PATH OF A PACKET IN THE LINUX KERNEL STACK

Ashwin Kumar Chimata
ashwinc@ittc.ku.edu
University of Kansas

July 11, 2005

Contents

1 INTRODUCTION **3**

2 TCP/IP - Overview **3**

3 When Data is sent through socket **4**

3.1 Application Layer 4
3.2 The Socket Interface 5
3.3 Transport Layer 7
3.4 Network layer (IP) 9
3.5 Data Link Layer 10

4 When data is received from the Medium **12**

4.1 Physical layer 12
4.2 Network Layer - IP 14
4.3 Transport Layer 14
4.4 Application Layer 17
4.5 Conclusions 19

1 INTRODUCTION

The flow of the packet through the linux network stack is quite intriguing and has been a topic for research, with an eye for performance enhancement in end systems. This document is based on the TCP/IP protocol suite in the linux kernel version 2.6.11 - the kernel core prevalent at the time of writing this document. The sole purpose of this document is to take the reader through the path of a network packet in the kernel with pointers to LXR targets where one can have a look at the functions in the kernel which do the actual magic.

This document can serve as a ready look up for understanding the network stack, and its discussion includes KURT DSKI instrumentation points, which are highly useful in monitoring the packet behavior in the kernel.

We base our discussion on the scenario where data is written to a socket and the path of the resulting packet is traced in a code walk through sense

2 TCP/IP - Overview

TCP/IP is the most ubiquitous network protocol one can find in today's network. The protocol has its roots in the 70's even before the formulation of the ISO OSI standards. Therefore, there are four well defined layers in the TCP/IP protocol suite which encapsulate the popular seven layered architecture, within it.

Relating TCP/IP to the OSI model - The application layer in the TCP/IP protocol suite comprises of the application, presentation and the sessions layer of the ISO OSI model.

The **socket** layer acts as the interface to and from the application layer to the transport layer. This layer is also called as the **Transport Layer Interface**. It is worth mentioning that there are two kinds of sockets which operate in this layer, namely the connection oriented (streaming sockets) and the connectionless (datagram sockets).

The next layer which exists in the stack is the **Transport Layer** which encapsulates the TCP and UDP functionality within it. This forms Layer 4 of the TCP/IP protocol stack in the kernel. The **Network Layer** in the TCP/IP protocol suite is called **IP layer** as this layer contains the information about the Network topology, and this forms Layer 3 of the TCP/IP protocol stack. This layer also understands the addressing schemes and the routing protocols.

Link Layer forms Layer 2 of the stack and takes care of the error correction routines which are required for error free and reliable data transfer.

The last layer is the **Physical Layer** which is responsible for the various modulation and electrical details of data communication.

3 When Data is sent through socket

Let us examine the packet flow through a TCP socket as a model, to visualize the Network Stack operations in the linux kernel.

NOTE: All **bold faced** text are LXR search strings and the corresponding files are mostly mentioned alongside each LXR target. In the event of file names not being mentioned, an identifier search with the LXR targets will lead to the correct location which is in context.

3.1 Application Layer

The journey of the network packet starts at the application layer where data is written to the socket by a user program. The user program mostly uses the socket API which provides the system calls for the user to perform the **read & write** operations to the socket. Most operations on a socket will be simlilar to those with a normal file descriptor, but all the main functionality are well abstracted in the kernel.

The API provides a rich set of options for the user to interact with the network, some of the common calls are *send, sendto, sendmsg, write, writev*. Out of these, send, write and writev only work with connected sockets, because they do not allow the caller to specify the destination address. The write system call takes in three arguments

```
        write(socket,buffer,length);
```

The *writev* call performs the same function as the write call, except that it uses a ”gather write” form, which allows an application program to write a message without copying the data to contiguous bytes of memory.

```
        writev(socket, iovector, vectorlen);
```

Where *iovector* gives the address of an array of type *iovec* that contains a sequence of pointers to the blocks of bytes that form the message.

When a message sending call like *send, write etc* is made, the control reaches the **sock_sendmsg** system call which is in *net/socket.c*, irrespective of the kind of system call. This checks if the user buffer is readable and if so, it obtains the *sock struct* by using the socket descriptor available from the user-level program which is issuing the call. It then creates the message header based on the message transmitted and a socket control message which has information about the *UID, PID and GID* of the process. All these operations are carried out in the process context.

The control calls the **__sock_sendmsg**, which traverses to the protocol specific sendmsg function. The protocol options are consulted, through the sendmsg field of the **proto_ops** structure and the, protocol specific function is invoked. Thus, if it is a *TCP socket* then the **tcp_sendmsg** function is called and if it is a *UDP socket* then the **udp_sendmsg** function is called. These decisions are made after the control passes over the *Transport Layer Interface* and a decision is made on which protocol specific function to call.

The **tcp_sendmsg** function, defined in file *linux/net/ipv4/tcp.c* is finally invoked whenever any user-level message sending is invoked on an open SOCK_STREAM type socket.

3.2 The Socket Interface

The Socket Interface layer is sometimes called the glue layer as it acts as an interface between the *Application layer* and the lower *Transport Layer.* This is also called the *Transport Layer Interface* and is responsible for extracting the `sock` structure and checking if it is functional. In effect this layer invokes the appropriate protocol for the connection. This function is carried out in **inet_sendmsg** which is in *net/ipv4/af_inet.c*

The other relevant operations which take place at this layer are the system call translation for the various *socket creation* routines. The main functionality corresponding to socket creation takes place in the *net/socket.c*. This is the region in the kernel where all the translations for the various socket related system calls like *bind, listen, accept, connect, send & recv* are present.

In a KURT enabled kernel, we can find various instrumentation points which can be turned on to give an elaborate narrative of when and how each of these system calls are being called. Some of the instrumentation points we can find in this layer are:

```
EVENT_SOCKET -> when a socket is created.
EVENT_BIND   -> Event when a socket is bound to an address.
EVENT_LISTEN -> Event when socket listen is called.
EVENT_CONNECT -> Event when the connect system call is called
                 from a client machine.
EVENT_ACCEPT -> Event when the server accepts the connection
                from a client.
EVENT_SOCK_SENDMSG -> When a message is written to the socket.
EVENT_SOCK_RECVMSG -> When a message is read from a socket.
```

There are some more instrumentation points in this level, which have been omitted in this discussion for the sake of clarity. For a list of all instrumentation points please refer **network.ns** *in kernel/scripts/dski/network.ns*

The Socket layer is responsible for identifying the type of the protocol and for directing the control to the appropriate protocol specific function. The protocol registration takes place here and the appropriate transport layer routines are invoked.

3.3 Transport Layer

When the protocol specific routines for sending message is called, the operations which take place now are in the Transport Layer of the Network stack. The function pointer which would have been set in the *proto* structure will direct to **tcp_sendmsg or udp_sendmsg** as the case may be.

As we are dealing with the TCP case, let us examine the **tcp_sendmsg** routine. The **tcp_sendmsg** is defined in *linux/net/ipv4/tcp.c* which performs the TCP specific work on the packet. It waits till the connection is established, as TCP cannot send data till a connection is established. This section of code is shown below, here it is checking if the connection is established before the timeout occurs.

```
/* Wait for a connection to finish. */
if ((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT))
   if ((err = sk_stream_wait_connect(sk, &timeo)) != 0)
      goto out_err;
```

The other operation which the **tcp_sendmsg** takes care of is setting up the *Maximum Segment Size* for the connection.

Once the connection is established, and other TCP specific operations are performed, the actual sending of message takes place. This is done through the IO vector structure, which is a mechanism for transferring data from the user space into the kernel space. This is the place where the **struct sk_buff *skb** is created and the user data gets copied from the user space to the socket buffers in this function part of the code.

The **tcp_sendmsg** checks if there is buffer space available in the previously allocated buffers. If so, it writes the user data on to that. Else a new buffer is requested for the write operation. Basically this structure, tries to copy user information into available socket buffers, if none are available, new allocation is made for the purpose.

Once the socket buffer is filled with data, **tcp_sendmsg** copies the data from user space to kernel space by calling the **skb_copy_to_page** function, which internally calls checksum routines before copying data into kernel space. There are other page fault handling functionality which is incorporated in the **tcp_sendmsg** code which can be looked up in the function. These are routines which take care of allocating pages when message copy routines need them and so on. This function finally calls the **tcp_push_one** function which is one of the paths to **tcp_transmit_skb** function, which is the main function which transmits the TCP segments. The other ways by which the **tcp_transmit_skb** can be called are through:

```
    extern int tcp_write_xmit(struct sock *, int nonagle);
    extern int tcp_retransmit_skb(struct sock *, struct sk_buff *);
    extern void tcp_xmit_retransmit_queue(struct sock *);
    extern void tcp_simple_retransmit(struct sock *);
    and so on ...
```

The **tcp_transmit_skb** does the actual packet transmission to the *IP Layer*. This function builds the TCP header and sends the packet to the IP layer. Building the header in effect means that the source and destination ip addresses, the TCP sequence number are all set up. The important data structures which are relevant in this section are **tcphdr** - which stores the header information, **tcp_skb_cb** - is the TCP control buffer structure which contains the flags for the partially generated TCP header.

This function also takes care of the TCP scaling options and the advertised window options are also determined here. Checksum calculations accompany any data additions to the header or the data section. Finally the **queue_xmit** function is called as shown below, this queues the packet to its destination. It can either be an internal destination or an external destination, but these are decided on the next layer.

```
  err = tp->af_specific->queue_xmit(skb, 0);
                if (err <= 0)
                    return err
    /* where tp is the tcp_sock structure */
```

A return value less than zero in this case indicates that the packet has been dropped. The relevant instrumentation points in a KURT enabled kernel are:

```
EVENT_TCP_SENDMSG -> When tcp_send_msg is called
EVENT_TCP_WRITEXMIT -> when tcp_write_xmit is called
EVENT_TCP_TRANSKB -> when tcp_transmit_skb is called
EVENT_TCP_RECVMSG -> the tcp receive message event
EVENT_TCP_DATA_QUEUE -> when tcp_data_queue is called
```

These instrumentation points are placed in the different stages of data & header formation. The EVENT_TCP_TRANSKB is the instrumentation point which is placed in the **tcp_transmit_skb** function. All these functions are still executed in process context.

3.4 Network layer (IP)

The IP layer receives the packet and builds the IP header for the packet. This layer takes care of the route lookup for the packets and also maintains the Time To live (TTL for these packets). In addition to IP, the ICMP and IGMP also go hand in hand with the IP layer. Therefore these protocols can also be thought of as a part of IP. This layer handles the route look up for incoming and outgoing packets in the same way. If it is an external address it is delivered to the lower *Link Layer* else if it is meant for local delivery (incoming packet) then it is delivered to the higher layer.

When the queue_xmit function is called from within the **tcp_sock** structure, the control passes to the *IP Layer* where the function **ip_queue_xmit** which is defined in */net/ipv4/ip_output.c* is called.

The mechanisms of forwarding and routing are also incorporated in this routine, by using the *Forwarding Information Base*(FIB), which mainly handled by using the **kern_rta** structure. The discussion about forwarding and routing is not included in this document.

The routing information is checked for possible routing at this level by using the **__sk_dst_check** The packet is fragmented, if needed, by calling the **ip_fragment** function.

After the checks are performed the function **ip_route_output_flow** is called, which is the main function which takes care of routing the packets by making use of the **flowi** structure, which stores the flow information. The **ip_route_output_flow** which is defined in */net/ipv4/route.c*, calls the **__ip_route_output_key** function which finds a route and checks if the **flowi** structure is non-zero. The **__ip_route_output_key** first searches the *route cache* (an area where recently accessed routes are stored) for fast route retrieval. If a route is found it is used, else it tries to find a route by searching the FIB.

The above function is meant for fast route retrieval, if it fails to find a route from either the *route cache* or the *Forwarding Information Base* then the slow route look up function,**ip_route_output_slow** is called, which is the main output route resolving function. These controls still happens in the process context.

The complexities which reside in the route look up code and the depth of forwarding has been omitted in this document to preserve clarity.

To state in simple terms, all the packet routing is done by setting up the output field of the *neighbour cache* structure. Once all the processing of an output packet is done one of the three things can happen:

- If the packet is meant to be forwarded then the *output pointer* of the neighbour cache structure will point to **ip_forward**.
- If there is an unresolved route for a packet even after all the processing is done, then the *output pointer* points to **ip_output** function.
- If there is a resolved route after at this stage, then the *output function pointer* of the neighbour cache function will point to the **dev_queue_xmit** function.

We will forward our discussion with the assumption that a route is resolved and the **dev_queue_xmit** function is called.

3.5 Data Link Layer

The *Data Link Layer* is responsible for a large set of operations apart from just handing over the packet to the device. This layer is sometimes referred to as the queuing layer as most of the queuing disciple implementation takes place in this region. Apart from queue disciples, traffic shaping functions are also carried out in this layer.

The **dev_queue_xmit** is the data link layer function which is called for any packet which is meant to be delivered to an external destination. This function checks if the device registered with the socket buffer, has an existing queue disciple. This function disables all *local bottom halves* before obtaining the devices' queue locks.

Here we find the DSKI instrumentation which identifies the event when a packet is about to be queued into its corresponding device queue. This event is named as EVENT_DEV_QUEUE and is placed right before the actual packet enqueuing takes place. The **dev_queue_xmit** calls the **qdisc_run** routine, in a vanilla kernel. This function first runs in the process context and checks if the device has packets which need to be transmitted. If there are packets present then it initiates the transmission. If the device is not free, then the same function is executed again in the *Soft IRQ* context, to initiate the transmission.

When **queue_disc** is called in the process context, it checks the state of the device with the **netif_queue_stopped** function. If the function confirms that the device state to be up, then it calls the **qdisc_restart** function which tries to transmits the packet in process context. It first tries to obtain the *xmit_lock* for the device, if it is successful then it calls the **dev->hard_start_xmit** which transmits the packet out of the system. This routine is a device specific routine and is implemented in the device driver code of the device.

The packet is sent out into the medium by calling a set of I/O instructions to copy the packet to harware and start transmitting. After the packet transmission is completed, the device frees the *sk_buff* space occupied by the packet in the hardware and records the time when the transmission took place.

If this transmission fails for any reason, then the packet is *requeued* again for processing at a future time. This is done from the error handling routines in the **qdisc_restart** function. If for some reason the packet transmission could not occur, then it calls the **netif_schedule** function, which schedules the packet tranmission in the *Soft IRQ* context.

The **netif_schedule** function calls the **__netif_schedule** function, which raises the **NET_TX_SOFTIRQ** for this transmission. The DSKI event which is inserted at this part of the packet transmission is called **EVENT_NET_TX_SOFTIRQ**.

After the packet transmission is scheduled again and completed in the next available time, the device frees the space occupied by the *sk_buff structure* and calls the **netif_wake_queue** which informs that the device is free and can take in more packets for transmission. This function also raises a *SOFT IRQ* to schedule the next packet sending.

This completes the discussion on how a packet is sent from the applicationm layer to the medium. The next section deals with the process when a packet is received from the medium into the system.

4 When data is received from the Medium

In this section we deal with the path of a network packet from the physical medium up to the application layer. The receive side is more complicated than the send side as the control flow does not follow a linear path.

4.1 Physical layer

When a packet arrives at the Network Interface Card of the machine, the card receives the packet and the packet is transferred into a **rx_ring** through DMA. The **rx_ring** is a ring in the kernel memory where the network card transfers the incoming packet through DMA. The raw data which is stored in the **rx_ring** structure is either copied into a **sk_buff** structure.

After the packet is transferred to the kernel memory, the card interrupts the CPU, to inform about the availability of a new packet. The CPU then transfers the control to the core ISR which will take care of the packet processing. As the processing in the interrupt context should be as low as possible, the Interrupt Service Routine initiates the **NET_RX_SOFTIRQ** which will take the packet processing further. The interrupt handler calls the **netif_rx_schedule** function which is defined in *(include/linux/netdevice.h)*, Which in turn calls **__netif_rx_schedule**.

The **__netif_rx_schedule** function puts a reference to the device into the **softnet_data** poll list, and schedules the **NET_RX_SOFTIRQ**. From now on, further processing of the packet is taken care of in the soft irq context.

Whenever the **NET_RX_SOFTIRQ** is scheduled by the scheduler, it executes its registered handler function which is the **net_rx_action** which is defined in *net/core/dev.c*. In a kurt enabled kernel we can find the **NET_DEVICE_LAYER_FAM** instrumentation point named as **EVENT_NET_RX_ACTION** which appears everytime the **net_rx_action** function is called.

The **net_rx_action** function, disables the interrupts, till all the packets in the **rx_ring** of each of the devices are handled by the *Soft IRQ*. This function polls each of the registered devices and processes the **rx_ring** of each of the devices. The **net_device** structure defined in *(include/linux/netdevice.h)* has a function pointer for **poll** which points to which **process_backlog** function which is defined in *(/net/core/dev.c)*.

The **backlog_dev** is a pseudo device which is added to the poll list, whenever **netif_rx** function is called, from the device driver, if it is not already present in the poll list. This is the mechanism, which is used to remove a packet which has been enqueued into the **input_pkt_queue** by the network drivers. The poll function of the **backlog_device**, which points to **process_backlog**, is used to remove the packets from the input queue, and to process each of them.

Therefore, when **net_rx_action** polls each of the devices, ultimately the **process_backlog** function gets called for each of the packets, enqueued by each of the devices.

The **process_backlog** function dequeues the packet from the input queue of the device by calling the **__skb_dequeue** function, into a **sk_buff** structure. Here we will have access to the actual device, from the **sk_buff** of the packet and the **netif_receive_skb** function is called for further processing.

The **netif_receive_skb** function, classifies the packet according to its type, and directs it to the appropriate packet handler function, for instance, in the case of *IP* the function **ip_rcv** is the registered packet handler.

4.2 Network Layer - IP

The main function which receives the packet from the *net device layer* is the **ip_rcv** function. This function checks the packet for errors and discards the *IP* header, defragments the packet if necessary. The packet passes through the pre-routing net filter hook and then reaches **ip_rcv_finish** function, which is defined in *net/ipv4/ip_input.c*.

In a KURT enabled kernel we will find the **IP_LAYER_FAM** family event **EVENT_IP_RCV**, which corresponds to the instance when a packet is received at the IP layer.

The **ip_rcv_finish** does the route look up for this packet and decides if the packet is to be delivered locally or is to be forwarded and finally calls the **dst_input** function.The **dst_input** function in turn calls the **ip_local_deliver** function, if the packet is meant to be delivered locally.

In a KURT enabled kernel the **IP_LAYER_FAM** event,
EVENT_IP_LOCAL_DELIVER is present at this stage of packet delivery. The **ip_local_deliver** function defined in */net/ipv4/ip_input.c*, de-fragments the packet if necessary and calls the **ip_local_deliver_finish** function which inturn calls the protocol specific functions to process the packet.

The **ip_local_deliver_finish** function strips the packet of its IP header and finds the protocol associated with the packet by using a hash function which hashes based on the protocol number. Based on the protocol identification number, the appropriate packet handlers for each protocol gets called.

If it is the TCP protocol then the **tcp_v4_rcv** function gets called. This signifies the transition to the transport layer of the network stack, during packet reception.

4.3 Transport Layer

TCP associates a handler function, by initializing an instance of the **inet_protocol** structure. This handler field is set to **tcp_v4_rcv** function. Therefore, **tcp_v4_rcv**, defined in *linux/net/ipv4/tcp_ipv4.c*, is called from ipv4 when the protocol type in the IP header contains the protocol number for TCP.

This function checks if the packet has a valid TCP header, by calling the **pskb_may_pull** function, which checks if the packet header field has a complete header. The tcp_v4_rcv function then initializes the checksum extracts required field from the TCP header, which is used for fast path processing.

This function internally uses a macro called **TCP_SKB_CB** to get to the TCP control buffer from the socket buffer. The TCP control buffer maintains state regarding parameters like selective acknowledgement, the tcp sequence number etc.

The next step for this function is to find an open socket for this incoming packet, this is done by calling the **__tcp_v4_lookup**, in the following segment of code:

```
sk = __tcp_v4_lookup(skb->nh.iph->saddr, th->source,
                     skb->nh.iph->daddr, ntohs(th->dest),
                     tcp_v4_iif(skb));
```

If no TCP socket is found by this call, then the packet is discarded at this stage. If a valid TCP socket is found then we continue with the processing of the packet. The IP security parameters are checked in this routine and the connection is checked to see if it is in the **TCP_TIME_WAIT** state. If this is the case, then delayed TCP segments are discarded.

The socket is checked if it is in the locked state, in top-half context, if so it cannot accept packets and so the other incoming packets are added to the backlog device. One the other hand if the socket is not in the locked state, then the packets are put in the *prequeue* struture. Once the packets are in the *prequeue* then the packets can be processed in the process context rather than in the kernel context. This transfer from kernel to user space is done by the function **tcp_prequeue** which is called from the **tcp_v4_rcv** function. If the **tcp_prequeue** returns zero, which meants that there was no current user task which was associated with the socket in hand, then **tcp_v4_do_rcv** is called which is the slow packet delivery path.

We will continue our discussion based on the tcp_prequeue semantics initially. The **tcp_prequeue** is defined as an inline function in *include/net/tcp.h*, and is the main function which is responsible for queuing up the buffers for any waiting user process. When a user level socket is woken up and a read is issued on the socket application, the **tcp_prequeue** immediately processes the socket's prequeue. This function checks if a user task is waiting for data from a socket, and processes the prequeue buffer. The following portion of code copies the socket buffer to the user's address space:

```
if (!sysctl_tcp_low_latency && tp->ucopy.task) {
__skb_queue_tail(&tp->ucopy.prequeue, skb);
tp->ucopy.memory += skb->truesize;
```

When the slow packet transfer path is taken, that is, when **tcp_v4_do_rcv** function is called, then the following semantics is followed. When **tcp_v4_do_rcv** is called, it checks if the TCP connection is in established state, by examining the state variable to be **TCP_ESTABLISHED**. If the connection is established then the **tcp_rcv_established** function is called.

The **tcp_rcv_established** function starts processing the packet with the assumption that the packets are to be processed in the fast path. If options are set in the packet then the packet processing is diverted to the slow path. This function checks to see if the packet sequence number is in order and then direct the packet to fast or slow path based on options set up in the packet.

If this function detects a fast path the packet is copied directly to the user space after checking to see if the global **current** is the same as the task that has requested the service. The following code segment is the place where this is done:

```
   if (tp->ucopy.task == current &&
       tp->copied_seq == tp->rcv_nxt &&
       len - tcp_header_len <= tp->ucopy.len &&
       sock_owned_by_user(sk)) {
               __set_current_state(TASK_RUNNING);
.
.
```

The slow path processing, if taken does various checks based on options and branches out depending on the options, after sucessful packet reception **ACK** is sent and finally the **tcp_data_queue** function is called which queues the data segments in the socket's normal receive queue.

For segments in the socket's receive queue, processing is done when the application calls the read system call, which in turn calls the **tcp_recvmsg** function.

In a KURT enabled system, this stage of processing is symbolized by the **EVENT_TCP_RCV_MESSAGE** instrumentation point.

4.4 Application Layer

Whenever an user application issues an API call like *read, recvfrom*, they are mapped onto system calls **sys_recv** defined in */net/socket.c* which gets translated into the **sys_recvfrom** call.

The **sys_recvfrom** and other *recv* system calls gets translated into the **sock_recvmsg** function defined in */net/socket.c*. In the case of INET sockets, the **inet_recvmsg** defined in */net/ipv4/af_inet.c* is called, which calls the protocol specific receive function. In the case of TCP the **tcp_recvmsg** is called.

All system calls get translated into the **tcp_recvmsg** function at the socket layer which is defined in *linux/net/ipv4/tcp.c* This is the function which copies data from an open socket buffer into a user buffer. The KURT instrumentation point **EVENT_TCP_RECVMSG** can be found in this function.

This function also includes processing, when an urgent data processing need is communicated through the SIGURG signal by any process. The basic mechanism involved in this function is, a target byte size is check and this is used for limiting the size of data transferred from the socket buffer to the user space.

This completes the path of the packet from the medium to the user application. The packet path can be visualized by the network instrumentation diagram attached at the end of the document. The diagram illustrates the salient instrumentation points at the different layers of the Network stack we have discussed till now. A pass through the diagram while going through the document can be very educational.

The Diagram is an illustration of a stimulus response loop which exists between a Master and a client(slave) machine. The path of the stimulus corresponds to the path of any network packet, in the TCP/IP network stack.

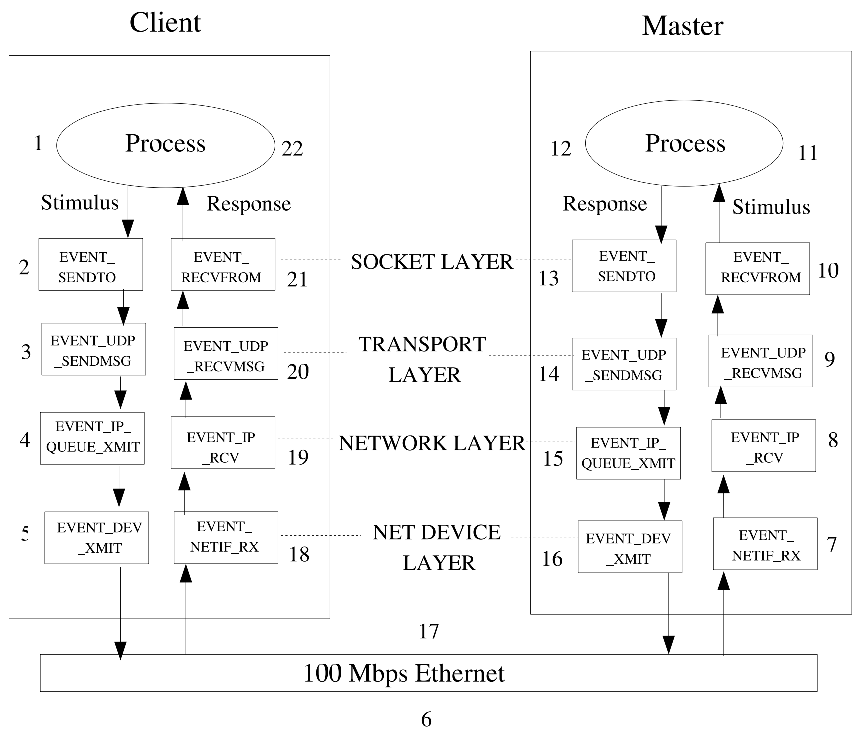

Figure 1: Linux Network Stack Instrumentation Points

4.5 Conclusions

The document presented a detailed flow through the linux TCP network protocol stack, for both the send and receive sides of the transmission. Though the document can be appreciated better by making LXR references pointed out in this document, following the document even otherwise is very helpful.

References

[1] The Linux TCP/IP Stack: Networking for Embedded Systems by THOMAS F. HERBERT

[2] Badri Subramanian. Real-Time Networking for Quality of Service on TDM based Ethernet Master's Thesis., University of Kansas, 2005

[3] Hariprasad Sampathkumar. Using Time Division Multiplexing to support Real-time Networking on Ethernet Master's Thesis., University of Kansas, 2005

[4] Ashwin Kumar Chimata & Senthil Shanmugham. Stimulus Response Characteristics of Ethernet TDM using star Topology Advanced Operating Systems, Semester Project, June 2005.