Contents

[1. Packet Queues 1](#_Toc163054283)

[1.1. Driver Queue (Ring Buffer) 1](#_Toc163054284)

[1.1.1 Socket kernel buffer 1](#_Toc163054285)

[The socket buffer, or "SKB", is the most fundamental data structure in the Linux networking code. Every packet sent or received is handled using this data structure. The most fundamental parts of the SKB structure are as follows: 1](#_Toc163054286)

[struct sk\_buff { 1](#_Toc163054287)

[/\* These two members must be first. \*/ 1](#_Toc163054288)

[2. Huge Packets from the Stack 1](#_Toc163054289)

[3. Starvation and Latency 1](#_Toc163054290)

[References: 1](#_Toc163054291)

# Packet Queues

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 effect of impacting latency. This article aims to explain where IP packets are queued on the transmit path of the Linux network stack, how interesting new latency-reducing features, such as BQL, operate and how to control buffering for reduced latency.

Diagram

Description automatically generated

Figure 1: Simplified High-Level Overview of the Queues on the Transmit Path of the Linux Network Stack

## Driver Queue (Ring Buffer)

The driver queue lies between the IP stack and the network interface controller (NIC). This queue typically is 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 the packet data. Instead, it consists of descriptors that point to other data structures called “socket kernel buffers”, which hold the packet data and are used throughout the kernel.

The input source for the driver queue is the IP stack that queues 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 it 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.

Diagram

Description automatically generated

Figure 2: Partially Full Driver Queue with Descriptors Pointing to SKBs

### Socket kernel buffer

### The socket buffer, or "SKB", is the most fundamental data structure in the Linux networking code. Every packet sent or received is handled using this data structure. The most fundamental parts of the SKB structure are as follows:

Defined in file: - [include](https://elixir.bootlin.com/linux/v4.20.17/source/include)/[linux](https://elixir.bootlin.com/linux/v4.20.17/source/include/linux)/[skbuff.h](https://elixir.bootlin.com/linux/v4.20.17/source/include/linux/skbuff.h)

### struct sk\_buff {

### /\* These two members must be first. \*/

struct sk\_buff \*next;

struct sk\_buff \*prev;

struct sk\_buff\_head \*list;

...

}

For more details on SKBs: [http://vger.kernel.org/%7Edavem/skb.html](http://vger.kernel.org/~davem/skb.html))

# Huge Packets from the Stack

Most NICs have a fixed maximum transmission unit (MTU), which is the biggest frame that 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](https://en.wikipedia.org/wiki/Jumbo_frame)) of up to 9,000 bytes. Inside the IP network stack, the MTU can manifest as a limit on the size of the packets that are sent to the device for transmission. For example, if an application writes 2,000 bytes to a TCP socket, 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 that 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 that 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 that can be queued in the driver queue. Figure 3 illustrates this concept in contrast with Figure 2.

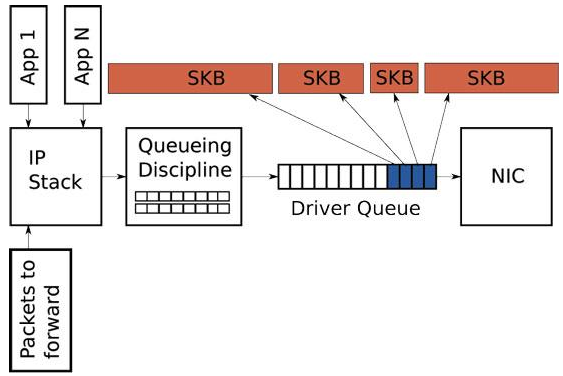


Figure 3: Large packets can be sent to the NIC when TSO, UFO or GSO are enabled. This can greatly increase the number of bytes in the driver queue.

Although the focus of this article is the transmit path, it is worth noting that Linux has receive-side optimizations that operate similarly to TSO, UFO and GSO and share the goal of reducing per-packet overhead. Specifically, generic receive offload (GRO, [http://vger.kernel.org/%7Edavem/cgi-bin/blog.cgi/2010/08/30](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 that is then passed to the IP stack. When the device forwards these large packets, GRO allows the original packets to be reconstructed, which is necessary to maintain the end-to-end nature of the IP packet flow. However, there is one side effect: when the large packet is broken up, it results in several packets for the flow being queued at once. This "micro-burst" of packets can negatively impact inter-flow latency.

# 3. Starvation and Latency

in progress

# DMA descriptors

A MAC module which interacts with Physical layer, and we have DMA which interacts with memory. So, the Ethernet frame flow is like this

**Transmit:**  
Memory-->DMA-->MAC-->Physical Layer  
**Receive:**  
Memory<--DMA<--MAC<--Physical Layer

Transmit and receive descriptors are basically used by DMA. In these descriptors, we give address of memory location from where DMA reads and writes back the Ethernet frame while transmitting and receiving respectively. Apart from memory location, many other important parameters which are required by DMA to process a frame are programmed in descriptors.

Once all the information is programmed, we give the control of descriptors to DMA. We can have one or many descriptors for one frame. When transmit or receive of frame is complete , DMA updates the status in descriptors and assign them back to user. Once we read status and frame data, we again give control of descriptors back to DMA and the process continues.

It's our choice how many descriptors we want to program. The more descriptors we program...the more frames we can process in one go (high performance) but yes at the cost of more kernel resources. There is always a trade off in kernel.

# Miscellaneous notes: -

1. For Gigabit operation, the clocks operate at 125 MHz; for 10/100 operation, the clock rates are 2.5 MHz/25 MHz.
2. TSO: TCP Segmentation Offload

# References:

1. <https://www.linuxjournal.com/content/queueing-linux-network-stack>
2. https://www.coverfire.com/articles/queueing-in-the-linux-network-stack/#:~:text=Driver%20Queue%20(aka%20ring%20buffer,does%20not%20contain%20packet%20data.
3. SKB (Socket Buffer) :

http://vger.kernel.org/~davem/skb.html