Linux Kernel Networking

Rami Rosen

ramirose@gmail.com

Haifux, August 2007

Disclaimer

Everything in this lecture shall not, under any circumstances, hold any legal liability whatsoever. Any usage of the data and information in this document shall be solely on the responsibility of the user. This lecture is not given on behalf of any company or organization.

Warning

- This lecture will deal with design functional

description side by side with many implementation details;

some knowledge of “C” is preferred.

General

- The Linux networking kernel code (including network device drivers) is a large part of the Linux kernel code.
- **Scope**: We will not deal with wireless, IPv6, and multicasting.
  - Also not with user space routing daemons/apps, and with security attacks (like DoS, spoofing, etc.) .
- Understanding a packet walkthrough in the kernel is a key to understanding kernel networking. Understanding it is a must if

  we want to understand Netfilter or IPSec internals, and more.
- There is a 10 pages Linux kernel networking walkthrouh document

General - Contd.

- Though it deals with 2.4.20 Linux kernel, most of it is relevant.
- This lecture will concentrate on this walkthrough (design and implementation details).
- References to code in this lecture are based on linux-2.6.23-rc2.
- There was some serious cleanup in 2.6.23

Hierarchy of networking layers

- The layers that we will deal with (based on the 7 layers model) are:

Transport Layer (L4) (udp,tcp...)

Network Layer (L3) (ip)

Link Layer (L2) (ethernet)

Networking Data Structures

- The two most important structures of linux kernel network layer are:
  - sk_buff (defined in *include/linux/skbuff.h*)
  - netdevice (defined in *include/linux/netdevice.h*)
- It is better to know a bit about them before delving into the walkthrough code.

SK_BUFF

- sk_buff represents data and headers.
- sk_buff API (examples)
  - sk_buff allocation is done with *alloc_skb()* or *dev_alloc_skb();* drivers use *dev_alloc_skb();*. (free by *kfree_skb()* and *dev_kfree_skb()*.
- *unsigned char* data* : points to the current header.
- *skb_pull(int len)* – removes data from the start of a buffer by advancing data to data+len and by decreasing len.
- Almost always sk_buff instances appear as “skb” in the kernel code.

SK_BUFF - contd

- sk_buff includes 3 unions; each corresponds to a kernel network layer:
- **transport_header** (previously called h) – for layer 4, the transport layer (can include tcp header or udp header or icmp header, and more)
- **network_header** – (previously called nh) for layer 3, the network layer (can include ip header or ipv6 header or arp header).
- **mac_header** – (previously called mac) for layer 2, the link layer.
- skb_network_header(skb), skb_transport_header(skb) and skb_mac_header(skb) return pointer to the header.

SK_BUFF - contd.

- **struct dst_entry *dst** – the route for this sk_buff; this route is determined by the routing subsystem.
  - It has 2 important function pointers:
    - *int (*input)(struct sk_buff*);*
    - *int (*output)(struct sk_buff*);*

int (*input)(struct sk_buff *);
int (*output)(struct net *net, struct sock *sk, struct sk_buff *skb);

- ***input()*** can be assigned to one of the following : ip_local_deliver, ip_forward, ip_mr_input, ip_error or dst_discard_in.
- **output()** can be assigned to one of the following :ip_output, ip_mc_output, ip_rt_bug, or dst_discard_out.
  - we will deal more with dst when talking about routing.

SK_BUFF - contd.

- In the usual case, there is only one dst_entry for every skb.
- When using IPSec, there is a linked list of dst_entries and only the last one is for routing; all other dst_entries are for IPSec transformers ; these other dst_entries have the DST_NOHASH flag set.
- **tstamp** (of type ktime_t) : time stamp of receiving the packet.
  - *net_enable_timestamp()* must be called in order to get values.

net_device

- net_device represents a network interface card.
- There are cases when we work with virtual devices.
  - For example, bonding (setting the same IP for two or more NICs, for load balancing and for high availability.)
  - Many times this is implemented using the private data of the device (the **void *priv** member of net_device);
  - In OpenSolaris there is a special pseudo driver called “vnic” which enables bandwidth allocation (project CrossBow).
- Important members:

net_device - contd

- **unsigned int mtu** – Maximum Transmission Unit: the maximum size of frame the device can handle.
- Each protocol has mtu of its own; the default is **1500** for Ethernet.
- you can change the mtu with ifconfig; for example,like this:
  - *ifconfig eth0 mtu 1400*
  - You cannot of course, change it to values higher than 1500 on 10Mb/s network:
  - *ifconfig eth0 mtu 1501* will give:
  - *SIOCSIFMTU: Invalid argument*

net_device - contd

- **unsigned int flags -** (which you see or set using ifconfig utility): for example, RUNNING or NOARP.
- **unsigned char dev_addr[MAX_ADDR_LEN]** : the MAC address of the device (6 bytes).
- **int (*hard_start_xmit)(struct sk_buff *skb,**

  struct net_device *dev);
  - a pointer to the device transmit method.
- **int promiscuity;** (a counter of the times a NIC is told to set to work in promiscuous mode; used to enable more than one sniffing client.)

net_device - contd

- You are likely to encounter macros starting with IN_DEV like:

IN_DEV_FORWARD() or IN_DEV_RX_REDIRECTS(). How are the related to net_device ? How are these macros implemented ?

- **void *ip_ptr**: IPv4 specific data. This pointer is assigned to a pointer to in_device in *inetdev_init()* (*net/ipv4/devinet.c*)

net_device - Contd.

- struct in_device have a member named cnf (instance of ipv4_devconf). Setting */proc/sys/net/ipv4/conf/all/forwarding*

  eventually sets the forwarding member of in_device to 1.

  The same is true to accept_redirects and send_redirects; both

  are also members of cnf (ipv4_devconf).
- In most distros, */proc/sys/net/ipv4/conf/all/forwarding=0*
- *But probably this is not so on your ADSL router.*

network interface drivers

- Most of the nics are PCI devices; there are also some USB network devices.
- The drivers for network PCI devices use the generic PCI calls, like *pci_register_driver()* and *pci_enable_device().*
- For more info on nic drives see the article "**Writing Network Device Driver for Linux**" (link no. 9 in links) and chap17 in **ldd3**.
- There are two modes in which a NIC can receive a packet.
  - The traditional way is interrupt-driven : each received packet is an asynchronous event which causes an interrupt.

NAPI

- NAPI (new API).
  - The NIC works in polling mode.
  - In order that the nic will work in polling mode it should be built with a proper flag.
  - Most of the new drivers support this feature.
  - When working with NAPI and when there is a very high load,

    packets are lost; but this occurs before they are fed into the network stack. (in the non-NAPI driver they pass into the stack)
  - in Solaris, polling is built into the kernel (no need to build drivers in any special way).

User Space Tools

- iputils (including ping, arping, and more)
- net-tools (ifconfig, netstat, , route, arp and more)
- IPROUTE2 (ip command with many options)
  - Uses rtnetlink API.
  - Has much wider functionalities; for example, you can create tunnels with “ip” command.
  - Note: no need for “-n” flag when using IPROUTE2 (because it does not work with DNS).

Routing Subsystem

- The routing table and the routing cache enable us to find the net device and the address of the host to which a packet will be sent.
- Reading entries in the routing table is done by calling *fib_lookup(const struct flowi *flp, struct fib_result *res)*
- FIB is the “Forwarding Information Base”.
- There are two routing tables by default: (non Policy Routing case)
  - local FIB table (*ip_fib_local_table* ; ID 255).
  - main FIB table (*ip_fib_main_table* ; ID 254)
  - *See : include/net/ip_fib.h.*

Routing Subsystem - contd.

- Routes can be added into the main routing table in one of 3 ways:
  - By sys admin command (route add/ip route).
  - By routing daemons.
  - As a result of ICMP (REDIRECT).
- A routing table is implemented by struct fib_table.

Routing Tables

- *fib_lookup()* first searches the local FIB table (ip_fib_local_table).
- In case it does not find an entry, it looks in the main FIB table (ip_fib_main_table).
- Why is it in this order ?
- There is one routing cache, regardless of how many routing tables there are.
- You can see the routing cache by running *"route -C".*
- Alternatively, you can see it by : "*cat /proc/net/rt_cache*".
  - con: this way, the addresses are in hex format

Routing Cache

- The routing cache is built of **rtable** elements:
- struct rtable (see: */include/net/route.h*)

```
{
union {
        struct dst_entry dst;
        } u;
...
}
```

Routing Cache - contd

- The **dst_entry** is the protocol-independent part.
  - Thus, for example, we have a dst_entry member (also called dst) in rt6_info in ipv6. (*include/net/ip6_fib.h*)
- The key for a lookup operation in the routing cache is an IP address (whereas in the routing table the key is a subnet).
- Inserting elements into the routing cache by : *rt_intern_hash()*
- There is an alternate mechanism for route cache lookup, called **fib_trie**, which is inside the kernel tree (*net/ipv4/fib_trie.c)*

Routing Cache - contd

- It is based on extending the lookup key.
- You should set: CONFIG_IP_FIB_TRIE (=y)
  - (instead of CONFIG_IP_FIB_HASH)
- By Robert Olsson et al (see links).

Creating a Routing Cache Entry

- Allocation of **rtable** instance (rth) is done by: *dst_alloc()*.
  - dst_alloc() in fact creates and returns a pointer to dst_entry and we cast it to rtable *(net/core/dst.c).*
- Setting input and output methods of dst:
  - (rth->u.dst.input and rth->u.dst.input)
- Setting the flowi member of dst (rth->fl)
  - Next time there is a lookup in the cache,for example , *ip_route_input()*, we will compare against rth->fl.

Routing Cache - Contd.

- A garbage collection call which delete eligible entries from the routing cache.
- Which entries are not eligible ?

Policy Routing (multiple tables)

- Generic routing uses destination-address based decisions.
- There are cases when the destination-address is not the sole parameter to decide which route to give; Policy Routing comes to enable this.

Policy Routing (multiple tables)-contd.

- Adding a routing table : by adding a line to: */etc/iproute2/rt_tables.*
  - For example: add the line “252 my_rt_table”.
  - There can be up to 255 routing tables.
- Policy routing should be enabled when building the kernel (CONFIG_IP_MULTIPLE_TABLES should be set.)
- Example of adding a route in this table:
- > ip route add default via 192.168.0.1 table my_rt_table
- Show the table by:
  - ip route show table my_rt_table

Policy Routing (multiple tables)-contd.

- You can add a rule to the **routing policy database (*RPDB*)** by "*ip rule add* ..."
  - The rule can be based on input interface, TOS, fwmark (from netfilter).
- *ip rule list* – show all rules.

Policy Routing: add/delete a rule - example

- *ip rule add tos 0x04 table 252*
  - This will cause packets with tos=0x08 (in the iphdr)

    to be routed by looking into the table we added (252)
  - So the default gw for these type of packets will be 192.168.0.1
  - ***ip rule show*** will give*:*
  - 32765: from all tos reliability lookup my_rt_table
  - ...

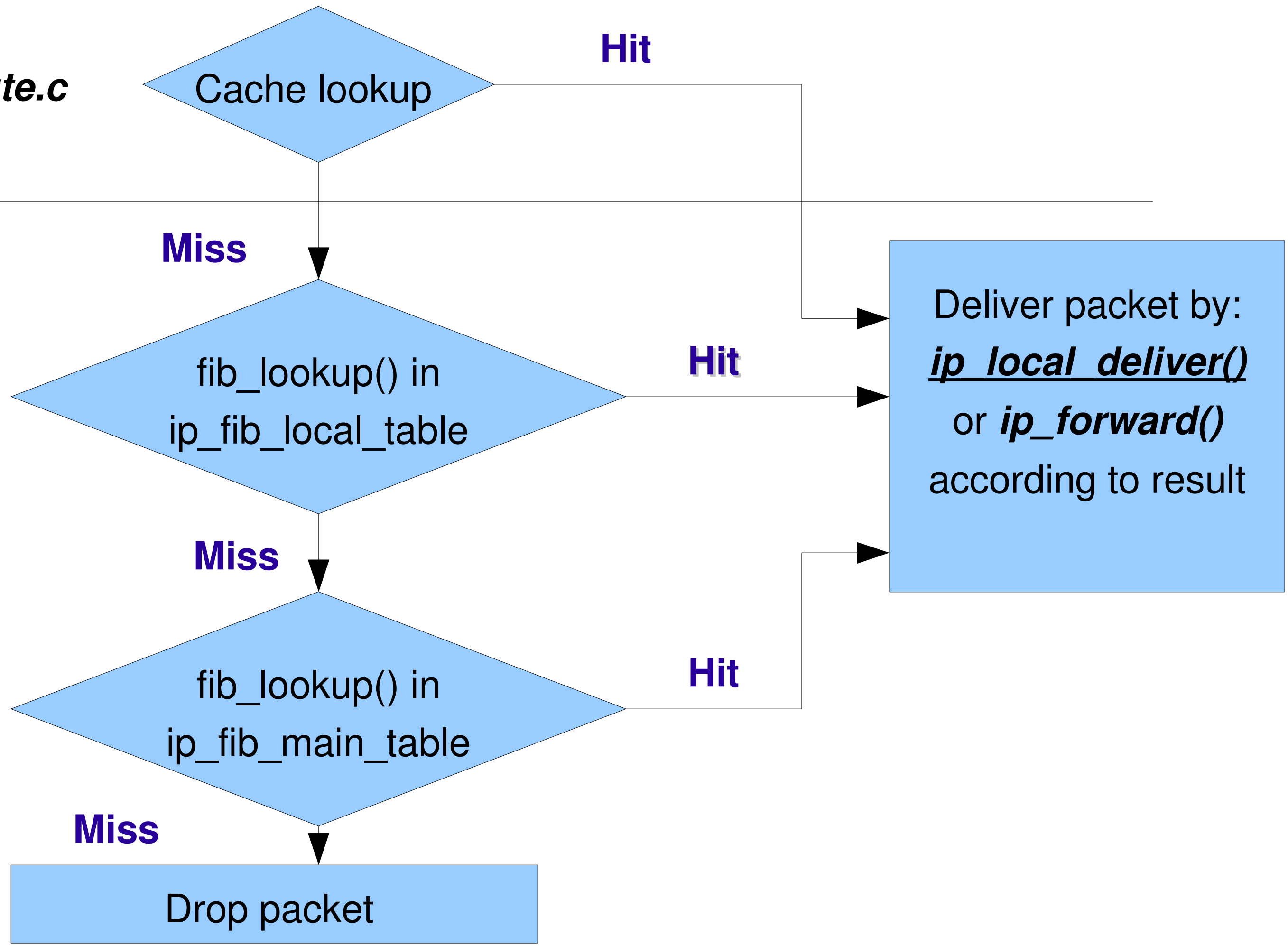
Policy Routing: add/delete a rule - example

- Delete a rule *: ip rule del tos 0x04 table 252*

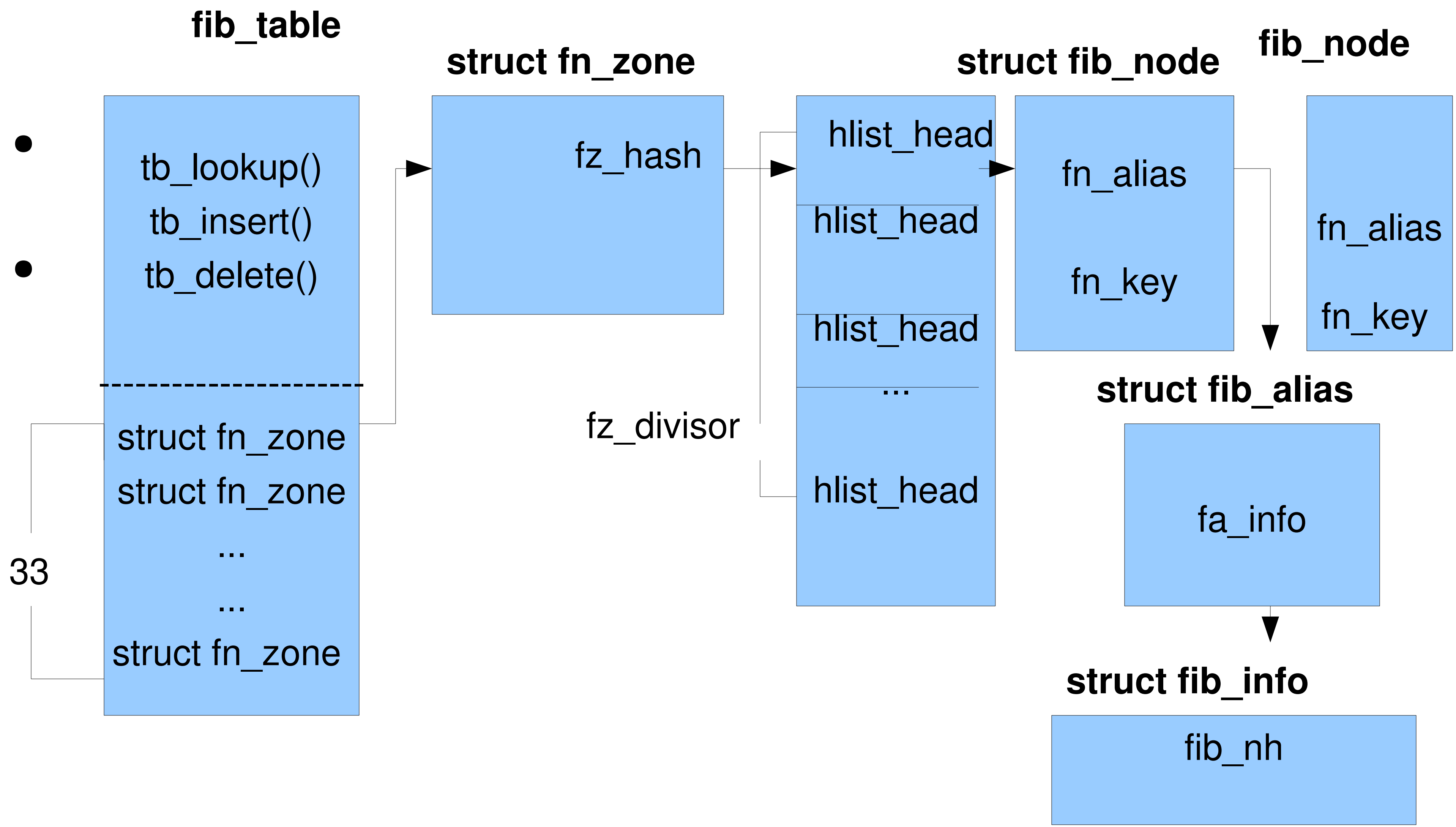
Routing Lookup

ip_route_input() in: net/ipv4/route.c

ip_route_input_slow()
in: net/ipv4/route.c



Routing Table Diagram



Routing Tables

- Breaking the fib_table into multiple data structures gives flexibility and enables fine grained and high level of sharing.
  - Suppose that we 10 routes to 10 different networks have the same next hop gw.
  - We can have one fib_info which will be shared by 10 fib_aliases.
  - fz_divisor is the number of buckets

Routing Tables - contd

- Each ***fib_ node*** element represents a unique subnet.
  - The ***fn_key*** member of fib_ node is the subnet (32 bit)

Routing Tables - contd

- Suppose that a device goes down or enabled.
- We need to disable/enable all routes which use this device.
- But how can we know which routes use this device ?
- In order to know it efficiently, there is the **fib_info_devhash** table.
- This table is indexed by the device identifier.
- See *fib_sync_down()* and *fib_sync_up()* in

  net/ipv4/fib_semantics.c

Routing Table lookup algorithm

- **LPM (Longest Prefix Match)** is the lookup algorithm.
- The route with the longest netmask is the one chosen.
- Netmask 0, which is the shortest netmask, is for the default gateway.
  - What happens when there are multiple entries with netmask=0?
  - *fib_lookup()* returns the **first entry it finds** in the fib table where netmask length is 0.

Routing Table lookup - contd.

- It may be that this is not the best choice default gateway.
- So in case that netmask is 0 (prefixlen of the fib_result returned from fib_look is 0) we call *fib_select_default()*.
- *fib_select_default()* will select the route with the lowest priority

  (metric) (by comparing to *fib_priority* values of all default gateways).

Receiving a packet

- When working in interrupt-driven model, the nic registers an interrupt handler with the IRQ with which the device works by calling *request_irq()*.
- This interrupt handler will be called when a frame is received
- The same interrupt handler will be called when transmission of a frame is finished and under other conditions. (depends on the NIC; sometimes, the interrupt handler will be called when there is some error).

Receiving a packet - contd

- Typically in the handler, we allocate sk_buff by calling *dev_alloc_skb()* ; also *eth_type_trans()* is called; among other things it advances the data pointer of the sk_buff to point to the IP header ; this is done by calling *skb_pull(skb, ETH_HLEN).*
- See : *net/ethernet/eth.c*
  - ETH_HLEN is 14, the size of ethernet header.

Receiving a packet - contd

- The handler for receiving a packet is *ip_rcv()*. (*net/ipv4/ip_input.c*)
- Handler for the protocols are registered at init phase.
  - Likewise, *arp_rcv()* is the handler for ARP packets.
- First, *ip_rcv()* performs some sanity checks. For example:

  if (iph->ihl < 5 || iph->version != 4)

  goto inhdr_error;

  - *iph* is the ip header ; iph->ihl is the ip header length (4 bits).
  - The ip header must be at least 20 bytes.
  - It can be up to 60 bytes (when we use ip options)

Receiving a packet - contd

- Then it calls *ip_rcv_finish*(), by:

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

- This division of methods into two stages (where the second has the same name with the suffix finish or slow, is typical for networking kernel code.)
- In many cases the second method has a “slow” suffix instead of “finish”; this usually happens when the first method looks in some cache and the second method performs a lookup in a table, which is slower.

Receiving a packet - contd

- *ip_rcv_finish()* implementation:

```
if (skb->dst == NULL) {
  int err = ip_route_input(skb, iph->daddr, iph->saddr, iph->tos,
                           skb->dev);
...
}
...
return dst_input(skb);
```

Receiving a packet - contd

- *ip_route_input()*:

  First performs a lookup in the routing cache to see if there is a match. If there is **no match (cache miss),** calls *ip_route_input_slow()* to perform a lookup in the routing table. (This lookup is done by calling *fib_lookup()*).

- *fib_lookup(const struct flowi *flp, struct fib_result *res)*

  The results are kept in fib_result.

- *ip_route_input()* returns 0 upon successful lookup. (also when there is a cache miss but a successful lookup in the routing table.)

Receiving a packet - contd

According to the results of *fib_lookup()*, we know if the frame is for **local delivery** or for **forwarding** or to be **dropped.**

- If the frame is for local delivery , we will set the input() function pointer of the route to *ip_local_deliver()*:

  rth->u.dst.input= ip_local_deliver;

- If the frame is to be forwarded, we will set the input() function pointer to *ip_forward()*:

  rth->u.dst.input = ip_forward;

Local Delivery

- Prototype:

  *ip_local_deliver(struct sk_buff *skb) (net/ipv4/ip_input.c).*

  - calls NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL,ip_local_deliver_finish);

- Delivers the packet to the higher protocol layers according to its type.

Forwarding

- Prototype:
  - *int ip_forward(struct sk_buff *skb)*
    - *(net/ipv4/ip_forward.c)*
  - *decreases the ttl in the ip header*
  - *If the ttl is <=1* , the methods send ICMP message *(ICMP_TIME_EXCEEDED)* and drops the packet.
  - Calls *NF_HOOK(PF_INET,NF_IP_FORWARD, skb, skb->dev, rt->u.dst.dev, ip_forward_finish);*

Forwarding- Contd

- *ip_forward_finish():* sends the packet out by calling *dst_output(skb)*.
- *dst_output(skb)* is just a wrapper, which calls

skb->dst->output(skb). (see *include/net/dst.h*)

Sending a Packet

- Handling of sending a packet is done by ***ip_route_output_key().***
- We need to perform routing lookup also in the case of transmission.
- In case of a cache miss, we calls *ip_route_output_slow()*, which looks in the routing table (by calling *fib_lookup()*, as also is done in ***ip_route_input_slow*****()**.)
- If the packet is for a remote host, we set dst->output to *ip_output()*

Sending a Packet-contd

- *ip_output()* will call *ip_finish_output()*
  - This is the NF_IP_POST_ROUTING point.
- *ip_finish_output()* will eventually send the packet from a neighbor by:
  - *dst->neighbour->output(skb)*
  - *arp_bind_neighbour()* sees to it that the L2 address of the next hop will be known*. (net/ipv4/arp.c)*

Sending a Packet - Contd.

- If the packet is for the local machine:
  - *dst->output = ip_output*
  - *dst->input   = ip_local_deliver*
  - *ip_output()* will send the packet on the loopback device,
  - Then we will go into *ip_rcv()* and *ip_rcv_finish()*, but this time dst is NOT null; so we will end in *ip_local_deliver()*.
- See: *net/ipv4/route.c*

Multipath routing

- This feature enables the administrator to set multiple next hops for a destination.
- To enable multipath routing, CONFIG_IP_ROUTE_MULTIPATH should be set when building the kernel.
- There was also an option for multipath caching: (by setting CONFIG_IP_ROUTE_MULTIPATH_CACHED).
- It was experimental and removed in 2.6.23 - See links (6).

mc - root@rr:/proc/net/stat

```
Linux Kernel v2.6.21-rc7 Configuration

                         Networking options
  Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters are
  hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes features.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ] excluded
  <M> module  < > module capable

        [ ] Network packet debugging
        <*> Packet socket
        [*]   Packet socket: mmapped IO
        <*> Unix domain sockets
        <*> Transformation user configuration interface
        [ ] Transformation sub policy support (EXPERIMENTAL)
        [ ] Transformation migrate database (EXPERIMENTAL)
        <M> PF_KEY sockets
        [ ]   PF_KEY MIGRATE (EXPERIMENTAL)
        [*] TCP/IP networking
        [*]   IP: multicasting
        [*]   IP: advanced router
                Choose IP: FIB lookup algorithm (choose FIB_HASH if unsure) (FIB_HASH)
        [*]   IP: policy routing
        [*]   IP: equal cost multipath
        [*]     IP: equal cost multipath with caching support (EXPERIMENTAL)
        <M>       MULTIPATH: round robin algorithm
        < >       MULTIPATH: random algorithm (NEW)
        < >       MULTIPATH: weighted random algorithm (NEW)
        < >       MULTIPATH: interface round robin algorithm (NEW)
        v(+)

                  <Select>    < Exit >    < Help >
```

Netfilter

- Netfilter is the kernel layer to support applying iptables rultes.
  - It enables:
    - Filtering
    - Changing packets (masquerading)
    - Connection Tracking

Netfilter rule - example

- Short example:
- Applying the following iptables rule:
  - iptables -A INPUT -p udp --dport 9999 -j DROP
- This is NF_IP_LOCAL_IN rule;
- The packet will go to:
- *ip_rcv()*
- and then: *ip_rcv_finish()*
- And then *ip_local_deliver()*

Netfilter rule - example (contd)

- but it will **NOT** proceed to *ip_local_deliver_finish()* as in the usual case, without this rule.
- As a result of applying this rule it reaches *nf_hook_slow()* with verdict == NF_DROP (calls *skb_free()* to free the packet)
- See */net/netfilter/core.c.*

ICMP redirect message

- ICMP protocol is used to notify about problems.
- A REDIRECT message is sent in case the route

  is suboptimal (inefficient).
- There are in fact 4 types of REDIRECT
- Only one is used :
  - Redirect Host (ICMP_REDIR_HOST)
- See **RFC 1812 (Requirements for IP Version 4 Routers).**

ICMP redirect message - contd.

- To support sending ICMP redirects, the machine should be configured to send redirect messages.
  - ***/proc/sys/net/ipv4/conf/all/send_redirects*** should be 1.
- In order that the other side will receive redirects, we should set

  /proc/sys/net/ipv4/conf/all/accept_redirects to 1.

ICMP redirect message - contd.

- Example:
- Add a suboptimal route on 192.168.0.31:
- route add -net 192.168.0.10 netmask 255.255.255.255 gw 192.168.0.121
- Running now “route” on 192.168.0.31 will show a new entry:

Destination	**Gateway**	Genmask	Flags	Metric	Ref	Use	Iface
192.168.0.10	**192.168.0.121**	255.255.255.255	UGH	0	0	0	eth0

ICMP redirect message - contd.

- Send packets from 192.168.0.31 to 192.168.0.10 :
- ping 192.168.0.10 (from 192.168.0.31)
- We will see (on 192.168.0.31):
  - From 192.168.0.121: icmp_seq=2 **Redirect Host(New nexthop: 192.168.0.10)**
- now, running on 192.168.0.121:
  - route -Cn | grep .10
- shows that there is a new entry in the routing cache:
-

ICMP redirect message - contd.

- 192.168.0.31 192.168.0.10 192.168.0.10 ri 0 0 34 eth0
- The “r” in the flags column means: RTCF_DOREDIRECT.
- The 192.168.0.121 machine had sent a redirect by calling *ip_rt_send_redirect()* from *ip_forward()*.

(net/ipv4/*ip_forward.c)*

ICMP redirect message - contd.

- And on 192.168.0.31, running “route -C | grep .10” shows now a new entry in the routing cache: (in case accept_redirects=1)
- 192.168.0.31 192.168.0.10 192.168.0.10 0 0 1 eth0
- In case accept_redirects=0 (on 192.168.0.31), we will see:
- 192.168.0.31 192.168.0.10 192.168.0.121 0 0 0 eth0
- which means that the gw is still 192.168.0.121 (which is the route that we added in the beginning).

ICMP redirect message - contd.

- Adding an entry to the routing cache as a result of getting ICMP REDIRECT is done in *ip_rt_redirect(), net/ipv4/route.c.*
- The entry in the routing table is not deleted.

Neighboring Subsystem

- Most known protocol: ARP (in IPV6: ND, neighbour discovery)
- ARP table.
- Ethernet header is 14 bytes long:
  - Source mac address (6 bytes).
  - Destination mac address (6 bytes).
  - Type (2 bytes).
    - 0x0800 is the type for IP packet (ETH_P_IP)
    - 0x0806 is the type for ARP packet (ETH_P_ARP)
    - see: *include/linux/if_ether.h*

Neighboring Subsystem - contd

- When there is no entry in the ARP cache for the destination IP address of a packet, a broadcast is sent (ARP request, `ARPOP_REQUEST`: who has IP address x.y.z...). This is done by a method called *arp_solicit()*. (*net/ipv4/arp.c*)
- You can see the contents of the arp table by running:

"*cat /proc/net/arp*" or by running the "arp" from a command line .

- You can delete and add entries to the arp table; see man arp.

Bridging Subsystem

- You can define a bridge and add NICs to it ("enslaving ports") using *brctl* (from bridge-utils).
- You can have up to 1024 ports for every bridge device (BR_MAX_PORTS) .
- Example:
- *brctl addbr mybr*
- *brctl addif mybr eth0*
- *brctl show*

Bridging Subsystem - contd.

- When a NIC is configured as a bridge port, the *br_port* member of net_device is initialized.
  - (br_port is an instance of *struct net_bridge_port*).
- When we receive a frame, *netif_receive_skb()* calls *handle_bridge().*

Bridging Subsystem - contd.

- The bridging forwarding database is searched for the destination MAC address.
- In case of a hit, the frame is sent to the bridge port with *br_forward()* *(net/bridge/br_forward.c).*
- If there is a miss, the frame is flooded on all bridge ports using *br_flood()* *(net/bridge/br_forward.c).*
- Note: this is not a broadcast !
- The ebtables mechanism is the L2 parallel of L3 Netfilter.

Bridging Subsystem- contd

- Ebtables enable us to filter and mangle packets

  at the link layer (L2).

IPSec

- Works at network IP layer (L3)
- Used in many forms of secured networks like VPNs.
- Mandatory in IPv6. (not in IPv4)
- Implemented in many operating systems: Linux, Solaris, Windows, and more.
- RFC2401
- In 2.6 kernel : implemented by Dave Miller and Alexey Kuznetsov.
- Transformation bundles.
- Chain of dst entries; only the last one is for routing.

IPSec-cont.

- User space tools: http://ipsec-tools.sf.net
- Building VPN : http://www.openswan.org/ (Open Source).
- There are also non IPSec solutions for VPN
  - example: pptp
- struct xfrm_policy has the following member:
  - struct dst_entry *bundles.
  - __xfrm4_bundle_create() creates dst_entries (with the DST_NOHASH flag) see: *net/ipv4/xfrm4_policy.c*
- Transport Mode and Tunnel Mode.

IPSec-contd.

- Show the security policies:
  - *ip xfrm policy show*
- Create RSA keys:
  - *ipsec rsasigkey --verbose 2048 > keys.txt*
  - *ipsec showhostkey --left   > left.publickey*
  - *ipsec showhostkey --right > right.publickey*

IPSec-contd.

Example: Host to Host VPN (using openswan)

in */etc/ipsec.conf:*

```
conn linux-to-linux
left=192.168.0.189
leftnexthop=%direct
leftrsasigkey=0sAQPPQ...
right=192.168.0.45
rightnexthop=%direct
rightrsasigkey=0sAQNwb...
type=tunnel
 auto=start
```

IPSec-contd.

- *service ipsec start* (to start the service)
- *ipsec verify* – Check your system to see if IPsec got installed and started correctly.
- *ipsec auto –status*
  - *If you see "IPsec SA established" , this implies success.*
- Look for errors in */var/log/secure* (fedora core) or in kernel syslog

Tips for hacking

- Documentation/networking/ip-sysctl.txt: networking kernel tunabels
- Example of reading a hex address:
- iph->daddr == 0x0A00A8C0 or

means checking if the address is 192.168.0.10 (C0=192,A8=168, 00=0,0A=10).

Tips for hacking - Contd.

- Disable ping reply:
- echo 1 >/proc/sys/net/ipv4/icmp_echo_ignore_all
- Disable arp: ***ip link set eth0 arp off*** (the NOARP flag will be set)
- Also ***ifconfig eth0 -arp*** has the same effect.
- How can you get the Path MTU to a destination (PMTU)?
  - Use tracepath (see man tracepath).
  - Tracepath is from iputils.

Tips for hacking - Contd.

- Keep iphdr struct handy (printout): (from linux/ip.h)

```
struct iphdr {

    __u8    ihl:4,
      version:4;
      __u8  tos;
      __be16        tot_len;
      __be16        id;
      __be16        frag_off;
      __u8  ttl;
      __u8  protocol;
      __sum16    check;
      __be32        saddr;
      __be32        daddr;
        /*The options start here. */

};
```

Tips for hacking - Contd.

- NIPQUAD() : macro for printing hex addresses
- CONFIG_NET_DMA is for TCP/IP offload.
- When you encounter: xfrm / CONFIG_XFRM this has to to do with IPSEC. (transformers).

New and future trends

- IO/AT.
- NetChannels (Van Jacobson and Evgeniy Polyakov).
- TCP Offloading.
- RDMA.
- Mulitqueus. : some new nics, like e1000 and IPW2200, allow two or more hardware Tx queues. There are already patches to enable this.

New and future trends - contd.

- See: “Enabling Linux Network Support of Hardware Multiqueue Devices”, OLS 2007.
- Some more info in: Documentation/networking/multiqueue.txt in recent Linux kernels.
- Devices with multiple TX/RX queues will have the NETIF_F_MULTI_QUEUE feature (include/linux/netdevice.h)
- MQ nic drivers will call *alloc_etherdev_mq()* or *alloc_netdev_mq()* instead of *alloc_etherdev()* or *alloc_netdev()*.

Links and more info

1) Linux Network Stack Walkthrough (2.4.20):

http://gicl.cs.drexel.edu/people/sevy/network/Linux_network_stack_wa

2) Understanding the Linux Kernel, Second Edition

By Daniel P. Bovet, Marco Cesati

Second Edition December 2002

chapter 18: networking.

- Understanding Linux Network Internals, Christian benvenuti

Oreilly , First Edition.

Links and more info

3) Linux Device Driver, by Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

   Third Edition February 2005.

   – Chapter 17, Network Drivers

4) Linux networking: (a lot of docs about specific networking topics)

   – http://linux-net.osdl.org/index.php/Main_Page

5) netdev mailing list: http://www.spinics.net/lists/netdev/

Links and more info

6) Removal of multipath routing cache from kernel code:

http://lists.openwall.net/netdev/2007/03/12/76

http://lwn.net/Articles/241465/

7) Linux Advanced Routing & Traffic Control :

http://lartc.org/

8) ebtables – a filtering tool for a bridging:

http://ebtables.sourceforge.net/

Links and more info

9) **Writing Network Device Driver for Linux**: (article)

   – http://app.linux.org.mt/article/writing-netdrivers?locale=en

Links and more info

10) Netconf – a yearly networking conference; first was in 2004.

- http://vger.kernel.org/netconf2004.html
- http://vger.kernel.org/netconf2005.html
- http://vger.kernel.org/netconf2006.html
- Next one: Linux Conf Australia, January 2008,Melbourne
- David S. Miller, James Morris , Rusty Russell , Jamal Hadi Salim ,Stephen Hemminger , Harald Welte, Hideaki YOSHIFUJI, Herbert Xu ,Thomas Graf ,Robert Olsson ,Arnaldo Carvalho de Melo and others

Links and more info

11) **Policy Routing With Linux** - Online Book Edition

   – by Matthew G. Marsh (Sams).

   – http://www.policyrouting.org/PolicyRoutingBook/

12) THRASH - A dynamic LC-trie and hash data structure:

Robert Olsson Stefan Nilsson, August 2006

http://www.csc.kth.se/~snilsson/public/papers/trash/trash.pdf

13) IPSec howto:

http://www.ipsec-howto.org/t1.html

Links and more info

14) Openswan: Building and Integrating Virtual Private Networks , by Paul Wouters, Ken Bantoft

http://www.packtpub.com/book/openswan/mid/061205jqdnh2by

publisher: Packt Publishing.