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.

```c
260 /*
261  *    Deliver IP Packets to the higher protocol layers.
262  */
263 int ip_local_deliver(struct sk_buff *skb)
264 {
265    /*
266     * Reassemble IP fragments.
267     */
268    
```
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.

```c
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.

```c
269    if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
270       skb = ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER);
271       if (!skb)
272          return 0;
273    }
```

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.

```c
275    return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev,
276               NULL, ip_local_deliver_finish);
277 }
```
Fragment management data structures

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.

```c
/* Describe an entry in the "incomplete datagrams" queue. */
struct ipq {
    struct hlist_node list;
    struct list_head lru_list; /* lru list member */
    u32      user;
    u32      saddr;
    u32      daddr;
    u16      id;
    u8       protocol;
    u8       last_in;
    struct sk_buff *fragments; /* linked list of fragments */
    int      len;        /* total length of datagram */
    int      meat;
    spinlock_t lock;
    atomic_t refcnt;
    struct timer_list timer; /* when will queue expire? */
    struct timeval stamp;
    int      iif;
    unsigned int rid;
    struct inet_peer *peer;
};
```
Functions of structure elements include:

- **list**: Used to link ipq structures in the same hash bucket.
- **lru_list**: Used in getting rid of old ipqs.
- **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 succeeded.
- **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, \textit{ip\_frag\_mem}, is used to track the \textit{total global amount} of memory used for packet reassembly. It is a global defined in net/ipv4/ip_fragment.c and initialized to 0.

\begin{verbatim}
/* Memory used for fragments */
130 atomic_t ip_frag_mem = ATOMIC_INIT(0);
\end{verbatim}

The variable \textit{sysctl_ipfrag\_high\_thresh} which is mapped in the /proc file system is declared and initialized in net/ipv4/ip_fragment.c. These haven't changed since kernel 2.4.

\begin{verbatim}
/* 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;
\end{verbatim}
ipq hash structures

ipq_hash is a hash table with sixty-four buckets used to manage struct ipq's. 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.

101 /* Hash table. */
102
103 #define IPQ_HASHSZ 64 // 64 hash buckets
104
105 /* Per-bucket lock is easy to add now. */
106 static struct hlist_head ipq_hash[IPQ_HASHSZ];
107 static DEFINE_RWLOCK(ipfrag_lock);
108 static u32 ipfrag_hash_rnd;
109 static LIST_HEAD(ipq_lru_list);
110 int ip_frag_nqueues = 0;

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

The lookup key consists of 4 elements taken from the IP header

- id (seq #)
- protocol
- saddr
- daddr
- A user supplied value IP_DEFRAG_LOCAL_DELIVER

126 static unsigned int ipqhashfn(u16 id, u32 saddr, u32 daddr, u8 prot)
127 {
128     return jhash_3words((u32)id << 16 | prot, saddr, daddr, ipfrag_hash_rnd) & (IPQ_HASHSZ - 1);
129 }

6
The Jenkins hash function

A surprisingly complex hashing function is employed:

```c
/* The golden ration: an arbitrary value */
#define JHASH_GOLDEN_RATIO      0x9e3779b9

/* A special ultra-optimized versions that knows they are
hashing exactly 3, 2 or 1 word(s).
* NOTE: In particular the "c += length; __jhash_mix(a,b,c);"
* normally done at the end is not done here.
*/
static inline u32 jhash_3words(u32 a, u32 b, u32 c, u32 initval)
{
    a += JHASH_GOLDEN_RATIO;
    b += JHASH_GOLDEN_RATIO;
    c += initval;
    __jhash_mix(a, b, c);
    return c;
}
```
4 /* jhash.h: Jenkins hash support. 
5 */
6 * Copyright (C) 1996 Bob Jenkins
7 * (bob_jenkins@burtleburtle.net)
8 *
9 * These are the credits from Bob's sources:
10 * lookup2.c, by Bob Jenkins, December 1996, Public Domain.
11 * hash(), hash2(), hash3, and mix() are externally useful functions.
12 * Routines to test the hash are included if SELF_TEST is defined.
13 * You can use this free for any purpose. It has no warranty.
14 *
15 * Copyright (C) 2003 David S. Miller (davem@redhat.com)
16 *
17 * I've modified Bob's hash to be useful in the Linux kernel, 
18 * and 
19 * any bugs present are surely my fault. -DaveM
20 */
21 */
22
23/* NOTE: Arguments are modified. */
24#define __jhash_mix(a, b, c) \ 
25{ \ 
26 a -= b; a -= c; a ^= (c>>13); \ 
27 b -= c; b -= a; b ^= (a<<8); \ 
28 c -= a; c -= b; c ^= (b>>13); \ 
29 a -= b; a -= c; a ^= (c>>12); \ 
30 b -= c; b -= a; b ^= (a<<16); \ 
31 c -= a; c -= b; c ^= (b>>5); \ 
32 a -= b; a -= c; a ^= (c>>3); \ 
33 b -= c; b -= a; b ^= (a<<10); \ 
34 c -= a; c -= b; c ^= (b>>15); \ 
35 }
This is the 2.4 kernel hash function:

```c
/*
   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;
123124     h ^= (h>>16)^id;
125     h ^= (h>>8)^prot;
126     return h & (IPQ_HASHSZ - 1);
127 }
```
The `ip_defrag()` function

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

```c
699 /* Process an incoming IP datagram fragment. */
700 struct sk_buff *ip_defrag(struct sk_buff *skb, u32 user)
701 {
702     struct iphdr *iph = skb->nh.iph;
703     struct ipq *qp;
704     struct net_device *dev;
705
706     IP_INC_STATS_BH(IPSTATS_MIB_REASMREQDS);
707
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.

```c
708     /* Start by cleaning up the memory. */
709     if (atomic_read(&ip_frag_mem) > sysctl_ipfrag_high_thresh)
710         ip_evictor();
711```
The reassembly driver

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.

```c
712    dev = skb->dev;
713
714    /* Lookup (or create) queue header */
715    if ((qp = ip_find(iph, user)) != NULL) {
716       struct sk_buff *ret = NULL;
717       spin_lock(&qp->lock);
718
720       ip_frag_queue(qp, skb);
```

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

```c
722       if (qp->last_in == (FIRST_IN|LAST_IN) &&
723             qp->meat == qp->len)
724           ret = ip_frag_reasm(qp, dev);
```

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.... out of order delivery...

```c
726       spin_unlock(&qp->lock);
727       ipq_put(qp);
728       return ret;
729    }
```

The call to ipq_put drops a reference to the queue. When reassembly is complete, the queue is no longer needed and it is destroyed here.
In case of any error, the fragment is silently discarded.

    731     IP_INC_STATS_BH(IpReasmFails);
    732     kfree_skb(skb);
    733     return NULL;
    734 }
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. The default number of hash queues is 64. The "user" parameter normally has the value **IP DEFRAG_LOCAL_DELIVER**.

```c
static struct ipq *ipq_hash[IPQ_HASHSZ];

/* Find the correct entry in the "incomplete datagrams" queue for this IP datagram, and create new one, if nothing is found. */

387 static inline struct ipq *ip_find(struct iphdr *iph, u32 user)
388 {
389    __be16 id = iph->id;
390    __u32 saddr = iph->saddr;
391    __u32 daddr = iph->daddr;
392    __u8 protocol = iph->protocol;
393    unsigned int hash;
394    struct ipq *qp;
395    struct hlist_node *n;
396    read_lock(&ipfrag_lock);
397    hash = ipqhashfn(id, saddr, daddr, protocol);
398    hash = ipqhashfn(id, saddr, daddr, protocol);
```
The \texttt{ip_find()} function continues by searching the chain indexed by \texttt{hash} for an \texttt{ipq} that matches \texttt{fragment's identification number, source address, destination address and protocol number}. If one is found a pointer to it is returned.

```c
399     hlist_for_each_entry(qp, n, &ipq_hash[hash], list) {
400       if(qp->id == id &&
401          qp->saddr == saddr &&
402          qp->daddr == daddr &&
403          qp->protocol == protocol &&
404          qp->user == user) {
405          atomic_inc(&qp->refcnt);
406          read_unlock(&ipfrag_lock);
407          return qp;
408     }
410    read_unlock(&ipfrag_lock);
411    return ip_frag_create(iph, user);
413 }
```

When the first fragment of a packet arrives, the search will fail. In this case, \texttt{ip_frag_create} is called to create a new fragments queue for enqueuing received fragment.
Creating a new ipq element

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

```c
351 static struct ipq *ip_frag_create(struct iphdr *iph,
                                      u32 user)
352 {
353    struct ipq *qp;
354
355    if ((qp = frag_alloc_queue()) == NULL)
356        goto out_nomem;
```

On return to `ip_frag_create` the newly created queue is initialized.

```c
358    qp->protocol = iph->protocol;
359    qp->last_in = 0;
360    qp->id = iph->id;
361    qp->saddr = iph->saddr;
362    qp->daddr = iph->daddr;
363    qp->user = user;
364    qp->len = 0;
365    qp->meat = 0;
366    qp->fragments = NULL;
367    qp->iif = 0;
368    qp->peer = sysctl_ipfrag_max_dist ?
                    inet_getpeer(iph->saddr, 1) : NULL;
369```

```c
```
Here 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 armed.

370    /* Initialize a timer for this entry. */
371    init_timer(&qp->timer);
372    qp->timer.data = (unsigned long) qp; /* pointer to queue */
373    qp->timer.function = ip_expire;     /* expire function */
374    spin_lock_init(&qp->lock);
375    atomic_set(&qp->refcnt, 1);
376

The *ip_frag_intern()* function is called to add the newly created fragments queue to the hash table that manages all such queues.

377    return ip_frag_intern(qp);

On failing to allocate a fragments queue structure, NULL is returned.

337    out_nomem:
338        NETDEBUG(if (net_ratelimit()) printk(KERN_ERR
339               "ip_frag_create: no memory left !\n"));
340    return NULL
341    }
379    out_nomem:
380        LIMIT_NETDEBUG(KERN_ERR
381               "ip_frag_create: no memory left !\n");
382    return NULL;
383    }

16
Allocating a new ipq

frag_alloc_queue is an inline function defined as below. atomic_add is used to add size of struct ipq structure kalloc'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??

```c
182 static __inline__ struct ipq *frag_alloc_queue(void)  
183 {  
184 struct ipq *qp = kmalloc(sizeof(struct ipq), GFP_ATOMIC);  
185  
186 if(!qp)  
187 return NULL;  
188 atomic_add(sizeof(struct ipq), &ip_frag_mem);  
189 return qp;  
190 }
```
Inserting the new ipq into the hash chain.

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

```c
306 static struct ipq *ip_frag_intern(struct ipq *qp_in) {
307    struct ipq *qp;
308
309    #ifdef CONFIG_SMP
310        struct hlist_node *n;
311    #endif
312    unsigned int hash;
313
314    write_lock(&ipfrag_lock);
315    hash = ipqhashfn(qp_in->id, qp_in->saddr, qp_in->daddr,
316                qp_in->protocol);
317    #ifdef CONFIG_SMP
318        /* With SMP race we have to recheck hash table, because
319           * such entry could be created on other cpu, while we
320           * promoted read lock to write lock.
321           */
322        hlist_for_each_entry(qp, n, &ipq_hash[hash], list) {
323            if(qp->id == qp_in->id &&
324                qp->saddr == qp_in->saddr &&
325                qp->daddr == qp_in->daddr &&
326                qp->protocol == qp_in->protocol &&
327                qp->user == qp_in->user) {
```

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.
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.

328          atomic_inc(&qp->refcnt);

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

329          write_unlock(&ipfrag_lock);
330          qp_in->last_in |= COMPLETE;
331          ipq_put(qp_in, NULL);
332          return qp;
333       }
334    }
335 #endif
Arming the timeout timer

Continuing in ip_frag_intern() the reassembly timeout timer is armed. IP_FRAG_TIME is defined in include/net/ip.h as 30 seconds. 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 mod_timer() is a little fuzzy. The value it returns is 0 if the timer was unarmed at the time of the call and 1 if it was already armed. Thus if the timer is not presently pending qp->refcount is incremented twice. Otherwise it is incremented only once.

The normal case here would seem to be that since this is a new ipq the timer will not be already armed and thus the double increment will occur. This will cause the ipq_put() call back in the ip_defrag() function not to free the ipq structure until the reassembly function also does a decrement.

57 int sysctl_ipfrag_time = IP_FRAG_TIME;

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

336     qp = qp_in;
337
338     if (!mod_timer(&qp->timer, jiffies + sysctl_ipfrag_time))
339        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.

341        atomic_inc(&qp->refcnt);
342        hlist_add_head(&qp->list, &ipq_hash[hash]);
343        INIT_LIST_HEAD(&qp->lru_list);
344        list_add_tail(&qp->lru_list, &ipq_lru_list);
345        ip_frag_nqueues++;
346        write_unlock(&ipfrag_lock);
347        return qp;
348 }
Adding new fragments to the ipq

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

465 /* Add new segment to existing queue. */
466 static void ip_frag_queue(struct ipq *qp,
467        struct sk_buff *skb)
468 {
469    struct sk_buff *prev, *next;
470    int flags, offset;
471    int ihl, end;

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

472    if (qp->last_in & COMPLETE)
473       goto err;

The index to the first byte in this fragment is stored in offset. The index of the next byte beyond this fragment is retained in end.

481    offset = ntohs(skb->nh.iph->frag_off);
482    flags = offset & ~IP_OFFSET;
483    offset &= IP_OFFSET;
484    offset <<= 3; /* offset is in 8-byte chunks */
485    ihl = skb->nh.iph->ihl * 4;
486    end = offset + skb->len - ihl;
Testing for the last fragment

When the received fragment is the last one, the LAST_IN flag is set and \textit{qp->len} is tested for consistency. If the \textit{end} of this fragment is less than \textit{qp->len} there is a definite problem. Another problem is seeing two LAST_IN fragments whose ends disagree.

\begin{verbatim}
490    /* Is this the final fragment? */
491    if ((flags & IP_MF) == 0) {
492       /* If we already have some bits beyond end
493        * or have different end, the segment is corrupted.
494        */
495       if (end < qp->len ||
496              ((qp->last_in & LAST_IN) && end != qp->len))
497          goto err;
498    }
499    qp->last_in |= LAST_IN;
500    qp->len = end;
\end{verbatim}

This looks like a legitimate last fragment.
Handling not the last fragment

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 check summing at this point remains something of a mystery.

```c
500    } else {
501       if (end&7) {
502          end &= ~7;
503          if (skb->ip_summed != CHECKSUM_UNNECESSARY)
504             skb->ip_summed = CHECKSUM_NONE;
505       }
```

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

```c
506       if (end > qp->len) {
507          /* Some bits beyond end -> corruption. */
508          if (qp->last_in & LAST_IN)
509             goto err;
510          qp->len = end;
511       }
512    }
```

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 received.

```c
513       if (end == offset)
514          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. It is important that skb->data point to the start of the transport header -- and not the fragments IP header.

```c
516   if (pskb_pull(skb, ihl) == NULL)
517       goto err;
```

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

```c
518   if (pskb_trim_rcsum(skb, end - offset))
519       goto err;
```
Determining 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.
```

521    /* Find out which fragments are in front and at the back of us
522    * in the chain of fragments so far. We must know where to put
523    * this fragment, right?
524    */
525    prev = NULL;
526    for(next = qp->fragments; next != NULL; next = next->next) {
527       if (FRAG_CB(next)->offset >= offset)
528          break;    /* bingo! */
529       prev = next;
530    }

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)
66     ((struct ipfrag_skb_cb*)((skb)->cb))
```
Front end overlap

If the received fragment overlaps the tail preceding fragment (prev), it is necessary to adjust "offset" and use skb_pull to remove the overlapping section from the received fragment. The variable "i" is positive in case of an overlap.

By removing the head of the newly received fragment, having to consider the possibility of overlapping multiple preceding fragments is avoided.

```c
530  /* We found where to put this one. Check for overlap with preceding fragment, and, if needed, align things so that any overlaps are eliminated.
535   */
536  if (prev) {
537      int i = (FRAG_CB(prev)->offset + prev->len) - offset;
538
539      if (i > 0) {
540         offset += i;
541      }
542   }
```

When offset is adjusted forward toward the end of the packet, it might pass end. In that case, the new fragment is discarded.

```c
541  if (end <= offset)
542      goto err;
543  if (!pskb_pull(skb, i))
544      goto err;
545  if (skb->ip_summed != CHECKSUM_UNNECESSARY)
546      skb->ip_summed = CHECKSUM_NONE;
547 }
548 }
```

```
prev->offset -------------------------------- prev->offset + prev->len
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td></td>
</tr>
</tbody>
</table>
offset                end
|----------------------|
<---------------------->
```

26
Back end overlap

If the received fragment overlaps succeeding fragments (next and others), the old stuff is replaced by the content of the new fragment. This requires an ugly loop in which multiple existing fragments may be discarded. Recall that end identifies the end of this fragment and that i is the length of the overlap.

```c
while (next && FRAG_CB(next)->offset < end) {
    int i = end - FRAG_CB(next)->offset; /* overlap is 'i' bytes */

    if (i < next->len) {

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

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.
New fragment covers next fragment

In case the received fragment completely covers the next fragment. The next fragment is detached from fragments queue and freed using function `frag_kfree_skb`.

```c
564       } else {
565          struct sk_buff *free_it = next;
566
567          /* Old fragment is completely overridden with
568           * new one drop it.
569           */
570          next = next->next;
571
572          if (prev)
573             prev->next = next;
574          else
575             qp->fragments = next;
576
577          qp->meat -= free_it->len;
578          frag_kfree_skb(free_it, NULL);
579       }
```
Freeing a fragment

The `frag_kfree_skb()` function is an inline function that adjusts the global variable `ip_frag_mem` and frees the fragment.

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

Adding the new fragment to the chain

Finally, the value of `offset` is saved in the control buffer (`struct ipfrag_skb_cb`) of the received fragment and the fragment inserted into the queue.

```
582    FRAG_CB(skb)->offset = offset;
583
584    /* Insert this fragment in the chain of fragments. */
585    skb->next = next;
586    if (prev)
587       prev->next = skb;
588    else
589       qp->fragments = skb;

The queue's timestamp is set to that of received fragment, and the true size of fragment is added to `ip_frag_mem`. The value `qp->meat` is incremented by size of the fragment.

```
591    if (skb->dev)
592       qp->iif = skb->dev->ifindex;
593    skb->dev = NULL;
594    skb_get_timestamp(skb, &qp->stamp);
595    qp->meat += skb->len;
596    atomic_add(skb->truesize, &ip_frag_mem);
```
If received fragment has a zero offset, the flag FIRST_IN is set to indicate that first fragment is on the queue.

```c
597    if (offset == 0)
598       qp->last_in |= FIRST_IN;
599
600    write_lock(&ipfrag_lock);
601    list_move_tail(&qp->lru_list, &ipq_lru_list);
602    write_unlock(&ipfrag_lock);
603    return;
604
605
```

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

```c
606 err:
607    kfree_skb(skb);
608 }
```
Physically reassembling the packet

The `ip_frag_reasm()` function is defined in net/ipv4/ip_fragment.c. When all fragments of a packet have been received, it links fragments together so that they logically form a single new datagram.

Recall that as the fragments arrive they are linked via the `skb->next` pointers of the struct `sk_buffs`. Having the all the buffers on the `next` chain of `head` makes it impossible for subsequent processors of the packet to recognize that this is really one logical packet. For proper semantics it is necessary for the chain to be owned by `head's skb_shinfo(head)->frag_list` pointer and not by its `next` pointer. All fragments except the first and second remain linked by the `skb->next` pointer.

```c
613 static struct sk_buff *ip_frag_reasm(struct ipq *qp,
614     struct net_device *dev)
615 {
616    struct iphdr *iph;
617    struct sk_buff *fp, *head = qp->fragments;
618    int len;
619    int ihlen;

620    ipq_kill(qp);
621
622    BUG_TRAP(head != NULL);
623    BUG_TRAP(FRAG_CB(head)->offset == 0);

625    /* Allocate a new buffer for the datagram. */
626    ihlen = head->nh.iph->ihl*4;
627    len = ihlen + qp->len;
628
629    if(len > 65535)
630       goto out_oversize;
```

As seen earlier `ipq_kill()` disarms the timer and removes the reassembly queue from its hash chain.

If the length of the IP datagram exceeds 64K, it is discarded and NULL is returned.
Dealing with a cloned head

If head, which points to the first sk_buff in the fragment chain is cloned, we have a problem. By definition cloning implies that both copies of the sk_buff point to the same data.

We are now about to change the skb_shared_info structure of the data that head owns. Therefore, it is necessary to ask the pskb_expand_head() function declone the sk_buff by reallocating the sk_buff kmalloc'ed part.

If the nhead and ntail parameters are 0, as they are here, an identical copy of the kmalloc'ed part is created.

```c
632    /* Head of list must not be cloned. */
633    if (skb_cloned(head) &&
          pskb_expand_head(head, 0, 0, GFP_ATOMIC))
634        goto out_nomem;
```
Dealing with an existing fragment list

Since head's frag_list pointer must be the base of the fragments chain, we have another problem if head already has sk_buffs on its frag_list. In this case it is necessary to create the new sk_buff structure, clone. It is unclear how this situation could possibly arise.

As described in the comments below, clone is inserted in the fragments queue following head and clone is given ownership of the fragment list formerly owned by head. The head retains ownership of any unmapped page fragments. We saw earlier that unmapped page buffers always precede the fragment chain in logical packet order.

```c
/* 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. */
if (skb_shinfo(head)->frag_list) {
    struct sk_buff *clone;
    int i, plen = 0;
    if ((clone = alloc_skb(0, GFP_ATOMIC)) == NULL)
        goto out_nomem;
    clone->next = head->next;
    head->next = clone;
    skb_shinfo(clone)->frag_list = skb_shinfo(head)->frag_list;
    skb_shinfo(head)->frag_list = NULL;
}
```

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 and the frag_list of the original queue head is set to NULL.

```c
clone->next = head->next;
head->next = clone;
skb_shinfo(clone)->frag_list = skb_shinfo(head)->frag_list;
skb_shinfo(head)->frag_list = NULL;
```
Adjusting the accounting information

Since data formerly owned by head has now been moved to clone, it is necessary to update both the len and data_len members of head and clone.

To do that *the total mass of the unmapped page fragments which are still owned by head is accumulated in plen.* Recall that len holds the amount of data associated with the sk_buff and data_len is the amount that resides in unmapped pages and the fragment list. Therefore, since the data owned by clone is exactly head’s old fragment list,

\[
\text{clone->len} == \text{clone->data_len} == \text{head->data_len - plen}.
\]

```c
for (i=0; i<skb_shinfo(head)->nr_frags; i++)
    plen += skb_shinfo(head)->frags[i].size;
clone->len = clone->data_len = head->data_len - plen;
head->data_len -= clone->len;
clone->csum = 0;
clone->ip_summed = head->ip_summed;
```

Add clone->truesize to the total global amount of memory used for fragments. The value of truesize is initialized by alloc_skb() to the size of the sk_buff structure plus the length of the kmalloc'ed area which in this case was 0.

```c
atomic_add(clone->truesize, &ip_frag_mem);
```
Moving the fragment chain from skb->next to the fragment list

Now the frag_list of head is set to point to the next fragment in the queue. This has the effect of placing all the fragments in the queue on frag_list of head (the first fragment). The call to skb_push() ensures that data points to the IP header. Recall that when a fragment was added to the queue, its data pointer was advanced by the size of IP header.

```c
659   skb_shinfo(head)->frag_list = head->next;
660   skb_push(head, head->data - head->nh.raw);
```

It is necessary now to reduce ip_frag_mem by the true size of each fragment on the queue and to update the len, data_len, truesize and csum elements of head to reflect that the remaining fragments are on its frag_list. As usual the checksumming remains unclear.

```c
661   atomic_sub(head->truesize, &ip_frag_mem);
662
663   for (fp=head->next; fp; fp = fp->next) {
664      head->data_len += fp->len;
665      head->len += fp->len;
666      if (head->ip_summed != fp->ip_summed) {  
667         head->ip_summed = CHECKSUM_NONE;
668      } else if (head->ip_summed == CHECKSUM_HW) {  
669         head->csum = csum_add(head->csum, fp->csum);
670      }
671      head->truesize += fp->truesize;
672      atomic_sub(fp->truesize, &ip_frag_mem);
673   }
```
Finally, after reassembling fragments into one IP datagram, the IP header element `frag_off` is set to zero and `tot_len` (total length) to combined length of all fragments. Since all of the fragments are now linked to the `frag_list` pointer associated with head `head->next` is set to NULL.

A pointer to reassembled datagram is returned.

```c
head->next = NULL;
head->dev = dev;
skb_set_timestamp(head, &qp->stamp);
iph = head->nh.iph;
iph->frag_off = 0;
iph->tot_len = htons(len);
IP_INC_STATS_BH(IPSTATS_MIB_REASMOKS);
qp->fragments = NULL;
return head;
```

In case of any error, NULL is returned.

```c
out_nomem:
LIMIT_NETDEBUG(KERN_ERR "IP: queue_glue: no memory for gluing "
    "queue %p\n", qp);
goto out_fail;
out_oversize:
if (net_ratelimit())
printk(KERN_INFO 
    "Oversized IP packet from %d.%d.%d.%d.\n", 
    NIPQUAD(qp->saddr));
out_fail:
IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
return NULL;
```
Removal due to time out

The ip_expire() function is the timer exit that is invoked when a queue has existed for 30 seconds.

```c
274 /*
275 * Oops, a fragment queue timed out. Kill it and send an
276 * ICMP reply.
277 */
278 static void ip_expire(unsigned long arg)
279 {
280    struct ipq *qp = (struct ipq *) arg;
281    spin_lock(&qp->lock);
282    if (qp->last_in & COMPLETE)
283       goto out;
284    ipq_kill(qp);
285    IP_INC_STATS_BH(IPSTATS_MIB_REASMTIMEOUT);
286    if ((qp->last_in&FIRST_IN) && qp->fragments != NULL) {
287       struct sk_buff *head = qp->fragments;
288       /* Send an ICMP "Fragment Reassembly Timeout" message. */
289       if ((head->dev = dev_get_by_index(qp->iif)) != NULL) {
290          icmp_send(head, ICMP_TIME_EXCEEDED,
291                      ICMP_EXC_FRAGTIME, 0);
292          dev_put(head->dev);
293       }
294    }
295 out:
296    spin_unlock(&qp->lock);
297    ipq_put(qp, NULL);
```
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`. Because a common DoS trick is to flood a system with fragments that can't be reassembled this is an important task.

```c
/* Memory limiting on fragments. Evictor trashes the oldest fragment queue until we are back under the low threshold. */

243 static void ip_evictor(void)
244 {
245    struct ipq *qp;
246    struct list_head *tmp;
247    int work;
248
249    work = atomic_read(&ip_frag_mem) -
250         sysctl_ipfrag_low_thresh;
251    if (work <= 0)
252        return;
```
The old FIFO Method

In this inner loop the oldest ipq on each hash queue, which is necessarily the last one on the queue is identified and freed. There are all manner of possible unfairness issues here, one of which is described below. Fortunately fragmentation is extremely rare.

```c
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               read_lock(&ipfrag_lock);
217               if ((qp = ipq_hash[i]) != NULL) {
218                   /* find the oldest queue for this 
hash bucket */
219                   while (qp->next)
220                      qp = qp->next;
221                   atomic_inc(&qp->refcnt);
222                   read_unlock(&ipfrag_lock);
223                   spin_lock(&qp->lock);
224                   if (!(qp->last_in & COMPLETE))
225                      ipq_kill(qp);
226                   spin_unlock(&qp->lock);
227               if (!((qp->last_in & COMPLETE))
228                   ipq_kill(qp);
229                   spin_unlock(&qp->lock);
```
Now call `ipq_put` to destroy it.

```c
ipq_put(qp);
IP_INC_STATS_BH(IpReasmFails);
progress = 1;
continue;
```

The new LRU method

```c
while (work > 0) {
    read_lock(&ipfrag_lock);
    if (list_empty(&ipq_lru_list)) {
        read_unlock(&ipfrag_lock);
        return;
    }
    tmp = ipq_lru_list.next;
    qp = list_entry(tmp, struct ipq, lru_list);
    atomic_inc(&qp->refcnt);
    read_unlock(&ipfrag_lock);
    spin_lock(&qp->lock);
    if (!(qp->last_in&COMPLETE))
        ipq_kill(qp);
    spin_unlock(&qp->lock);
    ipq_put(qp, &work);
    IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
}
```
Unlinking the ipq from its hash queue

The ipq_kill() function is defined in net/ipv4/ip_fragment.c. It deletes the timer associated with the queue and ensures that the ipq structure is unlinked from its hash chain. This makes clear the increment of refcnt that occurred when the timer was armed. The timer queue element does indeed hold a reference to the ipq that must be accounted for and del_timer returns 1 if the timer was armed.

225 /* Kill ipq entry. It is not destroyed immediately, 226 * because caller (and someone more) holds reference count. 227 */
228 static void ipq_kill(struct ipq *ipq)
229 {
230    if (del_timer(&ipq->timer))
231        atomic_dec(&ipq->refcnt);
232
233    if (!(ipq->last_in & COMPLETE)) {
234        ipq_unlink(ipq);
235        atomic_dec(&ipq->refcnt);
236        ipq->last_in |= COMPLETE;
237    }
238 }

The ipq_unlink() function calls __ipq_unlink() after obtaining the global write lock.

119 static __inline__ void ipq_unlink(struct ipq *ipq)
120 {
121    write_lock(&ipfrag_lock);
122    __ipq_unlink(ipq);
123    write_unlock(&ipfrag_lock);
124 }

The __ipq_unlink() function does the real work of removing the queue from its hash chain.

112 static __inline__ void __ipq_unlink(struct ipq *qp)
113 {
114    hlist_del(&qp->list);
115    list_del(&qp->lru_list);
116    ip_frag_nqueues--;
117 }

41
Destroying the ipq

After unlinking the queue, the ipq_put() function is called to free all fragments in the queue via the ip_frag_destroy() function when queue's reference count becomes zero.

```
219 static __inline__ void ipq_put(struct ipq *ipq, int *work)  
220 { 
221    if (atomic_dec_and_test(&ipq->refcnt))  
222       ip_frag_destroy(ipq, work);  
223 }  
```

The ip_frag_destroy() function does the real work of freeing fragments in the queue.

```
195 /* Complete destruction of ipq. */  
196 static void ip_frag_destroy(struct ipq *qp, int *work)  
197 { 
198    struct sk_buff *fp;  
199    BUG_TRAP(qp->last_in & COMPLETE);  
200    BUG_TRAP(del_timer(&qp->timer) == 0);  
201    if (qp->peer)  
202       inet_putpeer(qp->peer);  
203    /* Release all fragment data. */  
204    fp = qp->fragments;  
205    while (fp) {  
206       struct sk_buff *xp = fp->next;  
207       frag_kfree_skb(fp, work);  
208       fp = xp;  
209    }  
210    /* Finally, release the queue descriptor itself. */  
211    frag_free_queue(qp, work);  
212 }  
```
Finally, the queue descriptor is freed and `ip_frag_mem` is decremented by its size.

```
174  static __inline__ void frag_free_queue(struct ipq *qp,
                                  int *work)
175  {
176    if (work)
177        *work -= sizeof(struct ipq);
178    atomic_sub(sizeof(struct ipq), &ip_frag_mem);
179    kfree(qp);
180  }
```