Networking

Section Contents:

- Overview
- The Socket API
- SKBuffs
- Network Devices
- Linux IP and TCP/UDP

Networking Overview

Networking presents several problems to an O/S:

- Exposure to hostile, unvalidated activity
- Complex asynchronous operations
- Many protocols exist in many arrangements over many transports
- Performance is critical

Network Performance Issues

To maintain high performance:

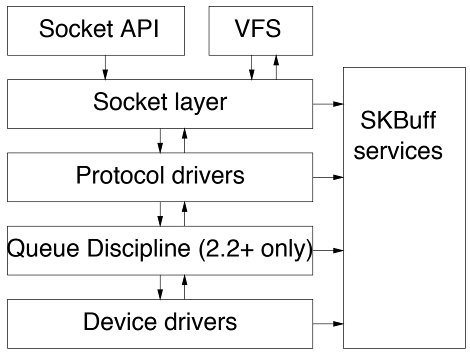
- Perform as few copies as possible
- Perform copy+checksum
- Align headers on cache boundaries
- Keep interrupts enabled as much as possible
- Cache all recent routing decisions

Networking Overview

We can define networking as a *stack* of interacting, interchangeable components

- Separate functionality from transport
- Provide a uniform API layer
- Enable modular implementation
- Compare with the Streams model. . . (compromise between modularity and performance)

Networking components



The Networking API

Linux implements a BSD Socket API:

- Allocate a `struct socket` for each open socket
- Associate a `struct inode` with each socket
- The API is entirely portable between protocols
- Use socket/inode strategy functions to implement protocol-specific routines

The `struct socket`

A socket deals only with the API's communication endpoint. It contains:

- Basic socket type information
- Strategy function entry points
- List of other sockets connected/awaiting connection

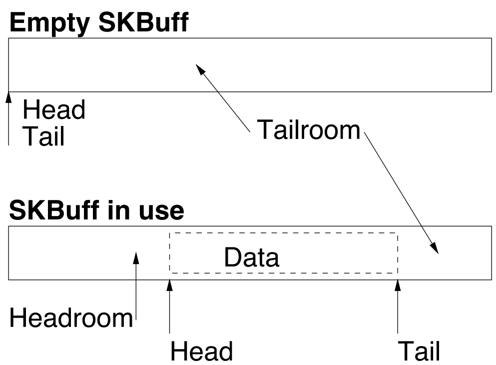
The socket encodes no information about the state of an actual protocol

Moving data between layers

All networking data is built up and passed around in `struct sk_buff` buffers.

- All packets are stored contiguously
- Data may be appended or prepended to the `sk_buff` if space allows
- Facilities are provided for easy queueing of `sk_buffs`
- Provide atomic (interrupt-safe) `sk_buff` operations

The SKBuff



Network Device Drivers

Provide a standard interface to networking device drivers. `struct net_device` describes any single network device, containing:

- Interface name
- Resources allocated to the device
- Interface status
- Device driver strategy routines

Getting data to a device driver

Sending is relatively simple. The device management layer maintains one queue of `sk_buffs` for each priority at each device.

- `dev_queue_xmit()` queues a `sk_buff` to a device
- If necessary, build driver-specific headers when packet is queued (ARP)
- Drop packets if driver queue length is exceeded (device's `LINK_STATE_XOFF` state bit is set to throttle output)
- Send the packet only if the driver is idle

Getting data to a device driver: Scheduling

Under 2.2+, device transmit queues have a `struct Qdisc` "queue discipline" to schedule packets.

- Presents a standard interface for inserting/removing packets on an output queue
- `net/sched/sch_*.c` implements multiple queuing policies: FIFO, prioritised, traffic shaper...
- `net/sched/cls_*.c` implements multiple classifier schemes: each packet can be selected on by route, firewall rull or by more complex policies.

Getting data to a device driver: `dst_entry`

We maintain a `dst_entry` structure to identify recent packet destinations, which:

- Corresponds to a higher protocol's routing decision (eg. IP's `struct rtable`)
- Maintains:
  - `hh_cache` pointer (for ARP resolution)
  - Per-path protocol state (MTU)
  - Rate limiting counters (RSVP)

Getting data from a device driver

Receiving data is harder: it is always received on an interrupt.

2.2 deals with synchronisation issues are dealt with by using the interrupt bottom-half

2.3 introduces the new *softnet* architecture:

- Separate transmit and receive soft_irqs are maintained for each CPU
- All net data structures are now properly SMP-spinlocked to allow concurrent interrupts
- Extensive use of r/w locks

Getting data from a device driver

The device driver receive path is first triggered in a top-half hard interrupt:

- Allocate a `sk_buff`
- Receive the packet into the `sk_buff` and queue it with `netif_rx()`, which:
  - Adds the packet to the input queue for this CPU (global `backlog` on 2.2)
  - Raises the soft IRQ to process the queue

The Network Soft IRQ

`net_rx_action` (`net_bh` on 2.2) must:

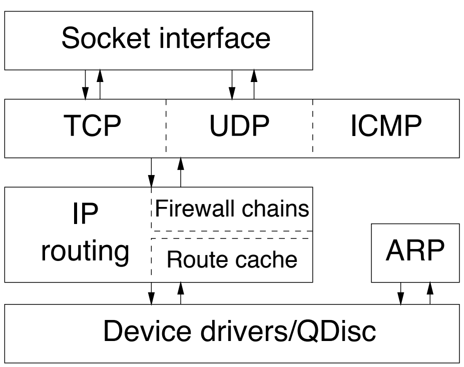
- Send any outstanding queued packets to their drivers/qdiscs
- Roll through the `backlog`:
  - Try to bridge or fastroute the packet first
  - Fetch the protocol ID (set by the device driver)
  - Pass the packet to the appropriate protocol: hash protocol lists by protocol ID

Linux and TCP/IP

Multiple protocols are supported well in the kernel. IP is just one, glued to the net stack with:

- All network devices maintain multiple protocol-specific pointers; for IP, use the per-device `in_device` (holds IP addresses etc.) struct
- The `struct sock` maintains much internal TCP-specific information for active, bound connections:
  - Sequence numbers
  - Window/congestion control
- Have one `struct sock` per connected `struct socket`

TCP/IP Components



IP Device Interface

`ip_rcv()` handles all packets coming from the device driver layers, and performs:

- Accounting/Firewalling
- Assignment to alias device
- Reassembly of IP fragments
- Delivery of packet to a local protocol handler, or
- Forwarding of routable packets via `ip_forward()`

IP Routing Decisions

`ip_rcv()` needs to distinguish between packets destined for the local machine and those to be forwarded.

- Perform a full `ip_route_input` when we see the packet
- We only make one pass over the routing tables
- The routing `dst_entry` is stored in the `sk_buff`
- Finally, pass the skb to the `input` method in the `dst_entry` for forwarding or local delivery

IP Fragment Management

Manage incoming fragments by maintaining a cache of incomplete datagrams:

- Maintain a `struct ipq` for each incomplete datagram
- Maintain a `struct ipfrag` for each fragment
- Hold all outstanding fragments on a `ipq` list, and all `ipqs` on the `ipq_hash` hash table

IP Fragment Processing

Incoming IP fragments are passed to `ip_defrag()` either on local delivery or if a netfilter module wants to assemble fragments:

- Search the `ipqueue` for an incomplete datagram (`ipq`) which matches this packet, and create a new `ipq` if necessary
- Set a timer to expire the `ipq` in 30 seconds
- When all fragments have arrived, call `ip_glue()` to merge them, and return a new `sk_buff`

IP Forwarding

`ip_forward()` deals with packets *not* destined for a local socket:

- Use the `sk_buff`'s existing routing information to work out the next hop
- Generate diagnostic ICMP for unroutable packets
- Call `ip_send()` to either fragment the packet, or directly `dev_queue_xmit()` on the destination interface
- Simply drop packets if we don't have enough memory

IP Routing

We maintain two separate routing databases:

- the permanent FIB (Forwarding Information Base)
  - Set up by the user
  - Indexed by route mask, type-of-service, and source address and interface;
- a transient route cache.

The Route Cache

The `struct rtable` encodes and caches a single routing decision:

- `ip_route_output` returns a `struct rtable` (processes like ICMP may want to know a route before they have a `sk_buff` to send)
- `ip_route_input` sets the `sk_buff` route directly.
- For performance, use a hash table to cache routes
- If route not cached, pass it to `ip_route_*_slow`

The Route Database

FIB organisation is optimised for performance:

- Group routes into *zones* according to the netmask length
- For each zone, maintain a separate 256-entry hash table of routing nodes
- Allow `fib_node` routing nodes to share as much data (interface, protocol, metrics etc.) as possible via shared `fib_info`
- Allow update via *netlink* character device

NetFilter

The 2.3 kernel's NetFilter code replaces the old firewalling/NAT code:

- Separate rule sets (*iptables*) for incoming; forwarded; locally received; locally injected; and output packets, plus user-defined iptables
- A generic `NF_HOOK` call can be called anywhere in any network stack, specifying which iptable to run
- New tables or rule types may be registered dynamically
- Rules may return a verdict of accept or drop, and may also modify or steal the `sk_buff`

NetFilter uses

The NetFilter framework implements many pieces of the old 2.2 IP stack:

- Firewalling is accomplished by calling the *filter* iptable
- The *NAT* iptable can be used to modify packets:
  - source-address NAT is used to implement masquerading
  - dest-address NAT is used to implement transparent proxying

IP Aliases

Kernel transparently supports IP aliases:

- Autodetect interface names of the form “dev:num”
- Link each alias to its root interface
- Routing logic reroutes packets destined to an alias to the root interface
- ARP support is automatic for all defined interfaces

The Address Resolution Protocol

ARP is the protocol which resolves IP addresses into ethernet HW addresses. The old style code special-cased ARP:

- `struct arp_table` maintains a single ARP entry
- Maintain a hash table of ARP entries
- Each entry references a list of `sk_buffs` held up for this ARP request
- ARP is called by the device’s `rebuild_header()`
- Use *netlink* interface to the arpd user-mode cache

Neighbourhood maps

The ARP database is just one special case of a *neighbourhood table* (also used for IPv6 neighbourhood discovery):

- `struct neigh_table` provides hashed lookup and management of `struct neighbours`
- Each `neighbour` references a `hh_cache` hardware header for the link level
- The ARP database creates provides neighbourhood methods for ARP solicitation

The UDP Protocol

UDP has no connections: all we need to do is route packets to open sockets.

- Rely on the socket API layer to create sockets for us
- Use `udp_v4_lookup()` to identify the destination socket for incoming packets
- Maintain a per-protocol hash table of sockets
- Maintain a single-entry last-used-socket cache
- Transmitted packets go straight to the IP layer

The TCP Protocol

TCP has major differences from UDP, including:

- Connections maintained: sockets have a backlog list of pending connections
- Many non-data types of packet to be dealt with: maintain connection state machine
- Data transport is reliable
- Maintain flow rates and round-trip times for flow control

TCP Reliable Data Stream

In order to present reliable communications to the API:

- Keep `sk_buffs` on the `struct sock` write queue until acknowledged
- Retry transmits automatically on timeouts
- Queue incoming packets until they can be presented in order