RTT): in milliseconds and accurate to the microsecond granularity
- Jitter: variation in the time between packets arriving in microseconds
- Smallest / Average / Largest Packet Size

Functionality

- Protocol statistics: percentage of time that each protocol (Ethernet, IP, TCP, etc…) was sent.
- MAC address statistics: dump the table of all MAC addresses that were seen and the frequency (in %) that was observed
- IP address statistics: dump the table of all IP addresses that were seen and the frequency (in %) that was observed
- Flow / session length: length of time in seconds with a microsecond granularity with respect to how long the flow / session was; the flow / session should fully configurable with respect to what fields make up the flow / session
- Layer statistics: percentage of packets that had at least one field examined at a particular layer (Ethernet, IP, TCP, etc…)
- Collect statistics on fields like “flow label” field in IPv6.

3.1.3 Design and Implementation Details

The TG is being designed with extensibility in mind to have the ability to add any new protocol, and control all field values in any of the protocols being used. In order to have full accessibility to the entire suite of protocol field values (Ethernet, IP, IPv6, TCP, UDP, etc…), we decided to implement the traffic generator from scratch and implement each of the protocols headers and field values. This sounds like a lot of work, but it really is better than what it sounds like. We use the header definitions of all these protocols (that have already been defined), and have a function for each protocol that goes through each field and initializes the corresponding fields with an appropriate values, and then we have a packet assembly function that takes all the various headers (structures) and copies them in a buffer in the order that they will appear when they are send over the wire. The raw device (ex. eth0) is open for writing, and the buffer (ex. Ethernet, IP, TCP, payload) is send over the raw device. This model is both extensible (new protocols can be added relatively easy), and is fully configurable (all fields in all layers are accessible). We would like to make a configuration file from which the TG could take all of its traffic specification, including the protocols, and the default/randomized field values.

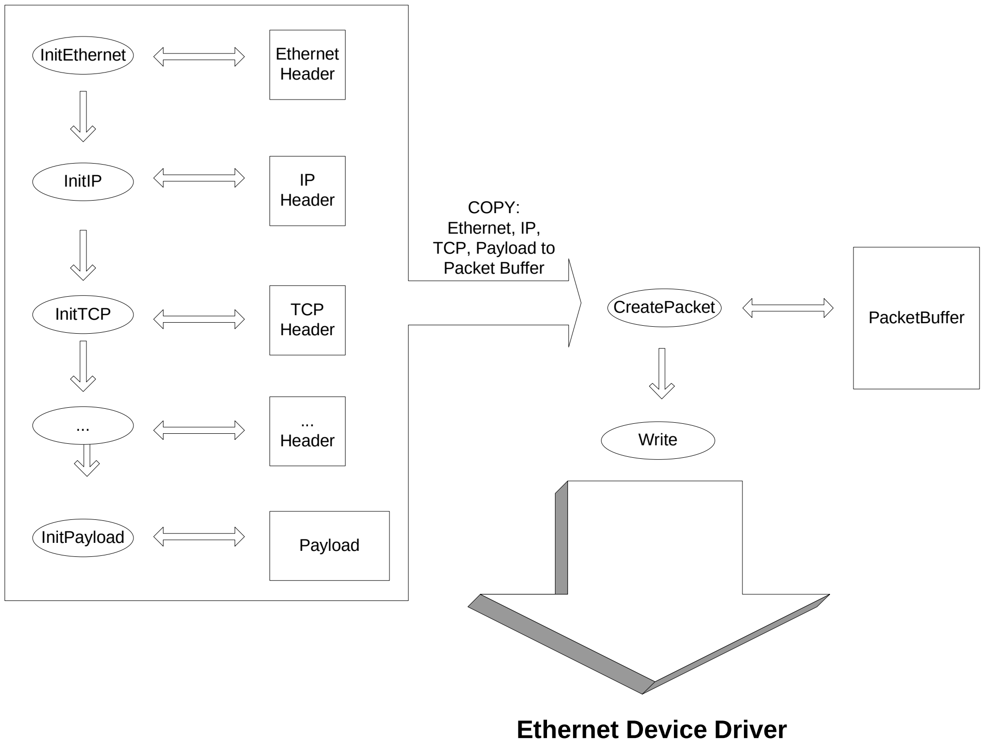

Figure 8: Traffic Generator Overview; the ovals represent function calls, while the rectangles represent memory buffer space, and the arrows represent flow of functionality and data.

Since the main purpose of the TG will be to test functionality of Neon, it must support testing service policies for various services. Each service application must have a different set of traffic policies due to the wide variety of applications and traffic patterns that will most likely exist in a real world scenario. Below are some examples on a few service applications that might run on the FED in the Neon architecture.

- Fire Wall
- Load Balancer
- Virus Scanner
- NAT
- Router

Firewalls usually look at the IPv4/IPv6 and TCP/UDP layer to make their policy decision. The allow / drop decision will most likely be made up of fields such as the IP address, port numbers, and the protocol that is running. This means that we must be able to generate a series of TCP and UDP packets as well as some arbitrary protocol that the firewall will not support. As for the IP addresses, we should be able to specify addresses from both pool of valid addresses and invalid ones that will cause the packets to be dropped. Similarly, we should be able to generate port numbers from both pool of port numbers. Based on the generated IP addresses, port numbers and protocols used, we should be able to figure out how many packets should be allowed through the firewall and how many should be dropped. At the very least, these numbers should correspond with the statistics collected at the TE and tTE.

Load balancers are very similar to firewalls in the sense that several layers must be examined and a decision must be taken to where to send the packet. Instead of operating just at the network (layer 3) and transport layer (layer 4), load balancers often operate at the application layer (layer 7) as well. In terms of the TG support for load balancers, first of all the TG must have the appropriate support for the layer 7 protocol that must be examined, such as HTTP, etc… Usually, load balancers are used to give access to a particular service (eg. Web server, ftp server, etc…) that are replicated over several physical machines. That means that the access to these various servers should be split evenly over the incoming requests. The testing of this kind of application is actually much easier to test than the firewall. Some traffic is generated that is intended for the particular service application and then the amount of traffic and workload experienced at the servers is analyzed; since a load balancer attempts to evenly spread the workload across various servers, the workload across the servers should be evenly distributed.

A virus scanner is a bit more complex to integrate into the TG because we are not only dealing with header information, but also with the payload information. A virus scanner should be able to detect any virus (or part of a virus) within the payload of the packet, and drop it if it suspects a particular packet to be infected. In order to make the FED work at its fullest, a batch of viruses should be randomly distributed over the traffic that will be generated so that they appear at random positions within the payload. This variety of viruses and the random positions of the virus will test both the performance and functionality of the virus scanner. The TE could watch out for any virus infected packet, and if any get through, then we know that the FED did not stop all virus infected packets. At that point, TAG information that is inserted at the tFED becomes very important to determine the reason why the FED failed to catch the infected packet.

In order to create a comprehensive list of features the TG must have to test the functionality of all these service applications, one must understand in depth how these applications work. A very detailed report that would discuss these features is another task all of its own, and is left for future work.

The state of the traffic generator is still very experimental, and most likely will not be ready by the time I leave. Since the TG was primarily the effort of another project (Mobile IPv6), the protocols it will first handle will probably be Ethernet, IPv6 and Mobility support, but the framework for the TG will be in place, and new protocols should be easily added.

3.2 Traffic Evaluator (TE)

Once the TG is complete, it should be relatively easy to build the TE in the same manner reading the raw device. This will allow to have access to the entire suite of protocol headers, and have the ability to keep very accurate statistics regarding the frequency of certain protocols, source / destination addresses, port numbers, etc… These kinds of statistics are definitely important for determining whether a particular service application, such as a firewall, performed its job of blocking certain traffic and letting other traffic through. The TE implementation will be very similar to that of Snoop, or EtherReal, in which the received buffer has to be parsed for information. This might be a time consuming process if all protocols in existence are to be implemented, but to implement a small suite of critical protocols, it should not be too difficult.

The TE should be able to capture all of the following statistics so it can generate an automated report at the end.

Performance

- Duration of Test: measured in seconds and accurate to the microsecond granularity
- Packet Rate: The packets per second rate that the TE received traffic at.

- Number of Packets Received: The total number of packets received on the network.
- Total Number of Bytes Received
- Throughput: Mbit/s data rates including header information; Mbit/s data rates of just the payload.
- Bandwidth: Capacity of the network.
- Round Trip Time (RTT): in milliseconds and accurate to the microsecond granularity
- Jitter: variation in the time between packets arriving in microseconds
- Smallest / Average / Largest Packet Size

Functionality

- Protocol statistics: percentage of time that each protocol (Ethernet, IP, TCP, etc…) was observed.
- MAC address statistics: dump the table of all MAC addresses that were seen and the frequency (in %) that was observed
- IP address statistics: dump the table of all IP addresses that were seen and the frequency (in %) that was observed
- Statistics on the Flow Label Field / TOS field
- Flow / session length: length of time in seconds with a microsecond granularity with respect to how long the flow / session was; the flow / session should fully configurable with respect to what fields make up the flow / session.

3.3 Traffic Evaluator with TAG Functionality (tTE)

This traffic evaluator would be relatively simple to implement (additionally on top of the TE), since each TAG packet that would arrive would carry a payload with useful information regarding the decisions that were made on that packet, including the entire suite of header information that packet had. The database of TAG values and their descriptive meanings would have be the identical between the tFED and the tTE, and therefore the same statistics that were captured by the TE could be augmented with a list of actions that were performed within the tFED, and therefore the lifetime of the packet can more easily be traced and a detailed list of actions can be kept.

In addition to the TE statistics it captures, the tTE can now also infer from the TAG information even more statistics; the tTE should be able to capture the following statistics in addition to those of the TE.

- Number of dropped packets due to errors
- Number of dropped packets due to policy, and policy code statistics
- Number of packets forwarded along with policy code statistics
- Statistics on the Flow Label Field / TOS field
- Statistics on which service application (firewall, load balancer, etc…) acted upon packets

3.4 Flow Enforcer Device with TAG Functionality (tFED)

This is probably going to be the most difficult task, especially since we are dealing with a relatively proprietary device (a network processor, or NP). First of all, network processors in general have an ingress port, a processing unit that makes some decision based on the header information, and then send packets to its egress port. Therefore, fundamentally, the NP (FED) will always have some decision to make. The modification that the tFED must have to add the

TAG functionality is every time a decision happens internally in the NP, the appropriate TAG information could be recorded in memory (most likely passed as an argument to the next logical processing step), and at the end of the packet's journey through the decision process of the NP, the NP could copy the packet, discard the data payload, and add a new payload that contains all the TAG information it has gathered all along. Note that this approach has its limitations since each NP is different, and some NPs have very well defined parts (ex. classification stage, packet modification stage, etc…), and sometimes very little information (and sometime none) can be passed from one stage to another. It is therefore imperative that TAG information be kept small (on the order of a byte or two), and that the particular NP actually has the support to transfer information between various stages. Note that most likely, the TAG packet must be generated in one of the last stages (most likely the packet modification stage), and that a new packet will not be able to be generated in one of the early stages, such as the classification stage. Another limitation might be the ability to inject new packets into the data stream, or even duplicate packets; some NP were never designed with that in mind, but they were rather designed to accept packets on ingress, perform some classification, and send it to the appropriate egress port. Once again, this is probably the hardest part of the entire project, and especially to implement the TAG functionality efficiently enough and with a good enough granularity that we could extract meaningful information from it at the tTE.

The tFED should keep track of the following information for each packet as it traverses the tFED.

- Dropped packets due to errors; duplicate packet, replace the data payload with the TAG information that the packet had errors.
- Dropped packets due to policy; duplicate packet, replace the data payload with the TAG information that the packet was dropped due to a specific policy.
- Every time a look-up occurs in some table / tree, and a decision is made based on the result of that look-up, the action should be recorded, and later placed in the TAG information of that packet; at the very end, after a packet has been through the entire tFED, and right before a packet is sent to the egress port, the packet should be duplicated, the payload should be stripped, and the saved TAG information should be stored in the data payload.
- Every time a different service application (firewall, load balancer, etc…) acts upon the each packet, the appropriate information should be stored, and later included with the TAG information.

4.0 Conclusion

The entire NEon project was a great experience! I learned a whole lot about flow management, resource management, and the motivation of the Neon architecture. I hope that my experiences and this report is enough to get someone else that much closer to being able to implement my ideas.

The state of the work that I did is the following:

- Traffic generator (TG)
    - Completed
        - Definition of the TG and the various features it should have
        - A basic framework (source code) that supports Ethernet, IPv6, and Mobile IPv6
    - Future Work
        - Adapt TG from Mobile IPv6 project to support more protocols
            - IPv4, TCP, UDP, etc…
- Traffic Evaluator (TE)
    - Completed
        - Definition of the TE and the various features it should have
    - Future Work
        - Implement the TE; can leverage from work done on the traffic generator
- Traffic Evaluator with TAG functionality (tTE)
    - Completed
        - Definition of the tTE and the various features it should have
    - Future Work
        - Implement the tTE; Once TAG functionality is defined, and the TE is complete, implementation of tTE should be straight forward and relatively simple
- Flow Enforcement Device with TAG functionality (tFED)
    - Completed
        - Definition of the tFED and the various features it should have
    - Future Work
        - Modify existing FED to support TAG functionality
        - Define specific traffic patterns that will be required to test each specific application service

5.0 Bibliography

[1] SMI: NTO. Project Neon.

[2] "Bprobe / Cprobe", http://cs-people.bu.edu/carter/tools/Tools.html.

[3] "ttcp", ftp://ftp.arl.mil/pub/ttcp/.

[4] "NetPerf", http://www.netperf.org.

[5] "netpipe", http://www.scl.ameslab.gov/netpipe/.

[6] "DBS", http://www.ai3.net/products/dbs/.

[7] "TReno", http://www.psc.edu/networking/treno_info.html.

[8] "TG", http://www.caip.rutgers.edu/~arni/linux/tg1.html.

[9] "NetSpec", http://www.ittc.ku.edu/netspec/.

[10] "Packet Shell", http://playground.sun.com/psh/.

[11] "Rude / Crude", http://www.atm.tut.fi/rude.

[12] "MGEN", http://manimac.itd.nrl.navy.mil/MGEN/.

[13] "UDPgen", http://www.fokus.fhg.de/usr/sebastian.zander/private/udpgen.

[14] "Iperf", http://dast.nlanr.net/Projects/Iperf/.