IP Reassembly

The *ip_local_deliver()* function, defined in net/ipv4/ip_input.c, is called by *ip_rcv_finish*.(). Its function is to *reassemble IP fragments that are destined for this host* and to *call ip_local_deliver_finish () to deliver a complete packet to transport layer.*

```
293 int ip_local_deliver(struct sk_buff *skb)
294 {
295     /*
296      *      Reassemble IP fragments.
297      */
```

The constants IP_MF and IP_OFFSET are defined in include/net/ip.h and are used to access the fragment management field of the IP header.

```
 73 #define IP_MF      0x2000 /* Flag: "More Fragments" */
 74 #define IP_OFFSET  0x1FFF /* "Fragment Offset" part */
```

When an IP packet has the IP_MF flag set or the 13 bit fragment offset is not 0, a call to the *ip_defrag()* function is made. The reason for the or condition is that the last fragment of a fragmented packet will not have IP_MF set but will have a non−zero offset. If the packet to which the received fragment belongs is still incomplete *ip_defrag()* returns NULL. In this case a return is made immediately. If this fragment completes the packet, a pointer to the reassembled packet is returned, and the packet is forwarded to *ip_local_deliver_finish()* via the *NF_HOOK()* call specifying the NF_IP_LOCAL_IN chain.

```
299     if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
300         skb = ip_defrag(skb);
301         if (!skb)
302             return 0;
303     }
```

When the packet is not fragmented or was completely reassembled by *ip_defrag(),* a call to *ip_local_deliver_finish* is made to deliver it to transport layer.

```
305     return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb,
306         skb->dev, NULL, ip_local_deliver_finish);
307 }
```

The remainder of this section is dedicated to the operation of ip_defrag() which is responsible for the reassembly of fragmented packets and is defined in net/ipv4/ip_fragment.c. Key data structures are also defined in ip_fragment.c

Each packet that is being reassembled is defined by a struct ipq which is defined in net/ipv4/ip_fragment.c.

```
68 struct ipq {
69     struct ipq *next;     /* linked list pointers */
70     u32 saddr;            /* These fields comprise */
71     u32 daddr;           /* the lookup key        */
72     u16 id;
73     u8 protocol;
74     u8 last_in;
75 #define COMPLETE 4
76 #define FIRST_IN 2
77 #define LAST_IN 1
78
79     struct sk_buff *fragments; /* linked list of frags*/
80     int             len;  /* total length of original pkt */
81     int             meat; /* number of bytes so far.  */
82     spinlock_t      lock;
83     atomic_t        refcnt;
84     struct timer_list timer; /* when will queue expire?  */
85     struct ipq **pprev;
86     int             iif;
87     struct timeval stamp;
88 };
```

Functions of structure elements include:

next:	Used to link ipq structures in the same hash bucket.	
len:	Offset of last data byte in the fragment queue. It is equal to the maximum value of fragment offset plus fragment length seen so far.	
fragments:	Points to first element in a list of received fragments.	
meat:	Sum of the length of the fragments that have been received so far. When the last fragment has been received and meat == len reassembly has succeded.	
last_in:	Flags field.	
	COMPLETE:	Fragments queue is complete.
	FIRST_IN:	First fragment (has offset zero) is on queue.
	LAST_IN:	Last fragment is on queue.
timer:	A timer used for cleaning up an old incomplete fragments queue.	

The variable, ip_frag_mem, is used to track the total amount of memory used for packet reassembly. It is a global defined in net/ipv4/ip_fragment.c and initialized to 0.

```
     /* Memory used for fragments */
130 atomic_t ip_frag_mem = ATOMIC_INIT(0);
```

The variable sysctl_ipfrag_high_thresh which is mapped in the /proc file system is declared and initialized in net/ipv4/ip_fragment.c.

```
    /*   Fragment cache limits. We will commit 256K at one time.
         Should we cross that limit we will prune down to 192K.
         This should cope with even the most extreme cases
         without allowing an attacker to measurably harm machine
         performance.
    */
 51 int sysctl_ipfrag_high_thresh = 256*1024;
 52 int sysctl_ipfrag_low_thresh = 192*1024;
```

The ip_defrag() function is passed a pointer to the sk_buff which is known here to contain an element of a fragmented IP packet.

```
596 /* Process an incoming IP datagram fragment. */
597 struct sk_buff *ip_defrag(struct sk_buff *skb)
598 {
599     struct iphdr *iph = skb->nh.iph;
600     struct ipq *qp;
601     struct net_device *dev;
602
603     IP_INC_STATS_BH(IpReasmReqds);
```

Its first order of business is to determine if there is a shortage of reassembly storage. When the value ip_frag_mem exceeds the high threshold value (sysctl_ipfrag_high_thresh), a call is made to the ip_evictor() function so that some partially reassembled packets can be discarded.

```
605     /* Start by cleaning up the memory. */
606     if (atomic_read(&ip_frag_mem) >
                              sysctl_ipfrag_high_thresh)
607         ip_evictor();
```

If the fragment being processed is the first fragment of a new packet to arrive, a queue is created to manage its reassembly. Otherwise, the fragment is enqueued in the existing queue. The ip_find() function is responsible for finding the queue to which a fragment belongs or creating a new queue if that is required. Its operation will be considered later.

```
609     dev = skb->dev;
610
611     /* Lookup (or create) queue header */
612     if ((qp = ip_find(iph)) != NULL) {
613          struct sk_buff *ret = NULL;
614
615          spin_lock(&qp->lock);
616
```

If the queue was found , the ip_frag_queue() function is used to add the sk_buff to the fragment queue.

```
617          ip_frag_queue(qp, skb);
```

When both the first and last fragments have been received and fragments queue (packet) becomes complete, ip_frag_reasm is called to perform reassembly. How could meat == len without FIRST_IN and LAST_IN set.

```
619          if (qp->last_in == (FIRST_IN|LAST_IN) &&
620                    qp->meat == qp->len)
621               ret = ip_frag_reasm(qp, dev);
622
```

With reassembly complete, the queue is no longer needed and it is destroyed here.

```
623          spin_unlock(&qp->lock);
624          ipq_put(qp);
625          return ret;
626     }
```

In case of any error, the fragment is discarded.

```
628     IP_INC_STATS_BH(IpReasmFails);
629     kfree_skb(skb);
630     return NULL;
631 }
```

Finding the ***ipq*** that owns the arriving ***sk_buff***

The ip_find() function is defined in net/ipv4/ip_fragment.c. Mapping of a fragment to a struct ipq is hash based.

```
    /*   Find the correct entry in the "incomplete
         datagrams" queue for this IP datagram, and create
         new one, if nothing is found.
     */
345 static inline struct ipq *ip_find(struct iphdr *iph)
346 {
347     __u16 id = iph->id;
348     __u32 saddr = iph->saddr;
349     __u32 daddr = iph->daddr;
350     __u8 protocol = iph->protocol;
351     unsigned int hash = ipqhashfn(id, saddr, daddr,
                                                  protocol);
```

The iphashfn() is defined below. It returns a hash value based on identification number, source address, destination address and protocol number of the fragment.

```
    /*
         Was: ((((id) >> 1) ^ (saddr) ^ (daddr) ^ (prot)) &
         (IPQ_HASHSZ - 1))
         I see, I see evil hand of bigendian mafia. On Intel all
         the packets hit one hash bucket with this hash function.
         8)
     */
120 static __inline__ unsigned int ipqhashfn(u16 id, u32
   saddr, u32 daddr, u8 prot)
121 {
122     unsigned int h = saddr ^ daddr;
123
124     h ^= (h>>16)^id;
125     h ^= (h>>8)^prot;
126     return h & (IPQ_HASHSZ - 1);
127 }
```

ipq_hash is a hash table with sixty−four buckets used to keep track of various fragments queues. ip_frag_nqueues denotes the total number of such queues in the hash table. The ipfrag_lock is a read/write lock used to protect insertion and removal of ipq's.

```
90 /* Hash table. */
91
92 #define IPQ_HASHSZ      64
93
94 /* Per−bucket lock is easy to add now. */
95 static struct ipq *ipq_hash[IPQ_HASHSZ];
96 static rwlock_t ipfrag_lock = RW_LOCK_UNLOCKED;
97 int ip_frag_nqueues = 0;
```

The ip_find() function continues by searching the chain indexed by hash for an ipq that matches fragment's identification number, source address, destination address and protocol number. If one is found a pointer to it is returned.

```
352     struct ipq *qp;
353
354     read_lock(&ipfrag_lock);
355     for(qp = ipq_hash[hash]; qp; qp = qp->next) {
356          if(qp->id == id          &&
357                    qp->saddr == saddr   &&
358                    qp->daddr == daddr   &&
359                    qp->protocol == protocol) {
360               atomic_inc(&qp->refcnt);
361               read_unlock(&ipfrag_lock);
362               return qp;
363          }
364     }
365     read_unlock(&ipfrag_lock);
```

When the first fragment of a packet to arrives, the search will fail. In this case, ip_frag_create is called to create a new fragments queue for enqueuing received fragment.

```
367     return ip_frag_create(hash, iph);
368 }
```

Creating a new ipq element

The ip_frag_create(), defined in net/ipv4/ip_fragment.c creates a new ipq element and inserts it into the proper hash chain.

```
310 /*  Add an entry to the 'ipq' queue for a newly
        received IP datagram. */
311 static struct ipq *ip_frag_create(unsigned hash,
        struct iphdr *iph)
312 {
313     struct ipq *qp;
314
315     if ((qp = frag_alloc_queue()) == NULL)
316          goto out_nomem;
```

frag_alloc_queue is an inline function defined as below. atomic_add is used to add size of struct ipq structure kmalloc'd to atomic_t type variable ip_frag_mem. Recall that ip_frag_mem denotes the amount of memory used in keeping track of fragments. Why was not the slab allocator used here?? Rareness of fragmentation??

```
145 static __inline__ struct ipq *frag_alloc_queue(void)
146 {
147     struct ipq *qp = kmalloc(sizeof(struct ipq),
                         GFP_ATOMIC);
148
149     if(!qp)
150          return NULL;
151     atomic_add(sizeof(struct ipq), &ip_frag_mem);
152     return qp;
153 }
```

On return to ip_frag_create the newly created queue is initialized.

```
318     qp->protocol = iph->protocol;
319     qp->last_in = 0;
320     qp->id = iph->id;
321     qp->saddr = iph->saddr;
322     qp->daddr = iph->daddr;
323     qp->len = 0;
324     qp->meat = 0;
325     qp->fragments = NULL;
326     qp->iif = 0;
```

Continuing in ip_frag_create the data and function members of the timer for this queue are initialized. Note that expires is not set and the timer is not yet added.

```
328     /* Initialize a timer for this entry. */
329     init_timer(&qp->timer);
330     qp->timer.data = (unsigned long) qp; /* pointer to
                                                queue */
331     qp->timer.function = ip_expire;    /* expire
                                                function  */
332     qp->lock = SPIN_LOCK_UNLOCKED;
333     atomic_set(&qp->refcnt, 1);
```

ip_frag_intern is called to add the newly created fragments queue to a hash table that manages all such queues.

```
335     return ip_frag_intern(hash, qp);
```

On failing to allocate a fragments queue structure, we return NULL.

```
337 out_nomem:
338     NETDEBUG(if (net_ratelimit()) printk(KERN_ERR
                  "ip_frag_create: no memory left !\n"));
339     return NULL
340 }
```

Inserting the new ipq into the hash chain.

The ip_frag_intern() function inserts the newly created ipq in the proper hash queue.

```
270 /* Creation primitives. */
271
272 static struct ipq *ip_frag_intern(unsigned int hash,
        struct ipq *qp_in)
273 {
274     struct ipq *qp;
275
```

On an SMP kernel, to avoid a race condition where another CPU creates a similar queue and adds it to the hash table, a recheck is enforced here. If the queue was added by another CPU, a pointer to the existing ipq is returned and the newly created ipq is destroyed..

```
276     write_lock(&ipfrag_lock);
277 #ifdef CONFIG_SMP
278     /*   With SMP race we have to recheck hash table,
        because such entry could be created on other
        cpu, while we promoted read lock to write lock.
281     */
282     for(qp = ipq_hash[hash]; qp; qp = qp->next) {
283         if(qp->id == qp_in->id &&
284                 qp->saddr == qp_in->saddr    &&
285                 qp->daddr == qp_in->daddr    &&
286                 qp->protocol == qp_in->protocol) {
```

This block is executed only if the queue that was about to be added already exists. As described above this situation can only occur on an SMP system and should be extremely rare there. The block destroys the new queue and returns a pointer to the existing one.

```
287             atomic_inc(&qp->refcnt);
288             write_unlock(&ipfrag_lock);
```

The COMPLETE flag needs to be set for ipq_put to destroy a fragments queue.

```
289             qp_in->last_in |= COMPLETE;
290             ipq_put(qp_in);
291             return qp;
292         }
293     }
294 #endif
```

Arming the timeout timer

The mod_timer() functions sets the expires member of timer qp->timer and adds it to the list of timers maintained by kernel. The dependence on the value returned by modtimer is not clear. The value it returns is actually computed as shown in detatch_timer(). Thus if the timer is not presently pending qp->refcount is incremented twice. Otherwise it is incremented only once.

```
198     if (!timer_pending(timer))
199          return 0;
200     list_del(&timer->list);
201          return 1;
```

```
 54 /*  Important NOTE! Fragment queue must be destroyed
        before MSL expires. RFC791 is wrong proposing to
        prolongate timer each fragment arrival by TTL.
    */
 57 int sysctl_ipfrag_time = IP_FRAG_TIME;
```

Continuing in ip_frag_intern() the reassembly timeout timer is armed. IP_FRAG_TIME is defined in include/net/ip.h as 30 seconds.

```
76 #define IP_FRAG_TIME   (30 * HZ) /* fragment lifetime */
```

```
295     qp = qp_in;
296
297     if (!mod_timer(&qp->timer, jiffies +
                                   sysctl_ipfrag_time))
298          atomic_inc(&qp->refcnt);
299
300     atomic_inc(&qp->refcnt);
```

The new queue is inserted at the head of corresponding hash bucket and the count of all reassembly queues is incremented.

```
301     if((qp->next = ipq_hash[hash]) != NULL)
302          qp->next->pprev = &qp->next;
303     ipq_hash[hash] = qp;
304     qp->pprev = &ipq_hash[hash];
305     ip_frag_nqueues++;
306     write_unlock(&ipfrag_lock);
307     return qp;
308 }
```

The ip_frag_queue() function, called by ip_defrag() , enqueues received fragment in the fragments queue returned by ip_find.

```
   /* Add new segment to existing queue. */
371 static void ip_frag_queue(struct ipq *qp, struct
        sk_buff *skb)
372 {
373     struct sk_buff *prev, *next;
374     int flags, offset;
375     int ihl, end;
```

If fragments queue is already complete or doomed for destruction, the fragment is discarded straightaway.

```
377     if (qp->last_in & COMPLETE)
378         goto err;
```

Determine the index of the last byte of this fragment and remember it in end.

```
380     offset = ntohs(skb->nh.iph->frag_off);
381     flags = offset & ~IP_OFFSET;
382     offset &= IP_OFFSET;
383     offset <<= 3;     /* offset is in 8-byte chunks */
384     ihl = skb->nh.iph->ihl * 4;
385
386     /* Determine the position of this fragment. */
387     end = offset + skb->len - ihl;
```

When received fragment is the last one, set LAST_IN flag and qp->len is updated if necessary.

```
389     /* Is this the final fragment? */
390     if ((flags & IP_MF) == 0) {
391         /*   If we already have some bits beyond end
392              or have different end, the segment is
                 corrrupted.
393          */
394         if (end < qp->len ||
395             ((qp->last_in & LAST_IN) && end != qp->len))
396             goto err;
```

This looks like a legitimate last fragment.

```
397         qp->last_in |= LAST_IN;
398         qp->len = end;
```

Fragment flags indicate that this should not be the last fragment. Since fragmentation occurs on 8 byte boundaries each fragment should end on an 8 byte boundary. However, since end was derived from skb->len it is possible that it does not represent an 8 byte boundary. In that case it is coerced to 8 byte alignment. As usual the checksumming at this point remains something of a mystery.

```
399     } else {
400          if (end&7) {
401               end &= ~7;
402               if (skb->ip_summed != CHECKSUM_UNNECESSARY)
403                    skb->ip_summed = CHECKSUM_NONE;
404          }
```

If the current end exceeds qp->len, then the packet is discarded if the last fragment has already been seen, or qp->len is reset to end if the last fragment has not been seen.

```
405          if (end > qp->len) {
406              /* Some bits beyond end -> corruption. */
407               if (qp->last_in & LAST_IN)
408                    goto err;
409               qp->len = end;
410          }
411     }
```

It would appear that the only way this could happen is that a fragment which is not the last fragment but had length < 8 was recieved.

```
412     if (end == offset)
413          goto err;
```

pskb_pull first ensures that IP header is entirely resident in kmalloc'd header and then advances "data " pointer by size of IP header.

```
415     if (pskb_pull(skb, ihl) == NULL)
416          goto err;
```

pskb_trim trims fragment, if required, to desired size (IP packet size minus IP header size).

```
417     if (pskb_trim(skb, end-offset))
418          goto err;
```

Determine the position of the received fragment in the fragments queue.

The offset of each fragment is held in the control buffer of the sk_buff.

offset: Fragment offset.
h: A pointer to a structure that contains IP options.

```
420     /*   Find out which fragments are in front and at
             the back of us in the chain of fragments so
             far.  We must know where to put this
             fragment, right?
423      */
424     prev = NULL;
425     for(next = qp->fragments; next != NULL; next =
                                        next->next) {
426          if (FRAG_CB(next)->offset >= offset)
427               break;  /* bingo! */
428          prev = next;
429     }
```

FRAG_CB is a macro defined in net/ipv4/ip_fragment.c. It returns a pointer to control buffer space in the fragment. It is type cast from char * to struct ipfrag_skb_cb *.

```
 59 struct ipfrag_skb_cb
 60 {
 61     struct inet_skb_parm    h;
 62     int                     offset;
 63 };

 65 #define FRAG_CB(skb)
                ((struct ipfrag_skb_cb*)((skb)->cb))
```

Front end overlap

Check to see if the received fragment overlaps the preceeding fragment (prev). Variable "i" is positive in case of an overlap. Adjust "offset " and use pskb_pull to remove overlapped section from received fragment.

```
431     /*   We found where to put this one.  Check for
             overlap with preceding fragment, and, if
             needed, align things so that any overlaps are
             eliminated.
434      */
435     if (prev) {
436          int i = (FRAG_CB(prev)->offset + prev->len)
                       - offset;
437
438          if (i > 0) {
439               offset += i;
440               if (end <= offset)
441                    goto err;
442               if (!pskb_pull(skb, i))
443                    goto err;
444               if (skb->ip_summed != CHECKSUM_UNNECESSARY)
445                    skb->ip_summed = CHECKSUM_NONE;
446          }
447     }
```

Back end overlap

Check if received fragment overlaps with succeeding fragments (next and others).

```
449     while (next && FRAG_CB(next)->offset < end) {
450          int i = end - FRAG_CB(next)->offset; /*
                        overlap is 'i' bytes */
451
452          if (i < next->len) {
```

The variable 'i' denotes number of bytes of overlap with this fragment. In this case, the overlap is partial. Use pskb_pull to remove this overlapped section and adjust "offset" of this fragment (next). Since pskb_pull() is extracting data from an existing fragment whose data has already been included in the reassembly count, it is necessary to decrement qp->meat.

```
453          /*   Eat head of the next overlapped
                  fragment and leave the loop. The
                  next ones cannot overlap.
455           */
456               if (!pskb_pull(next, i))
457                    goto err;
458               FRAG_CB(next)->offset += i;
459               qp->meat -= i;
460               if (next->ip_summed != CHECKSUM_UNNECESSARY)
461                    next->ip_summed = CHECKSUM_NONE;
462                    break;
```

In case the entire fragment overlaps with received fragment, detach it from fragments queue and free it using function frag_kfree_skb.

```
463          } else {
464               struct sk_buff *free_it = next;
465
466                /*  Old fragmnet is completely
                      overridden with new one drop it.
468                 */
469               next = next->next;
470
471               if (prev)
472                    prev->next = next;
473               else
474                    qp->fragments = next;
475
476               qp->meat -= free_it->len;
477               frag_kfree_skb(free_it);
478          }
479}
```

frag_kfree_skb is an inline function defined as below. atomic_sub is used to subtract true size of fragment from atomic_t type variable ip_frag_mem. Recall that ip_frag_mem denotes the amount of memory used in keeping track of fragments.

```
133 static __inline__ void frag_kfree_skb(struct sk_buff
                                        *skb)
134 {
135     atomic_sub(skb->truesize, &ip_frag_mem);
136     kfree_skb(skb);
137 }
```

Finally, we set "offset " field in control buffer (struct ipfrag_skb_cb) of received fragment and insert the fragment into its fragments queue.

```
481     FRAG_CB(skb)->offset = offset;
482
483  /* Insert this fragment in the chain of fragments. */
484     skb->next = next;
485     if (prev)
486          prev->next = skb;
487     else
488          qp->fragments = skb;
```

Set fragments queue timestamp to that of received fragment. Add the true size of fragment to ip_frag_mem. If received fragment has a zero offset, set flag FIRST_IN to indicate that first fragment is on the queue. qp->meat is incremented by size of the fragment.

```
490     if (skb->dev)
491          qp->iif = skb->dev->ifindex;
492     skb->dev = NULL;
493     qp->stamp = skb->stamp;
494     qp->meat += skb->len;
495     atomic_add(skb->truesize, &ip_frag_mem);

496     if (offset == 0)
497          qp->last_in |= FIRST_IN;
498
499     return;
```

In case of an error, discard the fragment, but not the queue.

```
501 err:
502     kfree_skb(skb);
503 }
```

Removal of old queues.

The ip_evictor() function is defined in net/ipv4/ip_fragment.c. It destroys old fragments queues until ip_frag_mem falls below sysctl_ipfrag_low_thresh.

```
    /*   Memory limiting on fragments.  Evictor trashes the
         oldest fragment queue until we are back under the
         low threshold.
    */
203 static void ip_evictor(void)
204 {
205     int i, progress;
206
207     do {
208          if (atomic_read(&ip_frag_mem) <=
             sysctl_ipfrag_low_thresh)
209               return;
210          progress = 0;
```

Run through each hash bucket, freeing the oldest fragments queue (if any) in it. Since, we always add a new fragments queue at the head of the hash bucket, the last fragment queue is the oldest in it. ipq_kill unlinks the given queue from its hash bucket and ipq_put destorys it.

```
211          /*   FIXME: Make LRU queue of frag heads.
                  -DaveM */
212          for (i = 0; i < IPQ_HASHSZ; i++) {
213               struct ipq *qp;
214               if (ipq_hash[i] == NULL)
215                    continue;
216
217               read_lock(&ipfrag_lock);
218               if ((qp = ipq_hash[i]) != NULL) {
```

Go to the end of the queue where the oldest element necessarily resides.

```
219                    /* find the oldest queue for this
                            hash bucket */
220                    while (qp->next)
221                         qp = qp->next;
222                    atomic_inc(&qp->refcnt);
223                    read_unlock(&ipfrag_lock);
224
```

If that queue is not complete (meaning already unlinked) call ipq_kill() to disarm its timer and unlink it.

```
225                     spin_lock(&qp->lock);
226                     if (!(qp->last_in&COMPLETE))
227                          ipq_kill(qp);
228                     spin_unlock(&qp->lock);
229
```

Call ipq_put to destroy it.

```
230                     ipq_put(qp);
231                     IP_INC_STATS_BH(IpReasmFails);
232                     progress = 1;
233                     continue;
234                }
235                read_unlock(&ipfrag_lock);
236           }
237      } while (progress);
238 }
```

ipq_kill is defined in net/ipv4/ip_fragment.c. It deletes timer for this queue from the list of timers maintained by the kernel.

```
     /*   Kill ipq entry. It is not destroyed immediately,
          because caller (and someone more) holds reference
          count.
     */
188  static __inline__ void ipq_kill(struct ipq *ipq)
189  {
190      if (del_timer(&ipq->timer))
191           atomic_dec(&ipq->refcnt);
192
193      if (!(ipq->last_in & COMPLETE)) {
194           ipq_unlink(ipq);
195           atomic_dec(&ipq->refcnt);
196           ipq->last_in |= COMPLETE;
197      }
198  }
```

ipq_unlink merely calls __ipq_unlink.

```
107 static __inline__ void ipq_unlink(struct ipq *ipq)
108 {
109     write_lock(&ipfrag_lock);
110     __ipq_unlink(ipq);
111     write_unlock(&ipfrag_lock);
112 }
```

__ipq_unlink does the real work of removing the queue from its hash bucket.

```
 99 static __inline__ void __ipq_unlink(struct ipq *qp)
100 {
101         if(qp->next)
102                 qp->next->pprev = qp->pprev;
103         *qp->pprev = qp->next;
104         ip_frag_nqueues--;
105 }
```

After unlinking the queue, ipq_put is called to free all fragments in the queue.
It calls ip_frag_destroy, when queue's reference count becomes zero.

```
179 static __inline__ void ipq_put(struct ipq *ipq)
180 {
181     if (atomic_dec_and_test(&ipq->refcnt))
182         ip_frag_destroy(ipq);
183 }
```

ip_frag_destroy does the real work of freeing fragments in the queue.

```
159 static void ip_frag_destroy(struct ipq *qp)
160 {
161     struct sk_buff *fp;
162
163     BUG_TRAP(qp->last_in&COMPLETE);
164     BUG_TRAP(del_timer(&qp->timer) == 0);
165
166     /* Release all fragment data. */
167     fp = qp->fragments;
168     while (fp) {
169          struct sk_buff *xp = fp->next;
171          frag_kfree_skb(fp);
172          fp = xp;
173     }
175     /* Finally, release the queue descriptor itself.*/
176     frag_free_queue(qp);
177 }
```

frag_kfree_skb frees the fragment and decreases ip_frag_mem by its true size.

```
133 static __inline__ void frag_kfree_skb(struct sk_buff
                                           *skb)
134 {
135     atomic_sub(skb->truesize, &ip_frag_mem);
136     kfree_skb(skb);
137 }
```

Finally, the queue descriptor is freed and ip_frag_mem is decremented by its size.

```
139 static __inline__ void frag_free_queue(struct ipq *qp)
140 {
141         atomic_sub(sizeof(struct ipq), &ip_frag_mem);
142         kfree(qp);
143 }
```

Physically reassembling the packet

The ip_frag_reasm() function is defined in net/ipv4/ip_fragment.c. It builds a new IP datagram from its fragments. As seen earlier ipq_kill() disarms the timer and removes the reassembly queue from its hash chain.

```
508 static struct sk_buff *ip_frag_reasm(struct ipq *qp,
        struct net_device *dev)
509 {
510     struct iphdr *iph;
511     struct sk_buff *fp, *head = qp->fragments;
512     int len;
513     int ihlen;
514
515     ipq_kill(qp);
516
517     BUG_TRAP(head != NULL);
518     BUG_TRAP(FRAG_CB(head)->offset == 0);
```

Validate length of IP datagram. If it exceeds 64K, we return NULL.

```
520     /* Allocate a new buffer for the datagram. */
521     ihlen = head->nh.iph->ihl*4;
522     len = ihlen + qp->len;
523
524     if(len > 65535)
525         goto out_oversize;
```

If head is cloned, pskb_expand_head is called to reallocate kmalloc'd header and skb_shinfo structure.

```
527     /* Head of list must not be cloned. */
528     if (skb_cloned(head) && pskb_expand_head(head, 0,
                                    0, GFP_ATOMIC))
529         goto out_nomem;
```

If head has sk_buffs on its frag_list, we allocate a new sk buff header "clone". This section handles the extremely ugly stuff that never occurs in practice.

```
531     /*   If the first fragment is fragmented itself,
             we split it to two chunks: the first with
             data and paged part and the second, holding
             only fragments. */

534     if (skb_shinfo(head)->frag_list) {
535          struct sk_buff *clone;
536          int i, plen = 0;
537
538          if ((clone = alloc_skb(0, GFP_ATOMIC)) ==
                  NULL)
539               goto out_nomem;
```

The clone is then linked next to head in the fragments queue. All the fragments from "frag_list" of head are moved into that of clone.

```
540          clone->next = head->next;
541          head->next = clone;
542          skb_shinfo(clone)->frag_list =
                  skb_shinfo(head)->frag_list;
543          skb_shinfo(head)->frag_list = NULL;
```

Update len and data_len members of head and clone. Increase ip_frag_mem by the true size of sk buff pointed to by clone.

```
544          for (i=0; i<skb_shinfo(head)->nr_frags; i++)
545               plen += skb_shinfo(head)->frags[i].size;
546          clone->len = clone->data_len =
                  head->data_len - plen;
547          head->data_len -= clone->len;
548          head->len -= clone->len;
549          clone->csum = 0;
550          clone->ip_summed = head->ip_summed;
551          atomic_add(clone->truesize,
             &ip_frag_mem);
552     }
```

Set frag_list of head to next fragment in the queue. This has the effect of placing all the fragments in the queue on frag_list of head (first fragment). skb_push ensures that "data" points to IP header .... did we see it being advanced beyond IP header ?? Yes, when a fragment was added to queue, its data pointer was advanced by size of IP header.

```
554     skb_shinfo(head)->frag_list = head->next;
555     skb_push(head, head->data - head->nh.raw);
```

Decrease ip_frag_mem by true size of each fragment on the queue.
Update len, data_len, truesize and csum members of head to reflect that remaining fragments are on its frag_list.

```
556     atomic_sub(head->truesize, &ip_frag_mem);
558     for (fp=head->next; fp; fp = fp->next) {
559          head->data_len += fp->len;
560          head->len += fp->len;
561          if (head->ip_summed != fp->ip_summed)
562               head->ip_summed = CHECKSUM_NONE;
563          else if (head->ip_summed == CHECKSUM_HW)
564                    head->csum = csum_add(head->csum,
                                            fp->csum);
565          head->truesize += fp->truesize;
566          atomic_sub(fp->truesize, &ip_frag_mem);
567     }
```

Finally, after reassembling fragments into one IP datagram, change IP header members, frag_off (fragment offset) to zero and tot_len (total length) to combined length of all fragments. A pointer to reassembled datagram is returned.

```
569     head->next = NULL;
570     head->dev = dev;
571     head->stamp = qp->stamp;
572
573     iph = head->nh.iph;
574     iph->frag_off = 0;
575     iph->tot_len = htons(len);
576     IP_INC_STATS_BH(IpReasmOKs);
577     qp->fragments = NULL;
578     return head;
```

In case of any error, we return NULL.

```
580 out_nomem:
581     NETDEBUG(if (net_ratelimit())
582                 printk(KERN_ERR
583                       "IP: queue_glue: no memory for
              gluing queue %p\n",
584                       qp));
585     goto out_fail;
586 out_oversize:
587     if (net_ratelimit())
588                 printk(KERN_INFO
589                       "Oversized IP packet from
              %d.%d.%d.%d.\n",
590                       NIPQUAD(qp->saddr));
591 out_fail:
592     IP_INC_STATS_BH(IpReasmFails);
593     return NULL;
594 }
```