HTB Linux queuing discipline manual - user guide

Martin Devera aka devik (devik@cdi.cz)
Manual: devik and Don Cohen
PDF Version: Eric Paynter (eric@arcticbears.com)
Last updated: 5.5.2002

New text is in red color. Coloring is removed on new text after 3 months. Currently they depicts HTB3 changes

- 1. Introduction
- 2. Link sharing
- 3. Sharing hierarchy
- 4. Rate ceiling
- 5. Burst
- 6. Priorizing bandwidth share
- 7. Understanding statistics
- 8. Making, debugging and sending error reports

1. Introduction

HTB is meant as a more understandable, intuitive and faster replacement for the CBQ qdisc in Linux. Both CBQ and HTB help you to control the use of the outbound bandwidth on a given link. Both allow you to use one physical link to simulate several slower links and to send different kinds of traffic on different simulated links. In both cases, you have to specify how to divide the physical link into simulated links and how to decide which simulated link to use for a given packet to be sent.

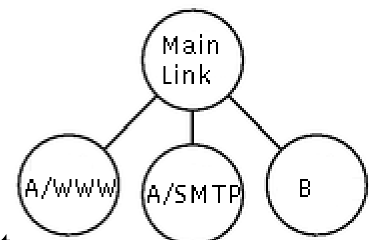
This document shows you how to use HTB. Most sections have examples, charts (with measured data) and discussion of particular problems.

This release of HTB should be also much more scalable. See comparison at HTB home page.

Please read: tc tool (not only HTB) uses shortcuts to denote units of rate. **kbps** means kilo**bytes** and **kbit** means **kilobits** ! This is the most FAQ about tc in linux.

2. Link sharing

Problem: We have two customers, A and B, both connected to the internet via eth0. We want to allocate 60 kbps to B and 40 kbps to A. Next we want to subdivide A's bandwidth 30kbps for WWW and 10kbps for everything else. Any unused bandwidth can be used by any class which needs it (in proportion of its allocated share).



HTB ensures that **the amount of service provided to each class is at least the minimum of the amount it requests and the amount assigned to it**. When a class requests less than the amount assigned, the remaining (excess) bandwidth is distributed to other classes which request service.

Also see document about HTB internals - it describes goal above in greater details.

Note: In the literature this is called "borrowing" the excess bandwidth. We use that term below to conform with the literature. We mention, however, that this seems like a bad term since there is no obligation to repay the resource that was "borrowed".

The different kinds of traffic above are represented by classes in HTB. The simplest approach is shown in the picture at the right.
Let's see what commands to use:

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

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

Note: In general (not just for HTB but for all qdiscs and classes in tc), handles are written x:y where x is an integer identifying a qdisc and y is an integer identifying a class belonging to that qdisc. The handle for a qdisc must have zero for its y value and the handle for a class must have a non-zero value for its y value. The "1:" above is treated as "1:0".

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

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

*Note: Sometimes people ask me why they have to repeat **dev eth0** when they have already used **handle** or **parent**. The reason is that handles are local to an interface, e.g., eth0 and eth1 could each have classes with handle 1:1.*

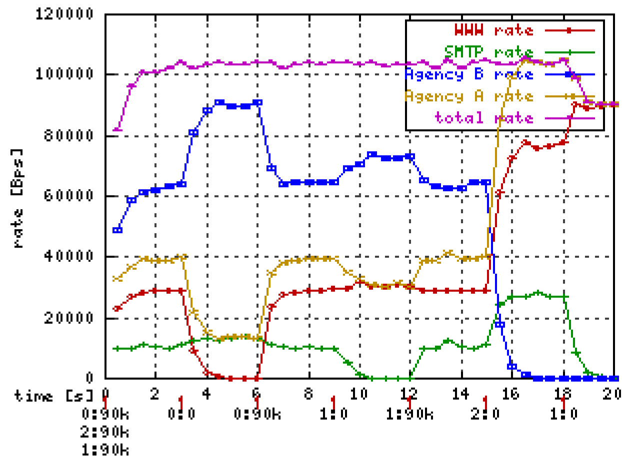
We also have to describe which packets belong in which class. This is really not related to the HTB qdisc. See the tc filter documentation for details. The commands will look something like this:

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

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

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

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



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

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

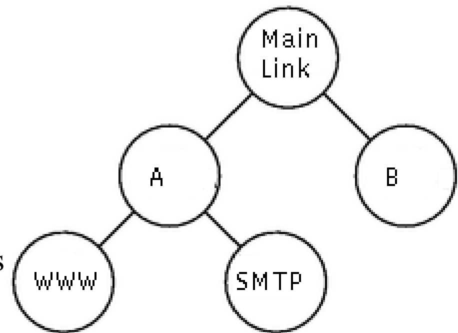
That's all the commands we need. Let's see what happens if we send packets of each class at 90kbps and then stop sending packets of one class at a time. Along the bottom of the graph are annotations like "0:90k". The horizontal position at the center of the label (in this case near the 9, also marked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

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

It might be good time to touch concept of **quantums** now. In fact when more classes want to borrow bandwidth they are each given some number of bytes before serving other competing class. This number is called quantum. You should see that if several classes are competing for parent's bandwidth then they get it in proportion of their quantums. It is important to know that for precise operation quantums need to be as small as possible and larger than MTU.
Normaly you don't need to specify quantums manualy as HTB chooses precomputed values. It computes classe's quantum (when you add or change it) as its rate divided by **r2q** global parameter. Its default value is 10 and because typical MTU is 1500 the default is good for rates from 15 kBps (120 kbit). For smaller minimal rates specify r2q 1 when creating qdisc - it is good from 12 kbit which should be enough. If you will need you can specify quantum manualy when adding or changing the class. You can avoid warnings in log if precomputed value would be bad. When you specify quantum on command line the r2q is ignored for that class.

This might seem like a good solution if A and B were not different customers. However, if A is paying for 40kbps then he would probably prefer his unused WWW bandwidth to go to his own other service rather than to B. This requirement is represented in HTB by the class hierarchy.

3. Sharing hierarchy



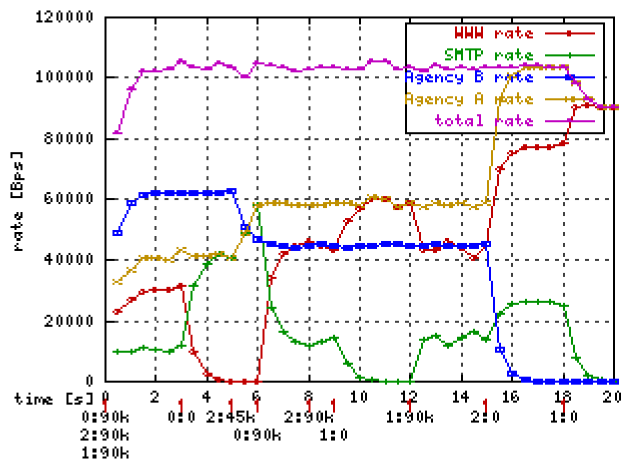
The problem from the previous chapter is solved by the class hierarchy in this picture. Customer A is now explicitly represented by its own class. Recall from above that **the amount of service provided to each class is at least the minimum of the amount it requests and the amount assigned to it**. This applies to htb classes that are not parents of other htb classes. We call these leaf classes. For htb classes that are parents of other htb classes, which we call interior classes, the rule is that **the amount of service is at least the minumum of the amount assigned to it and the sum of the amount requested by its children**. In this case we assign 40kbps to customer A. That means that if A requests less than the allocated rate for WWW, the excess will be used for A's other traffic (if there is demand for it), at least until the sum is 40kbps.

Notes: Packet classification rules can assign to inner nodes too. Then you have to attach other filter list to inner node. Finally you should reach leaf or special 1:0 class. The rate supplied for a parent should be the sum of the rates of its children.

The commands are now as follows:

```
tc class add dev eth0 parent 1: classid 1:1 htb rate 100kbps ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:2 htb rate 40kbps ceil 100kbps
tc class add dev eth0 parent 1:2 classid 1:10 htb rate 30kbps ceil 100kbps
tc class add dev eth0 parent 1:2 classid 1:11 htb rate 10kbps ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:12 htb rate 60kbps ceil 100kbps
```

We now turn to the graph showing the results of the hierarchical solution. When A's WWW traffic stops, its assigned bandwidth is reallocated to A's other traffic so that A's total bandwidth is still the assigned 40kbps.
If A were to request less than 40kbs in total then the excess would be given to B.

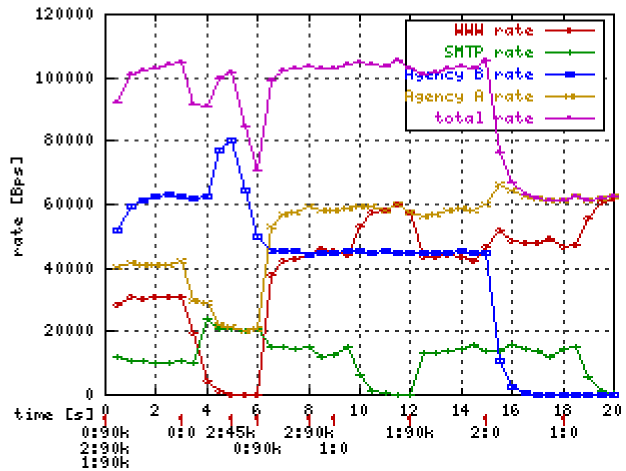


4. Rate ceiling

The **ceil** argument specifies the maximum bandwidth that a class can use. This limits how much bandwidth that class can borrow. The default ceil is the same as the rate. (That's why we had to specify it in the examples above to show borrowing.) We now change the **ceil 100kbps** for classes 1:2 (A) and 1:11 (A's other) from the previous chapter to **ceil 60kbps** and **ceil 20kbps**.

The graph at right differs from the previous one at time 3 (when WWW traffic stops) because A/other is limited to 20kbps. Therefore customer A gets only 20kbps in total and the unused 20kbps is allocated to B.
The second difference is at time 15 when B stops. Without the ceil, all of its bandwidth was given to A, but now A is only allowed to use 60kbps, so the remaining 40kbps goes unused.

This feature should be useful for ISPs because they probably want to limit the amount of service a given customer gets even when other customers are not requesting service. (ISPs probably want customers to pay more money for better service.) Note that root classes are not allowed to borrow, so there's really no point in specifying a ceil for them.



Notes: The ceil for a class should always be at least as high as the rate. Also, the ceil for a class should always be at least as high as the ceil of any of its children.

5. Burst

Networking hardware can only send one packet at a time and only at a hardware dependent rate. Link sharing software can only use this ability to approximate the effects of multiple links running at different (lower) speeds. Therefore the rate and ceil are not really instantaneous measures but averages over the time that it takes to send many packets. What really happens is that the traffic from one class is sent a few packets at a time at the maximum speed and then other classes are served for a while. The **burst** and **cburst** parameters control the amount of data that can be sent at the maximum (hardware) speed without trying to serve another class.

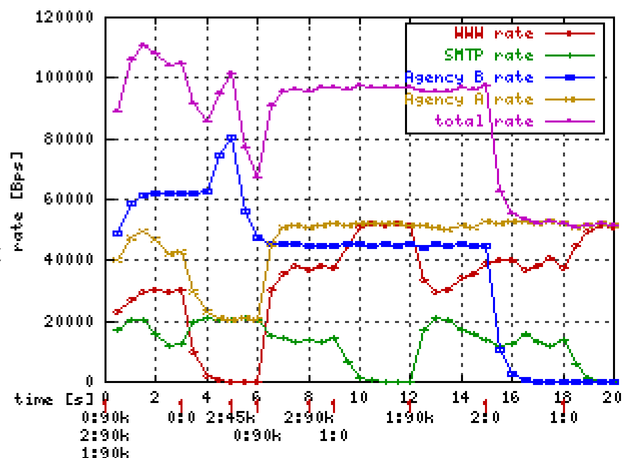
If **cburst** is smaller (ideally one packet size) it shapes bursts to not exceed **ceil** rate in the same way as TBF's peakrate does.

When you set **burst** for parent class smaller than for some child then you should expect the parent class to get stuck sometimes (because child will drain more than parent can handle). HTB will remember these negative bursts up to 1 minute.

You can ask **why I want bursts**. Well it is cheap and simple way how to improve response times on congested link. For example www traffic is bursty. You ask for page, get it in burst and then read it. During that idle period burst will "charge" again.

Note: The burst and cburst of a class should always be at least as high as that of any of it children.

On graph you can see case from previous chapter where I changed burst for red and yellow (agency A) class to 20kb but cburst remained default (cca 2 kb). Green hill is at time 13 due to burst setting on SMTP class. A class. It has underlimit since time 9 and accumulated 20 kb of burst. The hill is high up to 20 kbps (limited by ceil because it has cburst near packet size).
Clever reader can think why there is not red and yellow hill at time 7. It is because yellow is already at ceil limit so it has no space for furtner bursts.
There is at least one unwanted artifact - magenta

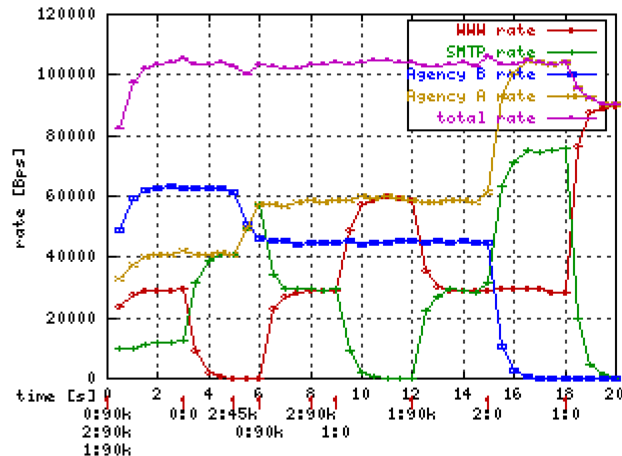


crater at time 4. It is because I intentionaly "forgot" to add burst to root link (1:1) class. It remembered hill from time 1 and when at time 4 blue class wanted to borrow yellow's rate it denied it and compensated itself.

Limitation: when you operate with high rates on computer with low resolution timer you need some minimal **burst** and **cburst** to be set for all classes. Timer resolution on i386 systems is 10ms and 1ms on Alphas. The minimal burst can be computed as max_rate*timer_resolution. So that for 10Mbit on plain i386 you needs burst 12kb.

If you set too small burst you will encounter smaller rate than you set. Latest tc tool will compute and set the smallest possible burst when it is not specified.

6. Priorizing bandwidth share



Priorizing traffic has two sides. First it affects how the excess bandwidth is distributed among siblings. Up to now we have seen that excess bandwidth was distibuted according to rate ratios. Now I used basic configuration from chapter 3 (hierarchy without ceiling and bursts) and changed priority of all classes to 1 except SMTP (green) which I set to 0 (higher). From sharing view you see that the class got all the excess bandwidth. The rule is that **classes with higher priority are offered excess bandwidth first**. But rules about guaranted **rate** and **ceil** are still met.

There is also second face of problem. It is total delay of packet. It is relatively hard to measure on ethernet which is too fast (delay is so neligible). But there is simple help. We can add simple HTB with one class rate limiting to less then 100 kbps and add second HTB (the one we are measuring) as child. Then we can simulate slower link with larger delays.
For simplicity sake I use simple two class scenario:

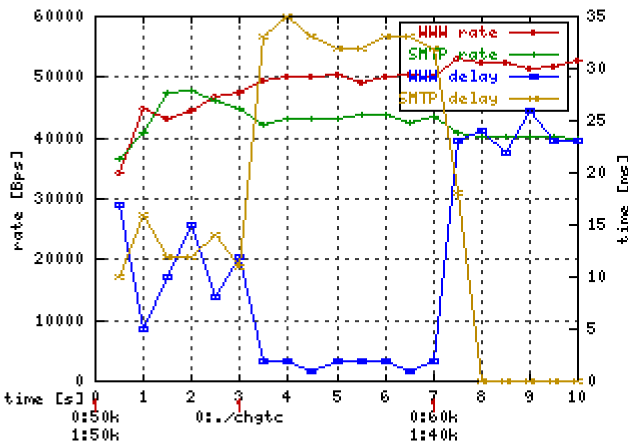
```
# qdisc for delay simulation
tc qdisc add dev eth0 root handle 100: htb
tc class add dev eth0 parent 100: classid 100:1 htb rate 90kbps

# real measured qdisc
tc qdisc add dev eth0 parent 100:1 handle 1: htb
AC="tc class add dev eth0 parent"
$AC 1: classid 1:1 htb rate 100kbps
$AC 1:2 classid 1:10 htb rate 50kbps ceil 100kbps prio 1
$AC 1:2 classid 1:11 htb rate 50kbps ceil 100kbps prio 1
tc qdisc add dev eth0 parent 1:10 handle 20: pfifo limit 2
tc qdisc add dev eth0 parent 1:11 handle 21: pfifo limit 2
```

Note: HTB as child of another HTB is NOT the same as class under another class within the same HTB. It is because when class in HTB can send it will send as soon as hardware equipment can. So that delay of underlimit class is limited only by equipment and not by ancestors.
In HTB under HTB case the outer HTB simulates new hardware equipment with all consequences (larger delay)

Simulator is set to generate 50 kbps for both classes and at time 3s it executes command:

```
tc class change dev eth0 parent 1:2 classid 1:10 htb \
 rate 50kbps ceil 100kbps burst 2k prio 0
```



As you see the delay of WWW class dropped nearly to the zero while SMTP's delay increased. When you priorize to get better delay it always makes other class delays worse.
Later (time 7s) the simulator starts to generate WWW at 60 kbps and SMTP at 40 kbps. There you can observe next interesting behaviour. When class is overlimit (WWW) then HTB priorizes underlimit part of bandwidth first.

What class should you priorize ? Generaly those classes where you really need low delays. The example could be video or audio traffic (and you will really need to use correct **rate** here to prevent traffic to kill other ones) or interactive (telnet, SSH) traffic which is bursty in nature and will not negatively affect other flows.
Common trick is to priorize ICMP to get nice ping delays even on fully utilized links (but from technical point of view it is not what you want when measuring connectivity).

7. Understanding statistics

The **tc** tool allows you to gather statistics of queuing disciplines in Linux. Unfortunately statistic results are not explained by authors so that you often can't use them. Here I try to help you to understand HTB's stats.
First whole HTB stats. The snippet bellow is taken during simulation from chapter 3.

```
# tc -s -d qdisc show dev eth0
 qdisc pfifo 22: limit 5p
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

 qdisc pfifo 21: limit 5p
 Sent 2891500 bytes 5783 pkts (dropped 820, overlimits 0)

 qdisc pfifo 20: limit 5p
 Sent 1760000 bytes 3520 pkts (dropped 3320, overlimits 0)

 qdisc htb 1: r2q 10 default 1 direct_packets_stat 0
 Sent 4651500 bytes 9303 pkts (dropped 4140, overlimits 34251)
```

First three disciplines are HTB's children. Let's ignore them as PFIFO stats are self explanatory.
overlimits tells you how many times the discipline delayed a packet. *direct_packets_stat* tells you how many packets was sent thru direct queue. Other stats are sefl explanatory. Let's look at class' stats:

```
tc -s -d class show dev eth0
class htb 1:1 root prio 0 rate 800Kbit ceil 800Kbit burst 2Kb/8 mpu 0b
    cburst 2Kb/8 mpu 0b quantum 10240 level 3
 Sent 5914000 bytes 11828 pkts (dropped 0, overlimits 0)
 rate 70196bps 141pps
 lended: 6872 borrowed: 0 giants: 0
```

```
class htb 1:2 parent 1:1 prio 0 rate 320Kbit ceil 4000Kbit burst 2Kb/8 mpu 0b
    cburst 2Kb/8 mpu 0b quantum 4096 level 2
 Sent 5914000 bytes 11828 pkts (dropped 0, overlimits 0)
 rate 70196bps 141pps
 lended: 1017 borrowed: 6872 giants: 0

class htb 1:10 parent 1:2 leaf 20: prio 1 rate 224Kbit ceil 800Kbit burst 2Kb/8 mpu
0b
    cburst 2Kb/8 mpu 0b quantum 2867 level 0
 Sent 2269000 bytes 4538 pkts (dropped 4400, overlimits 36358)
 rate 14635bps 29pps
 lended: 2939 borrowed: 1599 giants: 0
```

I deleted 1:11 and 1:12 class to make output shorter. As you see there are parameters we set. Also there are *level* and DRR *quantum* informations.
overlimits shows how many times class was asked to send packet but he can't due to rate/ceil constraints (currently counted for leaves only).
rate, pps tells you actual (10 sec averaged) rate going thru class. It is the same rate as used by gating.
lended is # of packets donated by this class (from its **rate**) and *borrowed* are packets for whose we borrowed from parent. Lends are always computed class-local while borrows are transitive (when 1:10 borrows from 1:2 which in turn borrows from 1:1 both 1:10 and 1:2 borrow counters are incremented).
giants is number of packets larger than mtu set in tc command. HTB will work with these but rates will not be accurate at all. Add mtu to your tc (defaults to 1600 bytes).

8. Making, debugging and sending error reports

If you have kernel 2.4.20 or newer you don't need to patch it - all is in vanilla tarball. The only thing you need is **tc** tool. Download HTB 3.6 tarball and use tc from it.

You have to patch to make it work with older kernels. Download kernel source and use **patch -p1 -i htb3_2.X.X.diff** to apply the patch. Then use **make menuconfig;make bzImage** as before. Don't forget to enable QoS and HTB.
Also you will have to use patched **tc** tool. The patch is also in downloads or you can download precompiled binary.

If you think that you found an error I will appreciate error report. For oopses I need ksymoops output. For weird qdisc behaviour add parameter **debug 3333333** to your **tc qdisc add .... htb**. It will log many megabytes to syslog facility kern level debug. You will probably want to add line like:
kern.debug -/var/log/debug
to your /etc/syslog.conf. Then bzip and send me the log via email (up to 10MB after bzipping) along with description of problem and its time.