Linux Networking Kernel

Version 0.1
February, 12, 2003

1 – Introduction

This report tries to describe the Networking part of the linux networking kernel. We try to describe the path the the packets follow in the forwarding path. We restrict this work to the IP code leaving other less used protocols like X.25 for other works. In the lower layers we also concentrate in the Ethernet protocol.

In the IP code we describe only the IPv4 code although the IPv6 code does not have many differences (bigger addressing, no fragmentation, etc).

Networking in the linux kernel

In Figure \ref{fig:tree} we can see where the relevant code is in the linux kernel.

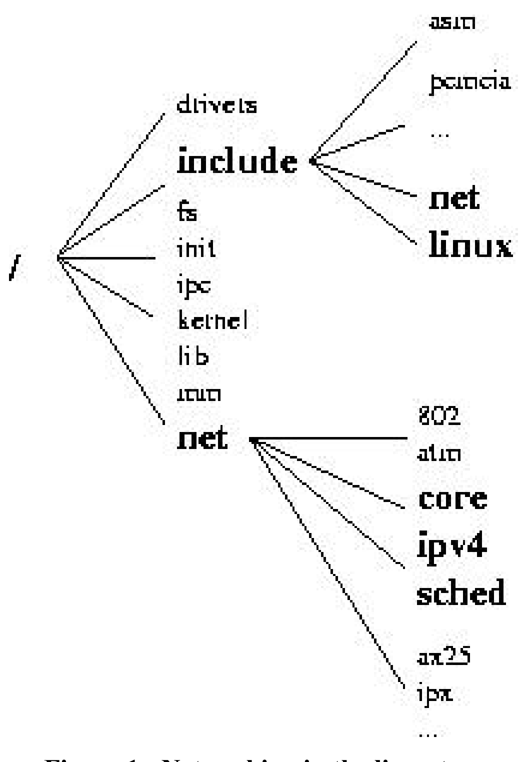


Figure 1 - Networking in the linux tree

Report Structure

The report follows a bottom-up approach when describing the kernel. A brief introduction to the most relevant data structures is presented in chapter ... In chapter ... the sub-ip layer is described. In chapter ... we describe the IP layer. TCP and UDP are described in chapters ...

The networking part of the kernel is populated with netfilter hooks where users can hang their code and analyse or change the packets. These are marked as HOOK in the kernel maps.

NOTE: We need References, specially RFCs (TCP, PAWS, SACKs, etc) and netfilter.
NOTE: We might need a final chapter on Socket layer and interation with the user space

Acknowledgments

We would like to thank .....

2 – Data Structures

The networking part of the kernel uses mainly two data structures. One to keep the state of a connection called sock and other to keep the status of each packet (incoming and/or outgoing) called *sk_buff*.

Here we proved a brief description of both. We also include a brief description of *tcp_opt*, a structure part of the sock structure which is used to maintain the tcp connection state. The details of tcp are presented in chapter \ref{chapter:tcp}.

sk_buff

The *sk_buff* structure is defined in *include/linux/skbuff.h* When a packet arrives to the kernel, either from the user space or from the network card one of these structures is created. Changing packet fields is achieved by changing its fields. In the next chapter practically every function is invoked with an *sk_buff* (the variable is usually called *skb*) as a parameter

The first fields are general ones. A pointer to the next and previous *skbs* in the list (packets are frequently put in lists or queues). The socket that owns the packet is stored in *sk* (note that if the packet is arriving from the network only at a later stage the socket owner is known).

The time of arrival is stored in stamp. The dev field stores the device the packet arrived and when and if the device to be used for transmission is known (for example by inspection of the routing table) the dev field is updated correspondingly.

```
struct sk_buff {
     /* These two members must be first. */
     struct sk_buff * next;             /* Next buffer in
list                     */
     struct sk_buff * prev;             /* Previous buffer
in list             */

     struct sk_buff_head * list;        /* List we are on
          */
     struct sock    *sk;           /* Socket we are owned by
          */
     struct timeval stamp;              /* Time we arrived
          */
     struct net_device   *dev;          /* Device we arrived
on/are leaving by        */
```

The transport section is a union that points to the corresponding transport layer structure (TCP, UDP, ICMP, etc).

```
/* Transport layer header */
union
{
     struct tcphdr  *th;
     struct udphdr  *uh;
     struct icmphdr *icmph;
     struct igmphdr *igmph;
     struct iphdr   *ipiph;
     struct spxhdr  *spxh;
     unsigned char  *raw;
} h;
```

The Network layer header points to the corresponding data structures (IPv4, IPv6, arp, raw, etc).

```
/* Network layer header */
union
{
     struct iphdr   *iph;
     struct ipv6hdr *ipv6h;
     struct arphdr  *arph;
     struct ipxhdr  *ipxh;
     unsigned char  *raw;
} nh;
```

The link layer is stored in this final union. Only a special case for ethernet is included. Other technologies will use the raw fields with appropriate casts.

```
/* Link layer header */
union
{
     struct ethhdr  *ethernet;
     unsigned char  *raw;
} mac;

struct  dst_entry *dst;
```

The rest of the packet info is stored in the rest of the structure. Length, data length, checksum, packet type, etc.

```
char         cb[48];
```

```
      unsigned int      len;              /* Length of actual data
            */
      unsigned int      data_len;
      unsigned int      csum;             /* Checksum
            */
      unsigned char     __unused,         /* Dead field, may be reused
            */
                  cloned,           /* head may be cloned (check refcnt
to be sure). */
                  pkt_type,         /* Packet class
      */
                  ip_summed;        /* Driver fed us an IP checksum
            */
      __u32       priority;         /* Packet queueing priority
      */
      atomic_t    users;                  /* User count - see
datagram.c,tcp.c        */
      unsigned short    protocol;         /* Packet protocol from
driver.           */
      unsigned short    security;         /* Security level of packet
            */
      unsigned int      truesize;         /* Buffer size
            */

      unsigned char     *head;                  /* Head of buffer
                  */
      unsigned char     *data;                  /* Data head pointer
                  */
      unsigned char     *tail;                  /* Tail pointer
                  */
      unsigned char     *end;             /* End pointer
            */
```

sock

The sock data structure keeps the state of a specific connection. When a socket is created in user space a sock structure is allocated. Data about the connection (TCP state for example).

The first fields contain source and destination addresses and ports.

```
struct sock {
      /* Socket demultiplex comparisons on incoming packets. */
      __u32             daddr;            /* Foreign IPv4 addr
      */
      __u32             rcv_saddr;  /* Bound local IPv4 addr
      */
      __u16             dport;            /* Destination port
      */
      unsigned short          num;        /* Local port
      */
      int               bound_dev_if;     /* Bound device index if != 0
      */
```

Among many fields the sock structure contains protocol specific information. These fields contain state information on each layer.

```
      union {
            struct ipv6_pinfo af_inet6;
      } net_pinfo;

      union {
            struct tcp_opt          af_tcp;
            struct raw_opt          tp_raw4;
            struct raw6_opt         tp_raw;
            struct spx_opt          af_spx;
      } tp_pinfo;

};
```

tcp_opt

One of the main components of the sock structure is the TCP field (tcp_opt). Both IP and UDP are stateless protocols with a minimum need to store information about their connections. TCP, however needs to store a big set of variables. They are stored in the fields of tcp_opt of which a relevant extract is shown below (comments are self-explanatory).

```
struct tcp_opt {
      int   tcp_header_len;   /* Bytes of tcp header to send
      */

      __u32 rcv_nxt;    /* What we want to receive next     */
      __u32 snd_nxt;    /* Next sequence we send            */

      __u32 snd_una;    /* First byte we want an ack for    */
      __u32 snd_sml;    /* Last byte of the most recently transmitted
small packet */
      __u32 rcv_tstamp; /* timestamp of last received ACK (for
keepalives) */
      __u32 lsndtime;   /* timestamp of last sent data packet (for
restart window) */

      /* Delayed ACK control data */
      struct {
            __u8  pending;    /* ACK is pending */
            __u8  quick;            /* Scheduled number of quick acks
      */
            __u8  pingpong;   /* The session is interactive       */
            __u8  blocked;    /* Delayed ACK was blocked by socket
lock*/
            __u32 ato;        /* Predicted tick of soft clock
      */
            unsigned long timeout;  /* Currently scheduled timeout
      */
            __u32 lrcvtime;   /* timestamp of last received data
packet*/
            __u16 last_seg_size;    /* Size of last incoming segment
      */
```

```
            __u16 rcv_mss;    /* MSS used for delayed ACK decisions
      */
      } ack;

      /* Data for direct copy to user */
      struct {
            struct sk_buff_head     prequeue;
            struct task_struct      *task;
            struct iovec            *iov;
            int               memory;
            int               len;
      } ucopy;

      __u32 snd_wl1;    /* Sequence for window update       */
      __u32 snd_wnd;    /* The window we expect to receive  */
      __u32 max_window; /* Maximal window ever seen from peer     */
      __u32 pmtu_cookie;      /* Last pmtu seen by socket         */
      __u16 mss_cache;  /* Cached effective mss, not including SACKS */
      __u16 mss_clamp;  /* Maximal mss, negotiated at connection setup
*/
      __u16 ext_header_len;   /* Network protocol overhead (IP/IPv6
options) */
      __u8  ca_state;   /* State of fast-retransmit machine       */
      __u8  retransmits;      /* Number of unrecovered RTO timeouts.
      */

      __u8  reordering; /* Packet reordering metric.        */
      __u8  queue_shrunk;     /* Write queue has been shrunk recently.*/
      __u8  defer_accept;     /* User waits for some data after accept()
*/

/* RTT measurement */
      __u8  backoff;    /* backoff                    */
      __u32 srtt;       /* smothed round trip time << 3           */
      __u32 mdev;       /* medium deviation                 */
      __u32 mdev_max;   /* maximal mdev for the last rtt period   */
      __u32 rttvar;           /* smoothed mdev_max                */
      __u32 rtt_seq;    /* sequence number to update rttvar */
      __u32 rto;        /* retransmit timeout               */

      __u32 packets_out;      /* Packets which are "in flight"    */
      __u32 left_out;   /* Packets which leaved network           */
      __u32 retrans_out;      /* Retransmitted packets out        */


/*
 *    Slow start and congestion control (see also Nagle, and Karn &
Partridge)
 */
      __u32 snd_ssthresh;     /* Slow start size threshold        */
      __u32 snd_cwnd;   /* Sending congestion window        */
      __u16 snd_cwnd_cnt;     /* Linear increase counter          */
      __u16 snd_cwnd_clamp; /* Do not allow snd_cwnd to grow above this
*/
      __u32 snd_cwnd_used;
      __u32 snd_cwnd_stamp;

      /* Two commonly used timers in both sender and receiver paths. */
      unsigned long           timeout;
      struct timer_list retransmit_timer; /* Resend (no ack)      */
      struct timer_list delack_timer;           /* Ack delay
      */
      struct sk_buff_head     out_of_order_queue; /* Out of order
segments go here */

      struct tcp_func         *af_specific;     /* Operations which are
AF_INET{4,6} specific   */
      struct sk_buff          *send_head; /* Front of stuff to transmit
            */
      struct page       *sndmsg_page;     /* Cached page for sendmsg
            */
      u32               sndmsg_off; /* Cached offset for sendmsg
      */

      __u32 rcv_wnd;    /* Current receiver window          */
      __u32 rcv_wup;    /* rcv_nxt on last window update sent     */
      __u32 write_seq;  /* Tail(+1) of data held in tcp send buffer */
      __u32 pushed_seq; /* Last pushed seq, required to talk to windows
*/
      __u32 copied_seq; /* Head of yet unread data          */
/*
 *      Options received (usually on last packet, some only on SYN
packets).
 */
      char  tstamp_ok,  /* TIMESTAMP seen on SYN packet           */
            wscale_ok,  /* Wscale seen on SYN packet        */
            sack_ok;    /* SACK seen on SYN packet          */
      char  saw_tstamp; /* Saw TIMESTAMP on last packet           */
        __u8      snd_wscale; /* Window scaling received from sender
      */
        __u8      rcv_wscale; /* Window scaling to send to receiver
      */
      __u8  nonagle;    /* Disable Nagle algorithm?             */
      __u8  keepalive_probes; /* num of allowed keep alive probes */

/*    PAWS/RTTM data    */
        __u32     rcv_tsval;  /* Time stamp value                 */
        __u32     rcv_tsecr;  /* Time stamp echo reply            */
        __u32     ts_recent;  /* Time stamp to echo next          */
        long      ts_recent_stamp;/* Time we stored ts_recent (for
aging) */

/*    SACKs data  */
      __u16 user_mss;   /* mss requested by user in ioctl */
      __u8  dsack;            /* D-SACK is scheduled              */
      __u8  eff_sacks;  /* Size of SACK array to send with next packet
*/
      struct tcp_sack_block duplicate_sack[1]; /* D-SACK block */
      struct tcp_sack_block selective_acks[4]; /* The SACKS themselves*/

      __u32 window_clamp;     /* Maximal window to advertise
      */
      __u32 rcv_ssthresh;     /* Current window clamp             */
      __u8  probes_out; /* unanswered 0 window probes       */
      __u8  num_sacks;  /* Number of SACK blocks            */
      __u16 advmss;           /* Advertised MSS           */

      __u8  syn_retries;      /* num of allowed syn retries */
      __u8  ecn_flags;  /* ECN status bits.                 */
      __u16 prior_ssthresh; /* ssthresh saved at recovery start   */
      __u32 lost_out;   /* Lost packets                     */
      __u32 sacked_out; /* SACK'd packets             */
      __u32 fackets_out;      /* FACK'd packets           */
      __u32 high_seq;   /* snd_nxt at onset of congestion   */

      __u32 retrans_stamp;    /* Timestamp of the last retransmit,
                         * also used in SYN-SENT to remember stamp of
                         * the first SYN. */
      __u32 undo_marker;      /* tracking retrans started here. */
      int   undo_retrans;     /* number of undoable retransmissions. */
      __u32 urg_seq;    /* Seq of received urgent pointer */
      __u16 urg_data;   /* Saved octet of OOB data and control flags */
      __u8  pending;    /* Scheduled timer event      */
      __u8  urg_mode;   /* In urgent mode       */
      __u32 snd_up;           /* Urgent pointer       */

};
```

3 – Lower Layers

This chapter describes the reception and handling of packets in the lower layers.

NOTE: Needs relevant files
NOTE: Needs some reordering to maybe follow the same structure

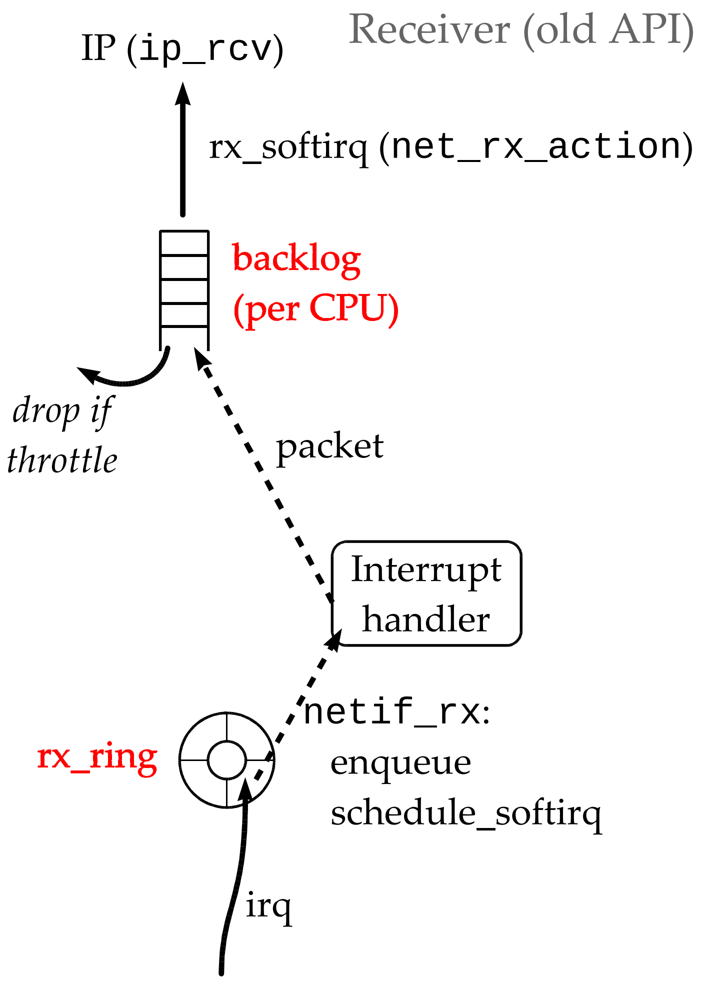


Figure 2 - Packet Reception with the old API

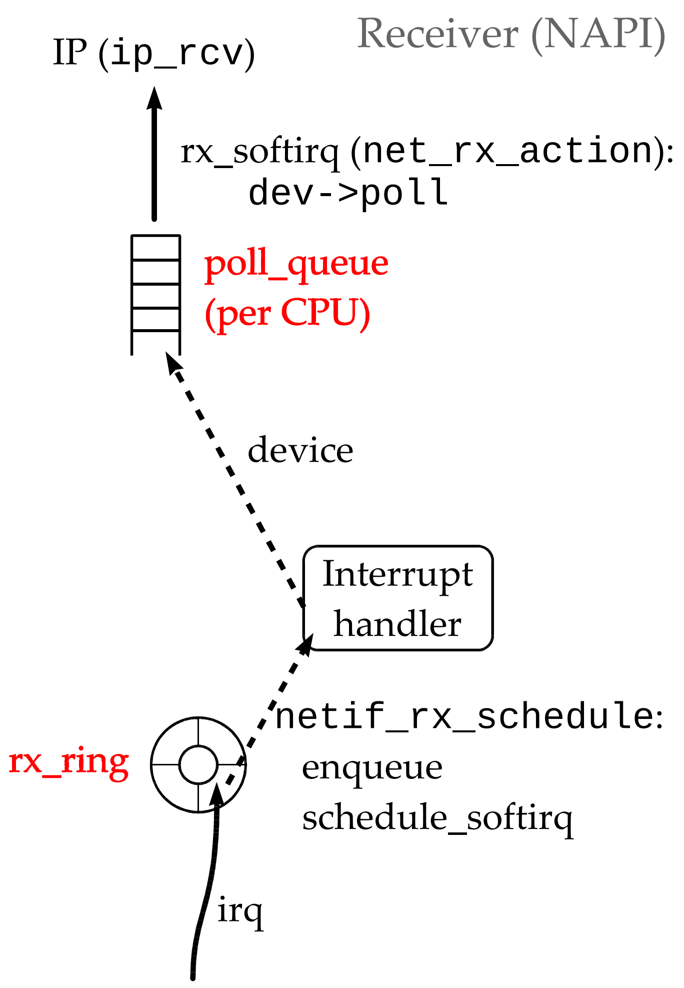


Figure 3 - Packet reception with the new API

When a packet is received the following steps are followed:

1. Packets are first received by the card. They are put in the *rx_ring* using DMA for recent cards. The *rx_ring* is a ring in the kernel memory where the card DMAs the incoming packets for the driver. The size of the ring is driver dependent. Older cards use the PIO scheme: it is the host CPU which transfers the data from the card into the host memory;

2. The card interrupts the CPU, which then jumps to the driver ISR code. Here arise some differences between the old subsystem (<= 2.4.19) and NAPI.

For the former, the interrupt handler calls the *netif_rx()* kernel procedure (*net/dev/core.c*,l. 1215). *netif_rx()* enqueues the received packet in the interrupted CPU's backlog queue and schedules a softirq (a kind of kernel thread, see `http://tldp.org/HOWTO/KernelAnalysis-HOWTO-5.html` or http://www.netfilter.org/unreliable-guides/kernel-hacking/lk-hacking-guide.html to know more about softirq), responsible for further processing of the packet (e.g. TCP/IP processing). The backlog is 300-packet long *(/proc/sys/net/core/netdev_max_backlog*). When it is full, it enters the throttle state and waits for being totally empty to reenter a normal state and allow again an enqueue (*netif_rx()*, *net/dev/core.c*). If the backlog is in the throttle state, *netif_rx* drops the packet. Backlog stats are available in */proc/net/softnet_stats*: one line per CPU, the first two columns are packets and drops counts. The third is the number of times the backlog entered the throttle state.

NAPI drivers act differently: the interrupt handler calls *netif_rx_schedule()* (*include/linux/netdevice.h*, l. 738). Instead of putting the packets in the backlog, it puts a reference to the device in a queue attached to the interrupted CPU (*softnet_data>poll_list*; *include/linux/netdevice.h*, l. 496). A softirq is schedules too, just like in the previous case. To insure backward compatibility, the backlog is considered as a device in NAPI, which can be enqueued just as an another card, to handle all the incoming packets. *netif_rx()* is rewritten to enqueue the backlog into the poll_list of the CPU after having enqueued the packet in the backlog;

3. When the softirq is scheduled, it executes *net_rx_action()* (*net/core/dev.c*, l. 1558). Since the last step differs between the older network subsystem and NAPI, this one does too.

For version <= 2.4.19, *net_rx_action* pulls all the packets in the backlog and calls for each of them the *ip_rcv()* procedure (*net/ipv4/ip_input.c*, l. 379) or another one depending on the type of the packet: arp, bootp, etc.

For NAPI, the CPU polls the devices present in his poll_list to get all the received packets from their rx_ring or from the backlog. The poll method of the backlog (process_backlog; *net/core/dev.c*, l. 1496) or of any device calls, for each received packet, *netif_receive_skb()* (*net/core/dev.c*, l. 1415) which roughly calls *ip_rcv().*

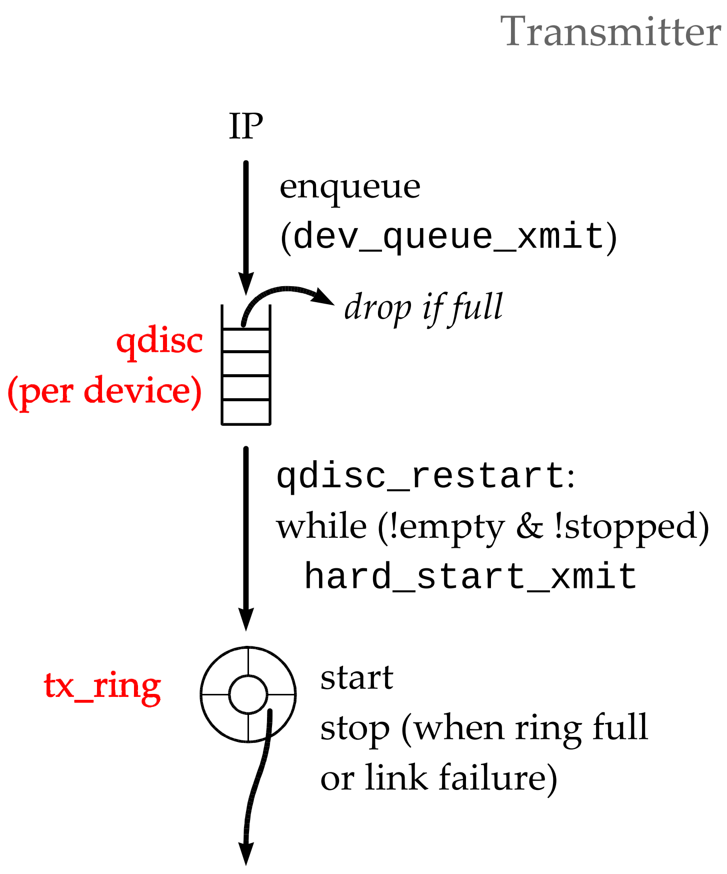


Figure 4 - Transmission of a packet

When a packet is created the following steps are followed:

- All the IP packets are built using the *arp_constructor()* method. Each packet contains a *dst* field, that provides the destination computed by the routing algorithm. The *dst* field provides an output method, which is *dev_queue_xmit()* for IP packets;

- The kernel provides lots of queueing disciplines between the kernel and the driver. It is intended to provide QoS support. The default queueing discipline is a FIFO queue. Default length is 100 packets (*ether_setup(): dev->tx_queue_len* ; *drivers/net/net_init.c*, l. 405). My understanding is that 'ifconfig' can override this value using the 'txqueuelen' option. You can't get stats for the default *qdisc*. The trick is to replace it with the same FIFO queue using the 'tc' command:
  - to replace the default qdisc, use : 'tc qdisc add dev eth0 root pfifo limit 100';
  - to get stats from this qdisc, use : 'tc -s -d qdisc show dev eth0';
  - to recover to default state, use : 'tc qdisc del dev eth0 root'.

1. For each packet to transmit from the IP layer, the *dev_queue_xmit()* procedure (*net/core/dev.c,l.* 991) is called. It queues a packet in the qdisc associated to the output

interface (determined by the routing). Then, if the device is not stopped (link failure, tx_ring full), all packets present in the qdisc are handled by *qdisc_restart()* (net/sched/sch_generic.c, l. 77);

2. The *hard_start_xmit()* virtual method is then called. This method is implemented in the driver code. The packet is placed in the tx_ring and the driver tells the card there are some packets to send;

3. Once the card has sent a packet or a group of packets, it communicates to the CPU that packets have been sent out. The CPU uses this information (*net_tx_action()*; *net/core/dev.c*, l. 1326) to put the packets into a completion_queue and to schedule a softirq for later deallocating the memory associated with these packets. This communication between the card and the CPU is card and driver-dependant, so I won't go into further details with respect to this.

4 – Network Layer

Introduction

The IP main files are:

- ip_input.c – Processing packets arriving to the host
- ip_output.c – Processing packets leaving the host
- ip_forward.c – Processing packets being routed by the host

Other less relevant files deal with IP packet fragmentation (*ip_fragment.c*), IP options (*ip_options.c*), multicast (*ipmr.c*) and IP over IP (*ipip.c*)

Figure \ref{fig:ip} describes the path that a packet traverses inside the IP linux layer. If the packet has reached the host from the network, it passes through the functions already described in chapter \ref{} until it reaches *net_rx_action()* that passes the packet to *ip_rcv()*. After passing the first *netfilter* hook (see chapter \ref{}) the packet reaches *ip_rcv_finish()* which verifies if the packet is for local delivery (it is addressed to this host) giving it to *ip_local_delivery()*. *ip_local_delivery()* will give it to the appropriate transport layer function (tcp, udp, etc).

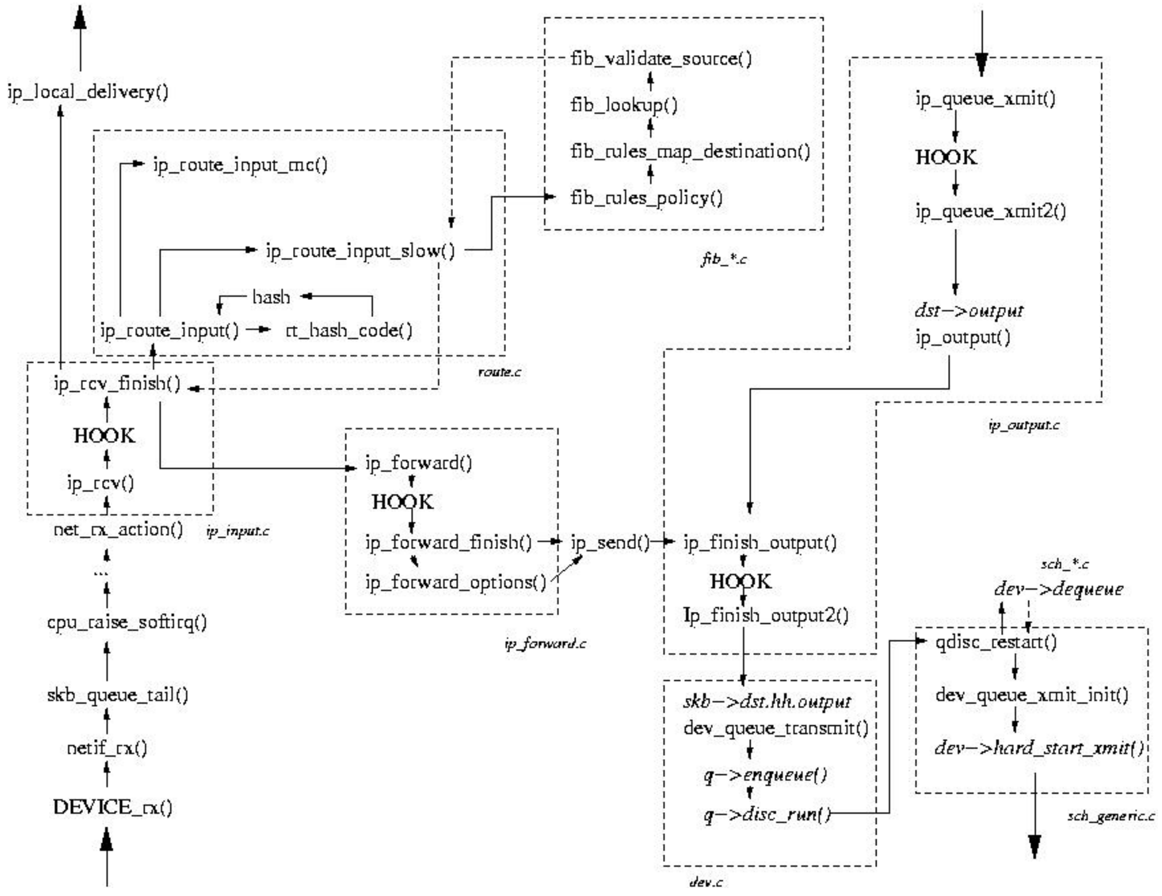


Figure 5 - Network layer data path

If the packet has other host as destination this means that our host is acting as a router (a frequent scenario in small networks). If the host is configured to execute forwarding (this can be sees and set through the value of the */proc/sys/net/ipv4/ip_forward* variable) it then has to be processed by a set of complex but very efficient functions.

The route is calculated by calling *ip_route_input()* which (if a fast hash does not exist) calls *ip_route_input_slow()*. *ip_route_input_slow()* calls the fib (Forward information base) set of functions in the fib*.c files. The FIB structure is quite complex (see for example \cite{}).

NOTE: Maybe we need a picture and a better explanation of this.

If the packet is a multicast packet the function that calculates the set of devices to transmit the packet (in this case the IP destination is unchanged) is *ip_route_input_mc()*.

After the route is calculated .... inserts the new IP destination in the IP packet and the output device in the *sk_buff* structure. The packet is then passed to the forwarding functions (*ip_forward()* and *ip_forward_finish()* which sends it to the output components (more about this in a bit).

A packet can also reach the IP layer coming from the upper layers (delivered by TCP or UDP). The first function to process the packet is *ip_queue_xmit()* which passes the packet to the output part through *ip_output()*.

In the output part the last changes on the packet are made... and the function *dev_queue_transmit()* is called which enqueues the packet in the output queue. It also tries to run the network scheduler mechanism by calling q$\-\>$disc$\-\>$run. This pointer will point to different functions depending on the scheduler installed (by default a fifo scheduler is installed but this can be changed with the tc utility).

The scheduling functions (*qdisc_restart()* and *dev_queue_xmit_init()*) run independently from the rest of the IP code.

NOTE: This chapter needs sections on: ARP, ICMP, Multicast (both IGMP and multicast routing),
NOTE: packet fragmentation and IPv6. None of these is very complicated.

5 – TCP

This chapter describes what is possibly the most complex part of the networking linux kernel: The TCP part.

Introduction

The main files that handle the tcp part are:

- tcp_input.c - This is the biggest one. It deals with incoming packets from the network.
- tcp_output.c - This deals with sending packets to the network
- tcp.c - General TCP code
- tcp_ipv4.c - IPv4 TCP specific code
- tcp_timer.c - Timer management
- tcp.h - TCP definitions

In Figures … and … we can see the TCP data path. On the left side the input processing and on the right side the output processing.

NOTE: These two figures should be in opposite pages

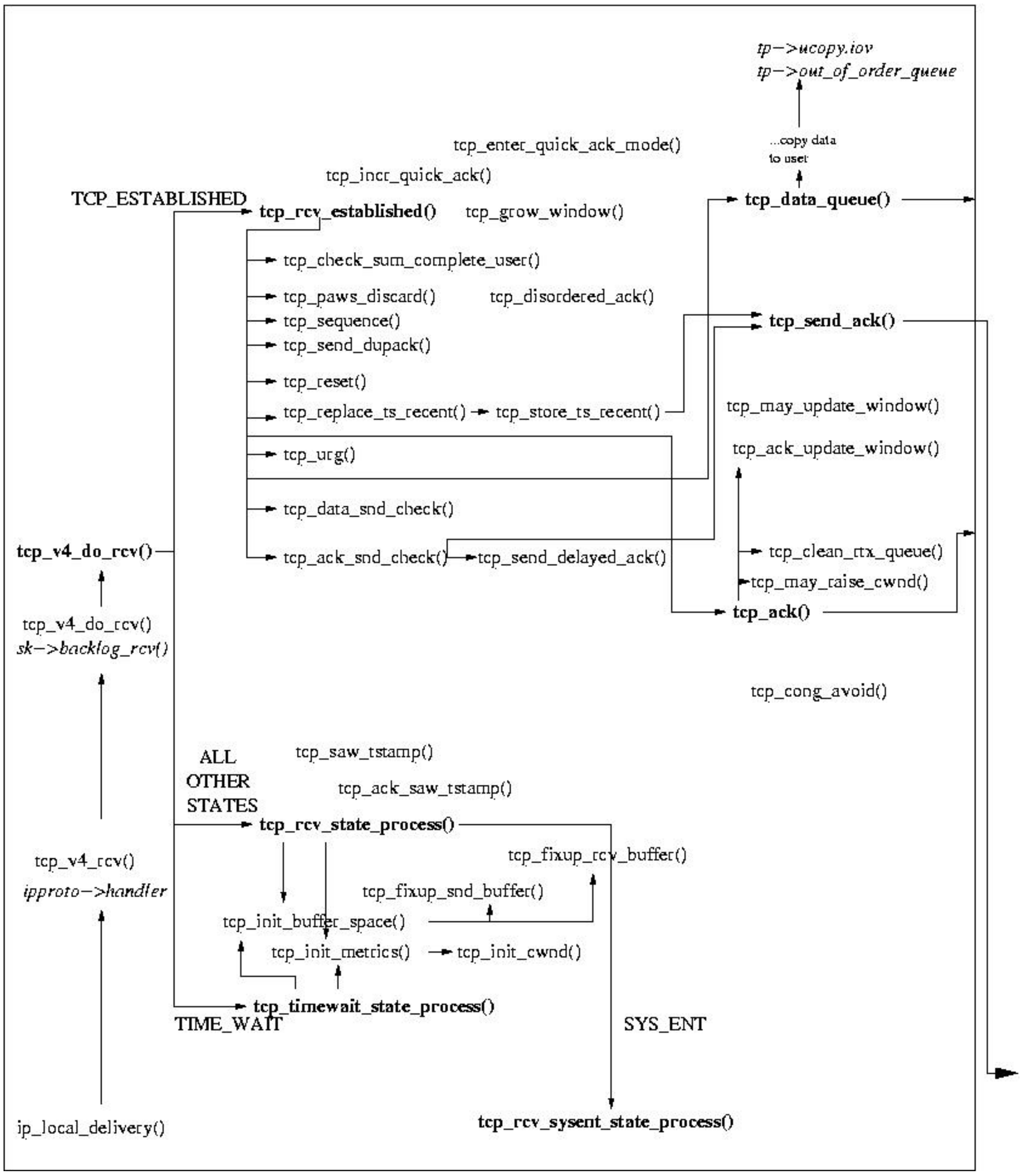


Figure 6 - TCP (part 1)

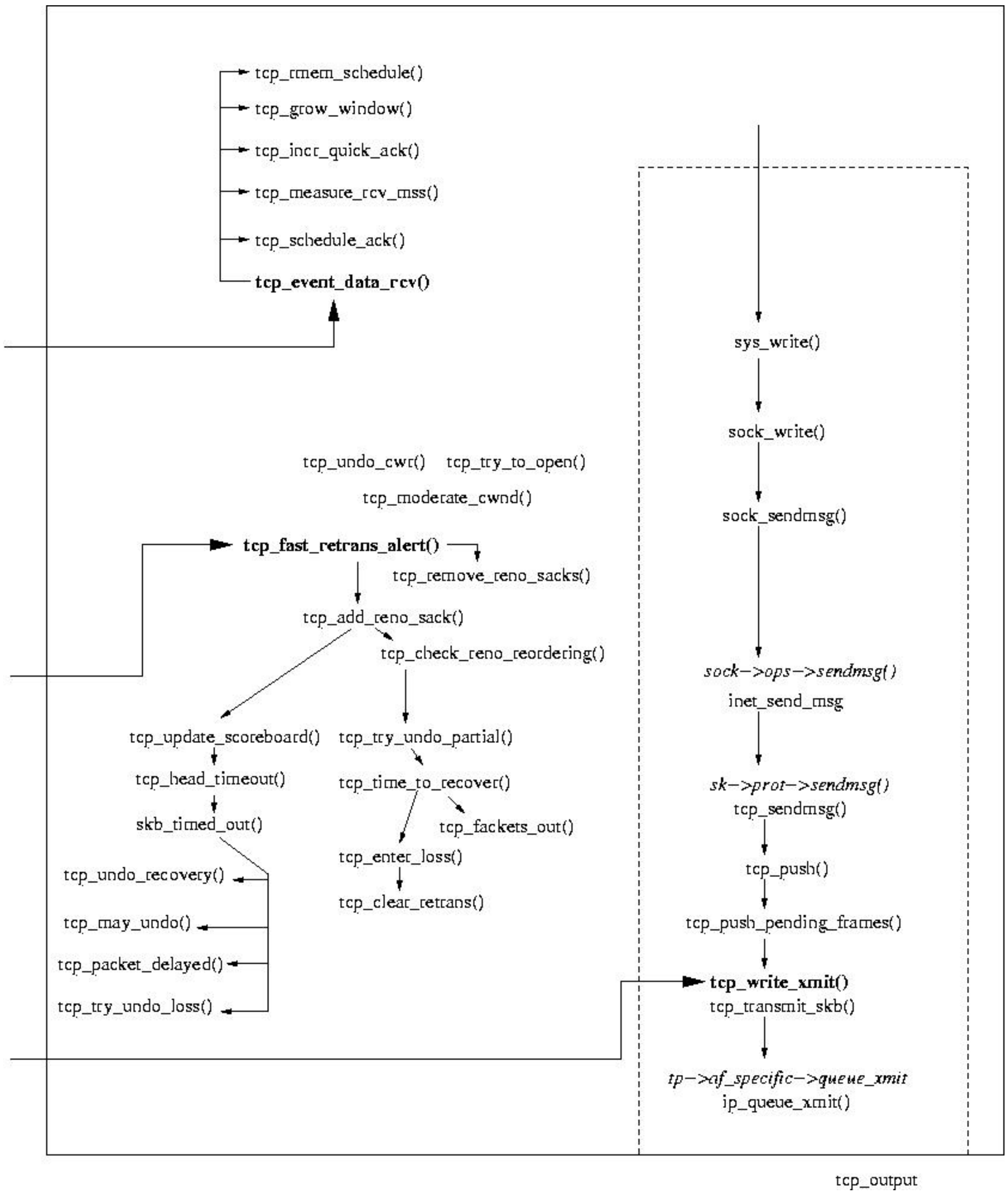


Figure 7 - TCP (part2)

TCP Input (mainly *tcp_input.c* }

This is the main sector of the TCP implementation. It deals with the reception of a TCP packet. The sender and receiver code is mixed up since an entity can be both at the same time.

The packet arrives to the TCP sector from the IP layer though *ip_local_deliver()* (left side of Figure~\ref{fig:tcp1}). This, gives the packet to the function pointed by *ipproto-*

>*handler*. In this case we are looking at the IPv4 implementation this is *tcp_v4_rcv()*. This calls *tcp_v4_do_rcv()*.

Depending on the TCP state the connection is a different function is called. If the connection is established (state is TCP_ESTABLISHED) it calls *tcp_rcv_established()*. This is the main case that we will look from now on. If the state is TIME_WAIT it calls *tcp_timewait_process()*.

I many other case it calls *tcp_rcv_state_process()*. This function will call *tcp_rcv_sysent_state_process()* if the state is SYS_ENT.

tcp_rcv_state_process() and *tcp_timewait_process()* have to initialise the TCP structures. They call *tcp_init_buffer_space()* and *tcp_init_metrics()*. *tcp_init_metrics()* initialises the congestion window by calling *tcp_init_cwnd()*.

tcp_rcv_established()

tcp_rcv_established() has to ways of operation: fast path and slow path. We first follow the slow path since it is more clear and leave the fast path for the end (note that in the code the fast path is dealt with before)

slow path

The slow path code follows the 7 steps on RFC ... with some other operations:

- The checksum is calculated with *tcp_checksum_complete_user()* . If is is not correct the packet is discarded.

- The PAWS (Protection Against Wrapped Sequence Numbers) is done with *tcp_paws_discard()*.

- STEP 1 - The sequence number of the packet is checked. If it is not in sequence the receiver sends a DUPACK with *tcp_send_dupack()*. *tcp_send_dupack()* may have to implement a SACK (*tcp_dsack_set()*) but if finishes by calling *tcp_send_ack()*.

- STEP 2 - Check the RST bit (*th->rst*) and if it is on calls *tcp_reset()*.

- STEP 3 - Check security and precedence (this is not implemented)

- STEP 4 - Check SYN bit. If it is on calls *tcp_reset()*...

- Calculate an estimative for the RTT (RTTM) by calling *tcp_replace_ts_recent()*

- STEP 5 - Check ACK field. If this is on, the packet brings an ACK and *tcp_ack()* is called (more details in section {sec:ta} below)
- STEP 6 - Check URG bit. If it is on call *tcp_urg()*
- STEP 7 - Process data on the packet. This is done by calling *tcp_data_queue()* (more details in Section \ref{sec:tdq} below).
- Checks if there is data to send by calling *tcp_data_snd_check()*. This function calls *tcp_write_xmit()* on the TCP output sector.
- Finally, check if there are ACKs to send with *tcp_ack_snd_check()*. This may result in sending an ACK straight away with *tcp_send_ack()* or scheduling a delayed ACK with *tcp_send_delayed_ack()*. The delayed ACK is stored in *tcp->ack.pending* .

tcp_data_queue()

tcp_data_queue() is the function responsible with giving the data to the user. If the packet arrived in order (all previous packets had already arrived) it copies the data to *tp->ucopy.iov* (*skb_copy_datagram_iovec(skb, 0, tp->ucopy.iov, chunk)*).

If the packet did not arrive in order it puts it in the out of order queue with *tcp_ofo_queue()*.

If a gap in the queue is filled RFC 2581 (section 4.2) \cite{} says to send an ACK immediately (*tp-$>$ack.pingpong = 0* and *tcp_ack_snd_check()* will send the ACK now).

The arrival of a packet has several consequences. These are dealt by calling *tcp_event_data_recv()*. It first schedules an ACK with *tcp_schedule_ack()*. It then estimates the MSS (Maximum Segment Size...) with *tcp_measure_rcv_mss()*.
In certain conditions (e.g we are in slow start) the receiver TCP should be in quickack mode (no delayed ACKS) and this function switches this on with *tcp_incr_quickack()*.
Finally it may have to increase the advertised window with *tcp_grow_window()*.

Finally *tcp_data_queue()* checks if the FIN bit is on, and if yes *tcp_fin()* is called.

tcp_ack()

Every time an ACK is received (this is the "sender" part) *tcp_ack()* is called. Not to confuse with *tcp_send_ack()* called by the "receiver" which calls *tcp_write_xmit()* to send ACKs.

The first thing it does is to check if the ACK is newer than sent or older than previous acks then we can probably ignore it *goto uninteresting_ack* and *goto old_ack*)

If everything is normal it updates the sender congestion window with *tcp_ack_update_window()* and/or *tcp_update_wl()*.

NOTE: When is everything normal ?

If the ACK is dubious (e.g there was nothing non-acknowledged on the retransmission queue, or ???) enter fast retransmit with *tcp_fastretrans_alert()* (see Section … below).

NOTE: In what exact conditions is fast_retransmit() called ?

If (???) enter slow start/congestion avoidance with *tcp_cong_avoid()*. This functions implements both the exponential increase in slow start and the linear increase in congestion avoidance.

NOTE: In which conditions is tcp_cong_avoid() is entered ?

tcp_fast_retransmit()

The *tcp_fast_retransmit_alert()* is entered from only one point (from *tcp_ack()*) in certain conditions. To understand these conditions we have to go through the Linux NewReno/SACK/FACK/ECN state machine. What follows is practically a copy of a comment in *tcp_input.c*. Note that this has nothing to do with the TCP state machine. The TCP state is almost certainly TCP_ESTABLISHED.

The Linux State machine can be:

- "Open" - Normal state, no dubious events, fast path.
- "Disorder" -In all the respects it is "Open",but requires a bit more attention. It is entered when we see some SACKs or dupacks. It is split of "Open" mainly to move some processing from fast path to slow one.
- "CWR" - CWND was reduced due to some Congestion Notification event. It can be ECN, ICMP source quench, local device congestion.
- "Recovery" - CWND was reduced, we are fast-retransmitting.
- "Loss" - CWND was reduced due to RTO timeout or SACK reneging.

The state is kept in *tp->ca_state* as TCP_CA_Open, TCP_CA_Disorder, TCP_CA_Cwr, TCP_CA_Recover or TCP_CA_Loss.

tcp_fastretrans_alert() is entered if state is not "Open" when an ACK is received or "strange" ACKs are received (SACK, DUPACK ECN ECE).

NOTE:In some non-"Open" conditions from is not entered...check this

fast path

The fast path is entered in certain conditions. For example a receiver usually enters the fast path since the TCP processing in the receiver side is much more simple. However this is not the only case.

NOTE: When is fast path entered and which steps does it follow.

SACKs

The Linux TCP implementation completely implements SACKS (selective ACKs) \cite{}. The connection SACK capabilities are stored in the *tp->sack_ok* field (FACKs are enabled if the 2 bit is one and DSACKS (delayed SACKS) are enabled if the 3 bit is 1).

SACKS occupies an unexpected great part of the TCP implementation. More than a dozen functions and significant parts of other functions are dedicated to implement SACKS.

NOTE: SACKs need more work...
NOTE: What are FACKs ?

quickacks

At certain times, the receiver enter quickack mode. That is, delayed ACKS are disabled. One example is in slow start when delaying ACKs would delay the slow start considerably.
tcp_enter_quick_ack_mode() is called by *tc_rcv_sysent_state_process()* since in the beginning of the connection this should be the state.

NOTE: When does the receiver enter quickack mode ?

Timeouts

Timeouts are vital for the correct behaviour of the TCP functions. They are used, for example, to infer a packet loss in the network. Here we trace the events on registering and triggering of the RETRANSMIT timer. These can be seen in Figures \ref{fig:timout1} and \ref{fig:timout2}

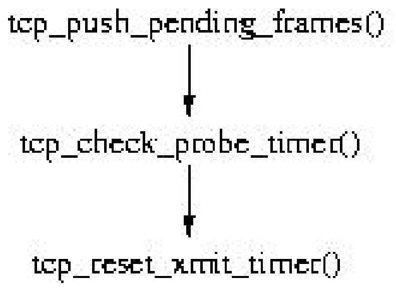
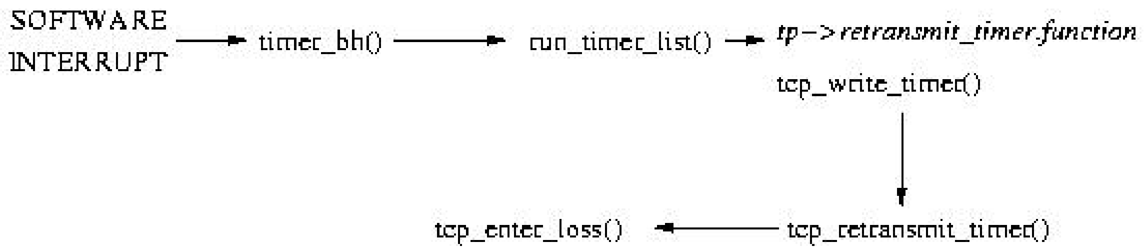


Figure 8 - Schedulling a timeout

The setting of the retransmit timer happens when a packet is sent (which will be seen in more detail in section \ref{}). *tcp_push_pending_frames()* calls *tcp_check_probe_timer()* which may call *tcp_reset_xmit_timer()*. This will schedule a software interrupt (which are dealt by non-networking parts of the kernel).

When the timeout expires a software interrupt is generated which calls *timer_bh()* which calls *run_timer_list()*. This will call *timer->function* which will, in this case, pointing to *tcp_wite_timer()*. This will call *tcp_retransmit_timer()* which will finally call *tcp_enter_loss()*. The state of the Linux machine will be set to CA_Loss and the *fastretransmit_alert()* function will schedule the retransmission of the packet.

NOTE: Why is fast retransmit called here ?



ECN

The ECN (Explicit Congestion Notification) \cite{} code is not difficult to trace. Almost all the code is in the *tcp_ecn.h* in the /include/net directory. It contains the code to receive and send the several ECN packet types.

On the packet input processing side we have some references:

- tcp_ack() which checks calls TCP_ECN_rcv_ecn_echo() to possibly process a ECN packet.
-
-

NOTE: We need some more detail about ECN.

TCP output

This part of the code (mainly *tcp_output.c*) deals with packets going out (both data packets from the "sender" and ACKs from the "receiver"). This is negotiated in the first SYN packets (as any other TCP option) (see *tcp_init_metrics()*).

NOTE: The TCP section might gain from a section on connection establishment and release.

6 – UDP

This chapter describes briefly the UDP part of the networking kernel. This is a significant simple piece of code than the TCP part. The absence of reliable delivery and congestion control allows a very simple design.

Introduction

The UDP code is mainly in one file:

- net/ipv4/udp.c

The UDP layer can be seen in Figure .... When a packet arrives from the IP layer through *ip_local_delivery()* it is passed to *udp_rcv()* (this is the equivalent of *tcp_v4_rcv()* in the TCP part). *udp_rcv()* puts the packet in the socket queue with *sock_put()*. This is the end of the delivery of the packet. The user will then call *inet_recvmsg()* (with the *recvmsg()* system call) which will, in this case, call *udp_recvmsg()* which calls *skb_rcv_datagram()*. This function will get the packets from the queue and fill the data structure that will be read in user space.

When a packet arrives. from the user the process is simpler. *inet_sendmsg()* calls *udp_sendmsg()* which build the UDP datagram with information taken from the *sk* structure (this information was put there when the socket was created and bound to the address).

After the UDP datagram is built it is passed to *ip_build_xmit()* which builds the IP packet with the possible help of *ip_build_xmit_slow()*.

NOTE: Why does UDP and tcp use different function on the IP layer ?

After the IP packet is built it is passed to *ip_output()* which, as was seen in chapter ..., finalises the delivery of the packet to the lower layers.

```
\begin{figure}
\epsfxsize=100mm
\centerline{\epsfbox{udp.eps}}
\caption{UDP}
\label{fig:udp}
\end{figure}
```

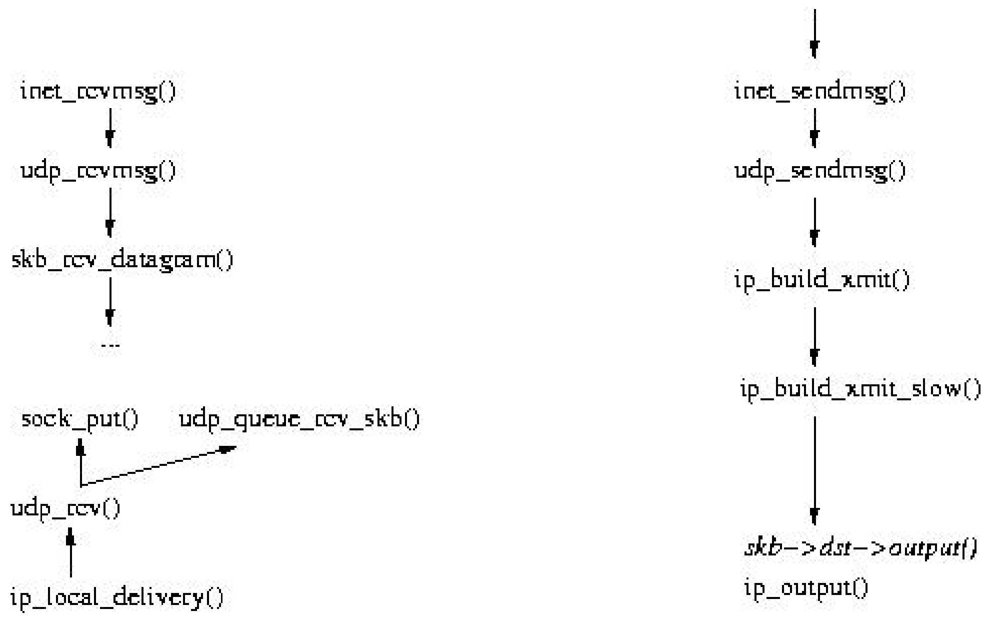

inet_rcvmsg()
udp_rcvmsg()
skb_rcv_datagram()
...
sock_put()
udp_queue_rcv_skb()
udp_rcv()
ip_local_delivery()
inet_sendmsg()
udp_sendmsg()
ip_build_xmit()
ip_build_xmit_slow()
skb->dst->output()
ip_output()

Figure 9 - UDP