Linux Network Stack Internals

Luca Abeni
luca.abeni@santannapisa.it

June 2, 2020

The Networking Stack

- Networking stack: network driver(s) + protocols
- A simple functioning implementation is not complex
  - Receiving and sending network packets is not difficult
  - The TCP/IP stack is fairly well understood
- However, the "`linux/net`" directory is quite complex
  - Lots of different protocols
  - This is all performance critical code!
- So, the modern Linux networking code is fine-tuned for performance in many different situations

Linux and Networking

- The Linux networking stack is used on many different devices
  - Ranging from Android phones / small embedded devices...
  - ...To big servers...
  - ...Passing through high-performance PCs and similar stuff!
- The code must be designed to perform well in all these situations
  - Low memory footprint / low CPU usage
  - High throughput, resilent to various DoS attacks
  - Low latency; performant for both TCP and UDP
  - ...

Evolution of the Linux Stack — 1

- The original netorking stack did “just work”
  - But was slow, and UP only
- Then, it was modified to run on multiple processors
  - But it was not able to take advantage of the hardware parallelism
  - The throughput did not scale with the number of CPUs
  - Issue: bottom half processing (only one bottom half can execute simultaneously, regardless of the number of CPU cores)
- Solution: use SoftIRQs
  - No per-core concurency, but multiple SoftIRQs can execute simultaneously on different cores

Evolution of the Linux Stack — 2

- Next issue: receive livelock
  - When packets arrive too fast, most of the time is lost in raising/serving interrupts
  - High userspace/kernelspace switch overhead, no time left for using the received packets!
- Solution: some form of interrupt mitigation / polling
  - NAPI: adaptive polling (in SoftIRQ context!), activated only when interrupts fire too often
  - Some kind of heuristic is used to activate the NAPI polling mode
- This solves some possible DoS attacks

Evolution of the Linux Stack — 3

- With the advent of Gb and 10Gb ethernet, new performance issues
  - Things work well for large packets (jumbo frames, etc...)
  - A lot of overhead for smaller packets
- Solution: Generic Receive Offload (GRO)
  - Try to merge multiple small packets in large buffers when possible
  - Process these small packets in batches (instead of processing them one at time)
  - Improves the receiving throughput a lot
- Of course, this makes the code much more complex!

The sk_buff Structure

- As the name suggests, `struct sk_buff` represents a packet that can be sent/received through a socket
  - More generically, through a network interface
- Easy in theory... But it is a quite complex structure!
- Passed through the various layers of the network stack, that can add/remove headers/trailers...
  - Must allow to efficiently add/remove them without copy
- Contains various kinds of fields
  - Related to lists
  - Data
  - ...

sk_buff Lists

- sk_buff structures are stored in lists
  - But they are not the “standard” Linux lists
  - Why? For efficiency reasons
  - Standard linux list: generic; sk_buff list: efficient
- Doubly linked lists: prev and next fields (pointers to struct sk_buff)
  - Must be the first fields of the structure
  - To match struct sk_buff_head
- struct sk_buff_head: head of a sk_buff list
  - The first 2 fields are the same contained in struct sk_buff
  - Also contains a spinlock and a len

Manipulating the Lists

- `sk_buff` lists are not regular Linux lists → need spacial functions to handle them
  - Defined in `net/core/skbuff.c` and `include/linux/skbuff.h`
  - In general every function has an unlocked "`__`" equivalent (often an inline function in `skbuff.h`)
- `skb_queue_head_init()`: initializes an `sk_buff` list head
- `skb_queue_head()`: insert an `sk_buff` at the head of a list
- `skb_queue_tail()`: insert an `sk_buff` at the tail
- `skb_dequeue()`: removes the first `sk_buff` from a list
- ...

sk_buff Data

- The structure contains different “data related” fields
- First, there are some lenghts, for example:
  - `len`: current size of the data
  - `data_len`: size of data contained in additional fragments
  - `truesize`: size of this buffer + `sk_buff` structure
- Then, there are various pointers to the buffer:
  - `head`: beginning of the buffer in memory
  - `data`: beginning of the data (= `head` + headroom)
  - `tail`: end of the data (= end of buffer - tailroom)
  - `end`: end of the buffer in memory

Adding/Removing Headers/Tailers

- When a `sk_buff` is allocated, `head` = `data` = `tail`; `end = head + size`
  - No headroom, everything is tailroom
- `len = 0`
  - No data in the buffer
- Then, the size of the buffer can be increased with `skb_put()` and `skb_push()`
  - Grow the buffer using tailroom and headroom
  - Need enough space in *room... But the headroom is initially empty! How can `skb_push()` work?

Making Space for Headers

- When a `sk_buff` is allocated, `head = data` ⇒ no headroom
  - But `skb_push()` works by decreasing `head`...
  - Before using `skb_push()` some space has to be created in the headroom!!!
- Space can be added to headroom with `skb_reserve()`
  - Does not actually copy data: just moves `head` (and `tail`)
  - Must be called before putting data in the buffer

Summing Up

- `alloc_skb()`: allocate empty (`len = 0`) buffer
- `skb_reserve()`: grow the headroom of a buffer (decreasing the tailroom)
- `skb_put()`: grow the buffer size (data len) at the end (getting memory from tailroom)
- `skb_push()`: grow the buffer size (data len) at the beginning (getting memory from headroom)
  - This makes space for a new protocol header
- `skb_pull()`: decrease the buffer size (data len) at the beginning (this removes a protocol header)

Fragmented sk_buffs

- Network packets can be split in various memory fragments
- The first fragment is described by the `sk_buff` structure
- What about the other ones?
  - At the end of the data buffer (`end` field), there is a `skb_shared_info` structure
  - A pointer to it can be obtained through the `end` field
- This structure contains information about the number of fragments, and a list to them

Cloning sk_buffs...

- Cloning a `sk_buff` is an unexpensive operation
  - Only the `sk_buff` structure is duplicated; the data buffer is shared
  - Specialized copy operation, to be more efficient!
- `cloned` flag set to 1
- There also is a usage counter (`dataref`)
  - Obviously, it cannot be in the `sk_buff` structure...
  - It is in the shared `skb_shared_info` structure!!!
- When a `sk_buff` is freed, the data buffer is released only if `dataref` is 0

...And Copying Them!

- The content of the data buffer of cloned `sk_buff`s is shared between all che clones
  - Hence, it cannot be modified!
  - Only (atomic) changes to some fields of `skb_shared_info` are allowed
- What to do if a real copy of a packet is needed?
- There is a function (`skb_copy()`) to duplicate both `sk_buff` and data buffer
  - `pskb_copy()` also duplicates fragments

Network Devices Structures

- A network device is handled by using a set of kernel structures
  - Traditionally, a `struct net_device` contained all the information
  - Even a pointer to the `poll()` method used by NAPI!
- Today, information are spread over multiple data structures
  - `net_device` is still the central one
  - But for receiving packets a `napi_struct` is used
  - Interrupts are associated to a NAPI structure, and the `net_device` structure is linked from it

The net_device Structure

- “Traditional” descriptor for a physical or virtual network device
  - Structure containing all the information needed to operate the device
- Various kinds of information
  - Related to hardware (or virtual description) of the device
  - General information about the device (name, state, list-related fields, ...)
  - Information about the interface (MTU, header size, queue len, ...)
  - Some kinds of device methods (function pointers, grouped in structures)
  - Some statistics

Hardware-Related Information

- Memory ranges for memory-mapped devices
- I/O base
- Used interrupt number
- Everything else that can be useful...
- Also, there is some “private state” for the driver
  - No pointer in the structure, but appended at the end
- Today, most of the important hardware-relate information are stored in the private structure, not in `struct net_device`
  - Example: `struct net_device` has only one `irq` field, but many modern NICs can raise multiple interrupts...

Device Information

- Device name
- Numeric identifier for the device (interface index `ifindex`)
- Information about the interface address
  - For example, permanent MAC address of the board, list of assigned MAC addresses, ...
- Some lists the network device can be into
  - Global list of network devices
  - Some additional lists for specific things (NAPI, devices being closed/unregistered, ...)

Device Methods

- The methods are grouped in various structures (eth methods, device methods, header-related methods, ...)
- Struct `net_device_ops` (`ndo_` methods)
  - `ndo_init()/ndo_uninit()`
  - `ndo_open()/ndo_stop()`
  - `ndo_start_xmit()`
  - ...
- Struct `header_ops`
  - `create()`
  - `parse()`
  - ...

Sending/Receiving Packets through Devices

- A packet is sent by invoking the `ndo_start_xmit()` method of `net_device`
  - Generally not invoked directly, but through netdev_start_xmit()
  - `dev_queue_xmit()` also passes through the network scheduling framework
- How is a packet received?
  - The device driver installs an interrupt handler that somehow manages to push the packet up to the network device structure...

Interrupt Handlers and NAPI

- The device driver installs ISRs with `request_irq()`
  - `request_irq()` allows to specify a data structure that will be passed to the ISR
  - Can be a device-private structure (see `igbx_main.c`), a per-irq structure (see `ixgbe_main.c`) or the `net_device` structure (see `e1000e/netdev.c`)
  - This structure contains a pointer to a `napi_struct`
- The ISR invokes `napi_schedule_prep()` to check if NAPI is already polling or is disabled
  - If `napi_schedule_prep()` returns true, `__napi_schedule()` is invoked

NAPI Processing

- `__napi_schedule()` disables interrupts, gets the per-cpu softirq context, and triggers the softirq (`____napi_schedule()`)
  - Notice: interrupt (and migration!) disabling is needed to use per-cpu data
- `____napi_schedule()` adds the NAPI structure to the per-cpu softnet data structure (it has a poll list)
- Then, it raises the `NET_RX_SOFTIRQ`
  - `net_rx_action()` is the handler for `NET_RX_SOFTIRQ`
  - It gets the per-cpu `softnet_data` and iterates on its `poll_list`, invoking `napi_poll()` on the enqueued napi structures

The Polling Method

- `napi_poll()` invokes the `poll()` method of the `napi_struct`
  - Function pointer named “poll”, member of `napi_struct`
- Then, it calls `napi_complete()`, `napi_gro_flush()` and finally `gro_normal_list()`
  - `napi_complete()` invokes `napi_complete_done()` → disable NAPI polling (can re-enable it if needed!)
- The driver’s `poll()` function (poll method in `napi_struct`) ends up calling `napi_gro_receive()`

GRO: Theory of Operation

- When a packet is received, the NIC computes a hash on it
  - The driver stores this “RSS hash” in the skbuff
- A NAPI structure has `GRO_HASH_BUCKET` (equal to 2^i) GRO lists (`gro_hash[]`)
  - A packet can go in the GRO list indicated by the i rightmost bits of its hash
  - If it is in the same flow of the other packets in the list, the it is inserted there
- If a packet is not inserted in any GRO list (GRO normal packet), it is inserted in `rx_list`
  - This allows to process packets in batches

GRO and Packet Queuing

- When the driver passes a packet to the network stack (`napi_gro_receive()`, it is inserted in `grow_hash[j]` or in `rx_list`
- `napi_gro_flush()` sends up the packets merged by GRO and pending on this `napi_struct` (stored in `grow_hash[]`)
  - Done by invoking `napi_gro_complete()` → invoke `gro_complete()` callbacks for higher level protocols
- `gro_normal_list()` invokes `netif_receive_skb_list_internal()` on the packets that have been received and enqueued on the `napi_struct` `rx_list` (sends them up)

Receiving Packets (with GRO Complications)

- In theory, `napi_gro_receive()` should just pass the packet up to higher-level protocols...
- ...But GRO complicates things a little bit!
  - `dev_gro_receive()` checks if the packet can be “merged” with other packets...
  - ...To do this, it needs to invoke higher-level callbacks (to check TCP/UDP flows, etc...)
  - Then, `napi_skb_finish()` passes up the packet (only if it has not been GROed!)
    - Invokes `gro_normal_one()`, that enqueues a packet to `rx_list` of the NAPI structure
- When enough packets have been enqueued, `gro_normal_list()` to send them!

Network Interface Receive

- `netif_receive_skb_list_internal()` processes lists of packets
- Another complication: RPS!
  - Up to now, processing happened on the core that received the interrupt
  - Can “migrate” the processing to another (less busy) core
  - This allows to automatically spread packet processin on all the cores!
- Finally, `__netif_receive_skb_list()` is invoked
- At the end of the story, `__netif_receive_skb_core()` will deliver the packet to the handlers of higher-level protocols (`deliver_skb()`)

Using Network Devices

- `struct net_device` and friends are used to manage hardware (or virtual devices)...
- ...Kernel code can use them directly, but user-space does not see these structures
- User-space code generally uses a higher-level programming interface exposing the whole networking stack through sockets
  - This includes higher-level (network and transport) protocols
- The networking stack transforms user buffers in `sk_buff`s

The Network Stack: Programmaer API

- Networking is accessed from user-space through *sockets*
  - Remember? Each socket has a “type”, a “domain”, and a “protocol”
  - The domain identifies a family of protocols
  - Example: `AF_INET`: internet protocols (IPv4)
- The domain (or protocol family) is mainly used when creating a socket, to select the appropriate protocol
- The kernel uses different data structures to represent the user-space interface of a socket and its internal representation

Socket Data Structures

- Data structure describing the “user-space vision” of a socket: `struct socket` (see `include/linux/net.h`)
  - Contains a (type and protocol dependent) set of operations, the type (stream, datagram, ...) and a link to an internal representation
- Data structure describing the socket’s internal representation: `struct sock` (see `include/net/sock.h`)

Higher Level Protocols

- Higher level protocols (for example IP, UDP, TCP, etc...) are registered at boot time
  - Example: `net/ipv4/af_inet.c::inet_init()`
  - Registers to socket the UDP and TCP protocols, plus some other protocols
  - Registers `AF_INET` sockets (INET family of protocols)
  - Registers TCP, UDP, ICMP and maybe IGMP to the IP network protocol ← mainly used for receiving packets
- The INET family provides a `create()` method (`inet_create()`), while the protocols provide the other methods to send packets, etc...

Creating a Socket and Sending a Packet

- When an INET socket is created, `inet_create()` ends up being called
  - `sys_socket()` searches for the protocol family registered as `AF_INET`
- It looks at type and protocol, searches for the appropriate inet protocol, and sets its operations in the socket structure
  - Example: for a datagram protocol (such as UDP), `inet_dgram_ops` is used
    - It also points to the UDP protocol operations: `udp_prot` (see `net/ipv4/udp.c`)

Sending a Packet

- The "operations structure" `ops` of a `struct socket` contains pointers to the user-invocable operations
  - Methods for operating on the socket (example: sending or receiving packets)
  - These methods are used by the syscalls
- Packets are sent with `sock_sendmsg()` (invoked, for example, by `sendto()`)
- `sock_sendmsg()` invokes `sock_sendmsg_nosec()`, which invokes `sock->ops->sendmsg()`
  - This points to `inet_msg()`, which invokes `sk->sk_prot->sendmsg()` (notice: this are protocol-dependent operations)

Sending a Packet — Down the Protocol Stack

- The protocol-specific `send()` function is invoked (example: `udp_sendmsg()` in `net/ipv4/udp.c`
- First of all, cope with “corked sockets” or similar things
- Then, get the destination address (from the message, or from the socket)
- Handle timestamps and “control messages” that do not need to be sent, IP options, and multicast
- Finally, route the packet!
  - Should be an IP protocol thing, but there is a fastpath in UDP as an optimization...
  - Call `ip_route_output_flow()` and buffer the result in `struct sock`

Sending a Packet — Identify the Destination

- `ip_route_output_flow()` returns a structure indicating how to send the data
  - Technically, it is a routing table entry!
  - First part: `dst_entry` structure
- It indicated the device to be used for sending the data
- It also indicates the next hop to which data has to be sent
  - Parts of it are filled using the ARP protocol
- It also contains function pointers for sending and receiving data!
  - For IPv4, they are set to `ip_output()` and `ip_local_deliver()`

Sending a Packet — Down the Protocol Stack

- After having a routing table entry and handling some other special situation (multicast, broadcast, ARP confirm, ...), the packet is passed down to the IP layer
  - `ip_make_skb()`, then `udp_send_skb()`
- `ip_make_skb()` (see `net/ipv4/ip_output.c`) generates an `sk_buff()` for the message
  - Complex code, because generic (supports corked sockets); for the non-corked case, creates some “fake” corking structures

Sending a Packet — Allocating and Initializing the skbuff

- __ip_append_data() allocates the `sk_buff`
  - Then reserves space for the headers and allocates the network header
  - Also notice "`skb->transport_header = ...`"
  - Finally, it copies the data...
- `__ip_make_skb()` fills the IP header and finishes the `sk_buff` initialization
  - Notice "`skb_dst_set(skb, &rt->dst)`" (and remember that `dst.output = ip_output()`!)

Sending a Packet — Down to Network Protocol

- `udp_send_skb()` fills the UDP header and finally passes the packet down: `ip_send_skb()`
- `ip_send_skb()` invokes `ip_local_out()`, that calls `__ip_local_out()` to set packet len and checksum, and then passes the packet to netfilter
- If netfilter agrees, then `ip_local_out()` calls `dst_output()` to send the packet
  - `dst_output()` does something like `skb_dst(skb)->output(skb)`
  - Looks at the `_skb_refdst` field of `sk_buff`... Set by `__ip_make_skb()` using info coming from the routing table entry

Sending a Packet — From Network to MAC Layer

- `dst_output()` ends up calling `ip_output()`
- `ip_output()` sets `skb->dev`, then calls `ip_finish_output()` → `ip_finish_output2()`
- `ip_output2()` searches for a "neighbour" to send the data, and invokes `neigh_output()` to it
  - We are finally out of the IP stack!!!
  - `neigh_output` checks if we know the MAC address of the neighbour, and if yes it invokes `neigh_hh_output()`
  - If not, some ARP stuff is needed!
- `neigh_hh_output()` fills some headers and finally calls `dev_queue_xmit()`
  - It will call the `ndo_start_xmit()` method of the device, when needed

Receiving a Packet

- How are packets received?
  - There is a `recvmsg` method in the socket operations...
  - ...But where does it get messages from?
- Remember `deliver_skb()`?
  - It searches for a network protocol handler
  - See for example `net/ipv4/af_inet.c::ip_packet_type`
- For IP, `ip_rcv()` ends up being called!
  - It searches for a dst (using early demultiplexing if needed)
  - This sets the `dst.input` pointer to `ip_local_deliver()`

Receiving a Packet — 2

- After checking some headers, `ip_local_deliver()` invokes `ip_local_deliver_finish()`
- The skbuff is then delivered to the appropriate transport protocol
  - Notice `skb_pull()` to remove the network header
  - `ip_protocol_deliver_rcu()` will invoke `tcp_v4_rcv()` or `udp_rcv`
- Then, the skbuff will be enqueued to a `sock` structure
- The `rcvmsg` method will get it from there...