IP Layer Implementatoin of Linux Kernel Stack

Fida Ullah Khattak
Department of Communication and Networking
School of Electrical Engineering
Aalto University
fidaullah.khattak@aalto.fi

ABSTRACT

The foundations of modern day communication networks are based on the IPv4 protocol. Though specifications of IPv4 are readily available, this is not the case with its implementation details. Open source operating systems provide a chance to look at IPv4 implementation. However, because of their changing nature, the documentation of open source operating systems is often out-dated. This work is an attempt to document the implementation details of IPv4 in the Linux kernel 3.5.4.
Detail description of IPv4 packet traversal paths at the Linux kernel is provided. An overview of routing subsystem and its role in the IPv4 packet traversal is also given.

Keywords

Linux Kernel Stack, IP Layer, Routing

1. INTRODUCTION

The linux IP layer [2] implementation can be broken down into the input, the output and the forwarding paths. The IP layer also interacts with supporting protocols, e.g. the ICMP [4], and subsytems, e.g. the routing subsytem, in the Linux stack for proper functioning.
An IP packet that enters the IP layer, has to traverse through one of the paths mentioned earlier. Based on its origin and destination, the packet is routed to calculate the path of the datagram inside the Linux kernel.
The functionalities of input, output and forwarding path traversal are implemented in their respective files as shown in table 1. It also shows the files that implements the routing functionality.
A packet inside the kernel is represented with an `sk_buff` structure. Apart from the datagram itself, this structure contains all necessary information required for routing the packet successfully. `sk_buff` is a common structure between all Layers of the IP stack. However, at a given time, only a single layer manipulates the values of this structure. In some cases, it is also possible to create multiple copies of the same structure for concurrent processing.
Table 2 shows some other structures important for packet processing at the IP layer. Few of these structures are exclusive to IP layer, while others are shared by different layers of the TCP/IP stack.
Figure 1 gives an overall view of how the datagrams are processed at different paths. There are several netfilter hooks included in the traversal logic for filtering packets at different points. The rest of the paper is structured as follows: Section 2 describes the ingress processing of IP datagram. Section 4 covers forwarding while Section 5 explain the egress processing. Routing subsystem is explained in section6.

2. PROCESSING OF INGRESS IP DATAGRAMS

The IP ingress traversal logic, defined in the `ip_input.c` file, is responsible for delivering the datagrams to a higher layer protocol or forwarding it to another host. It consists of functions and netfilter hooks that process an incoming datagram.
As shown by Figure 1, this path starts with the `ip_rcv()` function and ends at the `ip_local_deliver_finish()` function. The `ip_rcv()` function is registered as the handler function for incoming IP datagrams at the system startup by the `ip_init()` routine.
Before the control is passed to `ip_rcv()`, the lower layer function, `netif_receive_skb()`, sets the pointer of the socket buffer to the next layer (layer 3), such that the start of `skb->hdr` is pointing to the IP header. This way, the packet can be safely type-casted to an IP datagram. We start with a brief description of each function at the input traversal path of IP layer, explaining how a datagram is processed on its way up the stack.

1. **ip_rcv ()**

The first check performed by `ip_rcv()` is to drop the packets that are not addressed to the host but have been received by the networking device in promiscous mode. It may also clone the socket buffer if it is shared with the net device.
Next, `ip_rcv()` ensures that the IP header is entirely present in the dynamically allocated area. Afterwards, the length, version and checksum values of the IP header are verified. Once a packet has gone through all the sanity checks mentioned above, it is sent to the `NF_INET_PRE_ROUTING` hook. This hook works as a filter to catch unwanted packets in the early stage. If a packet gets through this hook, the control is passed to `ip_recv_finish()` function.

2. **ip_recv_finish()**

This function is passed as a function pointer to `NF_INET_PRE_ROUTING` hook and is called when the packet has successfully passed the hook.
It performs the key operation of routing the incoming datagrams using the `ip_route_input_no_ref()` function. If a packet is successfully routed, the routing subsytem initializes the `sk_buff->dst` and the `sk_buff->dst->input` function pointer is set to one of the following three functions:

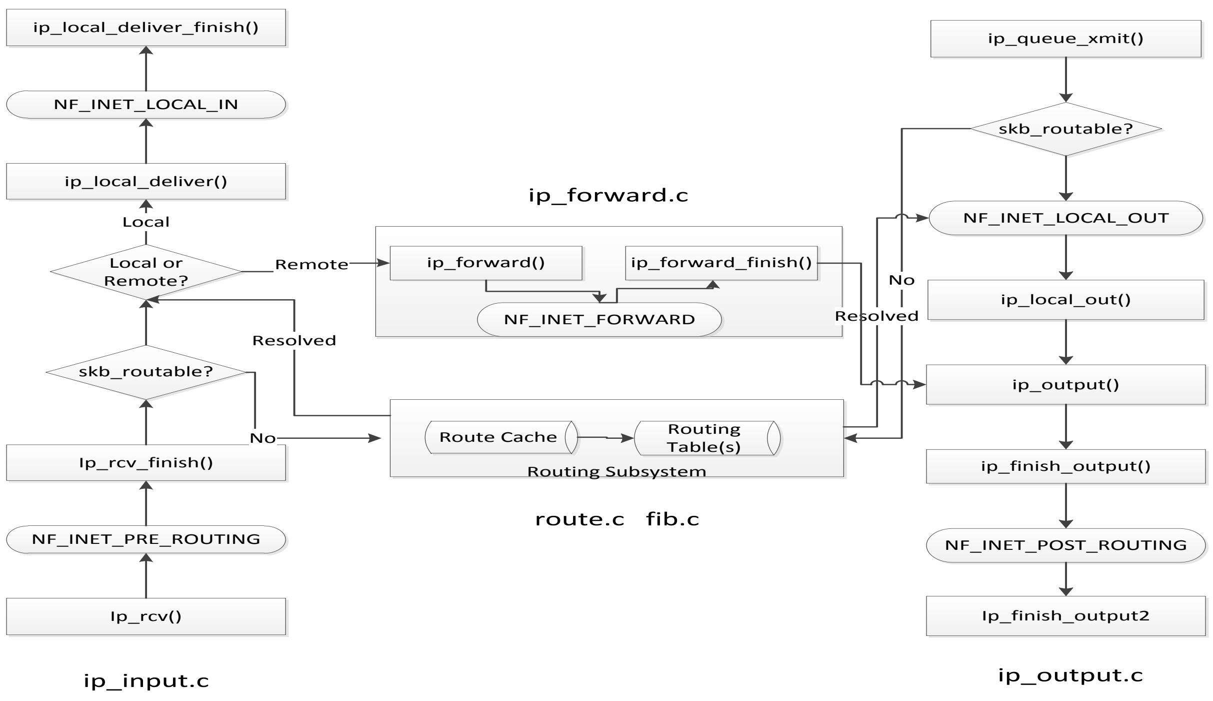

Figure 1: Traversal of an IP datagram through Linux stack

Table 1: IPv4 Important files

File Name	Explanation
net/ipv4_input.c	Functions related to ingress path
net/ipv4_output.c	Functions related to egress path
net/ipv4_forward.c	Functions related to forwarding path
net/ipv4_fragment.c	Functions related to IP fragmentation
net/ipv4_route.c	Implementation of the IP_router

(a)`ip_forward()` function is selected if the destination is not the local host and the packet is to be forwarded to another host.

(b)`ip_local_deliver()` function is selected if the packet is destined for the local host and has to be sent further up the stack.

(c)`ip_mr_input()` function is selected if the packet is a multicast packet.

Once the proper handler has been selected, IP options are processed by calling `ip_rcv_options()` function. The last step of this routine is the call to the `dest_input()` function, which in turn calls the function selected during the routing lookup by calling the `skb->dst->input()`.

3. **ip_local_deliver ()**
If the routing logic finds that a packet has to be delivered localy to a higher layer protocol, it assigns the pointer of `ip_local_delivery()` function in the `skb->dst->input()` function pointer.

The `ip_local_deliver()` routine is incharge of defragmentation. It calls the `ip_defrag()` function to queue the fragmented datagrams untill all the fragments have been received.
After receiving a complete, unfragmented datagram,`ip_local_deliver()` calls the `NF_INET_LOCAL_IN` hook and passes `ip_local_deliver_finish()` as the function pointer to be called after the netfilter processing.

4. **_local_deliver_finish ()**
When a datagram successfuly passes the netfilter hook called at the previous step, it is passed on to `ip_local_deliver_finish()` for some final processing at the IP layer. This function strips the IP header from the `sk_buffer` structure and finds the handler for further processing based on the value of protocol field in the IP header.

3. FORWARDING OF IP DATAGRAMS

As shown in Figure 1, datagrams that reach the forwarding section have already been routed by the input path and do not require another routing lookup. This makes the forwarding path relatively simple in comparison to the input or the output path. Forwarding path is responsible for checking

Table 2: IPv4 Related Data Structures

Data Structure	Location	Explanation
sk_buff	skbuff.h	Socket buffer data structure
flowi4	flow.h	Flow of traffic based on combination of fields
dst_entry	dst.h	protocol independent cached routes
iphdr	ip.h	IP header fields
ip_options	inet_sock.h	IP options of the header
ipq	ip_fragment.c	Incomplete datagram queue entry (fragment)
in_device	inet_device.h	IPv4 related configuration of network device

the optional features of "source routing" [1] and "router alert option" [3] before forwarding packets to other hosts. IPSEC policy checks for forwarded packets are also performed in the forwarding path.
Source routing option can be used by a sender to specify a list of routers through which a datagram should be delivered to the destination. Strict source routing implies that datagram must traverse all the nodes specified in the source routing list. If a datagram contains a "source routing option", the forwarding path checks if the next hop selected by the route lookup process matches the one indicated in the source routing list. In case the two hops are not the same, the packet is dropped and the source of the datagram is notified by sending an ICMP message .
Router alert option is used to indicate to the routers that a datagram needs special processing. This option is used by protocols like "Resource Reservation Protocol" [5] . A function can register itself as a handler for routing alert option, and if a datagram with a router alert option is received, the forwarding path calls this handler function.
We briefly take a look at the main functions, defined in `ip_forward.c` file, responsible for handling datagrams through the forwarding path.

1. **ip_forward()**

`ip_forward()` is the function that performs most of the forwarding related tasks, including the check for source routing and router alert options. Control is passed to the forwarding block when the routing logic decides that the packet is to be forwarded to another host and sets `skb->dst->input` pointer to the `ip_forward()` function.
Figure 3 shows the sequence of events in the `ip_forward()` function. It first checks for the IPSEC forwarding policies and the router alert option followed by checking the current TTL value of the datagram. If the router alert option is found, this function calls the corresponding handler responsible for implementing the route alert functionality and does not process the packet itself. If the TTL value of datagram is going to expire, an `ICMP TIME EXCEEDED` message is sent to the source and the packet is discarded.
Afterwards, the packet is checked for source routing. If the datagram contains a strict source routing option, it is made sure that the next hop, as mentioned in the source route list, is the same as calculated by the local route lookup. If not, the packet is discarded and source is notified.
At the end of the function, the netfilter hook, `NF_INET_FORWARD` is called , and ip_forward_finish() is passed as function pointer to the hook. This function is later executed if the packet successfully passes through the netfilter hook.

2. **ip_forward_finish()**

The `ip_forward_finish ()` function calls the `ip_forward_options()` funciton to finalize any processing required by the options included in the datagram and calculates the checksum.
At this stage the IP header has been created and the datagram has successfully traversed all checks in the forwarding block. To transmit the packet, `ip_forward_finish()` calls `dst_output()` function, another function set by the routing subsystem during the ingress traversal.

4. PROCESSING OF EGRESS IP DATAGRAMS

A higher layer protocol can pass data to IP layer in different ways. Protocols like TCP and SCTP, which fragment the packets themselves, interact with IP layer for outgoing datagrams through `ip_que_xmit()` function. Others protocols like UDP, that do not necessarily take care of the fragmentation, can call `ip_append_data()` and `ip_append_page()` functions to buffer data in Layer 3. This buffered data can then be pushed as a single datagram by calling the `ip_push_pending_frames()` function. TCP also uses `ip_build_and_send_pkt()` and `ip_send_reply()` functions for transmitting SYN ACKs and RESET messages respectively.
However, as `ip_queue_xmit()` is the most widely used method for interacting with upper layer protocols, here we discuss the sequence of events when this function is called.

(a) **ip_queue_xmit ()**

`ip_queue_xmit()` is the main function of egress path which performs many critical operations on the datagram.Figure 3 shows some of the main `ip_queue_xmit()` operations. It is in the `ip_queue_xmit()` function that the outgoing datagram is routed and a destination is set for the packet.
The routing information required by a datagram is stored in `skb->dst field`. In some cases, it is possible that the outgoing datagram has already been routed (e.g. by SCTP) and the `skb->dst` contains a valid route. In that case, `ip_queue_xmit()` skips the routing procedure, creates the IP header and sends the packet out.
In most cases, the `skb->dst` entry is empty and has to be filled by the `ip_queue_xmit()` function. There are three possible ways to route the packet.

i. The first option to find a valid route for an outgoing datagram is by using information from the "socket" structure. The socket structure is a part of `sk_buffer` structure and is passed as an argument to `__sk_dst_check()` function to search for an available route. If packets have already been sent by this socket, it will have a destination stored and can be used by any future packets originating from this socket.

ii. Another way to find a route is through a cache lookup

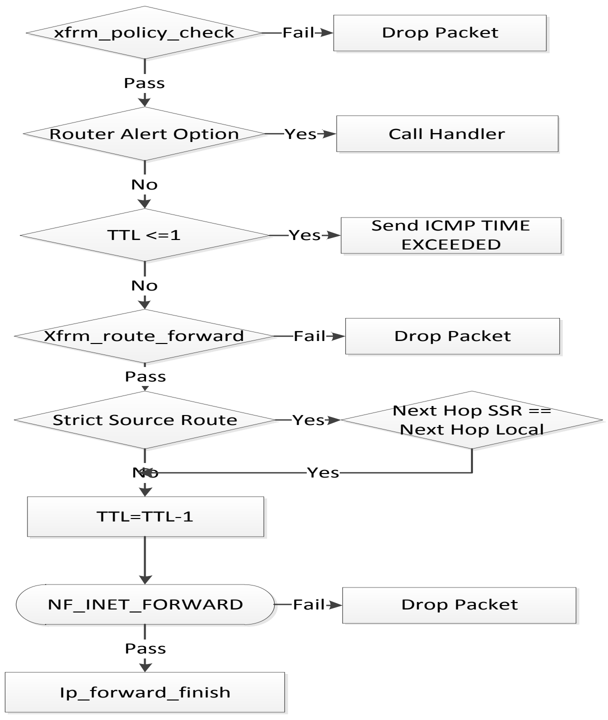

Figure 2: Forwarding of IP Datagrams

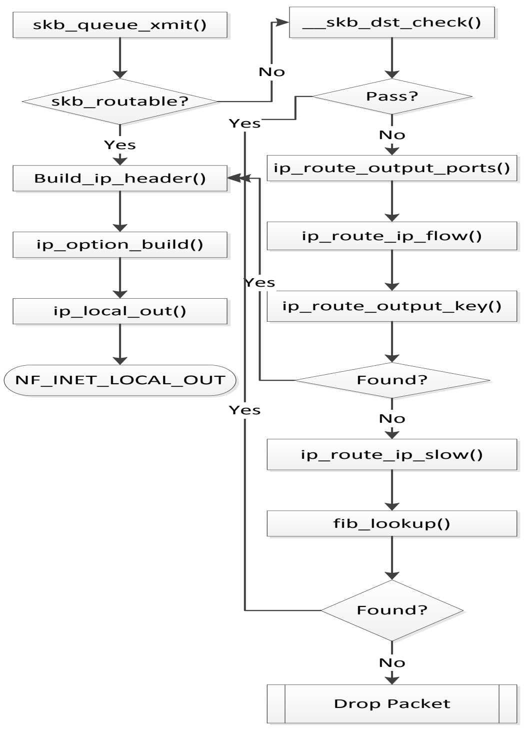

Figure 3: Egress Processing and Route Lookup

operation. If there is no route present in the socket structure `sk_buffer->sk`, the `ip_queue_xmit()` calls `__ip_route_output_key()` function for a lookup of a possible route in the routing cache.
This call is made through nested wrapper functions, `ip_route_output_ports()` and `ip_route_output_flow()`, which besides calling the `__ip_route_output_key()` function, perform IPSEC checks by calling the functions of XFRM framework and create a flowi4 structure. The `flowi` structure is later used by `__ip_route_output_key()` function for routing cache lookup.
iii. If route cache lookup also fails, `ip_route_output_slow()` is called as a last resort to find a possible route by performing a lookup on the routing table known as the forwarding information base (FIB) .

After resolving the route, the `ip_que_xmit()` function creates the IP header by using the `skb_push() function`. `Ip_options_build()` is called to build any IP options. As the last step, `dst_output()` function is called. If no routes exist to the host, the packet is dropped while incrementing the `IPSTATS_MIB_NOROUTES` counter.

(b) **ip_local_out()**
This function is a wrapper for `__ip_local_out()` which computes the checksum for outgoing datagram after initializing the value for "IP header total length". It then calls the netfilter hook `NF_INET_LOCAL_OUT`, passing the `dst_output()` function as a function pointer.

3. **dst_output()**
Like the `dst_input()` function used for processing of incoming datagrams, `dst_output()` function is a wrapper for the function pointer `skb->dst->output`. This function pointer is set to a specific function when the route lookup operation is performed and `skb->dst` is initialized.In the case when the outgoing datagram is a unicast packet, `the skb->dst->output` is set to `ip_output()` function. For multicast packets, the value of `skb->dst->output` will point to the `ip_mc_output()` function.
If a packet successfully passes through the `NF_INET_LOCAL_OUT` hook called in the previous step, it is passed to the `dst_output()` function.At this point the datagram has been routed and its IP header is in place. The dst_output () function is called not only at the egress path but also at the forwarding path to transmit the outgoing packets. As the next step, this function invokes the function assigned to the `skb->dst->output` pointer.

4. **ip_output()**
This function is invoked for transmission of unicast datagrams. It is responsible for updating the stats of the outgoing packets for the network device and calls the netfilter hook `NF_INET_POST_ROUTING` . `ip_finish_output()` is passed as a function pointer to the hook.

5. **Ip_finish_output()**
If the datagram traverses the `NF_INET_POST_ROUTING` hook successfully, it is passed to the `Ip_finish_output()` function for fragmentation checks. A packet with size more than the MTU is fragmented by calling the `ip_fragment()` function. When fragmentation is not required, the `ip_finish_output2()` function is called, which is the last hop for egress datagram processing at IP layer.

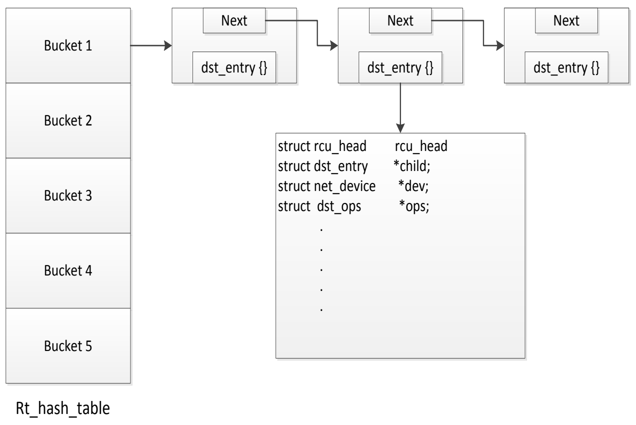

Figure 4: A simplified route cache

6. **ip_finish_output2()**
This function increments the counters for multicast and broadcast packets. It makes sure that the skb has enough space for the layer MAC header. It then tries to find the L2 neighbor address by searching for a prior cached entry for the destination. If that fails, it tries to resolve the L2 address by invoking the `neigh->output()` routine. After finding the L2 address, it calls the corresponding L2 handler for further processing. In case the search for L2 address fails, it drops the packet.

5. ROUTING SUBSYSTEM

The routing subsystem is essential to TCP/IP stack and it consists of a single routing cache and possibly many routing tables. The routing cache is a quick way for route resolution of datagrams. In case the route cache fails to provide an appropriate route entry, the routing tables are consulted to resolve the route.
Routing subsytem is initialized at the system startup by `ip_init()` function by calling `ip_rt_init()` routine. This routine initializes the routing cache by setting up the timers, defining the size of the cache and starting the garbage collector. The routing table is initialized by calling the `_ip_fib _init()` and `devinet_init()` functions to register handlers and initialize the "fib" database.
Routing subsytem is used by both input and output traversal paths of the stack to "route" a datagram. A successful lookup of route cache or routing table will result in a route entry for that particular datagram."Routing" a packet inside the kernel is equivalent to setting a valid `skb->dst` entry in the socket buffer.
This entry is critical for both incoming and outgoing datagrams. The `skb->dst->input` and `skb->dst->output` function pointers, used to direct the flow of control at various points, are set through the routing subsystem with appropriate handlers based on the destination address. Examples of such assignments are given in both ingress and egress traversal path.

5.1 Route Cache

A route cache lookup is the faster way to route packets in the Linux routing subsystem. The ip_route_input_no_ref() function of the incoming traversal path and the __ip_route _output_key() function of the output traversal path resolve routes for incoming and outgoing datagrams respectively, by performing the route lookup operation on the cache.
The route cache is implemented using the elements ofdst_e ntry structure, linked together to form "rt_hash_bucket". It is searched by using a "hash code" composed of the destination address, the source address, the TOS field values and the ingress or egress device. The computed "hash_code" is compared against the hash_code of the hash_buckets, and if a match is found, the entries in the bucket are compared against the values of flowi4 structure that is passed as an argument to the route lookup operation.
Every time a route entry is successfully returned after a route lookup, a reference counter to the route cache entry is incremented . As long as the reference count is positive, the entry is not deleted. A garbage collection mechanism for routing cache makes sure that old and unused cache entries are deleted to create space for new entries in the cache.

5.2 Routing Table

The routing table is a complicated data structure. A default routing table has two tables, an"ip fib local table" and an"ip fib main table".
The entries in the routing tables are accessed through hash lookups, which provide an efficient search mechanism for matching route entries.
The IP layer functions use `ip_route_output_slow` function to perform route lookups in the routing table, usually after route cache has failed. This function returns a pointer to routing table entry. structures of `net` and `flowi4` are given as arguments. Information about the source address, destination address, possible output interface is gathered from the flowi4 structure. The type of service, input interface (as loopback by default) and and scope of the flow (RT_SCOPE_LINK, RT_SCOPE_UNIVERSE) is gathered from the "net" argument and used for searching the route entries.

6. CONCLUSIONS

Even though IP is an old and mature protocol, its kernel implementation is constantly changing due to additions and enhancements. In this paper, we discussed the implementation of IPv4 in a state of the art Linux kernel[1]. The input, output and forwarding paths were discussed in detail and the role of the routing subsystem in the routing of IP datagrams was explained.
It will be interesting to experiment with the size of routing cache and its impact on route lookup process as an extension to this work in the future.

7. REFERENCES

[1] Ip source route options. `http://www.juniper.net/techpubs/software/junos-es/junos-es92/junos-es-swconfig-security/ip-source-route-options.html`.
[2] I. S. Institute. Internet Protocol. RFC 791, RFC Editor, September 1981.

[1]version 3.5.4

[3] D. Katz. IP Router Alert Option. RFC 2113, RFC Editor, February 1997.

[4] J. Postel. Internet Control Message Protocol. RFC 792, RFC Editor, September 1981.

[5] S. B. R. Braden, L. Zhang. Resource Reservation Protocol. RFC 2205, RFC Editor, September 1997.