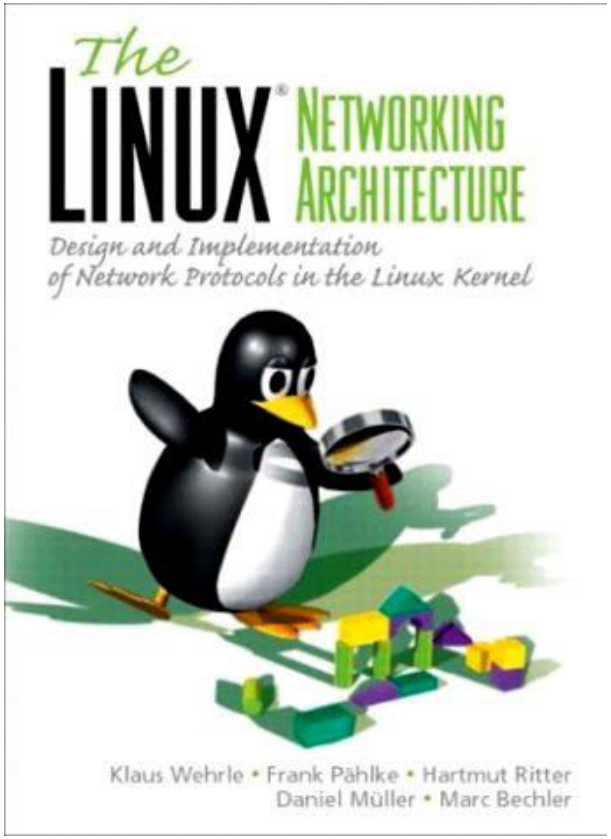
Linux Networking Architecture

Hugo

9/11/2014

Outline

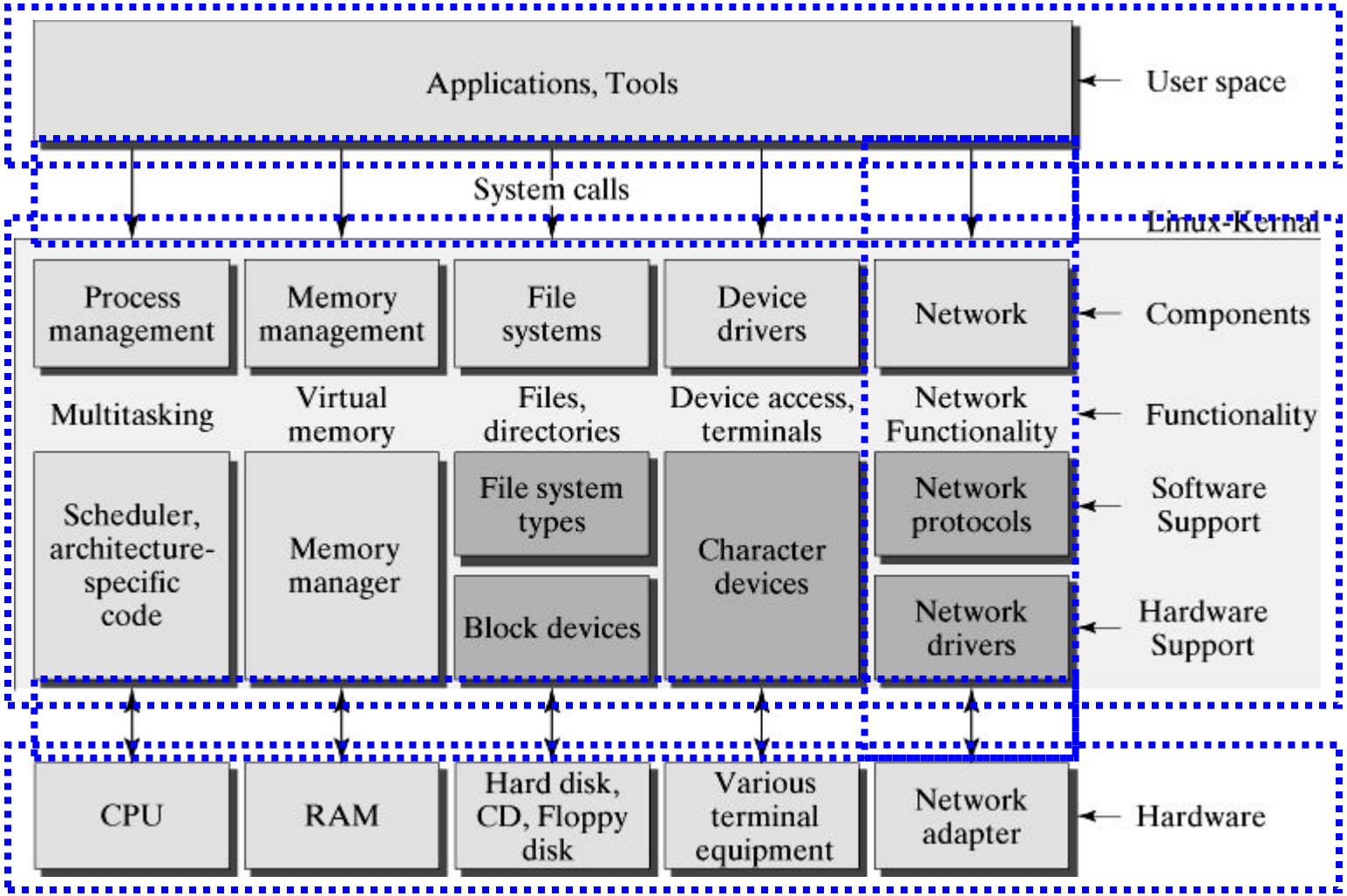
- Architecture of Communication System
- Managing Network Packets
- Network Device
- Data-Link Layer
- Network Layer
- Transport Layer
- Sockets in Linux Kernel
- Socket Programming



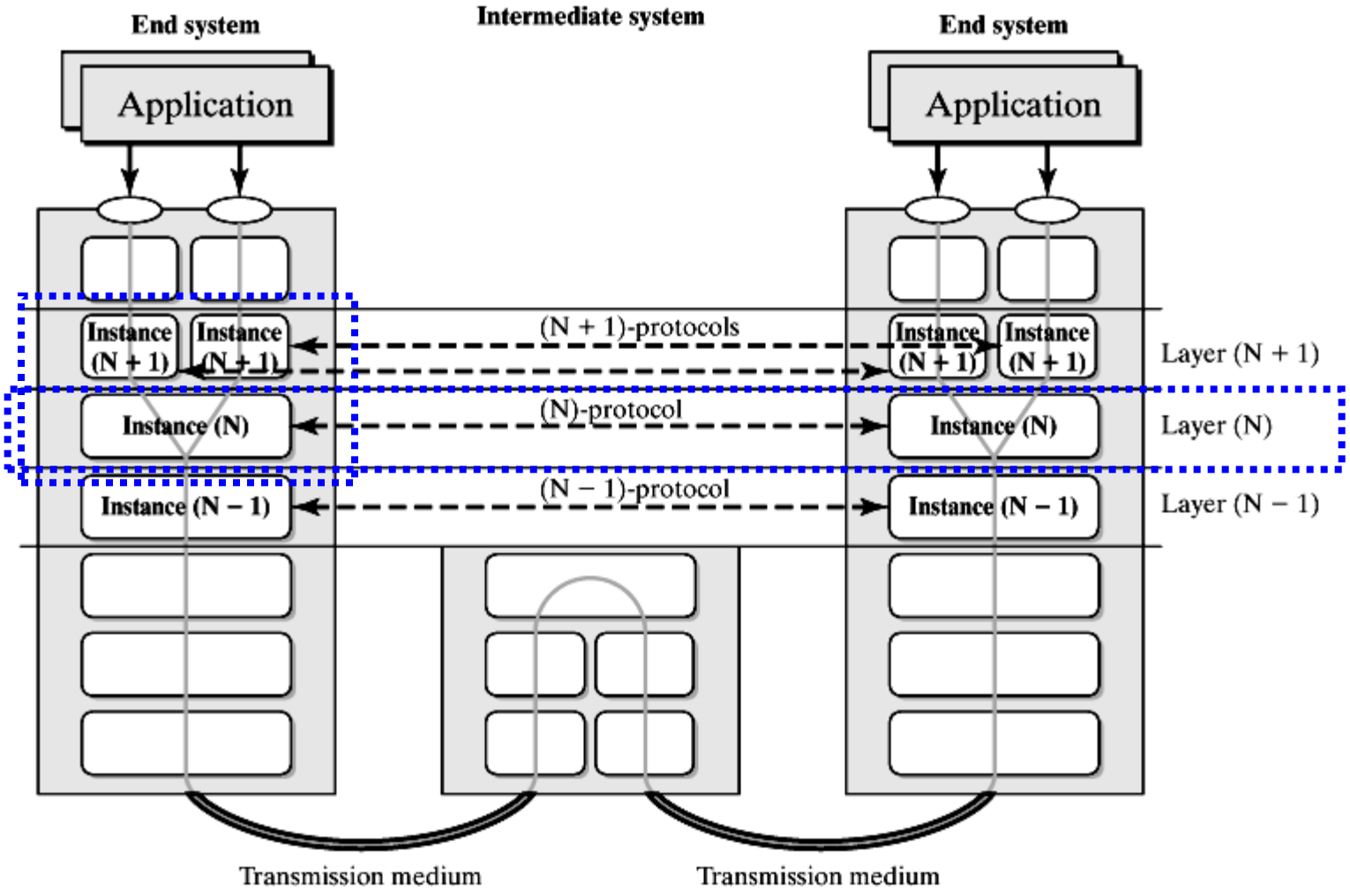
Outline

- Architecture of Communication System
- Managing Network Packets
- Network Device
- Data-Link Layer
- Network Layer
- Transport Layer
- Sockets in Linux Kernel
- Socket Programming

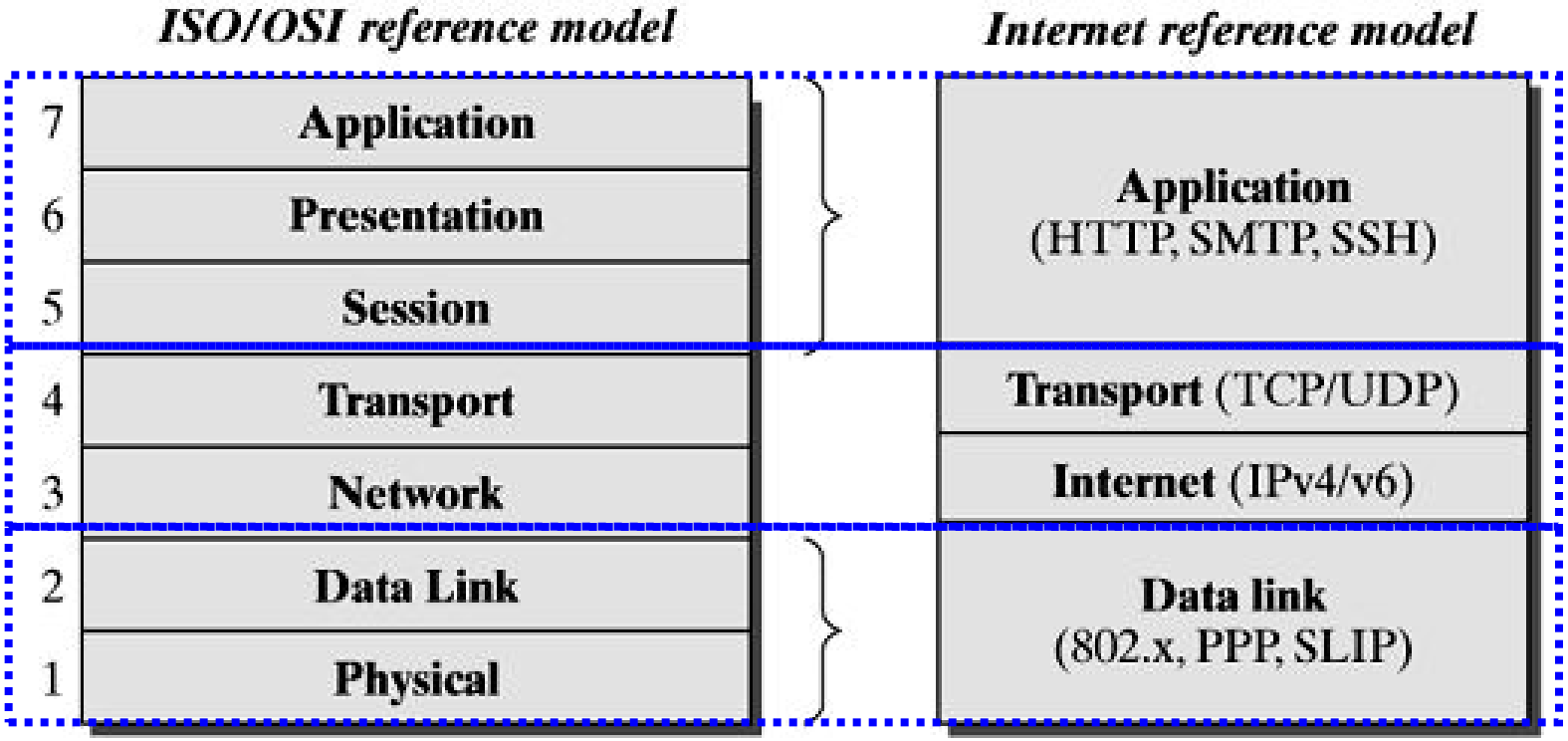
Linux Kernel Structure



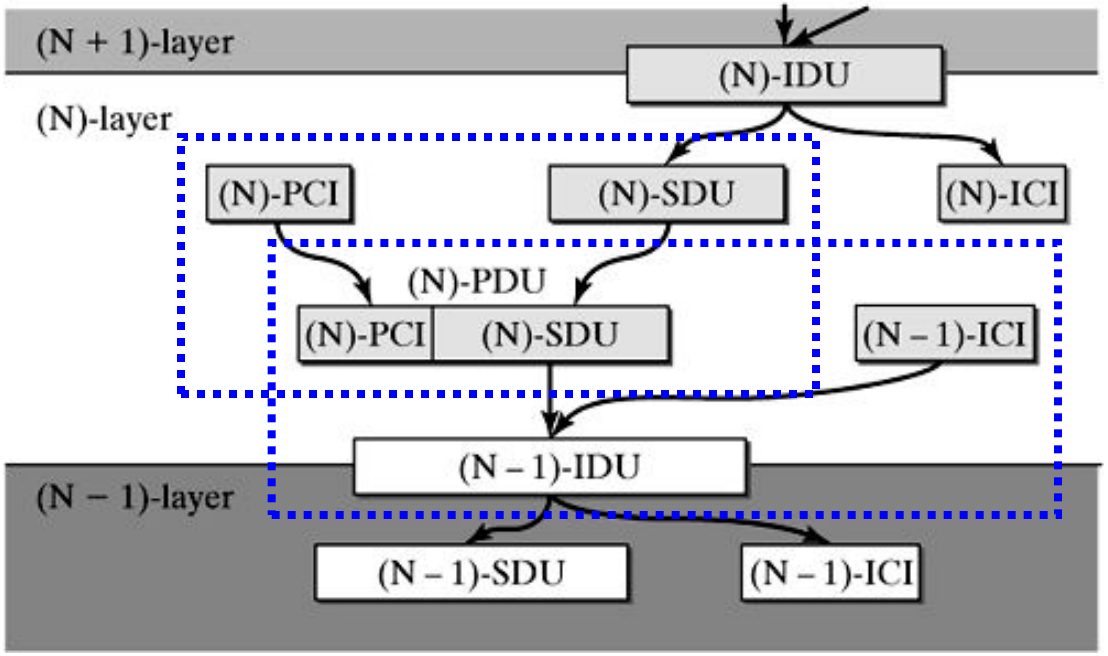
Layer-Based Communication



TCP/IP Reference Model



Vertical & Horizontal Comm.

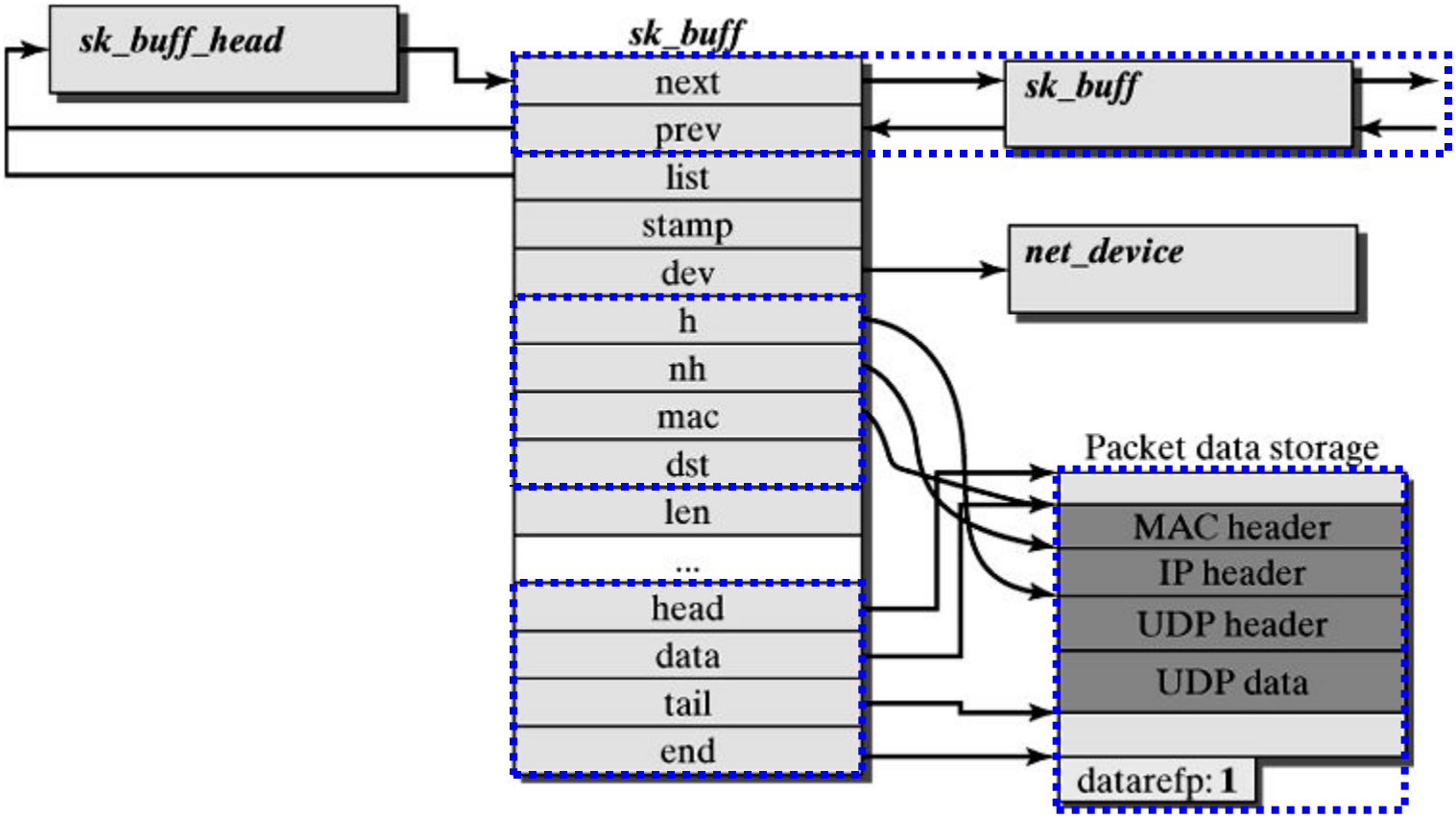


PDU	Protocol Data Unit
PCI	Protocol Control Information
SDU	Service Data Unit
ICI	Interface Control Information
IDU	Interface Data Unit

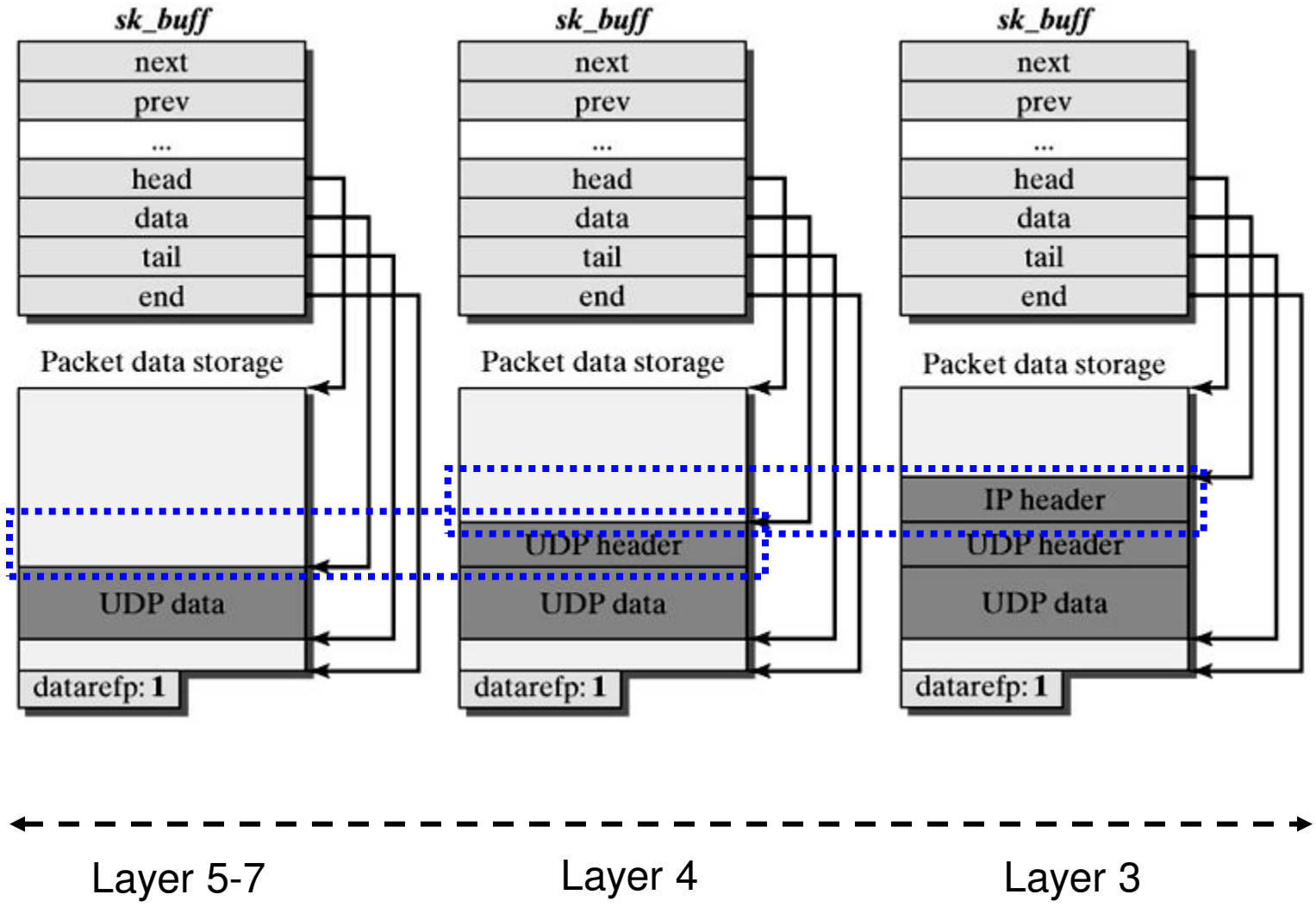
Outline

- Architecture of Communication System
- Managing Network Packets
- Network Device
- Data-Link Layer
- Network Layer
- Transport Layer
- Sockets in Linux Kernel
- Socket Programming

Socket Buffer



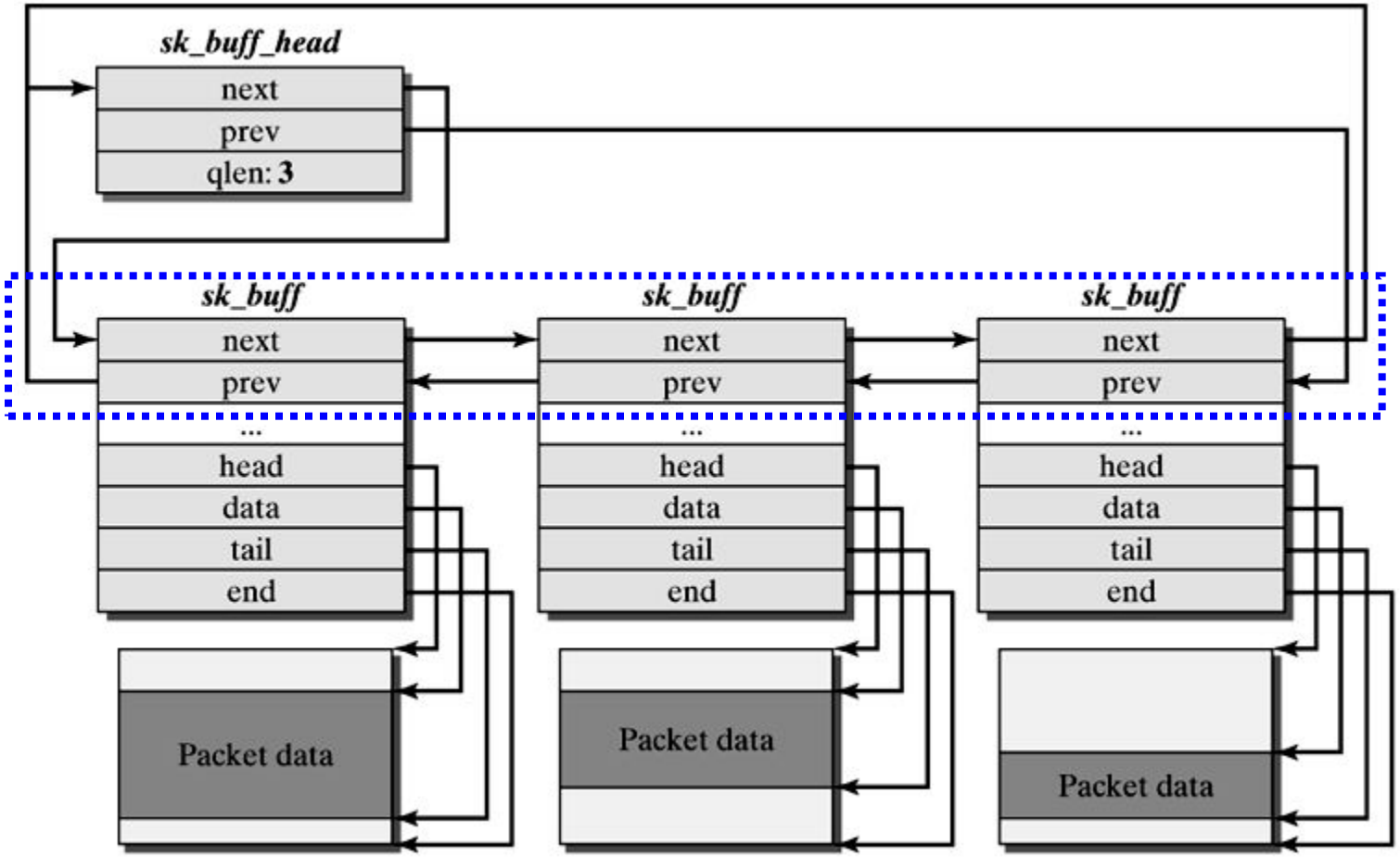
- socket buffers are data structures used to represent and manage packets


sk_buff
next
prev
...
head
data
tail
end
Packet data storage
UDP data
datarefp: 1
sk_buff
next
prev
...
head
data
tail
end
Packet data storage
UDP header
UDP data
datarefp: 1
sk_buff
next
prev
...
head
data
tail
end
Packet data storage
IP header
UDP header
UDP data
datarefp: 1
Layer 5-7
Layer 4
Layer 3

Operations on Socket Buffers

- Create, Release, Duplicate Socket Buffers
  - `alloc_skb()`, `skb_copy()`, `skb_copy_expand()`, `skb_clone()`, `kfree_skb()`, `skb_header_init()`
- Manipulate Packet Data Space
  - `skb_get()`, `skb_put()`, `skb_push()`, `skb_pull()`, `skb_tailroom()`, `skb_headroom()`, `skb_realloc_headroom()`, `skb_reserve()`, `skb_trim()`, `skb_cow()`
- Manage Socket Buffer Queues
  - `skb_cloned()`, `skb_over_panic()`, `skb_under_panic()`, `skb_head_to_pool()`, `skb_head_from_pool()`

Socket-Buffer Queue



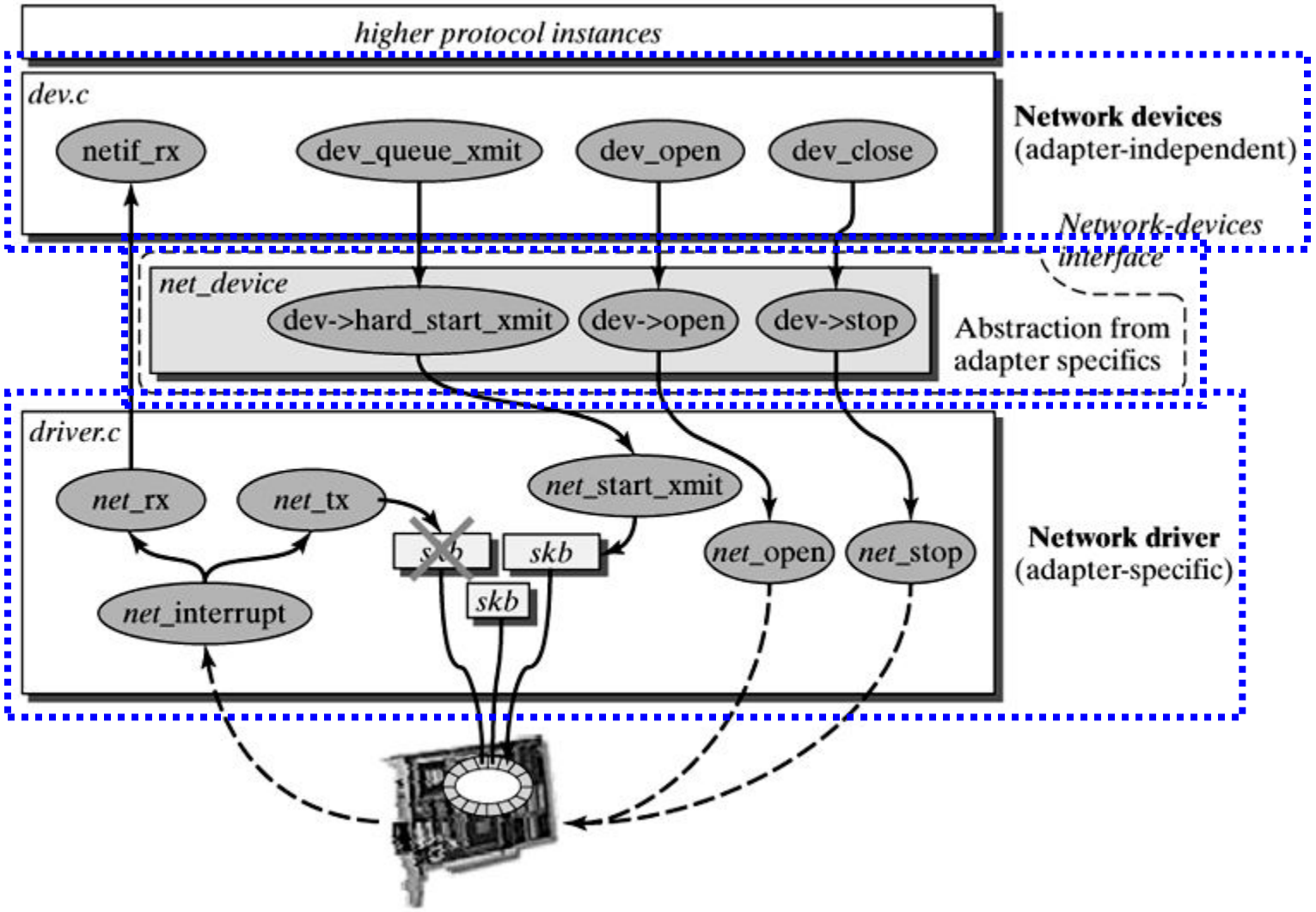
Operations on Socket-Buffer Queues

- Manage Queue Structures
  - `skb_queue_head_init(), skb_queue_empty(), skb_queue_len()`
- Manage Socket Buffers in Queues
  - `skb_queue_head(), skb_queue_tail(), skb_dequeue(), skb_dequeue_tail(), skb_queue_purge(), skb_insert(), skb_append(), skb_unlink(), skb_peek(), skb_peek_tail()`

Outline

- Architecture of Communication System
- Managing Network Packets
- Network Device
- Data-Link Layer
- Network Layer
- Transport Layer
- Sockets in Linux Kernel
- Socket Programming

Network Device Interface



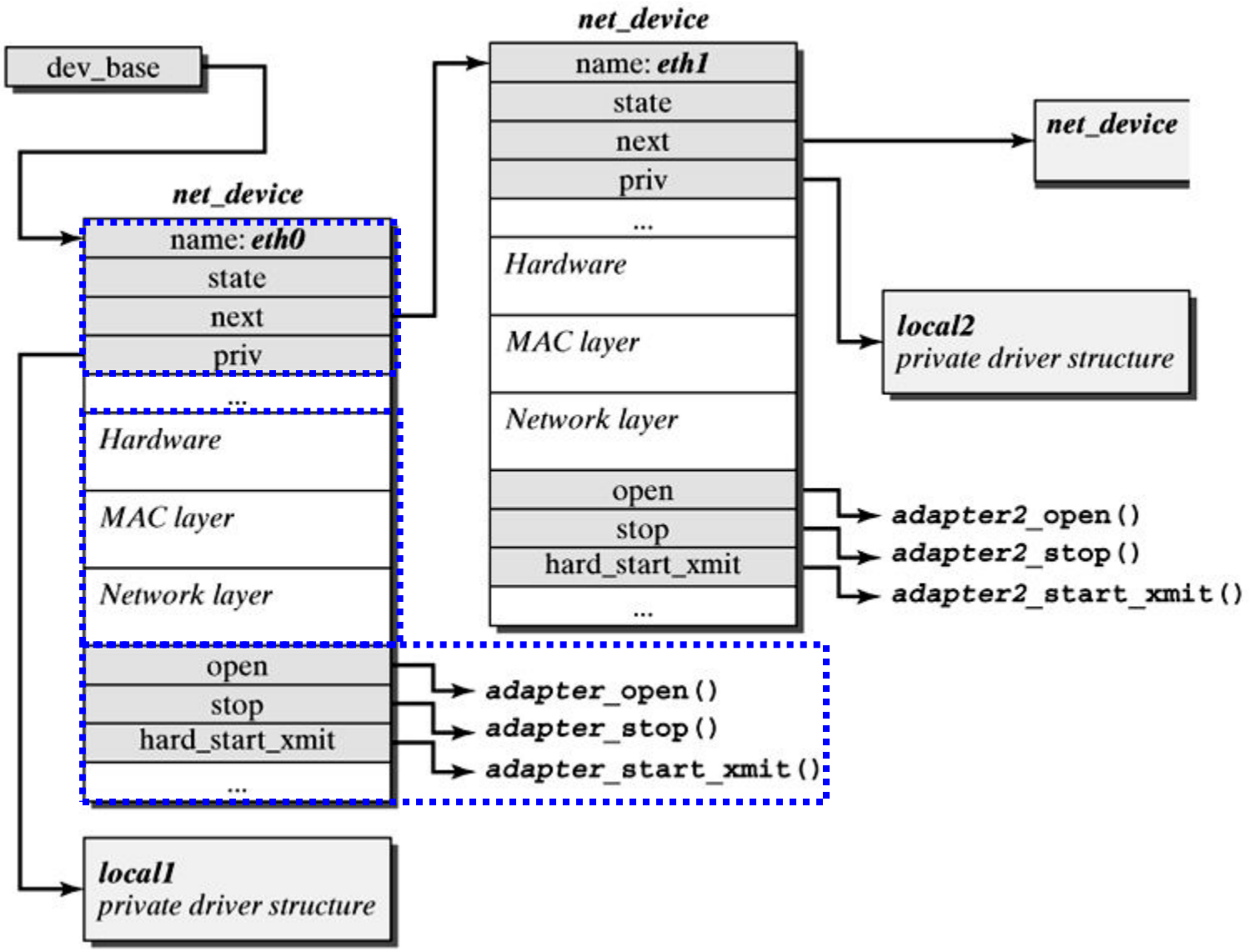
The net_device Structure

- General Fields of a Network Device
  - `name, next, owner, ifindex, iflink, state, trans_start, last_rx, priv, qdisc, refcnt, xmit_lock, xmit_lock_owner, queue_lock`
- Hardware-Specific Fields
  - `rmem_end, rmem_start, mem_end, mem_start , base_addr, irq, dma, if_port`
- Data on the Physical Layer
  - `hard_header_length, mtu, tx_queue_len, type, addr_len, dev_addr, broadcast, dev_mc_list, mc_count, watchdog_timeo, watchdog_timer`
- Data on the Network Layer
  - `ip_ptr, ip6_ptr, atalk_ptr, dn_ptr, ec_ptr, family, pa_alen, pa_addr, pa_braddr, pa_mask, pa_dstaddr, flags`
- Device-Driver Methods
  - `init(), uninit(), destructor(), open(), stop(), hard_start_xmit(), get_stats(), get_wireless_stats(), set_multicast_list(), watchdog_timeo(), do_ioctl(), set_config(), hard_header(), rebuild_header(), hard_header_cache(), header_cache_update(), hard_header_parse(), set_mac_address(), change_mtu()`

Managing Network Devices

- Registering and Unregistering Network Devices
  - `init_netdev()`, `init_etherdev()`, `ether_setup()`, `register_netdevice()`, `unregister_netdevice()`
- Opening and Closing Network Devices
  - `dev_open()`, `dev_close()`
- Creating and Finding Network Devices
  - `dev_alloc_name()`, `dev_alloc()`, `dev_get_by_name()`, `dev_get_by_index()`, `dev_load()`
- Notification Chains for State Changes
  - `notifier_call()`, `notifier_call_chain()`, `register_netdevice_notifier()`, `unregister_netdevice_notifier()`
- Transmitting over Network Devices
  - `dev_queue_xmit()`

Linked List of net_device



Managing Network Drivers

- Initializing Network Adapters
- Opening and Closing a Network Adapter
- Transmitting Data
- Problems In Transmitting Packets
- Runtime Configuration
- Adapter-Specific ioctl() Commands
- Statistical Information About a Network Device
- Multicast Support on Adapter Level

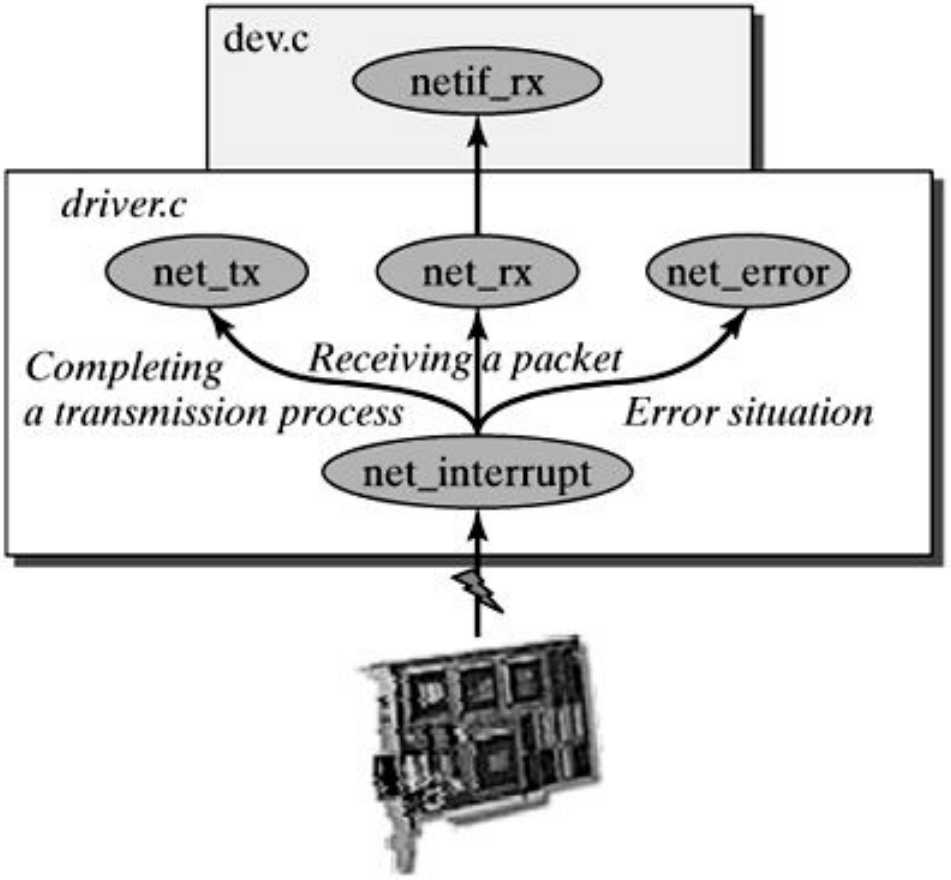
```
int __init netcard_probe(struct net_device *dev) {
    int i;
    for (i = 0; netcard_portlist[i]; i++) {
        int ioaddr = netcard_portlist[i];
        if (check_region(ioaddr, IO_NUM))
            continue;

        if (netcard_probe1(dev, ioaddr) == 0)
            return 0;
    }
    return -ENODEV;
}
```

```
static int __init netcard_probe1(struct net_device *dev, int ioaddr) {
    if (inb(ioaddr + 0) != SA_ADDRO
        || inb(ioaddr + 1) != SA_ADDR1 || inb(ioaddr + 2) != SA_ADDR2) {
        return -ENODEV;
    }
    /* Fill in the 'dev' fields. */
    dev->base_addr = ioaddr;
    request_region(ioaddr, IO_NUM, cardname);

    dev->open = net_open;
    dev->stop = net_close;
    dev->hard_start_xmit = net_send_packet;
    dev->get_stats = net_get_stats;
    dev->set_multicast_list = &set_multicast_list;
    dev->tx_timeout = &net_tx_timeout;
    dev->watchdog_timeo = MY_TX_TIMEOUT;

    ether_setup(dev);
    return 0;
}
```

Transmitting Data Packets

```
static int net_send_packet(struct sk_buff *skb, struct
   net_device *dev) {
     struct net_local *np = (struct net_local *)dev->priv;
     int ioaddr = dev->base_addr;
     short length = ETH_ZLEN < skb->len ? skb->len :
   ETH_ZLEN;
     unsigned char *buf = skb->data;

     hardware_send_packet(ioaddr, buf, length);
     np->stats.tx_bytes += skb->len;
     dev->trans_start = jiffies;
     if (inw(ioaddr) == /*RU*/81)
         np->stats.tx_aborted_errors++;
     dev_kfree_skb (skb);
```

Interrupts from Network Adapter



Receiving Packets from Adapter

```
static void net_interrupt(int irq, void *dev_id, struct
   pt_regs * regs) {
    ioaddr = dev->base_addr;
    np = (struct net_local *)dev->priv;
    status = inw(ioaddr + 0);
    if (status & RX_INTR) {
        net_rx(dev);
    }
    if (status & TX_INTR) {
        net_tx(dev);
        np->stats.tx_packets++;
        netif_wake_queue(dev);
    }
```

Receiving a Data Packet

```
static void net_rx(struct net_device *dev) {
    do {
        if (pkt_len == 0)
            break;
        if (status & 0x40) {
            /* There was an error. */
        }else{
            skb = dev_alloc_skb(pkt_len);
            ptr = skb_put(skb,pkt_len);
            memcpy(ptr, (void*)dev->rmem_start, pkt_len);
            netif_rx(skb);
        }
    } while (-boguscount);
```

Acknowledging a Transmission

```
void net_tx(struct net_device *dev) {
    spin_lock(&np->lock);

    while (tx_entry_is_sent(np, entry)) {
        struct sk_buff *skb = np->skbs[entry];
        np->stats.tx_bytes += skb->len;
        dev_kfree_skb_irq (skb);
        entry = next_tx_entry(np, entry);
    }

    if (netif_queue_stopped(dev) && ! tx_full(dev))
        netif_wake_queue(dev);

    spin_unlock(&np->lock);
```

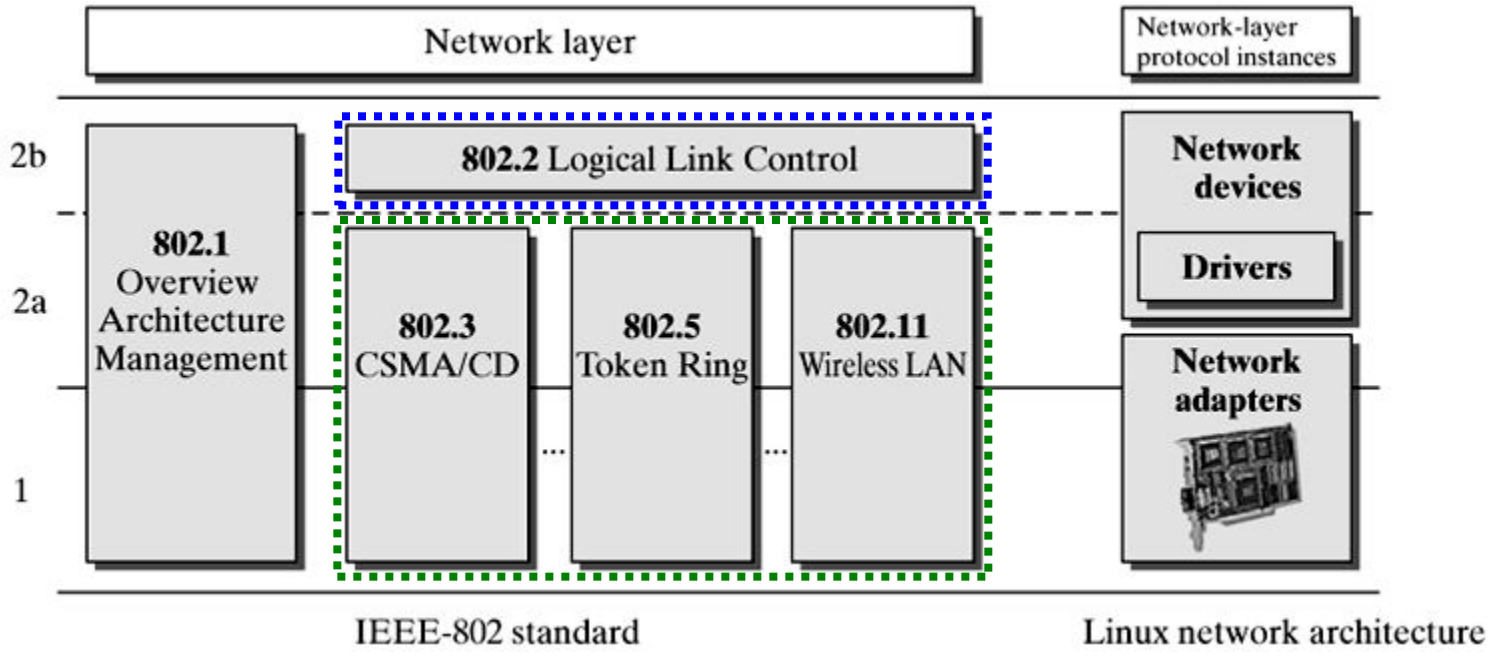
Outline

- Architecture of Communication System
- Managing Network Packets
- Network Device
- Data-Link Layer
- Network Layer
- Transport Layer
- Sockets in Linux Kernel
- Socket Programming

Media Access & Data-Link Layer

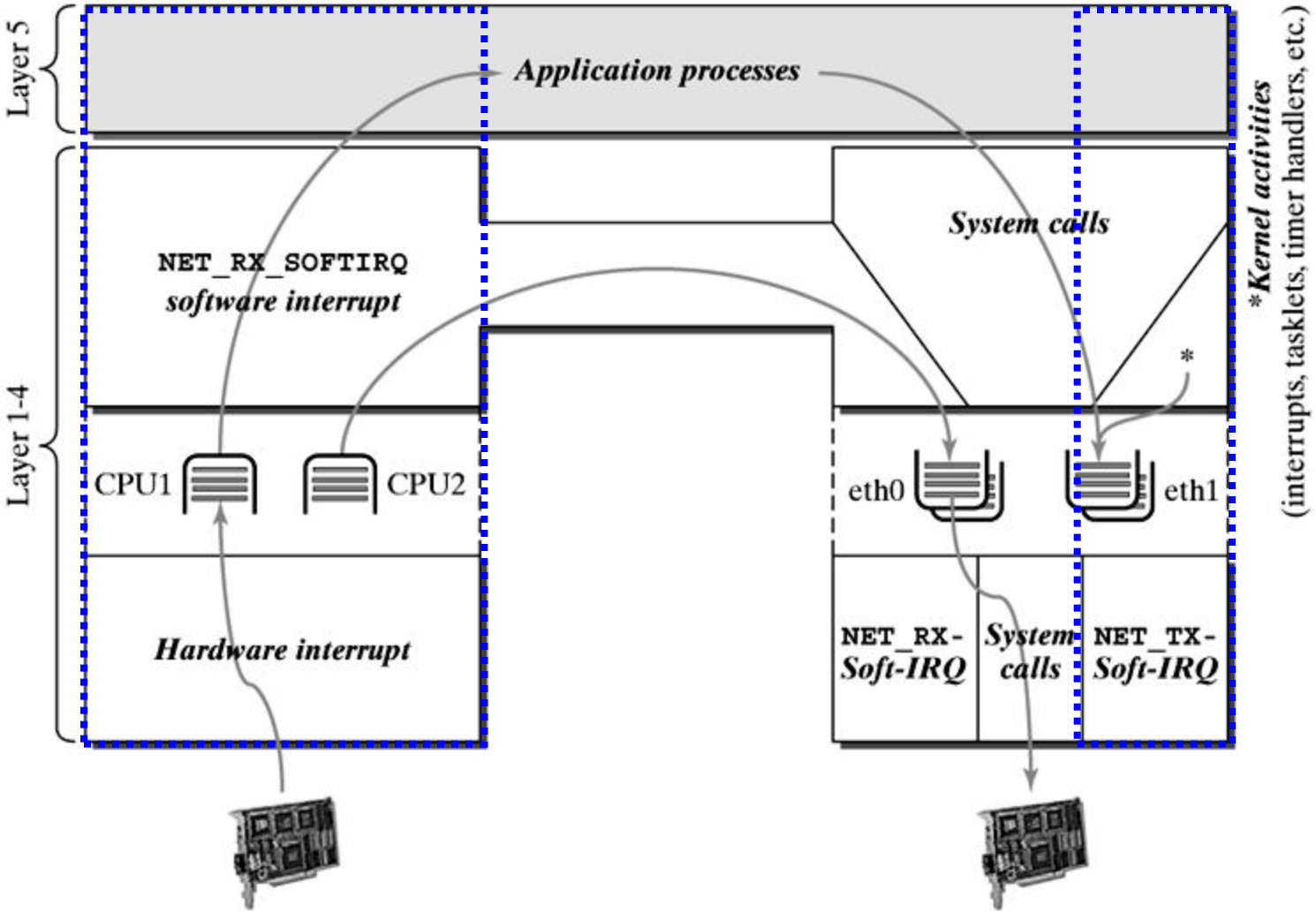
- Data-Link Layer (Ethernet)
- The Serial-Line Internet Protocol (SLIP)
- The Point-to-Point Protocol (PPP)
- PPP over Ethernet
- Asynchronous Transfer Mode—ATM
- Bluetooth in Linux
- Transparent Bridges

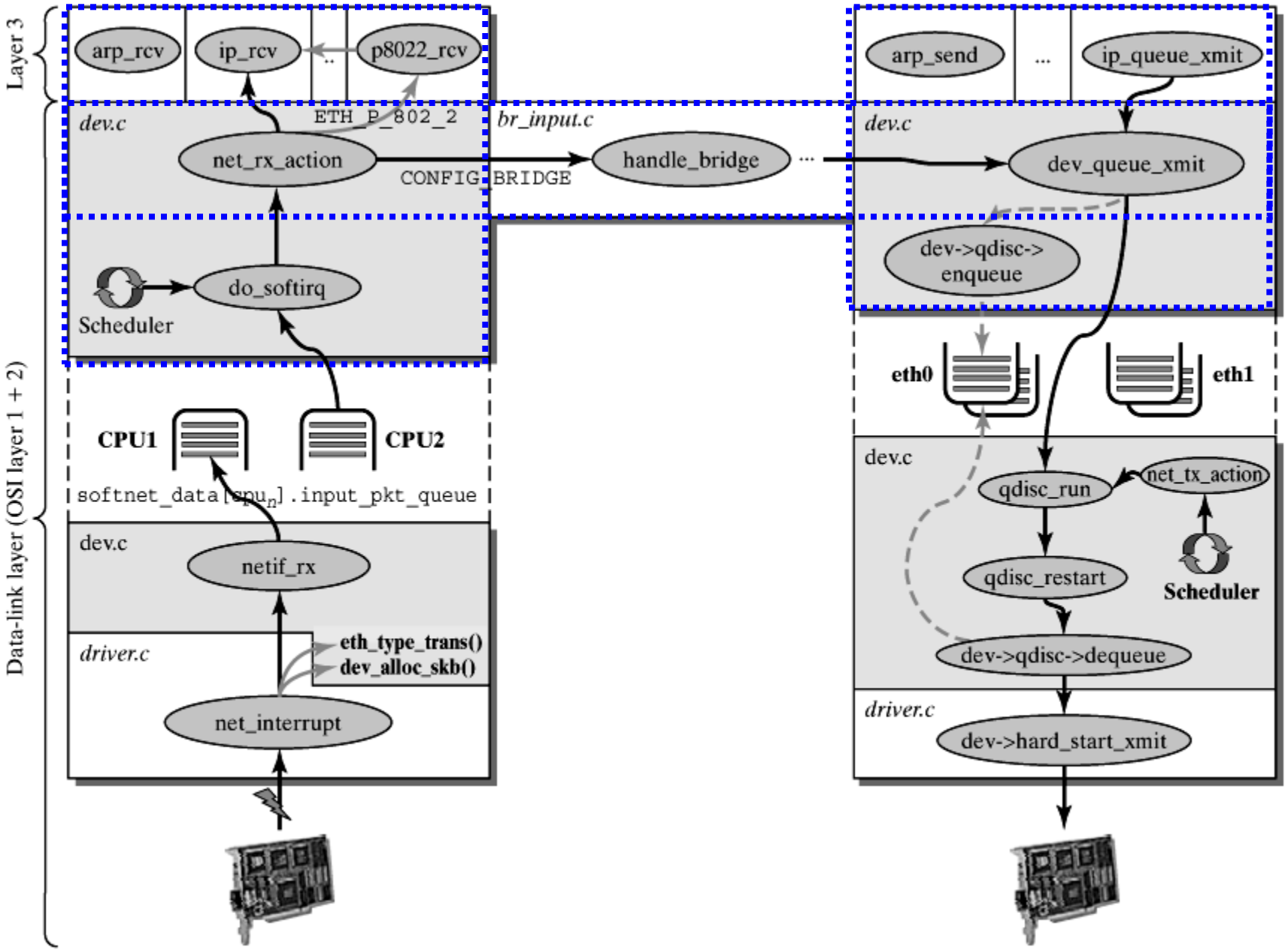
Implementation of Data Link Layer

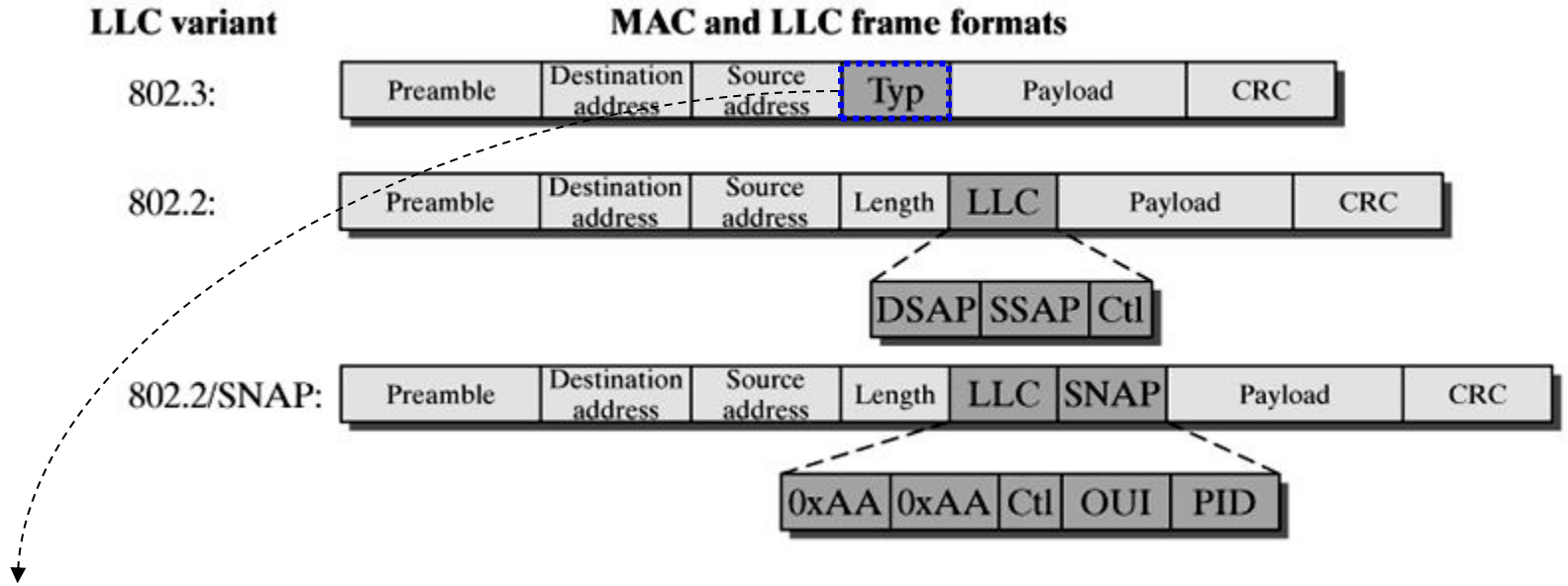


LLC (Logical-Link Control) provide a uniform interface for the upper layer.
MAC (Media-Access Control) handles the media-specific part.

Linux Network Activity



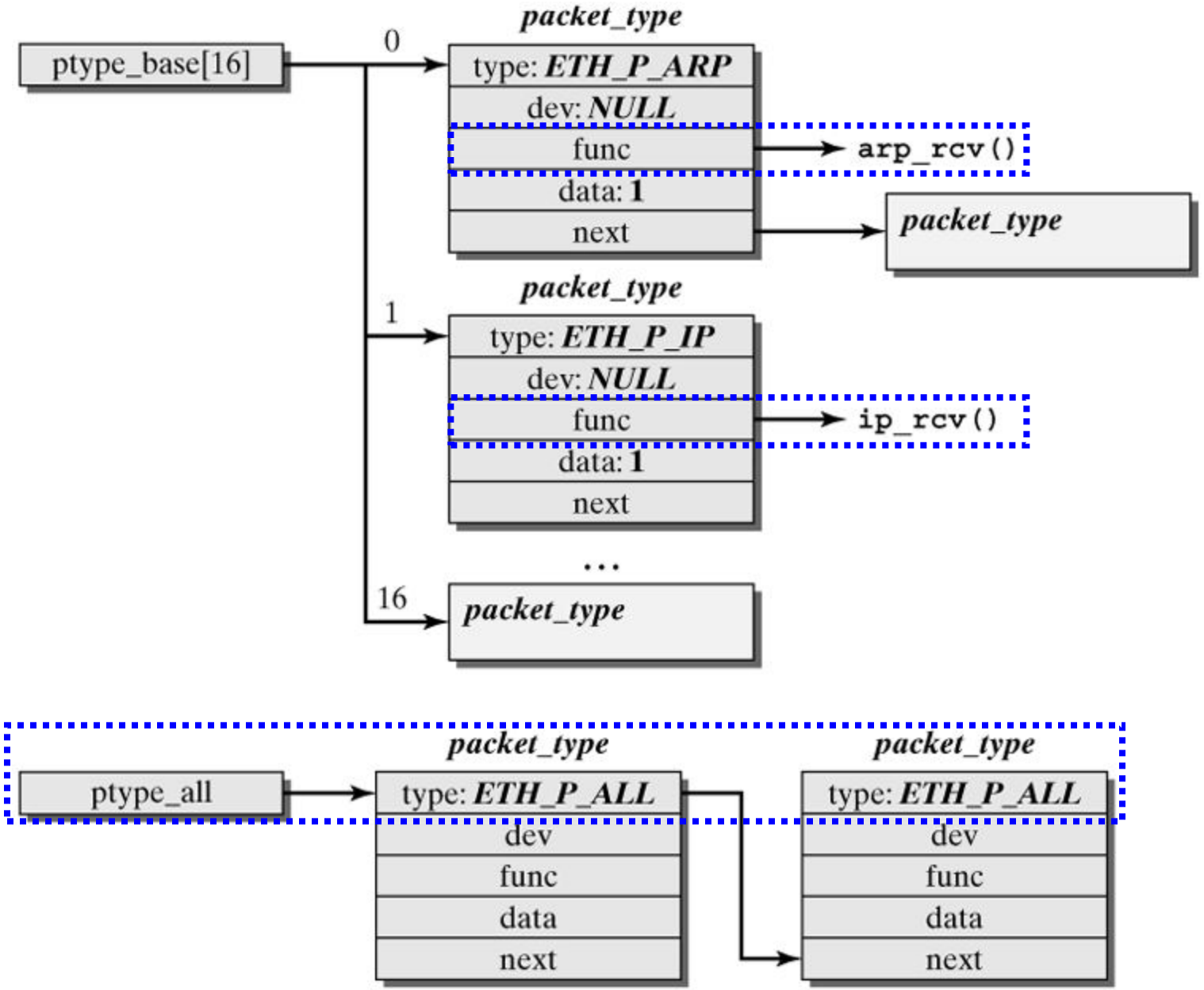

Layer 3
Data-link layer (OSI layer 1 + 2)
arp_rcv
ip_rcv
p8022_rcv
arp_send
ip_queue_xmit
dev.c
ETH_P_802_2
br_input.c
net_rx_action
CONFIG_BRIDGE
handle_bridge
dev_queue_xmit
dev->qdisc->
enqueue
do_softirq
Scheduler
eth0
eth1
CPU1
CPU2
softnet_data[cpu_n].input_pkt_queue
netif_rx
qdisc_run
net_tx_action
qdisc_restart
Scheduler
driver.c
eth_type_trans()
dev_alloc_skb()
net_interrupt
dev->qdisc->dequeue
dev->hard_start_xmit




```
/* These are the defined Ethernet Protocol ID's. */
ETH_P_LOOP      0x0060          /* Ethernet Loopback packet     */
ETH_P_IP        0x0800          /* Internet Protocol packet     */
ETH_P_X25       0x0805          /* CCITT X.25                   */
ETH_P_ARP       0x0806          /* Address Resolution packet    */
    ...
ETH_P_IPX       0x8137          /* IPX over DIX                 */
ETH_P_IPV6      0x86DD          /* IPv6                         */
```

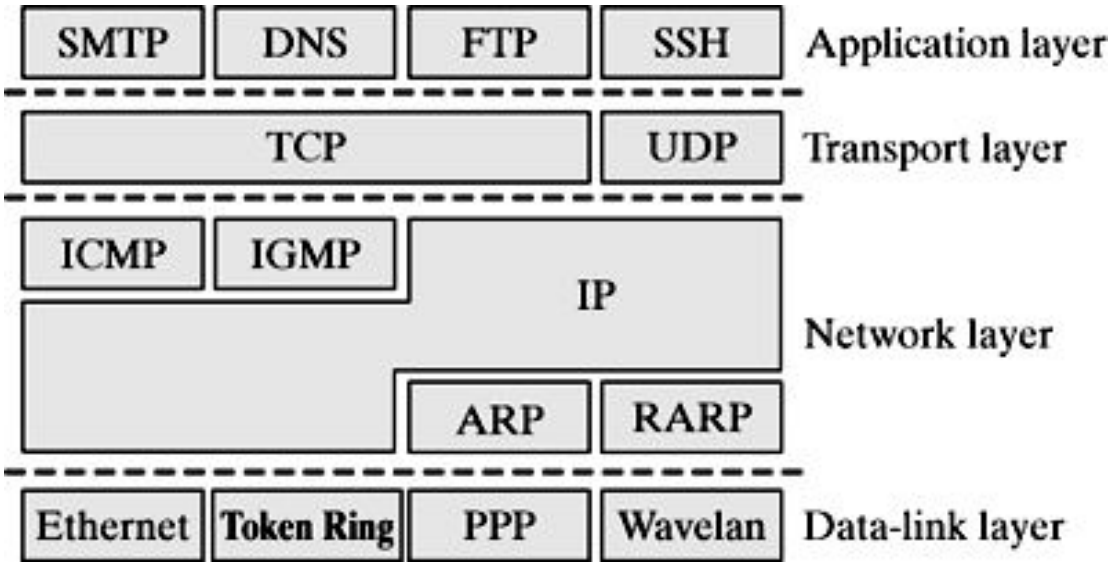
eth_type_trans(skb, dev)

- Recognize the LLC protocol type used and protocol ID of layer-3
- Identify the packet type (unicast, multicast, broadcast)
- Check whether the packet is addressed to the local computer


packet_type
ptype_base[16]
0
type: ETH_P_ARP
dev: NULL
func
arp_rcv()
data: 1
next
packet_type
packet_type
1
type: ETH_P_IP
dev: NULL
func
ip_rcv()
data: 1
next
...
16
packet_type
packet_type
packet_type
ptype_all
type: ETH_P_ALL
dev
func
data
next
type: ETH_P_ALL
dev
func
data
next

Internet Protocol Suite

SSH	Secure Socket Shell
FTP	File Transfer Protocol
DNS	Domain Name Service
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
IP	Internet Protocol
ICMP	Internet Control Message Protocol
IGMP	Internet Group Management Protocol
ARP	Address Resolution Protocol
RARP	Reverse Address Resolution Protocol



Outline

- Architecture of Communication System
- Managing Network Packets
- Network Device
- Data-Link Layer
- Network Layer
- Transport Layer
- Sockets in Linux Kernel
- Socket Programming

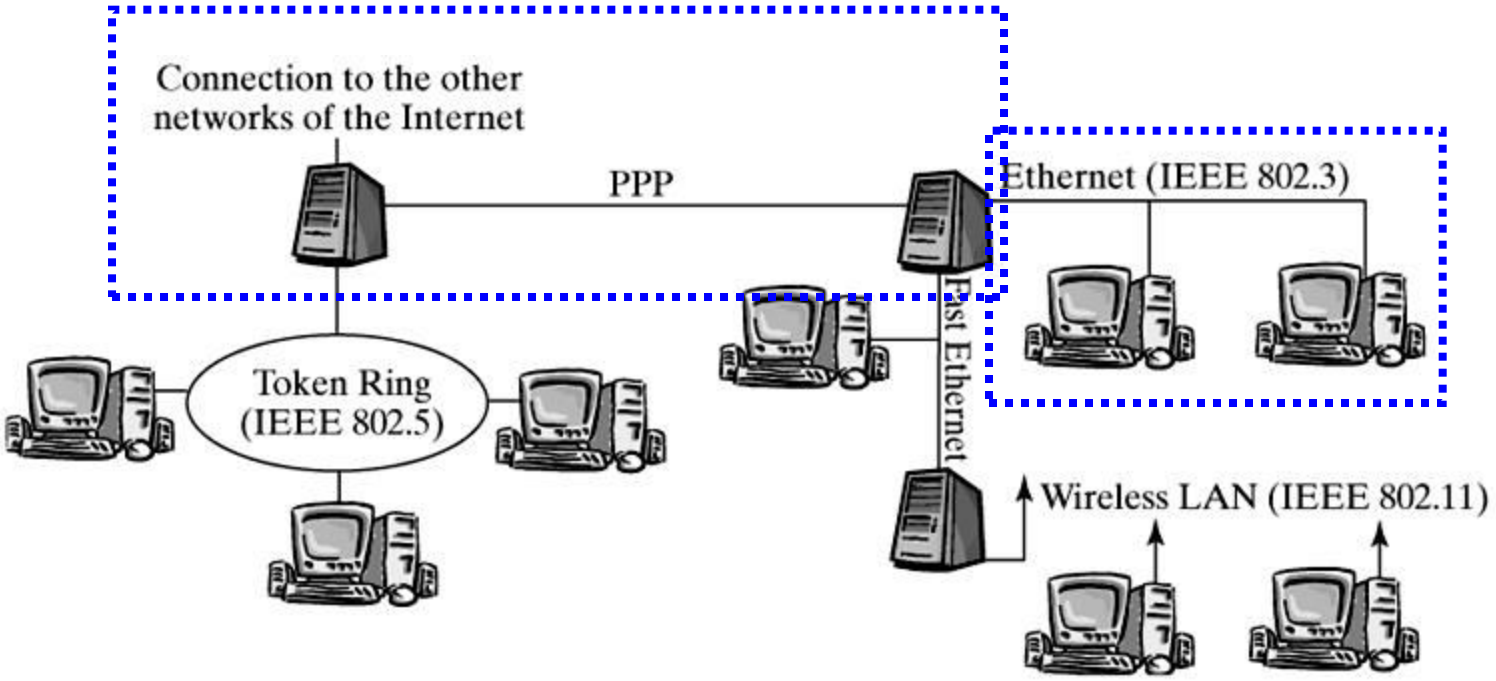
Network Layer

- IPv4 - Internet Protocol Version 4
- Internet Control Message Protocol (ICMP)
- Address Resolution Protocol (ARP)
- IP Routing
- IP Multicast for Group Communication
- Using Traffic Control to Support Quality of Service (QoS)
- Packet Filters and Firewalls
- Connection Tracking
- Network Address Translation (NAT)
- IPv6 - Internet Protocol Version 6

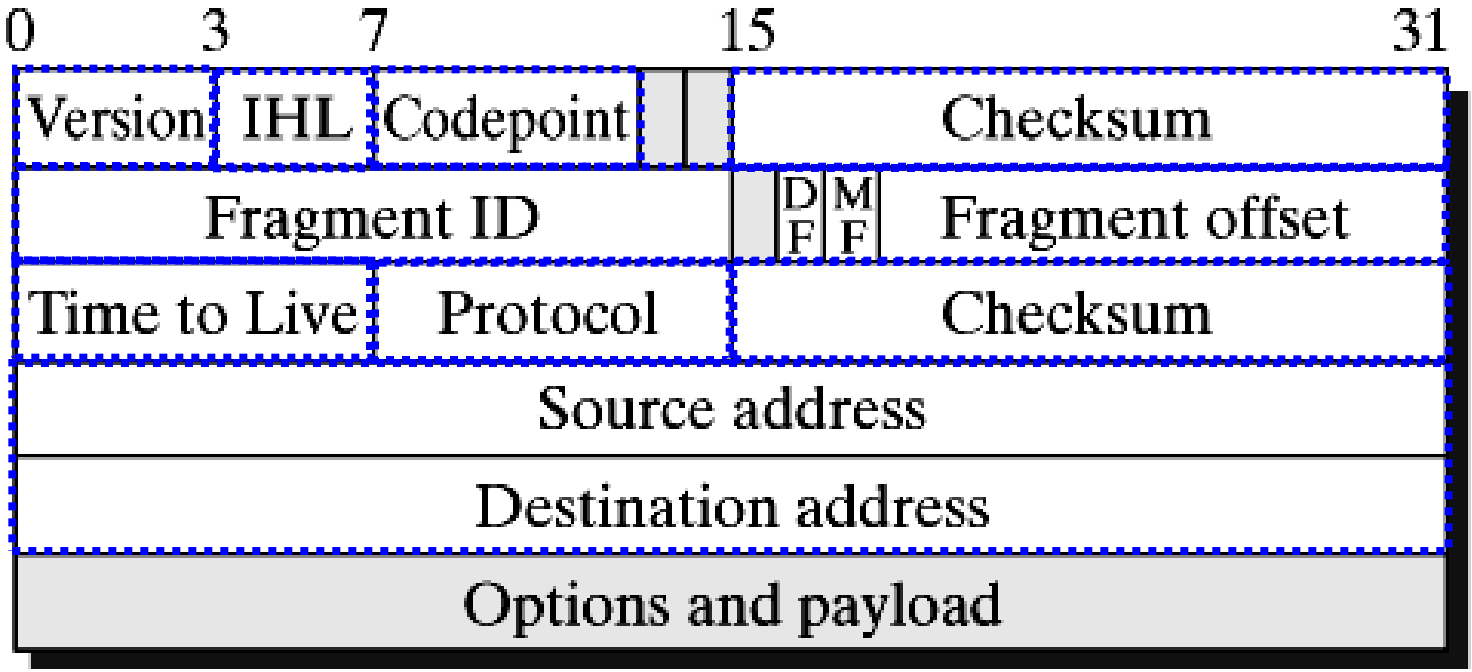
Internet Protocol

- Provides an unsecured connectionless datagram service
- Defines IP datagrams as basic units for data transmission
- Defines the IP addressing scheme
- Routes and forwards IP datagrams across interconnected networks
- Verifies the lifetime of packets
- Fragments and reassembles packets
- Uses ICMP to output errors

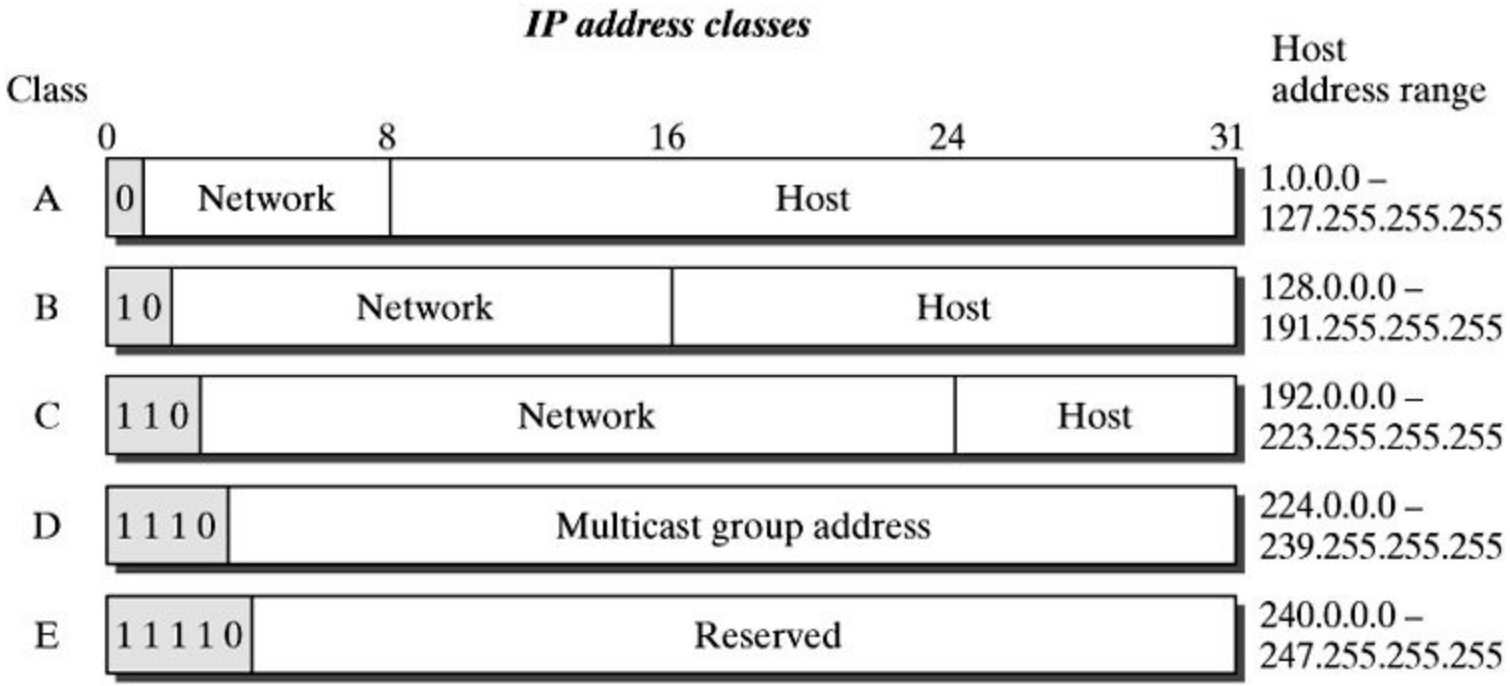
Routing IP Packets between LANs



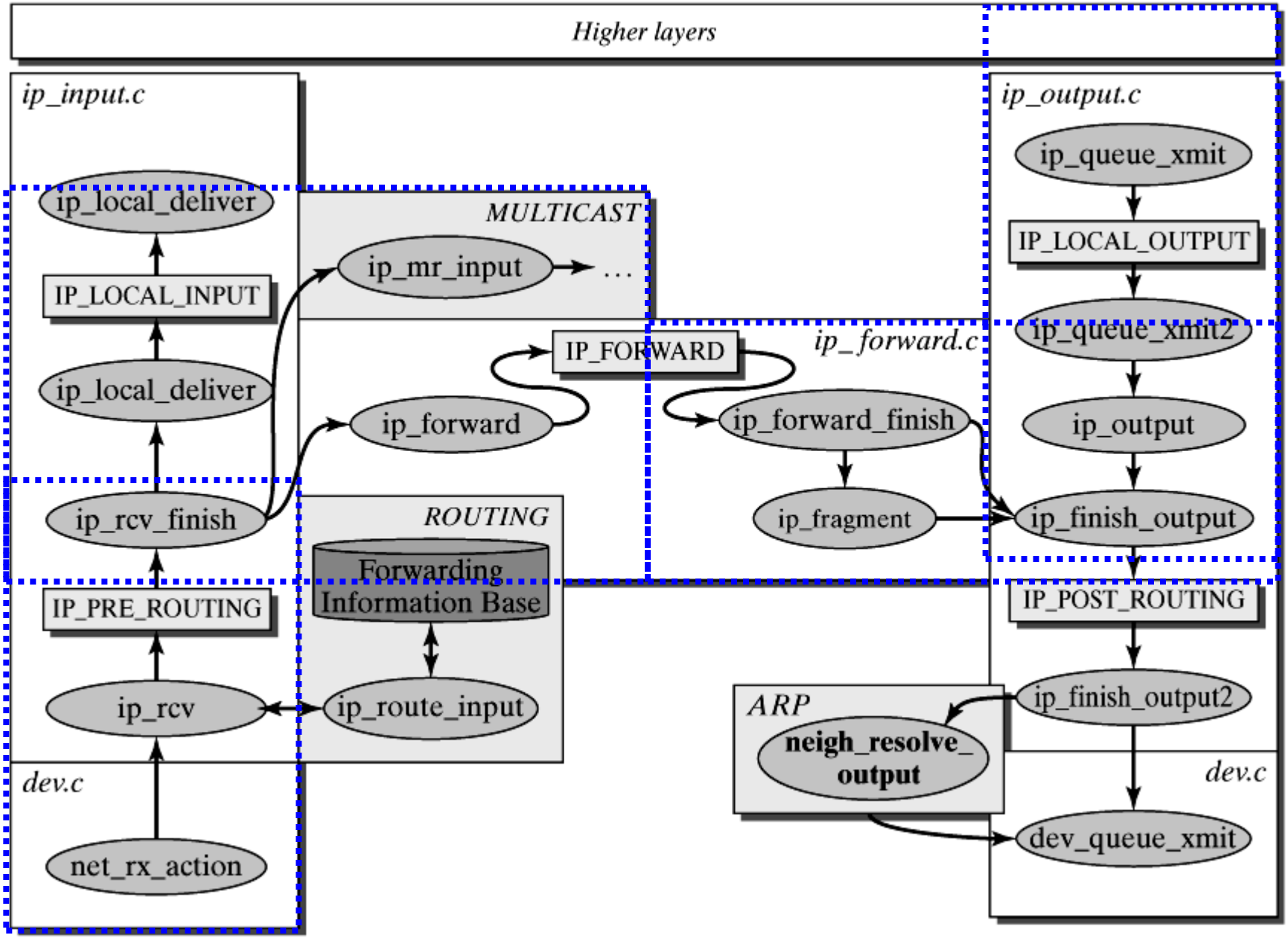
IP Packet Header

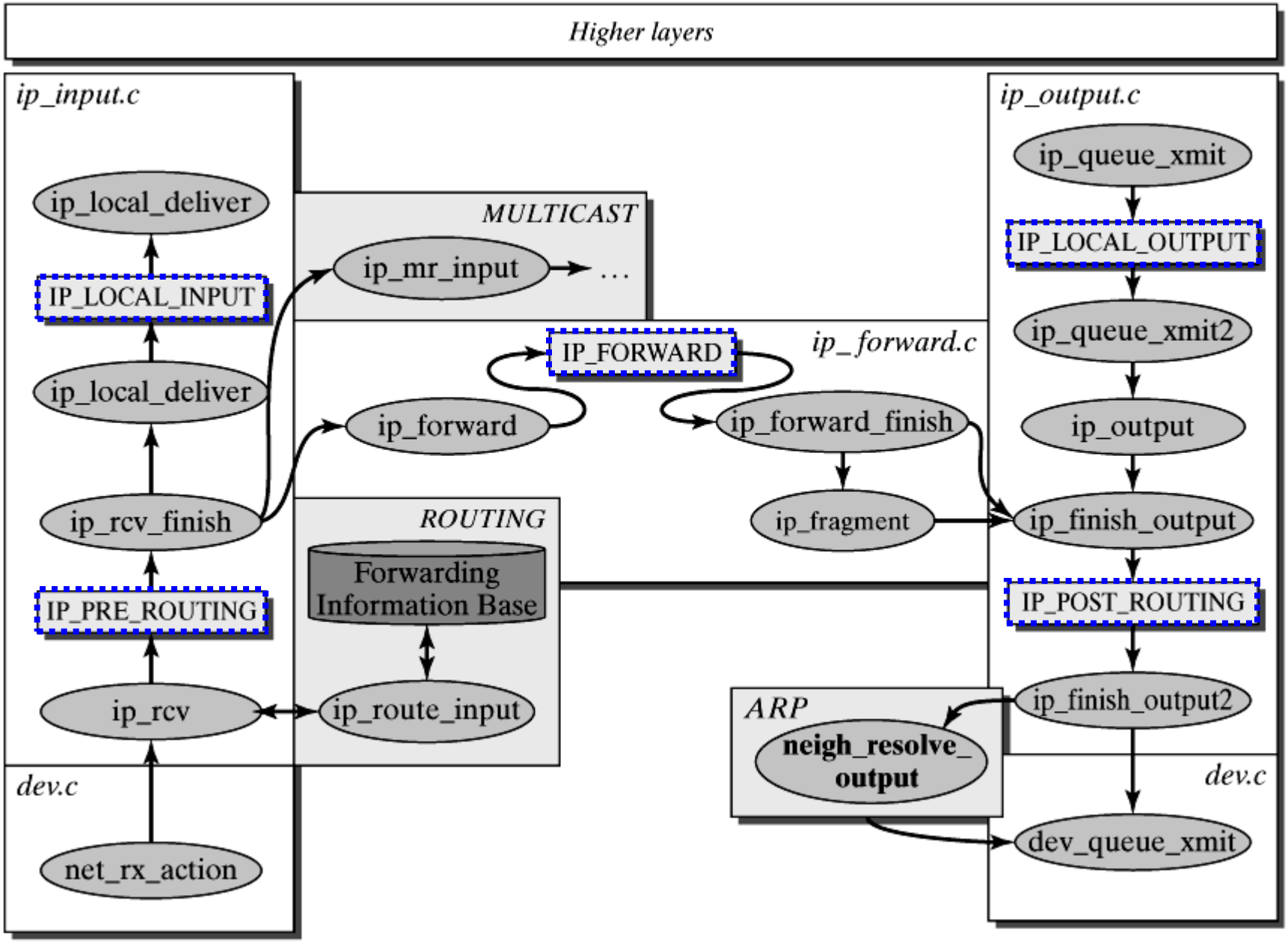


IP Address Classes



The class-A network address 127 represents the **loopback** network device of a computer.
An IP address with all bits of the computer part set to zero identifies the **network** itself.
An IP address where the computer part consists of 1-bits defines a **broadcast** address.

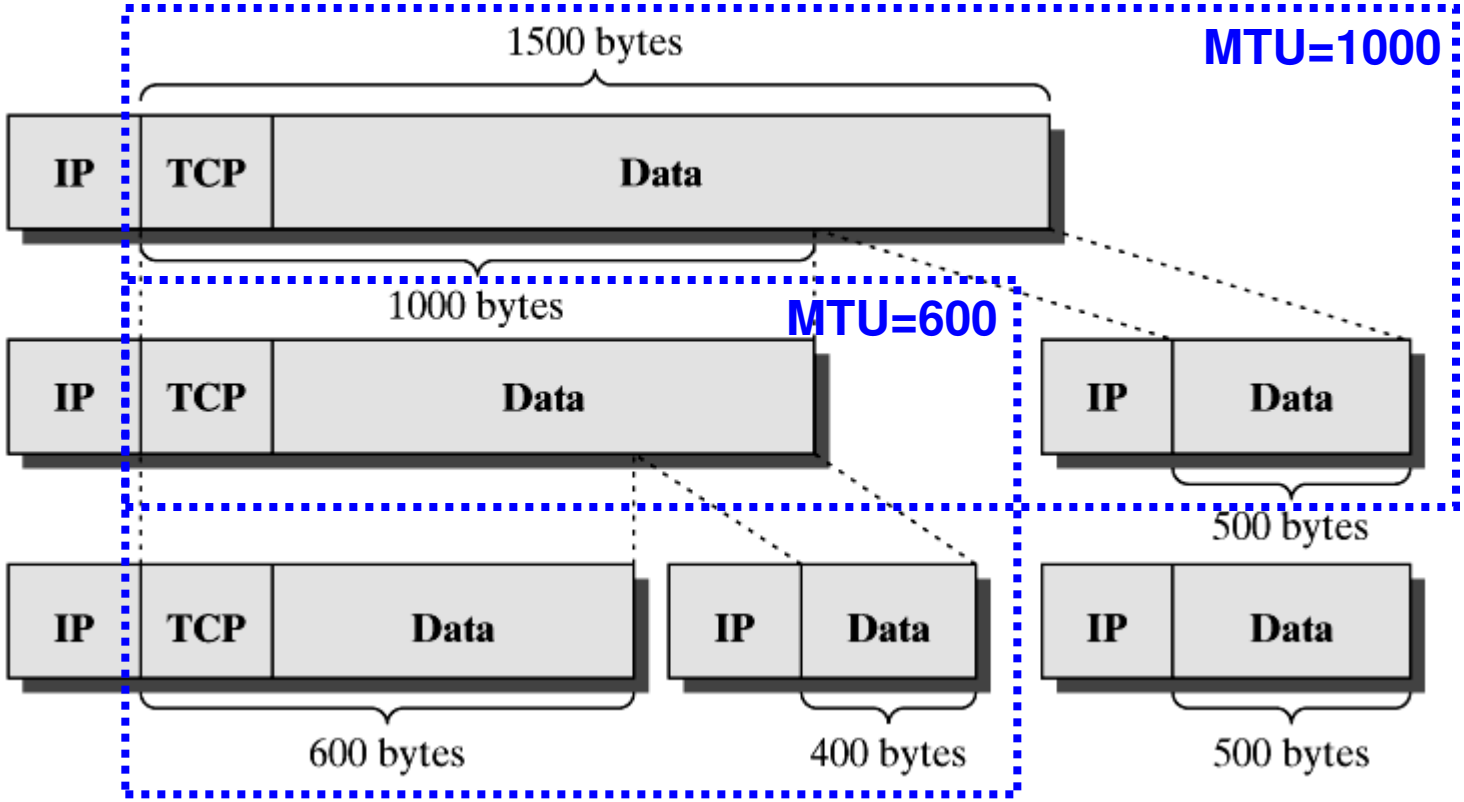

Higher layers
ip_input.c
ip_local_deliver
IP_LOCAL_INPUT
ip_local_deliver
ip_rcv_finish
IP_PRE_ROUTING
ip_rcv
dev.c
net_rx_action
MULTICAST
ip_mr_input
...
IP_FORWARD
ip_forward
ROUTING
Forwarding Information Base
ip_route_input
ip_forward.c
ip_forward_finish
ip_fragment
ARP
neigh_resolve_ output
ip_output.c
ip_queue_xmit
IP_LOCAL_OUTPUT
ip_queue_xmit2
ip_output
ip_finish_output
IP_POST_ROUTING
ip_finish_output2
dev.c
dev_queue_xmit


Higher layers
ip_input.c
ip_local_deliver
IP_LOCAL_INPUT
ip_local_deliver
ip_rcv_finish
IP_PRE_ROUTING
ip_rcv
net_rx_action
dev.c
MULTICAST
ip_mr_input
...
IP_FORWARD
ip_forward
ip_forward_finish
ip_forward.c
ROUTING
Forwarding Information Base
ip_route_input
ip_fragment
ARP
neigh_resolve_ output
ip_output.c
ip_queue_xmit
IP_LOCAL_OUTPUT
ip_queue_xmit2
ip_output
ip_finish_output
IP_POST_ROUTING
ip_finish_output2
dev.c
dev_queue_xmit

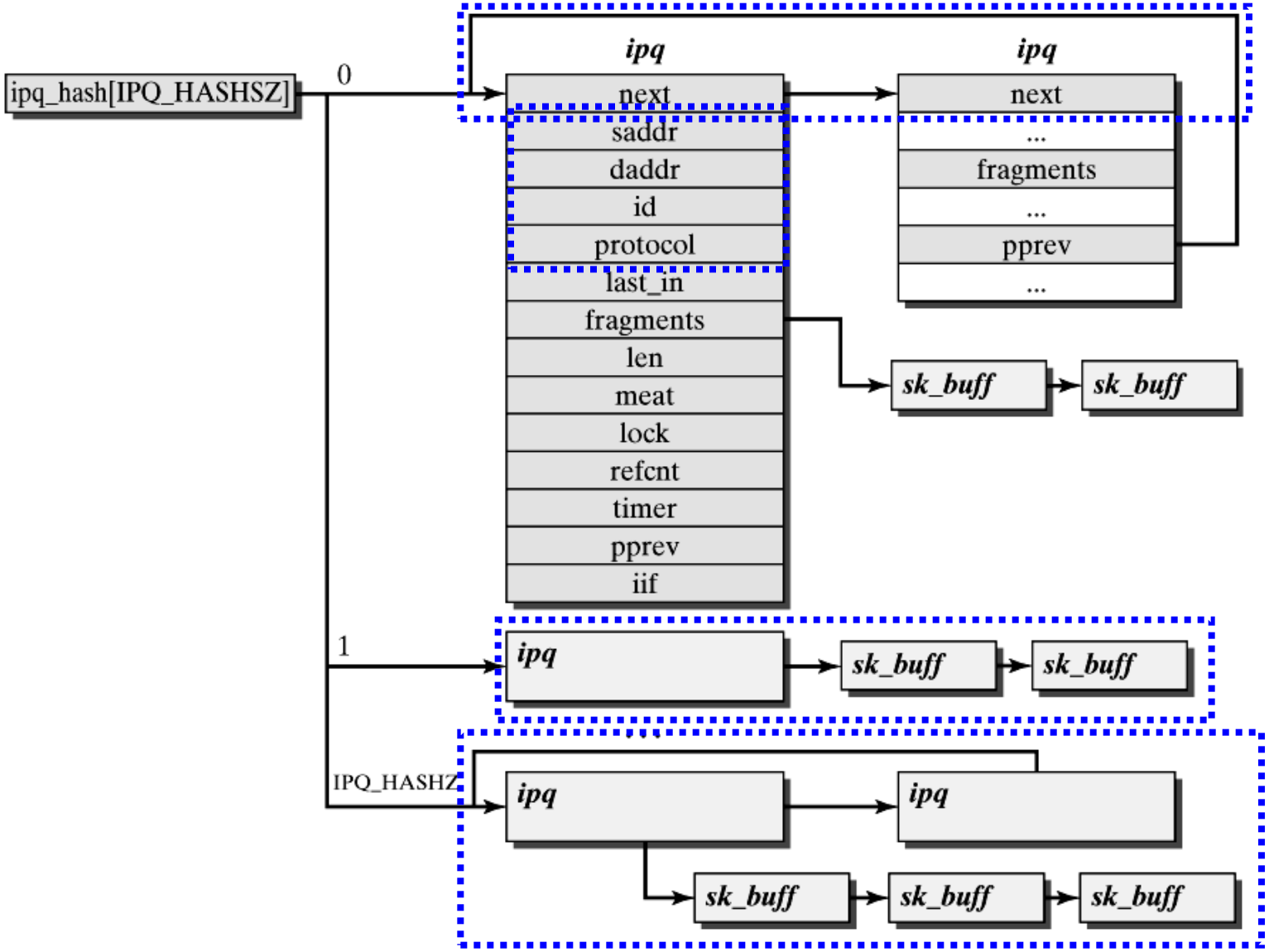
Delivering Packets Locally

- If `ip_route_input()` is the selected route, then the packet is addressed to the local computer. In this case, branching is to `ip_local_deliver()` rather than to `ip_forward()`.
  - Reassemble fragmented packets
  - ip_local_deliver_finish(): RAW-IP socket or up to transport layer

Fragmenting an IP Datagram

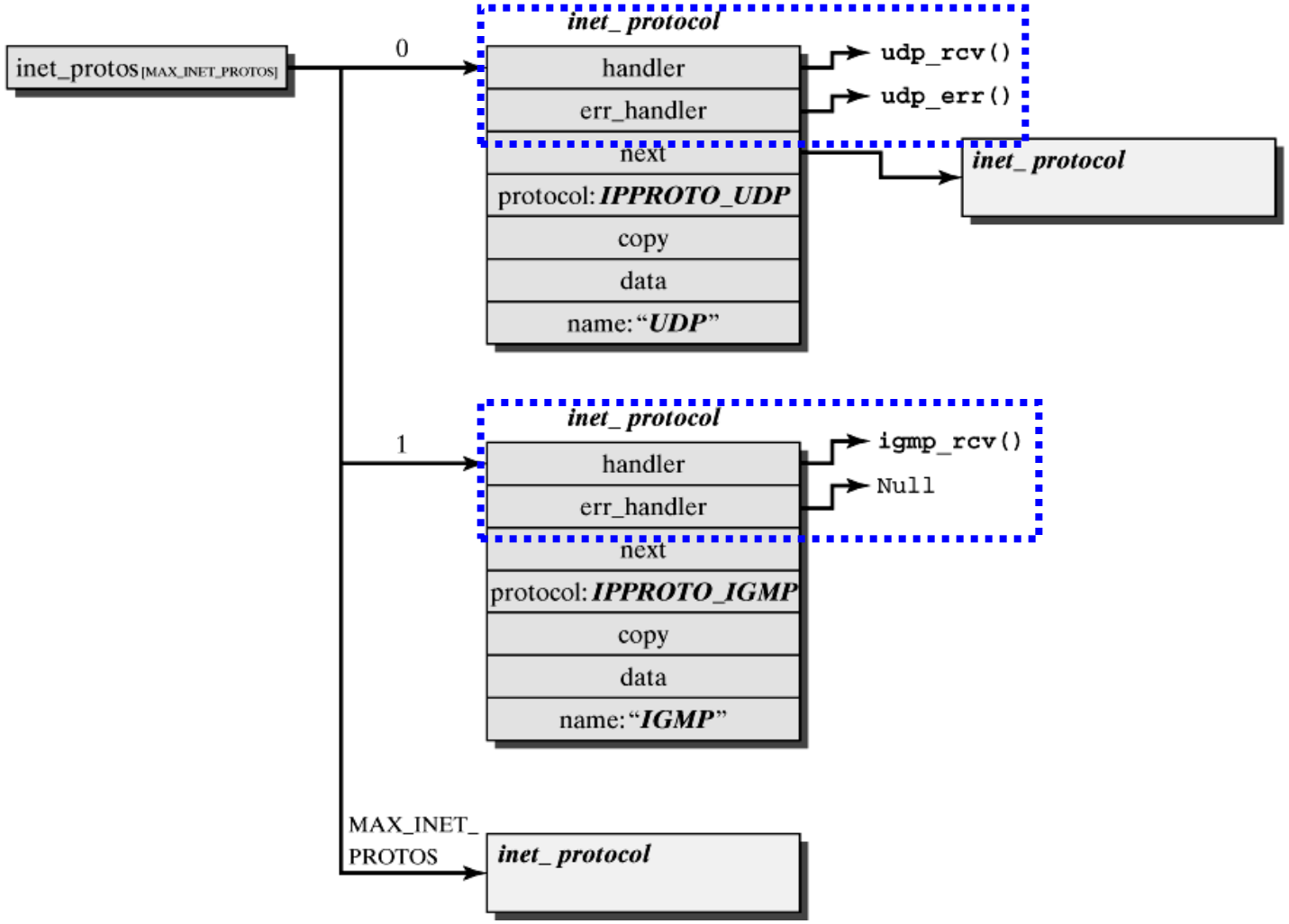


Each network has a maximum packet size, which is called Maximum Transfer Unit (MTU). If the MTU of a transmission medium is smaller than the size of a packet, then the packet has to be split into smaller IP packets.

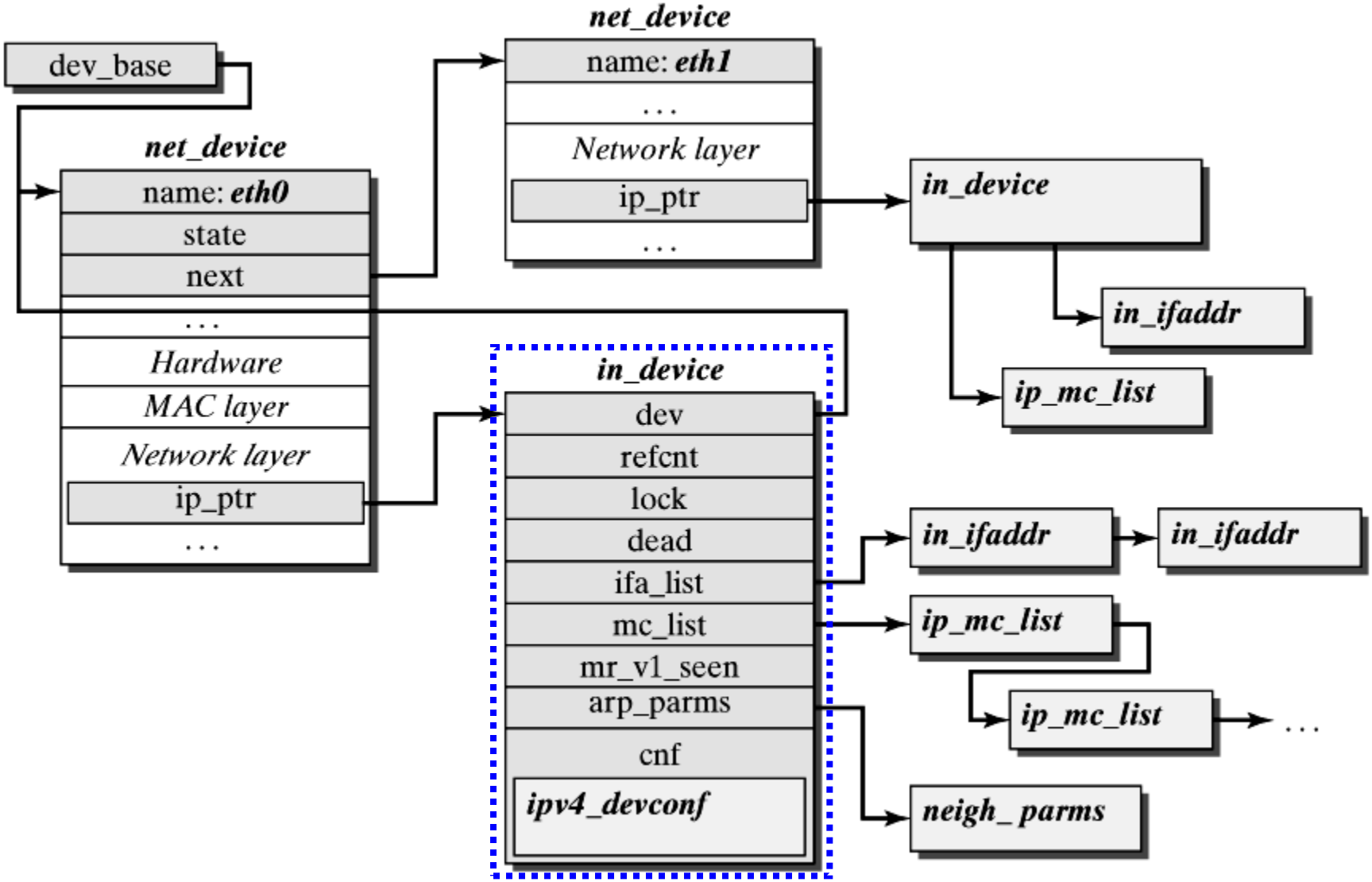


The **saddr**, **daddr**, **id**, and **protocol** elements are keys for the hash function and the allocation of incoming fragments to their IP datagrams.

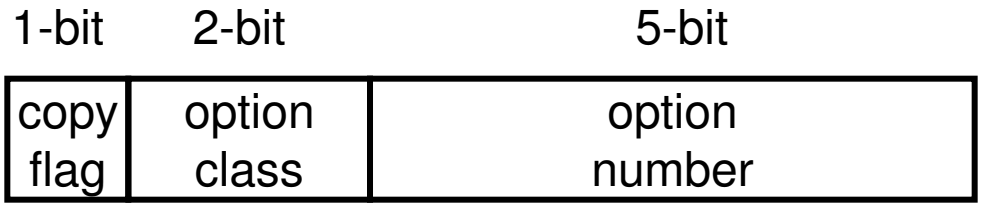
Transport-Layer Packets



IP Network Device

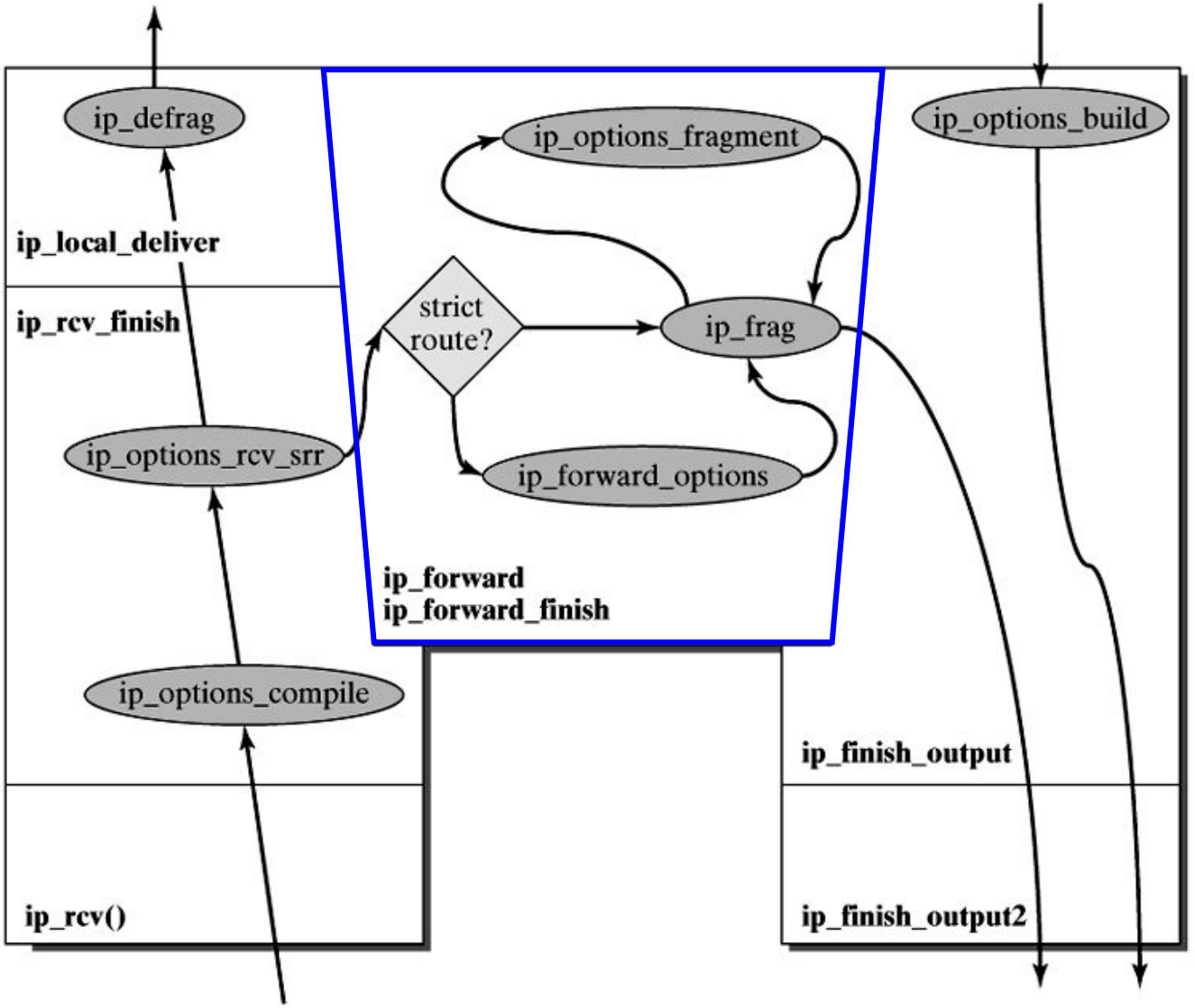


IP Options



Class	Number	Length	Name
0	0	-	End of Option
0	1	-	No Operation
0	2	11	Security
0	3	var	Loose Source Routing
0	9	var	Strict Source Routing
0	7	var	Record Route
0	8	4	Stream ID
2	4	var	Internet Timestamp

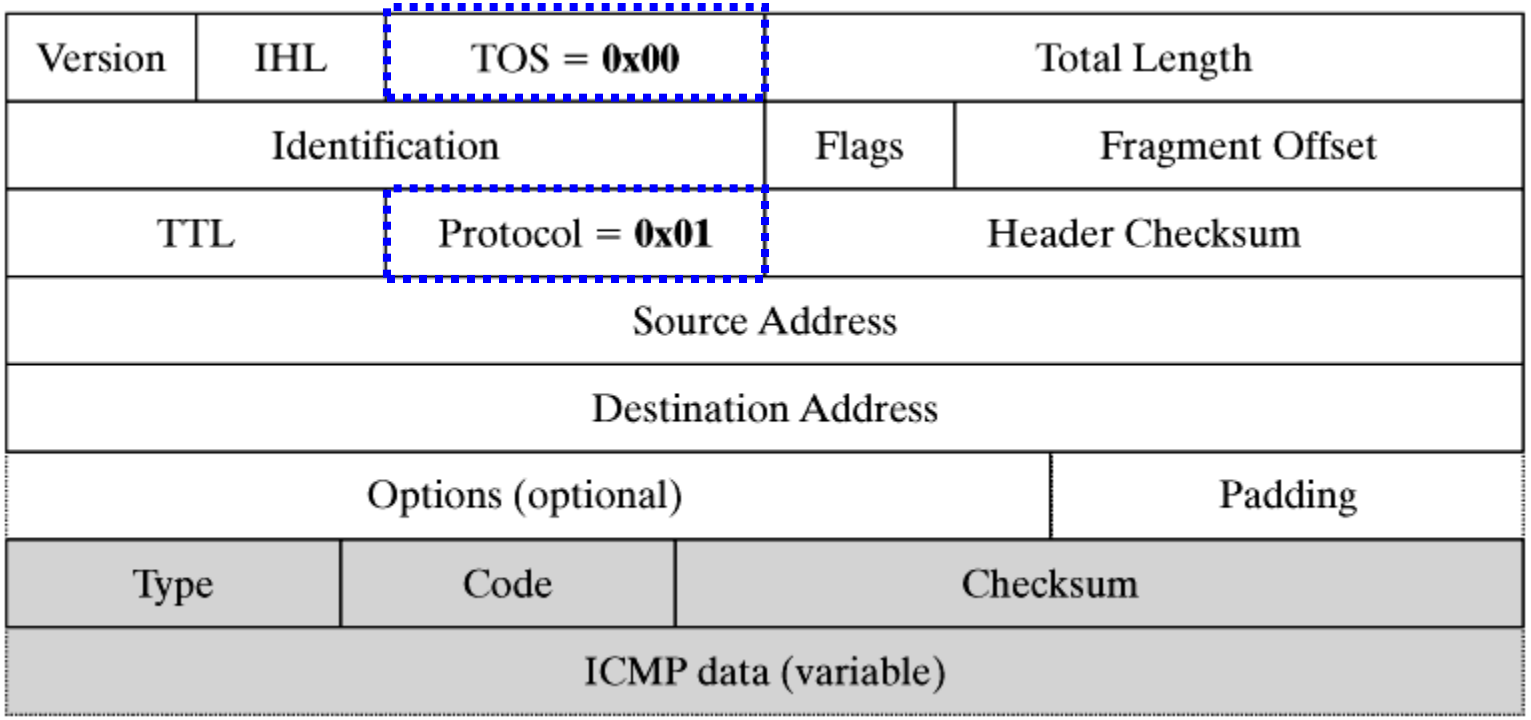
IP Options in the IP layer



Internet Control Message Protocol

- Error-report mechanism for the IP layer.
- The most popular application of ICMP is error detection or error diagnostics. (`ping`)

ICMP Packet Header



ICMP Packet Type (RFC 792)

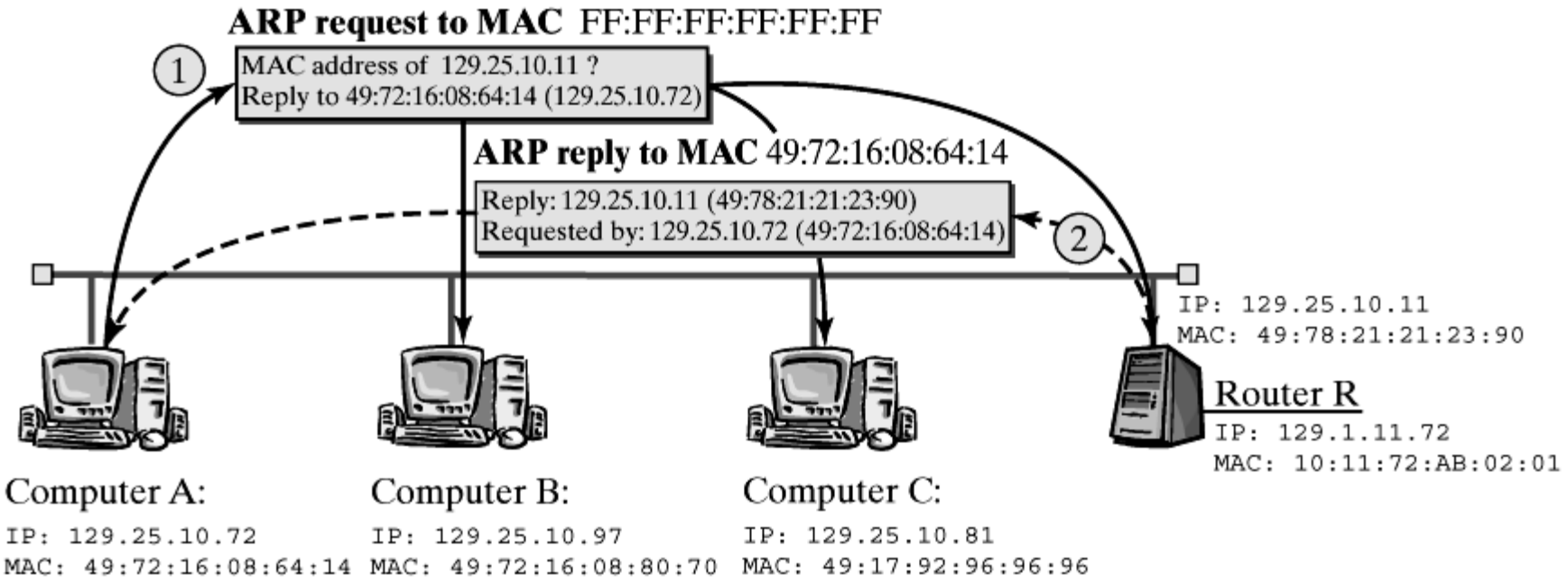
Type	Description
Destination Unreachable	The destination address cannot be reached.
Time Exceeded	A packet was discarded, because its TTL has expired.
Parameter Problem	Unknown or false options.
Source Quench	Informs the sender that IP packets were lost to overload.
Redirect	Enables path optimization.
Echo and Echo Reply	The data sent to the destination address is returned in a reply.
Timestamp and Timestamp Reply	The timestamp sent to the destination address is used-to reply with the timestamp of the destination address.
Information Request und Information Reply	Request/reply used to find the network a computer connects to.

ICMP in the Linux Kernel

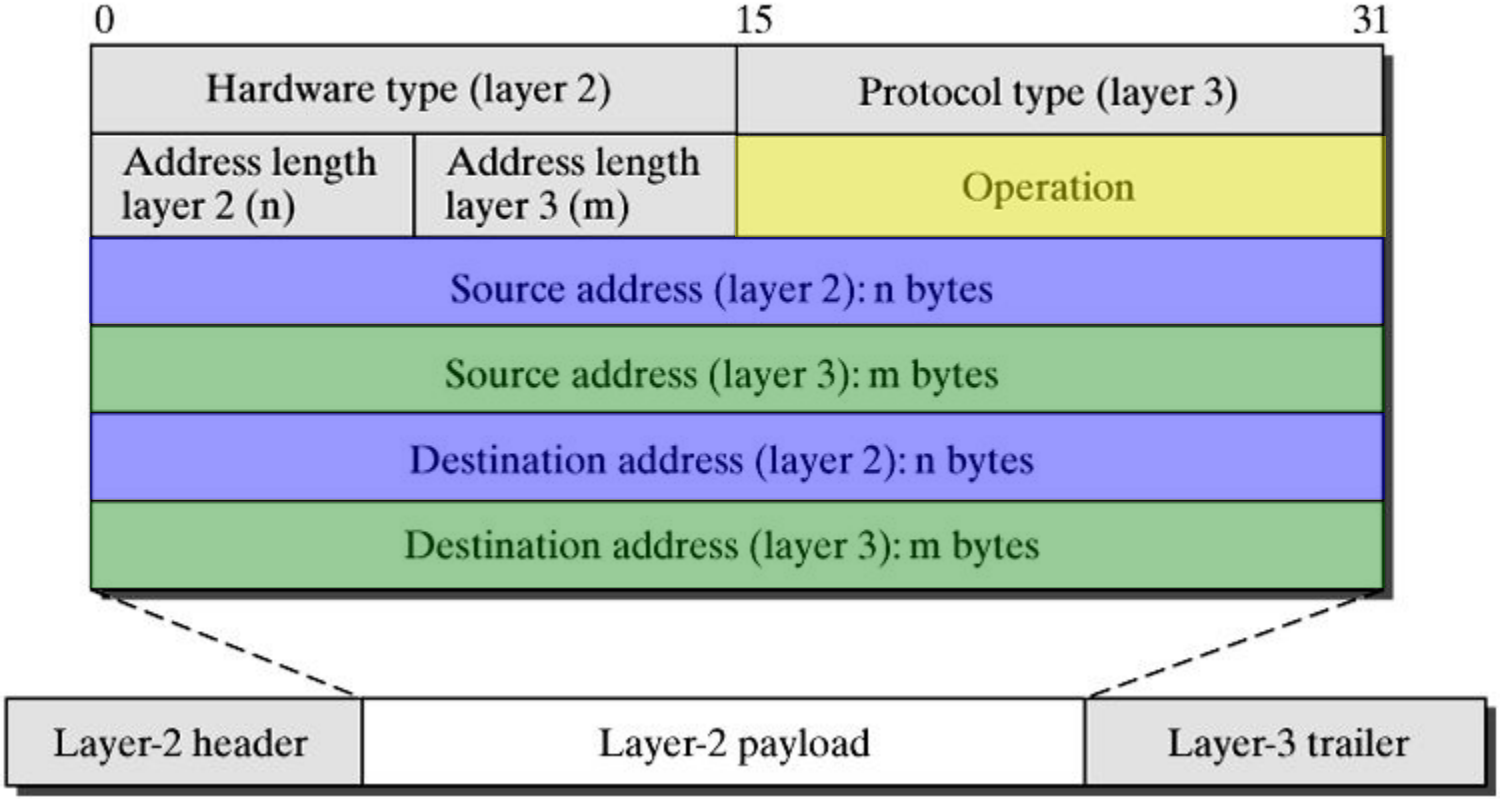
- Sending ICMP Packets
  - `icmp_send()`
- Handling Incoming ICMP Packets
  - `icmp_rcv()`, `icmp_reply()`, `icmp_redirect()`, `icmp_unreach()`, `icmp_echo()`, `icmp_timestamp()`, `icmp_addres()`, `icmp_address_reply()`
- ICMP messages generated within the kernel

Type	**Module**	**Reason**
Time Exceeded	Forward and defragment packets	A packet was discarded because it's TTL expired.
Parameter Problem	Detect packet options	Unknown or false options
Redirect	Packet routing	Obvious potential for optimization
Destination Unreachable	All modules that send, forward, or deliver IP packets	Inability to deliver a packet

Address Resolution Protocol



ARP protocol data unit



ARP request to FF:FF:FF:FF:FF:FF

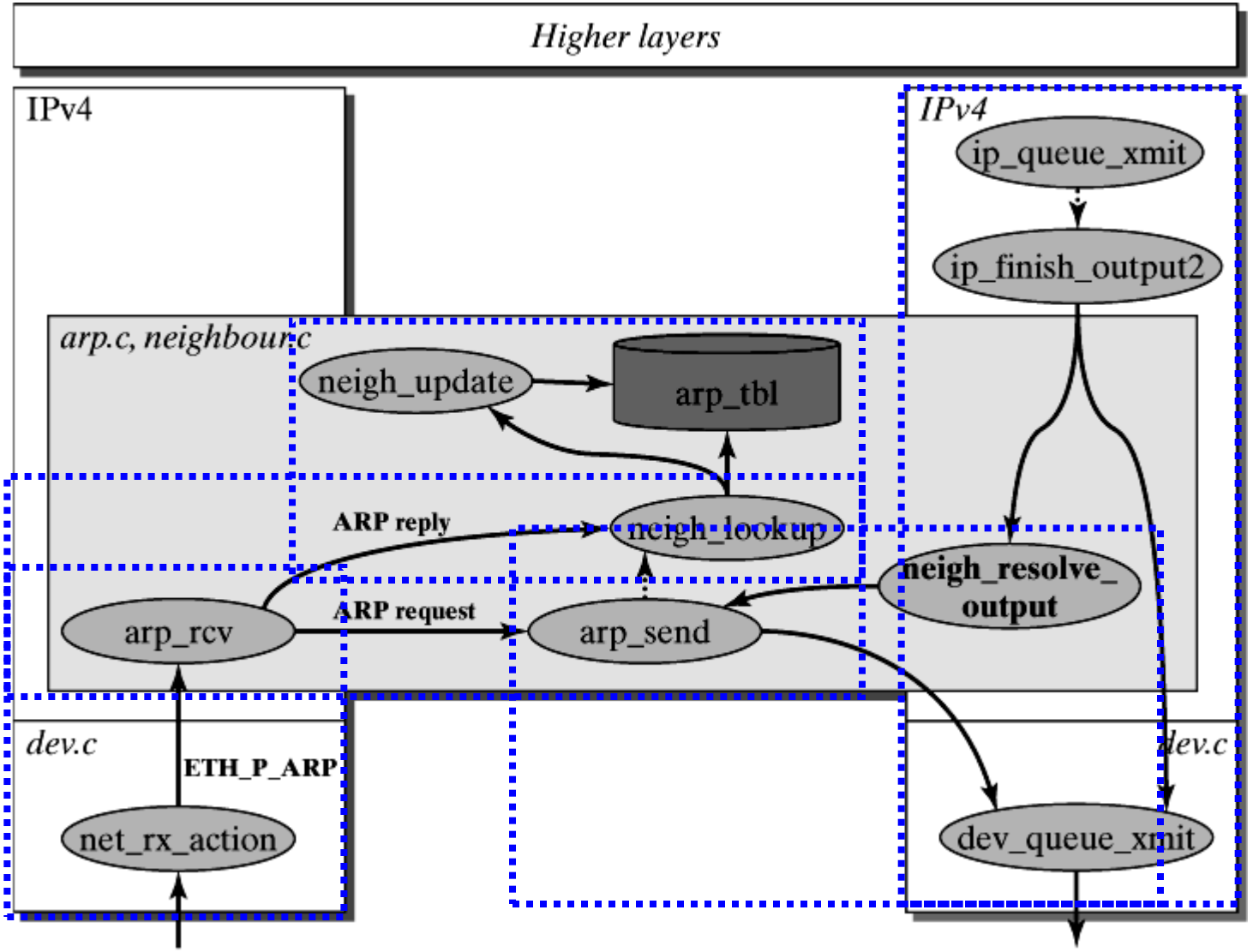
0		15	31
0x00 01 (*Ethernet*)		**0x80 00** (*Internet Protocol*)	
6	**4**	**0x00 01** (*ARP request*)	
49 72 16 08			
64 14		**129 25**	
10 72		**00 00**	
00 00 00 00			
129 25 10 11			

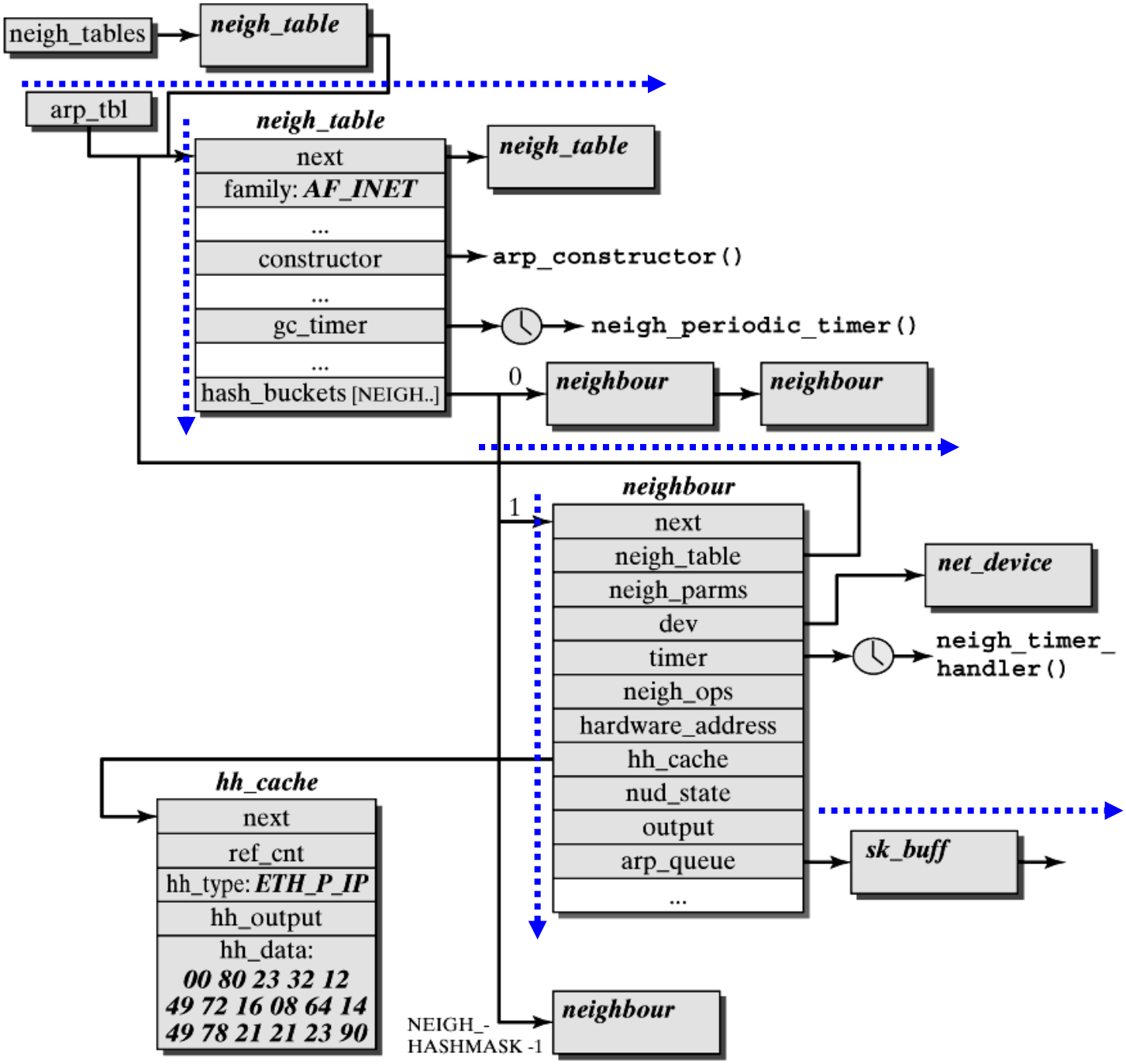
ARP reply to 49:72:16:08:64:14

0		15	31
0x00 01 (*Ethernet*)		**0x80 00** (*Internet Protocol*)	
6	**4**	**0x00 02** (*ARP reply*)	
49 72 16 08			
64 14		**129 25**	
10 72		**49 78**	
21 21 23 90			
129 25 10 11			

ARP Command

```
root@tux # arp -a
IP address HW type HW address
129.25.10.97 10Mbit/s Ethernet 49:72:16:08:80:70
129.25.10.72 10Mbit/s Ethernet 49:72:16:08:64:14
129.25.10.81 10Mbit/s Ethernet 49:17:92:96:96:96
```


Higher layers
IPv4
IPv4
ip_queue_xmit
ip_finish_output2
arp.c, neighbour.c
neigh_update
arp_tbl
ARP reply
neigh_lookup
neigh_resolve_
output
ARP request
arp_rcv
arp_send
dev.c
ETH_P_ARP
net_rx_action
dev.c
dev_queue_xmit


neigh_tables
neigh_table
arp_tbl
neigh_table
next
family: AF_INET
...
constructor
...
gc_timer
...
hash_buckets [NEIGH..]
neigh_table
arp_constructor()
neigh_periodic_timer()
0
neighbour
neighbour
neighbour
1
next
neigh_table
neigh_parms
dev
timer
neigh_ops
hardware_address
hh_cache
nud_state
output
arp_queue
...
net_device
neigh_timer_
handler()
sk_buff
hh_cache
next
ref_cnt
hh_type: ETH_P_IP
hh_output
hh_data:
00 80 23 32 12
49 72 16 08 64 14
49 78 21 21 23 90
NEIGH_-
HASHMASK -1
neighbour

Outline

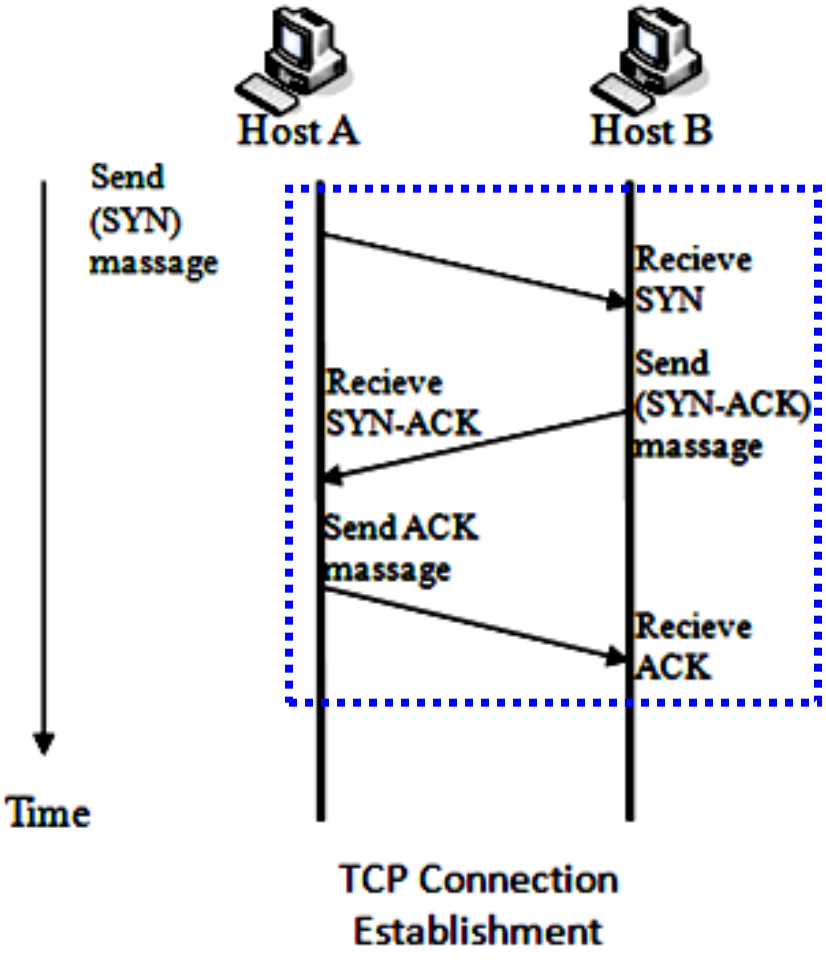
- Architecture of Communication System
- Managing Network Packets
- Network Device
- Data-Link Layer
- Network Layer
- Transport Layer
- Sockets in Linux Kernel
- Socket Programming

Transport Layer

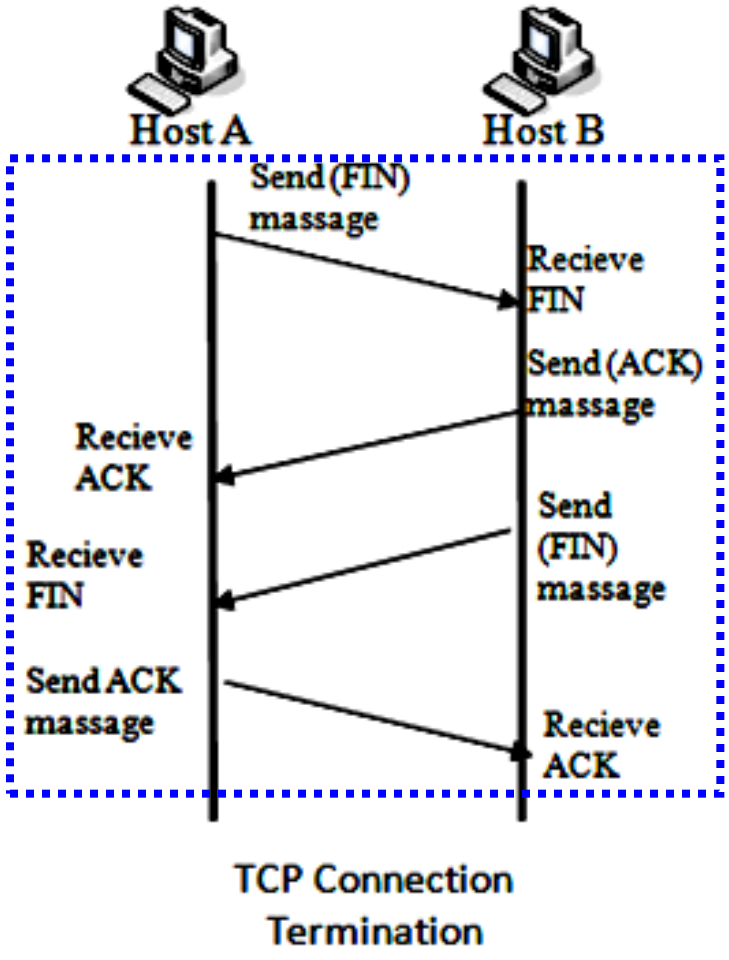
- Transmission Control Protocol (TCP)
- User Datagram Protocol (UDP)

Transmission Control Protocol

- Connection orientation
- Peer-to-peer communication
- Complete reliability
- Full-duplex communication
- Byte-stream interface
- Reliable connection startup
- Graceful connection shutdown

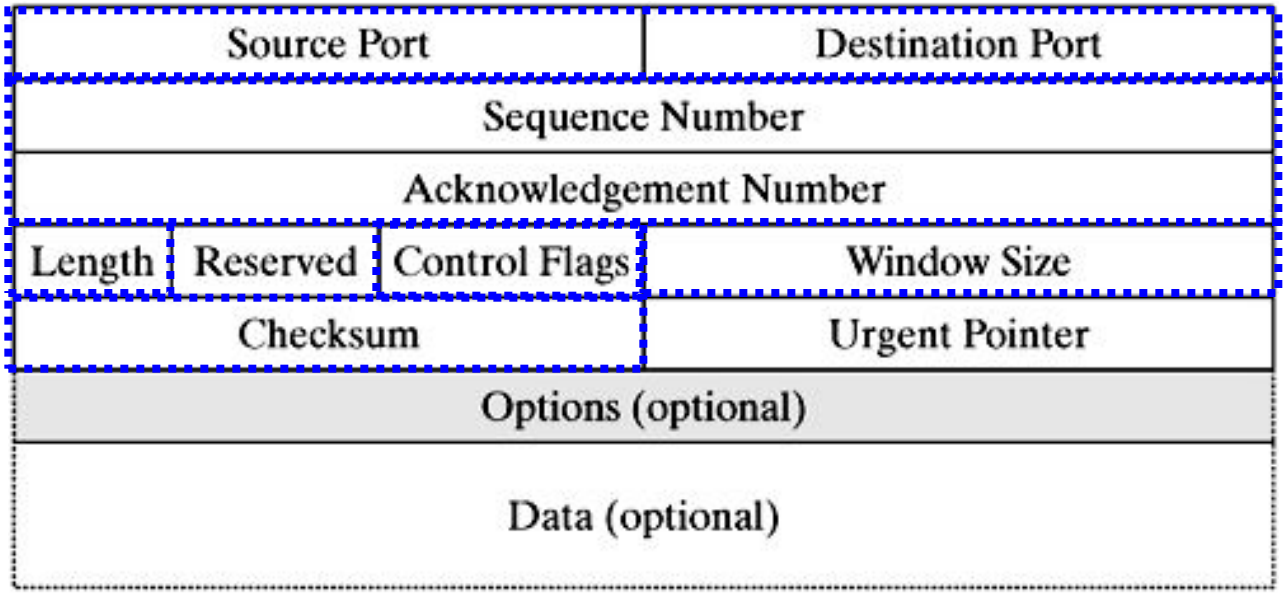


TCP Connection
Establishment

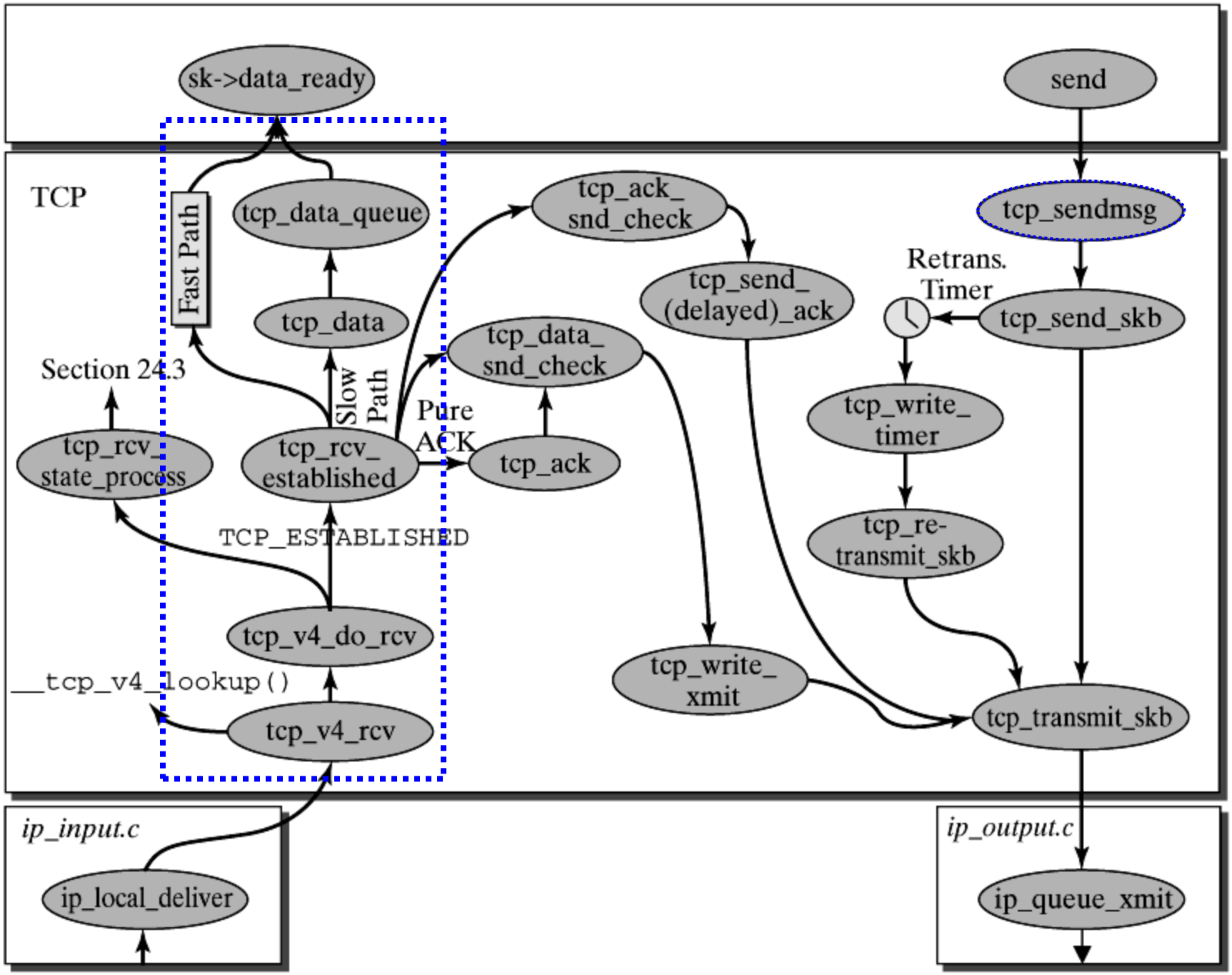


TCP Connection
Termination

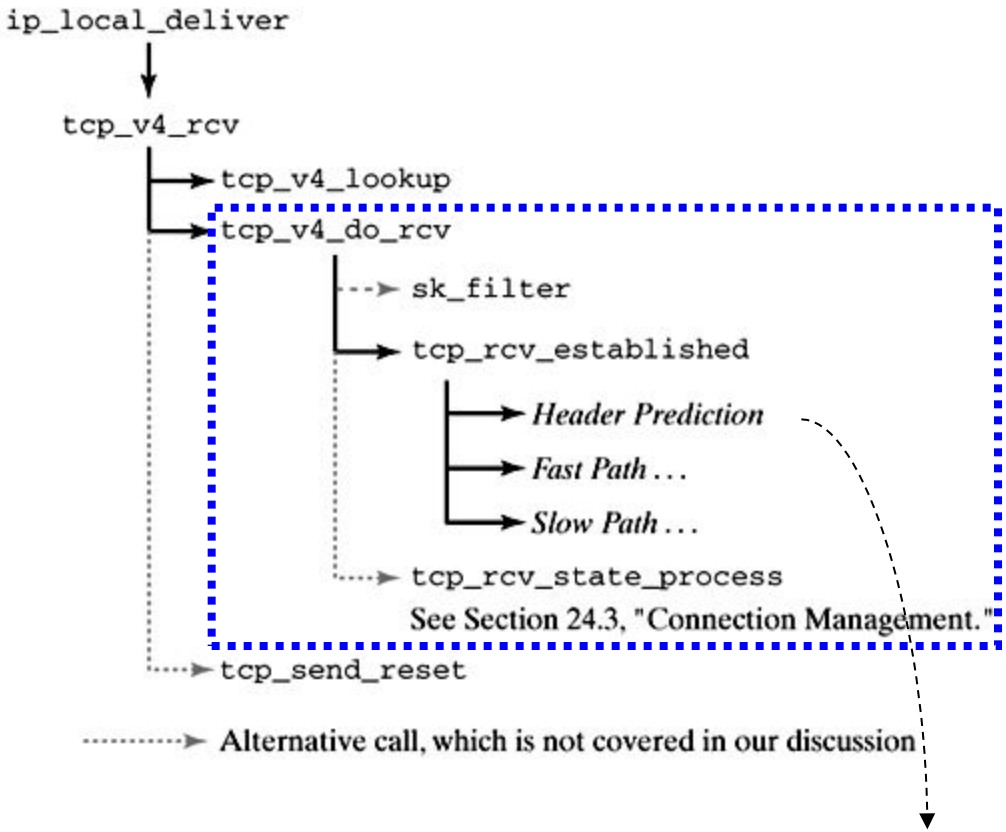
TCP Packet Header



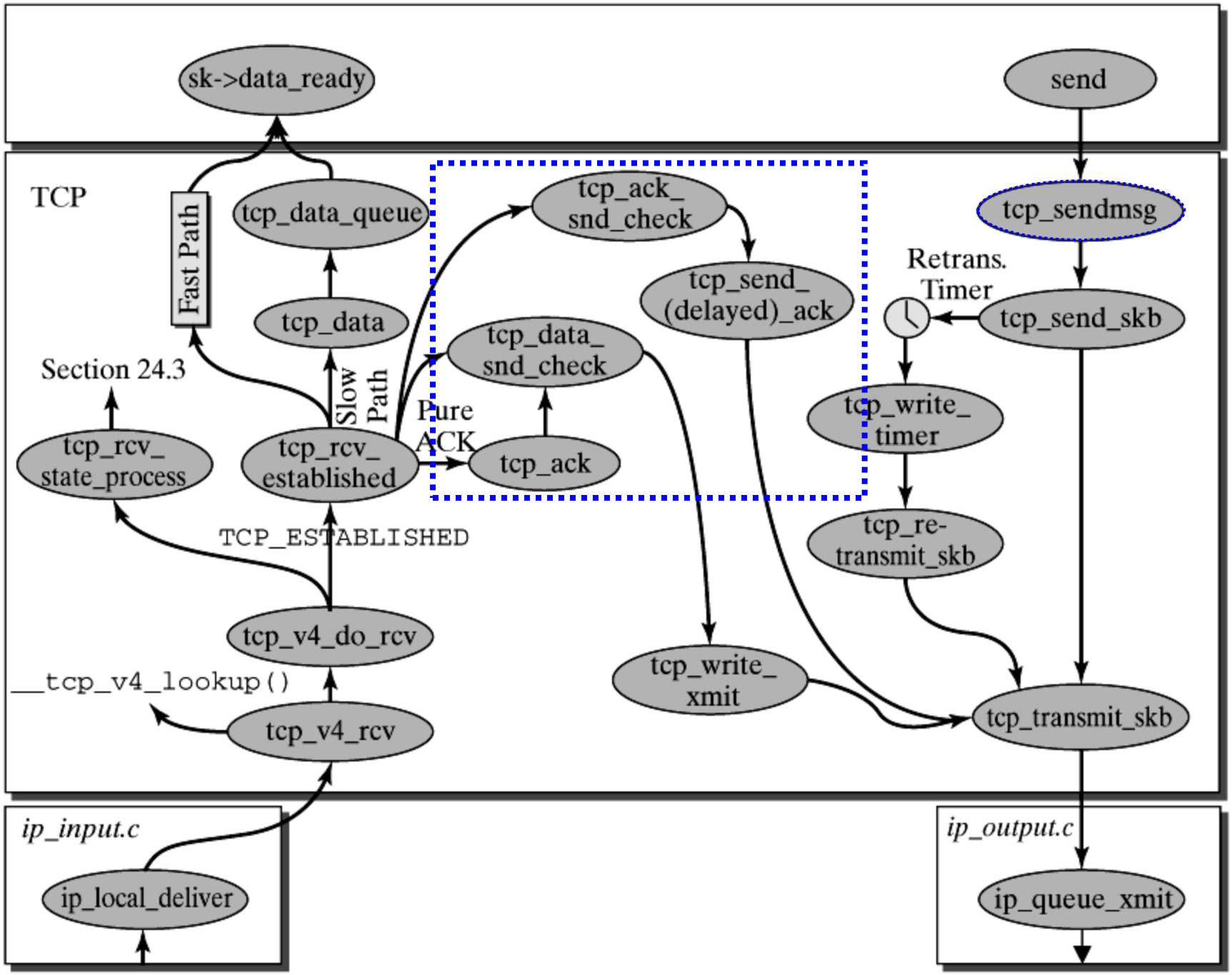
URG	points to important data that have to be forwarded immediately.
SYN	is used to establish connections. SYN = 1 denotes a connection request.
ACK	shows that the ACKNOWLEDGEMENT NUMBER field includes relevant data.
RST	can request a connection to be reset. RST = 1 denotes a request to reset a connection.
PSH	means that the TCP instance must immediately pass the data received to the higher layers.
FIN	means that the connection is to be torn down.


sk->data_ready
send
TCP
tcp_data_queue
Fast Path
tcp_ack_
snd_check
tcp_sendmsg
Retrans.
Timer
tcp_send_
(delayed)_ack
tcp_send_skb
tcp_data
tcp_data_
snd_check
Section 24.3
Slow
Path
Pure
ACK
tcp_write_
timer
tcp_rcv_
state_process
tcp_rcv_
established
tcp_ack
tcp_re-
transmit_skb
TCP_ESTABLISHED
tcp_v4_do_rcv
tcp_write_
xmit
__tcp_v4_lookup()
tcp_transmit_skb
tcp_v4_rcv
ip_input.c
ip_output.c
ip_local_deliver
ip_queue_xmit

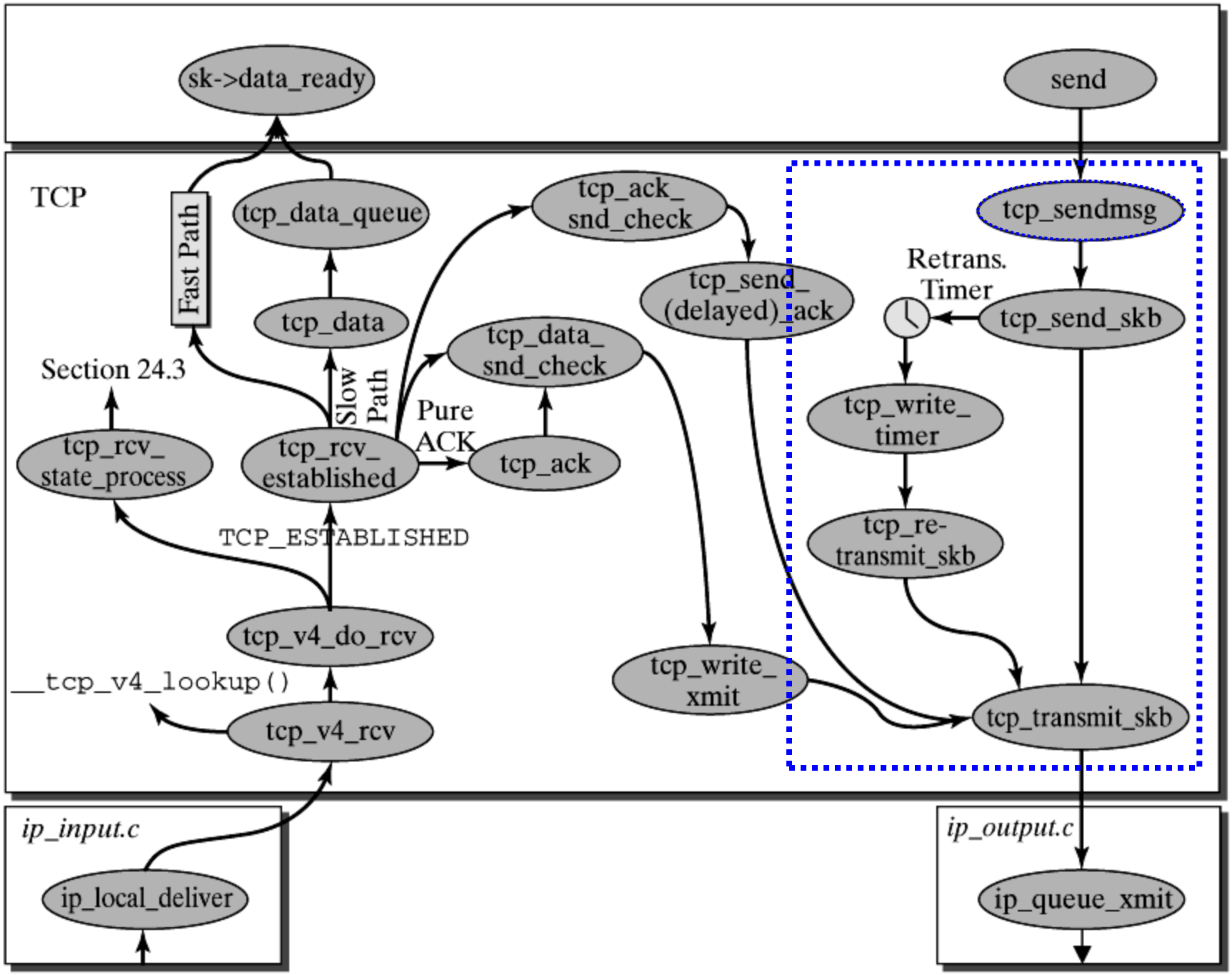
Receiving TCP Segment



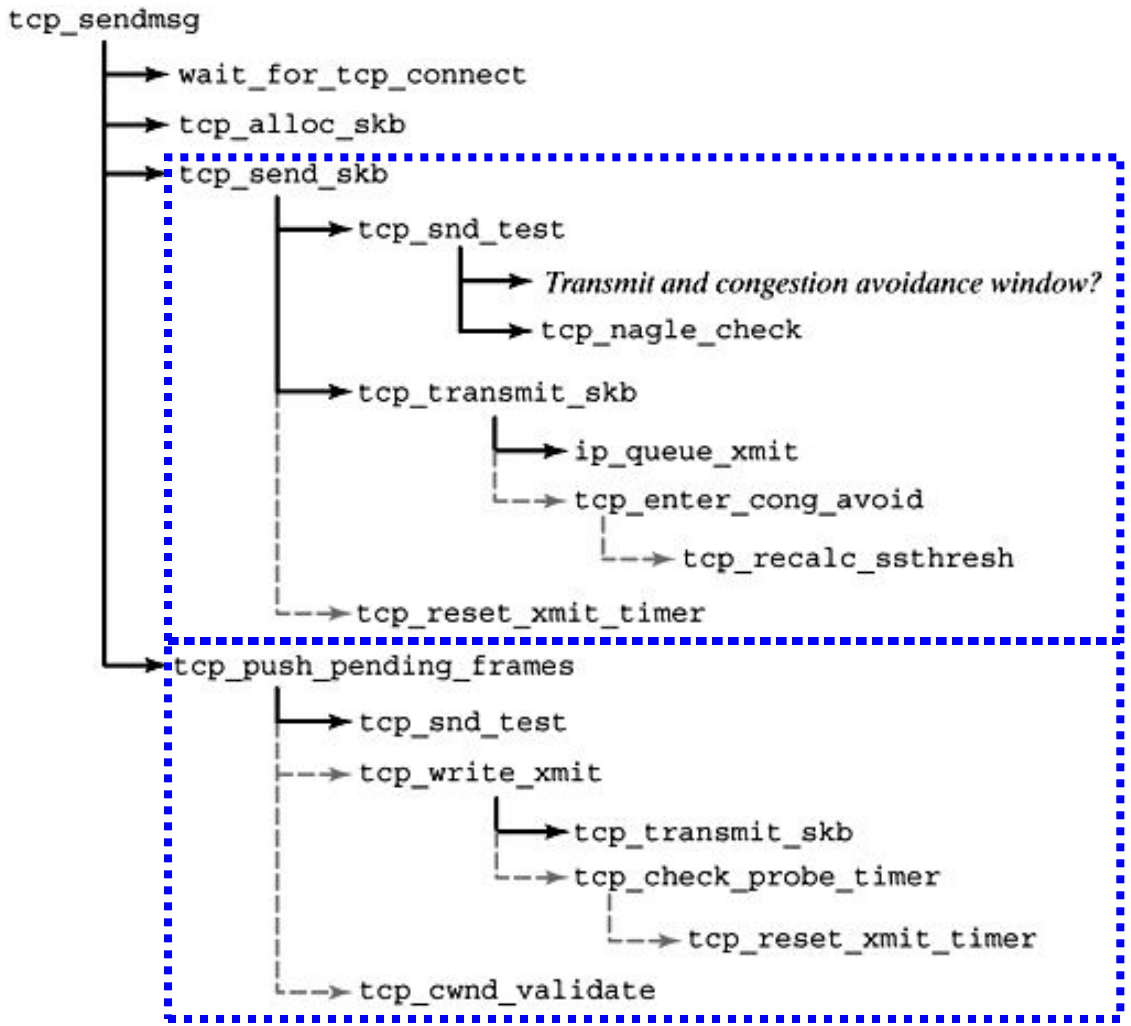

```
if ((tcp_flag_word(th) & TCP_HP_BIPS) == tp->pred_flags &&
        TCP_SKB_CB(skb)->seq == tp->rcv_nxt)
{ (...FAST PATH...) }
else
{ (...SLOW PATH...) }
```

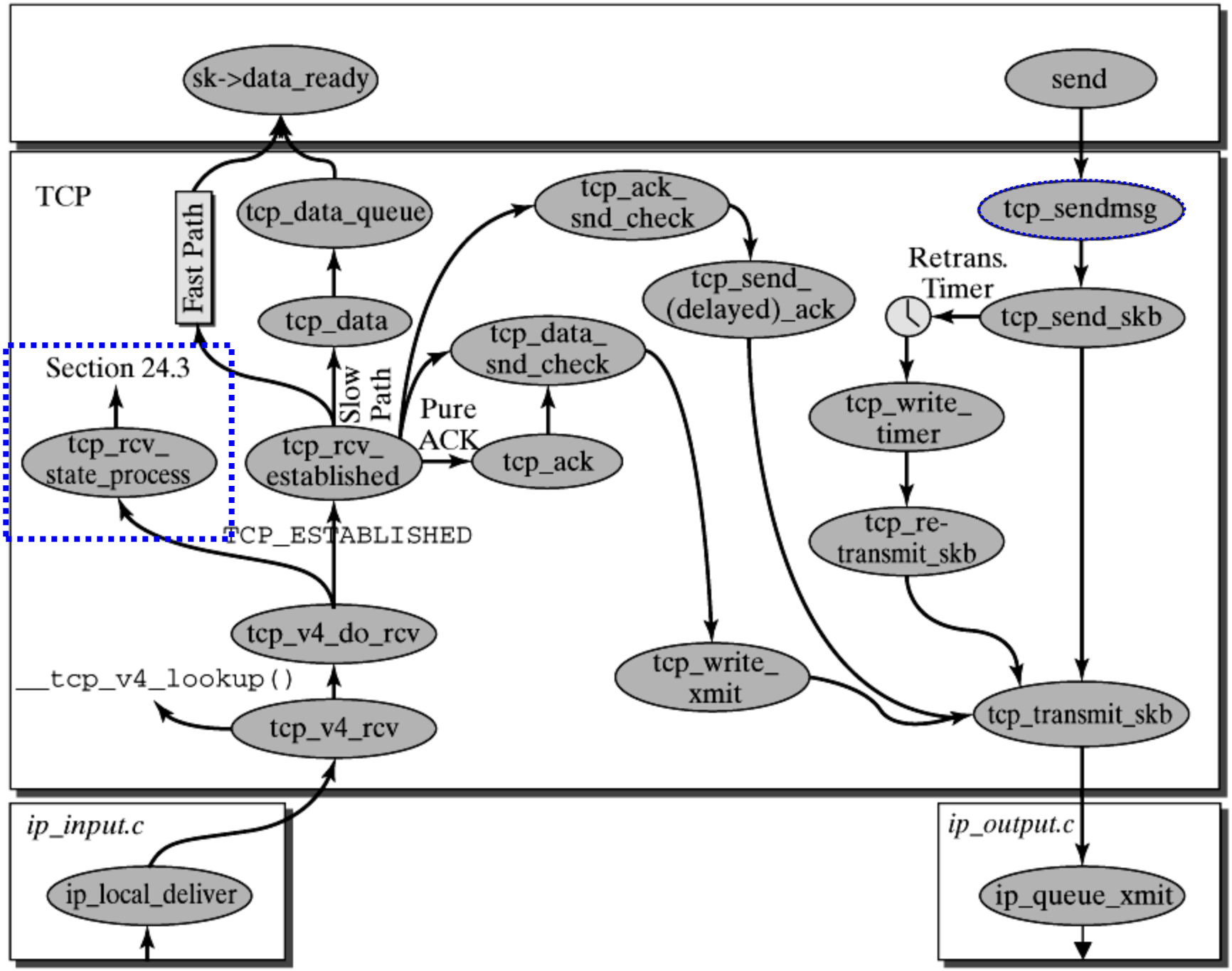

sk->data_ready
send
TCP
tcp_data_queue
tcp_ack_
snd_check
tcp_sendmsg
Fast Path
tcp_send_
(delayed)_ack
Retrans.
Timer
tcp_data
tcp_send_skb
tcp_data_
snd_check
Section 24.3
Slow
Path
Pure
ACK
tcp_write_
timer
tcp_rcv_
state_process
tcp_rcv_
established
tcp_ack
tcp_re-
transmit_skb
TCP_ESTABLISHED
tcp_v4_do_rcv
tcp_write_
xmit
__tcp_v4_lookup()
tcp_transmit_skb
tcp_v4_rcv
ip_input.c
ip_output.c
ip_local_deliver
ip_queue_xmit

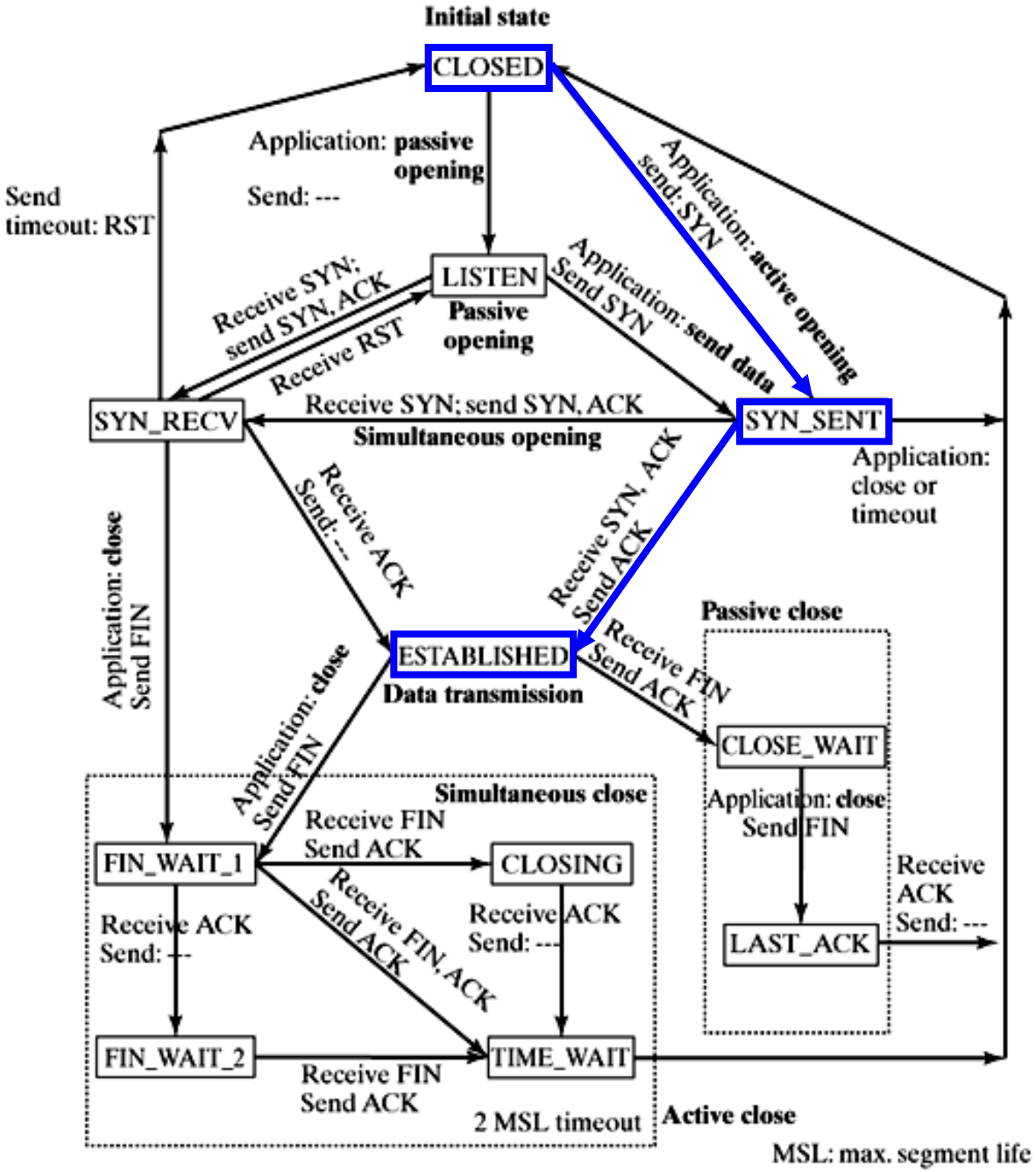
```
static __inline__ void tcp_data_snd_check(struct sock *sk) {
        struct sk_buff *skb = sk->tp_pinfo.af_tcp.send_head;
        struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
        if (skb != NULL)
        {
        if (after(TCP_SKB_CB(skb)->end_seq, tp->snd_una + tp->snd_wnd) ||
                tcp_packets_in_flight(tp) >= tp->snd_cwnd ||
                tcp_write_xmit(sk))
                        tcp_check_probe_timer(sk, tp);
        }
        tcp_check_space(sk);
}

static __inline__ void tcp_ack_snd_check(struct sock *sk) {
        struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
        if (!tcp_ack_scheduled(tp)) {
                /* We sent a data segment already. */
                return;
        }
                        /* More than one full frame received... */
        if (((tp->rcv_nxt - tp->rcv_wup) > tp->ack.rcv_mss
                        /* ... and right edge of window advances far enough.
                         * (tcp_recvmsg() will send ACK otherwise). Or... */
                 && __tcp_select_window(sk) >= tp->rcv_wnd) ||
                        /* We ACK each frame or ... */
                tcp_in_quickack_mode(tp) ||
                        /* We have out of order data. */
                (skb_peek(&tp->out_of_order_queue) != NULL)
        {
                tcp_send_ack(sk); /* Then ack it now */
        }else
        {
                tcp_send_delayed_ack(sk); /* Else, send delayed ack. */
        }
}
```


sk->data_ready
send
TCP
Fast Path
tcp_data_queue
tcp_ack_
snd_check
tcp_sendmsg
Retrans.
Timer
tcp_send_
(delayed)_ack
tcp_send_skb
tcp_data
tcp_data_
snd_check
Section 24.3
Slow
Path
Pure
ACK
tcp_write_
timer
tcp_rcv_
state_process
tcp_rcv_
established
tcp_ack
tcp_re-
transmit_skb
TCP_ESTABLISHED
tcp_v4_do_rcv
tcp_write_
xmit
__tcp_v4_lookup()
tcp_transmit_skb
tcp_v4_rcv
ip_input.c
ip_output.c
ip_local_deliver
ip_queue_xmit

Sending TCP Segments

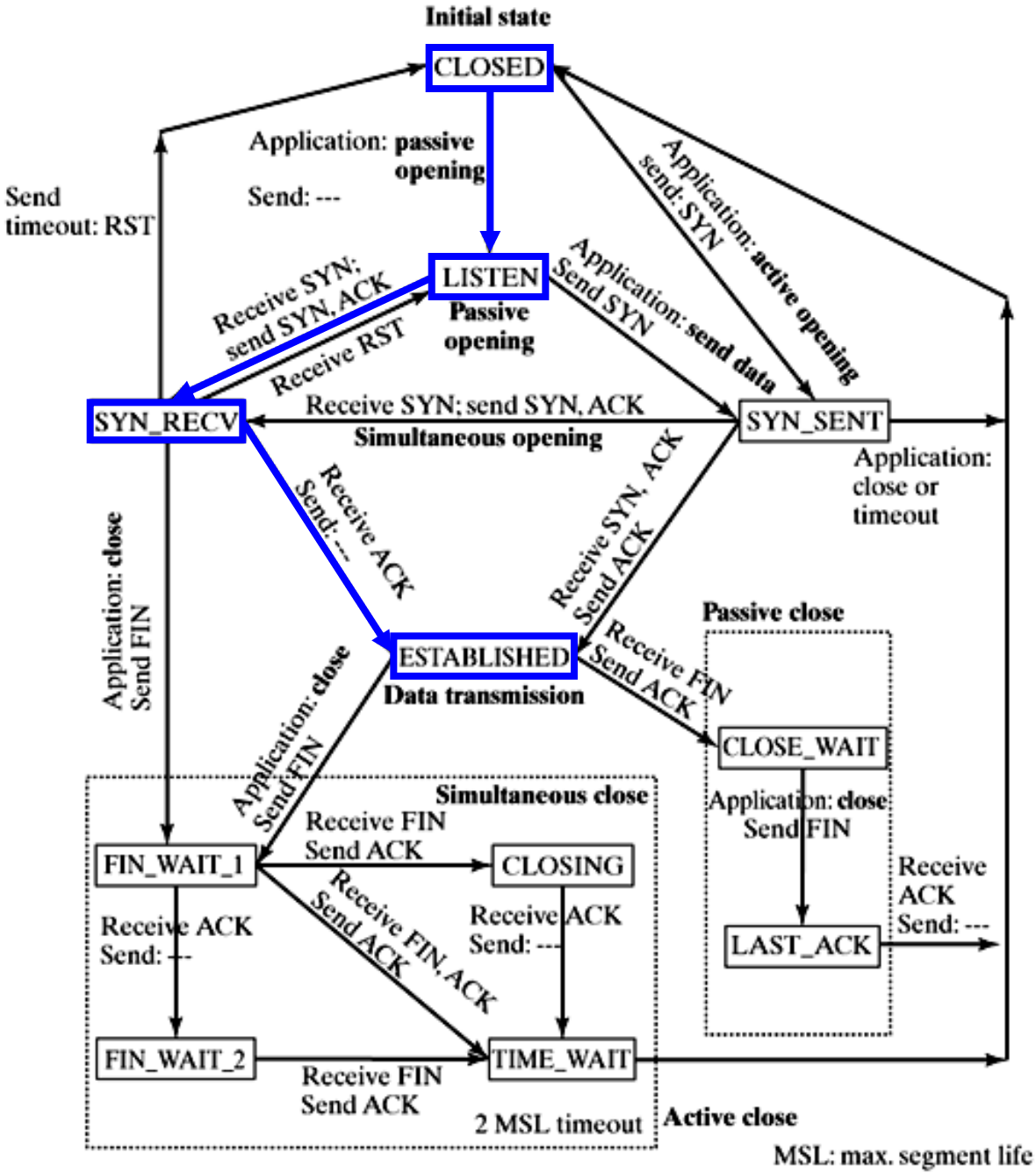



sk->data_ready
send
TCP
Fast Path
tcp_data_queue
tcp_ack_
snd_check
tcp_sendmsg
Retrans.
Timer
tcp_send_
(delayed)_ack
tcp_send_skb
tcp_data
tcp_data_
snd_check
Section 24.3
Slow
Path
Pure
ACK
tcp_write_
timer
tcp_rcv_
state_process
tcp_rcv_
established
tcp_ack
tcp_re-
transmit_skb
TCP_ESTABLISHED
tcp_v4_do_rcv
tcp_write_
xmit
__tcp_v4_lookup()
tcp_transmit_skb
tcp_v4_rcv
ip_input.c
ip_output.c
ip_local_deliver
ip_queue_xmit



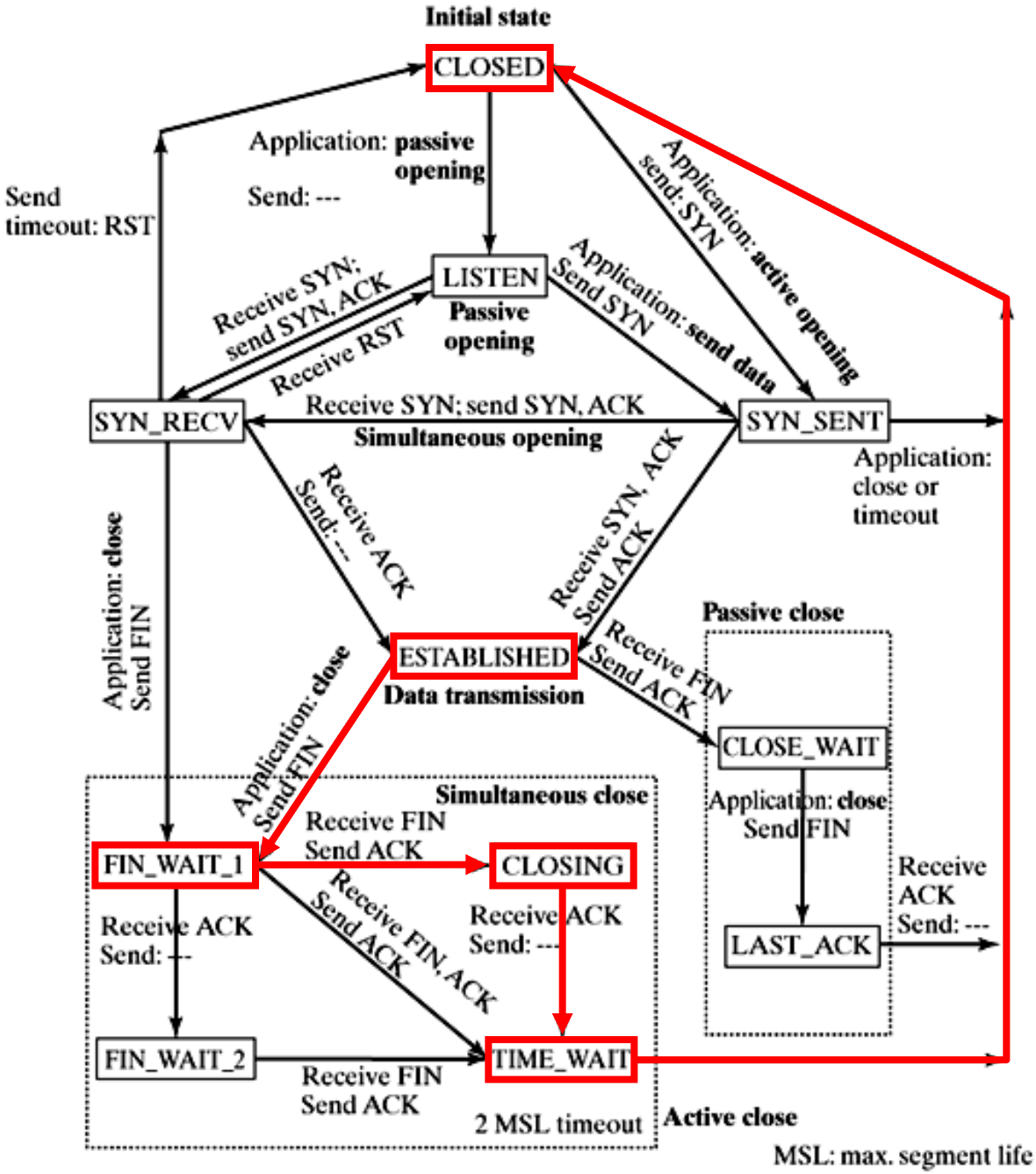

```
tcp_v4_connect()
        |
tcp_connect()
```

```
if (th->ack) {
   (...)
   if (!th->syn)
           goto discard;
   (...)
   tcp_set_state(sk, TCP_ESTABLISHED);
   (...)
   tcp_schedule_ack(tp);
   (...)
}
```



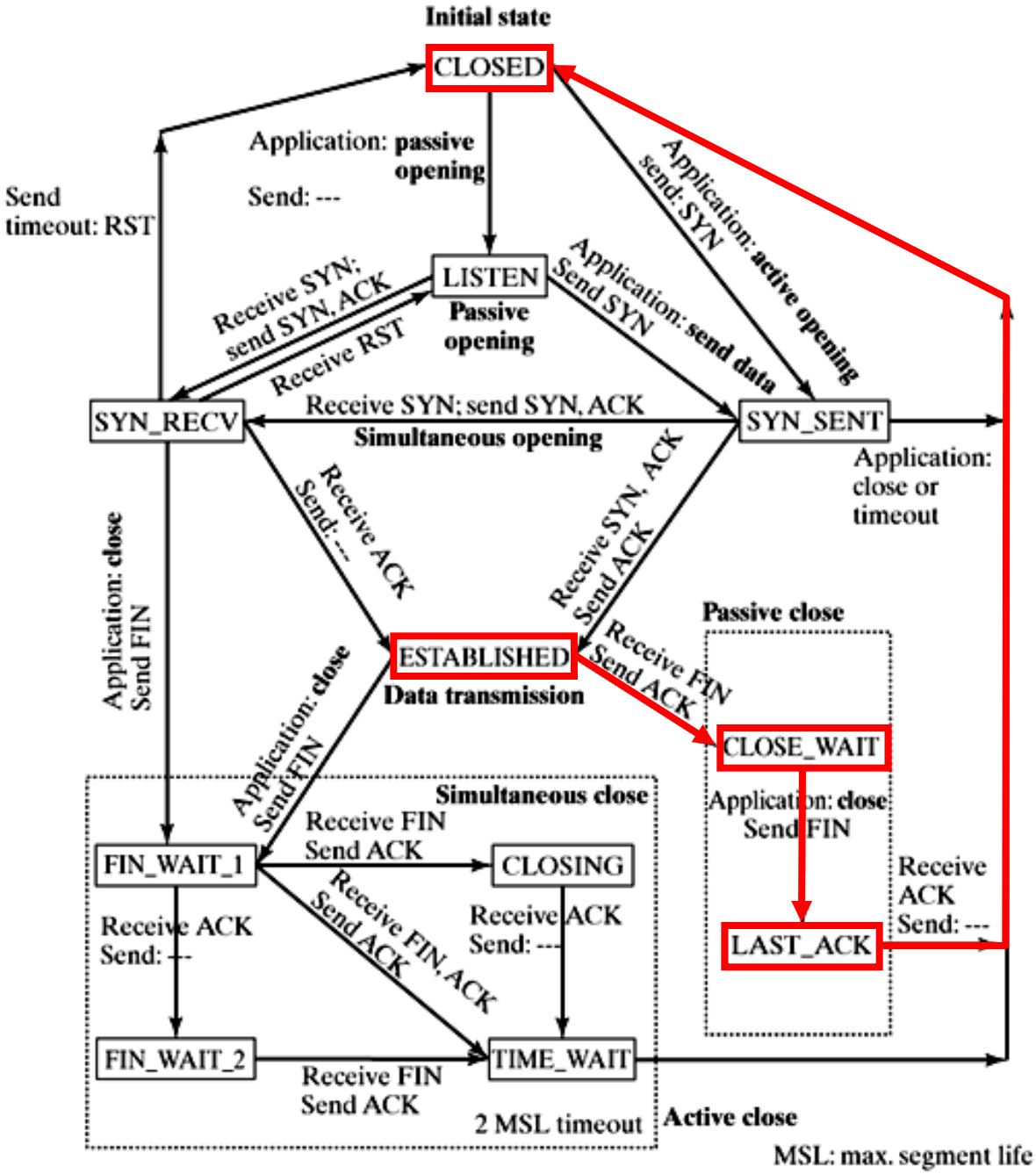

```
tcp_rcv_state_process()
        |
        +-------------------------------+
        |                               |
tcp_v4_hnd_req()                tcp_rcv_state_process()
        |
tcp_check_req()
        |
tcp_v4_syn_recv_sock()
        |
tcp_create_openreq_child()
```

```
if (th->ack) {
     switch(sk->state) {
     case TCP_SYN_RECV:
          (...)
          tcp_set_state(sk, TCP_ESTABLISHED);
    }
}
```



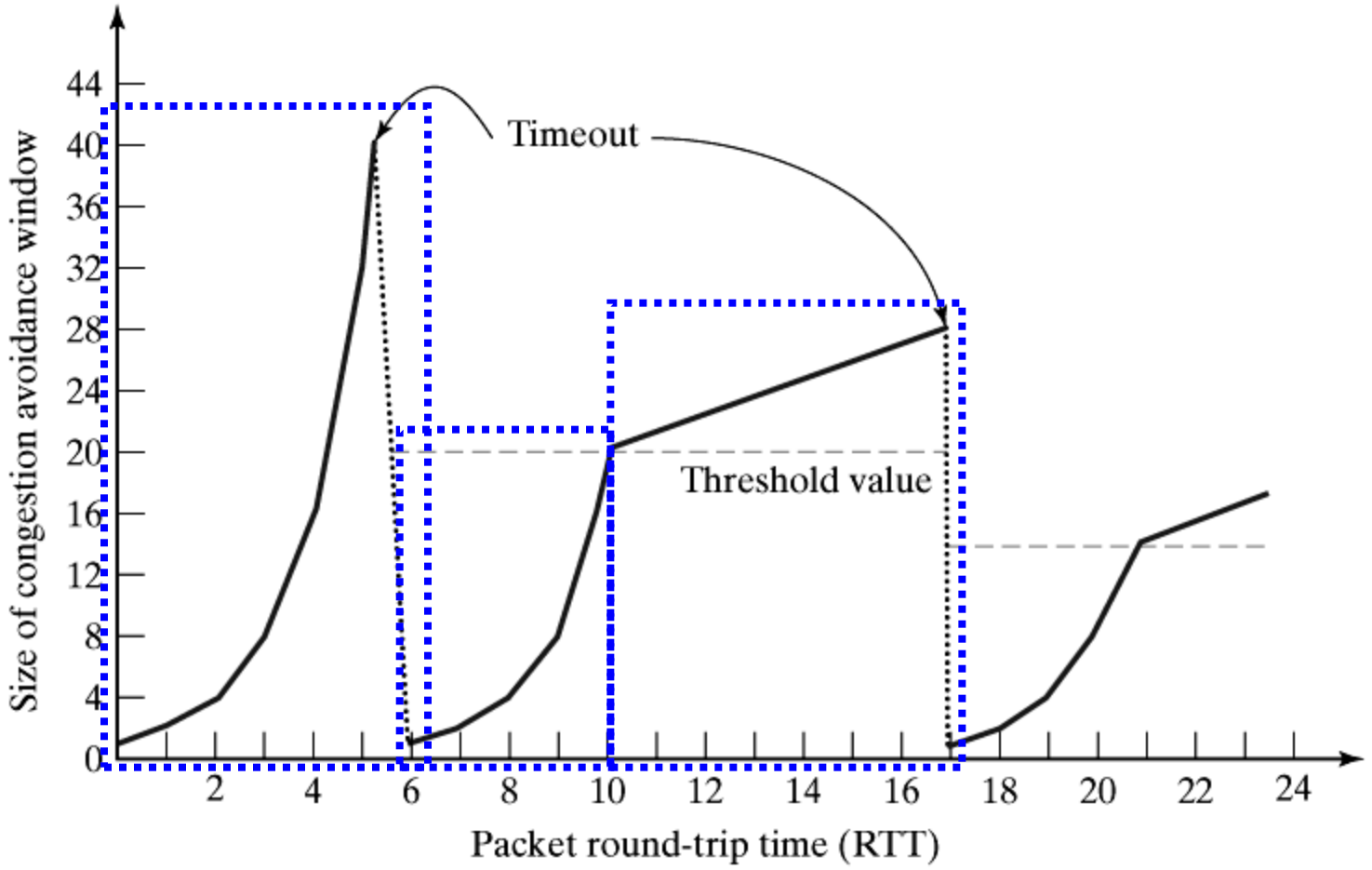

```
switch(sk->state) {
   (...)
   case TCP_FIN_WAIT1:
        tcp_send_ack(sk);
        tcp_set_state(sk, TCP_CLOSING);
   (...)
}
```

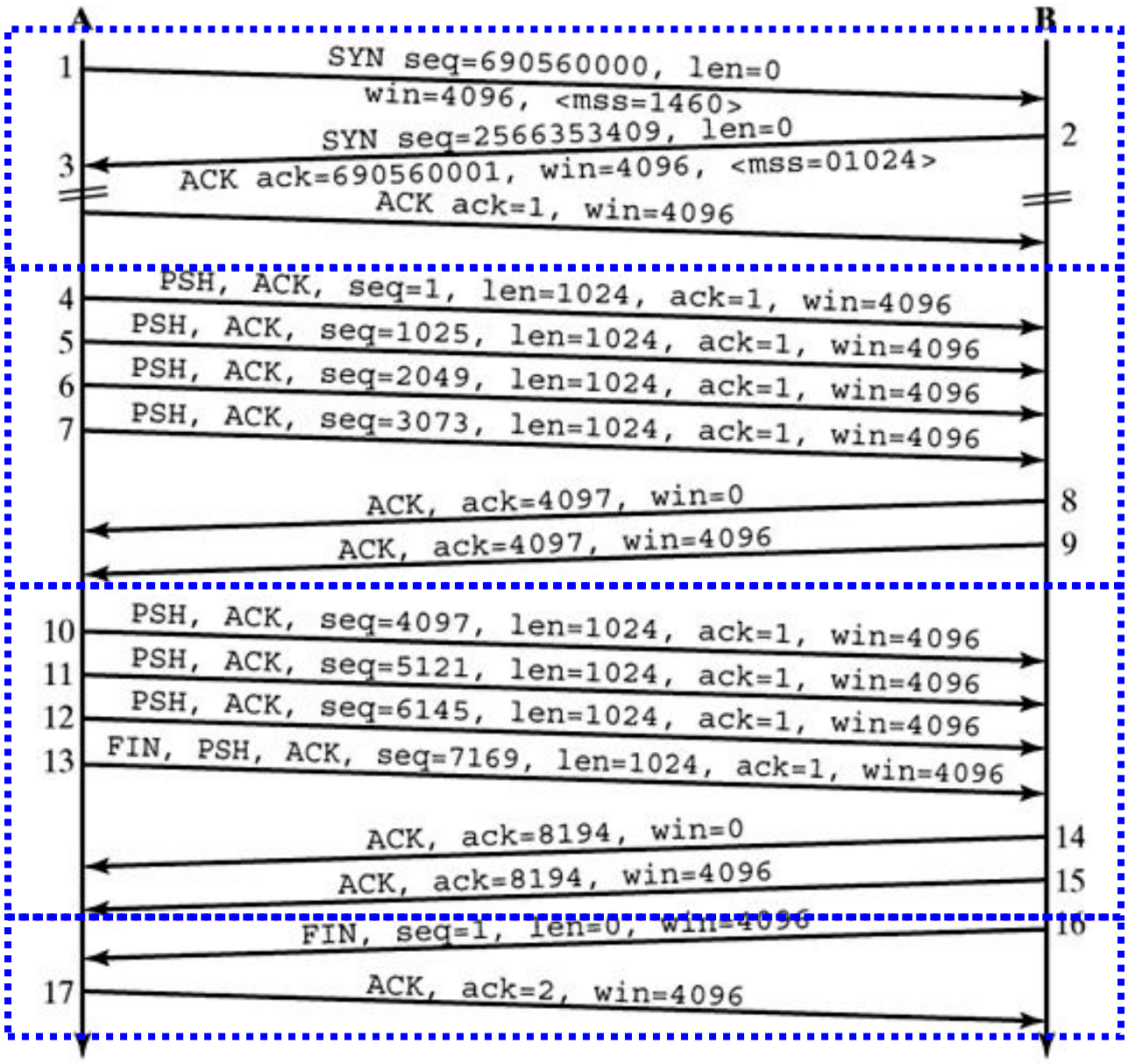
```
if (th->ack) {
    switch(sk->state) {
       (...)
       case TCP_CLOSING:
            (...)
            tcp_time_wait(sk, TCP_TIME_WAIT, 0
            (...)
       (...)
   }
}
```



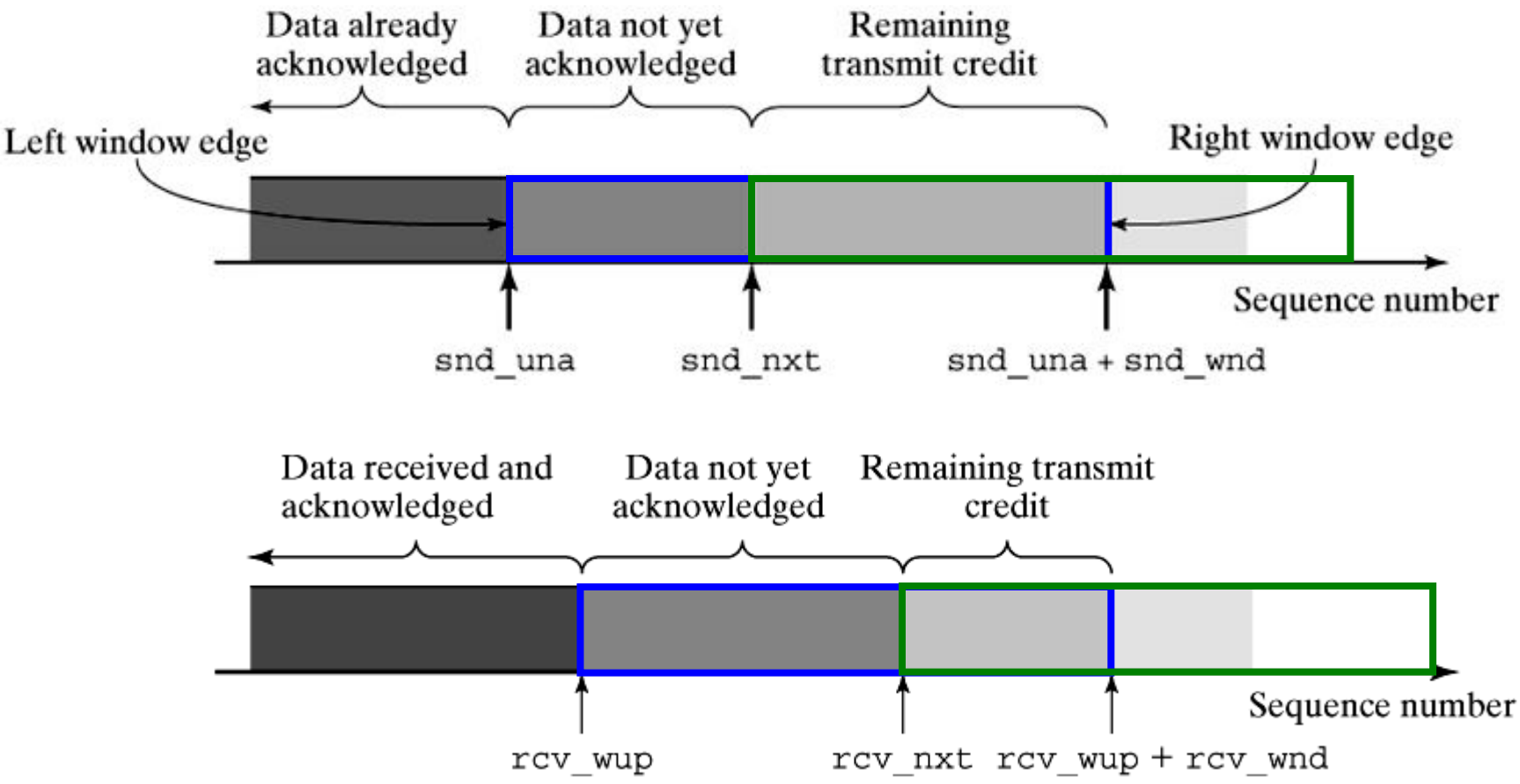

```
switch(sk->state) {
   case TCP_SYN_RECV:
   case TCP_ESTABLISHED:
     /* Move to CLOSE_WAIT */
     tcp_set_state(sk, TCP_CLOSE_WAIT);
       if (th->rst)
          sk->shutdown = SHUTDOWN_MASK;
       break;
   (...)
}


 /* ns: next state (Last ACK) */
  tcp_set_state(sk, ns);
```

Flow Control

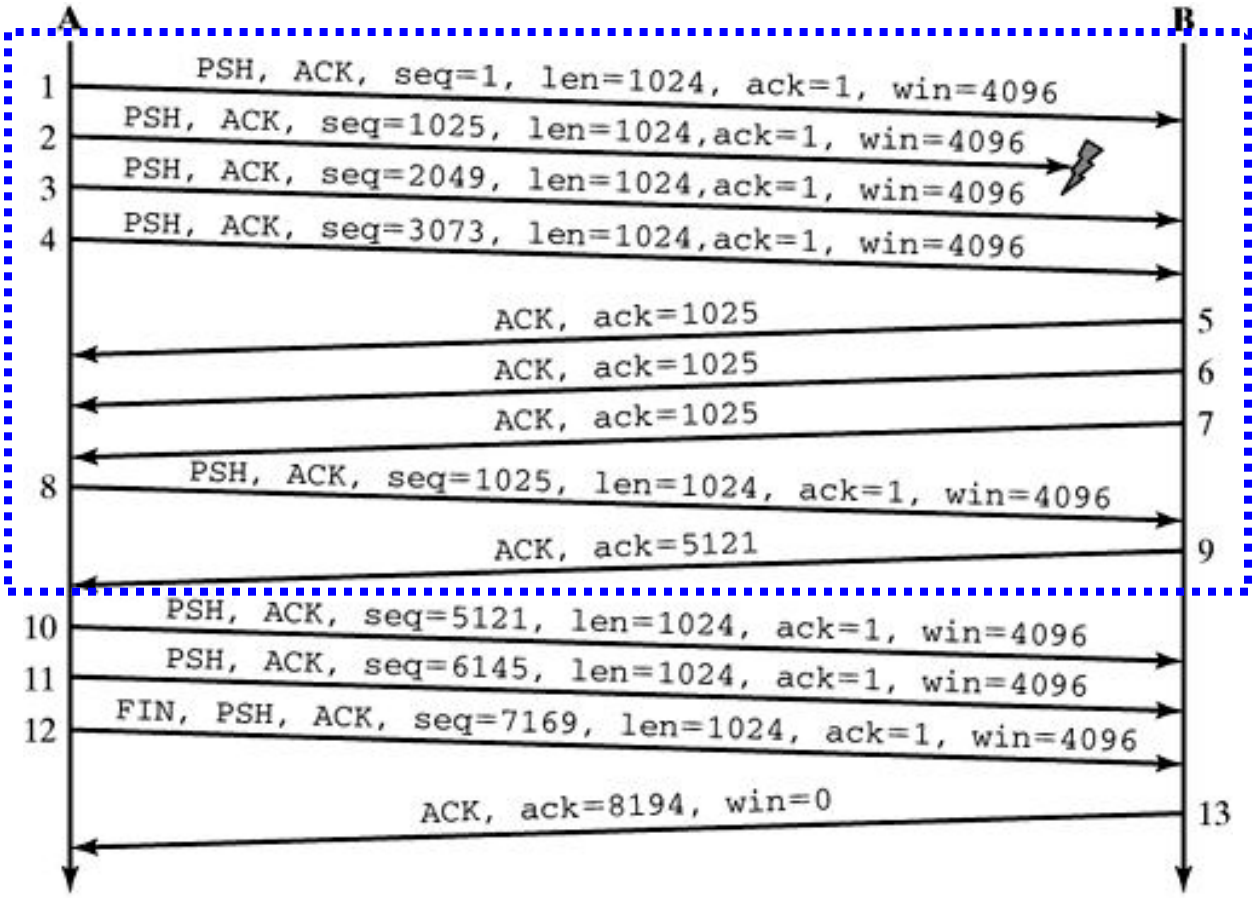



A
B
1
SYN seq=690560000, len=0
win=4096, <mss=1460>
SYN seq=2566353409, len=0
2
3
ACK ack=690560001, win=4096, <mss=01024>
ACK ack=1, win=4096
4
PSH, ACK, seq=1, len=1024, ack=1, win=4096
5
PSH, ACK, seq=1025, len=1024, ack=1, win=4096
6
PSH, ACK, seq=2049, len=1024, ack=1, win=4096
7
PSH, ACK, seq=3073, len=1024, ack=1, win=4096
ACK, ack=4097, win=0
8
ACK, ack=4097, win=4096
9
10
PSH, ACK, seq=4097, len=1024, ack=1, win=4096
11
PSH, ACK, seq=5121, len=1024, ack=1, win=4096
12
PSH, ACK, seq=6145, len=1024, ack=1, win=4096
13
FIN, PSH, ACK, seq=7169, len=1024, ack=1, win=4096
ACK, ack=8194, win=0
14
ACK, ack=8194, win=4096
15
FIN, seq=1, len=0, win=4096
16
17
ACK, ack=2, win=4096

Sliding-Window



Packet Loss & Fast Retransmit



Detecting & Handling Congestions

tcp_retransmit_timer

```
void tcp_enter_loss(struct sock *sk, int how)
{
        struct tcp_opt *tp = &sk->tp_pinfo.af_tcp;
        (...)
        /* Reduce ssthresh if it has not yet been
              made inside this window. */
        if ((tp->ca_state <= TCP_CA_Disorder)
                        || (tp->snd_una == tp->high_seq)
                        || (tp->ca_state == TCP_CA_Loss && !tp->retransmits))
        {
             tp->prior_ssthresh = tcp_current_ssthresh(tp);
             tp->snd_ssthresh = tcp_recalc_ssthresh(tp);
        }
        tp->snd_cwnd = 1;
        tp->snd_cwnd_cnt = 0;
        tp->snd_cwnd_stamp = tcp_time_stamp;
        (...)

static inline __u32 tcp_recalc_ssthresh(struct tcp_opt *tp)
{
        return max(tp->snd_cwnd>>1, 2);
}
```

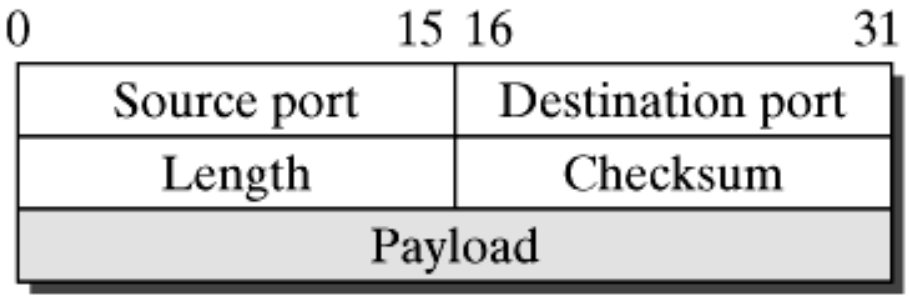
Congestion Avoidance

```
static __inline__ void tcp_cong_avoid(struct tcp_opt *tp)
{
        if (tp->snd_cwnd <= tp->snd_ssthresh)
        { /* In 'safe' area, increase. */
          if (tp->snd_cwnd < tp->snd_cwnd_clamp)
             tp->snd_cwnd++;
        }
        else
        { /* In dangerous area, increase slowly.
           * In theory this is tp->snd_cwnd += 1 / tp->snd_cwnd
           */
        if (tp->snd_cwnd_cnt >= tp->snd_cwnd)
        {
                if (tp->snd_cwnd < tp->snd_cwnd_clamp)
                        tp->snd_cwnd++;
                tp->snd_cwnd_cnt=0;
        }
        else
                tp->snd_cwnd_cnt++;
        }
}
```

User Datagram Protocol

- Same functionality as Internet Protocol (IP)
- Connectionless
- Unreliable service
- Cannot detect and handle lost or duplicate packets
- Transmitting data easily and quickly
- For audio or video streaming

UDP Packet Format



UDP Header & Payload

```
struct udphdr {
        __u16 source;
        __u16 dest;
        __u16 len;
        __u16 check;
};
```

```
struct msghdr {
        void            *msg_name;
        int             msg_namelen;
        struct iovec    *msg_iov;
        __kernel_size_t msg_iovlen;
        void            *msg_control;
        __kernel_size_t msg_controllen;
        unsigned        msg_flags;
};
```

```
struct iovec
{
        void            *iov_base;
        __kernel_size_t iov_len;
};
```

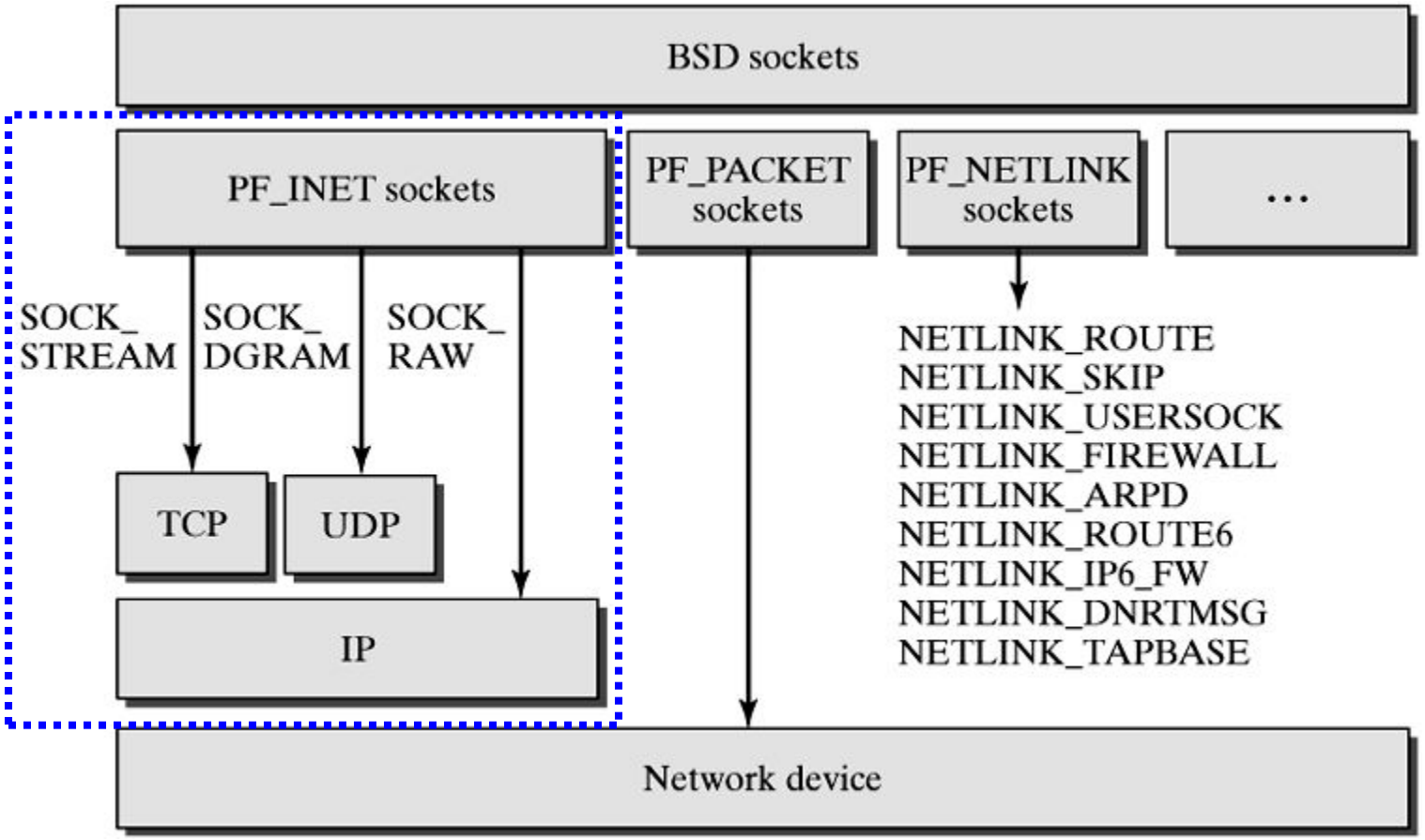
Service Interface to App Layer

```
struct proto udp_prot = {
        name:               "UDP",
        close:              udp_close,
        connect:            udp_connect,
        disconnect:         udp_disconnect,
        ioctl:              udp_ioctl,
        setsockopt:         ip_setsockopt,
        getsockopt:         ip_getsockopt,
        sendmsg:            udp_sendmsg,
        recvmsg:            udp_recvmsg,
        backlog_rcv:        udp_queue_rcv_skb,
        hash:               udp_v4_hash,
        unhash:             udp_v4_unhash,
        get_port:           udp_v4_get_port,
};


static struct inet_protocol udp_protocol = {
        handler:            udp_rcv,
        err_handler:        udp_err,
        next:               IPPROTO_PREVIOUS,
        protocol:           IPPROTO_UDP,
        name:               "UDP"
};
```

Outline

- Architecture of Communication System
- Managing Network Packets
- Network Device
- Data-Link Layer
- Network Layer
- Transport Layer
- Sockets in Linux Kernel
- Socket Programming

Socket Support in Linux Kernel



sys_socketcall()

```
if copy_from_user(a, args, nargs[call]))
  return -EFAULT;
a0=a[0];
a1=a[1];
switch(call) {
                <------- int socket (int family, int type, int protocol)
  case SYS_SOCKET:
    err = sys_socket(a0,a1,a[2]);
    break;
                <------- int bind(int sockfd, struct sockaddr *mAddress, int AddrLength)
  case SYS_BIND:
    err = sys_bind(a0, (struct sockaddr *)a1, a[2]);
    break;
                <------- int connect(int sockfd, struct sockaddr *ServAddr, int AddrLength)
  case SYS_CONNECT:
    err = sys_connect (a0, (struct sockaddr *)a1, a[2]);
    break;
                <------- int listen(int sockfd, int backlog)
  case SYS_LISTEN:
    err = sys_listen (a0,a1);
    break;
}
```

BSD Socket

```
struct socket {
    socket_state          state;
    unsigned long         flags;
    struct proto_ops      *ops;
    struct inode          *inode;
    struct fasync_struct  *fasync_list;
    struct file           *file;
    struct sock           *sk;
    wait_queue_head_t     wait;

    short                 type;
    unsigned char         passcred;
};
```

```
struct proto_ops {
int             family;
int             (*release)    (...);
int             (*bind)       (...);
int             (*connect)    (...);

int             (*socketpair) (...);
int             (*accept)     (...);
int             (*getname)    (...);
unsigned int    (*poll)       (...);
int             (*ioctl)      (...);
int             (*listen)     (...);
int             (*shutdown)   (...);
int             (*setsockopt) (...);
int             (*getsockopt) (...);
int             (*sendmsg)    (...);
int             (*recvmsg)    (...);
int             (*mmap)       (...);
ssize_t         (*sendpage)   (...);
};
```

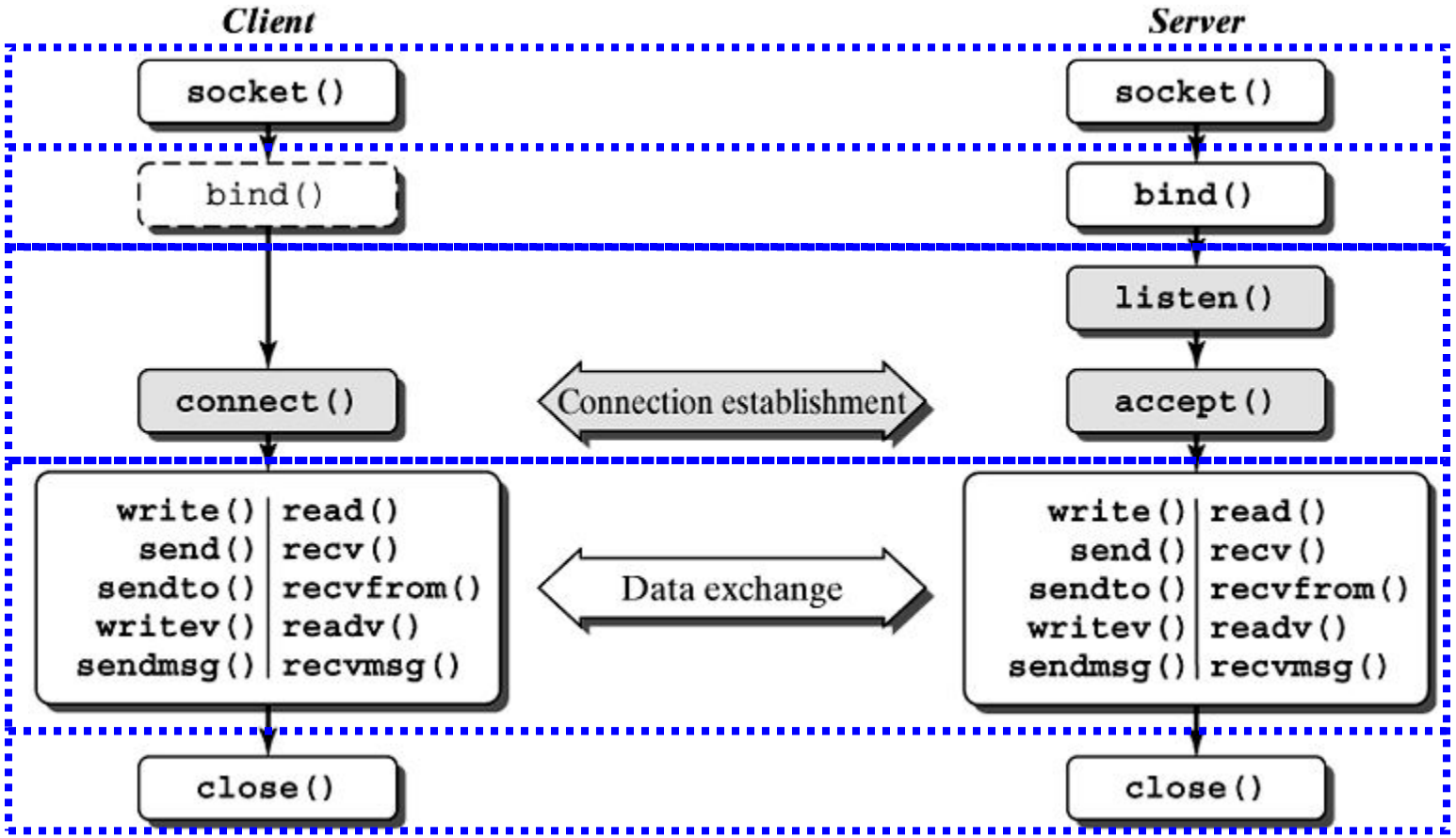
PF_INET Socket

```
struct proto {
void            (*close)            (...);
int             (*connect)          (...);
int             (*disconnect)       (...);
struct sock*    (*accept)           (...);
int             (*ioctl)            (...);
int             (*init)             (...);
int             (*destroy)          (...);
void            (*shutdown)         (...);
int             (*setsockopt)       (...);
int             (*getsockopt)       (...);
int             (*sendmsg)          (...);
int             (*recvmsg)          (...);
int             (*bind)             (...);
int             (*backlog_rcv)      (...);
void            (*hash)             (...);
void            (*unhash)           (...);
int             (*get_port)         (...);
char            name [32];
                struct {
                  int inuse;
                  u8 __pad[SMP_CACHE_BYTES - sizeof(int)];
                  } stats[NR_CPUS];
};
```

Outline

- Architecture of Communication System
- Managing Network Packets
- Network Device
- Data-Link Layer
- Network Layer
- Transport Layer
- Sockets in Linux Kernel
- Socket Programming

System Calls at Socket Interface



```c
#include <sys/types.h>
#include <sys/socket.h>

#define SERVER_TCP_PORT 2001

int sockfd;
struct sockaddr_in serv_addr;

/* Initialize address area. */
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(SERVER_TCP_PORT);

/*  Since a sockaddr_in structure was used above for the address,
    it has to be transformed to the more general sockaddr
    before using it as second parameter of the bind call. */
bind(sockfd, (struct sockaddr*) &serv_addr, sizeof(serv_addr));
```

```c
struct sockaddr_in   {
   sa_family_t             sin_family; /* Address family: AF_INET */
   unsigned short int      sin_port;   /* Port number */
   struct in_addr          sin_addr;   /* Internet address */

   /* Pad to size of 'struct sockaddr' . */
   unsigned char sin_zero[sizeof (struct sockaddr) -
                          sizeof (sa_family_t) -
                          sizeof (uint16_t) -
                          sizeof (struct in_addr)];
};
```

```
#include <sys/socket.h>

/* Create a socket and bind it to an address. */
listen(sockfd, 5);


#include      <sys/types.h>
#include      <sys/socket.h>

int newsockfd, clilen;
struct sockaddr_in cli_addr;
clilen = sizeof(cli_addr);

/* socket, bind, listen, ... */

newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
if (newsockfd < 0)
       printf("ERROR: Creating new socket");

if (childpid = fork()) < 0)
        printf("ERROR: Creating child process");
else if (childpid == 0)
{
             /* Child process */
        close(sockfd);              /* Parent socket */
        doit(newsockfd);            /* Handle client request */
        exit(0);
}
close(newsockfd);                   /* Parent closes the new socket */
```

```
#include <sys/types.h>
#include <sys/socket.h>

#define SERVER_TCP_PORT 2001
#define SERVER_HOST_ADDR "129.13.35.77"

struct sockaddr_in serv_addr;

/* A sockfd was already created ... */
/* Initialize data space. */
memset(&serv_addr, 0, sizeof(serv_addr));

/* Enter address family in address structure. */
serv_addr.sin_family = AF_INET;

/* Set IP address. */
serv_addr.sin_addr.s_addr = inet_addr(SERVER_HOST_ADDR);

/* Set port number. */
serv_addr.sin_port = htons(SERVER_TCP_PORT);

/* Establish connection to server. */
connect(sockfd, (struct sockaddr*) &serv_addr, sizeof(serv_addr));

/* Transmit data ... */
```

Data Transmission

- Transmitting data

```
size_t write(sockfd, buffer, length)
int send(sockfd, buffer, length, flags)
int sendto(sockfd, buffer, length, flags, destaddr, addrlen)
```

- Receiving data

```
size_t read(sockfd, buffer, length)
int recv(sockfd, buffer, length, flags)
int recvfrom (sockfd, buffer, length, flags, fromaddr,
  addrlen)
```

- Rransmit and receive an array of iovec structures

```
int readv(int sockfd, const struct iovec *vector, size_t
  count)
int writev(int sockfd, const struct iovec *vector, size_t
  count)
int sendmsg(int sockfd, const struct msghdr *msg, int flags)
int recvmsg(int sockfd, struct msghdr *msg, int flags)
```

Q & A

- Architecture of Communication System
- Managing Network Packets
- Network Device
- Data-Link Layer
- Network Layer
- Transport Layer
- Sockets in Linux Kernel
- Socket Programming