Research & Technological Development for a TransAtlantic Grid

DataTAG

Technical Report DataTAG-2004-1
FP5/IST DataTAG Project

A Map of the Networking Code
in Linux Kernel 2.4.20

M. Rio et al.

31 March 2004

www.datatag.org

A Map of the Networking Code in Linux Kernel 2.4.20

Technical Report DataTAG-2004-1, 31 March 2004

Miguel Rio
Department of Physics and Astronomy
University College London
Gower Street
London WC1E 6BT
UK
E-mail: M.Rio@ee.ucl.ac.uk
Web: http://www.ee.ucl.ac.uk/mrio/

Mathieu Goutelle
LIP Laboratory, INRIA/ReSO Team
ENS Lyon
46 allée d'Italie
69364 Lyon Cedex 07
France
E-mail: Mathieu.Goutelle@ecl2002.ec-lyon.fr
Web: http://perso.ens-lyon.fr/mathieu.goutelle/

Tom Kelly
Laboratory for Communication Engineering
Cambridge University
William Gates Building
15 J.J. Thomson Avenue
Cambridge CB3 0FD
UK
E-mail: ctk21@cam.ac.uk
Web: http://www-lce.eng.cam.ac.uk/~ctk21/

Richard Hughes-Jones
Department of Physics and Astronomy
University of Manchester
Oxford Road
Manchester M13 9PL
UK
E-mail: R.Hughes-Jones@man.ac.uk
Web: http://www.hep.man.ac.uk/~rich/

Jean-Philippe Martin-Flatin
IT Department
CERN
1211 Geneva 23
Switzerland
E-mail: jp.martin-flatin@ieee.org
Web: http://cern.ch/jpmf/

Yee-Ting Li
Department of Physics and Astronomy
University College London
Gower Street
London WC1E 6BT
UK
E-mail: ytl@cs.ucl.ac.uk
Web: http://www.hep.ucl.ac.uk/~ytl/

Abstract

In this technical report, we describe the structure and organization of the networking code of Linux kernel 2.4.20. This release is the first of the 2.4 branch to support network interrupt mitigation via a mechanism known as NAPI. We describe the main data structures, the sub-IP layer, the IP layer, and two transport layers: TCP and UDP. This material is meant for people who are familiar with operating systems but are not Linux kernel experts.

Contents

1 Introduction

When we investigated the performance of gigabit networks and end-hosts in the DataTAG testbed, we soon realized that some losses occurred in end-hosts, and that it was not clear where these losses occurred. To get a better understanding of packet losses and buffer overflows, we gradually built a picture of how the networking code of the Linux kernel works, and instrumented parts of the code where we suspected that losses could happen unnoticed.

This report documents our understanding of how the networking code works in Linux kernel 2.4.20 [1]. We selected release 2.4.20 because, at the time we began writing this report, it was the latest stable release of the Linux kernel (2.6 had not been released yet), and because it was the first sub-release of the 2.4 tree to support NAPI (New Application Programming Interface [4]), which supports network interrupt mitigation and thereby introduces a major change in the way packets are handled in the kernel. Until 2.4.20 was released, NAPI was one of the main novelties in the development branch 2.5 and was only expected to appear in 2.6; it was not supported by the 2.4 branch up to 2.4.19 included. For more introductory material on NAPI and the new networking features expected to appear in Linux kernel 2.6, see Cooperstein's online tutorial [5].

In this document, we describe the paths through the kernel followed by IP (Internet Protocol) packets when they are received or transmitted from a host. Other protocols such as X.25 are not considered here. In the lower layers, often known as the sub-IP layers, we concentrate on the Ethernet protocol and ignore other protocols such as ATM (Asynchronous Transfer Mode). Finally, in the IP code, we describe only the IPv4 code and let IPv6 for future work. Note that the IPv6 code is not vastly different from the IPv4 code as far as networking is concerned (larger address space, no packet fragmentation, etc).

The reader of this report is expected to be familiar with IP networking. For a primer on the internals of the Internet Protocol (IP) and Transmission Control Protocol (TCP), see Stevens [6] and Wright and Stevens [7]. Linux kernel 2.4.20 implements a variant of TCP known as NewReno, with the congestion control algorithm specified in RFC 2581 [2], and the selective acknowledgment (SACK) option, which is specified in RFCs 2018 [8] and 2883 [9]. The classic introductory books to the Linux kernel are Bovet and Cesati [10] and Crowcroft and Phillips [3]. For Linux device drivers, see Rubini *et al.* [11].

In the rest of this report, we follow a bottom-up approach to investigate the Linux kernel. In Section 2, we give the big picture of the way the networking code is structured in Linux. A brief introduction to the most relevant data structures is given in Section 3. In Section 4, the sub-IP layer is described. In Section 5, we investigate the network layer (IP unicast, IP multicast, ARP, ICMP). TCP is studied in Section 6 and UDP in Section 7. The socket Application Programming Interface (API) is described in Section 8. Finally, we present some concluding remarks in Section 9.

2   Networking Code: The Big Picture

Figure 1 depicts where the networking code is located in the Linux kernel. Most of the code is in `net/ipv4`. The rest of the relevant code is in `net/core` and `net/sched`. The header files can be found in `include/linux` and `include/net`.
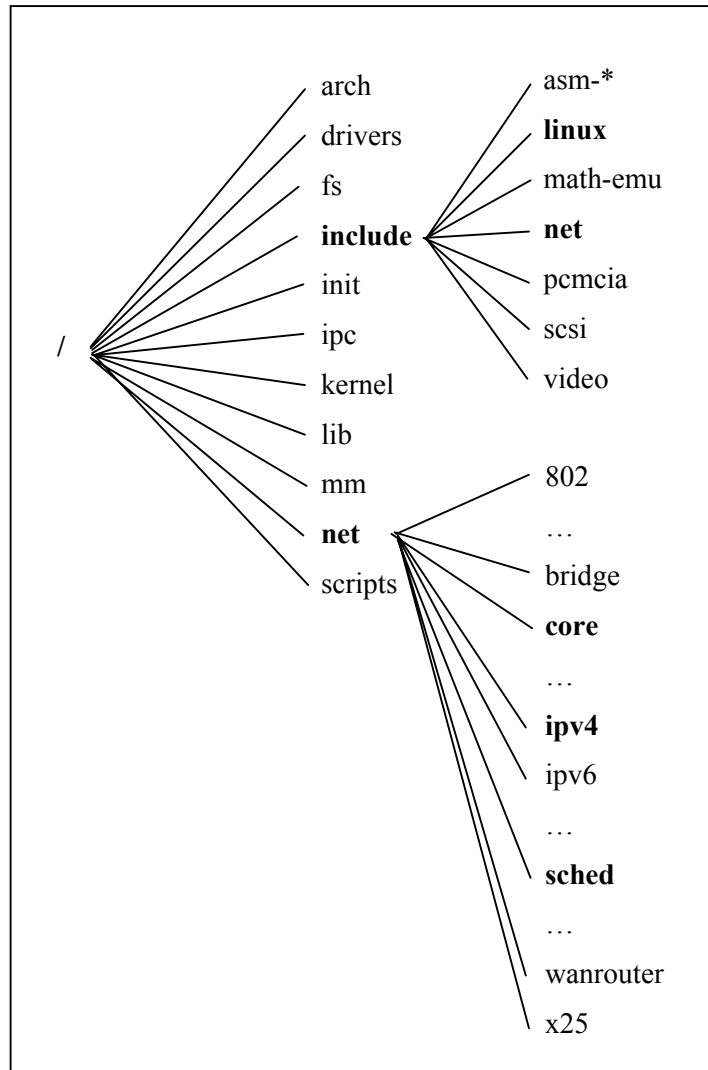
Figure 1: Networking code in the Linux kernel tree

The networking code of the kernel is sprinkled with *netfilter* hooks [16] where developers can hang their own code and analyze or change packets. These are marked as "HOOK" in the diagrams presented in this document.

Figure 2 and Figure 3 present an overview of the packet flows through the kernel. They indicate the areas where the hardware and driver code operate, the role of the kernel protocol stack and the kernel/application interface.

Figure 2: Handling of an incoming TCP segment

Figure 3: Handling of an outgoing TCP segment

3 General Data Structures

The networking part of the kernel uses mainly two data structures: one to keep the state of a connection, called *sock* (for "socket"), and another to keep the data and status of both incoming and outgoing packets, called *sk_buff* (for "socket buffer"). Both of them are des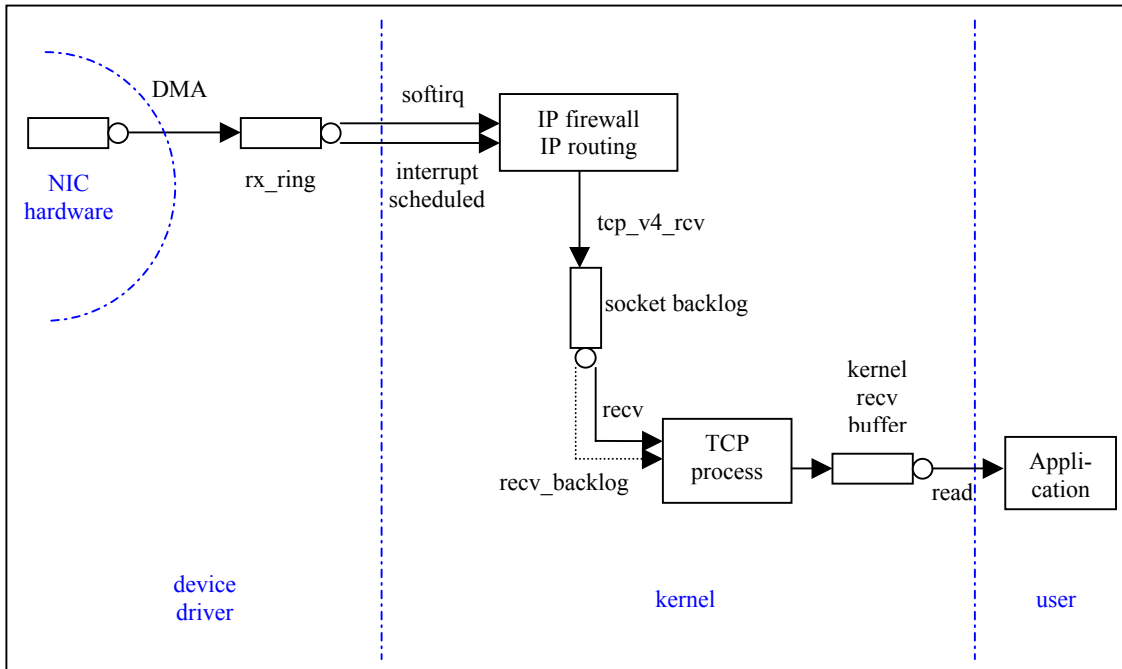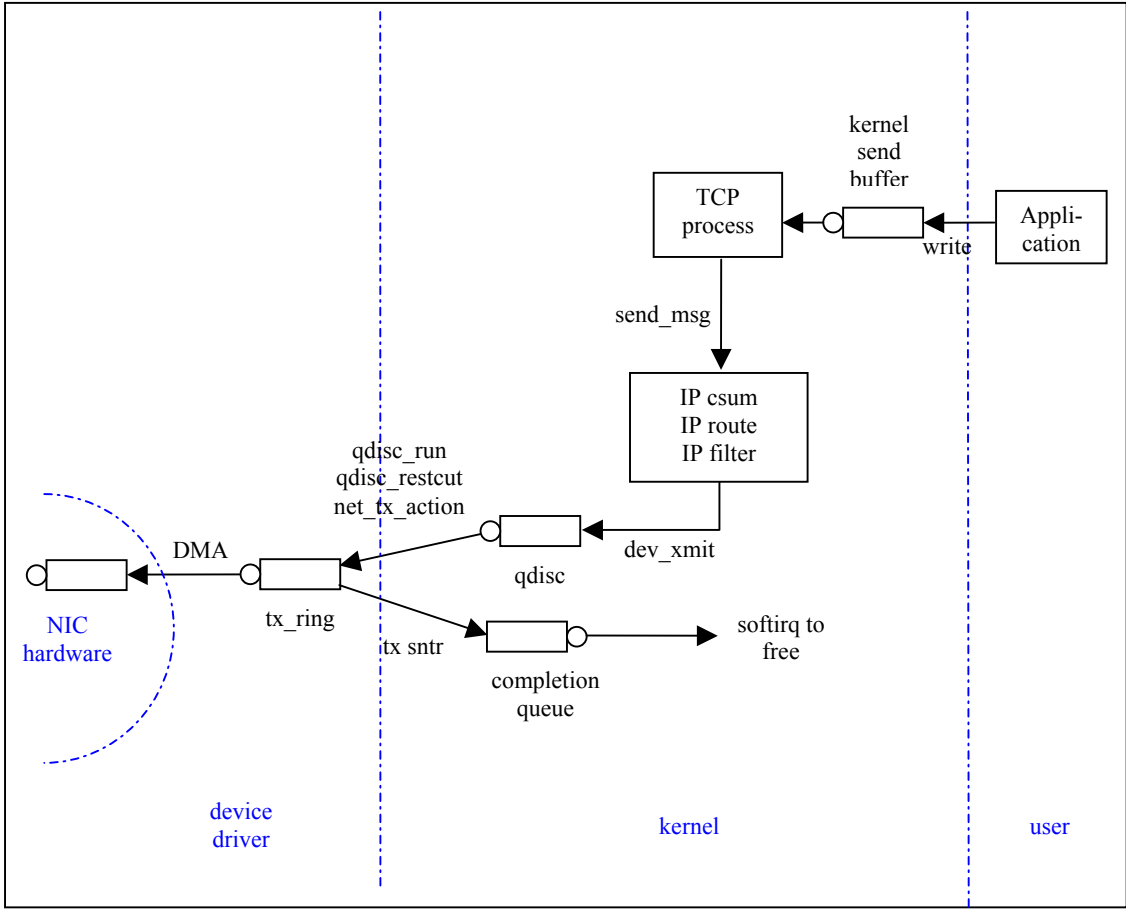cribed in this section. We also include a brief description of *tcp_opt*, a structure that is part of the *sock* structure and is used to maintain the TCP connection state. The details of TCP will be presented in section 6.

3.1 Socket buffers

The *sk_buff* data structure is defined in `include/linux/skbuff.h`.

When a packet is processed by the kernel, coming either from user space or from the network card, one of these data structures is created. Changing a field in a packet is achieved by updating a field of this data structure. In the networking code, virtually every function is invoked with an *sk_buff* (the variable is usually called *skb*) passed as a parameter.

The first two fields are pointers to the next and previous *sk_buff's* in the linked list (packets are frequently stored in linked lists or queues); *sk_buff_head* points to the head of the list.

The socket that owns the packet is stored in *sk* (note that if the packet comes from the network, the socket owner will be known only at a later stage).

The time of arrival is stored in a timestamp called *stamp*. The *dev* field stores the device from which the packet arrived, if the packet is for input. When the device to be used for transmission is known (for example, by inspection of the routing table), the *dev* field is updated correspondingly (see sections 4.1 and 4.3).

```
 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 a union called *mac*. 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;
```

Extra information about the packet such as length, data length, checksum, packet type, etc. is stored in the structure as shown below.

```
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_tusers;               /* 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 */
```

3.2   sock

The *sock* data structure keeps data about a specific TCP connection (e.g., TCP state) or virtual UDP connection. Whenever a socket is created in user space, a *sock* structure is allocated.

The first fields contain the source and destination addresses and ports of the socket pair.

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

Among many other fields, the *sock* structure contains protocol-specific information. These fields contain state information about 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;

    };
```

3.3 TCP options

One of the main components of the *sock* structure is the TCP option 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 large set of variables. These variables are stored in the fields of the *tcp_opt* structure; only the most relevant fields are 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 */
    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                       */

};
```

4 Sub-IP Layer

This section describes the reception and handling of packets by the hardware and the Network Interface Card (NIC) driver. This corresponds to layers 1 and 2 in the classical 7-layer network model. The driver and the IP layer are tightly bound with the driver using methods from both the kernel and the IP layer.

4.1 Memory management

The allocation of a packet descriptor is done in `net/core/skbuff.c` by the *alloc_skb()* function. This function is used each time a new buffer is needed, especially in the driver code. It gets the header from the pool of packets of the current processor (*skb_head_from_pool*). It allocates memory for the data payload (data area) with *kmalloc()*and sets up the data pointer and the state of the descriptor. It collects some memory statistics to debug all memory leaks.

Some packets are allocated through *skb_clone()* when only the meta-data (in the *sk_buff* struct) need to be duplicated for the same packet data. This is the case for packet between TCP and IP on the transmitter side. The difference between the two types of allocation lies in the deallocation: *skb*'s allocated by *alloc_skb()* are de-allocated at ACK arrival time, while those allocated by *skb_clone()* are de-allocated after receiving transmit completion events from the NIC.

The deallocation of *sk_buff* is done by the internal function *__kfree_skb()* (called by *kfree_skb()* in `net/core/skbuff.c`). It releases the *dst* fields with *dst_release()*. This field contains, among other things, the destination device of the packet. The function calls *skb->destructor()* if present to do some specific operations before cleaning. De-allocating an *skb* involves finally cleaning it (for future reuse) with *skb_headerinit()*, freeing its data part if it is not a clone, and inserting it into a free *skb* pool for future reuse with *kfree_skbmem()*.

4.2 Packet Reception

The main files that deal with transmitting and receiving the frames below the IP network layer are:

- `include/linux/netdevice.h`
- `net/core/skbuff.c`
- `net/core/dev.c`
- `net/dev/core.c`
- `arch/i386/irq.c`
- `drivers/net/net_init.c`
- `net/sched/sch_generic.c`

As well as containing data for the higher layers, the packets are associated with descriptors that provide information on the physical location of the data, the length of the data, and extra control and status information. Usually the NIC driver sets up the packet descriptors and organizes them as ring buffers when the driver is loaded. Separate ring buffers are used by the NIC's Direct Memory Access (DMA) engine to transfer packets to and from main memory. The ring buffers (both the *tx_ring* for transmission and the *rx_ring* for reception) are just arrays of *skbuff*'s, managed by the interrupt handler (allocation is performed on reception and deallocation on transmission of the packets).

IP layer *ip_rcv()*

rx_softirq (*net_rx_action()*)

backlog queue (per CPU)

Pointer to packet descriptor

Drop if in throttle state

Interrupt Handler

netif_rx():
enqueue packet in backlog
schedule softirq

Ring buffer

Kernel memory

Data packet

Free descriptor   **Updated descriptor**   Full packet
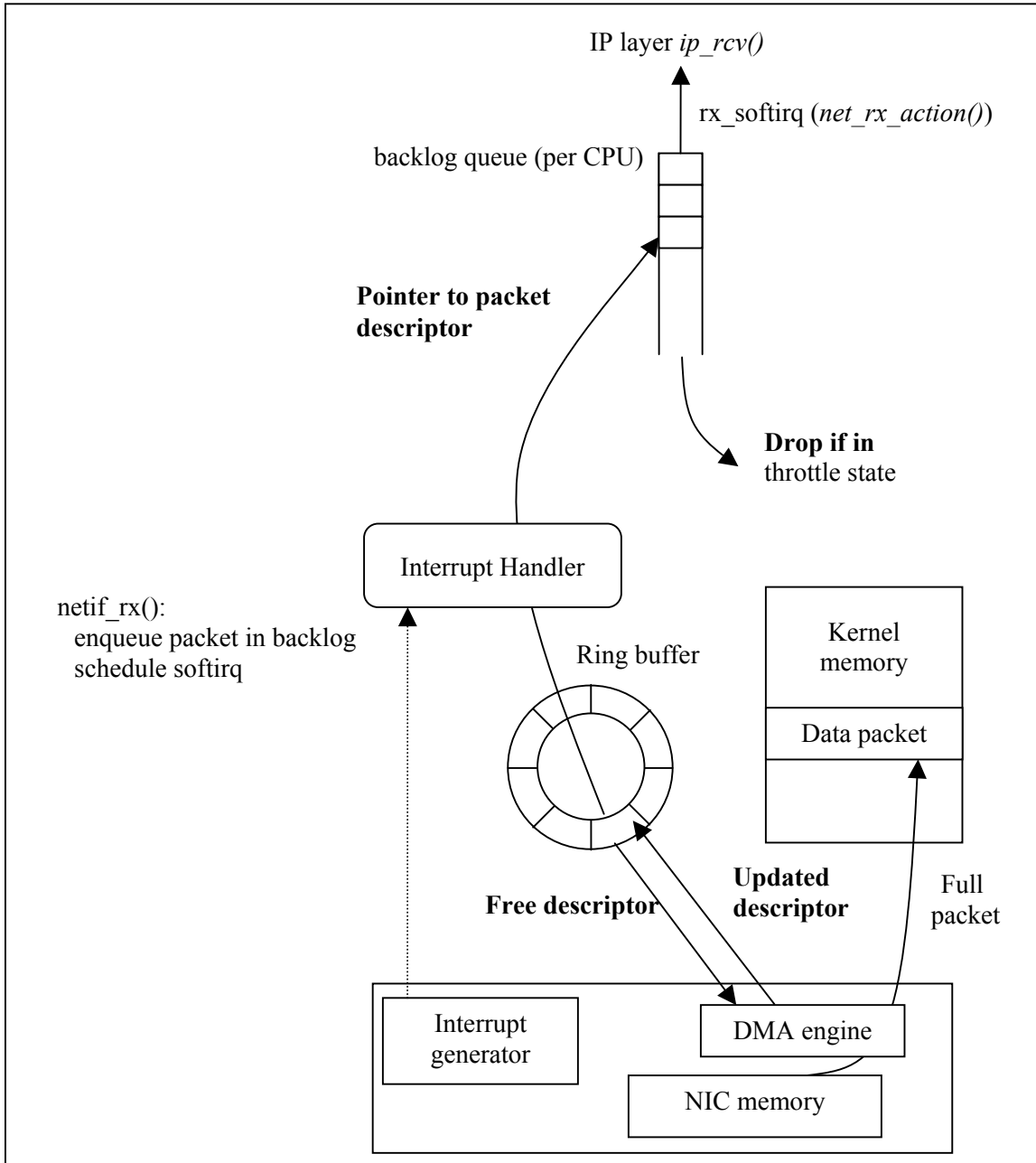
Interrupt generator   DMA engine

NIC memory

Figure 4: Packet reception with the old API until Linux kernel 2.4.19

Figure 4 and Figure 5 show the data flows that occur when a packet is received. The following steps are followed by a host.
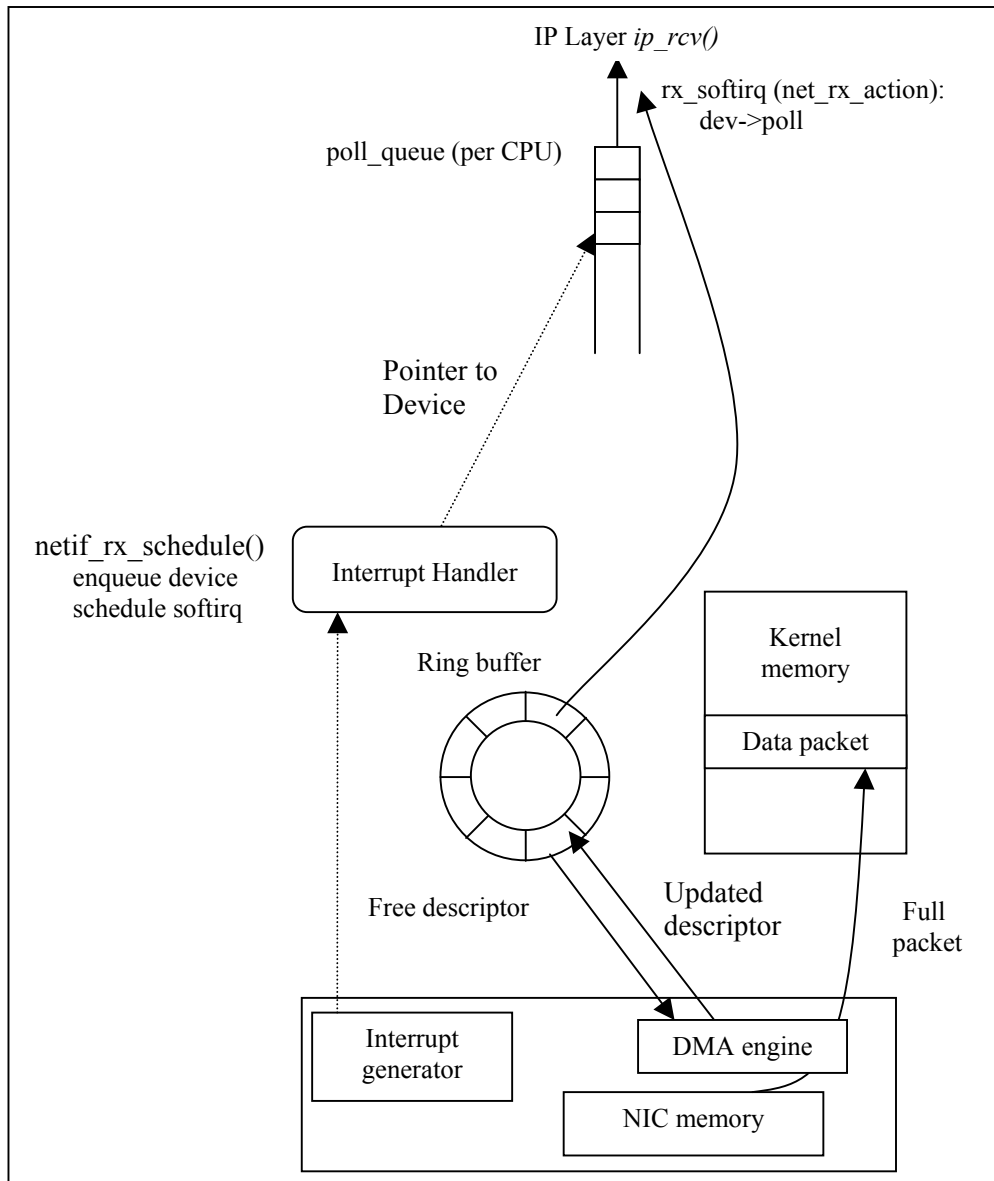
IP Layer *ip_rcv()*

rx_softirq (net_rx_action): dev->poll

poll_queue (per CPU)

Pointer to Device

netif_rx_schedule()
enqueue device
schedule softirq

Interrupt Handler

Ring buffer

Kernel memory

Data packet

Free descriptor

Updated descriptor

Full packet

Interrupt generator

DMA engine

NIC memory

Figure 5: Packet reception with Linux kernel 2.4.20: the new API (NAPI)

4.2.1 Step 1

When a packet is received by the NIC, it is put into kernel memory by the card DMA engine. The engine uses a list of packet descriptors that point to available areas of kernel memory where the packet may be placed. Each available data area must be large enough to hold the maximum size of packet that a particular interface can receive. This maximum size is specified by *maxMTU* (MTU stands for Maximum Transfer Unit). These descriptors are held in the *rx_ring* ring buffer in the kernel memory. The size of this ring buffer is driver and hardware dependent.

It is the interrupt handler (driver dependent) which first creates the packet descriptor (*struct sk_buff*). Then a pointer (*struct sk_buff**) is placed in the *rx_ring* and manipulated through the network stack. During subsequent processing in the network stack, the packet data remains at the

same kernel memory location. No extra copies are involved. Older cards use the Program I/O (PIO) scheme: it is the host CPU which transfers the data from the card into the host memory.

4.2.2 Step 2

The card interrupts the CPU, which then jumps to the driver Interrupt Service Routine (ISR) code. Here some differences arise between the old network subsystem (in kernels up to 2.4.19) and NAPI (from 2.4.20).

4.2.2.1 For old API kernels, up to 2.4.19

Figure 4 shows the routines called for network stacks prior to 2.4.20. The interrupt handler calls the *netif_rx()* kernel function (in `net/dev/core.c`, line 1215). The *netif_rx()* function enqueues the received packet in the interrupted CPU's backlog queue and schedules a *softirq[1]*, which is responsible for further processing of the packet (e.g. the TCP/IP processing). Only a pointer to the packet descriptor is actually enqueued in the backlog queue. Depending on settings in the NIC, the CPU may receive an interrupt for each packet or groups of packets (see Section 4.5).

By default, the backlog queue has a length of 300 packets, as defined in `/proc/sys/net/core/netdev_max_backlog`. If the backlog queue becomes full, it enters the throttle state and waits for being totally empty before re-entering a normal state and allowing further packets to be enqueued (*netif_rx()* in `net/dev/core.c`). If the backlog is in the throttle state, *netif_rx* drops the packet.

Backlog statistics are available from `/proc/net/softnet_stat`. The format of the output is defined in `net/core/dev.c`, lines 1804 onward. There is one line per CPU. The columns have the following meanings:

1. packet count;

2. drop count;

3. the *time squeeze* counter, i.e. the number of times the *softirq* took too much time to handle the packets from the device. When the budget of the *softirq* (i.e., the maximum number of packets it can dequeue in a row, which depends on the device, max = 300) reaches zero or when its execution time lasts more than one *jiffie* (10 ms, the smallest time unit in the Linux scheduler), the *softirq* stops dequeuing packets, increments the *time squeeze* counter of the CPU and reschedules itself for later execution;

4. number of times the backlog entered the throttle state;

5. number of hits in fast routes;

6. number of successes in fast routes;

7. number of defers in fast routes;

8. number of defers out in fast routes;

9. The right-most column indicates either latency reduction in fast routes or CPU collision, depending on a `#ifdef` flag.

An example of backlog statistics is shown below:

---

[1] A *softirq* (software interrupt request) is a kind of kernel thread [12] [13].

```
$ cat /proc/net/softnet_stat
94d449be 00009e0e 000003cd 0000000e 00000000 00000000 00000000 00000000 0000099f
000001da 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000005b
000002ca 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000b5a
000001fe 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000010
```

4.2.2.2 For NAPI drivers, from kernel 2.4.20 onward

NAPI drivers act differently. As shown in Figure 5, the interrupt handler calls *netif_rx_schedule()* (`include/linux/netdevice.h`, line 738). Instead of putting a pointer to the packet descriptor in the backlog queue, it puts a reference to the device in a queue attached to the interrupted CPU known as the *poll_list* (see *softnet_data->poll_list* in `include/linux/netdevice.h`, line 496). A *softirq* is then scheduled, just as in the previous case, but *receive* interruptions are disabled during the execution of the *softirq*.

To ensure backward compatibility with old drivers, the backlog queue is still implemented in NAPI-enabled kernels, but it is considered as a device to handle the incoming packets from the NICs whose drivers are not NAPI aware. It can be enqueued just as any other NIC device. The *netif_rx()* function is used only in the case of non-NAPI drivers, and has been rewritten to enqueue the backlog queue into the *poll_list* of the CPU after having enqueued the packet into the backlog queue.

4.2.3 Step 3

When the *softirq* is scheduled, it executes *net_rx_action()* (`net/core/dev.c`, line 1558). *Softirqs* are scheduled in *do_softirq()* (`arch/i386/irq.c`) when *do_irq* is called to do any pending interrupts. They can also be scheduled through the *ksoftirq* process when *do_softirq()* is interrupted by an interrupt, or when a *softirq* is scheduled outside an interrupt or a bottom-half of a driver. The *do_softirq()* function processes *softirqs* in the following order: `HI_SOFTIRQ`, `NET_TX_SOFTIRQ`, `NET_RX_SOFTIRQ` and `TASKLET_SOFTIRQ`. More details about scheduling in the Linux kernel can be found in [10]. Because step 2 differs between the older network subsystem and NAPI, step 3 does too.

For kernel versions prior to 2.4.20, *net_rx_action()* polls all the packets in the backlog queue and calls the *ip_rcv()* procedure for each of the data packets (`net/ipv4/ip_input.c`, line 379). For other types of packets (ARP, BOOTP, etc.), the corresponding *ip_xx()* routine is called.

For NAPI, the CPU polls the devices present in its *poll_list* (including the backlog for legacy drivers) to get all the received packets from their *rx_ring*. The *poll* method of any device (*poll()*, implemented in the NIC driver) or of the backlog (*process_backlog()* in `net/core/dev.c`, line 1496) calls *netif_receive_skb()* (`net/core/dev.c`, line 1415) for each received packet, which then calls *ip_rcv()*.

The NAPI network subsystem is a lot more efficient than the old system, especially in a high performance context (in our case, gigabit Ethernet). The advantages are:

- limitation of interruption rate (this may be seen as an adaptive interrupt coalescing mechanism);
- it is not prone to receive livelock [17];
- better data and instruction locality.

Because a device is always handled by a CPU, there is no packet reordering or cache default. One problem is that there is no parallelism in a Symmetric Multi-Processing (SMP) machine for traffic coming in from a single interface.

In the old API case, if the input rate is too high, the backlog queue becomes full and packets are dropped in the kernel, exactly between the rx_ring and the backlog in the enqueue procedure. In the NAPI case, exceeding packets are dropped earlier, before being put into the rx_ring. In this last case, an Ethernet pause packet halting the packet input if this feature is enabled.

4.3   Packet Transmission

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

Drop if full   IP Layer

enqueue (*dev_queue_xmit()*)

qdisc (per device)

qdisc_restart():
while not (empty or stopped)
hard_start_xmit()

tx_ring

Kernel
memory

Data packet
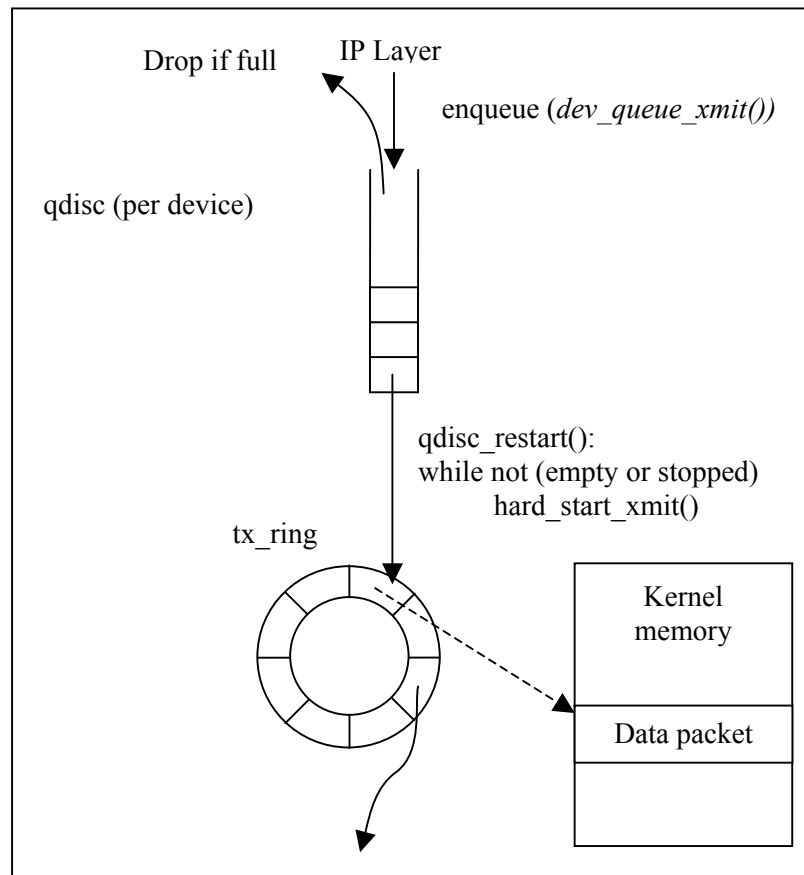
Figure 6: Transmission of a packet

The kernel provides multiple queuing disciplines (RED, CBQ, etc.) between the kernel and the driver. It is intended to provide QoS support. The default queuing discipline, or *qdisc*, consists of three FIFO queues with strict priorities and a default length of 100 packets for each queue (*ether_setup(): dev->tx_queue_len* ; `drivers/net/net_init.c`, line 405).

Figure 6 shows the different data flows that may occur when a packet is to be transmitted. The following steps are followed during transmission.

4.3.1 Step 1

For each packet to be transmitted from the IP layer, the *dev_queue_xmit()* procedure (`net/core/dev.c`, line 991) is called. It queues a packet in the *qdisc* associated to the output interface (as determined by the routing). Then, if the device is not stopped (e.g., due to link failure or the *tx_ring* being full), all packets present in the *qdisc* are handled by *qdisc_restart()* (`net/sched/sch_generic.c`, line 77).

4.3.2 Step 2

The *hard_start_xmit()* virtual method is then called. This method is implemented in the driver code. The packet descriptor, which contains the location of the packet data in kernel memory, is placed in the *tx_ring* and the driver tells the NIC that there are some packets to send.

4.3.3 Step 3

Once the card has sent a packet or a group of packets, it communicates to the CPU that the packets have been sent out by asserting an interrupt. The CPU uses this information (*net_tx_action()* in `net/core/dev.c`, line 1326) to put the packets into a *completion_queue* and to schedule a *softirq* for later deallocating (i) the meta-data contained in the *skbuff* struct and (ii) the packet data if we are sure that we will not need this data anymore (see Section 4.1). This communication between the card and the CPU is card and driver dependent.

4.4 Commands for monitoring and controlling the input and output network queues

The *ifconfig* command can be used to override the length of the output packet queue using the *txqueuelen* option. It is not possible to get statistics for the default output queue. The trick is to replace it with the same FIFO queue using the *tc* command:

- to replace the default *qdisc*: `tc qdisc add dev eth0 root pfifo limit 100`
- to get stats from this *qdisc*: `tc -s -d qdisc show dev eth0`
- to recover to default state: `tc qdisc del dev eth0 root`

4.5 Interrupt Coalescence

Depending on the configuration set by the driver, a modern NIC can either interrupt the host for each packet sent or received, or it can continue to transfer packets between the network and memory, using the descriptor mechanisms described above, but only informs the CPU of progress at intervals. This is known as *interrupt coalescence* and the details and options are hardware dependent. The NIC may generate interrupts after a fixed number of packets have been processed or after a fixed time from the first packet transferred after the last interrupt. In some cases, the NIC dynamically changes the interrupt coalescence times depending on the packet receive rate. Separate parameters are usually available for the transmit and receive functions of the NIC.

Interrupt coalescence, as the use of NAPI, reduces the amount of time the CPU spends context-switching to service interrupts. It is worth noting that the size of the transmit and receive ring buffers (and the kernel memory area for the packets) must be large enough to provide for the extra packets that will be in the system.

5 Network layer

The network layer provides end-to-end connectivity in the Internet across heterogeneous networks. It provides the common protocol (IP – Internet Protocol) used by almost all Internet traffic. Since Linux hosts can act as routers (and they often do as they provide an inexpensive way of building networks), an important part of the code deals with packet forwarding.

The main files that deal with the IP network layer are located in `net/ipv4`:

- `ip_input.c` – processing of the packets arriving at the host
- `ip_output.c` – processing of the packets leaving the host
- `ip_forward.c` – processing of the packets being routed by the host

Other files include:

- `ip_fragment.c` – IP packet fragmentation
- `ip_options.c` – IP options
- `ipmr.c` – IP multicast
- `ipip.c` – IP over IP

5.1 IP

5.1.1 IP Unicast

Figure 7 describes the path that an IP packet traverses inside the network layer. Packet reception from the network is shown on the left hand side and packets to be transmitted flow down the right hand side of the diagram. When the packet reaches the host from the network, it goes through the functions described in Section 4; when it reaches *net_rx_action()*, it is passed to *ip_rcv()*. After passing the first *netfilter* hook (see Section 2), the packet reaches *ip_rcv_finish()*, which verifies whether the packet is for local delivery. If it is addressed to this host, the packet is given to *ip_local_delivery()*, which in turn will give it to the appropriate transport layer function.

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

In the output part, the last changes to the packet are made in *ip_finish_output()* and the function *dev_queue_transmit()* is called; the latter enqueues the packet in the output queue. It also tries to run the network scheduler mechanism by calling *qdisc_run()*. This pointer will point to different functions, depending on the scheduler installed. A FIFO scheduler is installed by default, but this can be changed with the *tc* utility, as we have seen already.

The scheduling functions (*qdisc_restart()* and *dev_queue_xmit_init()*) are independent of the rest of the IP code.

When the output queue is full, *q->enqueue* returns an error which is propagated upward on the IP stack. This error is further propagated to the transport layer (TCP or UDP) as will be seen in Sections 6 and 7.
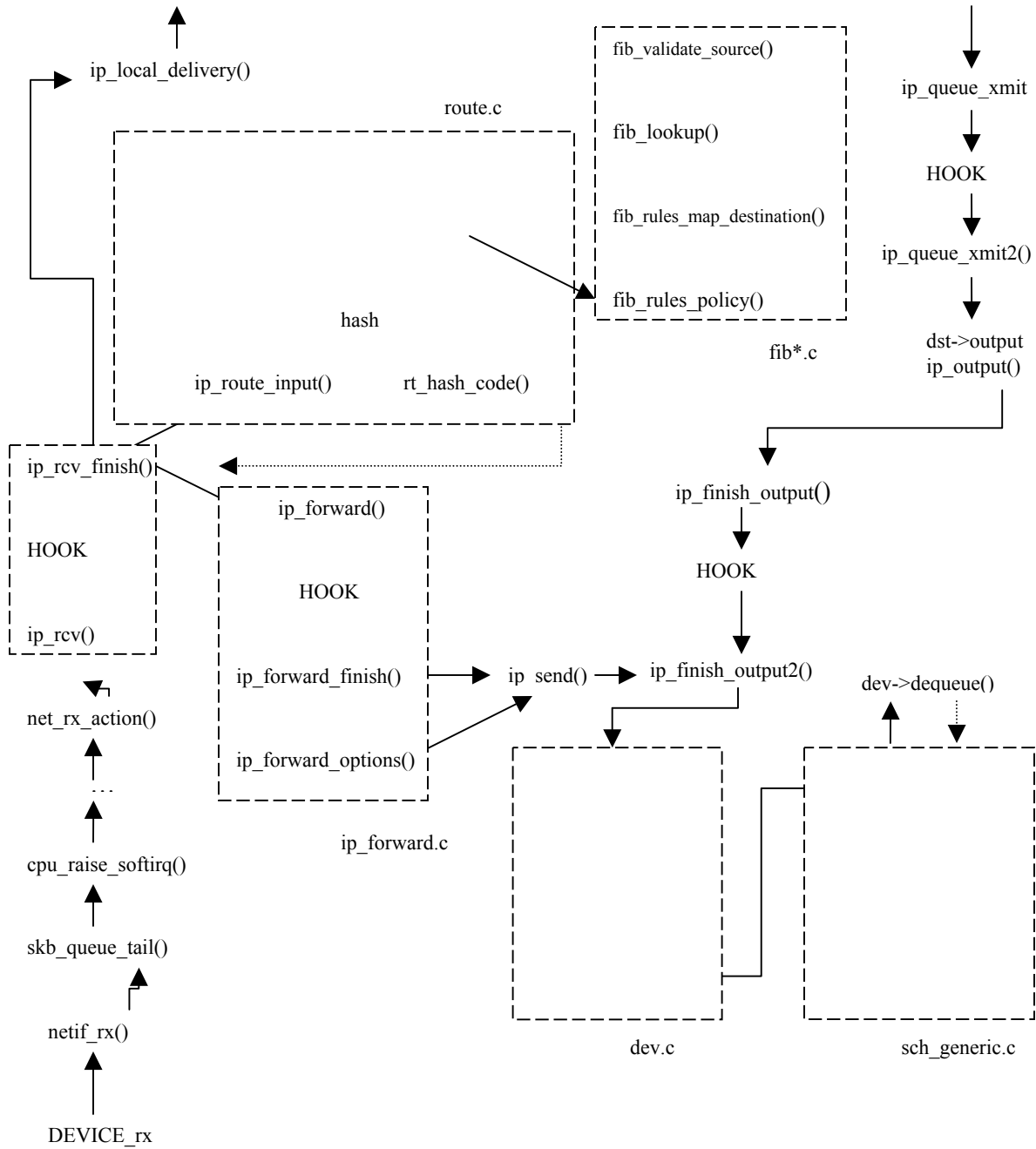
Figure 7: Network layer data path

5.1.2 IP Routing

If an incoming packet has a destination IP address other than that of the host, the latter acts as a router (a frequent scenario in small networks). If the host is configured to execute forwarding (this can be seen and set via /proc/sys/net/ipv4/ip_forward), it then has to be processed

by a set of complex but very efficient functions. If the *ip_forward* variable is set to zero, it is not forwarded.

The route is calculated by calling *ip_route_input()*, which (if a fast hash does not exist) calls *ip_route_input_slow()*. The *ip_route_input_slow()* function calls the FIB (Forward Information Base) set of functions in the `fib*.c` files. The FIB structure is quite complex [3].

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

After the route is calculated, *ip_rcv_finished()* 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 send it to the output components.

5.1.3 IP Multicast

The previous section dealt with unicast packets. With multicast packets, the system gets significantly more complicated. The user level (through a daemon like *gated*) uses the *setsockopt()* call on the UDP socket or *netlink* to instruct the kernel that it wants to join the group. The *set_socket_option()* function calls *ip_set_socket_option()*, which calls *ip_mc_join_group()* (or *ip_mc_leave_group()* when it wants to leave the group).

This function calls *ip_mc_inc_group()*. This makes a trigger expire and *igmp_timer_expire()* be called. Then *igmp_timer_expire()* calls *igmp_send_report()*.

When a host receives an IGMP (Internet Group Management Protocol) packet (that is, when we are acting as a multicast router), *net_rx_action()* delivers it to *igmp_rcv()*, which builds the appropriate multicast routing table information.

A more complex operation occurs when a multicast packet arrives at the host (router) or when the host wants to send a multicast packet. The packet is handled by *ip_route_output_slow()* (via *ip_route_input()* if the packet is coming in or via *ip_queue_xmit()* if the packet is going out), which in the multicast case calls *ip_mr_input()*.

Next, *ip_mr_input()* (`net/ipv4/ipmr.c`, line 1301) calls *ip_mr_forward()*, which calls *ipmr_queue_xmit()* for all the interfaces it needs to replicate the packet. This calls *ipmr_forward_finish()*, which calls *ip_finish_output()*. The rest can be seen on Figure 7.

5.2   ARP

Because ARP (Address Resolution Protocol) converts layer-3 addresses to layer-2 addresses, it is often said to be at layer 2.5. ARP is defined in RFC 826 and is the protocol that allows IP to run over a variety of lower layer technologies. Although we are mostly interested in Ethernet in this document, it is worth noting that ARP can resolve IP addresses for a wide variety of technologies, including ATM, Frame Relay, X.25, etc.

When an ARP packet is received, it is given by *nt_rx_action()* to *arp_rcv()* which, after some sanity checks (e.g., checking if the packet is for this host), passes it on to *arp_process()*. Then, *arp_process()* checks which type of ARP packet it is and, if appropriate (e.g., when it is an ARP request), sends a reply using *arp_send()*.

The decision of sending an ARP request deals with a much more complex set of functions depicted in Figure 8. When the host wants to send a packet to a host in its LAN, it needs to

convert the IP address into the MAC address and store the latter in the *skb* structure. When the host is not in the LAN, the packet is sent to a router in the LAN. The function *ip_queue_xmit()* (which can be seen in Figure 7) calls *ip_route_output()*, which calls *rt_intern_hash()*. This calls *arp_bind_neighbour()*, which calls *neigh_lookup_error()*.

The function *neigh_lookup_error()* tries to see if there is already any neighbor data for this IP address with *neigh_lookup()*. If there is not, it triggers the creation of a new one with *neigh_create()*. The latter triggers the creation of the ARP request by calling *arp_constructor()*. Then the function *arp_constructor()* starts allocating space for the ARP request and calls the function *neigh->ops->output()*, which points to *neigh_resolve_output()*. When *neigh_resolve_output()* is called, it invokes *neigh_event_send()*. This calls *neigh->ops->solicit()*, which points to *arp_solicit()*. The latter calls *arp_send()*, which sends the ARP message. The *skb* to be resolved is stored in a list. When the reply arrives (in *arp_recv()*), it resolves the *skb* and removes it from the list.

ip_queue_xmit()

ip_route_output()

neigh_lookup_error()

neigh_resolve_output()

rt_intern_hash()

neigh_create()

arp_solicit()

arp_bind_neighbour()

arp_constructor()

arp_send()
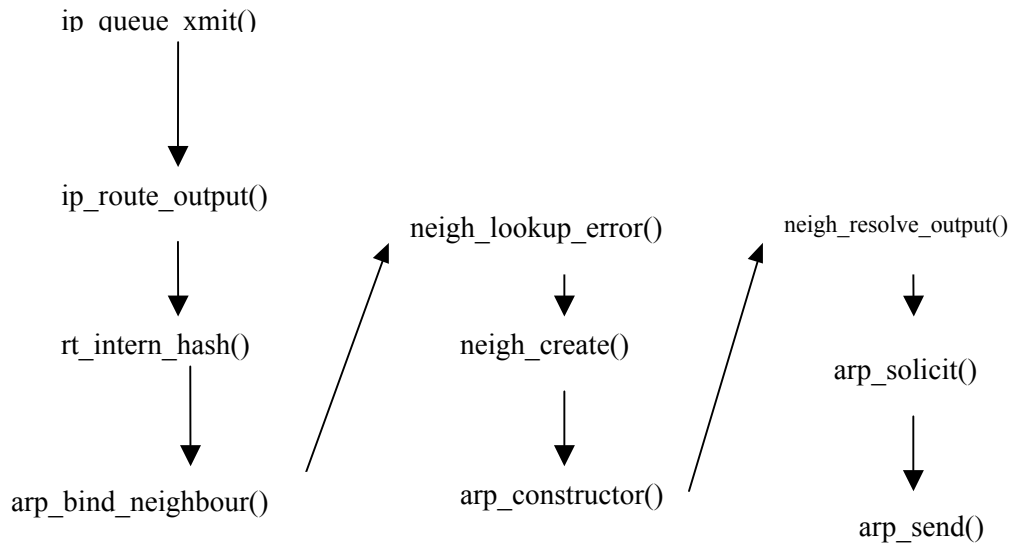
Figure 8: ARP

5.3 ICMP

The Internet Control Message Protocol (ICMP) plays an important role in the Internet. Its implementation is quite simple. Conceptually, ICMP is at the same level as IP, although ICMP datagrams use IP packets.

Figure 9 depicts the main ICMP functions. When an ICMP packet is received, *net_rx_action()* delivers it to *icmp_rcv()* where the ICMP field is checked; depending on the type, the appropriate function is called (this is done by calling *icmp_pointers[icmp->type].handler()*). In Figure 10, we can see the description of the main functions and types. Two of these functions, *icmp_echo()* and *icmp_timestamp()*, require a response to be sent to the original source. This is done by calling *icmp_reply()*.

Sometimes, a host needs to generate an ICMP packet that is not a mere reply to an ICMP request (e.g., the IP layer, the UDP layer and users—through raw sockets—can send ICMP packets). This is done by calling *icmp_send()*.

```
                              ━━►  icmp_discard()

                              ━━►  icmp_unreach()

                              ━━►  icmp_redirect()

icmp_rcv()  ━━━━━►            ━━►  icmp_timestamp()  ━━━━━►  icmp_reply()

                              ━━►  icmp_address()

                              ━━►  icmp_echo()  ━━━━━┘

                              ━━►  icmp_address_reply()
```

UDP	User	IP

icmp_send()
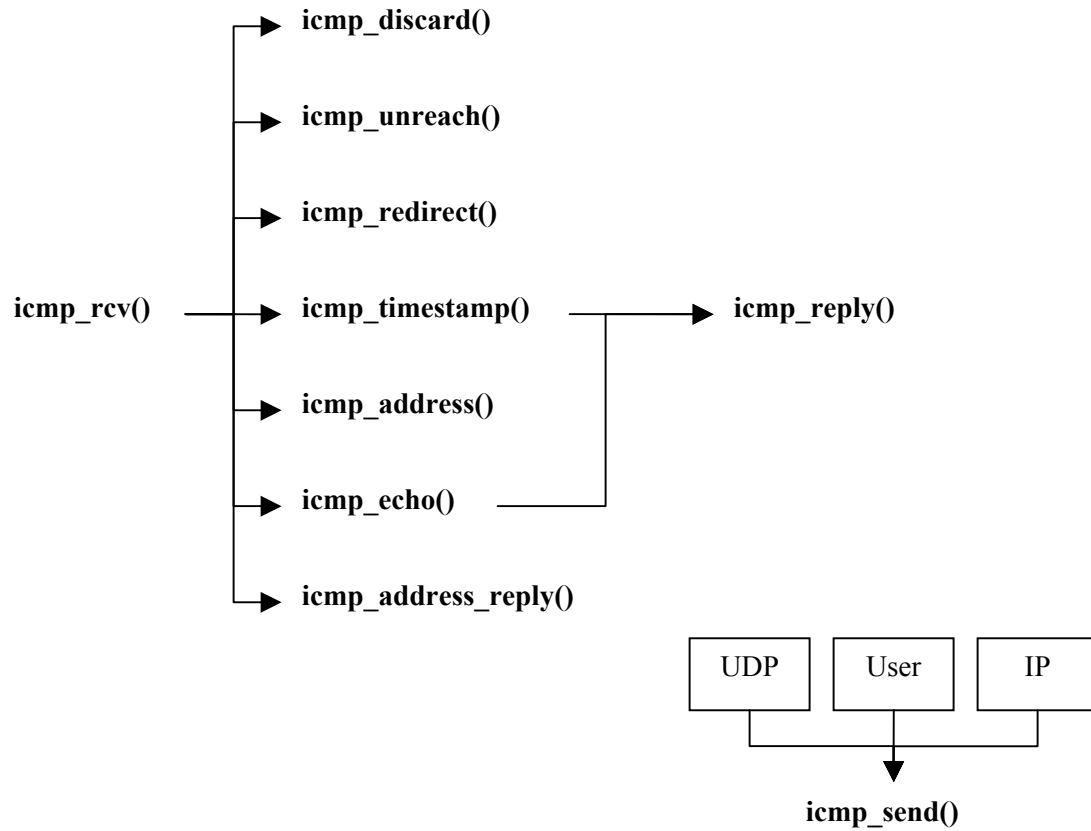
Figure 9: ICMP functions

ICMP function	Description
icmp_discard()	Discard the packet.
icmp_unreach()	Destination unreachable, ICMP time-exceed or ICMP source quench.
icmp_redirect()	ICMP redirect error. The router to which an IP packet was sent is saying that the datagram should have been sent to another router.
icmp_timestamp()	This host is being queried about the current timestamp (usually the number of seconds).
icmp_address()	Request for a network address mask . Typically used by a diskless system to obtain its subnet mask.
icmp_address_reply()	This message contains the reply to an ICMP address request.
icmp_echo()	ICMP echo command. This requires the host to send an ICMP echo reply to the original sender. This is how the *ping* command is implemented.

Figure 10: ICMP packet types

6   TCP

This section describes the implementation of the Transmission Control Protocol (TCP), which is probably the most complex part of the networking code in the Linux kernel.

TCP contributes for the vast majority of the traffic in the Internet. It fulfills two important functions: it establishes a reliable communication between a sender and a receiver by retransmitting non-acknowledged packets, and it implements congestion control by reducing the sending rate when congestion is detected.

Although both ends of a TCP connection can be sender and receiver simultaneously, we separate our code explanations for the "receiver" behavior (when the host receives data and sends acknowledgments) and the "sender" behavior (when the host sends data, receives acknowledgments, retransmits lost packets and adjusts congestion window and sending rate). The complexity of the latter is significantly higher.

The reader is assumed to be familiar with the TCP state machine, which is described in [6].

The main files of the TCP code are all located in `net/ipv4`, except header files which are in `include/net`. They are:

- `tcp_input.c` – Code dealing with incoming packets from the network.
- `tcp_output.c` – Code dealing with sending packets to the network.
- `tcp.c` – General TCP code. Links with the socket layer and provides some "higher" level functions to create and release TCP connections.
- `tcp_ipv4.c` – IPv4 TCP specific code.
- `tcp_timer.c` – Timer management.
- `tcp.h` – Definition of TCP constants.

Figure 11 and Figure 12 depict the TCP data path and are meant to be viewed side by side. Input processing is described in Figure 11 and output processing is illustrated by Figure 12.
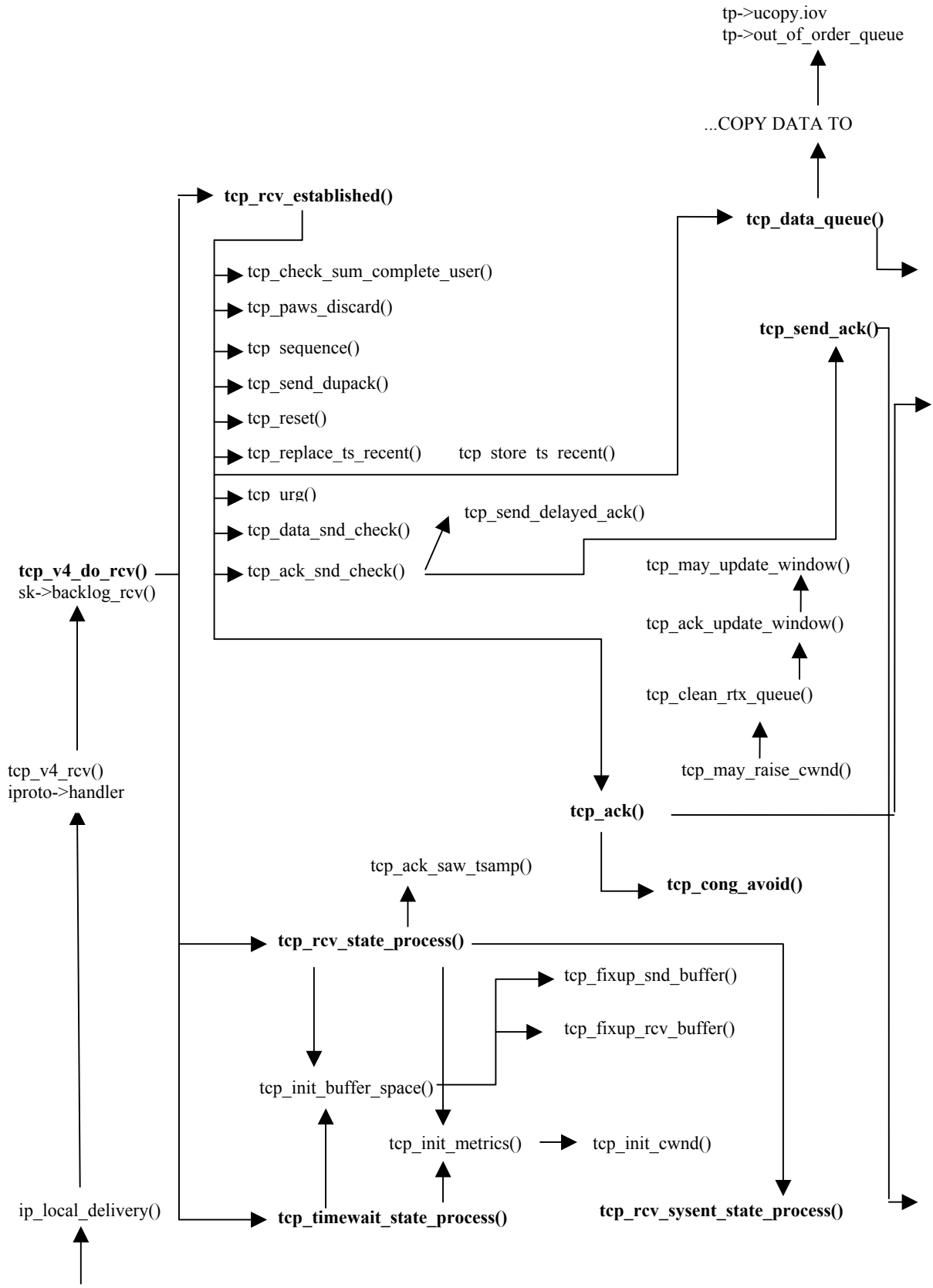
tp->ucopy.iov
tp->out_of_order_queue

...COPY DATA TO

tcp_rcv_established()

tcp_check_sum_complete_user()

tcp_paws_discard()

tcp sequence()

tcp_send_dupack()

tcp_reset()

tcp_replace_ts_recent()   tcp store ts recent()

tcp urg()

tcp_data_snd_check()

tcp_ack_snd_check()

tcp_data_queue()

tcp_send_ack()

tcp_send_delayed_ack()

tcp_may_update_window()

tcp_ack_update_window()

tcp_clean_rtx_queue()

tcp_may_raise_cwnd()

tcp_v4_do_rcv()
sk->backlog_rcv()

tcp_v4_rcv()
iproto->handler

tcp_ack()

tcp_ack_saw_tsamp()

tcp_cong_avoid()

tcp_rcv_state_process()

tcp_fixup_snd_buffer()

tcp_fixup_rcv_buffer()

tcp_init_buffer_space()

tcp_init_metrics()   tcp_init_cwnd()

ip_local_delivery()

tcp_timewait_state_process()

tcp_rcv_sysent_state_process()

Figure 11: TCP: input processing

tcp_rmem_shcedule()

tcp_grow_window()

tcp_incr_quick_ack()

tcp_measure_rcv_mss()

tcp_schedule_ack()

tcp_event_data_rcv()

tcp_moderate_cwnd()

tcp_fast_retrans_alert()

tcp_remove_reno_sacks()

tcp_add_reno_sack()

tcp_update_scoreboard()

tcp_head_timeout()

tcp_check_reno_reordering()

skb_timed_out()

tcp_undo_recovery()

tcp_try_undo_partial()

tcp_may_undo()

tcp_time_to_recover()

tcp_packet_delayed()

tcp_fackets_out()

tcp_enter_loss()

tcp_try_undo_loss()

tcp_clear_retrans()

sys_write()

sock_write()

sock_sendmsg()

sock->ops->sendmsg()
tcp_sendmsg()

sk->prot->sendmsg()
tcp_sendmsg()

tcp_push()

tcp_push_pending_frames()

tcp_write_xmit()

tcp_transmit_skb()

tp->af_specific->queue_xmit
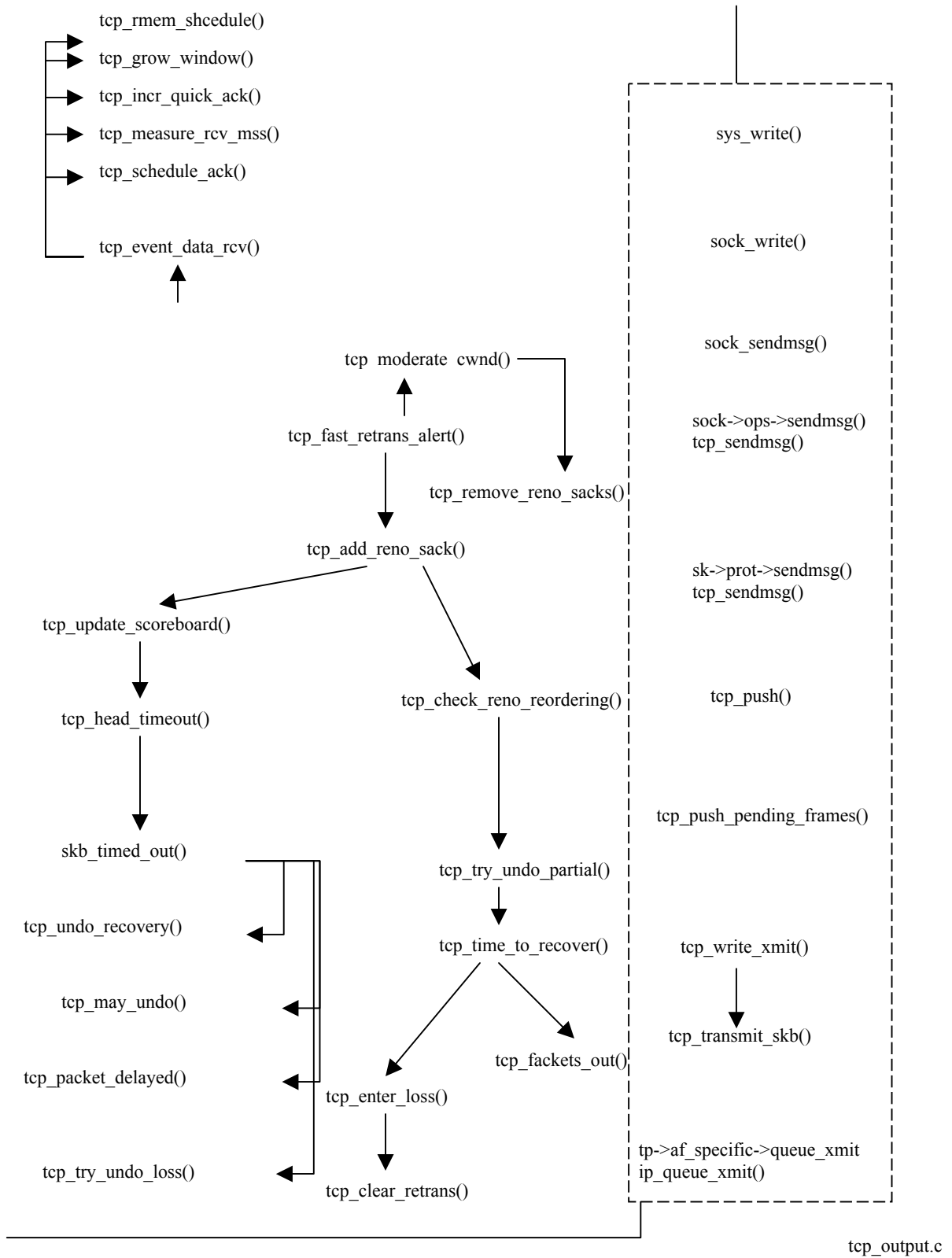ip_queue_xmit()

tcp_output.c

Figure 12: TCP: output processing

27

6.1   TCP Input

TCP input is mainly implemented in `net/ipv4/tcp_input.c`. This is the largest portion of the TCP code. It deals with the reception of a TCP packet. The sender and receiver code is tightly coupled as an entity can be both at the same time.

Incoming packets are made available to the TCP routines from the IP layer by *ip_local_delivery()* shown on the left side of Figure 11. This routine gives the packet to the function pointed by *ipproto->handler* (see structures in Section 2). For the IPv4 protocol stack, this is *tcp_v4_rcv()*, which calls *tcp_v4_do_rcv()*. The function *tcp_v4_do_rcv()* in turn calls another function depending on the TCP state of the connection (for more details, see [6]).

If the connection is established (state is TCP_ESTABLISHED), it calls *tcp_rcv_established()*. This is the main case that we will examine from now on. If the state is TIME_WAIT, it calls *tcp_timewait_process()*. All other states are processed by *tcp_rcv_state_process()*. For example, this function calls *tcp_rcv_sysent_state_process()* if the state is SYN_SENT.

For some TCP states (e.g., CALL_SETUP), *tcp_rcv_state_process()* and *tcp_timewait_process()* have to initialize the TCP structures. They call *tcp_init_buffer_space()* and *tcp_init_metrics()*. The latter initializes the congestion window by calling *tcp_init_cwnd()*.

The following subsections describe the actions of the functions shown in Figure 11 and Figure 12. The function *tcp_rcv_established()* has two modes of operation: fast path and slow path. We first describe the slow path, which is easier to understand, and present the fast path afterward. Note that in the code, the fast path is dealt with first.

6.1.1 tcp_rcv_established(): Slow Path

The slow path code follows the 7 steps defined in RFC 793, plus a few other operations:

- The checksum is calculated with *tcp_checksum_complete_user()* . If it is incorrect, the packet is discarded.
- The Protection Against Wrapped Sequence Numbers (PAWS) [14] 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()*. The latter may have to implement a SACK (*tcp_dsack_set()*) but it finishes by calling *tcp_send_ack()*.

STEP 2: It checks the RST (connection reset) bit (*th->rst*). If it is on, it calls *tcp_reset()*. An error must be passed on to the upper layers.

STEP 3: It is supposed to check security and precedence but this is not implemented.

STEP 4, part 1: It checks SYN bit. If it is on, it calls *tcp_reset()*. This synchronizes sequence numbers to initiate a connection.

STEP 4, part 2: It calculates an estimative for the RTT (RTTM) by calling *tcp_replace_ts_recent()*.

STEP 5: It checks the ACK bit. If this bit is set, the packet brings an acknowledgment and *tcp_ack()* is called (more details to come in Section 6.1.3).

STEP 6: It checks the URG (urgent) bit. If this bit is set, it calls *tcp_urg()*. This makes the receiver tell the process listening to the socket that the data is urgent.

STEP 7, part 1: It processes data on the packet. This is done by calling *tcp_data_queue()* (more details in Section 6.1.2 below).

STEP 7, part 2: It checks if there is data to send by calling *tcp_data_snd_check()*. This function calls *tcp_write_xmit()* on the TCP output sector.

STEP 7, part 3: It checks 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()*.

6.1.2 tcp_data_queue() & tcp_event_data_recv()

The *tcp_data_queue()* function is responsible for giving the data to the user. If the packet arrived in order (all previous packets having already arrived), it copies the data to *tp->ucopy.iov* (*skb_copy_datagram_iovec(skb, 0, tp->ucopy.iov, chunk)*); see structure *tcp_opt* in Section 3.

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, Section 4.2 of RFC 2581 [2] says that we should 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 with by calling *tcp_event_data_recv()*. This function first schedules an ACK with *tcp_schedule_ack()*, and then estimates the MSS (Maximum Segment Size) with *tcp_measure_rcv_mss()*.

In certain conditions (e.g., if we are in slow start), the receiver TCP should be in QuickACK mode where ACKs are sent immediately. If this is the situation, *tcp_event_data_recv()* switches this on with *tcp_incr_quickack()*. It may also have to increase the advertised window with *tcp_grow_window()*.

Finally *tcp_data_queue()* checks if the FIN bit is set; if it is, *tcp_fin()* is called.

6.1.3 tcp_ack()

Every time an ACK is received, *tcp_ack()* is called. The first thing it does is to check if the ACK is valid by making sure it is within the right hand side of the sliding window (*tp->snd_nxt*) or older than previous ACKs. If this is the case, then we can probably ignore it with *goto uninteresting_ack* and *goto old_ack* respectively and return 0.

If everything is normal, it updates the sender's TCP sliding window with *tcp_ack_update_window()* and/or *tcp_update_wl()*. An ACK may be considered "normal" if it acknowledges the next section of contiguous data starting from the pointer to the last fully acknowledged block of data.

If the ACK is dubious, it enters fast retransmit with *tcp_fastretrans_alert()* (see Section 6.1.4 below). If the ACK is normal and the number of packets in flight is not smaller than the congestion window, it increases the congestion window by entering slow start/congestion avoidance with *tcp_cong_avoid()*. This function implements both the exponential increase in slow start and the linear increase in congestion avoidance as defined in RFC 793. When we are in congestion avoidance, *tcp_cong_avoid()* utilizes the variable *snd_cwnd_cnt* to determine when to linearly increase the congestion window.

Note that *tcp_ack()* should not be confused with *tcp_send_ack()*, which is called by the "receiver" to send ACKs using *tcp_write_xmit()*.

6.1.4 tcp_fastretransmit_alert()

Under certain conditions, *tcp_fast_retransmit_alert()* is called by *tcp_ack()* (it is only called by this function). To understand these conditions, we have to go through the Linux {NewReno, SACK, FACK, ECN} finite state machine. This section is copied almost *verbatim* from a comment in `tcp_input.c`. Note that this finite state machine (also known as the *ACK state machine*) has nothing to do with the TCP finite state machine. The TCP state is usually TCP_ESTABLISHED.

The Linux finite state machine can be in any of the following states:

- *Open*: Normal state, no dubious events, fast path.
- *Disorder*: In all respects it is "*Open*", but it requires a bit more attention. It is entered when we see some SACKs or DupACKs. It is separate from "*Open*" primarily to move some processing from fast path to slow path.
- *CWR*: The congestion window should be reduced due to some congestion notification event, which can be ECN, ICMP source quench, three duplicate ACKs, or local device congestion.
- *Recovery*: The congestion window was reduced, so now we should be fast-retransmitting.
- *Loss*: The congestion window was reduced due to an RTO timeout or SACK reneging.

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

The function *tcp_fastretrans_alert()* is entered if the state is not "*Open*", when an ACK is received or "strange" ACKs are received (SACK, DUPACK, ECN). This function performs the following tasks:

- It checks flags, ECN and SACK and processes loss information.
- It processes the state machine, possibly changing the state.
- It calls *tcp_may_undo()* routines in case the congestion window reduction was too drastic (more on this in Section 6.7.1).
- Updates the scoreboard. The scoreboard keeps track of which packets were acknowledged or not.
- It calls *tcp_cong_down()* in case we are in CWR state, and reduces the congestion window by one every other ACK (this is known as *rate halving*). The function *tcp_cong_down()* is smart because the congestion window reduction is applied over the entire RTT by using *snd_cwnd_cnt()* to count which ACK this is.
- It calls *tcp_xmit_retransmit_queue()* to decide whether anything should be sent.

6.1.5 Fast path

The fast path is entered under certain conditions in *tcp_rcv_established()*. It uses the *header prediction* technique defined in RFC 1323 [14]. This happens when the incoming packet has the expected sequence number. Although the fast path is faster than the slow path, all of the following operations are done in order: PAWS is checked, *tcp_ack()* is called if the packet was an ACK, *tcp_data_snd_check()* is called to see if more data can be sent, data is copied to the user

with *tcp_copy_to_iovec()*, the timestamp is stored with *tcp_store_ts_recent()*, *tcp_event_data_recv()* is called, and an ACK is sent in case we are the receiver.

6.2 SACKs

Linux kernel 2.4.20 fully implements SACKs (Selective ACKs) as defined in RFC 2018 [8]. The connection SACK capabilities are stored in the *tp->sack_ok* field (FACKs are enabled if the 2nd bit is set and DSACKs (delayed SACKs) are enabled if the 3rd bit is set). When a TCP connection is established, the sender and receiver negotiate different options, including SACK.

The SACK code occupies a surprisingly large part of the TCP implementation. More than a dozen functions and significant parts of other functions are dedicated to implementing SACK. It is still fairly inefficient code, because the lookup of non-received blocks in the list is an expensive process due to the linked-list structure of the *sk_buff*'s.

When a receiver gets a packet, it checks in *tcp_data_queue()* if the *skb* overlaps with the previous one. If it does not, it calls *tcp_sack_new_ofo_skb()* to build a SACK response.

On the sender side (or receiver of SACKs), the most important function in the SACK processing is *tcp_sacktag_write_queue()*; it is called by *tcp_ack()*.

6.3 QuickACKs

At certain times, the receiver enters QuickACK mode, that is, delayed ACKS are disabled. One example is in slow start, when delaying ACKs would delay the slow start considerably.

The function *tcp_enter_quick_ack_mode()* is called by *tc_rcv_sysent_state_process()* because, at the beginning of the connection, the TCP state should be SYSENT.

6.4 Timeouts

Timeouts are vital for the correct behavior of the TCP functions. They are used, for instance, to infer packet loss in the network. The events related to registering and triggering the retransmit timer are depicted in Figure 13 and Figure 14.
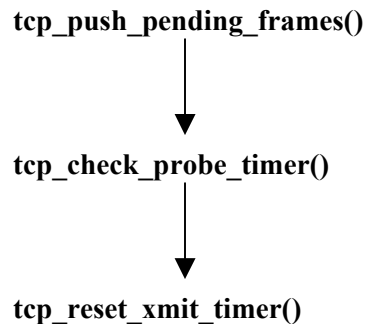
tcp_push_pending_frames()

↓

tcp_check_probe_timer()

↓

tcp_reset_xmit_timer()

Figure 13: Scheduling a timeout

The setting of the retransmit timer happens when a packet is sent. The function *tcp_push_pending_frames()* calls *tcp_check_probe_timer()*, which may call *tcp_reset_xmit_timer()*. This schedules a software interrupt, which is dealt with by non-networking parts of the kernel.

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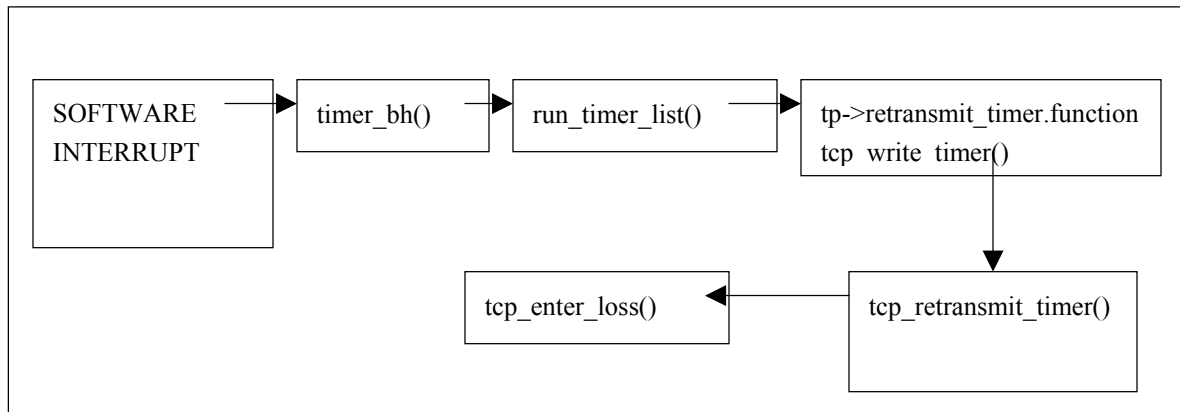

Figure 14: Timeout arrival

6.5 ECN

Linux kernel 2.4.20 fully implements ECN (Explicit Congestion Notification) to allow ECN-capable routers to report congestion before dropping packets. Almost all the code is in the *tcp_ecn.h* in the include/net directory. It contains the code to receive and send the different ECN packet types.

In *tcp_ack(),* when the ECN bit is on, *TCP_ECN_rcv_ecn_echo()* is called to deal with the ECN message. This calls the appropriate ECN message handling routine.

When an ECN congestion notification arrives, the Linux host enters the CWR state. This makes the host reduce the congestion window by one on every other ACK received. This can be seen in *tcp_fastrestrans_alert()* when it calls *tcp_cwnd_down()*.

ECN messages can also be sent by the kernel when the function *TCP_ECN_send()* is called in *tcp_transmit_skb()*.

6.6 TCP output

This part of the code (mainly net/ipv4/tcp_output.c) is illustrated in Figure 12. It deals with packets going out of the host and includes both data packets from the "sender" and ACKs from the "receiver". The function *tcp_transmit_skb()*, a crucial operation in the TCP output, executes the following tasks:

- Check *sysctl()* flags for timestamps, window scaling and SACK.
- Build TCP header and checksum.
- Set SYN packets.
- Set ECN flags.
- Clear ACK event in the socket.
- Increment TCP statistics through TCP_INC_STATS (*TcpOutSegs*).
- Call *ip_queue_xmit()*.

If there is no error, the function returns; otherwise, it calls *tcp_enter_cwr()*. This error may happen when the output queue is full. As we saw in Section 4.3.2, *q->enqueue* returns an error when this queue is full. The error is then propagated until here and the congestion control mechanisms react accordingly.

6.7   Changing the congestion window

The TCP algorithm adjusts its sending rate by reducing or increasing the size of the sending window. The basic TCP operation is straightforward. When it receives an ACK, it increases the congestion window by calling *tcp_cong_avoid()* either linearly or exponentially, depending on where we are (congestion avoidance or slow start). When it detects that a packet is lost in the network, it reduces the window accordingly.

TCP detects a packet loss when:

- The sender receives a triple ACK. This is done in *tcp_fastretrans_alert()* using the *is_dupack* variable.
- A timeout occurs, which causes *tcp_enter_loss()* to be called (see Section 6.6). In this case, the congestion window is set to 1 and *ssthresh* (the slow-start threshold) is set to half of the congestion window when the packet is lost. This last operation is done in *tcp_recalc_ssthresh()*.
- TX Queue is full. This is detected in *tcp_transmit_skb()* (the error is propagated from *q->enqueue* in the sub-IP layer) which calls *tcp_enter_cwr()*.
- SACK detects a hole.

Apart from these situations, the Linux kernel modifies the congestion window in several more places; some of these changes are based on standards, others are Linux specific. In the following sections, we describe these extra changes.

6.7.1 Undoing the Congestion Window

One of the most logically complicated parts of the Linux kernel is when it decides to undo a congestion window update. This happens when the kernel finds that a window reduction should not have been made. This can be found in two ways: the receiver can inform by a duplicate SACK (D-SACK) that the incoming segment was already received; or the Linux TCP sender can detect unnecessary retransmissions by using the TCP timestamp option attached to each TCP header. These detections are done in the *tcp_fastretransmit_alert()*, which calls the appropriate undo operations depending in which state the Linux machine is: *tcp_try_undo_recovery()*, *tcp_undo_cwr()*, *tcp_try_undo_dsack()*, *tcp_try_undo_partial()* or *tcp_try_undo_loss()*. They all call *tcp_may_undo_loss()*.

6.7.2 Congestion Window Moderation

Linux implements the function *tcp_moderate_cwnd()*, which reduces the congestion window whenever it thinks that there are more packets in flight than there should be based on the value of *snd_cwnd*. This feature is specific to Linux and is specified neither in an IETF RFC nor in an Internet Draft. The purpose of the function is to prevent large transient bursts of packets from being sent out during "dubious conditions". This is often the case when an ACK acknowledges more than three packets. As a result, the magnitude of the congestion window reduction can be very large at large congestion window sizes, and hence reduce throughput.

The primary calling functions for *tcp_moderate_cwnd()* are *tcp_undo_cwr()*, *tcp_try_undo_recovery()*, *tcp_try_to_open()* and *tcp_fastretrans_alert()*. In all cases, the function call is triggered by conditions being met in *tcp_fast_retrans_alert()*.

6.7.3 Congestion Window Validation

Linux implements congestion window validation defined in RFC 2861 [15]. With this technique, the sender reduces the congestion window size if it has not been fully used for one RTO estimate's worth of time.

This is done by *tcp_cwnd_restart()*, which is called if necessary by *tcp_event_data_sent()*. The function *tcp_event_data_sent()* is called by *tcp_transmit_skb()* every time TCP transmits a packet.

7 UDP

This section reviews the UDP part of the networking code in the Linux kernel. This is a significantly simpler piece of code than the TCP part. The absence of reliable delivery and congestion control allows for a very simple design.

Most of the UDP code is located in one file: `net/ipv4/udp.c`

The UDP layer is depicted in Figure 15. When a packet arrives from the IP layer through *ip_local_delivery()*, it is passed on to *udp_rcv()* (this is the equivalent of *tcp_v4_rcv()* in the TCP part). The function *udp_rcv()* puts the packet in the socket queue for the user application with *sock_put()*. This is the end of the delivery of the packet.

When the user reads the packet, e.g. with the *recvmsg()* system call, *inet_recvmsg()* is called, which in this case calls *udp_recvmsg()*, which calls *skb_rcv_datagram()*. The function *skb_rcv_datagram()* then gets the packets from the queue and fills the data structure that will be read in user space.

When a packet arrives from the user, the process is simpler. The function *inet_sendmsg()* calls *udp_sendmsg()*, which builds 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).

Once 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()*. If, for some reason, the packet could not be transmitted (e.g., if the outgoing ring buffer is full), the error is propagated to *udp_sendmsg()*, which updates statistics (nothing else is done because UDP is a non-reliable protocol).

Once the IP packet has been built, it is passed on to *ip_output()*, which finalizes the delivery of the packet to the lower layers.
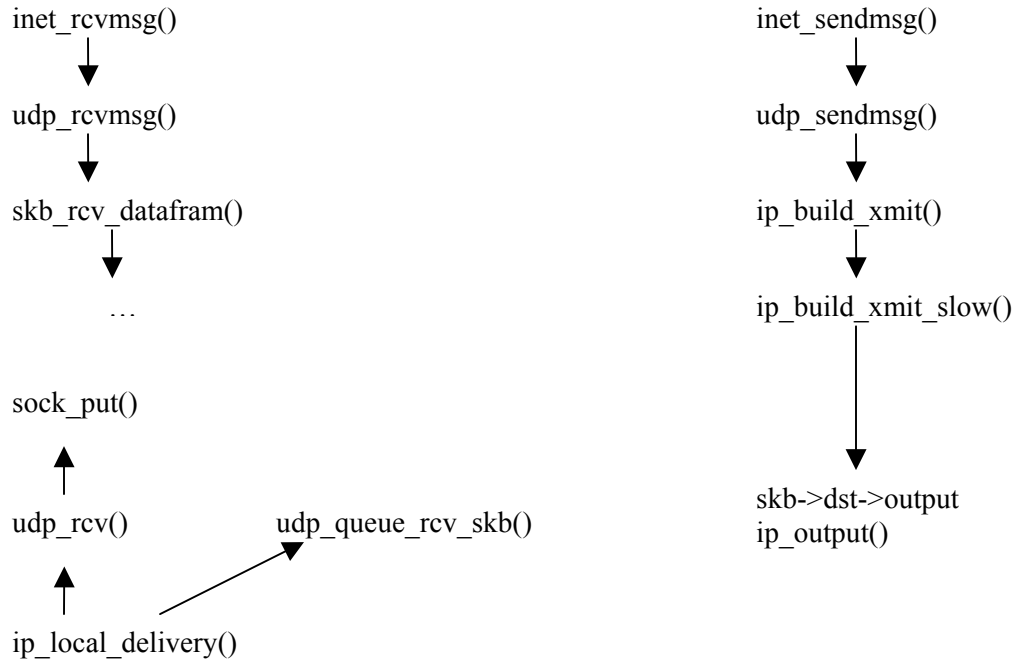
inet_rcvmsg()	inet_sendmsg()
↓	↓
udp_rcvmsg()	udp_sendmsg()
↓	↓
skb_rcv_datafram()	ip_build_xmit()
↓	↓
…	ip_build_xmit_slow()
	↓
sock_put()	
↑	skb->dst->output
udp_rcv() udp_queue_rcv_skb()	ip_output()
↑ ↗	
ip_local_delivery()	

Figure 15: UDP

8   The *socket* API

The previous sections have identified events inside the kernel. The main "actor" of the previous sections was the packet. In this section, we explain the relationships between events in system space and events in user space.

Applications use the socket interface to create connections to other hosts and/or to send information to the other end. We emphasize the chain of events generated in the TCP code when the *connect()* system called is used.

All network system calls reach *sys_socketcall()*, which gets the call parameters from the user (*copy_from_user(a, args, nargs[call])*) and calls the appropriate kernel function.

8.1   socket()

When a user invokes the *socket()* system call, this calls *sys_socket()* inside the kernel (see file `net/socket.c`). The *sys_socket()* function does two simple things. First, it calls *sock_create()*, which allocates a new *sock* structure where all the information about the socket/connection is

stored. Second, it calls *sock_map_fd()*, which maps the socket to a file descriptor. In this way, the application can access the socket as if it were a file—a typical Unix feature.

8.2   bind()

The *bind()* system call triggers *sys_bind()*, which simply puts information about the destination address and port in the *sock* structure.

8.3   listen()

The *listen()* system call, which triggers *sys_listen()*, calls the appropriate listen function for this protocol. This is pointed by *sock->ops->listen(sock, backlog)*. In the case of TCP, the listen function is *inet_listen()*, which in turn calls *tcp_listen_start()*.

8.4   accept() and connect()

The *accept()* system call triggers *sys_accept()*, which calls the appropriate accept function for that protocol (see *sock->ops->accept()*). In the case of TCP, the accept function is *tcp_accept()*.

When a user invokes the *connect()* system call, the function *sys_connect()* is called inside the kernel. UDP has no *connect* primitive because it is a connectionless protocol. In the case of TCP, the function *tcp_connect()* is called (by calling *sock->ops->connect()* on the socket). The *tcp_connect()* function initializes several fields of the *tcp_opt* structure, and an *skb* for the SYN packet is filled and transmitted at the end of the function.

Meanwhile, the server has created a socket, bound it to a port and called *listen()* to wait for a connection. This changed the state of the socket to LISTENING. When a packet arrives (which will be the TCP SYN packet sent by the client), this is dealt with by *tcp_rcv_state_process()*. The server then replies with a SYNACK packet that the client will process in *tcp_rcv_synsent_state_process()*; this is the state that the client enters after sending a SYN packet.

Both *tcp_rcv_state_process()* (in the server) and *tcp_rcv_sysent_state_process()* (in the client) have to initialize some other data in the *tcp_opt* structure. This is done by calling *tcp_init_metrics()* and *tcp_initialize_rcv_mss()*.

Both the server and the client acknowledge these packets and enter the ESTABLISHED state. From now on, every packet that arrives is handled by *tcp_rcv_established()*.

8.5   write()

Every time a user writes in a socket, this goes through the socket linkage to *inet_sendmsg()*. The function *sk->prot->sendmsg()* is called, which in turn calls *tcp_sendmsg()* in the case of TCP or *udp_sendmsg()* in the case of UDP. The next chain of events was described in the previous sections.

8.6 close()

When the user closes the file descriptor corresponding to this socket, the file system code calls *sock_close()*, which calls *sock_release()* after checking that the *inode* is valid. The function *sock_release()* calls the appropriate release function, in our case *inet_release()*, before updating the number of sockets in use. The function *inet_release()* calls the appropriate protocol-closing function, which is *tcp_close()* in the case of TCP. The latter function sends an active reset with *tcp_send_active_reset()* and sets the state to TCP_CLOSE_WAIT.

9 Conclusion

In this technical report, we have documented how the networking code is structured in release 2.4.20 of the Linux kernel. First, we gave an overview, showing the relevant branches of the code tree and explaining how incoming and outgoing TCP segments are handled. Next, we reviewed the general data structures (*sk_buff* and *sock*) and detailed TCP options. Then, we described the sub-IP layer and highlighted the difference in the handling of interrupts between NAPI-based and pre-NAPI device drivers; we also described interrupt coalescence, an important technique for gigabit end-hosts. In the next section, we described the network layer, which includes IP, ARP and ICMP. Then we delved into TCP and detailed TCP input, TCP output, SACKs, QuickACKs, timeouts and ECN; we also documented how TCP's congestion window is adjusted. Next, we studied UDP, whose code is easier to understand than TCP's. Finally, we mapped the socket API, well-known to Unix networking programmers, to kernel functions.

The need for such a document arises from the current gap between the abundant literature aimed at Linux beginners and the Linux kernel mailing list where Linux experts occasionally distil some of their wisdom. Because the technology evolves quickly and the Linux kernel code frequently undergoes important changes, it would be useful to keep up-to-date descriptions of different parts of the kernel (not just the networking code). We have experienced that this is a time-consuming endeavor, but documenting entangled code (the Linux kernel code notoriously suffers from a lack of code clean-up and reengineering) is the only way for projects like ours to understand in detail what the problems are, and to devise a strategy for solving them.

For the sake of conserving time, several important aspects have not been considered in this document. It would be useful to document how the IPv6 code is structured, as well as the Stream Control Transmission Protocol (SCTP). The description of SACK also deserves more attention, as we have realized that this part of the code is sub-optimal and causes problems in long-distance gigabit networks. Last, it would be useful to update this document to a 2.6.x version of the kernel.

Acknowledgments

Acronyms

ACK	Acknowledgment
API	Application Programming Interface
ARP	Address Resolution Protocol
ATM	Asynchronous Transfer Mode
BOOTP	Boot Protocol
CBQ	Class-Based Queuing
CPU	Central Processing Unit
DMA	Direct Memory Access
DupACK	Duplicate Acknowledgment
ECN	Explicit Congestion Notification
FIB	Forward Information Base
FIFO	First In First Out
ICMP	Internet Control Message Protocol
IGMP	Internet Group Management Protocol
IETF	Internet Engineering Task Force
I/O	Input/Output
IP	Internet Protocol
IPv4	IP version 4
IPv6	IP version 6
IRQ	Interrupt Request
ISR	Interrupt Service Routine
LAN	Local Area Network
MAC	Media Access Control
MSS	Maximum Segment Size
MTU	Maximum Transfer Unit

NAPI	New Application Programming Interface
NIC	Network Interface Card
PAWS	Protect Against Wrapped Sequence numbers
PIO	Program Input/Output
QuickACK	Quick Acknowledgment
RED	Random Early Discard
RFC	Request For Comment (IETF specification)
RST	Reset (TCP state)
RTT	Round Trip Time
SACK	Selective Acknowledgment
SCTP	Stream Control Transmission Protocol
SMP	Symmetric Multi-Processing
SYN	Synchronize (TCP state)
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

References

[1] Linux kernel 2.4.20. Available from *The Linux Kernel Archives* at: http://www.kernel.org/pub/linux/kernel/v2.4/patch-2.4.20.bz2

[2] M. Allman, V. Paxson and W. Stevens, *RFC 2581: TCP Congestion Control*, IETF, April 1999.

[3] J. Crowcroft and I. Phillips, *TCP/IP & Linux Protocol Implementation: Systems Code for the Linux Internet*, Wiley, 2002.

[4] J.H. Salim, R. Olsson and A. Kuznetsov, "Beyond Softnet". In *Proc. Linux 2.5 Kernel Developers Summit*, San Jose, CA, USA, March 2001. Available at <http://www.cyberus.ca/~hadi/usenix-paper.tgz>.

[5] J. Cooperstein, *Linux Kernel 2.6 – New Features III: Networking*. Axian, January 2003. Available at <http://www.axian.com/pdfs/linux_talk3.pdf>.

[6] W.R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, 1994.

[7] G.R. Wright and W.R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley, 1995.

[8]   M. Mathis, J. Mahdavi, S. Floyd and A. Romanow, *RFC 2018, TCP Selective Acknowledgment Options*, IETF, October 1996.

[9]   S. Floyd, J. Mahdavi, M. Mathis and M. Podolsky, *RFC 2883: An Extension to the Selective Acknowledgement (SACK) Option for TCP*, IETF, July 2000.

[10]   Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*, 2nd Edition, O'Reilly, 2002.

[11]   A. Rubini and J. Corbet, *Linux Device Drivers*, 2nd Edition, O'Reilly, 2001.

[12]   http://tldp.org/HOWTO/KernelAnalysis-HOWTO-5.html

[13]   http://www.netfilter.org/unreliable-guides/kernel-hacking/lk-hacking-guide.html

[14]   V. Jacobson, R. Braden and D. Borman, *RFC 1323: TCP Extensions for High Performance*, IETF, May 1992.

[15]   M. Handley, J. Padhye and S. Floyd, *RFC 2861: TCP Congestion Window Validation*, IETF, June 2000.

[16]   http://www.netfilter.org/

[17]   J. C. Mogul and K. K. Ramakrishnan. "Eliminating Receive Livelock in an Interrupt-Driven Kernel". In *Proc. of the 1996 Usenix Technical Conference*, pages 99–111, 1996.

Biographies

Miguel Rio is a Lecturer at the Department of Electronic and Electrical Engineering, University College London. He previously worked on Performance Evaluation of High Speed Networks in the DataTAG and MBNG projects and on Programmable Networks on the Promile project. He holds a Ph.D. from the University of Kent at Canterbury, as well as M.Sc. and B.Sc. degrees from the University of Minho, Portugal. His research interests include Programmable Networks, Quality of Service, Multicast and Protocols for Reliable Transfers in High-Speed Networks.

Mathieu Goutelle is a Ph.D. student in the INRIA RESO team of the LIP Laboratory at ENS Lyon. He is a member of the DataTAG Project and currently works on the behavior of TCP over a DiffServ-enabled gigabit network. In 2002, he graduated as a generalist engineer (equiv. to an M.Sc. in electrical and mechanical engineering) from Ecole Centrale in Lyon, France. In 2003, he received an M.Sc. in Computer Science from ENS Lyon.

Tom Kelly received a Mathematics degree from the University of Oxford in July 1999. His Ph.D. research on "Engineering Internet Flow Controls" was completed in February 2004 at the University of Cambridge. He has held research positions as an intern at AT&T Labs Research in 1999, an intern at the ICSI Center for Internet Research in Berkeley during 2001, and an IPAM research fellowship at UCLA in 2002. During the winter of 2002–03 he worked for CERN on the EU DataTAG project implementing the Scalable TCP proposal for high-speed wide area data transfer. His research interests include middleware, networking, distributed systems and computer architecture.

Richard Hughes-Jones leads e-science and Trigger and Data Acquisition development in the Particle Physics group at Manchester University. He has a Ph.D. in Particle Physics and has worked on Data Acquisition and Network projects for over 20 years, including evaluating and

field-testing OSI transport protocols and products. He is secretary of the Particle Physics Network Coordinating Group which has the remit to support networking for PPARC funded researchers. Within the UK GridPP project he is deputy leader of the network workgroup and is active in the DataGrid networking work package (WP7). He is also responsible for the High Throughput investigations in the UK e-Science MB-NG project to investigate QoS and various traffic engineering techniques including MPLS. He is a member of the Global Grid Forum and is co-chair of the Network Measurements Working Group. He was a member of the Program Committee of the 2003 PFLDnet workshop, and is a member of the UKLIGHT Technical Committee. His current interests are in the areas of real-time computing and networking including the performance of transport protocols over LANs, MANs and WANs, network management and modeling of Gigabit Ethernet components.

J.P. Martin-Flatin is Technical Manager of the European FP5/IST DataTAG Project at CERN, where he coordinates research activities in gigabit networking, Grid networking and Grid middleware. Prior to that, he was a principal technical staff member with AT&T Labs Research in Florham Park, NJ, USA, where he worked on distributed network management, information modeling and Web-based management. He holds a Ph.D. degree in Computer Science from the Swiss Federal Institute of Technology in Lausanne (EPFL). His research interests include software engineering, distributed systems and IP networking. He is the author of a book, *Web-Based Management of IP Networks and Systems*, published in 2002 by Wiley. He is a senior member of the IEEE and a member of the ACM. He is a co-chair of the GGF Data Transport Research Group and a member of the IRTF Network Management Research Group. He was a co-chair of GNEW 2004 and PFLDnet 2003.

Yee-Ting Li received an M.Sc. degree in Physics from the University of London in August 2001. He is now studying for a Ph.D. with the Centre of Excellence in Networked Systems at University College London, UK. His research interests include IP-based transport protocols, network monitoring, Quality of Service (QoS) and Grid middleware.