# Introduction

These example setups were given by ebtables users, and they are very much appreciated. If you have a configuration involving ebtables which you would like to share, please write a little story about it and send it to the webmaster so that it

can be added here.

# Real-life examples

## Choose one of the examples below



You can also view all examples on one page.

### Linux brouting, MAC snat and MAC dnat all in one

This setup and explanation was given by *Enrico Ansaloni*, who got things working together with *Alessandro Eusebi*.

#### Short description

Two 802.1Q VLANs, a HP 4000 M switch and a Linux bridge with iptables and ebtables.
The HP switch is used for VLAN switching. The Linux bridge is used for bridging specific frames and for IP routing.
Obviously good filtering is required to prevent duplicated traffic.

#### Enrico's story

Why I use the Linux bridge: I have to bridge the 2 VLANs to let DEC LAT traffic only go through, while IP traffic has to go under normal routing decisions: this is done with ebtables.

The bridge has 2 IP addresses, one for each VLAN so that routing will work, in conjuction with iptables eventually to do some traffic shaping and internal control (allowed tcp ports from inside to outside etc.)

**First try: assign the 2 IP addresses to the bridge device (br0)**

The problem with the HP switch is that it doesn't allow the same MAC address in 2 different ports which is what happens when I connect the bridge which has 2 cables - it connects into 2 ports - and has 1 MAC address; this occurs even if the ports belong to different VLANs: to put it in another way, the HP VLAN implementation doesn't fully isolate VLANs but checks for MAC address consistency in the whole switch, regardless of VLAN settings... instead, the Cisco Catalyst series does that right but it costs 3 times more and my customer won't pay that much...

**Second try: use ebtables MAC snat to give the two bridge IP addresses different MAC source addresses**

so I decided I could try with MAC natting, and I set up ebtables like this:

```
ebtables -t nat -A POSTROUTING -o $INSIDE_IF_NAME -s $DMZ_IF_MAC \
-j snat --to-source $INSIDE_IF_MAC
ebtables -t nat -A PREROUTING -i $INSIDE_IF_NAME -d $INSIDE_IF_MAC \
-j dnat --to-destination $DMZ_IF_MAC
```

As you can see, I'm tying to fool the switch into thinking the bridge has a different MAC address for each interface: the rule is correct, as I can see by capturing traffic with ethereal, but there's a problem: ebtables works at layer 2 only, thus correctly natting MAC address in the layer 2 ethernet frame; the switch now accepts the natted packets but the ARP packets are at layer 3 and they keep the original MAC address as from the linux kernel network stack, so the client's reply is wrong and never goes through...

**Third try: use brouting + MAC snat + MAC dnat**

As stated in the examples section on the ebtables hp, I started with this:

```
**********************************************************
Bridge table: broute
```

```
Bridge chain: BROUTE
Policy: ACCEPT
nr. of entries: 4
1. -p IPV4 -i eth0 -j DROP , count = 47959
2. -p IPV4 -i eth1 -j DROP , count = 47
3. -p ARP -i eth0 -j DROP , count = 371
4. -p ARP -i eth1 -j DROP , count = 141
********************************************************
```

But this is not enough... With this rule, IP routing and ARP stuff is perfectly working for both networks (VLANs) and I can control IP stuff with iptables, but bridge isn't working yet... that's because of the duplicate MAC in different VLANs problem of the HP switch: when I use bridging, the same MAC address of the bridged client appears on two ports, one for each VLAN, and the switch automatically deactivates the first port! But I solved this issue with ebtables MAC NAT, like this:

```
********************************************************
Bridge table: nat

Bridge chain: PREROUTING
Policy: ACCEPT
nr. of entries: 2
1. -d 10:50:da:e7:18:51 -i eth1 -j dnat --to-dst 0:50:da:e7:18:51 --dnat-target ACCEPT, count = 1260
2. -d 10:10:a4:9b:30:d -i eth0 -j dnat --to-dst 0:10:a4:9b:30:d --dnat-target ACCEPT, count = 1252

Bridge chain: OUTPUT
Policy: ACCEPT
nr. of entries: 0

Bridge chain: POSTROUTING
Policy: ACCEPT
nr. of entries: 2
1. -s 0:50:da:e7:18:51 -o eth1 -j snat --to-src 10:50:da:e7:18:51 --snat-target ACCEPT, count = 1362
2. -s 0:10:a4:9b:30:d -o eth0 -j snat --to-src 10:10:a4:9b:30:d --snat-target ACCEPT, count = 1346
********************************************************
```

The MAC addresses you see are two network client, one for each VLAN. When a client passes the bridge, I have to change his MAC (i change the first 00: with 10:) in order to make the switch happy.
I did a small script also, so you can specify a list of MAC addresses for each VLAN. Here's the script:

```
**********************************************************
#!/bin/bash
################################################################
# EBTables test script
################################################################
# Binaries

EBTABLES=/usr/local/sbin/ebtables
################################################################
# Interface names

INSIDE_IF_NAME=eth0
DMZ_IF_NAME=eth1
BRIDGE_IF_NAME=br0
################################################################
# Bridge mac address list

INSIDE_IF_MAC="00:04:76:14:74:99"
DMZ_IF_MAC="00:01:03:e2:e9:4c"
################################################################
# Client mac address list

LAN_CLIENT_MACS="00:50:DA:E7:18:51 00:50:DA:E7:F1:A0 00:10:A4:9B:E8:21"
DMZ_CLIENT_MACS="00:10:A4:9B:30:0D 00:01:03:E2:12:9C 00:50:DA:E7:11:2B"
NEW_PREFIX="10:"
################################################################
# Set default policy
#
$EBTABLES -P INPUT ACCEPT
$EBTABLES -P OUTPUT ACCEPT
$EBTABLES -P FORWARD ACCEPT
```

```
# clear existing tables
$EBTABLES -F
$EBTABLES -t nat -F
$EBTABLES -t broute -F

#################################################################
# BRoute

$EBTABLES -t broute -A BROUTE -p ipv4 -i $INSIDE_IF_NAME -j DROP
$EBTABLES -t broute -A BROUTE -p ipv4 -i $DMZ_IF_NAME -j DROP
$EBTABLES -t broute -A BROUTE -p arp -i $INSIDE_IF_NAME -j DROP
$EBTABLES -t broute -A BROUTE -p arp -i $DMZ_IF_NAME -j DROP

#################################################################
# Bridged clients

for MAC in $LAN_CLIENT_MACS; do
        NEW_MAC="${NEW_PREFIX}`echo ${MAC} | cut -f2- -d':'`"
        $EBTABLES -t nat -A POSTROUTING -o $DMZ_IF_NAME -s $MAC -j snat --to-source $NEW_MAC
        $EBTABLES -t nat -A PREROUTING -i $DMZ_IF_NAME -d $NEW_MAC -j dnat --to-destination $MAC
done

for MAC in $DMZ_CLIENT_MACS; do
        NEW_MAC="${NEW_PREFIX}`echo ${MAC} | cut -f2- -d':'`"
        $EBTABLES -t nat -A POSTROUTING -o $INSIDE_IF_NAME -s $MAC -j snat --to-source $NEW_MAC
        $EBTABLES -t nat -A PREROUTING -i $INSIDE_IF_NAME -d $NEW_MAC -j dnat --to-destination $MAC
done

#################################################################
# END
#################################################################
```

I hope this can be useful to you or someone else... If you're so unlucky to have to deal with HP 4000 switches and their VLAN implementation :)

# Filtering AppleTalk using ebtables

This setup and description was given by *Ashok Aiyar*. The original website where he posted this setup is no longer available but the website is archived here. The contents were edited to bring the original text, which dates from the Linux 2.4 days, up-to-date.

## Why filter AppleTalk?

There are many situations where it is appropriate to filter AppleTalk. Here's one of them. We tunnel/route AppleTalk between five networks using netatalk. There are very similarly named Tektronix Phaser printers in two of these networks, and often print jobs intended for one are unintentionally sent to the other. We would prefer for each of these Phasers to be visible only in the network in which it is located, and not in all five networks. Unlike CAP, netatalk does not support filtering. Therefore, on this page I describe one method to add external filters to netatalk, on the basis of the MAC address associated with an AppleTalk object or node.

There are pros and cons to filtering on the basis of MAC addresses. They have the advantage of being more robust because AppleTalk node numbers can change with every reboot, while the MAC address will not. They have the disadvantage of not being fine-grained; MAC-based filtering will block all the services associated with the filtered AppleTalk node. In general, AppleTalk nodes in our networks are associated with a single service.

## Iptables versus Ebtables

The Linux netfilter code supports filtering of IPV4, IPV6 and DECnet packets on the basis of MAC addresses. However such filters do not apply to any other type of ethernet frame. So, an `iptables` rule such as:

```
iptables -I INPUT -m mac --mac-source TE:KP:HA:SE:R8:60 -j DROP
```

results in only IPV4, IPV6 and DECnet packets from that source address being dropped. More to the point, DDP and AARP packets from the same source address are not dropped. Ebtables appeared to be perfectly suited to filter Ethernet frames on the basis of MAC address as well as ethernet protocol type. However, it only supports bridge interfaces, and not regular Ethernet interfaces. Bart De Schuymer, the author of ebtables brought to my attention that a Linux bridge

interface can have just a single Ethernet interface. Thanks to Bart's generous advice, a working Ethernet filtering setup is described below.

### Setting up Ebtables

To setup a bridge with a single interface, first create the bridge interface (br0). Then add the relevant ethernet interface to the bridge. Finally, assign to the bridge the IP address previously assigned to the ethernet interface. The commands to do this are detailed below:

```
brctl addbr br0             # create bridge interface
brctl stp br0 off           # disable spanning tree protocol on br0
brctl addif br0 eth0        # add eth0 to br0
ifconfig br0 aaa.bbb.ccc.ddd netmask 255.255.255.0 broadcast aaa.bbb.ccc.255
ifconfig eth0 0.0.0.0
route add -net aaa.bbb.ccc.0 netmask 255.255.255.0 br0
route add default gw aaa.bbb.ccc.1 netmask 0.0.0.0 metric 1 br0
```

Now network traffic will be routed through the br0 interface rather than the underlying eth0. `Atalkd` can be started to route AppleTalk between br0 and any other desired interfaces. Note that `atalkd.conf` has to be modified so that the reference to eth0 is replaced with br0. For example, the `atalkd.conf` for PC1 shown on my AppleTalk tunneling page is modified to:

```
br0  -seed -phase 2 -net 2253  -addr 2253.102  -zone "Microbio-Immun"
tap0 -seed -phase 2 -net 60000 -addr 60000.253 -zone "Microbio-Immun"
tap1 -seed -phase 2 -net 60001 -addr 60001.253 -zone "Microbio-Immun"
tap2 -seed -phase 2 -net 60002 -addr 60002.253 -zone "Microbio-Immun"
tap3 -seed -phase 2 -net 60003 -addr 60003.253 -zone "Microbio-Immun"
```

Verify that AppleTalk routing is working, and then proceed to set up Ethernet filters using `ebtables`. For this the MAC addresses of the AppleTalk nodes that are not to be routed must be known. One simple method of discovering the MAC address is to send the AppleTalk object a few `aecho` packets, and then read the MAC address from `/proc/net/aarp`. A sample `ebtables` filter is shown below:

```
ebtables -P INPUT ACCEPT
ebtables -P FORWARD ACCEPT
ebtables -P OUTPUT ACCEPT
ebtables -A INPUT -p LENGTH -s TE:KP:HA:SE:R8:60 -j DROP
```

Currently, `ebtables` doesn't support filtering of 802.2 and 802.3 packets such as the DDP and AARP packets used by AppleTalk. However all such packets can be dropped on the basis of the length field – if I understand Bart de Schuymer's explanation correctly. Therefore in the example above, all ethernet 802.2, 802.3 packets from the node with the MAC address TE:KP:HA:SE:R8:60 are dropped. This includes AppleTalk packets, but not IPV4, and ARP packets. This node is left visible in the network in which it is located, but not in any networks to which AppleTalk is routed.

**Acknowledgements, Final Comments and Useful Links:**

Bart de Schuymer's advice and patient explanations are greatly appreciated. In my experience `atalkd` bound to the br0 interface is as stable as `atalkd` bound to the eth0 interface. In addition the MAC address based filters described here work well for their intended purpose. While this works, there is a performance penalty associated with receiving all IP traffic through br0 and not eth0. This is because traffic destined for the bridge is queued twice (once more than normal) – that's a lot of overhead. The ebtables `broute` table can be used to circumvent this and directly route the traffic entering the bridge port. This way it will be queued only once, eliminating the performance penalty. In the example above:

```
brctl addbr br0
brctl stp br0 off
brctl addif br0 eth0
ifconfig br0 0.0.0.0
ifconfig eth0 a.b.c.d netmask 255.255.255.0 broadcast a.b.c.255
```

The following two ebtables BROUTE table rules should be used:

```
ebtables -t broute -A BROUTING -p IPv4 -i eth0 --ip-dst a.b.c.d -j DROP
ebtables -t broute -A BROUTING -p ARP -i eth0 -d MAC_of_eth0 -j DROP
```

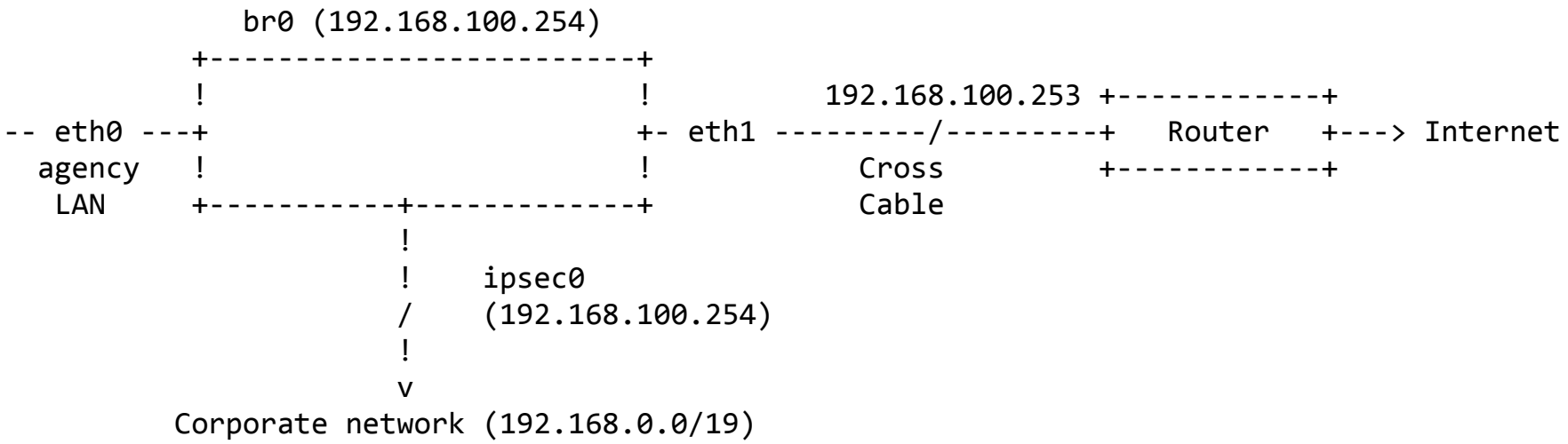
Atalkd should still be bound to br0, thus allowing AppleTalk to be filtered by ebtables. As best as I can tell this configuration eliminates the performance penalty on IP traffic throughput. Because we tunnel AppleTalk through IP, this

configuration removes any throughput penalties in using a bridge interface and ebtables to route AppleTalk.

## Transparent routing with Freeswan1

This setup was given by *Dominique Blas*.

### Bridge/router configuration




- eth0 and eth1 without any address.
- br0 is built upon eth0 and eth1.
- STP is off.
- ipsec0 defaults to br0's address.
- Default gateway is, of course, 192.168.100.253.
- When tunnel is established, route towards 192.168.0.0/19 is created by freeswan using interface ipsec0. Classical.

### Stations configuration

- Addressing scheme : 192.168.100.0/24.
- Default gateway is 192.168.100.253.

### Now the challenge

- Automatically inject packets, for which the destination address is within the corporate network addressing, to ipsec0.
- Keep the other packets going through eth1.
- That is: ping www.yahoo.com (Internet access through eth1) must respond, ping 192.168.0.33 (intranet access going through ipsec0) must answer.

One can ask why being so complicated. It's enough to configure default gateway (or a list a subnet in the routing table) on the stations to point towards the br0 address and the router will do its job routing toward intranet or towards Internet. Perfectly exact.

Yes but imagine you have several dozens of such local stations without DHCP to manage their network configuration and you are thousands of miles away. You can simply send the machine, ask someone to plug it in the router and in the hub with the correct cables you've provided.
That's all.

### A solution

So

```
ebtables -t nat -A PREROUTING -i eth0 -p ipv4 --ip-dst 192.168.0.0/19 \
-j dnat --to-dst  $MAC_ADDR_OF_IPSEC0 --dnat-target ACCEPT
```

actually works and is enough for the challenge.

A notice on `$MAC_ADDR_Of_IPSEC0`.
Since the rule is set before ipsec is launched the mac address of ipsec0 is not set at this time. This doesn't matter since ipsec0's mac address will be the same as that of the outbound interface that is equal to eth1's mac address.

Hence this rule:

```
ebtables -t nat -A PREROUTING -i eth0 -p ipv4 --ip-dst 192.168.0.0/19 \
-j dnat --to-dst $MAC_ADDR_OF_ETH1 --dnat-target  ACCEPT
```

Of course that's only the first step. One can then take into account a few other things:

- protecting eth1 at the ip level (via iptables) or at the mac level (anti spoofing via ebtables or iptables),
- protecting intranet from devices attempting to access the tunnel from eth1,
- be sure such mac address is associated with such ip address without using `arp -s`, using ebtables,
- etc, etc.

## By-pass dns client bug on user's inexpensive routers

This example was given by *Mike Ireton*.

**ebtables just saved our cookies big time. Here's how it did it:**

We're a wireless isp and our wireless subscribers all have inexpensive soho routers with fixed ip addresses. We recently had to change out the hardware at one of our main towers (software upgrades, faster cpu, etc etc) and as a result of doing this, we discovered that numerous subscribers all of a sudden could not resolve dns. Everyone continued to have ping and tcp connectivity, but dns wasn't resolving for them. Rebooting their routers would solve the problem, but there's a lot of subscribers who depend on this site and we coulnd't possibly call them all. After looking over packet dumps, it appears that most of these cheap soho routers have a very subtle bug which was responsable for the problem - their dns requests were still bearing the old mac address of the router, while test pings to them bore the new! (In effect, the embedded dns client in the router was not re-arping for the gateway and instead using an outdated cache). This is across several brands of routers too. So the solution was to install ebtables mac address nat'ing so that received frames destined for the old mac addresses would be changed to the new address, thus solving the problem site wide.

## Rate shaping

This example was given by *Dave Stahr* (dave<at>stewireless.com).

**Goal**

I'm running Fedora Core 3 with `bridge-utils-0.9.6-2` and `ebtables-v2.0.6` to rate shape bandwidth for our wireless subscribers, based on their MAC address. This bridge sits between our core wireless link and the switch connected to our servers and internet gateway. A perl script runs out of cron that connects to our mysql database to create the traffic queues automatically. I'm not including that here, but any programmer could figure out how to write their own suited to their own database, etc, or just write the config by hand for smaller networks. We're running on a little 400mhz PentiumII with 128mb RAM and it doesn't even sweat. We have around 500 customers, all rate shaped in their own individual queues. This could hypothetically support up to 20,000 individual traffic queues.

This is a very simple setup, but in trying to figure out how to do it, I found many unanswered posts on many web forums by people trying to do this.

**Bridge configuration**

This gives us the ability to log in to the bridge, as well as give the bridge the ability to connect out to our mysql database server, etc:

```
------------- ifcfg-br0 -------------
DEVICE=br0
ONBOOT=no
BOOTPROTO=static
IPADDR=192.168.111.11
NETMASK=255.255.255.0

----------- bridge_up.sh --------------------
#!/bin/bash

ifdown eth0
ifdown eth1

ifconfig eth0 0.0.0.0 up
ifconfig eth1 0.0.0.0 up

brctl addbr br0
```

```
brctl addif br0 eth0
brctl addif br0 eth1

ifconfig br0 up

----------- bridge_down.sh --------------------
#!/bin/bash

ifdown eth0
ifdown eth1
ifconfig br0 down
brctl delbr br0
```

### The rate shaping part

We're using `tc` to do the deed. This is my first attempt at this, so I may be doing some things wrong, especially with the `tc` commands - BUT IT WORKS - so I figure, I'll fix it later. You can use `ebtables -L --Lc` to see your customer's usage. I dump this out hourly, adding the `-Z` option to zero the counters out, then have a perl script parse that output and dump it into a mysql table where I can make better use of it.

```
--------------------------- rateshape ----------------------
#!/bin/bash
#
#  All Rates are in Kbits, so in order to gets Bytes divide by 8
#  e.g. 25Kbps == 3.125KB/s
#
TC=/sbin/tc
EBTABLES=/sbin/ebtables   # Location of ebtables

cd /usr/local/bridge

tc_start() {
    $TC qdisc add dev eth0 root handle 1:0 cbq bandwidth 100Mbit avpkt 1000 mpu 64
    $TC qdisc add dev eth1 root handle 1:0 cbq bandwidth 100Mbit avpkt 1000 mpu 64
```

```
#Customer A
#Two MACs: 00:0D:BD:A4:E1:C8 and 00:20:78:B0:25:7D
#256kbps download speed
${TC} class add dev eth0 parent 1:0 classid 1:1 cbq rate 256KBit allot 1514 prio 1 avpkt 1000 bounded
${TC} filter add dev eth0 parent 1:0 protocol ip handle 1 fw flowid 1:1
${EBTABLES} -A FORWARD -d 00:0D:BD:A4:E1:C8 -j mark --set-mark 1 --mark-target ACCEPT
${EBTABLES} -A FORWARD -d 00:20:78:B0:25:7D -j mark --set-mark 1 --mark-target ACCEPT
#128kbps upload speed
${TC} class add dev eth1 parent 1:0 classid 1:1 cbq rate 128KBit allot 1514 prio 1 avpkt 1000 bounded
${TC} filter add dev eth1 parent 1:0 protocol ip handle 1 fw flowid 1:1
${EBTABLES} -A FORWARD -s 00:0D:BD:A4:E1:C8 -j mark --set-mark 1 --mark-target ACCEPT
${EBTABLES} -A FORWARD -s 00:20:78:B0:25:7D -j mark --set-mark 1 --mark-target ACCEPT

#Customer B
#MAC Address: 00:0D:BD:A4:D6:54
#800kbps download speed
${TC} class add dev eth0 parent 1:0 classid 1:2 cbq rate 800KBit allot 1514 prio 1 avpkt 1000 bounded
${TC} filter add dev eth0 parent 1:0 protocol ip handle 2 fw flowid 1:2
${EBTABLES} -A FORWARD -d 00:0D:BD:A4:D6:54 -j mark --set-mark 2 --mark-target ACCEPT
#64kbps upload speed
${TC} class add dev eth1 parent 1:0 classid 1:2 cbq rate 64KBit allot 1514 prio 1 avpkt 1000 bounded
${TC} filter add dev eth1 parent 1:0 protocol ip handle 2 fw flowid 1:2
${EBTABLES} -A FORWARD -s 00:0D:BD:A4:D6:54 -j mark --set-mark 2 --mark-target ACCEPT

#Customer C
#MAC Address: 00:0A:5E:22:D1:A3
#do not rate shape!
${EBTABLES} -A FORWARD -s 00:0A:5E:22:D1:A3 -j ACCEPT
${EBTABLES} -A FORWARD -d 00:0A:5E:22:D1:A3 -j ACCEPT

#Block anything we didn't specify above.
${EBTABLES} -A FORWARD -j DROP --log

#<my config has over 500 customers and over 1100 MAC addresses>
```

```
#Just keep incrementing the classid, handle, flowid, and mark values for each customer's
#individual speed queues.
}

tc_stop() {

    ./save_and_reset_counters

    ${EBTABLES} -F

    $TC qdisc del dev eth0 root
    $TC qdisc del dev eth1 root
}

tc_restart() {

    tc_stop
    sleep 1
    tc_start

}

tc_show() {

    echo ""
    echo "eth0"
    $TC qdisc show dev eth0
    $TC class show dev eth0
    $TC filter show dev eth0
    echo ""
    echo "eth1"
    $TC qdisc show dev eth1
    $TC class show dev eth1
}
```

```
tc_stop() {

    ./save_and_reset_counters

    ${EBTABLES} -F

    $TC qdisc del dev eth0 root
    $TC qdisc del dev eth1 root
}

tc_restart() {

    tc_stop
    sleep 1
    tc_start

}

tc_show() {

    echo ""
    echo "eth0"
    $TC qdisc show dev eth0
    $TC class show dev eth0
    $TC filter show dev eth0
    echo ""
    echo "eth1"
    $TC qdisc show dev eth1
    $TC class show dev eth1
    $TC filter show dev eth1

}
case "$1" in

  start)
```

```
    echo -n "Starting bandwidth shaping: "
    tc_start
    echo "done"
    ;;

  stop)

    echo -n "Stopping bandwidth shaping: "
    tc_stop
    echo "done"
    ;;

  restart)

    echo -n "Restarting bandwidth shaping: "
    tc_restart
    echo "done"
    ;;

  show)

    tc_show
    ;;

  *)

    echo "Usage: rateshape {start|stop|restart|show}"
    ;;

esac
```