# Navigation of TCP/IP files in Linux

Divye Kapoor
Pracheer Agarwal
Swagat Konchada

## Background Information

# Linux Virtual Filesystem (VFS)

- It is the software layer in the kernel that provides a uniform filesystem interface to userspace programs
- It provides an abstraction within the kernel that allows for transparent working with a variety of filesystems.
- Thus it allows many different filesystem implementations to coexist freely
- Each socket is implemented as a "file" mounted on the sockfs filesystem.
    - file->private points to the socket information.

## Inodes and File Structures

- Inodes provide a method to access the actual data blocks allocated to a file. For sockets, they provide buffer space which can be used to hold socket specific data.
  - struct inode

- Every file is represented in the kernel as an object of the *file* structure. It requires an inode provided to it.
  - struct file

## Structure of Function Pointers

```
Struct operations {
        int (*read)(int, char *, int);
        void (*destroy_inode)(inode *);
        void (*dirty_inode) (struct inode *);
        int (*write_inode) (struct inode *, int);
        void (*drop_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
};
Sizeof(operations) = sizeof(function ptr)*6
```

Divye Kapoor

# Walkthrough of Sending

User Space
Socket, bind, listen, connect, send, recv, write, read etc.

Socket Functions (Kernel)
sys_socket, sys_bind, sys_listen, sys_connect etc. in socket.c

TCP/IP Layer Functions
inet_create, tcp_v4_connect, tcp_sendmsg, tcp_recvmsg

Ethernet Device Layer
dev_hard_start_xmit

# Socket(family, type, proto)

Sys_socket()

Sock_create()

Allocate a socket object
(internally an inode
Associated with a file object)

Locate the family requested and call the create function for that family

Inet_create()
Lower layer initialization

Sock_map_fd()

Sock_alloc_fd()
Allocate a file descriptor

Sock_attach_fd()

Fd_install()

# Sys_connect(fd, sockaddr *, len)

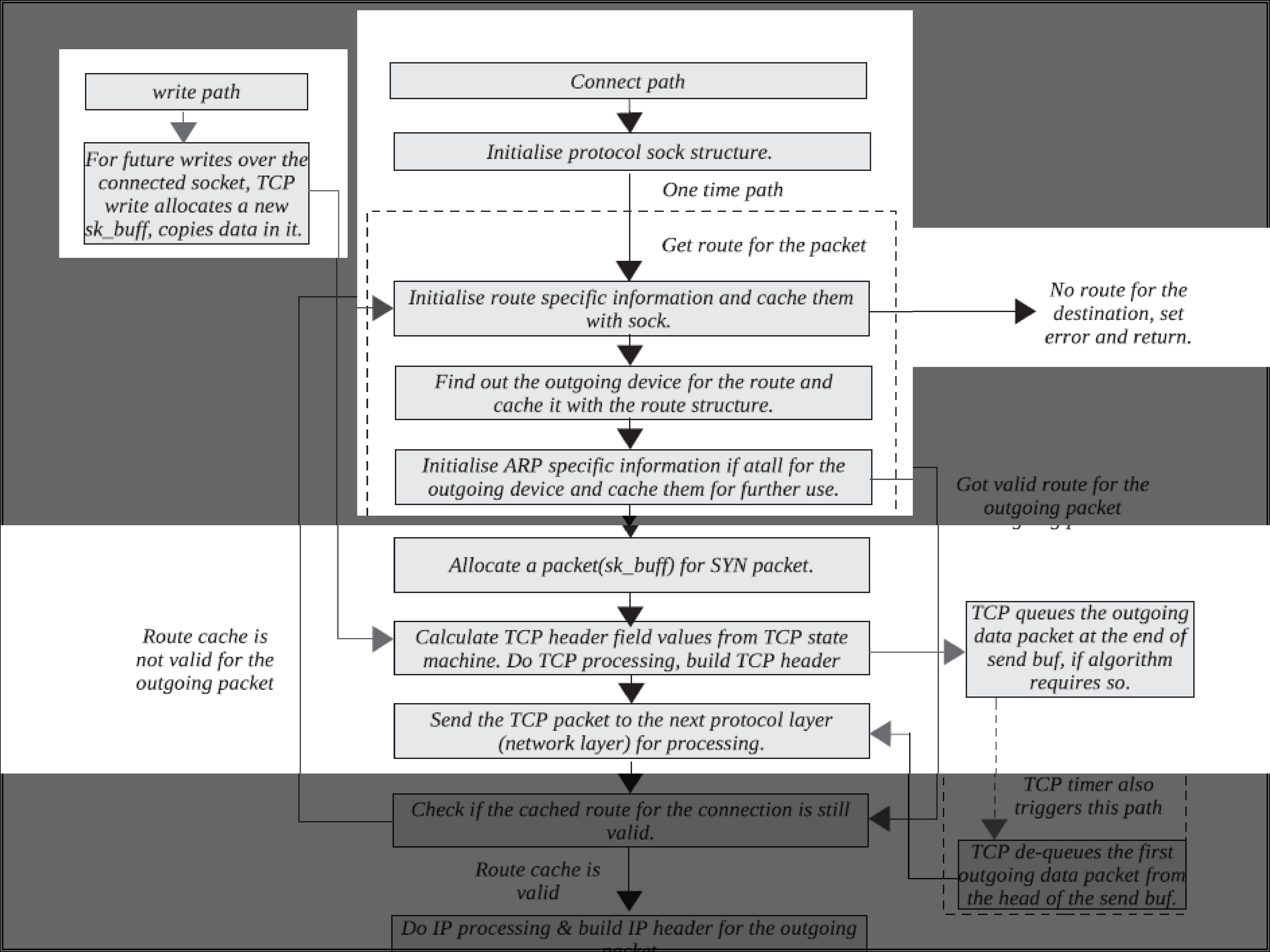
Sys_connect()

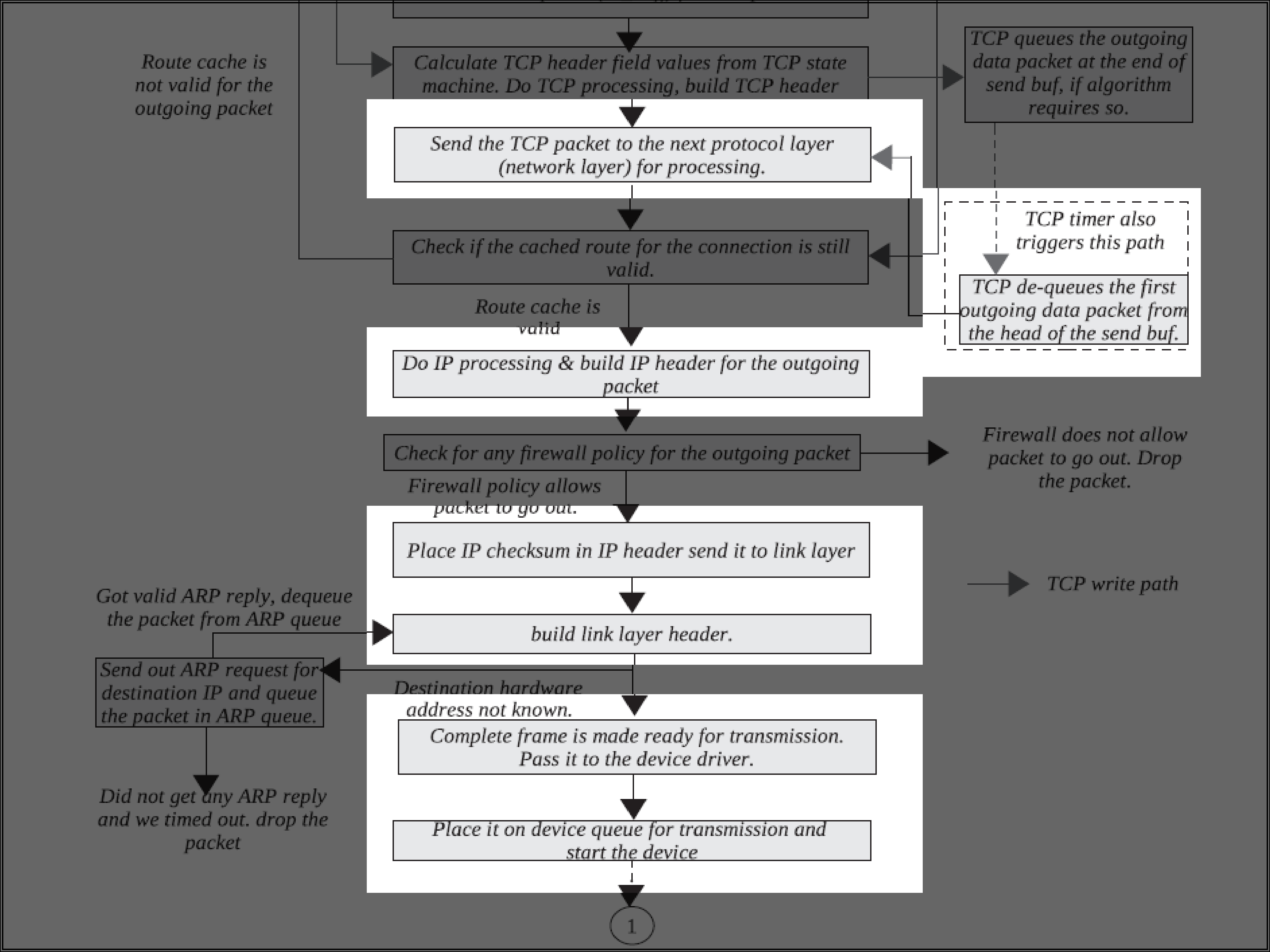
Sockfd_lookup_light()
Returns the socket object associated with the given fd

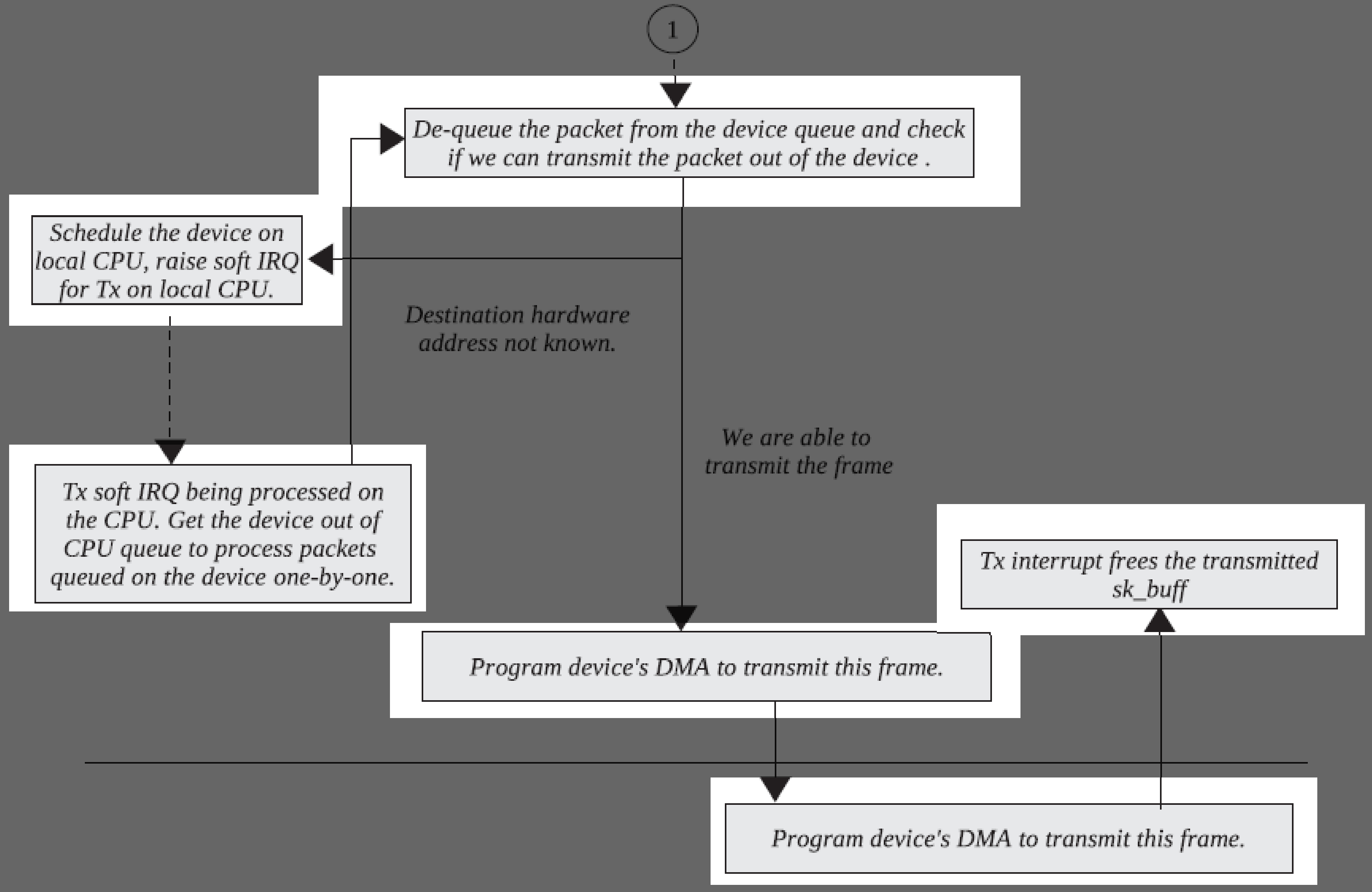
Move_addr_to_kernel()
For userspace sockaddr *

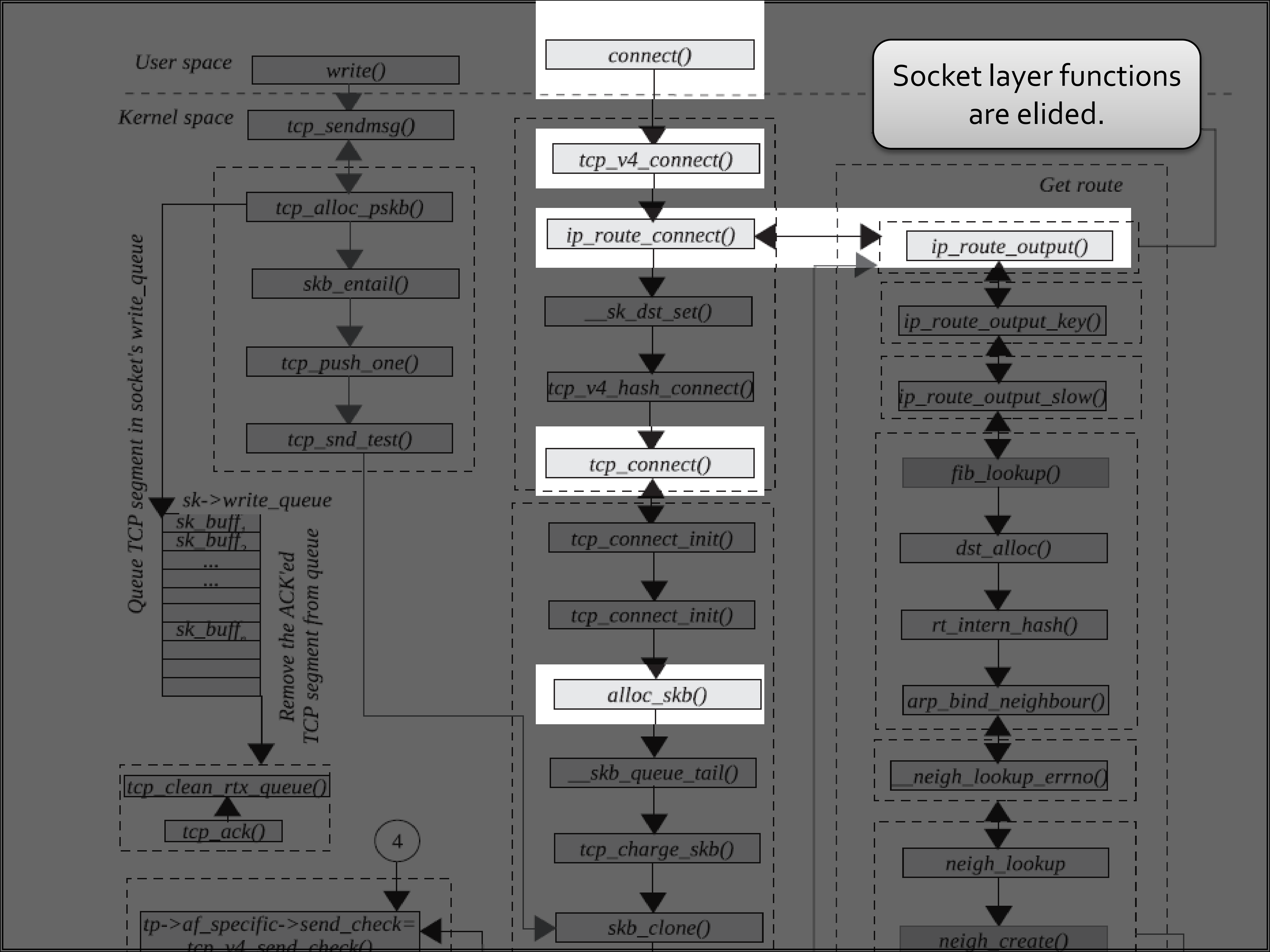
Sock->ops->connect()
Lower layer call

Tcp_v4_connect()


write path
For future writes over the connected socket, TCP write allocates a new sk_buff, copies data in it.
Connect path
Initialise protocol sock structure.
One time path
Get route for the packet
Initialise route specific information and cache them with sock.
No route for the destination, set error and return.
Find out the outgoing device for the route and cache it with the route structure.
Initialise ARP specific information if atall for the outgoing device and cache them for further use.
Got valid route for the outgoing packet
Allocate a packet(sk_buff) for SYN packet.
Route cache is not valid for the outgoing packet
Calculate TCP header field values from TCP state machine. Do TCP processing, build TCP header
TCP queues the outgoing data packet at the end of send buf, if algorithm requires so.
Send the TCP packet to the next protocol layer (network layer) for processing.
TCP timer also triggers this path
Check if the cached route for the connection is still valid.
TCP de-queues the first outgoing data packet from the head of the send buf.
Route cache is valid
Do IP processing & build IP header for the outgoing


Route cache is not valid for the outgoing packet
Calculate TCP header field values from TCP state machine. Do TCP processing, build TCP header
TCP queues the outgoing data packet at the end of send buf, if algorithm requires so.
Send the TCP packet to the next protocol layer (network layer) for processing.
TCP timer also triggers this path
Check if the cached route for the connection is still valid.
TCP de-queues the first outgoing data packet from the head of the send buf.
Route cache is valid
Do IP processing & build IP header for the outgoing packet
Check for any firewall policy for the outgoing packet
Firewall does not allow packet to go out. Drop the packet.
Firewall policy allows packet to go out.
Place IP checksum in IP header send it to link layer
TCP write path
Got valid ARP reply, dequeue the packet from ARP queue
build link layer header.
Send out ARP request for destination IP and queue the packet in ARP queue.
Destination hardware address not known.
Complete frame is made ready for transmission. Pass it to the device driver.
Did not get any ARP reply and we timed out. drop the packet
Place it on device queue for transmission and start the device
1


1
De-queue the packet from the device queue and check if we can transmit the packet out of the device .
Schedule the device on local CPU, raise soft IRQ for Tx on local CPU.
Destination hardware address not known.
We are able to transmit the frame
Tx soft IRQ being processed on the CPU. Get the device out of CPU queue to process packets queued on the device one-by-one.
Tx interrupt frees the transmitted sk_buff
Program device's DMA to transmit this frame.
Program device's DMA to transmit this frame.



Socket layer functions are elided.

## struct sk_buff

## struct sk_buff

Defined in *<include/linux/skbuff.h>*

- used by every network layer (except the physical layer)
- fields of the structure change as it is passed from one layer to another
- i.e., fields are layer dependent.

## Networking options

```
struct sk_buff {
   … … …
#ifdef CONFIG_NET_SCHED
   _ _u32    tc_index;
#ifdef CONFIG_NET_CLS_ACT
   _ _u32    tc_verd;
   _ _u32    tc_classid;
#endif
#endif
}
```

sk_buff is peppered with C preprocessor #ifdef directives.

CONFIG_NET_SCHED symbol should be defined at compile time for the structure to have the element tc_index.

enabled with some version of *make config* by an administrator.

# sk_buff list

- The kernel maintains all sk_buff structures in a doubly linked list.

```
struct sk_buff_head {/* only the head of the list */
    /* These two members must be first. */
    struct sk_buff    * next;
    struct sk_buff    * prev;

    _ _u32       qlen;
    spinlock_t    lock;/* atomicity in accessing a sk_buff list. */
};
```

## Element classification

- Layout
- General
- Feature-specific
- Management functions

## Layout

- **struct sock * sk**

**sock data structure of the socket that owns this buffer**

- unsigned int len

includes both the data in the main buffer (i.e., the one pointed to by head) and the data in the fragments

- unsigned int data_len

unlike len, data_len accounts only for the size of the data in the fragments.

- unsigned int truesize

skb->truesize = size + sizeof(struct sk_buff);

- atomic_t users

reference count, or the number of entities using this sk_buff buffer
atomic_inc and atomic_dec

## Layout

- struct sock * sk

sock data structure of the socket that owns this buffer

- **unsigned int len**

**includes both the data in the main buffer (i.e., the one pointed to by head) and the data in the fragments**

- unsigned int data_len

unlike len, data_len accounts only for the size of the data in the fragments.

- unsigned int truesize

skb->truesize = size + sizeof(struct sk_buff);

- atomic_t users

reference count, or the number of entities using this sk_buff buffer
atomic_inc and atomic_dec

## Layout

- struct  sock  * sk

 sock data structure of the socket that owns this buffer

- unsigned int len

includes both the data in the main buffer (i.e., the one pointed to by head) and the data in the fragments

- **unsigned int data_len**

**unlike len, data_len accounts only for the size of the data in the fragments.**

- unsigned int truesize

skb->truesize = size + sizeof(struct sk_buff);

- atomic_t users

reference count, or the number of entities using this sk_buff buffer
atomic_inc and atomic_dec

## Layout

- struct sock * sk

sock data structure of the socket that owns this buffer

- unsigned int len

includes both the data in the main buffer (i.e., the one pointed to by head) and the data in the fragments

- unsigned int data_len

unlike len, data_len accounts only for the size of the data in the fragments.

- **unsigned int truesize**

**skb->truesize = size + sizeof(struct sk_buff);**

- atomic_t users

reference count, or the number of entities using this sk_buff buffer
atomic_inc and atomic_dec

# Layout

- struct sock * sk

sock data structure of the socket that owns this buffer

- unsigned int len

includes both the data in the main buffer (i.e., the one pointed to by head) and the data in the fragments

- unsigned int data_len

unlike len, data_len accounts only for the size of the data in the fragments.

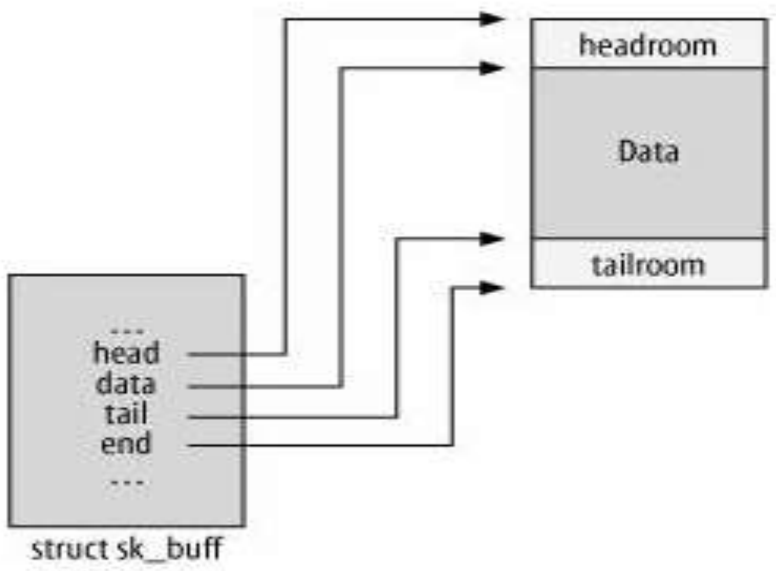
- unsigned int truesize

skb->truesize = size + sizeof(struct sk_buff);

- **atomic_t users**

**reference count, or the number of entities using this sk_buff buffer**
**atomic_inc() and atomic_dec()**

# position pointers



- unsigned char *head
- sk_buff_data_t end
- unsigned char *data
- sk_buff_data_t tail

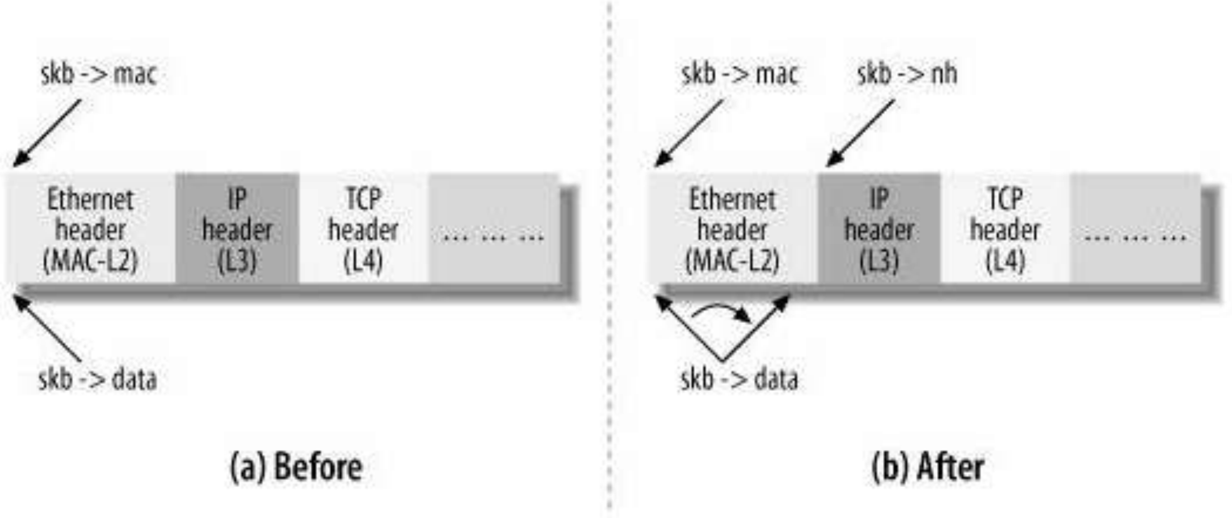
# sk_buf->dev

struct net_device *dev

- represents the receiving interface or the to be transmitted device(or interface) corresponding to the packet.
- usually represents the virtual device's(representation of all devices grouped) net_device structure.

- Pointers to protocol headers.
- sk_buff_data_t transport_header;
- sk_buff_data_t network_header;
- sk_buff_data_t mac_header;

# pointer modifications



updation of data is done using the *_header pointers

## Control block

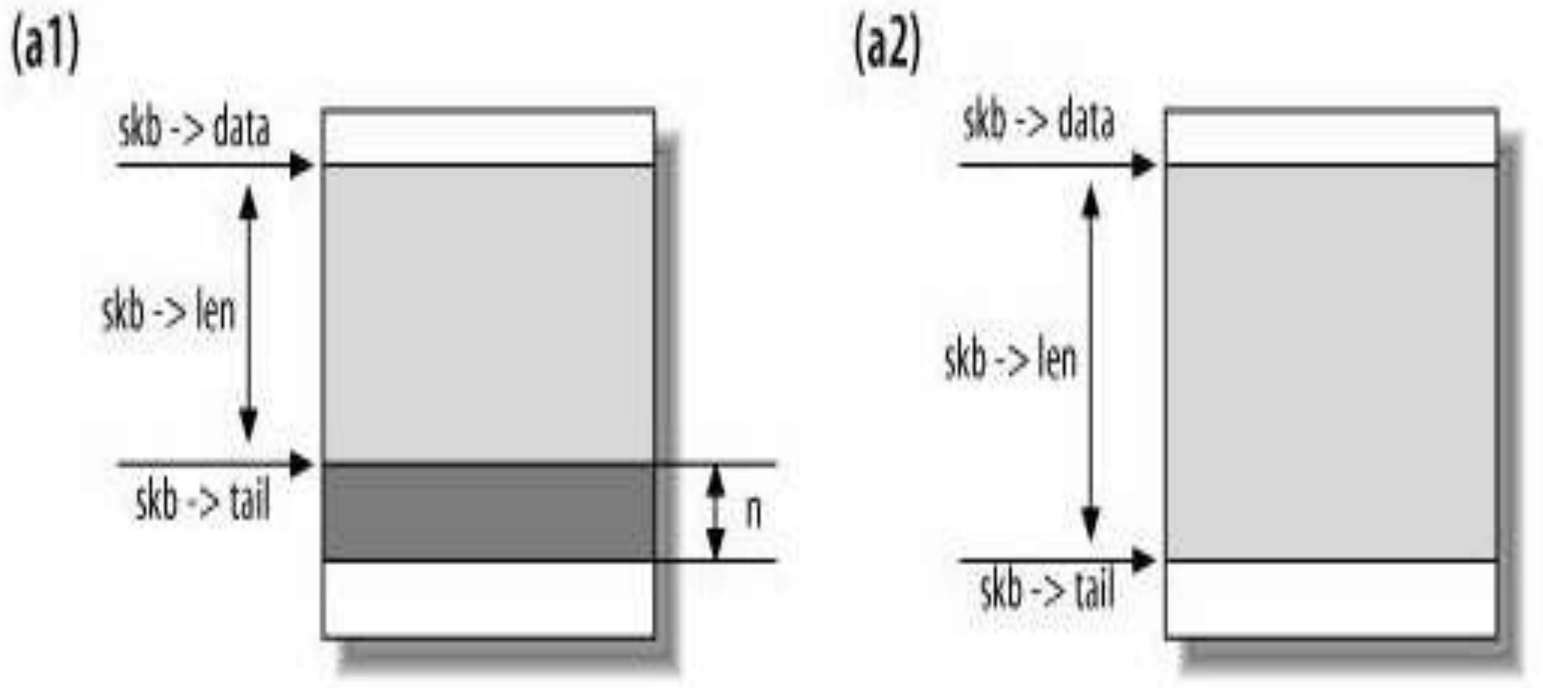
- char cb[40]
- This is a "control buffer," or storage for private information, maintained by each layer for internal use.

```
struct tcp_skb_cb {
 … … … _ _u32 seq; /* Starting sequence number */
_ _u32 end_seq; /* SEQ + FIN + SYN + datalen*/
_ _u32 when; /* used to compute rtt's */
_ _u8 flags; /* TCP header flags. */
… … …
};
```

## management functions

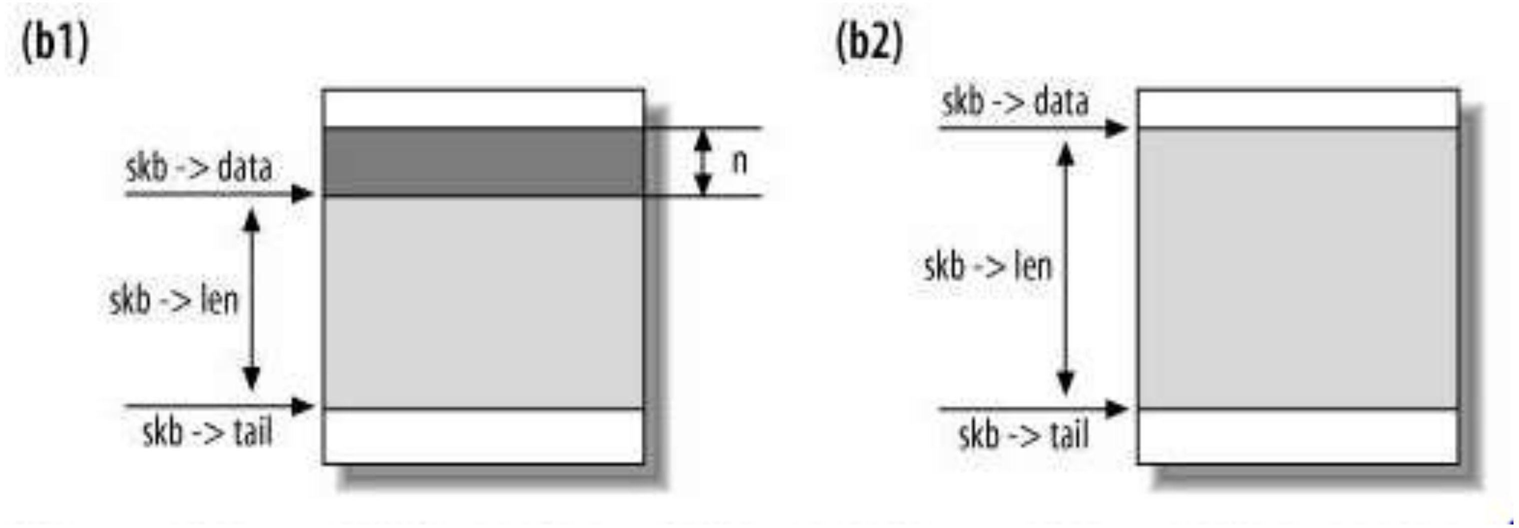
*Defined in <include/linux/skbuff.h>   &   <net/core/skbuff.c>*

skb_put(struct sk_buff *, usingned int len)



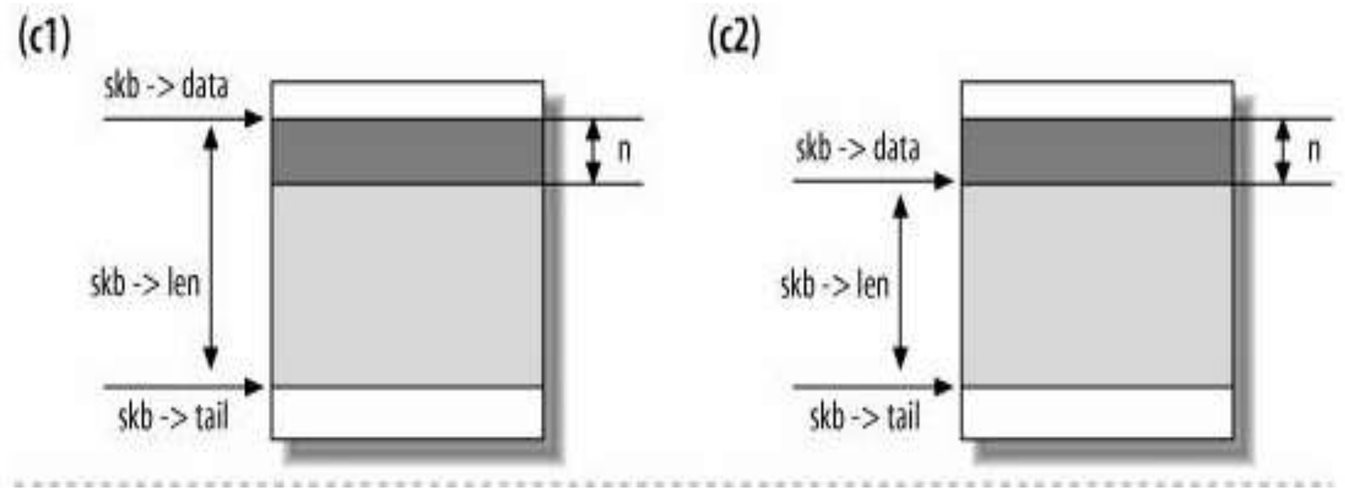
## management functions

skb_push(struct sk_buff *skb, unsigned int len)



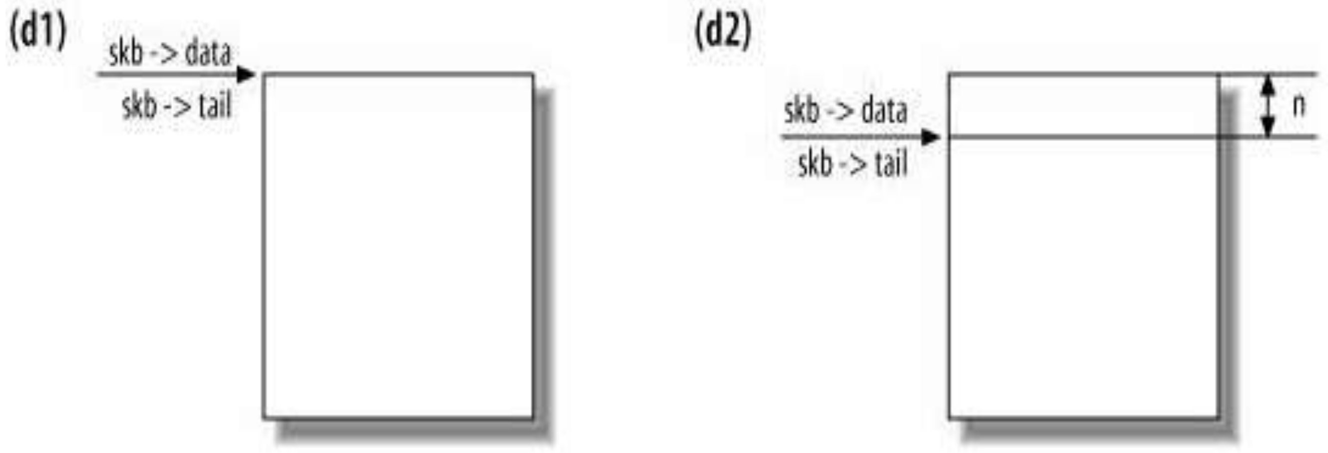
## management functions

skb_pull(struct sk_buff *skb, unsigned int len)



# management functions

skb_reserve(struct sk_buff *skb, int len)

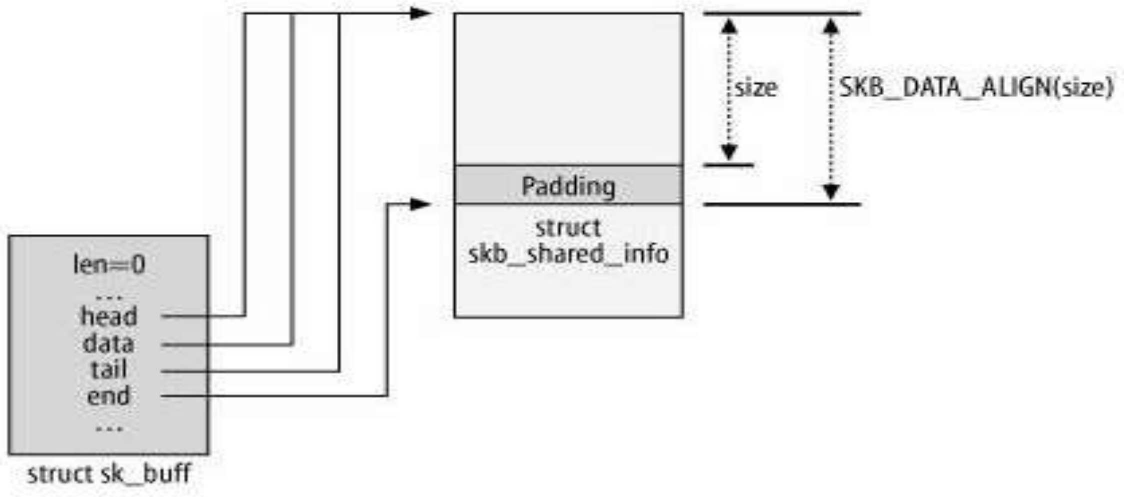


Each of the above four memory management functions return the data ptr.

# memory allocation

defined in <*net/core/skbuff.c*>

```
struct sk_buff *__alloc_skb(unsigned int size, gfp_t gfp_mask,
                            int fclone, int node)
...
size = SKB_DATA_ALIGN(size);
data = kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);
...
```



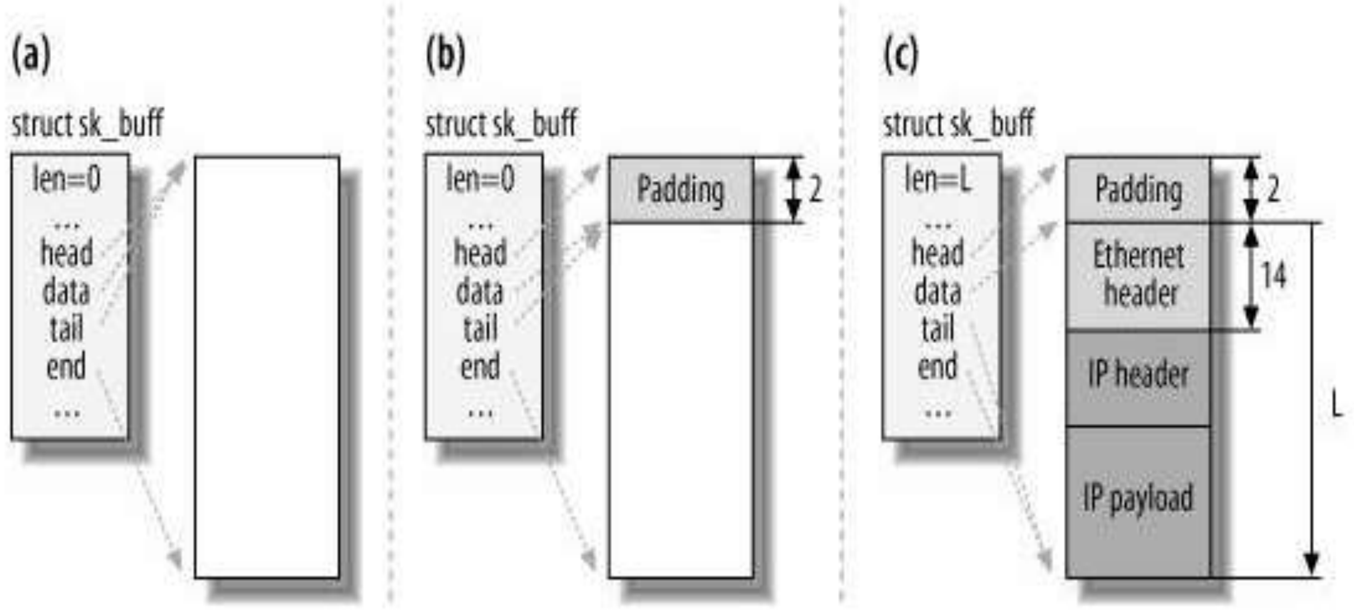
## memory allocation

struct sk_buff *__netdev_alloc_skb(struct net_device *dev,
unsigned int length, gfp_t gfp_mask)
The buffer allocation function meant for use by device drivers
Executed in interrupt mode

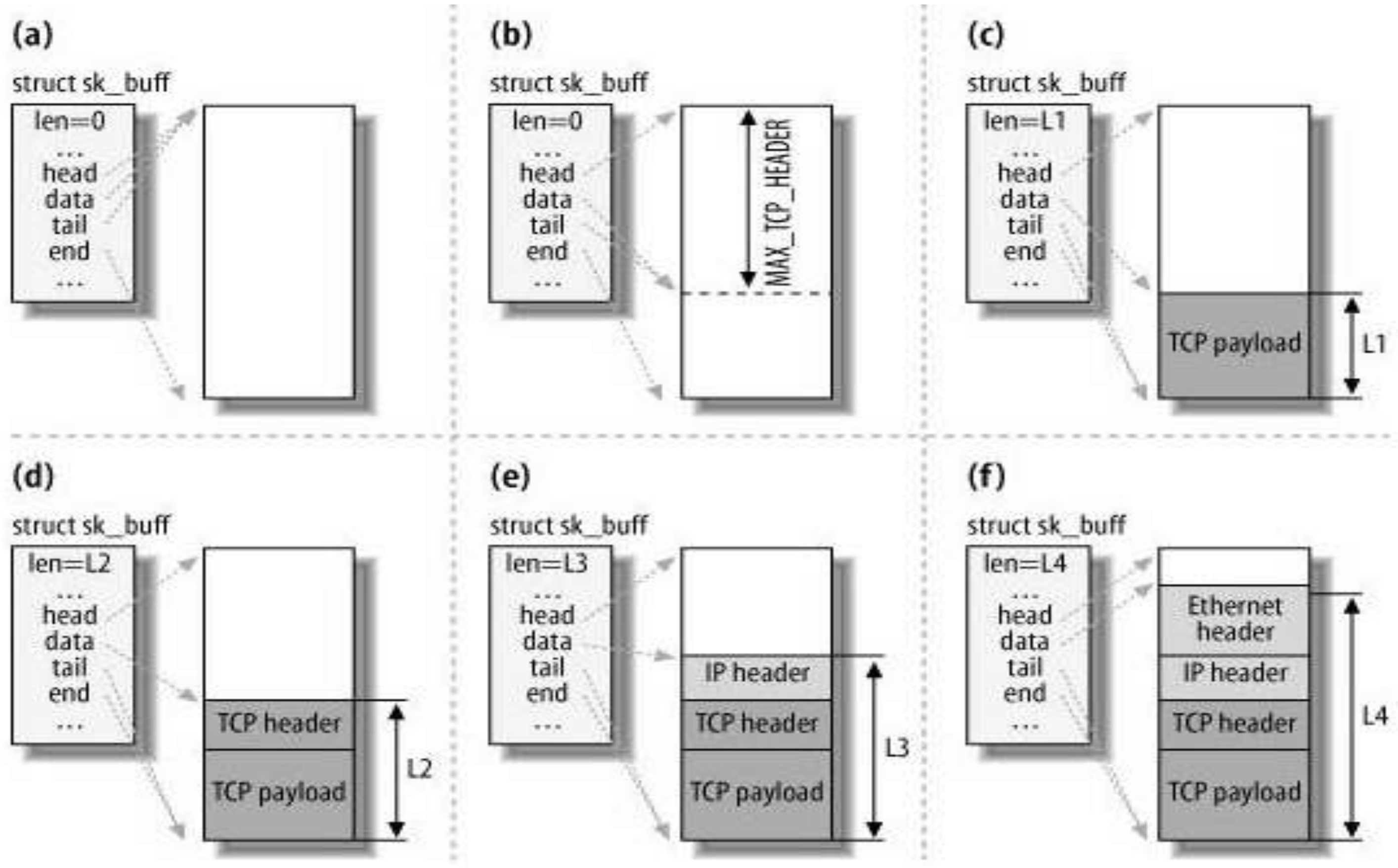
**Freeing memory: kfree_skb and dev_kfree_skb**

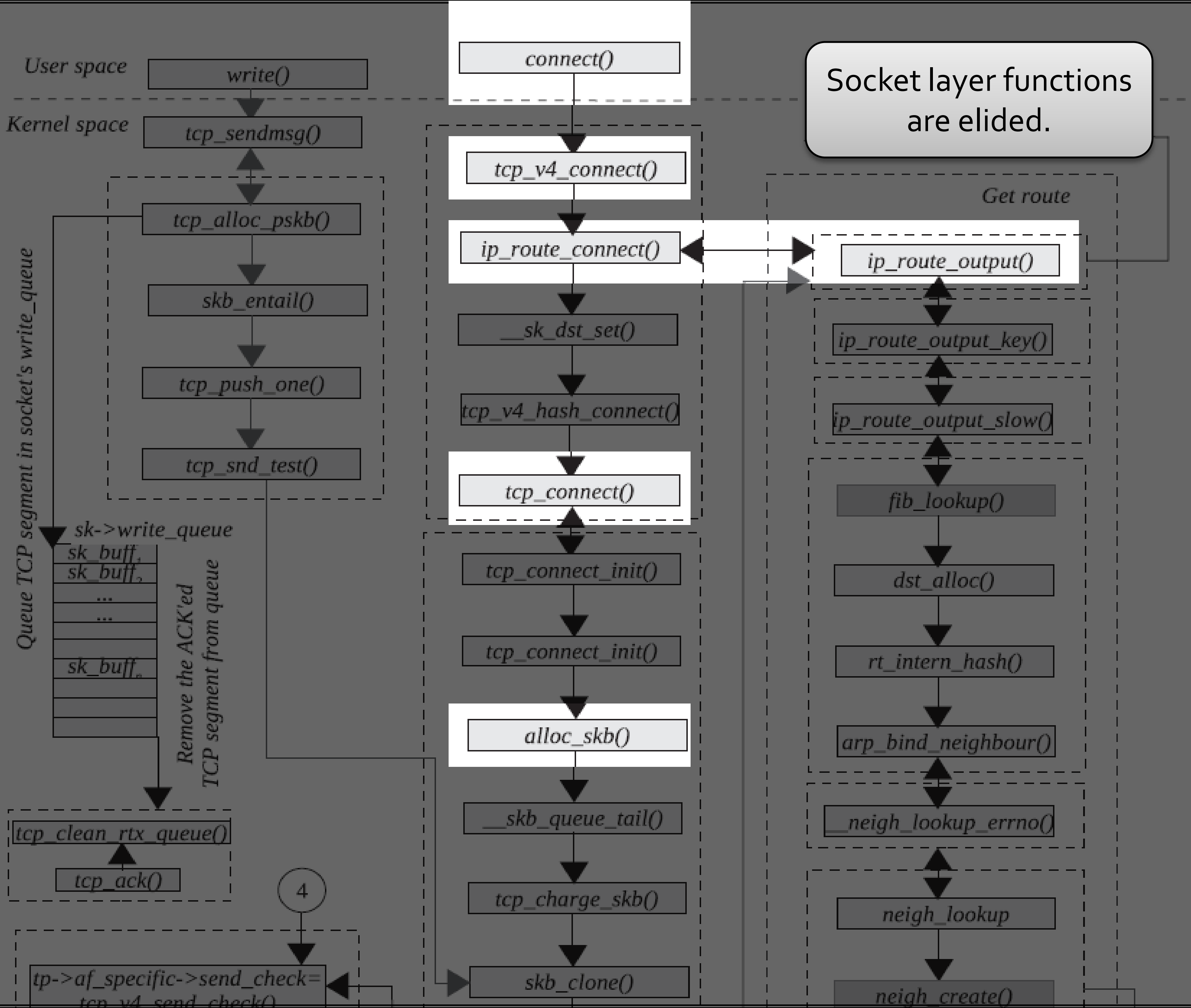
Release buffer back to the buffer-pool.
Buffer released only when skb_users counter is 1. If not, the counter is decremented.

## initializing buffer - reception

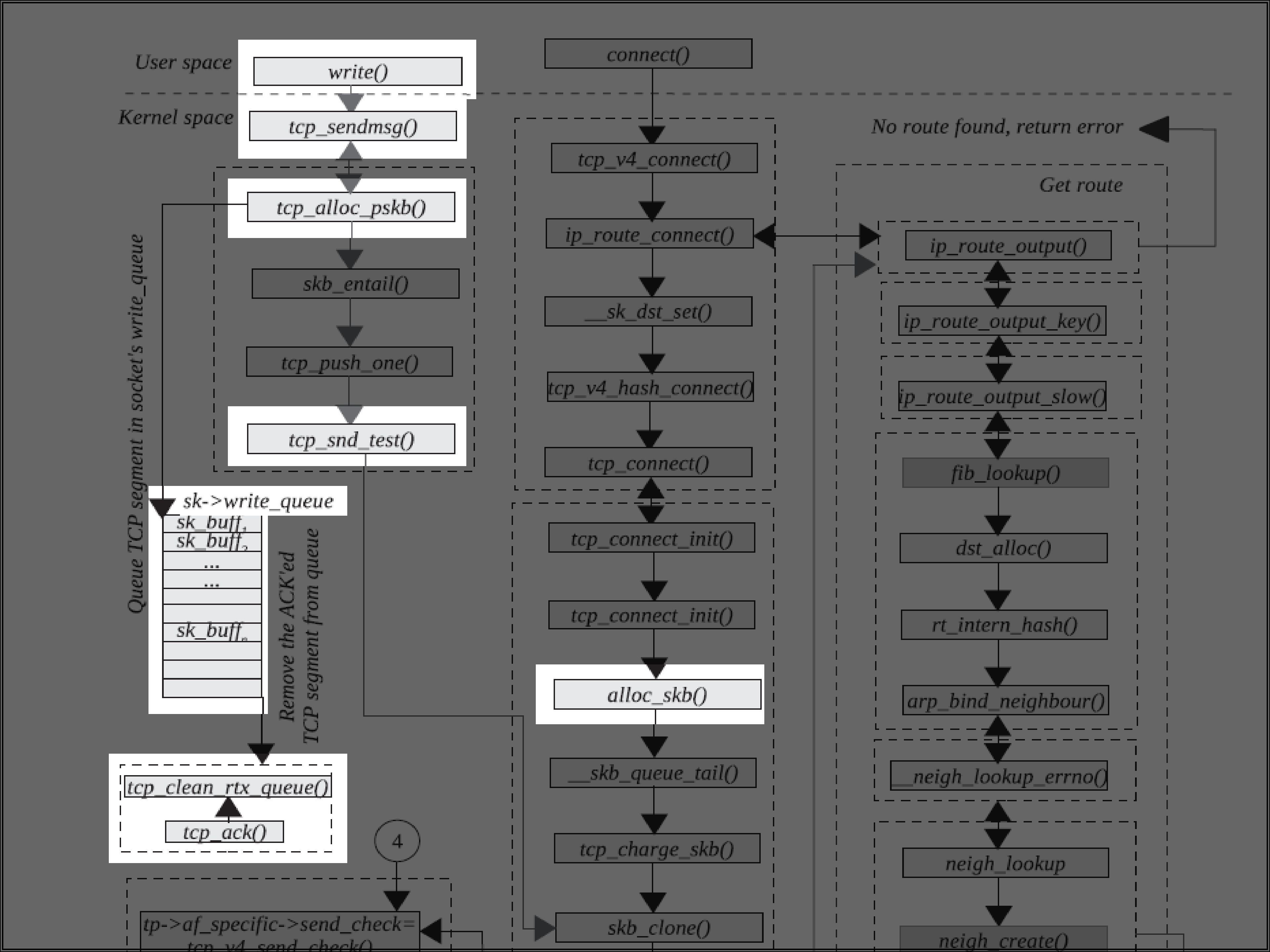


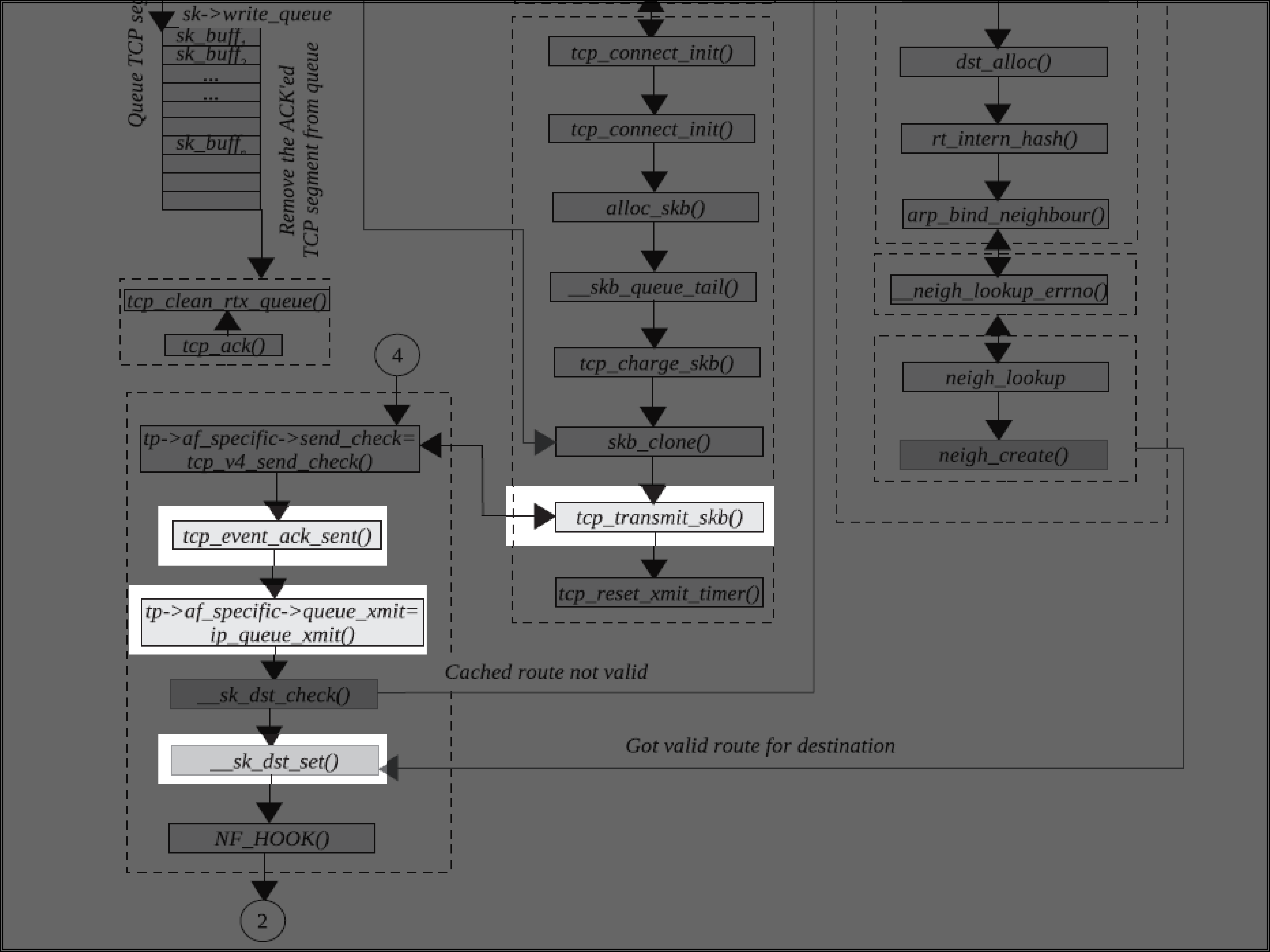
# initializing buffer - transmission

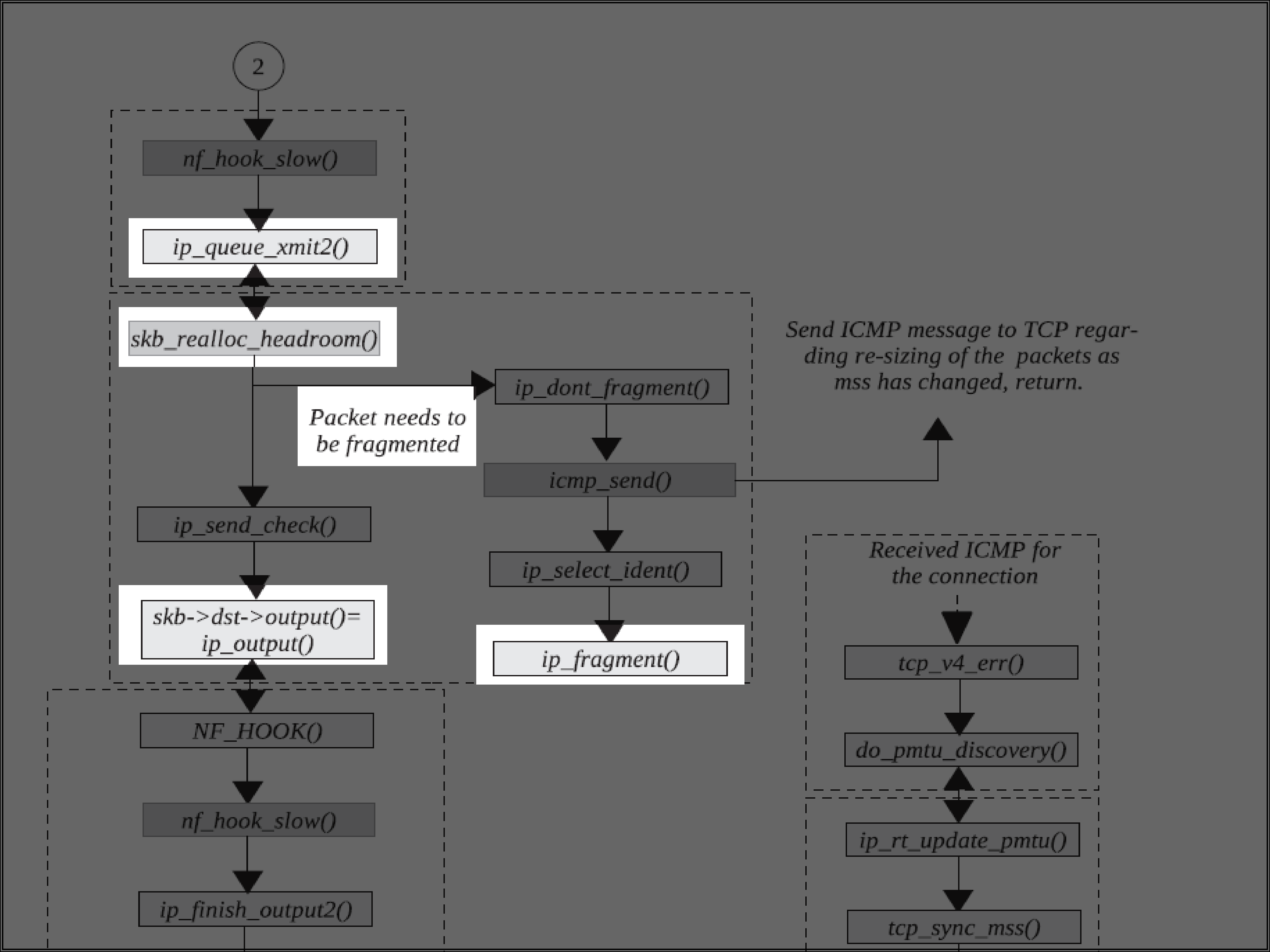


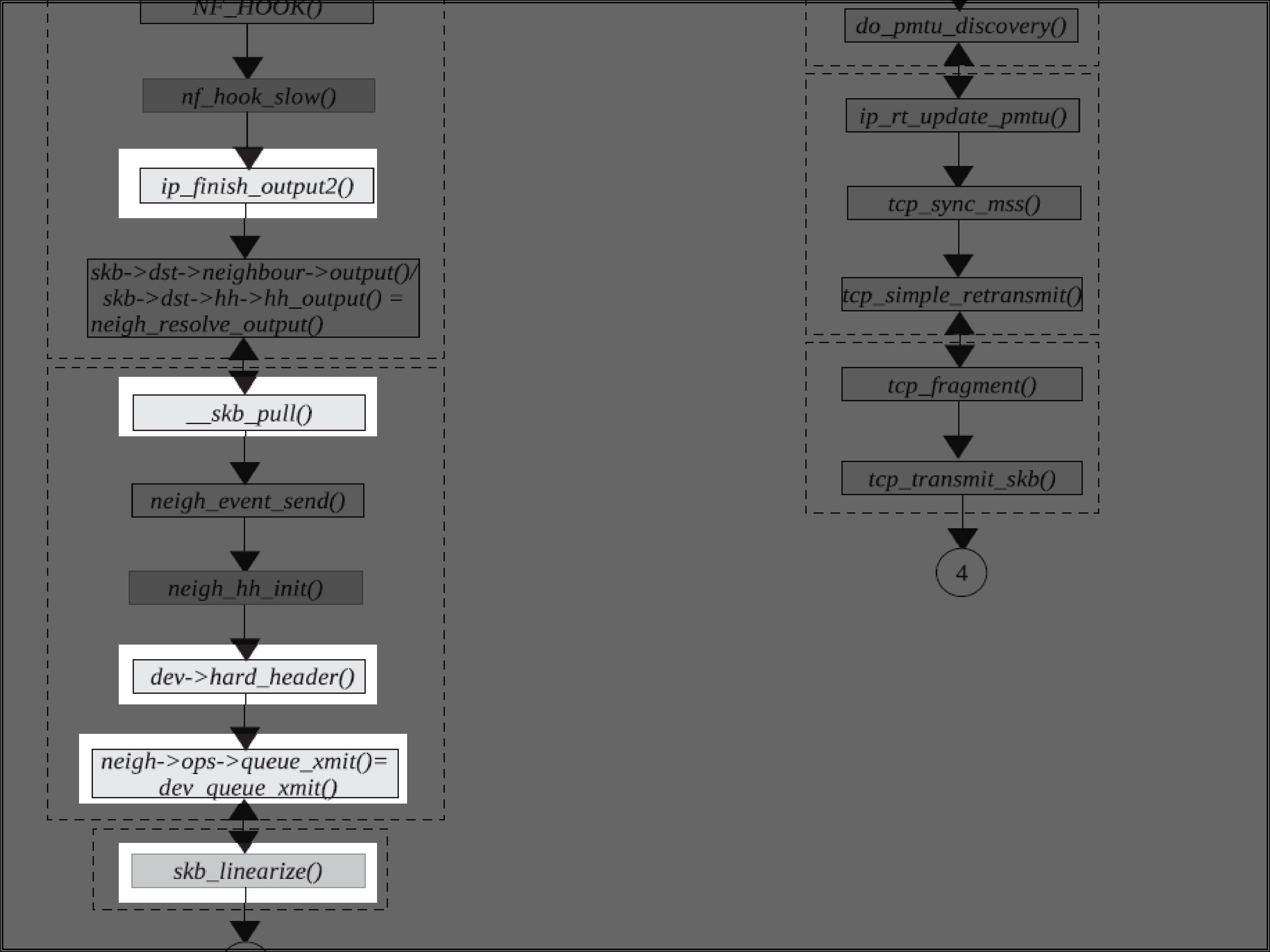


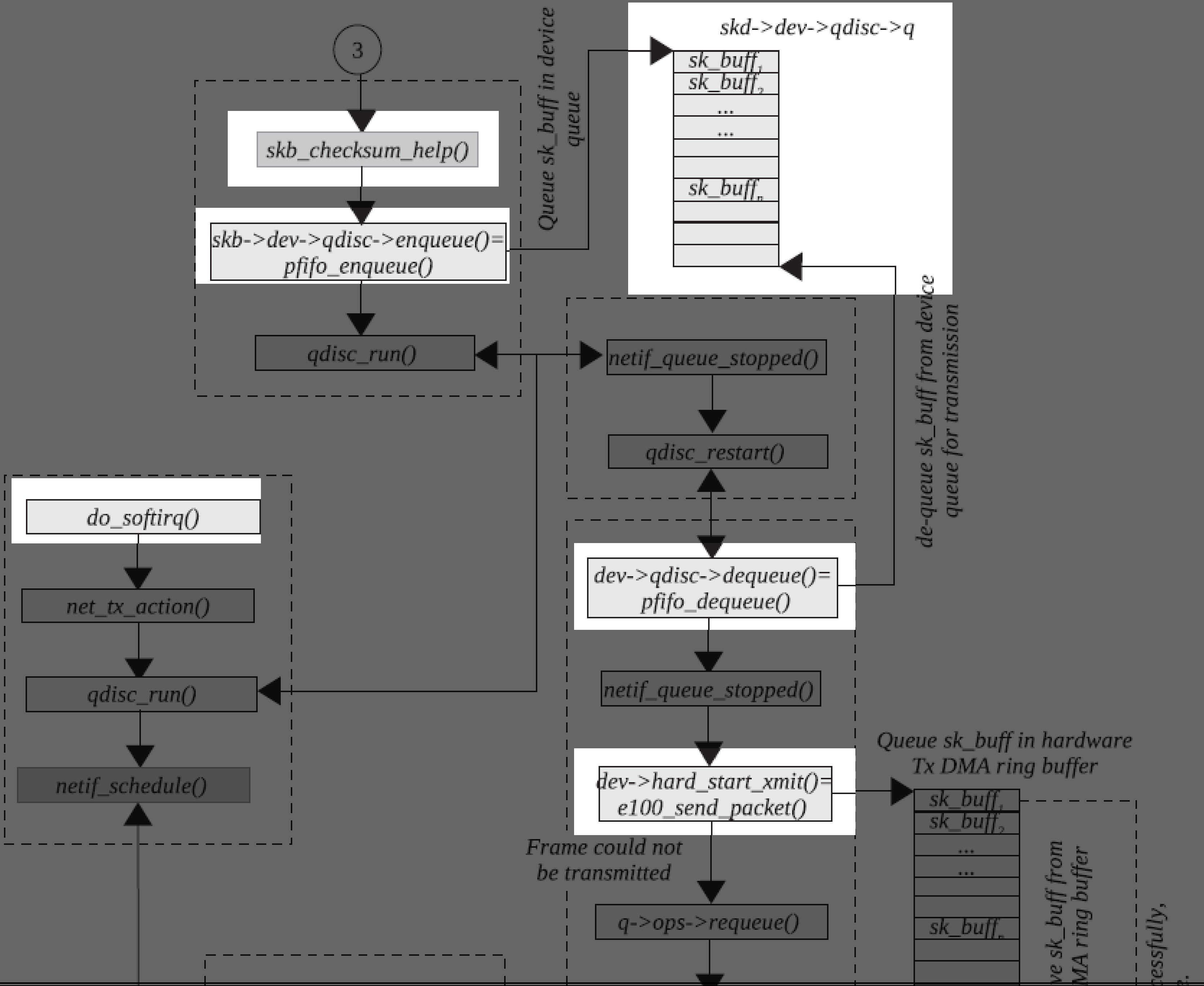
Socket layer functions are elided.

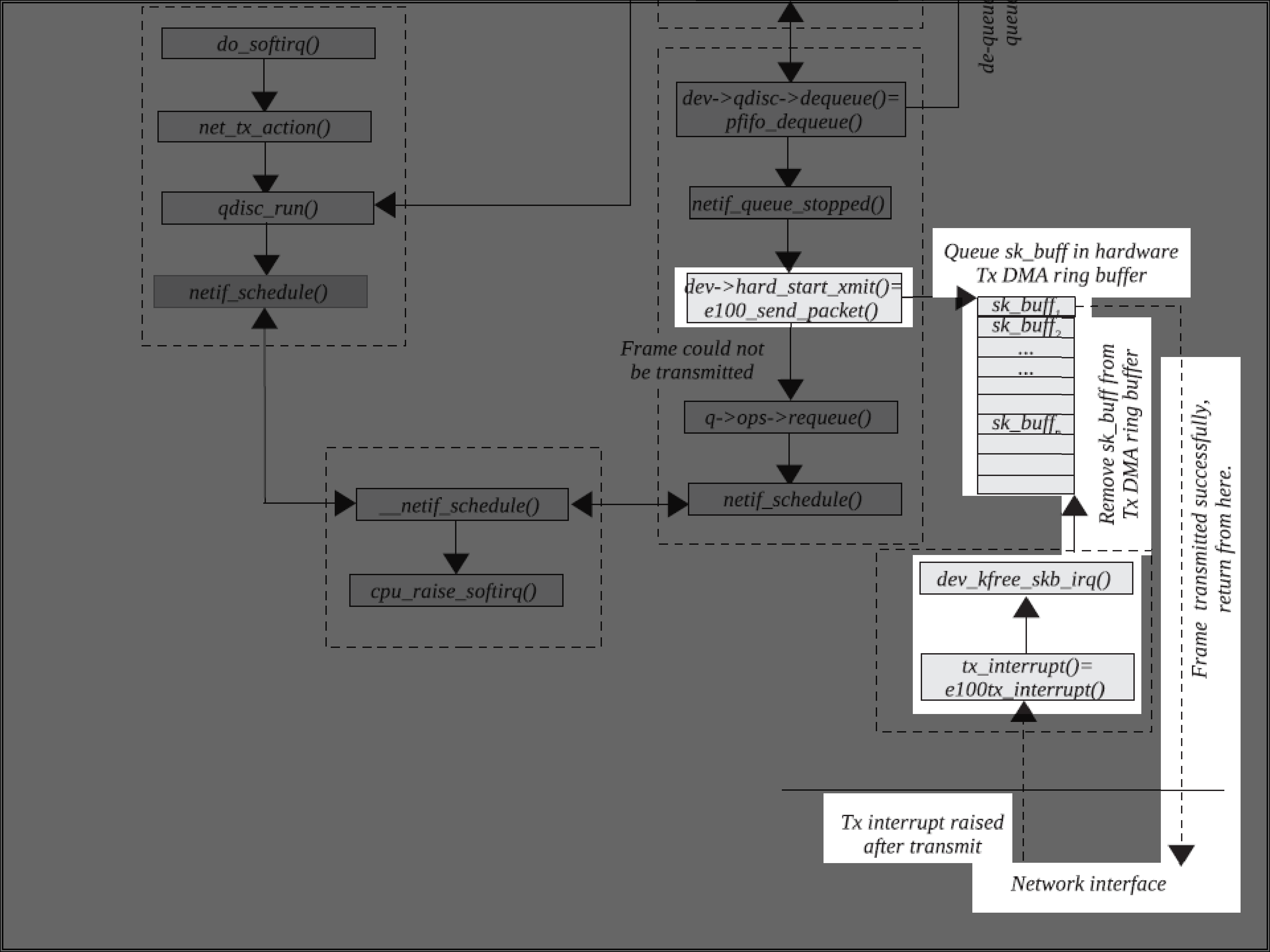

User space
write()
Kernel space
tcp_sendmsg()
tcp_alloc_pskb()
skb_entail()
tcp_push_one()
tcp_snd_test()
Queue TCP segment in socket's write_queue
sk->write_queue
sk_buff1
sk_buff2
...
...
sk_buffn
Remove the ACK'ed TCP segment from queue
tcp_clean_rtx_queue()
tcp_ack()
4
tp->af_specific->send_check= tcp_v4_send_check()
connect()
tcp_v4_connect()
ip_route_connect()
__sk_dst_set()
tcp_v4_hash_connect()
tcp_connect()
tcp_connect_init()
tcp_connect_init()
alloc_skb()
__skb_queue_tail()
tcp_charge_skb()
skb_clone()
No route found, return error
Get route
ip_route_output()
ip_route_output_key()
ip_route_output_slow()
fib_lookup()
dst_alloc()
rt_intern_hash()
arp_bind_neighbour()
__neigh_lookup_errno()
neigh_lookup
neigh_create()


sk->write_queue
Queue TCP seg
sk_buff,
sk_buff,
...
...
sk_buff,
Remove the ACK'ed TCP segment from queue
tcp_clean_rtx_queue()
tcp_ack()
4
tp->af_specific->send_check= tcp_v4_send_check()
tcp_event_ack_sent()
tp->af_specific->queue_xmit= ip_queue_xmit()
__sk_dst_check()
__sk_dst_set()
NF_HOOK()
2
tcp_connect_init()
tcp_connect_init()
alloc_skb()
__skb_queue_tail()
tcp_charge_skb()
skb_clone()
tcp_transmit_skb()
tcp_reset_xmit_timer()
Cached route not valid
Got valid route for destination
dst_alloc()
rt_intern_hash()
arp_bind_neighbour()
__neigh_lookup_errno()
neigh_lookup
neigh_create()

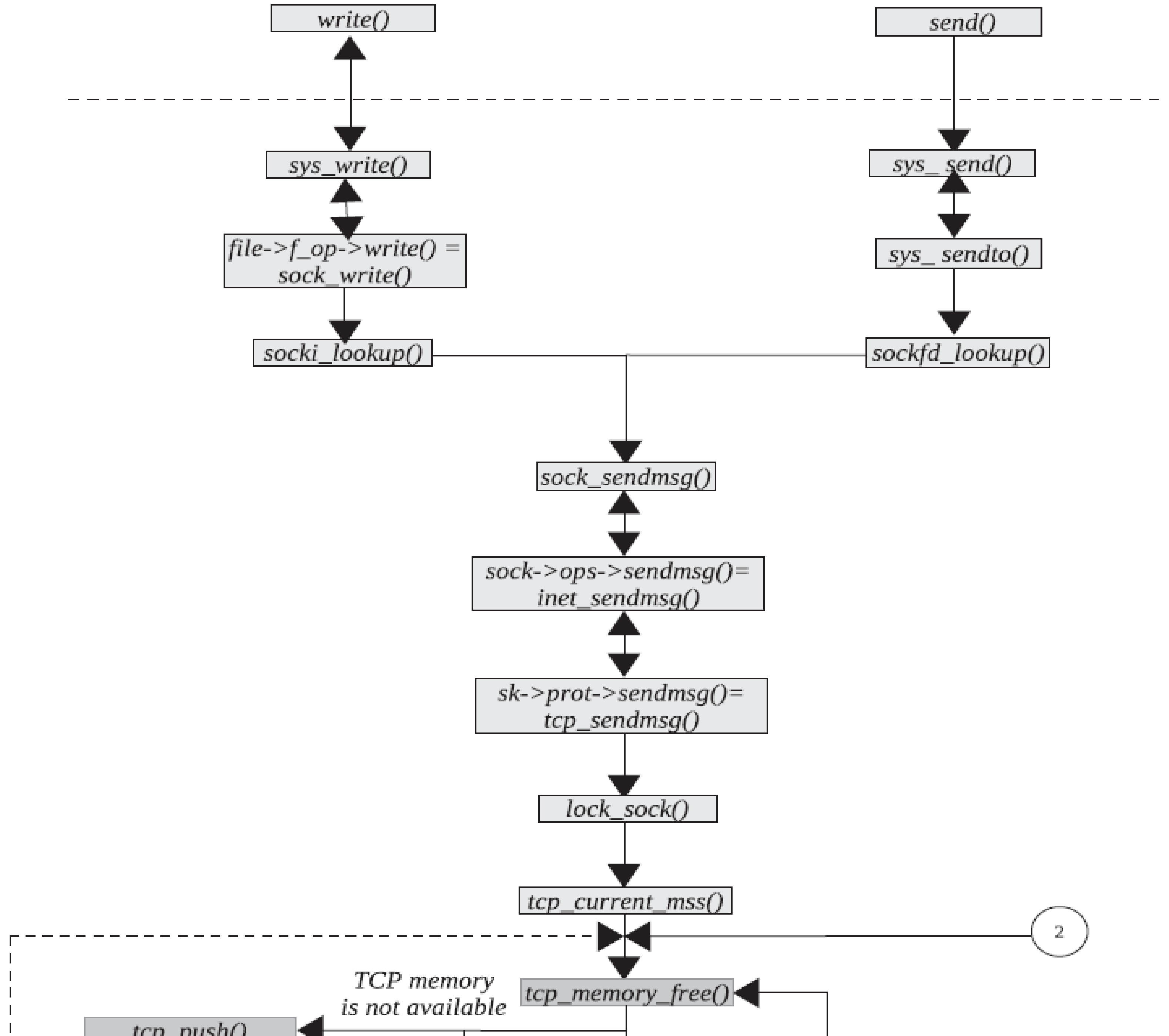

2
nf_hook_slow()
ip_queue_xmit2()
skb_realloc_headroom()
ip_dont_fragment()
Send ICMP message to TCP regarding re-sizing of the packets as mss has changed, return.
Packet needs to be fragmented
icmp_send()
ip_send_check()
ip_select_ident()
Received ICMP for the connection
skb->dst->output()=ip_output()
ip_fragment()
tcp_v4_err()
NF_HOOK()
do_pmtu_discovery()
nf_hook_slow()
ip_rt_update_pmtu()
ip_finish_output2()
tcp_sync_mss()


NF_HOOK()
nf_hook_slow()
ip_finish_output2()
skb->dst->neighbour->output()/ skb->dst->hh->hh_output() = neigh_resolve_output()
__skb_pull()
neigh_event_send()
neigh_hh_init()
dev->hard_header()
neigh->ops->queue_xmit()= dev_queue_xmit()
skb_linearize()
do_pmtu_discovery()
ip_rt_update_pmtu()
tcp_sync_mss()
tcp_simple_retransmit()
tcp_fragment()
tcp_transmit_skb()
4


3
skb_checksum_help()
skb->dev->qdisc->enqueue()=
pfifo_enqueue()
qdisc_run()
Queue sk_buff in device queue
skd->dev->qdisc->q
sk_buff1
sk_buff2
...
...
sk_buffn
de-queue sk_buff from device queue for transmission
netif_queue_stopped()
qdisc_restart()
dev->qdisc->dequeue()=
pfifo_dequeue()
netif_queue_stopped()
dev->hard_start_xmit()=
e100_send_packet()
Frame could not be transmitted
q->ops->requeue()
Queue sk_buff in hardware Tx DMA ring buffer
sk_buff1
sk_buff2
...
...
sk_buffn
do_softirq()
net_tx_action()
qdisc_run()
netif_schedule()


do_softirq()
net_tx_action()
qdisc_run()
netif_schedule()
dev->qdisc->dequeue()=
pfifo_dequeue()
de-queu
queu
netif_queue_stopped()
Queue sk_buff in hardware
Tx DMA ring buffer
dev->hard_start_xmit()=
e100_send_packet()
sk_buff1
sk_buff2
...
...
sk_buffn
Frame could not
be transmitted
q->ops->requeue()
Remove sk_buff from
Tx DMA ring buffer
Frame transmitted successfully,
return from here.
__netif_schedule()
netif_schedule()
cpu_raise_softirq()
dev_kfree_skb_irq()
tx_interrupt()=
e100tx_interrupt()
Tx interrupt raised
after transmit
Network interface

## Overall High Level Functional Overview of Sending


write()
send()
sys_write()
sys_ send()
file->f_op->write() = sock_write()
sys_ sendto()
socki_lookup()
sockfd_lookup()
sock_sendmsg()
sock->ops->sendmsg()= inet_sendmsg()
sk->prot->sendmsg()= tcp_sendmsg()
lock_sock()
tcp_current_mss()
2
tcp_memory_free()
TCP memory is not available
tcp_push()

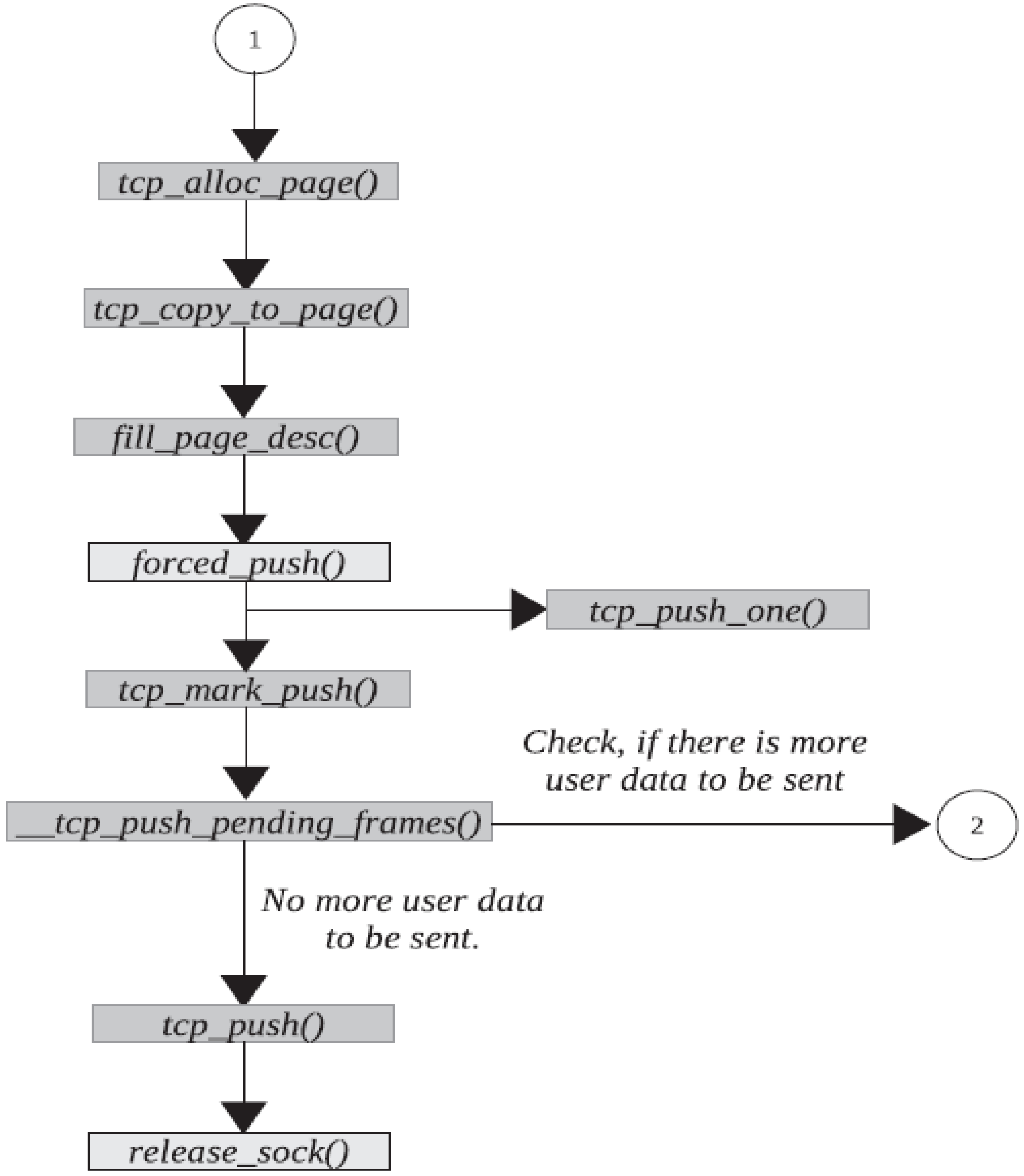


**Figure 7.6b.** Functional flow of TCP send process (*continued*).

## net_device

- *Defined in <include/linux/netdevice.h>*
- stores all information specifically regarding a network device
- one such structure for each device, both real ones (such as Ethernet NICs) and virtual ones
- Network devices can be classified into types such as *Ethernet cards* and *Token Ring cards*
- Each type may come in several models.
- Model specific parameters are initialized by device driver software.
- Parameters common for different models are initiated by kernel.

# configuration parameters

```
struct net_device{
     char                    name[IFNAMSIZ];
     int                     ifindex;

     /* device name hash chain, ex: eth0 */
     struct hlist_node       name_hlist;

     unsigned long                     mem_end;/* shared mem end     */
     unsigned long                     mem_start;           /* shared mem start   */
     unsigned long                     base_addr;           /* device I/O address  */
     unsigned int            irq;                 /* device IRQ number */
     unsigned char                     if_port;             /* Selectable AUI, TP,..*/
     unsigned char                     dma;                 /* DMA channel         */

...
```

# configuration parameters

```
struct net_device{
    char                name[IFNAMSIZ];
    int                 ifindex;

    /* device name hash chain, ex: eth0 */
    struct hlist_node   name_hlist;

    unsigned long             mem_end;      /* shared mem end     */
    unsigned long             mem_start;    /* shared mem start   */
    unsigned long             base_addr;    /* device I/O address */
    unsigned int        irq;                /* device IRQ number */
    unsigned char             if_port;      /* Selectable AUI, TP,..*/
    unsigned char             dma;          /* DMA channel        */

…
```

## configuration parameters

```
struct net_device{
      char                name[IFNAMSIZ];
      int                 ifindex;

      /* device name hash chain, ex: eth0 */
      struct hlist_node   name_hlist;

      unsigned long                 mem_end;/* shared mem end     */
      unsigned long                 mem_start;          /* shared mem start   */
      unsigned long                 base_addr;          /* device I/O address   */
      unsigned int                  irq;                /* device IRQ number */
      unsigned char                 if_port;            /* Selectable AUI, TP,..*/
      unsigned char                 dma;                /* DMA channel
         */
```

# configuration parameters

```
struct net_device{
    char                    name[IFNAMSIZ];
    /* device name hash chain, ex: eth0 */
    struct hlist_node       name_hlist;

    unsigned long                       mem_end;/* shared mem end      */
    unsigned long                       mem_start;          /* shared mem start   */
    unsigned long                       base_addr;          /* device I/O address  */
    unsigned int            irq;                  /* device IRQ number */
    unsigned char                       if_port;            /* Selectable AUI, TP,..*/
    unsigned char                       dma;                /* DMA channel         */
    unsigned short                      flags;      /*   interface flags (a la BSD)     */
...
```

# configuration parameters

```
struct net_device{
    char                    name[IFNAMSIZ];
    /* device name hash chain, ex: eth0 */
    struct hlist_node       name_hlist;

    unsigned long                   mem_end;/* shared mem end     */
    unsigned long                   mem_start;         /* shared mem start   */
    unsigned long                   base_addr;         /* device I/O address  */
    unsigned int            irq;             /* device IRQ number */
    unsigned char                   if_port;           /* Selectable AUI, TP,..*/
    unsigned char                   dma;               /* DMA channel        */
    unsigned short                  flags;             /*  interface flags (a la BSD)*/
...
```

# configuration parameters

```
struct net_device{
     char                    name[IFNAMSIZ];
     /* device name hash chain, ex: eth0 */
     struct hlist_node       name_hlist;

     unsigned long                      mem_end;/* shared mem end    */
     unsigned long                      mem_start;          /* shared mem start   */
     unsigned long                      base_addr;          /* device I/O address  */
     unsigned int            irq;                     /* device IRQ number*/
     unsigned char                      if_port;            /* Selectable AUI, TP,..*/
     unsigned char                      dma;                /* DMA channel        */
     unsigned short                     flags;              /*  interface flags (a la BSD)*/
/* ex : IFF_UP || IFF_RUNNING || IFF_MULTICAST */
```

# configuration parameters

```
struct net_device{

    …

    unsigned               mtu;                 /* interface MTU value      */
    unsigned short         type;                /* interface hardware type  */
    unsigned short         hard_header_len;     /* hardware hdr length      */

    unsigned char          dev_addr[MAX_ADDR_LEN];
    unsigned char          addr_len;            /* hardware address length  */

    unsigned char          broadcast[MAX_ADDR_LEN];
    unsigned int           promiscuity;

…
```

# configuration parameters

```
struct net_device{

    ...

     unsigned           mtu;               /* interface MTU value      */
    unsigned short      type;              /* interface hardware type*/
    unsigned short      hard_header_len;   /* hardware hdr length      */

    unsigned char       dev_addr[MAX_ADDR_LEN];
    unsigned char       addr_len;          /* hardware address length  */

    unsigned char       broadcast[MAX_ADDR_LEN];
    unsigned int        promiscuity;

...
```

# configuration parameters

```
struct net_device{

    ...

     unsigned          mtu;              /* interface MTU value       */
    unsigned short     type;             /* interface hardware type   */
    unsigned short     hard_header_len;/* hardware hdr length   */

    unsigned char      dev_addr[MAX_ADDR_LEN];
    unsigned char      addr_len;         /* hardware address length   */

    unsigned char      broadcast[MAX_ADDR_LEN];
    unsigned int       promiscuity;

...
```

# configuration parameters

```
struct net_device{

    ...

     unsigned              mtu;               /* interface MTU value          */
    unsigned short         type;              /* interface hardware type      */
    unsigned short         hard_header_len;   /* hardware hdr length          */

    unsigned char          dev_addr[MAX_ADDR_LEN];
    unsigned char          addr_len;          /* hardware address length*/

    unsigned char          broadcast[MAX_ADDR_LEN];
    unsigned int           promiscuity;

...
```

# configuration parameters

```
struct net_device{

      …

       unsigned              mtu;              /* interface MTU value          */
      unsigned short         type;             /* interface hardware type      */
      unsigned short         hard_header_len;  /* hardware hdr length          */

      unsigned char          dev_addr[MAX_ADDR_LEN];
      unsigned char          addr_len;         /* hardware address length      */

      unsigned char          broadcast[MAX_ADDR_LEN];
      unsigned int           promiscuity;

…
```

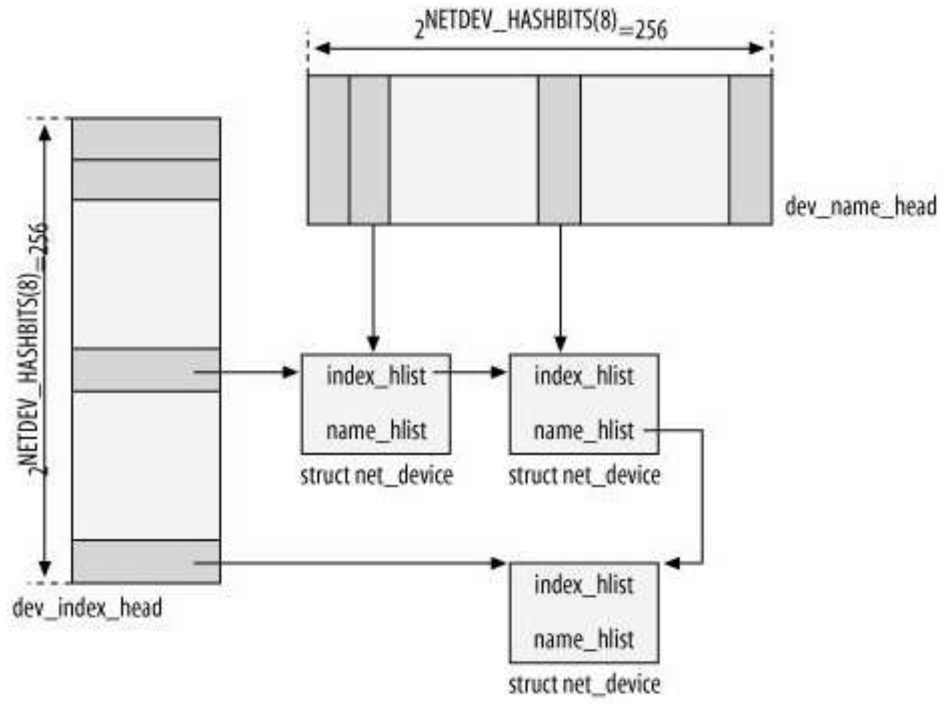
# configuration parameters

```
struct net_device{

    ...

     unsigned                 mtu;               /* interface MTU value          */
    unsigned short            type;              /* interface hardware type      */
    unsigned short            hard_header_len;   /* hardware hdr length          */

    unsigned char             dev_addr[MAX_ADDR_LEN];
    unsigned char             addr_len;          /* hardware address length      */

    unsigned char             broadcast[MAX_ADDR_LEN];
    unsigned int              promiscuity;

...
```

# list management

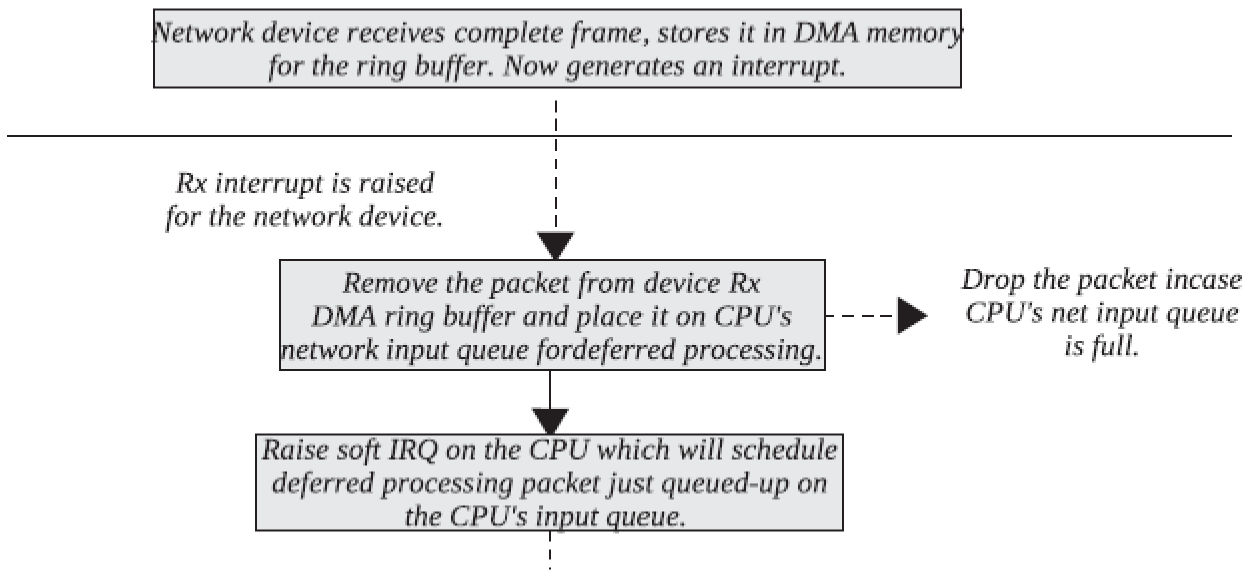
struct net_device{

    ...
    struct net_device *next;
    struct hlist_node name_hlist·
    struct hlist_node index_hlist

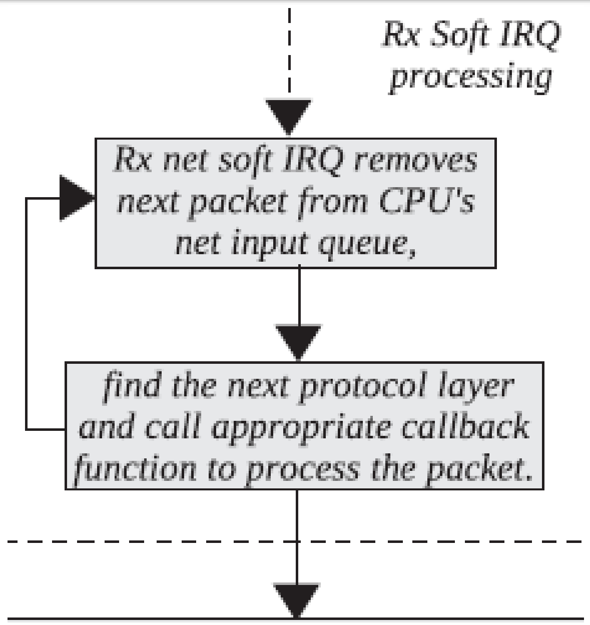


## Walkthrough Reception

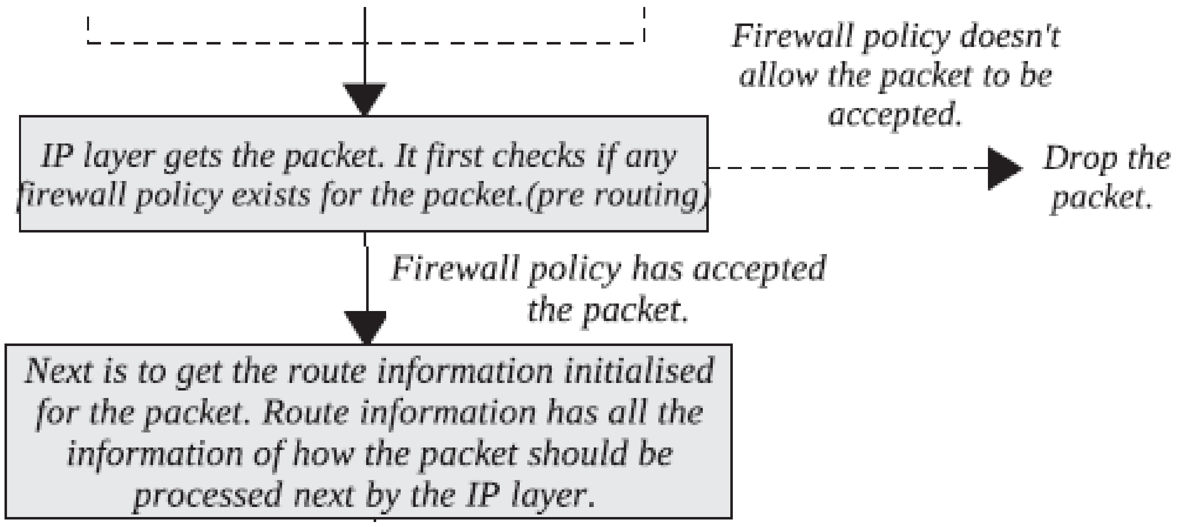
# Walkthrough Reception


Network device receives complete frame, stores it in DMA memory for the ring buffer. Now generates an interrupt.
Rx interrupt is raised for the network device.
Remove the packet from device Rx DMA ring buffer and place it on CPU's network input queue fordeferred processing.
Drop the packet incase CPU's net input queue is full.
Raise soft IRQ on the CPU which will schedule deferred processing packet just queued-up on the CPU's input queue.

- *We don't process the packet in the interrupt subroutine.*
- *Netif_rx() – raise the net Rx softIRQ.*
- *Net_rx_action() is called - start processing the packet*
- *Processing of packet starts with the protocol switching section*

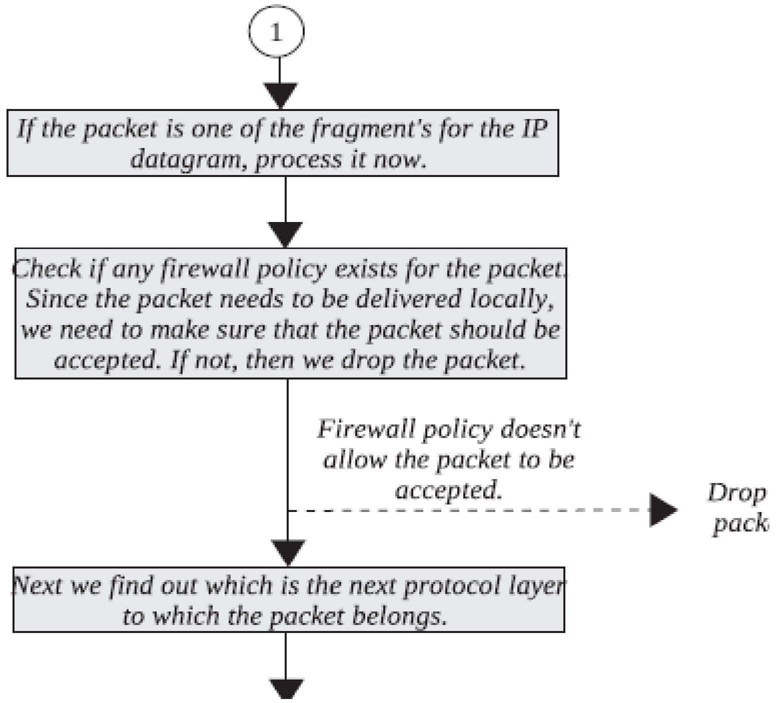

Rx Soft IRQ processing
Rx net soft IRQ removes next packet from CPU's net input queue,
find the next protocol layer and call appropriate callback function to process the packet.

- *Netif_receive_skb() is called to process the packet and find out the next protocol layer.*
- *Protocol family of the packet is extracted from the link layer header.*

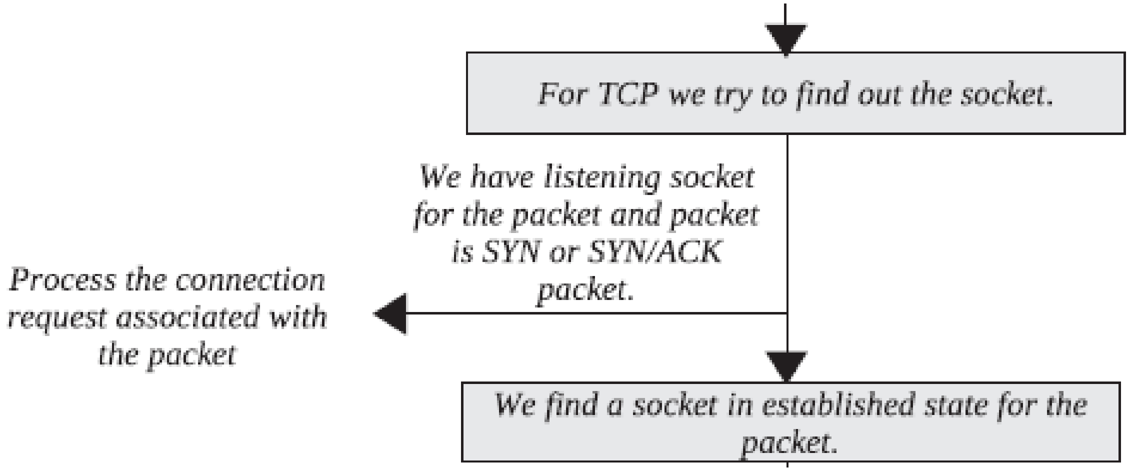

Firewall policy doesn't allow the packet to be accepted.
IP layer gets the packet. It first checks if any firewall policy exists for the packet.(pre routing)
Drop the packet.
Firewall policy has accepted the packet.
Next is to get the route information initialised for the packet. Route information has all the information of how the packet should be processed next by the IP layer.

- *ip_rcv() is an entry point for IP packets processing.*
- *Checks if the packet we have is destined for some other host (using PACKET_OTHERHOST)*
- *Check  the checksum of the packet by calling ip_fast_csum()*

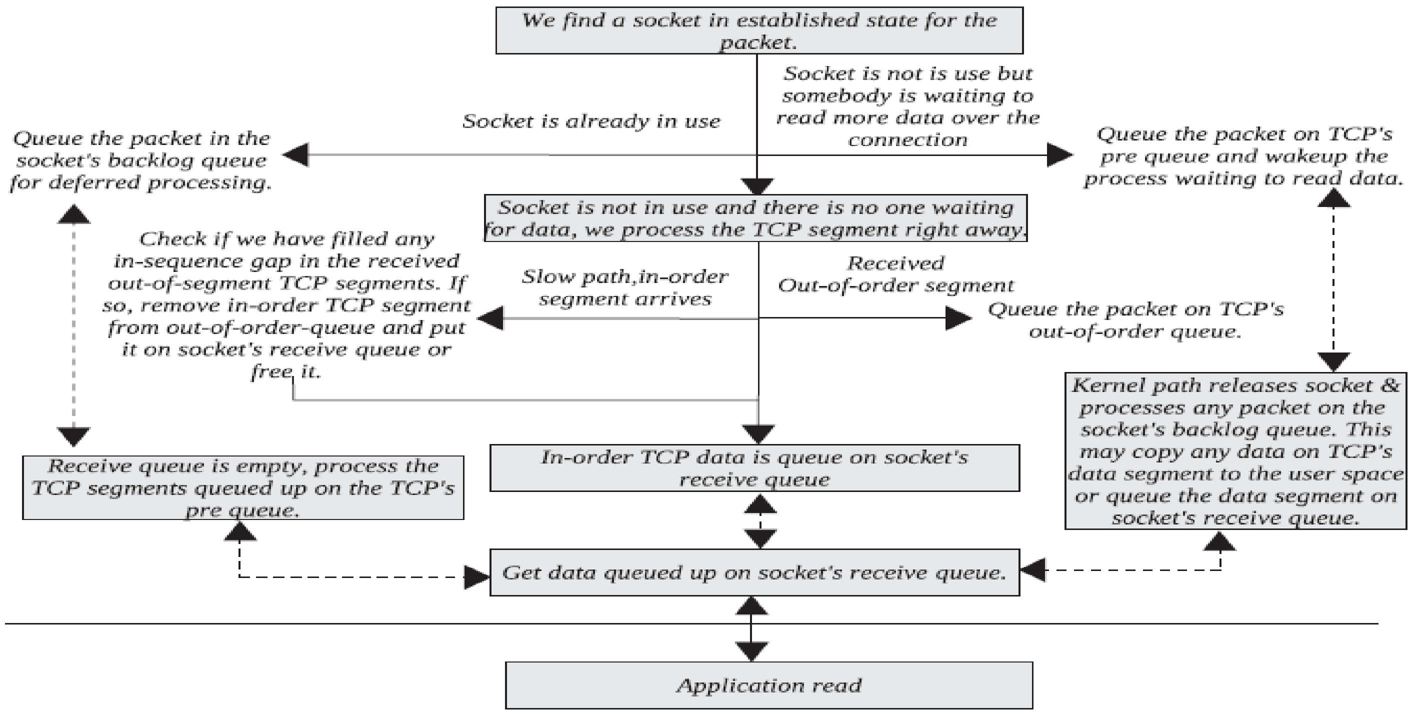
- *Call ip_route_input() , this routine checks kernel routing table rt_hash_table.*
- *If  packet needs to be forwarded input routine is ip_forward()*
- *Otherwise ip_local_deliver()*
- *ip_send() is called to check if the packet needs to be fragmented*
- *If yes , fragment the packet by calling ip_fragment()*
- *Packet output path – ip_finish_output()*
- *ip_local_deliver() – packets need to delivered locally*


1
If the packet is one of the fragment's for the IP datagram, process it now.
Check if any firewall policy exists for the packet. Since the packet needs to be delivered locally, we need to make sure that the packet should be accepted. If not, then we drop the packet.
Firewall policy doesn't allow the packet to be accepted.
Drop pack
Next we find out which is the next protocol layer to which the packet belongs.

- ➢*ip_defrag()*
- ➢*Protocol identifier field skb->np.iph->protocol (in IP header).*
- ➢*For TCP, we find the receive handler  as tcp_v4_rcv() (entry point for the TCP layer)*


For TCP we try to find out the socket.
We have listening socket for the packet and packet is SYN or SYN/ACK packet.
Process the connection request associated with the packet
We find a socket in established state for the packet.

- *_tcp_v4_lookup() – find the socket to which the packet belongs*
- *Establised sockets are maintained in the hash table tcp_ehash.*
- *Established socket not found – New connection request for any listening socket*
- *Search for listening socket – tcp_v4_lookup_listener()*
- *tcp_rcv_established()*


We find a socket in established state for the packet.
Socket is already in use
Socket is not is use but somebody is waiting to read more data over the connection
Queue the packet in the socket's backlog queue for deferred processing.
Queue the packet on TCP's pre queue and wakeup the process waiting to read data.
Socket is not in use and there is no one waiting for data, we process the TCP segment right away.
Check if we have filled any in-sequence gap in the received out-of-segment TCP segments. If so, remove in-order TCP segment from out-of-order-queue and put it on socket's receive queue or free it.
Slow path,in-order segment arrives
Received Out-of-order segment
Queue the packet on TCP's out-of-order queue.
Kernel path releases socket & processes any packet on the socket's backlog queue. This may copy any data on TCP's data segment to the user space or queue the data segment on socket's receive queue.
Receive queue is empty, process the TCP segments queued up on the TCP's pre queue.
In-order TCP data is queue on socket's receive queue
Get data queued up on socket's receive queue.
Application read

- *Application read the data  from the receive queue if it issues recv()*
- *Kernel routine to read data from TCP socket is tcp_recvmsg()*

Thank You
Any Questions?