TCP Receive

The *tcp_v4_rcv()* function defined in *ipv4/tcp_ipv4.c* is the main entry point for delivery of datagrams from the IP layer.   I thought this test for PACKET_HOST had been done before.

```
int tcp_v4_rcv(struct sk_buff *skb)
{
    struct tcphdr *th;
    struct sock *sk;
    int ret;

    if (skb->pkt_type != PACKET_HOST)
       goto discard_it;

    /* Count it even if it's bad */
    TCP_INC_STATS_BH(TCP_MIB_INSEGS);
```

As usual, the *pskb_may_pull()* function is responsible for ensuring that the TCP header, and in the next call, the header options are in the *kmalloc'ed* portion of the *sk_buff*.

```
    if (!pskb_may_pull(skb, sizeof(struct tcphdr)))
       goto discard_it;

    th = skb->h.th;

```

Ensure that the offset to the data in words exceeds size of tcp header.

```
    if (th->doff < sizeof(struct tcphdr) / 4)
       goto bad_packet;
```

Ensure the options are in the *kmalloc'ed* part.

```
    if (!pskb_may_pull(skb, th->doff * 4))
       goto discard_it;

```

Some other sanity checks such as data offset and packet length are validated later, but if the checksum (which covers the whole packet is bad the packet is ditched here.

```
/* An explanation is required here, I think.
 * Packet length and doff are validated by header prediction,
 * provided case of th->doff==0 is eliminated.
 * So, we defer the checks. */

if ((skb->ip_summed != CHECKSUM_UNNECESSARY &&
     tcp_v4_checksum_init(skb)))
   goto bad_packet;
```

Header processing

Attributes of the TCP header are collected in the control buffer. Numeric fields must be converted to host byte order.

- The value of *seq* is the sequence number of the first byte of this segment. *th->ack_seq* is the sequence number of the next byte the other end expects to receive from us.
- The value of *end_seq* is the sequence number just past the last byte carried in this segment.
- The unusual addition of the *th->syn* and *th->fin* bitfields is required because each consumes one byte of the sequence number space.
- The subtraction of *th->doff * 4* is removing the length of the TCP header.

```
th = skb->h.th;
TCP_SKB_CB(skb)->seq = ntohl(th->seq);
TCP_SKB_CB(skb)->end_seq =
       (TCP_SKB_CB(skb)->seq + th->syn + th->fin +
        skb->len - th->doff * 4);
TCP_SKB_CB(skb)->ack_seq = ntohl(th->ack_seq);
TCP_SKB_CB(skb)->when    = 0;
TCP_SKB_CB(skb)->flags   = skb->nh.iph->tos;
TCP_SKB_CB(skb)->sacked  = 0;

```

Socket lookup

The *__inet_lookup()* function used to be called *__tcp_v4_lookup()*. It attempts to find a *struct sock* that may be associated with this packet. Lookup elements include source and destination IP and port addresses along with any bound input interface.

```
    sk = __inet_lookup(&tcp_hashinfo,
            skb->nh.iph->saddr, th->source,
             skb->nh.iph->daddr, ntohs(th->dest),
             inet_iif(skb));

    if (!sk)
       goto no_tcp_socket;
```

TCP state is checked and the packet is passed to two filters.

```
process:
    if (sk->sk_state == TCP_TIME_WAIT)
       goto do_time_wait;

    if (!xfrm4_policy_check(sk, XFRM_POLICY_IN, skb))
       goto discard_and_relse;
    nf_reset(skb);

    if (sk_filter(sk, skb, 0))
       goto discard_and_relse;

    skb->dev = NULL;

```

Locking considerations

Sockets have a lock structure that is used in different ways depending upon whether the caller is a top-down or bottom-up caller. Bottom up callers block under penalty of death and must use the *bh_lock_sock()* functions. These functions obtain a spinlock that protects the updating of *lock.users* but the value of *lock.users* is the *real indicator* of whether or not the *sock* is available.

```
bh_lock_sock_nested(sk);
ret = 0;
```

If the socket is not locked, then *lock.users* will be 0 and *tcp_prequeue()* will be called to attempt to queue the packet on *tp->ucopy.prequeue* . If there is no process blocked waiting to consume data on the socket, then that won't work, and the packet must be processed immediately via the call to *tcp_v4_do_rcv()*. If the socket is locked, the packet must be added to the backlog queue.

```
if (!sock_owned_by_user(sk)) {
----------- net DMA stuff -----------
   {
      if (!tcp_prequeue(sk, skb))
      ret = tcp_v4_do_rcv(sk, skb);
   }
} else
   sk_add_backlog(sk, skb);
bh_unlock_sock(sk);

sock_put(sk);

return ret;
```

The *owner* variable is actually a Boolean flag. A value of 0 means that no application process owns the socket at present. A value of 1 means that *some* application process owns the socket.

```
#define sock_owned_by_user(sk)  ((sk)->sk_lock.owner)
```

Exit from *tcp_v4_rcv()*

This is the end of *tcp_v4_rcv()*. For a "regular" data packet on of three outcomes will have occurred:

- Packet is left on the prequeue (socket not locked and receiver sleeping in *tcp_recvmsg())*
- Packet is left on the backlog queue (socket locked... receiver active in *tcp_send/recvmsg*)
- Packet is processed (socket not locked and no receiver sleeping in *tcp_recvmsg())*

The first two outcomes are "lightweight" in terms of processing.

- In the first case the actual processing which occurs in *tcp_v4_do_rcv()* is performed in the context of the application that will actually receive the packet.

- In the second case, it will occur in the context of an application that owns the socket.

- In the *least desirable* third case it will be performed immediately in the context of the softirq. Note that in the first two cases *ack* generation is also deferred.

- In the COP protocol *ALL* packet processing is carried out as in the third case.

Handling of unsual conditions

The rest of this code handles exceptional conditions.

```
no_tcp_socket:
   if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))
      goto discard_it;

   if (skb->len < (th->doff << 2)
               || tcp_checksum_complete(skb)) {
bad_packet:
      TCP_INC_STATS_BH(TCP_MIB_INERRS);
   } else {
      tcp_v4_send_reset(skb);
   }

discard_it:
   /* Discard frame. */
   kfree_skb(skb);
   return 0;

discard_and_relse:
   sock_put(sk);
   goto discard_it;

```

Time wait processing

```
do_time_wait:
   if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
      inet_twsk_put((struct inet_timewait_sock *) sk);
      goto discard_it;
   }
   switch (tcp_timewait_state_process(
                    (struct inet_timewait_sock *)sk,
                  skb, th)) {
   case TCP_TW_SYN: {
      struct sock *sk2 = inet_lookup_listener(&tcp_hashinfo,
                     skb->nh.iph->daddr,
                     ntohs(th->dest),
                     inet_iif(skb));
      if (sk2) {
         inet_twsk_deschedule((struct inet_timewait_sock *)sk,
                    &tcp_death_row);
         inet_twsk_put((struct inet_timewait_sock *)sk);
         sk = sk2;
         goto process;
      }
      /* Fall through to ACK */
   }
   case TCP_TW_ACK:
      tcp_v4_timewait_ack(sk, skb);
      break;
   case TCP_TW_RST:
      goto no_tcp_socket;
   case TCP_TW_SUCCESS:;
   }
   goto discard_it;
}
```

The socket locking mechanism

In this section we take a more detailed look at the locking mechanism..

```
/* Used by processes to "lock" a socket state, so that
 * interrupts and bottom half handlers won't change it
 * from under us. It essentially blocks any incoming
 * packets, so that we won't get any new data or any
 * packets that change the state of the socket.
 *
 * While locked, BH processing will add new packets to
 * the backlog queue.  This queue is processed by the
 * owner of the socket lock right before it is released.
 *
 * Since ~2.3.5 it is also exclusive sleep lock serializing
 * accesses from user process context.
*/
#define sock_owned_by_user(sk)  ((sk)->sk_lock.owner)

```

Application context socket locking

The *lock_sock()* function may be used by *top half callers* which are allowed to block. If the number of users is already 1, *__lock_sock()* will be called and the process will block.

In *__lock_sock()* the spinlock will be dropped and retaken after the process wakes up and will return with the lock held.

```
void fastcall lock_sock(struct sock *sk)
{
        might_sleep();
        spin_lock_bh(&sk->sk_lock.slock);
        if (sk->sk_lock.owner)
                __lock_sock(sk);

        sk->sk_lock.owner = (void *)1;
        spin_unlock(&sk->sk_lock.slock);
        /*
         * The sk_lock has mutex_lock() semantics here:
         */
        mutex_acquire(&sk->sk_lock.dep_map, 0, 0, _RET_IP_);
        local_bh_enable();
}
```

The __*lock_sock* function.

Use of exclusive wait ensures FIFO (instead of thundering herd) queuing of processes blocked waiting for the socket. Note that it is almost always the case that there will be at most ONE process that is waiting in any case.

```
1232 static void __lock_sock(struct sock *sk)
1233 {
1234    DEFINE_WAIT(wait);
1235
1236    for(;;) {
1237        prepare_to_wait_exclusive(&sk->sk_lock.wq, &wait,
1238                                     TASK_UNINTERRUPTIBLE);
1239        spin_unlock_bh(&sk->sk_lock.slock);
1240        schedule();
```

After wakeup, the lock is obtained again and if the ownership value is 0, then the socket is free. The ownership value will be set to 1 in line 1540 above. This is safe because the spin lock is held ensuring that the the ownership cannot change "out from under us".

```
1241        spin_lock_bh(&sk->sk_lock.slock);
1242        if(!sock_owned_by_user(sk))
1243            break;
1244    }
1245    finish_wait(&sk->sk_lock.wq, &wait);
1246 }
```

The *bh_lock_sock()* facility

The *bh_lock_sock()* macro just obtains the spin lock. The spin lock is held by the caller while the caller checks to see if *lock.users* is 0. If it is not 0, then the bottom half, *which is not permitted to block*, must take a course of action that doesn't require the lock. This action is typically to queue the packet on the *backlog* queue.

In my view the explosive growth of *bh_lock** and *spin_lock** should be controlled. If it can't be controlled, then the entire locking architecture should be rethought.

```
/* BH context may only use the following locking interface. */
#define bh_lock_sock(__sk)
           spin_lock(&((__sk)->sk_lock.slock))

#define bh_lock_sock_nested(__sk) \
           spin_lock_nested(&((__sk)->sk_lock.slock), \
                                   SINGLE_DEPTH_NESTING)

#define bh_unlock_sock(__sk)
          spin_unlock(&((__sk)->sk_lock.slock))

void __lockfunc _spin_lock_nested(spinlock_t *lock,
                                       int subclass)
{
   preempt_disable();
   spin_acquire(&lock->dep_map, subclass, 0, _RET_IP_);
   _raw_spin_lock(lock);
}
```

Nested lock handlers -- new to kernel 2.6

This appears to be a new mechanism designed to support holding more than one lock at a time.. possibly a max of two! This is normally done in sane operating systems by using a hierarchical locking scheme.

```
void lock_acquire(struct lockdep_map *lock,
               unsigned int subclass,
               int trylock, int read, int check, unsigned long ip)
{
   unsigned long flags;

   if (unlikely(current->lockdep_recursion))
       return;

   raw_local_irq_save(flags);
   check_flags(flags);

   current->lockdep_recursion = 1;
    __lock_acquire(lock, subclass, trylock, read, check,
                     irqs_disabled_flags(flags), ip);
   current->lockdep_recursion = 0;
   raw_local_irq_restore(flags);
}
```

```
#ifdef CONFIG_DEBUG_LOCK_ALLOC
# ifdef CONFIG_PROVE_LOCKING
#  define spin_acquire(l, s, t, i)
              lock_acquire(l, s, t, 0, 2, i)
# else
#  define spin_acquire(l, s, t, i)
               lock_acquire(l, s, t, 0, 1, i)
# endif
# define spin_release(l, n, i)
                 lock_release(l, n, i)
#else
# define spin_acquire(l, s, t, i)   do { } while (0)
# define spin_release(l, n, i)      do { } while (0)
#endif
```

The socket release mechanism.

When the socket lock is released, if the *backlog* queue is not empty, then *__release_sock()* is called to drain the backlog queue and pass the packets to *tcp_v4_do_rcv()* for processing. If there are additional processes sleeping on the lock, since this is an EXCLUSIVE wait, one is awakened after the lock is released.

Note that an additional wait queue, *sk->sk_lock.wq* is associated with the socket.

```
void fastcall release_sock(struct sock *sk)
{
   /*
    * The sk_lock has mutex_unlock() semantics:
    */
   mutex_release(&sk->sk_lock.dep_map, 1, _RET_IP_);

   spin_lock_bh(&sk->sk_lock.slock);
   if (sk->sk_backlog.tail)
       __release_sock(sk);
   sk->sk_lock.owner = NULL;
   if (waitqueue_active(&sk->sk_lock.wq))
       wake_up(&sk->sk_lock.wq);
   spin_unlock_bh(&sk->sk_lock.slock);
}
```

Draining the backlog queue

Arriving packets are placed on the *backlog* queue when the *sock* is locked at arrival time. When the *sock* is unlocked they are passed to the *tcp_v4_do_rcv()*. To preserve packet order and improve efficiency, *this function must be called before the lock.owner field is reset to 0 via the call to bh_unlock_sock()*

Note that it uses *queue stealing* to assume ownership of the backlog queue before releasing exclusive ownership of the socket. This function is invoked only in *top_half* (application) context. The lock is dropped while the stolen queue is being drained and additional packets may be added to the backlog queue. The outer loop allows for multiple queue steals.

```
static void __release_sock(struct sock *sk)
{
   struct sk_buff *skb = sk->sk_backlog.head;

   do {
      sk->sk_backlog.head = sk->sk_backlog.tail = NULL;
      bh_unlock_sock(sk);

      do {
         struct sk_buff *next = skb->next;

         skb->next = NULL;
```

Here *sk->backlog_rcv()* is *tcp_v4_do_rcv().*

```
         sk->sk_backlog_rcv(sk, skb);

      /*
       * We are in process context here with softirqs
       * disabled, use cond_resched_softirq() to preempt.
       * This is safe to do because we've taken the backlog
       * queue private:
       */
         cond_resched_softirq();

         skb = next;
      } while (skb != NULL);

   bh_lock_sock(sk);
 } while((skb = sk->sk_backlog.head) != NULL);
}
```

The TCP prequeue mechanism.

The prequeue is managed via the *ucopy* substructure of the *tcp_sock.*

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

        } ucopy;
```

prequeue:	The queue itself
task:	Task struct of process sleeping on receive queue
iov:	Iovec associated with pending receive
memory:	Total amount of memory occupied by prequeue packets
len:	Remaining space in user supplied buffer.

Prequeue processing

When a packet arrives, the first objective is to simply place it on the *&tp->ucopy.prequeue*. The *sk->lock* must be held on entry to this function. If is *tp->ucopy.task* is NULL, then there is no sleeping task to wake up and *the prequeue is bypassed.* Hence the only time the prequeue can be used is when there is a process blocked waiting to receive data on this socket.

The use of the prequeue permits data to be processed in the context of the application to which it will eventually be delivered.

```
static inline int tcp_prequeue(struct sock *sk,
                                  struct sk_buff *skb)
{
   struct tcp_sock *tp = tcp_sk(sk);

   if (!sysctl_tcp_low_latency && tp->ucopy.task) {
        __skb_queue_tail(&tp->ucopy.prequeue, skb);
       tp->ucopy.memory += skb->truesize;
```

If the amount of data on the prequeue exceeds the receive buffer quota, the prequeue is drained and packets are passed to *sk->sk_backlog_rcv* which ponts *tcp_v4_do_rcv().* This function is the analog of *cop_rcv_ad()* and it may be invoked from either the top half or the bottom half context. This is in contrast to *cop_rcv_ad()* which is bottom half only.

```
       if (tp->ucopy.memory > sk->sk_rcvbuf) {
          struct sk_buff *skb1;

          BUG_ON(sock_owned_by_user(sk));

          while ((skb1 = __skb_dequeue(&tp->ucopy.prequeue))
                                    != NULL) {
             sk->sk_backlog_rcv(sk, skb1);
             NET_INC_STATS_BH(LINUX_MIB_TCPPREQUEUEDROPPED);
          }

         tp->ucopy.memory = 0;
```

Unblocking the application

If the amount of data on the prequeue doesn't exceed the buffer quota, *and* this is the first packet added to the prequeue, the application is awakened. Presumably, the objective of all of this is to cause as much processing as possible to be done in the context of the application that is actually receiving the data.

```
    } else if (skb_queue_len(&tp->ucopy.prequeue) == 1) {
        wake_up_interruptible(sk->sk_sleep);
        if (!inet_csk_ack_scheduled(sk))
           inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
                                     (3 * TCP_RTO_MIN) / 4,
                                     TCP_RTO_MAX);
               }
     return 1;
    }
    return 0;
}
```

The *sk_add_backlog()* function

This function adds a packet to the tail of the backlog queue. *The sk->sk_lock.spinlock must be held prior to calling this function!*

```
static inline void sk_add_backlog(struct sock *sk,
              struct sk_buff *skb)
{
   if (!sk->sk_backlog.tail) {
      sk->sk_backlog.head = sk->sk_backlog.tail = skb;
   } else {
      sk->sk_backlog.tail->next = skb;
      sk->sk_backlog.tail = skb;
   }
   skb->next = NULL;
}
```

The *tcp_recvmsg()* function.

This function is the analog of *udp_recvmsg()*. It is handles the *recvmsg()* system call and thus executes in an application context.

```
This routine copies from a sock struct into the user buffer.

Technical note: in 2.3 we work on _locked_ socket, so that
tricks with *seq access order and skb->users are not required.
Probably, code can be easily improved even more.
```

```
int tcp_recvmsg(struct kiocb *iocb, struct sock *sk,
                    struct msghdr *msg,
           size_t len, int nonblock, int flags, int *addr_len)
{
        struct tcp_sock *tp = tcp_sk(sk);
        int copied = 0;
        u32 peek_seq;
        u32 *seq;       /* Pointer to value holding sequence
                           number of head of unread data */
        unsigned long used;
        int err;
        int target;     /* Read at least this many bytes */
        long timeo;
        struct task_struct *user_recv = NULL;
        int copied_early = 0;
```

The caller will block here until the *struct sock* can be locked.

```
        lock_sock(sk);

        TCP_CHECK_TIMER(sk);

```

State verification

It is not permissible to receive data on a socket that is in the LISTEN state.

```
    err = -ENOTCONN;
    if (sk->sk_state == TCP_LISTEN)
       goto out;

    timeo = sock_rcvtimeo(sk, nonblock);

    /* Urgent data needs to be handled specially. */
    if (flags & MSG_OOB)
       goto recv_urg;
```

Setting the u32 *seq pointer

The value of **seq* is the *next* sequence number to be consumed by the application code. A comment in the code calls it the ``*head of yet unread data''*. It normally points to permanent holder in the tcpsock, but for peek requests it is redirected to a local variable.

```
    seq = &tp->copied_seq;
    if (flags & MSG_PEEK) {
       peek_seq = tp->copied_seq;
       seq = &peek_seq;
    }

```

The objective of the *low water* test is to set the *target* variable which is the minimum number of bytes that must be consumed before this *tcp_recvmsg* completes.

```
    target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);

```

Back in *sock_initdata()* the value of *sk->sk_rcvlowat* was set to 1 word.

```
static inline int sock_rcvlowat(const struct sock *sk,
                                         int waitall, int len)
{
     return (waitall ? len : min_t(int,
                         sk->sk_rcvlowat, len)) ? : 1;
}
```

The main receive loop.

This is the main receive loop. This loop contains subsections that process the different receive queues, and plenty of *goto's* that make life more interesting.

```
do {
   struct sk_buff *skb;
   u32 offset;

* Are we at urgent data?
  Stop if we have read anything or have SIGURG pending. */

   if (tp->urg_data && tp->urg_seq == *seq) {
      if (copied)
         break;
      if (signal_pending(current)) {
         copied = timeo ? sock_intr_errno(timeo) : -EAGAIN;
         break;
      }
   }
```

The receive queue.

Packets are placed in the *receive queue* by *tcp_v4_do_rcv.* The first step is to try to consume a buffer from the *receive* queue. Unlike the backlog queue, prequeue and out of order queue. Packets in the *receive queue* are

(1) already acked
(2) guaranteed in order
(3) contain no holes but
(4) apparently may contain overlapping data

If the *receive* queue is not empty *rcv_nxt* will be the next byte beyond its end.

Processing the receive queue

If the receive queue is empty then the *prequeue* and the *backlog* queue will be processed. The "before" test is possible because segments are forced to be in order on the *receive_queue*. If the test fails in means the first byte of the the packet on the receive queue has a sequence number that is after the next byte of unconsumed data ... i.e. there is a hole.

```
        /* Next get a buffer. */

        skb = skb_peek(&sk->sk_receive_queue);
        do {
           if (!skb)
              break;

           /* Now that we have two receive queues this
            * shouldn't happen.
            */
           if (before(*seq, TCP_SKB_CB(skb)->seq)) {
              printk(KERN_INFO "recvmsg bug: copied %X "
                     "seq %X\n", *seq, TCP_SKB_CB(skb)->seq);
              break;
           }
```

Presumably the normal case here is for offset to be 0, but it might be a positive number if this segment contains both retransmitted and new data. If the segment contains all retransmitted data it shouldn't have been placed on this queue in the first place. At any rate the value of *offset* indicates where in this *sk_buff* the new data not yet consumed starts.

In the normal case, the *goto* will cause the subsequent queue processing loops to be passed over.

```
           offset = *seq - TCP_SKB_CB(skb)->seq;
           if (skb->h.th->syn)
              offset--;
           if (offset < skb->len)
              goto found_ok_skb;
           if (skb->h.th->fin)
              goto found_fin_ok;
           BUG_TRAP(flags & MSG_PEEK);
           skb = skb->next;
        } while (skb != (struct sk_buff *)
                               &sk->sk_receive_queue);
```

Testing for "receive complete" conditions

Falling into this section means that either the receive queue was empty or it didn't contain any usable data. Although the comment indicates that it is necessary to process the *backlog* queue, there is no evidence of the *backlog* queue actually being processed here.

What is actually happening is various conditions are tested to see if this receive operation should be allowed to return to the user. The value of *copied* is initially 0. It seems to normally represent the number of bytes that have been copied to user space, but also as a catchall to indicate error situations. The value of *target* is the minimum number of bytes that must (normally) be returned to the user.

If at least target bytes have been consumed and the backlog queue is empty, then an exit from the main loop is made.

```
    /* Well, if we have backlog, try to process it now yet. */

      if (copied >= target && !sk->sk_backlog.tail)
         break;

      if (copied) {    // some data copied
         if (sk->sk_err ||
             sk->sk_state == TCP_CLOSE ||
             (sk->sk_shutdown & RCV_SHUTDOWN) ||
             !timeo ||
             signal_pending(current) ||
             (flags & MSG_PEEK))
            break;
```

```
} else {      // no data copied
   if (sock_flag(sk, SOCK_DONE))
      break;

   if (sk->sk_err) {
      copied = sock_error(sk);  // copied overloaded :-(
      break;
   }

   if (sk->sk_shutdown & RCV_SHUTDOWN)
      break;

   if (sk->sk_state == TCP_CLOSE) {
      if (!sock_flag(sk, SOCK_DONE)) {
         /* This occurs when user tries to read
          * from never connected socket.
          */
         copied = -ENOTCONN;
         break;
      }
      break;
   }

   if (!timeo) {
      copied = -EAGAIN;
      break;
   }

   if (signal_pending(current)) {
      copied = sock_intr_errno(timeo);
      break;
   }
}
```

The call to tcp_cleanup_rbuf() is primarily concerned with the fact that it may be necessary to schedule a window update type ack if the application has now consumed some data.

```
tcp_cleanup_rbuf(sk, copied);

```

Establishing the copy to user space parameters

Arrival at this spot indicates

- the receive queue is empty,
- no serious errors or state changes were noted and
- we haven't consumed sufficient data to return to the caller.

The *ucopy.task* variable appears to be used to ensure that when data is copied to user space it is being copied in the context of the process that first requested the data! Needless to say this would be a good idea. The value of *user_recv* is initially set to NULL at the front end of this function.

Presumably the normal case is for *both* to be NULL here on the first trip through the loop and to not change thereafter. If two processes are consuming from the same socket and one of them is already sleeping on the receive queue, the tp->ucopy.task will not be NULL here. The value of *len* is the user specified buffer length which is saved here in *tp->ucopy.len.*

```

        if (!sysctl_tcp_low_latency && tp->ucopy.task ==
             user_recv) {
           /* Install new reader */
          if (!user_recv && !(flags & (MSG_TRUNC | MSG_PEEK))) {
              user_recv = current;
              tp->ucopy.task = user_recv;
              tp->ucopy.iov = msg->msg_iov;
           }

           tp->ucopy.len = len;

           BUG_TRAP(tp->copied_seq == tp->rcv_nxt ||
               (flags & (MSG_PEEK | MSG_TRUNC)));
```

Testing the prequeue for empty

```
   /* Ugly... If prequeue is not empty, we have to
    * process it before releasing socket, otherwise
    * order will be broken at second iteration.
    * More elegant solution is required!!!
    *
    * Look: we have the following (pseudo)queues:
    *
    * 1. packets in flight
    * 2. backlog
    * 3. prequeue
    * 4. receive_queue
    *
    * Each queue can be processed only if the next ones
    * are empty. At this point we have empty receive_queue.
    * But prequeue _can_ be not empty after second iteration,
    * when we jumped to start of loop because backlog
    * processing added something to receive_queue.
    * We cannot release_sock(), because backlog contains
    * packets arrived _after_ prequeued ones.
    *
    * Shortly, algorithm is clear --- to process all
    * the queues in order. We could make it more directly,
    * requeueing packets from backlog to prequeue, if
    * is not empty. It is more elegant, but eats cycles,
    * unfortunately.
    */
```

This location is where the diagnostic

Apr 21 12:17:27 vmlnx2b kernel: tcp_recvmsg: release prequeue 1

was inserted.

```
          if (!skb_queue_empty(&tp->ucopy.prequeue))
             goto do_prequeue;

          /* __ Set realtime policy in scheduler __ */
       }
```

Processing the backlog queue

Arrival here implies that the prequeue was empty. The *release/lock* combination here causes the contents of *backlog* queue to be passed to *tcp_v4_do_rcv()* which will (possibly??) copy them to user space or possibly place them on the *receive queue* or the out of order queue.

A *previous* test was:

```
if (copied >= target && !sk->sk_backlog.tail)
   break;
```

Therefore, arrival here with *copied* >= *target* implies the *backlog* queue is non-empty and we already have enough data to return to the caller. If *copied* < *target* we don't know how much data might be on the backlog queue so we just sleep here and depend on *tcp_v4_do_rcv() or tcp_prequeue()* to wake us up at the appropriate time. This backlog release *never occurred* in the monitored connection.

```
if (copied >= target) {
   /* Do not sleep, just process backlog. */
   release_sock(sk);
   lock_sock(sk);
```

Waiting for data

The caller will sleep here but the sock will be released before sleeping and locked just after wakeup. This used to be called *tcp_data_wait()*

```
     } else
        sk_wait_data(sk, &timeo);
```

```
int sk_wait_data(struct sock *sk, long *timeo)
{
   int rc;
   DEFINE_WAIT(wait);

   prepare_to_wait(sk->sk_sleep, &wait, TASK_INTERRUPTIBLE);
   set_bit(SOCK_ASYNC_WAITDATA, &sk->sk_socket->flags);
   rc = sk_wait_event(sk, timeo,
               !skb_queue_empty(&sk->sk_receive_queue));
   clear_bit(SOCK_ASYNC_WAITDATA, &sk->sk_socket->flags);
   finish_wait(sk->sk_sleep, &wait);
   return rc;
}
```

This is a clever macro that could probably simplify the of "wait" functions in COP.

```
#define sk_wait_event(__sk, __timeo,__condition)             \
({    int rc;                                                \
    release_sock(__sk);                                      \
    rc =__condition;                                         \
    if (!rc){                                                \
          *(__timeo) = schedule_timeout(*(__timeo));         \
    }                                                        \
    lock_sock(__sk);                                         \
    rc = __condition;                                        \
    rc;                                                      \
})
```

Adjusting the length of the available buffer space.

Arrival here means that we either

- released the backlog queue or
- woke up after waiting for the receive queue to be non-empty

So hopefully there is new data in the receive queue now.

If *user_recv* is not NULL here then it will point to the task struct of the current process. Here *len* is the length of the user supplied buffer. *tp->ucopy.len* was initially set to *len* but it would appear that it may be decremented in *tcp_rcv_established.* The NET_ADD_STATS call is designed to accumulate how many bytes get delivered in this way.

```
        if (user_recv) {
            int chunk;

            /* __ Restore normal policy in scheduler __ */

            if ((chunk = len - tp->ucopy.len) != 0) {
NET_ADD_STATS_USER(LINUX_MIB_TCPDIRECTCOPYFROMBACKLOG, chunk);
                len -= chunk;
                copied += chunk;
            }
```

Processing the prequeue

This test ensures that every thing that has been received correctly and in order has also been consumed (i.e. the *receive queue is empty)*. That will not be so if the draining of the backlog queue caused new packets to end up on the receive queue and in that case the prequeue processing must be deferred! Note that processing the *prequeue* can also lead to a *chunk* being consumed by the application.

```
         if (tp->rcv_nxt == tp->copied_seq &&
             !skb_queue_empty(&tp->ucopy.prequeue)) {
do_prequeue:
            tcp_prequeue_process(sk);

            if ((chunk = len - tp->ucopy.len) != 0) {
NET_ADD_STATS_USER(LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE, chunk);
               len -= chunk;
               copied += chunk;
            }
         }
      }
```

Draining the prequeue may have caused new stuff to appear on the *receive queue.* The *continue* causes a jump back to the top of the loop to test receive queue.

```
      if ((flags & MSG_PEEK) && peek_seq != tp->copied_seq) {
         if (net_ratelimit())
            printk(KERN_DEBUG "TCP(%s:%d):
                    Application bug, race in MSG_PEEK.\n",
                    current->comm, current->pid);
         peek_seq = tp->copied_seq;
      }
      continue;
```

Copying data from the *receive* queue to application space.

Because of the preceding *continue* arrival here is *only* via *goto* from processing the *receive queue.*

```
found_ok_skb:
   /* Ok so how much can we use? */
   used = skb->len - offset;
   if (len < used)
      used = len;

   /* Do we have urgent data here? */
   if (tp->urg_data) {
      u32 urg_offset = tp->urg_seq - *seq;
      if (urg_offset < used) {
         if (!urg_offset) {
            if (!sock_flag(sk, SOCK_URGINLINE)) {
               ++*seq;
               offset++;
               used--;
               if (!used)
                  goto skip_copy;
            }
         } else
            used = urg_offset;
      }
   }
```

```
    if (!(flags & MSG_TRUNC)) {

----- NETDMA code


        {
            err = skb_copy_datagram_iovec(skb, offset,
                  msg->msg_iov, used);
            if (err) {
               /* Exception. Bailout! */
               if (!copied)
                  copied = -EFAULT;
               break;
            }
        }
    }
```

Updating buffer pointers and counters

The value of *used* is the amount just copied from this *skb*. The *tcp_rcv_space_adjust()* function is a fairly opaque algorithm designed to dynamically decide how much buffer space is available

```
    *seq += used;
    copied += used;
    len -= used;

    tcp_rcv_space_adjust(sk);
```

Where to copy...

This routine doesn't need to keep track of where to put the data as it is being copied. That is done in *memcpy_to_iovec()* which dynamically adjusts elements of the iovec to indicate how much of each buffer has been used up.... See *udprecv.pdf*

```
        iov->iov_len -= copy;
        iov->iov_base += copy;
```

End of the main loop

```
skip_copy:
      if (tp->urg_data && after(tp->copied_seq,
           tp->urg_seq)) {
         tp->urg_data = 0;
         tcp_fast_path_check(sk, tp);
      }
```

If more data exists in this *sk_buff* then back to the top of the loop.

```
      if (used + offset < skb->len)
         continue;

      if (skb->h.th->fin)
         goto found_fin_ok;
      if (!(flags & MSG_PEEK)) {
         sk_eat_skb(sk, skb, copied_early);
         copied_early = 0;
      }
      continue;

   found_fin_ok:
      /* Process the FIN. */
      ++*seq;
      if (!(flags & MSG_PEEK)) {
         sk_eat_skb(sk, skb, copied_early);
         copied_early = 0;
      }
      break;
   } while (len > 0);
```

Exit from the *main loop*

Finally after falling out of the main loop, the *prequeue* is drained one more time. Here *tp->ucopy.len* is set to *len* if *copied > 0* and zero otherwise. Finally the *tp->ucopy* structure is reinitialized to indicate that there is no application process actively trying to consume data.

```
    if (user_recv) {
       if (!skb_queue_empty(&tp->ucopy.prequeue)) {
          int chunk;

          tp->ucopy.len = copied > 0 ? len : 0;

          tcp_prequeue_process(sk);

          if (copied > 0 && (chunk = len - tp->ucopy.len) != 0)
{
NET_ADD_STATS_USER(LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE, chunk);
             len -= chunk;
             copied += chunk;
          }
       }

       tp->ucopy.task = NULL;
       tp->ucopy.len = 0;
    }
```

Return from tcp_recvmsg()

On return the socket must be released which means that the backlog queue must be drained once more..

```
    /* According to UNIX98, msg_name/msg_namelen are ignored
     * on connected socket. I was just happy when found this 8) --ANK
     */

    /* Clean up data we have read: This will do ACK frames. */
    tcp_cleanup_rbuf(sk, copied);

    TCP_CHECK_TIMER(sk);
    release_sock(sk);
    return copied;

out:
    TCP_CHECK_TIMER(sk);
    release_sock(sk);
    return err;

recv_urg:
    err = tcp_recv_urg(sk, timeo, msg, len, flags, addr_len);
    goto out;
}
```

Processing the *prequeue*

The *tcp_prequeue_process()* function simply dequeues *sk_buffs* from the *prequeue* and passes them to *tcp_v4_do_rcv()*. This function is called with *sk->sk_lock.owner = 1*. The usage of *local_bh_disable/enable* is not clear.

```
static void tcp_prequeue_process(struct sock *sk)
{
   struct sk_buff *skb;
   struct tcp_sock *tp = tcp_sk(sk);

   NET_INC_STATS_USER(LINUX_MIB_TCPPREQUEUED);

   /* RX process wants to run with disabled BHs, though it is
    *  not necessary */
   local_bh_disable();
   while ((skb = __skb_dequeue(&tp->ucopy.prequeue)) != NULL)
      sk->sk_backlog_rcv(sk, skb);
   local_bh_enable();

   /* Clear memory counter. */
   tp->ucopy.memory = 0;
}
```

The *tcp_v4_do_rcv()* function

This function is called directly from *tcp_v4_rcv()* when the *tp->user.task* pointer is found to be NULL in *tcp_prequeue*. It is also serves as the *backlog_rcv* function and is called each time a packet is removed from the backlog queue, when the prequeue overflows or by *tcp_prequeue_process*. Since *tcp_recvmsg()* calls both *release_sock()* and *tcp_prequeue_process()* this function may be called in *both* top end and bottom end contexts!

```
/* The socket must have it's spinlock held when we get
 * here.
 *
 * We have a potential double-lock case here, so even when
 * doing backlog processing we use the BH locking scheme.
 * This is because we cannot sleep with the original
 * spinlock held.
 */
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
   if (sk->sk_state == TCP_ESTABLISHED) { /* Fast path */
      TCP_CHECK_TIMER(sk);
      if (tcp_rcv_established(sk, skb, skb->h.th, skb->len))
         goto reset;
      TCP_CHECK_TIMER(sk);
      return 0;
   }
```

The *tcp_rcv_established()* function

```
/*
 * TCP receive function for the ESTABLISHED state.
 *
 * It is split into a fast path and a slow path.
 * The fast path is disabled when:
 * - A zero window was announced from us - zero window
 *   probing is only handled properly in the slow path.
 * - Out of order segments arrived.
 * - Urgent data is expected.
 * - There is no buffer space left
 * - Unexpected TCP flags/window values/header lengths are
 *   received(detected by checking the TCP header against pred_flags)
 * - Data is sent in both directions. Fast path only supports pure senders
 *   or pure receivers (this means either the sequence
 *   number or the ack value must stay constant)
 * - Unexpected TCP option.
 *
 * When these conditions are not satisfied it drops into a standard
 * receive procedure patterned after RFC793 to handle all cases.
 * The first three cases are guaranteed by proper pred_flags setting,
 * the rest is checked inline. Fast processing is turned on
 * in tcp_data_queue when everything is OK.
 */
```

The *tcp_rcv_established* function

A variant of Van J's algorithm is implemented here. His original algorithm was intended only for the downcall path, and the part that involves actual delivery to user space is used only when this function is called by the recipient of the data.

```
int tcp_rcv_established(struct sock *sk,
                          struct sk_buff *skb,
         struct tcphdr *th, unsigned len)
{
   struct tcp_sock *tp = tcp_sk(sk);

   /*
    * Header prediction.
    * The code loosely follows the one in the famous
    * "30 instruction TCP receive" Van Jacobson mail.
    *
    * Van's trick is to deposit buffers into socket queue
    * on a device interrupt, to call tcp_recv function
    * on the receive process context and checksum and copy
    * the buffer to user space. smart...
    *
    * Our current scheme is not silly either but we take the
    * extra cost of the net_bh soft interrupt processing...
    * We do checksum and copy also from device to kernel.
    */

   tp->rx_opt.saw_tstamp = 0;

```

```
/* pred_flags is 0xS?10 << 16 + snd_wnd
 * if header_prediction is to be made
 * 'S' will always be tp->tcp_header_len >> 2
 * '?' will be 0 for the fast path, otherwise pred_flags
 *  is 0 to turn it off (when there are holes in receive
 *  space for instance)
 * PSH flag is ignored.
 */

if ((tcp_flag_word(th) & TCP_HP_BITS) == tp->pred_flags &&
   TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {
   int tcp_header_len = tp->tcp_header_len;

```

Van J's magic is now complicated by the possible presence of the timestamp header option which didn't exist at the time of his 30 instruction tcp receive.

```
   /* Timestamp header prediction: tcp_header_len
    * is automatically equal to th->doff*4 due to pred_fla
    * pred_flags match.
    */

/* Check timestamp */
   if (tcp_header_len == sizeof(struct tcphdr) +
             TCPOLEN_TSTAMP_ALIGNED) {
      __u32 *ptr = (__u32 *)(th + 1);

     /* No? Slow path! */
      if (*ptr != ntohl((TCPOPT_NOP << 24) |
                (TCPOPT_NOP << 16) |
           (TCPOPT_TIMESTAMP << 8) | TCPOLEN_TIMESTAMP))
         goto slow_path;

      tp->rx_opt.saw_tstamp = 1;
      ++ptr;
      tp->rx_opt.rcv_tsval = ntohl(*ptr);
      ++ptr;
      tp->rx_opt.rcv_tsecr = ntohl(*ptr);

```

PAWs stands for Protection Against Wrapped Sequence Numbers. If the time stamp just went down this could well be a delayed duplicate.

```
        /* If PAWS failed, check it more carefully in slow path */
            if ((s32)(tp->rx_opt.rcv_tsval -
                       tp->rx_opt.ts_recent) < 0)
               goto slow_path;


            /* DO NOT update ts_recent here, if checksum fails
             * and timestamp was corrupted part, it will result
             * in a hung connection since we will drop all
             * future packets due to the PAWS test.
             */
         }

```

Updating recent timestamp

The packet must be a standalone *ack* or window update if *len* <= *tcp_header_len*. *The value of rcv_wup* is the value of *rcv_nxt* on the most recent window update that was sent. If the receiving process is a bulk sender, it should receive only ACKs and window updates. The comment on the next page explains that header only packets are checksummed on entry. It seems to me that if this process is a bulk sender then *tp->rcv_nxt* and *tcp->rcv_wup* should never change!

```
         if (len <= tcp_header_len) {
            /* Bulk data transfer: sender */
            if (len == tcp_header_len) {
               /* Predicted packet is in window by definition.
                * seq == rcv_nxt and rcv_wup <= rcv_nxt.
                * Hence, check seq<=rcv_wup reduces to:
                */
               if (tcp_header_len ==
                   (sizeof(struct tcphdr) +
                   TCPOLEN_TSTAMP_ALIGNED) &&
                   tp->rcv_nxt == tp->rcv_wup)
                  tcp_store_ts_recent(tp);
```

Standalone *ack* processing

The packet must be a standalone *ack* or window update if *len* <= *tcp_header_len.* As with *cop* when an ACK is received it is necessary to:

- free any sk_buffs that were acked
- see if there is pending data that can now be sent
- free the *ack* packet itself

```

                 /* We know that such packets are checksummed
                  * on entry.
                  */
                 tcp_ack(sk, skb, 0);
                 __kfree_skb(skb);
                 tcp_data_snd_check(sk, tp);
                 return 0;
              } else { /* Header too small */
                 TCP_INC_STATS_BH(TCP_MIB_INERRS);
                 goto discard;
              }
```

Validating sequence numbers

The value of *rcv_wup* is the *last ack number* sent. Because of delayed acks in TCP *rcv_nxt* may be beyond *rcv_wup.* The check is made to allow control information with otherwise invalid sequence numbers to get through.

```
/* Check segment sequence number for validity.
 *
 * Segment controls are considered valid, if the segment
 * fits to the window after truncation to the window. Acceptability
 * of data (and SYN, FIN, of course) is checked separately.
 * See tcp_data_queue(), for example.
 *
 * Also, controls (RST is main one) are accepted using RCV.WUP instead
 * of RCV.NXT. Peer still did not advance his SND.UNA when we
 * delayed ACK, so that hisSND.UNA<=ourRCV.WUP.
 * (borrowed from freebsd)
 */

static inline int tcp_sequence(struct tcp_sock *tp,
        u32 seq, u32 end_seq)
{
   return   !before(end_seq, tp->rcv_wup) &&
      !after(seq, tp->rcv_nxt + tcp_receive_window(tp));
}
```

Data packet processing

Arrival here means we are still in the fast path and that the packet contained data.

The value of *eaten* will be set to 1 if and only if all of the data in this packet is transferred to user space. The value of *tp->copied_seq* is the sequence number of the next byte of data to be copied to user space. If tp->copied_seq == tp->rcv_nxt, then everything up to *rcv_nxt (which is the start of this packet since we are in the fast path)* has already been copied to user space and so its safe to copy this packet. The test len - tcp_header_len <= tp->ucopy.len is done to ensure that the *entire* segment will fit in the user buffer.

Finally the copy can't happen unless the *sk_lock.owner* is set to 1 and the *ucopy.task* is the current process. The value of *sk_lock.owner* is 1 during processing of the *backlog_queue and tcp_process_prequeue* and *current* will match *ucopy.task.* But when *tcp_v4_do_receive* is called in bottom half processing *current* would not match *ucopy.task* even if the socket is locked and *sk_lock.owner* is 1. *(Note: this situation can't actually happen -- if the socket is locked the packet goes on the backlog queue..*

The *tcp_copy_to_iovec()* function will update both *tp->copied_seq* and *tp->ucopy.len.* The copy can occur only if the entire packet will fit in the space left in the user buffer.

```
    } else {
       int eaten = 0;
       int copied_early = 0;

       if (tp->copied_seq == tp->rcv_nxt &&
           len - tcp_header_len <= tp->ucopy.len) {
 --- net DMA ---
           if (tp->ucopy.task == current &&
               sock_owned_by_user(sk) && !copied_early) {
              __set_current_state(TASK_RUNNING);

              if (!tcp_copy_to_iovec(sk, skb,
                                     tcp_header_len))
                 eaten = 1;
           }
```

Packet fully consumed

Presumably, *tcp_copy_to_iovec()* performs checksumming so it is now safe to update the timestamp. The timestamp is updated only if this segment is the first segment beyond the last ack sent because the timestamp is used to compute the RTT.

```
                if (eaten) {
                   /* Predicted packet is in window by definition.
                    * seq == rcv_nxt and rcv_wup <= rcv_nxt.
                    * Hence, check seq<=rcv_wup reduces to:
                    */
                   if (tcp_header_len ==
                       (sizeof(struct tcphdr) +
                        TCPOLEN_TSTAMP_ALIGNED) &&
                       tp->rcv_nxt == tp->rcv_wup)
                      tcp_store_ts_recent(tp);

                   tcp_rcv_rtt_measure_ts(sk, skb);

                   __skb_pull(skb, tcp_header_len);
                   tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
                   NET_INC_STATS_BH(LINUX_MIB_TCPHPHITSTOUSER);
                }
```

Copied early is set by NET_DMA processing. The *tcp_cleanup_rbuf()* function is responsible for determining if a window update must be sent.

```
                if (copied_early)
                   tcp_cleanup_rbuf(sk, skb->len);
             }
```

This code was inserted just before the *tcp_copy_to_iovec()* and produced the log messages shown

```
                if (in_interrupt())
                {
                   printk("ucopy in interrupt! \n");
                   tp->ucopy_in_irq += 1;
                }
```

Packet not eaten

If the packet is not fully consumed it must go on the receive queue. It is also checksummed and time stamp processing is performed in yet another place here.

```
        if (!eaten) {
            if (tcp_checksum_complete_user(sk, skb))
                goto csum_error;

            /* Predicted packet is in window by definition.
             * seq == rcv_nxt and rcv_wup <= rcv_nxt.
             * Hence, check seq<=rcv_wup reduces to:
             */
            if (tcp_header_len ==
                 (sizeof(struct tcphdr) +
                 TCPOLEN_TSTAMP_ALIGNED) &&
                 tp->rcv_nxt == tp->rcv_wup)
                tcp_store_ts_recent(tp);

            tcp_rcv_rtt_measure_ts(sk, skb);

```

The usage of *sk_forward_alloc* is not clear, but if the size of the buffer (header and data) exceeds it then a jump into slow path processing is required.

```
            if ((int)skb->truesize > sk->sk_forward_alloc)
                goto step5;

            NET_INC_STATS_BH(LINUX_MIB_TCPHPHITS);

```

Here is where the packet is placed on the receive queue and *rcv_next* is updated.

```
            /* Bulk data transfer: receiver */
            __skb_pull(skb,tcp_header_len);
            __skb_queue_tail(&sk->sk_receive_queue, skb);
            sk_stream_set_owner_r(skb, sk);
            tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
        }

```

Ack Generation

The call to *tcp_event_data_recv()* deals with delayed ACK generation. The calls to *tcp_ack()* and *tcp_data_snd_check()* deal with processing the ack carried by this packet and possibly sending data from the send queue. The jump to *no_ack* occurs if there is already pending ack that has been scheduled for transmission..

```
            tcp_event_data_recv(sk, tp, skb);

            if (TCP_SKB_CB(skb)->ack_seq != tp->snd_una) {
               /* Well, only one small jumplet in fast path... */
               tcp_ack(sk, skb, FLAG_DATA);
               tcp_data_snd_check(sk, tp);
               if (!inet_csk_ack_scheduled(sk))
                  goto no_ack;
            }
```

The call to *_tcp_ack_snd_check()* is used to determine if we need to send an ack now.

```
            __tcp_ack_snd_check(sk, 0);

no_ack:
#ifdef CONFIG_NET_DMA
            if (copied_early)
               __skb_queue_tail(&sk->sk_async_wait_queue, skb);
            else
#endif
```

If the complete packet was consumed, then the buffer is freed. Otherwise, the *sk_data_ready()* function is invoked to wake up any sleeping application. In *COP* this is done automatically in the call to *sock_queue_rcv_skb(sk, skb).* This concludes the *fast path.*

```
            if (eaten)
               __kfree_skb(skb);
            else
               sk->sk_data_ready(sk, 0);
            return 0;
         }
```

The slow path

This path begins with checksumming. It then determines if the packet should be discarded because PAWS failed. Resets are accepted even if PAWS fails.

```
slow_path:
   if (len < (th->doff<<2) ||
          tcp_checksum_complete_user(sk, skb))
      goto csum_error;

   /*
    * RFC1323: H1. Apply PAWS check first.
    */
   if (tcp_fast_parse_options(skb, th, tp) &&
          tp->rx_opt.saw_tstamp &&
       tcp_paws_discard(sk, skb)) {
      if (!th->rst) {
         NET_INC_STATS_BH(LINUX_MIB_PAWSESTABREJECTED);
         tcp_send_dupack(sk, skb);
         goto discard;
      }
      /* Resets are accepted even if PAWS failed.

         ts_recent update must be made after we are sure
         that the packet is in window.
       */
   }
```

Sequence number validation

If this segment doesn't contain *tp->rcv_nxt* its necessary to send a duplicate ack unless it does contain a *reset* flag. Reset always causes instant death of the connection.

```
    if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq,
                           TCP_SKB_CB(skb)->end_seq)) {
/* RFC793, page 37: "In all states except SYN-SENT, all reset
 * (RST) segments are validated by checking their SEQ-fields."
 * And page 69: "If an incoming segment is not acceptable,
 * an acknowledgment should be sent in reply (unless the RST
 * bit is set, if so drop the segment and return)".

       if (!th->rst)
          tcp_send_dupack(sk, skb);
       goto discard;
    }

    if(th->rst) {
       tcp_reset(sk);
       goto discard;
    }

    tcp_replace_ts_recent(tp, TCP_SKB_CB(skb)->seq);

```

The packet is rejected in the *end* is before *rcv_wup* or the start is after the end of the window.

```
static inline int tcp_sequence(struct tcp_sock *tp,
                                   u32 seq, u32 end_seq)
{
    return   !before(end_seq, tp->rcv_wup) &&
       !after(seq, tp->rcv_nxt + tcp_receive_window(tp));
}
```

Because of the way the receive window is computed.. *rcv_nxt* + *receive_window* = *rcv_wup* + *rcv_nxt*.

```
/* Compute the actual receive window we are currently advertising.
 * Rcv_nxt can be after the window if our peer push more data
 * than the offered window.
 */
static inline u32 tcp_receive_window(const struct tcp_sock *tp)
{
        s32 win = tp->rcv_wup + tp->rcv_wnd - tp->rcv_nxt;

        if (win < 0)
                win = 0;
        return (u32) win;
}
```

If the packet carries SYN and the sequence number is *before rcv_nxt* the packet is considered a delayed duplicate and nothing happens here.

If the sequence number is beyond *rcv_nxt,* then the connection is reset. The *before()* function does deal with sequence number wrap. Since this end of the connection believes it is established, why not just ACK and let the other end reset if need be??

```
if (th->syn && !before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt))
{
   TCP_INC_STATS_BH(TCP_MIB_INERRS);
   NET_INC_STATS_BH(LINUX_MIB_TCPABORTONSYN);
   tcp_reset(sk);
   return 1;
}
```

```
/*
 * The next routines deal with comparing 32 bit unsigned ints
 * and worry about wraparound (automatic with unsigned
   arithmetic).
 */

static inline int before(__u32 seq1, __u32 seq2)
{
       return (__s32)(seq1-seq2) < 0;
}
```

If the *ack* flag is set in the header then *tcp_ack()* is called to process it.

```
step5:
   if(th->ack)
      tcp_ack(sk, skb, FLAG_SLOWPATH);

   tcp_rcv_rtt_measure_ts(sk, skb);

   /* Process urgent data. */
   tcp_urg(sk, skb, th);

   /* step 7: process the segment text */
   tcp_data_queue(sk, skb);
```

Check to see if its possible to send a new segment because the one just received *acked* new data or if it is necessary to send an ack

```
   tcp_data_snd_check(sk, tp);
   tcp_ack_snd_check(sk);
   return 0;
```

```
csum_error:
   TCP_INC_STATS_BH(TCP_MIB_INERRS);

discard:
   __kfree_skb(skb);
   return 0;
}
```

Queuing the new data

The receive queue consists only of in order and non overlapping segments. If this segment doesn't fall into that category it must go on the out of order queue. If it fills a hole, then a collection of segments from the out of order queue can move to the receive queue.

```
static void tcp_data_queue(struct sock *sk,
                                  struct sk_buff *skb)
{
    struct tcphdr *th = skb->h.th;
    struct tcp_sock *tp = tcp_sk(sk);
    int eaten = -1;
```

Check for no data in the packet.

```
    if (TCP_SKB_CB(skb)->seq == TCP_SKB_CB(skb)->end_seq)
       goto drop;
```

Update *data* pointer to point to actual data and then do ECN and *sack* processing

```
    __skb_pull(skb, th->doff*4);
    TCP_ECN_accept_cwr(tp, skb);
    if (tp->rx_opt.dsack) {
       tp->rx_opt.dsack = 0;
       tp->rx_opt.eff_sacks =
                min_t(unsigned int, tp->rx_opt.num_sacks,
                       4 - tp->rx_opt.tstamp_ok);
    }
```

Direct copy to user space may occur again here

If the sequence number is *rcv_nxt* then the packet goes on the receive queue or may be copied to user space. The segment will not be queued if the offered window is presently 0. For copy to user space to succeed the same conditions must prevail as before. The lock must be held and the current task must be the ucopy task.

```
/*  Queue data for delivery to the user.
 *  Packets in sequence go to the receive queue.
 *  Out of sequence packets to the out_of_order_queue.
 */
if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {
   if (tcp_receive_window(tp) == 0)
      goto out_of_window;

   /* Ok. In sequence. In window. */
   if (tp->ucopy.task == current &&
       tp->copied_seq == tp->rcv_nxt && tp->ucopy.len &&
       sock_owned_by_user(sk) && !tp->urg_data) {
      int chunk = min_t(unsigned int, skb->len,
                  tp->ucopy.len);

```

Exercise: If `tp->ucopy.task == current,` how can it *not* be running?? Here the copy to user space doesn't necessarily have to copy the entire segment, but it might. Even if it does *eaten* can't be set if the segment also contains a *fin.*

```
      __set_current_state(TASK_RUNNING);

      local_bh_enable();
      if (!skb_copy_datagram_iovec(skb, 0,
            tp->ucopy.iov, chunk)) {
         tp->ucopy.len -= chunk;
         tp->copied_seq += chunk;
         eaten = (chunk == skb->len && !th->fin);
         tcp_rcv_space_adjust(sk);
      }
      local_bh_disable();
   }
```

Adding the buffer to the *receive queue.*

The value of "eaten" is set to 1 if *all* of the data in this segment was copied to user space. Otherwise *eaten* will be -1 (no data copied) or 0 some but not all data copied.

If *eaten* is <= 0 then the packet will be queued on the receive queue. Queue pruning details are not clear. In the Linux implementation of TCP buffer quota's appear to be somewhat *soft* and subject to change as a function of global memory demand.

```
        if (eaten <= 0) {
queue_and_out:
            if (eaten < 0 &&
                (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf
                || !sk_stream_rmem_schedule(sk, skb))) {
                if (tcp_prune_queue(sk) < 0 ||
                    !sk_stream_rmem_schedule(sk, skb))
                    goto drop;
            }
            sk_stream_set_owner_r(skb, sk);
            __skb_queue_tail(&sk->sk_receive_queue, skb);
        }
        tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
```

As noted earlier TCP uses a variable ACK delay strategy that is implemented in *tcp_event_data_recv().*

```
        if(skb->len)
            tcp_event_data_recv(sk, tp, skb);
        if(th->fin)
            tcp_fin(skb, sk, th);
```

Checking for filled holes

If the out of order queue is not empty this segment may have filled a hole that necessarily exists between the *end of the receive queue* and the *start of the out of order queue.* It may now be possible to move more segments from the out of order queue to the receive queue. The *tcp_ofo_queue()* function handles this. It is also possible for holes to be filled internal to the O-o-O queue, but these events are irrelevant.

```
        if (!skb_queue_empty(&tp->out_of_order_queue)) {
            tcp_ofo_queue(sk);

            /* RFC2581. 4.2. SHOULD send immediate ACK, when
             * gap in queue is filled.
             */
            if (skb_queue_empty(&tp->out_of_order_queue))
                inet_csk(sk)->icsk_ack.pingpong = 0;
        }

        if (tp->rx_opt.num_sacks)
            tcp_sack_remove(tp);

        tcp_fast_path_check(sk, tp);

        if (eaten > 0)
            __kfree_skb(skb);
        else if (!sock_flag(sk, SOCK_DEAD))
            sk->sk_data_ready(sk, 0);
        return;
    }
```

End of segment before tp->rcv_nxt

In this case there is no data in the segment that can be used. It's necessary to ack but the segment must be dropped.

```
   if (!after(TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt)) {
      /* A retransmit, 2nd most common case.  Force an
         immediate ack. */
      NET_INC_STATS_BH(LINUX_MIB_DELAYEDACKLOST);
      tcp_dsack_set(tp, TCP_SKB_CB(skb)->seq,
                    TCP_SKB_CB(skb)->end_seq);

out_of_window:
      tcp_enter_quickack_mode(sk);
      inet_csk_schedule_ack(sk);
drop:
      __kfree_skb(skb);
      return;
   }
```

Received segment outside window

```
/* Out of window. F.e. zero window probe. */
if (!before(TCP_SKB_CB(skb)->seq,
       tp->rcv_nxt + tcp_receive_window(tp)))
   goto out_of_window;

```

Received packet overlaps *rcv_nxt*

```
tcp_enter_quickack_mode(sk);

if (before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
   /* Partial packet, seq < rcv_next < end_seq */
  SOCK_DEBUG(sk, "partial packet:
            rcv_next %X seq %X - %X\n",
         tp->rcv_nxt, TCP_SKB_CB(skb)->seq,
         TCP_SKB_CB(skb)->end_seq);

   tcp_dsack_set(tp, TCP_SKB_CB(skb)->seq, tp->rcv_nxt);


/* If window is closed, drop tail of packet. But after
 * remembering D-SACK for its head made in previous line.
 */
   if (!tcp_receive_window(tp))
      goto out_of_window;
   goto queue_and_out;
}

TCP_ECN_check_ce(tp, skb);

```

The packet may also be dropped due to full buffer quota.

```
if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf ||
    !sk_stream_rmem_schedule(sk, skb)) {
   if (tcp_prune_queue(sk) < 0 ||
       !sk_stream_rmem_schedule(sk, skb))
      goto drop;
}


/* Disable header prediction. */
tp->pred_flags = 0;
inet_csk_schedule_ack(sk);

SOCK_DEBUG(sk, "out of order segment: rcv_next %X seq %X -
%X\n",  tp->rcv_nxt, TCP_SKB_CB(skb)->seq,
         TCP_SKB_CB(skb)->end_seq);

sk_stream_set_owner_r(skb, sk);
```

Adding to the out-of-order queue

Handling an out of order segment occurs here. If the out of order queue is presently empty then this packet is added and SACK data is constructed.

```
    if (!skb_peek(&tp->out_of_order_queue)) {
       /* Initial out of order segment, build 1 SACK. */
       if (tp->rx_opt.sack_ok) {
          tp->rx_opt.num_sacks = 1;
          tp->rx_opt.dsack     = 0;
          tp->rx_opt.eff_sacks = 1;
          tp->selective_acks[0].start_seq =
                          TCP_SKB_CB(skb)->seq;
          tp->selective_acks[0].end_seq =
                   TCP_SKB_CB(skb)->end_seq;
       }
       __skb_queue_head(&tp->out_of_order_queue,skb);
```

If the out of order queue is not empty then its necessary to figure out where to put this segment. The most likely place is on the *end of the queue.*

```
    } else {
       struct sk_buff *skb1 = tp->out_of_order_queue.prev;
       u32 seq = TCP_SKB_CB(skb)->seq;
       u32 end_seq = TCP_SKB_CB(skb)->end_seq;
```

The new *skb* will be inserted after *skb1* which is the current last element on the queue.

```
       if (seq == TCP_SKB_CB(skb1)->end_seq) {
          __skb_append(skb1, skb, &tp->out_of_order_queue);

          if (!tp->rx_opt.num_sacks ||
              tp->selective_acks[0].end_seq != seq)
             goto add_sack;

          /* Common case: data arrive in order after hole. */
          tp->selective_acks[0].end_seq = end_seq;
          return;
       }
```

Not last segement on out of order queue

If this is not the last segment then a backward search through the queue must be done to find where to put it.

```
        /* Find place to insert this segment. */
        do {
           if (!after(TCP_SKB_CB(skb1)->seq, seq))
              break;
        } while ((skb1 = skb1->prev) !=
            (struct sk_buff*)&tp->out_of_order_queue);
```

Its possible that overlap of entire segments may occur. This may result in dropping this segment if all data it carries is already on the O-o-O queue. What happens if all of the data in this segment is presently contained in TWO segments in the O-o-O queue???

```

        /* Do skb overlap to previous one? */
        if (skb1 != (struct sk_buff*)&tp->out_of_order_queue &&
            before(seq, TCP_SKB_CB(skb1)->end_seq)) {
           if (!after(end_seq, TCP_SKB_CB(skb1)->end_seq)) {
              /* All the bits are present. Drop. */
              __kfree_skb(skb);
              tcp_dsack_set(tp, seq, end_seq);
              goto add_sack;
           }

           if (after(seq, TCP_SKB_CB(skb1)->seq)) {
              /* Partial overlap. */
              tcp_dsack_set(tp, seq, TCP_SKB_CB(skb1)->end_seq);
           } else {
              skb1 = skb1->prev;
           }
        }
        __skb_insert(skb, skb1, skb1->next,
                          &tp->out_of_order_queue);

```

Or it is also possible that this segment covers multiple existing ones that may now be discarded.

```
      /* And clean segments covered by new one as whole. */
      while ((skb1 = skb->next) !=
             (struct sk_buff*)&tp->out_of_order_queue &&
             after(end_seq, TCP_SKB_CB(skb1)->seq)) {
             if (before(end_seq, TCP_SKB_CB(skb1)->end_seq)) {
                tcp_dsack_extend(tp,
                         TCP_SKB_CB(skb1)->seq, end_seq);
                break;
             }
             __skb_unlink(skb1, &tp->out_of_order_queue);
             tcp_dsack_extend(tp, TCP_SKB_CB(skb1)->seq,
                          TCP_SKB_CB(skb1)->end_seq);
             __kfree_skb(skb1);
      }

add_sack:
      if (tp->rx_opt.sack_ok)
         tcp_sack_new_ofo_skb(sk, seq, end_seq);
   }
}

```