

Linux Network stack overview

Florian Westphal

July 2017

Problems

several general Linux are also true for network stack:

- Hardware:
 - 1 cpu vs. SMP
 - LTE? WLAN? 40Gbit Ethernet?
- Usage:
 - Desktop, Server, Router, Bridge, ...
 - Server \neq server: e.g. Database workload vs. file server

→ low Latency, high throughput, low cpu cycles/resource usage, large feature set (IPSEC, NAT, traffic shaping and accuting, ...)

Agenda

- RX: how do packets end up in a program?
- TX: how does data get from program out to the network?
- tricks: GSO, GRO, RSS, XPS, ...
- Byte Queue Limits, TSQ, ...

how does the kernel receive/collect packets from hardware?

3 Ways:

- ① interrupt + percpu backlog
- ② softinterrupt: NAPI
- ③ busypoll: `SO_BUSY_POLL` setsockopt – reduce latency at cpu cost

Simple ('obsolete') nic/driver

- (very) small memory buffer to hold 1 or 2 packets
- interrupt handler allocates memory, copies packets from hardware memory
- packet is placed on a per-cpu queue for later processing in softirq

Interrupt

- asynchronous event, interrupts currently running program
- high irq/s: → system becomes unusable

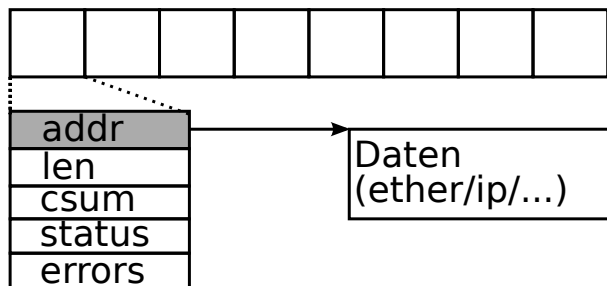
NAPI: mixes interrupt model with polling

- 1 first interrupt disables further interrupts from nic
- 2 schedule a polling cycle (`softirq`)
- 3 next softinterrupt will process packets from nic
- 4 softinterrupt takes too long?
`ksoftirqd` → network stack processing moves under scheduler control
- 5 no more packets? re-enable nic rx interrupt

new nic drivers

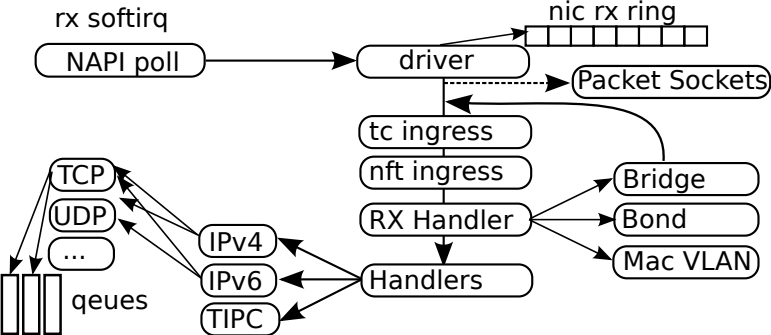
- 10, 40, 56 (oder mehr) GBit
- NAPI – all packet processing occurs via `sofirq`
- Ring-Buffer / Array ("RX Queues")
- driver allocates buffer for each ring buffer slot
- hardware DMAs incoming data into these buffers
- hardware also performs other tasks: checksum offload, rxhash, ...
- multiple TX and RX queues, interrupts per queue, RSS – map rxqueue → CPU

nic rx ring (example)



- driver checks status bit of next descriptor
- no data? reenale rx irq and done
- otherwise proess the pacet

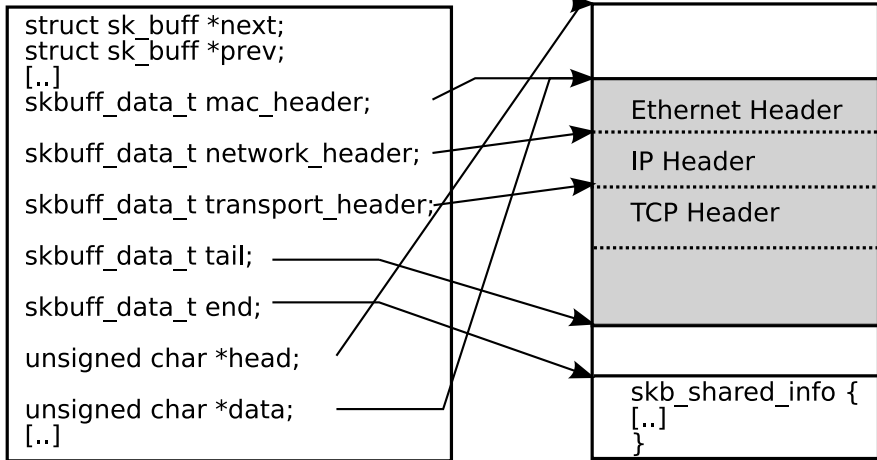
packet rx



socket buffer

- skb: "socket buffer", kernel data structure to represent a packet
- RX/incoming: allocation done in the driver
- locally generated packets: allocation done in tcp/udp etc. stack
- contains metadata, e.g. incoming interface, routing information, etc.
- len vs. truesize – socket mem accounting

sk_buff

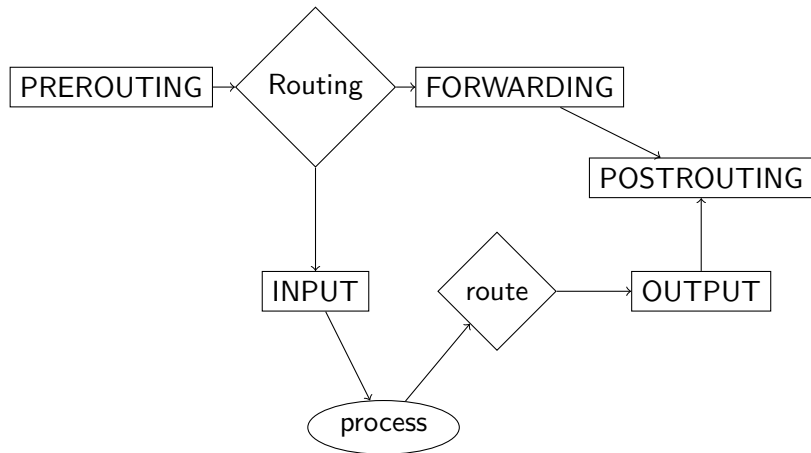


`next/prev`: for lists, e.g. send and receive queues

`sk_buff_data_t`: offset with start of particular protocol headers

others: socket, routing information, `rxhash`, IPSEC, ... on rx, most fields will be unset/empty and are filled in while packet is passed to next layers

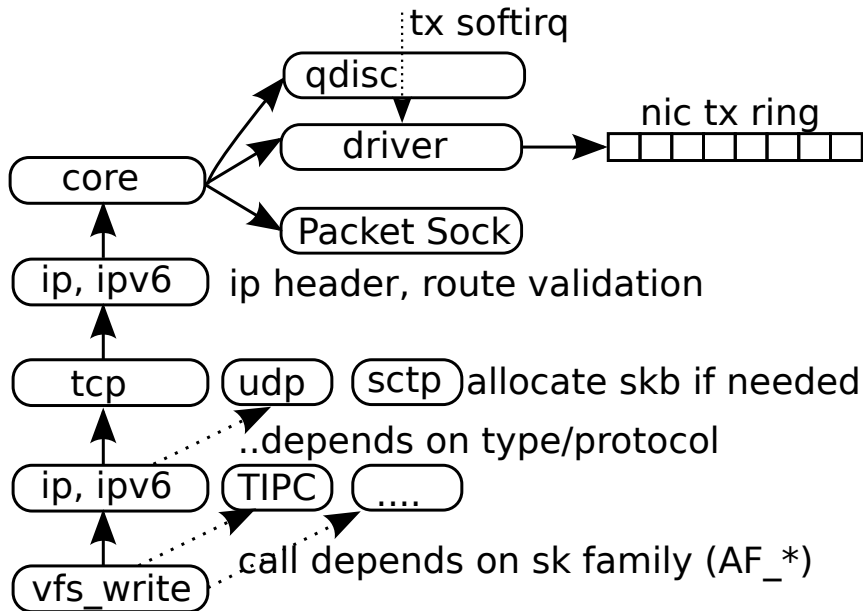
IPv4 Architektur



how does data end up in the network?

- ① direct: programm sent it
- ② timer, e.g. tcp retransmit, window probe, ...
- ③ via rx: forwarding (router, bridge), tcp acks, ...
- ④ tx softirq (qdisc)

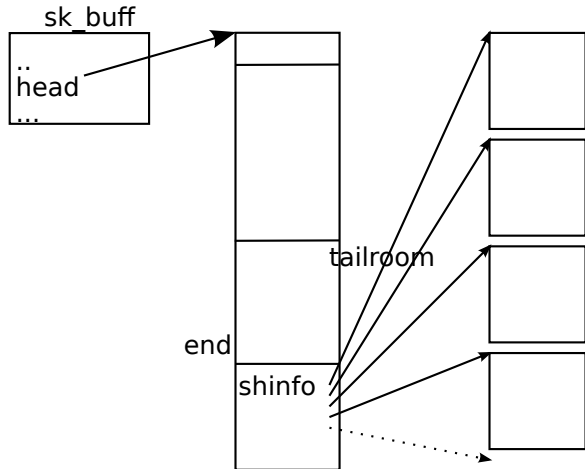
packet tx



GSO, GRO

- `skb != packet`
- `skbs` can be a lot larger than a physical `mtu`
- `transmit`:
 - TSO: TCP Segmentation offload
 - GSO: Generic Segmentation offload
- `rx`:
 - GRO: Generic Receive offload
 - occurs even before packet sockets
 - driver needs to use `napi_gro_receive()` api
 - merges logically contiguous packets in one GSO `skb`

nonlinear skbs



pages are references via shinfo area
sendpage, gro/gso, ...

scaling: RSS Receive Side Scaling, RPS

- one IRQ per rx-queue /proc/interrupts:

```
      CPU0 CPU1 CPU2 CPU3 CPU4 CPU5 CPU6 CPU7
28: 31189 4389 2853 2557 6315 4035 3002 2130 eth0
```

- /proc/irq/28/smp_affinity – ff – default
printf %x \$((1 << 7)) > /proc/irq/28/smp_affinity
- determines which CPU can perform rx softirq
- incoming packets are distributed among rx queues by hardware
- usually done by a hash on packet header data
- modern nics have programmable filters
- RPS: "RSS in software"
/sys/class/net/\$dev/queues/rx/rps_cpus

Documentation/networking/scaling.txt

scaling: XPS (Transmit Packet Steering)

- ideal case: CPU that runs application maps to a particular tx queue
- tx-completion occurs on the same cpu, i.e. alloc/free on same processor
- kernel saves used queue in skb/socket
- `/sys/class/net/$dev/queues/tx/xps_cpus` – set which cpus use tx queue
- mq scheduler, its possible to set a different qdisc for each tx queue

Byte Queue Limits (BQL)

- aims to reduce unneeded buffering:
 - buffering increases latency and jitter
 - can reduce throughput
- Idea: calculate the read transmit rate
 - 1 on tx start: how many bytes are transmitted?
 - 2 on tx completion/ende: how many bytes completed transmissio
- needs to be added at driver level
- prerequisite for `xmit_more` in drivers

TX Batching, xmit-more

- `xmit_more` flag in `skbuff`
- problem: update of nic hardware tx tail pointer gets too expensive
- would like to set it only once when multiple packets get sent
- this already happens with `gso/tso` offloads
- when nic is backlogged, `qdisc` starts to fill up
- uses `bql` to estimate how much data nic can take
- dequeue multiple `skbs`, set `skb->xmit_more` flag on all but last `skb`

allows drivers to not touch tx tail descriptor in some cases

TSQ (TCP small queues)

"BQL for sockets"

- similar goals as BQL
 - increase fairness among flows
 - conserve memory
- why read more data from disk if there are 100kb of unset data in local queues...?
- idea:
 - only 2 packets per stream in qdisc or 1 ms delay or
 - `net.ipv4.tcp_limit_output_bytes`
- uses socket wmem accounting
 - when tcp has to allocate new skb it checks `sk->sk_wmem_alloc`
 - if overlimit: block or `EWOULDBLOCK`
 - nic tx completion: `skb_orphan()`

(micro-)optimizations, design choices

- place functionality in kernel modules
- otherwise: static keys (netfilter hooks, udp encap/ipsec nat-t, ...)
- even atomic ops are too slow
 - percpu data
 - RCU
- Batching: GRO, GSO, xmit_more...
- optional HW offloads
- sysctl and other interfaces for further fine tuning depending on expected workload

current development focus

- zerocopy tx
- container/network namespaces/virtualization (vrf)
- offloading: ktls, ipsec
- XDP
- (unfortunate, imo) trend to move functionality from userspace to kernel (instead of other way around) because thats easier to do