Der Linux Netzwerk-Stack

Florian Westphal

August 2015

Probleme

Etliche generelle Linux-Probleme gelten auch für den Netzwerkwerkstack

- Hardware:
  - 1 cpu vs. SMP
  - LTE? WLAN? 40Gbit Ethernet?
- Einsatzzweck:
  - Desktop, Server, Router, Bridge, ...
  - Server $\neq$ server: z.B. Datenbank vs. Dateiserver

$\rightarrow$ geringe Latenz, Hoher Durchsatz, wenig CPU-Zyklen, Speichersparend, fairness, großes Feature-Set (IPSEC, NAT, Traffic Shaping, ...)

Agenda

- RX: Wie kommt ein Paket zum Programm?
- TX: Wie kommen Daten vom Programm ins Netzwerk?
- Tricks: GSO, GRO, RSS, XPS, ...
- Byte Queue Limits, TSQ, ...
- Aktuelle Schwerpunkte

Wie kommt ein Paket zum Kernel?

3 Wege:

1. Interrupt + percpu Backlog
2. Softinterrupt: NAPI
3. Busypoll: `SO_BUSY_POLL` setsockopt – reduziert Latenz auf Kosten von CPU-Zyklen

Einfache(r) ('obsolete') Netzwerkkarte/Treiber

- (sehr) kleiner Speicherbereich um 1 oder 2 pakete zu speichern
- Interrupthandler allokiert Speicherplatz, kopiert das Paket aus dem Hardware-Speicher
- Paket kommt in eine Pro-CPU Warteschlange, Weiterverarbeitung erst durch Softirq

Interrupt

- Unterbricht laufendes Programm
- zu viele Interrupts/s: $\rightarrow$ System unbenutzbar

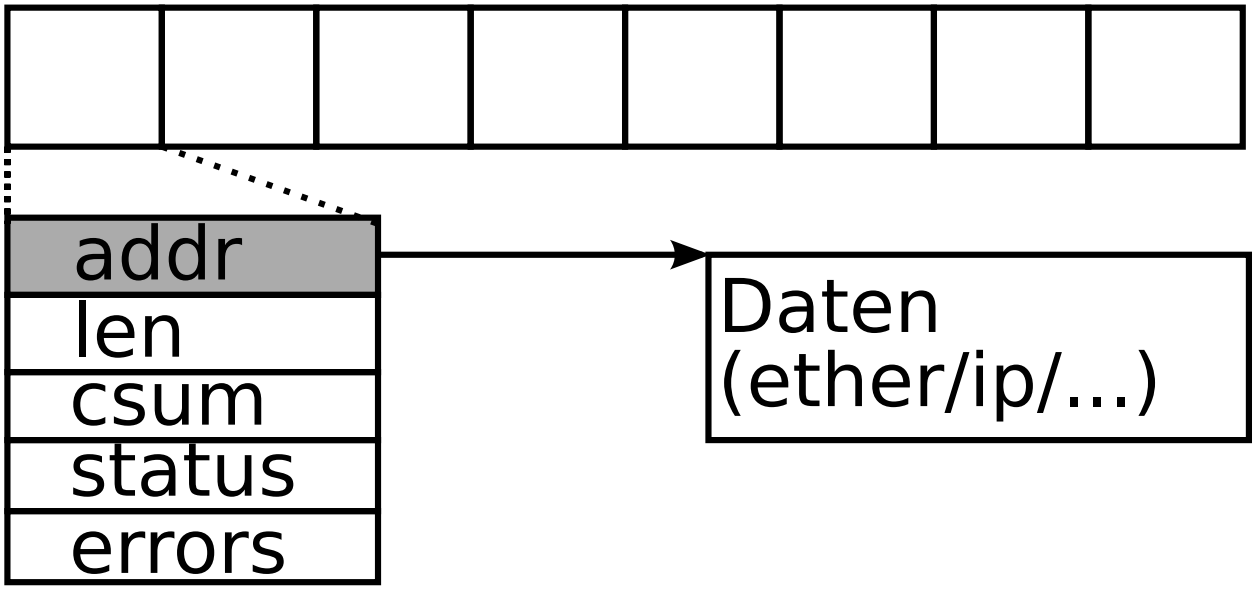
NAPI: Mischung aus Interrupt + Polling (periodisches Abfragen)

1. Mit erstem Interrupt auf Netzwerkkarte abschalten
2. 'Vormerken' eines Abfragezyklus
3. nächster Softinterrupt arbeitet anstehende Pakete ab
4. Softinterrupt dauert zu lange?
   `ksoftirqd` $\rightarrow$ Verarbeitung unterliegt dann Scheduler
5. Wenn keine Pakete mehr da: Interrupts der Netzwerkkarte werden wieder angeschaltet

Neuere Ethernet-Karten/Treiber

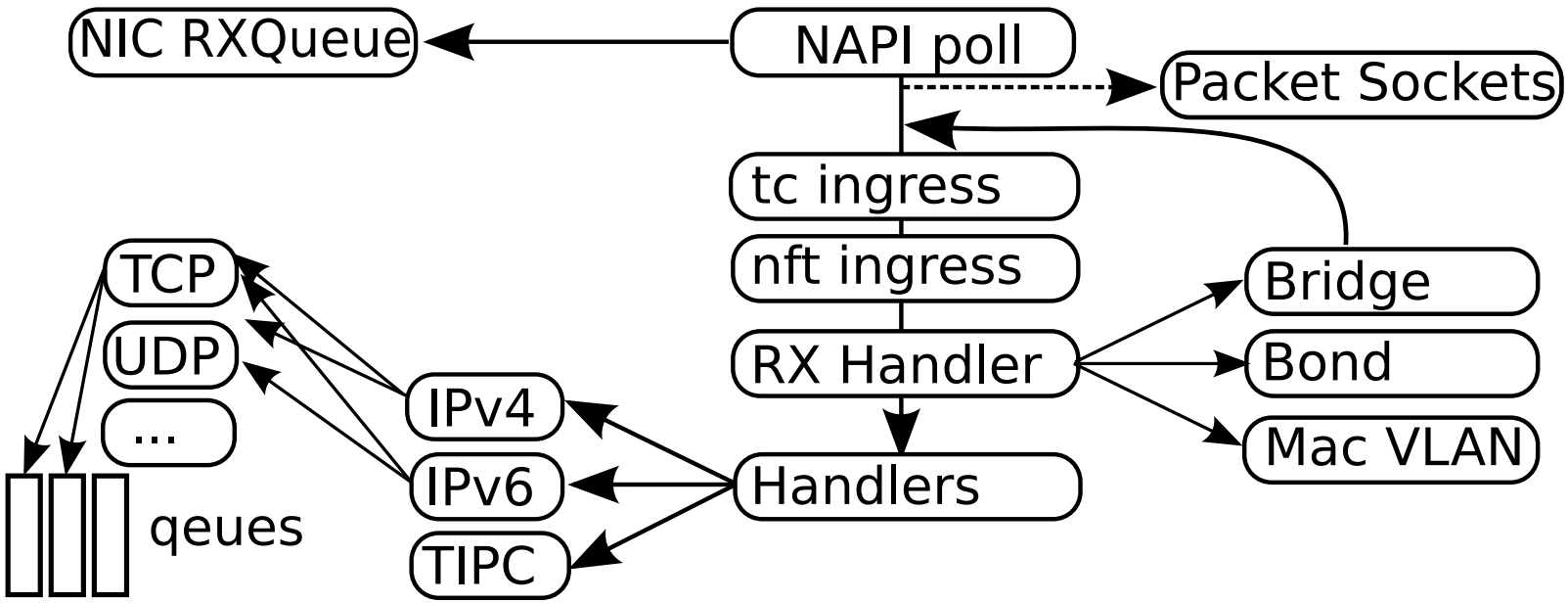
- 10, 40, 56 (und mehr) GBit
- Ring-Buffer / Array ("RX Queue")
- Treiber allokiert pro Ring-Buffer Element ("Deskriptor") einen Speicherblock
- Hardware schreibt eingehende Daten direkt in diese Blöcke (DMA) $\rightarrow$ keine Kopie durch CPU
- viele Features direkt in Hardware, z.B. Prüfsummen, programmierbare Filter, ...
- mehrere TX und RX-Queues, Interrupt per queue, RSS – Zuordnung Queue $\rightarrow$ CPU
- NAPI, d.h. Paketverarbeitung immer per Softirq

RX-Ring Netzwerkkarte (Beispiel)



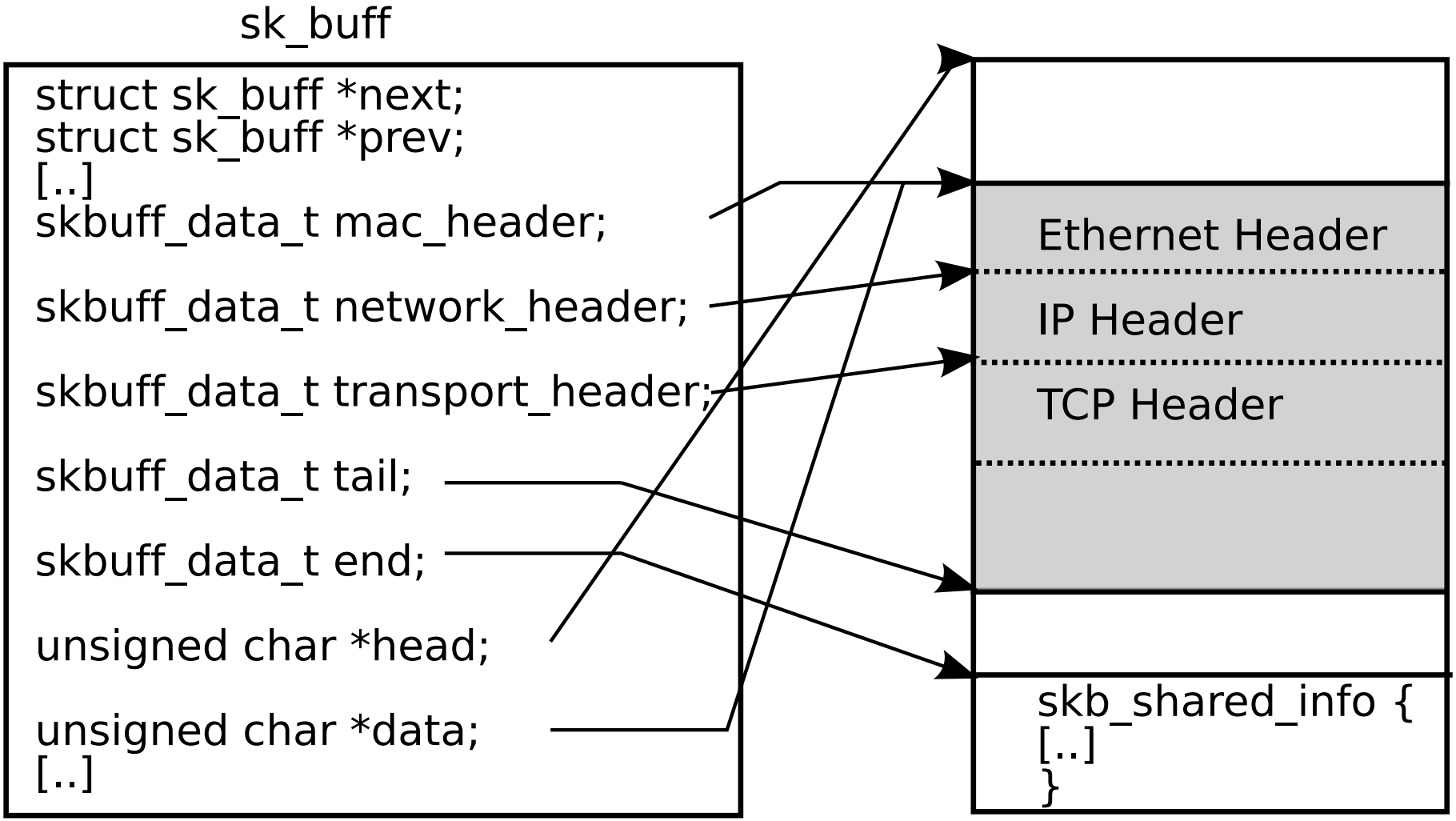
- Treiber testet status-bits des nächsten Deskriptors
- Wenn nichts empfangen: "Interrupts ein und ende"
- sonst wird Paket verarbeitet
- Addr wird vom Treiber ausgefüllt, alles andere durch HW

packet rx


NIC RXQueue
NAPI poll
Packet Sockets
tc ingress
nft ingress
RX Handler
Bridge
Bond
Mac VLAN
Handlers
IPv4
IPv6
TIPC
TCP
UDP
...
queues

socket buffer

- skb: "socketbuffer", Kernel-Datenstruktur, repräsentiert ein Paket
- RX/eingehend: Allokation durch Netzwerktreiber
- Lokal generierte Pakete: Allokation durch TCP/UDP Implementierung, etc.
- **DIE** Datenstruktur im Netzwerkstack
- Hält Metadaten
- len vs. truesize – socket mem accounting



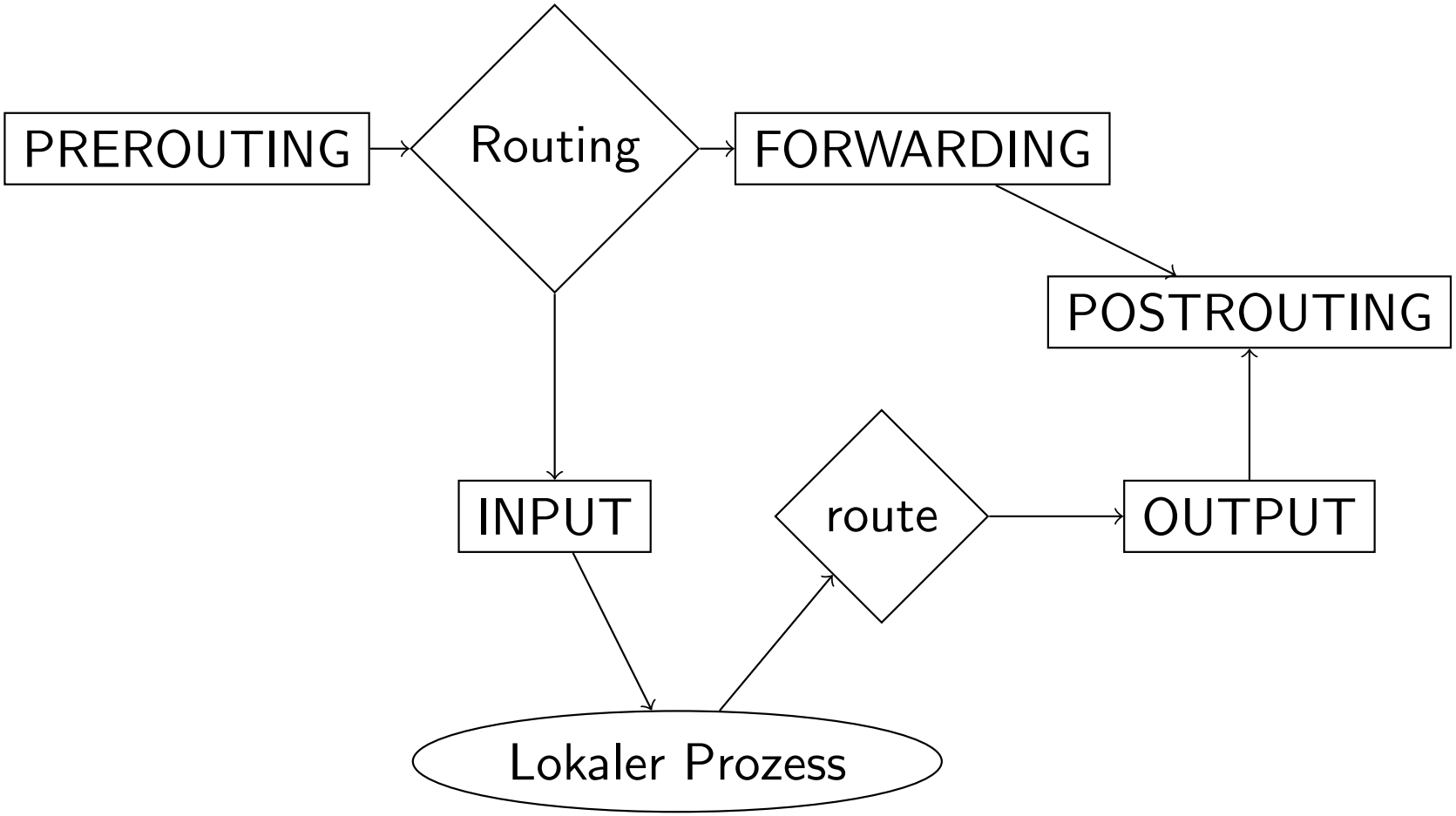
`next/prev`: Listen, z.B. sende/Empfangswarteschlange

`sk_buff_data_t`: Offset, Stelle an der bestimmter Header beginnt

Viele Weitere: Socket, routing-Informationen, rxhash, IPSEC,

. . . Fast alles zuerst "leer", wird nach-und-nach durch Stack "ausgefüllt"

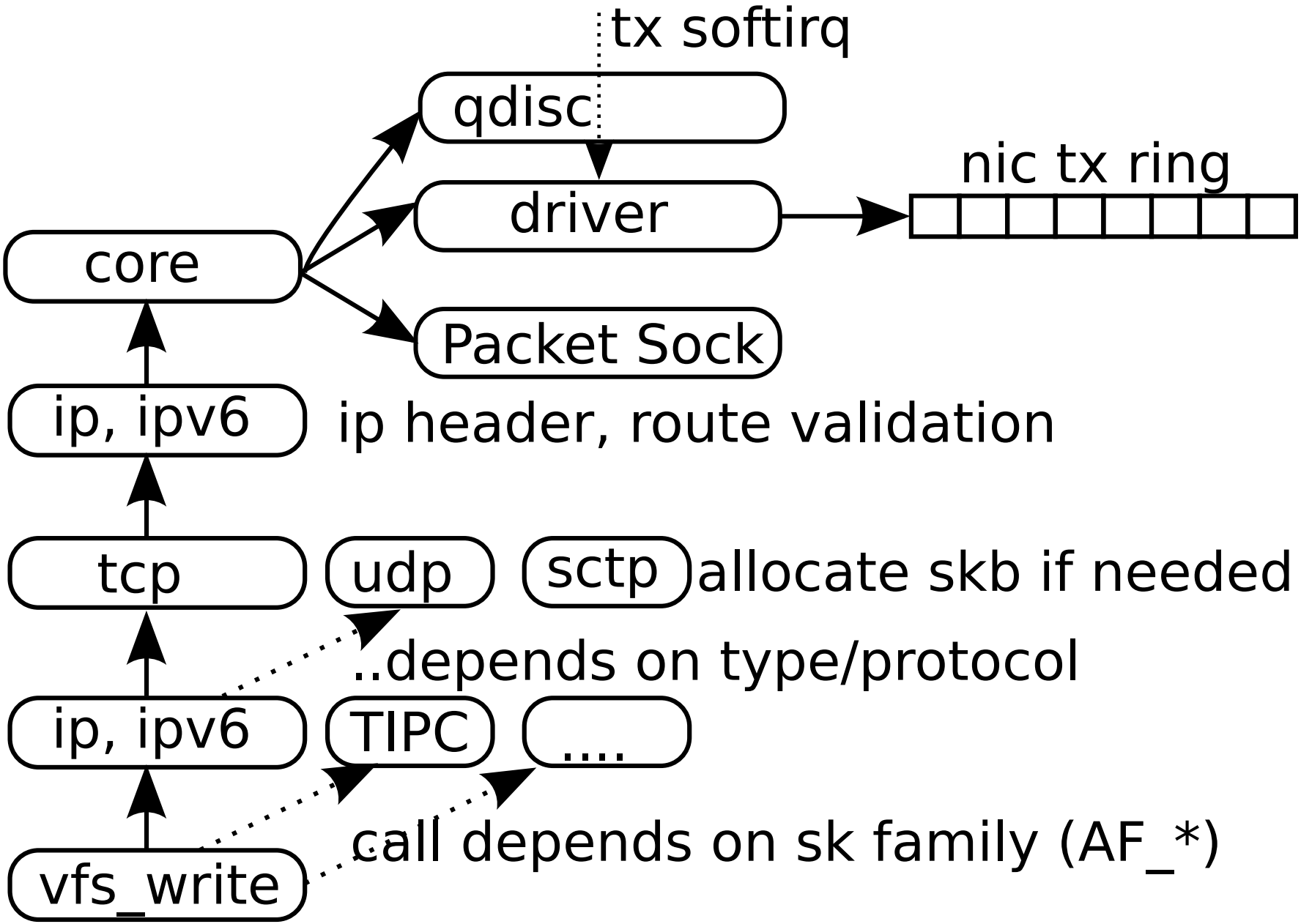
IPv4 Architektur



Wie kommt ein Paket zur Netzwerkkarte?

1. direkt: Programm sendet etwas
2. timer, z.B. tcp: retransmit, window probe, ...
3. via rx: forwarding (router, bridge), tcp acks, ...
4. tx softirq (qdisc)

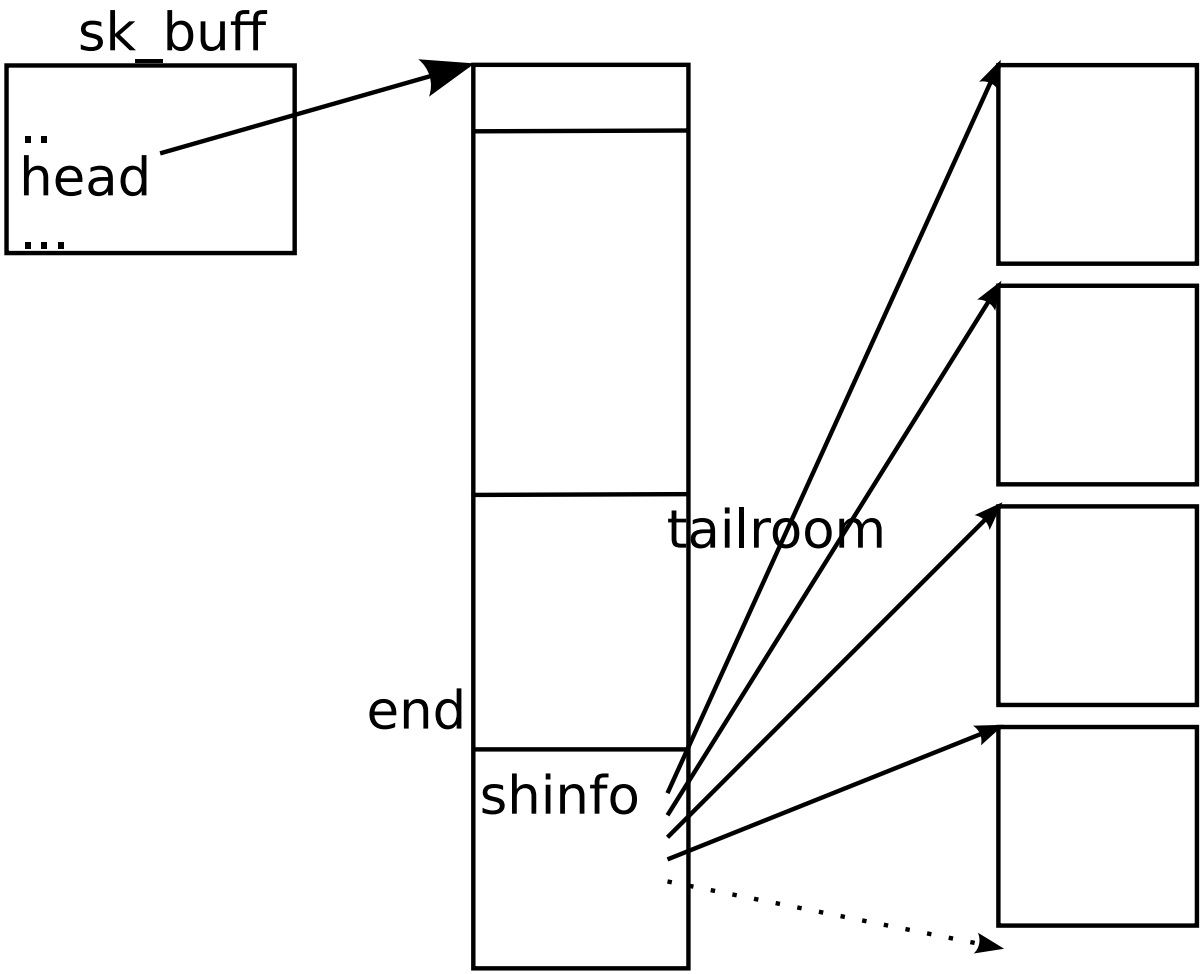
packet tx


tx softirq
qdisc
driver
nic tx ring
core
Packet Sock
ip, ipv6
ip header, route validation
tcp
udp
sctp
allocate skb if needed
..depends on type/protocol
ip, ipv6
TIPC
....
call depends on sk family (AF_*)
vfs_write

GSO, GRO

- skb ! = paket
- skb kann wesentlich grösser als MTU sein (bis 64k)
- transmit:
  - TSO: TCP Segementation offload
  - GSO: Generic Segmentation offload
- rx:
  - GRO: Generic Receive offload
  - noch vor packet-sockets
  - Treiber muss `napi_gro_receive()` api benutzen
  - "Verschmelzen" von paketen zu einem GSO-skb

nichtlineare skbs



pages werden via array in shinfo referenziert

sendpage, gro/gso, ...

Skalierung: RSS Receive Side Scaling, RPS

- ein IRQ pro 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 $((1 << 7)) > /proc/irq/28/smp_affinity`
- legt fest, auf welche CPU rx-softirq ausgeführt wird
- eingehende pakete werden von HW auf rx-queues aufgeteilt
- typischerweise hash-basiert (l3 und oder l4)
- neuere nics haben programmierbare filter
- RPS: "RSS in software"
  `/sys/class/net/$dev/queues/rx/rps_cpus`

`Documentation/networking/scaling.txt`

Skalierung: XPS (Transmit Packet Steering)

- idealfall: CPU auf der die Anwendung läuft verwendet eine bestimmte TX queue
- mapping cpu $\leftrightarrow$ tx queue
- tx-completion auf derselben cpu, d.h. auch alloc/free
- kernel speichert verwendete queue im skb/socket
- `/sys/class/net/$dev/queues/tx/xps_cpus` – legt fest welche CPUs tx queue benutzen
- `mq` scheduler, damit auch (verschiedene) qdiscs per tx-queue möglich

Byte Queue Limits (BQL)

- Ziel: Unnötiges Buffering verhindern
  - erhöht Latenz und Jitter
  - Kann auch Durchsatz mindern
- Idee: Messen der realen Senderate
  1. bei tx start: wieviele bytes werden gesendet?
  2. bei tx completion/ende: wieviel fertig übertragen?
- TSQ (TCP Small Queues) – `net.ipv4.tcp_limit_output_bytes`

- Batching: GRO, GSO, ...
- Funktionalität in Module auslagern
- sonst: static keys (netfilter hooks, udp encap/ipsec nat-t, ...)
- Parallelisierbarkeit
  - percpu Datenstrukturen (z.B. counter)
  - RCU
- optionale HW offloads
- `sysctl` Interfaces für weiteres Tuning

aktuelle Schwerpunkte

- Skalierbarkeit: "millions of sockets"
- zerocopy tx
- tcp: listen lock removal
- skb-alloc/free in batches
- container/network namespaces/virtualisierung
  - VRF
  - per-namespace netfilter hooks
- offloading: switchdev
- ebpf
- nftables