The journey of a packet through the linux 2.4 network stack

Harald Welte laforge@gnumonks.org
1.4, 2000/10/14 20:27:43

This document describes the journey of a network packet inside the linux kernel 2.4.x. This has changed drastically since 2.2 because the globally serialized bottom half was abandoned in favor of the new softirq system.

Contents

1 Preface 1

2 Receiving the packet 1

2.1 The receive interrupt 1

2.2 The network RX softirq 2

2.3 The IPv4 packet handler 2

3 Packet forwarding to another device 3

1 Preface

I have to excuse for my ignorance, but this document has a strong focus on the "default case": x86 architecture and ip packets which get forwarded.

I am definitely no kernel guru and the information provided by this document may be wrong. So don't expect too much, I'll always appreciate Your comments and bugfixes.

2 Receiving the packet

2.1 The receive interrupt

If the network card receives an ethernet frame which matches the local MAC address or is a linklayer broadcast, it issues an interrupt. The network driver for this particular card handles the interrupt, fetches the packet data via DMA / PIO / whatever into RAM. It then allocates a skb and calls a function of the protocol independent device support routines: `net/core/dev.c:netif_rx(skb)`.

If the driver didn't already timestamp the skb, it is timestamped now. Afterwards the skb gets enqueued in the apropriate queue for the processor handling this packet. If the queue backlog is full the packet is dropped at this place. After enqueuing the skb the receive softinterrupt is marked for execution via `include/linux/interrupt.h:__cpu_raise_softirq()`.

The interrupt handler exits and all interrupts are reenabled.

2.2 The network RX softirq

Now we encounter one of the big changes between 2.2 and 2.4: The whole network stack is no longer a bottom half, but a softirq. Softirqs have the major advantage, that they may run on more than one CPU simultaneously. bh's were guaranteed to run only on one CPU at a time.

Our network receive softirq is registered in `net/core/dev.c:net_init()` using the function `kernel/softirq.c:open_softirq()` provided by the softirq subsystem.

Further handling of our packet is done in the network receive softirq (NET_RX_SOFTIRQ) which is called from `kernel/softirq.c:do_softirq()`. do_softirq() itself is called from three places within the kernel:

1. from `arch/i386/kernel/irq.c:do_IRQ()`, which is the generic IRQ handler
2. from `arch/i386/kernel/entry.S` in case the kernel just returned from a syscall
3. inside the main process scheduler in `kernel/sched.c:schedule()`

So if execution passes one of these points, do_softirq() is called, it detects the NET_RX_SOFTIRQ marked an calls `net/core/dev.c:net_rx_action()`. Here the sbk is dequeued from this cpu's receive queue and afterwards handled to the apropriate packet handler. In case of IPv4 this is the IPv4 packet handler.

2.3 The IPv4 packet handler

The IP packet handler is registered via `net/core/dev.c:dev_add_pack()` called from `net/ipv4/ip_output.c:ip_init()`.

The IPv4 packet handling function is `net/ipv4/ip_input.c:ip_rcv()`. After some initial checks (if the packet is for this host, ...) the ip checksum is calculated. Additional checks are done on the length and IP protocol version 4.

Every packet failing one of the sanity checks is dropped at this point.

If the packet passes the tests, we determine the size of the ip packet and trim the skb in case the transport medium has appended some padding.

Now it is the first time one of the netfilter hooks is called.

Netfilter provides an generict and abstract interface to the standard routing code. This is currently used for packet filtering, mangling, NAT and queuing packets to userspace. For further reference see my conference paper 'The netfilter subsystem in Linux 2.4' or one of Rustys unreliable guides, i.e the netfilter-hacking-guide.

After successful traversal the netfilter hook, `net/ipv4/ipv_input.c:ip_rcv_finish()` is called.

Inside ip_rcv_finish(), the packet's destination is determined by calling the routing function `net/ipv4/route.c:ip_route_input()`. Furthermore, if our IP packet has IP options, they are processed now. Depending on the routing decision made by `net/ipv4/route.c:ip_route_input_slow()`, the journey of our packet continues in one of the following functions:

net/ipv4/ip_input.c:ip_local_deliver()

The packet's destination is local, we have to process the layer 4 protocol and pass it to an userspace process.

net/ipv4/ip_forward.c:ip_forward()

The packet's destination is not local, we have to forward it to another network

net/ipv4/route.c:ip_error()

An error occurred, we are unable to find an apropriate routing table entry for this packet.

net/ipv4/ipmr.c:ip_mr_input()

It is a Multicast packet and we have to do some multicast routing.

3 Packet forwarding to another device

If the routing decided that this packet has to be forwarded to another device, the function `net/ipv4/ip_forward.c:ip_forward()` is called.

The first task of this function is to check the ip header's TTL. If it is <= 1 we drop the packet and return an ICMP time exceeded message to the sender.

We check the header's tailroom if we have enough tailroom for the destination device's link layer header and expand the skb if neccessary.

Next the TTL is decremented by one.

If our new packet is bigger than the MTU of the destination device and the don't fragment bit in the IP header is set, we drop the packet and send a ICMP frag needed message to the sender.

Finally it is time to call another one of the netfilter hooks - this time it is the NF_IP_FORWARD hook.

Assuming that the netfilter hooks is returning a NF_ACCEPT verdict, the function `net/ipv4/ip_forward.c:ip_forward_finish()` is the next step in our packet's journey.

ip_forward_finish() itself checks if we need to set any additional options in the IP header, and has ip_optFIXME doing this. Afterwards it calls `include/net/ip.h:ip_send()`.

If we need some fragmentation, FIXME:ip_fragment gets called, otherwise we continue in `net/ipv4/ip_forward:ip_finish_output()`.

ip_finish_output() again does nothing else than calling the netfilter postrouting hook NF_IP_POST_ROUTING and calling ip_finish_output2() on successful traversal of this hook.

ip_finish_output2() calls prepends the hardware (link layer) header to our skb and calls `net/ipv4/ip_output.c:ip_output()`.