ADSL Bandwidth Management HOWTO

Dan Singletary

<dvsing@sonicspike.net>

Revision History

Revision 1.3	2003−04−07	Revised by: ds
Added links section.		
Revision 1.2	2002−09−26	Revised by: ds
Added link to new Email Discussion List. Added small teaser to caveat section regarding new and improved QoS for Linux designed specifically for ADSL to be released soon.		
Revision 1.1	2002−08−26	Revised by: ds
A few corrections (Thanks to the many that pointed them out!). Added informational caveat to implementation section.		
Revision 1.0	2002−08−21	Revised by: ds
Better control over bandwidth, more theory, updated for 2.4 kernels		
Revision 0.1	2001−08−06	Revised by: ds
Initial publication		

This document describes how to configure a Linux router to more effectively manage outbound traffic on an ADSL modem or other device with similar bandwidth properties (cable modem, ISDN, etc). Emphasis is placed on lowering the latency for interactive traffic even when the upstream and/or downstream bandwidth is fully saturated.

Table of Contents

1. Introduction

The purpose of this document is to suggest a way to manage outbound traffic on an ADSL (or cable modem) connection to the Internet. The problem is that many ADSL lines are limited in the neighborhood of 128kbps for upstream data transfer. Aggravating this problem is the packet queue in the ADSL modem which can take 2 to 3 seconds to empty when full. Together this means that when the upstream bandwidth is fully saturated it can take up to 3 seconds for any other packets to get out to the Internet. This can cripple interactive applications such as telnet and multi−player games.

1.1. New Versions of This Document

You can always view the latest version of this document on the World Wide Web at the URL: http://www.tldp.org.

New versions of this document will also be uploaded to various Linux WWW and FTP sites, including the LDP home page at http://www.tldp.org.

1.2. Email Discussion List

For questions and update information regarding ADSL Bandwidth Management please subscribe to the ADSL Bandwidth Management email list at http://jared.sonicspike.net/mailman/listinfo/adsl−qos.

1.3. Disclaimer

Neither the author nor the distributors, or any other contributor of this HOWTO are in any way responsible for physical, financial, moral or any other type of damage incurred by following the suggestions in this text.

1.4. Copyright and License

This document is copyright 2002 by Dan Singletary, and is released under the terms of the GNU Free Documentation License, which is hereby incorporated by reference.

1.5. Feedback and corrections

If you have questions or comments about this document, please feel free to contact the author at dvsing@sonicspike.net.

2. Background

2.1. Prerequisites

The method outlined in this document should work in other Linux configurations however it remains untested in any configuration but the following:

- Red Hat Linux 7.3
- 2.4.18−5 Kernel with QoS Support fully enabled (modules OK) and including the following kernel patches (which may eventually be included in later kernels):
    - HTB queue − http://luxik.cdi.cz/~devik/qos/htb/

      Note: it has been reported that kernels since version 2.4.18−3 shipped with Mandrake (8.1, 8.2) have already been patched for HTB.
    - IMQ device − http://luxik.cdi.cz/~patrick/imq/
- iptables v1.2.6a or later (version of iptables distributed with Red Hat 7.3 is missing the length module)

> Note: Previous versions of this document specified a method of bandwidth control that involved patching the existing sch_prio queue. It was found later that this patch was entirely unnecessary. Regardless, the newer methods outlined in this document will give you better results (although at the writing of this document *2* kernel patches are now necessary. :) Happy patching.)

2.2. Layout

In order to keep things simple, all references to network devices and configuration in this document will be with respect to the following network layout diagram:

```
              <-- 128kbit/s      --------------     <-- 10Mbit -->
 Internet <---------------------> | ADSL Modem | <--------------------
               1.5Mbit/s -->     --------------                     |
                                                                    | eth0
                                                                    V
                                                        -----------------
                                                        |               |
                                                        | Linux Router  |
                                                        |               |
                                                        -----------------
                                                         | .. | eth1..ethN
                                                         |    |
                                                         V    V

                                                      Local Network
```

2.3. Packet Queues

Packet queues are buckets that hold data for a network device when it can't be immediately sent. Most packet queues use a FIFO (first in, first out) discipline unless they've been specially configured to do otherwise. What this means is that when the packet queue for a device is completely full, the packet most recently placed in the queue will be sent over the device only after all the other packets in the queue at that time have been sent.

2.3.1. The Upstream

With an ADSL modem, bandwidth is asymmetric with 1.5Mbit/s typical downstream and 128kbit/sec typical upstream. Although this is the line speed, the interface between the Linux Router PC and the ADSL modem is typically at or above 10Mbit/s. If the interface to the Local Network is also 10Mbit/s, there will typically be NO QUEUING at the router when packets are sent from the Local Network to the Internet. Packets are sent out eth0 as fast as they are received from the Local Network. Instead, packets are queued at the ADSL modem since they are arriving at 10Mbit/s and only being sent at 128kbit/s. Eventually the packet queue at the ADSL modem will become full and any more packets sent to it will be silently dropped. TCP is designed to handle this and will adjust it's transmit window size accordingly to take full advantage of the available bandwidth.

While packet queues combined with TCP result in the most effective use of bandwidth, large FIFO queues can increase the latency for interactive traffic.

Another type of queue that is somewhat like FIFO is an n−band priority queue. However, instead of having just one queue that packets line up in, the n−band priority queue has n FIFO queues which packets are placed in by their classification. Each queue has a priority and packets are always dequeued from the highest priority queue that contains packets. Using this discipline FTP packets can be placed in a lower priority queue than telnet packets so that even during an FTP upload, a single telnet packet will jump the queue and be sent immediately.

This document has been revised to use a new queue in linux called the Hierarchical Token Bucket (HTB). The HTB queue is much like the n−band queue described above, but it has the capability to limit the rate of traffic in each class. In addition to this, it has the ability to set up classes of traffic beneath other classes creating a hierarchy of classes. Fully describing HTB is beyond the scope of this document, but more information can be found at http://www.lartc.org

2.3.2. The Downstream

Traffic coming inbound on your ADSL modem is queued in much the same way as outbound traffic, however the queue resides at your ISP. Because of this, you probably don't have direct control of how packets are queued or which types of traffic get preferential treatment. The only way to keep your latency low here is to make sure that people don't send you data too fast. Unfortunately, there's no way to directly control the speed at which packets arrive, but since a majority of your traffic is most likely TCP, there are some ways to slow down the senders:

- Intentionally drop inbound packets − TCP is designed to take full advantage of the available bandwidth while also avoiding congestion of the link. This means that during a bulk data transfer TCP will send more and more data until eventually a packet is dropped. TCP detects this and reduces it's transmission window. This cycle continues throughout the transfer and assures data is moved as quickly as possible.
- Manipulate the advertised receive window − During a TCP transfer, the receiver sends back a

continuous stream of acknowledgment (ACK) packets. Included in the ACK packets is a window size advertisement which states the maximum amount of unacknowledged data the receiver should send. By manipulating the window size of outbound ACK packets we can intentionally slow down the sender. At the moment there is no (free) implementation for this type of flow−control on Linux (however I may be working on one!).

3. How it Works

There are two basic steps to optimize upstream bandwidth. First we have to find a way to prevent the ADSL modem from queuing packets since we have no control over how it handles the queue. In order to do this we will throttle the amount of data the router sends out eth0 to be slightly less than the total upstream bandwidth of the ADSL modem. This will result in the router having to queue packets that arrive from the Local Network faster than it is allowed to send them.

The second step is to set up priority queuing discipline on the router. We'll investigate a queue that can be configured to give priority to interactive traffic such as telnet and multi−player games.

> By using the HTB queue we can accomplish bandwidth shaping and priority queuing at the same time while also assuring that no priority class is starved by another. Avoiding starvation wasn't possible using the method outlined in the 0.1 revision of this document.

The final step is to configure the firewall to prioritize packets by using fwmark.

3.1. Throttling Outbound Traffic with Linux HTB

Although the connection between the router and the modem is at 10Mbit/s, the modem is only able to send data at 128kbit/s. Any data sent in excess of that rate will be queued at the modem. Thus, a ping packet sent from the router may go to the modem immediately, but may take a few seconds to actually get sent out to the Internet if the queue in the modem has any packets in it. Unfortunately most ADSL modems provide no mechanism to specify how packets are dequeued or how large the queue is, so our first objective is to move the place where the outbound packets are queued to somewhere where we have more control over the queue.

We'll do this by using the HTB queue to limit the rate at which we send packets to the ADSL modem. Even though our upstream bandwidth may be 128kbit/s we'll have to limit the rate at which we send packets to be slightly below that. If we want to lower the latency we have to be SURE that not a single packet is ever queued at the modem. Through experimentation I have found that limiting the outbound traffic to about 90kbit/s gives me almost 95% of the bandwidth I could achieve without HTB rate control. With HTB enabled at this rate, we've prevented the ADSL modem from queuing packets.

3.2. Priority Queuing with HTB

> Note: previous claims in this section (originally named N−band priority queuing) were later found to be incorrect. It actually WAS possible to classify packets into the individual bands of the priority queue by only using the fwmark field, however it was poorly documented at the writing of version 0.1 of this document

At this point we still haven't realized any change in the performance. We've merely moved the FIFO queue from the ADSL modem to the router. In fact, with Linux configured to a default queue size of 100 packets we've probably made our problem worse at this point! But not for long...

Each neighbor class in an HTB queue can be assigned a priority. By placing different types of traffic in different classes and then assigning these classes different priorities, we can control the order in which packets are dequeued and sent. HTB makes it possible to do this while still avoiding starvation of any one class, since we're able to specify a minimum guaranteed rate for each class. In addition to this, HTB allows for us to tell a class that it may use any unused bandwidth from other classes up to a certain ceiling.

Once we have our classes set up, we set up filters to place traffic in classes. There are several ways to do this, but the method described in this document uses the familiar iptables/ipchains to mark packets with an fwmark. The filters place traffic into the classes of the HTB queue based on their fwmark. This way, we're able to set up matching rules in iptables to send certain types of traffic to certain classes.

3.3. Classifying Outbound Packets with iptables

Note: originally this document used ipchains to classify packets. The newer iptables is now used.

The final step in configuring your router to give priority to interactive traffic is to set up the firewall to define how traffic should be classified. This is done by setting the packet's fwmark field.

Without getting into too much detail, here is a simplified description of how outbound packets might be classified into 4 classes with the highest priority class being 0x00:

1. Mark ALL packets as 0x03. This places all packets, by default, into the lowest priority queue.
2. Mark ICMP packets as 0x00. We want ping to show the latency for the highest priority packets.
3. Mark all packets that have a destination port 1024 or less as 0x01. This gives priority to system services such as Telnet and SSH. FTP's control port will also fall into this range however FTP data transfer takes place on high ports and will remain in the 0x03 band.
4. Mark all packets that have a destination port of 25 (SMTP) as 0x03. If someone sends an email with a large attachment we don't want it to swamp interactive traffic.
5. Mark all packets that are going to a multi−player game server as 0x02. This will give gamers low latency but will keep them from swamping out the the system applications that require low latency.

   Mark any "small" packets as 0x02. Outbound ACK packets from inbound downloads should be sent promptly to assure efficient downloads. This is possible using the iptables length module.

Obviously, this can be customized to fit your needs.

3.4. A few more tweaks...

There are two more things that you can do to improve your latency. First, you can set the Maximum Transmittable Unit (mtu) to be lower than the default of 1500 bytes. Lowering this number will lower the average time you have to wait to send a priority packet if there is already a full−sized low−priority packet being sent. Lowering this number will also slightly decrease your throughput because each packet contains at least 40 bytes worth of IP and TCP header information.

The other thing you can do to improve latency even on your low−priority traffic is to lower your queue length from the default of 100, which on an ADSL line could take as much as 10 seconds to empty with a 1500 byte mtu.

3.5. Attempting to Throttle Inbound Traffic

By using the Intermediate Queuing Device (IMQ), we can run all incoming packets through a queue in the same way that we queue outbound packets. Packet priority is much simpler in this case. Since we can only (attempt to) control inbound TCP traffic, we'll put all non−TCP traffic in the 0x00 class, and all TCP traffic in the 0x01 class. We'll also place "small" TCP packets in the 0x00 class since these are most likely ACK packets for outbound data that has already been sent. We'll set up a standard FIFO queue on the 0x00 class, and we'll set up a Random Early Drop (RED) queue on the 0x01 class. RED is better than a FIFO (tail−drop) queue at controlling TCP because it will drop packets before the queue overflows in an attempt to slow down transfers that look like they're about to get out of control. We'll also rate−limit both classes to some maximum inbound rate which is less than your true inbound speed over the ADSL modem.

3.5.1. Why Inbound Traffic Limiting isn't all That Good

We want to limit our inbound traffic to avoid filling up the queue at the ISP, which can sometimes buffer as much as 5 seconds worth of data. The problem is that currently the only way to limit inbound TCP traffic is to drop perfectly good packets. These packets have already taking up some share of bandwidth on the ADSL modem only to be dropped by the Linux box in an effort to slow down future packets. These dropped packets will eventually be retransmitted consuming more bandwidth. When we limit traffic, we are limiting the rate of packets which we will accept into our network. Since the *actual* inbound data rate is somewhere above this because of the packets we drop, we'll actually have to limit our downstream to *much* lower than the actual rate of the ADSL modem in order to assure low latency. In practice I had to limit my 1.5mbit/s downstream ADSL to 700kbit/sec in order to keep the latency acceptable with 5 concurrent downloads. The more TCP sessions you have, the more bandwidth you'll waste with dropped packets, and the lower you'll have to set your limit rate.

A much better way to control inbound TCP traffic would be TCP window manipulation, but as of this writing there exists no (free) implementation of it for Linux (that I know of...).

4. Implementation

Now with all of the explanation out of the way it's time to implement bandwidth management with Linux.

4.1. Caveats

Limiting the actual rate of data sent to the DSL modem is not as simple as it may seem. Most DSL modems are really just ethernet bridges that bridge data back and forth between your linux box and the gateway at your ISP. Most DSL modems use ATM as a link layer to send data. ATM sends data in cells that are always 53 bytes long. 5 of these bytes are header information, leaving 48 bytes available for data. Even if you are sending 1 byte of data, an entire 53 bytes of bandwidth are consumed sent since ATM cells are always 53 bytes long. This means that if you are sending a typical TCP ACK packet which consists of 0 bytes data + 20 bytes TCP header + 20 bytes IP header + 18 bytes Ethernet header. In actuality, even though the ethernet packet you are sending has only 40 bytes of payload (TCP and IP header), the minimum payload for an Ethernet packet is 46 bytes of data, so the remaining 6 bytes are padded with nulls. This means that the actual length of the Ethernet packet plus header is 18 + 46 = 64 bytes. In order to send 64 bytes over ATM, you have to send two ATM cells which consume 106 bytes of bandwidth. This means for every TCP ACK packet, you're wasting 42 bytes of bandwidth. This would be okay if Linux accounted for the encapsulation that the DSL modem uses, but instead, Linux only accounts the TCP header, IP header, and 14 bytes of the MAC address (Linux doesn't count the 4 bytes CRC since this is handled at the hardware level). Linux doesn't count the minimum Ethernet packet size of 46 bytes, nor does it take into account the fixed ATM cell size.

What all of this means is that you'll have to limit your outbound bandwidth to somewhat less than your true capacity (until we can figure out a packet scheduler that can account for the various types of encapsulation being used). You may find that you've figured out a good number to limit your bandwidth to, but then you download a big file and the latency starts to shoot up over 3 seconds. This is most likely because the bandwidth those small ACK packets consume is being miscalculated by Linux.

I have been working on a solution to this problem for a few months and have almost settled on a solution that I will soon release to the public for further testing. The solution involves using a user−space queue instead of linux's QoS to rate−limit packets. I've basically implemented a simple HTB queue using linux user−space queues. This solution (so far) has been able to regulate outbound traffic SO WELL that even during a massive bulk download (several streams) and bulk upload (gnutella, several streams) the latency PEAKS at 400ms over my nominal no−traffic latency of about 15ms. For more information on this QoS method, subscribe to the email list for updates or check back on updates to this HOWTO.

4.2. Script: myshaper

The following is a listing of the script which I use to control bandwidth on my Linux router. It uses several of the concepts covered in the document. Outbound traffic is placed into one of 7 queues depending on type. Inbound traffic is placed into two queues with TCP packets being dropped first (lowest priority) if the inbound data is over−rate. The rates given in this script seem to work OK for my setup but your results may vary.

> This script was originally based on the ADSL WonderShaper as seen at the LARTC website.

```
#!/bin/bash
#
```

```
# myshaper - DSL/Cable modem outbound traffic shaper and prioritizer.
#            Based on the ADSL/Cable wondershaper (www.lartc.org)
#
# Written by Dan Singletary (8/7/02)
#
# NOTE!! - This script assumes your kernel has been patched with the
#          appropriate HTB queue and IMQ patches available here:
#          (subnote: future kernels may not require patching)
#
#       http://luxik.cdi.cz/~devik/qos/htb/
#       http://luxik.cdi.cz/~patrick/imq/
#
# Configuration options for myshaper:
#  DEV    - set to ethX that connects to DSL/Cable Modem
#  RATEUP - set this to slightly lower than your
#           outbound bandwidth on the DSL/Cable Modem.
#           I have a 1500/128 DSL line and setting
#           RATEUP=90 works well for my 128kbps upstream.
#           However, your mileage may vary.
#  RATEDN - set this to slightly lower than your
#           inbound bandwidth on the DSL/Cable Modem.
#
#
#  Theory on using imq to "shape" inbound traffic:
#
#     It's impossible to directly limit the rate of data that will
#  be sent to you by other hosts on the internet.  In order to shape
#  the inbound traffic rate, we have to rely on the congestion avoidance
#  algorithms in TCP.  Because of this, WE CAN ONLY ATTEMPT TO SHAPE
#  INBOUND TRAFFIC ON TCP CONNECTIONS.  This means that any traffic that
#  is not tcp should be placed in the high-prio class, since dropping
#  a non-tcp packet will most likely result in a retransmit which will
#  do nothing but unnecessarily consume bandwidth.
#     We attempt to shape inbound TCP traffic by dropping tcp packets
#  when they overflow the HTB queue which will only pass them on at
#  a certain rate (RATEDN) which is slightly lower than the actual
#  capability of the inbound device.  By dropping TCP packets that
#  are over-rate, we are simulating the same packets getting dropped
#  due to a queue-overflow on our ISP's side.  The advantage of this
#  is that our ISP's queue will never fill because TCP will slow it's
#  transmission rate in response to the dropped packets in the assumption
#  that it has filled the ISP's queue, when in reality it has not.
#     The advantage of using a priority-based queuing discipline is
#  that we can specifically choose NOT to drop certain types of packets
#  that we place in the higher priority buckets (ssh, telnet, etc).  This
#  is because packets will always be dequeued from the lowest priority class
#  with the stipulation that packets will still be dequeued from every
#  class fairly at a minimum rate (in this script, each bucket will deliver
#  at least it's fair share of 1/7 of the bandwidth).
#
#  Reiterating main points:
#   * Dropping a tcp packet on a connection will lead to a slower rate
#     of reception for that connection due to the congestion avoidance algorithm.
#   * We gain nothing from dropping non-TCP packets.  In fact, if they
#     were important they would probably be retransmitted anyways so we want to
#     try to never drop these packets.  This means that saturated TCP connections
#     will not negatively effect protocols that don't have a built-in retransmit like TCP.
#   * Slowing down incoming TCP connections such that the total inbound rate is less
#     than the true capability of the device (ADSL/Cable Modem) SHOULD result in little
#     to no packets being queued on the ISP's side (DSLAM, cable concentrator, etc).  Since
#     these ISP queues have been observed to queue 4 seconds of data at 1500Kbps or 6 megabits
#     of data, having no packets queued there will mean lower latency.
```

```
#
#  Caveats (questions posed before testing):
#   * Will limiting inbound traffic in this fashion result in poor bulk TCP performance?
#     - Preliminary answer is no!  Seems that by prioritizing ACK packets (small <64b)
#       we maximize throughput by not wasting bandwidth on retransmitted packets
#       that we already have.
#

# NOTE: The following configuration works well for my
# setup: 1.5M/128K ADSL via Pacific Bell Internet (SBC Global Services)

DEV=eth0
RATEUP=90
RATEDN=700  # Note that this is significantly lower than the capacity of 1500.
            # Because of this, you may not want to bother limiting inbound traffic
            # until a better implementation such as TCP window manipulation can be used.

#
# End Configuration Options
#

if [ "$1" = "status" ]
then
        echo "[qdisc]"
        tc -s qdisc show dev $DEV
        tc -s qdisc show dev imq0
        echo "[class]"
        tc -s class show dev $DEV
        tc -s class show dev imq0
        echo "[filter]"
        tc -s filter show dev $DEV
        tc -s filter show dev imq0
        echo "[iptables]"
        iptables -t mangle -L MYSHAPER-OUT -v -x 2> /dev/null
        iptables -t mangle -L MYSHAPER-IN -v -x 2> /dev/null
        exit
fi

# Reset everything to a known state (cleared)
tc qdisc del dev $DEV root    2> /dev/null > /dev/null
tc qdisc del dev imq0 root 2> /dev/null > /dev/null
iptables -t mangle -D POSTROUTING -o $DEV -j MYSHAPER-OUT 2> /dev/null > /dev/null
iptables -t mangle -F MYSHAPER-OUT 2> /dev/null > /dev/null
iptables -t mangle -X MYSHAPER-OUT 2> /dev/null > /dev/null
iptables -t mangle -D PREROUTING -i $DEV -j MYSHAPER-IN 2> /dev/null > /dev/null
iptables -t mangle -F MYSHAPER-IN 2> /dev/null > /dev/null
iptables -t mangle -X MYSHAPER-IN 2> /dev/null > /dev/null
ip link set imq0 down 2> /dev/null > /dev/null
rmmod imq 2> /dev/null > /dev/null

if [ "$1" = "stop" ]
then
        echo "Shaping removed on $DEV."
        exit
fi

###########################################################
#
# Outbound Shaping (limits total bandwidth to RATEUP)

# set queue size to give latency of about 2 seconds on low-prio packets
ip link set dev $DEV qlen 30
```

```
# changes mtu on the outbound device.  Lowering the mtu will result
# in lower latency but will also cause slightly lower throughput due
# to IP and TCP protocol overhead.
ip link set dev $DEV mtu 1000

# add HTB root qdisc
tc qdisc add dev $DEV root handle 1: htb default 26

# add main rate limit classes
tc class add dev $DEV parent 1: classid 1:1 htb rate ${RATEUP}kbit

# add leaf classes - We grant each class at LEAST it's "fair share" of bandwidth.
#                    this way no class will ever be starved by another class.  Each
#                    class is also permitted to consume all of the available bandwidth
#                    if no other classes are in use.
tc class add dev $DEV parent 1:1 classid 1:20 htb rate $[$RATEUP/7]kbit ceil ${RATEUP}kbit prio 0
tc class add dev $DEV parent 1:1 classid 1:21 htb rate $[$RATEUP/7]kbit ceil ${RATEUP}kbit prio 1
tc class add dev $DEV parent 1:1 classid 1:22 htb rate $[$RATEUP/7]kbit ceil ${RATEUP}kbit prio 2
tc class add dev $DEV parent 1:1 classid 1:23 htb rate $[$RATEUP/7]kbit ceil ${RATEUP}kbit prio 3
tc class add dev $DEV parent 1:1 classid 1:24 htb rate $[$RATEUP/7]kbit ceil ${RATEUP}kbit prio 4
tc class add dev $DEV parent 1:1 classid 1:25 htb rate $[$RATEUP/7]kbit ceil ${RATEUP}kbit prio 5
tc class add dev $DEV parent 1:1 classid 1:26 htb rate $[$RATEUP/7]kbit ceil ${RATEUP}kbit prio 6

# attach qdisc to leaf classes - here we at SFQ to each priority class.  SFQ insures that
#                                within each class connections will be treated (almost) fairly.
tc qdisc add dev $DEV parent 1:20 handle 20: sfq perturb 10
tc qdisc add dev $DEV parent 1:21 handle 21: sfq perturb 10
tc qdisc add dev $DEV parent 1:22 handle 22: sfq perturb 10
tc qdisc add dev $DEV parent 1:23 handle 23: sfq perturb 10
tc qdisc add dev $DEV parent 1:24 handle 24: sfq perturb 10
tc qdisc add dev $DEV parent 1:25 handle 25: sfq perturb 10
tc qdisc add dev $DEV parent 1:26 handle 26: sfq perturb 10

# filter traffic into classes by fwmark - here we direct traffic into priority class according to
#                                         the fwmark set on the packet (we set fwmark with iptables
#                                         later).  Note that above we've set the default priority
#                                         class to 1:26 so unmarked packets (or packets marked with
#                                         unfamiliar IDs) will be defaulted to the lowest priority
#                                         class.
tc filter add dev $DEV parent 1:0 prio 0 protocol ip handle 20 fw flowid 1:20
tc filter add dev $DEV parent 1:0 prio 0 protocol ip handle 21 fw flowid 1:21
tc filter add dev $DEV parent 1:0 prio 0 protocol ip handle 22 fw flowid 1:22
tc filter add dev $DEV parent 1:0 prio 0 protocol ip handle 23 fw flowid 1:23
tc filter add dev $DEV parent 1:0 prio 0 protocol ip handle 24 fw flowid 1:24
tc filter add dev $DEV parent 1:0 prio 0 protocol ip handle 25 fw flowid 1:25
tc filter add dev $DEV parent 1:0 prio 0 protocol ip handle 26 fw flowid 1:26

# add MYSHAPER-OUT chain to the mangle table in iptables - this sets up the table we'll use
#                                                      to filter and mark packets.
iptables -t mangle -N MYSHAPER-OUT
iptables -t mangle -I POSTROUTING -o $DEV -j MYSHAPER-OUT

# add fwmark entries to classify different types of traffic - Set fwmark from 20-26 according to
#                                                             desired class. 20 is highest prio.
iptables -t mangle -A MYSHAPER-OUT -p tcp --sport 0:1024 -j MARK --set-mark 23 # Default for low port traffic
iptables -t mangle -A MYSHAPER-OUT -p tcp --dport 0:1024 -j MARK --set-mark 23 # ""
iptables -t mangle -A MYSHAPER-OUT -p tcp --dport 20 -j MARK --set-mark 26     # ftp-data port, low prio
iptables -t mangle -A MYSHAPER-OUT -p tcp --dport 5190 -j MARK --set-mark 23   # aol instant messenger
iptables -t mangle -A MYSHAPER-OUT -p icmp -j MARK --set-mark 20               # ICMP (ping) - high prio, impress friends
iptables -t mangle -A MYSHAPER-OUT -p udp -j MARK --set-mark 21                # DNS name resolution (small packets)
iptables -t mangle -A MYSHAPER-OUT -p tcp --dport ssh -j MARK --set-mark 22    # secure shell
```

```
iptables -t mangle -A MYSHAPER-OUT -p tcp --sport ssh -j MARK --set-mark 22    # secure shell
iptables -t mangle -A MYSHAPER-OUT -p tcp --dport telnet -j MARK --set-mark 22 # telnet (ew...)
iptables -t mangle -A MYSHAPER-OUT -p tcp --sport telnet -j MARK --set-mark 22 # telnet (ew...)
iptables -t mangle -A MYSHAPER-OUT -p ipv6-crypt -j MARK --set-mark 24         # IPSec - we don't know what the payload is though...
iptables -t mangle -A MYSHAPER-OUT -p tcp --sport http -j MARK --set-mark 25   # Local web server
iptables -t mangle -A MYSHAPER-OUT -p tcp -m length --length :64 -j MARK --set-mark 21 # small packets (probably just ACKs)
iptables -t mangle -A MYSHAPER-OUT -m mark --mark 0 -j MARK --set-mark 26      # redundant- mark any unmarked packets as 26 (low prio)

# Done with outbound shaping
#
####################################################

echo "Outbound shaping added to $DEV.  Rate: ${RATEUP}Kbit/sec."

# uncomment following line if you only want upstream shaping.
# exit

####################################################
#
# Inbound Shaping (limits total bandwidth to RATEDN)

# make sure imq module is loaded

modprobe imq numdevs=1

ip link set imq0 up

# add qdisc - default low-prio class 1:21

tc qdisc add dev imq0 handle 1: root htb default 21

# add main rate limit classes
tc class add dev imq0 parent 1: classid 1:1 htb rate ${RATEDN}kbit

# add leaf classes - TCP traffic in 21, non TCP traffic in 20
#
tc class add dev imq0 parent 1:1 classid 1:20 htb rate $[$RATEDN/2]kbit ceil ${RATEDN}kbit prio 0
tc class add dev imq0 parent 1:1 classid 1:21 htb rate $[$RATEDN/2]kbit ceil ${RATEDN}kbit prio 1

# attach qdisc to leaf classes - here we at SFQ to each priority class.  SFQ insures that
#                                 within each class connections will be treated (almost) fairly.
tc qdisc add dev imq0 parent 1:20 handle 20: sfq perturb 10
tc qdisc add dev imq0 parent 1:21 handle 21: red limit 1000000 min 5000 max 100000 avpkt 1000 burst 50

# filter traffic into classes by fwmark - here we direct traffic into priority class according to
#                                          the fwmark set on the packet (we set fwmark with iptables
#                                          later).  Note that above we've set the default priority
#                                          class to 1:26 so unmarked packets (or packets marked with
#                                          unfamiliar IDs) will be defaulted to the lowest priority
#                                          class.
tc filter add dev imq0 parent 1:0 prio 0 protocol ip handle 20 fw flowid 1:20
tc filter add dev imq0 parent 1:0 prio 0 protocol ip handle 21 fw flowid 1:21

# add MYSHAPER-IN chain to the mangle table in iptables - this sets up the table we'll use
#                                                          to filter and mark packets.
iptables -t mangle -N MYSHAPER-IN
iptables -t mangle -I PREROUTING -i $DEV -j MYSHAPER-IN

# add fwmark entries to classify different types of traffic - Set fwmark from 20-26 according to
#                                                              desired class. 20 is highest prio.
iptables -t mangle -A MYSHAPER-IN -p ! tcp -j MARK --set-mark 20              # Set non-tcp packets to highest priority
iptables -t mangle -A MYSHAPER-IN -p tcp -m length --length :64 -j MARK --set-mark 20 # short TCP packets are probably ACKs
```

```
iptables -t mangle -A MYSHAPER-IN -p tcp --dport ssh -j MARK --set-mark 20    # secure shell
iptables -t mangle -A MYSHAPER-IN -p tcp --sport ssh -j MARK --set-mark 20    # secure shell
iptables -t mangle -A MYSHAPER-IN -p tcp --dport telnet -j MARK --set-mark 20 # telnet (ew...)
iptables -t mangle -A MYSHAPER-IN -p tcp --sport telnet -j MARK --set-mark 20 # telnet (ew...)
iptables -t mangle -A MYSHAPER-IN -m mark --mark 0 -j MARK --set-mark 21              # redundant- mark any unmarked packets as 26 (low prio)

# finally, instruct these packets to go through the imq0 we set up above
iptables -t mangle -A MYSHAPER-IN -j IMQ

# Done with inbound shaping
#
####################################################

echo "Inbound shaping added to $DEV.  Rate: ${RATEDN}Kbit/sec."
```

5. Testing the New Queue

The easiest way to test your new setup is to saturate the upstream with low−priority traffic. This depends how you have your priorities set up. For the sake of example, let's say you've placed telnet traffic and ping traffic at a higher priority (lower fwmark) than other high ports (that are used for FTP transfers, etc). If you initiate an FTP upload to saturate upstream bandwidth, you should only notice your ping times to the gateway (on the other side of the DSL line) increasing by a small amount compared to what it would increase to with no priority queuing. Ping times under 100ms are typical depending on how you've got things set up. Ping times greater than one or two seconds probably mean that things aren't working right.

6. OK It Works!! Now What?

Now that you've successfully started to manage your bandwidth, you should start thinking of ways to use it. After all, you're probably paying for it!

- Use a Gnutella client and SHARE YOUR FILES without adversely affecting your network performance
- Run a web server without having web page hits slow you down in Quake

7. Related Links

- Bandwidth Controller for Windows – http://www.bandwidthcontroller.com
- dsl_qos_queue – (beta) for Linux. No kernel patching, and better performance –