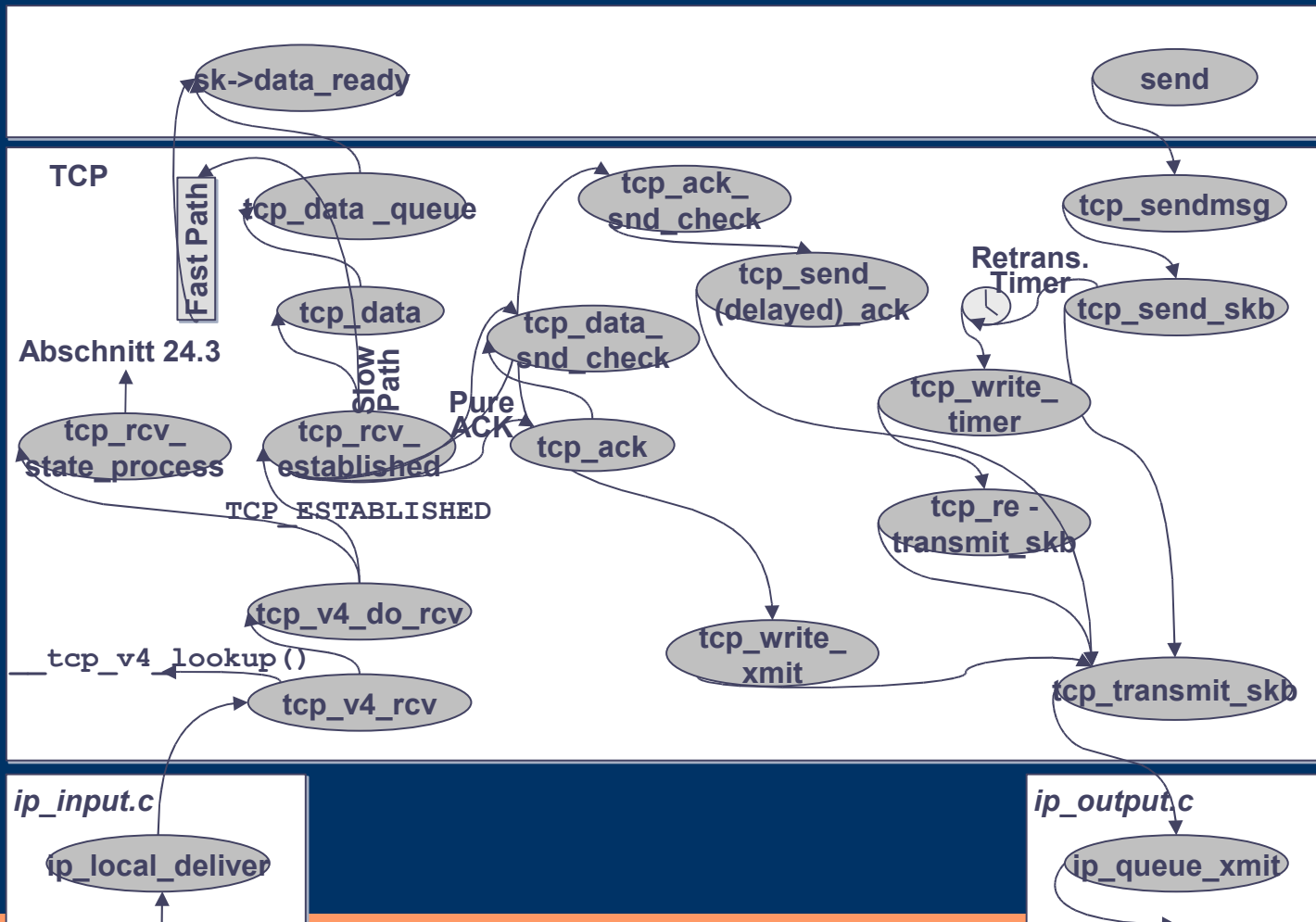


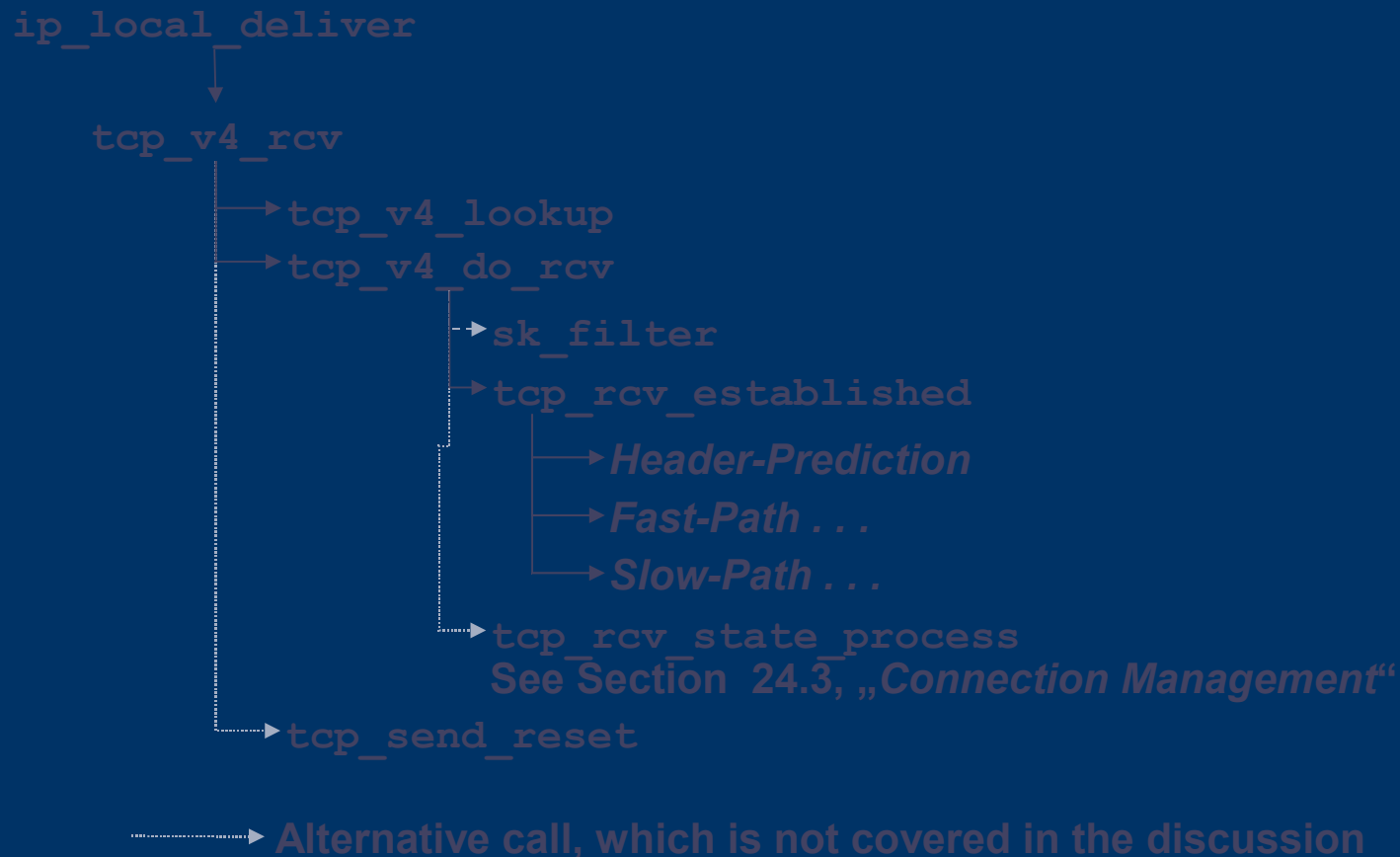
TCP Implementation in Linux



tcp_v4_rcv(skb, len)

- Checks if the packet is really addressed to the host (*skb* → *pkt_type* == *PACKET_HOST*). If not, the packet is discarded.
 - Invokes *tcp_v4_lookup()* to search the hash table of active sockets for the matching sock structure.
 - Source/destination IP addresses and ports and the network device index *skb* → *dst* → *rt_iif* at which the segment arrive are used to index into the hash table.
 - If a matched sock structure is located, *tcp_v4_do_rcv()* is invoked; otherwise, *tcp_send_reset()* sends a *RESET* segment.
-
-

Process of Receiving a Segment



tcp_v4_do_rcv()

- If the TCP state ($sk \rightarrow state$) is
 - TCP_ESTABLISHED, invokes *tcp_rcv_established()*.
 - One of the other states, invokes *tcp_rcv_state_process()*, i.e., the TCP state machine will be examined to determine state transition.
-
-

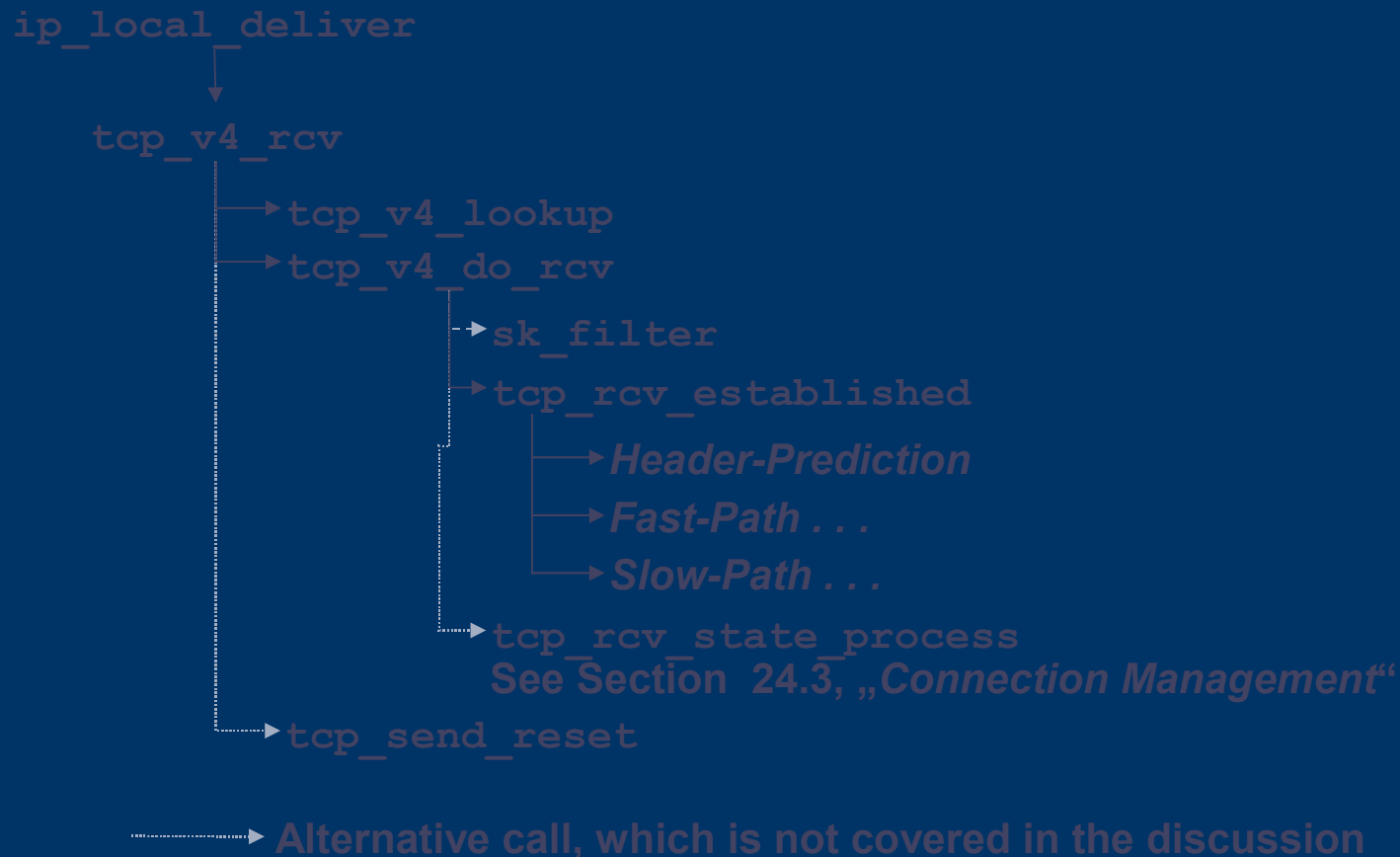
tcp_rcv_established(sk,skb,th,len)

- Dispatches packets to *fast path* or *slow path*
- Packets are processed in fast path if
 - The segment received is a pure ACK segment for the data sent last.
 - The segment received contains the data expected.

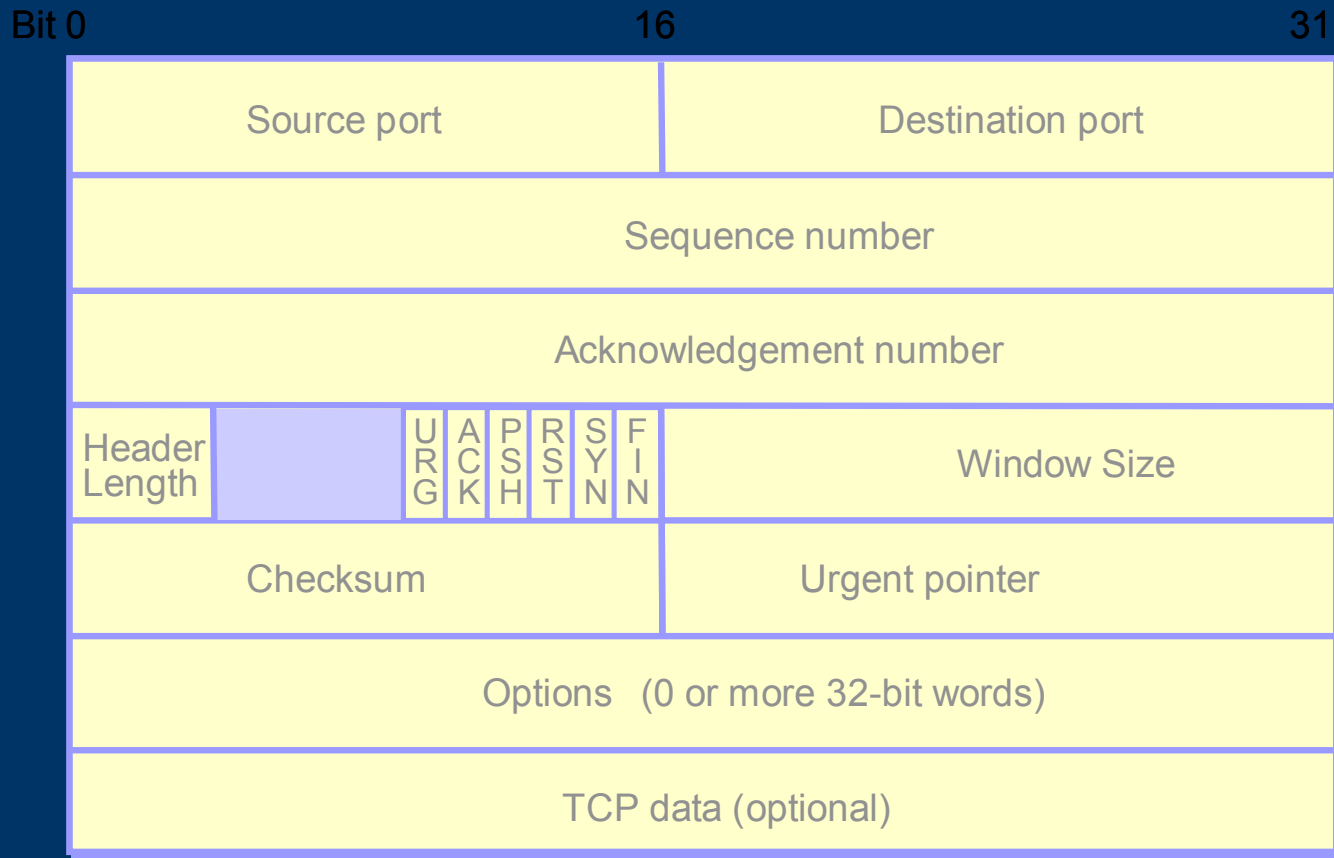
tcp_rcv_established(sk,skb,th,len)

- Packets are processed in slow path if
 - If SYN, URG, FIN, RST flag is set (detected in *Header Prediction*).
 - The SN of the segment does not correspond to *tp* → *rcv_nxt*.
 - The communication is two-way.
 - The segment contains a zero window advertisement.
 - The segment contains TCP options other than the timestamp option.
-
-

Process of Receiving a Segment



Header Prediction (TCP Header)



Header Prediction

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

Note that

7. `#define TCP_HP_BITS (~(TCP_RESERVED_BITS|TCP_FLAG_PSH))`
 8. `tp->pred_flags` is set in *tcp_fast_path_on()*
-
-

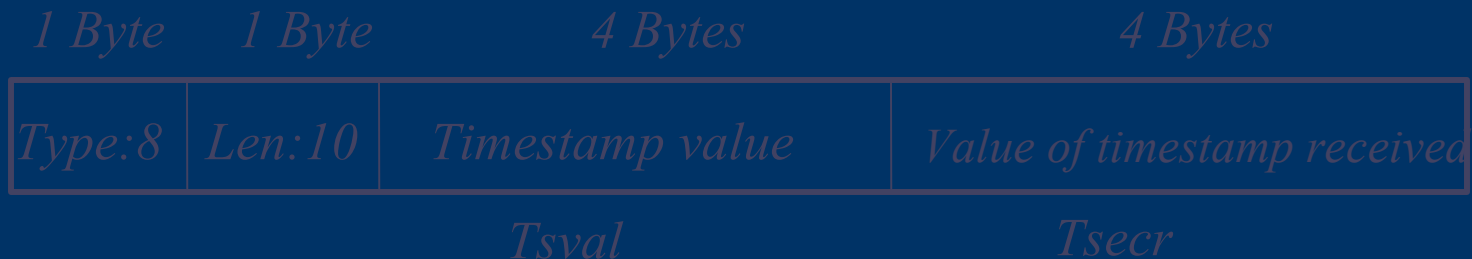
Header Prediction

```
static __inline__ void __tcp_fast_path_on(struct tcp_opt *
    tp, u32 snd_wnd)
{
    tp->pred_flags = htonl((tp->tcp_header_len << 26) |
        ntohl(TCP_FLAG_ACK) | snd_wnd);
}

static __inline__ void tcp_fast_path_on(struct tcp_opt *tp
    )
{
    __tcp_fast_path_on(tp, tp->snd_wnd>>tp->snd_wscale);
}
```

Fast Path in *tcp_rcv_established()*

1. $TCP_SKB_CB(skb) \rightarrow seq == tp \rightarrow rcv_nxt$? If so, proceed.
2. Checks if the timestamp option exists. If so,
 - the timestamp value, *Tsval* and *Tsecr* are read.
 - If the condition to update the $tp \rightarrow ts_recent$ timestamp is met (i.e., $tp \rightarrow rcv_tsval - tp \rightarrow ts_recent < 0$), the values are accepted by *tcp_store_ts_recent()*.



Fast Path in tcp_rcv_established()

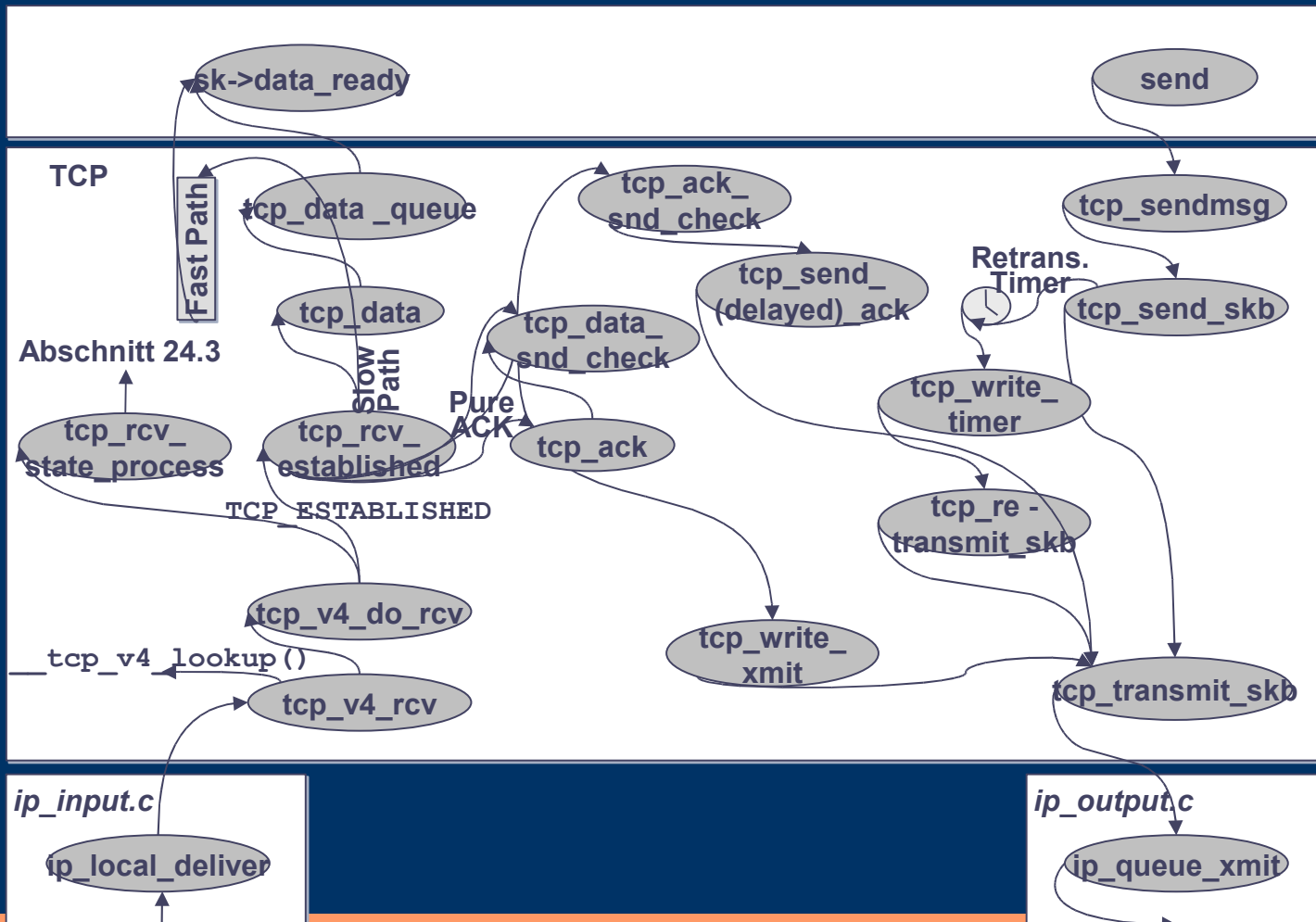
1. packet header length == segment length?
 2. Yes → ACM segment
 - Invokes *tcp_ack()* to process the ack.
 - Invokes *__kfree_skb()* to release the socket buffer
 - Invokes *tcp_data_snd_check()* to check if local packets can be sent (because of the send quota induced by the ack).
-
-

Fast Path in tcp_rcv_established()

1. No → Data segment

- If the payload can be copied directly into the user space,
 - the statistics of the connection are updated
 - the relevant process is informed
 - the payload is copied into the receive memory of the process
 - The sequence number expected next is updated
 - If the payload *cannot* be copied directly
 - Checks if the receive buffer for the socket is sufficient
 - The statistics of the connection are updated
 - The segment is added to the end of the receive queue of the socket
 - The sequence number expected next is updated.
-
-

TCP Implementation in Linux



Fast Path in tcp_rcv_established()

1. No → Data segment (cont'd)
 - Invokes *tcp_event_data_rcv()* to carry out various management tasks
 - If the segment contains an ack, then invoke *tcp_ack()* to process the ack and *tcp_data_snd_check()* to initiate transmission of waiting local data segments.
 - Checks if an ack has to be sent back in response to receipt of the segment, in the form of Delayed ACK or Quick ACK mode.
-
-

Helper Function – tcp_ack()

1. Adapt the receive window
(*tcp_ack_update_window()*)
 2. Delete acknowledged packets from the
retransmission queue (*tcp_clean_rtx_queue()*)
 3. Check for zero window probing
acknowledgement.
 4. Update RTT and RTO.
 5. Activate the fast retransmit mode if necessary.
-
-

Helper Function –

tcp_data_snd_check()

 *tcp_data_snd_check()* checks if local data in the transmit queue can be transmitted (as allowed by the sliding windows)

```
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, tp->nonagle))
            tcp_check_probe_timer(sk, tp);
    }
    tcp_check_space(sk);
}
```

Slow Path

- Checks the checksum.
 - Checks the timestamp option via *tcp_fast_parse_options()*; performs PAWS check via *tcp_paws_discard()*;
 - Invokes *tcp_sequence()* to check if the packet arrived out of order, and if so, activate the *QuickAck* mode to send acks asap.
 - If RST is set, invoke *tcp_reset()* to reset the connection and free the socket buffer.
 - If the TCP header contains a timestamp option, update the recent timestamp stored locally with *tcp_replace_ts_recent()*.
-
-

Slow Path

- If *SYN* is set to signal an error in an established connection, invokes *tcp_reset()* to reset the connection.
 - If *ACK* is set, invoke *tcp_ack()* to process the ack.
 - If *URG* Is set, invoke *tcp_urg()* to process the priority data.
 - Invokes *tcp_data()* and *tcp_data_queue()* to process the payload.
 - Checks if the receive queue of the sock structure has sufficient space.
 - Inserts the segment into the receive queue or the out of order queue.
 - Invokes *tcp_data_snd_check()* and *tcp_ack_snd_check()* to check whether data or acks waiting can be sent.
-
-

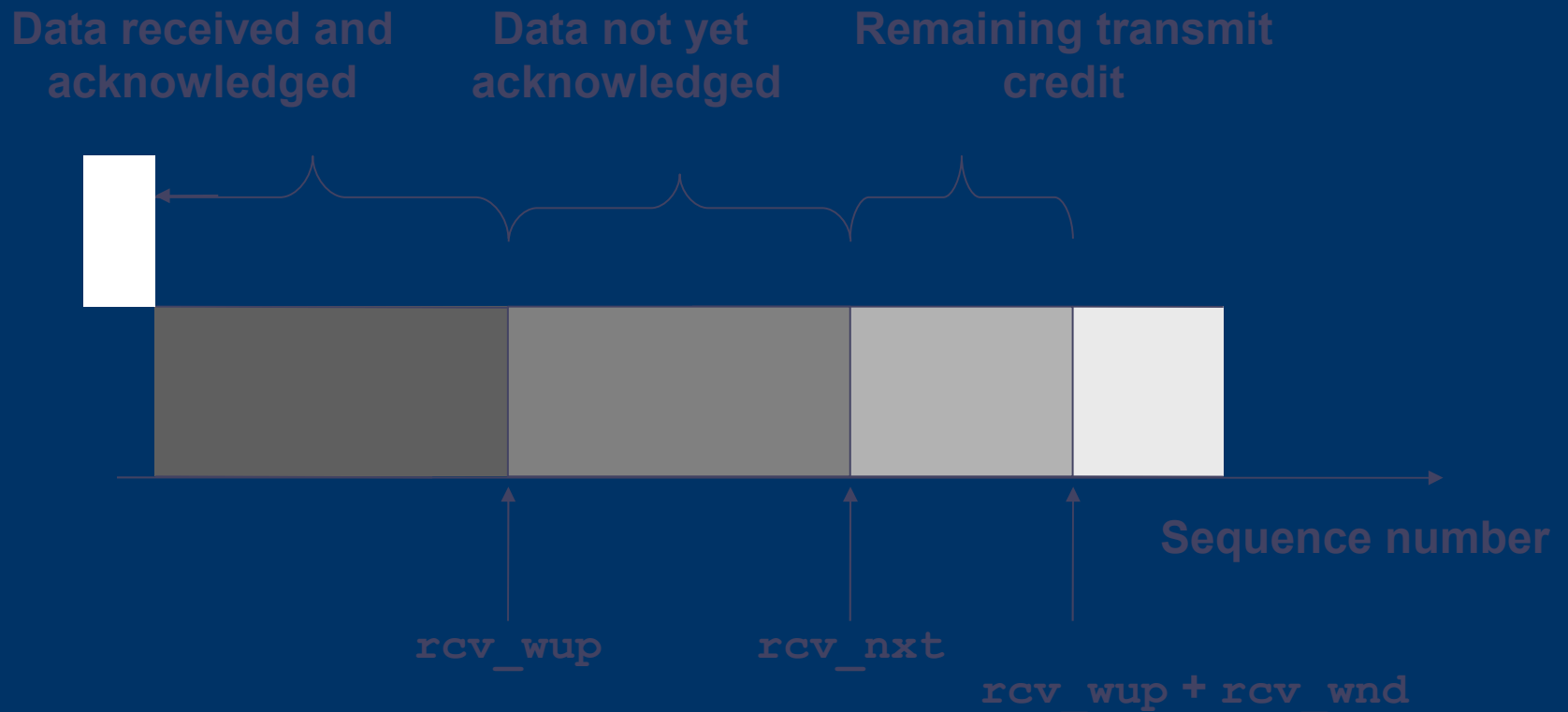
Helper Function –

tcp_ack_snd_check()

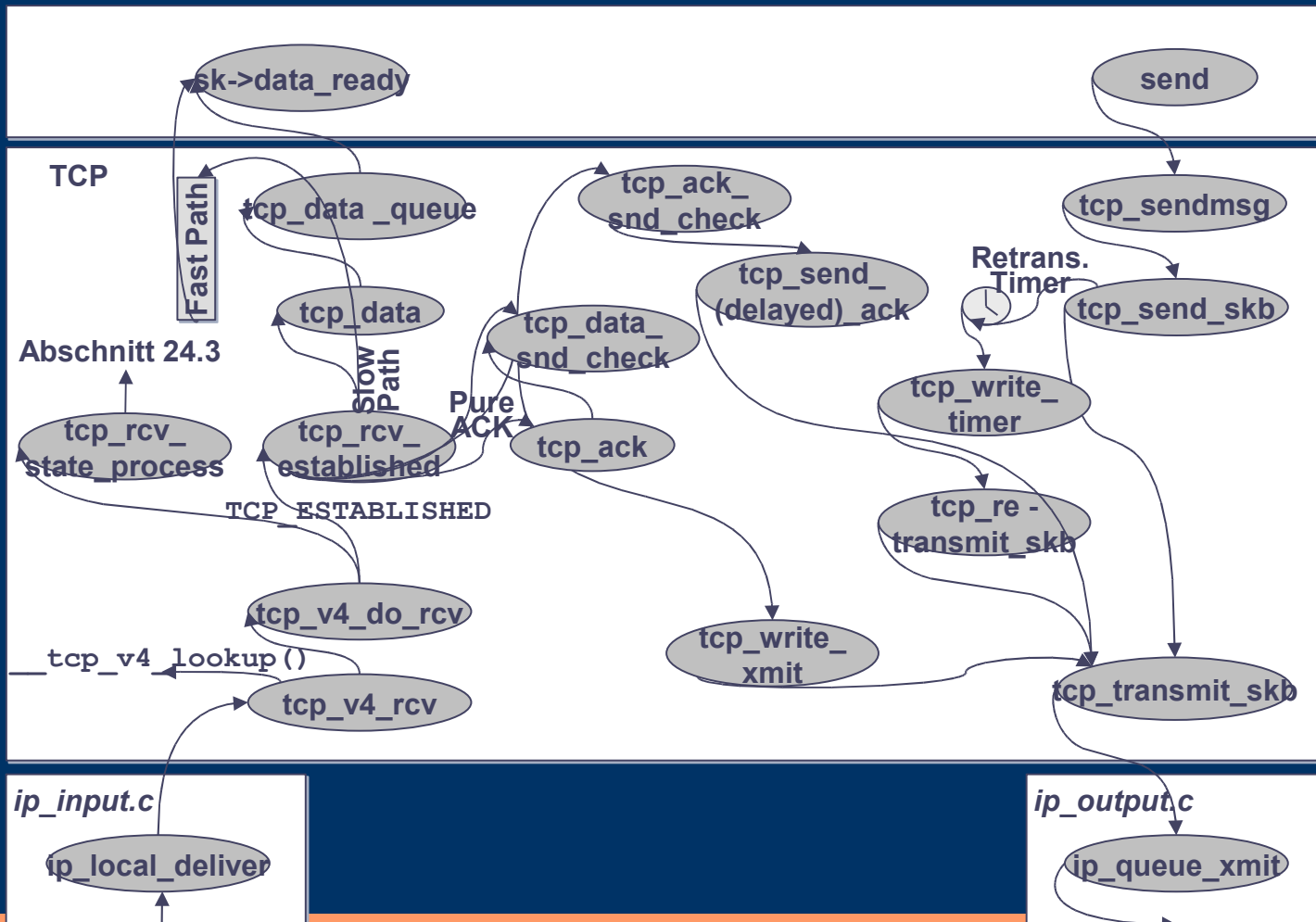
● *tcp_ack_snd_check(sk)* checks for various canases where acks can be sent.

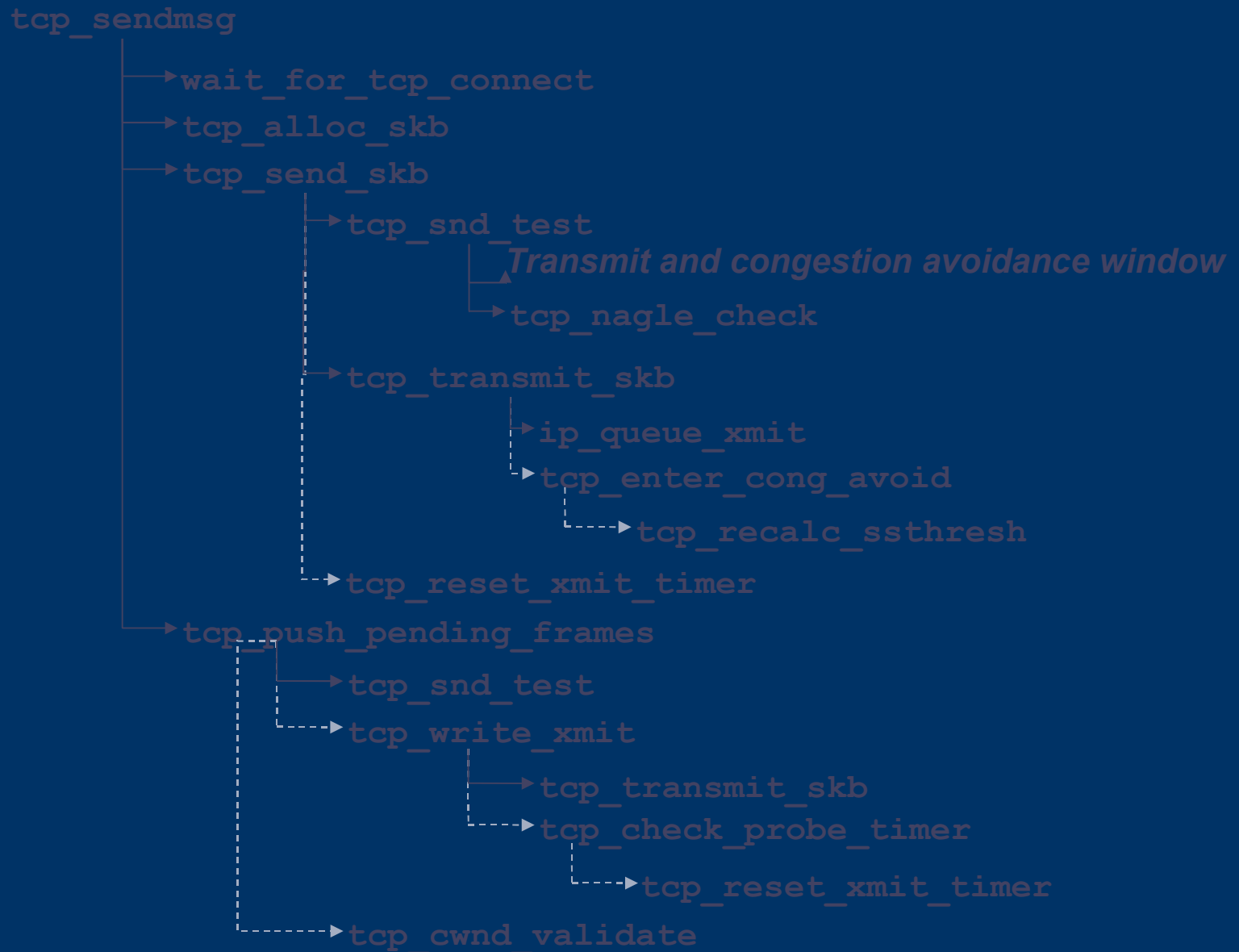
```
static __inline__ void tcp_ack_snd_check(struct sock *sk)
{
    struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
    if (!tcp_ack_scheduled(tp)) { * We sent a data segment already. */
        return;
    }
    /* More than one full frame received... */
    if (((tp->rcv_nxt - tp->rcv_wup) > tp->ack.rcv_mss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

Window Kept at the Receiver



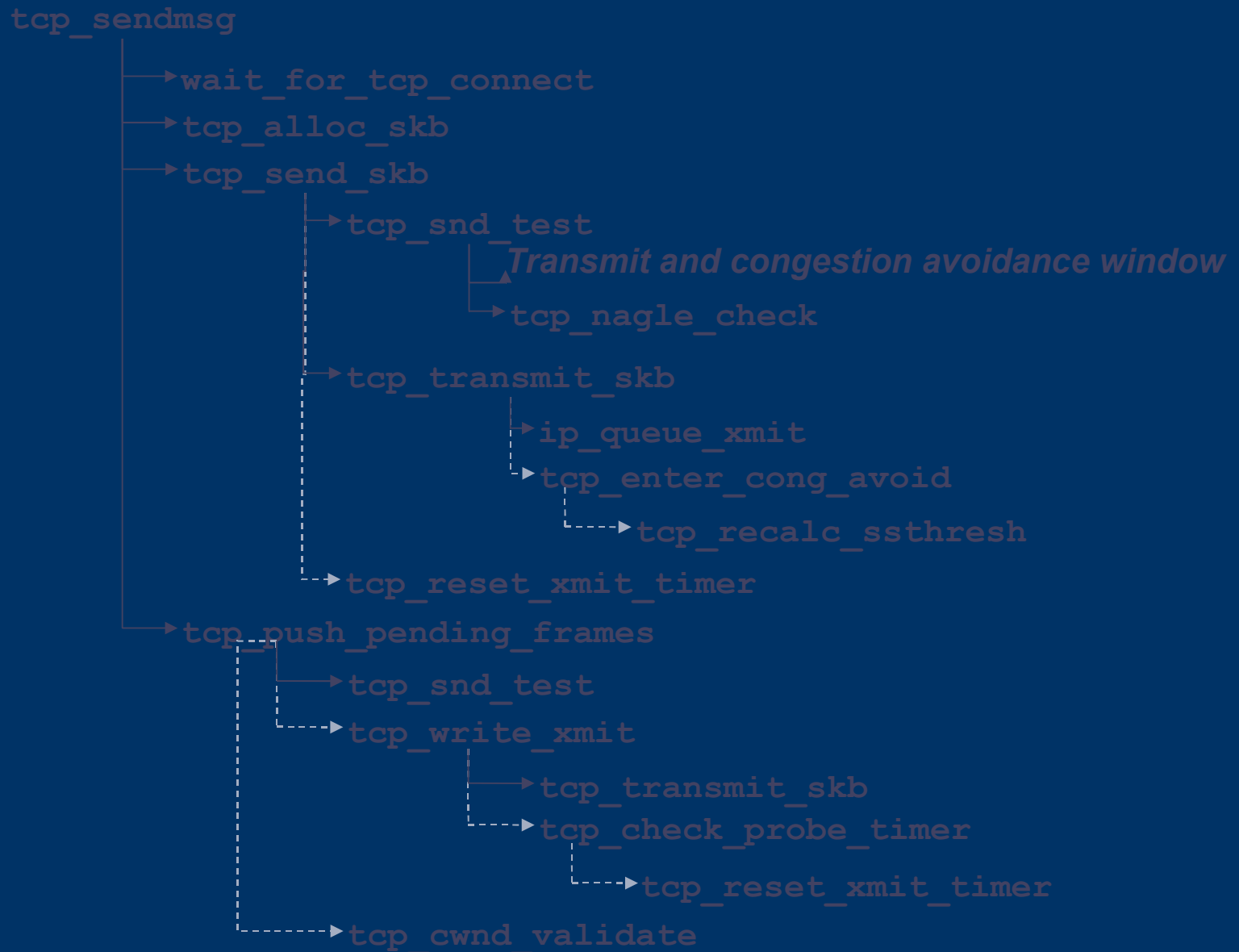
TCP Implementation in Linux





tcp_sendmsg()

- *tcp_sendmsg(sock, msg, size)* copies payload from the user space into the kernel space and send it in the form of TCP segments.
 1. Checks if the connection has already been established. If not, invokes *wait_for_tcp_connect()*.
 2. Computes the maximum segment size (*tcp_current_mss*).
 3. Invokes *tcp_alloc_skb()* and copies the data from the user space.
 4. Invokes *tcp_send_skb()* to put the socket buffer in the transmit queue of the sock structure.
 5. Invokes *-tcp_push_pending_frames()* to take segments from *tp→write_queue* and transmit them.



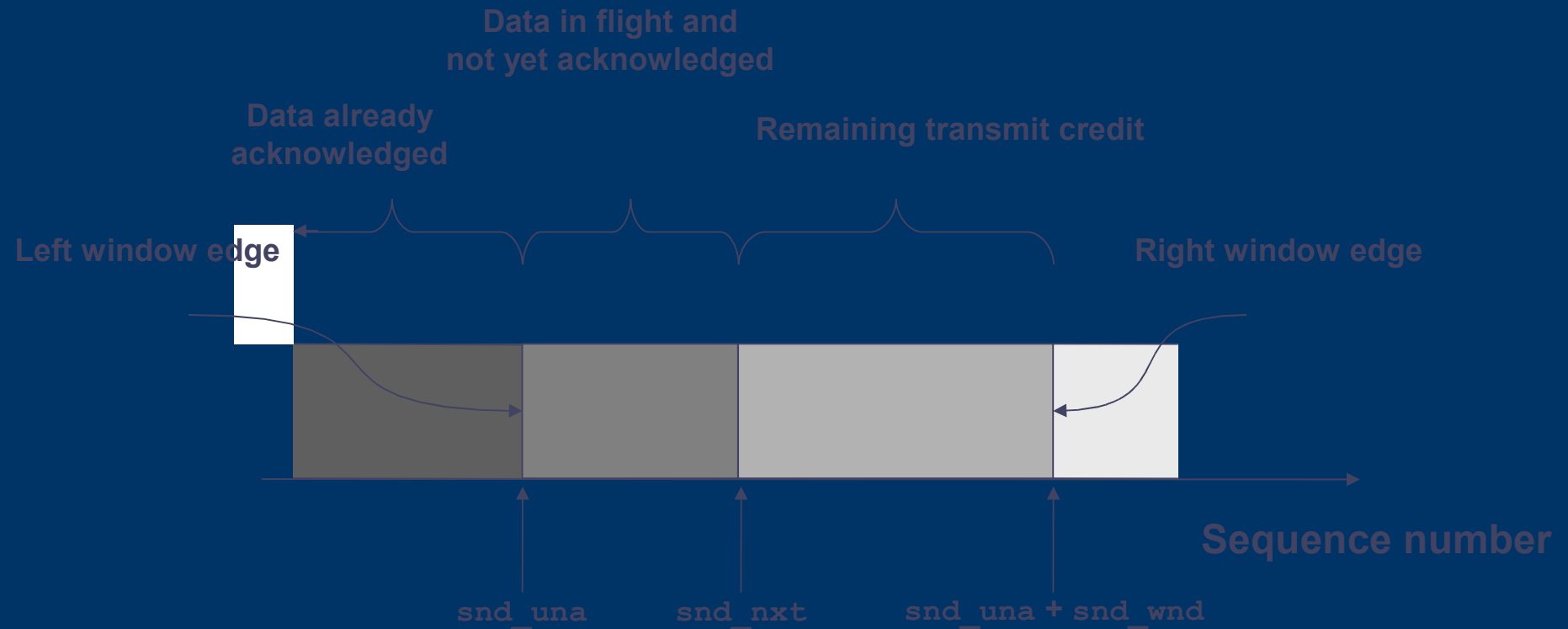
tcp_send_skb()

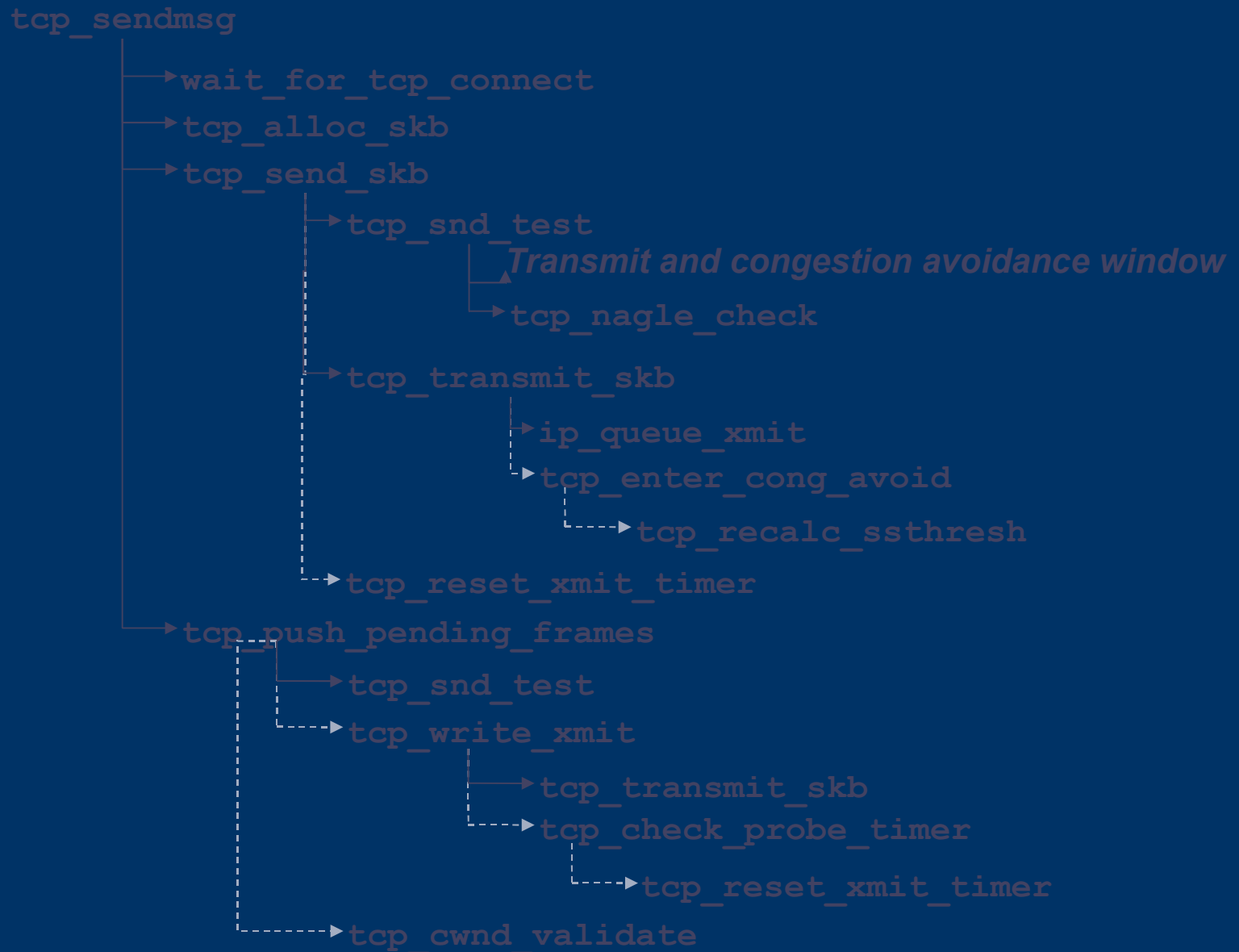
1. Adds the socket buffer, *skb*, to the transmit queue *sk*→*write_queue*
2. Invokes *tcp_snd_test()* to determine if the transmission can be started.
3. If so, invokes *tcp_transmit_skb()* to pass the segment to the IP layer.
4. Invokes *tcp_reset_xmit_timer()* for automatic retransmission.

tcp_snd_test()

```
static __inline__ int tcp_snd_test(struct tcp_opt *tp, struct  
    sk_buff *skb, unsigned cur_mss, int nonagle)  
{  
    return ((nonagle==1 || tp->urg_mode || !tcp_nagle  
        _check(tp, skb, cur_mss, nonagle)) &&  
        ((tcp_packets_in_flight(tp) < tp->snd_cwnd) ||  
            (TCP_SKB_CB(skb)->flags & TCPCB_FLAG_FIN))  
        && !after(TCP_SKB_CB(skb)->end_seq, tp->snd_una  
            + tp->snd_wnd));  
}
```

Window Kept at the Sender





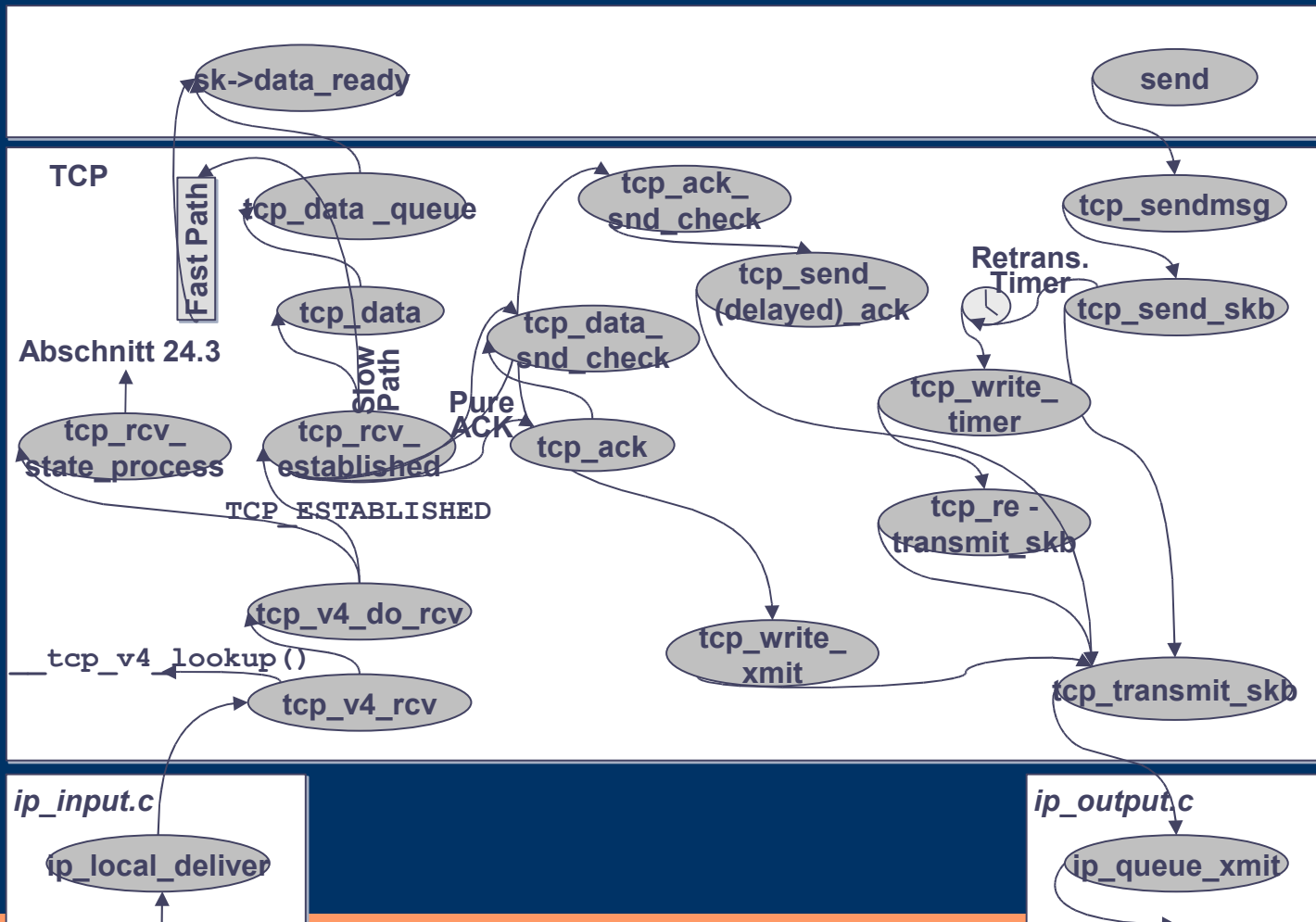
tcp_transmit_skb()

1. Fills the TCP header with the appropriate values from the *tcp_opt* structure.
 2. Invokes *tcp_syn_build_options()* to register the TCP options for a SYN packet and *tcp_build_and_update_options()* to register the option for all other packets.
 3. If ACK is set, the number of permitted *QuickAck* packets is decremented in *tcp_event_ack_sent()* method. The timer for delayed ACKs is stopped.
 4. If the segment contains payload, checks if the retransmission timer has expired. If so, the congestion window, *snd_cwnd*, is set to the minimum value (*tcp_cwnd_restart*).
-
-

tcp_transmit_skb()

1. Invokes *tp* → *af_specific* → *queue_xmit()* (i.e., *ip_queue_xmit()* for IPv4) to pass the socket buffer to the IP layer.
2. Invokes *tcp_enter_cwr()* to adapt the threshold value for the slow start algorithm (if the segment is the first segment of a connection).

TCP Implementation in Linux



tcp_push_pending_frames()

```
struct sk_buff *skb = tp->send_head;
```

```
if (skb) {  
    if (!tcp_skb_is_last(sk, skb))  
        nonagle = 1;  
    if (!tcp_snd_test(tp, skb, cur_mss, nonagle) || tcp  
        _write_xmit(sk, nonagle))  
        tcp_check_probe_timer(sk, tp);  
}  
tcp_cwnd_validate(sk, tp);
```

Continues to send segments
from the transmit queue of sk,
as long as it is allowed by
tcp_snd_test()

