Common Functionality in the 2.6 Linux* Network Stack

April, 2010

White Paper

Richard Kelly

Network Software Engineer

Joseph Gasparakis

Network Software Engineer

Intel Corporation

323704

Executive Summary

This paper is an outline of the network stack and the most widely used mechanisms of the 2.6 Linux kernel focusing mainly on the Internet Protocol (IP). Firstly, this work describes the functionality of the network stack itself briefly describing the NAPI, the socket buffer (sk_buff) structure and the Netfilter framework in order to give you some background. Then it focuses on reception of packets, bridging, IP forwarding, transmission of packets from local processes and QoS. It gives a more detailed description on each topic and where appropriate refers to the kernel functions involved and the Netfilter hooks traversed.

This paper is an outline of the network stack and the most widely used mechanisms of the 2.6 Linux kernel, focusing mainly on the Internet Protocol (IP).

Contents

Introduction

The Linux* kernel is a very challenging and fast changing software project. Patches, updates, optimizations and additions occur frequently, and any software engineer who develops software in kernel space needs to be aware of these updates.

Due to the extremely fast and dynamic nature of the development of the Linux kernel, attempts at fully documenting the current kernel cannot easily be kept up to date. In this paper we present some of the main networking functionality of the 2.6 Linux kernel.

We provide a background of some functionality found traditionally in Linux kernels such as the New Application Programming Interface (NAPI), Socket Buffer (sk_buff) structures and Netfilter.

Next, we introduce the network stack with references to packet reception with and without NAPI. Subsequently we provide information about the networking capabilities inside the Linux kernel such as Bridging, IPv4 Forwarding and Quality of Service (QoS).

This paper is based on the 2.6.28 kernel but does not include details on the multi-queue support in the latest kernels and does not compare the 2.6 kernel to previous versions; it merely gives an overview of the network stack.

General Overview

Functionality before NAPI (Older Kernels)

In a non-NAPI enabled kernel, the Network Interface Card (NIC) generates a hardware interrupt to make the device driver aware that a packet has been received. Subsequently, a software interrupt request (softIRQ) is generated to inform the CPU that it has a packet, ready and waiting to be processed. This essentially means that every time a new packet is received, a new interrupt will be generated. The thread for servicing the softIRQs has a higher priority than the thread which actually consumes the packet. In the case of a heavy traffic, all of these context switches can create severe bottlenecks in the system that can escalate to a Denial of Service (DOS) attack [3]. Some improvements in performance have been achieved by introducing a buffer in the NIC where packets can be stored and then sent up to the kernel with just one interrupt, known as interrupt coalescing. However a more efficient mechanism was needed. That mechanism was the NAPI.

Functionality with NAPI (Current Kernels)

NAPI aimed to fix the problem outlined above by using a combination of interrupt and polling rather than just a pure interrupt driven model as used in the non-NAPI case. NAPI is now built into the kernel and enabled by default in a typical Linux setup. It uses a technique called "adaptive interrupt coalescing" [1]. The idea is to use a mixture of interrupts and polling. Device polling does not rely on interrupts to be generated when the device needs attention but rather, it constantly polls it to determine if it has packets that need to be processed.

With NAPI, an interrupt is generated on reception of the first packet received. Interrupts are disabled and the device is put into polling mode where it is checked for packets during every polling cycle. If it has packets, they get processed accordingly. Otherwise interrupts are re-enabled. Research has shown that NAPI is much more efficient, and will never lead to a “livelock” [2]. A livelock is a form of resource starvation and can be defined as the case where many tasks are waiting for the others to finish, so neither of them actually does so.

Socket Buffers

Each packet, both sent and received, has an associated socket buffer as defined in <include/linux/skbuff.h> of the kernel source. The sk_buff is one of the most important data structures for the Linux networking stack. All the relevant data for a packet is kept, and maintained in this structure. All layers of the network stack, including the device driver are aware of it. The device driver has the responsibility to allocate memory and move each received packet’s data to its own associated sk_buff structure. Changing any field or any property in a packet is achieved by updating the appropriate field of that packets sk_buff structure. This structure maintains pointers to both the previous and next sk_buff in the queue. Also it contains information regarding the device on which the packet was received for ingress traffic, or will be transmitted on for egress traffic.

Netfilter

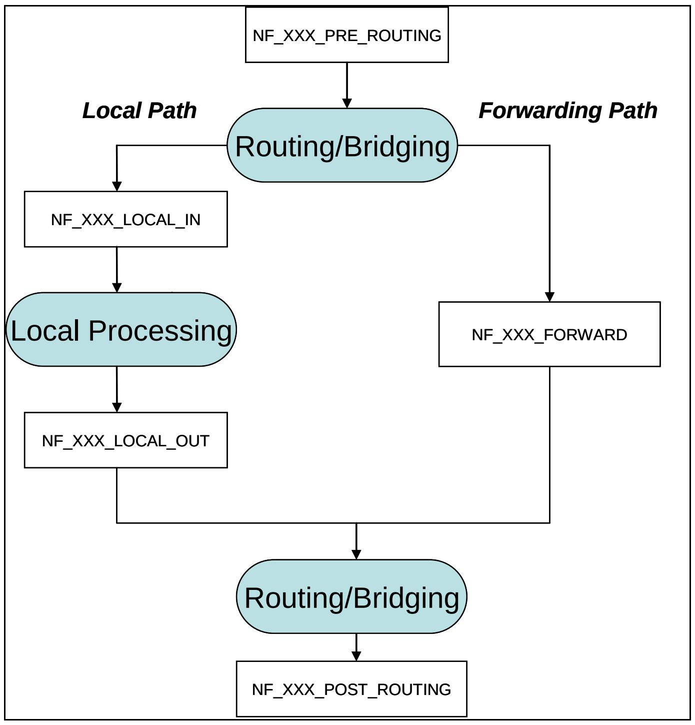
Another important mechanism inside the Linux kernel, introduced during the 2.3.x development was the Netfilter framework [12] [13]. Netfilter comprises of a set of hooks within the Linux network protocol stack. These hooks can be used to register kernel modules that can manipulate network packets at different stages, shown in Figure 1:

- NF_XXX_PRE_ROUTING: Pre-routing. All packets received by the NIC, hit this hook, which is reached before the routing or bridging decision and after all the sanity checks. Usually Network Address Port Translation (NAPT) and Destination Network Address Translation (DNAT) are implemented in this hook with regards to the IP stack.

- NF_XXX_LOCAL_IN: Local Input. All packets going to the local machine reach this hook.
- NF_XXX_FORWARD: Forwarding. Packets which are not destined for a local process hit this hook.
- NF_XXX_LOCAL_OUT: Local Output. This is the first hook in the egress packet path. Packets leaving the local machine always reach this hook.
- NF_XXX_POST_ROUTING: Post-routing. This hook is reached after the routing or bridging decision. Source Network Address Translation (SNAT) is registered to this hook for the IP stack. All packets leaving the local machine reach this hook.

The XXX implies functionality or protocol. For example NF_BR_PREROUTING refers to a hook for a packet (sk_buff) in Bridging at pre-routing, where NF_INET_FORWARD refers to a hook in the IP forwarding mechanism in the stack that is following the forwarding path.

Figure 1. Netfilter Hooks

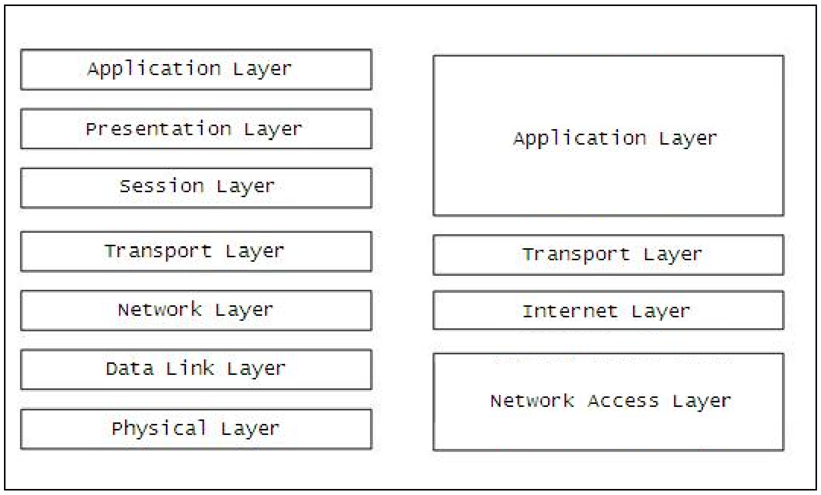


Netfilter can be configured by user applications such as iptables or conntrack.

Linux Network Stack

The basic network stack model which Linux adopted from the TCP/IP stack has four basic layers as shown in Figure 2.

Figure 2. Basic layers of the Network Stack



The application layer runs usually in user space, such as telnet or FTP client, although some exceptions also exist [11]. The transport layer is largely made up of two key protocols: TCP and UDP. The main protocol of the internet layer, also known as network layer, is IP. The network access layer, often called the Link layer, consists of the device driver and deals with the low level details of how a packet gets from the hardware interface and passed up the network stack.

As a packet moves through the network stack, data from each layer is appended or removed from each socket buffer as appropriate. Each layer adds data to the packet as it is passed down, known as encapsulation. For example, the network layer will add the source/destination IP address. When the packet is received at the destination, it will be passed up the stack once more. Each layer will consume the data which has been put on at the corresponding layer at the transmitter's side, known as de-capsulation.

Receiving a Packet

Upon reception of a packet, either in NAPI or Non-NAPI context, the driver allocates a socket buffer and stores the packet in an sk_buff structure. An important function that the device driver has to do, is to fill in the sk_buff–>protocol field. This will be used later to determine which protocol the packet should be sent to. To fill this in, the driver calls eth_type_trans(), which determines which protocol stack the packet should be delivered to.

Receiving a Packet Without NAPI

The driver then calls the netif_rx() function. This function is defined in <include/net/dev.c> in the kernel sources. The netif_rx() function checks to see if the packet has been timestamped, and if not, it does so. Then it checks the queue to see if there is available space for the packet, and if so, it inserts it into the queue and calls netif_rx_schedule(). Netif_rx_schedule then raises a NET_IF_SOFTIRQ so that the kernel will be aware that there is a packet to process.

When the kernel services the NET_IF_SOFTIRQ, it invokes the interrupt handler, which is NET_RX_ACTION. NET_RX_ACTION is the function which is the consumer of the packets. It looks up the poll_list (this is a list of devices which have packets waiting to be processed, which is worth mentioning has nothing to do with polling mechanism) and calls the poll method for each one.

If the poll method returns anything but 0, it means that all the packets on that queue were not processed and there are others still enqueued, so the device goes to the back to the poll_list.

For non-NAPI setups, the poll method for backlog_dev (backlog_dev represents a device that has scheduled a soft irq [4]) is initialized to be the process_backlog() function, e.g.:

```
queue->backlog_dev.poll = process_backlog;
```

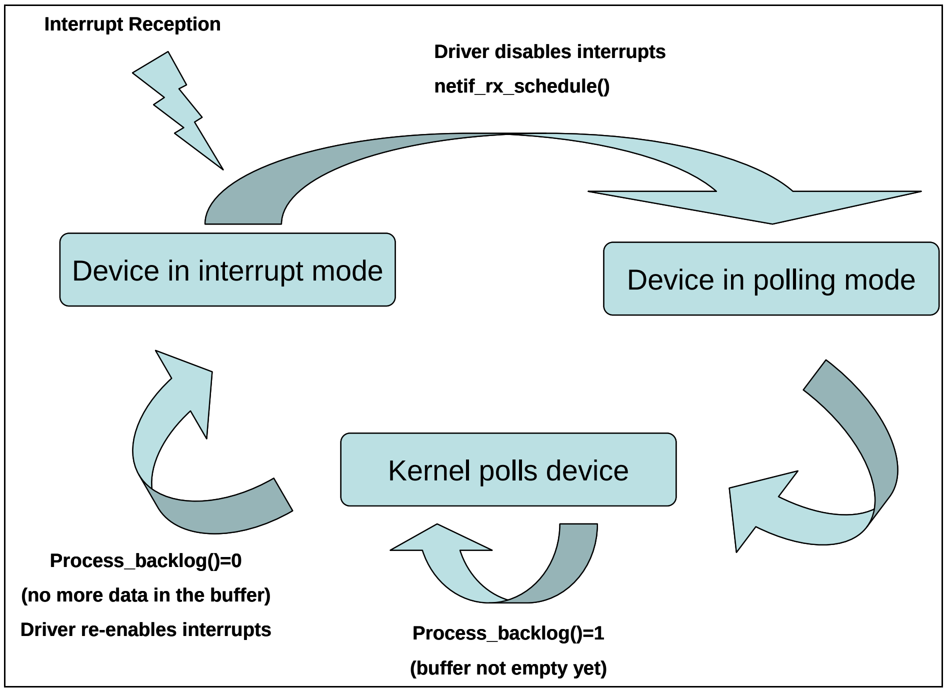
This process_backlog() function takes the packets from the queue and calls netif_receive_skb(). Netif_recieve_skb() looks up the protocol field of the sk_buff for that packet, and passes it to the intended handler of the protocol. As an example, for IP this is ip_rcv(). If all packets have been processed on a device then it is removed from poll_list.

Receiving a Packet with NAPI

The functionality of NAPI is shown in Figure 3. For a NAPI-enabled kernel, there is no netif_rx() function. The device driver disables further interrupts after the reception of the first packet and instead calls netif_rx_schedule() function straight away.

Another difference lies in the poll function. The poll function of a non-NAPI device really is a pointer to the process_backlog() function. The poll method in a NAPI-enabled setup checks if all the packets have been processed, and if so returns 0 to the net_rx_action() function and enables interrupts. This means that the device is taken out of the poll_list, and net_rx_schedule() will be called to raise another interrupt on reception of the next packet.

Figure 3. NAPI Mechanism



Bridging

Bridging in Linux conforms to the IEEE 802.1d standard. A bridge can connect different networks to each other. Packets are moved through it based on their MAC addresses and therefore traffic move through transparently. Bridging and firewalling is often done together and these concepts can be combined using Etables [9].

All the bridging network code resides on the Network Access Layer of the stack. (Layer one of the TCP/IP model). A bridge is implemented as a virtual device which cannot perform any action until such time as at least one real device is bound to it.

The bridge checks the forwarding database to see if it contains information about where the destination MAC address is. If it successfully determines where the MAC address is located, it sends the frame on the corresponding bridge port. If it cannot determine where to send the frame then the bridge “floods” the frame to all ports of the bridging device. This way, the frame will be sent to the correct location.

The spanning tree protocol (STP) eliminates loops in network topologies by disabling some redundant links. This protocol maintains a tree which aims to cope with configuration changes and it also has the ability to update the forwarding database dynamically. A Bridging Protocol Data Unit (BPDU) is essentially a management message which can be passed between bridges to exchange control information. The STP exchanges BPDUs to get information from its neighbors.

The entry point for the bridging code is the br_init() initialization function. This function sets up the forwarding database and sets up br_handle_frame_hook to point to br_handle_frame(). Br_handle_frame() is called each time a frame is received on the bridge.

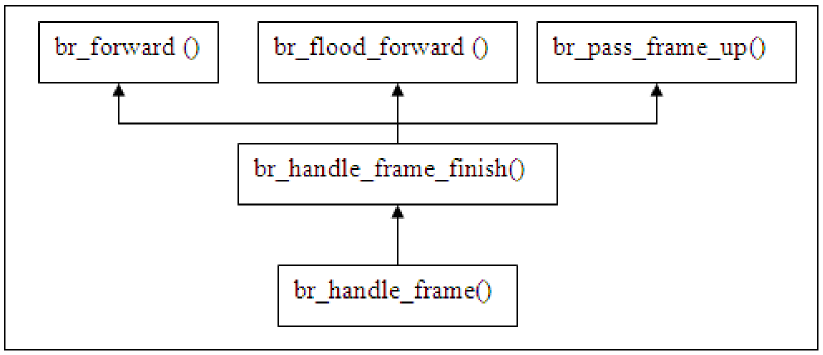
The net_bridge structure is the main data structure of the bridge. A bridge device is added by a call to br_add_bridge() and removed by a call to br_del_bridge().

The function netif_receive_skb(), passes a frame to the bridge_handle() function. This calls the br_handle_frame_hook() which points to br_handle_frame(). Next the br_handle_frame() function calls the br_handle_frame_finish() [10].

Another point of note is the br_handle_frame_finish() function. This adds the MAC address marked as the source, into the forwarding database. This then looks up the forwarding database to see if any information about the destination MAC address exists. If it does exist and the frame is not for local delivery, the br_forward() function is called, else the br_flood_forward() function is called which sends the message out on all bridging ports. This ensures the frame is sent to the right destination as mentioned above. If the frame is for local delivery br_pass_frame_up is called.[10].

An example of the very basic initial bridging steps as described can be seen in Figure 4.

Figure 4. Bridging Inside the Kernel



On the way out to the egress interface, deliver is being called and the NF_BR_LOCAL_OUT and consequently NF_BR_POST_ROUTING Netfilter hooks are traversed.

IP Forwarding for IPv4

IP Forwarding can be defined as the process of forwarding IP packets based on the routing tables. The Linux kernel has the capability to act as a router. For this facility to be turned on, the value in the /proc/sys/net/ipv4/ip_forward can be set to 1. All IP forwarding takes place in the network layer of the network stack [5].

As described before, when the IP network stack has a packet to process, ip_rcv(), the handler function of the IP is called. NF_INET_PRE_ROUTING exists in this function. There are multiple structures contained inside the kernel with regards to forwarding. A very important one is the routing cache. This holds information about recently routed packets. This memory is small in size but less expensive to access than a normal lookup in memory.

At a high level, the Forwarding Information Base (FIB) is the primary routing information storage area which contains details of how the packets will be forwarded. To read entries in the tables, a call is made to fib_lookup(). The key which is used in a lookup of the routing cache is an IP address, whereas in a FIB lookup, the key is actually a subnet.

The job of ip_rcv() function is to perform sanity checks on a packet, such as validating the check sum. After this validation, the check for whether or not the packet is for local consumption can be performed. If it is, it can be passed straight up the network stack as normal. To determine whether it is destined for a local process, or whether it should be forwarded to the corresponding interface for the intended recipient, ip_route_input() is eventually called to lookup the routing cache.

If there is a cache miss, ip_route_input_slow() calls a function to perform a lookup of the routing table. Depending on the value returned by ip_route_input() the kernel can determine if the packet is due for local delivery, or remote delivery. The IP address of the packet is looked up in the kernel routing tables and if it is to be forwarded, ip_forward() is called. If the packet is for local delivery, ip_local_deliver() is called and the NF_INET_LOCAL_IN hook will be fired of. In its way out from a local delivery, ip_local_out() is called and NF_INET_LOCAL_OUT hook is hit. The ip_forward() function checks if the time to live field has expired, and if so then the packet should be dropped. When it is dropped an Internet Control Message Protocol (ICMP) message is sent to advise that the time for the packet to live has been exceeded. This is where the Netfilter NF_INET_FORWARD hook is traversed and the ip_foward_finish() function is called which sends out the packet via an indirect function call to ip_output(). When ip_ finish_output() is called by ip_output(), the final Netfilter NF_INET_POST_ROUTING hook is traversed [6].

Transmission of Packets From a Local Process

When a packet comes from a local process, the ip_route_output_key() function is called. For transmission a routing lookup is also needed so the packet will be directed towards the right interface. If the needed information is not stored in the routing cache, then a similar process to ingress traffic needs to be followed. A call to ip_route_output_slow() is performed, where the FIB table is looked up. Then provided the packet is not destined for a local process (ie: loopback), ip_output() is called. The final Netfilter NF_INET_POST_ROUTING hook is traversed in a similar way to the IP forwarding case [6].

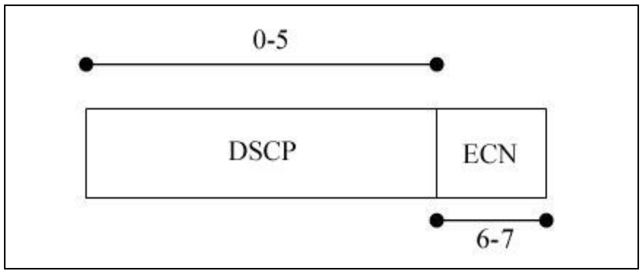
Quality of Service

Quality of Service (QoS) is commonly defined as a method for differentiating between different types of traffic so that they may be given a different priority. Naturally, a kernel may have multiple flows of packets to process. Some packets may have real time constraints which have a higher criticality than others. DiffServ (Differentiated Services) is a method of managing network traffic services by providing QoS guarantees to critical services like VoIP whilst also providing best effort guarantees to non-critical services such as web traffic, in a scalable manor. DiffServ does not make any assumptions about which traffic should be given a higher priority but rather it merely provides a mechanism for doing so. This mechanism was originally outlined by the IETF (Internet Engineering Task Force).

Diffserv aware routers offer Per-Hop Behaviors (PHBs) which define the packet forwarding properties associated with a class of traffic. All traffic of the same class, flowing through a DiffServ enabled router (in this case the Linux kernel) is called a "Behavior Aggregate" (BA).

DiffServ uses the ToS (Type of Service) field in the IP header (which is often called the DS field – or Differentiated Service field). ToS is an 8-bit segment that is broken into two main parts as shown in Figure 5.

Figure 5. ToS Breakdown



PHB's filters determine which class a packet gets classified into. This classification is done by a mechanism called the Linux traffic control. The four most common PHB filters are as follows:

- Default PHB: best effort.
- Assured Forwarding (AF) PHB: assurance of delivery as long as traffic does not exceed the subscription rate.
- Expedited Forwarding (EP) PHB: minimizes latency, jitter and packet loss.
- Class Selector (CS) PHB: for backward compatibility with Precedence field in ToS.

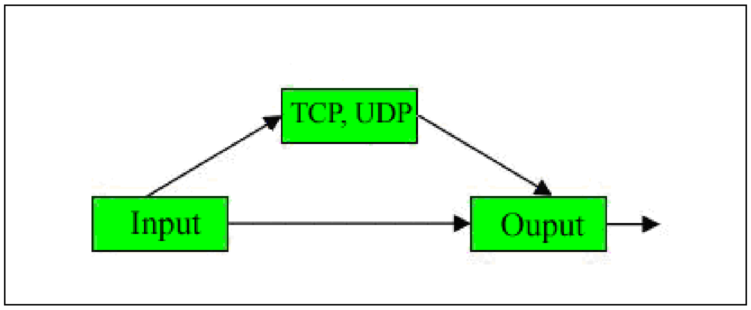
Before DiffServ existed, a Precedence field existed in the ToS byte to mark the priority of packets. However, this field was not taken advantage of, so the IETF decided to reuse the ToS field as its current DS incarnation. In order to maintain a degree of backwards compatibility with any device that still uses the precedence field, DiffServ supports the Class Selector PHB.

The first three bits of the Class Selector PHB are the precedence field. The precedence field can be mapped into a different DiffServ class. When a packet is received by a DiffServ enabled router (such as the Linux kernel) from a non-DiffServ enabled device, the DiffServ enabled one has the ability to classify the packet into different priority queue depending on the packet's class selector code point. Therefore, some degree of backwards compatibility is supported, and DiffServ is aware of packets which use the precedence field.

The ECN (the final two bits of the ToS field) provides notification of network congestion without the loss of packets. This feature is optional and only availed of when both end-points decide to use it.

Figure 6 shows the very basic path a packet can take through the kernel. In this diagram, the TCP/UDP block includes both ingress traffic that is not intended for the local machine (traffic to be forwarded) and packets sent from local host which need to be sent out. It is at this output queuing stage where Linux Traffic control resides.

Figure 6. High-level Path Overview



Traffic control can determine in which order packets are queued and sent, which packets are dropped and essentially, give preference to higher priority packets. As soon as traffic control has released a packet to be transmitted, the device driver can send it onto the wire. The main traffic control components can be categorized as below:

- Queuing disciplines (qdiscs)
- Classes (within a qdisc)
- Filters

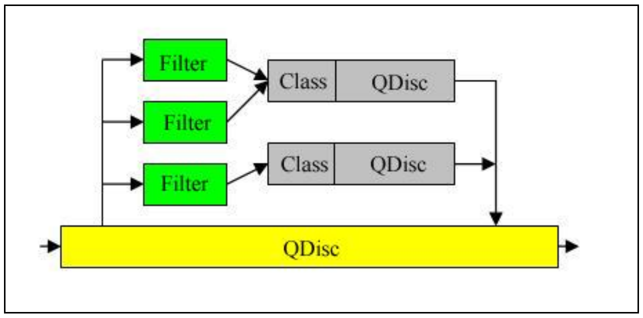
The qdiscs control how packets are queued for transmission on a device. In the simplest case, this can be a FIFO.

Expanding on this, filters can be used to classify a packet into a particular class. Each class may be processed in a different fashion and a class may be given priorities over a different one. This is done by giving each class its own sub qdisc as distinct from the overall qdisc.

dev_queue_xmit() is the function called when a buffer is ready to be transmitted. This then calls the enqueue method for the qdisc [7].

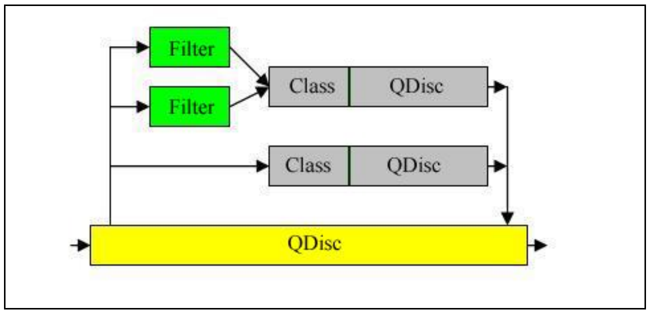
For example as shown in Figure 7, the overall qdisc may contain two classes, with two different priorities. Filters will decide within which qdisc of a class a particular packet should be placed. Then there may be a rule to always transmit the packets from the qdisc of the class with the highest priority, while the qdisc of that class is not empty. This essentially allows one class of traffic to take priority over another [8].

Figure 7. Qdisc Example



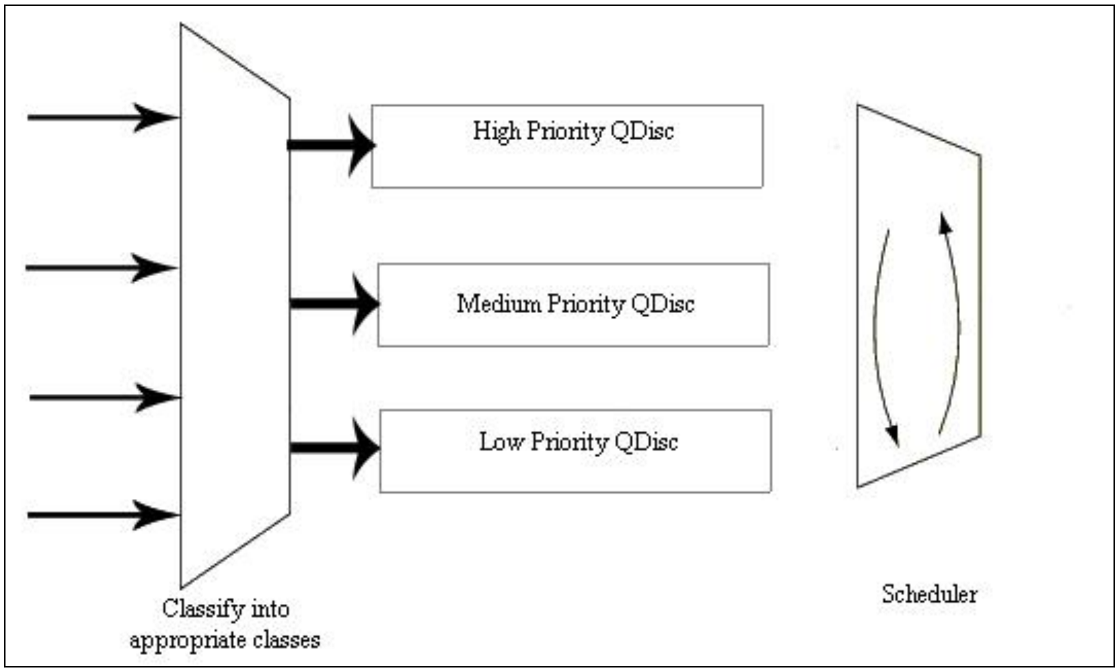
The filters may be the 4 PHB as described previously, or they could be based on a Multi-field classifier (MF-classifier), which takes a source/destination IP address and other fields into account. For example, if one packet does not match any filter, its default behavior will be Default PHB, which as shown earlier, is generally best effort delivery. An example of this can be seen in Figure 8.

Figure 8. Qdisc With Default Classifier



The full picture of the Linux traffic control mechanism is demonstrated in Figure 9. This shows the same set up which has been described previously, in which filters are used to classify flows into classes. These classes are assigned different priority. Finally, the scheduler takes control of what order these qdiscs of packets are sent onto the wire by the device driver based to the qdiscs relative priority.

Figure 9. Linux Traffic Control



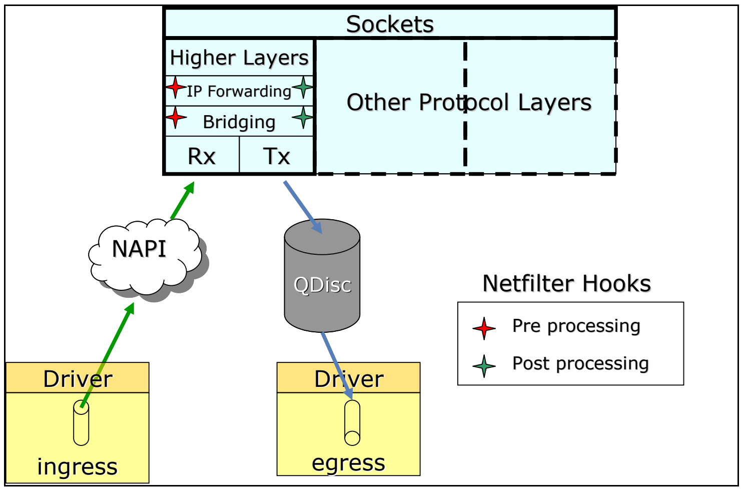
After a packet has been enqueued, _qdisc_run() is eventually called. Its purpose is to call qdisc_restart() as long as the device is able to transmit something.

qdisc_restart() dequeues a packet from the qdisc. It does so by eventually calling hard_start_xmit() function which is implemented in the driver.

Summary

In the beginning of this paper the outline of the Linux network stack of the 2.6 kernel was presented along with its basic structure, the sk_buff and the netfilter mechanism. The non-NAPI and NAPI mechanisms for packet reception were introduced, following a more detailed description with references to the kernel functions. Following this, bridging was discussed, IP Forwarding and packet transmission from a local process. Finally in this paper QoS and how it is implemented in the Linux Traffic Control was presented. The overview of the whole networking mechanism in the Linux kernel that was presented in this work can be summarized in Figure 10.

Figure 10. Overview of the Linux Networking Mechanism



The authors are hoping that the reader will find this information useful during the development of networking drivers, applications in kernel space or any form of analysis of the Linux network stack. The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. http://intel.com/embedded/edc.

References

[1] http://www.geocities.com/asimshankar/notes/linux-networking-napi.html

[2] Jamal Hadi Salim, Robert Olsson and Alexey Kuznetsov

- Beyond Softnet

[3] Luca Deri - Improving Passive Packet Capture: Beyond Device Polling

[4] Ivan Pronchev - Packet Capturing Using the Linux Netfilter Framework, July 2006

[5] Linux Network Software, UCL CS

[6] Rami Rosen - Linux Kernel Networking, http://www.haifux.org/lectures/172/netLec.pdf

[7] http://www.opalsoft.net/qos/DS-21.htm

[8] Wener Almesberger, Jamal Hadi Salim, Alexey Kuznetsov

- Differentiated Services on Linux

[9] http://ebtables.sourceforge.net/

[10] Christian Benvenuti - Understanding Linux Network Internals, O'Reilly Publications, Chapter 16: Bridging: Linux Implementation

[11] Philippe Joubert, Robert B. Kingy, Rich Neves, Mark Russinovichz, John M. Tracey - High-Performance Memory-Based Web Servers: Kernel and User-Space Performance

[12] Rusty Russell, Harald Welte - "Linux netfilter Hacking HOWTO": http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.txt

[13] http://www.netfilter.org/

Authors

Joseph Gasparakis is a network software engineer with the Embedded Computing Group.

Richard Kelly is a network software engineer with the Embedded Computing Group.

Acronyms

AF	Assured forwarding
BA	Behavior aggregate
BPDU	Bridging protocol data unit
CS	Class selector
DNAT	Destination network address translation
DOS	Denial of service
EP	Expedited forwarding
FIB	Forwarding information base
FIFO	First in first out
ICMP	Internet control message protocol
IETF	Internet engineering task force
IP	Internet protocol
NAPI	New application programming interface
NAPT	Network address port translation
NIC	Network interface card
PHB	Per hop behavior
QoS	Quality of Service
SNAT	Source network address translation
STP	Spanning tree protocol
TCP	Transmission control protocol
ToS	Type of Service
UDP	User datagram protocol