Network Citizenship

Author: A.J.Gillette
Date: DEc 05, 2012
Revision: 1.1

Table of Contents

Introduction

This document defines what it means to be a good network Citizen. It describes mechanisms for implementing and measuring network citizenship with a special focus on Linux based systems. The document will describe some of the built in capabilities that come with Linux distributions as well as what you can expect from Internet service providers.

Network Citizenship

The Internet and many corporate networks are “Best Effort” Ethernet networks that do not implement protocols to segregate network traffic. Mechanisms like MPLS and VLAN's can be used to manually control how a network segment is shared, but these schemes cannot be applied to a network like the Internet that is not owned by one organization. On public networks it is up to the connected devices to share the available bandwidth.

Network Citizenship is a measure of how well a device on a best effort network shares with other devices on the same network segment. Networks are typically designed to have layers where a local LAN is connected via routers to a firewall or gateway router that connects to the Internet. There can be various levels of Network Address Translation (NAT) before packets actually reach the Internet. Internet Service Providers (ISP's) then provide a level of service that may involve limiting the flow of data using devices like shapers and policers.

Many modern day operating systems have a built in ability to control the flow of data onto the network. This makes it possible for the OS to react better to shapers and policers improving throughput on public networks.. We will look at some tools and make some suggestions specific to the type of traffic that would be associated with visual communication sessions.

Internet Service Provides (Shaping and Policing)

Internet service providers are in business to provide bandwidth to consumers. Most ISP's have several levels of service targeted and everything from residential consumers to large enterprises. For business consumers Service Level Agreements or SLA's define parameters of the service like the amount of bandwidth that will be delivered. In some countries, ISP's are allowed to use devices like policers to manage the bandwidth into the network from each end point. This allows the ISP to engineer their network in a more deterministic fashion making it possible to guarantee bandwidth as defined in Business SLA's.

For the end user this means that bursts that exceed the purchased bandwidth rate will be discarded. This may seem rather brutal but in actual fact protocols like TCP will detect

the packet loss and automatically retransmit lost data at a slower rate. Some protocols like UDP however, have no mechanism for retransmission so the data is lost forever.

The way the ISP calculates a burst is not always public knowledge but we have determined through experiment that many ISP's simply look at the packet arrival rate.

This means that any small burst of data can result in packet loss. In order to prevent this type of loss, Linux systems can use “tc” to insure that packets are transmitted at regular intervals, thus staying below the radar so to speak. The only place where this approach can be a problem is where the Linux kernel is configured with a timer tick that makes servicing the outbound packet queue in real time impossible.

The latest Linux kernels for all desktop machines now come with a standard 1ms timer tick, but machines like laptops that are trying to conserve battery life have lower resolution ticks that make real time shaping more of a challenge.

With a 1ms timer tick the maximum data rate would be 1/1ms or 1000 packets/second. Most network links however are specified in bits/second so if we assume 1500 bytes/packet(mtu size) and 8 bits/byte we get a maximum data rate of 12Mb/s. Anything over that rate will cause the Linux kernel to process more than one packet per tick. When that happens we cannot control the time between the packets and we run the risk of loosing a packet to a network policer as a result.

Controlling packet flow seems fairly straight forward considering the calculations above, but in actual fact the algorithms used by ISP's are for the most part closely guarded secrets. They may allow short bursts of two or three packets because routers in the network all have buffers and it is impossible to control the resulting jitter. Using “tc” to space out how packets are placed onto the network will decrease packet loss thus improving network throughput far beyond the theoretical 12Mb/s limit.

It is important to note that all data flows between visual communication systems do not have to be treated as equals. Audio for example is running at a much lower data rate to start with and introducing a slight delay to a collaboration page update would hardly be noticed. By using a more complex queuing discipline, it is possible to further tune the system.

Being a good network Citizen.

In order to be a “good Network Citizen”, systems need to implement mechanisms that allow the network to be shared in an intelligent fashion. There are network centric solutions that guarantee fairness but they usually come at a higher cost and add complexity to the network implementation.

As we previously mentioned, good network Citizens take it upon themselves to share the network by monitoring network performance and adjusting their individual use .

This is not a new concept and protocols exist that were specifically designed to enable good behaviour. Media transmission in the Internet is typically done with UDP and RTP. This combination does not provide for a retransmission mechanism but does provide for detecting network performance by using the companion RTCP protocol. RTCP allows receivers to feed back information to transmitters about lost packets, jitter and other network performance parameters. Transmitters can then adjust things like video resolution to reduce demands on the network.

It is possible to layer on additional protocols to provide a way to reconstruct damaged packets without having to do retransmissions. This is usually called Forward Error Correction (FEC) and does increase the data rate of streams carrying this extra CRC data. If a network is already at capacity, adding FEC will make things worse so it is important that good network citizens only enable FEC when the packet loss is not due to limited bandwidth. Ideally, FEC should be dynamically switched in and out as required.

Linux TC (Traffic Control)

Every version of Linux ships with a process called “tc” that is short for “Traffic Control”. This process works with the Linux kernel to determine how data packets will be placed onto the local network. It contains queuing systems and other mechanisms that control how packets are received and transmitted on the Network Interface Cards.

By default, traffic control consists of a single queue which collects packets and dequeues them as fast as the underlying hardware can accept them. This is a First In First Out mechanism (FIFO) that has been given the name “pfifo_fast”.
All network links are serial in the way that data is received and transmitted so queues are required to manage data flow.

Queues can also be used to treat types of network traffic differently. Non real time network traffic can be slowed down to allow delay sensitive network traffic to have more bandwidth. This gives the people using the network the responsiveness that they demand while still allowing the gaps to be filled in with server traffic.

There are many different types of queues available and any number of queues can be assembled to create solutions for common network challenges like the ones listed below.

- Limit the total bandwidth to a fixed amount.
- Limit the amount of bandwidth available to a specific network user.
- Maximize network throughput for a specific network protocol (Like TCP)
- Reserve bandwidth for a specific application (Like VoIP)
- Adjust latency for a specific class of network traffic.
- Ensure that a specific type of network traffic is dropped (A policer).

When traffic control has been properly configured networks become more predictable. Predictability however comes with a price and that price for the most part is complexity. Traffic control also is dependant on the other users and routers on the network. Before we proceed we should define a few components of a traffic control system.

Definitions:

Queues
A queue is a location or buffer where packets wait to be transmitted. A queue does not make an promise with respect to when a packet will be serviced and is usually configured to be a fixed size..

Flows
A flow is a set of packets that represent a communication between two network connected devices. Flows can have a protocol associated with them like TCP or UDP.

Tokens and Buckets
In order to control the flow of data onto a network one commonly used mechanism is to generate tokens at a desired rate and to then place packets onto the network only when tokens are available. When the queue is empty the tokens are placed in a bucket for future use. The size of the burst of future traffic is then controlled by the bucket size. Controlling the flow of traffic using tokens and a bucket is considered shaping.

A traffic shaper
Is a device that contains buffers and some number of queues that adjust the flow of traffic onto or between network segments.

A Policer
Is a shaper without buffers. Flow is controlled by throwing packets away.

A Classifier
A classifier is a mechanism that allows a filter to inspect a packet and key on different characteristics. A common classifier used by "tc" is the "u32 classifier" which allows a filter to identify things like a UDP packet type.

A Filter
A traffic control filter is a mechanism that glues together other system components. A filter must contain a classifier phrase and may contain policer phrase.

A "qdisc"
"qdisc" is short for "Queuing Discipline". When ever an OS/Kernel needs to send a packet to an interface it sends it to a "qdisc". "qdisc" is a generic term to describe the many different types of queues.

A Handle

Every class and classful qdisc requires a unique identifier within "tc". This unique identifier is known as a handle and has two parts. The major number and a minor number can be assigned arbitrarily by the user in accordance with the following rules.

major/first number

This parameter is completely free of meaning to the kernel, however all objects in the traffic control structure with the same parent must share a major handle number. Conventional numbering schemes start at 1 for objects attached directly to the root qdisc.

minor/second number

This parameter unambiguously identifies the object as a qdisc if minor is 0. Any other value identifies the object as a class. All classes sharing a parent must have unique minor numbers.

NOTE: The special handle ffff:0 is reserved for the ingress qdisc.

The handle is used as the target in classid and flowid phrases of tc filter statements.

Queue Types

There are a number of different types of Queues that come standard with "tc" but we are only going to consider the most useful ones for visual communication. Queues can be classful or classless. The primary difference is that queues that support classes can contain multiple qdiscs. If a queue is classful, the kernel can dequeue a packet from any of the internal qdisc's. Examples of classless queues would be pfifo_fast, red(Random Early Detection), sfq(Stochastic Fairness Queuing) and tbf(Token Bucket Filter). Examples of classful queues would include cbq(Class Based Queuing), htb(Hierarchy Tocken Bucket) and prio.

pfifo_fast

This is the most common type of queue and is the default queue for "tc". It is a simple First In First Out buffer where no packet gets special treatment. This queue typically has three "bands" where "Type Of Service" bits can be used to place traffic in the bands. Band 0 packets are written out first and only when band 0 is empty will band 1 packets be written. The "pfifo_fast" queue is considered to be classless so you cannot add other qdiscs to it.

tbq (Token Bucket Filter)
This is a simple queue that uses tokens and buckets to regulate the flow of packets onto the network. Short bursts are allowed and controlled by the bucket size.

Htb (Hierarchical Token Bucket)
This is a classful queue that allows for an arbitrary number of token buckets arranged in a hierarchy. Users can define the size of the buckets, the rate of token generation and whether the queues are nested. The variable “ceil” sets the maximum bandwidth that the class is allowed to consume. (The master bucket)

TC Examples

"tc" can be used at a linux command line to modify, create or destroy queues in real time. The granularity with respect to packet transmission or token generation will be determined in part by the kernel timer tick value. Most linux kernels (post 2009) have a 1ms timer tick. Note that power sensitive devices like laptops use a larger tick value to improve battery life. The tick length may increase or decrease burstiness because the kernel may be forced to transmit multiple packets per tick. The reason why this is important is because ISP's can look at the packet arrival rate to determine whether to drop a packet.

NOTE: "tc" can be used to create a policer but only on ingress from the network.

"tc" syntax

To see all of the command options for "tc" you can access the built in " man page" by typing "man tc". There are always more options than we need to get started so the options below are the most commonly used.

NOTE: The command "tc" has been aliased on all TeleCollabortaion systems and must be unaliased before trying to test the examples below. To unalias the command type "unalias tc".

Here is an extract from the “tc” man page.

tc(8) - Linux man page

Name

tc - show / manipulate traffic control settings

Synopsis

tc qdisc [add | change | replace | link] dev DEV [parent qdisc-id | root] [handle qdisc-id] qdisc [qdisc specific parameters]

tc class [add | change | replace] dev DEV parent qdisc-id [classid class-id] qdisc [qdisc specific parameters]

tc filter [add | change | replace] dev DEV [parent qdisc-id | root] protocol protocol prio priority filtertype [filtertype specific parameters] flowid flow-id

tc [-s | -d] qdisc show [dev DEV]

tc [-s | -d] class show dev DEV

tc filter show dev DEV

“tc” Examples

Perhaps the best way to understand the options and usage for “tc” is to analyze some examples.

Example 1:
Here's a command to show the current “tc” configuration:

Command:

tc -s qdisc show dev eth0

Result:

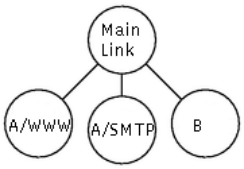
qdisc pfifo_fast 0: root bands 3 priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
Sent 43093029 bytes 440074 pkt (dropped 0, overlimits 0 requeues 0)
rate 0bit 0pps backlog 0b 0p requeues 0

Explanation:

- The qdisc being used is a pfifo_fast type.
- The root queue has three bands
- The priomap defines which band each of 16 TOS types goes.
- The last part of the message contains stats on bytes and packets sent.

Example 2:
(Extracted from the HTB Linux queuing discipline manual - user guide by Martin Devera)

In this example Martin considers an application where there are three traffic streams sharing one internet connection (Main Link). The example that follows will create a tree structure where three queues will be connected to the root queue that feeds the network.




```
tc qdisc add dev eth0 root handle 1: htb default 12
```

This command attaches queue discipline HTB to eth0 and gives it the "handle" 1:. This is just a name or identifier with which to refer to it below. The default 12 means that any traffic that is not otherwise classified will be assigned to class 1:12.

```
tc class add dev eth0 parent 1: classid 1:1 htb rate 100kbps ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:10 htb rate 30kbps ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:11 htb rate 10kbps ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:12 htb rate 60kbps ceil 100kbps
```

The first line creates a "root" class, 1:1 under the qdisc 1:. The definition of a root class is one with the htb qdisc as its parent. A root class, like other classes under an htb qdisc allows its children to borrow from each other, but one root class cannot borrow from another. We could have created the other three classes directly under the htb qdisc, but then the excess bandwidth from one would not be available to the others. In this case we do want to allow borrowing, so we have to create an extra class to serve as the root and put the classes that will carry the real data under that. These are defined by the next three lines. The ceil parameter is described below.

We also have to describe which packets belong in which class. This is really not related to the HTB qdisc. The commands will look something like this:

```
tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 match ip src 1.2.3.4 match ip dport 80 0xffff flowid 1:10
tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 match ip src 1.2.3.4 flowid 1:11
```

(We identify A by its IP address which we imagine here to be 1.2.3.4.)

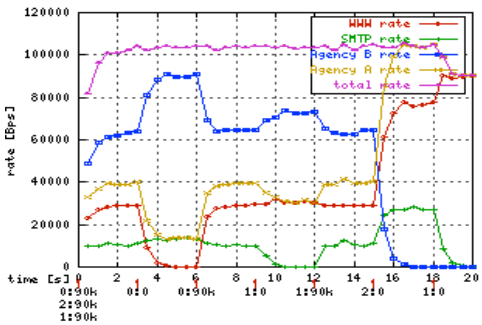
Note: The U32 classifier has an undocumented design bug which causes duplicate entries to be listed by "tc filter show" when you use U32 classifiers with different prio values.

You may notice that we didn't create a filter for the 1:12 class. It might be more clear to do so, but this illustrates the use of the default. Any packet not classified by the two rules above (any packet not from source address 1.2.3.4) will be put in class 1:12.

Now we can optionally attach queuing disciplines to the leaf classes. If none is specified the default is pfifo.

```
tc qdisc add dev eth0 parent 1:10 handle 20: pfifo limit 5
tc qdisc add dev eth0 parent 1:11 handle 30: pfifo limit 5
tc qdisc add dev eth0 parent 1:12 handle 40: sfq perturb 10
```

That's all the commands we need. Let's see what happens if we send packets of each class at 90kbps and then stop sending packets of one class at a time. Along the bottom of the graph are annotations like "0:90k". The horizontal position at the center of the label (in this case near the 9, also marked with a red "1") indicates the time at which the rate of some traffic class changes. Before the colon is an identifier for the class (0 for class 1:10, 1 for class 1:11, 2 for class 1:12) and after the colon is the new rate starting at the time where the annotation appears. For example, the rate of class 0 is changed to 90k at time 0, 0 (= 0k) at time 3, and back to 90k at time 6.



Initially all classes
generate 90kb. Since this is higher than any of the rates specified, each class is limited to its specified rate. At time 3 when we stop sending class 0 packets, the rate allocated to class 0 is reallocated to the other two classes in proportion to their allocations, 1 part class 1 to 6 parts class 2. (The increase in class 1 is hard to see because it's only 4 kbps.) Similarly at time 9 when class 1 traffic stops its bandwidth is reallocated to the other two (and the increase in class 0 is similarly hard to see.) At time 15 it's easier to see that the allocation to class 2 is divided 3 parts for class 0 to 1 part for class 1. At time 18 both class 1 and class 2 stop so class 0 gets all 90 kbps it requests.

Linux Netem

"netem" is a Linux queuing discipline written by Stephen Hemminger at OSDL and is based on NISTnet. Netem is used to emulate the properties of wide area networks. The current version of netem emulates variable delay, loss, duplication and re-ordering. If you run a current 2.6 distribution, (Fedora, OpenSuse, Gentoo, Debian, Mandriva), netem is already enabled in the kernel and a current version of iproute2 is also included. (The netem kernel component is enabled under: Networking/Networking Options/QoS and/or fair queuing/Network emulator

Netem is controlled by the command line tool 'tc' which is part of the iproute2 package of tools. The tc command uses shared libraries and data files in the /usr/lib/tc directory.

Netem is a useful tool for simulating networks and network impairments, but it is important to note that the netem queuing disipline would not be used on an endpoint.

Netem Examples

An egress network policer (by Jonathan Crone)
In this example Jonathan is defining interfaces and IP addresses for three linux lab machines that are connected on separate ports of a linux computer that has been dedicated for network emulation. The netem computer has four Ethernet interfaces labeled eth0-eth3.

The systems connected to each of the ports are Magor systems labeled "HDSolo, HDDuo and HDTrio". The remaining port not referenced in the script below is "eth0" which is connected to the main company network and is used for remote access. There are no systems of interest connected to that interface.

Note that this script is setting up the network emulator to simulate a policer. That means that packets will be lost if the systems try to exceed the specified bandwidth limits. It is also important to note that this script allows TCP traffic to pass unaffected. It was also discovered through experiment that in order to accurately simulate a real network there must always be some minimal amount of delay. When the three queues are created, 1:11 gets 10ms of delay and 1:12 and 1:13 get 50ms of delay. Creating queues with no delay can cause problems with algorithms that rely on a finite amount of delay to converge.

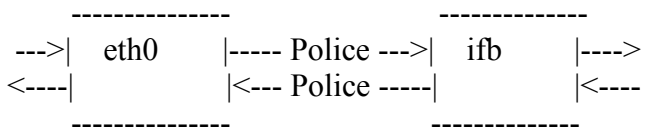
It was also discovered through experiment that the buffer size specified effects latency and that if the buffer is set to large it can effect the performance of end points because they get the false impression that there is more network bandwidth available.

```
#! /bin/bash
# Jonathan's ProtoType Lab Script for Netem
# invoke the HTB device so we can add classes and management to the bandwidth management
# This is the bandwidth management version:   we will have a packet loss version as well.
# This is the 'atlantic'  script:  eth1 to eth3

if [ `whoami` != "root" ]; then
        echo "Running a sudo to get the script working: $*"
        sudo /home/videowall/InfocomPolice.sh $*
        exit 0
fi

#Define the system interfaces and the connected system IP addresses
HDSolo=eth1
HDDuo=eth2
HDTrio=eth3

HDSoloIP=192.168.127.10
HDDuoIP=192.168.127.74
HDTrioIP=192.168.127.138

ratelimit=$1
buffersize=200k

# configure the impairments for the HDSolo.
# Create a root queuing discipline of type htb.
# Then set it's rate to 1Gb/s (wirespeed) so the queuing discipline has no effect.

/sbin/tc qdisc add dev $HDSolo root handle 1: htb
/sbin/tc class add dev $HDSolo parent 1: classid 1:1 htb rate 1000Mbps ceil 1000Mbps

# Now we create classes that can be attached to specific netem
# devices. Collab will get class 1:11, everything will go to class 1:12 or class 1:13
# Note that each class is set to wire speed so there is no effect on the flows.

/sbin/tc class add dev $HDSolo parent 1: classid 1:11 htb rate 1000Mbps
/sbin/tc class add dev $HDSolo parent 1: classid 1:12 htb rate 1000Mbps
/sbin/tc class add dev $HDSolo parent 1: classid 1:13 htb rate 1000Mbps

# Now we assign a queuing discipline to the classes and add a fixed delay of 50ms
# to 1:12 and 1:13

/sbin/tc qdisc add dev $HDSolo parent 1:11 handle 10: netem delay 10ms
/sbin/tc qdisc add dev $HDSolo parent 1:12 handle 20: netem delay 50ms limit 10000
/sbin/tc qdisc add dev $HDSolo parent 1:13 handle 30: netem delay 50ms limit 10000

# Now we apply a filter so that anything flowing from the HDDuo will go to flow 1:12
# and anything flowing from the HDTrio will go to flow 1.13.
# Anything using port 5900 (vnc) will go to flowid 1:11 and will be unencumbered
# Note that the rate limit is passed in as the first argument to the script.
# The buffer size is hard coded to 200K

/sbin/tc filter add dev $HDSolo protocol ip prio 1 u32 match ip protocol 6 0xff flowid 1:11
/sbin/tc filter add dev $HDSolo protocol ip prio 2 u32 match ip src $HDDuoIP police rate $ratelimit buffer $buffersize  drop  flowid
1:12
/sbin/tc filter add dev $HDSolo protocol ip prio 3 u32 match ip src $HDTrioIP police rate $ratelimit buffer $buffersize drop  flowid
1:13
```

```
# configure the impairments for the HDDuo.
# Create a root queuing discipline of type htb.
# Then set it's rate to 1Gb/s (wirespeed) so the queuing discipline has no effect.

/sbin/tc qdisc add dev $HDDuo handle 1: root htb
/sbin/tc class add dev $HDDuo parent 1: classid 1:1 htb rate 1000Mbps ceil 1000Mbps

# Now we create classes that can be attached to specific netem
# devices. Collab will get class 1:11, everything will go to class 1:12 or class 1:13
# Note that each class is set to wire speed so there is no effect on the flows.

/sbin/tc class add dev $HDDuo parent 1: classid 1:11 htb rate 1000Mbps
/sbin/tc class add dev $HDDuo parent 1: classid 1:12 htb rate 1000Mbps
/sbin/tc class add dev $HDDuo parent 1: classid 1:13 htb rate 1000Mbps

# Now we assign a queuing discipline to the classes and add a fixed delay of 50ms
# to 1:12 and 1:13

/sbin/tc qdisc add dev $HDDuo parent 1:11 handle 10: netem delay 10ms
/sbin/tc qdisc add dev $HDDuo parent 1:12 handle 20: netem delay 50ms  limit 10000
/sbin/tc qdisc add dev $HDDuo parent 1:13 handle 30: netem delay 50ms limit 10000

# Now we apply a filter so that anything flowing from the HDSolo will go to flow 1:12
# and anything flowing from the HDTrio will go to flow 1.13.
# Anything using port 5900 (vnc) will go to flowid 1:11 and will be unencumbered
# Note that the rate limit is passed in as the first argument to the script.
# The buffer size is hard coded to 200K

/sbin/tc filter add dev $HDDuo protocol ip prio 1 u32 match ip protocol 6 0xff flowid 1:11
/sbin/tc filter add dev $HDDuo protocol ip prio 2 u32 match ip src $HDSoloIP police rate $ratelimit buffer $buffersize drop  flowid
1:12
/sbin/tc filter add dev $HDDuo protocol ip prio 3 u32 match ip src $HDTrioIP police rate $ratelimit buffer $buffersize drop   flowid
1:13


# configure the impairments for the HDTrio.
# Create a root queuing discipline of type htb.
# Then set it's rate to 1Gb/s (wirespeed) so the queuing discipline has no effect.

/sbin/tc qdisc add dev $HDTrio root handle 1: htb
/sbin/tc class add dev $HDTrio parent 1: classid 1:1 htb rate 1000Mbps ceil 1000Mbps

# Now we create classes that can be attached to specific netem
# devices. Collab will get class 1:11, everything will go to class 1:12 or class 1:13
# Note that each class is set to wire speed so there is no effect on the flows.

/sbin/tc class add dev $HDTrio parent 1: classid 1:11 htb rate 1000Mbps
/sbin/tc class add dev $HDTrio parent 1: classid 1:12 htb rate 1000Mbps
/sbin/tc class add dev $HDTrio parent 1: classid 1:13 htb rate 1000Mbps

# Now we assign a queuing discipline to the classes and add a fixed delay of 50ms
# to 1:12 and 1:13

/sbin/tc qdisc add dev $HDTrio parent 1:11 handle 10: netem delay 10ms
/sbin/tc qdisc add dev $HDTrio parent 1:12 handle 20: netem delay 50ms  limit 10000
/sbin/tc qdisc add dev $HDTrio parent 1:13 handle 30: netem delay 50ms limit 10000
```

```
# Now we apply a filter so that anything flowing from the HDDuo will go to flow 1:12
# and anything flowing from the HDSolo will go to flow 1.13.
# Anything using port 5900 (vnc) will go to flowid 1:11 and will be unencumbered
# Note that the rate limit is passed in as the first argument to the script.
# The buffer size is hard coded to 200K

/sbin/tc filter add dev $HDTrio protocol ip prio 1 u32 match ip protocol 6 0xff flowid 1:11
/sbin/tc filter add dev $HDTrio protocol ip prio 2 u32 match ip src $HDDuoIP police rate $ratelimit buffer $buffersize drop  flowid
1:12
/sbin/tc filter add dev $HDTrio protocol ip prio 3 u32 match ip src $HDSoloIP police rate $ratelimit buffer $buffersize drop  flowid
1:13
```

A loss/delay network tester (by Jonathan Crone)
In this example Jonathan creates a virtual NIC in order to provide independent delay and loss on both ingress and egress network traffic. By definition "tc" only allows policing on egress so Jonathan connects his virtual NIC to the real NIC such that all packet traffic flows through both devices. Inbound traffic from eth0 is policed by eth0 in the normal way. Outbound traffic is policed by the virtual interface ifb.



Jonathan also provides a commented line in the script below where traffic is impaired for only one inbound source. This can be handy when you want to test a multi-node connection with impairments from only one participant.

Note that this script is designed to be edited each time it is used. It would be a simple modification to design in command line arguments that assigned the values without the need to edit the script. For the purpose of this example we decided to keep it simple.

```
#This lines sets the delay and loss for data arriving from the eth0 network interface
#Put 125 ms of delay with 2% loss on outbound interface.
tc qdisc add dev eth0 root netem delay 125ms loss 2%

# Create the pseudo interface ifb so that we can duplicate the packets.
modprobe ifb

# Bring up the interface
ip link set dev ifb0 up

# Create a queuing discipline for ingress packet flow.
tc qdisc add dev eth0 handle ffff: ingress

# Use TC Filtering rules to redirect ALL inbound packets through the IFB interface 0
tc filter add dev eth0 parent ffff:  protocol ip u32 match u32 0 0 flowid 1:1 action mirred egress redirect dev ifb0

# Here is an alternative:  where you only impair traffic from one inbound source, and route it to the ifb.
#  tc filter add dev eth0 parent ffff:  protocol ip u32 match ip src  192.168.217.216 flowid 1:1 action mirred egress redirect dev ifb0

# This line sets the delay and loss for the outbound packets
# assign 250ms  with 1.5 percent packet loss to the ifb: now inbound packets
# are impaired.
tc qdisc add dev ifb0 root netem delay 250ms loss 1.5%

# dump the statistics
tc -s -d qdisc show dev eth0
tc -s -d qdisc show dev ifb0

# These next few lines pause the script to allow results of the
# impairment to be collected before the script returns the network
# to normal operation.
echo "Type anything (and enter) to clear impairment and exit"
read input
echo " "

#dump the stats again as we exit
tc -s -d qdisc show dev eth0
tc -s -d qdisc show dev ifb0

# and clean up and exit
```

```
tc qdisc del dev eth0 handle ffff: ingress
tc qdisc del dev ifb0 root netem
tc qdisc del dev eth0 root netem
```

Conclusions and Recommendations

A queuing proposal for Visual Communication

A Visual Communication system is but one end point on a best effort network so we can use “tc” to be a better “Network Citizen” but we are still at the mercy of other devices on the LAN. That said, we will improve the probability that our traffic flows will traverse the network with less delay and loss.