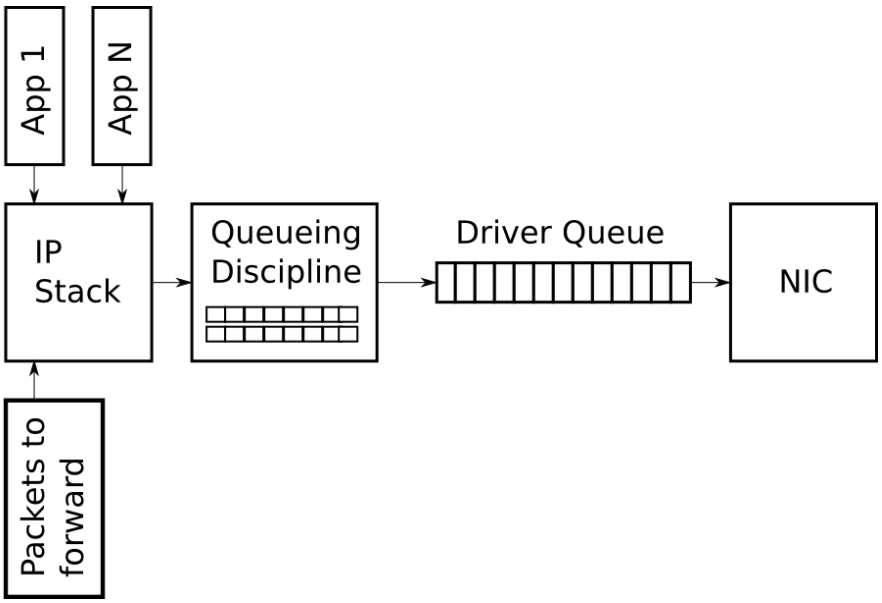
Queueing in the Linux Network Stack

[A slightly shorter and edited version of this article appeared in the July 2013 issue of Linux Journal < http://www.linuxjournal.com/content/july-2013-issue-linux-journal-networking> . Thanks to Linux Journal's great copyright policy I'm still allowed to post this on my site. Go here < https://www.pubservice.com/Subnew2page.aspx?PC=LJ> to subscribe to Linux Journal.]

Packet queues are a core component of any network stack or device. They allow for asynchronous modules to communicate, increase performance and have the side affect of impacting latency. This article aims to explain where IP packets are queued in the Linux network stack, how interesting new latency reducing features such as BQL operate and how to control buffering for reduced latency.

The figure below will be referenced throughout and modified versions presented to illustrate specific concepts.

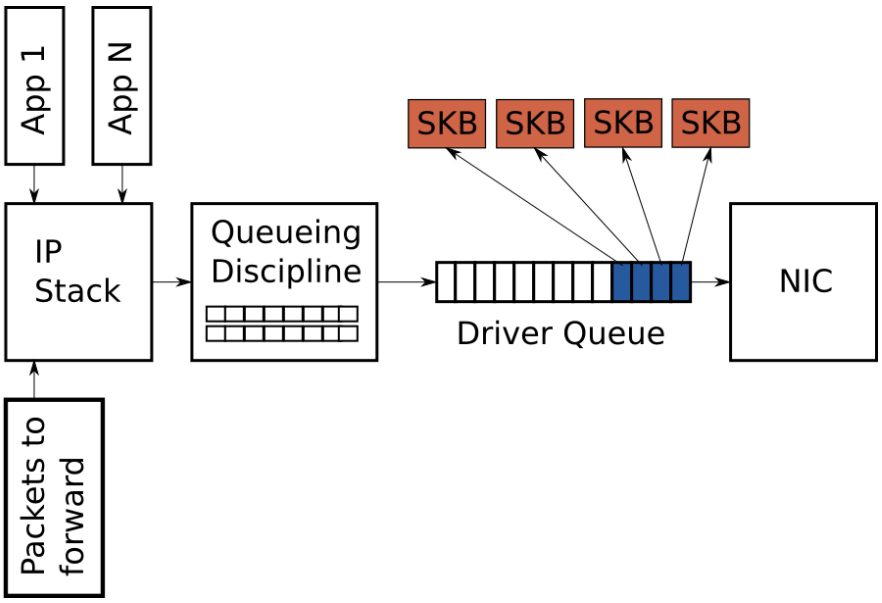


< https://www.coverfire.com/wp-content/uploads/2012/11/figure_1_v2.png>

Figure 1 – Simplified high level overview of the queues on the transmit path of the Linux network stack

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 < http://en.wikipedia.org/wiki/Circular_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) < http://vger.kernel.org/~davem/skb.html>** which hold the packet data and are used throughout the kernel.



< https://www.coverfire.com/wp-content/uploads/2012/11/figure_2_v2.png>

Figure 2 – 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

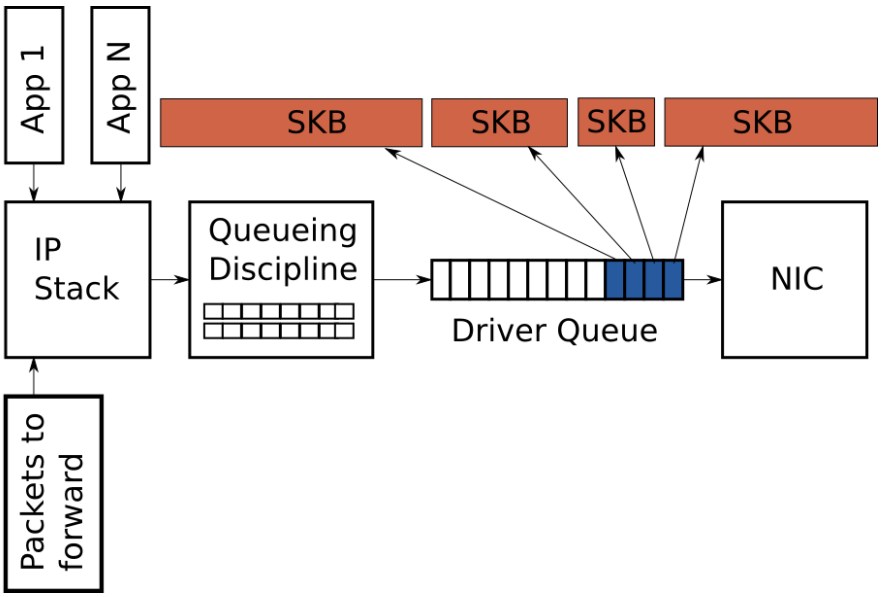
driver queue gives the IP stack a location to queue data asynchronously from the operation of the hardware. One alternative design would be for the NIC to ask the IP stack for data whenever the physical medium is ready to transmit. Since responding to this request cannot be instantaneous this design 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.

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 media. For Ethernet the default MTU is 1,500 bytes but some Ethernet networks support **Jumbo Frames <** **http://en.wikipedia.org/wiki/Jumbo_frame>** of up to 9,000 bytes. Inside 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 2,000 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 1,500 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 in contrast with figure 2.



< https://www.coverfire.com/wp-content/uploads/2012/11/figure_3_v2.png>

Figure 3 – 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.

While the rest of this article focuses on the transmit path it is worth noting that Linux also has receive side optimizations which operate similarly to TSO, UFO and GSO. These optimizations also have the goal of reducing per-packet overhead. Specifically, generic receive offload (**GRO < http://vger.kernel.org/~davem/cgi-bin/blog.cgi/2010/08/30>**) allows the NIC driver to combine received packets into a single large packet which is then passed to the IP stack. When forwarding packets, GRO allows for the original packets to be reconstructed which is necessary to maintain the end-to-end nature of IP packets. However, there is one side affect, when the large packet is broken up on the transmit side of the forwarding operation it results in several packets for the flow being queued at once. This 'micro-burst' of packets can negatively impact inter-flow latencies.

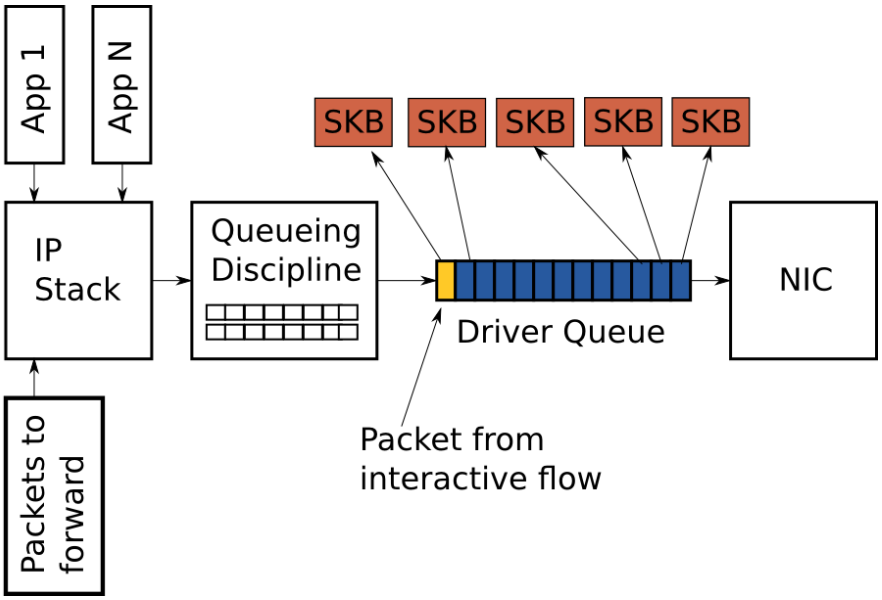
Starvation and Latency

Despite its necessity and benefits, the queue between the IP stack and the hardware 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.



< https://www.coverfire.com/wp-content/uploads/2012/11/figure_4_v2.png>

Figure 4 – Interactive packet (yellow) behind bulk flow packets (blue)

Figure 4 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.

Large latencies introduced by oversized, unmanaged buffers is known as **Bufferbloat. < http://en.wikipedia.org/wiki/Bufferbloat>** For a more detailed explanation of this phenomenon see **Controlling Queue Delay < http://queue.acm.org/detail.cfm?id=2209336>** and the **Bufferbloat < http://www.bufferbloat.net/>** project.

As the above discussion illustrates, 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.

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 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. See later parts of this article for how to accomplish this.

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 starv
        decrease LIMIT by the number of bytes not transmitted in th

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.

Queuing Disciplines (QDisc)

The driver queue is a simple first in, first out (FIFO) queue. It treats all packets equally and has no capabilities for distinguishing between packets of different flows. This design keeps the NIC driver software simple and fast. Note that more advanced Ethernet and most wireless NICs support multiple independent transmission queues but similarly each of these queues is typically a FIFO. A higher layer is responsible for choosing which transmission queue to use.

Sandwiched between the IP stack and the driver queue is the queueing discipline (QDisc) layer (see Figure 1). This layer implements the traffic management capabilities of the Linux kernel which include traffic classification, prioritization and rate shaping. The QDisc layer is configured through the somewhat opaque tc command. There are three key concepts to understand in the QDisc layer: QDiscs, classes and filters.

The QDisc is the Linux abstraction for traffic queues which are more complex than the standard FIFO queue. This interface allows the QDisc to carry out complex queue management behaviours without requiring the IP stack or the NIC driver to be

modified. By default every network interface is assigned a **pfifo_fast < http://lartc.org/howto/lartc.qdisc.classless.html>** QDisc which implements a simple three band prioritization scheme based on the TOS bits. Despite being the default, the pfifo_fast QDisc is far from the best choice because it defaults to having very deep queues (see txqueuelen below) and is not flow aware.

The second concept which is closely related to the QDisc is the class. Individual QDiscs may implement classes in order to handle subsets of the traffic differently. For example, the Hierarchical Token Bucket (**HTB < http://lartc.org/manpages/tc-htb.html>**) QDisc allows the user to configure 500Kbps and 300Kbps classes and direct traffic to each as desired. Not all QDiscs have support for multiple classes – those that do are referred to as classful QDiscs.

Filters (also called classifiers) are the mechanism used to classify traffic to a particular QDisc or class. There are many different types of filters of varying complexity. **u32 < http://www.lartc.org/lartc.html#LARTC.ADV-FILTER.U32>** being the most general and the flow filter perhaps the easiest to use. The documentation for the flow filter is lacking but you can find an example in **one of my QoS scripts < http://git.coverfire.com/?p=linux-qos-scripts.git;a=blob;f=src-3tos.sh;hb=HEAD>** .

For more detail on QDiscs, classes and filters see the **LARTC HOWTO < http://www.lartc.org/howto/>** and the tc man pages.

Buffering between the transport layer and the queueing disciplines

In looking at the previous figures you may have noticed that there are no packet queues above the queueing discipline layer. What this means is that the network stack places packets directly into the queueing discipline or else pushes back on the upper layers (eg socket buffer) if the queue is full. The obvious question that follows is what happens when the stack has a lot of data to send? This could occur as the result of a TCP connection with large congestion window or even worse an application sending UDP packets as fast as it can. The answer is that for a QDisc with a single queue, the same problem outlined in Figure 4 for the driver queue occurs. That is, a single high bandwidth or high packet rate flow can consume all of the space in the queue causing packet loss and adding significant latency to other flows. Even worse this creates another point of buffering where a **standing queue can form < http://queue.acm.org/detail.cfm?id=2209336>** which increases latency and causes problems for TCP's RTT and congestion window size calculations. Since Linux defaults to the pfifo_fast QDisc which effectively has a single queue (because most traffic is marked with TOS=0) this phenomenon is not uncommon.

As of Linux 3.6.0 (2012-09-30), the Linux kernel has a new feature called TCP Small Queues which aims to solve this problem for TCP. TCP Small Queues adds a per TCP flow limit on the number of bytes which can be queued in the QDisc and driver queue at any one time. This has the interesting side effect of causing the kernel to push back on the application earlier which allows the application to more effectively prioritize writes to the socket. At present (2012-12-28) it is still possible for single flows from other transport protocols to flood the QDisc layer.

Another partial solution to transport layer flood problem which is transport layer agnostic is to use a QDisc which has many queues, ideally one per network flow. Both the Stochastic Fairness Queueing (**SFQ < http://crpppc19.epfl.ch/cgi-bin/man/man2html?8+tc-sfq>**) and Fair Queueing with Controlled Delay (**fq_codel < http://linuxmanpages.net/manpages/fedora18/man8/tc-fq_codel.8.html>**) QDiscs fit this problem nicely as they effectively have a queue per network flow.

How to manipulate the queue sizes in Linux

Driver Queue

The **ethtool < http://linuxmanpages.net/manpages/fedora12/man8/ethtool.8.html>** 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:

```
[root@alpha net-next]# 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. Early in the Bufferbloat investigation 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.

```
[dan@alpha ~]$ 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.

Byte Queue Limits (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 < https://gettys.wordpress.com/2012/05/01/bufferbloat-goings-on/#comment-4053>** 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_queu
```

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
```

What is txqueuelen?

Often in early Bufferbloat discussions the idea of statically reducing the NIC transmission queue was mentioned. 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 (bold text):

```
[dan@alpha ~]$ 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.
          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 M
          Interrupt:23
```

```
 [dan@alpha ~]$ 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 1,000 packets which is a large amount of buffering especially at low bandwidths.

The interesting question is what exactly does this variable control? This wasn't clear to me so I spent some time spelunking in the kernel source. From what I can tell, the txqueuelen is only used as a default queue length for **some** of the queueing disciplines. Specifically:

- 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.

As an aside, I find it a little confusing that the ifconfig command shows low level details of the network interface such as the MAC address but the txqueuelen parameter refers to the higher level QDisc layer. It seems more appropriate for that ifconfig would show the driver queue size.

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

```
[root@alpha dan]# ip link set txqueuelen 500 dev eth0
```

Notice that the ip command uses "txqueuelen" but when displaying the interface details it uses "qlen" – another unfortunate inconsistency.

Queueing Disciplines

As introduced earlier, the Linux kernel has a large number of queueing disciplines (QDiscs) each of which implements its own packet queues and behaviour. Describing the details of how to configure each of the QDiscs is out of scope for this article. For full details see the tc man page (man tc). You can find details for each QDisc in 'man tc qdisc-name' (ex: 'man tc htb' or 'man tc fq_codel'). **LARTC < http://www.lartc.org/>** is also a very useful resource but is missing information on newer features.

Below are a few tips and tricks related to the tc command that may be helpful:

- The HTB QDisc implements a default queue which receives all packets if they are not classified with filter rules. Some other QDiscs such as DRR simply black hole traffic that is not classified. To see how many packets were not classified properly and were directly queued into the default HTB class see the direct_packets_stat in "tc qdisc show".
- The HTB class hierarchy is only useful for classification not bandwidth allocation. All bandwidth allocation occurs by looking at the leaves and their associated

priorities.

- The QDisc infrastructure identifies QDiscs and classes with major and minor numbers which are separated by a colon. The major number is the QDisc identifier and the minor number the class within that QDisc. The catch is that the tc command uses a hexadecimal representation of these numbers on the command line. Since many strings are valid in both hex and decimal (ie 10) many users don't even realize that tc uses hex. See one of **my tc scripts <** **http://git.coverfire.com/?p=linux-qos-scripts.git;a=blob;f=src-3tos.sh;hb=HEAD>** for how I deal with this.
- If you are using ADSL which is ATM (most DSL services are ATM based but newer variants such as VDSL2 are not always) based you probably want to add the "linklayer adsl" option. This accounts for the overhead which comes from breaking IP packets into a bunch of 53-byte ATM cells.
- If you are using PPPoE then you probably want to account for the PPPoE overhead with the 'overhead' parameter.

TCP Small Queues

The per-socket TCP queue limit can be viewed and controlled with the following /proc file:

```
/proc/sys/net/ipv4/tcp_limit_output_bytes
```

My understanding is that you should not need to modify this value in any normal situation.

Oversized Queues Outside Of Your Control

Unfortunately not all of the oversized queues which will affect your Internet performance are under your control. Most commonly the problem will lie in the device which attaches to your service provider (eg DSL or cable modem) or in the service providers equipment itself. In the later case there isn't much you can do because there is no way to control the traffic which is sent towards you. However in the upstream direction you can shape the traffic to slightly below the link rate. This will stop the queue in the device from ever having more than a couple packets. Many residential home routers have a rate limit setting which can be used to shape below the link rate. If you are using a Linux box as a router, shaping below the link rate also allows the kernel's queueing features to be effective. You can find many example tc

scripts online including the one I use < http://git.coverfire.com/?p=linux-qos-scripts.git;a=summary> with some related performance results < http://www.coverfire.com/archives/2013/01/01/improving-my-home-internet-performance/> .

Summary

Queueing in packet buffers is a necessary component of any packet network both within a device and across network elements. Properly managing the size of these buffers is critical to achieving good network latency especially under load. While static buffer sizing can play a role in decreasing latency the real solution is intelligent management of the amount of queued data. This is best accomplished through dynamic schemes such as BQL and active queue management < http://en.wikipedia.org/wiki/Active_queue_management> (AQM) techniques like Codel. This article outlined where packets are queued in the Linux network stack, how features related to queueing are configured and provided some guidance on how to achieve low latency.

Related Links

Controlling Queue Delay < http://queue.acm.org/detail.cfm?id=2209336> – A fantastic explanation of network queueing and an introduction to the Codel algorithm.

Presentation of Codel at the IETF < https://www.coverfire.com/archives/2012/08/13/codel-at-ietf/> – Basically a video version of the Controlling Queue Delay article.

Bufferbloat: Dark Buffers in the Internet < http://cacm.acm.org/magazines/2012/1/144810-bufferbloat/fulltext> – Early article Bufferbloat article.

Linux Advanced Routing and Traffic Control Howto (LARTC) < http://www.lartc.org/howto/> – Probably still the best documentation of the Linux tc command although it's somewhat out of date with respect to new features such as fq_codel.

TCP Small Queues on LWN < http://lwn.net/Articles/507065/>

Byte Queue Limits on LWN < http://lwn.net/Articles/454390/>

Thanks

Thanks to Kevin Mason, Simon Barber, Lucas Fontes and Rami Rosen for reviewing this article and providing helpful feedback.

 Up ↑