Traffic Control HOWTO

Version 1.1

Martin A. Brown, linux-ip.net [http://linux-ip.net/] <martin@linux-ip.net>
Federico Bolelli, Unimore [http://www.unimore.it/] <167439@studenti.unimore.it>
Natale Patriciello, Unimore [http://www.unimore.it/] <natale.patriciello@unimore.it>

2016-03-11

Revision History

Revision 1.1.0	2016-01-30	MAB
Incorporate new qdisc description sections written by Federico Bollelli and Natale Patriciello		
Revision 1.0.2	2006-10-28	MAB
Add references to HFSC, alter author email addresses		
Revision 1.0.1	2003-11-17	MAB
Added link to Leonardo Balliache's documentation		
Revision 1.0	2003-09-24	MAB
reviewed and approved by TLDP		
Revision 0.7	2003-09-14	MAB
incremental revisions, proofreading, ready for TLDP		
Revision 0.6	2003-09-09	MAB
minor editing, corrections from Stef Coene		
Revision 0.5	2003-09-01	MAB
HTB section mostly complete, more diagrams, LARTC pre-release		
Revision 0.4	2003-08-30	MAB
added diagram		
Revision 0.3	2003-08-29	MAB
substantial completion of classless, software, rules, elements and components sections		
Revision 0.2	2003-08-23	MAB
major work on overview, elements, components and software sections		
Revision 0.1	2003-08-15	MAB
initial revision (outline complete)		

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 enqueuing, policing, classifying, scheduling, shaping and dropping. This HOWTO provides an introduction and overview of the capabilities and implementation of traffic control under Linux.

Table of Contents

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 [http://lartc.org/howto/], documentation from individual projects and importantly the LARTC mailing list [http://vger.kernel.org/vger-lists.html#lartc] 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 [http://tldp.org/HOWTO/Traffic-Control-tcng-HTB-HOWTO/] and LARTC HOWTO [http://lartc.org/howto/] for immediate satisfaction.

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 the section called “Kernel requirements”), however, the ability to install and use software may be all that is required.

[1] See the section called “Software and Tools” for more details on the use or installation of a particular traffic control mechanism, kernel or command line utility.

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.

Conventions

This text was written in DocBook [http://www.docbook.org/] (version 4.2 [http://www.docbook.org/xml/4.2/index.html]) with **vim** [http://vim.sourceforge.net/]. All formatting has been applied by xsltproc [http://xmlsoft.org/XSLT/] based on DocBook XSL [http://docbook.sourceforge.net/projects/xsl/] and LDP XSL [http://www.tldp.org/LDP/LDP-Author-Guide/usingldpxsl.html] stylesheets. Typeface formatting and display conventions are similar to most printed and electronically distributed technical documentation.

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.

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 section of examples.
- A section detailing the classifiers.
- A section discussing the techniques for measuring traffic.
- A section covering meters.
- More details on **tcng**.
- Descriptions of newer qdiscs, specifically, Controlled Delay (codel), Fair Queue Controlled Delay (fq_codel), Proportional Integrated controller Enhanced (pie), Stochastic Fair Blue (sfb), Heavy Hitter Filter (hhf), Choke (choke).

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

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.

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 (if and) which 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 simplest possible model, 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. This is like a single toll booth for entering a highway. Every car must stop and pay the toll. Other cars wait their turn.

Linux provides this simplest traffic control tool (FIFO), and in addition offers a wide variety of other tools that allow all sorts of control over packet handling.

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 the section called “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 a set of tools allowing an administrator 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 at an IP-layer.

Why use it?

Traffic control tools allow the implementer to apply preferences, organizational or business policies to packets or network flows transmitted into a network. This control allows stewardship over the network resources such as throughput or latency.

Fundamentally, traffic control becomes a necessity because of packet switching in networks.

For a brief digression, to explain the novelty and cleverness of packet switching, think about the circuit-switched telephone networks that were built over the entire 20th century. In order to set up a call, the network gear knew rules about call establishment and when a caller tried to connect, the network employed the rules to reserve a circuit for the entire duration of the call or connection. While one call was engaged, using that resource, no other call or caller could use that resource. This meant many individual pieces of equipment could block call setup because of resource unavailability.

Let's return to packet-switched networks, a mid-20th century invention, later in wide use, and nearly ubiquitous in the 21st century. Packet-switched networks differ from circuit based networks in one very important regard. The unit of data handled by the network gear is not a circuit, but rather a small chunk of data called a packet. Inexactly speaking, the packet is a letter in an envelope with a destination address. The packet-switched network had only a very small amount of work to do, reading the destination identifier and transmitting the packet.

Sometimes, packet-switched networks are described as stateless because they do not need to track all of the flows (analogy to a circuit) that are active in the network. In order to be function, the packet-handling machines must know how to reach the destinations addresses. One analogy is a package-handling service like your postal service, UPS or DHL.

If there's a sudden influx of packets into a packet-switched network (or, by analogy, the increase of cards and packages sent by mail and other carriers at Christmas), the network can become slow or unresponsive. Lack of differentiation between importance of specific packets or network flows is, therefore, a weakness of such packet-switched networks. The network can be overloaded with data packets all competing.

In simplest terms, the traffic control tools allow somebody to enqueue packets into the network differently based on attributes of the packet. The various different tools each solve a different problem and many can be combined, to implement complex rules to meet a preference or business goal.

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 shows the types of common problems that can be solved by using traffic control tools to maximize the usability of the network.

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

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.

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.

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.

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

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.

In the image below a simplified high level overview of the queues on the transmit path of the Linux network stack:

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

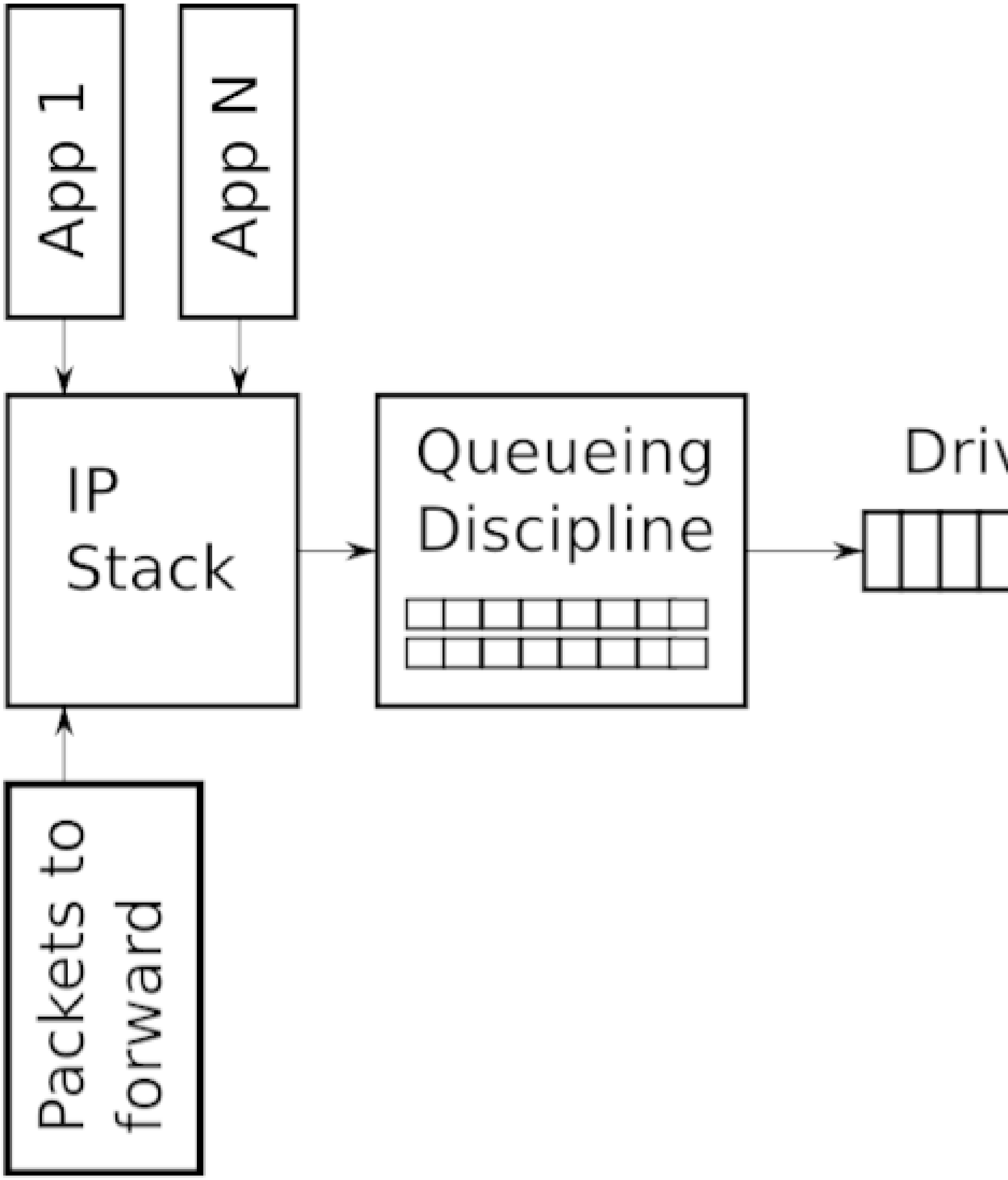


Figure 1: *Simplified high level overview of the queues on the transmit path of the Linux network stack.*

See 2.9 for details about NIC interface and 4.9 for details about driver queue.

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.

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.

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.

NIC, Network Interface Controller

A network interface controller is a computer hardware component, differently from previous ones thar are software components, that connects a computer to a computer network. The network controller implements the electronic circuitry required to communicate using a specific data link layer and physical layer standard such as Ethernet, Fibre Channel, Wi-Fi or Token Ring. Traffic control must deal with the physical constraints and characteristics of the NIC interface.

Huge Packets from the Stack

Most NICs have a fixed maximum transmission unit (MTU) which is the biggest frame which can be transmitted by the physical medium. For Ethernet the default MTU is 1500 bytes but some Ethernet networks support Jumbo Frames of up to 9000 bytes. Inside the IP network stack, the MTU can manifest as a limit on the size of the packets which are sent to the device for transmission. For example, if an application writes 2000 bytes to a TCP socket then the IP stack needs to create two IP packets to keep the packet size less than or equal to a 1500 byte MTU. For large data transfers the comparably small MTU causes a large number of small packets to be created and transferred through the driver queue.

In order to avoid the overhead associated with a large number of packets on the transmit path, the Linux kernel implements several optimizations: TCP segmentation offload (TSO), UDP fragmentation offload (UFO) and generic segmentation offload (GSO). All of these optimizations allow the IP stack to create packets which are larger than the MTU of the outgoing NIC. For IPv4, packets as large as the IPv4 maximum of 65,536 bytes can be created and queued to the driver queue. In the case of TSO and UFO, the NIC hardware takes responsibility for breaking the single large packet into packets small enough to be transmitted on the physical interface. For NICs without hardware support, GSO performs the same operation in software immediately before queueing to the driver queue.

Recall from earlier that the driver queue contains a fixed number of descriptors which each point to packets of varying sizes, Since TSO, UFO and GSO allow for much larger packets these optimizations have the side effect of greatly increasing the number of bytes which can be queued in the driver queue. Figure 3 illustrates this concept.

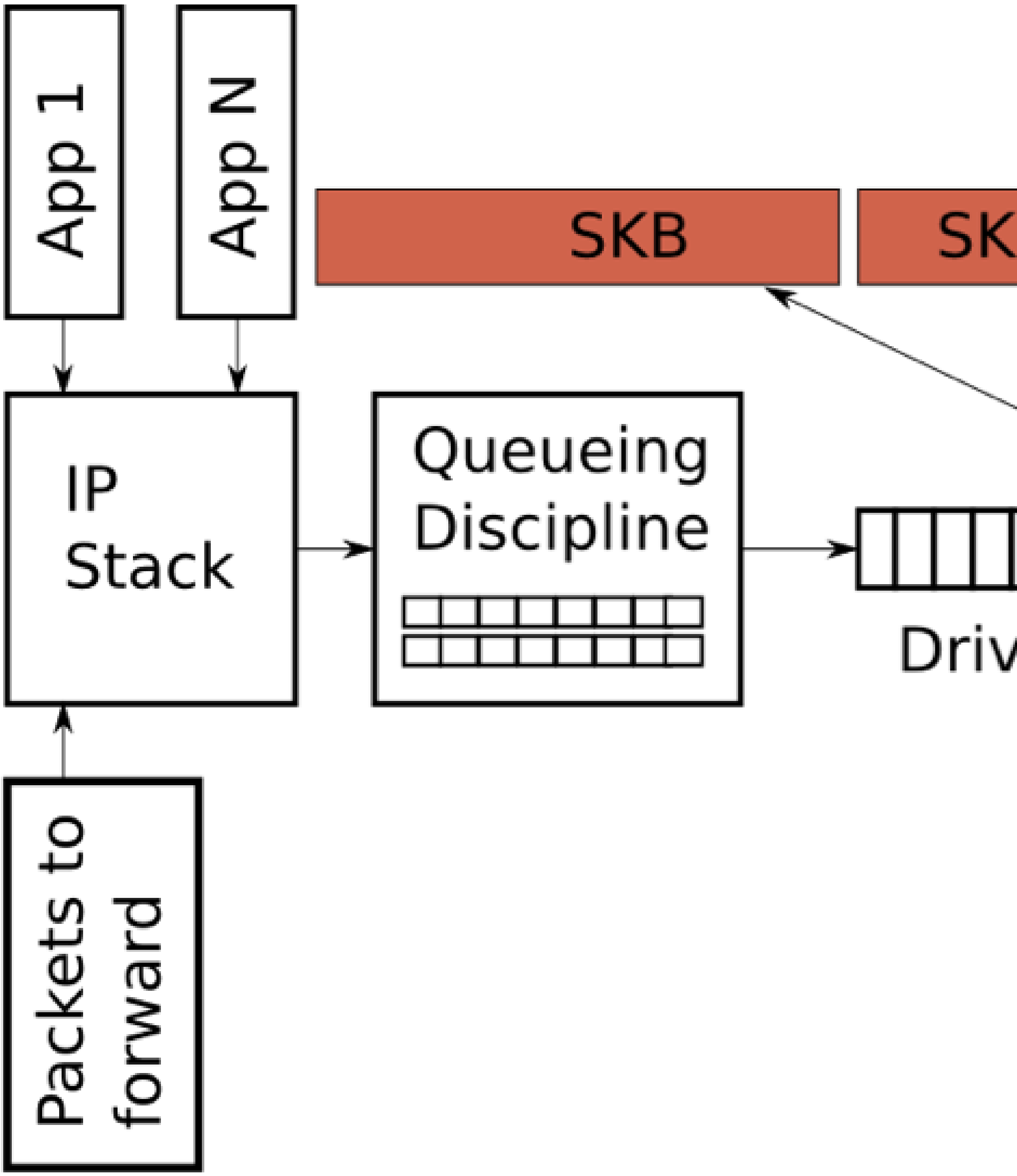


Figure 2: *Large packets can be sent to the NIC when TSO, UFO or GSO are enabled. This can greatly increase the number of bytes in the driver queue.*

Starvation and Latency

The queue between the IP stack and the hardware (see chapter 4.2 for detail about driver queue or see chapter 5.5 for how manage it) introduces two problems: starvation and latency.

If the NIC driver wakes to pull packets off of the queue for transmission and the queue is empty the hardware will miss a transmission opportunity thereby reducing the throughput of the system. This is referred to as starvation. Note that an empty queue when the system does not have anything to transmit is not starvation – this is normal. The complication associated with avoiding starvation is that the IP stack which is filling the queue and the hardware driver draining the queue run asynchronously. Worse, the duration between fill or drain events varies with the load on the system and external conditions such as the network interface’s physical medium. For example, on a busy system the IP stack will get fewer opportunities to add packets to the buffer which increases the chances that the hardware will drain the buffer before more packets are queued. For this reason it is advantageous to have a very large buffer to reduce the probability of starvation and ensures high throughput.

While a large queue is necessary for a busy system to maintain high throughput, it has the downside of allowing for the introduction of a large amount of latency.

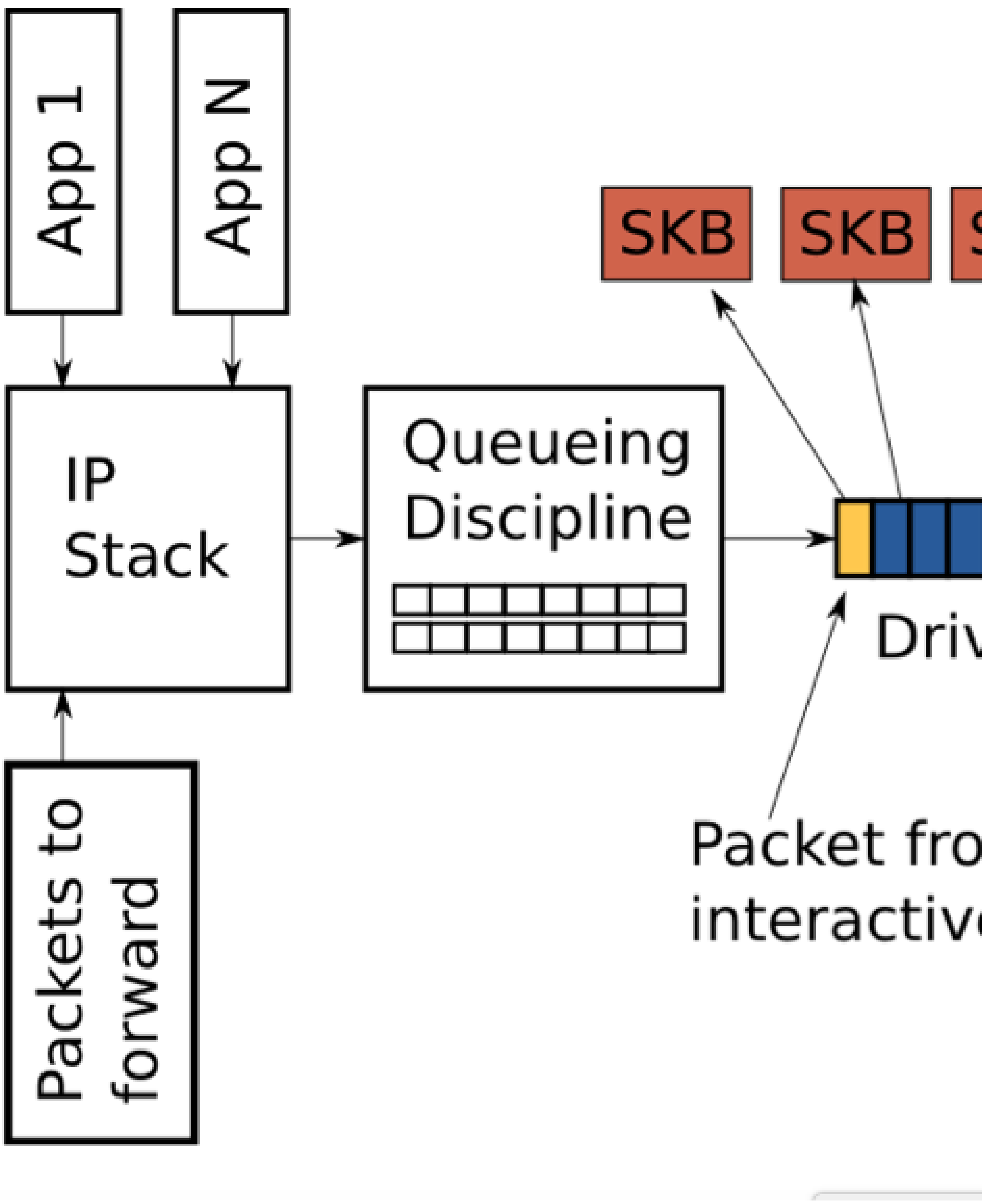


Figure 3: *Interactive packet (yellow) behind bulk flow packets (blue)*

Figure 3 shows a driver queue which is almost full with TCP segments for a single high bandwidth, bulk traffic flow (blue). Queued last is a packet from a VoIP or gaming flow (yellow). Interactive applications like VoIP or gaming typically emit small packets at fixed intervals which are latency sensitive while a high bandwidth data transfer generates a higher packet rate and larger packets. This higher packet rate can fill the buffer between interactive packets causing the transmission of the interactive packet to be delayed. To further illustrate this behaviour consider a scenario based on the following assumptions:

- A network interface which is capable of transmitting at 5 Mbit/sec or 5,000,000 bits/sec.
- Each packet from the bulk flow is 1,500 bytes or 12,000 bits.
- Each packet from the interactive flow is 500 bytes.
- The depth of the queue is 128 descriptors
- There are 127 bulk data packets and 1 interactive packet queued last.

Given the above assumptions, the time required to drain the 127 bulk packets and create a transmission opportunity for the interactive packet is (127 * 12,000) / 5,000,000 = 0.304 seconds (304 milliseconds for those who think of latency in terms of ping results). This amount of latency is well beyond what is acceptable for interactive applications and this does not even represent the complete round trip time – it is only the time required transmit the packets queued before the interactive one. As described earlier, the size of the packets in the driver queue can be larger than 1,500 bytes if TSO, UFO or GSO are enabled. This makes the latency problem correspondingly worse.

Choosing the correct size for the driver queue is a Goldilocks problem – it can’t be too small or throughput suffers, it can’t be too big or latency suffers.

Relationship between throughput and latency

In all traffic control systems, there is a relationship between throughput and latency. The maximum information rate of a network link is termed bandwidth, but for the user of a network the actually achieved bandwidth has a dedicated term, throughput.

latency	the delay in time between a sender's transmission and the recipient's decoding or receiving of the data; always non-negative and non-zero (time doesn't move backwards, then)
	in principle, latency is unidirectional, however almost the entire Internet networking community talks about bidirectional delay —the delay in time between a sender's transmission of data and some sort of acknowledgement of receipt of that data; cf. **ping**
	measured in milliseconds (ms); on Ethernet, latencies are typically between 0.3 and 1.0 ms and on wide-area networks (i.e. to your ISP, across a large campus or to a remote server) between 5 to 300 ms
throughput	a measure of the total amount of data that can be transmitted successfully between a sender and receiver
	measured in bits per second; the measurement most often quoted by complaining users after buying a 10Mbit/s package from their provider and receiving 8.2Mbit/s.

Note

Latency and throughput are general computing terms. For example, application developers speak of user-perceived latency when trying to build responsive tools. Database and filesystem people speak about disk throughput. And, above the network layer, latency of a website name lookup in DNS is a major contributor to the perceived performance of a website. The remainder of this document concerns latency in the network domain, specifically the IP network layer.

During the millenial fin de siècle, many developed world network service providers had learned that users were interested in the highest possible download throughput (the above mentioned 10Mbit/s bandwidth figure).

In order to maximize this download throughput, gear vendors and providers commonly tuned their equipment to hold a large number of data packets. When the network was ready to accept another packet, the network gear was certain to have one in its queue and could simply send another packet. In practice, this meant that the user, who was measuring download throughput, would receive a high number and was likely to be happy. This was desirable for the provider because the delivered throughput could more likely meet the advertised number.

This technique effectively maximized throughput, at the cost of latency. Imagine that a high priority packet is waiting at the end of the big queue of packets mentioned above. Perhaps, the theoretical latency of the packet on this network might be 100ms, but it needs to wait its turn in the queue to be transmitted.

While the decision to maximize throughput has been wildly successful, the effect on latency is significant.

Despite a general warning from Stuart Cheshire in the mid-1990s called It's the Latency, Stupid [http://www.stuartcheshire.org/rants/Latency.html], it took the novel term, bufferbloat, widely publicized about 15 years later by Jim Getty in an ACM Queue article Bufferbloat: Dark Buffers in the Internet [http://queue.acm.org/detail.cfm?id=2071893] and a Bufferbloat FAQ [https://gettys.wordpress.com/bufferbloat-faq/] in his blog, to bring some focus onto the choice for maximizing throughput that both gear vendors and providers preferred.

The relationship (tension) between latency and throughput in packet-switched networks have been well-known in the academic, networking and Linux development community. Linux traffic control core data structures date back to the 1990s and have been continuously developed and extended with new schedulers and features.

Traditional Elements of Traffic Control

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 the section called “Tokens and buckets” for further detail on tokens and buckets.

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

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).

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 the section called "`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 the section called "policer").

Policing

Policers measure and limit traffic in a particular queue.

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

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

Dropping

Dropping discards an entire packet, flow or classification.

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

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).

Components of Linux Traffic Control

Table 1. Correlation between traffic control elements and Linux components

traditional element	Linux component
enqueuing;	FIXME: receiving packets from userspace and network.
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.
enqueuing;	Between the scheduler's `qdisc` and the network interface controller (NIC) lies the driver queue. The driver queue gives the higher layers (IP stack and traffic control subsystem) a location to queue data asynchronously for the operation of the hardware. The size of that queue is automatically set by Byte Queue Limits (BQL).

`qdisc`

Simply put, a qdisc is a scheduler (the section called “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 `class`es, and provide a handle to which to attach `filter`s. There is no prohibition on using a classful qdisc without child classes, although this will usually consume cycles and other system resources for no benefit.

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

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

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

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

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

`class`

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

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

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

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

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.

[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.

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

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 the section called “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`”.

`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.

`handle`

Every `class` and classful `qdisc` (see also the section called “Classful Queuing Disciplines (`qdisc`s)”) 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.

[6] In this case, you'll have a `filter` which uses a `classifier` to select the packets you wish to drop. Then you'll use a `policer` with a with a drop action like this **police rate 1bps burst 1 action drop/drop**.
[7] I do not know the range nor base of these numbers. I believe they are u32 hexadecimal, but need to confirm this.

`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.

`txqueuelen`

The current size of the transmission queue can be obtained from the **ip** and **ifconfig** commands. Confusingly, these commands name the transmission queue length differently (emphasized text below):

```
$ifconfig eth0

eth0      Link encap:Ethernet  HWaddr 00:18:F3:51:44:10
          inet addr:69.41.199.58  Bcast:69.41.199.63 Mask:255.255.255.248
          inet6 addr: fe80::218:f3ff:fe51:4410/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:435033 errors:0 dropped:0 overruns:0 frame:0
          TX packets:429919 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:65651219 (62.6 MiB)  TX bytes:132143593 (126.0 MiB)
          Interrupt:23
```

```
$ip link

1: lo:  mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0:  mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:18:f3:51:44:10 brd ff:ff:ff:ff:ff:ff
```

The length of the transmission queue in Linux defaults to 1000 packets which could represent a large amount of buffering especially at low bandwidths. (To understand why, see the discussion on latency and throughput, specifically bufferbloat).

More interestingly, `txqueuelen` is only used as a default queue length for these queueing disciplines.

- `pfifo_fast` (Linux default queueing discipline)
- `sch_fifo`
- `sch_gred`
- `sch_htb` (only for the default queue)
- `sch_plug`
- `sch_sfb`

- `sch_teql`

Looking back at Figure 1, the txqueuelen parameter controls the size of the queues in the Queueing Discipline box for the QDiscs listed above. For most of these queueing disciplines, the “limit” argument on the tc command line overrides the txqueuelen default. In summary, if you do not use one of the above queueing disciplines or if you override the queue length then the txqueuelen value is meaningless.

The length of the transmission queue is configured with the ip or ifconfig commands.

```
ip link set txqueuelen 500 dev eth0
```

Notice that the ip command uses “txqueuelen” but when displaying the interface details it uses “qlen”.

Driver Queue (aka ring buffer)

Between the IP stack and the network interface controller (NIC) lies the driver queue. This queue is typically implemented as a first-in, first-out (FIFO) ring buffer – just think of it as a fixed sized buffer. The driver queue does not contain packet data. Instead it consists of descriptors which point to other data structures called socket kernel buffers (SKBs) which hold the packet data and are used throughout the kernel.

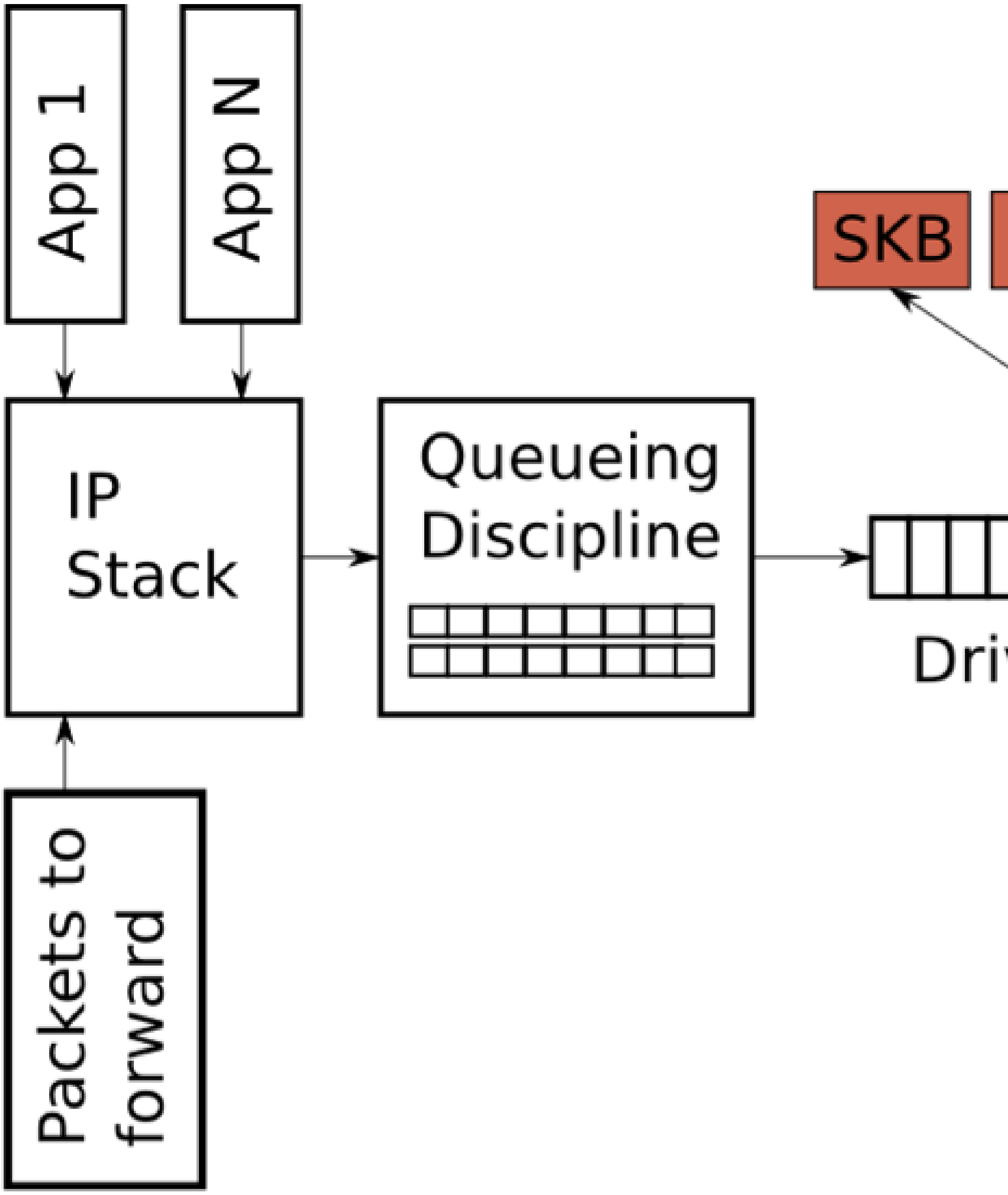


Figure 4: *Partially full driver queue with descriptors pointing to SKBs*

The input source for the driver queue is the IP stack which queues complete IP packets. The packets may be generated locally or received on one NIC to be routed out another when the device is functioning as an

IP router. Packets added to the driver queue by the IP stack are dequeued by the hardware driver and sent across a data bus to the NIC hardware for transmission.

The reason the driver queue exists is to ensure that whenever the system has data to transmit, the data is available to the NIC for immediate transmission. That is, the driver queue gives the IP stack a location to queue data asynchronously from the operation of the hardware. One alternative design would be for the NIC to ask the IP stack for data whenever the physical medium is ready to transmit. Since responding to this request cannot be instantaneous this design wastes valuable transmission opportunities resulting in lower throughput. The opposite approach would be for the IP stack to wait after a packet is created until the hardware is ready to transmit. This is also not ideal because the IP stack cannot move on to other work.

For detail how to set driver queue see chapter 5.5.

Byte Queue Limits (BQL)

Byte Queue Limits (BQL) is a new feature in recent Linux kernels (> 3.3.0) which attempts to solve the problem of driver queue sizing automatically. This is accomplished by adding a layer which enables and disables queuing to the driver queue based on calculating the minimum buffer size required to avoid starvation under the current system conditions. Recall from earlier that the smaller the amount of queued data, the lower the maximum latency experienced by queued packets.

It is key to understand that the actual size of the driver queue is not changed by BQL. Rather BQL calculates a limit of how much data (in bytes) can be queued at the current time. Any bytes over this limit must be held or dropped by the layers above the driver queue..

The BQL mechanism operates when two events occur: when packets are enqueued to the driver queue and when a transmission to the wire has completed. A simplified version of the BQL algorithm is outlined below. LIMIT refers to the value calculated by BQL.

```
****
** After adding packets to the queue
****

if the number of queued bytes is over the current LIMIT value then
        disable the queueing of more data to the driver queue
```

Notice that the amount of queued data can exceed LIMIT because data is queued before the LIMIT check occurs. Since a large number of bytes can be queued in a single operation when TSO, UFO or GSO (see chapter 2.9.1 aggiungi link for details) are enabled these throughput optimizations have the side effect of allowing a higher than desirable amount of data to be queued. If you care about latency you probably want to disable these features.

The second stage of BQL is executed after the hardware has completed a transmission (simplified pseudo-code):

```
****
** When the hardware has completed sending a batch of packets
** (Referred to as the end of an interval)
****

if the hardware was starved in the interval
    increase LIMIT
```

```
else if the hardware was busy during the entire interval (not starved) and there a
    decrease LIMIT by the number of bytes not transmitted in the interval

if the number of queued bytes is less than LIMIT
    enable the queueing of more data to the buffer
```

As you can see, BQL is based on testing whether the device was starved. If it was starved, then LIMIT is increased allowing more data to be queued which reduces the chance of starvation. If the device was busy for the entire interval and there are still bytes to be transferred in the queue then the queue is bigger than is necessary for the system under the current conditions and LIMIT is decreased to constrain the latency.

A real world example may help provide a sense of how much BQL affects the amount of data which can be queued. On one of my servers the driver queue size defaults to 256 descriptors. Since the Ethernet MTU is 1,500 bytes this means up to 256 * 1,500 = 384,000 bytes can be queued to the driver queue (TSO, GSO etc are disabled or this would be much higher). However, the limit value calculated by BQL is 3,012 bytes. As you can see, BQL greatly constrains the amount of data which can be queued.

An interesting aspect of BQL can be inferred from the first word in the name – byte. Unlike the size of the driver queue and most other packet queues, BQL operates on bytes. This is because the number of bytes has a more direct relationship with the time required to transmit to the physical medium than the number of packets or descriptors since the later are variably sized.

BQL reduces network latency by limiting the amount of queued data to the minimum required to avoid starvation. It also has the very important side effect of moving the point where most packets are queued from the driver queue which is a simple FIFO to the queueing discipline (QDisc) layer which is capable of implementing much more complicated queueing strategies. The next section introduces the Linux QDisc layer.

Set BQL

The BQL algorithm is self tuning so you probably don’t need to mess with this too much. However, if you are concerned about optimal latencies at low bitrates then you may want override the upper limit on the calculated LIMIT value. BQL state and configuration can be found in a /sys directory based on the location and name of the NIC. On my server the directory for eth0 is:

```
/sys/devices/pci0000:00/0000:00:14.0/net/eth0/queues/tx-0/byte_queue_limits
```

The files in this directory are:

- *hold_time:* Time between modifying LIMIT in milliseconds.
- *inflight:* The number of queued but not yet transmitted bytes.
- *limit:* The LIMIT value calculated by BQL. 0 if BQL is not supported in the NIC driver.
- *limit_max:* A configurable maximum value for LIMIT. Set this value lower to optimize for latency.
- *limit_min:* A configurable minimum value for LIMIT. Set this value higher to optimize for throughput.

To place a hard upper limit on the number of bytes which can be queued write the new value to the limit_max fie.

```
echo "3000" > limit_max
```

Software and Tools

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 [http://tldp.org/HOWTO/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 [http://tldp.org/HOWTO/Kernel-HOWTO/], as this document cannot adequately address questions about the use of the Linux kernel.

[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.

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 [http://linux-ip.net/gl/ip-cref/] and for a more expository discussion, the documentation at linux-ip.net [http://linux-ip.net/]. Of the tools in the **iproute2** package, the binary **tc** is the only one used for traffic control. This HOWTO will ignore the other tools in the suite.

Because it interacts with the kernel to direct the creation, deletion and modification of traffic control structures, the **tc** binary needs to be compiled with support for all of the `qdisc`s you wish to use. In particular, the HTB qdisc is not supported yet in the upstream **iproute2** package. See the section called “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 [http://lartc.org/howto/]. For even better understanding, consult the kernel and **iproute2** code.

Example 3. tc `qdisc`

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

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

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

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

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

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

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

Example 4. tc `class`

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

❶ Add a class. The verb could also be `del`.
❷ Specify the device onto which we are attaching the new class.
❸ Specify the parent `handle` to which we are attaching the new class.
❹ This is a unique `handle` (*`major`*:*`minor`*) identifying this class. The minor number must be any non-zero (0) number.
❺ Both of the classful qdiscs require that any children classes be classes of the same type as the parent. Thus an HTB qdisc will contain HTB classes.
❻❼ This is a class specific parameter. Consult the section called “HTB, Hierarchical Token Bucket” for more detail on these parameters.

Example 5. tc `filter`

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

❶ Add a filter. The verb could also be `del`.
❷ Specify the device onto which we are attaching the new filter.
❸ Specify the parent handle to which we are attaching the new filter.
❹ This parameter is required. It's use should be obvious, although I don't know more.
❺ The *`prio`* parameter allows a given filter to be preferred above another. The *`pref`* is a synonym.
❻ This is a `classifier`, and is a required phrase in every **tc** `filter` command.
❼❽ These are parameters to the classifier. In this case, packets with a type of service flag (indicating interactive usage) and matching port 22 will be selected by this statement.
❾ The *`flowid`* specifies the `handle` of the target class (or qdisc) to which a matching filter should send its selected packets.
❿ This is the `policer`, and is an optional phrase in every **tc** `filter` command.

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

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

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

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

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

tcng, Traffic Control Next Generation

FIXME; sing the praises of tcng. See also Traffic Control using tcng and HTB HOWTO [http://tldp.org/HOWTO/Traffic-Control-tcng-HTB-HOWTO/] and tcng documentation [http://linux-ip.net/gl/tcng/].

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

Netfilter

Netfilter is a framework provided by the Linux kernel that allows various networking-related operations to be implemented in the form of customized handlers. Netfilter offers various functions and operations for packet filtering, network address translation, and port translation, which provide the functionality required for directing packets through a network, as well as for providing ability to prohibit packets from reaching sensitive locations within a computer network.

Netfilter represents a set of hooks inside the Linux kernel, allowing specific kernel modules to register callback functions with the kernel's networking stack. Those functions, usually applied to the traffic in form of filtering and modification rules, are called for every packet that traverses the respective hook within the networking stack.

`iptables`

iptables is a user-space application program that allows a system administrator to configure the tables provided by the Linux kernel firewall (implemented as different Netfilter modules) and the chains and rules it stores. Different kernel modules and programs are currently used for different protocols; iptables applies to IPv4, ip6tables to IPv6, arptables to ARP, and ebtables to Ethernet frames.

iptables requires elevated privileges to operate and must be executed by user root, otherwise it fails to function. On most Linux systems, iptables is installed as /usr/sbin/iptables and documented in its man pages, which can be opened using man iptables when installed. It may also be found in /sbin/iptables, but since iptables is more like a service rather than an "essential binary", the preferred location remains /usr/sbin.

The term iptables is also commonly used to inclusively refer to the kernel-level components. x_tables is the name of the kernel module carrying the shared code portion used by all four modules that also provides the API used for extensions; subsequently, Xtables is more or less used to refer to the entire firewall (v4, v6, arp, and eb) architecture.

Xtables allows the system administrator to define tables containing chains of rules for the treatment of packets. Each table is associated with a different kind of packet processing. Packets are processed by sequentially traversing the rules in chains. A rule in a chain can cause a goto or jump to another chain, and this can be repeated to whatever level of nesting is desired. (A jump is like a “call”, i.e. the point that was jumped from is remembered.) Every network packet arriving at or leaving from the computer traverses at least one chain.

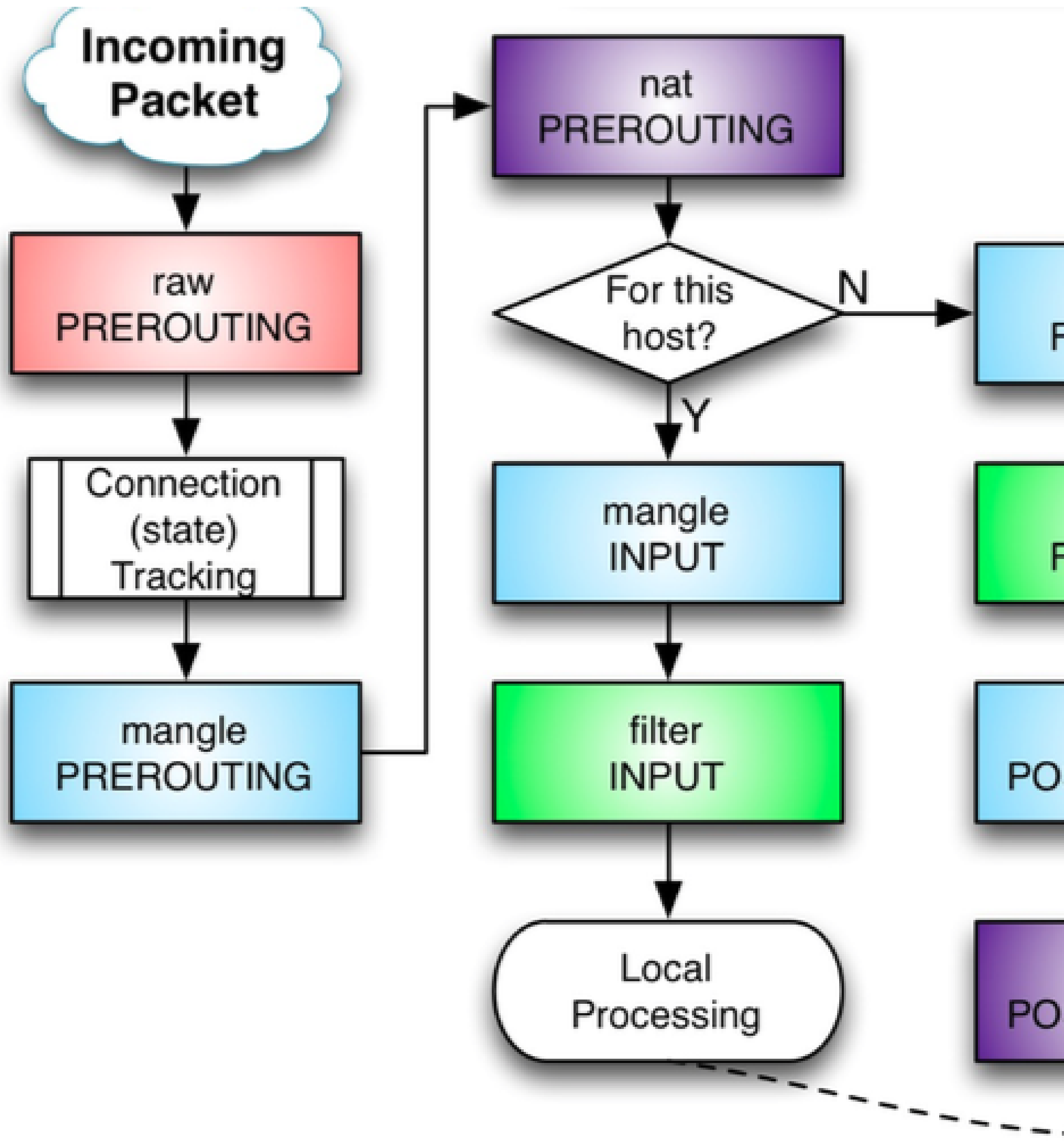


Figure 5: *Packet flow paths. Packets start at a given box and will flow along a certain path, depending on the circumstances.*

The origin of the packet determines which chain it traverses initially. There are five predefined chains (mapping to the five available Netfilter hooks, see figure 5), though a table may not have all chains.

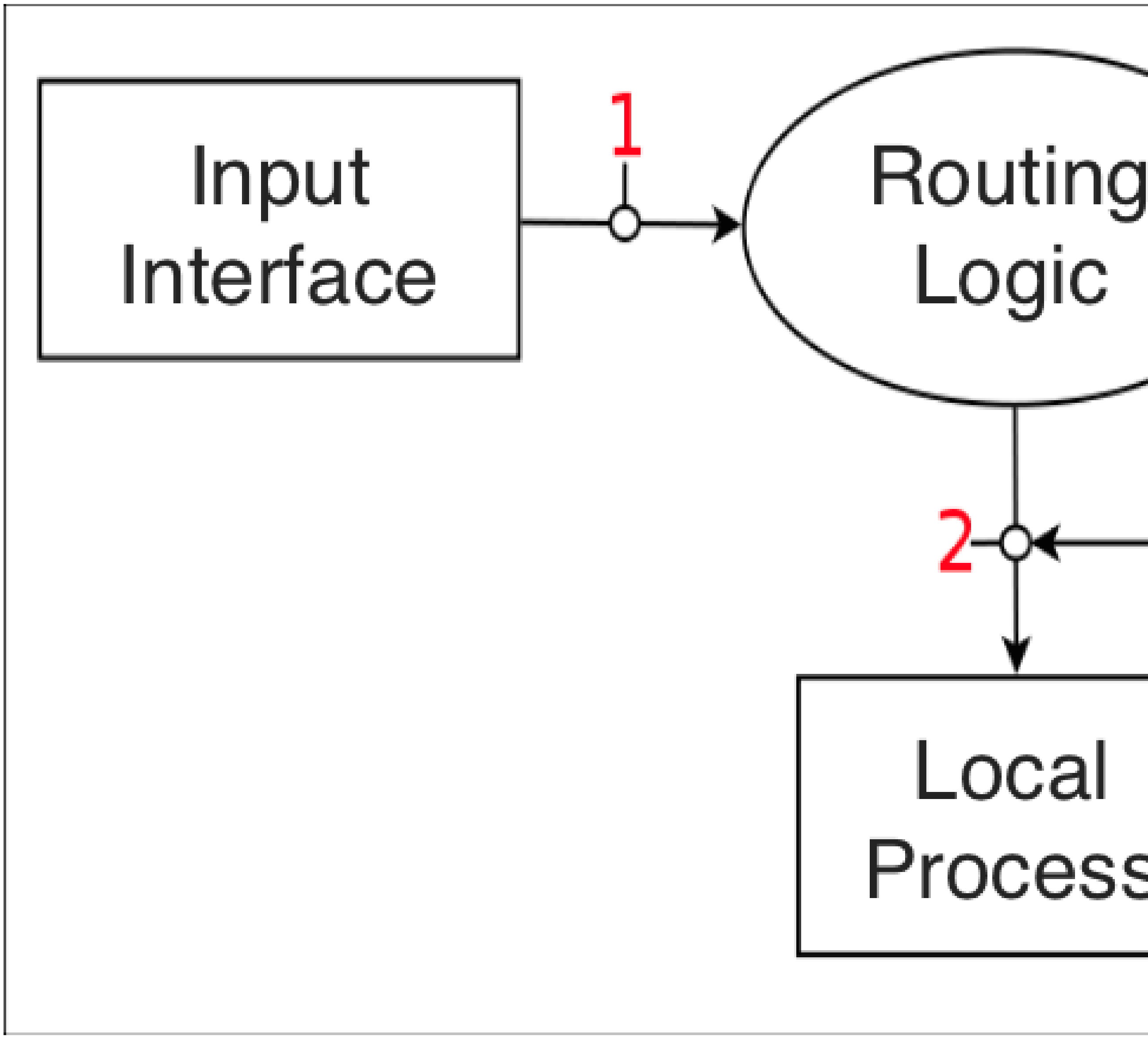


Figure 6: *netfilter's hook*

Predefined chains have a policy, for example DROP, which is applied to the packet if it reaches the end of the chain. The system administrator can create as many other chains as desired. These chains have no policy; if a packet reaches the end of the chain it is returned to the chain which called it. A chain may be empty.

- `PREROUTING:` Packets will enter this chain before a routing decision is made (point 1 in Figure 6).
- `INPUT:` Packet is going to be locally delivered. It does not have anything to do with processes having an opened socket; local delivery is controlled by the "local-delivery" routing table: ip route show table local (point 2 Figure 6).
- `FORWARD:` All packets that have been routed and were not for local delivery will traverse this chain (point 3 in Figure 6).

- `OUTPUT:` Packets sent from the machine itself will be visiting this chain (point 5 in Figure 6)
- `POSTROUTING:` Routing decision has been made. Packets enter this chain just before handing them off to the hardware (point 4 in Figure 6).

Each rule in a chain contains the specification of which packets it matches. It may also contain a target (used for extensions) or verdict (one of the built-in decisions). As a packet traverses a chain, each rule in turn is examined. If a rule does not match the packet, the packet is passed to the next rule. If a rule does match the packet, the rule takes the action indicated by the target/verdict, which may result in the packet being allowed to continue along the chain or it may not. Matches make up the large part of rulesets, as they contain the conditions packets are tested for. These can happen for about any layer in the OSI model, as with e.g. the --mac-source and -p tcp --dport parameters, and there are also protocol-independent matches, such as -m time.

The packet continues to traverse the chain until either

- a rule matches the packet and decides the ultimate fate of the packet, for example by calling one of the `ACCEPT` or `DROP`, or a module returning such an ultimate fate; or
- a rule calls the `RETURN` verdict, in which case processing returns to the calling chain; or
- the end of the chain is reached; traversal either continues in the parent chain (as if RETURN was used), or the base chain policy, which is an ultimate fate, is used.

Targets also return a verdict like ACCEPT (NAT modules will do this) or DROP (e.g. the REJECT module), but may also imply CONTINUE (e.g. the LOG module; CONTINUE is an internal name) to continue with the next rule as if no target/verdict was specified at all.

IMQ, Intermediate Queuing device

The Intermediate queueing device is not a qdisc but its usage is tightly bound to qdiscs. Within linux, qdiscs are attached to network devices and everything that is queued to the device is first queued to the qdisc and then to driver queue. From this concept, two limitations arise:

- Only egress shaping is possible (an ingress qdisc exists, but its possibilities are very limited compared to classful qdiscs, as seen before).
- A qdisc can only see traffic of one interface, global limitations can't be placed.

IMQ is there to help solve those two limitations. In short, you can put everything you choose in a qdisc. Specially marked packets get intercepted in netfilter NF_IP_PRE_ROUTING and NF_IP_POST_ROUTING hooks and pass through the qdisc attached to an imq device. An iptables target is used for marking the packets.

This enables you to do ingress shaping as you can just mark packets coming in from somewhere and/or treat interfaces as classes to set global limits. You can also do lots of other stuff like just putting your http traffic in a qdisc, put new connection requests in a qdisc, exc.

Sample configuration

The first thing that might come to mind is use ingress shaping to give yourself a high guaranteed bandwidth. Configuration is just like with any other interface:

```
tc qdisc add dev imq0 root handle 1: htb default 20

tc class add dev imq0 parent 1: classid 1:1 htb rate 2mbit burst 15k
tc class add dev imq0 parent 1:1 classid 1:10 htb rate 1mbit
tc class add dev imq0 parent 1:1 classid 1:20 htb rate 1mbit

tc qdisc add dev imq0 parent 1:10 handle 10: pfifo
tc qdisc add dev imq0 parent 1:20 handle 20: sfq



tc filter add dev imq0 parent 10:0 protocol ip prio 1 u32 match \ ip dst 10.0.0.23
```

In this example u32 is used for classification. Other classifiers should work as expected. Next traffic has to be selected and marked to be enqueued to imq0.

```
iptables -t mangle -A PREROUTING -i eth0 -j IMQ --todev 0

ip link set imq0 up
```

The IMQ iptables targets is valid in the PREROUTING and POSTROUTING chains of the mangle table. It's syntax is

```
IMQ [ --todev n ] n : number of imq device
```

An ip6tables target is also provided.

Please note traffic is not enqueued when the target is hit but afterwards. The exact location where traffic enters the imq device depends on the direction of the traffic (in/out). These are the predefined netfilter hooks used by iptables:

```
enum nf_ip_hook_priorities {
            NF_IP_PRI_FIRST = INT_MIN,
            NF_IP_PRI_CONNTRACK = -200,
            NF_IP_PRI_MANGLE = -150,
            NF_IP_PRI_NAT_DST = -100,
            NF_IP_PRI_FILTER = 0,
            NF_IP_PRI_NAT_SRC = 100,
            NF_IP_PRI_LAST = INT_MAX,
            };
```

For ingress traffic, imq registers itself with NF_IP_PRI_MANGLE + 1 priority which means packets enter the imq device directly after the mangle PREROUTING chain has been passed.

For egress imq uses NF_IP_PRI_LAST which honours the fact that packets dropped by the filter table won't occupy bandwidth.

`ethtool`, Driver Queue

The ethtool command is used to control the driver queue size for Ethernet devices. ethtool also provides low level interface statistics as well as the ability to enable and disable IP stack and driver features.

The -g flag to ethtool displays the driver queue (ring) parameters (see Figure 1) :

```
$ethtool -g eth0

    Ring parameters for eth0:
    Pre-set maximums:
    RX:        16384
    RX Mini:    0
    RX Jumbo:    0
    TX:        16384
    Current hardware settings:
    RX:        512
    RX Mini:    0
    RX Jumbo:    0
    TX:        256
```

You can see from the above output that the driver for this NIC defaults to 256 descriptors in the transmission queue. It was often recommended to reduce the size of the driver queue in order to reduce latency. With the introduction of BQL (assuming your NIC driver supports it) there is no longer any reason to modify the driver queue size (see the below for how to configure BQL).

Ethtool also allows you to manage optimization features such as TSO, UFO and GSO. The -k flag displays the current offload settings and -K modifies them.

```
$ethtool -k eth0

    Offload parameters for eth0:
    rx-checksumming: off
    tx-checksumming: off
    scatter-gather: off
    tcp-segmentation-offload: off
    udp-fragmentation-offload: off
    generic-segmentation-offload: off
    generic-receive-offload: on
    large-receive-offload: off
    rx-vlan-offload: off
    tx-vlan-offload: off
    ntuple-filters: off
    receive-hashing: off
```

Since TSO, GSO, UFO and GRO greatly increase the number of bytes which can be queued in the driver queue you should disable these optimizations if you want to optimize for latency over throughput. It’s doubtful you will notice any CPU impact or throughput decrease when disabling these features unless the system is handling very high data rates.

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`.

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

Note

Although a FIFO is one of the simplest elements of a queueing system, neither `pfifo` nor `bfifo` is the default qdisc on Linux interfaces. Be certain to see the section called “`pfifo_fast`, the default Linux qdisc” for the full details on the default (`pfifo_fast`) qdisc.

`pfifo, bfifo` Algorithm

The FIFO algorithm forms the underlying 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.

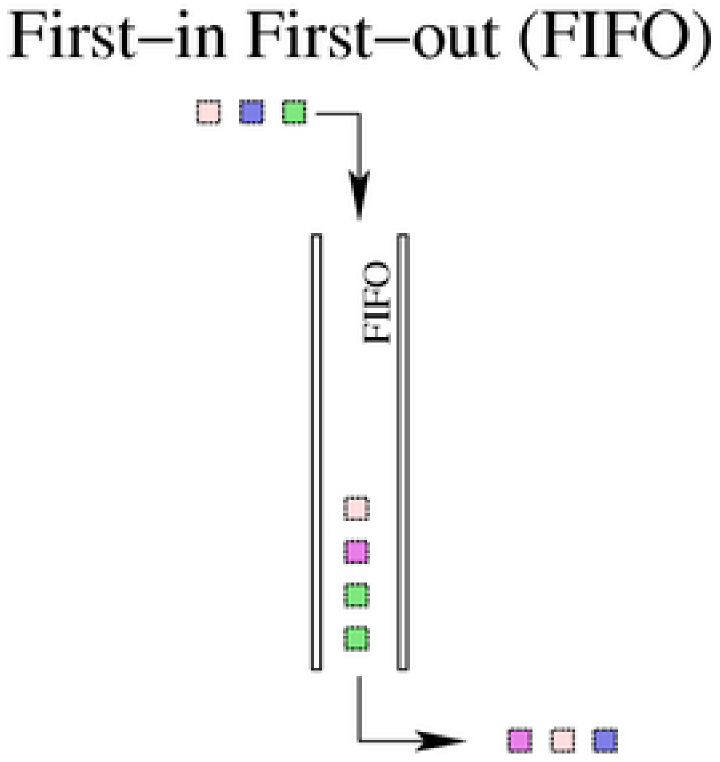


Figure 7: *FIFO qdisc*

A list of packets is maintained, when a packet is enqueued it gets inserted at the tail of a list. When a packet needs to be sent out to the network, it is taken from the head of the list.

`limit` parameter

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` *`(Packet limited First In, First Out queue)`* the unit is understood to be packets and for a `bfifo` *`(Byte limited First In, First Out queue)`* the unit is understood to be bytes.

For pfifo, defaults to the interface txqueuelen , as specified with ifconfig or ip. The range for this parameter is [0, UINT32_MAX].

For bfifo, it defaults to the txqueuelen multiplied by the interface MTU. The range for this parameter is [0, UINT32_MAX] bytes. Note: The link layer header was considered when counting packet length.

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
```

tc –s qdisc ls

The output of tc -s qdisc ls contains the limit, either in packets or in bytes, and the number of bytes and packets actually sent. An unsent and dropped packet only appears between braces and is not counted as 'Sent'.

In this example, the queue length is 100 packets, 45894 bytes were sentover 681 packets. No packets were dropped, and as the pfifo queue does not slow down packets, there were also no overlimits:

```
$ tc -s qdisc ls dev eth0
```

```
qdisc pfifo 8001: dev eth0 limit 100p
Sent 45894 bytes 681 pkts (dropped 0, overlimits 0)
```

If a backlog occurs, this is displayed as well.

`pfifo` and `bfifo`, like all non-default qdiscs, maintain statistics. This might be a reason to prefer that over the default.

`pfifo_fast`, the default Linux qdisc

The `pfifo_fast` *`(three-band first in, first out queue)`* qdisc is the default qdisc for all interfaces under Linux. Whenever an interface is created, the pfifo_fast qdisc is automatically used as a queue. If another qdisc is attached, it preempts the default pfifo_fast, which automatically returns to function when an existing qdisc is detached.

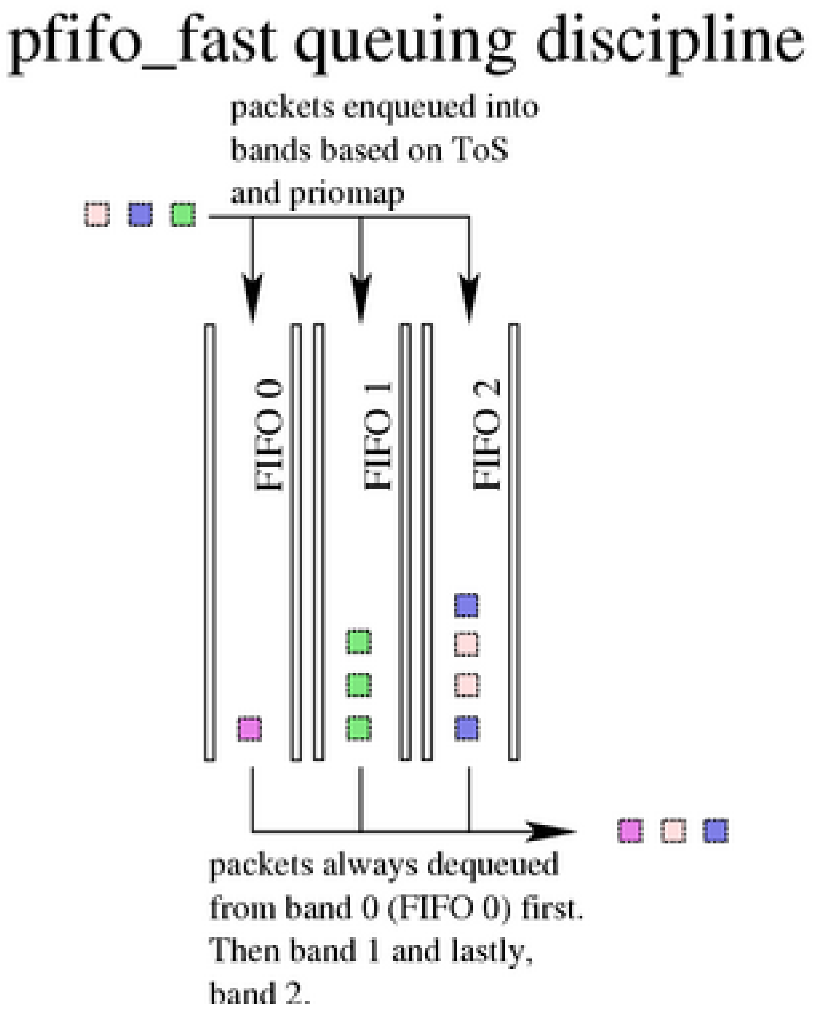

pfifo_fast queuing discipline
packets enqueued into bands based on ToS and priomap
FIFO 0
FIFO 1
FIFO 2
packets always dequeued from band 0 (FIFO 0) first. Then band 1 and lastly, band 2.

Figure 8: *pfifo_fast qdisc*

`pfifo_fast` algorithm

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.

The algorithm is very similar to that of the classful prio qdisc. The pfifo_fast qdisc is like three `pfifo` queues side by side, where an individual packet will be enqueued in one of the three FIFOs based on its Type of Service (ToS) bits. Not all three bands are dequeued simultaneously - as long as lower-numbered (higher priority) bands have traffic, higher-numbered bands are never dequeued. This can be used to prioritize interactive traffic or penalize 'lowest cost' traffic. Each band can be txqueuelen packets long, as configured with ifconfig or ip. Any additional packets arriving, once a specific band is full, are not enqueued but are instead dropped.

See chapter 6.2.3 for complete details on how ToS bits are translated into bands.

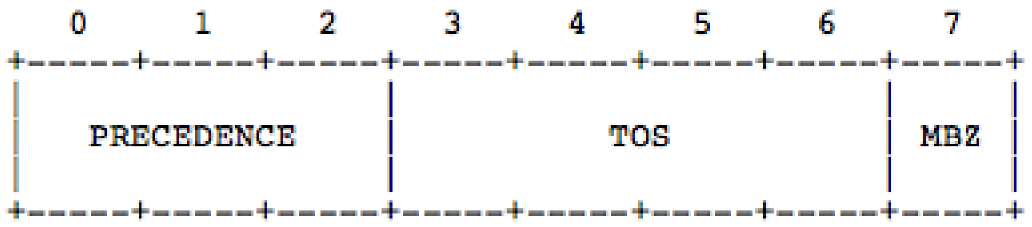
`txqueuelen` parameter

The length of the three bands depends on the interface txqueuelen, as specified with ifconfig or ip.

Bugs

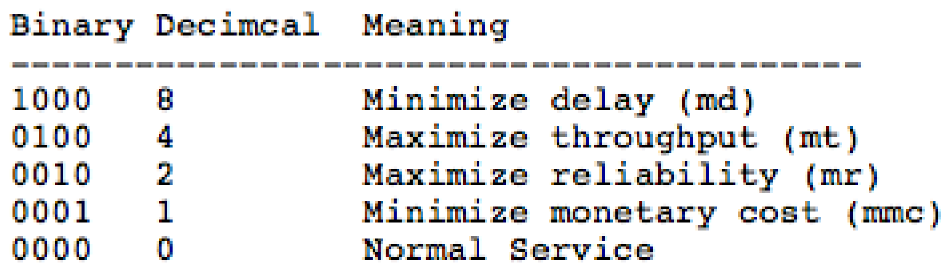
There is nothing configurable to the end user about the pfifo_fast qdisc. This is how it is configured by default:

`priomap` determines how packet priorities, as assigned by the kernel, map to bands. Mapping occurs based on the ToS octet of the packet, which looks like this:



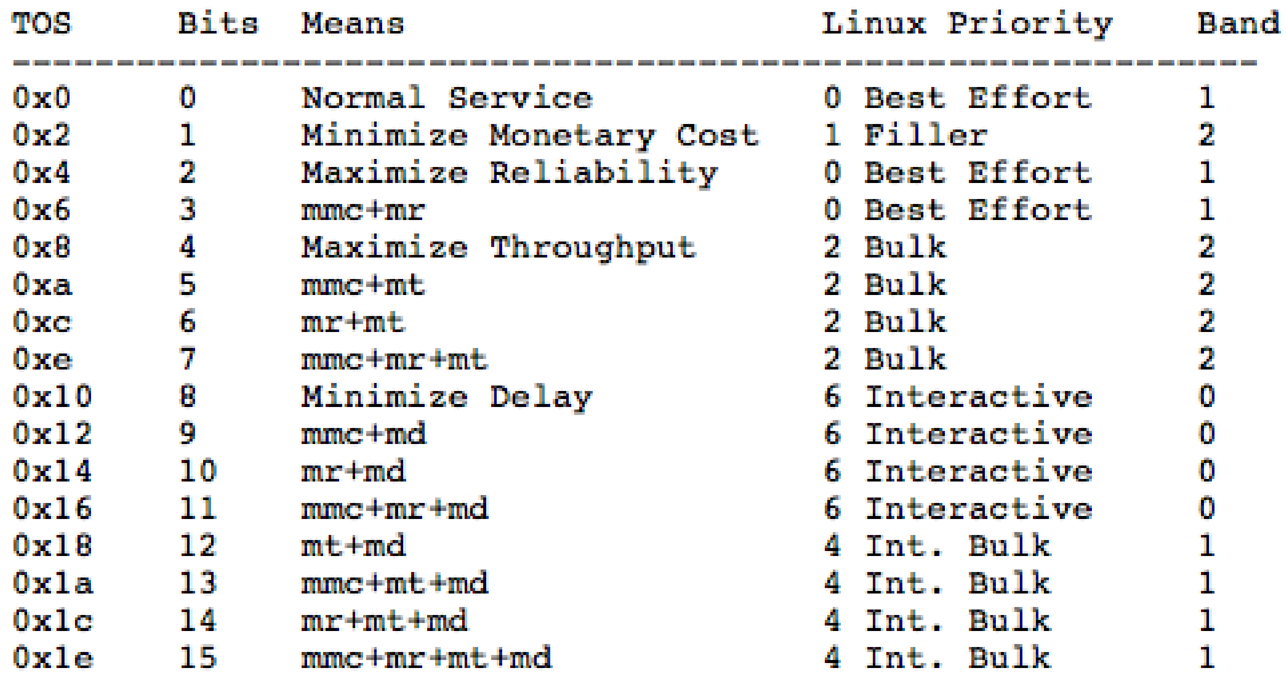
```
   0     1     2     3     4     5     6     7
+-----+-----+-----+-----+-----+-----+-----+-----+
|                 |                       |     |
|   PRECEDENCE    |          TOS          | MBZ |
|                 |                       |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

The four ToS bits (the 'ToS field') are defined as:



Binary	Decimcal	Meaning
1000	8	Minimize delay (md)
0100	4	Maximize throughput (mt)
0010	2	Maximize reliability (mr)
0001	1	Minimize monetary cost (mmc)
0000	0	Normal Service

As there is 1 bit to the right of these four bits, the actual value of the ToS field is double the value of the ToS bits. Running **tcpdump -v -v** shows you the value of the entire ToS field, not just the four bits. It is the value you see in the first column of this table:



TOS	Bits	Means	Linux Priority	Band
0x0	0	Normal Service	0 Best Effort	1
0x2	1	Minimize Monetary Cost	1 Filler	2
0x4	2	Maximize Reliability	0 Best Effort	1
0x6	3	mmc+mr	0 Best Effort	1
0x8	4	Maximize Throughput	2 Bulk	2
0xa	5	mmc+mt	2 Bulk	2
0xc	6	mr+mt	2 Bulk	2
0xe	7	mmc+mr+mt	2 Bulk	2
0x10	8	Minimize Delay	6 Interactive	0
0x12	9	mmc+md	6 Interactive	0
0x14	10	mr+md	6 Interactive	0
0x16	11	mmc+mr+md	6 Interactive	0
0x18	12	mt+md	4 Int. Bulk	1
0x1a	13	mmc+mt+md	4 Int. Bulk	1
0x1c	14	mr+mt+md	4 Int. Bulk	1
0x1e	15	mmc+mr+mt+md	4 Int. Bulk	1

Lots of numbers. The second column contains the value of the relevant four ToS bits, followed by their translated meaning. For example, 15 stands for a packet wanting Minimal Monetary Cost, Maximum Reliability, Maximum Throughput AND Minimum Delay.

The fourth column lists the way the Linux kernel interprets the ToS bits, by showing to which Priority they are mapped.

The last column shows the result of the default priomap. On the command line, the default priomap looks like this:

1, 2, 2, 2, 1, 2, 0, 0 , 1, 1, 1, 1, 1, 1, 1, 1

This means that priority 4, for example, gets mapped to band number 1. The priomap also allows you to list higher priorities (> 7) which do not correspond to ToS mappings, but which are set by other means.

Moreover, differently from other non standard qdisc, pfifo_fast does not maintain statistics and does not show up in tc qdisc ls. This is because it is the automatic default in the absence of a configured qdisc.

SFQ, Stochastic Fair Queuing

Stochastic Fairness Queueing is a classless queueing discipline available for traffic control with the tc command. SFQ does not shape traffic but only schedules the transmission of packets, based on 'flows'. The goal is to ensure fairness so that each flow is able to send data in turn, thus preventing any single flow from drowning out the rest. 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 (see 6.3.1 algorithm). Perturbation (the parameter *perturb*) sets this periodicity (see 6.3.3 parameters). This may in fact have some effect in mitigating a Denial of Service attempt. SFQ is work-conserving and therefore always delivers a packet if it has one available.

So, the SFQ qdisc attempts to fairly distribute opportunity to transmit data to the network among an arbitrary number of flows.

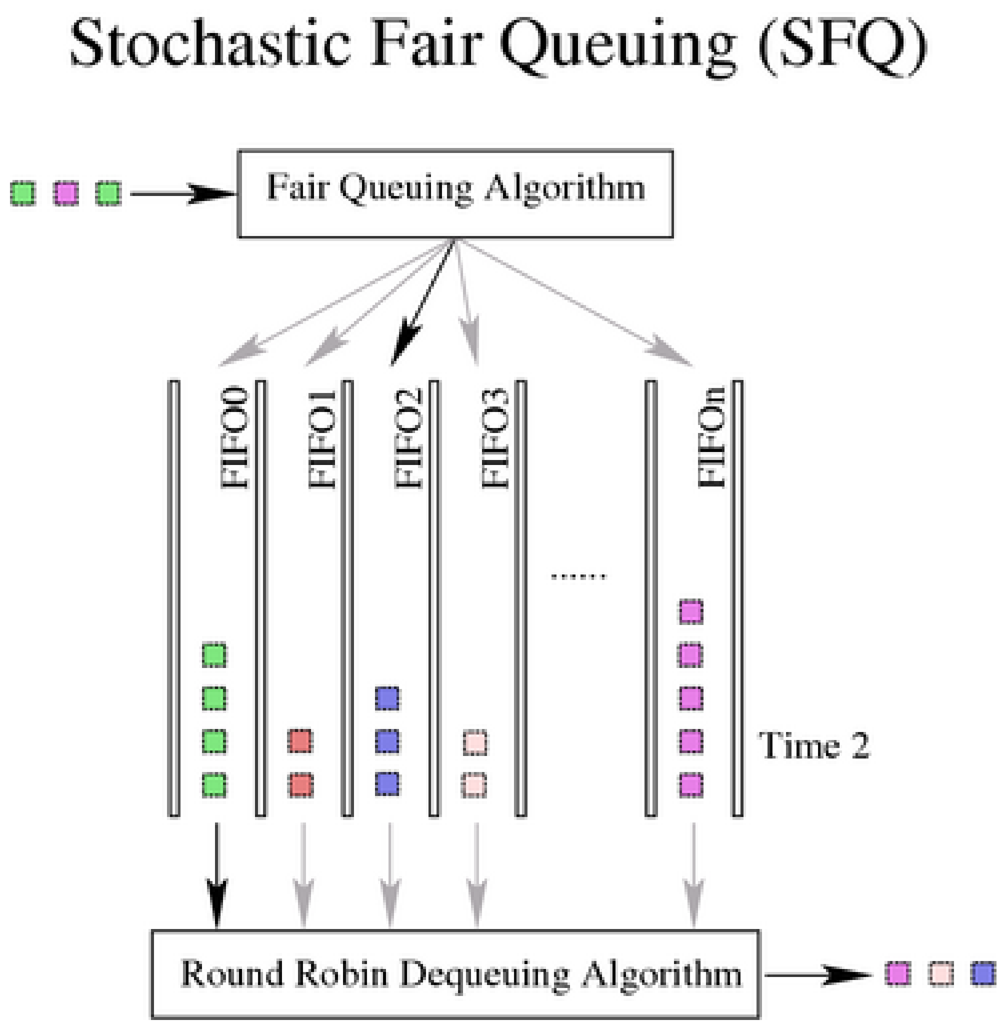

Stochastic Fair Queuing (SFQ)
Fair Queuing Algorithm
FIFO0
FIFO1
FIFO2
FIFO3
......
FIFOn
Time 2
Round Robin Dequeuing Algorithm

Figure 9: *Stochastic Fair Queuing (SFQ)*

SFQ Algorithm

On enqueueing, each packet is assigned to a hash bucket, based on the packets hash value. This hash value is either obtained from an external flow classifier (use tc filter to set them), or a default internal classifier if no external classifier has been configured. When the internal classifier is used, sfq uses

- Source address;
- Destination address;
- Source and Destination port;

if these are available.

SFQ knows about ipv4 and ipv6 and also UDP, TCP and ESP. Packets with other protocols are hashed based on the 32bits representation of their destination and source. A flow corresponds mostly to a TCP/IP connection.

Each of these buckets should represent a unique flow. Because multiple flows may get hashed to the same bucket, sfqs internal hashing algorithm may be perturbed at configurable intervals so that the unfairness lasts only for a short while. Perturbation may however cause some inadvertent packet reordering to occur. After linux-3.3, there is no packet reordering problem, but possible packet drops if rehashing hits one limit (number of flows or packets per flow)

When dequeuing, each hashbucket with data is queried in a round robin fashion.

Before linux-3.3, the compile time maximum length of the SFQ is 128 packets, which can be spread over at most 128 buckets of 1024 available. In case of overflow, tail-drop is performed on the fullest bucket, thus maintaining fairness.

After linux-3.3, maximum length of SFQ is 65535 packets, and divisor limit is 65536. In case of overflow, tail-drop is performed on the fullest bucket, unless headdrop was requested.

Synopsis

```
tc  qdisc ... [divisor hashtablesize] [limit packets] [perturb seconds] [quantum b
```

Parameters

- *divisor:* can be used to set a different hash table size, available from kernel 2.6.39 onwards. The specified divisor must be a power of two and cannot be larger than 65536. Default value is 1024.
- *limit:* upper limit of the SFQ. Can be used to reduce the default length of 127 packets. After linux-3.3, it can be raised.
- *depth:* limit of packets per flow (after linux-3.3). Default to 127 and can be lowered.
- *perturb:* interval in seconds for queue algorithm perturbation. Defaults to 0, which means that no perturbation occurs. Do not set too low for each perturbation may cause some packet reordering or losses. Advised value is 60. This value has no effect when external flow classification is used. Its better to increase divisor value to lower risk of hash collisions.
- *quantum:* amount of bytes a flow is allowed to dequeue during a round of the round robin process. Defaults to the MTU of the interface which is also the advised value and the minimum value.
- *flows:* after linux-3.3, it is possible to change the default limit of flows. Default value is 127.
- *headdrop:* default SFQ behavior is to perform tail-drop of packets from a flow. You can ask a headdrop instead, as this is known to provide a better feedback for TCP flows
- *redflowlimit:* configure the optional RED module on top of each SFQ flow. Random Early Detection principle is to perform packet marks or drops in a probabilistic way. Redflowlimit configures the hard limit on the real (not average) queue size per SFQ flow in bytes.
- *min:* average queue size at which marking becomes a possibility. Defaults to max /3.
- *max:* at this average queue size, the marking probability is maximal. Defaults to redflowlimit /4.
- *probability:* maximum probability for marking, specified as a floating point number from 0.0 to 1.0. Default value is 0.02.
- *avpkt:* specified in bytes. Used with burst to determine the time constant for average queue size calculations. Default value is 1000.

- *burst:* used for determining how fast the average queue size is influenced by the real queue size. Default value is: (2 * min + max) / (3 * avpkt).
- *ecn:* 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 depth packets.
- *harddrop:* if average flow queue size is above max bytes, this parameter forces a drop instead of ecn marking.

Example and Usage

To attach to device ppp0:

```
$ tc qdisc add dev ppp0 root sfq
```

Please note that SFQ, like all non-shaping (work-conserving) qdiscs, is only useful if it owns the queue. This is the case when the link speed equals the actually available bandwidth. This holds for regular phone modems, ISDN connections and direct non-switched ethernet links.

Most often, cable modems and DSL devices do not fall into this category. The same holds for when connected to a switch and trying to send data to a congested segment also connected to the switch. In this case, the effective queue does not reside within Linux and is therefore not available for scheduling. Embed SFQ in a classful qdisc to make sure it owns the queue.

It is possible to use external classifiers with sfq, for example to hash traffic based only on source/destination ip addresses

```
$ tc filter add ... flow hash keys src,dst perturb 30 divisor 1024
```

Note that the given divisor should match the one used by sfq. If you have changed the sfq default of 1024, use the same value for the flow hash filter, too.

Example 7. SFQ with optional RED mode

```
[root@leander]# tc qdisc add dev eth0 parent 1:1 handle 10: sfq limit 3000 flows 5
```

Example 8. 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 the section called "ESFQ, Extended Stochastic Fair Queuing" for an SFQ qdisc with more exposed parameters for the user to manipulate.

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 9. 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.

RED,Random Early Drop

Description

Random Early Detection is a classless qdisc which manages 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 if ECN is configured, marking either means dropping or purely marking a packet as overlimit.

Algorithm

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.

Synopsis

```
$ tc  qdisc ... red limit bytes [min bytes] [max bytes] avpkt bytes [burst packets
```

Parameters

- *min:* average queue size at which marking becomes a possibility. Defaults to max/3.
- *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. Default to limit/4.
- *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). Default is 0.02.
- *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 (min+min+max)/(3*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. Default is 10Mbit.
- *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. Recommended.
- *harddrop:* If average flow queue size is above max bytes, this parameterv forces a drop instead of ecn marking.
- *adaptive:* (added in linux-3.3) Sets RED in adaptive mode as described in http://icir.org/floyd/papers/adaptiveRed.pdf. Goal of Adaptive RED is to make 'probability' dynamic value between 1% and 50% to reach the target average queue: (max - min) / 2.

Example

```
# tc qdisc add dev eth0 parent 1:1 handle 10: red limit 400000 min 30000 max 90000
```

GRED, Generic Random Early Drop

Generalized RED is used in DiffServ implementation and it has virtual queue (VQ) within physical queue. Currently, the number of virtual queues is limited to 16.

GRED is configured in two steps. First the generic parameters are configured to select the number of virtual queues DPs and whether to turn on the RIO-like buffer sharing scheme. Also at this point, a default virtual queue is selected.

The second step is used to set parameters for individual virtual queues.

Synopsis

```
... gred DP drop-probability limit BYTES min BYTES max BYTES avpkt BYTES burst PAC

OR

... gred setup DPs "num of DPs" default "default DP" [grio]
```

Parameter

- *setup:* identifies that this is a generic setup for GRED;
- *DPs:* is the number of virtual queues;
- *default:* specifies default virtual queue;
- *grio:* turns on the RIO-like buffering scheme;
- *limit:* defines the virtual queue “physical” limit in bytes;
- *min:* defines the minimum threshold value in bytes;
- *max:* defines the maximum threshold value in bytes;
- *avpkt:* is the average packet size in bytes;
- *bandwidth:* is the wire-speed of the interface;
- *burst:* is the number of average-sized packets allowed to burst;
- *probability:* defines the drop probability in the range (0…);
- *DP:* identifies the virtual queue assigned to these parameters;
- *prio:* identifies the virtual queue priority if grio was set in general parameters;

TBF, Token Bucket Filter

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

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

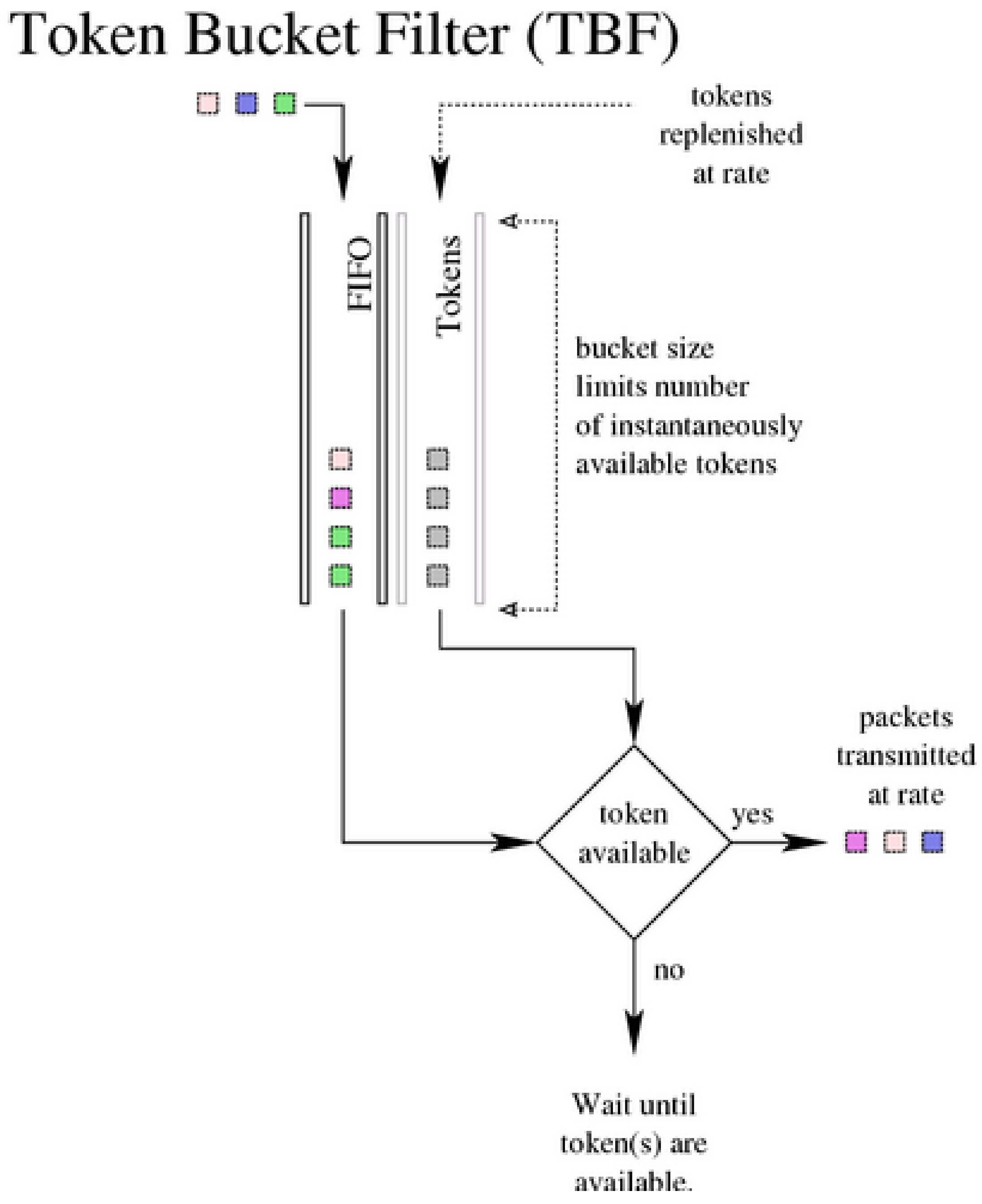

Token Bucket Filter (TBF)
tokens replenished at rate
FIFO
Tokens
bucket size limits number of instantaneously available tokens
token available
yes
packets transmitted at rate
no
Wait until token(s) are available.

Figure 10: *Token Bucket Filter (TBF)*

Algorithm

As the name implies, traffic is filtered based on the expenditure of tokens (non-work-conserving). Tokens roughly correspond to bytes, with the additional constraint that each packet consumes some tokens, no matter how small it is. This reflects the fact that even a zero-sized packet occupies the link for some time. On creation, the TBF is stocked with tokens which correspond to the amount of traffic that can be burst in one go. Tokens arrive at a steady rate, until the bucket is full. If no tokens are available, packets are queued, up to a configured limit. The TBF now calculates the token deficit, and throttles until the first packet in the queue can be sent. If it is not acceptable to burst out packets at maximum speed, a peakrate can be configured to limit the speed at which the bucket empties. This peakrate is implemented as a second TBF with a very small bucket, so that it doesn't burst.

Parameters

- *limit or latency:* 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). These two parameters are mutually exclusive.

- *Burst:* also known as buffer or maxburst. 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. The minimum buffer size can be calculated by dividing the rate by HZ.

  Token usage calculations are performed using a table which by default has a resolution of 8 packets. This resolution can be changed by specifying the cell size with the burst. For example, to specify a 6000 byte buffer with a 16 byte cell size, set a burst of 6000/16. You will probably never have to set this. Must be an integral power of 2.

- *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 (specified in bytes) for a packet. Defaults to zero.

- *Rate:* the speed knob. Furthermore, if a peakrate is desired, the following parameters are available:

- *peakrate:* maximum depletion rate of the bucket. The peakrate does not need to be set, it is only necessary if perfect millisecond timescale shaping is required.

- *mtu/minburst:* specifies the size of the peakrate bucket. For perfect accuracy, should be set to the MTU of the interface. If a peakrate is needed, but some burstiness is acceptable, this size can be raised. A 3000 byte minburst allows around 3mbit/s of peakrate, given 1000 byte packets. Like the regular burstsize you can also specify a cell size.

Example

Example 10. 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 r
```

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

HTB, Hierarchical Token Bucket

HTB is meant as a more understandable and intuitive replacement for the CBQ (see chapter 7.4) 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 oftraffic 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.

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

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

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

Example 11. tc usage for HTB

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

... class add ... htb rate R1 burst B1 [prio P] [slot S] [pslot PS]
```

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

TC HTB version 3.3
```

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 [http://luxik.cdi.cz/~devik/qos/htb/] for an **iproute2** patch to **tc**.

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.

Borrowing

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

Table 2. HTB class states and potential actions taken

type of class	**class state**	**HTB internal state**	**action taken**
leaf	< *`rate`*	*`HTB_CAN_SEND`*	Leaf class will dequeue queued bytes up to available tokens (no more than burst packets)
leaf	> *`rate`*, < *`ceil`*	*`HTB_MAY_BORROW`*	Leaf class will attempt to borrow tokens/ctokens from parent class. If tokens are available, they will be lent in *`quantum`* increments

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

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

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

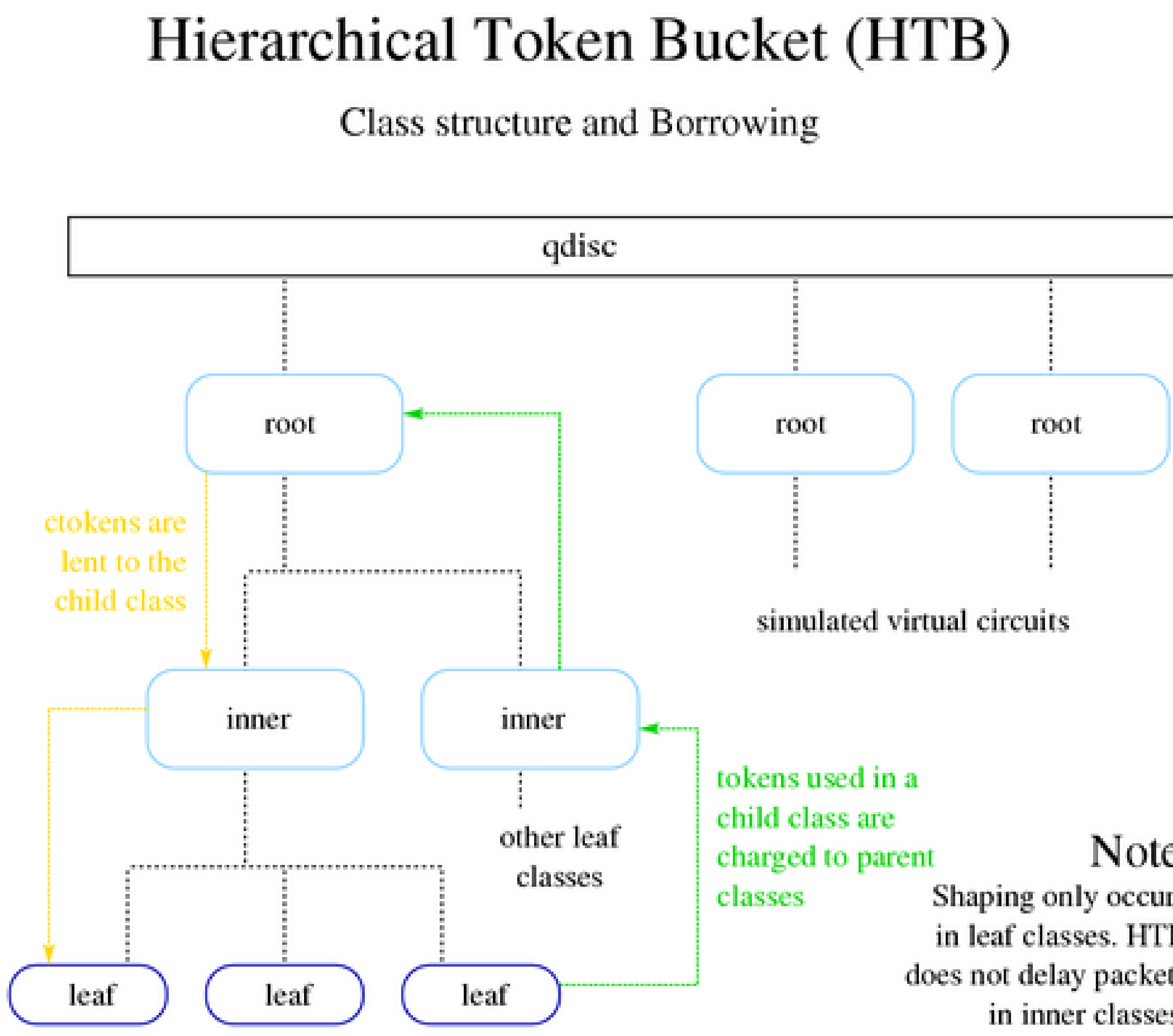

Hierarchical Token Bucket (HTB)
Class structure and Borrowing
qdisc
root
root
root
ctokens are lent to the child class
inner
inner
simulated virtual circuits
tokens used in a child class are charged to parent classes
other leaf classes
leaf
leaf
leaf
Note
Shaping only occur
in leaf classes. HT
does not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

Figure 11: *Hierarchical Token Bucket (HTB)*

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

HTB class parameters

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

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

`ceil`	Used to set the maximum desired speed to which to limit the transmitted traffic. The borrowing model should illustrate how this parameter is used. This can be considered the equivalent of “burstable bandwidth”.
`burst`	This is the size of the *`rate`* bucket (see Tokens and buckets). HTB will dequeue *`burst`* bytes before awaiting the arrival of more tokens.
`cburst`	This is the size of the *`ceil`* bucket (see Tokens and buckets). HTB will dequeue *`cburst`* bytes before awaiting the arrival of more ctokens.
`quantum`	This is a key parameter used by HTB to control borrowing. Normally, the correct *`quantum`* is calculated by HTB, not specified by the user. Tweaking this parameter can have tremendous effects on borrowing and shaping under contention, because it is used both to split traffic between children classes over *`rate`* (but below *`ceil`*) and to transmit packets from these same classes.
`r2q`	Also, usually calculated for the user, *`r2q`* is a hint to HTB to help determine the optimal *`quantum`* for a particular class.
`mtu`	
`prio`	In the round-robin process, classes with the lowest priority field are tried for packets first. Mandatory field.
`prio`	Place of this class within the hierarchy. If attached directly to a qdisc and not to another class, minor can be omitted. Mandatory field.
`prio`	Like qdiscs, classes can be named. The major number must be equal to the major number of the qdisc to which it belongs. Optional, but needed if this class is going to have children.

HTB root parameters

The root of a HTB qdisc class tree has the following parameters:

`parent major:minor | root`	This mandatory parameter determines the place of the HTB instance, either at the root of an interface or within an existing class.
`handle major:`	Like all other qdiscs, the HTB can be assigned a handle. Should consist only of a major number, followed by a colon. Optional, but very useful if classes will be generated within this qdisc.
`default minor-id`	Unclassified traffic gets sent to the class with this minor-id.

Rules

Below are some general guidelines to using HTB culled from http://www.docum.org/docum.org/ and the (new) LARTC mailing list [http://www.spinics.net/lists/lartc/] (see also the (old) LARTC mailing list archive [http://mailman.ds9a.nl/mailman/listinfo/lartc/]). 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 the section called “Shaping”.
- Because HTB does not shape in any class except the leaf class, the sum of the *`rate`*s of leaf classes should not exceed the *`ceil`* of a parent class. Ideally, the sum of the *`rate`*s of the children classes

would match the `rate` of the parent class, allowing the parent class to distribute leftover bandwidth (`ceil` - `rate`) among the children classes.

This key concept in employing HTB bears repeating. Only leaf classes actually shape packets; packets are only delayed in these leaf classes. The inner classes (all the way up to the root class) exist to define how borrowing/lending occurs (see also the section called "Borrowing").

- The `quantum` is only only used when a class is over `rate` but below `ceil`.
- The `quantum` should be set at MTU or higher. HTB will dequeue a single packet at least per service opportunity even if `quantum` is too small. In such a case, it will not be able to calculate accurately the real bandwidth consumed [9].
- Parent classes lend tokens to children in increments of `quantum`, so for maximum granularity and most instantaneously evenly distributed bandwidth, `quantum` should be as low as possible while still no less than MTU.
- A distinction between tokens and ctokens is only meaningful in a leaf class, because non-leaf classes only lend tokens to child classes.
- HTB borrowing could more accurately be described as "using".

Classification

Like see before, within the one HTB instance many classes may exist. Each of these classes contains another qdisc, by default tc-pfifo.When enqueueing a packet, HTB starts at the root and uses various methods to determine which class should receive the data. In the absence of uncommon configuration options, the process is rather easy. At each node we look for an instruction, and then go to the class the instruction refers us to. If the class found is a barren leaf-node (without children), we enqueue the packet there. If it is not yet a leaf node, we do the whole thing over again starting from that node.

The following actions are performed, in order at each node we visit, until one sends us to another node, or terminates the process.

- Consult filters attached to the class. If sent to a leafnode, we are done. Otherwise, restart.
- If none of the above returned with an instruction, enqueue at this node.

This algorithm makes sure that a packet always ends up somewhere, even while you are busy building your configuration.

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 [http://klaus.geekserver.net/hfsc/hfsc.html] or read a translation into English, HFSC Scheduling with Linux [http://linux-ip.net/tc/hfsc.en/]. The original research article, A Hierarchical Fair Service Curve Algorithm For Link-Sharing, Real-Time and Priority Services [http://acm.org/sigcomm/sigcomm97/program.html#ab011], also remains available.

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.

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. PRIO is a scheduler and never delays packets - it is a work-conserving qdisc, though the qdiscs contained in the classes may not be

Algorithm

On creation with **tc qdisc add**, a fixed number of bands is created. Each band is a class, although is not possible to add classes with **tc class add**. The number of bands to be created is fixed at the creation of the qdisc itself.

When dequeueing packets, band 0 is always checked first. If it has no packet to dequeue, then PRIO will try band 1, and so onwards. Maximum reliability packets should therefore go to band 0, minimum delay to band 1 and the rest to band 2.

As the PRIO qdisc itself will have minor number 0, band 0 is actually major:1, band 1 is major:2, etc. For major, substitute the major number assigned to the qdisc on 'tc qdisc add' with the handle parameter.

Synopsis

```
$ tc qdisc ... dev dev ( parent classid | root) [ handle major: ] prio [bands band
```

Classification

Three methods are available to determine the target band in which a packet will be enqueued.

- *From userspace*, a process with sufficient privileges can encode the destination class directly with SO_PRIORITY.
- *Programmatically*, a **tc filter** attached to the root qdisc can point any traffic directly to a class.
- And, typically, *with reference to the priomap*, a packet's priority, is derived from the Type of Service (ToS) assigned to the packet.

Only the priomap is specific to this qdisc.

Configurable parameters

bands	total number of distinct bands. If changed from the default of 3, priomap must be updated as well.
priomap	a tc filter attached to the root qdisc can point traffic directly to a class

The *priomap* specifies how this qdisc determines how a packet maps to a specific band. Mapping occurs based on the value of the ToS octet of a packet.

```
   0     1     2     3     4     5     6     7
+-----+-----+-----+-----+-----+-----+-----+-----+
|   PRECEDENCE    |          ToS          | MBZ |   RFC 791
+-----+-----+-----+-----+-----+-----+-----+-----+

   0     1     2     3     4     5     6     7
+-----+-----+-----+-----+-----+-----+-----+-----+
|    DiffServ Code Point (DSCP)     |  (unused) |   RFC 2474
+-----+-----+-----+-----+-----+-----+-----+-----+
```

The four ToS bits from the (the 'ToS field') are defined slightly differently in RFC 791 and RFC 2474. The later RFC supersedes the definitions of the former, but not all software, systems and terminology have caught up to that change. So, often packet analysis programs will still refer to Type of Service (ToS) instead of DiffServ Code Point (DSCP).

Table 3. RFC 791 [https://tools.ietf.org/rfc/rfc791.txt] interpretation of IP ToS header

Binary	Decimal	Meaning
1000	8	Minimize delay (md)
0100	4	Maximize throughput (mt)
0010	2	Maximize reliability (mr)
0001	1	Minimize monetary cost (mmc)
0000	0	Normal Service

As there is 1 bit to the right of these four bits, the actual value of the ToS field is double the value of the ToS bits. Running **tcpdump -v -v** shows you the value of the entire ToS field, not just the four bits. It is the value you see in the first column of this table:

Table 4. Mapping ToS value to priomap band

ToS Field	ToS Bits	Meaning	Linux Priority	Band
0x0	0	Normal Service	0 Best Effort	1
0x2	1	Minimize Monetary Cost (mmc)	1 Filler	2
0x4	2	Maximize Reliability (mr)	0 Best Effort	1
0x6	3	mmc+mr	0 Best Effort	1
0x8	4	Maximize Throughput (mt)	2 Bulk	2
0xa	5	mmc+mt	2 Bulk	2
0xc	6	mr+mt	2 Bulk	2
0xe	7	mmc+mr+mt	2 Bulk	2
0x10	8	Minimize Delay (md)	6 Interactive	0
0x12	9	mmc+md	6 Interactive	0
0x14	10	mr+md	6 Interactive	0
0x16	11	mmc+mr+md	6 Interactive	0

ToS Field	ToS Bits	Meaning	Linux Priority	Band
0x18	12	mt+md	4 Int. Bulk	1
0x1a	13	mmc+mt+md	4 Int. Bulk	1
0x1c	14	mr+mt+md	4 Int. Bulk	1
0x1e	15	mmc+mr+mt+md	4 Int. Bulk	1

The second column contains the value of the relevant four ToS bits, followed by their translated meaning. For example, 15 stands for a packet wanting Minimal Monetary Cost, Maximum Reliability, Maximum Throughput AND Minimum Delay.

The fourth column lists the way the Linux kernel interprets the ToS bits, by showing to which Priority they are mapped.

The last column shows the result of the default priomap. On the command line, the default priomap looks like this:

1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1

This means that priority 4, for example, gets mapped to band number 1. The priomap also allows you to list higher priorities (> 7) which do not correspond to ToS mappings, but which are set by other means.

Classes

PRIO classes cannot be configured further - they are automatically created when the PRIO qdisc is attached. Each class however can contain yet a further qdisc.

Bugs

Large amounts of traffic in the lower bands can cause starvation of higher bands. Can be prevented by attaching a shaper to these bands to make sure they cannot dominate the link.

CBQ, Class Based Queuing (CBQ)

Class Based Queuing (CBQ) is the classic implementation (also called venerable) of a traffic control system. CBQ is a classful qdisc that implements a rich link sharing hierarchy of classes. It contains shaping elements as well as prioritizing capabilities. Shaping is performed by calculating link idle time based on the timing of dequeue events and knowledge of the underlying link layer bandwidth.

Shaping Algorithm

Shaping is done using link idle time calculations, and actions taken if these calculations deviate from set limits.

When shaping a 10mbit/s connection to 1mbit/s, the link will be idle 90% of the time. If it isn't, it needs to be throttled so that it is idle 90% of the time.

From the kernel's perspective, this is hard to measure, so CBQ instead computes idle time from the number of microseconds that elapse between requests from the device driver for more data. Combined with the knowledge of packet sizes, this is used to approximate how full or empty the link is.

This is rather circumspect and doesn't always arrive at proper results. The physical link bandwidth may be ill defined in case of 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 not defined.

During operations, the effective idletime is measured using an exponential weighted moving average (EWMA). This calculation of activity against idleness values recent packets exponentially more than predecessors. The EWMA is an effective calculation to deal with the problem that a system is either active or inactive. For example, the Unix system load average is calculated in the same way.

The calculated idle time is subtracted from the EWMA measured one, the resulting number is called 'avgidle'. A perfectly loaded link has an avgidle of zero: packets arrive exactly at the calculated interval.

An overloaded link has a negative avgidle and if it gets too negative, CBQ throttles and is then 'overlimit'. Conversely, an idle link might amass a huge avgidle, which would then allow infinite bandwidths after a few hours of silence. To prevent this, avgidle is capped at maxidle.

If overlimit, in theory, the CBQ could throttle itself for exactly the amount of time that was calculated to pass between packets, and then pass one packet, and throttle again. Due to timer resolution constraints, this may not be feasible, see the minburst parameter below.

Classification

Under one installed CBQ qdisc many classes may exist. Each of these classes contains another qdisc, by default tc-pfifo.

When enqueueing a packet, CBQ starts at the root and uses various methods to determine which class should receive the data. If a verdict is reached, this process is repeated for the recipient class which might have further means of classifying traffic to its children, if any. CBQ has the following methods available to classify a packet to any child classes.

- skb>priority class encoding. Can be set from userspace by an application with the SO_PRIORITY setsockopt. The skb->priority class encoding only applies if the skb->priority holds a major:minor handle of an existing class within this qdisc.

- *tc filters attached to the class.*

- *The defmap of a class*, as set with the split and defmap parameters. The defmap may contain instructions for each possible Linux packet priority.

Each class also has a level. Leaf nodes, attached to the bottom of theclass hierarchy, have a level of 0.

Classification Algorithm

Classification is a loop, which terminates when a leaf class is found. At any point the loop may jump to the fallback algorithm. The loop consists of the following steps:

- If the packet is generated locally and has a valid classid encoded within its skb->priority, choose it and terminate.

- Consult the tc filters, if any, attached to this child. If these return a class which is not a leaf class, restart loop from he class returned. If it is a leaf, choose it and terminate.

- If the tc filters did not return a class, but did return a classid, try to find a class with that id within this qdisc. Checkif the found class is of a lower level than the current class. If so, and the returned class is not a leaf node, restart the loop at the found class. If it is a leaf node, terminate. If we found an upward reference to a higher level, enter the fallback algorithm.

- If the tc filters did not return a class, nor a valid reference to one, consider the minor number of the reference to be the priority. Retrieve a class from the defmap of this class for the priority. If this did not contain a class, consult the defmap of this class for the BEST_EFFORT class. If this is an upward reference, or no BEST_EFFORT class was defined, enter the fallback algorithm. If a valid class was found, and it is not a leaf node, restart the loop at this class. If it is a leaf, choose it and terminate. If neither the priority distilled from the classid, nor the BEST_EFFORT priority yielded a class, enter the fallback algorithm.

The fallback algorithm resides outside of the loop and is as follows.

- Consult the defmap of the class at which the jump to fallback occured. If the defmap contains a class for the priority of the class (which is related to the ToS field), choose this class and terminate.
- Consult the map for a class for the BEST_EFFORT priority. If found, choose it, and terminate.
- Choose the class at which break out to the fallback algorithm occurred. Terminate.

The packet is enqueued to the class which was chosen when either algorithm terminated. It is therefore possible for a packet to be enqueued not at a leaf node, but in the middle of the hierarchy.

Link Sharing Algorithm

When dequeuing for sending to the network device, CBQ decides which of its classes will be allowed to send. It does so with a Weighted Round Robin process in which each class with packets gets a chance to send in turn. The WRR process starts by asking the highest priority classes (lowest numerically - highest semantically) for packets, and will continue to do so until they have no more data to offer, in which case the process repeats for lower priorities.

Each class is not allowed to send at length though, they can only dequeue a configurable amount of data during each round.

If a class is about to go overlimit, and it is not bounded it will try to borrow avgidle from siblings that are not isolated. This process is repeated from the bottom upwards. If a class is unable to borrow enough avgidle to send a packet, it is throttled and not asked for a packet for enough time for the avgidle to increase above zero.

Root Parameters

The root qdisc of a CBQ class tree has the following parameters:

`parent` *`root`* | *`major:minor`*	this mandatory parameter determines the place of the CBQ instance, either at the root of an interface or within an existing class.
`handle` *`major:`*	like all other qdiscs, the CBQ can be assigned a handle. Should consist only of a major number, followed by a colon. This parameter is optional.
`avpkt` *`bytes`*	for calculations, the average packet size must be known. It is silently capped at a minimum of 2/3 of the interface MTU. This parameter is mandatory.
`bandwidth` *`rate`*	underlying available bandwidth; to determine the idle time, CBQ must know the bandwidth of your A) desired target bandwidth, B) underlying physical interface or C) parent qdisc. This is a vital parameter, more about it later. This parameter is mandatory.

`cell` *`size`*	the cell size determines the granularity of packet transmission time calculations. Must be an integral power of 2, defaults to 8.
`mpu` *`bytes`*	a zero sized packet may still take time to transmit. This value is the lower cap for packet transmission time calculations - packets smaller than this value are still deemed to have this size. Defaults to 0.
`ewma` *`log`*	CBQ calculates idleness using an Exponentially Weighted Moving Average (EWMA) which smooths out measurements easily accommodating short bursts. The *`log`* value determines how much smoothing occurs. Lower values imply greater sensitivity. Must be between 0 and 31. Defaults to 5.

A CBQ qdisc does not shape out of its own accord. It only needs to know certain parameters about the underlying link. Actual shaping is done in classes.

Class Parameters

Classes have a lot of parameters to configure their operation.

`parent` *`major:minor`*	place of this class within the hierarchy. If attached directly to a qdisc and not to another class, minor can be omitted. This parameter is mandatory.
`classid` *`major:minor`*	like qdiscs, classes can be named. The major number must be equal to the major number of the qdisc to which it belongs. Optional, but needed if this class is going to have children.
`weight` *`weightvalue`*	when dequeuing to the lower layer, classes are tried for traffic in a round-robin fashion. Classes with a higher configured qdisc will generally have more traffic to offer during each round, so it makes sense to allow it to dequeue more traffic. All weights under a class are normalized, so only the ratios matter. Defaults to the configured rate, unless the priority of this class is maximal, in which case it is set to 1.
`allot` *`bytes`*	allot specifies how many bytes a qdisc can dequeue during each round of the process. This parameter is weighted using the renormalized class weight described above.
`priority` *`priovalue`*	in the round-robin process, classes with the lowest priority field are tried for packets first. This parameter is mandatory.
`rate` *`bitrate`*	maximum aggregated rate at which this class (children inclusive) can transmit. The bitrate is specified using the **tc** way of specifying rates (e.g. '1544kbit'). This parameter is mandatory.
`bandwidth` *`bitrate`*	this is different from the bandwidth specified when creating a parent CBQ qdisc. The CBQ class `bandwidth` parameter is only used to determine maxidle and offtime, which, in turn, are only calculated when specifying maxburst or minburst. Thus, this parameter is only required if specifying `maxburst` or `minburst`.
`maxburst` *`packetcount`*,	this number of packets is used to calculate maxidle so that when avgidle is at maxidle, this number of average packets can be burst before avgidle drops to 0. Set it higher to be more tolerant of bursts. You can't set maxidle directly, only via this parameter.

`minburst` *`packetcount`*	as mentioned before, CBQ needs to throttle in case of overlimit. The ideal solution is to do so for exactly the calculated idle time, and pass 1 packet. However, Unix kernels generally have a hard time scheduling events shorter than 10ms, so it is better to throttle for a longer period, and then pass minburst packets in one go, and then sleep minburst times longer. The time to wait is called the offtime. Higher values of minburst lead to more accurate shaping in the long term, but to bigger bursts at millisecond timescales.
`minidle` *`microseconds`*,	*minidle:* if avgidle is below 0, we are overlimits and need to wait until avgidle will be big enough to send one packet. To prevent a sudden burst from shutting down the link for a prolonged period of time, avgidle is reset to minidle if it gets too low. Minidle is specified in negative microseconds, so 10 means that avgidle is capped at -10us.
`bounded` | `borrow`,	identifies a borrowing policy. Either the class will try to `borrow` bandwidth from its siblings or it will consider itself `bounded`. Mutually exclusive.
`isolated` | `sharing`	identifies a sharing policy. Either the class will engage in a `sharing` policy toward its siblings or it will consider itself `isolated`. Mutually exclusive.

split major:minor and defmap bitmap[/bitmap]: if consulting filters attached to a class did not give a verdict, CBQ can also classify based on the packet's priority. There are 16 priorities available, numbered from 0 to 15. The defmap specifies which priorities this class wants to receive, specified as a bitmap. The Least Significant Bit corresponds to priority zero. The split parameter tells CBQ at which class the decision must be made, which should be a (grand)parent of the class you are adding.

As an example, 'tc class add ... classid 10:1 cbq .. split 10:0 defmap c0' configures class 10:0 to send packets with priorities 6 and 7 to 10:1.

The complimentary configuration would then be: 'tc class add ... classid 10:2 cbq ... split 10:0 defmap 3f' Which would send all packets 0, 1, 2, 3, 4 and 5 to 10:1.

estimator interval timeconstant: CBQ can measure how much bandwidth each class is using, which tc filters can use to classify packets with. In order to determine the bandwidth it uses a very simple estimator that measures once every interval microseconds how much traffic has passed. This again is a EWMA, for which the time constant can be specified, also in microseconds. The time constant corresponds to the sluggishness of the measurement or, conversely, to the sensitivity of the average to short bursts. Higher values mean less sensitivity.

WRR, Weighted Round Robin

This qdisc is not included in the standard kernels.

The WRR qdisc distributes bandwidth between its classes using the weighted round robin scheme. That is, like the CBQ qdisc it contains classes into which arbitrary qdiscs can be plugged. All classes which have sufficient demand will get bandwidth proportional to the weights associated with the classes. The weights can be set manually using the tc program. But they can also be made automatically decreasing for classes transferring much data.

The qdisc has a built-in classifier which assigns packets coming from or sent to different machines to different classes. Either the MAC or IP and either source or destination addresses can be used. The MAC address can only be used when the Linux box is acting as an ethernet bridge, however. The classes are automatically assigned to machines based on the packets seen.

The qdisc can be very useful at sites where a lot of unrelated individuals share an Internet connection. A set of scripts setting up a relevant behavior for such a site is a central part of the WRR distribution.

Rules, Guidelines and Approaches

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.

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.

[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.

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.

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.

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.

Scripts for use with QoS/Traffic Control

wondershaper

More to come, see wondershaper [http://lartc.org/wondershaper/].

ADSL Bandwidth HOWTO script (`myshaper`)

More to come, see myshaper [http://www.tldp.org/HOWTO/ADSL-Bandwidth-Management-HOWTO/implementation.html].

`htb.init`

More to come, see `htb.init` [http://sourceforge.net/projects/htbinit/].

`tcng.init`

More to come, see `tcng.init` [http://linux-ip.net/code/tcng/tcng.init].

cbq.init

More to come, see cbq.init [http://sourceforge.net/projects/cbqinit/].

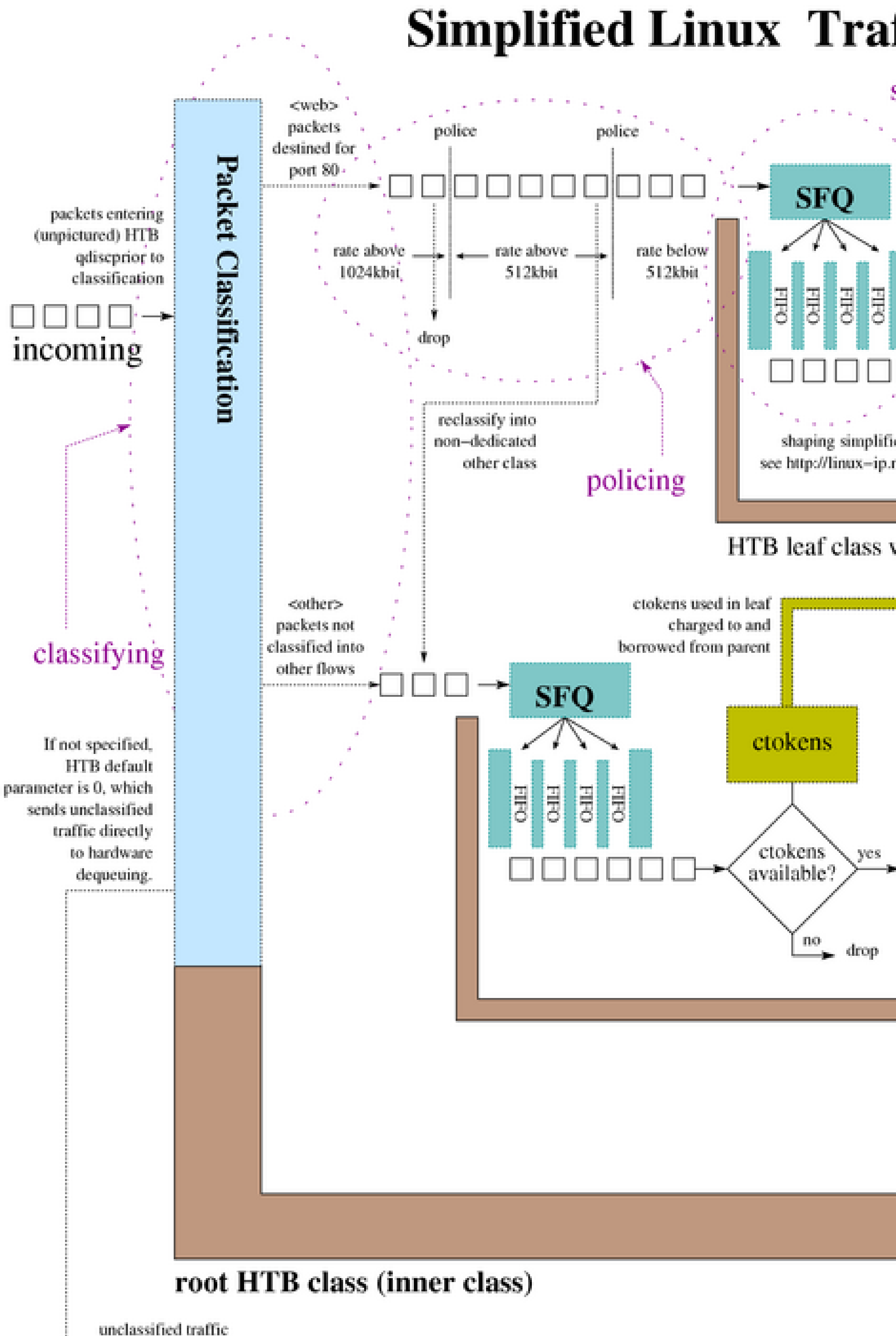
Diagram

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 [http://linux-ip.net/traffic-control/htb-class.png].

Example 12. An example HTB tcng configuration

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

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

dev eth0 {
    egress {

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

        htb () {  /* root qdisc */

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

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

            }
        }
    }
}
```


Simplified Linux Traf
Packet Classification
<web> packets destined for port 80
police
police
rate above 1024kbit
rate above 512kbit
rate below 512kbit
drop
SFQ
FIFO
FIFO
FIFO
FIFO
packets entering (unpictured) HTB qdiscprior to classification
incoming
reclassify into non-dedicated other class
policing
shaping simplifi
see http://linux-ip.n
HTB leaf class v
classifying
<other> packets not classified into other flows
ctokens used in leaf charged to and borrowed from parent
SFQ
FIFO
FIFO
FIFO
FIFO
ctokens
ctokens available?
yes
no
drop
If not specified, HTB default parameter is 0, which sends unclassified traffic directly to hardware dequeuing.
root HTB class (inner class)
unclassified traffic

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 [http://luxik.cdi.cz/~devik/qos/htb/], HTB user guide [http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm] and HTB theory [http://luxik.cdi.cz/~devik/qos/htb/manual/theory.htm] (*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 [http://www.docum.org/docum.org/] (see below).

- General Quality of Service docs [http://opalsoft.net/qos/] (*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 [http://opalsoft.net/qos/DS.htm] document among others.

- **tcng** (Traffic Control Next Generation) [http://tcng.sourceforge.net/] and **tcng** manual [http://linux-ip.net/gl/tcng/] (*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** [ftp://ftp.inr.ac.ru/ip-routing/] and **iproute2** manual [http://linux-ip.net/gl/ip-cref/] (*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 [http://luxik.cdi.cz/~devik/qos/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 [http://www.docum.org/] (*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 [http://lartc.org/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 [http://linux-ip.net/] (*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 [http://www.almesberger.net/cv/papers.html]

  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 [http://www.almesberger.net/cv/papers/tcio8.pdf] or PS [http://www.almesberger.net/cv/papers/tcio8.ps.gz] format.

- Linux DiffServ project [http://diffserv.sourceforge.net/]

  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.