Using Linux Traffic Control on Virtual Circuits

J. Zurawski – Internet2 – zurawski@internet2.edu
February 25nd 2013

1. Abstract

Research and Education (R&E) networks have experimented with the concept of Virtual Circuits (VC) for a number of years. This *connection-oriented* service emulates a physical point-to-point connection, using the underlying technology of packet-switched networks. ESnet's On-Demand Secure Circuits and Advance Reservation System (OSCARS) [OSCARS] is an implementation of VC control that facilitates network management within, and between, domains. The underlying mechanisms for management of these logical connections rely on Quality of Service (QoS) procedures to management utilization, latency, and throughput.

Recent research [OSCARSQoS] has highlighted some harmful characteristics of the QoS-based management of VC resources; notably it has shown that reliable protocols, such as Transmission Control Protocol (TCP), can react poorly to some of the actions implemented to guarantee bandwidth reservation and flow segmentation. A similar pattern emerges, where a traffic flow can be slowed down due to ***out of order*** behavior introduced, when multiple ingress queues are used on a particular interface. Similarly, lack of available buffering to support packet bursts, common in TCP, can drop traffic silently in some cases.

Methods to control the ***burst*** behavior of an application or protocol, as well as limit the overall bandwidth sent, can be introduced at the host. Tools such as **`tc`**, found in Linux based operations systems, offer a robust and predictable way to introduce QoS behavior at the source of traffic. When implemented correctly, these operating system mechanisms will aide in the end-to-end flow of traffic across VC resources.

2. Sample Network Environment

Testing of the recommendations found in this document were performed on hosts affiliated with the NSF funded DYNES project [DYNES]. DYNES facilitated the purchase of hardware resources for campuses and regional R&E networks, and encouraged the use of the Internet2 ION service [ION], a VC implementation based on OSCARS.

The testing machines were Dell R510 servers, with 10Gbps capable network cards. The CentOS 5 operating system, a Red Hat Enterprise Linux Variant, was used as the base control software along with OSCARS to handle the VC control plane, and DRAGON to facilitate communicate with the local switching resources.

The servers were connected to Dell 8024F Switches using 10G optical connections. These switches are designed nominally for data center usage, but are capable of performing in wide area environments. All switches feature a configurable (up to 128 KB) ingress buffering behavior, and were tested to ensure the capability to deliver multiple sustained 10Gbps network flows. Campus and regional network switches were uplinked to Internet2 ION directly, or through the use of statically configured (e.g. specific pre-configured VLANs) network devices.

All system and network devices were tuned, using recommendations from ESnet's Fasterdata resources [Fasterdata], when applicable. Host tuning recommendations regarding memory configuration and network interface settings were applied, along with recommendations regarding network setup to remove unnecessary firewalls and spurious networking devices from the path.

3. OSCARS Operation

OSCARS relies on the concept of a federation to facilitate the creation of end-to-end paths. Using certificate technology, it is possible to ***peer*** OSCARS instances that have control over a specific domain's resources. For example, ESnet has chosen to peer with Internet2 ION. Thus members of either site could invoke VC paths between the domains, as long as resources and permissions allowed this. To extend the example, if a school such as Texas Tech University peered with Internet2 ION, they could make a VC, if resources were available and permitted for use, into ESnet by exercising the peering relationship.

OSCARS has a notion of bandwidth guarantees, built on QoS, for hardware that is able to support it. Simplistically, OSCARS will install mechanisms on network devices that will treat traffic (up to the requested rate for the VC) in a special manner, more so than other traffic on the same interface that is not affiliated with the VC. For example, on a 10Gbps interface we could reserve 2Gbps for a VC. Those using that VC are guaranteed to have access to the full 2Gbps, even as the remaining traffic on the interface grows. This can be modeled into two basic classes of service:

- Expedited Traffic
- Best Effort Traffic

The QoS mechanisms also feature other behavior that will kick in if the requestor of the aforementioned VC tries to use more resources than were requested. In an extended example, lets assume that the application using the VC tries to push 2.5Gbps of traffic through the 2Gbps reservation. 2Gbps of the traffic will continue to be treated as ***expedited***, but the remaining 0.5Gbps will be sent into a 3rd queue: less than best effort service. This queue is treated on a lower priority than regular traffic on the interface. One can imagine that on an interface with no traffic, this separation of traffic will have minimal impact, on a heavily loaded interface; there is

the potential to disrupt the flow, particularly if delicate protocols such as TCP are used.

Experimentation [OSCARSQoS] has shown that this queuing behavior can cause disruption in traffic flows that utilize TCP. The queuing introduces ***out of order*** behavior, which causes a stall (and in some cases unnecessary ***retransmission***) of information. Investigations into better methods of management for the QoS are still being conducted, with ideas ranging from hard traffic drops (which in some cases are preferred to out of order behavior) to softer QoS management policies implemented by the switches and hosts under consideration.

Traffic management implemented within the network can be subjective. Without knowledge of what a particular flow is doing, networks can only judge based on information related to source, destination, and prior behavior. Alternatives for traffic management abound, with a likely answer being implemented by an application with knowledge of underlying network capabilities, or via the sending and receiving hosts directly.

We will investigate the latter option – allow the hosts to regulate the flow of traffic exiting the network interfaces. There is system software, some implemented within the operating system kernel, that can enforce a form of QoS using similar mechanisms to that of network devices. When put in place these rules help to smooth out fragile protocols, such as TCP, such that they do not exhibit behaviors that will burst beyond available buffers, or exceed bandwidth reservations on VC infrastructure.

4. Using TC

`tc` is used to show and manipulate traffic control settings within the Linux operating system. Essentially, we create queues on the host interface (similar to network device QoS) to categorize traffic, and handle it in different manners. E.g. we can make traffic adhere to a certain bandwidth, latency, or even emulate errors and drops on a faulty network. With queuing we determine the way in which data is ***sent***; it is important to realize that we can only shape data that we transmit.

A possible use cases for **`tc`**, and how we intend to use it, is to smooth bandwidth to a specific bottleneck range. For example, lets say we have a 10Gbps capable network card on the host, and a path that consists of 10Gbps links for 90% of the end to end path. There is a single 1Gbps link as the bottleneck, thus it benefits us to smooth the traffic to this bottleneck link instead of allowing TCP to fluctuate between over and under sending (with a resulting average that is less than the bottleneck). Smoothing at the host allows us to regulate our traffic more effectively than TCP can, especially as the distance between the end points grows.

`tc` consists of the following functionalities [TCMan]:

- ***shaping*** - When traffic is shaped, it's rate of transmission is under control. Shaping may be more than lowering the available bandwidth - it is also used to smooth out bursts in traffic for better network behavior. Shaping occurs on egress.
- ***scheduling*** - By scheduling the transmission of packets it is possible to improve interactivity for traffic that needs it while still guaranteeing bandwidth to bulk transfers. Reordering is also called prioritizing, and happens only on egress.
- ***policing*** - Where shaping deals with transmission of traffic, policing pertains to traffic arriving. Policing thus occurs on ingress.
- ***dropping*** - Traffic exceeding a set bandwidth may also be dropped forthwith, both on ingress and on egress.

Processing of traffic is controlled by three kinds of objects: `qdiscs`, `classes` and `filters`. We will use all three in our examples.

A `qdisc` is short for *queuing discipline*. Whenever the kernel needs to send a packet to an interface, it is en-queued to the `qdisc` configured for that interface. Immediately afterwards, the kernel tries to get as many packets as possible from the `qdisc`, for giving them to the network adaptor driver. Some `qdisc` can contain `classes`, which contain further `qdisc`. When the kernel tries to de-queue a packet from such a classful `qdisc` it can come from any of the `classes`. A `qdisc` may for example prioritize certain kinds of traffic by trying to de-queue from certain classes before others. A `filter` is used by a classful `qdisc` to determine in which class a packet will be en-queued. All filters attached to the `class` are called, until one of them returns with a verdict.

We propose some very specific uses of **`tc`** for traffic management on VCs: establish a simple Hierarchical Token Bucket (HTB) queue and traffic rates that are slightly below the requested circuit capacity. The following sections will outline our steps.

4.1. Creating a Circuit

OSCARS can be used to create circuits between participating sites. In general you need to have an account on one end of the circuit (source or destination), and the ability to *reference* dynamic resources through an address (e.g. URN string).

Figure 1 shows what a typical OSCARS login screen looks like. It is beyond the scope of this document to describe installation and use of OSCARS, so we assume this portion is done for a given domain:

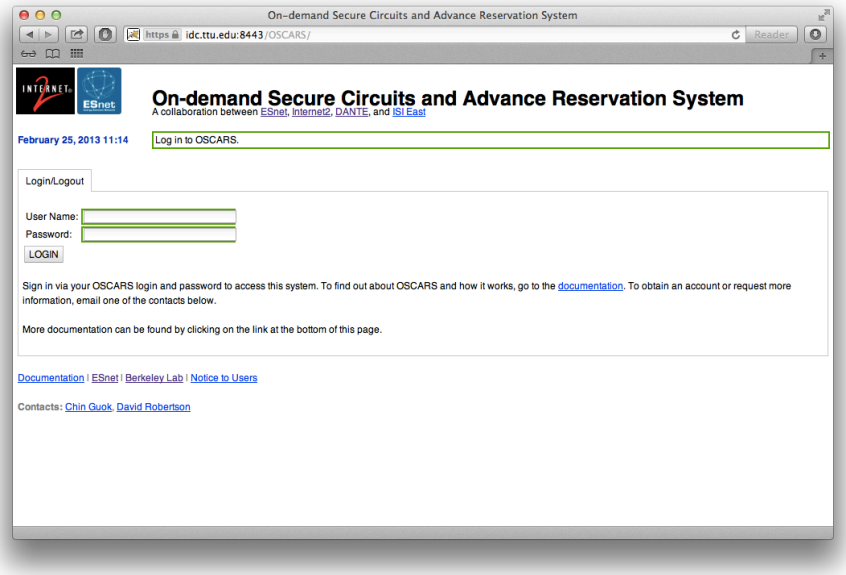

Figure 1 - Initial Loggin Screen - OSCARS Server at Texas Tech University

After logging in, Figure 2 shows the available options. We are interested in creating a circuit to another participating location:

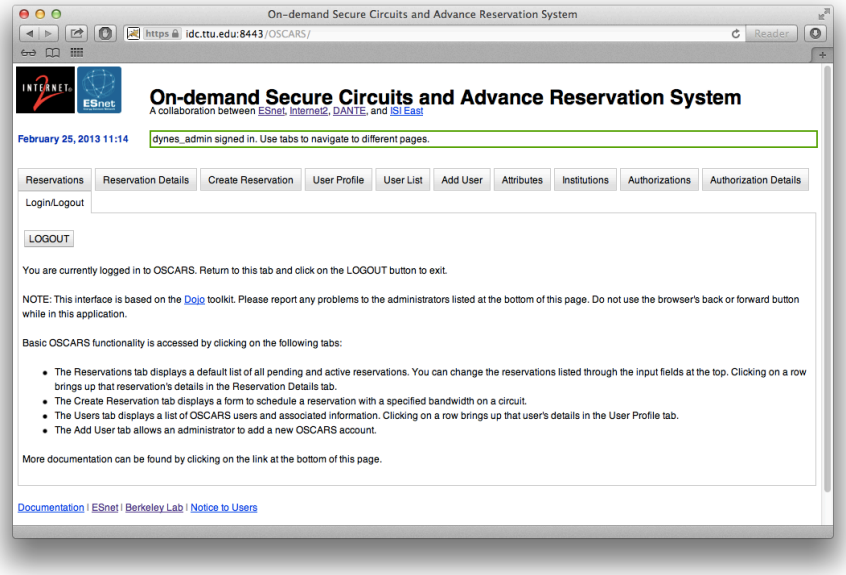

Figure 2 - OSCARS Options

Figure 3 shows details needed to create a circuit:

- URNs of endpoints
- Time of Circuit

- Bandwidth Requirements
- Path or VLAN specifics (if needed)

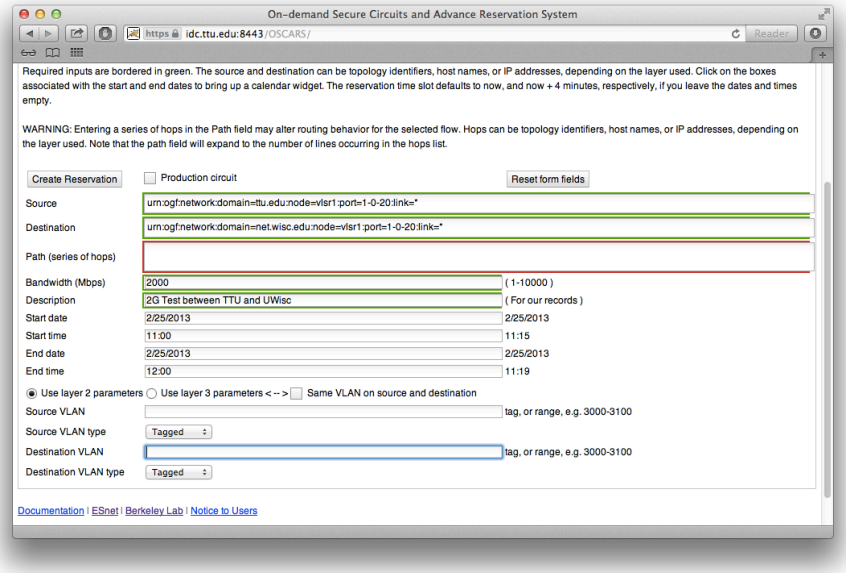

Figure 3 - Creating a Circuit with Known URNs (Texas Tech University and University of Wisconsin)

The circuit is shown active in Figure 4, and we see what VLANs we have been assigned.

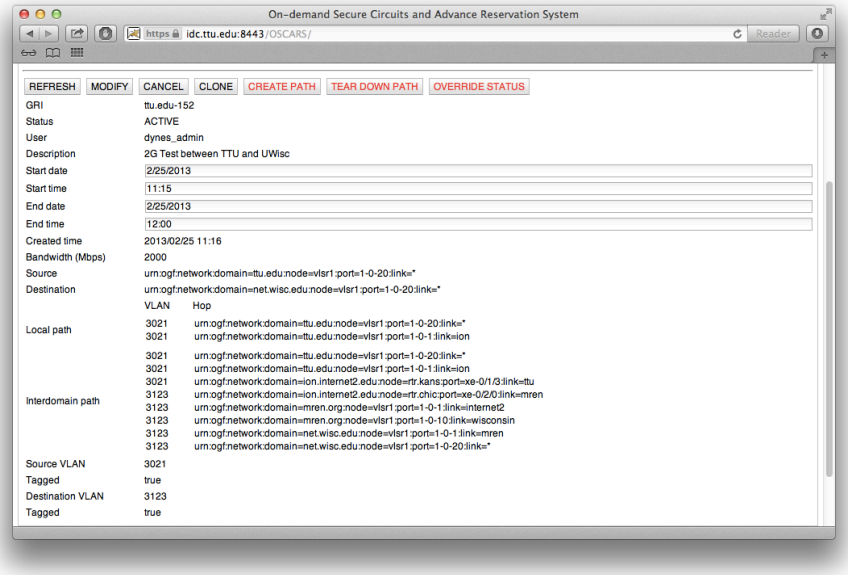

Figure 4 - Complete Circuit, with VLANs and Path Listed

It is now necessary to configure the local hosts that are available on this network (e.g. the switch where the circuit is known to terminate) to communicate on the designated VLAN.

4.2. Adding Machines to Circuit via VLANs and Private Address Space

Each machine that will communicate on the newly established circuit will need to be added to the VLANs for the different sites. Note we may not have the same VLANs on each end, as OSCARS performs a process called *VLAN Translation*. We know that the VLANs we care about are:

- Texas Tech University = 3021
- University of Wisconsin = 3123

We also must carve out private IP addresses for these hosts to communicate with, anything defined in RFC 1918 will do. For the sake of this example, we will use the **`10.10.200.0/24`** subnet on either end. We first look at the Texas Tech University end. We need to associate one of the server's interfaces with the VLAN. In this case we know that the server is connected to the switch that is now connected to the circuit via Ethernet interface 0 (e.g. **`eth0`**). We do the following:

```
sudo /sbin/vconfig add eth0 3021
sudo /sbin/ifconfig eth0.3021 10.10.200.10/24 up
sudo /sbin/ifconfig eth0.3021 txqueuelen 10000
```

This adds the VLAN to the interface, assigns an IP address (**`10.10.200.10`**) and network information, and creates a queue for data transmission. Establishing the **`txqueuelen`** is very important; this is required for **`tc`** control. Failure to do this will result in **`tc`** being unable to affect the traffic on the VLAN. Then we turn our attention to the University of Wisconsin end, which is very similar:

```
sudo /sbin/vconfig add eth0 3123
sudo /sbin/ifconfig eth0.3123 10.10.200.20/24 up
sudo /sbin/ifconfig eth0.3123 txqueuelen 10000
```

The hosts are now established on the circuit. We can do a simple test to check connectivity:

```
ping -c 5 10.10.200.10
PING 10.10.200.10 (10.10.200.10) 56(84) bytes of data.
64 bytes from 10.10.200.10: icmp_seq=1 ttl=64 time=36.3 ms
64 bytes from 10.10.200.10: icmp_seq=2 ttl=64 time=36.3 ms
64 bytes from 10.10.200.10: icmp_seq=3 ttl=64 time=36.2 ms
64 bytes from 10.10.200.10: icmp_seq=4 ttl=64 time=36.3 ms
64 bytes from 10.10.200.10: icmp_seq=5 ttl=64 time=36.2 ms

--- 10.10.200.10 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms
rtt min/avg/max/mdev = 36.296/36.313/36.352/0.209 ms
```

We see a latency that matches the distance, and we are able to see that the subnet definitions are correct.

4.3. Establishing TC Rules

We will establish some basic **tc** rules on the sender end. For the sake of this test we will send from University of Wisconsin to Texas Tech University only, thus the rules will be applied to the Wisconsin end. We will review the statements in order. This first rule clears out any existing **tc** rules that may have been defined before. It's always a good idea to do this.

```
sudo /usr/sbin/tc qdisc del dev eth0.3123 root
```

This second rile creates an HTB for the VLAN on our interface. This must be done before we go any further in defining classes:

```
sudo /usr/sbin/tc qdisc add dev eth0.3123 handle 1: root htb
```

Next, we add in a class to our route queue. We make a base transfer rate of 2Gbps (256 MBps – note that **tc** requires this):

```
sudo /usr/sbin/tc class add dev eth0.3123 parent 1: classid 1:1 htb
rate 256mbps
```

Lastly, we create a filter that restricts our **tc** queue and class to a specific source IP address (on the Wisconsin end):

```
sudo /usr/sbin/tc filter add dev eth0.3123 parent 1: protocol ip prio
16 u32 match ip src 10.10.200.20/32 flowid 1:1
```

We will make changes to this during experimentation, particularly to the rate.

5. Testing

We first make a baseline test between Texas Tech University and the University of Wisconsin. This test was performed using a 1Gbps circuit, on the aforementioned VLANs. Note that to do this, we must remove any settings that are in place. Do be safe; lets do this on both ends, first at Wisconsin:

```
sudo /usr/sbin/tc qdisc del dev eth0.3123 root
```

And at Texas Tech:

```
sudo /usr/sbin/tc qdisc del dev eth0.3021 root
```

We initiate a test using the nuttcp [nuttcp] tool, from the Wisconsin end, sending to Texas Tech. The Texas Tech end is running a server:

```
[dynes@fdt-texastech ~]$ nuttcp -S -p 5679 -P 5678 --nofork
```

The Wisconsin end is running the client:

```
[dynes@fdt-wisc ~]$ nuttcp -T 30 -i 1 -p 5679 -P 5678 10.10.200.10
    1.6875 MB /   1.00 sec =   14.1543 Mbps     9 retrans
    1.6875 MB /   1.00 sec =   14.1558 Mbps     3 retrans
    1.3750 MB /   1.00 sec =   11.5345 Mbps     0 retrans
    1.9375 MB /   1.00 sec =   16.2529 Mbps     0 retrans
    3.1250 MB /   1.00 sec =   26.2147 Mbps     0 retrans
    1.4375 MB /   1.00 sec =   12.0585 Mbps    21 retrans
    2.7500 MB /   1.00 sec =   23.0691 Mbps     0 retrans
    3.2500 MB /   1.00 sec =   27.2629 Mbps     8 retrans
    1.4375 MB /   1.00 sec =   12.0585 Mbps     0 retrans
    2.7500 MB /   1.00 sec =   23.0688 Mbps     0 retrans
    2.6250 MB /   1.00 sec =   22.0198 Mbps    37 retrans
    0.5625 MB /   1.00 sec =    4.7185 Mbps     0 retrans
    2.4375 MB /   1.00 sec =   20.4474 Mbps     0 retrans
    3.0000 MB /   1.00 sec =   25.1658 Mbps    20 retrans
    0.5000 MB /   1.00 sec =    4.1943 Mbps     0 retrans
    2.6250 MB /   1.00 sec =   22.0197 Mbps     0 retrans
    3.3750 MB /   1.00 sec =   28.3118 Mbps    13 retrans
    1.8125 MB /   1.00 sec =   15.2046 Mbps     0 retrans
    3.3125 MB /   1.00 sec =   27.7867 Mbps     0 retrans
    3.8125 MB /   1.00 sec =   31.9824 Mbps     0 retrans
    5.7500 MB /   1.00 sec =   48.2347 Mbps     0 retrans
    3.4375 MB /   1.00 sec =   28.8354 Mbps    14 retrans
    3.3125 MB /   1.00 sec =   27.7872 Mbps     0 retrans
    4.5625 MB /   1.00 sec =   38.2728 Mbps    23 retrans
    1.5625 MB /   1.00 sec =   13.1071 Mbps     0 retrans
    3.2500 MB /   1.00 sec =   27.2630 Mbps     0 retrans
    4.3125 MB /   1.00 sec =   36.1759 Mbps    23 retrans
    1.8750 MB /   1.00 sec =   15.7287 Mbps     0 retrans
    3.3125 MB /   1.00 sec =   27.7880 Mbps     0 retrans
    4.1875 MB /   1.00 sec =   35.1252 Mbps     0 retrans

   83.7159 MB /  30.45 sec =   23.0658 Mbps 0 %TX 0 %RX 171 retrans 36.69 msRTT
```

The performance does not look good considering that we had requested 1 Gbps speeds on the circuit. We see retransmissions (e.g. TCP is struggling for some reason), and generally low throughput. Using the tools **`tcpdump`** [TCPDump] and **`tcptrace`** [TCPTrace] we can capture what is going on with this particular transfer. The following graph shows a graphical representation of this transfer:

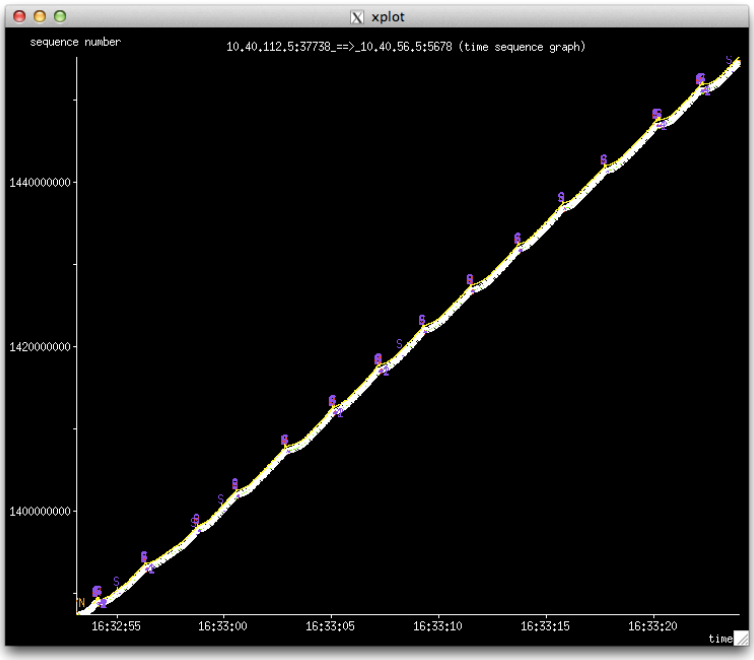

Figure 5 - TCPDump of 1Gbps Circuit Testing Using nuttcp

This graph plots sequence number over time, e.g. a linear graph indicates smooth sailing from a transmission perspective. The *blips* of activity are related to the trouble we saw with nuttcp; each indicates a stall in the process. Zooming in further on some of the purple points in Figure 6, we see the following:

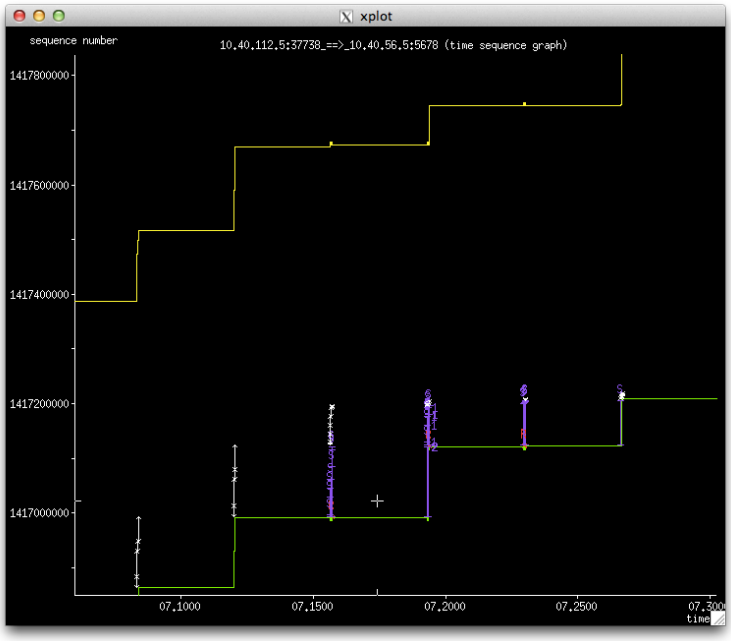

Figure 6 - Zoomed View of Data Retransmission

This indicates stalls or drops of data packets, which delays TCP. The sending end compensates by trying to recover with sending duplicate data into the network, which stalls things further. We end up in a continuous cycle, which reduces our throughput. We have surmised that this is being caused by a two factors:

- Low buffering (128K) on some of the switches in the path
- QoS provided by the switches in the path

With this in mind, we institute a **tc** rule that will limit our throughput to 900Mbps (under our reservation of 1 Gbps). Note that this is done on the Wisconsin side:

```
sudo /usr/sbin/tc qdisc del dev eth0.3123 root
sudo /usr/sbin/tc qdisc add dev eth0.3123 handle 1: root htb
sudo /usr/sbin/tc class add dev eth0.3123 parent 1: classid 1:1 htb
rate 112.5mbps
sudo /usr/sbin/tc filter add dev eth0.3123 parent 1: protocol ip prio
16 u32 match ip src 10.10.200.20/32 flowid 1:1
```

After adding this rule, we run the client again, and see the following performance:

```
[dynes@fdt-wisc ~]$ nuttcp -T 30 -i 1 -p 5679 -P 5678 10.10.200.10
    2.1875 MB /   1.00 sec =   18.3486 Mbps     0 retrans
    8.3125 MB /   1.00 sec =   69.7281 Mbps     1 retrans
   28.3125 MB /   1.00 sec =  237.5170 Mbps     0 retrans
   99.1875 MB /   1.00 sec =  832.0559 Mbps     0 retrans
  108.5000 MB /   1.00 sec =  910.1831 Mbps     0 retrans
  108.4375 MB /   1.00 sec =  909.6078 Mbps     0 retrans
  108.4375 MB /   1.00 sec =  909.6706 Mbps     0 retrans
  108.4375 MB /   1.00 sec =  909.6215 Mbps     0 retrans
  108.3125 MB /   1.00 sec =  908.5747 Mbps     0 retrans
  108.3750 MB /   1.00 sec =  909.1354 Mbps     0 retrans
  108.3750 MB /   1.00 sec =  909.1363 Mbps     0 retrans
  108.2500 MB /   1.00 sec =  908.0605 Mbps     0 retrans
  108.3750 MB /   1.00 sec =  909.1218 Mbps     0 retrans
  108.3125 MB /   1.00 sec =  908.5911 Mbps     0 retrans
  108.3125 MB /   1.00 sec =  908.5902 Mbps     0 retrans
  108.4375 MB /   1.00 sec =  909.6133 Mbps     0 retrans
  108.5000 MB /   1.00 sec =  910.1731 Mbps     0 retrans
  108.4375 MB /   1.00 sec =  909.6533 Mbps     0 retrans
  108.3750 MB /   1.00 sec =  909.1199 Mbps     0 retrans
  108.4375 MB /   1.00 sec =  909.6388 Mbps     0 retrans
  108.3750 MB /   1.00 sec =  909.1154 Mbps     0 retrans
  108.4375 MB /   1.00 sec =  909.6406 Mbps     0 retrans
  108.3750 MB /   1.00 sec =  909.1154 Mbps     0 retrans
  108.3125 MB /   1.00 sec =  908.5911 Mbps     0 retrans
  108.4375 MB /   1.00 sec =  909.6388 Mbps     0 retrans
  108.5000 MB /   1.00 sec =  910.1640 Mbps     0 retrans
  108.3125 MB /   1.00 sec =  908.5593 Mbps     0 retrans
  108.5000 MB /   1.00 sec =  910.1967 Mbps     0 retrans
  108.4375 MB /   1.00 sec =  909.6397 Mbps     0 retrans
  108.3125 MB /   1.00 sec =  908.5911 Mbps     0 retrans

 2965.6678 MB /  30.12 sec =  825.9052 Mbps 3 %TX 8 %RX 1 retrans 36.73 msRTT
```

This can be viewed graphically using the same **tcpdump** and **tcptrace** analysis:

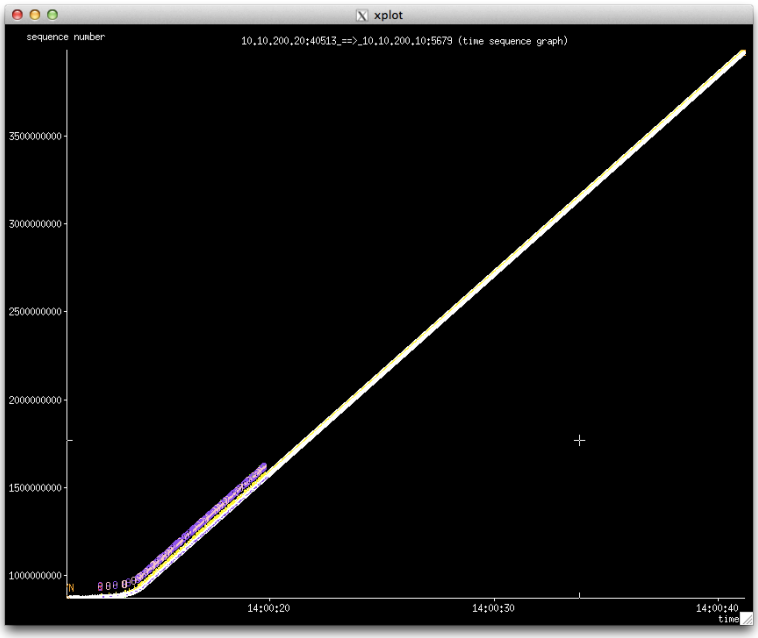

Figure 7 - nuttcp Performance After Use of tc

In Figure 7 we see a transfer that has much retransmission activity to start, before finally reaching a steady state of performance for the remainder of the transfer. We achieve an average that is very close to the original 900Mbps-shaping request, and over time we would see it come closer to this threshold. We next zoom in on the start of the transfer in Figure 8.

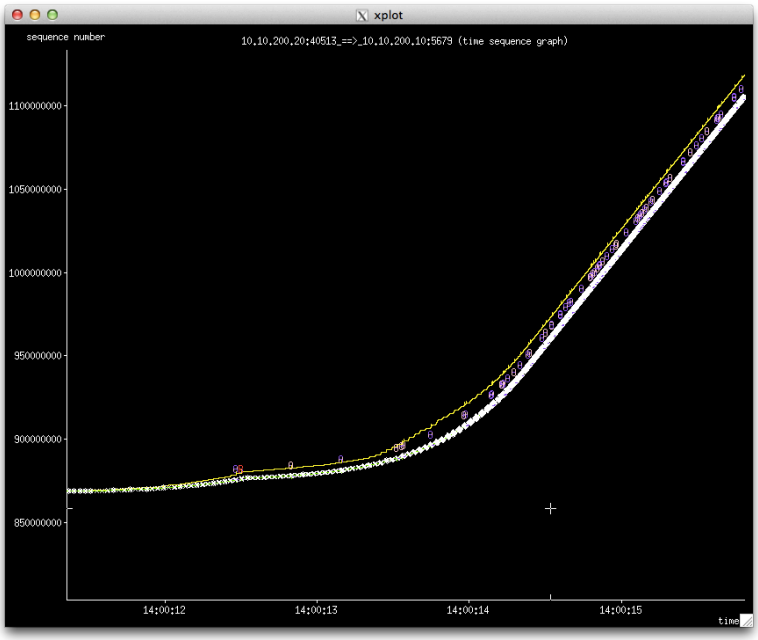

Figure 8 - Zoom of Start to nuttcp Transfer

This transfer demonstrates a loss event during the TCP *slow start* phase, and then minor shaping activities to smooth the flow and control the sending window for an additional amount of time. We next will try **`tc`** rule that will limit our throughput to 1Gbps, equal to our reservation. Note that this is done on the Wisconsin side:

```
sudo /usr/sbin/tc qdisc del dev eth0.3123 root
sudo /usr/sbin/tc qdisc add dev eth0.3123 handle 1: root htb
sudo /usr/sbin/tc class add dev eth0.3123 parent 1: classid 1:1 htb
rate 128mbps
sudo /usr/sbin/tc filter add dev eth0.3123 parent 1: protocol ip prio
16 u32 match ip src 10.10.200.20/32 flowid 1:1
```

We observe the following performance characteristics:

```
[dynes@fdt-wisc ~]$ nuttcp -T 30 -i 1 -p 5679 -P 5678 10.10.200.10
    2.8750 MB /   1.00 sec =   24.1153 Mbps     0 retrans
    6.8125 MB /   1.00 sec =   57.1492 Mbps     3 retrans
   15.1250 MB /   1.00 sec =  126.8811 Mbps     8 retrans
   17.1875 MB /   1.00 sec =  144.1652 Mbps     0 retrans
   19.0625 MB /   1.00 sec =  159.9147 Mbps     0 retrans
   22.5000 MB /   1.00 sec =  188.7422 Mbps     0 retrans
   29.4375 MB /   1.00 sec =  246.9406 Mbps     0 retrans
   31.9375 MB /   1.00 sec =  267.9114 Mbps     6 retrans
   15.8125 MB /   1.00 sec =  132.6459 Mbps     0 retrans
   21.5625 MB /   1.00 sec =  180.8795 Mbps     0 retrans
   24.6875 MB /   1.00 sec =  207.0925 Mbps     0 retrans
   30.2500 MB /   1.00 sec =  253.7435 Mbps     0 retrans
   39.7500 MB /   1.00 sec =  333.4735 Mbps     0 retrans
   44.0000 MB /   1.00 sec =  369.1102 Mbps     7 retrans
   21.6875 MB /   1.00 sec =  181.9228 Mbps     0 retrans
   29.3750 MB /   1.00 sec =  246.4070 Mbps     0 retrans
   32.1250 MB /   1.00 sec =  269.4830 Mbps     0 retrans
   37.8750 MB /   1.00 sec =  317.7239 Mbps     0 retrans
   46.9375 MB /   1.00 sec =  393.7466 Mbps     0 retrans
   57.3750 MB /   1.00 sec =  481.2993 Mbps     0 retrans
   31.8750 MB /   1.00 sec =  267.3751 Mbps     4 retrans
   33.9375 MB /   1.00 sec =  284.6907 Mbps     0 retrans
   36.8750 MB /   1.00 sec =  309.3503 Mbps     0 retrans
   41.1250 MB /   1.00 sec =  344.9805 Mbps     0 retrans
   48.9375 MB /   1.00 sec =  410.5187 Mbps     0 retrans
   18.3750 MB /   1.00 sec =  154.1303 Mbps     9 retrans
   27.8125 MB /   1.00 sec =  233.2900 Mbps     0 retrans
   30.0000 MB /   1.00 sec =  251.6952 Mbps     0 retrans
   35.6875 MB /   1.00 sec =  299.3684 Mbps     0 retrans
   44.0625 MB /   1.00 sec =  369.6230 Mbps     0 retrans

  906.7649 MB /  30.28 sec =  251.2284 Mbps 0 %TX 3 %RX 37 retrans 36.71 msRTT
```

We now turn to **`tcpdump`** and **`tcptrace`** again so we can see the graphical view of the output:

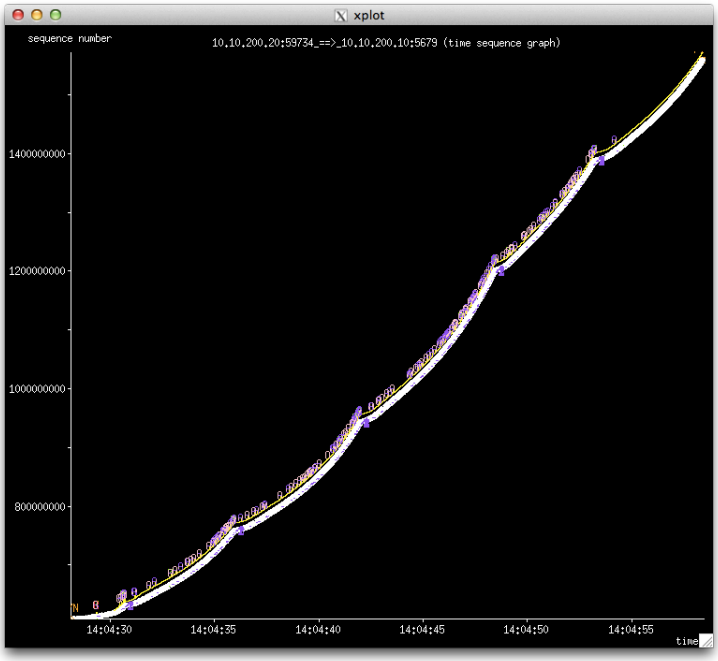

Figure 9 - Using tc with 1Gbps rate on a 1Gbps circuit

Figure 9 shows constant stalls, most likely due to the same factors we observed before related to the size of buffers and QoS implemented in OSCARS.

Additional experimentation for circuits of higher reservations has shown that this general pattern is repeated, and it is recommended that **`tc`** settings aim for 90% – 92% of the available bandwidth to reduce the chance of QoS impacting performance of TCP flows.

6. Conclusion

VC implementations offer the useful and powerful paradigm of establishing point-to-point Layer 2 connections (VLANs) between participating endpoints. These connections offer a litany of positive features, including stable latency and guaranteed bandwidth delivered via QoS mechanisms.

Existing applications that use protocols such as TCP require no adaption to use these new networking paradigms by default; however, it has been observed that performance is subjective due to the environment created using QoS. These mechanisms can force packets into an out-of-order state due to queuing, or may have limited buffering to handle bursty behavior; both situations will reduce overall TCP throughput.

Using simple host based solutions, such as the Linux **`tc`** program, we can implement mechanisms that smooth and control TCP traffic on an end to end basis. These

controls make it possible to use a high percentage (90-92% in practice) of bandwidth reservations, without needing to modify existing applications.

It is recommended that users of VC infrastructure follow these steps when utilizing resources.

7. References

[OSCARS] – On-Demand Secure Circuits and Advance Reservation System (OSCARS), http://www.es.net/services/virtual-circuits-oscars/

[OSCARSQoS] – Z. Yan, M. Veeraraghavan, C. Tracy, C. Guok, On how to provision Quality of Service (QoS) for large dataset transfers. January 2013.

[DYNES] - Dynamic Network System (DYNES), http://www.internet2.edu/ion/dynes.html

[ION] – Internet2 ION (Interoperable On-demand Network) Service, http://www.internet2.edu/ion/

[Fasterdata] – ESnet Fasterdata, http://fasterdata.es.net

[TCman] - tc(8) - Linux man page, http://linux.die.net/man/8/tc

[nuttcp] - Phil Dykstra's nuttcp quick start guide, http://www.wcisd.hpc.mil/nuttcp/Nuttcp-HOWTO.html

[TCPDump] – TCPDump, http://www.tcpdump.org

[TCPTrace] – TCPTrace, http://www.tcptrace.org