Sheet 5

Forwarding intro

ip_rcv

Right, so now we're going to look at routing. We start with net/ipv4/ip_input.c::ip_rcv, which is the main IP receive routine, and which needs to make a decision as to whether to deal with this packet locally or forward it.

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)
{
    struct iphdr *iph = skb->nh.iph;

```

There are several stages here. First, we're testing to see whether what we have is well formed.

```
    /* When the interface is in promisc. mode, drop all the crap
     * that it receives, do not try to analyse it.
     */
    if (skb->pkt_type == PACKET_OTHERHOST)
        goto drop;

    IP_INC_STATS_BH(IpInReceives);

    if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL)
        goto out;

    if (!pskb_may_pull(skb, sizeof(struct iphdr)))
        goto inhdr_error;

    iph = skb->nh.iph;
```

In particular, we require that

1. **Length at least the size of an ip header**
2. **Version of 4**
3. **Checksums correctly. [Speed optimisation for later, skip loopback checksums]**
4. **Doesn't have a bogus length**

Also, RFC1122: 3.1.2.2 MUST silently discard any IP frame that fails the checksum.

```
    if (iph->ihl < 5 || iph->version != 4)
        goto inhdr_error;

    if (!pskb_may_pull(skb, iph->ihl*4))
        goto inhdr_error;

    if (ip_fast_csum((u8 *)iph, iph->ihl) != 0)
        goto inhdr_error;

    {
        __u32 len = ntohs(iph->tot_len);
        if (skb->len < len || len < (iph->ihl<<2))
            goto inhdr_error;

```

Our transport medium may have padded the buffer out. Now we know it is IP we can trim to the true length of the frame. Note this now means skb->len holds ntohs(iph->tot_len).

```
        if (skb->len > len) {
            __pskb_trim(skb, len);
            if (skb->ip_summed == CHECKSUM_HW)
                skb->ip_summed = CHECKSUM_NONE;
        }
    }
```

AOK, so the work carries on in net/ipv4/ip_input.c::ip_rcv_finish, described below after netfiltering

```
    return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL, ip_rcv_finish);
```

Otherwise, fail quietly.

```
inhdr_error:
    IP_INC_STATS_BH(IpInHdrErrors);
drop:
    kfree_skb(skb);
out:
    return NET_RX_DROP;
}
```

ip_rcv_finish

At this point, we know that we have an IP packet, but we haven't decided what to do with it.

```
static inline int ip_rcv_finish(struct sk_buff *skb)
{
    struct net_device *dev = skb->dev;
    struct iphdr *iph = skb->nh.iph;

```

If we don't already know what to do with the packet, call net/ipv4/route.c::ip_route_input. This determines whether we should forward this on another interface or whether it is for us. We'll talk about this later.

```
    if (skb->dst == NULL) {
        if (ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, dev))
            goto drop;
    }

```

<deal with IP options>

This is either call net/ipv4/ip_forward.c::ip_forward, or local delivery. We'll deal with forwarding below.

```
    return skb->dst->input(skb);

inhdr_error:
    IP_INC_STATS_BH(IpInHdrErrors);
drop:
    kfree_skb(skb);
    return NET_RX_DROP;
}

```

ip_forward

We’ve decided that we need to forward the packet.

```
int ip_forward(struct sk_buff *skb)
{
    struct net_device     *dev2;   /* Output device */
    struct iphdr          *iph;    /* Our header */
    struct rtable         *rt;     /* Route we use */
    struct ip_options     *opt = &(IPCB(skb)->opt);
    unsigned short        mtu;

```

Housekeeping

```
    if (IPCB(skb)->opt.router_alert && ip_call_ra_chain(skb))
        return NET_RX_SUCCESS;

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

    skb->ip_summed = CHECKSUM_NONE;

    iph = skb->nh.iph;
    rt = (struct rtable*)skb->dst;

```

According to the RFC, we must first decrease the TTL field. If that reaches zero, we must reply an ICMP control message telling that the packet's lifetime expired.

```
    if (iph->ttl <= 1)
        goto too_many_hops;

```

Fail if we're supposed to be routing strictly and we can't meet the requirements. Strict routing permits no gatewaying

```
    if (opt->is_strictroute && rt->rt_dst != rt->rt_gateway)
        goto sr_failed;

```

Having picked a route we can now send the frame out after asking the firewall permission to do so.

```
    skb->priority = rt_tos2priority(iph->tos);
    dev2 = rt->u.dst.dev;
    mtu  = rt->u.dst.pmtu;

    /*
     *  We now generate an ICMP HOST REDIRECT giving the route
     *  we calculated.
     */
    if (rt->rt_flags&RTCF_DOREDIRECT && !opt->srr)
        ip_rt_send_redirect(skb);

```

We are about to mangle packet. Copy it! (cow = copy on write)

```
    if (skb_cow(skb, dev2->hard_header_len))
        goto drop;
    iph = skb->nh.iph;
```

Decrease ttl after skb cow done

```
    ip_decrease_ttl(iph);
```

Check if we need to fragment, because MTU is lower than the length of this packet. If we do, and the 'don't fragment' flag is set, then generate error.

```
    if (skb->len > mtu && (ntohs(iph->frag_off) & IP_DF))
        goto frag_needed;
```

<NAT stuff>

Carry on with net/ipv4/ip_forward.c::ip_forward_finish after a netfilter.

```
    return NF_HOOK(PF_INET, NF_IP_FORWARD, skb, skb->dev, dev2,
                          ip_forward_finish);
```

Generate diagnostic ICMP messages for unroutable packets

```
frag_needed:
    IP_INC_STATS_BH(IpFragFails);
    icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED, htonl(mtu));
    goto drop;

sr_failed:
    icmp_send(skb, ICMP_DEST_UNREACH, ICMP_SR_FAILED, 0);
    goto drop;
```

```

too_many_hops:
    icmp_send(skb, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL, 0);
drop:
    kfree_skb(skb);
    return NET_RX_DROP;
}

```

ip_forward_finish

Get the packet on its way.

```
static inline int ip_forward_finish(struct sk_buff *skb)
{
    struct ip_options * opt   = &(IPCB(skb)->opt);

    IP_INC_STATS_BH(IpForwDatagrams);

```

If we haven’t got any options, then come in here.

```
    if (opt->optlen == 0) {
```

<deal with fast routing>

Send it

```
        return (ip_send(skb));
    }

```

Deal with options.

```
    ip_forward_options(skb);
```

And send it.

```
    return (ip_send(skb));
```

```
}

```

ip_route_input

OK, so how did we decide whether to forward or deliver locally? Look back and you'll see that we called ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, dev) in net/ipv4/ip_input::ip_rcv_finish.

One important thing to note here is that we have two structures we use for routing. The first is a transient route cache, held in a hash table as below; the second is the permanent FIB (forwarding information base). The FIB is only consulted if we don't have a matching hash table entry.

```
int ip_route_input(struct sk_buff *skb, u32 daddr, u32 saddr, u8 tos, struct
net_device *dev)
{
    struct rtable   *rth;
    unsigned         hash;
    int              iif = dev->ifindex;

```

Generate an index into the routing table, based on the destination address, sending address, input interface and tos fields.

```
    tos &= IPTOS_RT_MASK;
    hash = rt_hash_code(daddr, saddr ^ (iif << 5), tos);

```

Now chain down that entry looking for something that matches

```
    read_lock(&rt_hash_table[hash].lock);
    for (rth = rt_hash_table[hash].chain; rth; rth = rth->u.rt_next) {
        if (rth->key.dst == daddr &&
            rth->key.src == saddr &&
            rth->key.iif == iif   &&
            rth->key.oif == 0     &&
#ifdef CONFIG_IP_ROUTE_FWMARK
            rth->key.fwmark == skb->nfmark &&
#endif
            rth->key.tos == tos) {
```

We found something. Update time and usage stats

```
             rth->u.dst.lastuse = jiffies;
             dst_hold(&rth->u.dst);
             rth->u.dst.__use++;
             rt_cache_stat[smp_processor_id()].in_hit++;
             read_unlock(&rt_hash_table[hash].lock);
```

Set the destination for this packet

```
             skb->dst = (struct dst_entry*)rth;
```

And return success

```
             return 0;
        }
    }
    read_unlock(&rt_hash_table[hash].lock);
```

<Deal with multicast traffic>

We need to look a bit harder.

```
    return ip_route_input_slow(skb, daddr, saddr, tos, dev);
}

```

ip_route_input_slow

The route we wanted was not in the route cache, so we need to consult the FIB. The FIB data is put in place by a routing algorithm, and we'll look at that later. What we care about is looking up the FIB at present. This is complicated by the need to do 'longest match' for CIDR addresses. See slides.

Take a look in the files include/net/ip_fib.h and net/ipv4/[fib_frontend.c, fib_hash.c, fib_rules.c, fib_semantics.c]

```
/*
 *   NOTE. We drop all the packets that have local source addresses, because every
 *   properly looped back packet must have correct destination already attached by
 *   output routine.
 *
 *   Such an approach solves two big problems:
 *   1. Not simplex devices are handled properly.
 *   2. IP spoofing attempts are filtered with 100% of guarantee.
 */

int ip_route_input_slow(struct sk_buff *skb, u32 daddr, u32 saddr, u8 tos, struct
net_device *dev)
{
    struct rt_key         key;
    struct fib_result     res;
    struct in_device    *in_dev   = in_dev_get(dev);
    struct in_device    *out_dev  = NULL;
    unsigned              flags    = 0;
    u32                   itag     = 0;
    struct rtable        *rth;
    unsigned              hash;
    u32                   spec_dst;
    int                   err      = -EINVAL;
    int                   free_res = 0;

```

Check to see if IP on this device is disabled.

```
    if (!in_dev)
        goto out;

```

Set up lookup key.

```
    key.dst      = daddr;
    key.src      = saddr;
    key.tos      = tos;
#ifdef CONFIG_IP_ROUTE_FWMARK
    key.fwmark   = skb->nfmark;
#endif
    key.iif      = dev->ifindex;
    key.oif      = 0;
    key.scope    = RT_SCOPE_UNIVERSE;

    hash = rt_hash_code(daddr, saddr ^ (key.iif << 5), tos);

```

Check for the most weird ‘martians’, which can be not detected by fib_lookup.

```
    if (MULTICAST(saddr) || BADCLASS(saddr) || LOOPBACK(saddr))
        goto martian_source;

    if (daddr == 0xFFFFFFFF || (saddr == 0 && daddr == 0))
        goto brd_input;

    if (ZERONET(saddr))
        goto martian_source;

    if (BADCLASS(daddr) || ZERONET(daddr) || LOOPBACK(daddr))
        goto martian_destination;


```

This is the key statement – lookup route in the FIB. We'll talk more about this later.

```
    if ((err = fib_lookup(&key, &res)) != 0) {
        if (!IN_DEV_FORWARD(in_dev))
            goto e_inval;
        goto no_route;
    }
```

Mark res as needing to be deallocated. This gets done at label 'done:'

```
    free_res = 1;

    rt_cache_stat[smp_processor_id()].in_slow_tot++;

```

<NAT stuff removed>

Check for broadcast

```
    if (res.type == RTN_BROADCAST)
        goto brd_input;
```

Accept locally. But before we do, validate the source – check that it's not broadcast or our local address and that it arrived on the right physical interface.

```
    if (res.type == RTN_LOCAL) {
        int result;
        result = fib_validate_source(saddr, daddr, tos, loopback_dev.ifindex, dev,
                                     &spec_dst, &itag);
        if (result < 0)
            goto martian_source;
        if (result)
            flags |= RTCF_DIRECTSRC;
        spec_dst = daddr;
        goto local_input;
    }
```

Check for more errors Only go past this point if we have a unicast direct route.

```
    if (!IN_DEV_FORWARD(in_dev))
        goto e_inval;
    if (res.type != RTN_UNICAST)
        goto martian_destination;
```

<multipath stuff removed>

include/linux/inetdevice.h::in_dev_get increments a ref count and returns a include/linux/inetdevice.h::in_device

```
    out_dev = in_dev_get(FIB_RES_DEV(res));
    if (out_dev == NULL) {
        if (net_ratelimit())
            printk(KERN_CRIT "Bug in ip_route_input_slow(). Please, report\n");
        goto e_inval;
    }

```

Validate the source – check that it's not broadcast or our local address and that it arrived on the right physical interface. Also, calculate the logical interface this packet arrived on and calculate the 'specific destination' address.

```
    err = fib_validate_source(saddr, daddr, tos, FIB_RES_OIF(res), dev,
                              &spec_dst, &itag);
    if (err < 0)
        goto martian_source;

    if (err)
        flags |= RTCF_DIRECTSRC;

    if (out_dev == in_dev && err && !(flags & (RTCF_NAT | RTCF_MASQ)) &&
            (IN_DEV_SHARED_MEDIA(out_dev) ||
             inet_addr_onlink(out_dev, saddr, FIB_RES_GW(res))))
        flags |= RTCF_DOREDIRECT;

```

Not IP (i.e. ARP). Do not create route, if it is invalid for proxy arp. Dynamic NAT (DNAT) routes are always valid.

```
    if (skb->protocol != __constant_htons(ETH_P_IP)) {
        if (out_dev == in_dev && !(flags & RTCF_DNAT))
            goto e_inval;
    }

```

Set up cache entry.

```
    rth = dst_alloc(&ipv4_dst_ops);
    if (!rth)
        goto e_nobufs;

    atomic_set(&rth->u.dst.__refcnt, 1);
    rth->u.dst.flags  = DST_HOST;
    rth->key.dst      = daddr;
    rth->rt_dst       = daddr;
    rth->key.tos      = tos;
#ifdef CONFIG_IP_ROUTE_FWMARK
    rth->key.fwmark   = skb->nfmark;
#endif
    rth->key.src      = saddr;
    rth->rt_src       = saddr;
    rth->rt_gateway   = daddr;

```

<NAT stuff removed>

```
    rth->rt_iif      =
    rth->key.iif     = dev->ifindex;
    rth->u.dst.dev   = out_dev->dev;
    dev_hold(rth->u.dst.dev);
    rth->key.oif     = 0;
    rth->rt_spec_dst = spec_dst;

    rth->u.dst.input  = ip_forward;
    rth->u.dst.output = ip_output;

    rt_set_nexthop(rth, &res, itag);

    rth->rt_flags    = flags;

```

<FASTROUTE stuff removed>

```
intern:
```

Insert entry into the route cache

```
    err = rt_intern_hash(hash, rth, (struct rtable**)&skb->dst);

done:
```

Reduce ref counts

```
    in_dev_put(in_dev);
    if (out_dev)
        in_dev_put(out_dev);
    if (free_res)
        fib_res_put(&res);
out: return err;

brd_input:
```
<Deal with broadcast>

```
local_input:
```
Come here if we are willing to accept locally.
```
    rth = dst_alloc(&ipv4_dst_ops);
    if (!rth)
        goto e_nobufs;

    rth->u.dst.output= ip_rt_bug;

    atomic_set(&rth->u.dst.__refcnt, 1);
    rth->u.dst.flags  = DST_HOST;
    rth->key.dst      = daddr;
    rth->rt_dst       = daddr;
    rth->key.tos      = tos;
#ifdef CONFIG_IP_ROUTE_FWMARK
    rth->key.fwmark   = skb->nfmark;
#endif
    rth->key.src      = saddr;
    rth->rt_src       = saddr;
```

```
#ifdef CONFIG_IP_ROUTE_NAT
    rth->rt_dst_map   = key.dst;
    rth->rt_src_map   = key.src;
#endif
#ifdef CONFIG_NET_CLS_ROUTE
    rth->u.dst.tclassid = itag;
#endif
    rth->rt_iif       =
    rth->key.iif      = dev->ifindex;
    rth->u.dst.dev    = &loopback_dev;
    dev_hold(rth->u.dst.dev);
    rth->key.oif      = 0;
    rth->rt_gateway   = daddr;
    rth->rt_spec_dst  = spec_dst;
    rth->u.dst.input  = ip_local_deliver;
    rth->rt_flags     = flags|RTCF_LOCAL;
    if (res.type == RTN_UNREACHABLE) {
        rth->u.dst.input   = ip_error;
        rth->u.dst.error   = -err;
        rth->rt_flags     &= ~RTCF_LOCAL;
    }
    rth->rt_type = res.type;
    goto intern;

no_route:
    rt_cache_stat[smp_processor_id()].in_no_route++;
    spec_dst = inet_select_addr(dev, 0, RT_SCOPE_UNIVERSE);
    res.type = RTN_UNREACHABLE;
    goto local_input;

martian_destination:
    **<Generate error>**

e_inval:
    err = -EINVAL;
    goto done;

e_nobufs:
    err = -ENOBUFS;
    goto done;

martian_source:
    **<Generate error>**
    goto e_inval;
}

```