A Practical Introduction to Quality of Service using Traffic Shaping

Thomas Graf <graf@swiss.nexus-ag.com>

28th March 2002

Contents

Chapter 1

Introduction

This document tries to give you some practical explanations and examples about howto implement quality of service using traffic shaping. The initial version of this document was created during a project of 10 work days as some kind of degree dissertation.

My appologies for the sometimes bad or uncommon english, if you find some unscrutable parts do not hesitate to contact me.

I tried to avoid errors but I cannot guarantee that everything is correct I wrote. Therefore do not rely on this informations.

This document is dedicated to Alexey N. Kuznetsov for writing all the cool linux networking code, the folks at lartc.org for their help and piro at megatokyo.com for making me laugh and cry with his web comic.

1.1 Disclaimer & License

1.2 Prior knowledge

I did not have the time to explain everything from the basics, thus a prior knowledge in networking and linux administration is needed. You should know the protocol specification of the TCP/IP protocol family and how packets are routed in the linux kernel. Further, knowledge about a UNIX shell is necessary is a must to understand the examples.

I suggest to look into the Linux Networking-concepts HOWTO [1] and take a look at the RFCs listed in the references at the end of this document.

Chapter 2

Network analysis

2.1 Introduction

A good and working Quality of Service setup requires a detailed network analysis. Make sure you know as much as possible about your network. You need to know exactly what is between you (i.e. the gateway you want to shape on) and your ISP[1].

2.2 Exploring the Routing Ways

Knowing the common routes is not a must when shaping, but can help track down problems. It's also good to know if you experience network problems between you and your ISP because you can check easily where the problem actually is and maybe change your ISP.

Finding out routes is quite easy, I use traceroute as described in [5, RFC1393]

```
# traceroute -n -I 62.2.89.177
traceroute to 62.2.89.177, 30 hops max, 38 byte packets
1  192.168.10.1 2.653 ms 2.472 ms 2.366 ms
2  146.228.52.21 11.888 ms 11.272 ms 11.221 ms
3  146.228.60.253 129.141 ms 150.139 ms 139.899 ms
4  146.228.4.50 154.203 ms 158.341 ms 153.568 ms
5  46.228.253.66 143.879 ms 128.648 ms 137.259 ms
6  194.42.48.7 144.640 ms 124.209 ms 72.644 ms
7  62.2.6.33 73.497 ms 35.078 ms 50.025 ms
8  62.2.18.169 35.964 ms 24.213 ms 16.791 ms
9  62.2.94.186 20.635 ms 16.283 ms 19.034 ms
10 62.2.89.177 178.370 ms 47.138 ms 29.101 ms
```

Let's go through the output step by step. The first node is our gateway identifiable by the local address and its fast reply. The next 4 nodes are part of the network 146.228.0.0/16 owned by EUnet Switzerland. 194.42.48.7 belongs to

[1] Internet Service Provider

192.42.48.0/24 (Telehouse) and seems to be the gateway to the Cablecom network. Then we already enter the final network 62.2.0.0/16 (Cablecom). Looking at the response times which are quite good and the routers are doing a good job.
Repeat this step for other destinations and compare the outputs.

2.3 Measure the Latency

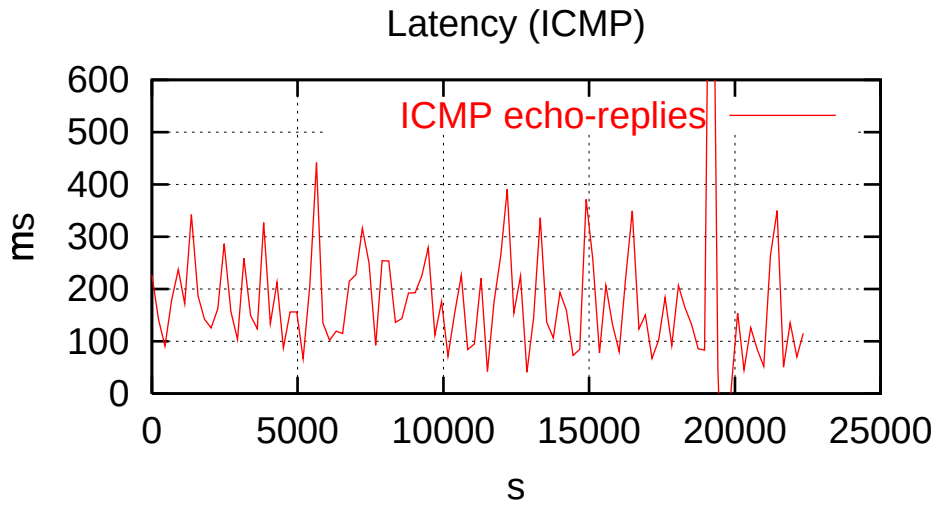
Knowing the latency is essential. You'll need it for the shaping configuration and to prove that you actually improved something with your shaping setup. The easiest way is to use the ping utility which uses the ICMP[2] protocol.

```
# ping -i 30 62.2.89.177 | tee ping.log
PING 62.2.89.177 (62.2.89.177): 56 data bytes
64 bytes from 62.2.89.177: icmp_seq=0 ttl=246 time=150.2 ms
64 bytes from 62.2.89.177: icmp_seq=1 ttl=246 time=95.1 ms
...
```

This sends ICMP echo-request packets to 62.2.89.177 every 30 seconds requesting ICMP echo-reply packets. We use tee to write the output to the file ping.log for further statistics. The following little perl script extracts the response times and prepare the data to feed gnuplot[3].

```
# cat ping.log | perl -e '$c=0; while(<>) { \
chomp; m/time=(.*?)\s/i; if($1 ne "") { \
print "$c\t$1\n"; $c += 30;}}'
```

This extracts the response times and prints them with a proper timestamp ready to feed gnuplot:



[2] Internet Control Message Protocol
[3] A tool to display numerical series.

The replies are irregular and the response times vary between 50 and 400ms. Settle this irregular behaviour and minimise the average latency is one goal to reach.

```
# cat ping.log | perl -e '$c=0; $t=0; while(<>) { \
chomp; m/time=(.*?)\s/i; if($1 ne "") { $t+=$1; $c++; } \
} printf "%.2f\n", $t/$c;'
208.23
```

The perl script above calculates the average ping response time.

2.4 Monitor the Traffic

The most important part is to analyse the traffic produced by customers, employees or other companies you share the link with. I suggest to capture all the traffic for some time using a sniffer like tcpdump:

```
# tcpdump -i eth0 -n -w `hostname`-`date -I`.raw
tcpdump: listening on eth0
```

This captures all packets on interface eth0 (first Ethernet device on linux) and writes them to a file called hostname-yyyy-mm-dd.raw. The advantage of this method compared to sniffing and analysing at the same time is, that you are able to generate different statistics with the same input data.

We'll use this data for further analysis from now on.

Be careful not to reach the maximum file size limit of your operating system, I've run tcpdump for 4.5 hours which resulted in a 250 MB file. I suggest to prefer real time statistics if you want to monitor more than a few hours or to split incoming and outgoing traffic into two different files.

2.4.1 Overall bandwidth usage

Let's get an overview over the traffic. Our company is using a synchron 256KBit line which provides Internet access for two networks using a different IP range.

Outgoing packets:

```
# tcpstat -r i.25-2-2002.raw -f 'src net 193.73.218.0/24' -a
Bytes/sec = 2.3 KB
Bytes/minute = 139.0 KB
Bytes/hour = 8.1 MB

# tcpstat -r i.25-2-2002.raw -f 'src net 195.48.118.0/24' -a
Bytes/sec = 16.5 B
Bytes/minute = 990.1 B
Bytes/hour = 58.0 KB
```

Incoming packets:

```
# tcpstat -r i.25-2-2002.raw -f 'dst net 193.73.218.0/24' -a
Bytes/sec = 18.9 KB
Bytes/minute = 1.1 MB
Bytes/hour = 66.4 MB

# tcpstat -r i.25-2-2002.raw -f 'dst net 195.48.118.0/24' -a
Bytes/sec = 393.5 B
Bytes/minute = 23.1 KB
Bytes/hour = 1.4 MB
```

The first network (193.73.218.0/24) is far more interesting and should be worth a closer look. In general the line is used quite regularly and is at it's upper limit. The outgoing traffic seems to be mostly control traffic whereas incoming traffic must be bulk data to reach such a rate.

2.4.2 Packet Size and ACKs[4]

```
# nstats -r i.25-2-2002.raw -f 'src net 193.73.218.0/24' -o ip
Protocol (IP)            #      %        bytes      %     avg
-----------------------------------------------------------------
IPv4:               224905 100.00     37979628 100.00   168.9
...

# nstats -r i.25-2-2002.raw -f 'dst net 193.73.218.0/24' -o ip
Protocol (IP)            #      %        bytes      %     avg
-----------------------------------------------------------------
IPv4:               263178 100.00    287553622 100.00 1092.6
...
```

225k packets went out with an average packet size of 168 bytes and 263k packets came in with an average packet size of 1092 bytes. So we have nearly the same amount of packets but totally different rates. The nearly equal packet count is quite normal for TCP[5].

```
# nstats -r i.25-2-2002.raw -f 'src net 193.73.218.0/24' \
-o tcpopt
TCP Options (IPv4)       #      %        bytes      %     avg
-----------------------------------------------------------------
URG:                     0   0.00            0   0.00     0.0
ACK:                204925  93.62     36524706  97.58   178.2
PSH:                 25700  11.74     11942546  31.91   464.7
RST:                  2335   1.07       149458   0.40    64.0
SYN:                 12390   5.66       804068   2.15    64.9
FIN:                  6550   2.99       518317   1.38    79.1
```

[4] Acknowledge (Flag in TCP header to mark packets verifying data packets)
[5] Transmission Control Protocol

```
# nstats -r i.25-2-2002.raw -f 'dst net 193.73.218.0/24' \
-o tcpopt
TCP Options (IPv4)       #      %        bytes      %     avg
-------------------------------------------------------------
URG:                     0   0.00            0   0.00     0.0
ACK:                255286  99.43    286354513  99.97  1121.7
PSH:                 51236  19.96     47670818  16.64   930.4
RST:                  2634   1.03       168596   0.06    64.0
SYN:                  9125   3.55       593192   0.21    65.0
FIN:                  8724   3.40      1771586   0.62   203.1
```

Except for the initial SYN packet, every packet in a connection must have the ACK bit set, that explains the high amount of packets which have the ACK bit set. Every packet must have the SYN, ACK, FIN-ACK or RST-ACK bit(s) set, other packets are illegal. So, the amount of ACK packets plus the amount of SYN packets should result around 100%. It's not exactly 100% as a SYN-ACK combination is also possible.

2.4.3 Protocol breakdown on layer 3

Let's break down the packets on OSI layer 3. We print out the number of ARP[6], IPv4 and IPv6 packets.

```
# nstats -r i.25-2-2002.raw -o ether,ip
Protocol (Ethernet)      #      %        bytes      %     avg
-------------------------------------------------------------
IP:                 513028  99.66    339687876  99.95   662.1
ARP                     50   0.01         3200   0.00    64.0
RARP:                    0   0.00            0   0.00     0.0
Other:                1709   0.33       171840   0.05   100.6
-------------------------------------------------------------
Total:              514787 100.00    339862916 100.00   660.2

Protocol (IP)            #      %        bytes      %     avg
-------------------------------------------------------------
IPv4:               513028 100.00    339687876 100.00   662.1
IPv6:                    0   0.00            0   0.00     0.0
-------------------------------------------------------------
Total:              513028 100.00    339687876 100.00   662.1
```

As you can see, there is some ARP traffic which is normal on a local Ethernet network, quite a lot of IPv4 traffic and no IPv6 traffic. There is some non-ip traffic, probably CDP[7]. Note that we used all traffic for this statistic to see if there is any IPv6 traffic.

[6] Address Resolution Protocol
[7] Cisco Discovery Protocol

2.4.4 Protocol breakdown on layer 4

Having only IPV4 makes it a bit easier, but let's look at the protocol breakdown of ICMP, TCP and UDP[8]:

```
# nstats -r i.25-2-2002.raw -o proto -f 'net 193.73.218.0/24'
Protocol (IPv4)          #      %        bytes      %     avg
-------------------------------------------------------------
TCP:                475636  97.45    323879857  99.49   680.9
UDP:                 10672   2.19      1477815   0.45   138.5
ICMP:                 1761   0.36       174822   0.05    99.3
Other:                   0   0.00            0   0.00     0.0
-------------------------------------------------------------
Total:              488069 100.00    325532494 100.00   667.0
```

As you probably guessed, most of the traffic is TCP. The UDP traffic is probably caused by the DNS[9] protocol, but we'll see it by looking at layer 5. There is a bit of ICMP which might be interesting.

2.4.5 ICMP type breakdown

```
# nstats -r i.25-2-2002.raw -o icmp -f 'net 193.73.218.0/24'
ICMP Types (IPv4)        #      %        bytes      %     avg
-------------------------------------------------------------
Echo-Request:          608  34.53        65548  37.49   107.8
Echo-Reply:            986  55.99        95464  54.61    96.8
Dest Unreach:           77   4.37         6142   3.51    79.8
Source Quench:           0   0.00            0   0.00     0.0
Redirect:                0   0.00            0   0.00     0.0
Time Exceeded:          90   5.11         7668   4.39    85.2
Parameter Prob:          0   0.00            0   0.00     0.0
Timetstamp:              0   0.00            0   0.00     0.0
Timetstamp Rpy:          0   0.00            0   0.00     0.0
Info Request:            0   0.00            0   0.00     0.0
Info Reply:              0   0.00            0   0.00     0.0
Address:                 0   0.00            0   0.00     0.0
Address Reply:           0   0.00            0   0.00     0.0
Other:                   0   0.00            0   0.00     0.0
```

There is some echo-request/echo-reply traffic which can be caused by many applications. What we can say for sure is, that some of the echo-requests were sent to a broadcast or multi-cast address. The destination unreachable and time exceeded could be interesting, so we'll have a closer look.

[8] User Datagram Protocol
[9] Domain Name System

```
# nstats -r i.25-2-2002.raw -o destu -f 'net 193.73.218.0/24'
ICMP Dest. Unreach.      #      %        bytes      %     avg
-------------------------------------------------------------
Net Unreach:            10  12.99          740  12.05    74.0
Host Unreach:           12  15.58          941  15.32    78.4
Packet Filtered:        19  24.68         1406  22.89    74.0
Other:                  36  46.75         3055  49.74    84.9
...
```

Network/Host unreachable is the normal ICMP message sent when a router can't reach a network or a host. We can't do anything with packet filtered.

```
# nstats -r i.25-2-2002.raw -o texc -f 'net 193.73.218.0/24'
ICMP Time Exceeded       #      %        bytes      %     avg
-------------------------------------------------------------
TTL:                    90 100.00         7668 100.00    85.2
Fragtime:                0   0.00            0   0.00     0.0
Other:                   0   0.00            0   0.00     0.0
```

All ICMP Time Exceeded packets are caused by packets whose TTL[10] was zero and a router dropped it.

2.4.6 Protocol breakdown on layer 5

```
# nstats -r i.25-2-2002.raw -o port -f 'net 193.73.218.0/24'
Port (IP)                #      %        bytes      %     avg
-------------------------------------------------------------
www                 386397  79.46    277261984  85.17   717.6
1214                 15718   3.23      8452895   2.60   537.8
smtp                 14870   3.06     11723986   3.60   788.4
domain               10391   2.14      1463098   0.45   140.8
https                 7757   1.60      3028940   0.93   390.5
...
ssh                   5335   1.10       996220   0.31   186.7
```

HTTP[11] gets the biggest slice which isn't unusual. There is 1.4MB DNS traffic which verifies my suppositions in chapter 2.4.4.

[10] Time To Live
[11] Hyper Text Transfer Protocol (Protocol used by web servers)

2.4.7 Type of Service

The TOS[12] field in the IP header is probably the best way to prioritise traffic. The field should be set by the application sending the packets to a proper value. Unfortunately this is not done by many applications. Fortunately, it's possible to correct the field on the gateway using iptables[13].

```
# nstats -r i.25-2-2002.raw -o tos
TOS (IPv4)                #      %        bytes      %     avg
-------------------------------------------------------------
Low-Delay:            10092   1.97      6779441   2.00   671.0
Throughput:             357   0.07       457366   0.13  1281.0
Reliability:              2   0.00          128   0.00    64.0
Low-Cost:                 0   0.00            0   0.00     0.0
None:                502577  97.96    332450941  97.87   661.0
```

This looks quite good, at least some applications set a proper TOS.

[12] Type of Service
[13] Packet Filter for Linux

Chapter 3

Configuration

3.1 Introduction

Having a good network analysis like the one worked out in chapter 2 gives a good basis to design and implement a CBQ[1] configuration. CBQ isn't the only way to go, HTB[2] or the Cisco IOS QoS software can do nearly the same. Anyway, CBQ is the preferred way for all examples in this document.

There are lots of articles about traffic shaping using the CBQ idea. Sally Floyd composed a reference page of articles and implementations about CBQ [7].

I'm not going to explain how CBQ works but I will create a CBQ setup using the CBQ implementation in the Linux Kernel 2.1-2.4 written by Alexey Kuznetsov [6].

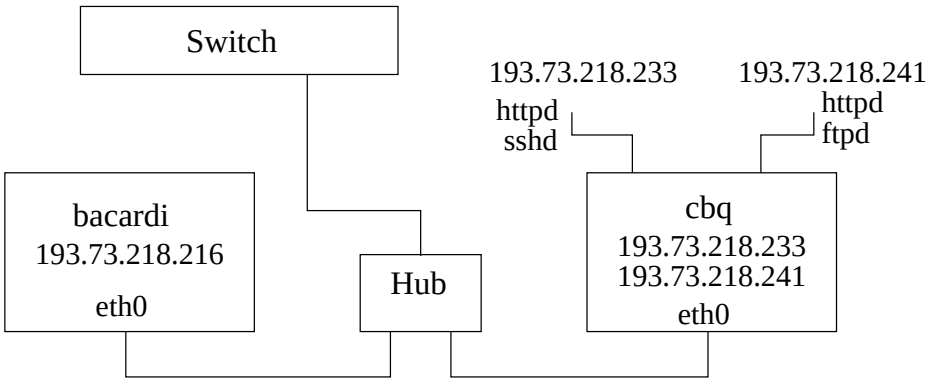
3.2 Requirements

- Two companies are using the same Internet connection with a capacity of 256KBit/s both ways. They pay equal rates, thus the bandwidth needs to be fairly divided.
- Both companies are running a web server which needs guaranteed 60KBit/s to make sure customers don't have to wait. If 60KBit/s aren't enough the bandwidth must be increased without interfering interactive traffic.
- Company 1 has a need for terminal connections (interactive traffic) using the Telnet or SSH[3] protocol. The users of such connections prefer low latency, therefore a guaranteed bandwidth of 30KBit/s is needed.
- Company 2 is running an FTP server which must not affect any other traffic but needs at least 40KBit/s.
- Both companies have needs for DoS[4] protection within the scope of possibility. This means if they can't be stopped they shouldn't be supported.

[1] Class Based Queueing
[2] Hierarchy Token Bucket
[3] Secure Shell
[4] Denial of Service

3.3 Network Setup

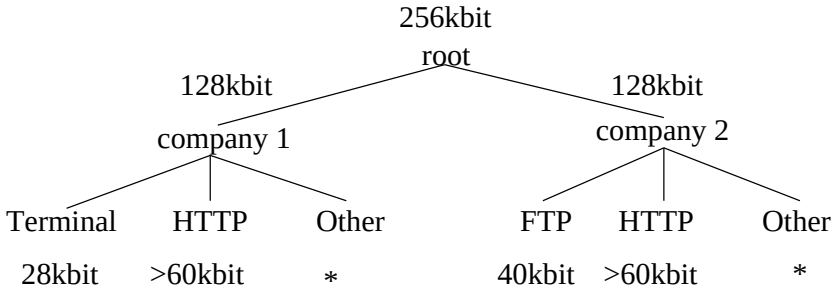
Creating a traffic shaping configuration needs a little test environment. You don't need more than two machines, it would be even possible to do everything on one matching, but it does only complicate things. Simulating two companies is quite easy, just assign two ip addresses to the interface and run the daemons twice and bind them to the specific ip address.



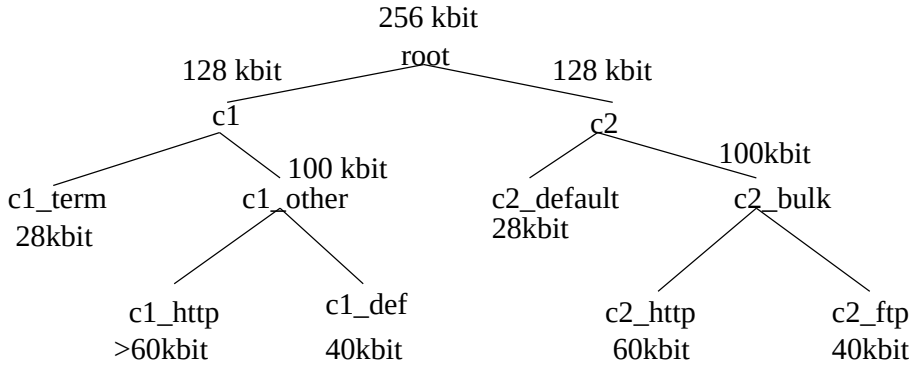
As you can see, we're using the normal LAN with the disadvantage of having some broadcast traffic on the interface.

3.4 Outgoing traffic

3.4.1 Design



This shows the hierarchy in a logic point of view. Each node represents a class where classes on the same level can consume the bandwidth of the upper class. Unfortunately, CBQ isn't that logical, thus we have to change the above hierarchy a bit to fit the needs of CBQ.



I changed the hierarchy to only have two subclasses under each class. It is possible to have more subclasses but I didn't get exact shaping results using more than two classes so far. The whole tree looks quite symmetric, this is true and you'll often see such trees for CBQ setups. Take a look at the node *c1*. You would think, that we're going to shape *c1_term* to 28KBit and leave *c1_other* the rest and shape that in subclasses of c1_other. This is not true, we're actually shaping *c1_other* to 100KBit and leave *c1_term* the rest and this is 28KBit. CBQ classes always need to know about the bandwidth of their parent to be able to rate.

c1_term will always get 28KBit because *c1_other* is bounded to it's 100KBit. Whereas *c1_def* on the third level can completely famish, because the *c1_http* class is allowed to borrow bandwidth.

3.4.2 Initialisation

Let's get to the tricky part. This is all fine on paper, but does it actually work and why did we do all this network analysis before?

The best way to answer your questions is to create the shell script that will load this configuration into the kernel so we can try out what we've just talked about. The complete script can be found in Appendix C.1.

```
#!/bin/sh
outdev=$1
tc qdisc del dev ${outdev} root >/dev/null 2>&1
```

This deletes the *root* handle which contains all the classes and filters we are going to create later. This makes the script rerun-able.

3.4.3 Root

```
tc qdisc add dev ${outdev} root handle 1:0 cbq \
    bandwidth 100mbit \
    avpkt 660 \
    mpu 64
```

This creates the root handle *(device)* assuming you are on a 100mbit Ethernet network. With *avpkt* we tell about the average packet size we expect (we use

the value we found out in chapter 2.4.3). *mpu* means minimum packet usage and is used because a zero-byte packet doesn't consume zero bandwidth. 64 is the *mpu* for Ethernet.

```
tc class add dev ${outdev} parent 1:0 classid 1:1 cbq \
    bandwidth 100mbit \
    rate 256kbit \
    weight 30 \
    allot 1514 \
    prio 7 \
    avpkt 660 \
    bounded
```

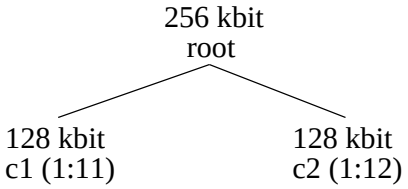
Creates the first CBQ class with classid *1:1 (root)* belonging to class *1:0 (device)*. All traffic passing this class will be shaped to 256kbit. Well this is not really true as they are actually shaped in subclasses but these subclasses think that the link capacity is 256kbit. *weight* is used to give classes with more bandwidth the chance to send more data in one cycle, the value is multiplied with *allot*. *allot x weight* is the number of bytes sent when each time the class gets it's turn. 1514 is calculated as follows: *MTU*[5] *+ 14 byte ethernet header*.

The MTU can be found out using the the *ip* command:

```
ip addr show dev eth0 |head -1 |sed 's/.*mtu \([0-9]*\).*/\1/'
1500
```

prio is the priority, the lower the better. *bounded* means that this class will not try to borrow bandwidth from other classes. This isn't possible anyway as there is only one class on this level.

3.4.4 Split traffic between companies




```
tc class add dev ${outdev} parent 1:1 classid 1:11 cbq \
    bandwidth 256kbit \
    rate 128kbit \
    weight 13 \
    allot 1514 \
    prio 2 \
    avpkt 660 \
    bounded
```

[5] Maximum Transmission Unit

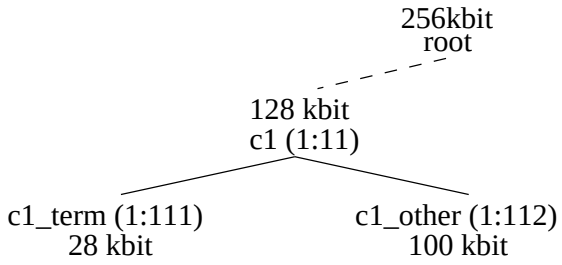
Another CBQ class 1:11 *(c1)* is created as child of 1:1 *(root)* which describes the bandwidth usable by Company 1. All subclasses of 1:11 *(c1)* see a link capacity of 128KBit. A common *weight* value of rate/10 is used. Company 2 needs exactly the same class but with classid 1:12.

```
tc filter add dev ${outdev} parent 1:0 \
    protocol ip \
    prio 1 \
    u32 match ip src 193.73.218.233/32 \
    flowid 1:11
```

Next step is to add a filter which matches all packets coming from network 193.73.218.233/32. Remember, we're shaping the outgoing traffic which means that we have to match the source address. You have to add the same filter again to match traffic from company 2. Packets that don't belong to either of them should be blocked using a packet filter.

What about those numbers (i.e. 1:0,1:1,1:11 etc.)? This are classids that must be unique. I add a new decimal place for every new level and increment if there are multiple classes on the same level. Using this scheme allows you to find out on which level the class is hooked up without reading through the whole setup.

3.4.5 Interactive Traffic for Company 1




```
tc class add dev ${outdev} parent 1:11 classid 1:111 cbq \
    bandwidth 128kbit \
    rate 128kbit \
    weight 3 \
    allot 1514 \
    prio 1 \
    avpkt 190 \
    bounded
```

This is the class called *c1_term* in the class tree. As already mentioned, we are not going to shape in this class, we'll shape in the other class on the same level and all bandwidth that is not used by the other class can be used by this one. This is because we know that this class only contains low traffic packets and will never get full, shaping this class would only increase the latency. The average packet size is much lower than before, I took the value of the SSH protocol from the analysis in chapter 2.4.6.

```
tc filter add dev ${outdev} parent 1:11 \
    protocol ip \
    prio 20 \
    u32 match ip tos 0x8 0xff \
    flowid 1:111

1000 -- minimize delay
0100 -- maximize throughput
0010 -- maximize reliability
0001 -- minimize monetary cost
0000 -- normal service
```

This filter matches on all packets having TOS (DSCP) set to minimize-delay by adding the mask 0xff to the TOS value in the IP header using a bitwise AND. This will result in one of the values in the table above which is taken from [4, RFC1349]. If that value is 0x8 (minimize-delay) the filter matches. Tools like ssh, rsh or telnet set the TOS of important packets to a proper value. You can find more information about the u32 select in one of the listed references in References.

Note, that this filter is only tried to match if the packet is coming from the network used by company 1. This is because the parent of the filter is set to 1:11 *(c1)* which is the top node for company 1.

```
tc filter add dev ${outdev} parent 1:11 \
    protocol ip \
    prio 20 \
    u32 match ip protocol 1 0xff \
    flowid 1:11
```

This filter matches all ICMP packets and puts them into the fast queue. We do this to be able to measure the latency in that queue using the ping or traceroute utility (in ICMP mode).

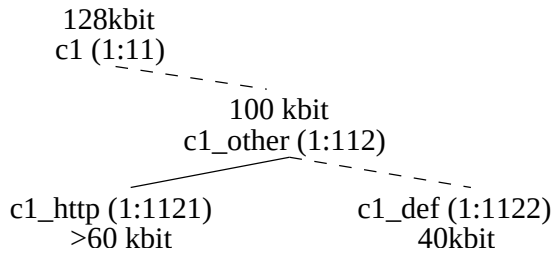
```
tc qdisc add dev ${outdev} parent 1:111 sfq \
    perturb 10
```

Installs a stochastic fairness queue for all traffic in class 1:111 *(c1_term)*. SFQ provides fair queueing for all conversations (or flow) which mostly corresponds to a TCP session or UDP stream. A FIFO would result in almost the same result as this class will hopefully never be full, but if some heavy traffic gets into this class the other connections won't be affected too much.

```
tc class add dev ${outdev} parent 1:11 classid 1:112 cbq \
    bandwidth 128kbit \
    rate 100kbit \
    weight 10 \
    allot 1514 \
    prio 1 \
    avpkt 720 \
    bounded
```

Creates a class on the same level as the class before, this means the bandwidth will be shared between them. Using a *bandwidth* of 128kbit and rate it down to 100kbit will leave 28kbit to the *c1_term* class. I slightly increased the average packet size as there is only bulk traffic in this class.

3.4.6 Guaranteed bandwidth for the web server




```
tc class add dev ${outdev} parent 1:112 classid 1:1121 cbq \
    bandwidth 100kbit \
rate 60kbit \
weight 6 \
    allot 1514 \
    prio 1 \
    avpkt 700 \
    borrow
```

To make sure the web server *(c1_http)* gets the guaranteed 60kbit we need another class which shapes to 60kbit with an average packet size of 700 (avg column of www in chapter 2.4.6). The interesting parameter is *borrow* which allows the class to borrow bandwidth from other classes except they've set the isolated flag.

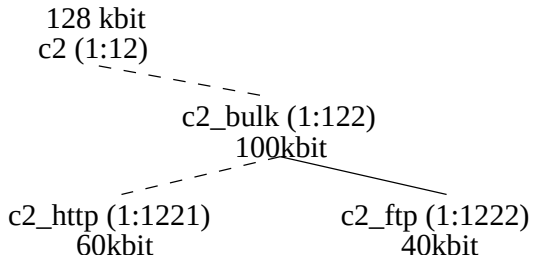
```
tc filter add dev ${outdev} parent 1:11 \
    protocol ip \
    prio 30 \
    u32 match ip sport 80 0xff \
    flowid 1:1121
```

Matching traffic produced by the web server is quite easy, as the source port is always 80 as long as it uses the default port.

```
tc qdisc add dev ${outdev} parent 1:1121 sfq \
    perturb 10
```

SFQ is recommended because there are multiple HTTP sessions at the same time. If a customer is downloading something he shouldn't affect other customers just reading the web site. SFQ leads to good results because the connection can be reused (HTTP/1.1).

3.4.7 Limiting the FTP Server




```
tc class add dev ${outdev} parent 1:122 classid 1:1222 cbq \
    bandwidth 100kbit \
    rate 40kbit \
    weight 4 \
    allot 1514 \
    prio 3 \
    avpkt 800 \
    bounded
```

The *c2_ftp* node needs it's own class shaped to 40kbit.

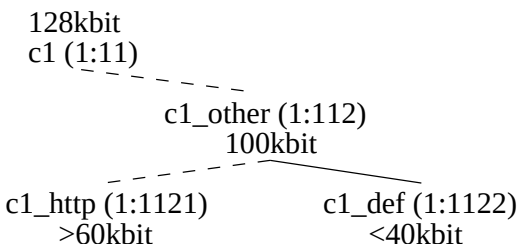
```
tc qdisc add dev ${outdev} parent 1:1222 sfq \
    perturb 10
```

Normal SFQ again as this queue is normally full and should be fair when serving multiple clients.

```
tc filter add dev ${outdev} parent 1:12 \
    protocol ip \7
    prio 20 \
    u32 match ip sport 20 0xff \
    flowid 1:222
```

Creates a filter matching active FTP traffic.

3.4.8 Default Traffic



```
tc class add dev ${outdev} parent 1:112 classid 1:1122 cbq \
    bandwidth 100kbit \
    rate 40kbit \
    weight 4 \
    allot 1514 \
    prio 1 \
    avpkt 600 \
    bounded
```

This creates the default class for company 1 which is responsible for all traffic not matching any other class. The class has to be set bounded to allow the *c1_http* class to borrow bandwidth.

```
tc filter add dev ${outdev} parent 1:11 \
    protocol ip \
    prio 100 \
    u32 match ip tos 0 0 \
    flowid 1:1122
```

All other traffic not matching any other filter should go to the default queue 1:1122. The u32 selector will match in any case as a bitwise AND with 0 in 0 results (A matching rule is mandatory).

3.5 Incoming traffic

3.5.1 Initialisation

```
indev=eth1
```

Put this at the top of the script and replace eth1 with the interface where the packets are coming in.

```
tc qdisc add dev ${indev} handle ffff: ingress
```

Creates an ingress handle which is needed to shape incoming traffic. It's not possible to shape ingress traffic using *CBQ*, we have to use *police* which is also part of the linux kernel.

3.5.2 Split between companies

```
tc filter add dev ${indev} parent ffff: \
    protocol ip \
    prio 50 \
    u32 match ip dst 193.73.218.233/32 \
    police \
        rate 128kbit \
        burst 10k \
        drop \
    flowid :1
```

This filter matches on all packet to ip address 193.73.218.218.233/32 which is the gateway doing NAT[6] for company 1. The *policer* will rate down traffic to 128kbit by *dropping* packets.

3.5.3 Restrict ICMP flood attacks

ICMP flood, was widely used in the past and is still a common DoS attack. It's popular as it's very easy to spoof and produces a reply (when sending a echo-request). It works by sending normal ICMP packets, normally echo-request, with a heavy payload and results in a higher response time of the attacked host.

```
tc filter add dev ${indev} parent ffff: \
    protocol ip \
    prio 40 \
    u32 match ip protocol 1 0xff \
    police \
        rate 10kbit \
        burst 10k \
        drop \
    flowid :1
```

This filter matches on all packets having the protocol field (at offset 9 in the ip header) set to 1 (ICMP)[7]. The filter is checked before the shaping for the companies is done and thus affects all ICMP traffic. The *policer* shapes down all the traffic to 10kbit.

[6] Network Address Translation
[7] You can find a list of protocols in /etc/procotols

Chapter 4

Testing

4.1 Framework

A testing framework is always a good idea as it allows to redo tests. Testing a shaping configuration has a need for a script that can produce the same traffic multiple times. Another script must be made to get and print statistics.

4.1.1 What should be tested

A few questions can be derived out of the requirements in chapter 3.2:

- Does the traffic splitting between the companies work?
- Does the web server always gets its 60kbit?
- What about the latency if the line gets busy?
- What happens if someone starts a ping flood attack?

4.1.2 Traffic

traffic is designed to produce reproducable traffic to test out a shaping configuration. It does take into account to reach as many limits as possible to allow answering the questions above.

```
#!/bin/sh
SESSION=$1
mkdir -p ${SESSION}

ping 193.73.218.233 > ${SESSION}/ping.raw&
```

First of all, a ping is started that runs during the whole test and measures the latency.

```
sleep 10
wget -O /dev/null -o /dev/null http://193.73.218.241/bulk2&
```

10 seconds later, the first download from the web server of company 2 is started. The download will consume 60kbit during the whole test and mustn't fall below that rate.

```
sleep 5
wget -O /dev/null -o /dev/null ftp://193.73.218.241/bulk&
sleep 10
wget -O /dev/null -o /dev/null ftp://193.73.218.241/bulk&
```

After that, two FTP downloads are launched. Each of them will consume 20kbit. Together with the download from the web server, those downloads already fill up all the bulk data capacity of company 2.

```
sleep 10
ping -f 193.73.218.233 -s 2000&
sleep 10
kill `ps a | grep -v grep | grep "ping -f"| cut -d' ' -f1`
```

The ping -f above does simulate a ping flood attack. The process gets killed after 10 seconds. This part should clarify about the latency on heavy load.

```
sleep 10
rsync -rvP root@cbq:bulk .&
```

Last but not least a download using company 1's bandwidth to check if the companies can interfere each other. This download mustnt't affect anything in the download statistics of company 2.

```
sleep 120
killall wget
killall ping
killall rsync
```

We let the whole thing run for additional 120 seconds to get some statistics over a longer period.

4.1.3 tcstat

This is all good, but we need to get some statistics while handling the traffic. tcstat was developed to just do that, it uses *tc -s class* to get statistics about the classes and prints them out. The useful feature is the *-g* option which tells tcstat to output in intervals readable by gnuplot.

```
# tcstat -i eth1 -g | tee cbq.log
```

Collects statistics about classes on eth1 and outputs to cbq.log in a gnuplot readable format:

```
# head cbq.log
0 c2_http 1514 1 0 0 0 0 5411 -1
0 c1_def 0 0 0 0 0 0 0 0
...
```

tcstat is also able to print out in a human readable format:

```
# tcstat -i eth1
Name        Bytes  Packets Dropped  Olimits  AvgIdle Backlog
-------------------------------------------------------------
c2_http   2424326     1607       0     9610  -153350       0
c1_def    1329962     1156       0     7135    -8904       0
c1        1496146     1952       0        0    -8737       0
device    5599468     5029      22        0       52       0
c2_bulk   4083828     2732       0        0       53       0
c1_term    166184      796       0      119    -8737       0
top       5586526     4758       0        0    -4337       0
c2_def       6552       74       0       39    -4742       0
c1_other  1329962     1156       0        0    -3529       0
c2_ftp    1659502     1125       0     8488  -172641       0
c2        4090380     2806       0        0       53       0
c1_http         0        0       0        0       53       0
```

Full documentation can be found in the manual pages in Appendix B.2.

4.1.4 Automated graph generation

A way to compare the throughput of shaping classes is to generate graphs including bit rate and packet rate. A few steps are required to create those graphs. This includes extracting the data of each class out of the log file, which contains data about all classes, and converting the data into the graphics. I use a Makefile[1] with the following variables and targets:

```
DIR=run
LOG=tcstat.log
```

Default values for the variables *DIR* and *LOG*. *DIR* is used as output directory for all commands and must contain the per class log files. *LOG* must point to the log file containing the raw data and is used in *extract*.

```
PS_OUT := $(patsubst gp/%.gp,%.ps,$(wildcard gp/*.gp))
```

Generates a list of all files matching *gp/*.gp*, replaces the extension with *.ps* and adds them to the list *PS_OUT*. A graphic will be generated for each item in *PS_OUT*, thus it's enough to just create the *gp/<class>.gp* file (gnuplot configuration) to generate a graph.

[1] The complete Makefile can be found in Appendix C.2.

```
extract: create_dir
    @for n in `cat $(LOG) | awk '{print $$2;}' | \
    sort | uniq`; do \
        cat $(LOG) | grep " $$n " | awk \
        '{print $$1"\t"$$3"\t"$$4;}' > $(DIR)/$$n.log; \
    done
```

Creates a *$(DIR)/<class>.log* file for each class in *$(LOG)*. The *create_ dir* target creates *$(DIR)* if it doesn't exists.

```
graph: create_dir $(PS_OUT)

%.ps: gp/%.gp
    @cd $(DIR); \
    if [ -f $(patsubst %.ps,%.log,$@) ]; then \
      gnuplot ../$< > $@; \
    fi
```

Runs gnuplot for each gnuplot configuration file if a log file for that class exist.

```
ping:
    @cat $(LOG) | perl -e '$$c=0; while(<>) { \
    chomp; m/time=(.*?)\s/i; if($$1 ne "") { \
    print "$$c\t$$1\n"; $$c++;}}' > $(DIR)/ping.log
```

Converts raw ping output into a gnuplot readable format. It assumes that ping was called in a one second interval.

```
clean:
    @rm -rf $(DIR)/*.ps
```

Removes all graphics.

```
.PHONY: extract clean graph ping
```

Tells make to not look for files called extract, clean or graph, otherwise make would look for those files and compare the modification time.

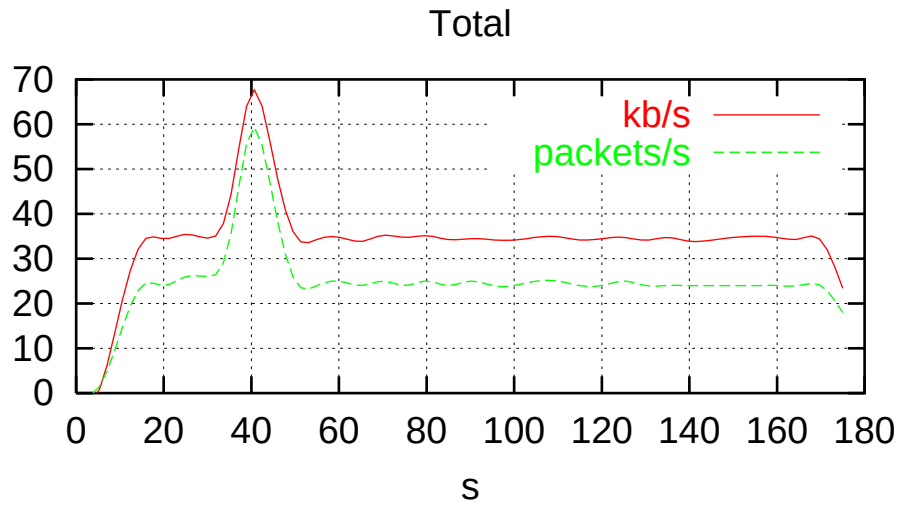
4.2 Test (Shaping Disabled)

4.2.1 Configuration

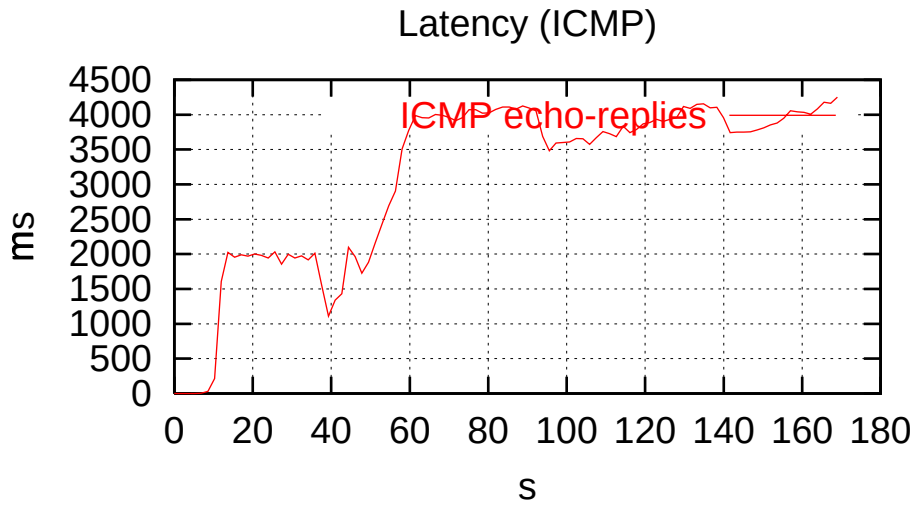
The following CBQ configuration does shape incoming and outgoing traffic to 256kbit/s which is the capacity of the line, thus no real shaping is done. Although, the results aren't the same as you would get over a real 256kbit line.

```
#!/bin/sh

indev=eth1
outdev=eth1

# delete old rules
tc qdisc del dev ${outdev} root >/dev/null 2>&1
tc qdisc del dev ${outdev} ingress >/dev/null 2>&1

# create root handle
tc qdisc add dev ${outdev} root handle 1:0 cbq \
    bandwidth 100mbit \
    avpkt 600

# default class, shape to 256kbit/s
tc class add dev ${outdev} parent 1:0 classid 1:1 cbq \
    bandwidth 100mbit \
    rate 256kbit \
    allot 1514 \
    bounded

# put all packets into the default class
tc filter add dev ${outdev} parent 1:0 \
    protocol ip \
    prio 100 \
    u32 match ip tos 0 0 \
    flowid 1:1

# create ingress handle
tc qdisc add dev ${indev} handle ffff: ingress

# shape all incoming traffic to 256kbit/s
tc filter add dev ${indev} parent ffff: \
    protocol ip \
    prio 50 \
    u32 match ip dst 193.73.218.0/24 \
    police rate 256kbit burst 10k drop \
    flowid :1
```

4.2.2 Results



The average rage is around 35KB/s which is about 280KBit/s, the shaping is working but it isn't very precise when shaping from 100MBit down to 256KBit. The ping flood attack is clearly visible and does exceed the limit.



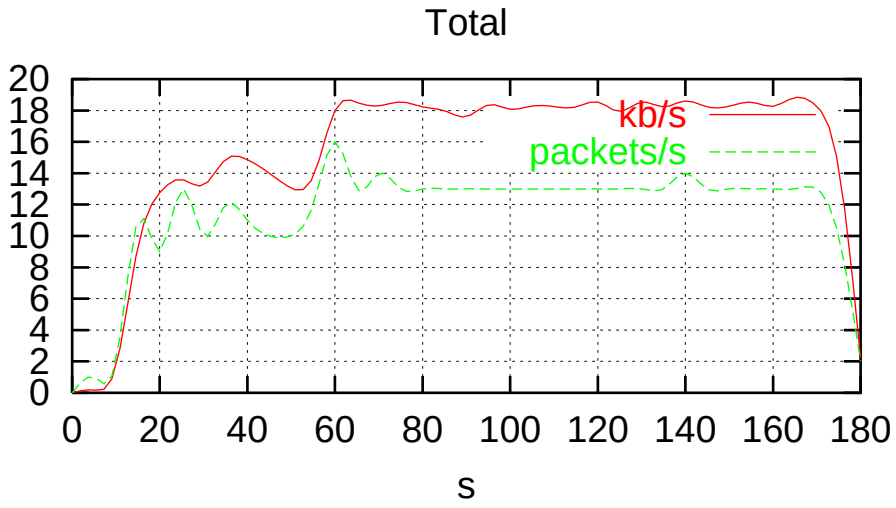
This is quite interesting, even the first HTTP download did increase the ping response time up to two seconds. The ping flood didn't affect much as the line was full anyway, but the rsync did increase it again to 4 seconds which is just unacceptable. This graphic should look at lot better when traffic shaping is enabled.

4.3 Test (Shaping Enabled)

Let's try out the configuration worked out in chapter 3.
What we expect:

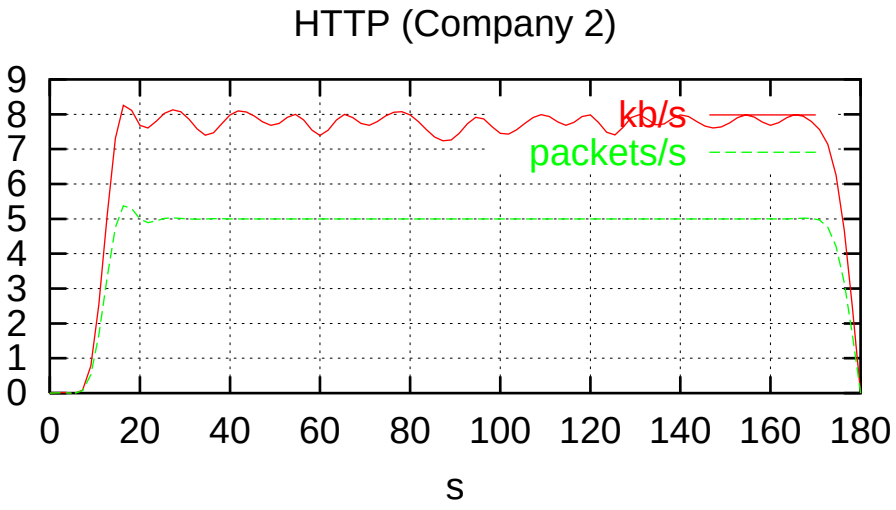
- The latency statistic must look better, the repsponse times should not over-react upon every event.
- The HTTP (Company 2) statistic must go straight up to 60KBit or more and stay there during the whole test.
- The FTP (Company 2) graph must stay around 40KBit, some irregular behaviour is accepted as the HTTP class has a better priority.
- The default (Company 1) statistic can contain any traffic but mustn't exceed 40KBit, especially during the rsync download.
- The Terminal (Company 1) statistic must stay below 30kbit and will probably show a peak during the ping flood that should not exceed 10KBit.

4.3.1 Total



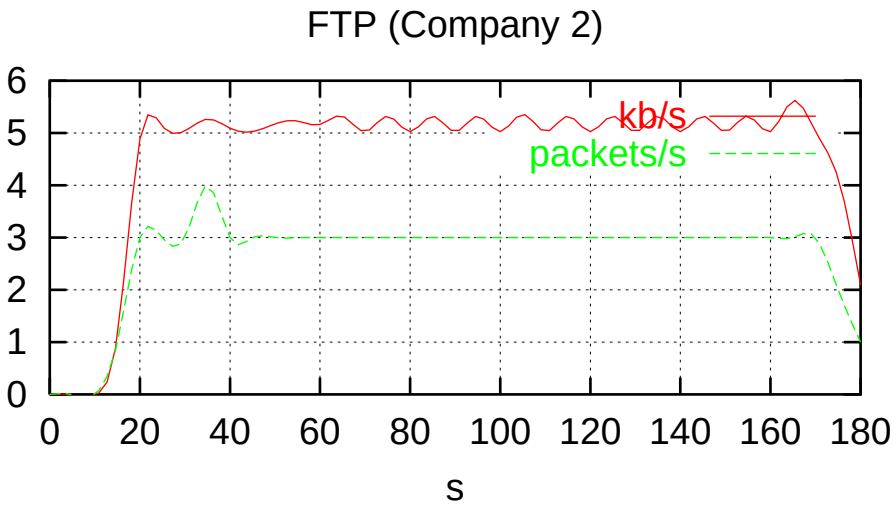
The peak is around 18KB/s, thus the line was never full. You could say that we don't use the line very efficiently, that may be true if you rate the results with the throughput of the line and the runtime of a download. But the goals were to have prioritised traffic and guaranteed bandwidth for the web servers.

4.3.2 HTTP



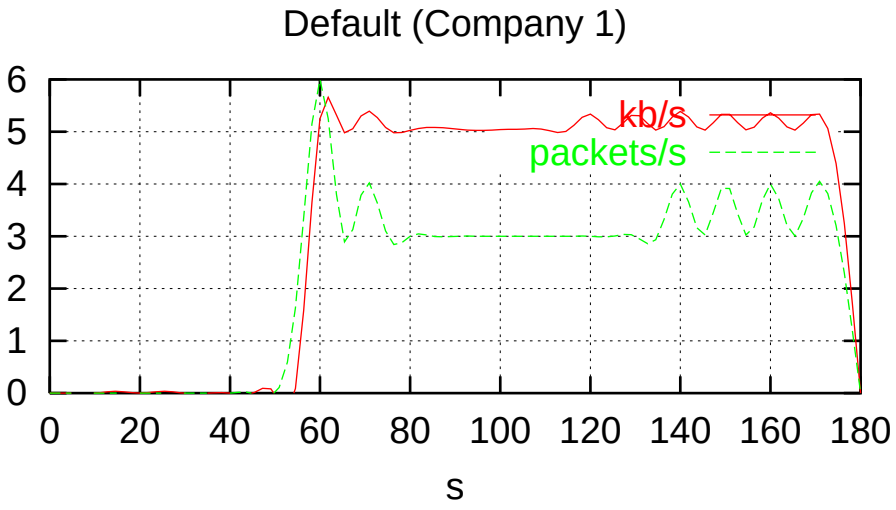
The HTTP download was started with a 10 seconds delay, you can see the rate going up and stay around 8KB/s. The ping flood started with a 30 seconds delay didn't affect anything.

4.3.3 FTP



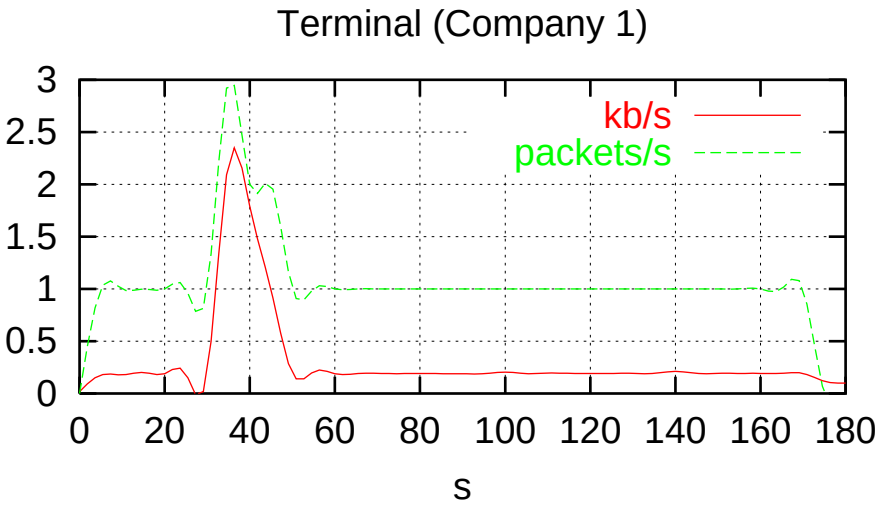
As you can see, the second download, started with a 30 seconds delay, didn't affect the total bandwidth used but it increased the packets per second for a short time before it settled to 3 packets per second.

4.3.4 Default



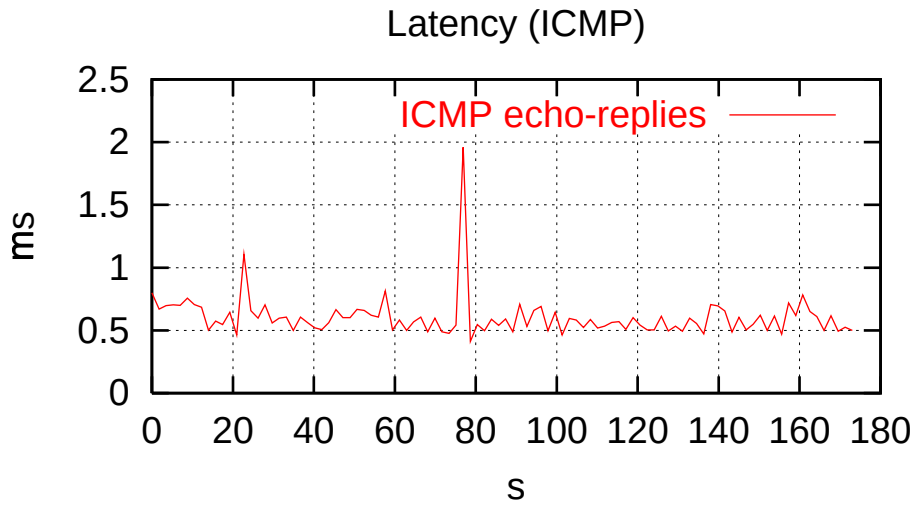
The default class is mainly used by the rsync traffic that is produced with the rsync download started with a 50 seconds delay.

4.3.5 Terminal



This class had a little bit of SSH traffic and the ping flood is clearly visible and it did exceed the defined 10kbit in chapter 3.5.3 a little bit but it keeps it within a limit, thus the protection against ICMP flood DoS attacks is working.

4.3.6 Latency



This looks *very* good, remember that the setup without shaping did result in a latency of 4000 ms. Even the ping flood didn't increase the latency. The peak at 80 seconds looks like a big latency increase but thats very normal as 2ms isn't unusual, could be caused by a physical network problem.

Chapter 5

conf2tc

5.1 Design

The configuration made in chapter 3 is working but is very long winded. conf2tc is designed to simplify this configuration by taking a XML like configuration file and convert it into tc calls. The XML like format is quite easy to read and write but does still allow to easily write a graphical user interface some when later. The tool doesn't replace experiences and knowledge about cbq though.

5.1.1 Tags

device Must be the first tag, all child's will use that device.
Needed: dev=<device>

qdisc Qdisc, Acts as root handle if parent is a device.
Optional: handle=<handle>, type=(cbq/sfq/tbf/etc) (default: cbq), type dependent arguments

class Class, needs another class or a qdisc as parent.
Needed: id=<classid>
Optional: type=(cbq/sfq/tbf/etc) (default: cbq), type dependent arguments

filter Filter, needs a class or qdisc as parent.
Needed: hook=<classid>
Optional: prio=<prio>, u32=<match>

u32 Adds an u32 match to the parent filter and therefore needs a filter as parent.
Needed: match=<match>

police Policer, adds a policer to the parent filter and therefore needs a filter as parent.
Needed: rate=<rate>, burst=<burst>
Optional: action=(drop/reclassify/continue) (default: drop)

5.1.2 Configuration File

Two examples:

```
<device dev="eth1">
    <qdisc handle="1:0" bandwidth="100mbit" avpkt="600"
           mpu="64">
        <class id="1:1" rate="10kbit">
            <qdisc type="sfq" />
            <filter prio="1" hook="1:0">
                <u32 match="ip src 193.73.218.233/32" />
            </filter>
        </class>
    </qdisc>
</device>
```

Simple engress (outgoing traffic) configuration similar to the pieces in chapter 3.4.3 and 3.4.5. The first qdisc will be converted to a root handle because it's parent is the device. Parameters like *avpkt* and *bandwidth* are automatically inherited to the child's of each class. It's possible to have multiple *u32* tags in a filter.

```
<device dev="eth1">
    <qdisc handle="ffff:" type="ingress">
        <filter prio="10" hook="ffff:">
            <u32 match="ip dst 193.73.218.233/32" />
            <police rate="42kbit" burst="10k" />
        </filter>
    </qdisc>
</device>
```

Ingress configuration shaping all traffic to 193.73.218.233 to 42kbit/s.

5.2 Test

5.2.1 Running Example 1

```
# conf2tc conf/ex1.xml
#!/bin/sh
tc qdisc del dev eth1 root > /dev/null 2>&1
tc qdisc del dev eth1 ingress > /dev/null 2>&1
tc qdisc add dev eth1 root handle 1:0 cbq \
    bandwidth 100mbit mpu 64 avpkt 600
tc class add dev eth1 classid 1:1 parent 1:0 cbq \
    weight 1 bandwidth 100mbit rate 10kbit allot bounded
tc qdisc add dev eth1 parent 1:1 sfq
tc filter add dev eth1 parent 1:0 prio 1 \
    protocol ip u32 match ip src 193.73.218.233/32 flowid :1
```

5.2.2 Running Examples 2

```
# conf2tc conf/ex2.xml
#!/bin/sh
tc qdisc del dev eth1 root > /dev/null 2>&1
tc qdisc del dev eth1 ingress > /dev/null 2>&1
tc qdisc add dev eth1 handle ffff: ingress
tc filter add dev eth1 parent ffff: prio 10 protocol ip \
    u32 match ip dst 193.73.218.233/32 \
    police rate 42kbit drop burst 10k  flowid :1
```

5.2.3 Error Handling

```
# conf2tc conf/fa.xml

mismatched tag at line 26, column 22, byte 1142:
                        <filter hook="1:11" ... />
                        <qdisc type="sfq">
                    </class>
====================^

                    <!-- c1_default -->
 at /usr/lib/perl5/XML/Parser.pm line 185
```

Chapter 6

Further Ideas

The way I used to go is a possible one but not the only one. Most of the stuff I tried out could possibly be done better. I will list some of the thoughts I have that could be tried out:

- Shape using a TBF instead of CBQ.
- Do UDP ingress restriction to protect sofware before UDP flood attacks.
- SFQ is currently being used as queueing discipline for all classes. I got better results than with a simple FIFO, although i think this is not the right way to go.
- HTB could simplify things and might increase performance.
- Does this setup work on a connection with higher bandwidth?

Appendix A

Used Tools

iproute2 Utilities (tc, ip) to control the kernel networking code
Alexey Kuznetsov <kuznet@ms2.inr.ac.ru>
ftp://ftp.inr.ac.ru/ip-routing/iproute2-2.4.7-now-ss010824.tar.gz
Version 010824

tcpdump Prints out the headers of packets on a network interface that match the boolean expression.
Van Jacobson, Craig Leres and Steven McCanne
http://www.tcpdump.org/
Version 3.6

nstats Prints statistics about ethernet network traffic. This includes protocol breakdown on several layers, counting packets and bytes per protocol, average packet size per protocol, TOS statistics, and TCP options usage.
Thomas Graf <tgraf@reeler.org>
http://reeler.org/nstats/
Version 0.3.2

tcpstat Reports certain network interface statistics much like vmstat(8) does for system statistics. Statistics include bandwidth being used, number of packets, average packet size, and much more.
Paul Herman <pherman@frenchfries.net>
http://www.frenchfries.net/paul/tcpstat/
Version 1.4

tcstat Prints statistics about all tc classes on a given interface. Can also be used to feed a plot programs like gnuplot.
Thomas Graf
Version 0.2.0

conf2tc Converts a given configuration in XML like format into a set of tc commands which can be executed.
Thomas Graf
Version 0.1

gnuplot Gnuplot is a command-driven interactive function plotting program.
Thomas Williams, Pixar Corporation and Colin Kelley.
http://www.gnuplot.vt.edu/
Version 3.7.1p1

traceroute Print the route packets take to network host.
Van Jacobson
ftp://ftp.ee.lbl.gov/traceroute.tar.Z
Version 1.4a5

ping Send ICMP ECHO_REQUEST packets to network hosts.
Mike Muuss
http://ftp.arl.mil/~mike/ping.html
Version 0.10

make GNU make utility to maintain groups of programs.
Richard Stallman, Roland McGrath and Paul D. Smith
http://www.gnu.org/software/make/
Version 3.79.1

perl Practical Extraction and Report Language.
Larry Wall <larry@wall.org>
http://www.perl.com/
Version 5.6.1

wget GNU Wget is a freely available network utility to retrieve files from the World Wide Web, using HTTP (Hyper Text Transfer Protocol) and FTP (File Transfer Protocol), the two most widely used Internet protocols.
Hrvoje Niksic
Version 1.7

rsync faster, flexible replacement for rcp.
Andrew Tridgell, Paul Mackerras
http://rsync.samba.org/
Version 2.5.2

bash Bash is an sh-compatible command language interpreter that executes commands read from the standard input or from a file.
Brian Fox, Chet Ramey
http://www.gnu.org/software/bash/
Version 2.05a

Appendix B

Manual Pages

B.1 conf2tc

```
conf2tc(8)                                                  conf2tc(8)


NAME
       conf2tc - config to tc command converter

SYNOPSIS
       conf2tc [ -d ] [ -o <file> ] <file>



DESCRIPTION
       conf2tc  converts a given configuration in XML like
       format into a set of tc commands which can be executed.


OPTIONS
       -d     Don't delete old rules.

       -o     Write output to file instead of standard output.


CONFIGURATION FORMAT
      The following tags can be used:

       device Must be the first tag, all childs use that device.
              Needed:  dev=<device>

       qdisc  Qdisc, acts as root handle if parent is a device.
              Optional:  handle=<handle>, type=(cbq|sfq|tbf|etc)
              (default: cbq), type dependent arguments

       class  Class, needs another class or a qdisc as parent.
              Needed:  id=<classid>
              Optional:   type=(cbq|sfq|tbf|etc) (default:cbq),
              type dependent arguments

       filter Filter, needs a class or a qdisc as parent.
              Needed:  hook=<classid>
              Optional:  prio=<prio>, u32=<match>
```

```
       u32    Adds an u32 match to the parent filter  and therefore
              needs a filter as parent.
              Needed:  match=<match>

       police Policer,  adds  a  policer to the parent filter and
              therefore needs a filter as parent.
              Needed:  rate=<rate>, burst=<burst>
              Optional:  action=(drop|reclassify|continue)
              (default: drop)


EXAMPLES
       To convert cbq.xml into tc commands and write them
       to standard output:
              conf2tc cbq.xml

       To convert the configuration from standard input
       and write it to cbq.script:
              conf2tc -o cbq.script -


SEE ALSO
       tc(8), tc-cbq(8), tcstat(8)


AUTHOR
       Thomas Graf <tgraf@reeler.org>


                           March 5, 2002                          1
```

B.2 tcstat

```
tcstat(8)                                                 tcstat(8)


NAME
       tcstat - traffic control statistics

SYNOPSIS
       tcstat -i <dev> [ -hg ] [ -w <interval> ] [ -f <file> ]



DESCRIPTION
       Prints  statistics  about all tc classes on a given
       interface. Can also be used to feed a plot programs
       like  gnuplot.

OPTIONS
       -h     Don't print header.

       -g     Print unformatted interval based output to generate
              graphs with gnuplot(1).

       -i     Selects which interface to get get statistics from.
```

-w Defines the *interval* used between printing the statistics.

-f Selects the *configfile*.

CONFIGURATION FILE

The configuration file is used to map classids to class names and is optional but improves readability of the output.

The format is:

<classid> <classname>
<classid> <classname>
EOF

EXAMPLES

To print statistics about all classes on interface eth0:

tcstat -i eth0

To print statistics in a gnuplot readable format every 10 seconds:

tcstat -i eth0 -g -w 10

FILES

~/.tcstatrc
Default path of configuartion file.

SEE ALSO

tc(8), tc-cbq(8), gnuplot(1)

AUTHOR

Thomas Graf <tgraf@reeler.org>

Appendix C

Listings

C.1 CBQ Configuration

```
#!/bin/sh

indev=eth1
outdev=eth1

echo "Deleting root handle"
tc qdisc del dev ${outdev} root >/dev/null 2>&1
tc qdisc del dev ${outdev} ingress >/dev/null 2>&1

echo "root handle on ${outdev} 1:0"
tc qdisc add dev ${outdev} root handle 1:0 cbq \
    bandwidth 100mbit \
    avpkt 860 \
    mpu 64

echo "cbq class 1:1 [1:0]"
tc class add dev ${outdev} parent 1:0 classid 1:1 cbq \
    bandwidth 100mbit \
    rate 256kbit \
    weight 30 \
    allot 1514 \
    prio 7 \
    avpkt 660 \
    bounded

#-------------------------------------------------------
# Company 1
# ------------------------------------------------------
echo "cbq class 1:11 [1:1] (128kbit)"
tc class add dev ${outdev} parent 1:1 classid 1:11 cbq \
    bandwidth 256kbit \
    rate 128kbit \
    weight 13 \
    allot 1514 \
    prio 6 \
    avpkt 660 \
    bounded

echo "cbq class 1:111 [1:11] (28kbit)"
tc class add dev ${outdev} parent 1:11 classid 1:111 cbq \
```

```
    bandwidth 128kbit \
    rate 128kbit \
    weight 3 \
    allot 1514 \
    prio 1 \
    avpkt 190 \
    bounded

echo "sfq [1:111]"
tc qdisc add dev ${outdev} parent 1:111 sfq \
    perturb 10

echo "cbq class 1:112 [1:11] (100kbit)"
tc class add dev ${outdev} parent 1:11 classid 1:112 cbq \
    bandwidth 128kbit \
    rate 100kbit \
    weight 10 \
    allot 1514 \
    prio 4 \
    avpkt 700 \
    bounded

echo "cbq class 1:1121 [1:112] (60kbit)"
tc class add dev ${outdev} parent 1:112 classid 1:1121 cbq \
    bandwidth 100kbit \
    rate 60kbit \
    weight 6 \
    allot 1514 \
    prio 2 \
    avpkt 700 \
    borrow

echo "sfq [1:1121]"
tc qdisc add dev ${outdev} parent 1:1121 sfq \
    perturb 10

echo "cbq class 1:1122 [1:112] (40kbit)"
tc class add dev ${outdev} parent 1:112 classid 1:1122 cbq \
    bandwidth 100kbit \
    rate 40kbit \
    weight 4 \
    allot 1514 \
    prio 3 \
    avpkt 600 \
    bounded

echo "sfq [1:1122]"
tc qdisc add dev ${outdev} parent 1:1122 sfq \
    perturb 10


# ------------------------------------------------------
# Company 2
# ------------------------------------------------------
echo "cbq class 1:12 [1:1] (128kbit)"
tc class add dev ${outdev} parent 1:1 classid 1:12 cbq \
    bandwidth 256kbit \
    rate 128kbit \
    weight 13 \
    allot 1514 \
    prio 5 \
    avpkt 660 \
```

```
    bounded

echo "cbq class 1:121 [1:12] (28kbit)"
tc class add dev ${outdev} parent 1:12 classid 1:121 cbq \
    bandwidth 128kbit \
    rate 128kbit \
    weight 3 \
    allot 1514 \
    prio 1 \
    avpkt 660 \
    bounded

echo "sfq [1:121]"
tc qdisc add dev ${outdev} parent 1:121 sfq \
    perturb 10

echo "cbq class 1:122 [1:12] (100kbit)"
tc class add dev ${outdev} parent 1:12 classid 1:122 cbq \
    bandwidth 128kbit \
    rate 100kbit \
    weight 10 \
    allot 1514 \
    prio 2 \
    avpkt 660 \
    bounded


echo "cbq class 1:1221 [1:122] (60kbit)"
tc class add dev ${outdev} parent 1:122 classid 1:1221 cbq \
    bandwidth 100kbit \
    rate 60kbit \
    weight 6 \
    allot 1514 \
    prio 2 \
    avpkt 700 \
    bounded

echo "sfq [1:1221]"
tc qdisc add dev ${outdev} parent 1:1221 sfq \
    perturb 10

echo "cbq class 1:1222 [1:122] (40kbit)"
tc class add dev ${outdev} parent 1:122 classid 1:1222 cbq \
    bandwidth 100kbit \
    rate 40kbit \
    weight 4 \
    allot 1514 \
    prio 3 \
    avpkt 800 \
    bounded

echo "sfq [1:1222]"
tc qdisc add dev ${outdev} parent 1:1222 sfq \
    perturb 10

# ------------------------------------------------------------------
# split up the companies
# ------------------------------------------------------------------
tc filter add dev ${outdev} parent 1:0 protocol ip prio 1 u32 \
    match ip src 193.73.218.233/32 flowid 1:11

tc filter add dev ${outdev} parent 1:0 protocol ip prio 1 u32 \
```

```
    match ip src 193.73.218.241/32 flowid 1:12

# ------------------------------------------------------------
# company 1
# ------------------------------------------------------------
# minimize-delay
tc filter add dev ${outdev} parent 1:11 protocol ip prio 20 u32 \
    match ip tos 0x8 0xff flowid 1:111

tc filter add dev ${outdev} parent 1:11 protocol ip prio 20 u32 \
    match ip protocol 1 0xff flowid 1:111

# webserver
tc filter add dev ${outdev} parent 1:11 protocol ip prio 20 u32 \
    match ip sport 80 0xff flowid 1:1121

# default
tc filter add dev ${outdev} parent 1:11 protocol ip prio 100 u32 \
    match ip tos 0 0 flowid 1:1122

# ------------------------------------------------------------
# company 2
# ------------------------------------------------------------

# webserver
tc filter add dev ${outdev} parent 1:12 protocol ip prio 20 u32 \
    match ip sport 80 0xff flowid 1:1221

# ftp
tc filter add dev ${outdev} parent 1:12 protocol ip prio 20 u32 \
    match ip sport 20 0xff flowid 1:1222

# default
tc filter add dev ${outdev} parent 1:12 protocol ip prio 100 u32 \
    match ip tos 0 0 flowid 1:121


# ------------------------------------------------
# Incoming
# ------------------------------------------------

tc qdisc add dev ${indev} handle ffff: ingress

# restrict icmp floods
tc filter add dev ${indev} parent ffff: \
    protocol ip \
    prio 40 \
    u32 match ip protocol 1 0xff \
    police rate 10kbit \
        burst 10k \
        drop \
        flowid :1

# company 1
tc filter add dev ${indev} parent ffff: \
    protocol ip \
    prio 50 \
    u32 match ip dst 193.73.218.233/32 \
    police rate 128kbit \
        burst 10k \
        drop \
        flowid :1
```

```
# company 2
tc filter add dev ${indev} parent ffff: \
    protocol ip \
    prio 50 \
    u32 match ip dst 193.73.218.241/32 \
    police rate 128kbit \
        burst 10k \
        drop \
        flowid :1
```

C.2 Makefile (Chapter 4.1.4)

```
DIR=run
LOG=tcstat.log
PS_OUT := $(patsubst gp/%.gp,%.ps,$(wildcard gp/*.gp))

all: extract graph

create_dir:
    @mkdir -p $(DIR)

graph: create_dir $(PS_OUT)

clean:
    @rm -rf $(DIR)/*.ps

extract: create_dir
    @for n in `cat $(LOG) | awk '{print $$2;}' | sort | uniq`; do \
    cat $(LOG) | grep " $$n " | awk '{print $$1"\t"$$3"\t"$$4;}' > \
    $(DIR)/$$n.log; \
    done

ping:
    @cat $(LOG) | perl -e '$$c=0; while(<>) { \
    chomp; m/time=(.*?)\s/i; if($$1 ne "") { \
    print "$$c\t$$1\n"; $$c++;}}' > $(DIR)/ping.log

%.ps: gp/%.gp
    @cd $(DIR); \
    if [ -f $(patsubst %.ps,%.log,$@) ]; then \
      gnuplot ../$< > $@; \
    fi

.PHONY: extract clean graph ping
```

C.3 Gnuplot Configuration Example (Chapter 4.1.4)

```
set term postscript color
set data style lines
set yrange [ 0 : ]
set grid
set size 0.5,0.4
set title "HTTP (Company 2)"
set xlabel "s"
plot "c2_http.log" using 1:($2/1024) smooth csplines title "kb/s" \
    ,"c2_http.log" using 1:($3) smooth csplines title "packets/s"
```

Bibliography

[1] Linux Networking-concepts HOWTO
http://netfilter.samba.org/unreliable-guides/networking-concepts-HOWTO/index.html

[2] Linux Advanced Routing and Traffic Control
http://lartc.org/

[3] RFC791: Internet Protocol
http://rfc.net/0791.html

[4] RFC1349: Type of Service in the Internet Protocol Suite
http://rfc.net/1349.html

[5] RFC1393: Traceroute Using an IP Option
http://rfc.net/1393.html

[6] Alexey Kuznetsov's ip routing tools.
ftp://ftp.inr.ac.ru/ip-routing/

[7] References on CBQ (Class-Based Queueing) by Sally Floyd
http://www.icir.org/floyd/cbq.html

[8] Linux - Advanced Networking Overview
http://qos.ittc.ukans.edu/howto/

[9] RFC792: Internet Control Message Protocol
http://rfc.net/rfc0792.html

[10] RFC3168: Transmission Control Protocol
http://rfc.net/rfc3168.html

[11] RFC768: User Datagram Protocol
http://rfc.net/rfc0768.html