Traffic Control using tcng and HTB HOWTO

Version 1.0.1

Martin A. Brown

linux−ip.net
Network Administration

<martin@linux-ip.net>

April 2006

Revision History

Revision 1.0.1	2006−10−28	Revised by: MAB
Updating contact information		
Revision 1.0	2003−04−16	Revised by: tab
Initial Release, reviewed by LDP		
Revision 0.5	2002−04−01	Revised by: MAB
submit to tldp, rename/retitle with HOWTO		
Revision 0.4	2002−03−31	Revised by: MAB
new example, bucket crash course		
Revision 0.3	2002−03−16	Revised by: MAB
corrections and notes from Jacob Teplitsky, raptor and Joshua Heling		
Revision 0.2	2002−03−15	Revised by: MAB
links, cleanup, publish		
Revision 0.1	2002−03−14	Revised by: MAB
initial revision		

Table of Contents

1. Introduction

This is a brief tutorial on using **tcng** (Traffic Control Next Generation) with HTB (Hierarchical Token Bucket) to perform traffic shaping on a Linux machine.

This tutorial is intended for systems administrators who have

- AT LEAST, a basic understanding of traffic control
- EITHER the capability to compile iproute2 and tcng from source

  OR the capability of building RPMS from provided SRPMs
- EITHER a modular kernel with support for htb and dsmark

  OR capability to compile a kernel with support for htb and dsmark

This article is neither comprehensive nor authoritative. The author solicits positive and negative feedback at `<martin@linux-ip.net>`. Corrections, additions, and further examples are always welcome.

1.1. What is traffic control and how does it work?

Traffic control is the term given to the entire packet queuing subsystem in a network or network device. Traffic control consists of several distinct operations. Classifying is a mechanism by which to identify packets and place them in individual flows or classes. Policing is a mechanism by which one limits the number of packets or bytes in a stream matching a particular classification. Scheduling is the decision−making process by which packets are ordered and re−ordered for transmission. Shaping is the process by which packets are delayed and transmitted to produce an even and predictable flow rate.

These many characteristics of a traffic control system can be combined in complex ways to reserve bandwidth for a particular flow (or application) or to limit the amount of bandwidth available to a particular flow or application.

One of the key concepts of traffic control is the concept of tokens. A policing or shaping implementation needs to calculate the number of bytes or packets which have passed at what rate. Each packet or byte (depending on the implementation), corresponds to a token, and the policing or shaping implementation will only transmit or pass the packet if it has a token available. A common metaphorical container in which an implementation keeps its token is the bucket. In short, a bucket represents the both the number of tokens which can be used instantaneously (the size of the bucket), and the rate at which the tokens are replenished (how fast the bucket gets refilled).

See Section 1.2 for an example of buckets in a linux traffic control system.

Under linux, traffic control has historically been a complex endeavor. The **tc** command line tool provides an interface to the kernel structures which perform the shaping, scheduling, policing and classifying. The syntax of this command is, however, arcane. The **tcng** project provides a much friendlier interface to the human by layering a language on top of the powerful **tc** command line tool. By writing traffic control configurations in **tcng** they become easily maintainable, less arcane, and importantly also more portable.

1.2. What is htb?

Hierarchichal Token Bucket is a classful qdisc written by Martin Devera with a simpler set of configuration parameters than CBQ. There is a great deal of documentation on the author's site and also on Stef Coene's website about HTB and its uses. Below is a very brief sketch of the HTB system.

Conceptually, HTB is an arbitrary number of token buckets arranged in a hierarchy (yes, you probably could have figured that out without my sentence). Let's consider the simplest scenario. The primary egress queuing discipline on any device is known as the `root` qdisc.

The `root` qdisc will contain one class (complex scenarios could have multiple classes attached to the `root` qdisc). This single HTB class will be set with two parameters, a `rate` and a `ceil`. These values should be the same for the top−level class, and will represent the total available bandwidth on the link.

In HTB, `rate` means the guaranteed bandwidth available for a given class and `ceil` is short for ceiling, which indicates the maximum bandwidth that class is allowed to consume. Any bandwidth used between `rate` and `ceil` is borrowed from a parent class, hence the suggestion that `rate` and `ceil` be the same in the top−level class.

A number of children classes can be made under this class, each of which can be allocated some amount of the available bandwidth from the parent class. In these children classes, the `rate` and `ceil` parameter values need not be the same as suggested for the parent class. This allows you to reserve a specified amount of bandwidth to a particular class. It also allows HTB to calculate the ratio of distribution of available bandwidth to the ratios of the classes themselves. This should be more apparent in the examples below.

Hierarchical Token Bucket implements a classful queuing mechanism for the linux traffic control system, and provides `rate` and `ceil` to allow the user to control the absolute bandwidth to particular classes of traffic as well as indicate the ratio of distribution of bandwidth when extra bandwidth becomes available (up to `ceil`).

Keep in mind when choosing the bandwidth for your top−level class that traffic shaping only helps if you are the bottleneck between your LAN and the Internet. Typically, this is the case in home and office network environments, where an entire LAN is serviced by a DSL or T1 connection.

In practice, this means that you should probably set the bandwidth for your top−level class to your available bandwidth minus a fraction of that bandwidth.

1.3. What is tcng?

Traffic Control Next Generation (tcng) is a project by Werner Almesberger to provide a powerful, abstract, and uniform language in which to describe traffic control structures. The **tcc** parser in the **tcng** distribution transforms tcng the language into a number of output formats. By default, **tcc** will read a file (specified as an argument or as STDIN) and print to STDOUT the series of **tc** commands (see **iproute2** below) required to create the desired traffic control structure in the kernel.

Consult the parameter reference for **tcng** to see the supported queuing disciplines. Jacob Teplitsky, active on the LARTC mailing list and a contributor to the tcng project, wrote the htb support for **tcng**.

The **tcc** tool can produce a number of different types of output, but this document will only consider the conventional and default output. Consult the TCNG manual for more detailed information about the use of

tcng.

The **tcsim** tool is a traffic control simulator which accepts tcng configuration files and reads a control language to simulate the behaviour of a kernel sending and receiving packets with the specified control structures. Although **tcsim** is a significant portion of the **tcng** project, **tcsim** will not be covered here at all.

---

2. Requirements

There are a few requirements in order for the kernel to support HTB and DSMARK, tc to support HTB and DSMARK, and tcng itself.

Specifically, support for HTB in the kernel and tc is absolutely required in order for this tutorial to be remotely useful (refer to the title if htere is any doubt in your mind). DSMARK support is, strictly speaking, optional, although some examples (class selection path, in particular, but maybe others) may not operate without dsmark support.

2.1. kernel requirements

The kernel requirements are very easy to meet. Kernel 2.4.20 and newer include support for HTB and dsmark, so simply be certain that these options are turned on in the QoS/Fair Queuing portion of your kernel configuration. For a brief summary of the options to select in kernel configuration, visit the DiffServ project kernel configuration notes.

For kernels older than 2.4.20, the following tarball containing a patch should be applied to your 2.4.17 or newer kernel tree.

2.2. tc requirements

The **tc** command is a part of the **iproute2** utility suite. For general documentation on **iproute2**, see http://linux−ip.net/ and the **iproute2** manual. The software itself is available directly from Alexey Kuznetsov'z FTP archive but commonly also via packages supplied with your linux distribution. If your distribution can make use of RPMS, you can download this SRPM and compile it on your own system.

If you need to compile **iproute2** yourself, use the patch to **tc** from this tarball at Martin Devera's HTB site in order to provide support for HTB in **tc**.

Your **tc** will also need to support dsmark, the diffserv marking mechanism. Fortunately, this is a simple change to the `Config` file from the **iproute2** source package. Simply change `TC_CONFIG_DIFFSERV=n` to `TC_CONFIG_DIFFSERV=y` and recompile.

The SRPM creates a **tc** binary with support for dsmark and for HTB, both of which are required for this example.

2.3. tcng requirements

Support for **tcng** is the easiest part of the process. Simply untar the tcng source and run **`./configure --no-tcsim`** before compiling.

If you are on an RPM−based system, you can use the SPEC file in `tcng/build/tcng.spec` to build for your distribution, or you can download and compile this SRPM. The SRPM produces two packages, tcc and tcc−devel. You need only tcc to create configurations.

In order to run the **tcc** parser, you will also need to have the **cpp** package installed. **tcc** uses cpp.

3. Configuration examples

Examples shown here will be modified examples of downloadable configurations available in this directory.

These examples can be used as standalone configuration files to be fed into a **tcc** parser, or they can be used in conjunction with the example SysV startup script. The startup script is a modification of a script posted on the LARTC mailing list by raptor.

If you are going to use the above startup script, take a look at this example `/etc/sysconfig/tcng`:

Example 1. `/etc/sysconfig/tcng`

```
# - tcng meta-configuration file
#   (I never meta-configuration file I didn't like)
#
# -- 2003-03-15 created; -MAB
# -- 2003-03-31 modified to allow ENVAR override; -MAB
#
# -- this directory will hold all of the tcng configurations
#    used on this host
#
TCCONFBASEDIR=${TCCONFBASEDIR:-/etc/sysconfig/tcng-configs}

# -- this is the active, desired tcng configuration
#    note, that, because tcng provides the #include construct,
#    the modularity of configuration can be built into the
#    configuration files in $TCCONFBASEDIR
#
TCCONF=${TCCONF:-$TCCONFBASEDIR/global.tcc}

tcstats=${tcstats:-no}   # -- will suppress statistical output
tcstats=${tcstats:-yes}  # -- will throw the "-s" option to tc

tcdebug=${tcdebug:-0}    # -- for typical startup script usage
tcdebug=${tcdebug:-1}    # -- for a bit of information about what's happening
tcdebug=${tcdebug:-2}    # -- for debugging information
#
#
# -- an additional measure to take, you can override the default tc and tcc
#    command line utilities by specifying their pathnames here, for example:
#
#  tc=/usr/local/bin/tc
#  tcc=/usr/local/tcng/bin/tcc
#
#
```

3.1. Using tcng to shape download only

Many general concepts will be introduced with this example. This example can be compiled to its **tc** output with the command **`tcc class-selection-path.tcc`**.

Example 2. `/etc/sysconfig/tcng/class-selection-path.tcc`

```
/*
 * Simply commented example of a tcng traffic control file.
 *
 *   Martin A. Brown <martin@linux-ip.net>
 *
 * Example:  Using class selection path.
 *
 * (If you are reading the processed output in HTML, the callouts are
 *  clickable links to the description text.)
 *
 */

#include "fields.tc"     ❶
#include "ports.tc"

#define INTERFACE  eth0  ❷

dev INTERFACE {
    egress { ❸

        /* In class selection path, the filters come first!  DSmark */ ❹

        class ( <$ssh> )    if tcp_sport ==  22 && ip_tos_delay == 1 ;
        class ( <$audio> )  if tcp_sport == 554 || tcp_dport == 7070 ;
        class ( <$bulk> ) \
            if tcp_sport == PORT_SSH || tcp_dport == PORT_HTTP ; ❺
        class ( <$other> )  if 1 ; ❻

        /* section in which we configure the qdiscs and classes */

        htb () { ❼
            class ( rate 600kbps, ceil 600kbps ) { ❽
                $ssh   = class ( rate  64kbps, ceil 128kbps ) { sfq; } ;
              ❾ $audio = class ( rate 128kbps, ceil 128kbps ) { sfq; } ;
                $bulk  = class ( rate 256kbps, ceil 512kbps ) { sfq; } ;
                $other = class ( rate 128kbps, ceil 384kbps ) { sfq; } ; ❿
            }
        }
    }
}
```

❶

The **tcng** language provides support for C–style include directives which can include any file. Files are included relative to the current directory or the **tcng** library (normally `/usr/lib/tcng/include`). Strictly speaking, it is not necessary to `#include` `ports.tc` and `fields.tc`, because **tcc** will include these by default.
The use of `#include` can allow for flexible definition of variables and inclusion of common traffic control elements.
See also the tcng manual on includes.

❷

These are CPP directives. The `#define` can be used to create macros or constants. For more on their use, you should see the **tcng** manual on variables.

❸

The `egress` keyword is synonymous with the `dsmark` keyword. The example here uses class selection path. It is the use of the `egress` keyword in this configuration which requires dsmark support in the kernel and **tc**.

❹

Class selection path is one approach to traffic shaping. In class selection path, the packet is marked

(DiffServ mark) upon entry into the router. The router may take any number of actions or apply any number of policing, scheduling or shaping actions on the packet as a result of this initial classification. Consult the **tcng** manual on class selection path for further details.

❺ This example shows the use of names for the ports instead of numbers. This is one of the conveniences of **tcng** afforded by the automatic inclusion of `ports.tc`. The ports are named in accordance with IANA port names. See IANA's registered ports for these names or examine the file `ports.tc`.
Names and numbers are equally acceptable and valid.

❻ Note this peculiar construct which classifies any packet which have not yet been classified. Any packet which has not been classified by the above classifiers is put into the class "$other" here. The `if 1` construct can be used to classify the remainder of unclassified traffic.

❼ This is the creation of the root qdisc which is attached to device, `eth0` in this case. Consult the reference material in the **tcng** appendix on queuing discipline parameters for valid parameters to each qdisc. Any qdisc parameters can be inserted into the parentheses in the same fashion as the class parameters further below in the example. If no parameters need be specified, the parentheses are optional.

❽ The top level class in this example sets the maximum bandwidth allowed through this class. Let's assume that `eth0` is the inside network interface of a machine. This limits the total bandwidth to 600 kilobits per second transmitted to the internal network.
The parameters `rate` and `ceil` should be familiar to anybody who has used HTB. These are HTB specific parameters and are translated properly by the **tcc** utility. See the table on **tcng** rate and speed specification.

❾ This is the assignment of a class to a variable. This is commonly done as part of class selection path.

❿ As suggested by Martin Devera on the HTB homepage, an embedded SFQ gives each class a fair queuing algorithm for distribution of resources to the contenders passing packets through that class. Note the absence of any parameters to the embedded queuing discipline.
If no queuing discipline is specified for leaf classes, they contain the default, a pfifo_fast qdisc. The inclusion of a stochastic fair queuing qdisc in the leaf classes inhibits the ability of a single connection to dominate in a given class.

3.2. Using a two−rate three−color meter

Example 3. `/etc/sysconfig/tcng/two-rate-three-color-meter.tcc`

```
/*
 * Simply commented example of a tcng traffic control file.
 *
 *   Martin A. Brown <martin@linux-ip.net>
 *
 * Example:  Using a meter.
 *
 * (If you are reading the processed output in HTML, the callouts are
 *  clickable links to the description text.)
 *
 */

#define   EXCEPTION      192.168.137.50
#define   INTERFACE      eth0
```

```
$meter = trTCM( cir 128kbps, cbs 10kB, pir 256kbps, pbs 10kB );  ❶

dev eth0 {
    egress {
        class ( <$full> )      if ip_src == EXCEPTION      ; ❷
        class ( <$fast> )      if trTCM_green( $meter )    ; ❸
        class ( <$slow> )      if trTCM_yellow( $meter )   ; ❹
        drop                   if trTCM_red( $meter )      ; ❺
        htb {
            class ( rate 600kbps, ceil 600kbps ) {
                $fast = class ( rate 256kbps, ceil 256kbps ) { sfq; } ;
                $slow = class ( rate 128kbps, ceil 128kbps ) { sfq; } ;
                $full = class ( rate 600kbps, ceil 600kbps ) { sfq; } ;
            }
        }
    }
}
```

❶
: This is the declaration of the meter to be used for classifying traffic. The underlying technology used to implement this meter is policing. See the tcng manual on meters for the different types of meters. This meter is a two−rate three−color meter, the most complex meter available in the **tcng** language. This meter returns the colors green, yellow and red, based on the rates offered in the committed and peak buckets. If the metered rate exceeds the committed rate, this meter will turn yellow, and if the metered rate exceeds the peak rate, this meter will turn red.
The variable `$meter` can be operated on by functions applicable to the meter type. In this case, there are three functions available for testing `$meter`'s state, `trTCM_green`, `trTCM_yellow`, and `trTCM_red`. For efficiency, consider also the accelerated counterparts.

❷
: In this example, the IP 192.168.137.50 is specifically excluded from the policing control applied to traffic departing on eth0.

❸
: Up to the committed information rate (`cir`), packets will pass through this class. Tokens will be removed from the `cir`/`cbs` bucket.
The meter is green.

❹
: Traffic flow exceeding the `cir`/`cbs` bucket will be classified here. The `pir`/`pbs` bucket (`pir` is peak information rate, `pbs` is peak burst size). This allows a particular flow to be guaranteed one class of service up to a given rate, and then be reclassified above that rate.
The meter is yellow.

❺
: Traffic flow exceeding the `pir`/`pbs` bucket will be classified here. A common configuration causes traffic to be dropped above peak rate, although traffic could be re−classified into a best−effort class from a guaranteed class.
The meter is red.

---

4. Miscellaneous Notes

Thankfully, **tcng** does away with one of the minor annoyances of **tc**. The following table maps the syntax and convention of these tools with English equivalents.

Table 1. Speed/Rate syntax: tcng vs. tc

tcng	**English**	**tc**
bps	bit(s) per second	bit
Bps	byte(s) per second	bps (argh!)
kbps	kilobit(s) per second	kbit
kBps	kilobyte(s) per second	kbps
Mbps	megabit(s) per second	mbit or Mbit
MBps	megabyte(s) per second	mbps or Mbps
pps	packet per second	??

Note that this means a slight adjustment for longtime users of **tc**, but a much better choice for intuitive usablity for English speakers.

For example, we can use conventional expressions of rate in **tcng** configurations: **`100Mbps`**, **`128kbps`**, and even **`2Gpps`**. See also the **tcng** manual on units.

In order for traffic control to be effective, it is important to understand where the bottlenecks are. In most cases, you'll want to perform the traffic control at or near the bottleneck.

5. Links and Further documentation

- the linux DiffServ project
- HTB site (*Martin "devik" Devera*)
- Traffic Control Next Generation (**tcng**)

  TCNG manual (*Werner Almesberger*)
- iproute2 (*Alexey Kuznetsov*)

  iproute2 manual (*Alexey Kuznetsov*)
- Research and documentation on traffic control under linux (*Stef Coene*)
- LARTC HOWTO (*bert hubert, et. al.*)
- guide to IP networking with linux (*Martin A. Brown*)