IP Layer Transmission Functions

The ip_build_xmit() function

This function is called by udp_sendmsg() to construct the IP header and initiate the transmission. This function is not exported by the kernel so you will have to implement the necessary components within your project. Parameters passed to the ip_build_xmit() function are:

- The address of the struct sock,
- The protocol specific callback routine used to retrieve the user data to be transmitted,
- a caller supplied pointer that will be passed back to the callback,
- the length of user data plus transport header,
- the ip cookie,
- a pointer to the route cache element to be used and
- the flags from the msg structure.

When called by UDP the frag pointer is set to point to the UDP fake header structure. The packet must have been routed prior to calling this function.

```c
624 /*
625 *      Fast path for unfragmented packets.
626 */
627 int ip_build_xmit(struct sock *sk,
            int getfrag(const void*, char*, unsigned int, unsigned int),
            const void *frag, unsigned length, struct ipcm_cookie *ipc,
            struct rtable *rt, int flags)
637 {
638     int err;
639     struct sk_buff *skb;
640     int df;
641     struct iphdr *iph;
```
Fast and slow path transmission

The fast path handles all packets in which the caller provides the IP header (SOCK_RAW) and those other packets which require neither header options nor fragmentation. (The value sk->protinfo.af_inet.hdrincl is set to 1 during initialization of a socket of type SOCK_RAW.) If the IP packet must be fragmented or ip options are being used, ip_build_xmit_slow() must be called. If socket is not of type SOCK_RAW, the length is incremented by the size of the standard 20 byte header here.

644 /*  Try the simple case first. This leaves fragmented
645   * frames, and by choice RAW frames within 20 bytes of
646   * maximum size(rare) to the long path
648   */
649 if (!sk->protinfo.af_inet.hdrincl) {
    length += sizeof(struct iphdr);
}

The route cache element contains a guesstimate of the path MTU. The IP cookie contains a pointer to header options if they are present.

651 /*  Check for slow path. */
654 if (length > rt->u.dst.pmtu || ipc->opt != NULL)
655   return ip_build_xmit_slow(sk, getfrag, frag,
656                               length, ipc, rt, flags);

The else case covers sockets of type SOCK_RAW. If packets associated with raw sockets must be fragmented, they must be fragmented in user space. Raw packets of size larger than the device MTU are rejected here.

656 } else {
657   if (length > rt->u.dst.dev->mtu) {
658     ip_local_error(sk, EMSGSIZE, rt->rt_dst,
659                    sk->dport, rt->u.dst.dev->mtu);
660     return -EMSGSIZE;
661   }
662 }
Fast path transmission

Arrival at this point in the code indicates that “fast” transmit path is appropriate. The function of the MSG_PROBE bit is said to be for generating probe packets to determine path MTU, but a search of the source shows no place where the bit gets set and the MSG_PROBE flag is not defined in the man pages as user specifiable either.

```c
if (flags & MSG_PROBE) goto out;
```

The function `ip_dont_fragment()` defined in `include/net/ip.h` returns true if path MTU discovery option is set for the socket.

```c
/*
 * Do path mtu discovery if needed.
 */
int df = 0;
if (ip_dont_fragment(sk, &rt->u.dst))
    df = htons(IP_DF);
```

Note that the following block is not dependent upon the if statement above. The value of `hh_len` is set to the nearest multiple of 16 larger than the actual value which is stored in the `net_device` structure associated with the outgoing interface. Then the `sk_buff` is allocated with 15 more bytes than that. Note that this logic requires that the packet be routed before the `sk_buff` is allocated, and that this code fragment should be used in your send module. Any waiting caused by exceeding buffer quota is handled internally in `sock_alloc_send_skb()`.

```c
/*
 * Fast path for unfragmented frames without options.
 */
{ 
    int hh_len = (rt->u.dst.dev->hard_header_len + 15)&~15;
    skb = sock_alloc_send_skb(sk, length+hh_len+15,
        flags & MSG_DONTWAIT, &err);

    if(skb==NULL) goto error;
    skb_reserve(skb, hh_len);
}
```

With the `sk_buff` allocated, `ip_build_xmit` continues. The `skb_reserve()` function reserves space at the start of the buffer for the hardware (MAC) header by unconditionally setting the data and tail pointers to the offset specified in `hh_len`. You should do this too.

```c
if(skb==NULL) goto error;
skb_reserve(skb, hh_len);
```
Attaching the route cache element to the sk_buff.

The skb inherits its priority field from the struct sock. The use of the priority field is not well understood. The call to dst_clone returns increments the __refcnt field of the struct route that is being used and simply returns the pointer it was passed after incrementing the __refcnt field. Your cop_alloc_skb() should do this too.

685 skb->priority = sk->priority;
686 skb->dst = dst_clone(&rt->u.dst);

IP header construction

The skb_put() function advances both the tail pointer and the len field by the amount specified. It then returns the original value of the tail pointer. The value of length at this point is the sum of the lengths of the IP header, UDP header, and the user data. A useful exercise is to illustrate with a diagram the impact of the skb_xxx family of functions.

688 skb->nh.iph = iph = (struct iphdr *)skb_put(skb, length);

If the header is not included in the user data, ip_build_xmit() builds it. You will to incorporate an adapted version of this block directly into your cop_make_iphdr() function. Yours should memcpy() the skeleton from the cop_sock, set the length, the ident, and then do the checksum.

690 if(!sk->protinfo.af_inet.hdrincl) {
691   iph->version=4;
692   iph->ihl=5;
693   iph->tos=sk->protinfo.af_inet.tos;
694   iph->tot_len = htons(length);
695   iph->frag_off = df;
696   iph->ttl=sk->protinfo.af_inet.mc_ttl;
697   ip_select_ident(iph, &rt->u.dst, sk);
698   if (rt->rt_type != RTN_MULTICAST)
699     iph->ttl=sk->protinfo.af_inet.ttl;
700   iph->protocol=sk->protocol;
701   iph->saddr=rt->rt_src;
702   iph->daddr=rt->rt_dst;
703   iph->check=0;
704   iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl);
Invoking the transport layer callback

Here the call-back to the caller supplied `getfrag()` routine (which is in this case `udp_getfrag` or `udp_getfrag_nosum`). The parameter, `frag`, was supplied by the caller of `ip_build_xmit()` and in this case points to the UDP fake header constructed by the UDP layer. The second parameter is pointer to the buffer location just past the end of the IP header and the fourth, third is the amount of free space left in the `sk_buff`. The third parameter is an offset which is always set to 0 for unfragmented packets consisting of a single `iov` element. However, when packets are fragmented it contains the fragment offset.

```c
705   err = getfrag(frag, ((char *)iph)+iph->ihl*4, 0, 
706                       length-iph->ihl*4);
```

If the header was included the `getfrag()` routine must supply the whole packet.

```c
707   else
708   err = getfrag(frag, (void *)iph, 0, length);
710   if (err)
711      goto error_fault;
```
Passing the packet through the netfilter

The IP packet is passed to the filter and device layer using NF_HOOK. If the the netfilter facility accepts the packet, it will be passed to the function output_maybe_reroute().

```
713       err = NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb,
714            NULL, rt->u.dst.dev, output_maybe_reroute);
715       if (err > 0) {
716           err = sk->protinfo.af_inet.recverr ?
717               net_xmit_errno(err) : 0;
718       } if (err) goto error;
719     out:
720     return 0;
722   error_fault:
723     err = -EFAULT;
724   kfree_skb(skb);
725 error:
726   IP_INC_STATS(IpOutDiscards);
727   return err;
728 }
```

The ip_dont_fragment() function defined in include/net/ip.h returns true if path MTU discovery option is set for the socket.

```
181 static inline
182 int ip_dont_fragment(struct sock *sk, struct dst_entry *dst) {
184       return (sk->protinfo.af_inet.pmtudisc ==
185           IP_PMTUDISC_DO || (sk->protinfo.af_inet.pmtudisc ==
186           IP_PMTUDISC_WANT && !(dst->mxlock&(1<<RTAX_MTU))));
187 }
```
### An `output_maybe_reroute` function

```c
316 /**/
317 int cop_output_maybe_reroute(
318 struct sk_buff *skb)
319 {
320    int err;
321    struct timeval tod;
322    cop_sock_t *cpo = 0;
323    cop_hdr_t *cph;
324    
325    if (skb->sk)
326    {
327        cpo = skb->sk->user_data;
328        if (cpo)
329            {  
330            cpo->txtotal += 1;
331        }
332    }
333    
336    if (skb->dst->output)
337    {
338        err = skb->dst->output(skb);
339        dprintk("cop_output_: Output returned %d \n", err);
340        return(err);
341    }
342    else
343    {
344        dprintk("cop_output_: No output function! \n");
345        return(-1);
346    }
347    }
```

The pointer `skb->dst->output` was set in routing to point to either `ip_output()` function or the `ip_local_deliver()` function.
Transmission of fragmented IP packets and those with header options

/*
 *   Build and send a packet, with as little as one copy
 * Doesn't care much about ip options... option length
 * can be different for fragment at 0 and other fragments.
 * Note that the fragment at the highest offset is sent
 * first, so the getfrag routine can fill in the TCP/UDP
 * checksum header field in the last fragment it sends...
 * actually it also helps the reassemblers, they can put
 * most packets in at the head of the fragment queue,
 * and they know the total size in advance. This last
 * feature will measurably improve the Linux fragment
 * handler one day.
 * The callback has five args, an arbitrary pointer (copy
 * of frag), the source IP address (may depend on the
 * routing table), the destination address (char *), the
 * offset to copy from, and the length to be copied.
 */

static int ip_build_xmit_slow(struct sock *sk,
   int getfrag(const void *,char *,unsigned int,unsigned int),
   const void *frag,
   unsigned length,
   struct ipcm_cookie *ipc,
   struct rtable *rt,
   int flags)
{
    unsigned int fraglen, maxfraglen, fragheaderlen;
    int err;
    int offset, mf;
    int mtu;
    u16 id;
    int hh_len = (rt->u.dst.dev->hard_header_len + 15)&~15;
    int nfrags=0;
    struct ip_options *opt = ipc->opt;
    int df = 0;

    mtu = rt->u.dst.pmtu;
    if (ip_dont_fragment(sk, &rt->u.dst))
      df = htons(IP_DF);
Computing the length of fragments to be sent

Recall that `ip_build_xmit()` incremented `length` by the size of a standard IP header. Here it is decremented to recover the length of user data and transport header. Then the length of each fragment's IP header is saved in `fragheaderlen` and the maximum size of the remainder of each datagram is saved in `maxfraglen`.

```
454    length -= sizeof(struct iphdr);
456    if (opt) {
457        fragheaderlen = sizeof(struct iphdr) + opt->optlen;
458        maxfraglen = ((mtu-sizeof(struct iphdr)-
                         opt->optlen) & ~7) + fragheaderlen;
459    } else {
460        fragheaderlen = sizeof(struct iphdr);
462    /*
463       Fragheaderlen is the size of 'overhead' on each
464       buffer. Now work out the size of the frames to send.
465    */
467        maxfraglen = ((mtu-sizeof(struct iphdr)) & ~7)
                         + fragheaderlen;
468    }
```

Check to ensure packet size is within the 64k limit.

```
470    if (length + fragheaderlen > 0xFFFF) {
471        ip_local_error(sk, EMSGSIZE, rt->rt_dst,
                         sk->dport, mtu);
472        return -EMSGSIZE;
473    }
```
IP packets are always constructed last fragment to first. Here the offset and size of last fragment are computed.

475 /*
476 * Start at the end of the frame by handling the remainder.
477 */
479 offset = length - (length % (maxfraglen -
480 fragheaderlen));
481 /* Amount of memory to allocate for final fragment. */
485 fraglen = length - offset + fragheaderlen;
487 if (length-offset==0) {
488 fraglen = maxfraglen;
489 offset -= maxfraglen-fragheaderlen;
490 }
492 /* The last fragment will not have MF (more fragments) set.
494 */
496 mf = 0;
498 /*
499 * Don't fragment packets for path mtu discovery.
500 */
502 if (offset > 0 &&
503 sk->protinfo.af_inet.pmtudisc==IP_PMTUDISC_DO) {
504 ip_local_error(sk, EMSGSIZE, rt->rt_dst,
505 sk->dport, mtu);
506 return -EMSGSIZE;
507 }
509 if (flags & MSG_PROBE)
510 goto out;
511 /*
512 */
The value of \textit{af\_inet\_id} was initialised to 0 during socket initialization and is incremented for each fragment sent on this \textit{struct sock}. However, this particular value of \textit{id} may or may not actually end up in the packet due to a complex combination of circumstances. One iteration of the lengthy \textit{do} block below which ends at line 610 is performed for each fragment.

\begin{verbatim}
513     id = sk->protinfo.af_inet.id++;  
515     do {  
516         char *data;  
517         struct sk_buff * skb;  

An \textit{sk\_buff} header and data of size (fraglen+hh\_len+15) are allocated.

\begin{verbatim}
519     /* Get the memory we require with some space left for  
520      * alignment.  
523     skb = sock_alloc_send_skb(sk, fraglen+hh_len+15,  
524                         flags&MSG_DONTWAIT, &err);  

524     if (skb == NULL)  
525         goto error;  
528     skb->priority = sk->priority;  
529     skb->dst = dst_clone(&rt->u.dst);  

skb\_reserve reserves space for the hardware header at the front/head.

533     skb_reserve(skb, hh_len);  
\end{verbatim}
\end{verbatim}
The `skb_put` function adds the `fraglen` to the tail index to reserve space for the IP fragment and returns the original value of the tail index.

```c
535    /*
536    *      Find where to start putting bytes.
537    */
539    data = skb_put(skb, fraglen);
540    skb->nh.iph = (struct iphdr *)data;
```

This comment is clearly misleading..

```c
542    /*
543    *      Only write IP header onto non-raw packets
544    */
546    {
      struct iphdr *iph = (struct iphdr *)data;
549      iph->version = 4;
550      iph->ihl = 5;

551      if (opt) {
552        iph->ihl += opt->optlen>>2;
553        ip_options_build(skb, opt, ipc->addr, rt, offset);
555      }
556      iph->tos = sk->protinfo.af_inet.tos;
557      iph->tot_len =
558          htons(fraglen - fragheaderlen + iph->ihl*4);
559      iph->frag_off = htons(offset>>3)|mf|df;
```

This assignment will be overridden if this is the last fragment of a multifragment packet.

```c
560      iph->id = id;
```
Assigning an identifier to a fragmented packet

The outer *if* condition will be true only for the last or first and only fragment of a packet. The inner *if* will be true if the fragment is not the first fragment or fragmentation is allowed. Thus __ip_select_ident() will be called

- for the last fragment of a multi-fragment packet and also
- for the first and only fragment of a packet that carries header options but does not carry the *df* flag.

The *id* field of the first and only fragment of a packet that carries header options *and* the *df* flag appears to come from protinfo.af_inet.id.

```c
if (!mf) {
    if (offset || !df) {
        /* Select an unpredictable ident only
         * for packets without DF or having
         * been fragmented.
         */
        __ip_select_ident(iph, &rt->u.dst);
        id = iph->id;
    }
    /* Any further fragments will have MF set. */
    mf = htons(IP_MF);
}
if (rt->rt_type == RTN_MULTICAST)
    iph->ttl = sk->protinfo.af_inet.mc_ttl;
else
    iph->ttl = sk->protinfo.af_inet.ttl;
iph->protocol = sk->protocol;
iph->check = 0;
iph->saddr = rt->rt_src;
iph->daddr = rt->rt_dst;
}
```

IP header checksum is computed by *ip_fast_csum*.

```c
iph->check =
    ip_fast_csum((unsigned char *)iph, iph->ihl);
data += iph->ihl*4;
```
The UDP fragment and checksum routine is passed the fragment address and size.

587 /*
588  *      User data callback
589 */
590 if (getfrag(frag, data, offset, fraglen-
fragheaderlen)) {
592     err = -EFAULT;
593     kfree_skb(skb);
594     goto error;
595 }

Compute offset and fraglen for remaining fragments.

597     offset -= (maxfraglen-fragheaderlen);
598     fraglen = maxfraglen;
600     nfrags++;

The IP fragment or the IP packet with IP options is pushed to the device/filter layer.

602     err = NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL,
603                      skb->dst->dev, output_maybe_reroute);
604     if (err) {
605         if (err > 0)
606             err = sk->protinfo.af_inet.recverr ?
607                 net_xmit_errno(err) : 0;
608         if (err)
609             goto error;
610     } while (offset >= 0);
612     if (nfrags>1)
613         ip_statistics[smp_processor_id()*2 +
614                          !in_softirq()].IpFragCreates += nfrags;
615     out:
616     return 0;
617 error:
618     IP_INC_STATS(IpOutDiscards);
619     if (nfrags>1)
620         ip_statistics[smp_processor_id()*2 +
621                          !in_softirq()].IpFragCreates += nfrags;
621     return err;
622 }

14
Selecting an identifier number for an IP packet

A packet identifier is used in the reassembly of IP packets. If multiple sources on a single host are sending fragmented packets to a common destination it is critical that the id numbers come from a global counter and not per connection counters. The peer structure kept in the AVL tree play a key role in this.

The ip_select_ident() function is called unconditionally from the fast path of ip_build_xmit(), but ip_build_xmit_slow() sometimes uses the value in the protinfo.af_inet.id field of the struct sock.

```c
191 static inline void ip_select_ident(struct iphdr *iph, struct dst_entry *dst, struct sock *sk) {
192     if (iph->frag_off & __constant_htons(IP_DF)) {
193         /* This is only to work around buggy Windows95/2000
194          * VJ compression implementations. If the ID field
195          * does not change, they drop every other packet in
196          * a TCP stream using header compression.
197          */
198         iph->id = ((sk && sk->daddr) ?
199                     htons(sk->protinfo.af_inet.id++) : 0);
200     } else
201         __ip_select_ident(iph, dst);
202 }
```
Obtaining the id through the peer structure

The action of __ip_select_ident() is dependent upon whether the packet has an associated route. Normal UDP packets always will at this point.

720 void __ip_select_ident(struct iphdr *iph,  
    struct dst_entry *dst)  
721 {  
722     struct rtable *rt = (struct rtable *) dst;  
723  
724    if (rt) {  
725       if (rt->peer == NULL)  
726          rt_bind_peer(rt, 1);  
727  
728        /* If peer is attached to dest, it is never detached,  
729         so that we need not to grab a lock to dereference it.  
730        */  
731        if (rt->peer) {  
732          iph->id = htons(inet_getid(rt->peer));  
733          return;  
734        }  
735    } else  
736       printk(KERN_DEBUG "rt_bind_peer(0) @$p\n",  
737           NET_CALLER(iph));  
738  
739 }

Reaching this point means that rt_bind_peer() didn't succeed.

737  
738    ip_select_fb_ident(iph);  
739 }

The inet_getid() function assigns numbers serially from the peer structure.

56 static inline __u16 inet_getid(struct inet_peer *p)  
57 {  
58    __u16 id;  
59  
60    spin_lock_bh(&inet_peer_idlock);  
61    id = p->ip_id_count++;  
62    spin_unlock_bh(&inet_peer_idlock);  
63    return id;  
64 }

16
When no peer can be bound

As stated in a comment in the code: “Peer allocation may fail only in serious out-of-memory conditions. However we still can generate some output. Random ID selection looks a bit dangerous because we have no chances to select ID being unique in a reasonable period of time. But broken packet identifier may be better than no packet at all.”

```c
707 static void ip_select_fb_ident(struct iphdr *iph) {
708     static spinlock_t ip_fb_id_lock = SPIN_LOCK_UNLOCKED;
709     static u32 ip_fallback_id;
710     u32 salt;
711     spin_lock_bh(&ip_fb_id_lock);
712     salt = secure_ip_id(ip_fallback_id ^ iph->daddr);
713     iph->id = htons(salt & 0xFFFF);
714     ip_fallback_id = salt;
715     spin_unlock_bh(&ip_fb_id_lock);
716 }
717
718 
719 ```
Hardware assisted random number selection.

This function relies on random number generating hardware on Intel motherboards to pick a random id number. In addition to being called when a peer structure can't be created, it is also called by inet_getpeer() to initialize the starting id number when a new peer structure is created.

```c
__u32 secure_ip_id(__u32 daddr) {
    static time_t rekey_time;
    static __u32 secret[12];
    time_t t;

    /* Pick a random secret every REKEY_INTERVAL seconds. */
    t = CURRENT_TIME;
    if (!rekey_time || (t - rekey_time) > REKEY_INTERVAL) {
        rekey_time = t;
        /* First word is overwritten below. */
        get_random_bytes(secret+1, sizeof(secret)-4);
    }

    /* Pick a unique starting offset for each IP dest. */
    secret[0]=daddr;
    return halfMD4Transform(secret+8, secret);
}
```
void rt_bind_peer(struct rtable *rt, int create) {
    static spinlock_t rt_peer_lock = SPIN_LOCK_UNLOCKED;
    struct inet_peer *peer;
    peer = inet_getpeer(rt->rt_dst, create);
    spin_lock_bh(&rt_peer_lock);
    if (rt->peer == NULL) {
        rt->peer = peer;
        peer = NULL;
    }
    spin_unlock_bh(&rt_peer_lock);
    if (peer)
        inet_putpeer(peer);
}