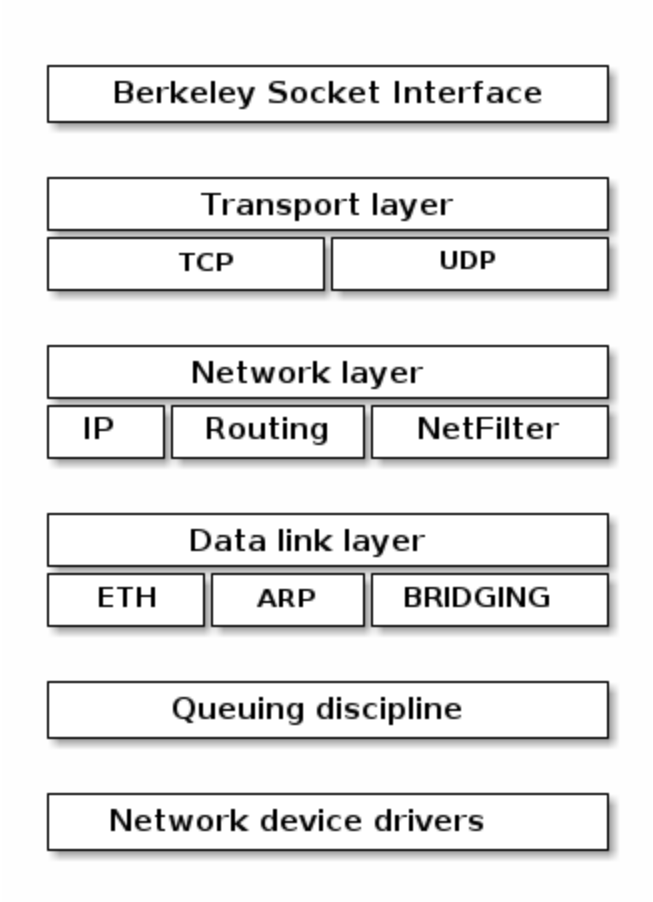
SO2 Lecture 10 - Networking

View slides

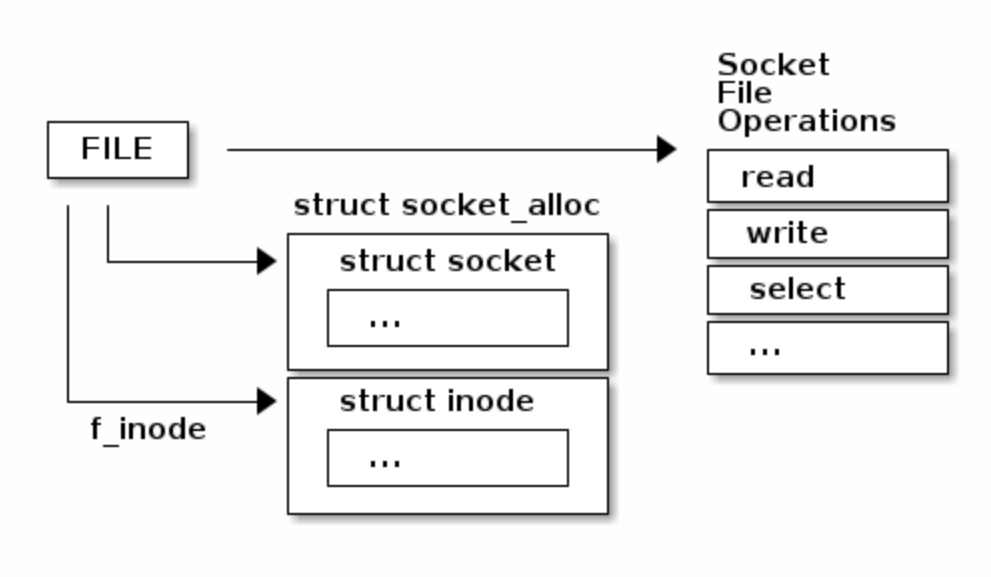
Lecture objectives:

- Socket implementation
- Routing implementation
- Network Device Interface
- Hardware and Software Acceleration Techniques

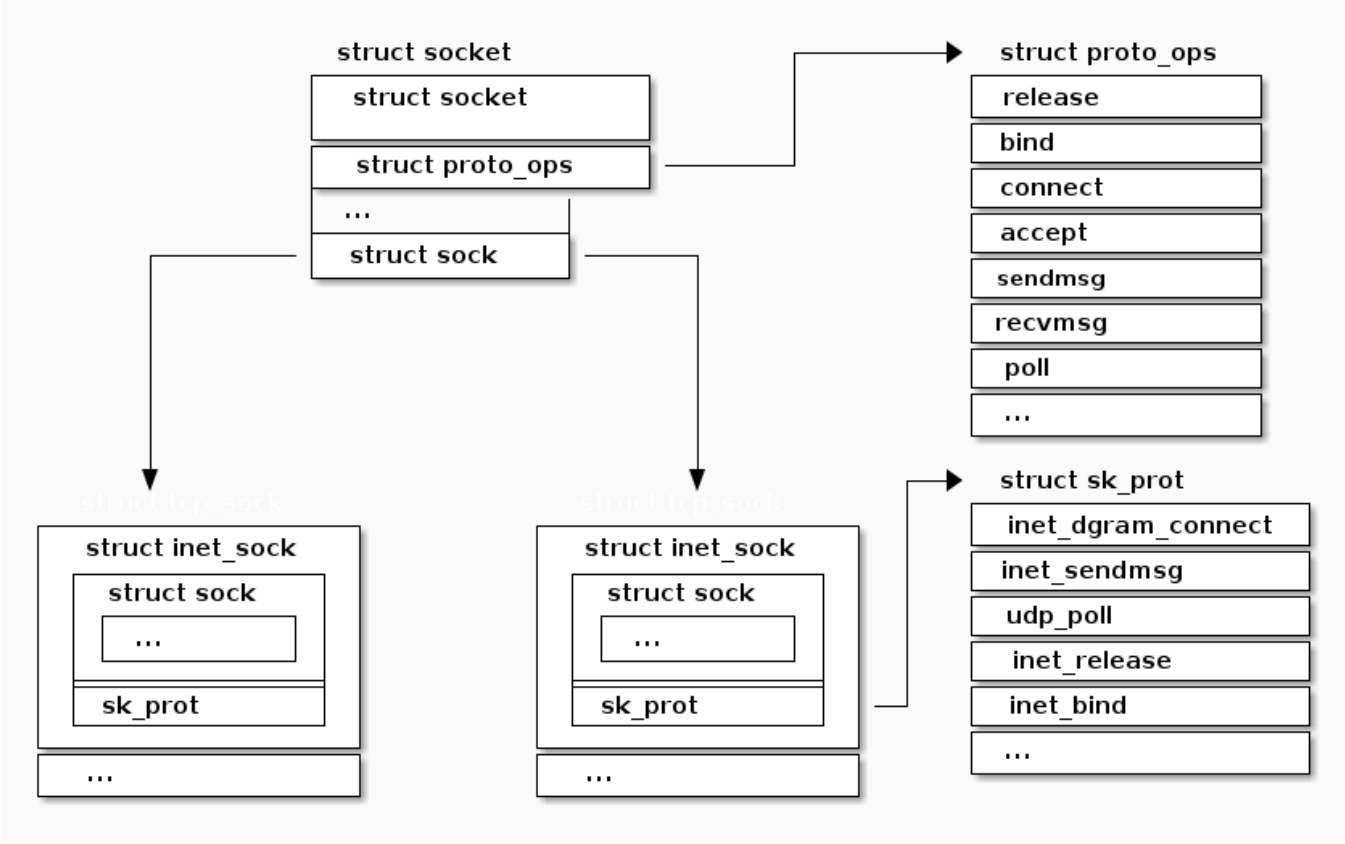
Network Management Overview



Sockets Implementation Overview

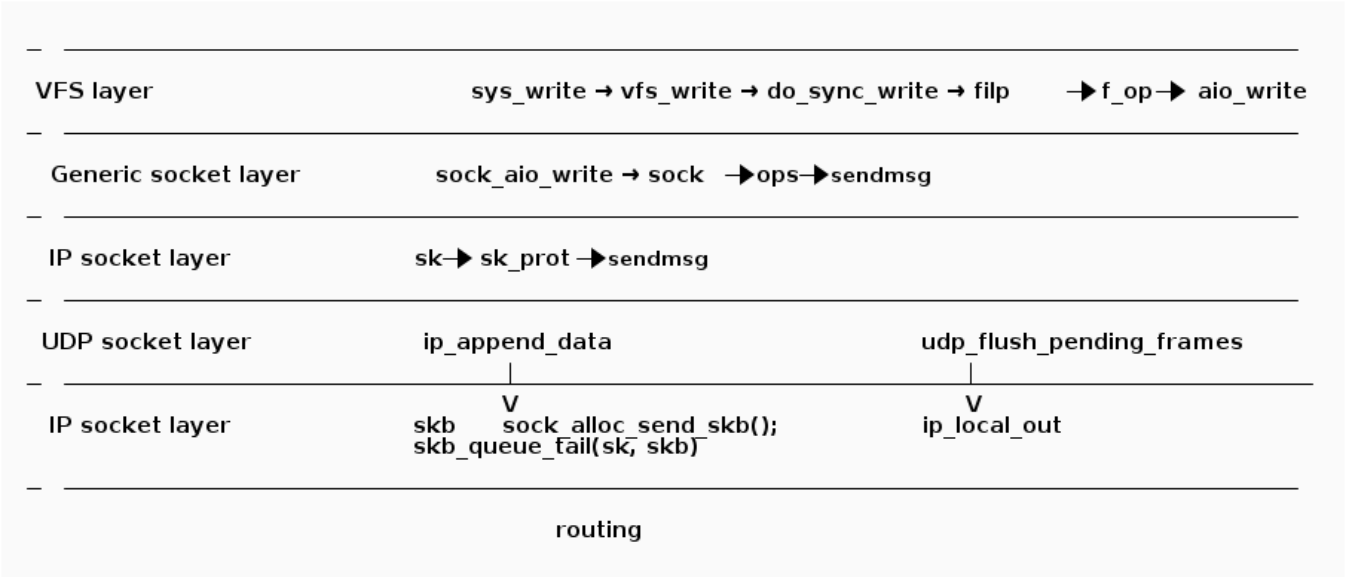


Sockets Families and Protocols



Example: UDP send

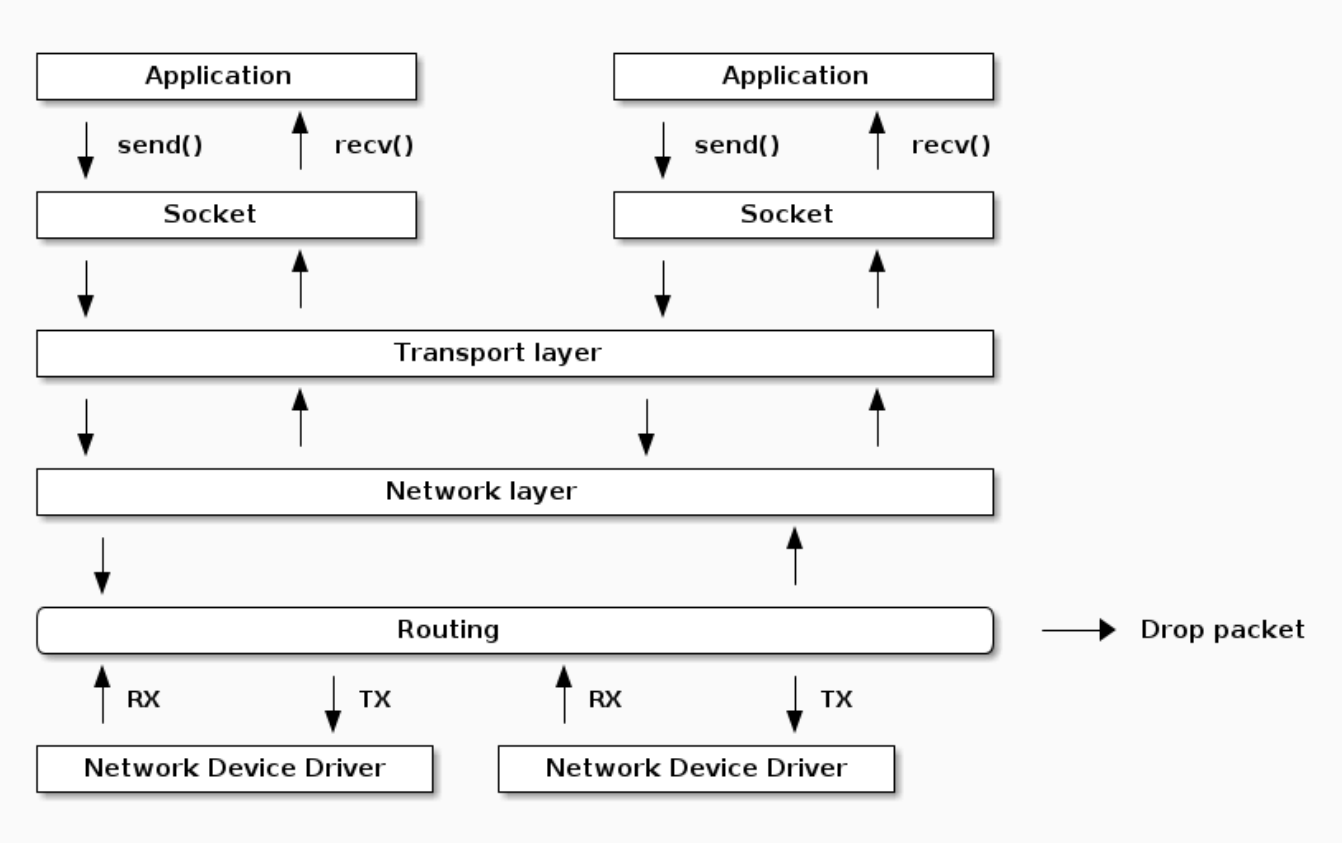
```
char c;
struct sockaddr_in addr;
int s;

s = socket(AF_INET, SOCK_DGRAM, 0);
connect(s, (struct sockaddr*)&addr, sizeof(addr));
write(s, &c, 1);
close(s);
```



Network processing phases

- Interrupt handler - device driver fetches data from the RX ring, creates a network packet and queues it to the network stack for processing
- NET_SOFTIRQ - packet goes through the stack layer and it is processed: decapsulate Ethernet frame, check IP packet and route it, if local packet decapsulate protocol packet (e.g. TCP) and queues it to a socket
- Process context - application fetches data from the socket queue or pushes data to the socket queue

Packet Routing



Routing Table(s)

```
tavi@desktop-tavi:~/src/linux$ ip route list table main
default via 172.30.240.1 dev eth0
172.30.240.0/20 dev eth0 proto kernel scope link src 172.30.249.241

tavi@desktop-tavi:~/src/linux$ ip route list table local
broadcast 127.0.0.0 dev lo proto kernel scope link src 127.0.0.1
local 127.0.0.0/8 dev lo proto kernel scope host src 127.0.0.1
local 127.0.0.1 dev lo proto kernel scope host src 127.0.0.1
broadcast 127.255.255.255 dev lo proto kernel scope link src 127.0.0.1
broadcast 172.30.240.0 dev eth0 proto kernel scope link src 172.30.249.241
local 172.30.249.241 dev eth0 proto kernel scope host src 172.30.249.241
broadcast 172.30.255.255 dev eth0 proto kernel scope link src 172.30.249.241

tavi@desktop-tavi:~/src/linux$ ip rule list
0:      from all lookup local
32766:  from all lookup main
32767:  from all lookup default
```

Routing Policy Database

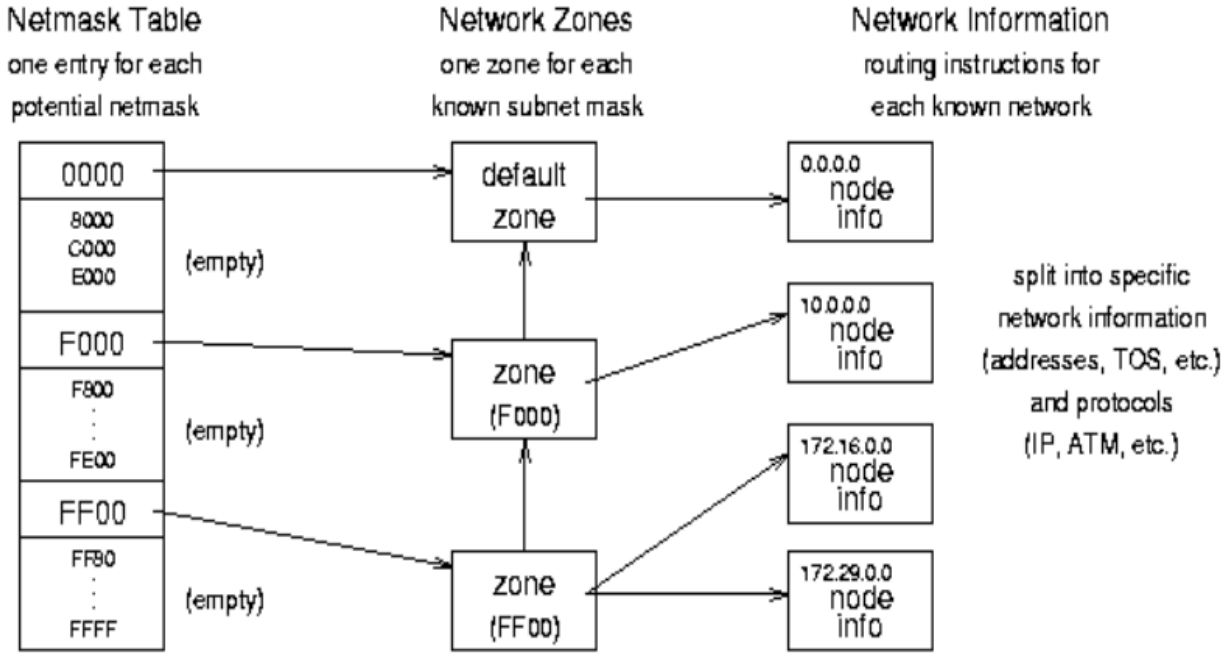
- "Regular" routing only uses the destination address
- To increase flexibility a "Routing Policy Database" is used that allows different routing based on other fields such as the source address, protocol type, transport ports, etc.

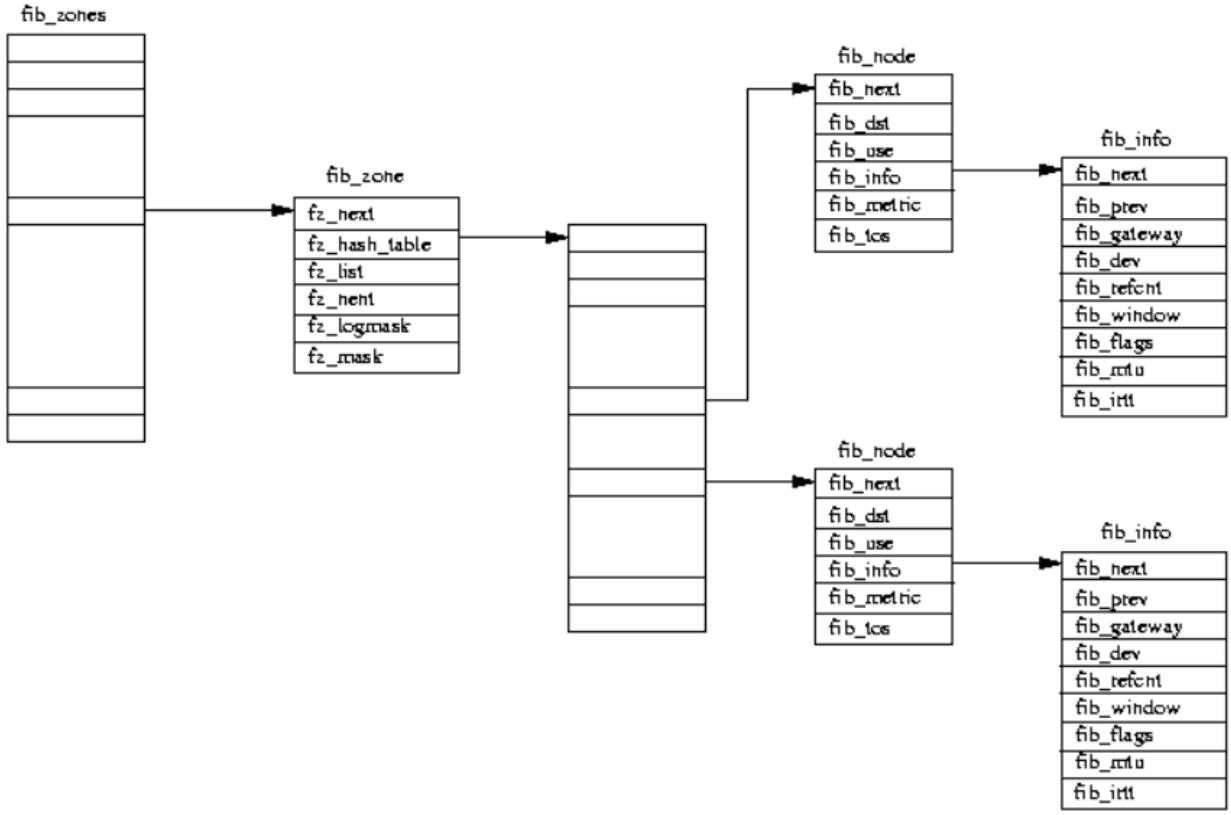
- This is encoded as a list of rules that are evaluated based on their priority (priority 0 is the highest)
- Each rule has a selector (how to match the packet) and an action (what action to take if the packet matches)
- Selectors: source address, destination address, type of service (TOS), input interface, output interface, etc.
- Action: lookup / unicast - use given routing table, blackhole - drop packet, unreachable - send ICMP unreachable message and drop packet, etc.

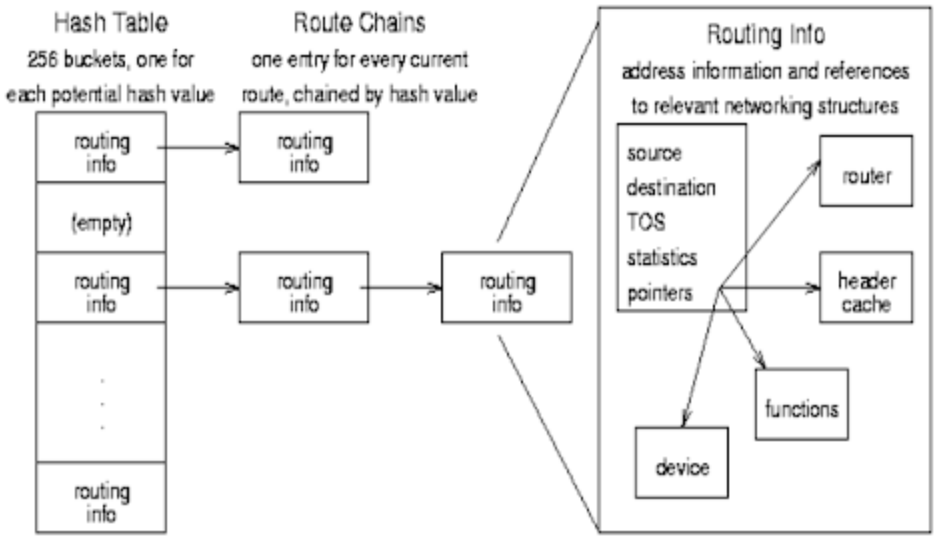
Routing table processing

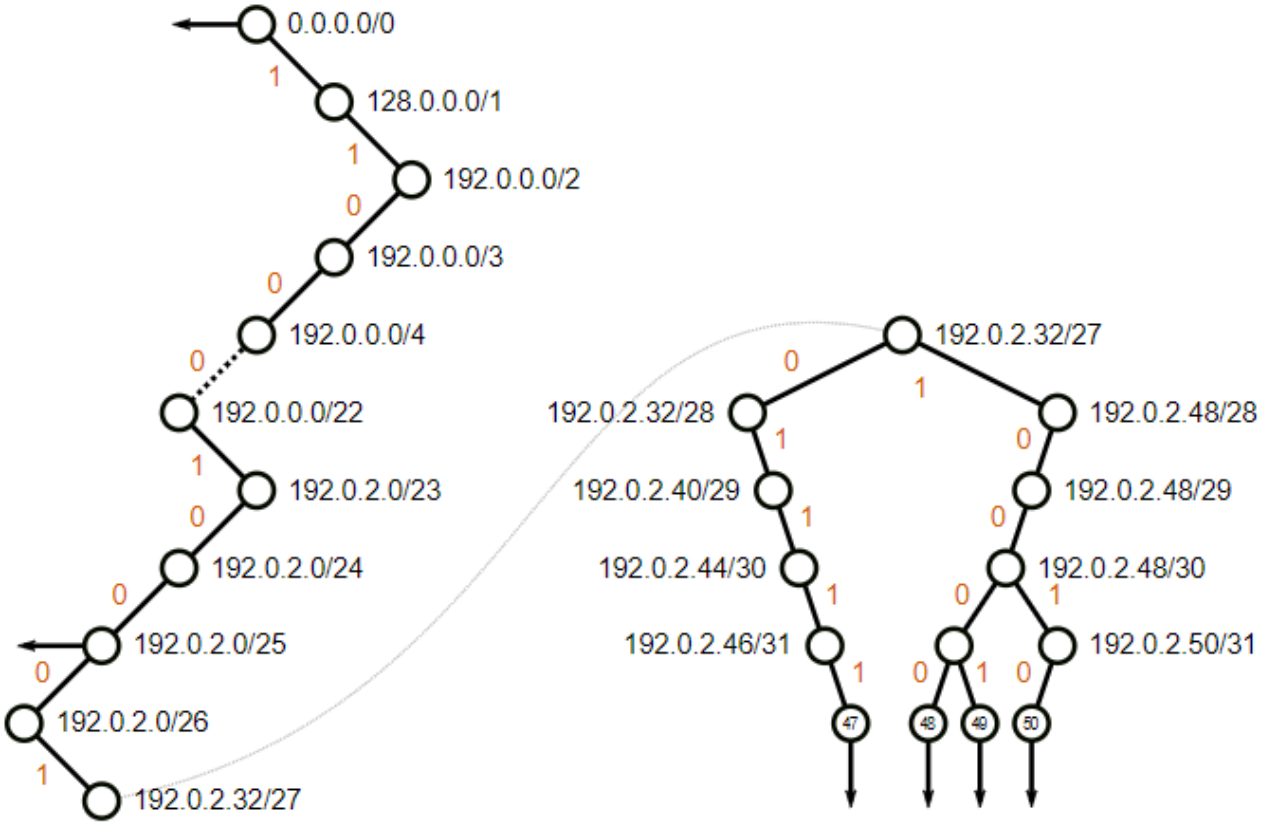
- Special table for local addreses -> route packets to sockets based on family, type, ports
- Check every routing entry for starting with the most specific routes (e.g. 192.168.0.0/24 is checked before 192.168.0.0/16)
- A route matches if the packet destination addreess logical ORed with the subnet mask equals the subnet address
- Once a route matches the following information is retrieved: interface, link layer next-hop address, network next host address

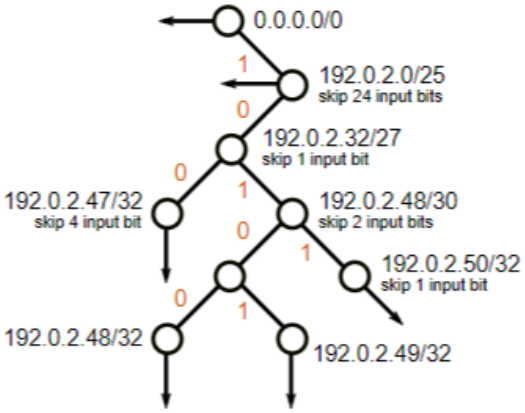
Forwarding Information Database

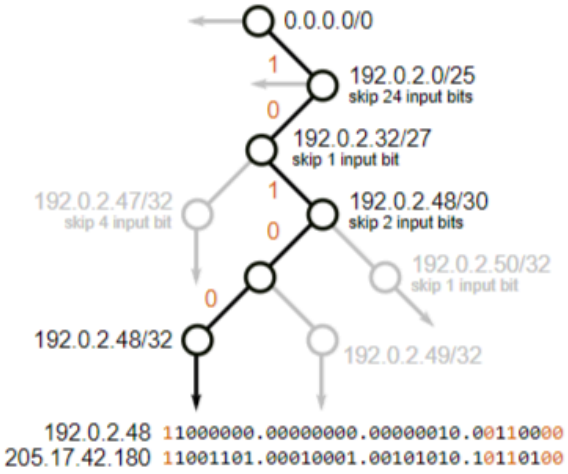

Netmask Table
one entry for each
potential netmask
Network Zones
one zone for each
known subnet mask
Network Information
routing instructions for
each known network
0000
8000
C000
E000
(empty)
F000
F800
FE00
(empty)
FF00
FF80
FFFF
(empty)
default
zone
zone
(F000)
zone
(FF00)
0.0.0.0
node
info
10.0.0.0
node
info
172.16.0.0
node
info
172.29.0.0
node
info
split into specific
network information
(addresses, TOS, etc.)
and protocols
(IP, ATM, etc.)


fib_zones
fib_zone
fz_next
fz_hash_table
fz_list
fz_nent
fz_logmask
fz_mask
fib_node
fib_next
fib_dst
fib_use
fib_info
fib_metric
fib_tos
fib_info
fib_next
fib_prev
fib_gateway
fib_dev
fib_refcnt
fib_window
fib_flags
fib_mtu
fib_irtt
fib_node
fib_next
fib_dst
fib_use
fib_info
fib_metric
fib_tos
fib_info
fib_next
fib_prev
fib_gateway
fib_dev
fib_refcnt
fib_window
fib_flags
fib_mtu
fib_irtt


Hash Table
256 buckets, one for each potential hash value
Route Chains
one entry for every current route, chained by hash value
routing info
routing info
(empty)
routing info
routing info
routing info
routing info
routing info
Routing Info
address information and references to relevant networking structures
source
destination
TOS
statistics
pointers
router
header cache
functions
device


0.0.0.0/0
128.0.0.0/1
192.0.0.0/2
192.0.0.0/3
192.0.0.0/4
192.0.0.0/22
192.0.2.0/23
192.0.2.0/24
192.0.2.0/25
192.0.2.0/26
192.0.2.32/27
192.0.2.32/28
192.0.2.40/29
192.0.2.44/30
192.0.2.46/31
192.0.2.48/28
192.0.2.48/29
192.0.2.48/30
192.0.2.50/31
47
48
49
50

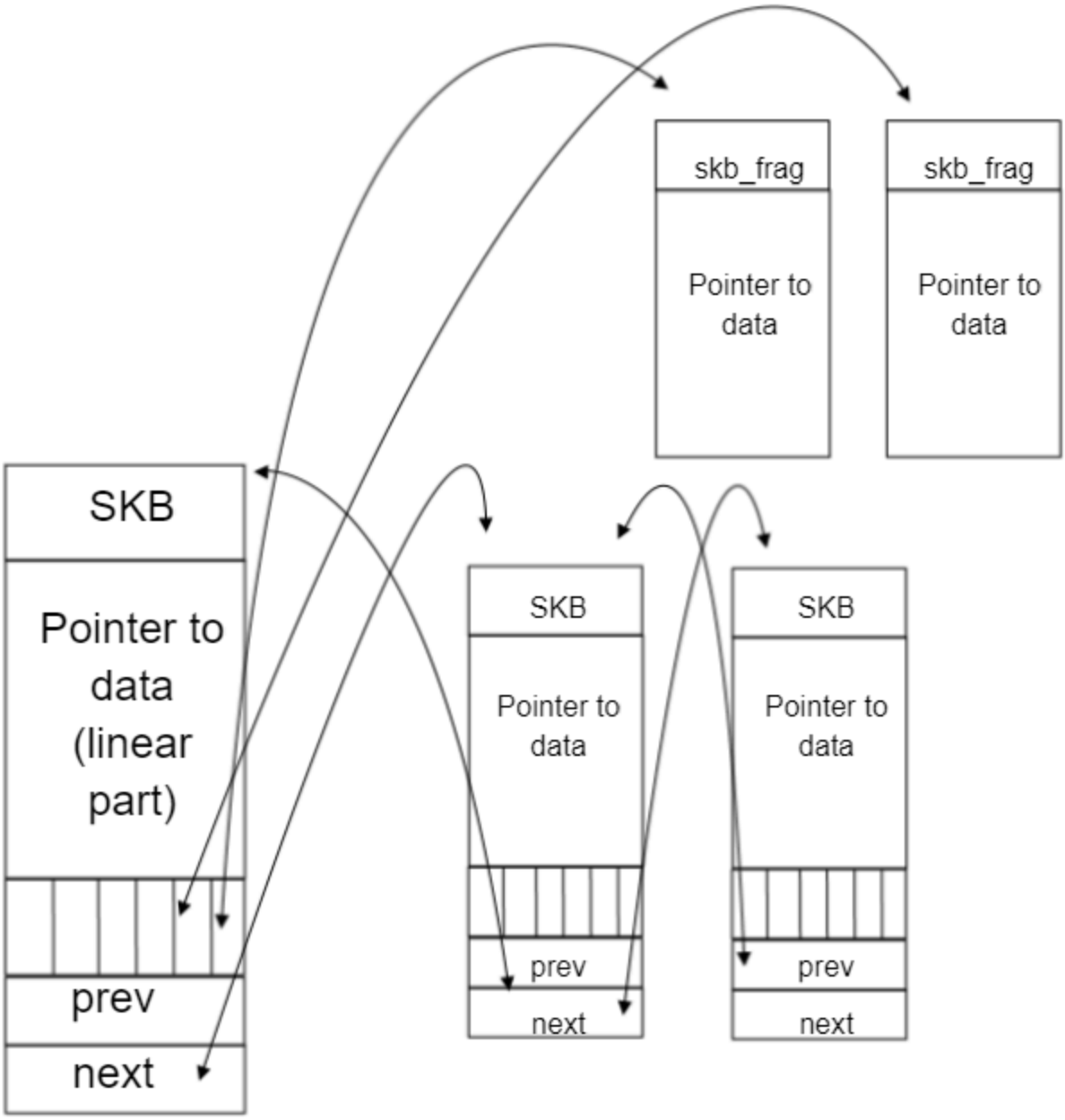




Netfilter

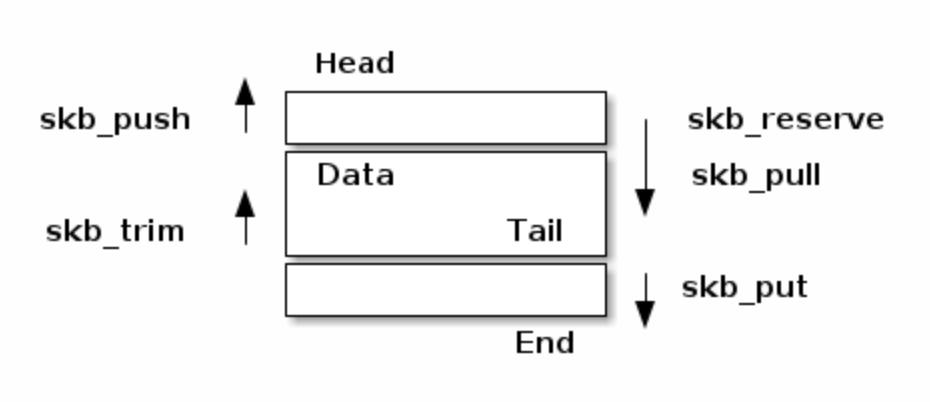
- Framework that implements packet filtering and NAT
- It uses hooks inserted in key places in the packet flow:
  - NF_IP_PRE_ROUTING
  - NF_IP_LOCAL_IN
  - NF_IP_FORWARD
  - NF_IP_LOCAL_OUT
  - NF_IP_POST_ROUTING
  - NF_IP_NUMHOOKS

Network packets / skbs (struct sk_buff)

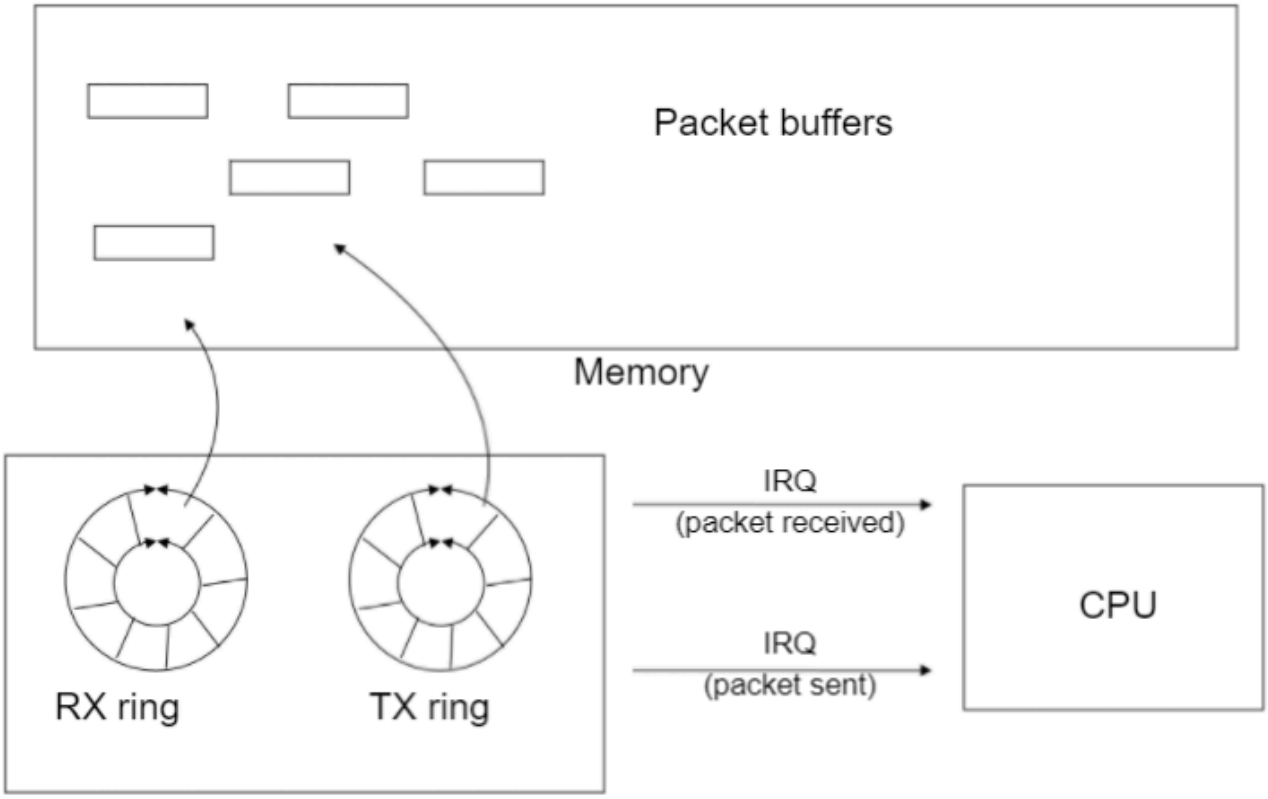


```
struct sk_buff {
    struct sk_buff *next;
    struct sk_buff *prev;

    struct sock *sk;
    ktime_t tstamp;
    struct net_device *dev;
    char cb[48];

    unsigned int len,
    data_len;
    __u16 mac_len,
    hdr_len;

    void (*destructor)(struct sk_buff *skb);

    sk_buff_data_t transport_header;
    sk_buff_data_t network_header;
    sk_buff_data_t mac_header;
    sk_buff_data_t tail;
    sk_buff_data_t end;

    unsigned char *head,
    *data;
    unsigned int truesize;
    atomic_t users;
```

```
/* reserve head room */
void skb_reserve(struct sk_buff *skb, int len);

/* add data to the end */
unsigned char *skb_put(struct sk_buff *skb, unsigned int len);

/* add data to the top */
unsigned char *skb_push(struct sk_buff *skb, unsigned int len);

/* discard data at the top */
unsigned char *skb_pull(struct sk_buff *skb, unsigned int len);

/* discard data at the end */
unsigned char *skb_trim(struct sk_buff *skb, unsigned int len);

unsigned char *skb_transport_header(const struct sk_buff *skb);

void skb_reset_transport_header(struct sk_buff *skb);

void skb_set_transport_header(struct sk_buff *skb, const int offset);

unsigned char *skb_network_header(const struct sk_buff *skb);

void skb_reset_network_header(struct sk_buff *skb);

void skb_set_network_header(struct sk_buff *skb, const int offset);

unsigned char *skb_mac_header(const struct sk_buff *skb);

int skb_mac_header_was_set(const struct sk_buff *skb);

void skb_reset_mac_header(struct sk_buff *skb);

void skb_set_mac_header(struct sk_buff *skb, const int offset);
```



Network Device



- Scatter-Gather
- Checksum offloading: Ethernet, IP, UDP, TCP
- Adaptive interrupt handling (coalescence, adaptive)

Hardware and Software Acceleration Techniques

- Full offload - Implement TCP/IP stack in hardware
- Issues:
  - Scaling number of connections
  - Security
  - Conformance

- Performance is proportional with the number of packets to be processed
- Example: if an end-point can process 60K pps
  - 1538 MSS -> 738Mbps
  - 2038 MSS -> 978Mbps
  - 9038 MSS -> 4.3Gbps
  - 20738 MSS -> 9.9Gbps

- The networking stack processes large packets
- TX path: the hardware splits large packets in smaller packets (TCP Segmentation Offload)

- RX path: the hardware aggregates small packets into larger packets (Large Receive Offload - LRO)