Network stack code walkthrough in Linux kernel

Chidambaram Ramanathan[1], Himanshu Shah[1]

Abstract
This document gives information about the path that a TCP packet takes, starting from the time it enters the NIC over the network cable, up to the user space and back from the user space down till it leaves over the wire. The path of the packet is demonstrated by identifying the corresponding functions being called in each of the layers of the network stack in the Linux 3.8.1 kernel.

Keywords
TCP/IP Network — Linux Kernel — Code Walkthrough

[1] *Department of Computer Science, Stony Brook University, New York, United States*

Contents

Introduction

This document provides a code walkthrough of the network stack for the Linux kernel. It explains the sequence of functions that are executed on a TCP packet right from the user space till it gets to the wire and back from the physical layer. This document is based on the TCP/IP protocol suite in the linux kernel version 3.8.0 - the kernel core prevalent at the time of writing this document.

1. Data structures

In this section we will discuss some of the most prominent data structures used in the kernel TCP/IP stack.

1.1 SOFTNET_DATA

The structure is defined in include/linux/netdevice.h. Its a per CPU data structure. i.e Each CPU has its own data structure to manage ingress and egress packets. Thus there is no need for any locking mechanisms among different CPU. The structure includes both fields used for reception and fields used for transmission. Ingress frames are queued to input_ pkt_ queue, and egress frames are placed into the specialized queues handled by Traffic Control (the QoS layer) instead of being handled by softirqs and the softnet_ data structure, but softirqs are still used to clean up transmitted buffers afterward, to keep that task from slowing transmission

1.2 NET_DEVICE

The net_ device structure is defined in include/linux/netdevice.h. It is the very core structure for all networking device. All the information, that a system needs to keep track of about a networking device is maintained in this structure. There is one structure for each networking device.

1.3 SK_BUFF

The sk_ buff structure is defined in include/linux/skbuff.h. When a packet arrives to the kernel, either from the user space or from the network card one of these structures is created.

Practically every function during reception and sending of packets is invoked with sk_buff as a parameter.

2. Link Layer

Below are the primary functions provided by the link layer used for transmitting and receiving a packet.

2.1 REGISTER_NETDEVICE

net/core/dev.c

As mentioned before, the network device is represented by the net_device structure. The network device is registered with the kernel by invoking the register_netdevice method. It sets the ifindex and other parameters of the device to the net_device structure. It also sets the __link _state _flag to indicate that the device is on. The device is added to a linked list which indicates all the protocols that a new device is up through notifier _call _chain. All protocols that needs to be known when a new device has been added, needs to register to this notifier chain. For example, if the IP layer wants this information, it registers by the calling the function devinet _init. This function internally calls the main function.

$$register\_netdevice\_notifier(\&ip\_netdev\_notifier);$$

The network device pre-allocates a set of socket buffers for receiving packets. The device interrupt handler takes the sk_buff and performs operations on it.

In the case of an ethernet device, the function alloc_etherdev is used to fill up the structure with generic ethernet data.

NAPI In old linux kernels, the device driver generates an interrupt for each frame it receives. But the disadvantage here is, the time spent in handling the interrupts. It can go really high when packets come at a faster rate. So instead of using a pure interrupt driven mechanism, new linux kernels follow a method called NAPI which is a combination of interrupts and polling. In this technique, if new frames arrive when the kernel is still processing, then there is no need for the driver to generate other interrupt. It would put the packets from the queue puts in the queue and kernel processes the packets from the queue. This reduces the number of interrupts.

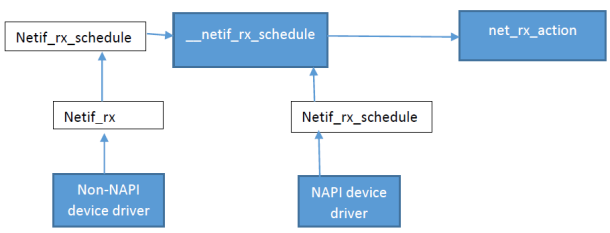

Figure illustrating the flow of packets from device drivers.

2.2 NETIF_RX

net/core/dev.c

All non-NAPI drivers use this function to receive a packet from the device driver and queue it for the upper protocol levels to process. It is called by an interrupt handler. It stores the received packet in a socket buffer called "struct sk_buff *skb" and initializes it. The function then calls the NETPOLL _RX function to check if it can accept or drop the packet. If NETPOLL_RX returns NET_RX_DROP, the packet is dropped, otherwise if it returns NET_RX_SUCCESS, the packet is accepted and further processing takes place. Each CPU has its own queue for incoming frames. The data structure for this queue is softnet_data. It then checks for congestion in the cpu queues. If all tests are satisfactory, the buffer is queued into the input queue with $\_\_skb\_queue\_tail(\&queue->input\_pkt\_queue, skb)$, the IRQ's status is restored for the CPU, and the function returns. It then schedules a softirq with the flag NET_RX_SOFTIRQ.

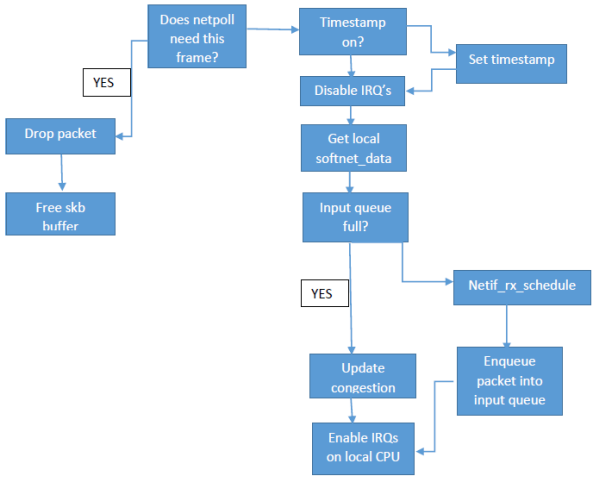

Figure illustrating the flow of packets from device drivers

The NAPI drivers call netif_rx_schedule which calls __raise_softirq_irqoff(NET_RX_SOFTIRQ).

Both the drivers queue the input device to polling list poll_list and schedule the NET_RX_SOFTIRQ which is handled by net_rx_action.

2.3 NET_RX_ACTION

net/core/dev.c

The net_rx_action processes the frames from two locations. In the case of non- NAPI drivers, the netif_rx function enqueues the frames in the softnet_data queue. In the case of NAPI drivers, it directly fetches from the device memory using the poll method. The softirq is handled by the NET_RX_ACTION function. The handlers are setup in the following fashion. The NET_DEV_INIT function sets up the queues and installs the handlers by the following two functions.

$$open\_softirq(NET\_TX\_SOFTIRQ, net\_tx\_action, NULL);$$

$$open\_softirq(NET\_RX\_SOFTIRQ, net\_rx\_action, NULL);$$

The function polls a set of devices which are in polling state through the poll function which was registered in a round robin fashion.

The NET_RX_ACTION disables the interrupts, till all the packets in each of the ring of each devices are handled by softirq. It polls each of the registered device and processes the RX_RING of each devices. RX_RING is nothing but a ring buffer where packets are enqueued at one end, and dequeued at the other end. When NET_RX_ACTION polls each of the devices, the PROCESS_BACKLOG function will be called for each of the enqueued packets.

It sets the softirq pending flag of the corresponding CPU by the following function call.

$$\_\_raise\_softirq\_irqoff(NET\_RX\_SOFTIRQ);$$

If net_rx_action was forced to break because its time limit has exceeded, then it again schedules NET_RX_ACTION so that it can process the next time through the following statements,

$$softnet\_break$$

$$\_\_get\_cpu\_var(netdev\_rx\_stat).time\_squeeze++;$$

$$\_\_raise\_softirq\_irqoff(NET\_RX\_SOFTIRQ);$$

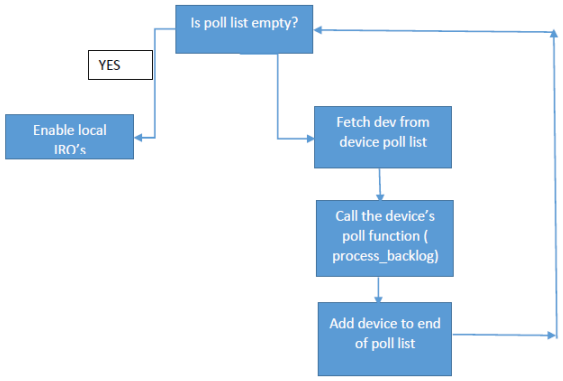

Figure illustrating the job done by net_ rx_ action

PROCESS_BACKLOG The poll virtual function of the net_device data structure, is executed by net_rx_action to process the backlog queue of a device. It is initialized by default to process_backlog in net_dev_init for those devices not using NAPI. It dequeues the packets from the input_pkt_queue through the following function.

__skb_dequeue(& queue− >input_pkt_queue);

netif_receive_skb(skb);

The function handlers are added by the corresponding protocol. For example, in the case of the IP protocol, it adds to the list of handlers through the following call. It is basically a function pointer.

```
int *func(struct sk_buff *,
struct net_device *,
struct packet_type*);
```

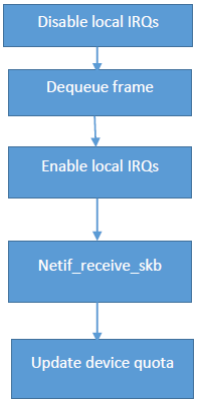

Figure illustrating the working of process_ backlog function

2.4 NETIF_RECEIVE_SKB

net/core/dev.c

This is the helper function used by the poll function which the device has registered. The NETIF_RECEIVE_SKB function classifies the input packets according to the protocol type and delivers it to the appropriate protocol through the following function call.

$$deliver\_skb(skb, pt\_prev, orig\_dev);$$

It mainly takes care of identifying the L2 and L3 headers. This function passes the copy of the frame to each protocol which is listening and specifically pass a copy of the frame to the L3 protocol handler. The protocol is specified by skb− >protocol. If there is no protocol registered, then the packet is dropped.

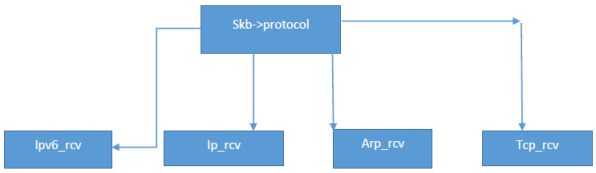

Figure illustrating the decision made by netif_ receive_ skb

3. IP Layer

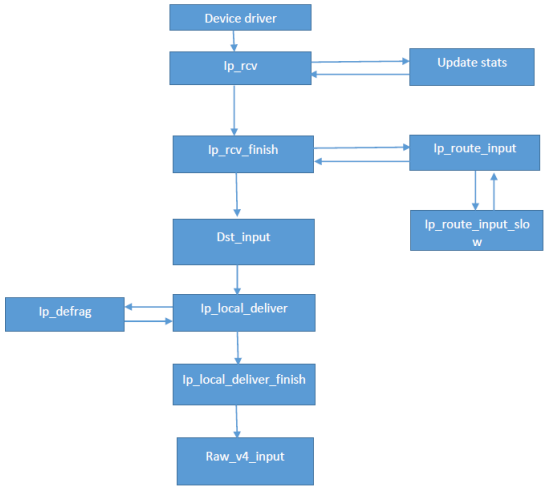

Figure illustrating the overall happenings in an L3 sub-system

3.1 L3 SUBSYSTEM INITIALISATION

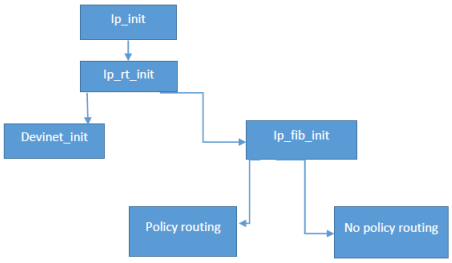

Figure illustrating the initialization of L3 sub-system

Ip_ri_init initializes certain important data structures like routing cache (rt_hash_table), Devinet_init registers with the notifier chain for information regarding devices. Ip_fib_init creates the routing tables (ip_fib_local_table , ip_fib_main_table) and it registers with the (fib_netdev_notifier and fib_inetaddr_notifier).

```
static struct notifier_block
fib_inetaddr_notifier = {
        .notifier_call =fib_inetaddr_event,
};
static struct notifier_block
fib_netdev_notifier = {
        .notifier_call =fib_netdev_event,
};
```

The netdev notifier function call is fib_netdev_event which gives information about the network device like network down, network up etc. The inetdev_notifier function call is fib_inetaddr_event which gives information about the events like ip address being added etc.

3.2 INET_INIT

net/ipv4/af_inet.c

The IP protocol registers itelf through the following function inet_ init. It creates a large route cache for the routing entries through the ip_rt_init function call. Various modules which are initialized through this routines are:

```
rt_hash_table
rt_flush_timer
rt_periodic_timer
devinet_init -> Registry to get details
about the network device
ip_fib_init -> Initializes the fib module
rt_secret_timer
ip_rt_acct
```

3.3 IP_RCV

net/ipv4/ip_input.c

The netif_receive_skb function sets the pointer to the L3 protocol $(skb->nh)$ at the end of the L2 header. Once the packet has been delivered by the lower layer to the upper layer, the corresponding packet handler which was registered will be called. This function handles IPV4 packets. If packets are destined for other hosts, then it simply drops them. The netif_receive_skb function, which is the one that calls ip_rcv, increments the reference count before it calls a protocol handler. If the handler sees a reference count bigger than 1, it creates its own copy of the buffer so that it can modify the packet. It is done by the following check

$$(skb = skb\_share\_check(skb, GFP\_ATOMIC))$$

It then makes the necessary sanity checks. If it is sure that the packet is correct and valid, then it calls ip_rcv_finish.

3.4 IP_RCV_FINISH

net/ipv4/route.c

The ip_rcv function just did some basic sanity checks and set the NF_IP_PRE_ROUTING flag indicating that it has not taken any routing decisions yet. It is the ip_rcv_finish function that deals with routing of packets. It decides whether the packets has to be locally directed or routed. It calls the ip_route_input function to search from the routing table to determine the function to call. i.e it calls the above function to determine the next hop. Once decisions have been taken it handles the packet to the upper layer protocols through the following statement.

$$skb->dst->input(skb)$$

where input is a function pointer to the registered function.

3.5 IP_ROUTE_INPUT

net/ipv4/route.c

This method first tries to find a suitable destination structure in the route cache and if that fails it invokes ip_route_input_slow method to perform a FIB lookup.

It calls rt_hash_code which returns an index into the route cache. The hash code is derived from the source address, destination address, interface index and type of service through the following line.

$$hash = rt\_hash\_code(daddr, saddr^{(}iif << 5), tos);$$

The hash code returned by the above function is used to determine the proper chain from the rt_hash_table structure. If it is a multicast packet it calls ip_route_input_mc which takes care of the processing of a multicast packet.

$if(MULTICAST(daddr))$

It calls ip_input_slow function in case of a route cache miss.

3.6 IP_ROUTE_INPUT_SLOW

net/ipv4/route.c

The function ip_ route_ input_ slow is called if an entry in the route cache does not exist and the FIB has to be asked for routes.

$$fib\_lookup(\&fl, \&res))$$

Broadcast and unicast entries are handled appropriately. If the final destination is not on this host, then the packet is deemed unreachable.

if(!IN_ DEV_ FORWARD(in_ dev)) goto e_ hostunreach;

If such an entry is found in the FIB it adds the entry in route cache.
rth = dst_alloc(& ipv4_dst_ops);

If it is destined to local machine, then it is delivered to higher layer protocols through the following call.

$$rth->u.dst.input = ip\_local\_deliver$$

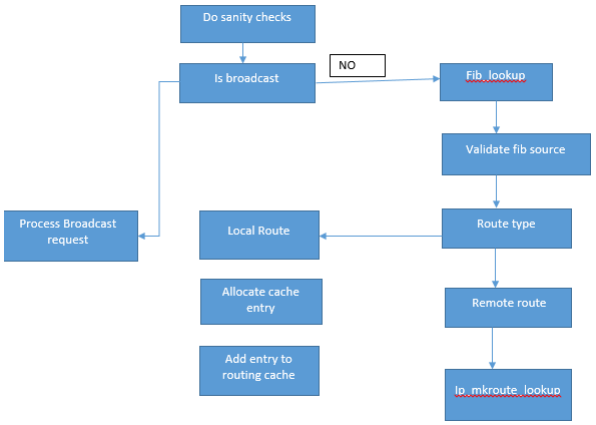

Figure illustrating the processing of ip_ route_ input_ slow function

3.7 IP_ FORWARD

net/ipv4/ip_ forward.c

The skb− >dst− >input function pointer is dynamically assigned a value depending on whether the packet has to be locally delivered or forwarded to a remote machine. In the former case it calls ip_ local_ deliver, in the latter case where the packet has to be forwarded, p_ forward is called. The call to ip_ rout_ input in ip_ rcv_ finish makes the sk_ buff buffer to contain all the information needed to forward the packet. The function is able to make decisions from the ip header represented by the struct iphdr *iph structure.

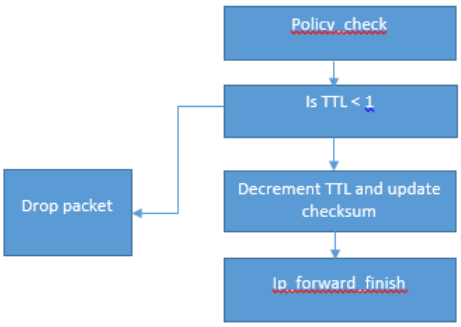

Figure illustrating the major work in ip_ forward

3.8 IP_ FORWARD_ FINISH

net/ipv4/ip_ forward.c When a packet reaches this place, it means it is actually ready for forwarding and has passes all checks. It calls the ip_ forward_ options. It checks the flags opt− >rr_ needaddr and offsets. The packet is finally transmitted with dst_ output

3.9 IP_LOCAL_DELIVER

net/ipv4/ip_input.c

This function delivers the packets to higher protocol layers. Before it does, it re-assembles the fragments and sends it.

$$skb = ip\_defrag(skb, IP\_DEFRAG\_LOCAL\_DELIVER);$$

It returns a pointer to the completely defragmented packet or NULL if the packet defragmentation is still not complete.

3.10 IP_DEFRAG

net/ipv4/ip_fragment.c

In this function, the incoming fragments are assembled into a single packet prior passing to the L4 layer using the ip_local_deliver function. The fragmentation / defragmentation module is initialized by ipfrag_init function during the execution of inet_init. Whether the packet is a fragment or not is told by looking at the IP header.

$$skb->nh.iph->frag\_off \& htons(IP\_MF|IP\_OFFSET)$$

Fragments must be stored in kernel memory until they are totally processed by the network subsystem. Ip_evictor function is called when we are hitting high threshold and need to free up some memory space. Ip_find finds the packet associated with the fragment being processed. The lookup is based on ID, source IP, destination IP and the L4 protocol. Ip_frag_queue queues a fragment into a list of fragments associated with the same packet. If it is the last entry in the queue, then we reassemble the packet using the ip_frag_reasm function.

3.11 IP_LOCAL_DELIVER_FINISH

net/ipv4/ip_input.c

This marks the point of end of IP layer. It then handles the packets over to the transport layer. It makes sure it removes the network header in the packet. The skb− >data points to the transport header. Here the transport protocol number is extracted from the IP header, converted to a hash key. If the hash chain is empty, there is by definition no raw handler that needs to see this packet.The protocol is identified by the following statement.

$$int protocol = skb->nh.iph->protocol;$$

If there is any raw socket, then it is identified by the following call.

$$raw\_sk = sk\_head(\&raw\_v4\_htable[hash]);$$

Various transport protocols register in the hash table, inet_protos, by the inet_ init function. This is the hash table for transport protocols.

$$struct net\_protocol * inet\_protos[MAX\_INET\_PROTOS];$$

4. TCP Layer

The TCP layer is initialized as follows from the programmer's perspective. The initial setup is done when the application issues a socket call. The kernel returns a socket descriptor if the socket was successfully created.

Lets look at the each of the functions in greater depth.

4.1 TCP_ INITIALIZATION

net/ipv4/af_ inet.c

This is a preliminary function which registers various protocols and its handlers to the hash table. The handlers are invoked from this hash table.

(inet_ add_ protocol(&icmp_ protocol, IPPROTO_ ICMP) < 0)
(inet_ add_ protocol(&udp_ protocol, IPPROTO_ UDP) < 0)
(inet_ add_ protocol(&tcp_ protocol, IPPROTO_ TCP) < 0)
(inet_ add_ protocol(&igmp_ protocol, IPPROTO_ IGMP) < 0)

In our case, the TCP protocol registers its handlers through this function.

```
static struct net_protocol tcp_protocol = {
.handler = tcp_v4_rcv,
.err_handler = tcp_v4_err,
.no_policy = 1,
};
```

The tcp_ init function allocates a large cache for tcp entries. It also allocates a bucket for the two states of TCP. "TCP established" and "TCP bind". It also registers with congestion control routine through tcp_ register_ congestion_ control function.

4.2 TCP_ V4_ RCV

net/ipv4/tcp_ ipv4.c

Packets intended for TCP are first processed by tcp_ v4_ rcv(). It makes a decision whether the packet needs to be processed or needs to be queued in either backlog or prequeue queues. This function is the main entry point for all TCP packets which have been delivered from the L3 sub-system. The very first job of the function is to find out socket for the TCP packet. It is done through the following statement which returns an established or a listening socket.

```
sk = __inet_lookup(&tcp_hashinfo,
skb->nh.iph->saddr, th->source,
skb->nh.iph->daddr, ntohs(th->dest),
inet_iif(skb));

return __inet_lookup_established(
hashinfo, saddr, sport, daddr,
hnum, dif); ? : inet_lookup_listener(
hashinfo, daddr, hnum, dif);
```

The __ inet_ lookup function finds a struct sock that would be associated with this packet and returns a pointer to it. Lookup is based on source port address and destination port address. If the socket is found to be in a TIMED_ WAIT state then it does the corresponding operation.

```
if (sk->sk_state == TCP_TIME_WAIT)
goto do_time_wait;
```

The four cases of TCP_ TIME_ WAIT are handled as follows. case TCP_ TW_ SYN case TCP_ TW_ ACK case TCP_ TW_ RST case TCP_ TW_ SUCCESS:

The pskb_ may_ pull function is responsible for ensuring that the TCP header is present in the kernel memory. It makes necessary sanity checks and starts to process the TCP header. The TCP header can be located by the statement

```
{
th = skb->h.th;
seq -> sequence number for the 1st byte
in the segment.  Ack\_ seq -> The next
sequence number the other end expects
from us End\_ seq -> The sequence number
just after the last byte in the segment
```

In this function we need to make a decision on whether the packet needs to be processed or needs to be queued in either backlog or prequeue queues. If the socket is locked by the user, then it is added to a backlog queue for future process when the lock has been obtained. If there is no application blocking on this socket to receive data, the packet has to be processed immediately. It is done through the tcp_ v4_ do_ rcv function call.

QUEUES FOR RECEIVING PACKETS

There are three queues to receive incoming TCP segments: They are Backlog queue, Prequeue queue and Receive queue.

sk − > receive_ queue contains processed TCP segments, which means that all the protocol headers are stripped and data are ready to be copied to the user application. sk − > receive_ queue contains all those data segments that are received in correct order. TCP segments in the other two queues are the ones that need to be processed.

4.3 TCP_ V4_ DO_ RCV

net/ipv4/tcp_ ipv4.c

The function tcp_ v4_ do_ rcv() does further processing of the connection request. It Is performed in the context of the application that will actually receive the packet.

4.4 TCP_ PREQUEUE

include/net/tcp.h

When we receive a tcp packet from tcp_ v4_ rcv and if no one is using the socket currently or some user process is waiting for the data on this socket, then we will add it to the prequeue or If the socket is already in use, we first try to queue the TCP packet in the prequeue queue by calling tcp_ prequeue(). We process all the segment in the prequeue one by one by calling callback routine sk $-$ > backlog_ rcv , backlog_ rcv points to tcp_ v4_ do_ rcv().

4.5 TCP_ RCV_ ESTABLISHED

net/ipv4/tcp_ input.c

The data packet is processed in this function. It tries to copy data to user buffer. The packet is either copied to the data buffer of the user or it is put into the receive queue. It makes some sanity checks as follows. It sees if the current task is the one which installed the receiver (tp$-$ > ucopy.task == current), and it checks if the sequence number which the user is expecting is actually the sequence number being copied (tp$-$ >copied_ seq == tp$-$ >rcv_ nxt) and checks if the length is the one requested by the user (len - tcp_ header_ len <= tp$-$ >ucopy.len) If the conditions are satisfied, then data is copied to the user buffer through tcp_ copy_ to_ iovec() function call.

5. References

Linux IP Networking
http://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html

Linux network stack walkthrough
http://edge.cs.drexel.edu/GICL/people/sevy/network/Linux_ network_ stack_ walkthrough.html

Input Routing
http://people.cs.clemson.edu/ westall/853/notes/routein.pdf

Input Routing
http://people.cs.clemson.edu/ westall/853/notes/iprecv3.pdf

Linux Networking
http://www.tldp.org/HOWTO/KernelAnalysis-HOWTO-8.html

Network Packet Processing in Linux
http://raghavclv.wordpress.com/article/network-packet-processing-in-linux-8m286fvf764g-5/

PATH OF A PACKET IN THE LINUX KERNEL STACK
http://hsnlab.tmit.bme.hu/twiki/pub/Targyak/Mar11Cikkek/Network_ stack.pdf

Networking slides Vyas Sekar
http://www.cs.stonybrook.edu/ porter/courses/cse506/f12/slides/netwo handout.pdf

TCP Implementation in Linux
http://www.ece.virginia.edu/cheetah/documents/papers/TCPlinux.pdf

FIRST STEPS IN THE LINUX KERNEL
http://people.cs.clemson.edu/w̃estall/853/notes/routein.pdf

TCP Implementation in Linux
www.ece.virginia.edu/cheetah/documents/papers/TCPlinux.pdf

MultiPath TCP - Linux Kernel implementation
http://mptcp.info.ucl.ac.be/