Traffic Control HOWTO

Version 1.0.2

Martin A. Brown

linux−ip.net

<martin@linux-ip.net>

"Oct 2006"

Revision History

Revision 1.0.2	2006−10−28	Revised by: MAB
Add references to HFSC, alter author email addresses		
Revision 1.0.1	2003−11−17	Revised by: MAB
Added link to Leonardo Balliache's documentation		
Revision 1.0	2003−09−24	Revised by: MAB
reviewed and approved by TLDP		
Revision 0.7	2003−09−14	Revised by: MAB
incremental revisions, proofreading, ready for TLDP		
Revision 0.6	2003−09−09	Revised by: MAB
minor editing, corrections from Stef Coene		
Revision 0.5	2003−09−01	Revised by: MAB
HTB section mostly complete, more diagrams, LARTC pre−release		
Revision 0.4	2003−08−30	Revised by: MAB
added diagram		
Revision 0.3	2003−08−29	Revised by: MAB
substantial completion of classless, software, rules, elements and components sections		
Revision 0.2	2003−08−23	Revised by: MAB
major work on overview, elements, components and software sections		
Revision 0.1	2003−08−15	Revised by: MAB
initial revision (outline complete)		

Traffic control encompasses the sets of mechanisms and operations by which packets are queued for transmission/reception on a network interface. The operations include enqueuing, policing, classifying, scheduling, shaping and dropping. This HOWTO provides an introduction and overview of the capabilities and implementation of traffic control under Linux.

Table of Contents

Table of Contents

1. Introduction to Linux Traffic Control

Linux offers a very rich set of tools for managing and manipulating the transmission of packets. The larger Linux community is very familiar with the tools available under Linux for packet mangling and firewalling (netfilter, and before that, ipchains) as well as hundreds of network services which can run on the operating system. Few inside the community and fewer outside the Linux community are aware of the tremendous power of the traffic control subsystem which has grown and matured under kernels 2.2 and 2.4.

This HOWTO purports to introduce the concepts of traffic control, the traditional elements (in general), the components of the Linux traffic control implementation and provide some guidelines . This HOWTO represents the collection, amalgamation and synthesis of the LARTC HOWTO, documentation from individual projects and importantly the LARTC mailing list over a period of study.

The impatient soul, who simply wishes to experiment right now, is recommended to the Traffic Control using tcng and HTB HOWTO and LARTC HOWTO for immediate satisfaction.

1.1. Target audience and assumptions about the reader

The target audience for this HOWTO is the network administrator or savvy home user who desires an introduction to the field of traffic control and an overview of the tools available under Linux for implementing traffic control.

I assume that the reader is comfortable with UNIX concepts and the command line and has a basic knowledge of IP networking. Users who wish to implement traffic control may require the ability to patch, compile and install a kernel or software package [1]. For users with newer kernels (2.4.20+, see also Section 5.1), however, the ability to install and use software may be all that is required.

Broadly speaking, this HOWTO was written with a sophisticated user in mind, perhaps one who has already had experience with traffic control under Linux. I assume that the reader may have no prior traffic control experience.

1.2. Conventions

This text was written in DocBook (version 4.2) with **vim**. All formatting has been applied by xsltproc based on DocBook XSL and LDP XSL stylesheets. Typeface formatting and display conventions are similar to most printed and electronically distributed technical documentation.

1.3. Recommended approach

I strongly recommend to the eager reader making a first foray into the discipline of traffic control, to become only casually familiar with the **tc** command line utility, before concentrating on **tcng**. The **tcng** software package defines an entire language for describing traffic control structures. At first, this language may seem daunting, but mastery of these basics will quickly provide the user with a much wider ability to employ (and deploy) traffic control configurations than the direct use of **tc** would afford.

Where possible, I'll try to prefer describing the behaviour of the Linux traffic control system in an abstract manner, although in many cases I'll need to supply the syntax of one or the other common systems for defining these structures. I may not supply examples in both the **tcng** language and the **tc** command line, so the wise user will have some familiarity with both.

1.4. Missing content, corrections and feedback

There is content yet missing from this HOWTO. In particular, the following items will be added at some point to this documentation.

- A description and diagram of GRED, WRR, PRIO and CBQ.
- A section of examples.
- A section detailing the classifiers.
- A section discussing the techniques for measuring traffic.
- A section covering meters.
- More details on **tcng**.

I welcome suggestions, corrections and feedback at <`martin@linux-ip.net`>. All errors and omissions are strictly my fault. Although I have made every effort to verify the factual correctness of the content presented herein, I cannot accept any responsibility for actions taken under the influence of this documentation.

2. Overview of Concepts

This section will introduce traffic control and examine reasons for it, identify a few advantages and disadvantages and introduce key concepts used in traffic control.

2.1. What is it?

Traffic control is the name given to the sets of queuing systems and mechanisms by which packets are received and transmitted on a router. This includes deciding which (and whether) packets to accept at what rate on the input of an interface and determining which packets to transmit in what order at what rate on the output of an interface.

In the overwhelming majority of situations, traffic control consists of a single queue which collects entering packets and dequeues them as quickly as the hardware (or underlying device) can accept them. This sort of queue is a FIFO.

The default qdisc under Linux is the `pfifo_fast`, which is slightly more complex than the FIFO.

There are examples of queues in all sorts of software. The queue is a way of organizing the pending tasks or data (see also Section 2.5). Because network links typically carry data in a serialized fashion, a queue is required to manage the outbound data packets.

In the case of a desktop machine and an efficient webserver sharing the same uplink to the Internet, the following contention for bandwidth may occur. The web server may be able to fill up the output queue on the router faster than the data can be transmitted across the link, at which point the router starts to drop packets (its buffer is full!). Now, the desktop machine (with an interactive application user) may be faced with packet loss and high latency. Note that high latency sometimes leads to screaming users! By separating the internal queues used to service these two different classes of application, there can be better sharing of the network resource between the two applications.

Traffic control is the set of tools which allows the user to have granular control over these queues and the queuing mechanisms of a networked device. The power to rearrange traffic flows and packets with these tools is tremendous and can be complicated, but is no substitute for adequate bandwidth.

The term Quality of Service (QoS) is often used as a synonym for traffic control.

2.2. Why use it?

Packet−switched networks differ from circuit based networks in one very important regard. A packet−switched network itself is stateless. A circuit−based network (such as a telephone network) must hold state within the network. IP networks are stateless and packet−switched networks by design; in fact, this statelessness is one of the fundamental strengths of IP.

The weakness of this statelessness is the lack of differentiation between types of flows. In simplest terms, traffic control allows an administrator to queue packets differently based on attributes of the packet. It can even be used to simulate the behaviour of a circuit−based network. This introduces statefulness into the stateless network.

There are many practical reasons to consider traffic control, and many scenarios in which using traffic control makes sense. Below are some examples of common problems which can be solved or at least ameliorated with these tools.

The list below is not an exhaustive list of the sorts of solutions available to users of traffic control, but introduces the types of problems that can be solved by using traffic control to maximize the usability of a network connection.

Common traffic control solutions

- Limit total bandwidth to a known rate; TBF, HTB with child class(es).
- Limit the bandwidth of a particular user, service or client; HTB classes and classifying with a `filter`. traffic.
- Maximize TCP throughput on an asymmetric link; prioritize transmission of ACK packets, wondershaper.
- Reserve bandwidth for a particular application or user; HTB with children classes and classifying.
- Prefer latency sensitive traffic; PRIO inside an HTB class.
- Managed oversubscribed bandwidth; HTB with borrowing.
- Allow equitable distribution of unreserved bandwidth; HTB with borrowing.
- Ensure that a particular type of traffic is dropped; `policer` attached to a `filter` with a `drop` action.

Remember, too that sometimes, it is simply better to purchase more bandwidth. Traffic control does not solve all problems!

2.3. Advantages

When properly employed, traffic control should lead to more predictable usage of network resources and less volatile contention for these resources. The network then meets the goals of the traffic control configuration. Bulk download traffic can be allocated a reasonable amount of bandwidth even as higher priority interactive traffic is simultaneously serviced. Even low priority data transfer such as mail can be allocated bandwidth without tremendously affecting the other classes of traffic.

In a larger picture, if the traffic control configuration represents policy which has been communicated to the users, then users (and, by extension, applications) know what to expect from the network.

2.4. Disdvantages

Complexity is easily one of the most significant disadvantages of using traffic control. There are ways to become familiar with traffic control tools which ease the learning curve about traffic control and its mechanisms, but identifying a traffic control misconfiguration can be quite a challenge.

Traffic control when used appropriately can lead to more equitable distribution of network resources. It can just as easily be installed in an inappropriate manner leading to further and more divisive contention for resources.

The computing resources required on a router to support a traffic control scenario need to be capable of handling the increased cost of maintaining the traffic control structures. Fortunately, this is a small incremental cost, but can become more significant as the configuration grows in size and complexity.

For personal use, there's no training cost associated with the use of traffic control, but a company may find that purchasing more bandwidth is a simpler solution than employing traffic control. Training employees and ensuring depth of knowledge may be more costly than investing in more bandwidth.

Although traffic control on packet−switched networks covers a larger conceptual area, you can think of traffic control as a way to provide [some of] the statefulness of a circuit−based network to a packet−switched network.

2.5. Queues

Queues form the backdrop for all of traffic control and are the integral concept behind scheduling. A queue is a location (or buffer) containing a finite number of items waiting for an action or service. In networking, a queue is the place where packets (our units) wait to be transmitted by the hardware (the service). In the simplest model, packets are transmitted in a first−come first−serve basis [2]. In the discipline of computer networking (and more generally computer science), this sort of a queue is known as a FIFO.

Without any other mechanisms, a queue doesn't offer any promise for traffic control. There are only two interesting actions in a queue. Anything entering a queue is enqueued into the queue. To remove an item from a queue is to dequeue that item.

A queue becomes much more interesting when coupled with other mechanisms which can delay packets, rearrange, drop and prioritize packets in multiple queues. A queue can also use subqueues, which allow for complexity of behaviour in a scheduling operation.

From the perspective of the higher layer software, a packet is simply enqueued for transmission, and the manner and order in which the enqueued packets are transmitted is immaterial to the higher layer. So, to the higher layer, the entire traffic control system may appear as a single queue [3]. It is only by examining the internals of this layer that the traffic control structures become exposed and available.

2.6. Flows

A flow is a distinct connection or conversation between two hosts. Any unique set of packets between two hosts can be regarded as a flow. Under TCP the concept of a connection with a source IP and port and destination IP and port represents a flow. A UDP flow can be similarly defined.

Traffic control mechanisms frequently separate traffic into classes of flows which can be aggregated and transmitted as an aggregated flow (consider DiffServ). Alternate mechanisms may attempt to divide bandwidth equally based on the individual flows.

Flows become important when attempting to divide bandwidth equally among a set of competing flows, especially when some applications deliberately build a large number of flows.

2.7. Tokens and buckets

Two of the key underpinnings of a shaping mechanisms are the interrelated concepts of tokens and buckets.

In order to control the rate of dequeuing, an implementation can count the number of packets or bytes dequeued as each item is dequeued, although this requires complex usage of timers and measurements to limit accurately. Instead of calculating the current usage and time, one method, used widely in traffic control, is to generate tokens at a desired rate, and only dequeue packets or bytes if a token is available.

Consider the analogy of an amusement park ride with a queue of people waiting to experience the ride. Let's imagine a track on which carts traverse a fixed track. The carts arrive at the head of the queue at a fixed rate. In order to enjoy the ride, each person must wait for an available cart. The cart is analogous to a token and the person is analogous to a packet. Again, this mechanism is a rate−limiting or shaping mechanism. Only a certain number of people can experience the ride in a particular period.

To extend the analogy, imagine an empty line for the amusement park ride and a large number of carts sitting on the track ready to carry people. If a large number of people entered the line together many (maybe all) of them could experience the ride because of the carts available and waiting. The number of carts available is a concept analogous to the bucket. A bucket contains a number of tokens and can use all of the tokens in bucket without regard for passage of time.

And to complete the analogy, the carts on the amusement park ride (our tokens) arrive at a fixed rate and are only kept available up to the size of the bucket. So, the bucket is filled with tokens according to the rate, and if the tokens are not used, the bucket can fill up. If tokens are used the bucket will not fill up. Buckets are a key concept in supporting bursty traffic such as HTTP.

The TBF qdisc is a classical example of a shaper (the section on TBF includes a diagram which may help to visualize the token and bucket concepts). The TBF generates `rate` tokens and only transmits packets when a token is available. Tokens are a generic shaping concept.

In the case that a queue does not need tokens immediately, the tokens can be collected until they are needed. To collect tokens indefinitely would negate any benefit of shaping so tokens are collected until a certain number of tokens has been reached. Now, the queue has tokens available for a large number of packets or bytes which need to be dequeued. These intangible tokens are stored in an intangible bucket, and the number of tokens that can be stored depends on the size of the bucket.

This also means that a bucket full of tokens may be available at any instant. Very predictable regular traffic can be handled by small buckets. Larger buckets may be required for burstier traffic, unless one of the desired goals is to reduce the burstiness of the flows.

In summary, tokens are generated at rate, and a maximum of a bucket's worth of tokens may be collected. This allows bursty traffic to be handled, while smoothing and shaping the transmitted traffic.

The concepts of tokens and buckets are closely interrelated and are used in both TBF (one of the classless qdiscs) and HTB (one of the classful qdiscs). Within the **tcng** language, the use of two− and three−color meters is indubitably a token and bucket concept.

2.8. Packets and frames

The terms for data sent across network changes depending on the layer the user is examining. This document will rather impolitely (and incorrectly) gloss over the technical distinction between packets and frames although they are outlined here.

The word frame is typically used to describe a layer 2 (data link) unit of data to be forwarded to the next recipient. Ethernet interfaces, PPP interfaces, and T1 interfaces all name their layer 2 data unit a frame. The frame is actually the unit on which traffic control is performed.

A packet, on the other hand, is a higher layer concept, representing layer 3 (network) units. The term packet is preferred in this documentation, although it is slightly inaccurate.

3. Traditional Elements of Traffic Control

3.1. Shaping

Shapers delay packets to meet a desired rate.

Shaping is the mechanism by which packets are delayed before transmission in an output queue to meet a desired output rate. This is one of the most common desires of users seeking bandwidth control solutions. The act of delaying a packet as part of a traffic control solution makes every shaping mechanism into a non−work−conserving mechanism, meaning roughly: "Work is required in order to delay packets."

Viewed in reverse, a non−work−conserving queuing mechanism is performing a shaping function. A work−conserving queuing mechanism (see PRIO) would not be capable of delaying a packet.

Shapers attempt to limit or ration traffic to meet but not exceed a configured rate (frequently measured in packets per second or bits/bytes per second). As a side effect, shapers can smooth out bursty traffic [4]. One of the advantages of shaping bandwidth is the ability to control latency of packets. The underlying mechanism for shaping to a rate is typically a token and bucket mechanism. See also Section 2.7 for further detail on tokens and buckets.

3.2. Scheduling

Schedulers arrange and/or rearrange packets for output.

Scheduling is the mechanism by which packets are arranged (or rearranged) between input and output of a particular queue. The overwhelmingly most common scheduler is the FIFO (first−in first−out) scheduler. From a larger perspective, any set of traffic control mechanisms on an output queue can be regarded as a scheduler, because packets are arranged for output.

Other generic scheduling mechanisms attempt to compensate for various networking conditions. A fair queuing algorithm (see SFQ) attempts to prevent any single client or flow from dominating the network usage. A round−robin algorithm (see WRR) gives each flow or client a turn to dequeue packets. Other sophisticated scheduling algorithms attempt to prevent backbone overload (see GRED) or refine other scheduling mechanisms (see ESFQ).

3.3. Classifying

Classifiers sort or separate traffic into queues.

Classifying is the mechanism by which packets are separated for different treatment, possibly different output queues. During the process of accepting, routing and transmitting a packet, a networking device can classify the packet a number of different ways. Classification can include marking the packet, which usually happens on the boundary of a network under a single administrative control or classification can occur on each hop individually.

The Linux model (see Section 4.3) allows for a packet to cascade across a series of classifiers in a traffic control structure and to be classified in conjunction with policers (see also Section 4.5).

3.4. Policing

Policers measure and limit traffic in a particular queue.

Policing, as an element of traffic control, is simply a mechanism by which traffic can be limited. Policing is most frequently used on the network border to ensure that a peer is not consuming more than its allocated bandwidth. A policer will accept traffic to a certain rate, and then perform an action on traffic exceeding this rate. A rather harsh solution is to drop the traffic, although the traffic could be reclassified instead of being dropped.

A policer is a yes/no question about the rate at which traffic is entering a queue. If the packet is about to enter a queue below a given rate, take one action (allow the enqueuing). If the packet is about to enter a queue above a given rate, take another action. Although the policer uses a token bucket mechanism internally, it does not have the capability to delay a packet as a shaping mechanism does.

3.5. Dropping

Dropping discards an entire packet, flow or classification.

Dropping a packet is a mechanism by which a packet is discarded.

3.6. Marking

Marking is a mechanism by which the packet is altered.

This is not `fwmark`. The **iptables**target `MARK`and the **ipchains**`--mark`are used to modify packet metadata, not the packet itself.

Traffic control marking mechanisms install a DSCP on the packet itself, which is then used and respected by other routers inside an administrative domain (usually for DiffServ).

4. Components of Linux Traffic Control

Table 1. Correlation between traffic control elements and Linux components

traditional element	Linux component
shaping	The `class` offers shaping capabilities.
scheduling	A `qdisc` is a scheduler. Schedulers can be simple such as the FIFO or complex, containing classes and other qdiscs, such as HTB.
classifying	The `filter` object performs the classification through the agency of a `classifier` object. Strictly speaking, Linux classifiers cannot exist outside of a filter.
policing	A `policer` exists in the Linux traffic control implementation only as part of a `filter`.
dropping	To `drop` traffic requires a `filter` with a `policer` which uses "drop" as an action.
marking	The `dsmark` `qdisc` is used for marking.

4.1. `qdisc`

Simply put, a qdisc is a scheduler (Section 3.2). Every output interface needs a scheduler of some kind, and the default scheduler is a FIFO. Other qdiscs available under Linux will rearrange the packets entering the scheduler's queue in accordance with that scheduler's rules.

The qdisc is the major building block on which all of Linux traffic control is built, and is also called a queuing discipline.

The classful qdiscs can contain `class`es, and provide a handle to which to attach `filter`s. There is no prohibition on using a classful qdisc without child classes, although this will usually consume cycles and other system resources for no benefit.

The classless qdiscs can contain no classes, nor is it possible to attach filter to a classless qdisc. Because a classless qdisc contains no children of any kind, there is no utility to classifying. This means that no filter can be attached to a classless qdisc.

A source of terminology confusion is the usage of the terms `root` qdisc and `ingress` qdisc. These are not really queuing disciplines, but rather locations onto which traffic control structures can be attached for egress (outbound traffic) and ingress (inbound traffic).

Each interface contains both. The primary and more common is the egress qdisc, known as the `root` qdisc. It can contain any of the queuing disciplines (`qdisc`s) with potential `class`es and class structures. The overwhelming majority of documentation applies to the `root` qdisc and its children. Traffic transmitted on an interface traverses the egress or `root` qdisc.

For traffic accepted on an interface, the `ingress` qdisc is traversed. With its limited utility, it allows no child `class` to be created, and only exists as an object onto which a `filter` can be attached. For practical purposes, the `ingress` qdisc is merely a convenient object onto which to attach a `policer` to limit the amount of traffic accepted on a network interface.

In short, you can do much more with an egress qdisc because it contains a real qdisc and the full power of the traffic control system. An `ingress` qdisc can only support a policer. The remainder of the documentation will concern itself with traffic control structures attached to the `root` qdisc unless otherwise specified.

4.2. `class`

Classes only exist inside a classful `qdisc` (*e.g.*, HTB and CBQ). Classes are immensely flexible and can always contain either multiple children classes or a single child qdisc [5]. There is no prohibition against a class containing a classful qdisc itself, which facilitates tremendously complex traffic control scenarios.

Any class can also have an arbitrary number of `filter`s attached to it, which allows the selection of a child class or the use of a filter to reclassify or drop traffic entering a particular class.

A leaf class is a terminal class in a qdisc. It contains a qdisc (default FIFO) and will never contain a child class. Any class which contains a child class is an inner class (or root class) and not a leaf class.

4.3. `filter`

The filter is the most complex component in the Linux traffic control system. The filter provides a convenient mechanism for gluing together several of the key elements of traffic control. The simplest and most obvious role of the filter is to classify (see Section 3.3) packets. Linux filters allow the user to classify packets into an output queue with either several different filters or a single filter.

- A filter must contain a `classifier` phrase.
- A filter may contain a `policer` phrase.

Filters can be attached either to classful `qdisc`s or to `class`es, however the enqueued packet always enters the root qdisc first. After the filter attached to the root qdisc has been traversed, the packet may be directed to any subclasses (which can have their own filters) where the packet may undergo further classification.

4.4. classifier

Filter objects, which can be manipulated using **tc**, can use several different classifying mechanisms, the most common of which is the `u32` classifier. The `u32` classifier allows the user to select packets based on attributes of the packet.

The classifiers are tools which can be used as part of a `filter` to identify characteristics of a packet or a packet's metadata. The Linux classfier object is a direct analogue to the basic operation and elemental mechanism of traffic control classifying.

4.5. policer

This elemental mechanism is only used in Linux traffic control as part of a `filter`. A policer calls one action above and another action below the specified rate. Clever use of policers can simulate a three–color meter. See also Section 10.

Although both policing and shaping are basic elements of traffic control for limiting bandwidth usage a policer will never delay traffic. It can only perform an action based on specified criteria. See also Example 5.

4.6. `drop`

This basic traffic control mechanism is only used in Linux traffic control as part of a `policer`. Any policer attached to any `filter` could have a `drop` action.

The only place in the Linux traffic control system where a packet can be explicitly dropped is a policer. A policer can limit packets enqueued at a specific rate, or it can be configured to drop all traffic matching a particular pattern [6].

There are, however, places within the traffic control system where a packet may be dropped as a side effect. For example, a packet will be dropped if the scheduler employed uses this method to control flows as the GRED does.

Also, a shaper or scheduler which runs out of its allocated buffer space may have to drop a packet during a particularly bursty or overloaded period.

4.7. `handle`

Every `class` and classful `qdisc` (see also Section 7) requires a unique identifier within the traffic control structure. This unique identifier is known as a handle and has two constituent members, a major number and a minor number. These numbers can be assigned arbitrarily by the user in accordance with the following rules [7].

The numbering of handles for classes and qdiscs

`major`
: This parameter is completely free of meaning to the kernel. The user may use an arbitrary numbering scheme, however all objects in the traffic control structure with the same parent must share a *`major`* handle number. Conventional numbering schemes start at 1 for objects attached directly to the `root` qdisc.

`minor`
: This parameter unambiguously identifies the object as a qdisc if *`minor`* is 0. Any other value identifies the object as a class. All classes sharing a parent must have unique *`minor`* numbers.

The special handle ffff:0 is reserved for the `ingress` qdisc.

The handle is used as the target in *`classid`* and *`flowid`* phrases of **tc** `filter` statements. These handles are external identifiers for the objects, usable by userland applications. The kernel maintains internal identifiers for each object.

5. Software and Tools

5.1. Kernel requirements

Many distributions provide kernels with modular or monolithic support for traffic control (Quality of Service). Custom kernels may not already provide support (modular or not) for the required features. If not, this is a very brief listing of the required kernel options.

The user who has little or no experience compiling a kernel is recommended to Kernel HOWTO. Experienced kernel compilers should be able to determine which of the below options apply to the desired configuration, after reading a bit more about traffic control and planning.

Example 1. Kernel compilation options [8]

```
#
# QoS and/or fair queueing
#
CONFIG_NET_SCHED=y
CONFIG_NET_SCH_CBQ=m
CONFIG_NET_SCH_HTB=m
CONFIG_NET_SCH_CSZ=m
CONFIG_NET_SCH_PRIO=m
CONFIG_NET_SCH_RED=m
CONFIG_NET_SCH_SFQ=m
CONFIG_NET_SCH_TEQL=m
CONFIG_NET_SCH_TBF=m
CONFIG_NET_SCH_GRED=m
CONFIG_NET_SCH_DSMARK=m
CONFIG_NET_SCH_INGRESS=m
CONFIG_NET_QOS=y
CONFIG_NET_ESTIMATOR=y
CONFIG_NET_CLS=y
CONFIG_NET_CLS_TCINDEX=m
CONFIG_NET_CLS_ROUTE4=m
CONFIG_NET_CLS_ROUTE=y
CONFIG_NET_CLS_FW=m
CONFIG_NET_CLS_U32=m
CONFIG_NET_CLS_RSVP=m
CONFIG_NET_CLS_RSVP6=m
CONFIG_NET_CLS_POLICE=y
```

A kernel compiled with the above set of options will provide modular support for almost everything discussed in this documentation. The user may need to **modprobe** ***`module`*** before using a given feature. Again, the confused user is recommended to the Kernel HOWTO, as this document cannot adequately address questions about the use of the Linux kernel.

5.2. iproute2 tools (tc)

iproute2 is a suite of command line utilities which manipulate kernel structures for IP networking configuration on a machine. For technical documentation on these tools, see the iproute2 documentation and for a more expository discussion, the documentation at linux−ip.net. Of the tools in the **iproute2** package, the binary **tc** is the only one used for traffic control. This HOWTO will ignore the other tools in the suite.

Because it interacts with the kernel to direct the creation, deletion and modification of traffic control structures, the **tc** binary needs to be compiled with support for all of the `qdisc`s you wish to use. In particular, the HTB qdisc is not supported yet in the upstream **iproute2** package. See Section 7.1 for more information.

The **tc** tool performs all of the configuration of the kernel structures required to support traffic control. As a result of its many uses, the command syntax can be described (at best) as arcane. The utility takes as its first non−option argument one of three Linux traffic control components, `qdisc`, `class` or `filter`.

Example 2. tc command usage

```
[root@leander]# tc
Usage: tc [ OPTIONS ] OBJECT { COMMAND | help }
where  OBJECT := { qdisc | class | filter }
       OPTIONS := { −s[tatistics] | −d[etails] | −r[aw] }
```

Each object accepts further and different options, and will be incompletely described and documented below. The hints in the examples below are designed to introduce the vagaries of **tc** command line syntax. For more examples, consult the LARTC HOWTO. For even better understanding, consult the kernel and **iproute2** code.

Example 3. tc `qdisc`

```
[root@leander]# tc qdisc add    \ ❶
>                  dev eth0     \ ❷
>                  root         \ ❸
>                  handle 1:0   \ ❹
>                  htb            ❺
```

❶ Add a queuing discipline. The verb could also be `del`.

❷ Specify the device onto which we are attaching the new queuing discipline.

❸ This means "egress" to **tc**. The word `root` must be used, however. Another qdisc with limited functionality, the `ingress` qdisc can be attached to the same device.

❹ The `handle` is a user−specified number of the form *`major`*:*`minor`*. The minor number for any queueing discipline handle must always be zero (0). An acceptable shorthand for a `qdisc` handle is the syntax "1:", where the minor number is assumed to be zero (0) if not specified.

❺ This is the queuing discipline to attach, HTB in this example. Queuing discipline specific parameters will follow this. In the example here, we add no qdisc−specific parameters.

Above was the simplest use of the **tc** utility for adding a queuing discipline to a device. Here's an example of the use of **tc** to add a class to an existing parent class.

Example 4. tc `class`

```
[root@leander]# tc class add    \ ❶
>                  dev eth0     \ ❷
>                  parent 1:1   \ ❸
>                  classid 1:6  \ ❹
>                  htb          \ ❺
>                  rate 256kbit \ ❻
>                  ceil 512kbit   ❼
```

❶ Add a class. The verb could also be `del`.

❷ Specify the device onto which we are attaching the new class.

❸ Specify the parent `handle` to which we are attaching the new class.

❹ This is a unique `handle` (*`major`*:*`minor`*) identifying this class. The minor number must be any non−zero (0) number.

❺ Both of the classful qdiscs require that any children classes be classes of the same type as the parent. Thus an HTB qdisc will contain HTB classes.

❻❼ This is a class specific parameter. Consult Section 7.1 for more detail on these parameters.

Example 5. tc `filter`

```
[root@leander]# tc filter add               \ ❶
>                  dev eth0                 \ ❷
>                  parent 1:0               \ ❸
>                  protocol ip              \ ❹
>                  prio 5                   \ ❺
>                  u32                      \ ❻
>                  match ip port 22 0xffff  \ ❼
>                  match ip tos 0x10 0xff   \ ❽
>                  flowid 1:6               \ ❾
>                  police                   \ ❿
>                  rate 32000bps            \ (11)
>                  burst 10240              \ (12)
>                  mpu 0                    \ (13)
>                  action drop/continue       (14)
```

❶ Add a filter. The verb could also be `del`.

❷ Specify the device onto which we are attaching the new filter.

❸ Specify the parent handle to which we are attaching the new filter.

❹ This parameter is required. It's use should be obvious, although I don't know more.

❺ The *`prio`* parameter allows a given filter to be preferred above another. The *`pref`* is a synonym.

❻ This is a `classifier`, and is a required phrase in every **tc** `filter` command.

❼❽ These are parameters to the classifier. In this case, packets with a type of service flag (indicating interactive usage) and matching port 22 will be selected by this statement.

❾ The *`flowid`* specifies the `handle` of the target class (or qdisc) to which a matching filter should send its selected packets.

⑩
This is the `policer`, and is an optional phrase in every **tc** `filter` command.

(11)
The policer will perform one action above this rate, and another action below (see action parameter).

(12)
The *burst* is an exact analog to *burst* in HTB (*burst* is a buckets concept).

(13)
The minimum policed unit. To count all traffic, use an *mpu* of zero (0).

(14)
The *action* indicates what should be done if the *rate* based on the attributes of the policer. The first word specifies the action to take if the policer has been exceeded. The second word specifies action to take otherwise.

As evidenced above, the **tc** command line utility has an arcane and complex syntax, even for simple operations such as these examples show. It should come as no surprised to the reader that there exists an easier way to configure Linux traffic control. See the next section, Section 5.3.

5.3. tcng, Traffic Control Next Generation

FIXME; sing the praises of tcng. See also Traffic Control using tcng and HTB HOWTO and tcng documentation.

Traffic control next generation (hereafter, **tcng**) provides all of the power of traffic control under Linux with twenty percent of the headache.

5.4. IMQ, Intermediate Queuing device

FIXME; must discuss IMQ. See also Patrick McHardy's website on IMQ.

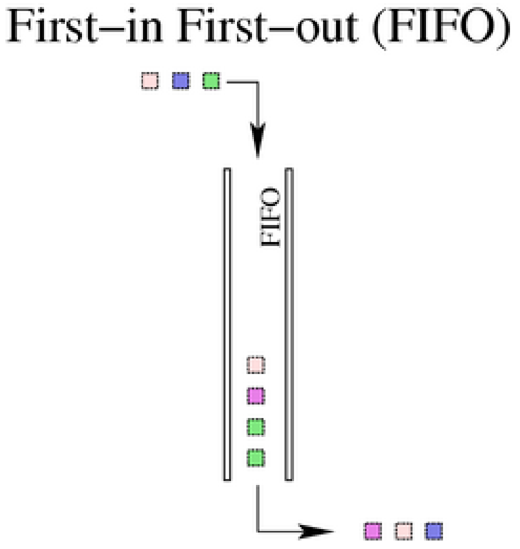
6. Classless Queuing Disciplines (`qdisc`s)

Each of these queuing disciplines can be used as the primary qdisc on an interface, or can be used inside a leaf class of a classful qdiscs. These are the fundamental schedulers used under Linux. Note that the default scheduler is the `pfifo_fast`.

6.1. FIFO, First−In First−Out (`pfifo` and `bfifo`)

This is not the default qdisc on Linux interfaces. Be certain to see Section 6.2 for the full details on the default (`pfifo_fast`) qdisc.

The FIFO algorithm forms the basis for the default qdisc on all Linux network interfaces (`pfifo_fast`). It performs no shaping or rearranging of packets. It simply transmits packets as soon as it can after receiving and queuing them. This is also the qdisc used inside all newly created classes until another qdisc or a class replaces the FIFO.



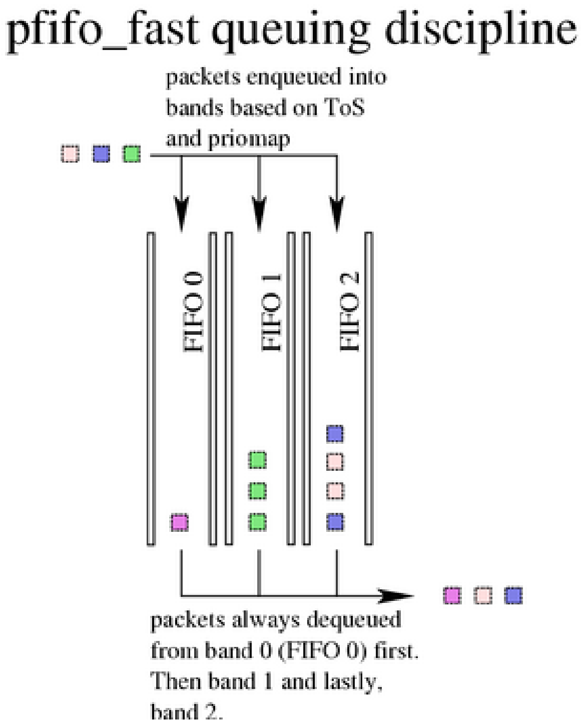
A real FIFO qdisc must, however, have a size limit (a buffer size) to prevent it from overflowing in case it is unable to dequeue packets as quickly as it receives them. Linux implements two basic FIFO `qdisc`s, one based on bytes, and one on packets. Regardless of the type of FIFO used, the size of the queue is defined by the parameter *`limit`*. For a `pfifo` the unit is understood to be packets and for a `bfifo` the unit is understood to be bytes.

Example 6. Specifying a *`limit`* for a packet or byte FIFO

```
[root@leander]# cat bfifo.tcc
/*
 * make a FIFO on eth0 with 10kbyte queue size
 *
 */

dev eth0 {
    egress {
        fifo (limit 10kB );
    }
}
[root@leander]# tcc < bfifo.tcc
# ================================ Device eth0 ================================

tc qdisc add dev eth0 handle 1:0 root dsmark indices 1 default_index 0
tc qdisc add dev eth0 handle 2:0 parent 1:0 bfifo limit 10240
[root@leander]# cat pfifo.tcc
/*
 * make a FIFO on eth0 with 30 packet queue size
 *
 */

dev eth0 {
    egress {
        fifo (limit 30p );
    }
}
[root@leander]# tcc < pfifo.tcc
# ================================ Device eth0 ================================

tc qdisc add dev eth0 handle 1:0 root dsmark indices 1 default_index 0
tc qdisc add dev eth0 handle 2:0 parent 1:0 pfifo limit 30
```

6.2. `pfifo_fast`, the default Linux qdisc

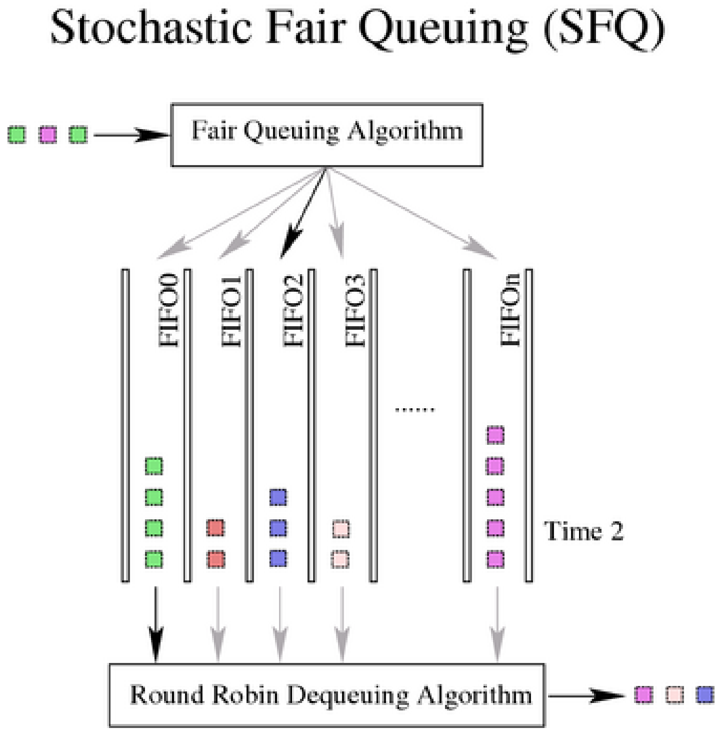
The `pfifo_fast` qdisc is the default qdisc for all interfaces under Linux. Based on a conventional FIFO qdisc, this qdisc also provides some prioritization. It provides three different bands (individual FIFOs) for separating traffic. The highest priority traffic (interactive flows) are placed into band 0 and are always serviced first. Similarly, band 1 is always emptied of pending packets before band 2 is dequeued.


pfifo_fast queuing discipline
packets enqueued into bands based on ToS and priomap
FIFO 0
FIFO 1
FIFO 2
packets always dequeued from band 0 (FIFO 0) first. Then band 1 and lastly, band 2.

There is nothing configurable to the end user about the `pfifo_fast` qdisc. For exact details on the `priomap` and use of the ToS bits, see the pfifo–fast section of the LARTC HOWTO.

6.3. SFQ, Stochastic Fair Queuing

The SFQ qdisc attempts to fairly distribute opportunity to transmit data to the network among an arbitrary number of flows. It accomplishes this by using a hash function to separate the traffic into separate (internally maintained) FIFOs which are dequeued in a round–robin fashion. Because there is the possibility for unfairness to manifest in the choice of hash function, this function is altered periodically. Perturbation (the parameter *`perturb`*) sets this periodicity.


Stochastic Fair Queuing (SFQ)
Fair Queuing Algorithm
FIFO0
FIFO1
FIFO2
FIFO3
......
FIFOn
Time 2
Round Robin Dequeuing Algorithm

Example 7. Creating an SFQ

```
[root@leander]# cat sfq.tcc
/*
 * make an SFQ on eth0 with a 10 second perturbation
 *
 */

dev eth0 {
    egress {
        sfq( perturb 10s );
    }
}
[root@leander]# tcc < sfq.tcc
# ================================ Device eth0 ================================

tc qdisc add dev eth0 handle 1:0 root dsmark indices 1 default_index 0
tc qdisc add dev eth0 handle 2:0 parent 1:0 sfq perturb 10
```

Unfortunately, some clever software (*e.g.* Kazaa and eMule among others) obliterate the benefit of this attempt at fair queuing by opening as many TCP sessions (flows) as can be sustained. In many networks, with well−behaved users, SFQ can adequately distribute the network resources to the contending flows, but other measures may be called for when obnoxious applications have invaded the network.

See also Section 6.4 for an SFQ qdisc with more exposed parameters for the user to manipulate.

6.4. ESFQ, Extended Stochastic Fair Queuing

Conceptually, this qdisc is no different than SFQ although it allows the user to control more parameters than its simpler cousin. This qdisc was conceived to overcome the shortcoming of SFQ identified above. By allowing the user to control which hashing algorithm is used for distributing access to network bandwidth, it is possible for the user to reach a fairer real distribution of bandwidth.

Example 8. ESFQ usage

```
Usage: ... esfq [ perturb SECS ] [ quantum BYTES ] [ depth FLOWS ]
        [ divisor HASHBITS ] [ limit PKTS ] [ hash HASHTYPE]

Where:
HASHTYPE := { classic | src | dst }
```

FIXME; need practical experience and/or attestation here.

6.5. GRED, Generic Random Early Drop

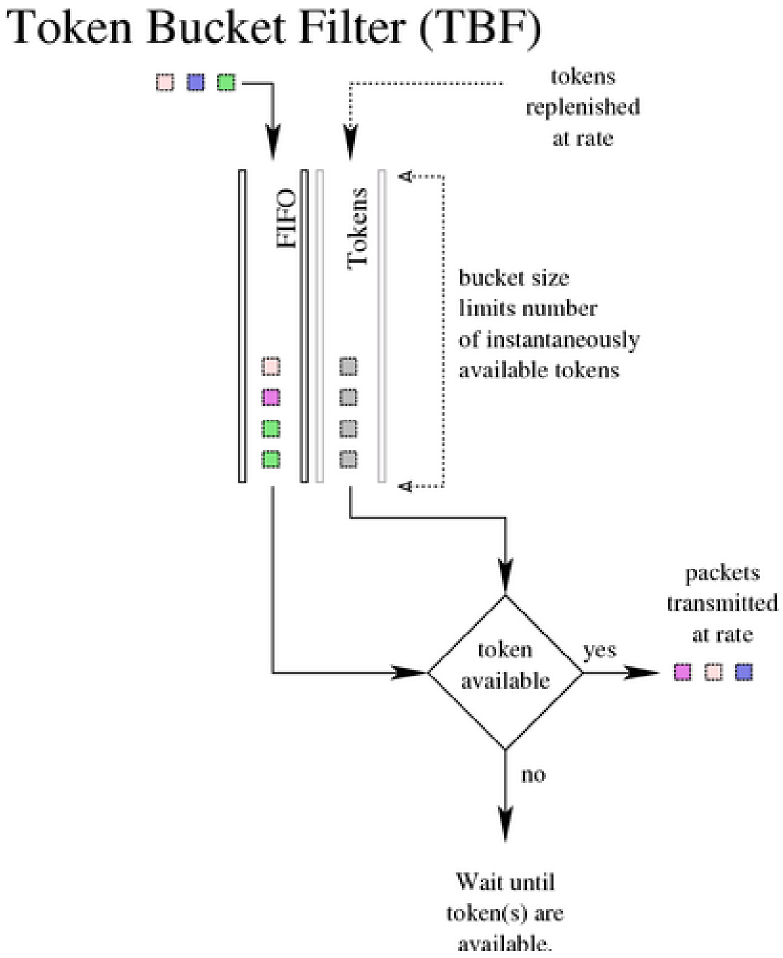
FIXME; I have never used this. Need practical experience or attestation.

Theory declares that a RED algorithm is useful on a backbone or core network, but not as useful near the end−user. See the section on flows to see a general discussion of the thirstiness of TCP.

6.6. TBF, Token Bucket Filter

This qdisc is built on tokens and buckets. It simply shapes traffic transmitted on an interface. To limit the speed at which packets will be dequeued from a particular interface, the TBF qdisc is the perfect solution. It simply slows down transmitted traffic to the specified rate.

Packets are only transmitted if there are sufficient tokens available. Otherwise, packets are deferred. Delaying packets in this fashion will introduce an artificial latency into the packet's round trip time.


Token Bucket Filter (TBF)
tokens replenished at rate
FIFO
Tokens
bucket size limits number of instantaneously available tokens
token available
yes
packets transmitted at rate
no
Wait until token(s) are available.

Example 9. Creating a 256kbit/s TBF

```
[root@leander]# cat tbf.tcc
/*
 * make a 256kbit/s TBF on eth0
 *
 */

dev eth0 {
    egress {
        tbf( rate 256 kbps, burst 20 kB, limit 20 kB, mtu 1514 B );
    }
}
[root@leander]# tcc < tbf.tcc
# ================================ Device eth0 ================================

tc qdisc add dev eth0 handle 1:0 root dsmark indices 1 default_index 0
tc qdisc add dev eth0 handle 2:0 parent 1:0 tbf burst 20480 limit 20480 mtu 1514 rate 32000bps
```

7. Classful Queuing Disciplines (`qdisc`s)

The flexibility and control of Linux traffic control can be unleashed through the agency of the classful qdiscs. Remember that the classful queuing disciplines can have filters attached to them, allowing packets to be directed to particular classes and subqueues.

There are several common terms to describe classes directly attached to the `root` qdisc and terminal classes. Classess attached to the `root` qdisc are known as root classes, and more generically inner classes. Any terminal class in a particular queuing discipline is known as a leaf class by analogy to the tree structure of the classes. Besides the use of figurative language depicting the structure as a tree, the language of family relationships is also quite common.

7.1. HTB, Hierarchical Token Bucket

HTB uses the concepts of tokens and buckets along with the class−based system and `filter`s to allow for complex and granular control over traffic. With a complex borrowing model, HTB can perform a variety of sophisticated traffic control techniques. One of the easiest ways to use HTB immediately is that of shaping.

By understanding tokens and buckets or by grasping the function of TBF, HTB should be merely a logical step. This queuing discipline allows the user to define the characteristics of the tokens and bucket used and allows the user to nest these buckets in an arbitrary fashion. When coupled with a classifying scheme, traffic can be controlled in a very granular fashion.

Below is example output of the syntax for HTB on the command line with the **tc** tool. Although the syntax for **tcng** is a language of its own, the rules for HTB are the same.

Example 10. tc usage for HTB

```
Usage: ... qdisc add ... htb [default N] [r2q N]
 default  minor id of class to which unclassified packets are sent {0}
 r2q      DRR quantums are computed as rate in Bps/r2q {10}
 debug    string of 16 numbers each 0-3 {0}

... class add ... htb rate R1 burst B1 [prio P] [slot S] [pslot PS]
                      [ceil R2] [cburst B2] [mtu MTU] [quantum Q]
 rate     rate allocated to this class (class can still borrow)
 burst    max bytes burst which can be accumulated during idle period {computed}
 ceil     definite upper class rate (no borrows) {rate}
 cburst   burst but for ceil {computed}
 mtu      max packet size we create rate map for {1600}
 prio     priority of leaf; lower are served first {0}
 quantum  how much bytes to serve from leaf at once {use r2q}

TC HTB version 3.3
```

7.1.1. Software requirements

Unlike almost all of the other software discussed, HTB is a newer queuing discipline and your distribution may not have all of the tools and capability you need to use HTB. The kernel must support HTB; kernel version 2.4.20 and later support it in the stock distribution, although earlier kernel versions require patching.

To enable userland support for HTB, see HTB for an **iproute2** patch to **tc**.

7.1.2. Shaping

One of the most common applications of HTB involves shaping transmitted traffic to a specific rate.

All shaping occurs in leaf classes. No shaping occurs in inner or root classes as they only exist to suggest how the borrowing model should distribute available tokens.

7.1.3. Borrowing

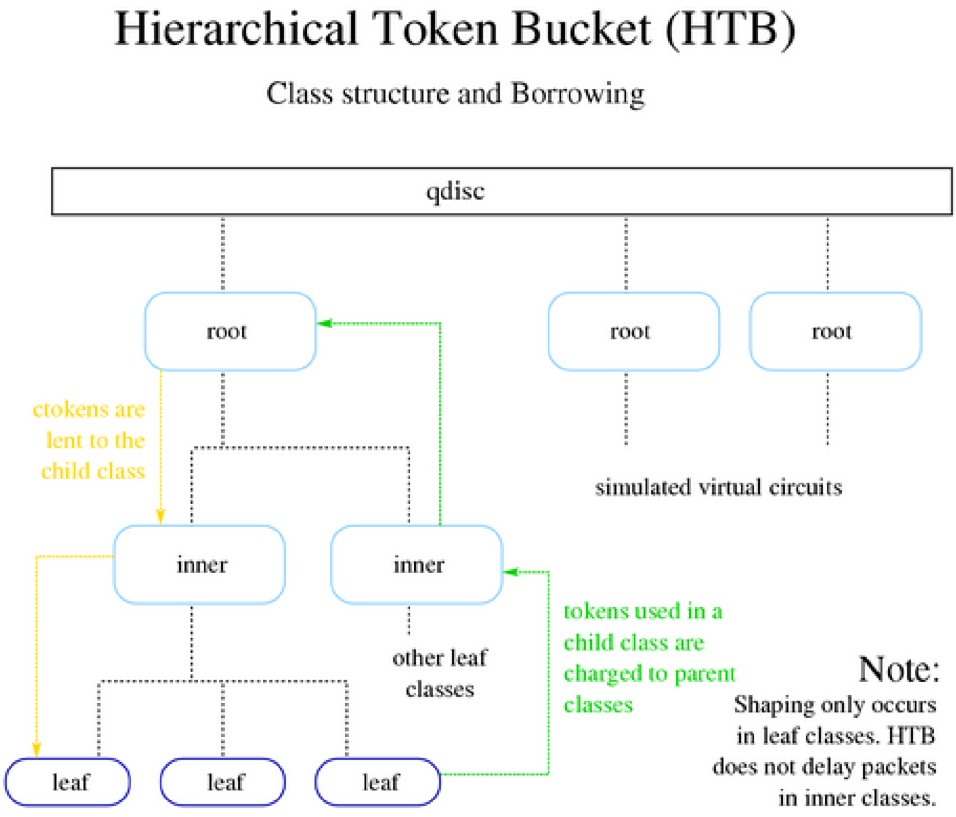
A fundamental part of the HTB qdisc is the borrowing mechanism. Children classes borrow tokens from their parents once they have exceeded `rate`. A child class will continue to attempt to borrow until it reaches `ceil`, at which point it will begin to queue packets for transmission until more tokens/ctokens are available. As there are only two primary types of classes which can be created with HTB the following table and diagram identify the various possible states and the behaviour of the borrowing mechanisms.

Table 2. HTB class states and potential actions taken

type of class	**class state**	**HTB internal state**	**action taken**
leaf	< *rate*	*HTB_CAN_SEND*	Leaf class will dequeue queued bytes up to available tokens (no more than burst packets)
leaf	> *rate*, < *ceil*	*HTB_MAY_BORROW*	Leaf class will attempt to borrow tokens/ctokens from parent class. If tokens are available, they will be lent in *quantum* increments and the leaf class will dequeue up to *cburst* bytes
leaf	> *ceil*	*HTB_CANT_SEND*	No packets will be dequeued. This will cause packet delay and will increase latency to meet the desired rate.
inner, root	< *rate*	*HTB_CAN_SEND*	Inner class will lend tokens to children.
inner, root	> *rate*, < *ceil*	*HTB_MAY_BORROW*	Inner class will attempt to borrow tokens/ctokens from parent class, lending them to competing children in *quantum* increments per request.
inner, root	> *ceil*	*HTB_CANT_SEND*	Inner class will not attempt to borrow from its parent and will not lend tokens/ctokens to children classes.

This diagram identifies the flow of borrowed tokens and the manner in which tokens are charged to parent classes. In order for the borrowing model to work, each class must have an accurate count of the number of tokens used by itself and all of its children. For this reason, any token used in a child or leaf class is charged to each parent class until the root class is reached.

Any child class which wishes to borrow a token will request a token from its parent class, which if it is also over its `rate` will request to borrow from its parent class until either a token is located or the root class is reached. So the borrowing of tokens flows toward the leaf classes and the charging of the usage of tokens flows toward the root class.


Hierarchical Token Bucket (HTB)
Class structure and Borrowing
qdisc
root
root
root
ctokens are lent to the child class
inner
inner
simulated virtual circuits
other leaf classes
tokens used in a child class are charged to parent classes
Note:
Shaping only occurs in leaf classes. HTB does not delay packets in inner classes.
leaf
leaf
leaf

Note in this diagram that there are several HTB root classes. Each of these root classes can simulate a virtual circuit.

7.1.4. HTB class parameters

`default`
: An optional parameter with every HTB `qdisc` object, the default *`default`* is 0, which cause any unclassified traffic to be dequeued at hardware speed, completely bypassing any of the classes attached to the `root` qdisc.

`rate`
: Used to set the minimum desired speed to which to limit transmitted traffic. This can be considered the equivalent of a committed information rate (CIR), or the guaranteed bandwidth for a given leaf class.

`ceil`
: Used to set the maximum desired speed to which to limit the transmitted traffic. The borrowing model should illustrate how this parameter is used. This can be considered the equivalent of "burstable bandwidth".

`burst`
: This is the size of the *`rate`* bucket (see Tokens and buckets). HTB will dequeue *`burst`* bytes before awaiting the arrival of more tokens.

`cburst`
: This is the size of the *`ceil`* bucket (see Tokens and buckets). HTB will dequeue *`cburst`* bytes before awaiting the arrival of more ctokens.

`quantum`
: This is a key parameter used by HTB to control borrowing. Normally, the correct *`quantum`* is

calculated by HTB, not specified by the user. Tweaking this parameter can have tremendous effects on borrowing and shaping under contention, because it is used both to split traffic between children classes over *rate* (but below *ceil*) and to transmit packets from these same classes.

r2q

Also, usually calculated for the user, *r2q* is a hint to HTB to help determine the optimal *quantum* for a particular class.

mtu

prio

7.1.5. Rules

Below are some general guidelines to using HTB culled from http://docum.org/ and the LARTC mailing list. These rules are simply a recommendation for beginners to maximize the benefit of HTB until gaining a better understanding of the practical application of HTB.

- Shaping with HTB occurs only in leaf classes. See also Section 7.1.2.
- Because HTB does not shape in any class except the leaf class, the sum of the *rate*s of leaf classes should not exceed the *ceil* of a parent class. Ideally, the sum of the *rate*s of the children classes would match the *rate* of the parent class, allowing the parent class to distribute leftover bandwidth (*ceil* − *rate*) among the children classes.

  This key concept in employing HTB bears repeating. Only leaf classes actually shape packets; packets are only delayed in these leaf classes. The inner classes (all the way up to the root class) exist to define how borrowing/lending occurs (see also Section 7.1.3).
- The *quantum* is only only used when a class is over *rate* but below *ceil*.
- The *quantum* should be set at MTU or higher. HTB will dequeue a single packet at least per service opportunity even if *quantum* is too small. In such a case, it will not be able to calculate accurately the real bandwidth consumed [9].
- Parent classes lend tokens to children in increments of *quantum*, so for maximum granularity and most instantaneously evenly distributed bandwidth, *quantum* should be as low as possible while still no less than MTU.
- A distinction between tokens and ctokens is only meaningful in a leaf class, because non−leaf classes only lend tokens to child classes.
- HTB borrowing could more accurately be described as "using".

7.2. HFSC, Hierarchical Fair Service Curve

The HFSC classful qdisc balances delay−sensitive traffic against throughput sensitive traffic. In a congested or backlogged state, the HFSC queuing discipline interleaves the delay−sensitive traffic when required according service curve definitions. Read about the Linux implementation in German, HFSC Scheduling mit Linux or read a translation into English, HFSC Scheduling with Linux. The original research article, A Hierarchical Fair Service Curve Algorithm For Link−Sharing, Real−Time and Priority Services, also remains available.

This section will be completed at a later date.

7.3. PRIO, priority scheduler

The PRIO classful qdisc works on a very simple precept. When it is ready to dequeue a packet, the first class is checked for a packet. If there's a packet, it gets dequeued. If there's no packet, then the next class is checked, until the queuing mechanism has no more classes to check.

This section will be completed at a later date.

7.4. CBQ, Class Based Queuing

CBQ is the classic implementation (also called venerable) of a traffic control system. This section will be completed at a later date.

8. Rules, Guidelines and Approaches

8.1. General Rules of Linux Traffic Control

There are a few general rules which ease the study of Linux traffic control. Traffic control structures under Linux are the same whether the initial configuration has been done with **tcng** or with **tc**.

- Any router performing a shaping function should be the bottleneck on the link, and should be shaping slightly below the maximum available link bandwidth. This prevents queues from forming in other routers, affording maximum control of packet latency/deferral to the shaping device.
- A device can only shape traffic it transmits [10]. Because the traffic has already been received on an input interface, the traffic cannot be shaped. A traditional solution to this problem is an ingress policer.
- Every interface must have a `qdisc`. The default qdisc (the `pfifo_fast` qdisc) is used when another qdisc is not explicitly attached to the interface.
- One of the classful qdiscs added to an interface with no children classes typically only consumes CPU for no benefit.
- Any newly created class contains a FIFO. This qdisc can be replaced explicitly with any other qdisc. The FIFO qdisc will be removed implicitly if a child class is attached to this class.
- Classes directly attached to the `root` qdisc can be used to simulate virtual circuits.
- A `filter` can be attached to classes or one of the classful qdiscs.

8.2. Handling a link with a known bandwidth

HTB is an ideal `qdisc` to use on a link with a known bandwidth, because the innermost (root−most) class can be set to the maximum bandwidth available on a given link. Flows can be further subdivided into children classes, allowing either guaranteed bandwidth to particular classes of traffic or allowing preference to specific kinds of traffic.

8.3. Handling a link with a variable (or unknown) bandwidth

In theory, the PRIO scheduler is an ideal match for links with variable bandwidth, because it is a work−conserving `qdisc` (which means that it provides no shaping). In the case of a link with an unknown or fluctuating bandwidth, the PRIO scheduler simply prefers to dequeue any available packet in the highest priority band first, then falling to the lower priority queues.

8.4. Sharing/splitting bandwidth based on flows

Of the many types of contention for network bandwidth, this is one of the easier types of contention to address in general. By using the SFQ qdisc, traffic in a particular queue can be separated into flows, each of which will be serviced fairly (inside that queue). Well−behaved applications (and users) will find that using SFQ and ESFQ are sufficient for most sharing needs.

The Achilles heel of these fair queuing algorithms is a misbehaving user or application which opens many connections simultaneously (e.g., eMule, eDonkey, Kazaa). By creating a large number of individual flows, the application can dominate slots in the fair queuing algorithm. Restated, the fair queuing algorithm has no idea that a single application is generating the majority of the flows, and cannot penalize the user. Other methods are called for.

8.5. Sharing/splitting bandwidth based on IP

For many administrators this is the ideal method of dividing bandwidth amongst their users. Unfortunately, there is no easy solution, and it becomes increasingly complex with the number of machine sharing a network link.

To divide bandwidth equitably between *N* IP addresses, there must be *N* classes.

9. Scripts for use with QoS/Traffic Control

9.1. wondershaper

More to come, see wondershaper.

9.2. ADSL Bandwidth HOWTO script (`myshaper`)

More to come, see myshaper.

9.3. `htb.init`

More to come, see `htb.init`.

9.4. `tcng.init`

More to come, see `tcng.init`.

9.5. `cbq.init`

More to come, see `cbq.init`.

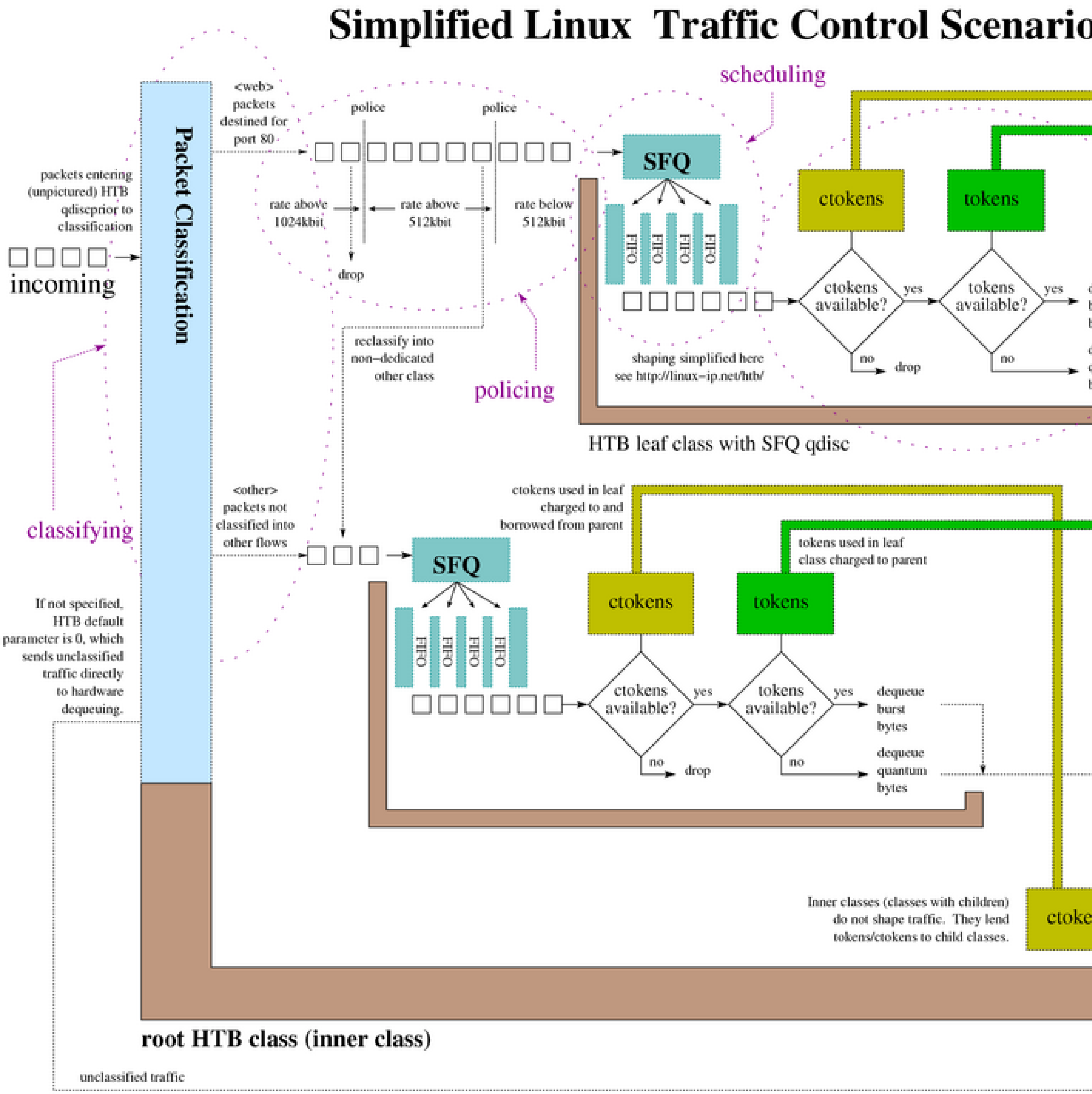
10. Diagram

10.1. General diagram

Below is a general diagram of the relationships of the components of a classful queuing discipline (HTB pictured). A larger version of the diagram is available.

Example 11. An example HTB tcng configuration

```
/*
 *
 *  possible mock up of diagram shown at
 *  http://linux-ip.net/traffic-control/htb-class.png
 *
 */

$m_web = trTCM (
                 cir 512  kbps,  /* commited information rate */
                 cbs 10   kB,    /* burst for CIR */
                 pir 1024 kbps,  /* peak information rate */
                 pbs 10   kB     /* burst for PIR */
               ) ;

dev eth0 {
    egress {

        class ( <$web> )  if tcp_dport == PORT_HTTP &&  __trTCM_green( $m_web );
        class ( <$bulk> ) if tcp_dport == PORT_HTTP && __trTCM_yellow( $m_web );
        drop              if                              __trTCM_red( $m_web );
        class ( <$bulk> ) if tcp_dport == PORT_SSH ;

        htb () {  /* root qdisc */

            class ( rate 1544kbps, ceil 1544kbps ) {  /* root class */

                $web  = class ( rate 512kbps, ceil  512kbps ) { sfq ; } ;
                $bulk = class ( rate 512kbps, ceil 1544kbps ) { sfq ; } ;

            }
        }
    }
}
```


Simplified Linux Traffic Control Scenario
scheduling
packets entering
(unpictured) HTB
qdiscprior to
classification
incoming
Packet Classification
<web>
packets
destined for
port 80
police
police
rate above
1024kbit
rate above
512kbit
rate below
512kbit
drop
reclassify into
non−dedicated
other class
policing
SFQ
FIFO
FIFO
FIFO
FIFO
ctokens
tokens
ctokens
available?
yes
tokens
available?
yes
no
drop
no
shaping simplified here
see http://linux−ip.net/htb/
HTB leaf class with SFQ qdisc
classifying
<other>
packets not
classified into
other flows
ctokens used in leaf
charged to and
borrowed from parent
tokens used in leaf
class charged to parent
SFQ
ctokens
tokens
FIFO
FIFO
FIFO
FIFO
If not specified,
HTB default
parameter is 0, which
sends unclassified
traffic directly
to hardware
dequeuing.
ctokens
available?
yes
tokens
available?
yes
dequeue
burst
bytes
no
drop
no
dequeue
quantum
bytes
Inner classes (classes with children)
do not shape traffic. They lend
tokens/ctokens to child classes.
ctoke
root HTB class (inner class)
unclassified traffic

11. Annotated Traffic Control Links

This section identifies a number of links to documentation about traffic control and Linux traffic control software. Each link will be listed with a brief description of the content at that site.

- HTB site, HTB user guide and HTB theory (*Martin "devik" Devera*)

  Hierarchical Token Bucket, HTB, is a classful queuing discipline. Widely used and supported it is also fairly well documented in the user guide and at Stef Coene's site (see below).
- General Quality of Service docs (*Leonardo Balliache*)

  There is a good deal of understandable and introductory documentation on his site, and in particular has some excellent overview material. See in particular, the detailed Linux QoS document among others.
- **tcng** (Traffic Control Next Generation) and **tcng** manual (*Werner Almesberger*)

  The **tcng** software includes a language and a set of tools for creating and testing traffic control structures. In addition to generating **tc** commands as output, it is also capable of providing output for non−Linux applications. A key piece of the **tcng** suite which is ignored in this documentation is the **tcsim** traffic control simulator.

  The user manual provided with the **tcng** software has been converted to HTML with **latex2html**. The distribution comes with the TeX documentation.
- **iproute2** and **iproute2** manual (*Alexey Kuznetsov*)

  This is a the source code for the **iproute2** suite, which includes the essential **tc** binary. Note, that as of iproute2−2.4.7−now−ss020116−try.tar.gz, the package did not support HTB, so a patch available from the HTB site will be required.

  The manual documents the entire suite of tools, although the **tc** utility is not adequately documented here. The ambitious reader is recommended to the LARTC HOWTO after consuming this introduction.
- Documentation, graphs, scripts and guidelines to traffic control under Linux (*Stef Coene*)

  Stef Coene has been gathering statistics and test results, scripts and tips for the use of QoS under Linux. There are some particularly useful graphs and guidelines available for implementing traffic control at Stef's site.
- LARTC HOWTO (*bert hubert, et. al.*)

  The Linux Advanced Routing and Traffic Control HOWTO is one of the key sources of data about the sophisticated techniques which are available for use under Linux. The Traffic Control Introduction HOWTO should provide the reader with enough background in the language and concepts of traffic control. The LARTC HOWTO is the next place the reader should look for general traffic control information.
- Guide to IP Networking with Linux (*Martin A. Brown*)

  Not directly related to traffic control, this site includes articles and general documentation on the behaviour of the Linux IP layer.
- Werner Almesberger's Papers

Werner Almesberger is one of the main developers and champions of traffic control under Linux (he's also the author of **tcng**, above). One of the key documents describing the entire traffic control architecture of the Linux kernel is his Linux Traffic Control – Implementation Overview which is available in PDF or PS format.

- Linux DiffServ project

  Mercilessly snipped from the main page of the DiffServ site...

  > Differentiated Services (short: Diffserv) is an architecture for providing different types or levels of service for network traffic. One key characteristic of Diffserv is that flows are aggregated in the network, so that core routers only need to distinguish a comparably small number of aggregated flows, even if those flows contain thousands or millions of individual flows.

Notes

[1] See Section 5 for more details on the use or installation of a particular traffic control mechanism, kernel or command line utility.

[2] This queueing model has long been used in civilized countries to distribute scant food or provisions equitably. William Faulkner is reputed to have walked to the front of the line for to fetch his share of ice, proving that not everybody likes the FIFO model, and providing us a model for considering priority queuing.

[3] Similarly, the entire traffic control system appears as a queue or scheduler to the higher layer which is enqueuing packets into this layer.

[4] This smoothing effect is not always desirable, hence the HTB parameters burst and cburst.

[5] A classful qdisc can only have children classes of its type. For example, an HTB qdisc can only have HTB classes as children. A CBQ qdisc cannot have HTB classes as children.

[6] In this case, you'll have a `filter` which uses a `classifier` to select the packets you wish to drop. Then you'll use a `policer` with a with a drop action like this **police rate 1bps burst 1 action drop/drop**.

[7] I do not know the range nor base of these numbers. I believe they are u32 hexadecimal, but need to confirm this.

[8] The options listed in this example are taken from a 2.4.20 kernel source tree. The exact options may differ slightly from kernel release to kernel release depending on patches and new schedulers and classifiers.

[9] HTB will report bandwidth usage in this scenario incorrectly. It will calculate the bandwidth used by *`quantum`* instead of the real dequeued packet size. This can skew results quickly.

[10] In fact, the Intermediate Queuing Device (IMQ) simulates an output device onto which traffic control structures can be attached. This clever solution allows a networking device to shape ingress traffic in the same fashion as egress traffic. Despite the apparent contradiction of the rule, IMQ appears as a device to the kernel. Thus, there has been no violation of the rule, but rather a sneaky reinterpretation of that rule.