

Delivery of an IP packet to the transport layer

The `ip_local_deliver_finish()` function is defined in `net/ipv4/ip_input.c`. It is the *okfn* that is invoked indirectly by `ip_local_deliver()` when an unfragmented packet has been received or when reassembly completes.

```
275     return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev,
276                   NULL, ip_local_deliver_finish);
277 }

199 static inline int ip_local_deliver_finish(
    struct sk_buff *skb)
200 {
201     int ihl = skb->nh.iph->ihl*4;
```

Removal of the network layer header

Yet another instance of the *pskb* family of functions used to be called to ensure that the IP header is entirely resident in the `kmalloc'd` area of the `sk_buff`. Apparently now that is guaranteed to be the case so, the `skb_pull()` function is used to advance `skb->data` so that it now *points to the transport header and to reduce `skb->len` by the length of the IP header.*

```
203     __skb_pull(skb, ihl);
204
205     /* Point into the IP datagram, just past the header. */
206     skb->h.raw = skb->data;
```

Delivery to raw sockets

Recall that the *struct sock*'s associated with sockets of type SOCK_RAW are linked in the *raw_v4_htable[]* which is a statically allocated hash table defined in *net/ipv4/raw.c*. The hash key is derived from the transport protocol number.

```
68 struct sock *raw_v4_htable[RAWV4_HTABLE_SIZE];
```

At this point *skb->data* points to the transport header. Here the transport protocol number is extracted from the IP header, converted to a hash key. If the hash chain is empty, there is by definition no raw handler that needs to see this packet.

```
208     rcu_read_lock();
209     {
210         /* Note: See raw.c and net/raw.h,
                RAWV4_HTABLE_SIZE==MAX_INET_PROTOS */
211         int protocol = skb->nh.iph->protocol;
212         int hash;
213         struct sock *raw_sk;
214         struct net_protocol *ipprot;
215
216         resubmit:
217         hash = protocol & (MAX_INET_PROTOS - 1);
218         raw_sk = sk_head(&raw_v4_htable[hash]);
219
220         /* If there is a raw socket we must check - if not we
221          * don't care less
222          */
```

If *raw_v4_input* did *not* deliver anything then *raw_sk* is reset to NULL.

```
223         if (raw_sk && !raw_v4_input(skb, skb->nh.iph, hash))
224             raw_sk = NULL;
```

Delivery to "real" transports

Continuing in `ip_local_deliver_finish()` the "real" transport protocols are handled. Recall that various transport protocols (along with their receive handlers) were registered in the hash table, `inet_protos`, by the `inet_init` function.

Here `ipprot` is set to point to a hash bucket in `inet_protos` corresponding to hash value obtained earlier. It used to be the case that a packet could be delivered to multiple rcv handlers for the *same transport protocol* and that fast/slow path delivery were differentiated. A fast path was taken to its corresponding handler (`udp_rcv` for UDP and `tcp_v4_rcv` for TCP) when the following three conditions are met:

- No raw socket was matched to the received packet.
- The hash bucket has only one entry (`ipprot->next == NULL`)
- That entry's protocol field matches that of the packet.

Now the packet is delivered only to the first eligible handler on the hash queue. Since there is only one possible recipient now, the problem of knowing whether or not to clone goes away.

```
226         if ((ipprot = rcu_dereference(inet_protos[hash]))
227             != NULL) {
228             int ret;
229             if (!ipprot->no_policy) {
230                 if (!xfrm4_policy_check(NULL,
231                     XFRM_POLICY_IN, skb)) {
232                     kfree_skb(skb);
233                     goto out;
234                 }
235                 nf_reset(skb);
236                 ret = ipprot->handler(skb);
237                 if (ret < 0) {
238                     protocol = -ret;
239                     goto resubmit;
240                 }
241                 IP_INC_STATS_BH(IPSTATS_MIB_INDELIVERS);
```

Unmatched packets

If no handlers were found the packet is dropped. If the packet was delivered to a raw socket then that counts as delivered. Either way the *sk_buff* must be freed because raw delivery unconditionally clones.

```
242     } else {
243         if (!raw_sk) {
244             if (xfrm4_policy_check(NULL,
245                                     XFRM_POLICY_IN, skb)) {
246                 IP_INC_STATS_BH(IPSTATS_MIB_INUNKNOWNPROTOS);
247                 icmp_send(skb, ICMP_DEST_UNREACH,
248                             ICMP_PROT_UNREACH, 0);
249             }
250         } else
251             IP_INC_STATS_BH(IPSTATS_MIB_INDELIVERS);
252         kfree_skb(skb);
253     }
254 out:
255     rcu_read_unlock();
256
257     return 0;
258 }
259
```

Delivery to raw sockets

If hash chain is non-empty, the `raw_v4_input()`, defined in `net/ipv4/raw.c` handles the delivery.

```
/* IP input processing comes here for RAW socket delivery.
This is fun as to avoid copies we want to make no surplus
copies. RFC 1122: SHOULD pass TOS value up to the transport
layer. -> It does. And not only TOS, but all IP header.*/

153 int raw_v4_input(struct sk_buff *skb,
                   struct iphdr *iph, int hash)
154 {
155     struct sock *sk;
156     struct hlist_head *head;
157     int delivered = 0;
```

As usual, after locking the queue, another test is made to ensure that the `struct sock` is still there.

```
159     read_lock(&raw_v4_lock);
160     head = &raw_v4_htable[hash];
161     if (hlist_empty(head))
162         goto out;
```

The `__raw_v4_lookup()` function returns the next socket in the chain based at its first parameter `sk` that matches the given protocol number, source address, destination address and device index.

```
163     sk = __raw_v4_lookup(__sk_head(head), iph->protocol,
164                          iph->saddr, iph->daddr,
165                          skb->dev->ifindex);
```

If a matching *struct sock* was found, then there is a destination to which the packet should be distributed. Unlike the loop in the device layer, in which cloning and delivery to the *previously* identified socket takes place, here the *sk_buff* is unconditionally cloned. The "original" will be delivered to a single "regular" transport later, or it will be freed.

```
167     while (sk) {
168         delivered = 1;
169         if (iph->protocol != IPPROTO_ICMP ||
170             !icmp_filter(sk, skb)) {
171             struct sk_buff *clone = skb_clone(skb, GFP_ATOMIC);
172             /* Not releasing hash table! */
173             if (clone)
174                 raw_rcv(sk, clone);
175         }
176         sk = __raw_v4_lookup(sk_next(sk), iph->protocol,
177                             iph->saddr, iph->daddr,
178                             skb->dev->ifindex);
179     }
```

The value returned is either last matched raw socket or NULL(when no raw socket matched). Delivery to the last matched raw socket takes place later.

```
180 out:
181     read_unlock(&raw_v4_lock);
182     return delivered;
183 }
```

Matching sockets to packets

The `__raw_v4_lookup()` function searches a chain of *struct sock*'s and returns the address of the first one which matches with respect to the parameters. On no match, NULL will be returned when the end of the chain is reached. The parameter *num* is the protocol number. The parameters *raddr* and *laddr* refer to *remote* and *local* address. The parameter *dif* is the index of the *net_device* on which the packet arrived.

```
105 struct sock *__raw_v4_lookup(struct sock *sk,
                               unsigned short num,
106                             __be32 raddr, __be32 laddr,
107                             int dif)
108 {
109     struct hlist_node *node;
110
```

Recall that for sockets of type SOCK_RAW, the value of *sk->num* is the protocol number and not the port number. In addition to a protocol number match, it is necessary to handle the cases in which the socket may be:

- connected to a remote address (*s->daddr*)
- bound to a local address (*s->rcv_saddr*)
- bound to a specific local interface (*s->bound_dev_if*)

```
111     sk_for_each_from(sk, node) {
112         struct inet_sock *inet = inet_sk(sk);
113
114         if (inet->num == num &&
115             !(inet->daddr && inet->daddr != raddr) &&
116             !(inet->rcv_saddr && inet->rcv_saddr != laddr) &&
117             !(sk->sk_bound_dev_if &&
118               sk->sk_bound_dev_if != dif))
119             goto found; /* gotcha */
120     }
121     sk = NULL;
122 found:
123     return sk;
124 }
```

Filtering of ICMP packets

For a packet with protocol as IPPROTO_ICMP, a call to function *icmp_filter* is made to decide whether to deliver packet to given raw socket or not. This decision is based on the field *sk->tp_pinfo.tp_raw4.filter.data* ...

```
    /*
        0 - deliver
        1 - block
    */

129 static __inline__ int icmp_filter(struct sock *sk, struct
                                   sk_buff *skb)
130 {
131     int type;
132
133     if (!pskb_may_pull(skb, sizeof(struct icmphdr)))
134         return 1;
135
136     type = skb->h.icmph->type;
137     if (type < 32) {
138         __u32 data = raw_sk(sk)->filter.data;
139
140         return ((1 << type) & data) != 0;
141     }
142
143     /* Do not block unknown ICMP types */
144     return 0;
145 }
```

Completing raw delivery

The `raw_rcv()` function is defined in `net/ipv4/raw.c`. For a raw socket, the "`data`" pointer is moved backward so that it points to IP header, using the function `skb_push`. A call to `raw_rcv_skb` is made.

```
252 int raw_rcv(struct sock *sk, struct sk_buff *skb)
253 {
254     if (!xfrm4_policy_check(sk, XFRM_POLICY_IN, skb)) {
255         kfree_skb(skb);
256         return NET_RX_DROP;
257     }
258     nf_reset(skb);
259
260     skb_push(skb, skb->data - skb->nh.raw);
261
262     raw_rcv_skb(sk, skb);
263     return 0;
264 }
265
```

The `raw_rcv_skb()` function invokes the function `sock_queue_rcv_skb()` to enqueue the packet in the receive queue of given socket.

```
239 static int raw_rcv_skb(struct sock * sk,
                        struct sk_buff * skb)
240 {
241     /* Charge it to the socket. */
242
243     if (sock_queue_rcv_skb(sk, skb) < 0) {
244         /* FIXME: increment a raw drops counter here */
245         kfree_skb(skb);
246         return NET_RX_DROP;
247     }
248
249     return NET_RX_SUCCESS;
250 }
```