

Traffic Control HOWTO

Version 1.0.2

Author: Martin A. Brown

linux-ip.net

`<martin@linux-ip.net>`

© 2006, Martin A. Brown

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no invariant sections, with no Front-Cover Texts, with no Back-Cover Text. A copy of the license is located at <http://www.gnu.org/licenses/fdl.html>.

"Oct 2006"

Abstract

Traffic control encompasses the sets of mechanisms and operations by which packets are queued for transmission/reception on a network interface. The operations include enqueueing, 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

1. Introduction to Linux Traffic Control

- 1.1. Target audience and assumptions about the reader
- 1.2. Conventions
- 1.3. Recommended approach
- 1.4. Missing content, corrections and feedback

2. Overview of Concepts

- 2.1. What is it?
- 2.2. Why use it?
- 2.3. Advantages
- 2.4. Disdvantages
- 2.5. Queues
- 2.6. Flows
- 2.7. Tokens and buckets

2.8. Packets and frames

3. Traditional Elements of Traffic Control

3.1. Shaping

3.2. Scheduling

3.3. Classifying

3.4. Policing

3.5. Dropping

3.6. Marking

4. Components of Linux Traffic Control

4.1. `qdisc`

4.2. `class`

4.3. `filter`

4.4. `classifier`

4.5. `policer`

4.6. `drop`

4.7. `handle`

5. Software and Tools

5.1. Kernel requirements

5.2. `iproute2` tools (`tc`)

5.3. `tcng`, Traffic Control Next Generation

5.4. IMQ, Intermediate Queuing device

6. Classless Queuing Disciplines (`qdiscs`)

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

6.2. `pfifo_fast`, the default Linux `qdisc`

6.3. SFQ, Stochastic Fair Queuing

6.4. ESFQ, Extended Stochastic Fair Queuing

6.5. GRED, Generic Random Early Drop

6.6. TBF, Token Bucket Filter

7. Classful Queuing Disciplines (`qdiscs`)

7.1. HTB, Hierarchical Token Bucket

7.2. HFSC, Hierarchical Fair Service Curve

7.3. PRIO, priority scheduler

7.4. CBQ, Class Based Queuing

8. Rules, Guidelines and Approaches

8.1. General Rules of Linux Traffic Control

8.2. Handling a link with a known bandwidth

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

8.4. Sharing/splitting bandwidth based on flows

8.5. Sharing/splitting bandwidth based on IP

9. Scripts for use with QoS/Traffic Control

9.1. `wondershaper`

9.2. ADSL Bandwidth HOWTO script (`myshaper`)

9.3. `htb.init`

9.4. `tcng.init`

9.5. `cbq.init`

10. Diagram

10.1. General diagram

11. Annotated Traffic Control Links

Next

**1. Introduction to Linux Traffic
Control**

1. Introduction to Linux Traffic Control

[Prev](#)[Next](#)

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, “Kernel requirements”](#)), 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>](mailto: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.

^[1] See [Section 5, “Software and Tools”](#) for more details on the use or installation of a particular traffic control mechanism, kernel or command line utility.

2. Overview of Concepts

[Prev](#)[Next](#)

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.



Note

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, “Queues”](#)). 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.

^[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 queueing.

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

3. Traditional Elements of Traffic Control

[Prev](#)[Next](#)

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, "Tokens and buckets"](#) 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](#), “`filter`”) 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](#), “`policer`”).

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 enqueueing). 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.



Note

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

[Prev](#)

[Next](#)

2. Overview of Concepts

[Home](#)

4. Components of Linux Traffic Control

4. Components of Linux Traffic Control

[Prev](#)[Next](#)

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, “Scheduling”](#)). 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 [classes](#), and provide a handle to which to attach [filters](#). 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 ([qdiscs](#)) with potential [classes](#) 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 `filters` 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, “Classifying”](#)) 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 qdiscs or to classes, 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 classifier 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, “Diagram”](#).

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, “tc filter”](#).

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.



Note

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, “Classful Queuing Disciplines \(qdiscs\)”](#)) 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] 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 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.

[Prev](#)

[Next](#)

[3. Traditional Elements of Traffic Control](#)

[Home](#)

[5. Software and Tools](#)

5. Software and Tools

[Prev](#)[Next](#)

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 [qdiscs](#) you wish to use. In particular, the HTB [qdisc](#) is not supported yet in the upstream **iproute2** package. See [Section 7.1, “HTB, Hierarchical Token Bucket”](#) 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.

4

The `handle` is a user-specified number of the form `major:minor`. The minor number for any queuing 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.

5

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 \ 1
> dev eth0 \ 2
> parent 1:1 \ 3
> classid 1:6 \ 4
> htb \ 5
> rate 256kbit \ 6
> ceil 512kbit \ 7
```

1

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

2

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

3

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

4

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

5

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.

6 7

This is a class specific parameter. Consult [Section 7.1, “HTB, Hierarchical Token Bucket”](#) for more detail on these parameters.

Example 5. `tc filter`

```
[root@leander]# tc filter add \ 1
```

```

> dev eth0 \ 2
> parent 1:0 \ 3
> protocol ip \ 4
> prio 5 \ 5
> u32 \ 6
> match ip port 22 0xffff \ 7
> match ip tos 0x10 0xff \ 8
> flowid 1:6 \ 9
> police \ A
> rate 32000bps \ B
> burst 10240 \ C
> mpu 0 \ D
> action drop/continue E

```

1

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

2

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

3

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

4

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

5

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

6

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

7 8

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.

9

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

A

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

B

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

C

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 surprise to the reader that there exists an easier way to configure Linux traffic control. See the next section, [Section 5.3, “tcng, Traffic Control Next Generation”](#).

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](#).

[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.

[Prev](#)

[Next](#)

[4. Components of Linux Traffic Control](#)

[Home](#)

[6. Classless Queuing Disciplines](#)
(qdiscs)

6. Classless Queuing Disciplines (qdiscs)

[Prev](#)[Next](#)

6. Classless Queuing Disciplines (qdiscs)

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](#).

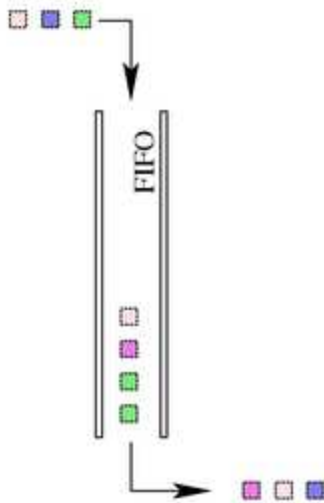
6.1. FIFO, First-In First-Out (pfifo and bfifo)

**Note**

This is not the default qdisc on Linux interfaces. Be certain to see [Section 6.2](#), “[pfifo_fast](#), the default Linux qdisc” 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.

First-in First-out (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 qdiscs, 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 );
    }
}

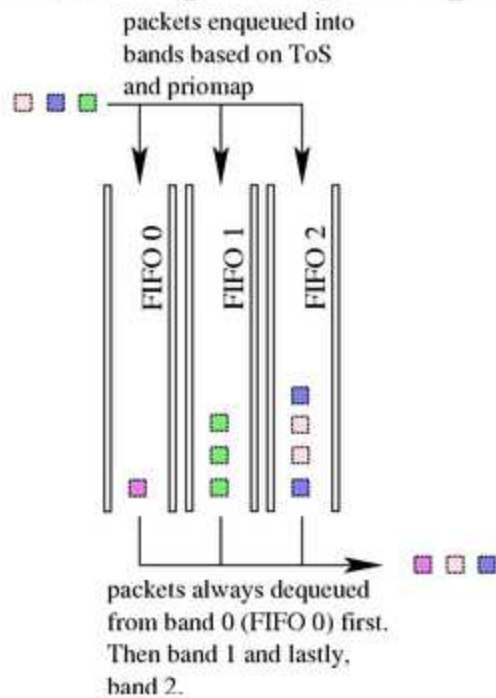
[root@leander]# tcc < pfifo.tcc
# ===== Device eth0 =====

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

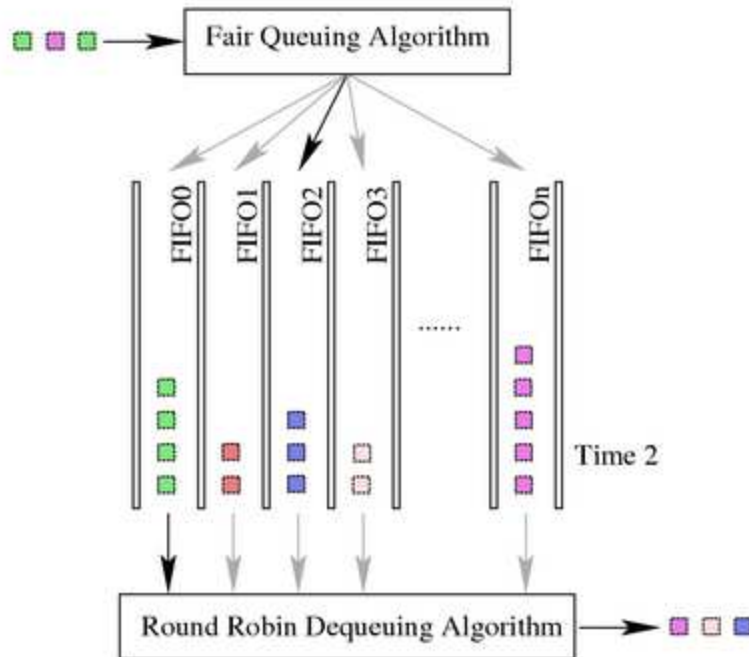


There is nothing configurable to the end user about the `pfifo_fast` qdisc. For exact details on the `priomap` and use of the ToS bits, see the [pfifo-fast](#) section of the [LARTC HOWTO](#).

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)



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, “ESFQ, Extended Stochastic Fair Queuing”](#) 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.

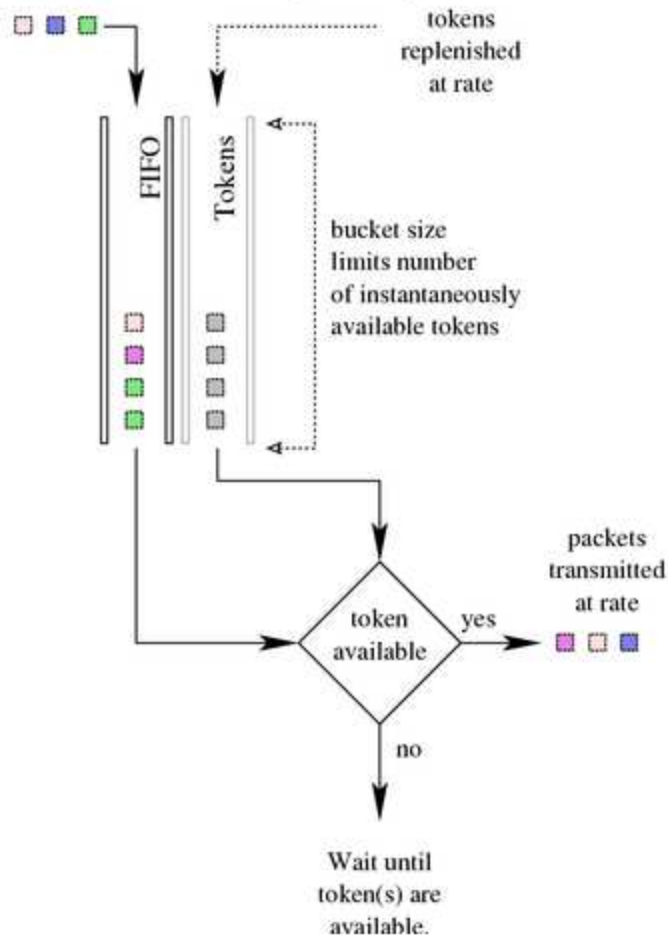
Theory declares that a RED algorithm is useful on a backbone or core network, but not as useful near the end-user. See the section on [flows](#) to see a general discussion of the thirstiness of TCP.

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)



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
```

[Prev](#)
[Next](#)
[5. Software and Tools](#)
[Home](#)
[7. Classful Queuing Disciplines \(qdiscs\)](#)

7. Classful Queuing Disciplines (qdiscs)

[Prev](#)[Next](#)

7. Classful Queuing Disciplines (qdiscs)

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. Classes 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 `filters` 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 quantum is 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)

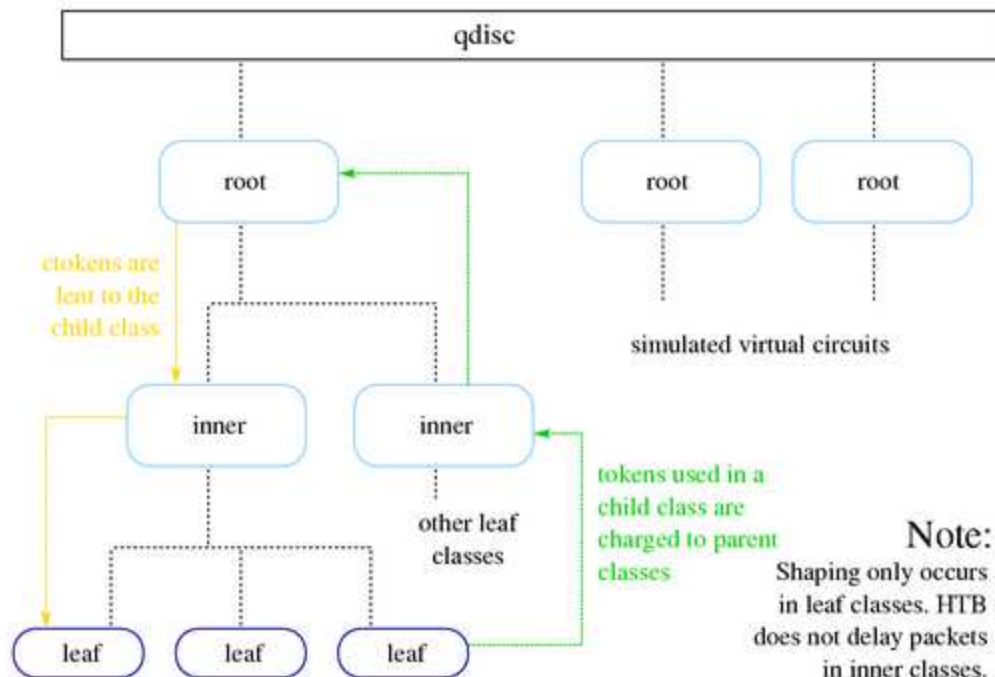
type of class	class state	HTB internal state	action taken
leaf	$> rate$, $< ceil$	<i>HTB_MAY_BORROW</i>	Leaf class will attempt to borrow tokens/ctokens from parent class. If tokens are available, they will be lent in <i>quantum</i> increments and the leaf class will dequeue up to <i>cburst</i> bytes
leaf	$> ceil$	<i>HTB_CANT_SEND</i>	No packets will be dequeued. This will cause packet delay and will increase latency to meet the desired rate.
inner, root	$< rate$	<i>HTB_CAN_SEND</i>	Inner class will lend tokens to children.
inner, root	$> rate$, $< ceil$	<i>HTB_MAY_BORROW</i>	Inner class will attempt to borrow tokens/ctokens from parent class, lending them to competing children in <i>quantum</i> increments per request.
inner, root	$> ceil$	<i>HTB_CANT_SEND</i>	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



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 tokens.

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, “Shaping”](#).
- Because HTB does not shape in any class except the leaf class, the sum of the *rates* of leaf classes should not exceed the *ceil* of a parent class. Ideally, the sum of the *rates* 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, “Borrowing”](#)).

- The *quantum* is 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.

[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.

[Prev](#)

[Next](#)

6. Classless Queuing Disciplines
(qdiscs)

[Home](#)

8. Rules, Guidelines and Approaches

8. Rules, Guidelines and Approaches

[Prev](#)[Next](#)

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.

^[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.

[Prev](#)

[7. Classful Queuing Disciplines \(qdiscs\)](#)

[Home](#)

[Next](#)

[9. Scripts for use with QoS/Traffic Control](#)

9. Scripts for use with QoS/Traffic Control

[Prev](#)[Next](#)

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](#).

[Prev](#)[Next](#)[8. Rules, Guidelines and Approaches](#)[Home](#)[10. Diagram](#)

10. Diagram

[Prev](#)[Next](#)

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, /* committed 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 */

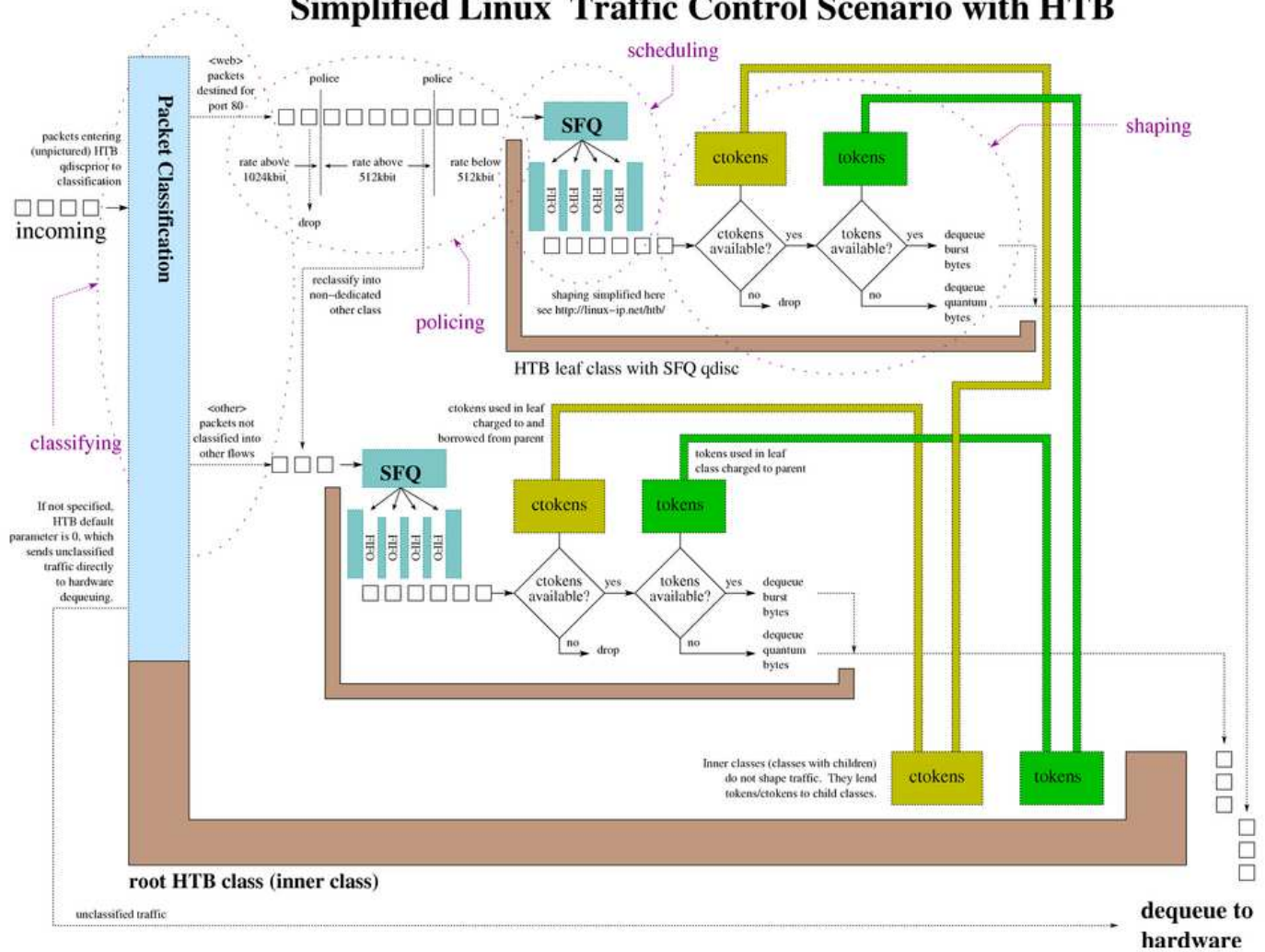
                $web = class ( rate 512kbps, ceil 512kbps ) { sfq ; } ;
                $bulk = class ( rate 512kbps, ceil 1544kbps ) { sfq ; } ;

            }

        }

    }
}
```

Simplified Linux Traffic Control Scenario with HTB


[Prev](#)
[9. Scripts for use with QoS/Traffic Control](#)
[Home](#)
[Next](#)
[11. Annotated Traffic Control Links](#)

11. Annotated Traffic Control Links

[Prev](#)

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.

[Prev](#)

[10. Diagram](#)

[Home](#)

HTB Linux queuing discipline manual - user guide

Martin Devera aka devik (devik@cdi.cz)

Manual: devik and Don Cohen

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.

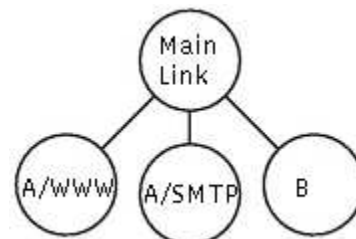
This document shows you how to use HTB. Most sections have examples, charts (with measured data) and discussion of particular problems.

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 kilobytes 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
```

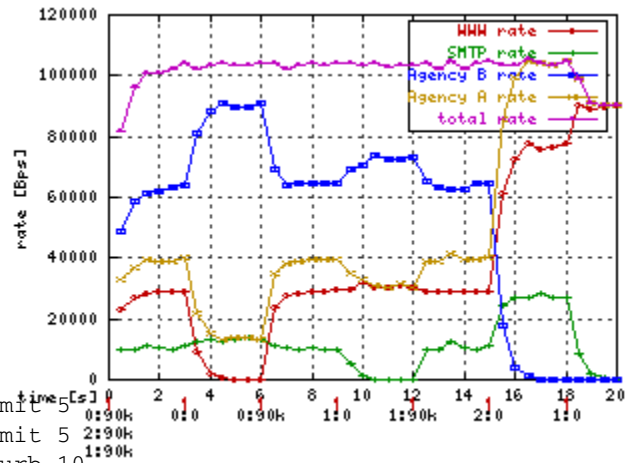
(We identify A by its IP address which we imagine here to be 1.2.3.4.)

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 quanta 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 quanta. It is important to know that for precise operation quanta need to be as small as possible and larger than MTU.

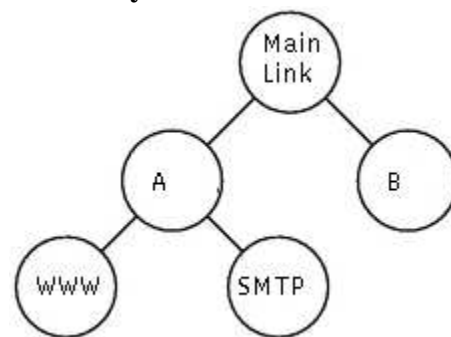
Normally you don't need to specify quanta manually as HTB chooses precomputed values. It computes class's quantum (when you add or change it) as its rate divided by r2q global parameter. Its default value is 10 and because typical MTU is 1500 the default is good for rates from 15 kbps (120 kbit). For 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 manually 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 minimum 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
```

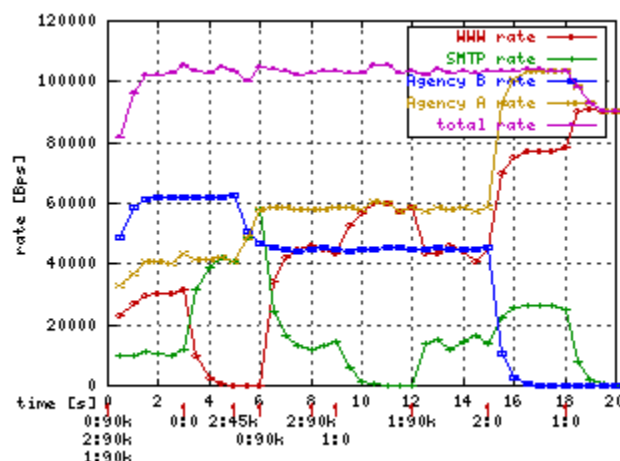
We now turn to the graph showing the results of the hierarchical solution. When A's WWW traffic stops, its assigned bandwidth is reallocated to A's other traffic so that A's total bandwidth is still the assigned 40kbps. If A were to request less than 40kbs in total then the excess would be given to B.

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.

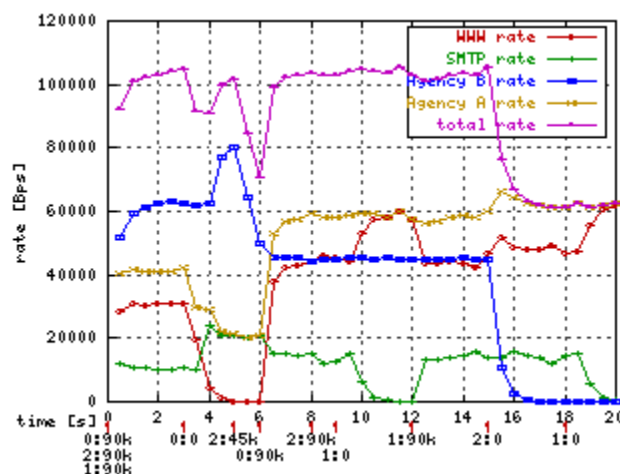
The second difference is at time 15 when B stops. Without the ceil, all of its bandwidth was given to A,



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.

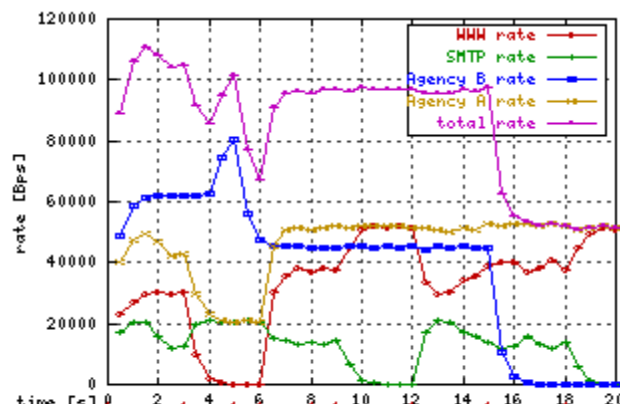
When you set burst for parent class smaller than for some child then you should expect the parent class to get stuck sometimes (because child will drain more than parent can handle). HTB will remember these negative bursts up to 1 minute.

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 its 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 further bursts.

There is at least one unwanted artifact - magenta crater at time 4. It is because I intentionally "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 $\text{max_rate} \times \text{timer_resolution}$. 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 distributed 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 guaranteed rate and ceil are still met.

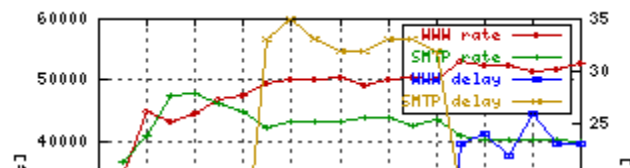
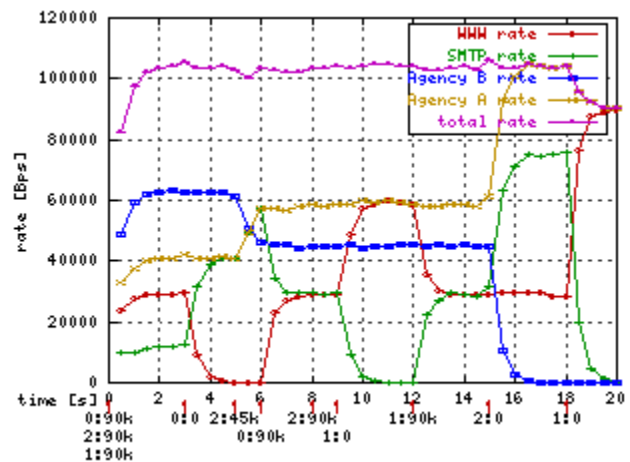
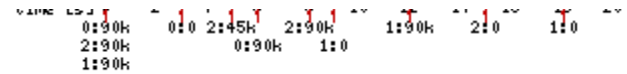
There is also second face of problem. It is total delay of packet. It is relatively hard to measure on ethernet which is too fast (delay is so negligible). But there is simple help. We can add simple HTB with one class rate limiting to less then 100 kbps and add second HTB (the one we are measuring) 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
```

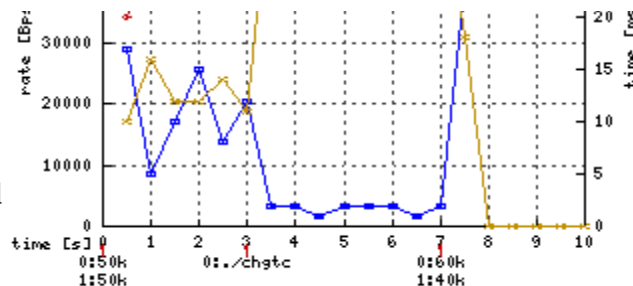
Note: HTB as child of another HTB is NOT the same as class under another class within the same HTB. It is because when class in HTB can send it will send as soon as hardware equipment can. So that delay of underlimit



*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 prioritize 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 prioritizes underlimit part of bandwidth first.

What class should you prioritize ? 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 prioritize 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 self 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
    lendeds: 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
    lendeds: 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 bziping) along with description of problem and its time.

Hierarchical token bucket theory

Martin Devera aka devik (devik@cdi.cz)

Last change: 5.5.2002

Definitions

Let's define HTB's goal formally. Some definitions first:

- Class has associated assured rate AR, ceil rate CR, priority P, level and quantum Q. It can has parent. We also know value of actual rate or R. It is rate of packet flow leaving the class and is measured over small period. For inner classes it is R sum of all descendant leaves.
- Leaf is class which has no children. Only leaf can hold packet queue.
- Level of class determines its position in hierarchy. Leaves has level 0, root classes LEVEL_COUNT-1 and each inner class has level one less than its parent. See pictures below (LEVEL_COUNT=3 there).
- Mode of class is artificial value which can be computed from R,AR,CR. Possible modes are
 - Red: $R > CR$
 - Yellow: $R \leq CR$ and $R > AR$
 - Green otherwise
- D(c) is list of all backlogged leaves which are descendants of c and where all classes between such leaf (and including it) and c are yellow. In other words D(c) is list of all leaves which would want to borrow from c.

Link sharing goal

Now we can define link sharing goal as definition for R. For each class c it should hold

$$R_c = \min(CR_c, AR_c + B_c) \quad [\text{eq1}]$$

where Bc is rate borrowed from ancestors and is defined as

$$B_c = \frac{Q_c R_p}{\sum [Q_i \text{ over } D(p) \text{ where } P_i = P_c]} \quad \text{iff } \min [P_i \text{ over } D(p)] \geq P_c \quad [\text{eq2}]$$

$$B_c = 0 \quad \text{otherwise} \quad [\text{eq3}]$$

where p is c's parent class. If there is no p then $B_c = 0$. Two definitions for Bc above reflects priority queuing - when there are backlogged descendants with numerically lower prio in D(p) then these should be served, not us. Fraction above shows us that excess rate (R_p) is divided between all leaves on the same priority according to Q values. Because R_p in [2] is defined again by [1] the formulas are recursive.

In more human way we can say that AR's of all classes are maintained if there is demand and no CR's are exceeded. Excess bw is subdivided between backlogged descendant leaves with highest priority according to Q ratios.

Auxiliary delay goal

We also want to ensure class isolation. So that changes in rate of one class should not affect delay in other class unless both are actually borrowing from the same ancestor. Also hi-prio class should have lower delays than lo-prio assuming they are both borrowing from the same level.

CBQ note

If you look at [1] you will see that our goals are more restricted subset of CBQ goals. So that if we can satisfy HTB goals then we also satisfy CBQ's one. Thus HTB is kind of CBQ.

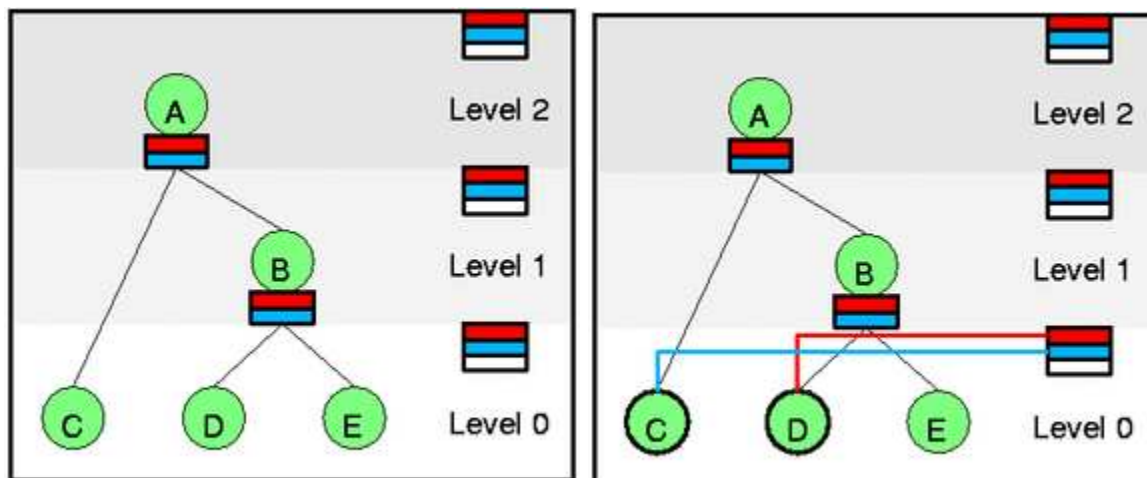
HTB scheduler

Here I will often put Linux implementation specific function or variable names into parens. It will help you to read source code.

There is tree of classes (struct htb_class) in HTB scheduler (struct htb_sched). There is also global so called self feed list (htb_sched::row). It is the rightmost column on pictures. The self feed is comprised of self slots. There is one slot per priority per level. So that there are six self slots on the example (ignore while slot now). Each self slot holds list of classes - the list is depicted by colored line going from the slot to class(es). All classes in slot have the same level and priority as slot has. The self slot contains list of all green classes whose have demand (are in D(c) set).

Each inner (non-leaf) class has inner feed slots (htb_class::inner.feed). Again there is one inner feed slot per priority (red-high, blue-low) and per inner class. As with self slots there is list of classes with the same priority as slot has and the classes must be slot owner's children.

Again the inner slot holds list of yellow children which are in D(c).



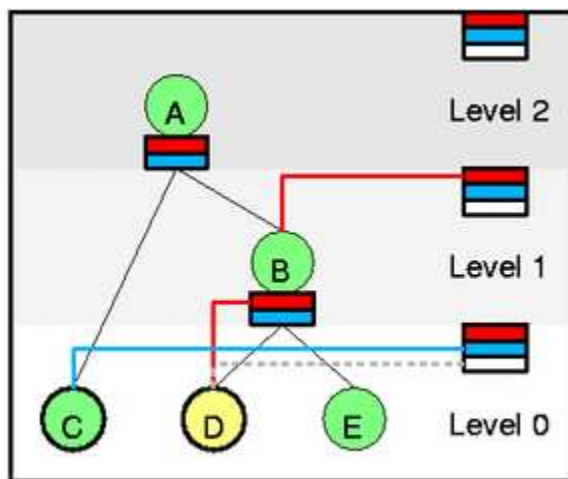
1. no packets here, all classes can send 2. Packets arrived for C and D

The white slot on self feed doesn't really belong there but it was more convenient to draw it as such. It is wait queue per level (htb_sched::wait_pq, htb_class::pq_node). It holds list of all classes on that level which are either red or yellow. Classes are sorted by wall time (htb_class::pq_key) at which they will change color (htb_class::mode). It is because the color change is asynchronous.

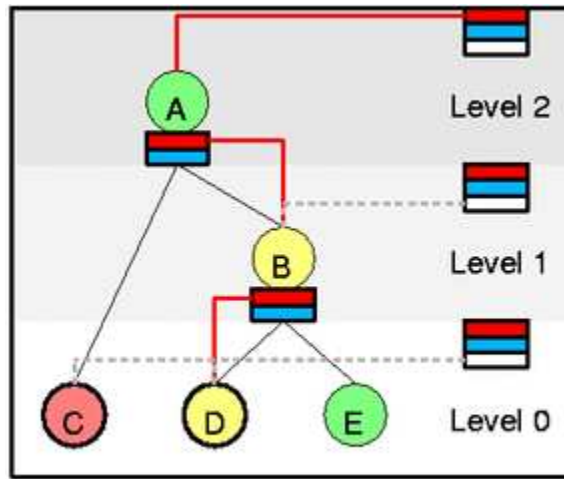
Probably you already see that if we are able to keep all classes on correct lists then selecting of next

packet dequeue is very simple. Just look at the self feed list and select the nonempty slot (one with a line) with lowest level and highest prio. There is no such one at pict 1. so that no packet can be send. On 2. it is red slot at level 0 thus class D can send now.

Let's look closely at the first two pictures. At 1. there is no backlogged leaf (all such leaves are drawn by thin circle) so that there is nothing to do. At 2. packets arrived for both C and D. Thus we need to activate these classes (`htb_activate`) and because they are both green we add them directly to their appropriate self slots. Dequeue would select D now and continue to C if it can't drain packet from C (unlikely).



3. D is over its rate



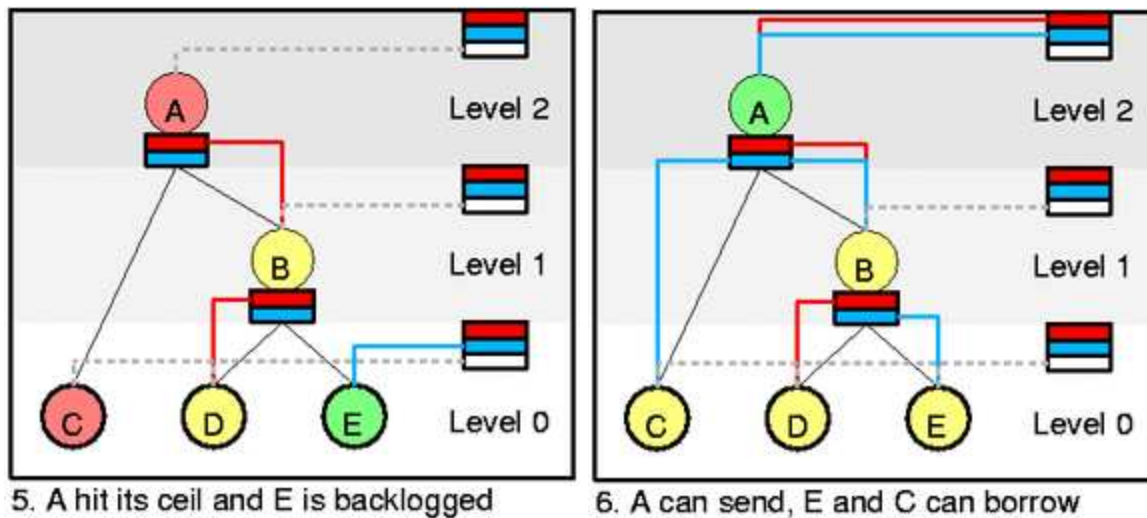
4. C too but it also hit ceil, B reached its rate

Let's assume that packet was dequeued (`htb_dequeue`) from D (`htb_dequeue_tree`) and we charged the packet size to leaky bucket (`htb_charge_class`). It forced D to go over its rate and changes color to yellow. As part of this change (`htb_change_class_mode`) we need to remove it from self feed (`htb_deactivate_prios` and `htb_remove_class_from_row`) and add to B's inner feed (`htb_activate_prios`). It also recursively adds B to self feed (`htb_add_class_to_row`). Because D will return to green state after some time we add it to priority event queue (`htb_add_to_wait_tree`) at its level (0).

Dequeue would now select class C even if it is lower prio than D. It is because C is at lower level and to compute [eq2] correctly we have to serve lower levels first. It is also intuitively ok - why to borrow if someone can send without it.

Let's assume that we dequeued C now and it changed state to red (ceil was reached). In this state C can't even borrow. Also assume that some packets was then drained from D and thus B changed to yellow. Now you should be able to explain all steps: remove B from self list, add it to wait and add A to self. Add B to A's inner feed. Add C to wait.

Now you can clearly see how self and inner feed list creates path for borrowing. It is the line at pict 4. going from the top self slot down to D. And yes, D is now the only one who can send.



Let's complicate it more. A hits its ceil and E starts to be backlogged. C returned to green. Changes are trivial only you can see that inner feeds are maintained even if the are not used. Look at the red feed which ends at A ... This picture also shows that more than one class can be on the same feed. E and C can be dequeued at the same time. To ensure correct distribution by Q as requested in [eq2] then as long as this state persist we have to apply DRR and cycle over both classes. It is simpler to say but harder to do. Let's look closer at it.

The class lists attached to inner or self feeds are rb-trees in reality. Hence each such list is sorted by its classid. Classid is constant for hiven hierarchy. We remember for each self slot active class from the list (`htb_sched::ptr`). Then it is fast to find leaf to dequeue - just follow (`htb_lookup_leaf`) ptrs from self feed thru inner feeds (`htb_class::inner.ptr`) to a leaf. When DRR decides that its quantum was exhausted (`htb_dequeue_tree`) we increment ptr (`htb_next_rb_node`) which led us to the leaf. The next tree traversal (`htb_lookup_leaf`) will propagate the change to the upper levels.

You can ask why the list should be sorted by classid. It is simple - DRR theory assumes that backlogged class remain on the same position. Only then DRR properties hold. In HTB the class migrates between lists too often and this would adversely affect precision of ratios.

The last and very interesting picture. Three classes changed, A can send E and C are yellow now. The important thing here is to understand that inner class can be active for more priorities (`htb_class::prio_activity`) while leaf only for one (`htb_class::aprio`). So that you can see red/blue pair going from self feed to A and B and forking to D and E. Also you can see the same as at 5. but with inner feed now. A serves both C and B (and thus E) at blue/low prio. If there were not D active we would have to dequeue C and E using DRR.

Implementation details

Paragraph above should be enough to understand the code. Only a few notes. There are `htb_sched::row_mask[level]` which tells us which prios are active on single level. For picture 6 the `row_mask = { 0,0,3 }`, for pict 3 it would be `{ 2,1,0 }`. It allows for faster eligible level lookup.

The `htb_dequeue` code applies all pending changes in even queue for the level which is going to dequeue. It is also reason why event queue is split per level - we need not to apply changes to level 1 or 2 classes as in picture 5 when it is enough to apply only those from level 0 and then dequeue level 0. It is

because of fact that on higher level prio queue can be no event which would prevent lower level from having green classes.

There is short-circuit for event existence test. Because we often need to test whether there is event for this time I added `htb_sched::near_ev_cache` per level which holds time of nearest event in jiffies. Fast test is then enough. Typically 10-30% speedup.

Color is called mode in the code. Corelation is 0=Red=HTB_CANT_SEND, 1=Yellow=HTB_MAY_BORROW, 2=Green=HTB_CAN_SEND. It is computed in `htb_class_mode` as leaky bucket [2]. Note the there is globally defined HTB_HYSTERESIS which adds hysteresis of "burst" size to the mode computation. It means that there is less mode changes and I measured 15% speedup.

Waitlist presence rule. The class is on its waitlist everytime when class mode is not HTB_CAN_SEND (Green).

Linux qdisc should support explicit drop request. It is very hard to do it here so that I have to add special `htb_class::leaf.drop_list` along with `htb_sched::drops` list per priority. These holds all active classes for given level. During drop I select the first one from lowest prio active list.

HTB uses "real" DRR as defined in [4]. CBQ in Linux uses one where the quantum can be lower than MTU - it is more generic but it is also no longer $O(1)$ complexity. It also means that you have to use right scale for rate->quantum conversion so that all quanta are larger than MTU.

Acknowledgments

I'd like to say thanks to numerous people from Linux world who helped and motivated me a lot.

[1] Link-sharing and Resource Management models for Packet Networks, Sally Floyd and Van Jacobson, 1995

[2] Leaky Bucket, J. Turner, IEEE vol 24 1986

[3] HTB for Linux, <http://luxik.cdi.cz/~devik/qos/htb>

[4] Efficient Fair Queuing using Deficit Round Robin, M. Shreedhar and G. Varghese

[5] WF2Q: Worst-case Fair Weighted Fair Queuing, J.C.R. Bennet and Hui Zhang

[Practical QOS](#)

[Next](#)

Differentiated Service on Linux HOWTO


Search

Written by Leonardo Balliache

August 2.003

<http://opalsoft.net/qos>

Things are easier than they appear; all you need is the right focus

Leonardo Balliache

[1.0. Differentiated Service Theory](#)

[1.1 Introduction](#)

[1.2 The specifications](#)

[1.3 The DS field](#)

[1.4 The architecture](#)

[1.5 Assure Forwarding PHB Group](#)

[1.6 Expedited Forwarding PHB](#)

[2.0. Linux Queuing Disciplines](#)

[2.1.- Linux Traffic Control](#)

[2.2.- FIFO queuing discipline](#)

[2.3.- PRIO queuing discipline](#)

[2.4.- TBF queuing discipline](#)

[2.5.- SFQ queuing discipline](#)

[2.6.- RED queuing discipline](#)

[2.7.- GRED queuing discipline](#)

[2.8.- HTB queuing discipline](#)

[2.9.- DSMARK queuing discipline](#)

[2.10.- TCINDEX classifier](#)

[2.11.- Policing](#)

3.0. Differentiated Service on Linux distribution

3.1.- Edge1

3.2.- Edge2

3.3.- Edge31-ca-u32

3.4.- Edge32-ca-u32

3.5.- Edge31-cb-tables

3.6.- Edge32-cb-tables

3.7.- Edge32-cb-u32

3.8.- Afhtb

3.9. ef-prio

3.10.- ef-htb

?? References

Practical QOS

Next

[Previous](#)[Content](#)[Next](#)

1.0.- Differentiated Service Theory



One of the problems to be addressed on the Internet environment is how we can provide better and differentiated services to our users and customers. Based on the idea that different quality of services must be offered to fulfill different customer needs and requirements, the differentiated service (diffserv) architecture was developed. Using differentiated service technology we can improve our services and offer better quality and richer option menus to our users and customers, to have a competitive advantage over our competition.

[Previous](#)[Content](#)[Next](#)

1.1.- Introduction

The diffserv architecture is based on a network model implemented over a complete Autonomous System (AS) or domain. Being this domain under our administrative control, we can take provisions to establish clear and consistent rules to manage traffic entering and flowing through the networks that conform the domain. If we are an ISP, our domain is used as a transit space for traffic to our customers and users, or inclusive to other ISP domains.

To do what we want, an architecture is implemented, where traffic entering the network at the edges of the domain is classified and assigned to different behavior aggregates. Each aggregate is identified by previously marking header of packets belonging to each behavior, when they enter the domain.

Inside the domain, packets belonging to the same behavior aggregate are forwarded according to previous established rules; this way, what we are really doing is creating classes of flows that travel through our networks. Each flow is treated along the domain according to the class to which it belongs. Using this class discrimination, we can have flows class A, B, C, D, etc., where each class receives a different treatment that, previously, we have established what is going to be.

Our domain becomes in some kind of discriminatory world, where depending of the class to which each flow belongs it will be treated different, perhaps as a king or queen, perhaps very well, perhaps not so well, or perhaps (for flows we don't want) really bad or very bad.

Let us see something graphic to represent what we are talking about; I'm going to ask you for some effort to imagine what I'm trying to draw, because I'm a very bad artist:

[Paper: Mobile Backhaul](#)

Expanding 3G backhaul capacity without jeopardizing 2G revenues
www.ciena.com

[QoS Management Tools](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[SIP call troubleshooter](#)

Real-time VoIP Traffic monitoring Free Trial & Support. Download now!
www.vqmanager.com

[AscenLink - Load Balancer](#)

ISP link Aggregation, VPN Trunking. MPLS / IPVPN Backup & Load Balance
www.xtera-ip.com

[Zigbee / IEEE802.15.4](#)

Semiconductor devices for low power wireless sensor networks
www.jennic.com

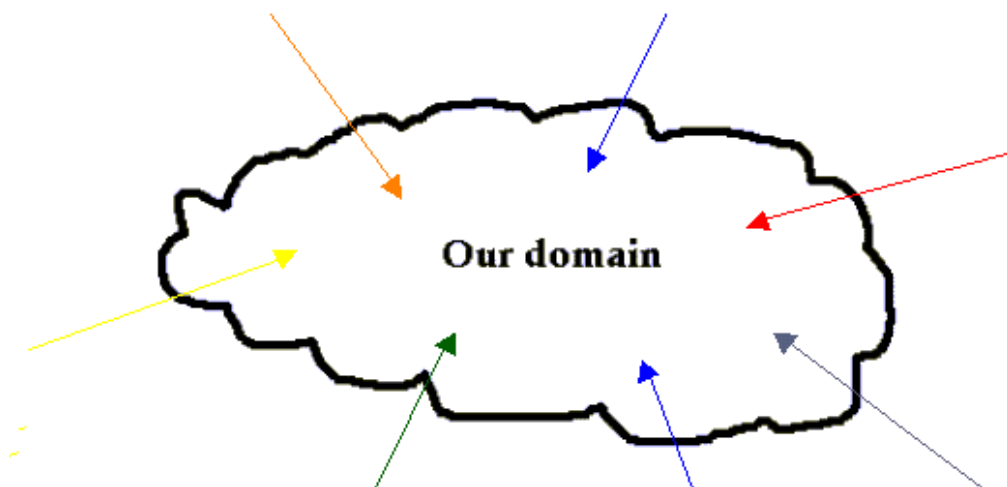


Figure 1.1.1

The cloud is representing our domain; arrows entering to it are different flows that we are receiving from outside. Flows are of different colors indicating that not all of them are of the same importance or interest for us. Some of them are from customers that pay for class A service, other from customers engage in standard services at lower costs; some flows are from mission critical services that require a special no loss and fast response dispatching; some are from services less critical that can accept some delay and perhaps losses without generating problems to the application they are trying to serve; some are from general but acceptable traffic that we can treat using the best-effort policy; and some are from unidentified places, but we don't want them because they are malicious Trojans, viruses and Spam e-mails that consume our network bandwidth and cause a lot of problems to our users, customers and technical people.

What we are going to do now is zooming one of those places at some edge of our domain where flows are entering to study better the situation on it; again a diagram:

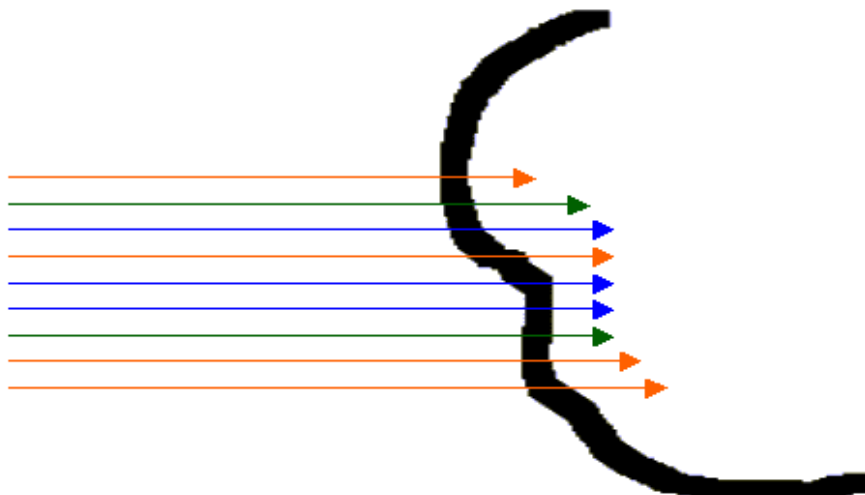


Figure 1.1.2

In this example, we have nine flows entering our domain at some edge of it; let us suppose that after a conscientious study of the situation we have decided that these flows can be classified using 3 different classes: the blue class is going to content 3 of the flows, the red class is going to content 4 of the flows and the green class is going to content 2 of the flows. To have some coherence with previous explanations let us suppose that green class is an A or Gold class, blue class is a B or Silver class, and red class is a C or Bronze class. For now it does not matter what gold, silver or bronze class means, just that they are different and have different requirements to be met.

When we classify these 9 flows into 3 classes, and after thinking that they could be 20, 30 or several hundred of them, classified again into 3 classes (or 4, 5 or 10 of them), we are understanding and using one of the basic and more important characteristic of differentiated service: it operates on behavior aggregates. What does it means? That we can have many flows but we classify previously by their behavior, aggregating them in several classes that always are going to be less than the original flows.

What do we get with this? We reduce the flow state information to be required to maintain on each router; instead of having state information for every flow, we reduce dramatically the amount of resources required by managing every class of flow, instead of every flow. As RFC 2386 "A Framework for QoS-based Routing in the Internet" points out: "An important issue in interdomain routing is the amount of flow state to be processed by transit ASs. Reducing the flow state by aggregation techniques must therefore be seriously considered. Flow aggregation means that transit traffic through an AS is classified into a few aggregated stream rather than been routed at the individual flow level".

Okay, but we have to prepare our domain to do that. It has to classify flows entering on it in some manageable number of classes, or behavior aggregates, and afterward, it has to have clear rules for each of these classes to be treated or managed (routed, shaped, policed, dropped, delayed, marked, remarked, forwarded, etc.), when they cross through the domain.

What we are talking about for flows entering our domain must be valid too for flows leaving from it. Let us suppose that, as we are an ISP, we can consider ourselves as a black box that do not generate flows directly, but instead, we transport them for our users, customers and other domains. As soon as we are implementing alone this new differentiated service technology, we have to take providence to not damage or confuse other people with packets marked by us. This means that because we are going to mark packets entering our domain to apply our idea of having differentiated service on it, we have also to respect our similar leaving packets going out our domain without any mark; we have to clean it out what we put on packets to do our fantastic experiments; and we are going

[AscenLink - Load Balancer](#)

ISP link Aggregation, VPN Trunking, MPLS / IPVPN Backup & Load Balance
www.xtera-ip.com

[End to End Monitoring](#)

Monitor performance of Java Applications in SOA environments
www.nastel.com

[End to End QoS and more](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[IP Address Management](#)

DNS/DHCP and IPAM solutions Free on-site Trial
www.BlueCatNetworks.com

[CESoPSN & SAToP Solutions](#)

E1/T1/J1 Over IP,MPLS,EPON Networks For Next Generation Networks
www.techaya.com

to do that, beyond a shadow of a doubt.

If we are successful with our ideas and we do implement differentiated service in our domain we can try later to reach some deal with our customers to offer these special services by what is called a SLA (Service Level Agreement). SLA will define forwarding services that our customers will receive. Also, we can sign with them what is called a TCA (Traffic Conditioning Agreement) which usually specifies traffic profiles and action to be taken to treat in-profile and out-of-profile packets.

And being more prolific, what about if we have some ISP neighbor as inventive as we are and we can extend our concept of differentiated service beyond our domain frontiers? Then, we could sign those SLA and TCA contracts with our peers and forward to them, and receive from them, marked packets that will be treated in both domains following specifics and previously agreed rules.

All these ideas that I have outlined have been taken from what the differentiated service architecture is promising the new Internet world is going to be. But, landing again to the real world, let us continue studying how we are going to implement differentiated service. Next step will be to explain in more detail the architecture.



[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

1.3.- The DS field

RFC 2474 define the field to be used on packets to print our mark; this mark will be used afterward to identify to which group the packet marked belongs to. Our discussion will be concentrated in **IPV4** packets. Reading from **RFC 2474** we have:

Differentiated services enhancements to the Internet protocol are intended to enable scalable service discrimination in the Internet without the need for per-flow state and signaling at every hop. A variety of services may be built from a small, well-defined set of building blocks which are deployed in network nodes. The services may be either end-to-end or intra-domain; they include both those that can satisfy quantitative performance requirements (e.g., peak bandwidth) and those based on relative performance (e.g., "class" differentiation). Services can be constructed by a combination of:

- *setting bits in an IP header field at network boundaries (autonomous system boundaries, internal administrative boundaries, or hosts),*
- *using those bits to determine how packets are forwarded by the nodes inside the network, and*
- *conditioning the marked packets at network boundaries in accordance with the requirements or rules of each service.*

Well. We touch briefly about this in our initial explanation. They indicate that "the enhancement to the Internet protocol covered by this specification are intended to enable scalable service discrimination without the need of per-flow state and signaling at every hop". We explained above that because our intention was to classify flows to aggregate them in groups before deciding how to forward them, we don't need to control states at routers on a per-flow based granularity but instead using aggregate of flows. This way our architecture will be easily scalable because using a few several aggregate groups we can manage the forwarding of many more individual flows.

Also the signaling, this means, classifying and marking, is done at the border routers only (edges of the domain) not requiring to signal at every hop of the domain. This explanation is really very important to the success of any new architecture, because service are scalable, this means, amount of resources required to implement and manage the model are not proportional to the number of flows to be forwarded but instead to some previously defined few "behavior aggregate".

After explaining briefly what kind of service could be implemented using the new architecture the specification gives us also some explanation of how they are intending to do that, this means: 1.- setting bits in the IP packets header field at network boundaries. 2.- using those bits to determine how packets are forwarded by the nodes inside the network, and 3.- conditioning the marked packets at network boundaries in accordance with the requirements or rules of each service.

We talked a little about points 1 and 2, this means, marking the packet when entering the domain by setting some bits in a field (not yet defined) in the IP header and using this mark to decide how to forward the packets inside the domain. But third intention is a new one. We can also conditioning the marked packets at network boundaries in accordance to some requirements or rules to be defined later. We can

[End to End QoS and more](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[Network/Ethernet Tester](#)

Tester for Switch & SOHO Router Network TAP / Power Monitor
www.xtramus.com

[SIP Monitor-Analyzer](#)

Call flow ladders, packet displays Linux & Windows support. Free trial!
www.vamanager.com

[Temperature via Internet](#)

Thermometer accessible through Ethernet. Price from 96 EUR!
www.papouch.com/en/

[Network Packet Analyzer](#)

Advanced Network Analyzer Solution. Start Analyzing Today. Download!
www.Paessler.com/Package-Sniffi

condition the packets when they are entering the domain (ingress) or when they are leaving it (egress).

But, what conditioning means? It means preparing the packets to fulfill some rules, perhaps marking them using some previous multi-field (MF) classification (by source and/or destination address, by source and/or destination ports, by protocol, etc.), perhaps shaping or policing them before entering or leaving the domain or perhaps remarking them if they are previously marked. For example, we talked that if our neighbors were not as inventive as we are, we have to clear any mark on packets we made before they leave our domain. This is an example of conditioning packets following a previous requirement or rule (we can't bother our similar with our inventions).

Reading again from the specification we have:

A differentiated services-compliant network node includes a classifier that selects packets based on the value of the DS field, along with buffer management and packet scheduling mechanisms capable of delivering the specific packet forwarding treatment indicated by the DS field value. Setting of the DS field and conditioning of the temporal behavior of marked packets need only be performed at network boundaries and may vary in complexity.

Well, they are talking about a new term called DS field that we have not defined yet. If we read the RFC specification, for all of us that are not versed in network terminology we find frequently some holes. Before continuing let us to attempt a brief explanation or definition about some terms commonly used when talking about differentiated service. Let's start identifying the DS field. The DS (for Differentiated Service) field is where we are going to mark our packets. This field is on the IP packet header. A figure can help us to understand this:

IP HEADER

0		15 16		31		
4 bit version	4 bit header length	8 bit type of service (TOS)	16 bit total length (in bytes)			4
16 bit identification			3 bit flags	13 bit fragmentation offset		8
8 bit time to live (TTL)	8 bit protocol	16 bit header checksum				12
32 bit source ip address						16
32 bit destination ip address						20
options (if any: next tcp udp header)						
DATA						

Figure 1.3.1

Here we have a diagram of the IP header. People who created the differentiated service architecture decided to use the second octet of the header identified in the figure as "8 bit type of service (TOS)" to implement the model, renaming the field as "DS field". In fact this octet had been traditionally used as a medium to signaling type of service to be lent to the packet. They merely redefine the utilization of the field to be incorporated to the new architecture.

They also talk about something called a classifier, basically to say that based on content of the DS field the classifier selects packets and applies to each different group of packet aggregation, identify by a distinct DS field, a differentiated treatment in terms of buffer management and packet scheduling. The term classifier, along with some other like dropper, marker, meter, shaper, policer and scheduler are defined in RFC 2475 but because they are needed to understand better what we are studying, let us to jump to it and then come back again. As it was written before problem when you try to follow specifications is that you find some holes than are covered later. Let us try to order things a little to make easier to understand concepts.

Reading on RFC 2475 we have:

Microflow: a single instance of an application-to-application flow of packets which is identified by source address, source port, destination address, destination port and protocol id.

Here we have a traditional definition of a flow between two applications. Any flow is identified by the 5-tuple (src addr, src port, dest addr, dest port, protocol). These 5 pieces of information are located in the IP/TCP/UDP header. Continuing with RFC 2475 we have now:

Classifier: an entity which selects packets based on the content of packet headers according to defined rules.

MF Classifier: a multi-field (MF) classifier which selects packets based on the content of some arbitrary number of header fields; typically some combination of source address, destination address, DS field, protocol ID, source port and destination port.

Basically the classifier is a mechanism that looks at the IP header to get some information that permit to classify the packet in some group. Classifier could use the DS field where we are going to put our mark to select packets. Or it can use a more complex selection using other fields like the source and/or destination address, source and/or destination port, and/or protocol identification.

Let us suppose that we want to separate in our domain flows that are TCP from those that are UDP. We know that we have to be very pending about UDP flows. Those flows are unresponsive (see foot note), this means, when congestion appears they do not adjust automatically its throughput to alleviate the link as TCP does. Because of this they can starve other flows and worse congestion behavior. To approach this problem we decided to implement a classifier on edge routers in our domain that select packets entering to it to classify them into two groups: TCP packets and UDP packets. In this case the classifier looks into the packet header protocol identification to select and classify them before entering our domain.

More complex selection and classification can be done. We can select, for example, packets coming from a specific external network or going to a specific internal network or perhaps those that are servicing a special service like ssh, ftp or telnet. The classifier is the mechanism that is in charge to select and classify the packets. When using different fields from the IP/TCP/UDP header to make the classification they are called multi-field (MF) classifiers.

But also classifiers can use only the DS field to classify packets. Let us suppose that we mark the UDP flows entering our domain with a specific mark on the DS field (later we are going to see how to mark packets using the DS field). After being marked we let packets enter our domain. At the same time we prepare routers inside the domain (they are called core routers to distinguish them from edge routers located at the domain frontiers) to forward these packets in some way. Then core routers will need to classify packets using the DS field instead of other fields in the IP packet header.

What we are talking is state-of-the-art of networking. Do not forget that as RFC 2386 - "A Framework for QoS-based Routing in the Internet" states: *limiting flow specific information is very important in any routing model to achieve scalation*. This is true for any network model. When limiting per-flow multi-field classification just at edge routers we are walking on this direction; think that except for high speed trunks between domains (where some other model has to be implemented) rest of links are from customers where maximum bandwidth is limited, permitting to control per-flow state classification more easily.

From RFC 2475 again:

Marking: the process of setting the DS codepoint in a packet based on defined rules; pre-marking, re-marking.

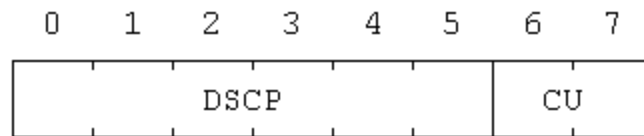
Marker: a device that performs marking.

Pre-mark: to set the DS codepoint of a packet prior to entry into a downstream DS domain.

Re-mark: to change the DS codepoint of a packet, usually performed by a marker in accordance with a TCA.

Marking and Marker are self-explanatory. But the specification uses now the expression "DS codepoint" instead of "DS field". What happens is that differentiated service uses only the 6 leftmost bits, from the eight of the DS field, to mark the packets. Bits 0 to 5 are used and bits 6 and 7 are respected. The 6 leftmost bits of the DS field form the DS codepoint. It's very important to not confuse DS field and DS codepoint, also called DSCP. Next figure taken from RFC 2474 clears what we are talking about:

The DS field structure is presented below:



DSCP: differentiated services codepoint

CU: currently unused

Figure 1.3.2

In a DSCP value notation 'xxxxxx' (where 'x' may equal '0' or '1') used in this document, the left-most bit signifies bit 0 of the DS field (as shown above), and the right-most bit signifies bit 5.

As you can see bits 6 and 7 are unused by differentiated service but used by other technologies to indicate ECN (Explicit Congestion Notification); anyway this theme is out of the scope of this document.

Above, when talking about marking, they wrote pre-marking or re-marking. When different domains implementing differentiated service interact to exchange DS packets, they can reach the edge of any domain previously marked in another domain. The packets are pre-marked. If not previously marked by another domain, they could be pre-marked by an edge router of our domain before entering it; these packets are pre-marked too.

Also it could be that when those packets reach our domain previously marked in another domain we re-mark them before entering our. The packets are then re-marked. Finally, packets could be re-marked before leaving our domain. For example, if we are implementing differentiated services alone in our domain we have to leave packets not marked before departing from it. Or perhaps, having some agreement with another domain we have to mark or re-mark packets leaving from our domain and going to the former.

Reading again from RFC 2475:

Metering: the process of measuring the temporal properties (e.g., rate) of a traffic stream selected by a classifier. The instantaneous state of this process may be used to affect the operation of a marker, shaper, or dropper, and/or may be used for accounting and measurement purposes.

Meter: a device that performs metering.

Here metering is defined. Normally this process is implemented, but not limited, at edge routers of domains. The idea is as follows. We can have some agreement with another domain to accept flows coming from it subject to some predefined rules or we simply define our own rules to define the characteristics of flows we are accorded to accept. Rules are related mainly to flow maximum throughput. Well, to be sure that these rules are fulfilled and maximum rates are not exceeded we have to measure those flows before entering our domain. This process called metering is done by devices called meters. Later we are going to see how these devices are implemented in the router.

Also, depending of the instantaneous state of this process of measuring we have to take some decision of what to do with flows violating our rules. For example, let us suppose that because we want to protect our networks from misbehave flows, one of our rules is that udp flows coming from network 211.32.120/24 must not exceed 1.5 Mbps of throughput when entering our domain. As soon as this rate is not exceeded there is no problem; we simply admit the flows. But when our meters tell us that the throughput is exceeded we have to take some advance to make sure our rules are respected. Metering and meters tell us about flows behavior. To take actions we can implement marking, dropping, policing or shaping. Let us continue reading from RFC 2475 for these definitions.



www.vyatta.com
Ads by Goooooogle

Dropping: the process of discarding packets based on specified rules; policing.

Dropper: a device that performs dropping.

Policing: the process of discarding packets (by a dropper) within a traffic stream in accordance with the state of a corresponding meter enforcing a traffic profile.

First approach to deal with flows not respecting our rules is to drop them. Dropping is the process of discarding those packets. For example, we can accept all packets up to the maximum rate allowed and discard (drop) all of them exceeding it. Dropper are the devices that perform dropping. Policing comprehends the all process of metering and dropping, when trying to enforce our traffic profile.

Another approach could be not to drop packets but instead to shape them as much as is possible to cause them to conform our rules. But let us take the definition directly from RFC 2475:

Shaping: the process of delaying packets within a traffic stream to cause it to conform to some defined traffic profile.

Shaper: a device that performs shaping.

Definitions are self explanatory. As soon as we have enough buffering capacity we can delay packets to conform the profile previously defined.

Combinations of these approaches (metering, marking, dropping and shaping) can be used freely by network administrators to enforce traffic profile entering and leaving administered domains. For example, a hierarchical approach could be to accept and mark with some DSCP, packets up to a predefined rate; then mark with another DSCP, packets from previous rate and up to another higher predefined rate, and finally to drop all packets over this last rate. Inside the domain packets marked with the first DSCP could have a special and fast forwarding treatment with no drop; and packets marked with the second DSCP a restringed treatment where they are dropped following some aleatory selection of them.

Let us return back again to the RFC 2474:

This document concentrates on the forwarding path component. In the packet forwarding path, differentiated services are realized by mapping the codepoint contained in a field in the IP packet header to a particular forwarding treatment, or per-hop behavior (PHB), at each network node along its path. The codepoints may be chosen from a set of mandatory values defined later in this document, from a set of recommended values to be defined in future documents, or may have purely local meaning. PHBs are expected to be implemented by employing a range of queue service and/or queue management disciplines on a network node's output interface queue: for example weighted round-robin (WRR) queue servicing or drop-preference queue management.

In this paragraph the authors define the term PHB or per-hop behavior. Basically a per-hop behavior (PHB) is a particular forwarding treatment that a group of packets marked with a

specific codepoint (DSCP) will going to receive at each network node along its path. It's very important to note that a mapping must be established between the DSCPs and the different PHBs to be defined. Also codepoint (DSCP) are going to be chosen from a set of mandatory to be defined in the own document; PHBs will be implemented using different resources to be offered by routers at network nodes. Those resources are basically queuing management disciplines and we will see later on how they are implemented. If we continue reading we have:

Behavior Aggregate: a collection of packets with the same codepoint crossing a link in a particular direction. The terms "aggregate" and "behavior aggregate" are used interchangeably in this document.

This definition reinforce definitions given above. "Behavior Aggregate" or simply "Aggregate" (or BA) is a collection of packets having the same DSCP. It's very important to say that any "Aggregate" will be, by mapping, assigned to a PHB, but be adviced that more than one BA can be assigned to the same PHB. DHCP-PHB mapping can be a N:1 relationship.

Traffic Conditioning: control functions that can be applied to a behavior aggregate, application flow, or other operationally useful subset of traffic, e.g., routing updates. These MAY include metering, policing, shaping, and packet marking. Traffic conditioning is used to enforce agreements between domains and to condition traffic to receive a differentiated service within a domain by marking packets with the appropriate codepoint in the DS field and by monitoring and altering the temporal characteristics of the aggregate where necessary.

Traffic Conditioner: an entity that performs traffic conditioning functions and which MAY contain meters, policers, shapers, and markers. Traffic conditioners are typically deployed in DS boundary nodes (i.e., not in interior nodes of a DS domain).

These definitions taken from RFC 2474 round our ideas about differentiated services. Conditioning is a compound process based on metering, policing, shapping and packet marking to be applied to a behavior aggregate. Using traffic conditioning we enforce any previous agreement made between differentiated service domains or our own rules used to differentiate quality of service to be given to different aggregates. Again from RFC 2474:

To summarize, classifiers and traffic conditioners are used to select which packets are to be added to behavior aggregates. Aggregates receive differentiated treatment in a DS domain and traffic conditioners MAY alter the temporal characteristics of the aggregate to conform to some requirements. A packet's DS field is used to designate the packet's behavior aggregate and is subsequently used to determine which forwarding treatment the packet receives. A behavior aggregate classifier which can select a PHB, for example a differential output queue servicing discipline, based on the codepoint in the DS field SHOULD be included in all network nodes in a DS domain. The classifiers and traffic conditioners at DS boundaries are configured in accordance with some service specification, a matter of administrative policy outside the scope of this document.

A new restriction is given in this paragraph; if you define a behavior aggregate identify by a specific DSCP and you map it to a determined PHB, this PHB should be included in all network nodes of the DS domain. Something that common sense indicates. Any packet belonging to a

behavior aggregate mapped to a PHB has to encounter at any node its PHB implemented to obtain adequate forwarding.

Let us now to talk a little more about the DS codepoint to complete this theme. Reading from RFC 2474 we have:

Implementors should note that the DSCP field is six bits wide. DS-compliant nodes MUST select PHBs by matching against the entire 6-bit DSCP field, e.g., by treating the value of the field as a table index which is used to select a particular packet handling mechanism which has been implemented in that device. The value of the CU field MUST be ignored by PHB selection. The DSCP field is defined as an unstructured field to facilitate the definition of future per-hop behaviors.

Have a look to the DS field figure somewhere above. First of all, mapping between DSCPs and PHBs must be done against the entire 6-bit DSCP field; this means that partial or individual bits matching is not allowed. DSCP must be consider an atomic value which we can use to enter in a table as an index to get the corresponding per-hop behavior. Also last 2-bits (CU field) must be ignored for PHB selections.

A "default" PHB MUST be available in a DS-compliant node. This is the common, best-effort forwarding behavior available in existing routers as standardized in [RFC1812]. When no other agreements are in place, it is assumed that packets belong to this aggregate. Such packets MAY be sent into a network without adhering to any particular rules and the network will deliver as many of these packets as possible and as soon as possible, subject to other resource policy constraints. A reasonable implementation of this PHB would be a queueing discipline that sends packets of this aggregate whenever the output link is not required to satisfy another PHB. A reasonable policy for constructing services would ensure that the aggregate was not "starved". This could be enforced by a mechanism in each node that reserves some minimal resources (e.g, buffers, bandwidth) for Default behavior aggregates. This permits senders that are not differentiated services-aware to continue to use the network in the same manner as today. The impact of the introduction of differentiated services into a domain on the service expectations of its customers and peers is a complex matter involving policy decisions by the domain and is outside the scope of this document.

The RECOMMENDED codepoint for the Default PHB is the bit pattern '000000'; the value '000000' MUST map to a PHB that meets these specifications. The codepoint chosen for Default behavior is compatible with existing practice [RFC791]. Where a codepoint is not mapped to a standardized or local use PHB, it SHOULD be mapped to the Default PHB

A default PHB is defined in these paragraphs and it is associated to the actual "best-effort" behavior. Some common sense tells us that the ultimate service we can provide is at least the actual "best-effort" service; assuming that any other PHB was not established we have to implement our architecture in some way to treat common flows (those not specially marked) in a previously determined PHB. This PHB does not have any special treatment except that some providence has to be taken to assure that in presence of other priority flows, they can not be starved, to allow the normal flow of them. Normally these providences are taken by controlling maximum bandwidth allowed to priority flows so that resources are left available for "best-effort" flows.

Also it's natural to select the codepoint '000000' to be mapped to this "best-effort" PHB. This way we respect other RFCs and common practices. Remember that somewhere above we talked about the necessity to leave our domain departing packets being not marked when not having a special agreement with other domains. Re-marking those packets with a codepoint '000000' guarantee we are respecting our similars. Observe also that packets not having any predefined codepoint in our implementation have to be associated to this "best-effort" PHB. This way we guarantee to ourself that all flows will be forwarded at least on a "best-effort" policy. If we forgot to assign some flows to a special codepoint they will be treated by our implementation as "best-effort" flows.

Let us see now how RFC 2474 authors approach the class definition in DS architecture. Continue reading we have:

A specification of the packet forwarding treatments selected by the DS field values of 'xxx000|xx', or DSCP = 'xxx000' and CU subfield unspecified, are reserved as a set of Class Selector Codepoints. PHBs which are mapped to by these codepoints MUST satisfy the Class Selector PHB requirements in addition to preserving the Default PHB requirement on codepoint '000000' (Sec. 4.1).

To begin defining how DSCP will be used authors define a class selector codepoint. Using the first 3 bits of the DSCP they establish that they are going to be used to identify a class. Every class has its codepoint defined in some way that satisfy the pattern 'xxx000|xx', when talking about the DS field (all 8 bits being considered) or 'xxx000', when talking about DSCP (6 leftmost bits being considered). What all this means? Basically that we can define classes of flow and use a pattern like 'xxx000' to identify them.

[Deep Packet Inspection](#)

A User-driven Community for DPI/DPP Learn, discuss, and collaborate
www.dPacket.org

[WAN News](#)

Network IT News, Info, Trends, White Papers & Case Studies- Free!
NetworkWorld.com

[MCF5213 Embedded Software](#)

MQX RTOS, TCP/IP, CAN and HTTP Free Software Evaluation Available
www.Embedded-Access.com

[Ip](#)

Soluções Completas de Telecom Confira Nossos Produtos e Serviços!
www.CTTecom.com.br

[Next Generation Mobile](#)

Personalised Messaging & Media For all mobile networks, everywhere
www.iinnv.ie

For example, we could invent a new class named "My best friend class" and select a codepoint for it respecting the specification. Something like 101000, or 111000, or 001000, or 110000, etc. Last 3 DSCP bits will be always zero. With this restriction we can have a maximum of 8 classes using the different combination permitted for the three leftmost bits.

Unresponsive flow: When an end-system responds to indications of congestion by reducing the load it generates to try to match the available capacity of the network it is referred to as a responsive. M.A. Parris [12].

	This site is visited regularly for very specialized people; it's then the best place to advertise your very specialized product. First month at no cost for you to verify this aseveration.
---	--

[Previous](#)[Content](#)[Next](#)

[Previous](#)[Content](#)[Next](#)

1.4.- The architecture

RFC 2475 deals with architecture of differentiated services. Let us start this part of the study presenting some paragraphs of this specification to continue with our discussion; you will see that reading from it our knowledge about differentiated services will be rounded:

The differentiated services architecture is based on a simple model where traffic entering a network is classified and possibly conditioned at the boundaries of the network, and assigned to different behavior aggregates. Each behavior aggregate is identified by a single DS codepoint. Within the core of the network, packets are forwarded according to the per-hop behavior associated with the DS codepoint.

All this explanation has been treated for us along this document; nothing new we have here. Traffic entering will be classified and possibly conditioned at the boundaries of our domain and later on assigned to a behavior aggregate according to the mapping relationship between codepoints and PHBs. Within the core of the network packets will be forwarded according to the per-hop behavior associated to each codepoint.

A DS domain is a contiguous set of DS nodes which operate with a common service provisioning policy and set of PHB groups implemented on each node. A DS domain has a well-defined boundary consisting of DS boundary nodes which classify and possibly condition ingress traffic to ensure that packets which transit the domain are appropriately marked to select a PHB from one of the PHB groups supported within the domain. Nodes within the DS domain select the forwarding behavior for packets based on their DS codepoint, mapping that value to one of the supported PHBs using either the recommended codepoint->PHB mapping or a locally customized mapping [DSFIELD]. Inclusion of non-DS-compliant nodes within a DS domain may result in unpredictable performance and may impede the ability to satisfy service level agreements (SLAs).

Ratification about our knowledge. Observe that they insist in having all nodes in DS-complaint mode; any non-DS-complaint node may result in unpredictable performance. Don't forget that we can protect ourself using the default DSCP ('000000' or not defined) to assign those flows to the "best-effort" behavior aggregate.

A DS domain consists of DS boundary nodes and DS interior nodes. DS boundary nodes interconnect the DS domain to other DS or non-DS-capable domains, whilst DS interior nodes only connect to other DS interior or boundary nodes within the same DS domain.

Observe here that we can connect our domain to other DS or non DS-capable domains; latter case we have to respect our similar leaving packets without any mark. Also, as common sense tells us, interior nodes (core routers) only connect to other interior nodes or to boundary nodes (edge routers) but in the same DS domain. This way our domain is a black-box to external DS-capable or non DS-capable domains.

Interior nodes may be able to perform limited traffic conditioning functions such as DS codepoint re-marking. Interior nodes which implement more complex classification and traffic conditioning functions are analogous to DS boundary nodes.

[QoS Management Tools](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[Paper: Mobile Backhaul](#)

Expanding 3G backhaul capacity without jeopardizing 2G revenues
www.ciena.com

[Free Packet Sniffer](#)

Packet Sniffer - It's Free. Very advanced networking tool.
www.Paessler.com

[AirMagnet Free-Trial](#)

Test AirMagnet's Laptop Tool and Police/Audit/Fix your WLAN!
www.airmagnet.com

[DDoS Protection Leader](#)

Renowned anti-DDoS service. Full SLA provided. Global network.
www.prolexic.com

To protect our scalable capacity it's very important to respect this rule: as soon as is possible interior nodes should perform only limited traffic conditioning; complex conditionings must be left to boundary nodes where, perhaps, lower throughputs make easier to implement them. See RFC 2386 recommendation somewhere above.

DS boundary nodes act both as a DS ingress node and as a DS egress node for different directions of traffic. Traffic enters a DS domain at a DS ingress node and leaves a DS domain at a DS egress node. A DS ingress node is responsible for ensuring that the traffic entering the DS domain conforms to any TCA between it and the other domain to which the ingress node is connected. A DS egress node may perform traffic conditioning functions on traffic forwarded to a directly connected peering domain, depending on the details of the TCA between the two domains.

DS boundary nodes act as an "ingress" node or an "egress" node depending of the direction of the traffic. In both cases conditioning must be performed to ensure that TCA between domains are respected. When those TCAs don't exist, providence must be taken to ensure that egress traffic does not create problems to other non DS-complaint domains or those DS-complaint not having a special SLA with us.

A differentiated services region (DS Region) is a set of one or more contiguous DS domains. DS regions are capable of supporting differentiated services along paths which span the domains within theregion.

The DS domains in a DS region may support different PHB groups internally and different codepoint->PHB mappings. However, to permit services which span across the domains, the peering DS domains must each establish a peering SLA which defines (either explicitly or implicitly) a TCA which specifies how transit traffic from one DS domain to another is conditioned at the boundary between the two DS domains.

Differentiated services are extended across a DS domain boundary by establishing a SLA between an upstream network and a downstream DS domain. The SLA may specify packet classification and re-marking rules and may also specify traffic profiles and actions to traffic streams which are in- or out-of-profile (see Sec. 2.3.2). The TCA between the domains is derived (explicitly or implicitly) from this SLA.

Here we have the first definition of collaborative between differentiated service capable domains. Contiguous DS-capable domains constitute a DS Region. Observe that internally DS domains act as black boxes and their PHB groups and mapping with their codepoints are freely manage by each administrator. But when interacting with other DS-capable domains (services must spam across the domains) SLAs must be established which specifies TCAs indicating how traffic will be conditioned to cross for one domain to another and viceversa.

Also they talk about in-profile or out-of-profile traffic. When SLAs are established between domains the agreement generally include some level where traffic can be considered in-profile or out-of-profile. For example, let's suppose that we sign a SLA establishing that UDP traffic will be accepted under certain conditions up to 3.5 Mbps; above this level UDP traffic will be consider as

non-friendly and treated as it (non-friendly) depending on current network condition. Then UDP traffic up to 3.5 Mbps is considered as in-profile and UDP traffic above 3.5 Mbps is considered as out-of-profile and treated according. The final treatment will be depend on current condition of each network; in extreme cases out-of-profile traffic will be totally dropped if required.

Traffic conditioning performs metering, shaping, policing and/or re-marking to ensure that the traffic entering the DS domain conforms to the rules specified in the TCA, in accordance with the domain's service provisioning policy. The extent of traffic conditioning required is dependent on the specifics of the service offering, and may range from simple codepoint re-marking to complex policing and shaping operations. The details of traffic conditioning policies which are negotiated between networks is outside the scope of this document.

Packet classifiers select packets in a traffic stream based on the content of some portion of the packet header. We define two types of classifiers. The BA (Behavior Aggregate) Classifier classifies packets based on the DS codepoint only. The MF (Multi-Field) classifier selects packets based on the value of a combination of one or more header fields, such as source address, destination address, DS field, protocol ID, source port and destination port numbers, and other information such as incoming interface.

Nothing new here. Just ratification of what we talked somewhere above. It's very important to note that MF classifiers (where more resources are required but perhaps less throughputs have to be managed) are normally implemented at boundary nodes (edge routers) and BA classifiers (where less resources are required but perhaps more throughputs have to be managed) are normally implemented at interior nodes (core routers). This way we keep network scalability as high as is possible.

A traffic profile specifies the temporal properties of a traffic stream selected by a classifier. It provides rules for determining whether a particular packet is in-profile or out-of-profile. The concept of in- and out-of-profile can be extended to more than two levels, e.g., multiple levels of conformance with a profile may be defined and enforced.

Different conditioning actions may be applied to the in-profile packets and out-of-profile packets, or different accounting actions may be triggered. In-profile packets may be allowed to enter the DS domain without further conditioning; or, alternatively, their DS codepoint may be changed. The latter happens when the DS codepoint is set to a non-Default value for the first time [DSFIELD], or when the packets enter a DS domain that uses a different PHB group or codepoint->PHB mapping policy for this traffic stream. Out-of-profile packets may be queued until they are in-profile (shaped), discarded (policed), marked with a new codepoint (re-marked), or forwarded unchanged while triggering some accounting procedure. Out-of-profile packets may be mapped to one or more behavior aggregates that are "inferior" in some dimension of forwarding performance to the BA into which in-profile packets are mapped.

Here authors explain some interesting concepts: A traffic profile permits us to determine if a packet is in-profile or out-of-profile. The rule must be explicit and clear; for example, we talked above about UDP flows and we established a traffic profile that tells us that up to 3.5 Mbps the

traffic is in-profile and above 3.5 Mbps the traffic is out-of-profile.

But, we can have more than two levels; for example, we can establish a new traffic profile as follows: up to 3.5 Mbps traffic is considered in-profile and it will be treated as gold class traffic; from 3.5 Mbps and up to 5.0 Mbps traffic is considered out-of-profile priority-1 and it will be treated as silver class traffic; above 5.0 Mbps traffic is considered as out-of-profile priority-2 and will be treated as bronze class traffic.

For this example (gold, silver and bronze class traffic) different conditioning actions may be applied to each type of them as is explained in the second paragraph of the specification. Conditioning actions to be applied are only limited by the network administrator creativity or necessity. These actions, depending on flow class, include but are not limited to: packets may be allowed to enter without further condition; they may be allowed to enter after some accounting procedure; the DS codepoint could be set (if not previously set), i.e., marking; also it could be changed (if previously set), i.e., re-marking; out-of-profile packets may be shaped to put them in-profile; or they may be dropped; or re-marked to assign them to a low priority and/or quality behavior aggregate; etc. Possibilities are endless and a very powerful architecture is emerging to handle different environments and/or requirements.

A traffic conditioner may contain the following elements: meter, marker, shaper, and dropper. A traffic stream is selected by a classifier, which steers the packets to a logical instance of a traffic conditioner. A meter is used (where appropriate) to measure the traffic stream against a traffic profile. The state of the meter with respect to a particular packet (e.g., whether it is in- or out-of-profile) may be used to affect a marking, dropping, or shaping action.

When packets exit the traffic conditioner of a DS boundary node the DS codepoint of each packet must be set to an appropriate value.

Fig. 1.4.1 shows the block diagram of a classifier and traffic conditioner. Note that a traffic conditioner may not necessarily contain all four elements. For example, in the case where no traffic profile is in effect, packets may only pass through a classifier and a marker.

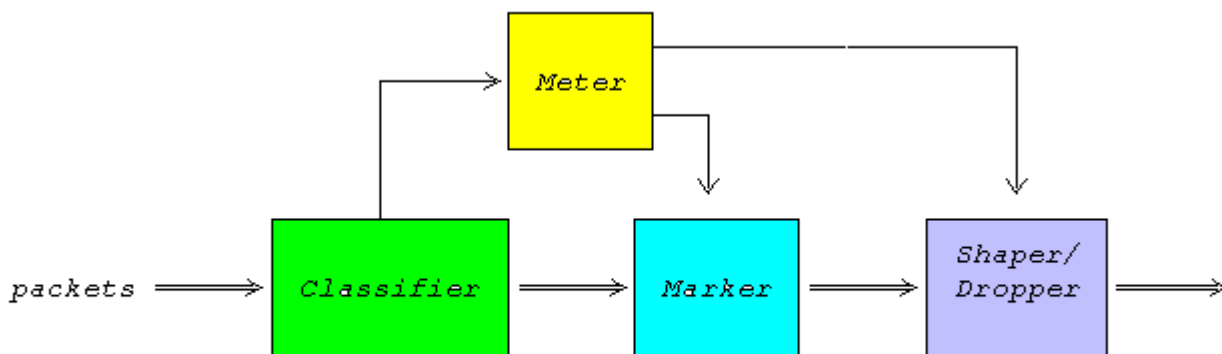


Figure 1.4.1

These paragraphs of the specification clears what we saw before when we talked about classifiers, meters, markers, shapers and droppers. The diagram shows a typical DS traffic conditioner and its elements. Conditioners are implemented at edge routers (boundary nodes) or at core routers (interior nodes). A conditioner should have at least a classifier and a marker; in this simple case incoming packets are classified, perhaps using a multi-field (MF) based classification (for example, based in the 5-tuple: source address, source port, destination address, destination port, protocol); then marked (DS codepoint is set) according to each classification and finally allowed to enter the domain. Inside the domain the DS codepoint may be used for DS based classifiers at core router conditioners to implement some other required cascading conditioning.

More complex conditioners implement also a meter that normally takes a measure of the incoming flow throughputs previously classified by classes by the classifier (using a MF classification, for example); for every class the throughput is measured and depending on their values the packets are segregated in different levels of in-profile or out-of-profile packets. Observe then that you can have in the same class different hierarchical levels of aggregation. For each level of aggregation a different action can be taken.

Some aggregations can be simply marked and allowed to enter the domain; or packets can be marked first and then passed through the shaper/dropper for shapping or policing and then allowed to enter the domain. After metering, packets can be passed directly to the shaper/dropper where they are shaped or policed by behavior aggregate and then allowed to enter the domain without having been previously marked; then they will be marked later at core routers (normally this is not done because it spoils differentiated service philosophy). As was said before possibilities are endless and the architecture is very flexible and powerful.

Next the specification defines meters, markers, shapers and droppers; we talked a little about them before but to rounding our knowledge it's a good idea to present here how the RFC 2475 specification approaches a definition of these concepts in a way that is really excellent:

Meters

Traffic meters measure the temporal properties of the stream of packets selected by a classifier against a traffic profile specified in a TCA. A meter passes state information to other conditioning functions to trigger a particular action for each packet which is either in- or out-of-profile (to some extent).

Markers

Packet markers set the DS field of a packet to a particular codepoint, adding the marked packet to a particular DS behavior aggregate. The marker may be configured to mark all packets which are steered to it to a single codepoint, or may be configured to mark a packet to one of a set of codepoints used to select a PHB in a PHB group, according to the state of a meter. When the marker changes the codepoint in a packet it is said to have "re-marked" the packet.

Shapers

Shapers delay some or all of the packets in a traffic stream in order to bring the stream into compliance with a traffic profile. A shaper usually has a finite-size buffer, and packets may be discarded if there is not sufficient buffer space to hold the delayed packets.

Droppers

Droppers discard some or all of the packets in a traffic stream in order to bring the stream into compliance with a traffic profile. This process is known as "policing" the stream. Note that a dropper can be implemented as a special case of a shaper by setting the shaper buffer size to zero (or a few) packets.

Overwhelming. Any additional word is unnecessary.

Next specification makes some advices about where traffic conditioners and MF classifiers have to be located; because it is a very important matter we are going to copy here these paragraphs from the specification and make some comments when required:

Location of Traffic Conditioners and MF Classifiers

[Free Packet Sniffer](#)

Packet Sniffer - It's Free.
Very advanced
networking tool.
www.Paessler.com

[AscenLink - Load Balancer](#)

ISP link Aggregation,
VPN Trunking. MPLS /
IPVPN Backup & Load
Balance
www.xtera-ip.com

[End to End Monitoring](#)

Monitor performance of
Java Applications in SOA
environments
www.nastel.com

[Deep Packet Inspection](#)

A User-driven Community
for DPI/DPP Learn,
discuss, and collaborate
www.dPacket.org

[DDoS Protection Leader](#)

Renowned anti-DDoS
service. Full SLA
provided. Global network.
www.prolexic.com

Traffic conditioners are usually located within DS ingress and egress boundary nodes, but may also be located in nodes within the interior of a DS domain, or within a non-DS-capable domain.

Observe that traffic conditioners can be located in boundary and/or interior nodes of the domain (we know this already) but also within a non-DS-capable domain; last asseveration implies that we can pre-conditioning flows before entering the DS-capable-domain and this work can be done on non-DS-capable-domains. Later this is explained better.

1. Within the Source Domain

We define the source domain as the domain containing the node(s) which originate the traffic receiving a particular service. Traffic sources and intermediate nodes within a source domain may perform traffic classification and conditioning functions. The traffic originating from the source domain across a boundary may be marked by the traffic sources directly or by intermediate nodes before leaving the source domain. This is referred to as initial marking or "pre-marking".

Consider the example of a company that has the policy that its CEO's packets should have higher priority. The CEO's host may mark the DS field of all outgoing packets with a DS codepoint that indicates "higher priority". Alternatively, the first-hop router directly connected to the CEO's host may classify the traffic and mark the CEO's packets with the correct DS codepoint. Such high priority traffic may also be conditioned near the source so that there is a limit on the amount of high priority traffic forwarded from a particular source.

There are some advantages to marking packets close to the traffic source. First, a traffic source can more easily take an application's preferences into account when deciding which packets should receive better forwarding treatment. Also, classification of packets is much simpler before the traffic has been aggregated with packets from other sources, since the number of classification rules which need to be applied within a single node is reduced.

Since packet marking may be distributed across multiple nodes, the source DS domain is responsible for ensuring that the aggregated traffic towards its provider DS domain conforms to the appropriate TCA. Additional allocation mechanisms such as bandwidth brokers or RSVP may be used to dynamically allocate resources for a particular DS behavior aggregate within the provider's network [2BIT, Bernet]. The boundary node of the source domain should also monitor conformance to the TCA, and may police, shape, or re-mark packets as necessary.

They define here a source domain; this domain generates the traffic and it could be a DS-capable domain or a non-DS-capable domain. It doesn't matter. If the domain is a DS-capable domain traffic can be marked in intermediate nodes or even by the application that generate it; within a non-DS-capable domain traffic could be marked by the application itself. The CEO example shows how traffic could be conditioned by the application with advantages. As closer as is possible to the source it will be better and easier to make the conditioning. The limiting quantity of traffic justify this because less resources are required and a finer granularity can be gained. Finally it is responsibility of the source domain, being DS-capable or not, to ensure that traffic leaving from it and going to a DS-capable domain conform the appropriate TCA.

2. At the Boundary of a DS Domain

Traffic streams may be classified, marked, and otherwise conditioned on either end of a boundary link (the DS egress node of the upstream domain or the DS ingress node of the downstream domain). The SLA between the domains should specify which domain has responsibility for mapping traffic streams to DS behavior aggregates and conditioning those aggregates in conformance with the appropriate TCA. However, a DS ingress node must assume that the incoming traffic may not conform to the TCA and must be prepared to enforce the TCA in accordance with local policy.

When packets are pre-marked and conditioned in the upstream domain, potentially fewer classification and traffic conditioning rules need to be supported in the downstream DS domain. In this circumstance the downstream DS domain may only need to re-mark or police the incoming behavior aggregates to enforce the TCA. However, more sophisticated services which are path- or source-dependent may require MF classification in the downstream DS domain's ingress nodes.

If a DS ingress node is connected to an upstream non-DS-capable domain, the DS ingress node must be able to perform all necessary traffic conditioning functions on the incoming traffic.

When conditioning is done at the boundary of a DS domain (at DS egress node when flows are leaving the domain or a DS ingress node when flows are entering the domain) the SLA between the domains should specify which domain has responsibility to assigning traffic streams to behavior aggregates and later conditioning those aggregates, but, a very important consideration has to be taken: no matter where the flows are coming, it is responsibility of the DS ingress node of any DS-capable domain to check (re-check) the entering flows and to be prepared to enforce the TCA in accordance with local policy. This way we protect the DS-capable domain from flows coming to it.

Also, probably, less resources are required to classifying and conditioning traffic in the downstream DS domain. Being closer to the source less aggregation has to be managed over lower throughput flows. Of course it is going to depend of the kind of services to be offered. Finally, as is expected, being the upstream domain a non-DS-capable domain all classification and conditioning must be done, when necessary, at the downstream receiving domain.

3. In non-DS-Capable Domains

Traffic sources or intermediate nodes in a non-DS-capable domain may employ traffic conditioners to pre-mark traffic before it reaches the ingress of a downstream DS domain. In this way the local policies for classification and marking may be concealed.

This paragraph talk about interaction between non-DS-capable domain and DS-capable domains. Some conditioning could be done at the upstream non-DS-capable domain before flows reach and enter the downstream DS-capable domain. Again downstream DS-capable domain has to enforce the TCA to fullfil its local policies.

4. In Interior DS Nodes

Although the basic architecture assumes that complex classification and traffic conditioning functions are located only in a network's ingress and egress boundary nodes, deployment of these functions in the interior of the network is not precluded. For example, more restrictive access policies may be enforced on a transoceanic link, requiring MF classification and traffic conditioning functionality in the upstream node on the link. This approach may have scaling limits, due to the potentially large number of classification and conditioning rules that might need to be maintained.

Normally, as we have seen through our explanations, conditioning is better done at boundary nodes where aggregation is fewer and lower throughput has to be managed. However, when required, deployment of these functions can be done in the interior nodes always having into account to preserve scaling management of the network.

Rest of the RFC 2475 specification is dedicated to the Per-Hop Behavior definition and a long explanation describing guidelines for PHB specifications. To preserve the integrity of the Differentiated Service architecture any PHB to be proposed for standardization should satisfy these guidelines. We are not going to go deeper with this theme and those of you interested in better information are encouraged to have a read to the original RFC 2475 specification. However, we will present some brief approach to the PHB definition taken directly from the specification, with some comments to clear what we are reading.

A per-hop behavior (PHB) is a description of the externally observable forwarding behavior of a DS node applied to a particular DS behavior aggregate. "Forwarding behavior" is a general concept in this context.

Useful behavioral distinctions are mainly observed when multiple behavior aggregates compete for buffer and bandwidth resources on a node. The PHB is the means by which a node allocates resources to behavior aggregates, and it is on top of this basic hop-by-hop resource allocation mechanism that useful differentiated services may be constructed.

The most simple example of a PHB is one which guarantees a minimal bandwidth allocation of X% of a link (over some reasonable time interval) to a behavior aggregate. This PHB can be fairly easily measured under a variety of competing traffic conditions. A slightly more complex PHB would guarantee a minimal bandwidth allocation of X% of a link, with proportional fair sharing of any excess link capacity.

Okay. We have to remember that first we classify flows by classes called "Behavior Aggregate" (BA); next we select a DS codepoint to identify each BA. When a flow is entering our domain, we, using our classifier (MF or DS codepoint), classify it into one of our predefined BAs. Depending on the BA selected we mark or re-mark the DS-codepoint on each packet header. Also, probably, we can make some conditioning at this time, mainly to protect ourself from misbehaved flows and trying that everyone entering the domain respect our inner rules. Up to here everything is clear.

But, what happen within the domain with all these flows classified by BA? We need to have some mechanism to assign different treatments because as we stated before each BA will be treated differently; some will be treated as kings or queens, some very well, some not so well, some bad and

some really very bad. Our domain is a discriminatory world. Well, these treatments are what Differentiated Service architecture called Per Hop Behavior (PHB). How each BA will be forwarded within our domain it's going to depend of the PHB assigned to the BA. We have here a mapping between the BAs and the PHBs. Every BA is mapped to its corresponding PHB.

How do we define or establish these PHBs or treatments? Really easy, by assigning resources of our domain to each of them. It's like the world; some are crude rich, some are really rich, some just rich, and going down, some are poor, some very poor, and finally some are crude poor. What are the resources we are going to distribute between our PHBs? Basically buffer and bandwidth resources. Authors give also two very simple examples: a PHB which guarantee a minimal bandwidth allocation of X% of the total link bandwidth and another PHB with the same policy but having the possibility of a proportional fair sharing of any excess link capacity.

PHBs may be specified in terms of their resource (e.g., buffer, bandwidth) priority relative to other PHBs, or in terms of their relative observable traffic characteristics (e.g., delay, loss). These PHBs may be used as building blocks to allocate resources and should be specified as a group (PHB group) for consistency. PHB groups will usually share a common constraint applying to each PHB within the group, such as a packet scheduling or buffer management policy.

PHBs are implemented in nodes by means of some buffer management and packet scheduling mechanisms. PHBs are defined in terms of behavior characteristics relevant to service provisioning policies, and not in terms of particular implementation mechanisms. In general, a variety of implementation mechanisms may be suitable for implementing a particular PHB group. Furthermore, it is likely that more than one

PHB group may be implemented on a node and utilized within a domain. PHB groups should be defined such that the proper resource allocation between groups can be inferred, and integrated mechanisms can be implemented which can simultaneously support two or more groups. A PHB group definition should indicate possible conflicts with previously documented PHB groups which might prevent simultaneous operation.

When specifying resource allocation we can use some relative measure between PHBs always based on the total resources available, or we can assign some absolute values. In general it's better to use relative distribution of resources, this way when those resources increment a fair sharing of them can be achieved. On the other hand some upper levels or maximum resource consuming values have to be implemented to be sure that misbehaved flows will not starve our domain behavior.

An example is useful here to clear what we are trying to say: in a boundary node we can have 3 flows that we decided to distribute in this form: A (30%); B (40%); and C (30%). These are relative values based on the total bandwidth available at the boundary node. But also we can establish some absolute limits to these flows; talking about maximum bandwidth permitted we can have: A (3 Mbps); B (1.5 Mbps); and C (2 Mbps). Let's suppose that at some time we can rely on with 4 Mbps at this node; as soon as flows A, B and C can reclaim its right, A can rely on with 1.2 Mbps, B with 1.6 Mbps, and C with 1.2 Mbps. Having all these flows enough throughput to violate their levels then A=1.2 Mbps, B=1.6 Mbps, and C=1.2 Mbps will be the throughput levels.

But, what about when one of these flows is using less of its share bandwidth permitted? This time other flows can reclaim and use this free bandwidth for them. Now upper levels established enter the game. Every flow could, as soon as bandwidth is available, have a higher share of the total bandwidth but the upper levels to be accepted will be: A (3 Mbps); B (1.5 Mbps); and C (2 Mbps).

Staying reading from the specification we have:

As described in [DSFIELD], a PHB is selected at a node by a mapping of the DS codepoint in a received packet. Standardized PHBs have a recommended codepoint. However, the total space of codepoints is larger than the space available for recommended codepoints for standardized PHBs, and [DSFIELD] leaves provisions for locally configurable mappings. A codepoint->PHB mapping table may contain both 1->1 and N->1 mappings.

All codepoints must be mapped to some PHB; in the absence of some local policy, codepoints which are not mapped to a standardized PHB in accordance with that PHB's specification should be mapped to the Default PHB.

The implementation, configuration, operation and administration of the supported PHB groups in the nodes of a DS Domain should effectively partition the resources of those nodes and the inter-node links between behavior aggregates, in accordance with the domain's service provisioning policy. Traffic conditioners can further control the usage of these resources through enforcement of TCAs and possibly through operational feedback from the nodes and traffic conditioners in the domain. Although a range of services can be deployed in the absence of complex traffic conditioning functions (e.g., using only static marking policies), functions such as policing, shaping, and dynamic re-marking enable the deployment of services providing quantitative performance metrics.

[DSFIELD] is the RFC 2474 specification which we have talked above. Refreshing knowledge a mapping exists between a BA identified by its specific DS-codepoint and once of our PHBs. PHBs are descriptions or specifications of how a specific BA will be treated throughout the domain; how much resources are going to be reserved for the BA and which are going to be the rules to be followed to manage it.

Because, probably, the total space of codepoints could be larger than the total space of standardized PHBs the mapping table could contain 1 ® 1 relations or N ® 1 relations. It's very important to have clear that PHB space should be standardized; this means, to propose a new PHB,



www.vyatta.com
Ads by Goooooogle

including the DS-codepoint suggested, the proponent has to follow the specification guidelines outlined in RFC 2475 specification. The proposition has to be revised and approved before being accepted as a standard.

To avoid problems with orphaned BAs every codepoint must be mapped to some PHB, but, when your domain doesn't find a mapping between an entering DS-codepoint and the PHB availability a default PHB must be already implemented to manage these cases. Normally, the default PHB is not more than the always implemented per hop behavior known as "best-effort".

Finally, it is responsibility of the domain administration to implement, configure, operate and manage the domain such that an effectively and fair distribution of available resources can be done at boundary and internal nodes between behavior aggregates to be managed, in accordance with the domain's service provisioning policy. To reach these goals a judicious employment of available tools has to be done to implement the traffic conditioners required for the employment of services, providing quantitative performance metrics.

To step ahead with our study of Differentiated Service architecture we are going to put our eyes on the, up to now, proposed and accepted PHB that exist. These are two: "Assure Forwarding PHB Group" and "Expedited Forwarding PHB". They are specified in RFC 2597 and RFC 2598 specifications, respectively.



[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

1.5.- Assure Forwarding PHB Group



Search

Continuing with our method of study let's start again presenting the original specification, this time the RFC 2597 specification, and then doing some comments when required.

This document defines a general use Differentiated Services (DS) [Blake] Per-Hop-Behavior (PHB) Group called Assured Forwarding (AF). The AF PHB group provides delivery of IP packets in four independently forwarded AF classes. Within each AF class, an IP packet can be assigned one of three different levels of drop precedence. A DS node does not reorder IP packets of the same microflow if they belong to the same AF class.

According to this, the new PHB is going to have four classes to be known as AF classes. We saw somewhere above that Differentiated Service architecture is based on classes of service that are identified by using the 3-leftmost bits of the DS-codepoint. But something very interesting is being proposed here: within each AF class an IP packet can be assigned to one of three different levels of drop precedence. What do we have here? First, IP packets of the same microflow can not be reordered; just common sense to protect the connection behavior. Second, within a class we can have three different treatments, or better yet, three different subclasses. How do we discriminate between subclasses? Just by using something called "drop precedence". Let's continue reading for a better definition.

Within each AF class IP packets are marked (again by the customer or the provider DS domain) with one of three possible drop precedence values. In case of congestion, the drop precedence of a packet determines the relative importance of the packet within the AF class. A congested DS node tries to protect packets with a lower drop precedence value from being lost by preferably discarding packets with a higher drop precedence value.

Very interesting. The drop precedence of a packet is not more than the relative importance of the packet within the class. As higher the drop precedence of a packet is it's going to be higher the probability that this packet can be discarded (dropped) when things go worse and congestion begin to destroy our happy world. Observe here that what they are trying to implement is what we called before a "discriminatory world". We have not only four different classes to classify our packets (citizens); using our criteria we can assign different network resources to each of these classes, but also, within the same class we can extend our hierarchy even more allowing some packets to have a better probability to survive than other, in case of congestion.

Observe that congestion is the devil that fires this last sub-hierarchy. Even when congestion is present or not resource distribution is going to be done between AF classes according to some previously specified rules (policies). This first hierarchy defines primarily a resource distribution. But when congestion appears we fire our second hierarchy control treating some packets better than other according to what is called the "drop precedence". Up to here everything is clear but let's continue reading the specification to see what they are reserved for our knowledge appetite.

In a DS node, the level of forwarding assurance of an IP packet thus depends on (1) how much forwarding resources has been allocated to the AF class that the packet belongs to, (2) what is the current load of the AF class, and, in case of congestion within the class, (3) what is the drop precedence of the packet.

[End to End QoS and more](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[Packet Sniffing](#)

Advanced Network Analyzer Solution. Start Analyzing Today. Download!
www.Paessler.com/Packet-Sniffing

[SIP call troubleshooter](#)

Real-time VoIP Traffic monitoring Free Trial & Support. Download now!
www.vqmanager.com

[AirMagnet Free-Trial](#)

Test/Audit/Fix your WLAN with Industry-leading Wi-Fi analyzer
www.airmagnet.com

[CESoPSN & SAToP Solutions](#)

E1/T1/J1 Over IP,MPLS,EPON Networks For Next Generation Networks
www.techaya.com

For example, if traffic conditioning actions at the ingress of the provider DS domain make sure that an AF class in the DS nodes is only moderately loaded by packets with the lowest drop precedence value and is not overloaded by packets with the two lowest drop precedence values, then the AF class can offer a high level of forwarding assurance for packets that are within the subscribed profile (i.e., marked with the lowest drop precedence value) and offer up to two lower levels of forwarding assurance for the excess traffic.

Overwhelming. No doubt. These paragraphs show us that we are in presence of one of the most flexible and powerful technology for QoS services with the additional advantage of requiring limited resources to be implemented. Flexible, powerful and scalable. Possibilities are endless. Really an amazing technology.

Assured Forwarding (AF) PHB group provides forwarding of IP packets in N independent AF classes. Within each AF class, an IP packet is assigned one of M different levels of drop precedence. An IP packet that belongs to an AF class i and has drop precedence j is marked with the AF codepoint AFij, where $1 \leq i \leq N$ and $1 \leq j \leq M$. Currently, four classes ($N=4$) with three levels of drop precedence in each class ($M=3$) are defined for general use. More AF classes or levels of drop precedence MAY be defined for local use.

A DS node SHOULD implement all four general use AF classes. Packets in one AF class MUST be forwarded independently from packets in another AF class, i.e., a DS node MUST NOT aggregate two or more AF classes together.

A DS node MUST allocate a configurable, minimum amount of forwarding resources (buffer space and bandwidth) to each implemented AF class. Each class SHOULD be serviced in a manner to achieve the configured service rate (bandwidth) over both small and large time scales.

An AF class MAY also be configurable to receive more forwarding resources than the minimum when excess resources are available either from other AF classes or from other PHB groups. This memo does not specify how the excess resources should be allocated, but implementations MUST specify what algorithms are actually supported and how they can be parameterized.

Here authors begin to refine the PHB Group specification. We can have up to N classes and up to M levels of drop precedence within each class. Initially four classes and three level of drop precedence are defined for general use. However, more AF classes and levels of drop precedence can be defined for local use. Each node should implement all four AF classes and for each class a minimum amount of forwarding resources has to be allocated. These resources are the minimum guaranteed to each class but when more resources are available they have to be shared between the AF classes being implemented.

It is also very important to note how the different subclasses are named; an IP packet that belong to an AF class i having drop precedence j is marked with the AF codepoint AFij. Having four classes and three drop precedences the twelve subclasses would be named: AF11..AF13; AF21..AF23; AF31..AF33; and AF41..AF43.

Within an AF class, a DS node MUST NOT forward an IP packet with smaller probability if it contains a drop precedence value p than if it contains a drop precedence value q when $p < q$. Note that this requirement can be fulfilled without needing to dequeue and discard already-queued packets.

Within each AF class, a DS node MUST accept all three drop precedence codepoints and they MUST yield at least two different levels of loss probability. In some networks, particularly in enterprise networks, where transient congestion is a rare and brief occurrence, it may be reasonable for a DS node to implement only two different levels of loss probability per AF class. While this may suffice for some networks, three different levels of loss probability SHOULD be supported in DS domains where congestion is a common occurrence.

If a DS node only implements two different levels of loss probability for an AF class x, the codepoint AFx1 MUST yield the lower loss probability and the codepoints AFx2 and AFx3 MUST yield the higher loss probability.

A DS node MUST NOT reorder AF packets of the same microflow when they belong to the same AF class regardless of their drop precedence. There are no quantifiable timing requirements (delay or delay variation) associated with the forwarding of AF packets.

The probability of any packet to be forwarded is higher as its drop precedence is lower; in any circumstances when a packet need to be dropped those of them having higher drop precedence will have also the higher probability to be selected for dropping. A DS node must accept packets with all three drop precedence codepoint and must implement at least two level of loss probability when transient congestion is rare, and all three levels of loss probability when congestion is a common occurrence. In those cases where only two levels of loss probability are implemented, packets belonging to classes having codepoint AFx1 will be subjugated to the lower loss probability, and those belonging to classes having codepoint AFx2 and AFx3 will be subjugated to the higher loss probability.

Observe also that the definition of Assure Forwarding PHB Group does not establish any quantifiable requirement to the delay or delay variation (jitter) that a packet could be suffering during the forwarding process.

A DS domain MAY at the edge of a domain control the amount of AF traffic that enters or exits the domain at various levels of drop precedence. Such traffic conditioning actions MAY include traffic shaping, discarding of packets, increasing or decreasing the drop precedence of packets, and reassigning of packets to other AF classes. However, the traffic conditioning actions MUST NOT cause reordering of packets of the same microflow.

Okay. Nothing really new. Observe, however, that remarking allow to change the class or subclass assigned to a packet. Conditioning must respect the packet ordering of the same microflow.

An AF implementation MUST attempt to minimize long-term congestion within each class, while allowing short-term congestion resulting from bursts. This requires an active queue management algorithm. An example of such an algorithm is Random Early Drop (RED) [Floyd]. This memo does not specify the use of a particular algorithm, but does require that several properties hold.

An AF implementation MUST detect and respond to long-term congestion within each class by dropping packets, while handling short-term congestion (packet bursts) by queueing packets. This implies the presence of a smoothing or filtering function that monitors the instantaneous congestion level and computes a smoothed congestion level. The dropping algorithm uses this smoothed congestion level to determine when packets should be discarded.

The dropping algorithm MUST be insensitive to the short-term traffic characteristics of the microflows using an AF class. That is, flows with different short-term burst shapes but identical longer-term packet rates should have packets discarded with essentially equal probability. One way to achieve this is to use randomness within the dropping function.

The dropping algorithm MUST treat all packets within a single class and precedence level identically. This implies that for any given smoothed congestion level, the discard rate of a particular microflow's packets within a single precedence level will be proportional to that flow's percentage of the total amount of traffic passing through that precedence level.

The congestion indication feedback to the end nodes, and thus the level of packet discard at each drop precedence in relation to congestion, MUST be gradual rather than abrupt, to allow the overall system to reach a stable operating point. One way to do this (RED) uses two (configurable) smoothed congestion level thresholds. When the smoothed congestion level is below the first threshold, no packets of the relevant precedence are discarded. When the smoothed congestion level is between the first and the second threshold, packets are discarded with linearly increasing probability, ranging from zero to a configurable value reached just prior to the second threshold. When the smoothed congestion level is above the second threshold, packets of the relevant precedence are discarded with 100% probability.

I took all this part of the specification in a block because they are shouting here that you have to use RED queuing discipline to implement Assure Forwarding PHB Group. The specification calls for not specifying the use of a particular algorithm, but what they are really specifying here is not more than the RED queuing discipline behavior. RED gateways were studied by various authors but finally, in 1993, Floyd and Jacobson presented a very complete study in their paper "*Random Early Detection Gateways for Congestion Avoidance*" [13]. Later below, when studying tools for implementing Differentiated Service, we are going to talk a little long about RED queuing discipline. Because of this we will postpone additional comments to a better occasion.

Recommended codepoints for the four general use AF classes are given below. These codepoints do not overlap with any other general use PHB groups.

The RECOMMENDED values of the AF codepoints are as follows:

AF11 = '001010', AF12 = '001100', AF13 = '001110',
 AF21 = '010010', AF22 = '010100', AF23 = '010110',
 AF31 = '011010', AF32 = '011100', AF33 = '011110',
 AF41 = '100010', AF42 = '100100', AF43 = '100110'.

The table below summarizes the recommended AF codepoint values.

	Class 1	Class 2	Class 3	Class 4
Low Drop Prec	001010	010010	011010	100010
Medium Drop Prec	001100	010100	011100	100100
High Drop Prec	001110	010110	011110	100110

Table 1.5.1

Finally we have the recommended values of the AF codepoints. Four classes and three subclasses (drop precedence) for each of them. But let's stop a little here to have a look to codepoints. They are six bits long as RFC 2474 specification calls. Remember also that the class should be defined using the 3-leftmost bits. Taking these we have a simple way to remember the class part of the codepoints:

001 = 1 = class 1
 010 = 2 = class 2
 011 = 3 = class 3
 100 = 4 = class 4

Next let's do something similar with the 3-rightmost bits that are used to specify the drop precedence or subclass:

010 = 2 = low drop precedence
 100 = 4 = medium drop precedence
 110 = 6 = high drop precedence

Okay. The rule is very simple. Classes are defined with the first 3-bits and they are just 1-2-3-4. Subclasses are defined with the last 3-bits and they are just 2-4-6.

[Paper: Mobile Backhaul](#)

Expanding 3G backhaul capacity without jeopardizing 2G revenues
www.ciena.com

[Online Surface Tension](#)

Measure surface tension on-line or at-line for process control
www.ksvlt.com

[Deep Packet Inspection](#)

A User-driven Community for DPI/DPP Learn, discuss, and collaborate
www.dPacket.org

[DDoS Protection Leader](#)

Renowned anti-DDoS service. Full SLA provided. Global network.
www.prolexic.com

[Calif. Rail Quiet Zone](#)

complex transportation, project mgt traffic congestion relief
www.ChrisBecker.com

You must be thinking why I'm bother you with all this explanation about classes, subclasses and bits. However, when trying to implement differentiated services it is absolutely necessary to have a clear understanding of how to compose the codepoint of a class. For example, how to compose the codepoint of class AF32?

Very easy. The class is 3 then the 3-leftmost bits are 011. The drop precedence is 2 (medium); because low-medium-high correspond to 2-4-6, medium drop precedence is 4, then the 3-rightmost bits are 100. Then AF32 is 011100. Try now yourself to find the codepoint for class AF43 as an exercise. Next check above with values taken directly from the specification.

To end the theme about codepoint you may be asking: numbering classes as 1-2-3-4 is really nice, but, why are drop precedence identified by 2-4-6 instead of 1-2-3? Reason is that they are trying to preserve the rightmost bit (bit number six) to indicate another condition. Do you remember what we talked before about in-profile and out-of-profile traffic? Let's refresh what we studied. A flow is entering our domain but we have established what is called a threshold for this kind of traffic. A throughput up to 1.5 Mbps (just as an example) is considered as in-profile traffic because our TCA calls for fullfil this condition. Above this level (our threshold) the traffic is considered out-of-profile traffic. Well, having an absolute independence of the final class where this traffic is going to be located (class 1, 2, 3 or 4), or even the drop precedence (subclass 2, 4 or 6), we can extend even more our already two level hierarchy, marking out-of-profile packets by setting the rightmost bit of the DS-codepoint. Let's suppose that packets belonging to this traffic are going to be assigned to class AF23. What the codepoints are going to be?

For in-profile traffic the codepoint will be 010110 (have a look to the table above or even better use your aid rule to get the code). For out-of-profile traffic we simply set the rightmost bit. Then codepoint for these packets will be 010111. Really nice!!.

Let's step ahead a little more. Let's imagine how to create a PHB for these packets. Again, as an example, we could say: traffic whose packets belong to this class (class 2), are going to be reserved 12% of the available resources at our router; being its drop precedence high (subclass 3), they are going to be subjugated to a drop probability of 4% (for every 100 packets 4 of them, in case of congestion, are probably killed). Up to a throughput of 1.5 Mbps these packets are considered in-profile and treated as was indicated (12% of share - 4% of drop probability). Above this rate traffic is considered out-of-profile and we can change our treatment. How? Well, as you decide; it's a matter of the domain manager responsibility (and the agreements, of course). You could decide or have a deal that out-of-profile traffic is simply not admitted and discarded before entering. Or perhaps a somewhat less strict policy, for example, they could be accepted but being subjugated to a drop policy of 100% in case of congestion.

Have a look to this table:

Class	Resources	Drop probability					
		Subclass 1		Subclass 2		Subclass 3	
		In	Out	In	Out	In	Out
1	40%	2%	50%	3%	60%	4%	70%
2	25%	2%	60%	4%	80%	5%	80%
3	20%	3%	70%	5%	80%	10%	90%
4	15%	3%	100%	6%	100%	10%	100%

Table 1.5.2

This is just an example from my exhausted mind. Do you catch the endless possibilities? The DS architecture allows you a broad room of conditioning for packet forwarding. It's really a very flexible and powerful technology requiring only, as we are going to see later, limited resources to be implemented. Continuing with RFC 2597 we have:

The AF codepoint mappings recommended above do not interfere with the local use spaces nor the Class Selector codepoints recommended in [Nichols]. The PHBs selected by those Class Selector codepoints may thus coexist with the AF PHB group and retain the forwarding behavior and relationships that was defined for them. In particular, the Default PHB codepoint of '000000' may remain to be used for conventional best effort traffic. Similarly, the codepoints '11x000' may remain to be used for network control traffic.

It's just a matter to respect our similar. We must coexist with some other technologies. The Default PHB codepoint of '000000', where we do not touch the packet header, should remain being used as an indicator of best effort traffic; the PHB that everybody know and use just now.

And finally, to finish with RFC 2597, let's copy here an example taken from the Appendix's specification. I'm not going to make comments about it. Reach yourself your own conclusions. Next we are going to study "Expedited Forwarding PHB" outlined in RFC 2598 specification.

The AF PHB group could be used to implement, for example, the so-called Olympic service, which consists of three service classes: bronze, silver, and gold. Packets are assigned to these three classes so that packets in the gold class experience lighter load (and thus have greater probability for timely forwarding) than packets assigned to the silver class. Same kind of relationship exists between the silver class and the bronze class. If desired, packets within each class may be further separated by giving them either low, medium, or high drop precedence.

The bronze, silver, and gold service classes could in the network be mapped to the AF classes 1, 2, and 3. Similarly, low, medium, and high drop precedence may be mapped to AF drop precedence levels 1, 2, or 3.

The drop precedence level of a packet could be assigned, for example, by using a leaky bucket traffic policer, which has as its parameters a rate and a size, which is the sum of two burst values: a committed burst size and an excess burst size. A packet is assigned low drop precedence if the number of tokens in the bucket is greater than the excess burst size, medium drop precedence if the number of tokens in the bucket is greater than zero, but at most the excess burst size, and high drop precedence if the bucket is empty. It may also be necessary to set an upper limit to the amount of high drop precedence traffic from a customer DS domain in order to avoid the situation where an avalanche of undeliverable high drop precedence packets from one customer DS domain can deny service to possibly deliverable high drop precedence packets from other domains.

Another way to assign the drop precedence level of a packet could be to limit the user traffic of an Olympic service class to a given peak rate and distribute it evenly across each level of drop precedence. This would yield a proportional bandwidth service, which equally apportions available capacity during times of congestion under the assumption that customers with high bandwidth microflows have subscribed to higher peak rates than customers with low bandwidth microflows.

The AF PHB group could also be used to implement a loss and low latency service using an over provisioned AF class, if the maximum arrival rate to that class is known a priori in each DS node. Specification of the required admission control services, however, is beyond the scope of this document. If low loss is not an objective, a low latency service could be implemented without over provisioning by setting a low maximum limit to the buffer space available for an AF class.



www.vyatta.com
Ads by Goooooogle



[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

1.6.- Expedited Forwarding PHB

Continuing our study of Differentiated Service we are going to see now our last PHB, named "Expedited Forwarding PHB" (EF PHB). Without losing any time let's start copying here for comments the introduction taken directly from the RFC 2598 specification:

Network nodes that implement the differentiated services enhancements to IP use a codepoint in the IP header to select a per-hop behavior (PHB) as the specific forwarding treatment for that packet [RFC2474, RFC2475]. This memo describes a particular PHB called expedited forwarding (EF). The EF PHB can be used to build a low loss, low latency, low jitter, assured bandwidth, end-to-end service through DS domains. Such a service appears to the endpoints like a point-to-point connection or a "virtual leased line". This service has also been described as Premium service [2BIT].

Loss, latency and jitter are all due to the queues traffic experiences while transiting the network. Therefore providing low loss, latency and jitter for some traffic aggregate means ensuring that the aggregate sees no (or very small) queues. Queues arise when (short-term) traffic arrival rate exceeds departure rate at some node. Thus a service that ensures no queues for some aggregate is equivalent to bounding rates such that, at every transit node, the aggregate's maximum arrival rate is less than that aggregate's minimum departure rate.

Well. This PHB is some sort of fast-track service. Low all. Guaranteed bandwidth. The golden dream of any application requiring a "First Class" service. An a lot of applications are just as demanding. Specially those of them that have to deal with real-time services. Real-time service traffic requires a very fast and safe service for the moving of data from sources to destinations. EF PHB is the answer offered by DS to these kind of demanding flows. The service is so guarantee that is as having a private pipe or "virtual leased line" to move our data. But, wait a minute. How can we offer this kind of excellent service to our VIP customers?

As the specification assertively explains, loss, latency and jitter are all due to the queues flows experience while transit the network. These queues are formed on routers along the path. Because is almost impossible to avoid these queues, then given to some flows this VIP service is a matter of creating for them a special path on routers that bypass the queues. Something similar as occur at custom's airports when diplomatic people have one special and priviledged attention desk that hurry up the bothering process. Sonething very disagreeable for us, normal mortals, but that in fact exists and is used.

To ensure this special fast service we have to configure our routers such that, for these flows, the aggregate's minimum departure rate should be always greater than the aggregate's maximum arrival rate. For reaching this goal our routers have to be enough powerful to forward, as fast as possible, the total throughput that is intended to cross through them; and for class-I flows this forwarding will be done on special fast-track queues specifically designed and configured to fullfil these requirements. Do not forget this. Routers have to be enough capable. It doesn't have any sense to configure special fast-track queues on routers that do not have enough capacity to forward, comfortably, the total throughput that you are thinking to move through them.

Creating such a service has two parts:

[QoS Management Tools](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[Network Traffic Analyzer](#)

Advanced Packet Analysis. A "must have" networking tool.
www.Paessler.com

[TransModeler](#)

The world's most powerful traffic simulation software
www.caliber.com/transmodeler

[Paper: Mobile Backhaul](#)

Expanding 3G backhaul capacity without jeopardizing 2G revenues
www.ciena.com

[Aimsun and Aimsun Online](#)

Powerful simulators for transport planning and traffic management.
www.aimsun.com

1. *Configuring nodes so that the aggregate has a well-defined minimum departure rate. ("Well-defined" means independent of the dynamic state of the node. In particular, independent of the intensity of other traffic at the node.)*
2. *Conditioning the aggregate (via policing and shaping) so that its arrival rate at any node is always less than that node's configured minimum departure rate.*

The EF PHB provides the first part of the service. The network boundary traffic conditioners described in [RFC2475] provide the second part.

Okay. EF PHB will be in charge of guaranteeing a minimum departure rate for some BA; this rate will be independent of the rest of the traffic to be forwarded by the router at the same time. On the other hand conditioning, as was seen when we studied RFC 2475 specification, will be in charge of controlling arriving rate to fulfill the requirement that departure rate is always greater than arrival rate.

The EF PHB is defined as a forwarding treatment for a particular diffserv aggregate where the departure rate of the aggregate's packets from any diffserv node must equal or exceed a configurable rate. The EF traffic SHOULD receive this rate independent of the intensity of any other traffic attempting to transit the node. It SHOULD average at least the configured rate when measured over any time interval equal to or longer than the time it takes to send an output link MTU sized packet at the configured rate. (Behavior at time scales shorter than a packet time at the configured rate is deliberately not specified.) The configured minimum rate MUST be settable by a network administrator (using whatever mechanism the node supports for non-volatile configuration).

Here authors describe EF PHB behavior. Observe that specification concentrates in guaranteeing a forwarding treatment for a particular BA having a departure rate that must be equal or exceed a configurable rate. Conditioning on enter will guarantee that arrival rate is going to be below this previously selected rate.

If the EF PHB is implemented by a mechanism that allows unlimited preemption of other traffic (e.g., a priority queue), the implementation MUST include some means to limit the damage EF traffic could inflict on other traffic (e.g., a token bucket rate limiter). Traffic that exceeds this limit MUST be discarded. This maximum EF rate, and burst size if appropriate, MUST be settable by a network administrator (using whatever mechanism the node supports for non-volatile configuration). The minimum and maximum rates may be the same and configured by a single parameter.

This part is very important. It's really no easy to establish quantifiable settings to slippery concepts such as loss, delay and jitter. What is normally done with traffic asking for these requirements is to put them in specially designed queues with the highest priority dispatch and the lowest drop probability. Flows managed in these queues are forwarded as soon as possible having priority over other concurrent flows (preemption). They are immune to drop policies and only in case of

extremely congestion situations could the router attempt against them. But, beware, implementing this kind of "royalty" will certainly destroy the forwarding opportunity of other flows. If the "royalty" is not put under strict control surely it will starve the performance behavior of the rest of mortals. That's because specification calls for an upper limit for the rate, and burst size if appropriate, for these aggregates. These maximums must be settable by the network administrator. Traffic that exceeds this limit must be discarded.

Several types of queue scheduling mechanisms may be employed to deliver the forwarding behavior described in section 2.1 and thus implement the EF PHB. A simple priority queue will give the appropriate behavior as long as there is no higher priority queue that could preempt the EF for more than a packet time at the configured rate. (This could be accomplished by having a rate policer such as a token bucket associated with each priority queue to bound how much the queue can starve other traffic.)

It's also possible to use a single queue in a group of queues serviced by a weighted round robin scheduler where the share of the output bandwidth assigned to the EF queue is equal to the configured rate. This could be implemented, for example, using one PHB of a Class Selector Compliant set of PHBs [RFC2474].

Another possible implementation is a CBQ [CBQ] scheduler that gives the EF queue priority up to the configured rate.

Here the authors try to give some ideas of how to implement the queuing discipline required to lend the EF PHB. They point out three possibilities. The simplest one is a priority queue where providences are taking to avoid starving EF for other flows or starving other flows for EF. A priority queue is a simple scheme where several queues are served in a hierarchy; as soon as there are packets in the highest priority queue they are served (forwarded) before the next priority queue is attended. When the highest priority queue is empty the next one in the hierarchy is served; following this way until the lowest one is served. As is obvious higher priority queues must have upper limits to the throughput to be forwarded as a mechanism to avoid starving lower priority queues.

The second scheme is a weighted round robin scheduler. In this scheme the queues are served in a round robin fashion and weights are assigned to point out how much of the total time is intended for serving each queue. Playing with the weights, the share of the output bandwidth assigned to the EF queue can be set to the configured rate (minimum



departure rate). More sophisticated schemes can include prioritization (preemption) of some queues.

The last scheme is based on the state-of-the-art CBQ (Class Based Queue) scheduler. CBQ is a highly sophisticated scheme where several mechanism basically merge PQ (priority queue) and fair capabilities to provide different kind of services to data traffic. CBQ distribute bandwidth between classes in a hierarchical link-sharing structure where each class correspond to some aggregation of traffic. Later on we are going to see how different queuing discipline implementations can be used to build the DS architecture. But now let's continue with the specification:

Codepoint 101110 is recommended for the EF PHB.

Finally the recommended codepoint is given. Just to make an analogy with AF PHB Group, observe that the class is number 5 (the next one). But the analogy borns and dies here because if we go ahead the drop precedence would be "high" and this is a contradiction. Why? Also the rightmost bit is preserved for signalling in-profile or out-of-profile traffic.

Well. With this we finished the theoretical part of our HOWTO. Next the practical part begins. I preferred to focus the issue this way because it doesn't have any sense to talk about DS architecture, behavior aggregates, per hop behaviors, marking, shapping, dropping, classes, etc., that will confuse the lector, if we do not establish, previously, a framework where everyone can talk, listen and understand the same. Perhaps someone could say that I misunderstood the specifications and then the framework is wrong. I admit this possibility. But in this case all of it is wrong. No untied pieces as generally occurs. And the confusion grows as you try to join the different authors.



This site is visited regularly for very specialized people; it's then the best place to advertise your very specialized product.
First month at no cost for you to verify this aseveration.

[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

2.0.- Linux Queuing Disciplines

To understand how Linux can help us in our attempt to build a Differentiated Service architecture is very important to understand first how Linux processes packets. To assemble our explanation let's ask for a little help from Werner Almesberger [6], Bert Hubert and Lartc people [7], Saravanan Radhakrishnan [8], Christian Worm Mortensen [9] and Werner Almesberger, Jamal Hadi Salim and Alexey Kuznetsov [10].

[Enterprise Systems Theory](#)

International Conference on Enterprises as Systems
www.enterprisesystemtheory.net

[Network Operating Systems](#)

Get the latest news, white papers, discussion threads, and much more.

Networking.ITtoolbox.com

[Free Download](#)

LogMeIn - Really 100% Free & Easy Access to PCs from Anywhere

www.LogMeIn.com

[End to End QoS for MPLS](#)

Automatic QoS for 600+ WAN Apps. Learn How - Free MPLS Whitepaper.

Packeteer.com

Ads by Google

The theme is very ample and I do not presume to give here a magisterial class (I'm not prepare, anyway); just a shallow review of concepts sufficiently enough to reach our goal of a better understanding of the Differentiated Service architecture on Linux. Those of you interested in having a deeper knowledge are encouraged to go directly to the sources, being the ultimate of them, the C Language code from the Linux Operating System kernel sources.

Note: you need to prepare your Linux box to implement the scripts given in this document. Have a look to <http://diffserv.sourceforge.net/> where a very clear explanation of how to configure your box can be found.

[Previous](#)[Content](#)[Next](#)

2.1.- Linux Traffic Control

Google

Search

Each network card (NIC = Network Interface Card) on a Linux box is driven by a Network Driver which control the hardware. The Network Driver acts as an exchange mechanism of packets between the Linux Networking Code and the physical network. This first approach can be viewed this way:

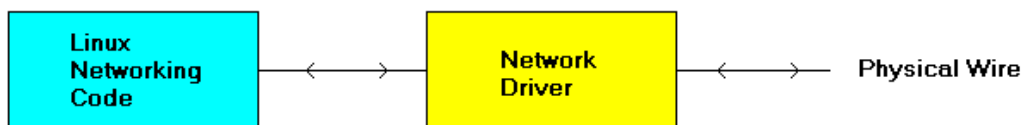


Figure 2.1.1

In this diagram by Christian [9] the Linux Networking Code can request the Network Driver to send a packet to the physical network (packet is leaving the box) or the Network Driver can give a packet it has received from the physical network to the Linux Networking Code (packet is entering the box).

Because our interest is to use Linux as a pure router to forward packets from one interface to another in the same box, it will not generate or consume any packet and our diagram can be extended to include two NICs:

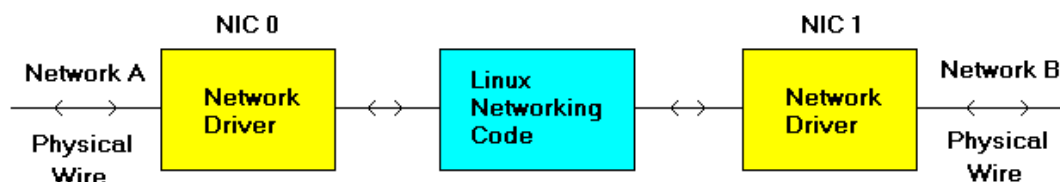


Figure 2.1.2

This way packets can enter from the physical network A to the NIC number 0; the NIC 0's Network Driver gives the received packets to the Linux Networking Code where they are forwarded; finally the Linux Networking Code request the NIC 1's Network Driver to send the packets to the physical network B.

Let's now put our eyes in the Linux Networking Code. Reading from [8] Saravanan wrote:

The basic principle involved in the implementation of QoS in linux is shown in figure below. This figure shows how the kernel processes incoming packets, and how it generates packets to be sent to the network. The input de-multiplexer examines the incoming packets to determine if the packets are destined for the local node. If so, they are sent to the higher layer for further processing. If not, it sends the packets to the forwarding block. The forwarding block, which may also received locally generated packets from the higher layer, looks up the routing table and determines the next hop for the packet. After this, it queues the packets to be transmitted on the output interface. It is at this point that the linux traffic control comes into play. Linux traffic control can be used to build a complex combination of queuing disciplines, classes and filters that control the packets that are sent on the output interface.

[Network/Ethernet Tester](#)

Tester for Switch & SOHO Router Network TAP / Power Monitor
www.xtramus.com

[SIP call troubleshooter](#)

Real-time VoIP Traffic monitoring Free Trial & Support. Download now!
www.vqmanager.com

[End to End QoS and more](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[AirMagnet Free-Trial](#)

Test/Audit/Fix your WLAN with Industry-leading Wi-Fi analyzer
www.airmagnet.com

[DDoS Protection Leader](#)

Renowned anti-DDoS service. Full SLA provided. Global network.
www.prolexic.com

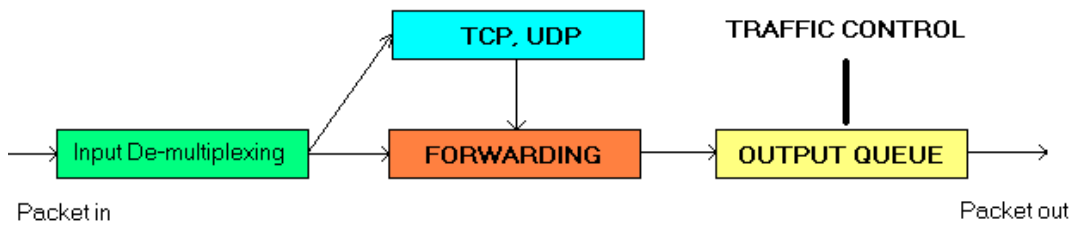


Figure 2.1.3

Very well. But we are going to refine things a little. First, we are not interested in local process, this means, we will not accept packets for local and we will not generate packets from local. Our Linux box will be a pure router: no packets are generated, no packets are consumed, packets are just forwarded or perhaps dropped. This way your box is a transfer machine and it is a lot better if you keep it as this. Do not lend any service except forwarding with a router. Don't do that.

Observe also that after packets are forwarded by the forwarding block they are enqueued in an output queue previously to be transmitted on the output interface. On this output queue something called Linux Traffic Control takes care of packet management. Reading from Werner [8] we can have a better and detailed information including a very nice figure:

Figure below shows roughly how the kernel processes data received from the network, and how it generates new data to be sent on the network: packets arrive on via an input interface, where they may be policed. Policing discards undesirable packets, e.g. if traffic is arriving too fast. After policing, packets are either directly forwarded to the network (e.g. on a different interface, if the machine is acting as a router or a bridge), or they are passed up to higher layers in the protocol stack (e.g. to a transport protocol like UDP or TCP) for further processing. Those higher layers may also generate data on their own and hand it to the lower layers for tasks like encapsulation, routing, and eventually transmission.

"Forwarding" includes the selection of the output interface, the selection of the next hop, encapsulation, etc. Once all this is done, packets are queued on the respective output interface. This is the second point where traffic control comes into play. Traffic control can, among other things, decide if packets are queued or if they are dropped (e.g. if the queue has reached some length limit, or if the traffic exceeds some rate limit), it can decide in which order packets are sent (e.g. to give priority to certain flows), it can delay the sending of packets (e.g. to limit the rate of outbound traffic), etc. Once traffic control has released a packet for sending, the device driver picks it up and emits it on the network.

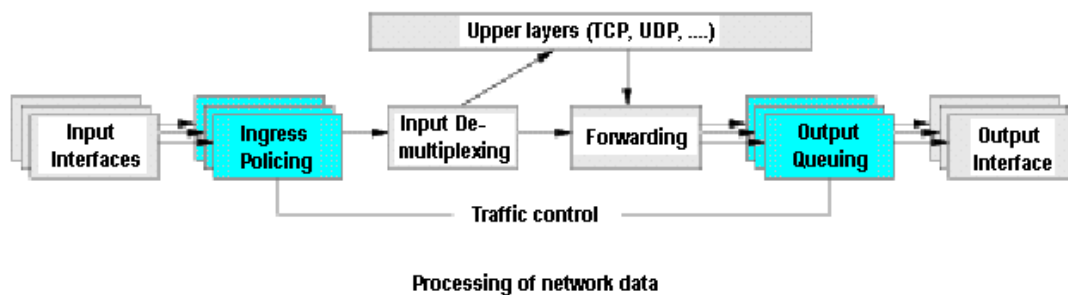


Figure 2.1.4

Werner explanation helps us a lot. Again to keep our box as a pure router no packets will be sent to the upper layers and no packets will be received from the upper layers. Our packet route will be then: Input Interface → Ingress Policing → Input Demultiplexing (just to say that packets are not for local use) → Forwarding → Output Queuing → Output Interface.

Where are we going to exercise control over our packets? Just into the blue blocks: Ingress Policing and Output Queuing. These blocks are called the "Traffic Control Code of the Linux kernel". Ingress policing will be our first point of control; here policing discards undesirable packets to enforce an upper limit on the entering throughput. The second

point will be the Output (Egress) Queue; here packets are queued and then dropped, delayed or prioritized according to the policy we are interested to implement.

Drilling down let's see how the Linux Traffic Control Code is implemented. As its name says it is just C Language code consisting of four major conceptual components:

- Queuing disciplines
- Classes
- Filters
- Policers

Queuing disciplines are algorithms which control how packets enqueued are treated. A very simple queuing discipline could be a FIFO (First In-First Out) queue of 16 packets. In this queue packets enter at the tail and leave the queue from the head. Something as simple like picture in figure 2.1.5 shown below.

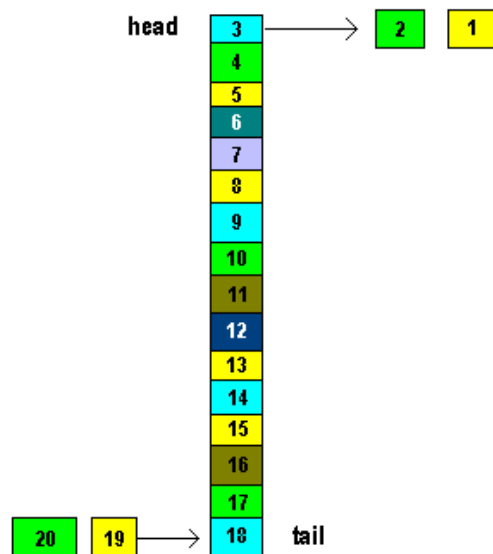


Figure 2.1.5

In this the queue is moving 20 packets. Packets 1 and 2 already entered and left the queue, packets 3 to 18 are just in the queue and packets 19 and 20 are in the process of entering the queue. Of course, before a new packet can be accepted another has to be dispatched. If packets arrive very fast, faster than they can be dispatched, the router doesn't have any other choice than dropping them. But we don't want this; we want to conserve our packets to protect TCP protocol behavior and applications than send/receive these packets. Or better yet, we want to exercise a true control over them: which are going to be forwarded and how fast; which are going to be delayed; and which, lamentably, dropped. The queue is an area of memory on our router. If we are expecting packets of say, 1000-byte size, we have to reserve an area of 16KB on memory to implement this simple 16-packets FIFO queue.

To step ahead we would like to extend now a little our knowledge about the Linux Traffic Control code. At this time Christian [9] explanation of the 'Class' concept come to us as a ring to the finger:

The basic building block of the Traffic Control system is qdiscs as described above. The queuing disciplines mentioned in the previous section are comparatively simple to deal with. That is, they are setup and maybe given some parameters. Afterwards packets can be enqueued to them and dequeued from them as described.

But many queuing are of a different nature. These qdiscs do not store packets themselves. Instead, they contain other qdiscs, which they give packets to and take packets from. Such qdiscs are known as qdiscs with classes.

For example, one could imagine a priority-based queuing discipline with the following properties:

1. Each packet enqueued to the queuing discipline is assigned a priority. For example the priority could be deduced from the source or destination IP address of the packet. Let us say that the priority is a number between 1 and 5.
2. When a packet is dequeued it will always select a packet it contains with the lowest priority number.

A way to implement such a queuing discipline is to make the priority-based queuing discipline contain 5 other queuing disciplines numbered from 1 to 5. The priority-based queuing discipline will then do the following:

1. When a packet is enqueued, it calculates the priority number, i.e. a number between 1 and 5. It then enqueues the packet to the queuing discipline indicated by this number
2. When a packet is dequeued it always dequeues from the non-empty queuing discipline with the lowest number.

What is interesting about this, is that the 5 contained queuing disciplines could be arbitrary queuing disciplines. For example sfq queues or any other queue.

In Linux this concept is handled by classes. That is, a queuing discipline might contain classes. In this example, the priority queuing discipline has 5 classes. Each class can be viewed as a socket to which you can plug in any other queuing discipline. When a qdisc with classes is created, it will typically assign simple FIFO queues to the classes it contains. But these can be replaced with other qdiscs by the tc program.

Bravo!! Some clearing: qdisc means queuing discipline; the qdisc he mentioned in a previous section was just a FIFO queue. But what he is magisterially describing here is a qdisc called PRIO queue; a queue with classes, certainly. SFQ is another of these qdiscs. And tc is a program written by Alexey Kuznetsov to manage qdiscs on Linux.

But Werner in [6] has something to say about this stuff; let's see:

Queuing disciplines and classes are intimately tied together: the presence of classes and their semantics are fundamental properties of the queuing discipline. But ability doesn't end yet - classes normally don't store their packets themselves, but they use another queuing discipline to take care of that. That queuing discipline can be arbitrarily chosen from the set of available queuing disciplines, and it may well have classes, which in turn use queuing disciplines, etc.

Nice, really nice. If we don't misunderstood the explanation we can have a tree with hierarchies; queuing disciplines have classes but these classes normally don't store their packets themselves, instead they use another queuing discipline to take care of that. And these queuing disciplines can have classes again which in turn use queuing disciplines and so on.

But let's review Christian [9] explanation above to extend even more our knowledge about Linux Traffic Control code. He wrote: "For example the priority could be deduced from the source or destination IP address of the packet". It sounds known for us. Search back this document and have a look to the "classifier" concept. Christian is trying to tell us that the PRIO qdisc he described has five classes numbered 1, 2, 3, 4 and 5. When a packet arrives to our router we snoop its IP header to see its source or destination address and based on either of these, or both, we select a priority from 1 to 5 for the packet; and with this priority on hand we put the packet in one of the classes of the qdisc according.

What are we doing? Just classifying the packet. We are telling: okay, you are coming from network, say, 203.16/16, then your priority will be 3; ... take the door number 3 please, where you will be attended.

[Deep Packet Inspection](#)

A User-driven Community for DPI/DPP Learn, discuss, and collaborate
www.dPacket.org

[ViewMaster -New & Vintage](#)

World marketplace for reels, packets, viewers, & supplies
www.3dstereo.com

[CESoPSN & SAToP Solutions](#)

E1/T1/J1 Over IP,MPLS,EPON Networks For Next Generation Networks
www.techaya.com

[Free Packet Sniffer](#)

Packet Sniffer - It's Free. Very advanced networking tool.
www.Paessler.com

[Free Erlang Calculator](#)

Model your call center staffing needs. What-if analysis, graphs
www.kooltoolz.com

To do this we need some kind of filter for selecting packets. Someone like a usher that when you arrive to the theater ask you: what your chair number is? Okay. Go there and search for row number 26, etc., etc. If you search back above a little, a filter is the third conceptual component of Werner [6] explanation. Let's read from him again:

Each network device has a queuing discipline associated with it, which controls how packets enqueued on that device are treated. Figure 2.1.6 shows the symbol we use for a queuing discipline without externally visible internal structure. For example, `sch_fifo` is such a simple queuing discipline, which just consists of a single queue, where all packets are stored in the order in which they have been enqueued, and which is emptied as fast as the respective device can send. More elaborate queuing disciplines may use filters to distinguish among different classes of packets and process each class in a specific way, e.g. by giving one class priority over other classes. Figure 2.1.7 shows an example of such a queuing discipline. Note that multiple filters may map to the same class.

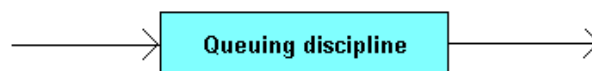


Figure 2.1.6

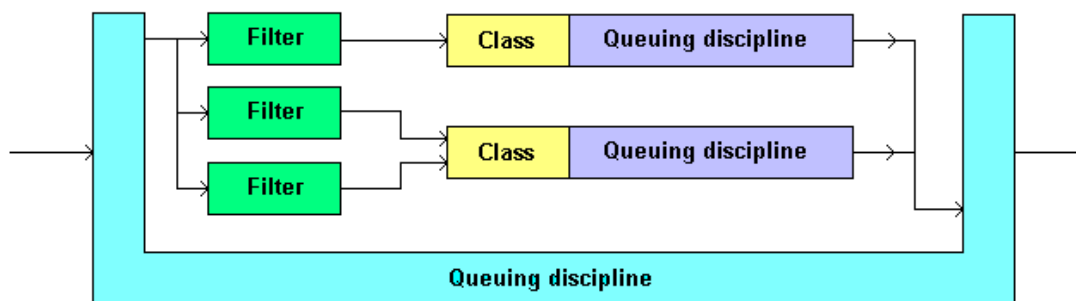


Figure 2.1.7

Eureka!! Now we have some pictures. It helps more than a hundred words. In fact this kind of pictures are going to be our everyday bread when dealing with queuing disciplines. It's a schematic and easy form of representing them. Looking at the pictures we can have a fast mental scheme of how is the qdisc behavior.

For example, looking at figure 2.1.7, packets enter the main qdisc by the left; immediately the classifier takes care of them and using filters select the class where each packet has to be put. When already in the class the queuing discipline on it takes care now. If this last qdisc doesn't have classes, as in the figure is shown, the packet is delivered to be sent to the physical network. Observe also something interesting: two (or more) filters can point out to the same class. Let's see now another of these diagrams and its explanation taken from the Werner [6] document:

Figure 2.1.8 shows an example of such a stack: first, there is a queuing discipline with two delay priorities. Packets which are selected by the filter go to the high-priority class, while all other packets go to the low-priority class. Whenever there are packets in the high-priority queue, they are sent before packets in the low-priority queue (e.g. the `sch_prio` queuing discipline works this way). In order to prevent high-priority traffic from starving low-priority traffic, we use the token bucket filter (TBF) queuing discipline, which enforces a rate of at most 1 Mbps. Finally, the queuing of low-priority packets is done by a FIFO queuing discipline. Note that there are better ways to accomplish what we've done here, e.g. by using class-based queuing (CBQ).

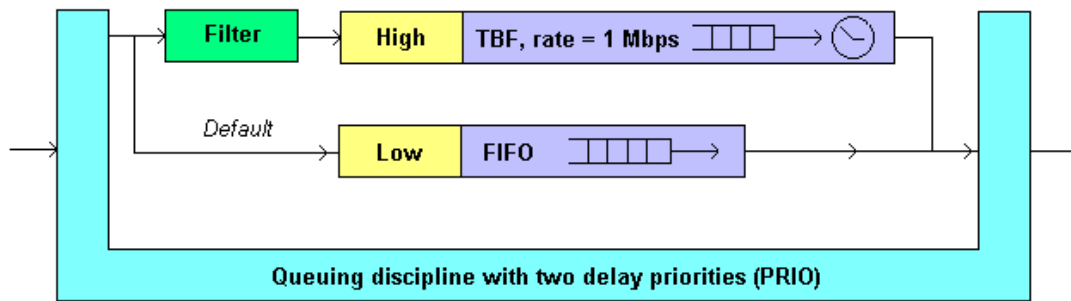


Figure 2.1.8

Great!! Werner talked about four Linux queuing disciplines (FIFO, PRIO, TBF and CBQ) and clears a lot our understanding about this stuff. The diagram implements three of these qdiscs. Now we know we can implement a default class where packets are sent when they do not match any of our filters. Seems like something well-known for us; something like "leave a default codepoint and assign it to the best-effort PHB". Also Werner's figure can be associated with those taken from RFC 2475 specification; see figure 1.4.1 somewhere above.

Let's see now with a little detail how the diagram represents our queuing discipline. The main qdisc is the PRIO queue which receives the packets. It applies the filter and select those of them marked as "high priority". How the mark is implemented? We don't know where the packets were marked but we know that marking could be based on MF (multi-field selection) or DS-codepoint.

Either being the case the classifier put them in the "High" class above. Rest of the packets (not marked with our high priority identifier) go to the "Low" class below. In the "High" class a TBF qdisc is implemented. As Werner explained before: *Typically, each class "owns" one queue...* This time the queue assigned to the "High" class is a TBF queue. What this queue is trying to do? Just controlling the maximum throughput traversing through it to 1 Mbps (have a look to the diagram). To the right of the TBF queue representation there's a little queue and a little clock shown. They represent the queue behavior and that some sort of metering is been made on it to know how much is the flow flowing.

The "Low" class queue is a FIFO queue. We saw something about it somewhere above. A simple FirstIn-FirstOut queue. It doesn't try to exercise any control just to enqueue and dequeue packets as they are arriving. Finally qdiscs for both classes deliver the packets on the right side of the main queue to be dispatched to the physical network.

A very interesting observation to be done here is that the "High" priority class is controlled by an upper level of throughput (implemented by TBF). We talked something about this before. High priority class traffic has to be controlled to avoid "starving" of lower priority classes. Of course, with this PRIO qdisc implementation this will work just if the transport protocol is responsive, like TCP. Why?

To step ahead let's now read from Saravanan [8] document:

This section discusses queuing disciplines, which form a basic building block for the support of QoS in linux. It also discusses the various queuing disciplines that are supported in linux. Each network device has a queue associated with it. There are 11 types of queuing disciplines that are currently supported in linux, which includes:

- *Class Based Queue (CBQ)*
- *Token Bucket Flow (TBF)*
- *Clark-Shenker-Zhang (CSZ)*
- *First In First Out (FIFO)*
- *Priority Traffic Equalizer (TEQL)*
- *Stochastic Fair Queuing (SFQ)*
- *Asynchronous Transfer Mode (ATM)*
- *Random Early Detection (RED)*
- *Generalized RED (GRED)*

- *Diff-Serv Marker (DS_MARK)*

Queues are identified by a handle <major number:minor number>, where the minor number is zero for queues. Handles are used to associate classes to queuing disciplines. Classes are discussed in the next subsection.

Gosh!! A lot of queues. Actually there are some more just developed but not officially implemented in the Linux kernel. Like HTB (Hierarchical Token Bucket) written by Martin Devera; WRR (Weighted Round Robin) written by Christian Worm Mortensen and IMQ (Intermediate Queuing Discipline) written by Patrick McHardy. Additionally some other exist not implemented yet on Linux, like CBT (Class-Based Threshold); FRED (Flow-Based RED); DRR (Differential Round Robin) and D-CBQ (Decoupled-CBQ). Cisco also has some polished queues like WFQ, D-WFQ, WRED, CBWFQ, CQ, PQ and stop you from count; there are a lot.

But don't be afraid. For our work we will concentrate on these: FIFO, PRIO, TBF, SFQ, RED, GRED, CBQ and DSMARK. (*Note: really, after this work was started I decided to use HTB instead of CBQ to implement DS; then CBQ will be replaced with HTB along this document*) It's going to be a very nice trip. At least I will do my best effort. By the way, for dealing with all these beasts we will need some mechanism to talk with them, this means, a tool which permit us to be the conductor of this orchestra. This tool was developed some years ago by Alexey Kuznetsov and it is called tc (traffic control). tc is not as friendly as most of us would like it should be, but, it's what we have. Recently, Werner Almesberger, really impressed for the lovely care that everybody feel for tc, wrote tcng (traffic control new generation). People still screaming. I'm just joking, of course.

This time our strategic will be very simple. First we will present a brief approach to the behavior of each queue; some theoretical but no so deeper support to understand how the queue was conceived. Next, how the queue is configurated using tc, and within this battle, an explanation about parameters and recommended settings.

For this part of our study we will have the little help of our friends: Bert Hubert and Lartc people [7]; Bert Hubert and his excellent work writing a manpage for these stuffs; Werner Almesberger, Jamal Hadi Salim and Alexey Kuznetsov [10] mainly for understanding GRED and DSMARK; some shallow diving into Alexey Kuznetsov code and documentation to understand how tc works; and last but not least: for illustrating as better as possible because Linux people are not distinguished for being good pedagoge teachers, I'm going to use some figures and concepts taken from "Supporting Differentiated Service Classes: Queue Scheduling Disciplines" by Chuck Semeria of Juniper Networks [11]. Hoping to not having rights problem with these people (in case of having please make a collect to help me on my defense). I'm going to dare to use their excellent documentation to support our explanation. Without losing more time let's start with the simplest one, the FIFO queuing discipline.



[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

2.2.- FIFO queuing discipline

FIFO is absolutely explained by Chuck Sumeria [11] in a few sentences and a figure; let's see how:

Firstin, firstout (FIFO) queuing is the most basic queue scheduling discipline. In FIFO queuing, all packets are treated equally by placing them into a single queue, and then servicing them in the same order that they were placed into the queue. FIFO queuing is also referred to as Firstcome, firstserved (FCFS) queuing. (See Figure 2.2.1)

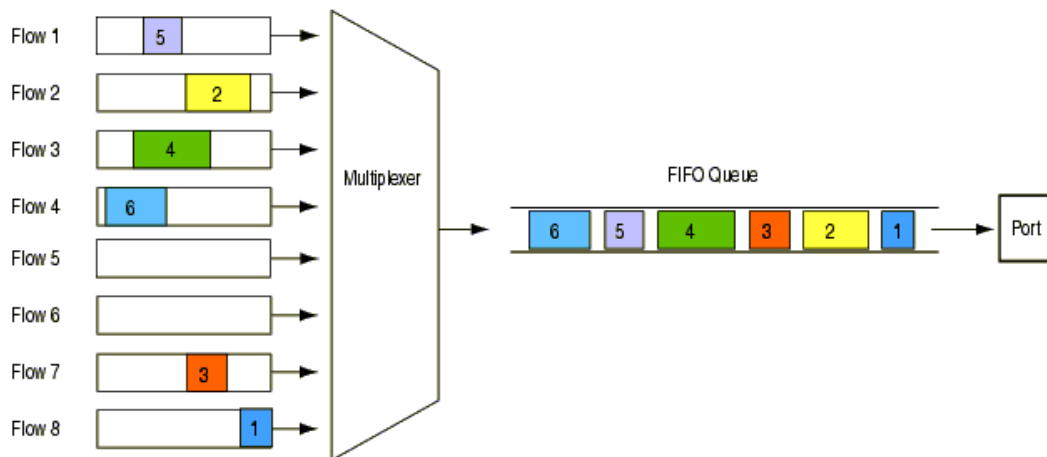


Figure 2.2.1

Easy. Isn't it? By the way, FIFO is the default queuing discipline used by Linux and most of the routers around the world. In case you don't specify any specific qdisc Linux assembles its interfaces with this type of queue. Let's see now how we can setup a FIFO queue on ethernet interface eth0 using tc; at the Linux command prompt just write:

```
# tc qdisc add dev eth0 root pfifo limit 10
```

cm-01

The command is self-explanatory. 'tc' is the utility; 'qdisc' for telling tc we are configuring a queue discipline (it could be 'class' or 'filter' if we are configuring a class or a filter respectively); 'add' because we are adding a new qdisc; 'dev eth0' for adding the qdisc to the device or interface Ethernet eth0; 'root' because it is a root qdisc (it doesn't apply to pfifo that is classless - only a root qdisc exists - but required to normalize the command utilization); 'pfifo' because our queue is a pfifo (packet-fifo) queue. Finally pfifo requires only one parameter: just 'limit' to indicate the length of the queue (number of packets that the queue can hold). Our pfifo queue is a 10 packets queue.

After creating our queueing discipline we can ask tc what we have configured; the following command:

[Network/Ethernet Tester](#)

Tester for Switch & SOHO Router Network TAP / Power Monitor
www.xtramus.com

[Free Erlang Calculator](#)

Model your call center staffing needs. What-if analysis, graphs
www.kooltoolz.com

[QoS Management Tools](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[Q-net Queueing Systems](#)

Queue Management System and Info Kiosk in high-tech design.
www.q-net.com

[Simulation Modeling Tools](#)

Learn the benefit of ExtendSim simulation tools. Free Demo!
www.extendssim.com

```
# tc qdisc show dev eth0
```

cm-02

will respond with:

```
qdisc pfifo 8001: dev eth0 limit 10 p
```

cm-03

The command just says: show us what qdisc we have on device eth0. tc responds by telling us that we have a pfifo qdisc numbered as 8001: (this really means 8001:0) with a limit capacity of 10 packets. Qdiscs and components are numbered, or better yet identified by a 32 bits handler formed by a 16 bits major number and a 16 bits minor number. The minor number is always zero for queues. When we added our pfifo queue we didn't assign it any handler and then tc created one for it (8001:0).

FIFO has spread as the grass but it has benefits and limitations. Chuck [11] exposes about this in a magisterial approach:

FIFO queuing offers the following benefits:

- *For softwarebased routers, FIFO queuing places an extremely low computational load on the system when compared with more elaborate queue scheduling disciplines.*
- *The behavior of a FIFO queue is very predictable---packets are not reordered and the maximum delay is determined by the maximum depth of the queue.*
- *As long as the queue depth remains short, FIFO queuing provides simple contention resolution for network resources without adding significantly to the queuing delay experienced at each hop.*

FIFO queuing also poses the following limitations:

- *A single FIFO queue does not allow routers to organize buffered packets, and then service one class of traffic differently from other classes of traffic.*
- *A single FIFO queue impacts all flows equally, because the mean queuing delay for all flows increases as congestion increases. As a result, FIFO queuing can result in increased delay, jitter, and loss for realtime applications traversing a FIFO queue.*
- *During periods of congestion, FIFO queuing benefits UDP flows over TCP flows. When experiencing packet loss due to congestion, TCPbased applications reduce their transmission rate, but UDPbased applications remain oblivious to packet loss and continue transmitting packets at their usual rate. Because TCPbased applications slow their transmission rate to adapt to changing network conditions, FIFO queuing can result in increased delay, jitter, and a reduction in the amount of output bandwidth consumed by TCP applications traversing the queue.*
- *A bursty flow can consume the entire buffer space of a FIFO queue, and that causes all other flows to be denied service until after the burst is serviced. This can result in increased delay, jitter, and loss for the other wellbehaved TCP and UDP flows traversing the queue.*

More clearly, impossible. Well, with all these problems let's delete the pfifo queue from our device eth0:

```
# tc qdisc del dev eth0 root
```

cm-04

[VoIP Call Quality Monitor](#)

Real-time & History QoS reporting Pre-empt VoIP downtime. Free Trial!
www.vqmanager.com

[ActiveMQ: Open Source JMS](#)

Enterprise-class JMS messaging Free downloads, papers, docs, forum
open.iona.com/

[ViewMaster -New & Vintage](#)

World marketplace for reels, packets, viewers, & supplies
www.3dstereo.com

[Java Server](#)

65 µs latency, 950 Mb/s/second. Download a demo.
www.rti.com

[Multithread Components](#)

We offer efficient components for atomic multithreaded programming!
www.pss-ab.com

That brought to the english just means: delete from device eth0 the root queue discipline.
Okay. We are over with FIFO. Now let's try with PRIO.



[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

2.3.- PRIO queuing discipline

For explaining PRIO queuing discipline Chuck [11] treatment of the issue is again impeccable; just some sentences and a figure:

Priority queuing (PQ) is the basis for a class of queue scheduling algorithms that are designed to provide a relatively simple method of supporting differentiated service classes. In classic PQ, packets are first classified by the system and then placed into different priority queues. Packets are scheduled from the head of a given queue only if all queues of higher priority are empty. Within each of the priority queues, packets are scheduled in FIFO order. (See Figure 2.3.1).

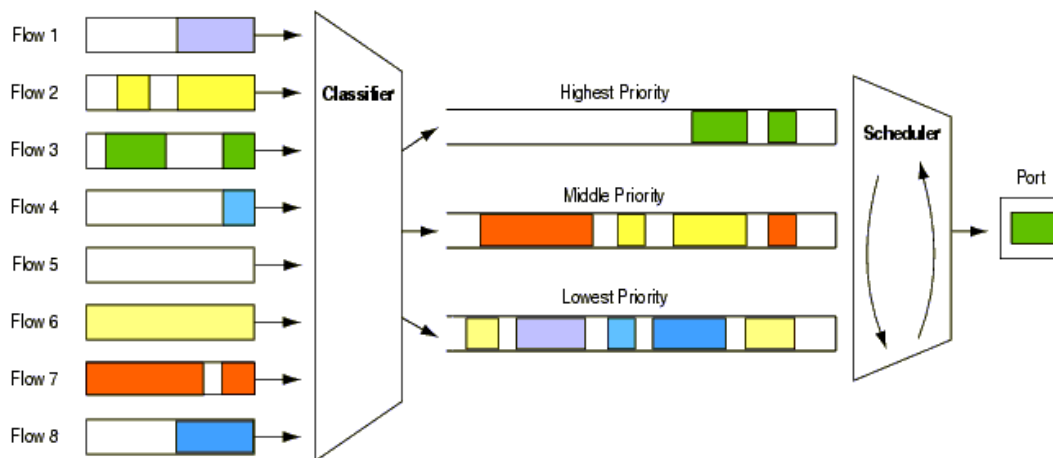


Figure 2.3.1

On Linux things are a little bit complicated. Reading from the man page and Lartc people [7] we can build an explanation. The PRIO qdisc is a classful queuing discipline that contains an arbitrary number of classes of different priority. When a packet is enqueued a sub-qdisc is chosen based on a filter command that you give with tc. When you create a new PRIO queue three pfifo sub-queuing disciplines are created. In fact, automatically 3 classes named m:1, m:2 and m:3 are created where m is the major number of the queuing discipline. Each of these classes is assembled with a pfifo as its own qdisc. A diagram, please, to understand better this stuff!!

[EXCELIANCE Load Balancer](#)

Appli. Load balancing & High avail. download free evaluation soft.
www.Software-Appliance.com

[Network/Ethernet Tester](#)

Tester for Switch & SOHO Router Network TAP / Power Monitor
www.xtramus.com

[End to End QoS and more](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[Transas Sets the Standard](#)

Onboard Systems, Simulators and VTS IT Solutions At Sea and Ashore
www.transas.com

[CESoPSN & SAToP Solutions](#)

E1/T1/J1 Over IP, MPLS, EPON Networks For Next Generation Networks
www.techaya.com

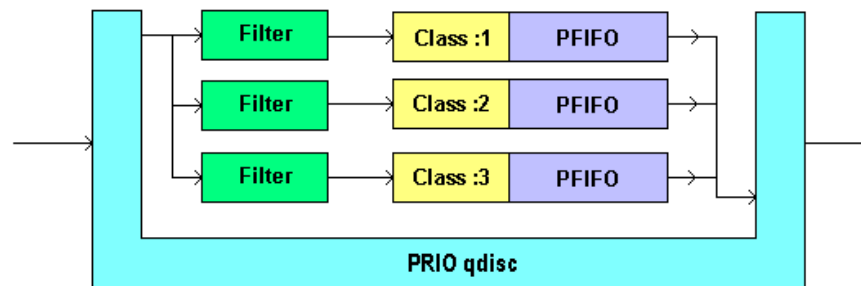


Figure 2.3.2

Have a look to figure 2.3.2. Whenever a packet needs to be dequeued class :1 is tried first. When it is empty class :2 is tried and finally class :3. As is almost obvious this queue could be really a problem such as it is. Suppose you are implementing a DS-AF architecture and you decide to associate DS classes AF11, AF21 and AF31 with the 3 classes offered by the PRIO qdisc, respectively. Perhaps you were lucky and things work at all. But, left your mind flying: what happen if a scenario appears where departure rate is less than arrival rate due to some congestion problem in the output link? And at the same time AF11 flows' arrival rate is higher than the departure rate. In this case always you will have AF11 packets waiting in the class :1 queue and classes :2 and :3 will not be served.

AF21 and AF31 customers are going to call you really very angry with you: hey, guy, what's going on there? This problem can be fixed changing the class :1 pfifo qdisc for some type of qdisc that put an upper level to AF11 flows. For example a TBF qdisc. Have a look to a Werner's diagram somewhere above for an example of this issue.

As you probably deduced this qdisc, when configuring with care, can be very useful in case you want to prioritize certain kind of traffic in favor of other. Let's see now how to manage this pet with tc. But first a fast trip over the parameters. Reading from Lartc [7] we have:

The following parameters are recognized by tc:

bands

Number of bands to create. Each band is in fact a class. If you change this number, you should probably also change the priomap.

priomap

"If you do not provide tc filters to classify traffic", the PRIO qdisc looks at the TC_PRIO priority to decide how to enqueue traffic. The kernel assigns each packet a TC_PRIO priority, based on TOS flags or socket options passed by the application.

The TC_PRIO is decided based on the TOS, and mapped as follows:

<i>TC_PRIO..</i>	<i>Num</i>	<i>TOS</i>	<i>Band</i>
<i>BESTEFFORT</i>	0	<i>Maximize Reliability (0x04)</i>	1
<i>FILLER</i>	1	<i>Minimize Cost (0x02)</i>	2
<i>BULK</i>	2	<i>Maximize Throughput (0x08)</i>	2
<i>INTERACTIVE_BULK</i>	4		2
<i>INTERACTIVE</i>	6	<i>Minimize Delay (0x10)</i>	1
<i>CONTROL</i>	7		2

The bands are classes, and are called major:1 to major:3 by default, so if your PRIO qdisc is called 12:, tc filter traffic to 12:1 (band 0) to grant it more priority. Reiterating, band 0 goes to minor number 1, band 1 to minor number 2, etc.

Well, nice but not too much, for our goal at least. 'priomap' parameter tells us that for using PRIO queuing discipline we have to provide filters because default behavior is not for us. Why? Protecting TOS (Why?) we will show PRIO qdisc setup using filters:

```
# tc qdisc add dev eth0 root handle 1: prio
```

cm-05

We know this command already. However, this time we are numbering our PRIO qdisc ourselves as 1:0 (the zero can be avoided when working with tc). Because PRIO is some sort of automatic queue this command instantly create classes 1:1, 1:2 and 1:3 and each of them has its pfifo queue already installed.

Let's see how to implement the filters. We will insist in our original idea, this means, to assign AF classes AF11, AF21 and AF31 to prio classes 1:1, 1:2 and 1:3. Do you remember our tip rule to compose codepoints? Using it we have:

$$AF11 = 1-2 = 001010$$
$$AF21 = 2-2 = 010010$$
$$AF31 = 3-2 = 011010$$

These are the six leftmost bits (DS-codepoint); the complete TOS byte will have two zero-bit to the right. Finally we have:

$$AF11 = 00101000 = 0x28$$
$$AF21 = 01001000 = 0x48$$
$$AF31 = 01101000 = 0x68$$

Last hexadecimal number is what we will use to build our filters. First, the first of them:

```
# tc filter add dev eth0 parent 1:0 prio 1 protocol ip u32 \
    match ip tos 0x28 0xff flowid 1:1
```

cm-06

Oh, my god. Let's try to decipher this. 'tc filter add dev eth0' is less than obvious: "hey, tc, add a filter on device eth0, thanks!". 'parent 1:0' means "the parent of the filter is going to be the object identify with the number 1:0 that happens to be our PRIO qdisc". 'prio 1' means "try this filter with priority 1"; if happens to be some other filter with prio 2, 3, 4, etc., try them in this order. 'protocol ip' means "we are working with the ip protocol". 'u32' means "our filter is just an u32 filter"; is just a type. There are other types as fwmark, route, etc. but we are not interested on them now. 'match ip' means "what follows has to be matched against the ip header of the packet". 'tos 0x28 0xff' means "match exactly tos byte 0x28". tc always multiplies the first term with the second one to get the number to be matched: $0x28 * 0xff$ is just 0x28. Using 0xf0, for example, we can change the matching number. Finally 'flowid 1:1' means "flows matching this filter have to be sent, directly, to the class identify with number 1:1"; this happens to be our PRIO 1:1 class.

Easy. Isn't it? Now let's write the next two commands:

```
# tc filter add dev eth0 parent 1:0 prio 2 protocol ip u32 \
    match ip tos 0x48 0xff flowid 1:2
# tc filter add dev eth0 parent 1:0 prio 3 protocol ip u32 \
    match ip tos 0x58 0xff flowid 1:3
```

cm-07

Eureka!! After this we have assembled our prio qdisc represented in the figure above. Congratulations.

But, as we saw previously when studying FIFO, PRIO has its benefits and problems too (who doesn't have?). Again, Chuck [11] approach is clear and very complete:

PQ offers a couple of benefits:

- *For softwarebased routers, PQ places a relatively low computational load on the system when compared with more elaborate queuing disciplines.*
- *PQ allows routers to organize buffered packets, and then service one class of traffic differently from other classes of traffic. For example, you can set priorities so that realtime applications, such as interactive voice and video, get priority over applications that do not operate in real time.*

But PQ also results in several limitations:

- *If the amount of highpriority traffic is not policed or conditioned at the edges of the network, lowerpriority traffic may experience excessive delay as it waits for unbounded higherpriority traffic to be serviced.*
- *If the volume of higherpriority traffic becomes excessive, lowerpriority traffic can be dropped as the buffer space allocated to lowpriority queues starts to overflow. If this occurs, it is possible that the combination of packet dropping, increased latency, and packet retransmission by host systems can ultimately lead to complete resource starvation for lowerpriority traffic. Strict PQ can create a network environment where a reduction in the quality of service delivered to the highestpriority service is delayed until the entire network is devoted to processing only the highestpriority service class.*
- *A misbehaving highpriority flow can add significantly to the amount of delay and jitter experienced by other highpriority flows sharing the same queue.*
- *PQ is not a solution to overcome the limitation of FIFO queuing where UDP flows are favored over TCP flows during periods of congestion. If you attempt to use PQ to place TCP flows into a higherpriority queue than UDP flows, TCP window management and flow control mechanisms will attempt to consume all of the available bandwidth on the output port, thus starving your lowerpriority UDP flows.*

It is very important to know the advantages and disadvantages of each qdisc to have better decisions when applying them. There is a nice guy, Stef Coene at www.docum.org, that has carried this bag in Linux since a long time. Have a look to his site. There is a lot of useful information on there. To continue our next queuing discipline will be TBF.

[Ne.....?](#)

[Analyzer](#)

Advanced Network Analyzer Solution. Start Analyzing Today. Download!
www.Paessler.com/Packet-Sniffing

[DDoS Protection Leader](#)

Renowned anti-DDoS service. Full SLA provided. Global network.
www.prolexic.com

[Paper: Mobile Backhaul](#)

Expanding 3G backhaul capacity without jeopardizing 2G revenues
www.ciena.com

[AirMagnet Free-Trial](#)

Test AirMagnet's Laptop Tool and Police/Audit/Fix your WLAN!
www.airmagnet.com

[Aimsun and Aimsun Online](#)

Powerful simulators for transport planning and traffic management.
www.aimsun.com



[Previous](#)[Content](#)[Next](#)

[Previous](#)[Content](#)[Next](#)

2.4.- TBF queuing discipline



There is an old story in my country that tells more or less as this: there was once a mule breeder for selling. One day a farmer looking for purchase some mules for his farm, told the peasant. How many mules do you have? I don't know, the breeder answered. The farmer asked then: What's the price of each of them? Just one buck for each one, the peasant answered. Okay, the farmer told: here you have 100 bucks; let me take now 100 of them. No, no, the peasant very worry told. I don't like the business that way. Just put one buck on my hat and for each buck you put on it, I will take out a mule for you from the corral. It's very simple: buck that drops, mule that crosses.

TBF queuing discipline works very similar to this funny story. You are interested in controlling the maximum rate that packets are dispatched from a queue of them; then you create a buffer (bucket) constantly filled by some virtual pieces of information called tokens, at a specific rate (token-rate).

As soon as there are tokens in the bucket and packets in the queue you pick up a token and let a packet go. Making the simil, the bucket is the peasant's hat, token are the bucks, the queue is the corral and packets are the peasant's mules. As fast as the token bucket is filled the packets are dequeued from the queue. The constant rate at which you fill the bucket of token is the maximum rate you are interested packets leave the queue.

Of course relationship cannot be one-to-one; this means, one-token one-packet, because packets are most of the time of different size. But you can create a one-token one-byte relationship. Let's say, for example, that you want to have a maximum rate of saying, 500 kbps. For doing this you have to dispatch 62.5 KB/sec, this means, 64000 bytes per second. Okay, we are going to fill our bucket to a rate of 64000 tokens per second. Let's suppose now that we check the head of the packet queue and we have a 670 bytes packet (the length is in the header packet). Then we pick up 670 tokens from the bucket and let the packet go. We will try to keep our packet queue empty; as soon as we have a packet and enough tokens to let it go, we will do that.

Let's see now a brief summary of how people from Lartc [7] explain this in a more technical way:

The Token Bucket Filter (TBF) is a simple queue that only passes packets arriving at a rate which is not exceeding some administratively set rate, with the possibility to allow short bursts in excess of his rate.

[QoS Management Tools](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
[Packeteer.com](#)

[Network Packet Analyzer](#)

Advanced Network Analyzer Solution. Start Analyzing Today. Download!
[www.Paessler.com/Packet-Sniffi](#)

[TDM Over IP Solutions](#)

E1/T1/J1 Over IP,MPLS, PON Networks For Wi-MAX,Radio and Cell. Backhaul
[www.techava.com](#)

[AirMagnet Free-Trial](#)

Test/Audit/Fix your WLAN with Industry-leading Wi-Fi analyzer
[www.airmagnet.com](#)

[SIP call troubleshooter](#)

Real-time VoIP Traffic monitoring Free Trial & Support. Download now!
[www.vqmanager.com](#)

"TBF is very precise, network- and processor friendly. It should be your first choice if you simple want to slow an interface down!"

The TBF implementation consists of a buffer (bucket), constantly filled by some virtual pieces of information called tokens, at a specific rate (token rate). The most important parameter of the bucket is its size, that is the number of tokens it can store.

Each arriving token collects one incoming data packet from the data queue and is then deleted from the bucket. Associating this algorithm with the two flows -- token and data, gives us three possible scenarios:

- *The data arrives in TBF at a rate that is equal to the rate of incoming tokens. In this case each incoming packet has its matching token and passes the queue without delay.*
- *The data arrives in TBF at a rate that is smaller than the token rate. Only a part of the tokens are deleted at output of each data packet that is sent out the queue, so the tokens accumulate, up to the bucket size. The unused tokens can then be used to send data at a speed that is exceeding the standard token rate, in case short data bursts occur.*
- *The data arrives in TBF at a rate bigger than the token rate. This means that the bucket will soon be devoid of tokens, which causes the TBF to throttle itself for a while. This is called an "overlimit situation". If packets keep coming in, packets will start to get dropped.*

The last scenario is very important, because it allows to administratively shape the bandwidth available to data that is passing the filter.

The accumulation of tokens allows a short burst of overlimit data to be still passed without loss, but any lasting overload will cause packets to be constantly delayed, and then dropped. Please note that in the actual implementation, tokens correspond to bytes, not packets.

Reading above observe that the second scenario (when some token are saved) permits us for allowing some short data burst until tokens are again totally consumed.

Next a parameter description is required before having a look of how to use tc for configuring this queue; reading from Lartc [7] and TBF man page we have:

Even though you will probably not need to change them, tbh has some knobs available. First the parameters that are always available:

limit or latency "(size -in bytes- of the packets queue)"

Limit is the number of bytes that can be queued waiting for tokens to become available. You can also specify this the other way around by setting the latency parameter, which specifies the maximum amount of time a packet can sit in the TBF. The latter calculation takes into account the size of the bucket, the rate and possibly the peakrate (if set).

burst/buffer/maxburst "(size -in bytes- of the token queue)"

Size of the bucket, in bytes. This is the maximum amount of bytes that tokens can be available for instantaneously. In general, larger shaping rates require a larger buffer. For 10mbit/s on Intel, you need at least 10kbyte buffer if you want to reach your configured rate! If your buffer is too small, packets may be dropped because more tokens arrive per timer tick than fit in your bucket.

mpu

A zero-sized packet does not use zero bandwidth. For ethernet, no packet uses less than 64 bytes. The Minimum Packet Unit determines the minimal token usage for a packet.

rate

The speedknob. See remarks above about limits!

If the bucket contains tokens and is allowed to empty, by default it does so at infinite speed. If this is unacceptable, use the following parameters:

peakrate

If tokens are available, and packets arrive, they are sent out immediately by default, at "lightspeed" so to speak. That may not be what you want, especially if you have a large bucket. The peakrate can be used to specify how quickly the bucket is allowed to be depleted. If doing everything by the book, this is achieved by releasing a packet, and then wait just long enough, and release the next. We calculated our waits so we send just at peakrate. However, due to default 10ms timer resolution of Unix, with 10.000 bits average packets, we are limited to 1mbit/s of peakrate!

mtu/minburst

The 1mbit/s peakrate is not very useful if your regular rate is more than that. A higher peakrate is possible by sending out more packets per timertick, which effectively means that we create a second bucket! This second bucket defaults to a single packet, which is not a bucket at all. To calculate the maximum possible peakrate, multiply the configured mtu by 100 (or more correctly, HZ, which is 100 on intel, 1024 on Alpha).

Let's bring now a simple configuration taken from the TBF man page:

```
# tc qdisc add dev eth0 root tbf rate 0.5mbit burst 5kb \
    latency 70ms peakrate 1mbit minburst 1540
```

cm-08

A little bit more complicated than configuring a FIFO or PRIO qdisc. Isn't it? But not too much. First part of the command (including 'tbft') is well known already. The maximum sustained rate is selected as 0.5 mbps with a 5 kilobyte buffer and a peak rate of 1.0 mbps for short burst of packets. The bucket queue size is calculated so that a maximum of 70ms of latency a packet will suffer on the queue. The minburst parameter is selected as the MTU (maximum transmitted unit) of the interface.

Finally to close this section: do you remember the PRIO qdisc configuration we did somewhere above? Let's replace the class 1:1 FIFO qdisc with a TBF qdisc to protect class 1:2 and class 1:3 flows from being starved by class 1:1 flows. The new diagram will be:

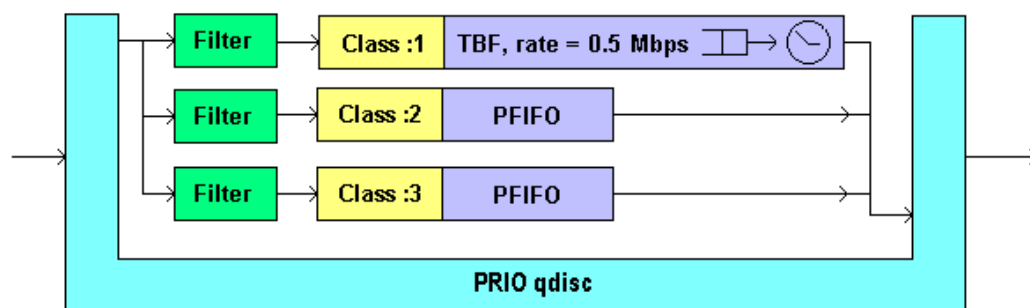


Figure 2.4.1

And the tc commands to build this scheme will be:

```
# tc qdisc add dev eth0 root handle 1: prio
# tc filter add dev eth0 parent 1:0 prio 1 protocol ip u32 \
    match ip tos 0x28 0xff flowid 1:1
# tc filter add dev eth0 parent 1:0 prio 2 protocol ip u32 \
    match ip tos 0x48 0xff flowid 1:2
# tc filter add dev eth0 parent 1:0 prio 3 protocol ip u32 \
    match ip tos 0x58 0xff flowid 1:3
# tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 0.5mbit burst 5kb \
    latency 70ms peakrate 1mbit minburst 1540
```

cm-09

Observe that last command is very similar to the TBF configuration example but this time the word 'root' is omitted and replaced by the words 'parent 1:1 handle 10:'. Now our TBF qdisc is not a root qdisc, but instead, it is a child of class 1:1 (class 1:1 own queue) and we number it as 10:0. This is a simple example of how, using tc, we assign a qdisc to a class to create a hierarchy.

Well, we are ready with TBF queuing discipline. To continue we will study now the SFQ queuing discipline.

[DDoS Protection Leader](#)

Renowned anti-DDoS service. Full SLA provided. Global network.
www.prolexic.com

[Java Server](#)

65 µs latency, 950 Mb/s/second. Download a demo.
www.rti.com

[Deep Packet Inspection](#)

A User-driven Community for DPI/DPP. Learn, discuss, and collaborate
www.dPacket.org

[ActiveMQ: Open Source MQ](#)

Enterprise-class JMS messaging. Free downloads, papers, guides
open.ionas.com/

[Paper: 3G Mobile Backhaul](#)

Using Ethernet to grow profitable 3G services while reducing OPEX
www.ciena.com



This site is visited regularly for very specialized people; it's then the best place to advertise your very specialized product.
First month at no cost for you to verify this aseveration.



Ads required

Your company can have presence here for as low as US \$0.001 × square pixel × page × month

[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

2.5.- SFQ queuing discipline

Stochastic Fairness Queuing (SFQ) belongs to the family of queuing disciplines based on the fair queuing algorithm. This algorithm was proposed by John Nagle in 1987. Let's build our explanation beginning with Chuck [11] approach to this issue including a very nice figure to clear what we are studying:

Fair queuing (FQ) was proposed by John Nagle in 1987. FQ is the foundation for a class of queue scheduling disciplines that are designed to ensure that each flow has fair access to network resources and to prevent a bursty flow from consuming more than its fair share of output port bandwidth. In FQ, packets are first classified into flows by the system and then assigned to a queue that is specifically dedicated to that flow. Queues are then serviced one packet at a time in roundrobin order. Empty queues are skipped. FQ is also referred to as perflow or flowbased queuing. (See Figure 2.5.1)

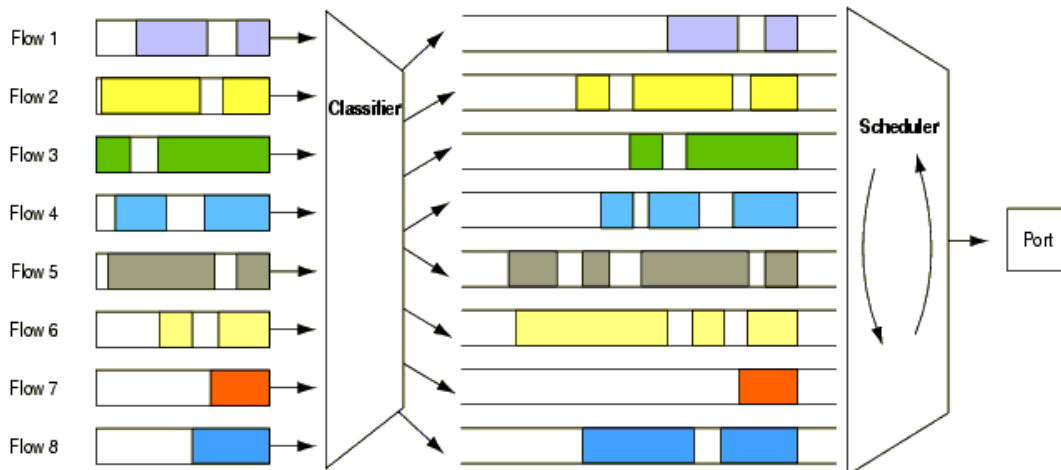


Figure 2.5.1

FQ is like having several doors. When a packet arrives it is classified by the classifier and assigned to one of the doors. The door is the entry to a queue that is served together with some other, one packet at a time in a round-robin order. This way the service is 'fair' for every queue.

The key to classify a flow is a conversation, this means, a numeric representation based on the tuple [source address, source port, destination address]. More sophisticated implementations can use the tuple [source address, source port, destination address, protocol] for classification. Because it is not practical to have one queue for each conversation SFQ employs a hashing algorithm which divides the traffic over a limited number of queues. Let's see a summary of how people from Lartc [7] define this discipline:

Stochastic Fairness Queueing (SFQ) is a simple implementation of the fair queueing algorithms family. It is less accurate than others, but it also requires less calculations while being almost perfectly fair.

The key word in SFQ is conversation (or flow), which mostly corresponds to a TCP session or a UDP stream. Traffic is divided into a pretty large number of FIFO queues, one for each conversation. Traffic is then sent in a round robin fashion, "giving each session the chance to send data in turn".

[RS232 Packet Sniffer](#)

www.hhdsoftware.com

[ActiveMQ: Open Source JMS](#)

Enterprise-class JMS messaging Free downloads, papers, docs, forum
open.iona.com/

[QoS Management Tools](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[Network Packet Analyzer](#)

Advanced Packet Analysis. A "must have" networking tool.
www.Paessler.com

[Deep Packet Inspection](#)

A User-driven Community for DPI/DPP Learn, discuss, and collaborate
www.dPacket.org

"This leads to very fair behaviour and disallows any single conversation from drowning out the rest". SFQ is called "Stochastic" because it does not really allocate a queue for each session, it has an algorithm which divides traffic over a limited number of queues using a hashing algorithm.

Because of the hash, multiple sessions might end up in the same bucket, which would halve each session's chance of sending a packet, thus halving the effective speed available. To prevent this situation from becoming noticeable, SFQ changes its hashing algorithm quite often so that any two colliding sessions will only do so for a small number of seconds.

It is important to note that "SFQ is only useful in case your actual outgoing interface is really full!" If it is not then there will be no queue on your linux machine and hence no effect. Later on we will describe how to combine SFQ with other qdiscs to get a best-of-both worlds situation.

Some concepts have to be stood out here: first, SFQ allocates a pretty large number of FIFO queues; as was said it is not practical to have one queue for each connection or conversation. Nevertheless, increasing as large as possible the number of queues helps the fairness of the algorithm. Also, each queue is a FIFO queue. Don't forget this; we will touch this theme again when talking about RED queuing discipline later. Because number of queues are normally less than number of flows, a hashing mechanism is implemented to assign flows based on their tuple representation, to queues. Assignment is stochastic and a disturb mechanism has to be implemented to reconfigure hashing trying to improve fairness. Parameters are simple, let's see from Lartc [7]:

The SFQ is pretty much selftuning:

perturb

Reconfigure hashing once this many seconds. If unset, hash will never be reconfigured. Not recommended. "10 seconds is a good value."

quantum

Amount of bytes a stream is allowed to dequeue before the next queue gets a turn. Defaults to 1 maximum sized packet (MTU-sized). "Do not set below the MTU!"

And configuration is a dream:

```
# tc qdisc add dev eth0 root sfq perturb 10
```

cm-10

You can have some statistics using the 'tc show' command; again from Lartc [7]:

```
# tc -s -d qdisc show # -s = statistics; -d = details
```

```
qdisc sfq 800c: dev eth0 quantum 1514b limit 128p flows 128/1024 perturb 10sec
Sent 4812 bytes 62 pkts (dropped 0, overlimits 0)
```

- The number 800c: is the automatically assigned handle number.
- Limit means that 128 packets can wait in this queue.
- There are 1024 hashbuckets available for accounting, of which 128 can be active at a time (no more packets fit in the queue!).
- Once every 10 seconds, the hashes are reconfigured.

Okay. SFQ is a very friendly queuing discipline. Because it deals with so many queues to manage as many flows as possible it is well suitable to put some order on them. Have you thought the same as we have? Of course, when we have a 'default PHB' many flows escape this way, therefore, this is a job made-to-order of our hero, the SFQ discipline. Let's reconfigure our PRIO example for having AF11 flows on class 1:1, AF21 flows on class 1:2 and rest of the flows (a default PHB) on class 1:3. We will put a TBF qdisc to control throughput on classes 1:1 and 1:2 and a SFQ qdisc on class 1:3. First, our well-known diagram:

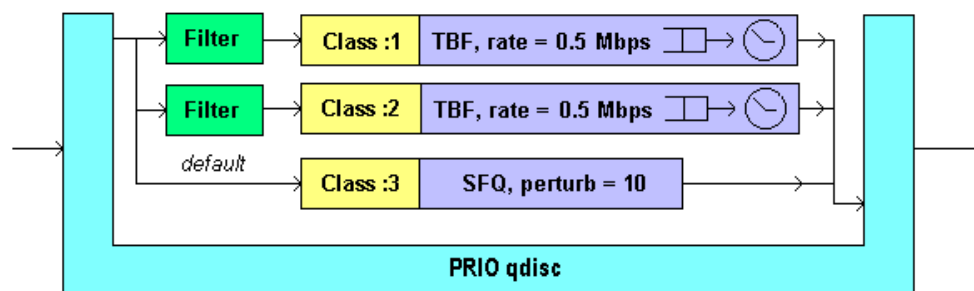


Figure 2.5.2

and then, the configuration commands:

```
# tc qdisc add dev eth0 root handle 1: prio
# tc filter add dev eth0 parent 1:0 prio 1 protocol ip u32 \
    match ip tos 0x28 0xff flowid 1:1
# tc filter add dev eth0 parent 1:0 prio 2 protocol ip u32 \
    match ip tos 0x48 0xff flowid 1:2
# tc filter add dev eth0 parent 1:0 prio 3 protocol ip u32 \
    match ip src 0/0 flowid 1:3
# tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 0.5mbit burst 5kb \
    latency 70ms peakrate 1mbit minburst 1540
# tc qdisc add dev eth0 parent 1:2 handle 20: tbf rate 0.5mbit burst 5kb \
    latency 70ms peakrate 1mbit minburst 1540
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq perturb 10
```

cm-11

The third filter command says: anything unmatched so far should go to class 1:3, the next-higher priority.

Well, we finished with SFQ. At this place of our trip RED queuing discipline time has come.

[Network Packet Analyzer](#)

Advanced Packet Analysis. A "must have" networking tool.
www.Paessler.com

[Use Genetic Algorithms](#)

Neural net software with GAs. Download free evaluation software.
www.neuralware.com

[FlagStone Encrypted HDD](#)

Full Disk Encryption, Tamper Proof FIPS, DIPCOG, CSE, CESG Approved
www.flagstonesecure.com

[Voice Compression Expert](#)

Dedicated to Voice Compression Up to 20:1 Compression Rate
www.broadgalaxy.com

[DNS DDoS Attacks](#)

Secure64 Survived the Attacks Learn how with a Free White Paper!
www.Secure64.com



This site is visited regularly for very specialized people; it's then the best place to advertise your very specialized product. First month at no cost for you to verify this aseveration.

[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

2.6.- RED queuing discipline

All queuing disciplines we have seen so far (FIFO, PRIO, TBF, SFQ) and also HTB that we will see later, base their queuing algorithm in the well-known FIFO queue. As you remember, in this queuing discipline packets enter the queue at the tail and leave it from the head. This, that could be considered something very simple and natural, pose a serious problem to those flows based on the, de-facto standard, TCP transport protocol.

Problems arise when queues overflow due to bursty of packets. At this moment more packets can not be admitted and they have to be dropped where they are entering, this means, at the tail of the queue. That's the reason why all queuing discipline based on the FIFO queue are known as DropTail queues.

Mark Anthony Parris in his excellent work "*Class-Based Thresholds-Lightweight Active Router-Queue Management for Multimedia Networking*" [12], approaches this theme in a magisterial way. Let's read from his work:

Firstinfirstout, droptailwhenfull was the original queue management scheme used in Internet routers. With this scheme packets are enqueued at the tail of a queue as they arrive and dequeued from the head of queue when there is capacity on the link. Droptail is the policy of dropping the arriving packet when the queue is full. (Other alternatives include dropping the packet at the head of the queue.) Droptail and FIFO are used interchangeably in this dissertation.

Braden, et al., point out several problems with simple droptailonfull and recommends that Internet routers employ more sophisticated techniques for managing their queues [Braden98]. The two major problems they identify are the problems of lockout and fullqueues.

Lockout refers to a phenomenon in which the shared resource, link bandwidth, is unfairly consumed exclusively by a small number of flows. The remaining flows are lockedout of (i.e., denied access to) the queue and, consequently, locked out of the outbound link. In this phenomenon the queue is occupied only by the packets from a small number of flows while the packets associated with most flows are consistently discarded. As a result, most flows receive none of the link bandwidth, and starve.

This phenomenon occurs because of timing effects which result in some flows' packets always arriving to find the queue full. For example, consider a situation where many sources are periodically sending bursts of packets that in aggregate exceeds the queue's capacity. If these sources become synchronized, all sending nearly simultaneously, the first packets to arrive (e.g. from the source closer to the bottleneck link) will find a queue with some available capacity while the subsequent packets will be discarded. If the same relative order is maintained between the sources, those sources that send first will consistently make progress while the other flows will consistently have all packets discarded and, thus, starve.

Fullqueues are queues that are usually occupied to capacity. If bursts of packets arrive to a full queue, many packets are dropped simultaneously. This can lead to large oscillations in the network utilization. If the dropped packets are from different flows there may be synchronized responses (backoff) among multiple flows. Synchronized backoff is a phenomenon in which many sources simultaneously receive congestion notification and reduce their generated load. As a result, the overall load on the network may drop below the capacity of the link and then rise back to exceed the link's capacity resulting

[Network/Ethernet Tester](#)

Tester for Switch & SOHO Router Network TAP / Power Monitor
www.xtramus.com

[Network Packet Analyzer](#)

Get complete real-time analysis across your entire network.
www.Paessler.com/Packet-Sniffing

[SIP call troubleshooter](#)

Real-time VoIP Traffic monitoring Free Trial & Support. Download now!
www.vqmanager.com

[Paper: Mobile Backhaul](#)

Expanding 3G backhaul capacity without jeopardizing 2G revenues
www.ciena.com

[MEF-8 Solutions](#)

E1/T1/J1 Over MPLS, EPON Networks For Next Generation Networks
www.techaya.com

in a full queue and once again leading to simultaneous drops. This oscillating behavior is exactly counter to the buffer's intended function, acting as a smoothing filter.

Many authors have studied these problems. In 1993, Sally Floyd and Van Jacobson presented their paper "Random Early Detection Gateways for Congestion Avoidance" [13] where the RED gateway was outlined. The idea behind RED is to provide, as soon as is possible, a feedback to responsive flows (like TCP) before the queue overflows in an effort to indicate that congestion is imminent, instead of waiting until the congestion has become excessive. Also, packet drops are distributed more fairly across all flows. Let's see how Floyd & Jacobson [13] explain guidelines for RED gateway design in their paper. Sorry, but this part is a little long; also if it is necessary I will insert some minor comments:

This section summarizes some of the design goals and guidelines for RED gateways. The main goal is to provide congestion avoidance by controlling the average queue size. Additional goals include the avoidance of global synchronization and of a bias against bursty traffic and the ability to maintain an upper bound on the average queue size even in the absence of cooperation from transport-layer protocols.

Okay, let's explain this a little. They try to avoid congestion by controlling the average queue size at the gateway or router. First problem when network get congested is that router's queue overflows and begins to drop packets. Avoiding packet drops (being as conservative as is possible) the global synchronization problem is minimized (where several TCP connections reduce their throughput to a minimum at the same time due, mainly, to a packet massacre at router queue). Also RED gateway pretends to be as condescending as possible with bursty traffic, trying again to protect packet survival. Finally, by controlling the maximum average queue size an indirect control is exercised over those 'bad citizen' unresponsive flows, like UDP.

The first job of a congestion avoidance mechanism at the gateway is to detect incipient congestion. As defined in [18], a congestion avoidance scheme maintains the network in a region of low delay and high throughput. The average queue size should be kept low, while fluctuations in the actual queue size should be allowed to accommodate bursty traffic and transient congestion. Because the gateway can monitor the size of the queue over time, the gateway is the appropriate agent to detect incipient congestion. Because the gateway has a unified view of the various sources contributing to this congestion, the gateway is also the appropriate agent to decide which sources to notify of this congestion.

In a network with connections with a range of roundtrip times, throughput requirements, and delay sensitivities, the gateway is the most appropriate agent to determine the size and duration of short-lived bursts in queue size to be accommodated by the gateway. The gateway can do this by controlling the time constants used by the low-pass filter for computing the average queue size. The goal of the gateway is to detect incipient congestion that has persisted for a "long time" (several roundtrip times).

Again a little explanation. A low-pass filter is a mechanism to soft or smooth the current aleatory behavior of the queue length throughout the time. Have a look to figure 2.6.2 below where two curves representing current queue size and average queue size (this obtained applying a low-pass filter to the current queue size) are depicted. They use what is called an Exponential Weighted Moving Average (EWMA); something simple but great invented by a genius mathematician (I don't know who) sometime ago.

The second job of a congestion avoidance gateway is to decide which connections to notify of congestion at the gateway. If congestion is detected before the gateway buffer is full, it is not necessary for the gateway to drop packets to notify sources of congestion. In this paper, we say that the gateway marks a packet, and notifies the source to reduce the window for that connection. This marking and notification can consist of dropping a packet, setting a bit in a packet header, or some other method understood by the transport protocol. The current feedback mechanism in TCP/IP networks is for the gateway to drop packets, and the simulations of RED gateways in this paper use this approach.

One goal is to avoid a bias against bursty traffic. Networks contain connections with a range of burstiness, and gateways such as Drop Tail and Random Drop gateways have a bias against bursty traffic. With Drop Tail gateways, the more bursty the traffic from a particular connection, the more likely it is that the gateway queue will overflow when packets from that connection arrive at the gateway [7].

Another goal in deciding which connections to notify of congestion is to avoid the global synchronization that results from notifying all connections to reduce their windows at the same time. Global synchronization has been studied in networks with

Drop Tail gateways [37], and results in loss of throughput in the network. Synchronization as a general network phenomena has been explored in [8].

In order to avoid problems such as biases against bursty traffic and global synchronization, congestion avoidance gateways can use distinct algorithms for congestion detection and for deciding which connections to notify of this congestion. The RED gateway uses randomization in choosing which arriving packets to mark; with this method, the probability of marking a packet from a particular connection is roughly proportional to that connection's share of the bandwidth through the gateway. This method can be efficiently implemented without maintaining per-connection state at the gateway.

One goal for a congestion avoidance gateway is the ability to control the average queue size even in the absence of cooperating sources. This can be done if the gateway drops arriving packets when the average queue size exceeds some maximum threshold (rather than setting a bit in the packet header). This method could be used to control the average queue size even if most connections last less than a roundtrip time (as could occur with modified transport protocols in increasingly high-speed networks), and even if connections fail to reduce their throughput in response to marked or dropped packets.

Okay. These technological monsters, Floyd & Jacobson, explain the problem and some outlines of how to approach it to be resolved. The solution they propose is overwhelming. Reading again from Floyd & Jacobson [13]:

This section describes the algorithm for RED gateways. The RED gateway calculates the average queue size, using a low-pass filter with an exponential weighted moving average. The average queue size is compared to two thresholds, a minimum threshold and a maximum threshold. When the average queue size is less than the minimum threshold, no packets are marked. When the average queue size is greater than the maximum threshold, every arriving packet is marked. If marked packets are in fact dropped, or if all source nodes are cooperative, this ensures that the average queue size does not significantly exceed the maximum threshold.

When the average queue size is between the minimum and the maximum threshold, each arriving packet is marked with probability p_a , where p_a is a function of the average queue size avg. Each time that a packet is marked, the probability that a packet is marked from a particular connection is roughly proportional to that connection's share of the bandwidth at the gateway. The general RED gateway algorithm is given in Figure 2.6.1.

```

Initialization
    avg = 0
    count = 1
end
for each packet arrival
    calculate the new average queue size 'avg'
    if the queue is nonempty
        avg = (1 - wq) * avg + wq * q // this is the low-pass filter EWMA
    else
        m = f(time - q_time)
        avg = (1 - wq)^m * avg
    end
    if minth <= avg < maxth
        increment count
        calculate probability 'pa'
        pb = maxp * (avg - minth) / (maxth - minth)
        pa = pb / (1 - count * pb)
    end
    with probability 'pa'
        mark the arriving packet
        count = 0
    end
end
else if maxth <= avg
    mark the arriving packet
    count = 0
end
else
    count = -1
end
end
end

when queue becomes empty
    q_time = time
end

Saved Variables:
    avg : average queue size
    q_time : start of the queue idle time
    count : packets since last marked packet

Fixed parameters:
    wq : queue weight
    minth : minimum threshold for queue
    maxth : maximum threshold for queue
    maxp : maximum value for pb

Other:
    pa : current packet-marking probability
    q : current queue size
    time : current time
    f(t) : a linear function of the time t

```

Figure 2.6.1

Thus the RED gateway has two separate algorithms. The algorithm for computing the average queue size determines the degree of burstiness that will be allowed in the gateway queue. The algorithm for calculating the packet-marking probability determines how frequently the gateway marks packets, given the current level of congestion. The goal is for the gateway to mark packets at fairly evenly-spaced intervals, in order to avoid biases and to avoid global synchronization, and to mark packets sufficiently frequently to control the average queue size.

These guys are not joking. RED queuing discipline is a very ingenious algorithm. To understand this even better we are going to present a figure created by using the ns-2 network simulator, taken from NS by Example [14] and then retouched by me; the simulation has to deal with a link where a RED, instead of a DropTail, queuing discipline is used. The figure represents the current queue length and the average queue length against time of the RED gateway used in the link. It can help us a lot. Have a look to figure 2.6.2 below.

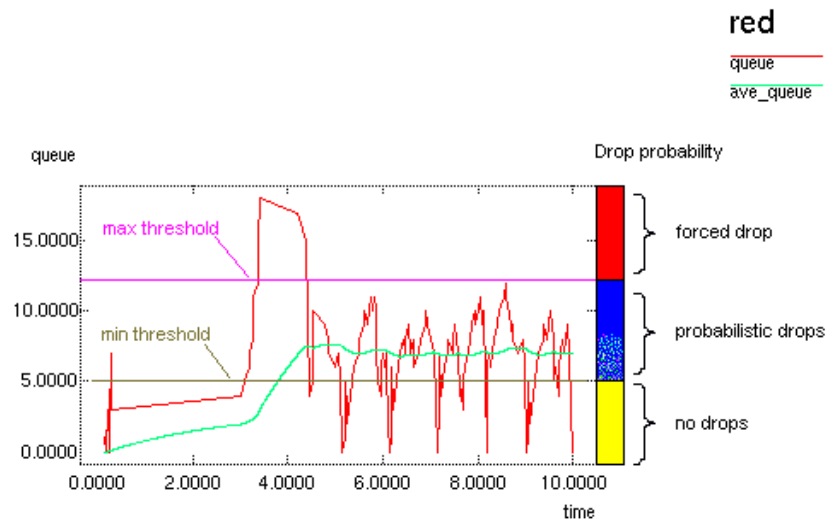


Figure 2.6.2

The red curve is the current queue size of the router; the green curve is the average queue size calculated using the EWMA low-pass filter. The minimum threshold is 5 packets and the maximum threshold is 12 packets. Observe that current queue has a very severe burst of packets in the interval between 3 and 4.2 seconds, more or less. In fact, the maximum threshold of 12 packets is violated. But the smoothed average queue size calculated using the EWMA low-pass filter is a lot slower. It follows the current queue length but like a turtle follows a rabbit. The 'drop probability' is selected based on the average queue size, not the current queue size. But let's Parris [12] to explain this better for us; I did some minor changes to adapt his explanation to our figure:

RED's packet dropping decisions are mode-based. Figure 2.6.2 illustrates the ideas behind the RED algorithm. This figure shows the instantaneous (red color) and weighted average (green color) queue size (in packets) over time.

The current mode, indicated on the right hand side, is determined by the relation between the average queue size and the minimum and maximum thresholds. When the average queue size is less than the minimum threshold this indicates no congestion, so no action is taken. This is no drop mode (yellow band on the figure) and the probability that an arriving packet will be dropped is zero. In this mode arriving packets are always enqueued.

At the other extreme, when the average queue size is greater than the maximum threshold, or if the queue is full, the algorithm infers persistent and severe congestion. All arriving packets are dropped. The probability an arriving packet will be dropped is one. This mode is referred to as forced drop mode (red band on the figure).

Finally, when the average queue size is between the two thresholds the algorithm operates in a probabilistic (i.e., random) drop mode. In this mode, arriving packets are dropped at random. The probability that a given packet will be dropped ranges between zero and one as a function of three parameters: maxp, the current average queue size avg, and count. The input parameter, maxp, determines the maximum probability that two consecutive packets will be dropped while in probabilistic drop mode. The variable, count, tracks how many packets have been enqueued since the last packet was dropped.

Very nice. In figure 2.6.1 above the RED algorithm shows how the three parameters that Parris talked about are used to implement the queuing discipline behavior. Observe also that even when current queue size breaks the 12 packets maximum threshold when the burst of packets arrive between seconds 3 and 4.2 approximately, no packet is dropped because the average queue size is below the maximum threshold. This way, the RED gateway manages intelligently bursty traffic while controlling the average queue size when the congestion becomes permanent.

In their paper, Floyd & Jacobson [13] showed that using RED gateways the network power is improved. Reading again from them:

Because RED gateways can control the average queue size while accommodating transient congestion, RED gateways are well-suited to provide high throughput and low average delay in high-speed networks with TCP connections that have large windows. The RED gateway can accommodate the short burst in the queue required by TCP's slow-start phase; thus RED gateways control the average queue size while still allowing TCP connections to smoothly open their windows. Figure 2.6.3 shows the results of simulations of the network below with two TCP connections, each with a maximum window of 240 packets, roughly equal to the delay-bandwidth product. The two connections are started at slightly different times. The simulations compare the performance of Drop Tail and of RED gateways.

[WAP Gateway](#)

WAP 2.0 High Performance
Gateway Wireless Profiled
HTTP/TCP & WSP
www.nowwap.com

[Multithread Components](#)

We offer efficient
components for atomic
multithreaded
programming!
www.pss-ab.com

[Deep Packet Inspection](#)

A User-driven Community
for DPI/DPP Learn,
discuss, and collaborate
www.dPacket.org

[MCF5213 Embedded Software](#)

MQX RTOS, TCP/IP, CAN
and HTTP Free Software
Evaluation Available
www.Embedded-Access.com

[Switching Gig Ethernets?](#)

Carrier Ethernet Transport
switches lambdas, Gig Es,
Ethernet tunnels
www.meriton.com/products/cet.php

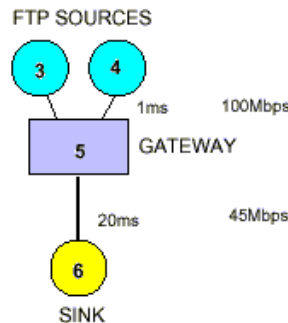
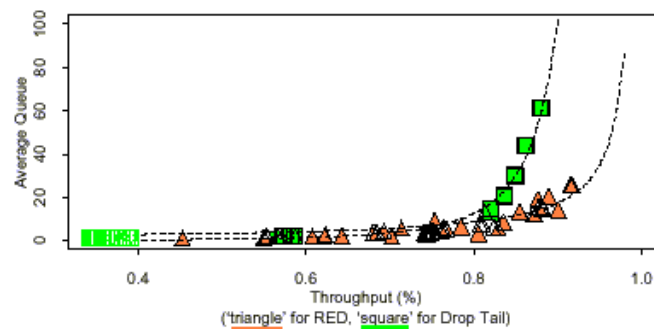


Figure 2.6.3

In the figure the x-axis shows the total throughput as a fraction of the maximum possible throughput on the congested link. The y-axis shows the average queue size in packets (as seen by arriving packets). Our simulator does not use the 4.3-Tahoe TCP code directly but we believe it is functionally identical.

Five 5-second simulations were run for each of 11 set of parameters for Drop Tail gateways, and for 11 sets of parameters for RED gateways; each mark in the figure shows the result of one of these five-second simulations. The simulations with Drop Tail gateways were run with the buffer size ranging from 15 to 140 packets; as the buffer size is increased, the throughput and the average queue size increase correspondingly.

In order to avoid phase effects in the simulations with Drop Tail gateways, the source node takes a random time drawn from the uniform distribution on $[0, t]$ seconds to prepare an FTP packet for transmission, where t is the bottleneck service time of 0.17 ms. [7].

The simulations with RED gateways were all run with a buffer size of 100 packets, with minth ranging from 3 to 50 packets. For the RED gateways, maxth is set to 3minth, with $wq = 0.002$ and $maxp = 1/50$.

The dashed lines show the average delay (as a function of throughput) approximated by $1.73/(1-x)$ for the simulations with RED gateways, and approximated by $0.1/(1-x)^3$ for the simulations with Drop Tail gateways. For this simple network with TCP connections with large windows, the network power (the ratio of throughput to delay) is higher with RED gateways than with Drop Tail gateways. There are several reasons for this difference. With Drop Tail gateways with a small maximum queue, the queue drops packets while the TCP connection is in the slow-start phase of rapidly increasing its window, reducing throughput.

On the other hand, with Drop Tail gateways with a large maximum queue the average delay is unacceptably large. In addition, Drop Tail gateways are more likely to drop packets from both connections at the same time, resulting in global synchronization and a further loss of throughput.

Okay, having the theoretical aspect of RED more or less controlled let's see now how this queuing discipline is implemented on Linux. Reading from the RED man page, we have:

Random Early Detection is a classless qdisc which limits its queue size smartly. Regular queues simply drop packets from the tail when they are full, which may not be the optimal behaviour. RED also performs tail drop, but does so in a more gradual way.

Once the queue hits a certain average length, packets enqueued have a configurable chance of being marked (which may mean dropped). This chance increases linearly up to a point called the max average queue length, although the queue might get bigger.

This has a host of benefits over simple taildrop, while not being processor intensive. It prevents synchronous retransmits after a burst in traffic, which cause further retransmits, etc.

The goal is to have a small queue size, which is good for interactivity while not disturbing TCP/IP traffic with too many sudden drops after a burst of traffic.

Depending on whether ECN is configured, marking either means dropping or purely marking a packet as overlimit.

The average queue size is used for determining the marking probability. This is calculated using an Exponential Weighted Moving Average, which can be more or less sensitive to bursts.

When the average queue size is below min bytes, no packet will ever be marked. When it exceeds min, the probability of doing so climbs linearly up to probability, until the average queue size hits max bytes. Because probability is normally not set to 100%, the queue size might conceivably rise above max bytes, so the limit parameter is provided to set a hard maximum for the size of the queue.

Okay. Bert Hubert explanation just round and confirm our understanding of this discipline. Next the parameters are explained:

min

Average queue size at which marking becomes a possibility.

max

At this average queue size, the marking probability is maximal. Should be at least twice min to prevent synchronous retransmits, higher for low min.

probability

Maximum probability for marking, specified as a floating point number from 0.0 to 1.0. Suggested values are 0.01 or 0.02 (1 or 2%, respectively).

limit

Hard limit on the real (not average) queue size in bytes. Further packets are dropped. Should be set higher than max+burst. It is advised to set this a few times higher than max.

burst

*Used for determining how fast the average queue size is influenced by the real queue size. Larger values make the calculation more sluggish, allowing longer bursts of traffic before marking starts. Real life experiments support the following guideline:
 $(\text{min} + \text{min} + \text{max}) / (3 * \text{avpkt})$.*

avpkt

Specified in bytes. Used with burst to determine the time constant for average queue size calculations. 1000 is a good value.

bandwidth

This rate is used for calculating the average queue size after some idle time. Should be set to the bandwidth of your interface. Does not mean that RED will shape for you! Optional.

ecn

As mentioned before, RED can either 'mark' or 'drop'. Explicit Congestion Notification allows RED to notify remote hosts that their rate exceeds the amount of bandwidth available. Non-ECN capable hosts can only be notified by dropping a packet. If this parameter is specified, packets which indicate that their hosts honor ECN will only be marked and not dropped, unless the queue size hits limit bytes. Needs a tc binary with RED support compiled in. Recommended.

And finally, to finish this long part we have to learn how to configure one of this RED beast using tc. First, a general command:

```
# tc qdisc add dev eth0 root limit <bytes> min <bytes> max <bytes> \
    avpkt <bytes> burst <packets> probability <number> bandwidth <kbps> \
    [ecn]
```

cm-12

Yellows indicate values we have to supply. Let's see how:

<max> is maximum threshold. To set this value we use link bandwidth and maximum desired latency. Assuming a bandwidth of 512 kbps and a maximum desired latency of 500 ms we have:

$$512 \text{ kbps} \sim 512000 \text{ bps} = 64000 \text{ bytes / sec}$$

$$\text{<max>} = 64000 \text{ bytes / sec} * 0.5 \text{ sec} = 32000 \text{ bytes}$$

Above <max> threshold we will have a packet massacre and latency doesn't matter.

<min> is minimum threshold. Must be set half of <max> threshold. Following Floyd & Jacobson, let's set <min> threshold such that <max> ~ 3 * <min>. Then we will set <min> to 12000 bytes.

<limit> is hard limit on the real queue size. This limit should never be reached. Following Lartc [7] recommendation let's set this as 8 * <max>; then <limit> will be 256000 bytes.

<avpkt> is average packet size. We set this to 1000 bytes.

<burst> allows longer burst of traffic before marking (dropping) starts; just to accommodate bursty flows. Following RED man page recommendations we have:

$$\text{<burst>} = (2 * \text{<min>} + \text{<max>}) / (3 * \text{<avpkt>})$$

$$\text{<burst>} = (2 * 12000 + 32000) / (3 * 1000) = 18.67 \sim 20.$$

<probability> is maximum probability of marking; using same value used by Floyd & Jacobson we set this to 0.02.

<bandwidth> is link bandwidth for calculating queue length when it's idle; then, bandwidth will be 512.

[ecn] is an optional parameter. If our end TCP systems are configured to respond to 'early congestion notification' you can use this flag to avoid packet dropping when average queue size is above <min> threshold and below <max> threshold. Above <max> threshold all packets are dropped; this perhaps occurs when dealing with unresponsive flows.

Well, we have all parameters completed; then using tc we order:

```
# tc qdisc add dev eth0 root red limit 256000 min 12000 max 32000
  avpkt 1000 burst 20 probability 0.02 bandwidth 512
```

cm-13

or,

```
# tc qdisc add dev eth0 root red limit 256000 min 12000 max 32000
  avpkt 1000 burst 20 probability 0.02 bandwidth 512 ecn
```

cm-14

if we have ecn responsive end-systems.

A final word about RED. When implementing it to make the work it was conceived to, this means, to feedback senders as soon as possible about incipient congestion, configure it as a root queue. It should be the queue faced to the sender. Don't configure it behind a real killer, as CBQ for instance, because you are spoiling RED nature.

And RED nature is very interesting for implementing some other behaviors instead of incipient congestion notification. Because it can drop packets at random based on a configurable probability parameter, it is tailor-made for implementing Assure Forwarding behavior. Do you remember? Same class, different drop precedences. Drop precedences can be implemented using RED. It was Werner Almesberger idea when they (Almesberger, Kuznetsov and Salim) created GRED, our next queuing discipline, of course.

[TDM Over IP Solutions](#)

E1/T1/J1 Over IP, MPLS, PON Networks For Wi-MAX, Radio and Cell. Backhaul
www.techaya.com

[SmartBits and Ixia](#)

Cost effective alternatives for SmartBits IP packet generators
www.omnicor.com

[DNS DDoS Attacks](#)

Secure64 Survived the Attacks Learn how with a Free White Paper!
www.Secure64.com

[Switching Gig Ethernets?](#)

Carrier Ethernet Transport switches lambdas, Gig Es, Ethernet tunnels
www.meriton.com/products/cet.php

[QoS Management Tools](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com



This site is visited regularly for very specialized people; it's then the best place to advertise your very specialized product. First month at no cost for you to verify this aseveration.

[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

2.7.- GRED queuing discipline

 Search

To explain why GRED was invented who's going to do that better than their creators, Werner Almesberger, Jamal Hadi Salim (*C sources indicate that Jamal wrote this code*) and Alexey Kuznetsov? Reading from their paper "*Differentiated Service on Linux*" [10], we have:

Finally, we need a queuing discipline to support multiple drop priorities as required for Assured Forwarding. For this, we designed GRED, a generalized RED. Besides four delay priorities, which can again be implemented with already existing components, Assured Forwarding also needs three drop priorities, which is more than the current implementation of RED supports. We therefore added a new queuing discipline which we call "generalized RED" (GRED). GRED uses the lower bits of `skb->tc_index` to select the drop class and hence the corresponding set of RED parameters.

Let's go slowly because this part requires to be studied carefully. We saw in previous section that RED has an inherent mechanism to mark packets randomly. Mark just means to mark packets for notifying senders of incipient congestion, or merely, dropping them. TCP with ECN option enabled can respond to ECN marked packets; TCP without ECN responds only to dropped packets. UDP and other unresponsive flows respond to nothing and packet killing is the only way to control them.

Independently if we mark or drop packets to notify senders about congestion, RED queuing discipline has the advantage that it can do that randomly using a probabilistic approach. Its original design was to be as fair as possible with entering flows, picking packets at random for trying to resolve the clustering nature of TCP behavior, for trying to control the queue length at routers and for trying to notify as soon as possible to the senders about congestion, for them to take correcting actions.

But for DS, we will not use RED this way, instead we want to take advantage of its probabilistic dropping mechanism to implement AF classes. Each of these classes need to support a three level drop precedence being the difference between levels the dropping probability. RED only offer one queue and one dropping probability, then our friends exhanced it to support, because 16 is better than 3 for computers, a 16 level of dropping probabilities just by creating 16 virtual queues with independent parameter setting. That's just GRED. 16 REDs in one, among other little tricks that we will see later.

At this time it would be very nice that a diagram comes to save us of this long-winded speech; here we have one:

[AirMagnet Free-Trial](#)

Test AirMagnet's Laptop Tool and Police/Audit/Fix your WLAN!

www.airmagnet.com

[Paper: Mobile Backhaul](#)

Expanding 3G backhaul capacity without jeopardizing 2G revenues

www.ciena.com

[Network Packet Analyzer](#)

Advanced Packet Analysis. A "must have" networking tool.

www.Paessler.com

[End to End QoS and more](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.

Packeteer.com

[SIP call troubleshooter](#)

Real-time VoIP Traffic monitoring Free Trial & Support. Download now!

www.vqmanager.com

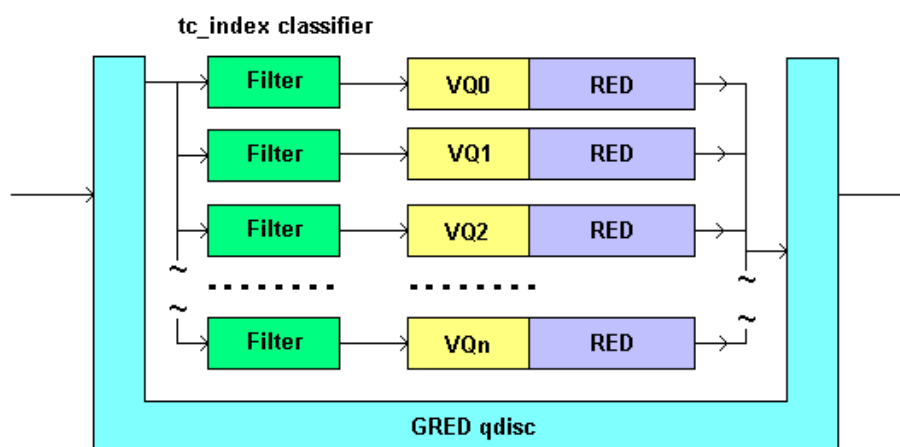


Figure 2.7.1

Okay, it looks like a full class queue. Each class has its own virtual queue. Also it has some kind of filter to select the queue each packet is going to be placed. Here the tricks begin. Designers extend the packet buffer on Linux to add a new field called *tc_index*. The packet buffer is accessed using a pointer called *skb*. Then to access the field they use *skb->tc_index*. The new field is an integer and its 4-rightmost (least significant) bits are used to select the virtual queue (VQ) where a packet is going to be located when the time of doing this comes.

The real process is really of a tricky nature that for us, simple mortals, it's good enough to know that when the packet is in front of GRED, the 4-rightmost bits of its *tc_index* field buffer are snooped and accordingly to its value (0, 1, 2, ... 15), one of the sixteen VQs is selected to receive the packet. It is very important to be clear that the *tc_index* field is not on the packet header, but instead, in the memory buffer assigned to manage the packet within the Linux Traffic Control process.

Another queuing discipline, DSMARK, that we will see later, is in charge of copying the DS field in the packet header onto the *skb->tc_index* field (previously it applies some bitwise operation to the DS field to convert it to a DSCP), and previously to the packet to be dispatched to the interface's driver, of copying back the *skb->tc_index* field onto the DS field in the packet header (previously it applies an inverse bitwise operation to the *skb->tc_index* field to convert the DSCP it contains back to a DS). All this mischievous process is executed by the DSMARK queuing discipline and by some mechanism called *tc_index* classifier, as we will see better, we hope, later on.

To continue let's see in short the GRED parameters; as you hope they are very similar what we saw before when talking about RED, but this time being GRED a multi-RED queue it is configured in two steps. First, the generic parameters are configured to select number of VQs, default VQ and the buffer sharing scheme. *tc* generic configuration is as follows:

```
#tc qdisc add dev eth0 root gred setup DPs <VQs> default <VQ> [prio]
```

cm-15

Again, yellows indicate values we have to supply. Let's see this in detail. This command sets our GRED monster as the root queue of ethernet interface 0. <VQs> will be the number of virtual queues we want. If we were trying to assemble an AF class (we need one GRED for each AF class to be implemented), <VQs> parameter should be set to 3. Its value goes from 1 (a simple RED queue) to 16 (a 16 multi-RED queue). Then, 'setup DPs 3' is very nice for us.

Next, we have to select a default queue from our queues. This queue will be selected when 4-rightmost *skb->tc_index* bits lay out of the number of VQs we selected. For example, you select 3 virtual queues and a packet arrives with its bits set to 9; this case the packet will be assigned to the default VQ. For our example the last VQ is good enough, then 'default 3' is okay.

Finally, we have an optional parameter [grio]. Adding it we will get what is called a 'Priority Buffer Scheme'. It is something similar what we saw before with PRIO qdisc, but a little bit complicated. Let's see how to explain this. If you set your GRED queuing discipline as 'grio' you can assign a priority to each VQ. Priorities range from 1 to 16 with 1 being the highest. As you remember when reading about RED, discipline behavior depends on the calculated value of the average queue length; let's call this value *qave* to use the same token used in the GRED C code. When *qave* is below *minimum threshold* we are in *no-drop* state. When *qave* is above *minimum threshold* and below *maximum threshold* we are in *probabilistic-drop* state. When *qave* is above *maximum threshold* we are in *force-drop* state.

All this means that *qave* value is crucial to define the queuing behavior. Having defined *minimum threshold* and *maximum threshold* as larger *qave* is, it's going to be higher the packet dropping probability. Then playing with *qave* value you could implement some kind of priority scheme. How the GRED designers did this? Well, when dealing with a priority 1 VQ the *qave* value is just its value (just the calculated value based on measurement). But when dealing with a priority 2 VQ its *qave* value will be its own calculated value plus the *qave* value of the priority 1 VQ. The *qave* value of a priority 5 VQ will be its own value plus the *qave* values of all VQs with higher priority index; this case *qave* from VQ from priority 4 to priority 1.

I don't like too much this scheme because it seems a little bit confuse for me. Lower priority VQs will have a *qave* value that will depend of *qave* of higher priority VQs. Of course, using this scheme we are creating a priority response. Higher priority VQs will be in better position to dispatch a packet than lower ones. But, because thresholds are individually set things complicate further. I feel that we can't control things. Well, it's a matter of personal taste; I prefer the simple scheme where we can define each VQ behaviour based on its own parameters. Anyway below is an example using grio option.

GRED using 'grio' could be used to implement an in-profile and out-of-profile scheme into the same drop precedence. For example, using two VQs with the same drop probability and different priorities. Differentiated Service architecture strength is that it establishes just a framework where we have, as joiners, enough field to play the game we want.

Well, summarizing our GRED generic command configuration could be:

```
#tc qdisc add dev eth0 root gred setup DPs 3 default 3
```

cm-16

for using the default buffer scheme, and:

```
#tc qdisc add dev eth0 root gred setup DPs 3 default 3 grio
```

cm-17

for using the priority buffer scheme.

The second step is to set parameters for individual virtual queues. This case, tc is used as follows:

```
# tc qdisc change dev eth0 root gred limit <bytes> min <bytes> \
    max <bytes> burst <packets> avpkt <bytes> bandwidth <kbps> \
    DP <vq> probability <number> [ prio <number> ]
```

cm-18

As you see, configuration is very similar what we used before with RED. Now the word 'change' is used instead of 'add' because the queue already exists created by the generic command. This time we are just 'changing' the virtual queue setting.

Comparing with RED new parameters are: *DP <vq>* where <vq> is the number, from 1 to 16, of the VQ we are configuring; and the optional parameter [*prio <number>*] where <number> is the priority assigned to this VQ. This parameter is to be used only when 'grio' option is used in the generic configuration. Let's see an example with 'grio' option:

```
# tc qdisc change dev eth0 root gred limit 60KB min 15K \
    max 45KB burst 20 avpkt 1000 bandwidth 10Mbit \
    DP 1 probability 0.02 prio 2
```

cm-19

To continue we would like to configure one of these 'severe doorman' completely as root on our ethernet interface 0 using a priority scheme. We begin explaining our goal with a diagram:

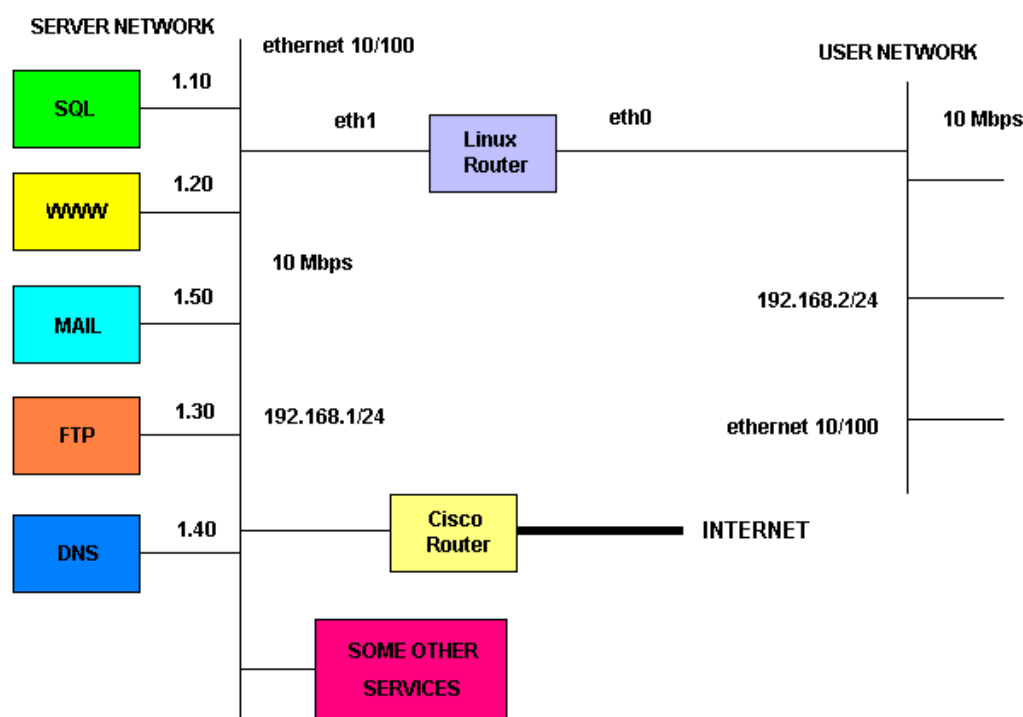


Figure 2.7.2

This is a very simple intranet domain on a 10 Mbps ethernet network. To the left the servers network is implemented under address space 192.168.1/24 having five servers: SQL, WWW, FTP, DNS and MAIL server. To the right the users network is implemented under address space 192.168.2/24 and is connected to the servers network through a Linux Router having the ethernet interface 1 on the servers side and the ethernet interface 0 on the users side. Also we have some VoIP traffic coming from Internet to our users network through the Cisco router. This traffic has to be protected when it reaches our network from some other stronger traffic. Have a look at [Voice over IP](#) in this site for more information about these fragile flows.

Our goal is to implement a GRED queuing discipline on Linux Router's ethernet interface eth0. Our monster is going to be an eight virtual queues GRED where VQ1 will correspond to the VoIP traffic, VQs 2 to 6 will correspond to DNS, SQL, WWW, FTP, and MAIL server respectively, VQ7 will correspond to UDP flows (other than UDP generated by our DNS server) and VQ8 will be reserved as default for rest of the network flows. We want to prioritize traffic from priority 1 to 8 as follows: VoIP, DNS, SQL, WWW, FTP, MAIL, other UDP and rest of traffic.

We need some expected bandwidth distribution, maximum desired latency, estimated average packet length and packet dropping probability. Using the given customer requirements and after thinking somewhat we decided to use the following table:

VQ	SERVICE	EXPECTED BANDWIDTH SHARE	MAXIMUM DESIRED LATENCY	AVERAGE PACKET LENGTH	DROP PROBABILITY
1	VoIP	3%	20ms	128	N/A
2	DNS	2%	50ms	256	N/A
3	SQL	25%	100ms	1024	1%
4	WWW	8%	100ms	512	1%
5	FTP	20%	100ms	1024	4%
6	MAIL	10%	100ms	1024	4%
7	OTHER-UDP	12%	50ms	1024	N/A
8	OTHER	20%	100ms	1024	4%

Table 2.7.1

VQ8 will be our default queue. N/A means that we will not apply drop probability to UDP flows. Below you can find an explanation for this.

For each individual flow we will have:

$$\begin{aligned} & 0.01 * \text{Bandwidth Share} * \text{Desired Latency} * \text{Network Bandwidth} \\ \text{Maximum Threshold} &= \frac{0.01 * \text{Bandwidth Share} * \text{Desired Latency} * \text{Network Bandwidth}}{8 \text{ bits/byte} * 1000 \text{ ms/sec}} \\ \text{Minimum Threshold} &= 1/2 * \text{Maximum Threshold} \\ \text{Avpkt} &= \text{Average Packet Length} \\ \text{Burst} &= (2 * \text{MinThreshold} + \text{MaxThreshold}) / (3 * \text{Avpkt}) \\ \text{Limit} &= 4 * \text{MaxThreshold} \end{aligned}$$

$$\text{Network Bandwidth (10Mbps)} = 10 * 1024 * 1024 \text{ bits/sec} = 10.485.760 \text{ bits/sec}$$

For example, for the SQL flow we will have:

$$\begin{aligned} & 0.01 * 25 * 100 * 10.485.760 \\ \text{Maximum Threshold} &= \frac{0.01 * 25 * 100 * 10.485.760}{8 * 1000} = 32768 \text{ bytes} \\ \text{Minimum Threshold} &= 1/2 * 32768 = 16384 \\ \text{Avpkt} &= 1024 \\ \text{Burst} &= (2 * 32768 + 16384) / (3 * 1024) = 27 \\ \text{Limit} &= 4 * 32768 = 131072 \end{aligned}$$

After some calculation we can fill our table completely:

SERVICE	SHARE	LATENCY	AVPKT	MAX-TH	MIN-TH	BURST	LIMIT	PKT/MIN
VoIP	3%	20	128	393	393	5	3146	3
DNS	2%	50	256	655	655	4	5243	2
SQL	25%	100	1024	32768	16384	27	131072	16
WWW	8%	100	512	10486	5243	17	41943	10
FTP	20%	100	1024	26214	13107	21	104858	12
MAIL	10%	100	1024	13107	6554	11	52429	6
OTHER-UDP	12%	50	1024	3932	3932	6	31457	3
OTHER	20%	100	1024	26214	13107	21	104858	12
TOTAL	100%						475005	

Table 2.7.2

Last column is the number of packets per queue when minimum threshold is almost reached. This value is very important to check that minimum threshold admits at least one entire packet (this means that each type of flow can have at least one packet enqueue in *no-drop* state). Below total buffer required is calculated as 475005 bytes.

Let's explain now why we set MaxThreshold = MinThreshold (really we can't be set them equal because tc complaints when doing this; but we set them very close) for UDP flows where drop probability is not a matter (in fact we set this to 1 in the configuration script; see below).

When things go nice and all flows approximate to their minimum threshold no packet is dropped and everybody is happy. When going above minimum threshold packets begin to be dropped at random but because UDP flows are unresponsive (they don't adjust their throughput themselves when packets are being dropped) we don't want them to starve TCP flows that are aggressively responsive. Then we set maximum threshold equal to minimum threshold on UDP flows to guarantee them just the minimum threshold occupancy, but not more than this; then for UDP flows every packet will be dropped above the minimum threshold. As you see dropping probability is not a matter in this case because it is zero (0%) when queue occupancy is below common minimum-maximum threshold and 1 (100%) when above.

Okay. We are ready to configure the ethernet interface eth0. Here we have the commands (don't forget that they have to be typed in one line):

Generic GRED setup

```
# tc qdisc add dev eth0 root gred setup DPs 8 default 8 prio
```

cm-20

VoIP service

```
# tc qdisc change dev eth0 root gred limit 3146 min 393 max 394
burst 5 avpkt 128 bandwidth 10Mbit DP 1 probability 1 prio 1
```

cm-21

[CESoPSN & SAToP Solutions](#)

E1/T1/J1 Over
IP,MPLS,EPON Networks
For Next Generation
Networks
www.techaya.com

[Free Packet Sniffer](#)

Packet Sniffer - It's Free.
Very advanced networking
tool.
www.Paessler.com

[Embedded AFDX/ARINC 664](#)

DO-178B Level A
certifiable software-based
solution
www.embvue.com/astack.htm

[ActiveMQ: Open Source MQ](#)

Enterprise-class JMS
messaging. Free
downloads, papers, guides
open.ionas.com/

[Deep Packet Inspection](#)

A User-driven Community
for DPI/DPP Learn,
discuss, and collaborate
www.dPacket.org

DNS server service

```
#tc qdisc change dev eth0 root gred limit 5243 min 655 max 656  
burst 4 avpkt 256 bandwidth 10Mbit DP 2 probability 1 prio 2
```

cm-22

SQL server service

```
#tc qdisc change dev eth0 root gred limit 131072 min 16384 max 32768  
burst 27 avpkt 1024 bandwidth 10Mbit DP 3 probability 0.01 prio 3
```

cm-23

WWW server service

```
#tc qdisc change dev eth0 root gred limit 41943 min 5243 max 10486  
burst 17 avpkt 512 bandwidth 10Mbit DP 4 probability 0.01 prio 4
```

cm-24

FTP server service

```
#tc qdisc change dev eth0 root gred limit 104858 min 13107 max 26214  
burst 21 avpkt 1024 bandwidth 10Mbit DP 5 probability 0.04 prio 5
```

cm-25

MAIL server service

```
#tc qdisc change dev eth0 root gred limit 52429 min 6554 max 13107  
burst 11 avpkt 1024 bandwidth 10Mbit DP 6 probability 0.04 prio 6
```

cm-26

OTHER-UDP services

```
#tc qdisc change dev eth0 root gred limit 31457 min 3932 max 3933  
burst 6 avpkt 1024 bandwidth 10Mbit DP 7 probability 1 prio 7
```

cm-27

OTHER services

```
#tc qdisc change dev eth0 root gred limit 104858 min 13107 max 26214  
burst 21 avpkt 1024 bandwidth 10Mbit DP 8 probability 0.04 prio 8
```

cm-28

Checking our GRED queue using 'tc qdisc show' we have:

```
# tc -d -s qdisc show dev eth0
```

```
qdisc gred 8002:
```

```
DP:1 (prio 1) Average Queue 0b Measured Queue 0b
```

```
Packet drops: 0 (forced 0 early 0)
```

```
Packet totals: 0 (bytes 0)
```

```
limit 3146b min 393b max 394b ewma 2 Plog 1 Scell_log 5
```

```
DP:2 (prio 2) Average Queue 0b Measured Queue 0b
```

```
Packet drops: 0 (forced 0 early 0)
```

```
Packet totals: 0 (bytes 0)
```

```
limit 5243b min 655b max 656b ewma 2 Plog 1 Scell_log 6
```

```
DP:3 (prio 3) Average Queue 0b Measured Queue 0b
```

```
Packet drops: 0 (forced 0 early 0)
```

```
Packet totals: 0 (bytes 0)
```

```
limit 128Kb min 16Kb max 32Kb ewma 4 Plog 21 Scell_log 10
```

```
DP:4 (prio 4) Average Queue 0b Measured Queue 0b
```

```
Packet drops: 0 (forced 0 early 0)
```

```
Packet totals: 0 (bytes 0)
```

```
limit 41943b min 5243b max 10486b ewma 4 Plog 20 Scell_log 9
```

```
DP:5 (prio 5) Average Queue 0b Measured Queue 0b
```

```
Packet drops: 0 (forced 0 early 0)
```

```
Packet totals: 0 (bytes 0)
```

```
limit 104858b min 13107b max 26214b ewma 4 Plog 19 Scell_log 10
```

```
DP:6 (prio 6) Average Queue 0b Measured Queue 0b
```

```
Packet drops: 0 (forced 0 early 0)
```

```
Packet totals: 0 (bytes 0)
```

```
limit 52429b min 6554b max 13107b ewma 3 Plog 18 Scell_log 9
```

```
DP:7 (prio 7) Average Queue 0b Measured Queue 0b
```

```
Packet drops: 0 (forced 0 early 0)
```

```
Packet totals: 0 (bytes 0)
```

```
limit 31457b min 3932b max 3933b ewma 2 Plog 1 Scell_log 8
```

```
DP:8 (prio 8) Average Queue 0b Measured Queue 0b
```

```
Packet drops: 0 (forced 0 early 0)
```

```
Packet totals: 72 (bytes 7488)
```

```
limit 104858b min 13107b max 26214b ewma 4 Plog 19 Scell_log 10
```

```
Sent 7488 bytes 72 pkts (dropped 0, overlimits 0)
```

Well, first half of the work is done; first half because we have to implement some way of placing different service packets on their specific virtual queues. We are going to do that on ethernet interface eth1 using what is known as an *ingress queue*. This queue combined with the u32 classifier will be good enough to reach our goal. But first, what is an ingress queue? Lartc [7] has the answer; reading from it:

All qdiscs discussed so far are egress qdiscs. Each interface however can also have an ingress qdisc which is not used to send packets out to the network adaptor. Instead, it allows you to apply tc filters to packets coming in over the interface, regardless of whether they have a local destination or are to be forwarded.

As the tc filters contain a full Token Bucket Filter implementation, and are also able to match on the kernel flow estimator, there is a lot of functionality available. This effectively allows you to police incoming traffic, before it even enters the IP stack.

The ingress qdisc itself does not require any parameters. It differs from other qdiscs in that it does not occupy the root of a device. Attach it like this:

```
# tc qdisc add dev eth0 ingress
```

This allows you to have other, sending, qdiscs on your device besides the ingress qdisc.

Very nice and clear. Simple command above creates an ingress queue on device eth0 and numbers it by default as *ffff*:. We can also number the queue explicitly; for example to create an ingress queue in our Linux router ethernet interface eth1 we just type:

```
# tc qdisc add dev eth1 handle ffff: ingress
```

cm-29

Identifying flows from our servers is very easy; we just use source address as the selection criteria to do this. This way DNS, SQL, WWW, FTP and MAIL internal provided services are easily identified.

UDP protocol packets are also easily identified. Our problem is that it is not as easy to identify properly VoIP packets from other UDP packets. Then we are going to use a trick with the help of *netfilter*. Nevertheless it is not an infallible solution. Some non-VoIP packets will be accepted by our not so tricky rule. Well, I haven't found a better solution for this. Then we will go ahead with what we have.

There is an *iptables* extension called 'length patch' by James Morris; have a look to *Netfilter Extensions HOWTO [15]* for more information. The extension adds a new match that allows you to match a packet based on its length. Because VoIP packets are very small we will match as VoIP packets all UDP packets being their length between 40 and 160 bytes. Some non-VoIP packets are going to trick us, but as I told you before, if you have a better solution please let me know as soon as you can. Also, we will match only port numbers above 5000 that correspond to general applications (current practice within the Unix operating system where port numbers below 1024 can only be used by privileged processes and port numbers between 1024 and 5000 are automatically assigned by the operating system). This way we can filter, for example, DNS UDP (port 53) packets from VoIP UDP packets.

We will mark VoIP packets (and some other UDP dwarfity) entering eth1 using this *iptables* command:

```
# iptables -t mangle -A PREROUTING -i eth1 -p udp --dport 5001:
--sport 5001: -m length --length 40:160 -j MARK --set-mark 1
```

cm-30

To place packets on different GRED VQs we will use the *u32* classifier. Let's see an example of this monster:

```
# tc filter add dev eth1 parent ffff: protocol ip prio 1 u32
match ip src 192.168.1.1/32 flowid :1
```

cm-31

This command will set to 1 the *skb->tc_index* of packets entering eth1 coming from host 192.168.1.1 (or having 192.168.1.1 as source address). *flowid :1* is the instruction that makes this possible. Then when being forwarded to our outgoing eth0 interface those packets will be placed on GRED VQ number 1. (*GRED uses the lower bits of skb->tc_index to select the VQ; read this somewhere above*) Great!!

What about VoIP packets? We will use a different approach but using the same tool; let's see how:

```
# tc filter add dev eth1 parent ffff: protocol ip prio 1  
    handle 2 fw flowid :4
```

cm-32

This time we are using *fwmark* to select our packets. All of them having their *fwmark* set to 2 (*handle 2 fw*) will be set to 4 on their *skb->tc_index* (*flowid :4*). And we can set the *fwmark* using the *iptables* command above (see command *cm-30*). Ja, Ja... We have all those packets controlled, haven't we? Using *iptables* we mark *fwmark* for small UDP packets as 2, for example, and then with the *fw* filter we set the *skb->tc_index* to 4 for these packets. But this means that they will be put on GRED VQ number 4 when being forwarded to the interface *eth0*. Great, again!!

And, what to do with other UDP packets? Very easy. UDP is protocol number 17. Then we can use this *u32* filter variation:

```
# tc filter add dev eth1 parent ffff: protocol ip prio 2 u32  
    match ip protocol 17 0xff flowid :5
```

cm-33

This rule matches UDP packets putting them on GRED VQ number 5. But observe that now the *prio* option is 2. Filter are treated following the *prio* option. *prio 1* is treated first, then *prio 2*, and so on. Then we put our small UDP packets (VoIP) on a *prio1* filter to place them on some GRED VQ; this filter will be treated first and all the small UDP packets will be placed on GRED VQ 1, for example. Having next another filter with a higher *prio* number, we can select rest of UDP packets placing them on another GRED VQ. It's very nice!! Isn't it?

Okay, fellows. Having all these explanations 'on-board' we have here the filter commands for each one of our services:

VoIP service

```
# iptables -t mangle -A PREROUTING -i eth1 -p udp --dport 5001:
--sport 5001: -m length --length 40:160 -j MARK --set-mark 1
# tc filter add dev eth1 parent ffff: protocol ip prio 1
handle 1 fw flowid :1
```

cm-34

DNS server service

```
# tc filter add dev eth1 parent ffff: protocol ip prio 2 u32
match ip src 192.168.1.40/32 flowid :2
```

cm-35

SQL server service

```
# tc filter add dev eth1 parent ffff: protocol ip prio 3 u32
match ip src 192.168.1.10/32 flowid :3
```

cm-36

WWW server service

```
# tc filter add dev eth1 parent ffff: protocol ip prio 4 u32
match ip src 192.168.1.20/32 flowid :4
```

cm-37

FTP server service

```
# tc filter add dev eth1 parent ffff: protocol ip prio 5 u32
match ip src 192.168.1.30/32 flowid :5
```

cm-38

MAIL server service

```
# tc filter add dev eth1 parent ffff: protocol ip prio 6 u32
match ip src 192.168.1.50/32 flowid :6
```

cm-39

OTHER-UDP services

[Deep Packet Inspection](#)

A User-driven Community for DPI/DPP Learn, discuss, and collaborate
www.dPacket.org

[Switching Gig Ethernets?](#)

Carrier Ethernet Transport switches lambdas, Gig Es, Ethernet tunnels
www.meriton.com/products/cet.php

[DNS Server Attacks](#)

We survived and you can too! Get the free whitepaper
www.Secure64.com

[Porta-Pak Tank Deodorizer](#)

by Walex Products Company Just Drop it in! Dissolvable Packet
www.rvsanitation.com

[Acuerate Performance Cues](#)

Acuerate Cue Technology. Stephen Hendry plays Acuerate...
www.acuerate.com

```
# tc filter add dev eth1 parent ffff: protocol ip prio 7 u32  
  match ip protocol 17 0xff flowid :7
```

cm-40

Rest of flows being not caught by our filters will be placed on default VQ number 8.

Well, with this explanation we have finished with GRED. Our next target in our DS tour will be HTB (Hierarchical Token Bucket) queuing discipline, a new qdisc from Martin Devera that was included on Linux beginning with the kernel 2.4.20. I invite you to keep on reading our next section.



This site is visited regularly for very specialized people; it's then the best place to advertise your very specialized product. First month at no cost for you to verify this aseveration.

[Previous](#)[Content](#)[Next](#)

[Previous](#)[Content](#)[Next](#)

2.8.- HTB queuing discipline

Finally, after doing some test, we decided to use HTB instead of CBQ to implement our Differentiated Service on Linux HOWTO. The original *Differentiated Service on Linux* [10] implementation was developed using CBQ when it was necessary. Basically two of the example scripts that were included in the original implementation were based on CBQ: AFCBQ that implements an example of the DS Assure Forwarding PHB (AF-PHB) and EFCBQ that implements an example of the DS Expedited Forwarding PHB (EF-PHB).

Then, why do we decide to base our HOWTO on HTB instead of CBQ? Perhaps reading a little from Lartc [7] we can give an answer to this question. Let's see how.

People from Lartc [7] don't like too much CBQ queuing discipline; let's read what they say about this:

As said before, CBQ is the most complex qdisc available, the most hyped, the least understood, and probably the trickiest one to get right. This is not because the authors are evil or incompetent, far from it, it's just that the CBQ algorithm isn't all that precise and doesn't really match the way Linux works.

Besides being classful, CBQ is also a shaper and it is in that aspect that it really doesn't work very well. It should work like this. If you try to shape a 10mbit/s connection to 1mbit/s, the link should be idle 90% of the time. If it isn't, we need to throttle so that it IS idle 90% of the time.

This is pretty hard to measure, so CBQ instead derives the idle time from the number of microseconds that elapse between requests from the hardware layer for more data. Combined, this can be used to approximate how full or empty the link is.

This is rather circumspect and doesn't always arrive at proper results. For example, what if the actual link speed of an interface that is not really able to transmit the full 100mbit/s of data, perhaps because of a badly implemented driver? A PCMCIA network card will also never achieve 100mbit/s because of the way the bus is designed - again, how do we calculate the idle time?

It gets even worse if we consider not-quite-real network devices like PPP over Ethernet or PPTP over TCP/IP. The effective bandwidth in that case is probably determined by the efficiency of pipes to userspace - which is huge.

People who have done measurements discover that CBQ is not always very accurate and sometimes completely misses the mark.

Love enough, don't you think? Initially, when this HOWTO was started, I decided to base the document on CBQ. My decision was based in the following arguments:

- CBQ was already part of the standard kernel. HTB not yet.
- Differentiated Service on Linux designers used CBQ to implement their models. When required DS examples in the original Diffserv package were based on CBQ.

[Network Traffic Control](#)

Control network traffic easy! Set quotas and calculate expenses.
www.ldecoGateway.com

[Free File Transfer](#)

Transfer Files over the Internet Easy, Secure and Private!
www.powerfolder.com

[FireWall Download](#)

Download the latest Software Titles 30 Days Risk Free Trial
ITPro.co.uk/Firewall

[Firewall Log Analyzer](#)

F/W Log Analysis & Reporting Tool For B/W & User Activity Monitoring.
www.fwanalyzer.com

[Free Download](#)

LogMeIn - Leading Way For Geeks To Access PCs from Anywhere Free
www.LogMeIn.com

But, finally, at last, HTB was included in the standard Linux kernel. Immediately I wrote this note.

Note: Reading again from [HTB user guide](#) I discovered (too late, I admit this) that beginning from kernel 2.4.20, HTB is on standard Linux kernel already. Being the case it would be really a lot better to re-write Differentiated Service scripts to porting them from CBQ to HTB. I really prefer to use HTB. Its design is clearer, command are simpler and support and a very good user documentation exists thank to Martin Devera, the designer. Also Stef Coene does an invaluable work supporting HTB through the LARTC list. I'm going to evaluate all this to study the possibility to step back this HOWTO and trying to use HTB instead of CBQ for implementing DS.

I did my work implementing AFCBQ and EFCBQ using HTB. The results were excellent. General behavior was much better using HTB than using CBQ. Then I renamed AFCBQ and EFCBQ as AFHTB and EFHTB. This HOWTO is based on these scripts instead of the original ones.

What are the real problems with CBQ? First, its configuration is very complicated. Too many parameters make the work tedious. Second, it isn't accurate. Almost everyone agree that HTB is better than CBQ to make the work they are intended to do. In fact, Alexey Kuznetsov, the CBQ Linux implementation designer admitted (I read this in Lartc list some time ago), that HTB did the work better. Martin Devera (the HTB designer) was very happy that day.

HTB (and CBQ too) is what is known as a hierachical link-sharing queuing discipline. If you want to know more about this type of queuing discipline it is a good idea to have a read to the paper *Link-sharing and Resource Management Models for Packet Networks* written by Sally Floyd and Van Jacobson in 1.995.

Some concepts taken from this document are very interesting to understand the model behind HTB design. They wrote, for example:

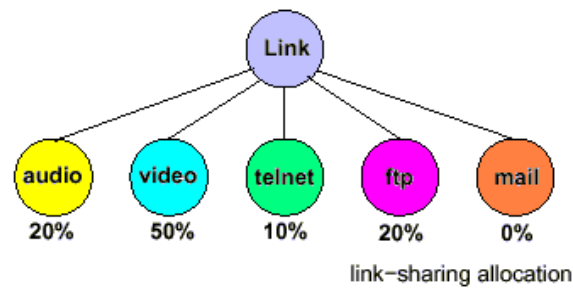
One requirement for link-sharing is to share bandwidth on a link between multiple organizations, where each organization wants to receive a guaranteed share of the link bandwidth during congestion, but where bandwidth that is not being used by one organization should be available to other organizations sharing the link.

Well, the idea is the same. It is very difficult to be always overprovisioned. Time will come when you will be underprovisioned and perhaps congested. When this occur you would like to have some mechanism to guarantee a share of the available bandwidth to the organizations that receive the service. Organizations share the link and after having some agreement you are responsible for guaranteeing the respect to the terms of such agreement.

Floyd & Jacobson recognize two type of structures to give bandwidth distribution to the organizations interested to be served. The first of them is know as *flat link-sharing* structure. Let's take some paragraphs from the paper to clear our knowledge. They wrote:

For a flat link-sharing structure such as in Figure 2.8.1, the link-sharing requirements are fairly straightforward. A link-sharing bandwidth is allocated to each class (expressed in Figure 2.8.1 as a percentage of the overall link bandwidth). These link-sharing allocations could be either static (permanently assigned by the network administrator) or dynamic (varying in response to current conditions on the network, according to some predetermined algorithm). The first link-sharing goal is that each class with sufficient demand should be able to receive roughly its allocated bandwidth, over some interval of time, in times of congestion.

As a consequence of this link-sharing goal, in times of congestion some classes might be restricted to their link-sharing bandwidth. For a class with a link-sharing allocation of zero, such as the mail class in Figure 2.8.1, the bandwidth received by this class is determined by the other scheduling mechanisms at the gateway; the link-sharing mechanisms do not "guarantee" any bandwidth to this class in times of congestion.

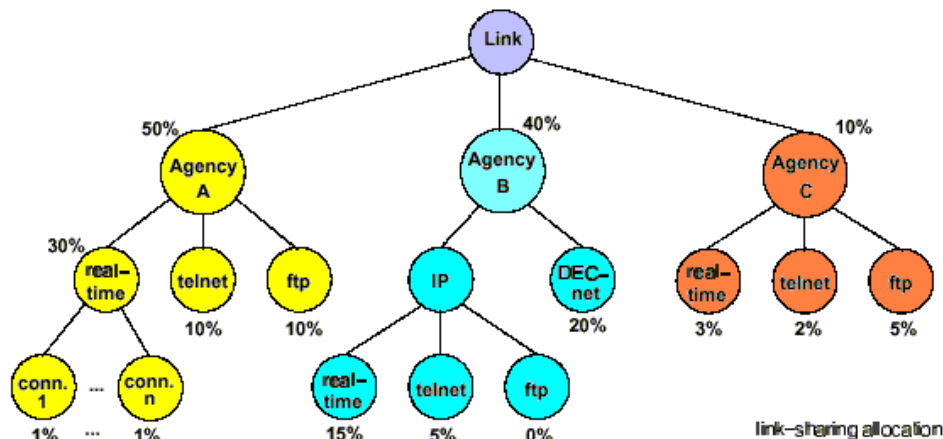


Link-sharing between service classes.

Figure 2.8.1

The second type is known as *hierarchical link-sharing* structure. Again, Floyd & Jacobson explain this as follows:

Multiple link-sharing constraints at a gateway can be expressed by a hierarchical link-sharing structure such as in Figure 2.8.2. The link-sharing structure in Figure 2.8.2 illustrates link-sharing between organizations, between protocol families, between service classes, and between individual connections within a service class; this is not meant to imply that all link-sharing structures at all links should include all of these forms of link-sharing. All arriving packets at the gateway are assigned to one of the leaf classes; the interior classes are used to designate guidelines about how 'excess' bandwidth should be allocated. Thus, the goal is that the three service classes for agency A should collectively receive 50% of the link bandwidth over appropriate time intervals, given sufficient demand. If the real-time class for agency A has little data to send, the hierarchical link-sharing structure specifies that the 'excess' bandwidth should be allocated to other subclasses of agency A.



A hierarchical link-sharing structure.

Figure 2.8.2

The link-sharing goals can be summarized as follows:

1. Each interior or leaf class should receive roughly its allocated link-sharing bandwidth over appropriate time intervals, given sufficient demand.
2. If all leaf and interior classes with sufficient demand have received at least their allocated link-sharing bandwidth, the distribution of any 'excess' bandwidth should not be arbitrary, but should follow some set of reasonable guidelines.

Observe that this structure is more flexible than the flat one. In the flat approach you have only one level of bandwidth distribution. In the hierarchical approach you can have multiple levels of bandwidth distribution increasing the flexibility to distribute the available bandwidth between the classes to be served.

Have a look to the hierarchical structure. Observe that the classes could be organizations or agencies (first level in the figure); protocol families (second level in the B agency); traffic types, as is shown in the second level of A and C agencies and the third level of B agency; or connections as is shown in the bottom level of A agency.

To implement a hierarchical link-sharing structure you have to put a lot of resources in the game because the model consumes CPU cycles generously. With the current technology managing high-speed WAN links using hierarchical approach is yet non-practical. For example, Cisco doesn't have yet this type of queuing discipline implemented in its routers. In this case Linux is one step ahead from Cisco because two hierarchical queuing disciplines are already implemented: CBQ and HTB.

Let us suppose we want to implement a hierarchical link-sharing structure using HTB. HTB is so nice that it is not necessary to begin with a long explanation about parameters and all those complicated matters. We can go directly, on a fast-track explanation. Let's begin by drawing our network example scheme:

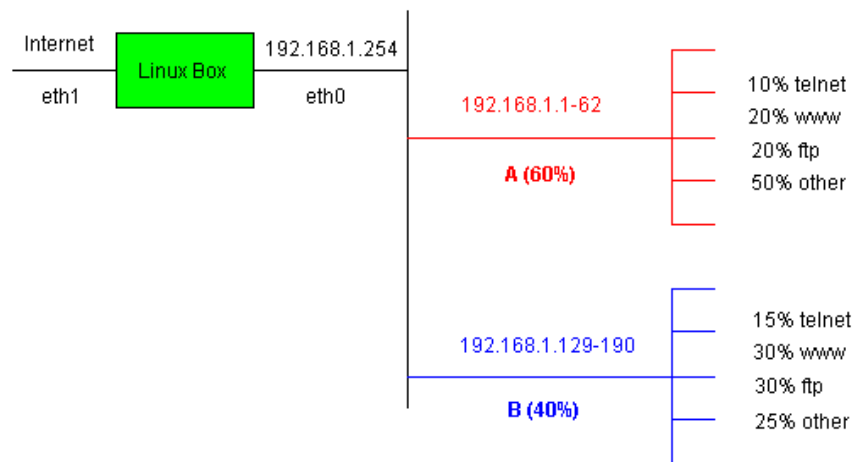


Figure 2.8.3

Two subnetworks, A and B, are going to be fed from a 2.048Mbps Internet connection through the 192.168.1.254 interface of our green Linux box depicted in the figure. A HTB queuing discipline will be installed on ethernet interface eth0 to control bandwidth distribution. Subnetwork A will have 60% of the available bandwidth guaranteed; subnetwork B the rest (40%). Within each subnetwork the guarantee flows for each type of service are as is indicated in the figure.

Next we draw our traditional queuing discipline diagram:

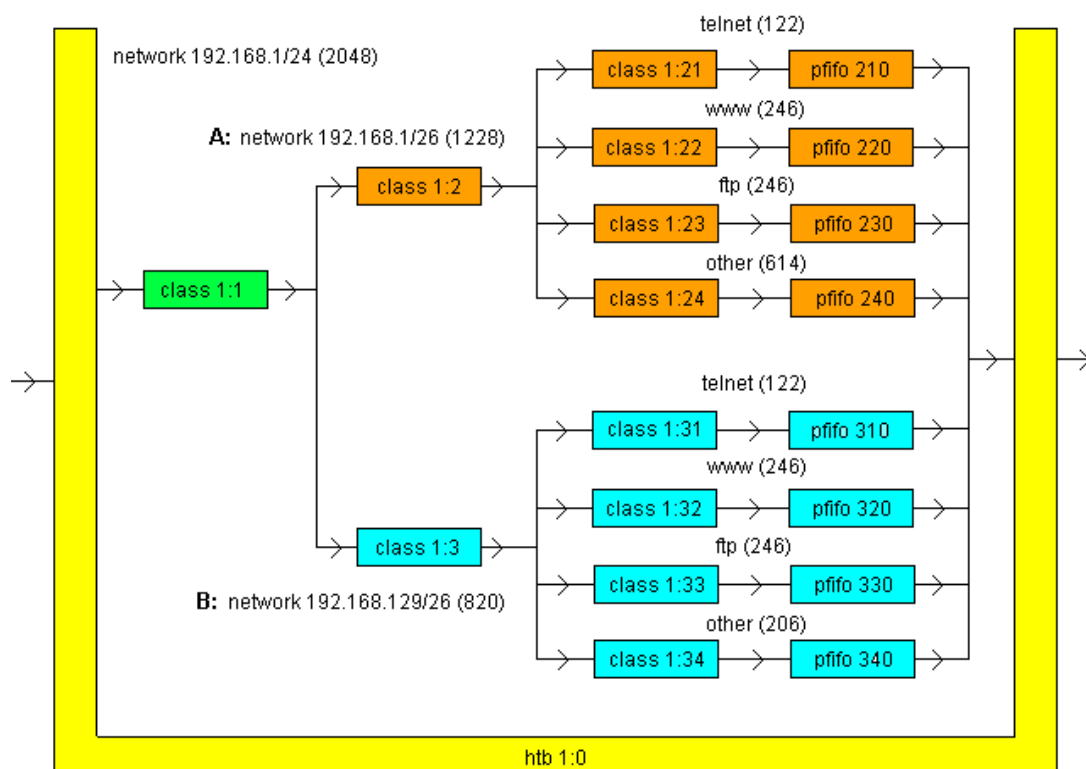


Figure 2.8.4

Yellow figure represents our HTB queuing discipline named 1:0. To permit borrowing between classes (more about this is explained below) HTB qdisc requires to create a root class depicted in the figure as a green rectangle; the root class is named 1:1. From this class we set two subclasses: class 1:2 will be used to control traffic to the network 192.168.1/26 and class 1:3 will be used to control traffic to the network 192.168.129/26. These classes represent the second level of hierarchy in our implementation. The third level is represented by the subclasses 1:21, 1:22, 1:23 and 1:24 belonging to the class 1:2 and the subclasses 1:31, 1:32, 1:33 and 1:34 belonging to the subclass 1:3. These subclasses will be used to distribute bandwidth at the service (third) level. For each of these subclasses a pfifo queuing discipline is configured for packet enqueueing. Final network, sub-networks and services bandwidth requirements are shown in parenthesis in kilobits per second (kbit).

Our next step will be to configure the HTB queuing discipline. We depict each command first and then a brief explanation is given to clear what we are doing.

```
#tc qdisc add dev eth0 root handle 1:0 htb
```

cm-41

[End to End QoS and more](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.

Packeteer.com

[FireWall Download](#)

Read Latest Articles & Comments Sign Up-Exclusive Member Benefits!

ITPro.co.uk/Firewall

[Network Bandwidth Test](#)

Find - Network Bandwidth Test. Network Bandwidth Test Guide.

www.InternetSpeedTestChest.com

[Bandwidth Meter](#)

Network usage & monitor bandwidth. Download now. (Freeware available)

www.Paessler.com/BandwidthMeter/

[Go Sailing with SailTime](#)

Hot new fractional sailing program gets you on the water today.

www.sailtime.com

This command creates the root HTB queuing discipline on ethernet interface eth0. Nothing more is required for our example. The queuing discipline is called 1:0.

```
# tc class add dev eth0 parent 1:0 classid 1:1 htb rate 2048kbit
```

cm-42

Next we create the root class 1:1. The *rate* parameter establishes the maximum bandwidth that will be permitted by the queuing discipline.

```
# tc class add dev eth0 parent 1:1 classid 1:2 htb \
    rate 1228kbit ceil 2048kbit
# tc class add dev eth0 parent 1:1 classid 1:3 htb \
    rate 820kbit ceil 2048kbit
```

cm-43

Now we create the class 1:2 to control flows to the A network (192.168.1/26) and the class 1:3 to control flows to the B network (192.168.129/26). The *rate* parameter is the minimum bandwidth that will be guaranteed to a class when all the classes in the queuing discipline are underprovisioned; this means, when flows reclaim rates that are equal or above the values set by the *rate* parameters. Let's explain this better. Being the A network flows equal or above 1228kbit and (simultaneously) the B network flows equal or above 820kbit, the minimum bandwidth to be guaranteed to classes 1:2 and 1:3 will be 1228kbit and 820kbit respectively.

HTB behavior is explained in *HTB User Guide* [17] as follows: *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.*

What about if a class is overprovisioned? Then the *ceil* parameter enters the game. For example, being the class 1:3 overprovisioned (consuming less than 820kbit), the class 1:2 can reclaim the excess left by the 1:3 class and uses this excess up to the value set by the *ceil* parameter, this means, up to 2048kbit. But, and this is very important, to allowing this borrowing (between classes 1:2 and 1:3), the HTB queuing discipline has to be configured including the root class 1:1.

But, what about if we don't want this *borrowing*? For example, let's suppose A and B network owners pay a fixed rate for their connections. They don't want that when he/she is not using his/her bandwidth another people uses it to take advantage of the share he/she pays. To avoid this problem we can re-write the last two commands using the value of the *rate* parameter as the *ceil* parameter; this way:

```
# tc class add dev eth0 parent 1:1 classid 1:2 htb \
    rate 1228kbit ceil 1228kbit
# tc class add dev eth0 parent 1:1 classid 1:3 htb \
    rate 820kbit ceil 820kbit
```

cm-44

Our next step will be to configure the service classes on each network. Let's begin with network A:

```
# tc class add dev eth0 parent 1:2 classid 1:21 htb \
    rate 122kbit ceil 1228kbit
# tc class add dev eth0 parent 1:2 classid 1:22 htb \
    rate 246kbit ceil 1228kbit
# tc class add dev eth0 parent 1:2 classid 1:23 htb \
    rate 246kbit ceil 1228kbit
# tc class add dev eth0 parent 1:2 classid 1:24 htb \
    rate 614kbit ceil 1228kbit
```

cm-45

Observe here that we set the rates defined for each service but also we define a *ceil* parameter equal to the share permitted to the network A. This is another way to limit the maximum amount to be assigned to network A. Being the *ceil* parameter of classes 1:2 and 1:3 set to 2048kbit will be not matter because the top level will be controlled by the *ceil* parameter on service level subclasses.

The commands are very similar to configure the network B:

```
# tc class add dev eth0 parent 1:3 classid 1:31 htb \
    rate 122kbit ceil 820kbit
# tc class add dev eth0 parent 1:3 classid 1:32 htb \
    rate 246kbit ceil 820kbit
# tc class add dev eth0 parent 1:3 classid 1:33 htb \
    rate 246kbit ceil 820kbit
# tc class add dev eth0 parent 1:3 classid 1:34 htb \
    rate 206kbit ceil 820kbit
```

cm-46

With these commands we have configured the part corresponding to the HTB queuing discipline. As you see HTB configuration is a very easy and friendly process. Having defined your class hierarchy you have to deal only with the *rate* and *ceil* parameters to get the job done. Really HTB has some other parameters to refine its behavior. But because this is the Differentiated Service HOWTO and not the HTB HOWTO, I suggest that those of you interested on having more HTB information have a look to the [HTB user guide](#). As was told before Martin Devera (HTB designer) and Stef Coene (one of the Lartc maintainer) have done an excellent work supporting HTB through the Lartc list.

To continue let's add a pfifo queuing discipline to each of the service classes; the commands are as follows:

```
#tc qdisc add dev eth0 parent 1:21 handle 210: pfifo limit 10
#tc qdisc add dev eth0 parent 1:22 handle 220: pfifo limit 10
#tc qdisc add dev eth0 parent 1:23 handle 230: pfifo limit 10
#tc qdisc add dev eth0 parent 1:24 handle 240: pfifo limit 10
#tc qdisc add dev eth0 parent 1:31 handle 310: pfifo limit 10
#tc qdisc add dev eth0 parent 1:32 handle 320: pfifo limit 10
#tc qdisc add dev eth0 parent 1:33 handle 330: pfifo limit 10
#tc qdisc add dev eth0 parent 1:34 handle 340: pfifo limit 10
```

cm-47

Well, fellows, we are almost ready. But, first, we have to configure the filter that place the packets on each class. Let's see how to do this beginning with network A:

```
#tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \
    match ip dst 192.168.1/26 match ip sport 23 0xffff flowid 1:21
#tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \
    match ip dst 192.168.1/26 match ip sport 80 0xffff flowid 1:22
#tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \
    match ip dst 192.168.1/26 match ip sport 20 0xffff flowid 1:23
#tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \
    match ip dst 192.168.1/26 match ip sport 21 0xffff flowid 1:23
#tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \
    match ip dst 192.168.1/26 flowid 1:24
```

cm-48

All these filter elements (the filter is just one, the *prio1* filter) match to destination ip address range 192.168.1/26 corresponding to network A. The *flowid* parameter indicates the class where the packets matching the corresponding filter element will be placed. Source ports 23, 80, 20 and 21 correspond to service telnet, www, ftp and ftp-data respectively from the remote servers. Last filter element matches any packet belonging to the network 192.168.1/26 not being previously matched by upper elements (in fact, the rest of A network flows). Because 192.168.1/24 is a private address range some kind of NAT server has to be implemented on the Linux box in front to the Internet connection. But this is not a matter to this HOWTO.

Commands to configure B network filter elements are very similar:

[Network Bandwidth Test](#)

Find - Network Bandwidth Test. Network Bandwidth Test Guide.
www.InternetSpeedTestChest.com

[Bandwidth](#)

Find - Bandwidth. Bandwidth Guide.
Bandwidthknowledge.com

[Full Linux Management](#)

App Discovery, Monitoring, Control Up in 20 mins. Download free copy!
Hyperic.com

[Your Own Server - \\$29.95](#)

Don't Pay More for Shared Hosting! 40GB Space, 200GB Data Transfer.
www.ServerPronto.com

[IP3 Networks Store](#)

Network Management price leader 866-891-3602
www.DoubleRadius.com

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \  
    match ip dst 192.168.129/26 match ip sport 23 0xffff flowid 1:31  
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \  
    match ip dst 192.168.129/26 match ip sport 80 0xffff flowid 1:32  
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \  
    match ip dst 192.168.129/26 match ip sport 20 0xffff flowid 1:33  
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \  
    match ip dst 192.168.129/26 match ip sport 21 0xffff flowid 1:33  
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \  
    match ip dst 192.168.129/26 flowid 1:34
```

cm-49

Okay, estimable lecturers, we have finished certainly with HTB. Our next step will be one of the Differentiated Service key queuing disciplines: the DSMARK queuing discipline in charge of marking DS packets. But this is a matter for another section. Let us go on.



This site is visited regularly for very specialized people; it's then the best place to advertise your very specialized product. First month at no cost for you to verify this aseveration.

[Previous](#)[Content](#)[Next](#)

[Previous](#)[Content](#)[Next](#)

2.9.- DSMARK queuing discipline

DSMARK queuing discipline was specially designed by Werner Almesberger (*C sources indicate that Werner wrote this code*), Jamal Hadi Salim and Alexey Kuznetsov to fulfill with the Differentiated Service specifications. It is in charge of packet marking in the DS Linux implementation.

Its behavior is really very simple when it is compared with some other Linux Traffic Control queuing disciplines. People complain about how to use DSMARK because the available documentation is really highly technical. I will try to clear in this section, as far as my understanding permits this, how to configure and how to use the DSMARK queuing discipline.

Let us start by telling that, on the contrary to other queuing disciplines, DSMARK doesn't shape, control or police traffic. It doesn't prioritize, delay, reorder or drop packets. It just marks packets using the DS field. If you, dear lector, jump directly to this section to know, ASAP, how DSMARK work, I would suggest you to have a brief read to section 1.3 where the [DS Field](#) is explained.

To begin with our explanation we draw our traditional diagram, this time related to the DSMARK qdisc:

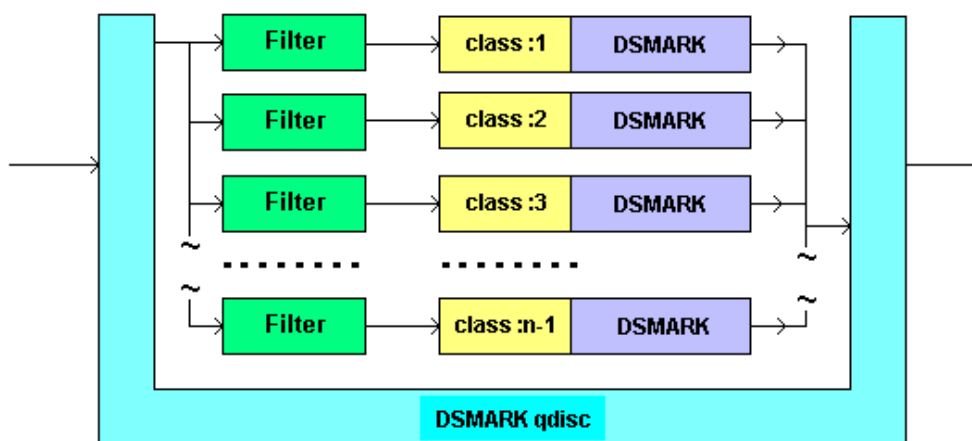


Figure 2.9.1

Okay, DSMARK looks like a full class queue. And in fact it is. Classes are numbered as 1, 2, 3, 4,..., $n-1$, where n is a parameter that defines the size of one internal table required to implement the queuing discipline behavior. This parameter is called *indices*. Being q the queue number, the element $q:0$ is the main

[Field Day T-Shirts](#)

New 2008 Designs Are In!
Free Survival Kit with
each T-Shirt
www.4jaxco.com

[SIP call troubleshooter](#)

Real-time VoIP Traffic
monitoring Free Trial &
Support. Download now!
www.vqmanager.com

[QoS Management Tools](#)

Automatic QoS for 600+
Applications - Free
Performance Survival
Guide.
Packeteer.com

[Pinnacle Oil Tools](#)

Canadian Supplier of Oil
Well Tools Packers,
Retainers, Inflates, Plugs
www.PinnacleOilTools.ca

[Network Traffic Control](#)

Control network traffic
easy! Set quotas and
calculate expenses.
www.ldecoGateway.com

queue itself. Elements from $q:1$ to $q:n-1$ are the classes of the queuing discipline (represented in the figure above as yellow rectangles). Each of these classes can be selected using a filter (its elements are represented as green rectangles) attached to the queuing discipline; the packets being selected by the filter are placed in the respective DSMARK class.

What is the difference between DSMARK and a full class qdisc as HTB, for example? Basically the class behavior. A HTB class just control the upper rate of packets flowing through it while a DSMARK class just marks packets flowing through it.

Where, what and when does the DSMARK class mark packets? Packets are marked on the DS field. They are marked with an *integer value* that we define for each class when we configure the queuing discipline. Packets are marked just before they leave the queuing discipline to be placed on the outgoing network driver interface.

To create a new DSMARK queuing discipline we use this command:

```
# tc add qdisc dev eth0 handle <hd> root dsmark \  
indices <id> [ default_index <did> ] [ set_tc_index ]
```

cm-50

Yellows indicate values we have to supply. Let's see this in detail. This command sets our DSMARK as the root queue of the ethernet interface 0. $\langle hd \rangle$ is the handle number of the queue. $\langle id \rangle$ is the size of the internal table that defines the number of classes ($id-1$) contained in the queue. $\langle did \rangle$ is the default index; those packets that don't match any of the existing classes will be placed in the default class defined by this index. This parameter is optional. Final (optional) parameter is set_tc_index ; more about this will be explained below.

Some configuration commands are:

```
# tc qdisc add dev eth0 handle 1:0 root dsmark indices 32
```

cm-51

This command creates a DSMARK queuing discipline as root on ethernet interface eth0. The qdisc is numbered as 1:0 and contains a 32 elements table (from element 0 to element 31). Elements 1 to 31 are usable as classes 1:1 to 1:31.

```
# tc qdisc add dev eth0 parent 1:1 handle 2:0 dsmark \  
indices 8 default_index 7
```

cm-52

This command creates a DSMARK queuing discipline having class 1:1 (from other qdisc) as parent. The qdisc is numbered as 2:0 and contains a 8 elements table. Default index is number 7 which correspond to the dsmark class 2:7.

Okay, fine. But, how does DSMARK mark packets? To answer this question we need a schematic representation of the DSMARK internal table. Next picture helps to understand better this stuff:

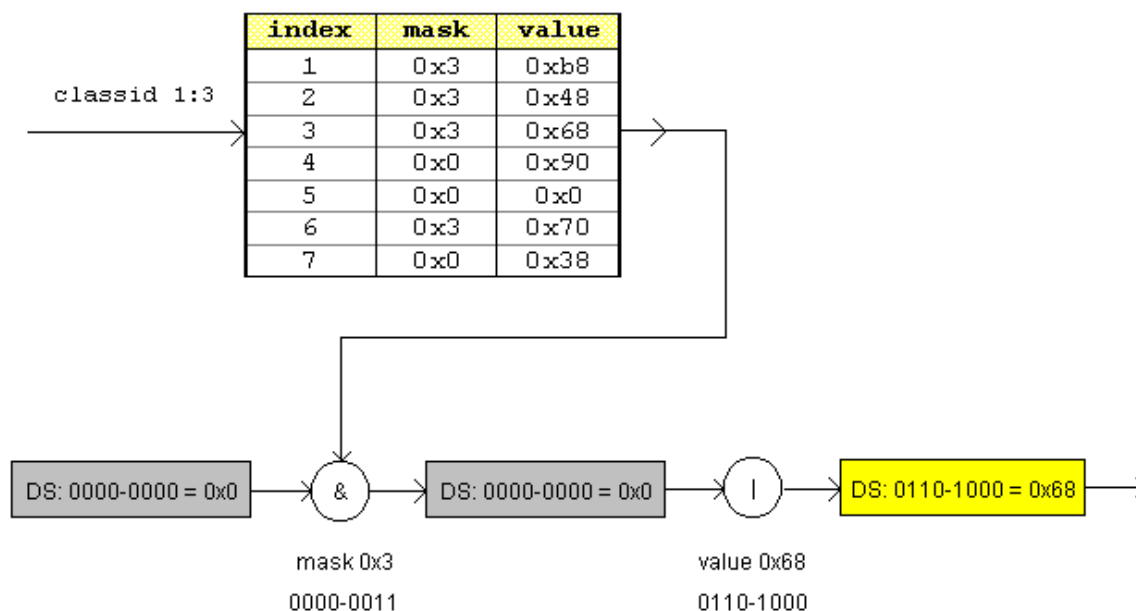


Figure 2.9.2

In this figure we enter by the left with a class number corresponding to an index value. In the example, the class number is 1:3 (index 3 assuming DSMARK as 1:0). The internal table has two columns called *mask* and *value* containing hexadecimal integer values. The integer values selected are: *mask* = 0x3; *value* = 0x68.

With these two values DSMARK goes to the packet, extract the DS field integer value and applies the following operation:

$$\text{New_DS} = (\text{Old_DS} \& \text{mask}) | \text{value}$$

where & and | are the bitwise *and* and *or* operators. The new calculated DS field value is placed back on the packet's DS field. In the figure a packet with the DS field set to 0x0 (best-effort DSCP) is entering. When DSMARK takes care the packet leaves the queue marked as DS field 0x68 corresponding to the DS Assure Forwarding class AF31. Observe that the packet enters as a common best-effort packet and leaves the queue with some money in its pocket; as a gentleman AF31 packet. Exactly what we wanted to do.

Observe that 0x3 *mask* preserves the two rightmost bits of the packet in case they (one or both of them) are set. This is a valid approach being the packet already marked to indicate some ECN condition. The 0x68 *value* bitwise *or* operation sets packet's DS field to 0x68.

What we have to learn now is how to fill our internal table, this means, how we make the relationship between *index (or classid)* values and pairs of *mask-value* values. But first let us present table 2.9.1 that helps a lot when dealing with those complicated DSCPs and DSs, even more because we have to deal with them in binary representation and hexadecimal representation.

CLASS	DP	DSCP	b-DSCP	x-DCSP	b-DS	x-DS
AF1	1	001010	0000-1010	0xa	0010-1000	0x28
	2	001100	0000-1100	0xc	0011-0000	0x30
	3	001110	0000-1110	0xe	0011-1000	0x38
AF2	1	010010	0001-0010	0x12	0100-1000	0x48
	2	010100	0001-0100	0x14	0101-0000	0x50
	3	010110	0001-0110	0x16	0101-1000	0x58
AF3	1	011010	0001-1010	0x1a	0110-1000	0x68
	2	011100	0001-1100	0x1c	0111-0000	0x70
	3	011110	0001-1110	0x1e	0111-1000	0x78
AF4	1	100010	0010-0010	0x22	1000-1000	0x88
	2	100100	0010-0100	0x24	1001-0000	0x90
	3	100110	0010-0110	0x26	1001-1000	0x98
EF		101110	0010-1110	0x2e	1011-1000	0xb8

Table 2.9.1

Great!! In the table we have DS classes AF1 to AF4 and EF. DP is the 'drop precedence'. From left to right we have the *DSCP* value defined by the DS specification; the binary *b-DSCP* (add two zeroes to the left of the DSCP value); the hexadecimal *x-DSCP* value; the binary *b-DS* field value (add two zeroes to the right of the DSCP value); and the hexadecimal *x-DS* field value.

With this table and the *mask* and *value* values we can make practically what we want. For example:

What you want to do

We would like to set any packet entering the classid as DS class AF23:

We would like to set any packet entering the classid as DS class AF12 but preserving the ECN bits:

We would like to change the 'drop precedence' to any DS AF packet entering the classid from what it has to 'drop precedence' 2; we want to preserve ECN bits. (0xe3 is 11100011; it preserves the first three class bits and the last two ECN bits. It sets to zero the 'drop precedence' bits. 0x10 is 00010000 which sets to 2 the 'drop precedence' bits).

We would like to change the class to any DS AF packet entering the classid from what it has to class AF3. The 'drop precedence' and ECN bits must be preserved. (0x1f is 00011111; it sets to zero the AF class bits and preserves the 'drop precedence'

DDoS Protection Leader

Renowned anti-DDoS service. Full SLA provided. Global network. www.prolexic.com

TDM Over IP Solutions

E1/T1/J1 Over IP, MPLS, PON Networks For Wi-MAX, Radio and Cell. Backhaul www.techaya.com

Field Service

Dependable, quality equipment installation services www.johnsonscreens.com

Field Data Collection

See How Wireless Solutions Improve Efficiency & Reliability. See More. www.honeywell.com/ps/wireless

MSMQ Queue Explorer

Manage MSMQ messages like files. Copy, delete, save, locally/remote! www.cogin.com

<u>mask</u>	<u>value</u>
0x0	0x58
0x3	0x30
0xe3	0x10
0x1f	0x60

and ECN bits. 0x60 is 01100000 which sets to 3 the AF class bits).

Okay fellows!! We are almost experts working with these things. Let's see now how to create the DSMARK table. Suppose we want to create the following table:

index	mask	value
1	0x0	0xb8
2	0x3	0x58
3	0xe3	0x10
4	0x1f	0x60
5	0x0	0x30
6	0x3	0x70
7	0x0	0x0

Table 2.9.2

Thinking a little to understand what this table does with the packets is left as an exercise. Configuring the queue is very easy. We will use an eight elements DSMARK table:

```
# tc qdisc add dev eth0 handle 1:0 root dsmark indices 8
# tc class change dev eth0 classid 1:1 dsmark mask 0x0 value 0xb8
# tc class change dev eth0 classid 1:2 dsmark mask 0x3 value 0x58
# tc class change dev eth0 classid 1:3 dsmark mask 0xe3 value 0x10
# tc class change dev eth0 classid 1:4 dsmark mask 0x1f value 0x60
# tc class change dev eth0 classid 1:5 dsmark mask 0x0 value 0x30
# tc class change dev eth0 classid 1:6 dsmark mask 0x3 value 0x70
# tc class change dev eth0 classid 1:7 dsmark mask 0x0 value 0x0
```

cm-53

There's really nothing to explain. First command creates the DSMARK queuing discipline. Next seven commands change each class (already created with the first command) to build our table.

Well, my friends, all this stuff is clear but we left behind how to place the packets entering the DSMARK into the different classes. We need a little help from a DSMARK attached filter. We will use the *u32* filter classifier assuming (to simplify things because we want just to show how to set the filter) that each DSMARK class from 1 to 7 correspond to one network ranging from 192.168.1/24 to 192.168.7/24 respectively. You can select more complicated filter elements to make your tests:

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \
    match ip src 192.168.1/24 flowid 1:1

# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \
    match ip src 192.168.2/24 flowid 1:2

# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \
    match ip src 192.168.3/24 flowid 1:3

# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \
    match ip src 192.168.4/24 flowid 1:4

# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \
    match ip src 192.168.5/24 flowid 1:5

# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \
    match ip src 192.168.6/24 flowid 1:6

# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \
    match ip src 192.168.7/24 flowid 1:7
```

cm-54

This filter, for example, places flows coming from network 192.168.1/24 on DSMARK class 1:1; packets from this network will leave DSMARK with their DS field value changed to 0xb8, which corresponds to DS class EF (Expedited Forwarding PHB).

DSMARK goes very nice. We know already how to set the discipline, how to create its classes and how to attach a filter for packet classifying. But, what about the `set_tc_index` parameter? To understand this we are going to steal next figure from *Differentiated Services on Linux* [10]:

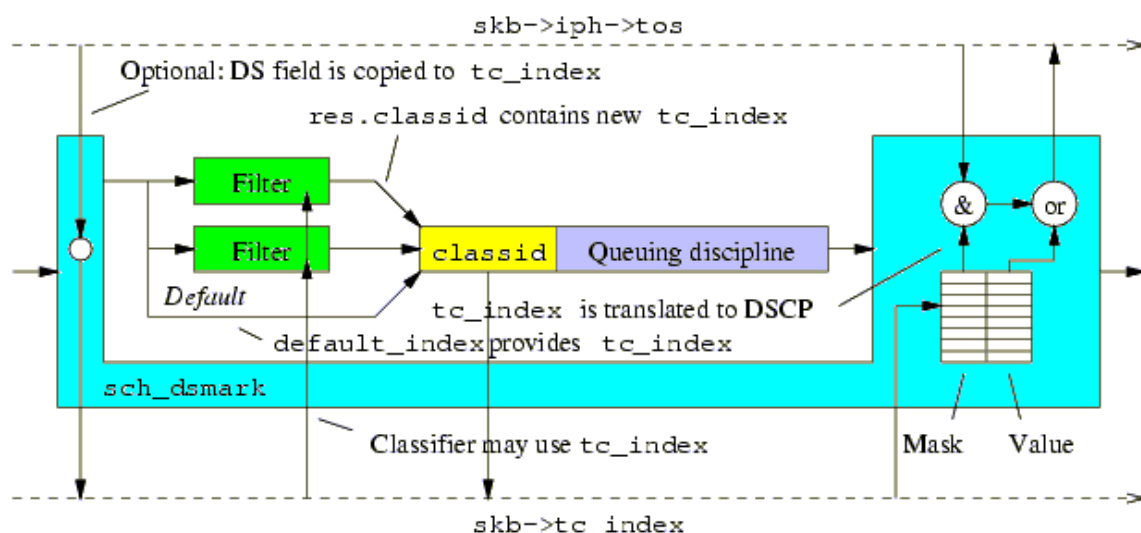


Figure 2.9.3

(Stolen from *Differentiated Services on Linux* [10])

This figure represents a DSMARK queuing discipline. Recall when we studied GRED that DS on Linux designers extend the packet buffer to add a new field called `tc_index`. The packet buffer is accessed using a

pointer called *skb*. Then, to access the new field *skb->tc_index* is used. This field is represented by the bottom dashed line in the figure.

The packet's buffer structure (*struct sk_buff*) contains the pointer *iph* that points to another structure (*struct iphdr*). This structure contains the field *tos* where the packet's TOS field is copied when the packet buffer is created. Then, to access the packet's TOS field *skb->iph->tos* is used. This field is represented by the top dashed line in the figure.

When you create a new DSMARK using the optional *set_tc_index* parameter, the queuing discipline will copy the packet's TOS field value (the DS field, in fact) contained in *skb->iph->tos* onto *skb->tc_index* on packet entrance. This is represented in the figure by the vertical line going from top to bottom, just in the DSMARK queue entrance. Perhaps, you should be asking: but, why?

The *skb->tc_index* field is a very important component of the Differentiated Service on Linux architecture. Recall, for example, that the 4-rightmost bits of this field are used to select a virtual RED queue where the packet is going to be placed when using the GRED queuing discipline.

But *skb->tc_index* usefulness doesn't end here. This field is also read by the special *tcindex* classifier to select a class identifier. This is shown in the figure by the next vertical line, this time going from bottom to top. The *tcindex* classifier reads the *skb->tc_index* value, could perform (or not) some bitwise operations on it, and use the final result to select a class identifier to the next inner queuing discipline. This is shown in the figure by the arrow going from the *tcindex* green filter element to the yellow classid of the inner queuing discipline.

When the *tcindex* classifier returns a class identifier value to the DSMARK queuing discipline as was explained above, the discipline (not the classifier, as was very intelligently pointed out to me for the german student Steffen Moser in a very pleasing e-mail information exchange), copies back this value again onto the *skb->tc_index* field. This is shown in the figure by the next (third) vertical line, this time going back from the yellow classid block to the *skb->tc_index* bottom field.

The *skb->tc_index* value is also used to select a class identifier (an index) to enter in the DSMARK internal table and get the *mask* and *value* values. The command *tc class change dev eth0 classid 1:1 dsmark mask 0x3 value 0x28*, for example, is in fact ordering: packets with its *skb->tc_index* field set to 1 (the minor value of the classid) must go to the index 1 internal DSMARK table register, and get from here the *mask* and *value* values. Next the packet's DS field value is read from the *skb->iph->tos* field, the bitwise *and* and *or* operation is applied, and the final value is placed back onto the *skb->iph->tos* field, where finally it will be copied back in the actual packet's DS field, just before it leaves the queuing discipline to be passed to the outgoing driver interface. This entire process is shown in the internal table schematic representation to the right of the DSMARK figure, above.

[Deep Packet Inspection](#)

A User-driven Community for DPI/DPP Learn, discuss, and collaborate
www.dPacket.org

[Track And Field](#)

Track and Field Coaching Secrets. 120 Proven Track and Field Drills.
www.TrackPracticePlans.com

[Paper: Mobile Backhaul](#)

Expanding 3G backhaul capacity without jeopardizing 2G revenues
www.ciena.com

[NSOM/SPM & Confocal](#)

Microscopes Integrated with Raman, SEM & Standard Optical Microscopes
www.nanonics.co.il

[Windows Packet Filtering](#)

Ndis Intermediate and Tdi Filter driver samples.
www.pcausa.com

I think (surely you share this mind, too) that *tcindex* classifier behavior understanding is so important for the Differentiated Service on Linux implementation, that I have reserved the next section to study this terrific and horrific monster. For now, it's over. See you later, alligator...



[Previous](#)

[Content](#)

[Next](#)

Google™

Search

The *tcindex* classifier bases its behavior in the *skb->tc_index* field located in the packet *sk_buff* buffer structure. This buffer space is created for every packet entering into or being created from the Linux box. Because of this, the *tcindex* classifier must be used only with those queuing disciplines that can recognize and set the *skb->tc_index* field; these are: GRED, DSMARK and INGRESS queuing disciplines.

I think it is easier to approach the *tcindex* classifier study by analyzing which qdisc/class/filter writes and which qdisc/class/filter reads the *skb->tc_index* field. Let's start by copying the figure 2.9.3 from the previous section, renumbered as 2.10.1 here, to be used as reference:

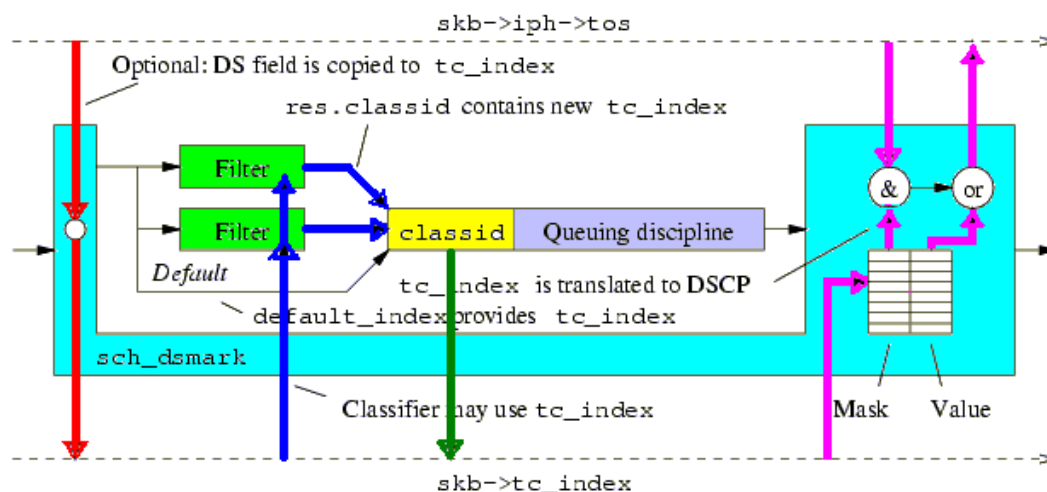


Figure 2.10.1

(Stolen from Differentiated Services on Linux [10])

Next we have to go to the C code (sorry, but it's better) to poke around. We will use this procedure:

1. We write one asseveration.
2. We present the C code that sustains it.
3. We make the reference with the figure above.
4. We present the tc command required to get that behavior.

Asseveration: The *skb->tc_index* value is set by the *dsmark* queuing discipline when the *set_tc_index* parameter is set. The *skb->iph->tos*, which contains the packet's DS field value, is copied onto the *skb->tc_index* field.

SIP Monitor-Analyzer

Call flow ladders, packet displays Linux & Windows support. Free trial!
www.vqmanager.com

End to End QoS and more

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

Network Traffic Control

Full control over network traffic. Download a 70-day free trial !
www.ldecoGateway.com

Pinnacle Oil Tools

Canadian Supplier of Oil
Well Tools Packers,
Retainers, Inflates, Plugs
www.PinnacleOilTools.ca

DDoS Protection Leader

Renowned anti-DDoS service. Full SLA provided. Global network.
www.prolexic.com

```

if (p->set_tc_index) {
    switch (skb->protocol) {
        case _constant_htons(ETH_P_IP):
            skb->tc_index = ipv4_get_dsfield(skb->nh.iph);
            break;
        case _constant_htons(ETH_P_IPV6):
            skb->tc_index = ipv6_get_dsfield(skb->nh.ipv6h);
            break;
        default:
            skb->tc_index = 0;
            break;
    };
}

```

C code taken from sch_dsmark.c file

Figure 2.10.2

In the figure, this process is represented by the big red vertical line going from top (*skb->iph->tos* field) to bottom (*skb->tc_index* field) in the *dsmark* entrance. As an example, the next tc command is used to get this behavior:

```
# tc qdisc add dev eth0 handle 1:0 root dsmark indices 32 \
    set_tc_index
```

cm-55

Asseveration: the *skb->tc_index* field value is read by the *tcindex* classifier; then, the filter applies a bitwise operation on a copy of the value (this means, the original value is not modified); the final value obtained from this operation is passed down to the filter elements to get a match. Having a match, the class identifier corresponding to this filter element is returned back and passed to the queuing discipline as the resulted class identifier.

```

result = TC_POLICE_OK; /* be nice to gcc */
if (TC_H_MAJ(skb->priority) == sch->handle) {
    skb->tc_index = TC_H_MIN(skb->priority);
} else {
    result = tc_classify(skb, p->filter_list, &res);
    D2PRINTK("result %d class 0x%04x\n", result, res.classid);
    switch (result) {
        #ifdef CONFIG_NET_CLS_POLICE
        case TC_POLICE_SHOT:
            kfree_skb(skb);
        }
    }
}

```

Here the tcindex classifier is invoked

C code taken from sch_dsmark.c file

```

static int tcindex_classify(struct sk_buff *skb, struct tcf_proto *tp,
                          struct tcf_result *res)
{
    struct tcindex_data *p = PRIV(tp);
    struct tcindex_filter_result *f;

    D2PRINTK("tcindex_classify(skb %p, tp %p, res %p), p %p\n", skb, tp, res, p);

    f = lookup(p, (skb->tc_index & p->mask) >> p->shift);
    if (!f) {
        if (!p->fall_through)
            return -1;
        res->classid = TC_H_MAKE(TC_H_MAJ(tp->q->handle),
                                (skb->tc_index & p->mask) >> p->shift);
        res->class = 0;
        D2PRINTK("alg 0x%x\n", res->classid);
        return 0;
    }
    *res = f->res;
    D2PRINTK("map 0x%x\n", res->classid);
}

```

Here a lookup is called

C code taken from cls_tcindex.c

Here the class id obtained is returned in res pointer

Figure 2.10.3

Okay, the classifier lookup is done by applying first the following bitwise operation to the *skb->tc_index* field:

$$(skb->tc_index \& p.mask) \gg p.shift$$

mask and *shift* are integer values we have to define in the main filter. Let's see a command to understand better this complicated part:

```

# tc qdisc add dev eth0 handle 1:0 root dsmark indices 64 \
  set_tc_index

# tc filter add dev eth0 parent 1:0 protocol ip prio 1 tcindex \
  mask 0xfc shift 2 pass_on

# tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
  handle 10 tcindex classid 1:111

# tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
  handle 12 tcindex classid 1:112

# tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
  handle 14 tcindex classid 1:113

```

The first command sets the DSMARK queuing discipline. Because *set_tc_index* is set, the packet's DS field value is copied onto the *skb->tc_index* field when the packet enters the qdisc. The second command is the main filter. For this example, it has three elements (next 3-commands). In the main filter we define *mask=0xfc* and *shift=2*. This filter reads the *skb->tc_index* value (containing the packet's DS field value), applies the bitwise operation, and the value resulted is passed down to the elements to look for a match. *pass_on* means: *if a match is not found in this element continue with the next one*.

Let's suppose that a packet having its DS field value marked as 0x30 (corresponding to the AF12 class) is entering the qdisc. The value 0x30 is copied by *dsmark* from the packet's DS field onto the *skb->tc_index* field. Next the classifier reads this field. It applies the bitwise operation to the value. What happens?

```
(skb->tc_index & p.mask) >> p.shift = (0x30 & 0xfc) >> 2 =
(00110000 & 11111100) = 00110000 >> 2 = 00001100 = 0xc
```

Final value after bitwise operation is 0xc which corresponds to the decimal value 12. This value is passed down to the filter elements. The first element doesn't match because it matches decimal value 10 (*handle 10 tcindex*).

Next element matches because it matches decimal value 12 (*handle 12 tcindex*); then, the class identifier returned back to the queuing discipline will be 1:112 (*classid 1:112*). In the figure, this process is represented by the big blue vertical line going from bottom (*skb->tc_index* field) to the green filter elements (to get a class identifier) and then from the green filter elements to the yellow class identifier returned back from the classifier to the queuing discipline. Now let's see what's going to do the *dsmark* queuing discipline with the class identifier value returned back.

Asseveration: the minor part value of the class identifier returned to the DSMARK queuing discipline by the *tcindex* classifier is copied back by the queuing discipline onto the *skb->tc_index* field.

```

    } else {
        result = tc_classify(skb,p->filter_list,&res);
        D2PRINTK("result %d class 0x%04x\n",result,res.classid);
        switch (result) {
            #ifdef CONFIG_NET_CLS_POLICE
                case TC_POLICE_SHOT:
                    kfree_skb(skb);
                    break;

            #if 0
                case TC_POLICE_RECLASSIFY:
                    /* FIXME: what to do here ??? */

            #endif
            #endif
                case TC_POLICE_OK:
                    skb->tc_index = TC_H_MIN(res.classid);
                    break;
                case TC_POLICE_UNSPEC:
                    /* fall through */
                default:
                    if (p->default_index != NO_DEFAULT_INDEX)
                        skb->tc_index = p->default_index;
        }
    }

```

Here the minor value (TC_H_MIN) of the class identifier is copied back onto *skb->tc_index* field

C code taken from *sch_dsmark.c* file

Figure 2.10.4

Well, fellows, finally the class identifier returned back again (it likes to travel, doesn't it?) to the *skb->tc_index* field. But, be careful. The value copied back was the class identifier's minor value, this means, 112 from the *classid 1:112*. It is very important to interpret well this ubiquitous value. 112 doesn't mean decimal value 112. Each of this numbers is a nibble (4-bits). Then 112 is really: 000100010010. It's better if we separe the nibbles, then the number is: 0001-0001-0010. This is now the new value contained in the *skb->tc_index* field.

In the figure, this process is represented by the big green vertical line going from yellow rectangle *classid* to the bottom *skb->tc_index* field.

Asseveration: On DSMARK queuing discipline the `skb->tc_index` value is used as an index to the internal table of *mask-value* pairs to get the pair to be used. Then, the pair selected is used, when dequeuing, to modify the packet's DS field value using a combined *and-or* bitwise operation.

We saw this already in previous section. Let's show first the C code:

```
static struct sk_buff *dsmark_dequeue(struct Qdisc *sch)
{
    struct dsmark_qdisc_data *p = PRIV(sch);
    struct sk_buff *skb;
    int index;

    D2PRINTK("dsmark_dequeue(sch %p,[qdisc %p])\n",sch,p);
    skb = p->q->ops->dequeue(p->q);
    if (!skb)
        return NULL;
    sch->q.len--;
    index = skb->tc_index & (p->indices-1);
    D2PRINTK("index %d->%d\n",skb->tc_index,index);
    switch (skb->protocol) {
        case constant_htons(ETH_P_IP):
            ipv4_change_dsfield(skb->nh.iph,
                               p->mask[index],p->value[index]);
            break;
```

Here the `skb->tc_index` value is copied to `index` after checking it doesn't go out of a valid indice

Here the packet's DS field value is changed using the mask and value pairs corresponding to the selected index. A bitwise operation is performed over the original value.

C code taken from `sch_dsmark.c` file

Figure 2.10.5

The commands above, are taken from the *afcbq* example of the Differentiated Service on Linux distribution (we will see every example on the DS on Linux distribution in detail, but later on). For now, to explaining this part, we will use a different set of commands; then we have:

```
# tc qdisc add dev eth0 handle 1:0 root dsmark indices 16 \
    set_tc_index
# tc class change dev eth0 classid 1:1 dsmark mask 0x1f value 0x40
# tc class change dev eth0 classid 1:2 dsmark mask 0x1f value 0x40
# tc class change dev eth0 classid 1:3 dsmark mask 0x1f value 0x40
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 tcindex \
    mask 0xfc shift 2 pass_on
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
    handle 10 tcindex classid 1:1
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
    handle 12 tcindex classid 1:2
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
    handle 14 tcindex classid 1:3
```

cm-57

This example is not as intelligent as it should be, but, for what we are trying to explain it is good enough. The first command sets the *dsmark* queuing discipline 1:0. Next 3-commands define the classes of the discipline. Now is the turn for the main filter. As we saw above, this filter reads the *skb->tc_index* field containing the packet's DS field value and after doing a bitwise operation on a copy of it, passes down the result obtained to the filter elements.

This commands are in fact changing the AF class of packets marked as AF1x to AF2x. This is what is called *re-marking* when talking on differentiated service terminology. The class is changed preserving the rest of the bits (drop precedence and ecn bits). When one AF1x's packet enters, its DS field is copied onto the *skb->tc_index* field by the *dsmark* queuing discipline, just because the *set_tc_index* parameter is set. Next the main filter is invoked. Let's suppose that one AF12 packet is entering. After the copy the new *skb->tc_index* value will be 0x30.

The main filter takes a copy of this value (0x30) and applies its bitwise operation with *mask=0xfc* and *shift=2*; then we have:

$$(0x30 \& 0xfc) \gg 2 = (00110000 \& 11111100) \gg 2 = 00110000 \gg 2 = 00001100 = 0xc = 12$$

Great!! Final value is decimal 12. This value is passed down to the filter elements. Second element matches and the class id value 1:2 is returned back to the *dsmark* queuing discipline. As we saw above, immediately *dsmark* strips the class id major value and copies back the minor value again onto the *skb->tc_index* field. The new value of *skb->tc_index* is now decimal 2.

Now is the turn again for *dsmark* queuing discipline when the packet is leaving out the discipline. The discipline reads the *skb->tc_index* field in the buffer's packet. The value is decimal 2. With this value it enters its own internal table. But this table was built for us with the 3 commands following the queuing discipline creation. Entering with decimal 2 index, the table contains the values *mask=0x1f* and *value=0x40*. The example is idiot because all classes have the same *mask-value* pair parameters. But, anyway, I'm tired and I don't want to think too much, just enough to explain how this stuff makes

[DDoS Protection Leader](#)

Renowned anti-DDoS service. Full SLA provided. Global network.
www.prolexic.com

[Linux Certification](#)

Linux Certification Training Now Available Online. Free Info Online!
www.CertificationTrainingOnline.com

[TDM Over IP Solutions](#)

E1/T1/J1 Over IP, MPLS, PON Networks For Wi-MAX, Radio and Cell. Backhaul
www.techaya.com

[Paper: Mobile Backhaul](#)

Expanding 3G backhaul capacity without jeopardizing 2G revenues
www.ciena.com

[MSMQ QueueExplorer](#)

Manage MSMQ messages like files. Copy, delete, save, locally/remote!
www.cogin.com

its work.

Finally the *dsmark* queuing discipline does the following operation over the AF12 marked packet.

$$(0x30 \& 0x1f) | 0x40 = (00110000 \& 00011111) | (01000000) = \\ (00010000 | 01000000) = 01010000 = 0x50$$

Okay, *0x50* is the value which corresponds to the class *AF22*. The packet enters as class-drop precedence *AF12* and departs as class-drop precedence *AF22*.

The really important thing to understand here is that *dsmark* reads the *skb->tc_index* value to select a class or an index into the internal table of *mask-value* pairs, for getting the pair to be used, later on, to update the DS field from the dequeuing packet. This entire process is represented by the big purple lines and arrows and the internal *dsmark* table representation to the right of the figure 2.10.1 above.

Asseveration: The 4-rightmost bits of the *skb->tc_index* field are used by the GRED queuing discipline to select a *red* virtual queue (VQ) for the packet entering the discipline. If the value (4-rightmost bits) is out of range of number of virtual queues, the *skb->tc_index* field is set (it shouldn't be) to the number of the default virtual queue by the GRED queuing discipline.

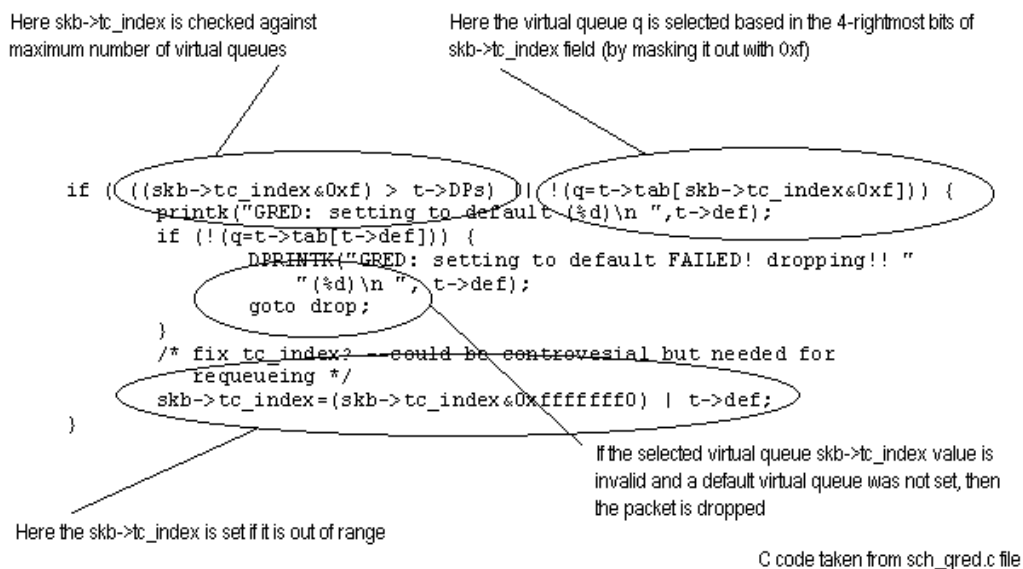


Figure 2.10.6

In this case, we don't have to put explicitly the packet into one virtual queue using a filter. It's good enough to set the *skb->tc_index* field value of the packet's buffer with the number of the virtual queue we want to select. For setting the *skb->tc_index* field we can use a *dsmark* qdisc and its attached filter, or we can use the *ingress* queuing discipline as it will be explained later on. Let's see how to set an example of this configuration:

```
# tc qdisc add dev eth0 handle 1:0 root dsmark indices 16 \
    set_tc_index

# tc filter add dev eth0 parent 1:0 protocol ip prio 1 tcindex \
    mask 0xfc shift 2 pass_on

# tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
    handle 10 tcindex classid 1:1

# tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
    handle 12 tcindex classid 1:2

# tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
    handle 14 tcindex classid 1:3

# tc qdisc add dev eth0 parent 1:0 gred setup DPs 3 default 3

# tc qdisc change dev eth0 parent 1:0 gred limit 3000 min 400 \
    max 800 burst 5 avpkt 128 bandwidth 10Mbit DP 1 \
    probability 0.01

# tc qdisc change dev eth0 parent 1:0 gred limit 6000 min 800 \
    max 1600 burst 12 avpkt 256 bandwidth 10Mbit DP 2 \
    probability 0.02

# tc qdisc change dev eth0 parent 1:0 gred limit 12000 min 1600 \
    max 3200 burst 20 avpkt 512 bandwidth 10Mbit DP 3 \
    probability 0.03
```

cm-58

These commands show a GRED configuration using DSMARK to select the virtual queue. The first command creates the *dsmark* queuing discipline. Packet's DS field will be copied onto the *skb->tc_index* field on entrance. Next command sets the main filter. Packets having DS field values corresponding to classes *AF11*, *AF12* and *AF13* will generate values *10*, *12* and *14* respectively, after the *(DS field & 0xfc) >> 2* bitwise operation is applied.

These values are passed down to the filter elements which are set using the next 3 commands. Class id *1:1*, *1:2* and *1:3* are returned back for classes *AF11*, *AF12* and *AF13* respectively. When the *dsmark* queuing discipline receives back the class id returned values, it sets the *skb->tc_index* with the minor values of them. This way, *skb->tc_index* is set to *1*, *2*, or *3* for packet's class *AF11*, *AF12*, or *AF13* respectively. It's great!! We have already set the *skb->tc_index* field for the *gred* queuing discipline.

Next command sets the main *gred* queuing discipline having as parent the *dsmark* queuing discipline. Last 3 commands set the *gred virtual queues* number *1*, *2* and *3* respectively. But, we don't have to worry about how to put packets into the *gred virtual queues*. GRED does itself its work by reading the *skb->tc_index* value and placing the packets into the corresponding virtual queues.

Our last asseveration: When using the INGRESS queuing discipline, *skb->tc_index* field is set with the minor part of the class identifier returned by the used attached filter.

```

static int ingress_enqueue(struct sk_buff *skb, struct Qdisc *sch)
{
    struct ingress_qdisc_data *p = PRIV(sch);
    struct tcf_result res;
    int result;

    D2PRINTK("ingress_enqueue(skb %p, sch %p, [qdisc %p])\n", skb, sch, p);
    result = tc_classify(skb, p->filter_list, &res);
    D2PRINTK("result %d class 0x%04x\n", result, res.classid);
    /*
     * Unlike normal "enqueue" functions, ingress_enqueue returns a
     * firewall FW_* code.
     */
    switch (result) {
#ifdef CONFIG_NET_CLS_POLICE
        case TC_POLICE_SHOT:
            result = NF_DROP;
            break;
        case TC_POLICE_RECLASSIFY: /* DSCP remarking here ? */
        case TC_POLICE_OK:
        case TC_POLICE_UNSPEC:
        default:
            result = NF_ACCEPT;
            break;
    }
#endif
    };

    skb->tc_index = TC_H_MIN(res.classid);
    return result;
}

```

Here the attached classifier is invoked to get a class id

C code taken from sch_ingress.c file

Here the minor value of the class id returned is copied onto skb->tc_index

Figure 2.10.7

The *ingress* queuing discipline is not a queue at all. It just invokes the attached classifier and when the class identifier is returned, it extracts the minor part from it and copies the result onto the *skb->tc_index* field.

The *ingress* qdisc's classifier could be a *u32* classifier or a *fw* classifier. The *tcindex* classifier cannot be used because it requires that the *skb->tc_index* field is set, and because the setting is done by the *ingress* queuing discipline itself, the initial *skb->tc_index* value will be zero. Excluding the *tcindex* classifier, I suppose we can use any kind of classifier to be attached to the *ingress* queuing discipline. Being *u32* or *fw* the used classifier, in both cases you can police the flows entering at the same time by implementing a policer into the classifier. Because this is specially important for the Differentiated Service architecture, we are going to explain a little more about policing in the next section. For now, we are going to show two examples using the *fw* classifier and the *u32* classifier.

```
# iptables -t mangle -A FORWARD -i eth1 -s 0/0 \  
-j MARK --set-mark 2  
  
# iptables -t mangle -A FORWARD -i eth1 -s 10.2.0.0/24 \  
-j MARK --set-mark 1  
  
# tc qdisc add dev eth1 handle ffff: ingress  
  
# tc filter add dev eth1 parent ffff: protocol ip prio 1 \  
handle 1 fw flowid :1  
  
# tc filter add dev eth1 parent ffff: protocol ip prio 1 \  
handle 2 fw flowid :2  
  
# tc qdisc add dev eth0 handle 1:0 root dsmark indices 16  
  
# tc class change dev eth0 classid 1:1 dsmark \  
mask 0x3 value 0x88  
  
# tc class change dev eth0 classid 1:2 dsmark \  
mask 0x3 value 0x90
```

cm-59

In this example we use the *fw* classifier. Traffic enters through the *eth1* interface and leaves the router through the *eth0* interface. The *ingress* queuing discipline is configured on interface *eth1*. On this interface we, previously, set *iptables* to mark any flow entering with *fw mark*=2, and then flows from network 10.2.0.0/24 with *fw mark*=1.

Using two filter elements we set *skb->tc_index* field with the value 1 (*flowid :1*) for packets with *fw* set to 1 (*handle 1 fw*), and with the value 2 (*flowid :2*) for packets with *fw* set to 2 (*handle 2 fw*).

Finally we configure a *dsmark* queuing discipline on outgoing interface *eth0*. Packets leaving the router with its *skb->tc_index* field set to 1 (*classid 1:1*) are marked on its *DS field* applying the bitwise operation $((DS \& mask) | value)$. Then, packets from network 10.2.0.0/24 (identified with *skb->tc_index*=1) are marked as 0x88 (which corresponds to *DS class AF41*), and rest of traffic (identified with *skb->tc_index*=2), is marked as 0x90 (which corresponds to *DS class AF42*).

The *u32* classifier is used in a similar way; but we don't need *iptables* for this case. For example:

```
#tc qdisc add dev eth1 handle ffff: ingress
#tc filter add dev eth1 parent ffff: protocol ip prio 1 u32 \
    match ip tos 0x28 0xfc flowid :1
#tc filter add dev eth1 parent ffff: protocol ip prio 1 u32 \
    match ip tos 0x30 0xfc flowid :2
#tc qdisc add dev eth0 handle 1:0 root dsmark indices 16
#tc class change dev eth0 classid 1:1 dsmark \
    mask 0x3 value 0xb8
#tc class change dev eth0 classid 1:2 dsmark \
    mask 0x3 value 0x28
```

cm-60

As you see this configuration is even simpler than when using the *fw* classifier. We configure the *ingress* queuing discipline, then using two filter elements attached to it, we set *skb->tc_index* field with the value 1 (*flowid :1*) for packets with *DS* field set to 0x28 (*match ip tos 0x28 0xfc*), preserving the *ecn* bits, and with the value 2 (*flowid :2*) for packets with *DS* field set to 0x30 (*match ip tos 0x30 0xfc*), again preserving the *ecn* bits. These packets happen to be the differentiated service classes *AF11* and *AF12*, respectively.

Our setting is some kind of "promoting packets" configuration. The *dsmark* queuing discipline marks packets leaving the router with its *skb->tc_index* field set to 1 (*classid 1:1*), i.e., *AF11*'s class packets, as 0xb8 (which corresponds to *DS* class *EF*), and packets leaving the router with its *skb->tc_index* field set to 2 (*classid 1:2*), i.e., *AF12*'s class packets, as 0x28 (which corresponds to *DS* class *AF11*). Then *DS AF11*'s class packets are promoted to *DS EF* and *DS AF12*'s class packets are promoted to *DS AF11*.

Well, fellows. With this explanation we finish the *TCINDEX* classifier. Next section will be dedicated to explore a little about the filter's police capability.

[Network Packet Analyzer](#)

Get complete real-time analysis across your entire network.

www.Paessler.com/Packet-Sniffing

[Paper: Mobile Backhaul](#)

Expanding 3G backhaul capacity without jeopardizing 2G revenues

www.ciena.com

[Virtual Machine](#)

Run any PC OS - Windows, Linux - on Mac OS X. Love your Mac -Free Trial

www.Parallels.com

[Eclipse UML - Free Trial](#)

Visual designer, Roundtrip Engine. Direct edit, Export as XMI & More

www.myeclipseide.com

[JAVA Programmers needed](#)

Join GetAFreelancer.com and bid on projects. Free and quick signup.

www.GetAFreelancer.com



This site is visited regularly for very specialized people; it's then the best place to advertise your very specialized product. First month at no cost for you to verify this aseveration.

[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

2.11.- Policing

Policing flows before entering the domain is a very important matter when dealing with Differentiated Service architecture. In fact, the specifications employ a lot of their contents to insist about these features. Let's start by recalling the policing definition taken from the *RFC 2475 [2]* specification:

Policing: the process of discarding packets (by a dropper) within a traffic stream in accordance with the state of a corresponding meter enforcing a traffic profile.

Dropper: a device that performs dropping.

Dropping: the process of discarding packets based on specified rules; policing.

Well, Policing → Dropper → Dropping → Policing, again. We have a loop here, haven't we? Anyway, the explanation is clear. Using a dropper (some kind of packet's killer), we have to discard packets to enforce a traffic profile.

The policing process is better done at ingress nodes of the domain. Why? Because it doesn't have any sense to permit packets violating a traffic profile to enter the domain, because sooner than later, the profile must be respected, then the offending packets should be dropped, but now after they consume a share of our valuable resources; then, why don't kill these 'bad citizen' packets before entering our domain? This is the idea behind the policing process.

What do we need to implement a policer? First of all, we should follow the recommendation above, this means, to place our policer on ingress edge router's domain. Avoid brute flows from entering the domain by policing them before they enter. Avoid consuming ingress edge router's resources by advancing the policing process, as soon as is possible to do that.

Beginning with this principle, we need also some way to classify incoming packets to place them in behavior aggregates (do not forget that this is a basic differentiated service architecture principle). Having them in BA, we can check next our *traffic profile rules* to know what to do when: a.- they respect the rules, i.e., they are *in-profile*. b.- They do not respect the rules, i.e., they are *out-of-profile*.

How do we know if they respect or not the rules? We need some kind of metering device. Finally, when we are obligated to apply the law, we need also some kind of dropper device, to drop the offending packets.

[EXCELIANCE Load Balancer](#)

Appli. Load balancing & High avail. download free evaluation soft.
www.Software-Appliance.com

[Network Traffic Control](#)

Full control over network traffic. Download a 70-day free trial !
www.IdecoGateway.com

[network traffic](#)

Analyze your firewall logs for b/w, traffic, user activity, & more!
www.FWAnalyzer.com

[LED Traffic Signal Lights](#)

BBE LED Traffic Signal Lights LED Bulbs, Street Lights Factory
www.bbeled.com

[Network/Ethernet Tester](#)

Tester for Switch & SOHO Router Network TAP / Power Monitor
www.xtramus.com

Linux has one *all-in-one* solution for all these requirements we have. It's just a classifier with policing. To follow the first recommendation above, we are going to implement it on the INGRESS queuing discipline (well, just for doing this work this pet was created, wasn't it?). *Ingress* will be implemented in the incoming interface on any domain edge router. *Ingress*, as was said before, doesn't enqueue packets at all, then we are for sure that resource consumption is minimum. The attached policer filter will be in charge of:

- Classify entering packets to put them in behavior aggregates (classes).
- Apply policing rules we are going to define to fulfill the service level agreement.
- Drop those packets that violate the rules.

To do all this work the edge router is going to consume resources, of course, but when remaining packets are forwarded to the outgoing interface to enter the domain, the flows are already clean, and the resource consumption in the outgoing interface queuing discipline and in the inner domain's core routers will be exactly what we need (no more) to manage the traffic we are committed to handle according to the TCAs.

What kind of classifier will we use to implement policing? Just the same we used before to explain the *ingress* queuing behavior when dealing with the *skb->tc_index* field (have a look to previous section). The *fw* classifier and the *u32* classifier. Let's start by presenting one example taken (partially) from the Differentiated Service on Linux distribution:

```
# iptables -t mangle -A FORWARD -i eth1 -s 0/0 \
-j MARK --set-mark 2

# iptables -t mangle -A FORWARD -i eth1 -s 10.2.0.0/24 \
-j MARK --set-mark 1

# tc qdisc add dev eth1 handle ffff: ingress

# tc filter add dev eth1 parent ffff: protocol ip prio 4 \
  handle 1 fw police rate 1500kbit burst 90k \
  continue flowid 4:1

# tc filter add dev eth1 parent ffff: protocol ip prio 5 \
  handle 1 fw police rate 1500kbit burst 90k \
  continue flowid 4:2

# tc filter add dev eth1 parent ffff: protocol ip prio 6 \
  handle 1 fw police rate 1000kbit burst 60k \
  drop flowid 4:3

# tc filter add dev eth1 parent ffff: protocol ip prio 6 \
  handle 2 fw police rate 1000kbit burst 60k \
  drop flowid 4:4
```

cm-61

This example is implemented using the *fw* classifier. First three command are known for us. *iptables* is used to mark entering packets according to their *source ip address*. Next four commands implement the *fw* classifier with policing. Let's concentrate our attention in these commands.

What are we expecting from the classifier? According to what was explained in this section, it should:

Classify entering packets to put them in behavior aggregates (classes)

This is done using *iptables* which is the tool in charge to classifying packets for this case. After doing the classification, it just marks the *fw* field in the packet's buffer to signaling the result of this. Above, in the example, packets from network *10.2/24* are marked as *fw 1*. Rest of packets as *fw 2*. Perhaps, we could think that this is a very coarse classification. But it's just a matter to use all the power behind *iptables*. We can rewrite the commands as we like to include on them the kind of classification we want. Final result will be that packets will pass down to the filter elements previously **fw** marked.

But, we are expecting flow aggregation too. This is done using the filter elements. Have a look above to the commands. Packets marked as *fw 1* (from network *10.2/24*) are placed in three different classes. Did you catch it? *flowid 4:1*, *flowid 4:2* and *flowid 4:3* are, in fact, three different classes of the same traffic, *where the aggregation is done depending on actual throughput and bursty conditions of this* (more about this below).

If you revise back you should remember that these instructions set the packet *skb->tc_index* field as *1*, *2* or *3* respectively. Having the *skb->tc_index* field set, is just a matter of using *dsmark* queuing discipline to mark their *DS field*, or using *gred* queuing discipline to place them in different *red virtual queues* for differentiated dropping precedence treatment. Rest of packets are all placed on class *4 (flowid 4:4)*.

Apply policing rules we are going to define to fulfill the service level agreement

Keywords for policing are as follows:

- **police rate <rate>** : defines the maximum rate (throughput) admitted for this type of traffic.
- **burst <burst>** : defines the maximum burst admitted for this type of traffic.
- **continue** : this means, packets violating this rule must be passed to the next police rule (next filter element).
- **drop** : this means, packets violating this rule must be dropped.

For example, have a look above to the *first police rule*. It says:

Traffic marked by *iptables* as *fw 1 (handle 1 fw)* will be admitted and aggregated to class number 1 (*flowid 4:1*) up to a maximum rate of *1500kbps* with a maximum burst of *90KB*. Traffic that *exceeds* this setting *must be passed* to the next police rule (*continue*).

Second police rule says:

Traffic that is passed to this rule is those that: a.- *does not match the first police rule*, or, b.- *has matched but exceeds the first police rule*. Over this traffic the second rule is applied as follows: traffic marked by *iptables* as *fw 1 (handle 1 fw)* will be admitted and aggregated to class number 2 (*flowid 4:2*) up to a maximum rate of *1500kbps* with a maximum burst of *90KB*. Traffic that *exceeds* this setting *must be passed* to the next police rule (*continue*).

Third police rule says:

Traffic that is passed to this rule is those that: a.- *does not match the second police rule*, or, b.- *has matched but exceeds the second police rule*. Over this traffic the third rule is applied as follows: traffic marked by *iptables* as *fw 1 (handle 1 fw)* will be admitted and aggregated to class number 3 (*flowid 4:3*) up to a maximum rate of *1000kbps* with a maximum burst of *60KB*. Traffic that *exceeds* this setting *must be dropped (drop)*.

Fourth police rule says:

Traffic that is passed to this rule is those that *does not match the third police rule* (neither the first and second, of course). We can't have traffic that match the three previous rules here (i.e., *handle 1 fw*), because all this traffic was dropped by the third rule. Traffic we have here is those that match the condition *fw 2 (handle 2 fw)*. Over this traffic the fourth rule is applied as follows: traffic marked by *iptables* as *fw 2 (handle 2 fw)* will be admitted and aggregated to class number 4 (*flowid 4:4*) up to a maximum rate of *1000kbps* with a maximum burst of *60KB*. Traffic that *exceeds* this setting *must be dropped (drop)*.

[LED Traffic Signal Lights](#)

BBE LED Traffic Signal
Lights LED Bulbs, Street
Lights Factory
www.bbeled.com

[Free Packet Sniffer](#)

Packet Sniffer - It's Free.
Very advanced networking
tool.
www.Paessler.com

[Voice Compression Expert](#)

Dedicated to Voice
Compression Up to 20:1
Compression Rate
www.broadgalaxy.com

[Simulation Modeling Tools](#)

Learn the benefit of
ExtendSim simulation
tools. Free Demo!
www.extendsim.com

[XRate! SNMP OPC Server](#)

Real time Network
Monitoring for OPC or
DDE clients. Free Trial.
www.XRate!.com/snmp.asp

Resuming, our service level agreement is easy: we admit traffic from network 10.2/24 up to a maximum of 4000kbps, but in three scales. First scale admits traffic from 0 up to 1500kbps, with a maximum burst of 90KB; second scale admits traffic above 1500kbps, but up to a maximum of 1500kbps more, with a maximum burst of 90KB; third scale admits traffic above 3000kbps, but up to a maximum of 1000kbps more, with a maximum burst of 60KB. Traffic from network 10.2/24 exceeding this profile is dropped. From any other network, we admit traffic from 0 up to 1000kbps, with a maximum burst of 60KB; no scales are implemented in this case. Traffic from other networks and exceeding this profile will be dropped.

You should be asking. But, what's the different between these three network 10.2/24's scales? From the point of view of the sending network, perhaps they are equal, but not from our point of view. Different network 10.2/24's scale traffic will be different treated in our domain. To do this we aggregate them using three different classes. They will receive different treatment depending of how we implement our router's queuing discipline to forward these classes within our domain.

There are two additional points that should be observed before we finish this part and jump to the next example. First is *prio*. Have a look to the *prio* keyword above. Filter elements are treated according to its *prio* number. In the example above, *prio 4* is treated first, then *prio 5*, *prio 6*, and so on. This mechanism guarantees how our filter elements will be searched by the queuing discipline to match entering packets.

Second is *burst*. Be careful with this. Bursting here is not synonymous of buffering. There's no any packet buffering here because *ingress* queuing discipline doesn't enqueue packets. Policing filters act using a *token bucket* to control bandwidth. You can learn a little more about this by reading the section [TBF queuing discipline](#). The same principle is applied here. When you define in this context '*police rate 1500kbit burst 90k*', what you are saying is: when throughput is less than *1500kbit*, tokens are saved (because they are injected into the bucket to a *1500kbit* rate and retired to a lower rate) to be used later, when throughput increases and this saving is needed. Then, *burst 90k* means that you can save up to a maximum of 90KB of equivalent tokens. This setting permits to deal with bursty flows, but at the same time controlling the maximum permitted burstiness.

Our second example uses the *u32* classifier instead of the *fw* one. This time the configuration is easier because we don't need *iptables* and the classifier itself is in charge of doing all the work, i.e., classifying and policing entering packets. Let's see how this goes:

```
# tc qdisc add dev eth1 handle ffff: ingress

# tc filter add dev eth1 parent ffff: protocol ip prio 4 u32 \
    match ip src 10.2.0.0/24 police rate 1500kbit burst 90k \
    continue flowid 4:1

# tc filter add dev eth1 parent ffff: protocol ip prio 5 u32 \
    match ip src 10.2.0.0/24 police rate 1500kbit burst 90k \
    continue flowid 4:2

# tc filter add dev eth1 parent ffff: protocol ip prio 6 u32 \
    match ip src 10.2.0.0/24 police rate 1000kbit burst 60k \
    drop flowid 4:3

# tc filter add dev eth1 parent ffff: protocol ip prio 6 u32 \
    match ip src 0/0 police rate 1000kbit burst 60k \
    drop flowid 4:4
```

cm-62

This configuration fulfills the same requirement of the previous one, but using the *u32* classifier. The *u32* classifier's classification capability can be used to aggregate flows previously marked from another differentiated service capable domain. To finish this section, let's show another example where we take advantage of this capability to remark entering packets:

```
# tc qdisc add dev eth1 handle ffff: ingress

# tc filter add dev eth1 parent ffff: protocol ip prio 4 u32 \
    match ip tos 0x88 0xfc police rate 1500kbit burst 90k \
    continue flowid 4:1

# tc filter add dev eth1 parent ffff: protocol ip prio 5 u32 \
    match ip tos 0x88 0xfc police rate 1500kbit burst 90k \
    continue flowid 4:2

# tc filter add dev eth1 parent ffff: protocol ip prio 6 u32 \
    match ip tos 0x88 0xfc police rate 1000kbit burst 60k \
    drop flowid 4:3

# tc filter add dev eth1 parent ffff: protocol ip prio 6 u32 \
    match ip tos 0x90 0xfc police rate 1000kbit burst 60k \
    drop flowid 4:4
```

cm-63

Matching by the packet's *tos* field we aggregate flows entering our domain marked as *DS class AF41 (tos 0x88)* into three classes according to the throughput and bursting condition. When they are already into, we can remark them using a *dsmark* queuing discipline (not showed here) that reads the packet *skb->tc_index* field.

[TransModeler](#)

The world's most powerful traffic simulation software
www.caliper.com/transmodeler

[Deep Packet Inspection](#)

A User-driven Community for DPI/DPP Learn, discuss, and collaborate
www.dPacket.org

[Speed Bumps](#)

SNG Company Traffic and Roadway Safety Products
www.sng.co.yu

[Rotapanel International](#)

Prismatic Variable Message Signs for traffic, Worldwide
www.rotapaneltraffic.com

[Alarm Panel,Solar Traffic](#)

Fire Alarm Bell, Call Point Solar Road Stud,Solar traffic light
www.multice.com

Same for packets marked as *DS class AF42 (tos 0x90)*.

With this example we are ready with this section. Next we are going to study in detail the examples included in the Differentiated Service On Linux distribution.



[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

3.0. Differentiated Service on Linux distribution

When I started to study the Differentiated Service on Linux distribution (included in the iproute2 distribution), it was really funny because, trying to understand what they were talking about, I searched a lot looking for better information. Finally, after attempting several times, I was tired and surprised that, perhaps with some words more or some words less, every documentation available finished with this advice: *I (we) suggest you to have a look to the Differentiated Service on Linux distribution's examples where more information can be found..., etc., etc.*

Of course, having such a look doesn't help to much either because too many questions remain unanswered. And, at the present time, some of them stay. That's the problem with Linux. It would be great if people who make Linux, could make also a little effort to improve the available documentation. One lose too much time trying to figure out what they were trying to do and how. Precious time that could be better utilized learning new paradigms. Well, nothing can be perfect. But, no one can prohibit me to say that this is exactly, one of the reasons why Linux use doesn't propagate faster. Too many people desert after breaking their brains trying to understand Linux. Finally, they joint the anti-Linux group telling that the system is for gurus and just limited for experimental implementations. On these circumstances, some known commercial companies take advantages to sell, very expensive indeed, some other operating systems that are technologically several years behind Linux. Really, a pity.

Well, we are not here to moan, but to learn Linux. In this section we are going to study each one of the examples included in the Differentiated Service on Linux distribution, with one exception, that the *afcbq* example will be replaced to use the *HTB* queuing discipline instead of the *CBQ* queuing discipline.

The examples in the distribution are:

1. [Edge1](#)
2. [Edge2](#)
3. [Edge31-ca-u32](#)
4. [Edge32-ca-u32](#)

[WAN Traffic Shaping Tool](#)

Free Guide: Optimize Performance of Critical Apps w/ Traffic Shaping.
[Packeteer.com](#)

[Free Utility](#)

Get Disk Performance Analyzer for Networks now!
[www.Diskeeper.com](#)

[Easy Scripts](#)

We have many easy scripts Install them in less then 5 min
[easy-scripts.net](#)

[Linux Networking](#)

Get the latest news, tutorials, white papers, and much more.
[Networking.ITtoolbox.com](#)

[SIP call troubleshooter](#)

Real-time VoIP Traffic monitoring Free Trial & Support. Download now!
[www.vqmanager.com](#)

5. [Edge31-cb-chains](#) (using *iptables*)
6. [Edge32-cb-chains](#) (using *iptables*)
7. [Edge32-cb-u32](#)
8. [afcbq](#) (using *htb*)
9. [ef-prio](#)
10. [efcbq](#) (using *htb*)

As you can figure out, *Edge** examples are for implementing edge routers. *afcbq* implements differentiated service *Assure Forwarding PHB* using *CBQ*, *DSMARK* and *GRED*. This example will be presented but replacing *CBQ* for the new *HTB* queuing discipline. *ef-prio* implements differentiated service *Expedited Forwarding PHB* using *PRIO*, *TBF*, *DSMARK* and *RED*. *efcbq* implements differentiated service *Expedited Forwarding PHB* using *CBQ*, *DSMARK* and *RED*. Again, this example will be presented but replacing *CBQ* for *HTB*.

Edge??-??-u32 examples use the *u32* classifier to classify packets, but *Edge-??-??-chains* examples use *ipchains* as the classifying tool. However, we are not going to use *ipchains* but instead *iptables*. *iptables* replaced *ipchains* some time ago and it is now the tool used to manage the Linux firewall. Another thing is that the examples were originally presented using *sh (bash)* or *perl* scripts. To help the explanation we will expand these scripts to get the final commands executed by the shell executor.

Well, let's start with *Edge-1*, kind lectors.



[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

3.1. Edge1

 Search

The example's script is as follows:

```
#!/bin/bash

##### Ingress side #####

iptables -t mangle -A FORWARD -i eth2 -s 10.2.0.0/24 -j MARK --set-mark 3
iptables -t mangle -A FORWARD -i eth2 -s 10.2.0.24 -j MARK --set-mark 1
iptables -t mangle -A FORWARD -i eth2 -s 10.2.0.3 -j MARK --set-mark 2

##### Egress side #####

tc qdisc add dev eth1 handle 1:0 root dsmark indices 64 set_tc_index

tc class change dev eth1 classid 1:1 dsmark mask 0x3 value 0xb8
tc class change dev eth1 classid 1:2 dsmark mask 0x3 value 0x28
tc class change dev eth1 classid 1:3 dsmark mask 0x3 value 0x48

tc filter add dev eth1 parent 1:0 protocol ip prio 4 handle 1 fw classid 1:1
tc filter add dev eth1 parent 1:0 protocol ip prio 4 handle 2 fw classid 1:2
tc filter add dev eth1 parent 1:0 protocol ip prio 4 handle 3 fw classid 1:3
```

Here we are trying to implement a very simple DS edge router. The interface *eth2* faces to outside of the domain. The interface *eth1* faces to inside of the domain.

First thing to be noted here is the use of *iptables* instead of *ipchains*. In the original example they use *ipchains -A input*, but in our implementation we use *iptables -A FORWARD* because we are trying to implement a router. Packets, then, are not going to enter the router to the upper layers, but instead they will be forwarded directly from interface *eth2* to interface *eth1*. In the FORWARD's *iptables* chain, every packet coming from the network selected will be marked (not the packet itself, but the *fw* field in the packet's buffer) as *1*, *2* or *3* to indicate its origin. *iptables* rules are treated in the same sequence as they were included. Then, first command marks all packets from network *10.2/24* as *fw 3*; second command picks just packets from host *10.2.0.24* and mark them as *fw 1*; and third command picks just packets from host *10.2.0.3* and mark them as *fw 2*.

In the *eth1* output interface a *DSMARK* queuing discipline is configured. When a packet enters this discipline its *DS* field value is copied onto the *skb->tc_index* field. This is not really necessary in this case because the final class selection will be done using the *fw* classifier, not the *tcindex* classifier. This example then goes fine by omitting *set_tc_index* in the *DSMARK*'s command.

To continue, the *fw* classifier is invoked now. This classifier reads the *fw* field on the packet's buffer and depending of its value (*1*, *2*, or *3*), returns classes *1*, *2*, or *3* to the *dsmark* queuing discipline respectively, which in turn sets the *skb->tc_index* value as *1*, *2* or *3* respectively. This because according to the filter commands above, the *fw* value was set the same as the minor part of the *classid* value.

Let's explain this very clearly, the command: *...handle 1 fw classid 1:1*, returns the *classid 1:1*, which *dsmark* uses to set the *skb->tc_index* field as *1* for packets whose *fw* field value is *1*. But, we could have written: *...handle 1 fw classid 1:2*, and then the *classid 1:2* will be returned, which *dsmark* will use to set the *skb->tc_index* field as *2* for packets whose *fw* field value is *1*. Be very careful with this.

Well, being here our assignment has been fulfilled; i.e., *skb->tc_index* field value is set according to the packet's source network. This means, any packet from host *10.2.0.24* has its *skb->tc_index* field set to *1*, any packet from host *10.2.0.3* has its *skb->tc_index* field set to *2*, and rest of packets from network *10.2/24* have their *skb->tc_index* field set to *3*.

[QoS Management Tools](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[Network/Ethernet Tester](#)

Tester for Switch & SOHO Router Network TAP / Power Monitor
www.xtramus.com

[WAN News](#)

Networking Webcasts, White Papers Trends & Breaking News. Get Info!
NetworkWorld.com

[Network Packet Analyzer](#)

Advanced Network Analyzer Solution. Start Analyzing Today. Download!
www.Paessler.com/Packet-Sniffi

[Configure DNS](#)

IP Address Management Simple & Secure DNS/DHCP
www.BlueCatNetworks.com

Final work is done by the *DSMARK* queuing discipline classes. Packets having marked their *skb->tc_index* field as 1, 2, or 3 will be placed in the corresponding *dsmark* classes 1:1, 1:2, or 1:3 respectively. Being in the classes, when they leave the *dsmark* queuing discipline, their *DS* field will be set to 0xb8, 0x28, or 0x48, respectively (but preserving the ecn bits), which will correspond to differentiated service classes *EF*, *AF11* and *AF21*, again, respectively.

This edge router then assigns differentiated service classes as follows:

Traffic coming from *host 10.2.0.24* will be aggregated to differentiated service *EF class*.

Traffic coming from *host 10.2.0.3* will be aggregated to differentiated service *AF11 class*.

Rest of traffic coming from *network 10.2/24* will be aggregated to differentiated service *AF21 class*.

Packets from these traffics will start their travel through our domain already marked as is indicated above. That's all folks, at least for this simple and funny example.



[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

3.2. Edge2

The example's script is as follows:

```
#!/bin/bash

##### Ingress side #####

iptables -t mangle -A INPUT -i eth2 -s 10.2.0.0/24 -j MARK --set-mark 3
iptables -t mangle -A INPUT -i eth2 -s 10.2.0.24 -j MARK --set-mark 1
iptables -t mangle -A INPUT -i eth2 -s 10.2.0.3 -j MARK --set-mark 2

tc qdisc add dev eth2 handle ffff: ingress

tc filter add dev eth2 parent ffff: protocol ip prio 50 handle 3 fw \
police rate 1500kbit burst 90k mtu 9k drop flowid :1

##### Egress side #####

tc qdisc add dev eth1 handle 1:0 root dsmark indices 64

tc class change dev eth1 classid 1:1 dsmark mask 0x3 value 0xb8
tc class change dev eth1 classid 1:2 dsmark mask 0x3 value 0x28
tc class change dev eth1 classid 1:3 dsmark mask 0x3 value 0x48

tc filter add dev eth1 parent 1:0 protocol ip prio 4 handle 1 fw classid 1:1
tc filter add dev eth1 parent 1:0 protocol ip prio 4 handle 2 fw classid 1:2
tc filter add dev eth1 parent 1:0 protocol ip prio 4 handle 3 fw classid 1:3
```

This example is a slight variation from the previous one. This time the core router is configured to police some traffic entering the domain. First three commands are similar than before using *iptables* to mark the *fw* field of the entering packets.

To continue, the differences begin, because an *ingress* queuing discipline is implemented to police traffic using a single element filter. This policer, matching *fw 3* packets, controls traffic from network 10.2/24 but excluding hosts 10.2.0.24 and 10.2.0.3. This because traffic from these hosts is marked as *fw 1* and *fw 2* respectively. The police protects our domain not permitting that traffic beyond 1500kbps enter from this

[EXCELIANCE Load Balancer](#)

Appli. Load balancing & High avail. download free evaluation soft.
www.Software-Appliance.com

[End to End QoS and more](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[Network Traffic Analyzer](#)

Get complete real-time analysis across your entire network.
www.Paessler.com/Packet-Sniffi

[AscenLink - Load Balancer](#)

ISP link Aggregation, VPN Trunking. MPLS / IPVPN Backup & Load Balance
www.xtera-ip.com

[WAN News](#)

Networking Webcasts, White Papers Trends & Breaking News. Get Info!
NetworkWorld.com

network (*drop police*), unless they are coming from the two selected hosts. The setting allows up to 90KB to burst and a minburst (mtu) of 9KB. Have a look to the section [TBF queuing discipline](#) for more information about the minburst parameter.

This example goes easy, doesn't it? Let's continue with Edge31-ca-u32.



[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

3.3. Edge31-ca-u32

The example's script is as follows:

```
#!/bin/bash

##### Ingress side #####

tc qdisc add dev eth0 handle ffff: ingress

tc filter add dev eth0 parent ffff: protocol ip prio 4 handle 1: u32 \
divisor 1

tc filter add dev eth0 parent ffff: protocol ip prio 4 u32 \
match ip tos 0x88 0xfc \
police index 1 rate 1500kbit burst 90k \
continue flowid :1

tc filter add dev eth0 parent ffff: protocol ip prio 5 u32 \
match ip tos 0x88 0xfc \
police index 2 rate 1000kbit burst 90k \
continue flowid :2

tc filter add dev eth0 parent ffff: protocol ip prio 6 u32 \
match ip tos 0x88 0xfc \
police index 3 rate 1000kbit burst 60k \
drop flowid :3

tc filter add dev eth0 parent ffff: protocol ip prio 5 u32 \
match ip tos 0x90 0xfc \
police index 2 rate 1000kbit burst 90k \
continue flowid :2

tc filter add dev eth0 parent ffff: protocol ip prio 6 u32 \
match ip tos 0x90 0xfc \
police index 3 rate 1000kbit burst 60k \
drop flowid :3

tc filter add dev eth0 parent ffff: protocol ip prio 6 u32 \
match ip tos 0x98 0xfc \
police index 3 rate 1000kbit burst 60k \
drop flowid :3

tc filter add dev eth0 parent ffff: protocol ip prio 7 u32 \
match ip src 0/0 \
police index 4 rate 1500kbit burst 60k \
drop flowid :4

##### Egress side #####

tc qdisc add dev eth1 handle 1:0 root dsmark indices 64
```

[En](#) [s and](#)
[more](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[Network Traffic Control](#)

Full control over network traffic. Download a 70-day free trial !
www.ldecoGateway.com

[Network Traffic Analyzer](#)

Get complete real-time analysis across your entire network.
www.Paessler.com/Packet-Sniffing

[Network/Ethernet Tester](#)

Tester for Switch & SOHO Router Netwok TAP / Power Monitor
www.xtramus.com

[Free Download](#)

LogMeIn - Really 100% Free & Easy Access to PCs from Anywhere
www.LogMeIn.com

```
tc class change dev eth1 classid 1:1 dsmark mask 0x3 value 0x88
tc class change dev eth1 classid 1:2 dsmark mask 0x3 value 0x90
tc class change dev eth1 classid 1:3 dsmark mask 0x3 value 0x98
tc class change dev eth1 classid 1:4 dsmark mask 0x3 value 0x0
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 1 tcindex classid 1:1
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 2 tcindex classid 1:2
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 3 tcindex classid 1:3
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 4 tcindex classid 1:4
```

We have some new keywords in this example. It handles four classes identified by *tcindex* values 1, 2, 3 and 4. The idea is to mark packets in this way:

tcindex	marked as	DS class
1	0x88	AF41
2	0x90	AF42
3	0x98	AF43
4	0x0	BE

Table 3.3.1

This is done using the commands on the *egress* side. More explanations are not required for the *egress eth1 interface* configuration. Let's concentrate now in the *ingress eth2 interface* configuration.

Here they use what is known as *shared meters*. Each meter is identified by the keywords *police index n*, where *n* is the meter number. Four meters are used here. Meter 1 is attached to filter element 4 (*prio 4*), meters 2, 3 and 4 are attached to filter elements 5, 6, and 7, respectively.

This example assume that traffic arrives to the ingress router previously marked, probably by another DS capable domain. There are three rules to be applied to traffic that enter the ingress router marked as *DS field 0x88 (DS class AF41)*.

rule #	conditioning	tcindex	marked as	traffic was
1	up to 1500 kbps, max burst 90KB	1	0x88 = AF41	keepd on
2	next 1000 kbps, max burst 90KB	2	0x90 = AF42	remarked (demoted)
3	next 1000 kbps, max burst 60KB	3	0x98 = AF43	remarked (demoted)

Table 3.3.2

DS AF41 traffic above *3500kbps* is dropped. As you can see this edge router is really remarking entering flows. The service level agreement calls to keep on the *DS AF41* traffic with its original classification up to *1500kbps, 90KB burst*. This traffic is accepted and forwarded as it is. Above this *CIR* (committed information rate), for the next *1000kbps, 90KB burst, DS AF41* traffic is accepted but remarked (demoted) as *DS AF42*. Above this, next *1000kbps, 60KB burst*, again from the *DS AF41* traffic, is accepted but remarked (demoted) as *DS AF43*. Rest of the *DS AF41* traffic is *dropped*.

The job that our very nice edge (ingress) router is doing is conditioning the entering traffic.

Is left to the gentle reader to draw its own tables for entering traffic marked as *DS AF42* and *DS AF43*. It is a good exercise, then go ahead. By the way, what about with entering traffic not marked as *DS AF41, AF42 or AF43*? How does the router deal with them?

Okay, it's time to continue with Edge32-ca-u32.

[z/OS TCP/IP](#)

Maximize z/OS TCP/IP network performance and availability.
www.sdsusa.com

[Voice Compression Expert](#)

Dedicated to Voice Compression Up to 20:1 Compression Rate
www.broadgalaxy.com

[Cisco Router IOS Tutorial](#)

Master the Cisco Router and earn Big Bucks as a Network Engineer
www.routerkeys.com

[Configure DNS](#)

IP Address Management Simple & Secure DNS/DHCP
www.BlueCatNetworks.com

[Ip](#)

Soluções Completas de Telecom Confira Nossos Produtos e Serviços!
www.CTTelecom.com.br



This site is visited regularly for very specialized people; it's then the best place to advertise your very specialized product. First month at no cost for you to verify this aseveration.

[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

3.4. Edge32-ca-u32



Search

The example's script is as follows:

```
#!/bin/sh -x

##### Ingress side #####

tc qdisc add dev eth2 handle ffff: ingress

tc filter add dev eth2 parent ffff: protocol ip prio 1 u32 \
match ip tos 0x88 0xfc \
police index 1 rate 1000kbit burst 90k \
continue flowid :1

tc filter add dev eth2 parent ffff: protocol ip prio 2 u32 \
match ip tos 0x88 0xfc \
police index 2 rate 1000kbit burst 30k \
continue flowid :1

tc filter add dev eth2 parent ffff: protocol ip prio 3 u32 \
match ip tos 0x88 0xfc \
police index 3 rate 500kbit burst 90k \
continue flowid :2

tc filter add dev eth2 parent ffff: protocol ip prio 4 u32 \
match ip tos 0x88 0xfc \
police index 4 rate 500kbit burst 30k \
continue flowid :2

tc filter add dev eth2 parent ffff: protocol ip prio 5 u32 \
match ip tos 0x88 0xfc \
police index 5 rate 500kbit burst 90k \
continue flowid :3

tc filter add dev eth2 parent ffff: protocol ip prio 6 u32 \
match ip tos 0x88 0xfc \
police index 6 rate 500kbit burst 30k \
drop flowid :3

tc filter add dev eth2 parent ffff: protocol ip prio 8 u32 \
match ip tos 0x90 0xfc \
police index 3 rate 500kbit burst 90k \
continue flowid :2
```

[Network Traffic Control](#)

Full control over network traffic. Download a 70-day free trial !
www.ldecoGateway.com

[QoS Management Tools](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[Network Protocol Analyzer](#)

Advanced Network Analyzer Solution. Start Analyzing Today. Download!
www.Paessler.com/Packet-Sniffi

[AirMagnet Free-Trial](#)

Test/Audit/Fix your WLAN with Industry-leading Wi-Fi analyzer
www.airmagnet.com

[Configure DNS](#)

IP Address Management Simple & Secure DNS/DHCP
www.BlueCatNetworks.com

```
tc filter add dev eth2 parent ffff: protocol ip prio 9 u32 \  
match ip tos 0x90 0xfc \  
police index 4 rate 500kbit burst 30k \  
continue flowid :2
```

```
tc filter add dev eth2 parent ffff: protocol ip prio 10 u32 \  
match ip tos 0x90 0xfc \  
police index 5 rate 500kbit burst 90k \  
continue flowid :3
```

```
tc filter add dev eth2 parent ffff: protocol ip prio 11 u32 \  
match ip tos 0x90 0xfc \  
police index 6 rate 500kbit burst 30k \  
drop flowid :3
```

```
tc filter add dev eth2 parent ffff: protocol ip prio 13 u32 \  
match ip tos 0x98 0xfc \  
police index 5 rate 500kbit burst 90k \  
continue flowid :3
```

```
tc filter add dev eth2 parent ffff: protocol ip prio 14 u32 \  
match ip tos 0x98 0xfc \  
police index 6 rate 500kbit burst 30k \  
drop flowid :3
```

```
tc filter add dev eth2 parent ffff: protocol ip prio 16 u32 \  
match ip src 0/0\  
police index 7 rate 1000kbit burst 90k \  
drop flowid :4
```

```
##### Egress side #####
```

```
tc qdisc add dev eth1 handle 1:0 root dsmark indices 64
```

```
tc class change dev eth1 classid 1:1 dsmark mask 0x3 value 0x88  
tc class change dev eth1 classid 1:2 dsmark mask 0x3 value 0x90  
tc class change dev eth1 classid 1:3 dsmark mask 0x3 value 0x98  
tc class change dev eth1 classid 1:3 dsmark mask 0x3 value 0x0
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 1 tcindex classid 1:1
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 2 tcindex classid 1:2
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 3 tcindex classid 1:3
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 4 tcindex classid 1:4
```

This new example is very similar to the previous one. The egress side works exactly the same way. In the ingress side, seven (7) *shared meters* are used. This time the same meter can be shared between two or more filter elements. For example, the meter number 3 (identified by *police index 3*) is shared by the filter elements 3 and 8 (*prio 3 and prio 8*); have a look to the script above.

Taking the *AF41* entering traffic as example (marked with *tos 0x88*), in the first *combo rule* the flows are assigned to class 1 up to a rate of 1000 (*CIR1*) + 1000 (*PIR1*) kbps and a bursting of 90 (*CBS1*) + 30 (*EBS1*) KB. *CIR* stands for *Committed Information Rate*, *PIR* as *Peak Information Rate*, *CBS* as *Committed Burst Size* and *EBS* as *Extended Burst Size*. Then *AF41* traffic is passed on with a *tcindex* value = 1 if it doesn't exceed its $CIR1/CBS1 + PIR1/EBS1$.

If *AF41* traffic exceeds the first rule, but not an extra rate/burst of $CIR2/CBS1 + PIR2/EBS1$, it is passed on with a *tcindex* value = 2. Here, $CIR2 = 500\text{kbps}$ and $PIR2 = 500\text{kbps}$.

If *AF41* traffic exceeds the second rule, but not an extra rate/burst $CIR2/CBS2 + PIR2/EBS2$, it is passed on with a *tcindex* value = 3. Here, $CBS2 = 90\text{KB}$ and $EBS2 = 30\text{KB}$.

AF41 traffic exceeding above rule is *dropped*.

Same explanation applies to other type of traffic. For *AF42* traffic a 2-level *combo rule* is applied. For *AF43* traffic a 1-level *combo rule* is applied. For the rest of traffic a 1-level *single rule* is applied to passes this on with a *tcindex* value = 4, to be finally marked as *BE*.

Okay, it's time to continue with Edge31-cb-tables.

[Configure DNS](#)

IP Address Management
Simple & Secure
DNS/DHCP
www.BlueCatNetworks.com

[WAN News](#)

Networking Webcasts,
White Papers Trends &
Breaking News. Get Info!
NetworkWorld.com

[TDM Over IP Solutions](#)

E1/T1/J1 Over IP,MPLS,
PON Networks For
Wi-MAX,Radio and Cell.
Backhaul
www.techaya.com

[IP Microwave Experts](#)

Dragonwave, Exalt,
Ceragon Soln's 5.8 &
6-23 GHz Scalable to
GigE.
www.starmicrowave.com

[Free Download](#)

LogMeIn - Really 100%
Free & Easy Access to
PCs from Anywhere
www.LogMeIn.com



This site is visited regularly for very specialized people; it's then the best place to advertise your very specialized product.
First month at no cost for you to verify this aseveration.

[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

3.5. Edge31-cb-tables

In the original distribution this example was implemented using *ipchains*. Our version is implemented using the new Linux's firewall packet filter tool *iptables*. The modify example's script is as follows:

```
#!/bin/bash

##### Ingress side #####

iptables -t mangle -A INPUT -i eth2 -s 0/0 -j MARK --set-mark 2
iptables -t mangle -A INPUT -i eth2 -s 10.2.0.0/24 -j MARK --set-mark 1

tc qdisc add dev eth2 handle ffff: ingress

tc filter add dev eth2 parent ffff: protocol ip prio 4 handle 1 fw \
police rate 1500kbit burst 90k continue flowid 4:1

tc filter add dev eth2 parent ffff: protocol ip prio 5 handle 1 fw \
police rate 1500kbit burst 90k continue flowid 4:2

tc filter add dev eth2 parent ffff: protocol ip prio 6 handle 1 fw \
police rate 1000kbit burst 60k drop flowid 4:3

tc filter add dev eth2 parent ffff: protocol ip prio 6 handle 2 fw \
police rate 1000kbit burst 60k drop flowid 4:4

##### Egress side #####

tc qdisc add dev eth1 handle 1:0 root dsmark indices 64

tc class change dev eth1 classid 1:1 dsmark mask 0x3 value 0x88
tc class change dev eth1 classid 1:2 dsmark mask 0x3 value 0x90
tc class change dev eth1 classid 1:3 dsmark mask 0x3 value 0x98
tc class change dev eth1 classid 1:4 dsmark mask 0x3 value 0x0

tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 1 tcindex classid 1:1

tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 2 tcindex classid 1:2

tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 3 tcindex classid 1:3

tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
```

[Download Firewall](#)

Award Winning Internet
Firewall Sign Up -
Exclusive Member
Benefits
ITPro.co.uk/Firewall

[IP Communications](#)

IP Communications
solutions Visit our site
www.artilium.com

[Firewall Log Analyzer](#)

F/W Log Analysis &
Reporting Tool For B/W &
User Activity Monitoring.
www.fwanalyzer.com

[Download Iptables](#)

Free directory of security
tools, news, videos, and
podcasts.
www.AttackPrevention.com/Firewall

[Network Protocol Analyzer](#)

Get complete real-time
analysis across your
entire network.
www.Paessler.com/Packet-Sniffi

handle 4 tcindex classid 1:4

Let's start by the *egress side* because is easy. Leaving packets are assigned to four *DS classes*: *AF41*, *AF42*, *AF43* and *BE*, which will correspond to *tcindex* value of 1, 2, 3 and 4, respectively.

In the ingress side *iptables* is used to mark packets from network *10.2/24* as *fw 1*. Rest of packets are marked as *fw 2*. These packets (not coming from network *10.2/24*) are marked as *best-effort (BE)* when leaving the router using the *tcindex 4* rule with a police *rate/burst* of *1000kbps/60KB*.

Packets coming from network *10.2/24* are assigned to *DS class AF41* with the *tcindex 1* rule with a police *rate/burst* of *1500kbps/90KB*; next *1500kbps/90KB* are assigned to *DS class AF42* with *tcindex 2* rule; and next *1000kbps/60KB* are assigned to *DS class AF43* with *tcindex rule 3*.

As you see this example is even simpler than the previous one.

Okay, it's time to continue with Edge32-cb-tables.



[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

3.6. Edge32-cb-tables

Again, in the original distribution this example was implemented using *ipchains*. Our version is implemented using the new Linux's firewall packet filter tool *iptables*. The modify example's script is as follows:

```
#!/bin/sh -x

iptables -t mangle -A INPUT -i eth2 -s 0/0 -j MARK --set-mark 2
iptables -t mangle -A INPUT -i eth2 -s 10.2.0.0/24 -j MARK --set-mark 1

tc qdisc add dev eth2 handle ffff: ingress

tc filter add dev eth2 parent ffff: protocol ip prio 1 handle 1 fw \
police rate 1500kbit burst 90k \
continue flowid :1

tc filter add dev eth2 parent ffff: protocol ip prio 2 handle 1 fw \
police rate 500kbit burst 90k \
continue flowid :1

tc filter add dev eth2 parent ffff: protocol ip prio 3 handle 1 fw \
police rate 1500kbit burst 90k \
continue flowid :2

tc filter add dev eth2 parent ffff: protocol ip prio 4 handle 1 fw \
police rate 500kbit burst 90k \
continue flowid :2

tc filter add dev eth2 parent ffff: protocol ip prio 5 handle 1 fw \
police rate 500kbit burst 90k \
continue flowid :3

tc filter add dev eth2 parent ffff: protocol ip prio 6 handle 1 fw \
police rate 500kbit burst 90k \
drop flowid :3

tc filter add dev eth2 parent ffff: protocol ip prio 7 handle 2 fw \
police rate 1500kbit burst 90k \
drop flowid :4

##### Egress side #####

tc qdisc add dev eth1 handle 1:0 root dsmark indices 64
```

[Download Firewall](#)

Read Articles And
Comments Online Sign
Up- Exclusive Member
Benefits!
ITPro.co.uk/Firewall

[WinGate Proxy & Firewall](#)

NAT firewall, transparent
proxy, secure mail server
with antivirus
www.wingate.com

[Firewall Log Analyzer](#)

F/W Log Analysis &
Reporting Tool For B/W &
User Activity Monitoring.
www.fwanalyzer.com

[Found: IP Communications](#)

IP Communications
Artilium has all solutions
www.artilium.com

[Download Iptables](#)

Free directory of security
tools, news, videos, and
podcasts.
www.AttackPrevention.com/Firew

```
tc class change dev eth1 classid 1:1 dsmark mask 0x3 value 0x88
tc class change dev eth1 classid 1:2 dsmark mask 0x3 value 0x90
tc class change dev eth1 classid 1:3 dsmark mask 0x3 value 0x98
tc class change dev eth1 classid 1:4 dsmark mask 0x3 value 0x0
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 1 tcindex classid 1:1
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 2 tcindex classid 1:2
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 3 tcindex classid 1:3
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 4 tcindex classid 1:4
```

The *egress side* is the same as previous example. Leaving packets are assigned to four *DS classes*: *AF41*, *AF42*, *AF43* and *BE*, which will correspond to *tcindex* value of 1, 2, 3 and 4, respectively.

In the *ingress side* traffic is divided into two blocks: traffic coming from the network *10.2/24* and traffic coming from any other network. Traffic from any other network, except network *10.2/24*, is marked as *best-effort* by using the last rule of the filter chain (*prio7*), up to a maximum *rate/burst* of *1500kbps/90KB*. Traffic coming from these networks, but violating this setting, is dropped.

Traffic from network *10.2/24* is conditioned using a *3-level hierarchy*. *Level-1 (tcindex 1)* is marked as *DS class AF41*. *Level-2 (tcindex 2)* is marked as *DS class AF42*. *Level-3 (tcindex 3)* is marked as *DS class AF42*. Rest of this traffic is dropped. The *rate/burst* settings are *combo* (formed each by *2-chained rules*) as follows:

Level-1 (prio1 + prio2) is *1500kbps/90KB + 500kbps/90KB*.

Level-2 (prio3 + prio4) is *1500kbps/90KB + 500kbps/90KB*.

Level-3 (prio5 + prio6) is *500kbps/90KB + 500kbps/90KB*.

Well, this example is over. Next one will be *Edge32-cb-u32*



This site is visited regularly for very specialized people; it's then the best place to advertise your very specialized product. First month at no cost for you to verify this asevation.

[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

3.7. Edge32-cb-u32



Again, in the original distribution this example was implemented using *ipchains*. Our version is implemented using the new Linux's firewall packet filter tool *iptables*. The modify example's script is as follows:

```
#!/bin/sh

##### Ingress side #####

tc qdisc add dev eth2 handle ffff: ingress

tc filter add dev eth2 parent ffff: protocol ip prio 1 u32 \
match ip src 10.2.0.0/24 police rate 1000kbit burst 90k \
continue flowid :1

tc filter add dev eth2 parent ffff: protocol ip prio 2 u32 \
match ip src 10.2.0.0/24 police rate 1000kbit burst 15k \
continue flowid :1

tc filter add dev eth2 parent ffff: protocol ip prio 3 u32 \
match ip src 10.2.0.0/24 police rate 1000kbit burst 90k \
continue flowid :2

tc filter add dev eth2 parent ffff: protocol ip prio 4 u32 \
match ip src 10.2.0.0/24 police rate 500kbit burst 90k \
continue flowid :2

tc filter add dev eth2 parent ffff: protocol ip prio 5 u32 \
match ip src 10.2.0.0/24 police rate 1000kbit burst 90k \
continue flowid :3

tc filter add dev eth2 parent ffff: protocol ip prio 6 u32 \
match ip src 10.2.0.0/24 police rate 500kbit burst 15k \
drop flowid :3

tc filter add dev eth2 parent ffff: protocol ip prio 7 u32 \
match ip src 0/0 police rate 1000kbit burst 90k \
drop flowid :4

##### Egress side #####

tc qdisc add dev eth1 handle 1:0 root dsmark indices 64

tc class change dev eth1 classid 1:1 dsmark mask 0x3 value 0x88
tc class change dev eth1 classid 1:2 dsmark mask 0x3 value 0x90
tc class change dev eth1 classid 1:3 dsmark mask 0x3 value 0x98
tc class change dev eth1 classid 1:3 dsmark mask 0x3 value 0x0

tc filter add dev eth1 parent 1:0 protocol ip prio 1 \
handle 1 tcindex classid 1:1
```

[End to End QoS and more](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
Packeteer.com

[iptables firewall logs](#)

Analyze & Generate B/W & Web Usage Reports Based On Firewall Logs.
www.fwanalyzer.com

[Free Download](#)

LogMeIn - Really 100% Free & Easy Access to PCs from Anywhere
www.LogMeIn.com

[AirMagnet Free-Trial](#)

Test/Audit/Fix your WLAN with Industry-leading Wi-Fi analyzer
www.airmagnet.com

[Configure DNS](#)

IP Address Management Simple & Secure DNS/DHCP
www.BlueCatNetworks.com

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \  
handle 2 tcindex classid 1:2
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \  
handle 3 tcindex classid 1:3
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 \  
handle 4 tcindex classid 1:4
```

The *egress side* is the exactly same as previous example. Leaving packets are assigned to four *DS classes*: *AF41*, *AF42*, *AF43* and *BE*, which will correspond to *tcindex* value of 1, 2, 3 and 4, respectively. If you are a little observant soon you will discover that this example is almost the same as the previous one. This time the classification is done by using the *u32 classifier* instead of *iptables* as was done in the last example. Some little differences in the *committed/peak rate* and *burst*, i.e., they use *1000kbps/90KB* here and they used *1500kbps/90KB* there. The networks are the same. Traffic from network *10.2/24* will be treated different from traffic of rest of networks. In this example the network classification is done by using the *u32 classifier*, but the classification results are the same.

I'm going to be lazy and will give you as home work to compare both examples and analyze similarities and differences.

For not losing more time our next example will be *afcbq* that we rename as *afhtb*.



This site is visited regularly for very specialized people; it's then the best place to advertise your very specialized product. First month at no cost for you to verify this aseveration.

[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

3.8. Afhtb

In the original distribution this example was implemented using *cbq*. Our version is implemented using the new queuing discipline *htb*. The modify example's script is as follows. Because the example is long and a little bit complicated, I re-arrange the commands to simplify explanation and a mark with a **blue number** identify each command, or group of them:

```
#!/bin/bash

# 1 --- Main dsmark
tc qdisc add dev eth0 handle 1:0 root dsmark indices 64 set_tc_index

# 2 --- Main dsmark classifier
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
tcindex mask 0xfc shift 2 pass_on

# --- Main dsmark classifier's elements

# 3 --- AF Class 1
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
handle 10 tcindex classid 1:111
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
handle 12 tcindex classid 1:112
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
handle 14 tcindex classid 1:113

# 4 --- AF Class 2
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
handle 18 tcindex classid 1:121
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
handle 20 tcindex classid 1:122
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
handle 22 tcindex classid 1:123

# 5 --- AF Class 3
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
handle 26 tcindex classid 1:131
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
handle 28 tcindex classid 1:132
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
handle 30 tcindex classid 1:133

# 6 --- AF Class 4
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
```

[Enhance MPLS Performance](#)

Automatic QoS for 600+ WAN Apps. Learn How - Free MPLS Whitepaper. Packeteer.com

[SIP Monitor-Analyzer](#)

Call flow ladders, packet displays Linux & Windows support. Free trial! www.vqmanager.com

[CESoPSN & SAToP Solutions](#)

E1/T1/J1 Over IP, MPLS, EPON Networks For Next Generation Networks www.techaya.com

[Configure DNS](#)

IP Address Management Simple & Secure DNS/DHCP www.BlueCatNetworks.com

[CESOP Monitoring & Testing](#)

Perform testing and monitoring for TDM over IP network solutions www.prisma-eng.com

```
handle 34 tcindex classid 1:141
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
handle 36 tcindex classid 1:142
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
handle 38 tcindex classid 1:143

# 7 --- BE
tc filter add dev eth0 parent 1:0 protocol ip prio 2 \
handle 0 tcindex mask 0 classid 1:1

# 8 --- Main htb qdisc
tc qdisc add dev eth0 parent 1:0 handle 2:0 htb

# 9 --- Main htb class
tc class add dev eth0 parent 2:0 classid 2:1 htb rate 10Mbit ceil 10Mbit

# 10 --- Main htb classifier
tc filter add dev eth0 parent 2:0 protocol ip prio 1 \
tcindex mask 0xf0 shift 4 pass_on

# --- Main htb classifier's elements

# 11 --- AF Class 1
tc filter add dev eth0 parent 2:0 protocol ip prio 1 \
handle 1 tcindex classid 2:10

# 12 --- AF Class 2
tc filter add dev eth0 parent 2:0 protocol ip prio 1 \
handle 2 tcindex classid 2:20

# 13 --- AF Class 3
tc filter add dev eth0 parent 2:0 protocol ip prio 1 \
handle 3 tcindex classid 2:30

# 14 --- AF Class 4
tc filter add dev eth0 parent 2:0 protocol ip prio 1 \
handle 4 tcindex classid 2:40

# 15 --- BE
tc filter add dev eth0 parent 2:0 protocol ip prio 1 \
handle 0 tcindex classid 2:50

# 16 --- AF Class 1 specific setup---
tc class add dev eth0 parent 2:1 classid 2:10 htb rate 1500Kbit ceil 10Mbit
tc qdisc add dev eth0 parent 2:10 gred setup DPs 3 default 2 prio

# 17 --- AF Class 1 DP 1---
tc qdisc change dev eth0 parent 2:10 gred limit 60KB min 15KB max 45KB \
burst 20 avpkt 1000 bandwidth 10Mbit DP 1 probability 0.02 prio 2

# 18 --- AF Class 1 DP 2---
tc qdisc change dev eth0 parent 2:10 gred limit 60KB min 15KB max 45KB \
burst 20 avpkt 1000 bandwidth 10Mbit DP 2 probability 0.04 prio 3

# 19 --- AF Class 1 DP 3---
tc qdisc change dev eth0 parent 2:10 gred limit 60KB min 15KB max 45KB \
```

burst 20 avpkt 1000 bandwidth 10Mbit DP 3 probability 0.06 prio 4

20 --- AF Class 2 specific setup---

tc class add dev eth0 parent 2:1 classid 2:20 htb rate 1500Kbit ceil 10Mbit

tc qdisc add dev eth0 parent 2:20 gred setup DPs 3 default 2 prio

21 --- AF Class 2 DP 1---

tc qdisc change dev eth0 parent 2:20 gred limit 60KB min 15KB max 45KB burst 20 \

avpkt 1000 bandwidth 10Mbit DP 1 probability 0.02 prio 2

22 --- AF Class 2 DP 2---

tc qdisc change dev eth0 parent 2:20 gred limit 60KB min 15KB max 45KB \

burst 20 avpkt 1000 bandwidth 10Mbit DP 2 probability 0.04 prio 3

23 --- AF Class 2 DP 3---

tc qdisc change dev eth0 parent 2:20 gred limit 60KB min 15KB max 45KB \

burst 20 avpkt 1000 bandwidth 10Mbit DP 3 probability 0.06 prio 4

24 --- AF Class 3 specific setup---

tc class add dev eth0 parent 2:1 classid 2:30 htb rate 1500Kbit ceil 10Mbit

tc qdisc add dev eth0 parent 2:30 gred setup DPs 3 default 2 prio

25 --- AF Class 3 DP 1---

tc qdisc change dev eth0 parent 2:30 gred limit 60KB min 15KB max 45KB \

burst 20 avpkt 1000 bandwidth 10Mbit DP 1 probability 0.02 prio 2

26 --- AF Class 3 DP 2---

tc qdisc change dev eth0 parent 2:30 gred limit 60KB min 15KB max 45KB \

burst 20 avpkt 1000 bandwidth 10Mbit DP 2 probability 0.04 prio 3

27 --- AF Class 3 DP 3---

tc qdisc change dev eth0 parent 2:30 gred limit 60KB min 15KB max 45KB \

burst 20 avpkt 1000 bandwidth 10Mbit DP 3 probability 0.06 prio 4

28 --- AF Class 4 specific setup---

tc class add dev eth0 parent 2:1 classid 2:40 htb rate 1500Kbit ceil 10Mbit

tc qdisc add dev eth0 parent 2:40 gred setup DPs 3 default 2 prio

29 --- AF Class 4 DP 1---

tc qdisc change dev eth0 parent 2:40 gred limit 60KB min 15KB max 45KB \

burst 20 avpkt 1000 bandwidth 10Mbit DP 1 probability 0.02 prio 2

30 --- AF Class 4 DP 2---

tc qdisc change dev eth0 parent 2:40 gred limit 60KB min 15KB max 45KB \

burst 20 avpkt 1000 bandwidth 10Mbit DP 2 probability 0.04 prio 3

31 --- AF Class 4 DP 3---

tc qdisc change dev eth0 parent 2:40 gred limit 60KB min 15KB max 45KB \

burst 20 avpkt 1000 bandwidth 10Mbit DP 3 probability 0.06 prio 4

32 -----BE Queue setup-----

tc class add dev eth0 parent 2:1 classid 2:50 htb rate 1500Kbit ceil 10Mbit

tc qdisc add dev eth0 parent 2:50 red limit 60KB min 15KB max 45KB \

burst 20 avpkt 1000 bandwidth 10Mbit probability 0.4

The precedence **blue number** above are matched with the numbers below:

A very important note: to simplify the script understanding, I wrote some commands before some that require the firsts. For example, to configure the classifier element in the command 11 above:

```
tc filter add dev eth0 parent 2:0 protocol ip prio 1 handle 1 tcindex classid 2:10
```

You need to create **first** the htb class 2:10, as in the command 16 above, that is:

```
tc class add dev eth0 parent 2:1 classid 2:10 htb rate 1500Kbit ceil 10Mbit
```

The relative position was moved because it is easier to understand the commands when they are presented in this (natural) order; check your script to avoid a command from being executed before some other that it depends to.

1. The main *dsmark* queuing discipline is configured. Because *set_tc_index* is indicated, the discipline copies the *DS* field from the entering packet onto the *tc_index* field.
2. The main *dsmark*'s attached classifier is configured. It takes a copy of the value contained in the *tc_index* field, applies on it the bitwise operation (*value & 0xfc*) >> 2 and passes down the result to the classifier's elements.
3. Classifier's elements which correspond to the differentiated service class *AF1*. The *DS-field* values of the classes *AF11*, *AF12* and *AF13* are *0x28*, *0x30* and *0x38* respectively. Applying on these values the bitwise operation indicated in 2 (above), the new values will be *0xa*, *0xc* and *0xe*, which correspond to the decimal values *10*, *12* and *14* respectively. These values (*10,12,14*) will be matched by the classifier's elements, for example: *handle 10 tcindex* matches the value 10. The classifier's element returns back to the *dsmark* queuing discipline the *minor-value* of its *class identifier*. Then *handle 10 tcindex classid 1:111*, will return the class identifier *111*; but, be careful, *111* just means *0x0111*. *
4. Similar as number 3 above, but for the differentiated service class *AF2*. This time classes *AF21*, *AF22* and *AF23* are *0x48*, *0x50* and *0x58* respectively. After the bitwise operation the new values will be *0x12*, *0x14* and *0x16*, which correspond to the decimal values *18*, *20* and *22* respectively. These values will be matched by the classifier's elements returning the class identifier minor-values *121*, *122* and *123*.
5. Similar as number 3 above, but for the differentiated service class *AF3*. This time classes *AF31*, *AF32* and *AF33* are *0x68*, *0x70* and *0x78* respectively. After the bitwise operation the new values will be *0x1a*, *0x1c* and *0x1e*, which correspond to the decimal values *26*, *28* and *30* respectively. These values will be matched by the classifier's elements returning the class identifier minor-values *131*, *132* and *133*.
6. Similar as number 3 above, but for the differentiated service class *AF4*. This time classes

AF41, AF42 and AF43 are 0x88, 0x90 and 0x98 respectively. After the bitwise operation the new values will be 0x22, 0x24 and 0x26, which correspond to the decimal values 34, 36 and 38 respectively. These values will be matched by the classifier's elements returning the class identifier minor-values 141, 142 and 143.

7. Similar as number 3 above, but for the differentiated service class *BE (best-effort)*. Class *BE* is *DS field 0x0*. After the bitwise operation the new value continue being *0x0*. This value will be matched by this classifier's element (*handle 0 tcindex*) returning the class identifier minor-values *1*.
8. The main *htb* queuing discipline is configured here as a child of the main *dsmark* queuing discipline.
9. *htb* queuing discipline requires a main class to distribute bandwidth and implement *link-sharing*. This class is configured here as the *classid 2:1*, having a rate and ceiling bandwidth of *10 Mbps*.
10. The *htb* main classifier is configured in this line of our long script. Be specially careful with the *bitwise operation* implement by this pet. It masks the classes with *0xf0 (11110000)*, stripping this way the *4-rightmost bits*, and then applies a *4-bit right shift*; this means, it gets the *4-leftmost bits* of the *classid* value. What do you thing is going to occur when you apply this monster to one of the *classid* returned by the filter's elements identified above by the numbers 3, 4, 5, 6 and 7?. Let's take an example. I like the second element of the *AF Class 3* filter (identified by the number 5 above). The second element is:

```
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
handle 28 tcindex classid 1:132
```

Then, this little beetle returns (for the *dsmark* queuing discipline setting the *tc_index* field value) the *classid 132* (minor-value). Applying over this the bitwise operation we have (this will be occurring when the *main classifier* in this explanation puts its hands over the *tc_index* field value previously set by the *dsmark* queuing discipline):

$$0x0132 \& 0x00f0 = 0000\ 0001\ 0011\ 0010 \& 0000\ 0000\ 1111\ 0000 = \\ 0000\ 0000\ 0011\ 0000 = 0x30$$

$$0x30 = 0011\ 0000 \gg 4 = 0000\ 0011 = 0x3$$

Eureka!! What do we have here? Just the *AF class 3* identification. Do not forget that this precious value will be passed

[Free Download](#)

LogMeIn - Really 100% Free & Easy Access to PCs from Anywhere
www.LogMeIn.com

[AirMagnet Free-Trial](#)

Test AirMagnet's Laptop Tool and Police/Audit/Fix your WLAN!
www.airmagnet.com

[Free Packet Sniffer](#)

Packet Sniffer - It's Free. Very advanced networking tool.
www.Paessler.com

[WAN News](#)

Networking Webcasts, White Papers Trends & Breaking News. Get Info!
NetworkWorld.com

[Multithread Components](#)

We offer efficient components for atomic multithreaded programming!
www.pss-ab.com

down to the *filter's elements* of this classifier. See below, under numbers 11, 12, 13, 14 and 15.

That's the reason why I insisted before in handling very well these bit manipulating operations.

11. The first filter's element of the *htb's* main classifier. This element, and the next four below, are going to receive the value passed down from the *htb's* main classifier. The value passed, will be matched against the *tcindex handle* managed by each element. For this element (the first one), the value to be matched is *1 (handle 1 tcindex)*, see the command above, please. Okay. This element matches the value *1*. And, what is the meaning of receiving a value `== 1`. Just that the class of the packet is *AF class 1*. Then, this filter element *will match packet belonging to the DS class AF1*.

Nice!! But, what is the job entrusted to this filter element? Let's have a look to the command:

```
tc filter add dev eth0 parent 2:0 protocol ip prio 1 \
handle 1 tcindex classid 2:10
```

Our friend is then in charge to send these packets (*class AF1 packets*) to the *htb class* identified with the *classid 2:10*. Ja, ja... little by little we put our *AF1 class packets* in the place we want, just in the *htb class 2:10* where we can control them. But, keep on reading, this *htb class* is explained below...

12. Same as 11 for the *AF2 class*. Goes fast now, doesn't it? Using *htb's class 2:20* this time.
13. Same as 11 for the *AF3 class*. Using *htb's class 2:30*.
14. Same as 11 for the *AF4 class*. Using *htb's class 2:40*.
15. Same as 11 for the *BE class*. Using *htb's class 2:50*. Be careful, the *classid* matches by this element is *0x0* which correspond to the *BE filter* identified above with the number 7. Why?
16. The second *htb* queuing discipline's class is added (the first class was the main class *2:1* configured in 9). This command corresponds to the *htb class 2:10* which will be in charge to receive packets from *DS class AF1* (see explanation above, under 11). Using the *htb link-sharing* capacity, this class is being configured for a normal rate of *1500 kbps* and a ceiling rate of *10 Mbps*.

Immediately, using a second command, a *gred queuing discipline* is attached to this class. The *gred* is configured with *3 virtual queues*, being the queue number 2 the *default queue* and using a *priority buffer scheme (prio)*. The *three virtual queues* are configured using the commands 17, 18 and 19. See below.

17. The *htb class 2:10's gred virtual queue number 1* is configured here. This virtual queue will be in charge of dealing with packets belonging to the *DS class AF11*.
18. The *htb class 2:10's gred virtual queue number 2* is configured here. This virtual queue will be in charge of dealing with packets belonging to the *DS class AF12*.
19. The *htb class 2:10's gred virtual queue number 3* is configured here. This virtual queue will be in charge of dealing with packets belonging to the *DS class AF13*.
20. Now is the turn for adding a new *htb link-sharing class*. This command corresponds to the *htb class 2:20* which will be in charge to receive packets from *DS class AF2* (see explanation above, under 11). This class is being configured for a normal rate of *1500 kbps* and a ceiling rate of *10 Mbps*.

Same as under 16, immediately, using a second command, a *gred queuing discipline* is attached to this class. Again, the *gred* is configured with *3 virtual queues*, being the queue number 2 the *default queue* and using a *priority buffer scheme (prio)*. The *three virtual queues* are configured using the commands 21, 22 and 23. See below.

21. The *htb class 2:20's gred virtual queue number 1* is configured here. This virtual queue will be in charge of dealing with packets belonging to the *DS class AF21*.
22. The *htb class 2:20's gred virtual queue number 2* is configured here. This virtual queue will be in charge of dealing with packets belonging to the *DS class AF22*.
23. The *htb class 2:20's gred virtual queue number 3* is configured here. This virtual queue will be in charge of dealing with packets belonging to the *DS class AF23*.
24. Again, the turn is for adding a new *htb link-sharing class*. This command corresponds to the *htb class 2:30* which will be in charge to receive packets from *DS class AF3* (see explanation above, under 11). This class is being configured for a normal rate of *1500 kbps* and a ceiling rate of *10 Mbps*.

Same as under 16, immediately, using a second command, a *gred queuing discipline* is attached to this class. Again, the *gred* is configured with *3 virtual queues*, being the queue number 2 the *default queue* and using a *priority buffer scheme (prio)*. The *three virtual queues* are configured using the commands 25, 26 and 27. See below.

25. The *htb class 2:30's gred virtual queue number 1* is configured here. This virtual queue will be in charge of dealing with packets belonging to the *DS class AF31*.
26. The *htb class 2:30's gred virtual queue number 2* is configured here. This virtual queue will be in charge of dealing with packets belonging to the *DS class AF32*.

27. The *htb class 2:30's gred virtual queue number 3* is configured here. This virtual queue will be in charge of dealing with packets belonging to the *DS class AF33*.
28. Again, the turn is for adding a new *htb link-sharing class*. This command corresponds to the *htb class 2:40* which will be in charge to receive packets from *DS class AF4* (see explanation above, under 11). This class is being configured for a normal rate of *1500 kbps* and a ceiling rate of *10 Mbps*.

Same as under 16, immediately, using a second command, a *gred queuing discipline* is attached to this class. Again, the *gred* is configured with *3 virtual queues*, being the queue number 2 the *default queue* and using a *priority buffer scheme (prio)*. The *three virtual queues* are configured using the commands 29, 30 and 31. See below.

29. The *htb class 2:40's gred virtual queue number 1* is configured here. This virtual queue will be in charge of dealing with packets belonging to the *DS class AF41*.
30. The *htb class 2:40's gred virtual queue number 2* is configured here. This virtual queue will be in charge of dealing with packets belonging to the *DS class AF42*.
31. The *htb class 2:40's gred virtual queue number 3* is configured here. This virtual queue will be in charge of dealing with packets belonging to the *DS class AF43*.
32. Finally, the last *htb link-sharing class* is added. This command corresponds to the *htb class 2:50* which will be in charge to receive packets from *DS class BE* (see explanation above, under 11). This class is being configured for a normal rate of *1500 kbps* and a ceiling rate of *10 Mbps*.

Immediately, using a second command, a *red queuing discipline* (be careful, is a *red*, not a *gred* queueing discipline) is attached to this class. This queue is individually configured as it was a *gred* virtual queue.

Well, fine. The commands are explained. But, I think that two additional pieces of information could help to understand well enough how this stuff makes its work. Then, we are going to add an scheme (it helps more than a hundred of words) and an explanation of how a selected packet flows throughout the discipline (to understand how our tiny friend travels and is treated when it is enough courageous to cross, alone?, with some similars? our bewitched Linux router).

[If you have a 17 inches screen the scheme is just here](#)

In the scheme we are trying to explain, as an example, how a *DS class AF11* packet flows throughout the queuing discipline. The packet flow is explained, step by step, as follows (the small blue number at the right is related to the commands in the script above).

How a packet flows throughout the *dsmark* queuing discipline

1. The *AF11* packet enters the *dsmark* queuing discipline by the left (this is not really true, it's just a way to understand the diagram). Its *TOS* field contains the value *0x28*. (1)
2. The *TOS* field value is copied by the *dsmark* queuing discipline onto the *skb->tc_index* field. (1)
3. The *dsmark* queuing discipline's main classifier reads the *skb->tc_index* value (*0x28*) and applies the bitwise operation: *0x28 & 0xfc >> 2* ; the result of this operation is *0xa*. (2)
4. This value (*0xa*) which corresponds to decimal 10, is passed down to the *dsmark*'s main classifier elements. The first of these elements matches the decimal 10 value and returns back the *class id* value *0x111* to the *dsmark* queuing discipline. The *dsmark* queuing discipline then copies back this value again onto the *skb->tc_index* field. (3)
5. The packet enters the *htb* queuing discipline. (8)
6. The *htb* queuing discipline's main classifier reads the *skb->tc_index* value (*0x111*) and applies the bitwise operation: *0x111 & 0xf0 >> 4* ; the result of this operation is *0x1*. (10)
7. This value (*0x1*) which corresponds to decimal 1, is passed down to the *htb*'s main classifier elements. The first of these elements matches the decimal 1 value and returns back the *class id* value 10 to the *htb* queuing discipline. (11)
8. The *htb* queuing discipline then looks for this class number (10) in its child classes; it finds the class 2:10 and it puts the packet on it. Because this class is rated to *1500kbps*, the *AF11* packets are then allowed to flow up to *1500kbps*. (16)
9. The packet enters the *gred* queuing discipline. (16)
10. The *gred* queuing discipline needs to know in which *virtual queue* to place the packet; to do this it reads the packet's buffer *skb->tc_index* field value. The value *0x111* is found. The *gred* queuing discipline then uses the *last four bits* of this number to select a *VQ*. Last four bits correspond to the value *0x1*. Then the *VQ number 1* is selected. The *AF11* packets are then subjugated to a dropping probability of *0.02* (2%) when being buffered on this *VQ*. (17)
11. Finally the packet leave the *gred*, *htb* and *dsmark* queuing disciplines. Because the *dsmark* is configured as classless in this configuration, no additional transformation is applied to the *packet's DS field* when leaving the *dsmark* queuing discipline. The packet is then forwarded using the same DS class it had when entering the discipline.

[Voiceglobe IP Telephony](#)

Free your Voice Around the Globe Next Generation VoIP Technology
www.voiceglobe.net

[Ip](#)

Soluções Completas de Telecom Confira Nossos Produtos e Serviços!
www.CTTecom.com.br

[Get Your Own Server](#)

True Dedicated Hosting Only \$29.95. High Speed, 40GB Space. Free Setup!
www.ServerPronto.com

[Taiwan OEM manufacturer](#)

POS Touchscreen OEM Barcode Scanner Magstripe Reader, & Receipt Printer
www.barcode-manufacturer.com

[Acuerate Performance Cues](#)

Acuerate Cue Technology. Stephen Hendry plays Acuerate...
www.acuerate.com

Okay, fellows. I really hope this explanation had been good enough to having clear this DS on Linux scheme. Our next step will be ef-prio.

** Please, make your homework, have a read to the previous chapters...*

	This site is visited regularly for very specialized people; it's then the best place to advertise your very specialized product. First month at no cost for you to verify this aseveration.
---	--

[Previous](#)

[Content](#)

[Next](#)

[Previous](#)[Content](#)[Next](#)

3.10. efhtb

 Search

The example's script is as follows:

```
#!/bin/bash
```

```
# - Main dsmark & classifier
```

```
tc qdisc add dev eth1 handle 1:0 root dsmark indices 64 set_tc_index
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 1 tcindex mask 0xfc shift 2
```

```
# - Main htb qdisc & class
```

```
tc qdisc add dev eth1 parent 1:0 handle 2:0 htb
```

```
tc class add dev eth1 parent 2:0 classid 2:1 htb rate 10Mbit ceil 10Mbit
```

```
# - EF Class (2:10)
```

```
tc class add dev eth1 parent 2:1 classid 2:10 htb rate 1500Kbit ceil 10Mbit
```

```
tc qdisc add dev eth1 parent 2:10 pfifo limit 5
```

```
tc filter add dev eth1 parent 2:0 protocol ip prio 1 handle 0x2e tcindex \  
classid 2:10 pass_on
```

```
# - BE Class (2:20)
```

```
tc class add dev eth1 parent 2:1 classid 2:20 htb rate 5Mbit ceil 10Mbit
```

```
tc qdisc add dev eth1 parent 2:20 red limit 60KB min 15KB max 45KB \  
burst 20 avpkt 1000 bandwidth 10Mbit probability 0.4
```

```
tc filter add dev eth1 parent 2:0 protocol ip prio 2 handle 0 tcindex \  
mask 0 classid 2:20 pass_on
```

This is a very simple implementation of the *Expedited Forwarding (EF)* class using an *htb queuing discipline*. To begin with a *dsmark* queuing discipline is configured. *dsmark* will be in charge of copying the *DS field* from entering packets to the `skb->tc_index` field. The *dsmark's* main filter reads this value and applies over it the bitwise operation `DS & 0xfc >> 2`. This operation returns back the class value `0x2e` for packets marked as *EF*, this means, having the *DS field* set to `0xb8` when entering the discipline. This *class id* value is then copied back onto the `skb->tc_index` field by the *dsmark* queuing discipline.

A *htb* queuing discipline is configured having as parent the *dsmark* queuing discipline. The *htb 2:0* queuing discipline is configured with two classes (besides the *2:1* main class): the first class, *class 2:10*, is selected to place on it the packets belonging to the *DS EF* class. Because these packets, having the `skb->tc_index` field set to `0x2e` are matched against the first classifier element (*prio 1 handle 0x2e*), the class id *2:10* is returned and the packets are finally placed on it.

In the *htb class 2:10*, a *pfifo queuing discipline* is configured. The *maximum throughput* for *EF* traffic is controlled through the *htb class 2:10*, up to a maximum of *1.5 Mbps*, avoiding this way starvation over low priority traffic (*BE*) going through *class 2:20* (see below). We need to limit as much as possible the length of this *pfifo* queue to avoid long queue of *EF* packets that could increase latency in case of congestion; the *pfifo* queue is configured with *five packets length*.

Best effort (BE) packets are matched with the second classifier element (*prio 2 handle 0*), being then assigned to *htb class 2:20*. On this class a *red queuing discipline* is configured. *Red* is the best solution for *BE traffic* not subjugated to any kind of additional preemptive queuing control. It avoids phase effects, allows moderate bursting, doesn't impose any bias to well behaved flows and is as fair as is possible when dropping packets is a necessity. Have a look to [Red queuing discipline](#) for more information about this.

Some similar as the previous [ef-prio](#) scheme, we are just implementing a simple way for *EF packets* to kick them out as soon as is possible from the router to avoiding them to loss their time traveling through it. This scheme is normally used for *real-time* traffic where any delay in the transmission time affect the behavior quality. However the [ef-prio](#) scheme works a little better than this, being simpler and faster.

[End to End QoS and more](#)

Automatic QoS for 600+ Applications - Free Performance Survival Guide.
[Packeteer.com](http://www.packeteer.com)

[SIP Monitor-Analyzer](#)

Call flow ladders, packet displays Linux & Windows support. Free trial!
www.vqmanager.com

[Network/Ethernet Tester](#)

Tester for Switch & SOHO Router Network TAP / Power Monitor
www.xtramus.com

[Videocollaboration](#)

Real-time videocommunication & collaboration for the enterprise
www.dialcom.com

[Free Download](#)

LogMeIn - Leading Way For Geeks To Access PCs from Anywhere Free
www.LogMeIn.com

Well, fellows!! We are ready with all the examples of the original *Linux* *Diffserv* distribution; our next step in this study will be trying to configure some testbed to check the theoretical part. Then, let's go on.

Sorry, this work is yet in progress. I will try to add a little everyday as soon as I can steal some time from my schedule...



[Previous](#)

[Content](#)

[Next](#)