A Guide to Using Raw Sockets

By **Subodh Saxena** - March 21, 2015

In this tutorial, let us take a look at how raw sockets can be used to receive data packets and send those packets to specific user applications, bypassing the normal TCP/IP protocols.

If you have no knowledge of the Linux kernel, yet are interested in the contents of network packets, raw sockets are the answer. A raw socket is used to receive raw packets. This means packets received at the Ethernet layer will directly pass to the raw socket. Stating it precisely, a raw socket bypasses the normal TCP/IP processing and sends the packets to the specific user application (see Figure 1).

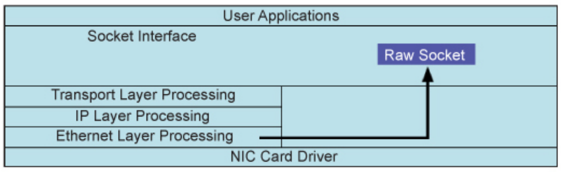


Figure 1: Graphical demonstration of a raw socket

A raw socket vs other sockets

Other sockets like stream sockets and data gram sockets receive data from the transport layer that contains no headers but only the payload. This means that there is no information about the source IP address and MAC address. If applications running on the same machine or on different machines are communicating, then they are only exchanging data.

The purpose of a raw socket is absolutely different. A raw socket allows an application to directly access lower level protocols, which means a raw socket receives un-extracted packets (see Figure 2). There is no need to provide the port and IP address to a raw socket, unlike in the case of stream and datagram sockets.

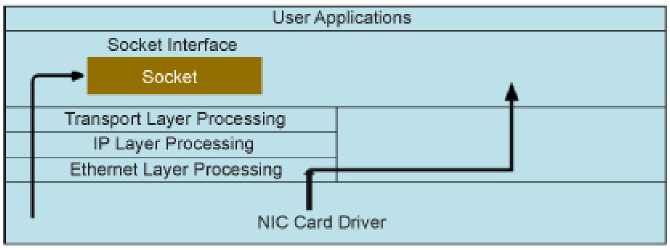
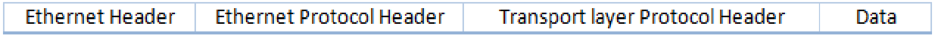


Figure 2: Graphical demonstration of how a raw socket works compared to other sockets

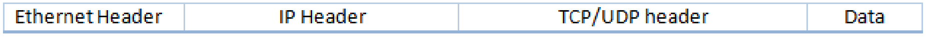
Network packets and packet sniffers

When an application sends data into the network, it is processed by various network layers. Before sending data, it is wrapped in various headers of the network layer. The wrapped form of data, which contains all the information like the source and destination address, is called a network packet (see Figure 3). According to Ethernet protocols, there are various types of network packets like Internet Protocol packets, Xerox PUP packets, Ethernet Loopback packets, etc. In Linux, we can see all protocols in the *if_ether.h* header file (see Figure 4).



Ethernet Header	Ethernet Protocol Header	Transport layer Protocol Header	Data

Figure 3: A generic representation of a network packet



Ethernet Header	IP Header	TCP/UDP header	Data

Figure 4: Network Packet for internet Protocol

When we connect to the Internet, we receive network packets, and our machine extracts all network layer headers and sends data to a particular application. For example, when we type *www.google.com* in our browser, we receive packets sent from Google, and our machine extracts all the headers of the network layer and gives the data to our browser.

By default, a machine receives those packets that have the same destination address as that of the machine, and this mode is called the non-promiscuous mode. But if we want to receive all the packets, we have to switch into the promiscuous mode. We can go into the promiscuous mode with the help of *ioctls.*

If we are interested in the contents or the structure of the headers of different network layers, we can access these with the help of a packet sniffer. There are various packet sniffers available for Linux, like Wireshark. There is a command line sniffer called *tcpdump,* which is also a very good packet sniffer. And if we want to make our own packet sniffer, it can easily be done if we know the basics of C and networking.

A packet sniffer with a raw socket

To develop a packet sniffer, you first have to open a raw socket. Only processes with an effective user ID of 0 or the *CAP_NET_RAW* capability are allowed to open raw sockets. So, during the execution of the program, you have to be the root user.

Opening a raw socket

To open a socket, you have to know three things – the socket family, socket type and protocol. For a raw socket, the socket family is *AF_PACKET*, the socket type is *SOCK_RAW* and for the protocol, see the if_ether.h header file. To receive all packets, the macro is *ETH_P_ALL* and to receive IP packets, the macro is *ETH_P_IP* for the protocol field.

```
int sock_r;
sock_r=socket(AF_PACKET,SOCK_RAW,htons(ETH_P_ALL));
if(sock_r<0)
{
printf(“error in socket\n”);
return -1;
}
```

Reception of the network packet

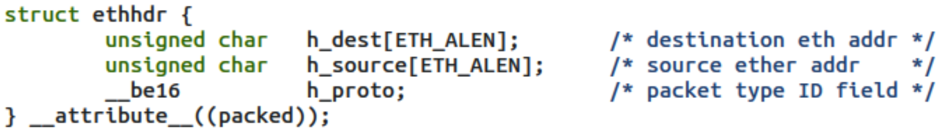
After successfully opening a raw socket, it's time to receive network packets, for which you need to use the *recvfrom api.* We can also use the *recv api*. But recvfrom provides additional information.

```
unsigned char *buffer = (unsigned char *) malloc(65536); //to receive data
memset(buffer,0,65536);
struct sockaddr saddr;
int saddr_len = sizeof (saddr);

//Receive a network packet and copy in to buffer
buflen=recvfrom(sock_r,buffer,65536,0,&saddr,(socklen_t *)&saddr_len);
if(buflen<0)
{
printf(“error in reading recvfrom function\n”);
return -1;
}
```

In *saddr*, the underlying protocol provides the source address of the packet.

Extracting the Ethernet header

Now that we have the network packets in our buffer, we will get information about the Ethernet header. The Ethernet header contains the physical address of the source and destination, or the MAC address and protocol of the receiving packet. The *if_ether.h* header contains the structure of the Ethernet header (see Figure 5).



```
struct ethhdr {
        unsigned char   h_dest[ETH_ALEN];       /* destination eth addr */
        unsigned char   h_source[ETH_ALEN];     /* source ether addr    */
        __be16          h_proto;                /* packet type ID field */
} __attribute__((packed));
```

Figure 5: Structure of Ethernet header

Now, we can easily access these fields:

```
struct ethhdr *eth = (struct ethhdr *)(buffer);
printf(“\nEthernet Header\n”);
printf(“\t|-Source Address : %.2X-%.2X-%.2X-%.2X-%.2X-%.2X\n”,eth->h_source[0],eth->h_source[1],eth->h_
printf(“\t|-Destination Address : %.2X-%.2X-%.2X-%.2X-%.2X-%.2X\n”,eth->h_dest[0],eth->h_dest[1],eth->h
printf(“\t|-Protocol : %d\n”,eth->h_proto);
```

h_proto gives information about the next layer. If you get 0x800 *(ETH_P_IP)*, it means that the next header is the IP header. Later, we will consider the next header as the IP header.

Note 1: *The physical address is 6 bytes*.

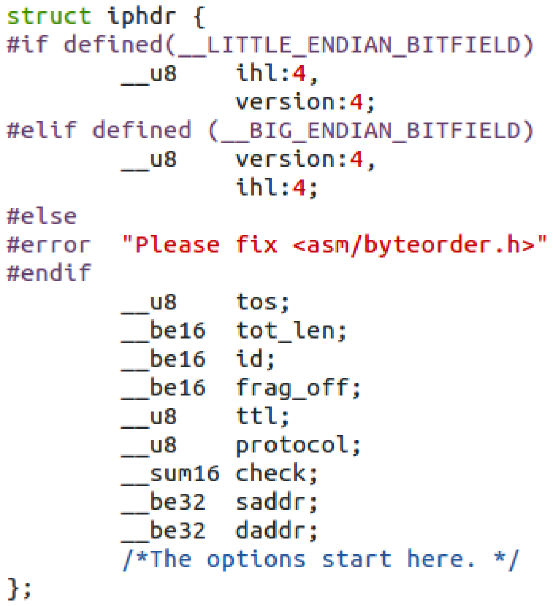
Note 2: *We can also direct the output to a file for better understanding.*

```
fprintf(log_txt,”\t|-Source Address : %.2X-%.2X-%.2X-%.2X-%.2X-%.2X\n”,eth->h_source[0],eth->h_source[1
```

Use *fflush* to avoid the input-output buffer problem when writing into a file.

Extracting the IP header

The IP layer gives various pieces of information like the source and destination IP address, the transport layer protocol, etc. The structure of the IP header is defined in the *ip.h header file* (see Figure 6).

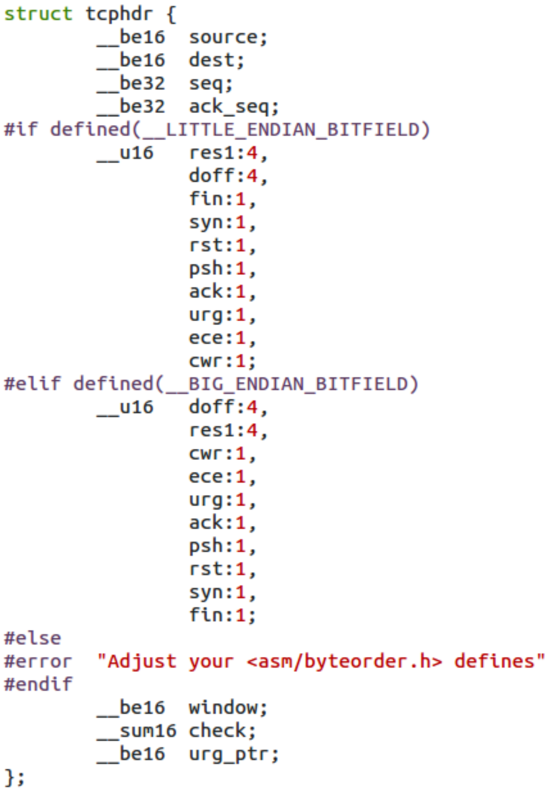


```
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u8    ihl:4,
                version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
        __u8    version:4,
                ihl:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
        __u8    tos;
        __be16  tot_len;
        __be16  id;
        __be16  frag_off;
        __u8    ttl;
        __u8    protocol;
        __sum16 check;
        __be32  saddr;
        __be32  daddr;
        /*The options start here. */
};
```

Figure 6: Structure of IP Header

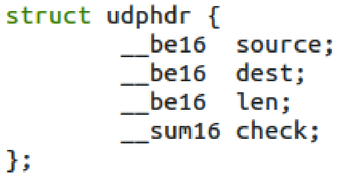
Now, to get this information, you need to increment your buffer pointer by the size of the Ethernet header because the IP header comes after the Ethernet header:

```
unsigned short iphdrlen;
struct iphdr *ip = (struct iphdr*)(buffer + sizeof(struct ethhdr));
memset(&source, 0, sizeof(source));
source.sin_addr.s_addr = ip->saddr;
memset(&dest, 0, sizeof(dest));
dest.sin_addr.s_addr = ip->daddr;

fprintf(log_txt, “\t|-Version : %d\n”,(unsigned int)ip->version);

fprintf(log_txt , “\t|-Internet Header Length : %d DWORDS or %d Bytes\n”,(unsigned in

fprintf(log_txt , “\t|-Type Of Service : %d\n”,(unsigned int)ip->tos);
fprintf(log_txt , “\t|-Total Length : %d Bytes\n”,ntohs(ip->tot_len));

fprintf(log_txt , “\t|-Identification : %d\n”,ntohs(ip->id));

fprintf(log_txt , “\t|-Time To Live : %d\n”,(unsigned int)ip->ttl);

fprintf(log_txt , “\t|-Protocol : %d\n”,(unsigned int)ip->protocol);

fprintf(log_txt , “\t|-Header Checksum : %d\n”,ntohs(ip->check));

fprintf(log_txt , “\t|-Source IP : %s\n”, inet_ntoa(source.sin_addr));

fprintf(log_txt , “\t|-Destination IP : %s\n”,inet_ntoa(dest.sin_addr));
```



```
struct tcphdr {
        __be16  source;
        __be16  dest;
        __be32  seq;
        __be32  ack_seq;
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u16   res1:4,
                doff:4,
                fin:1,
                syn:1,
                rst:1,
                psh:1,
                ack:1,
                urg:1,
                ece:1,
                cwr:1;
#elif defined(__BIG_ENDIAN_BITFIELD)
        __u16   doff:4,
                res1:4,
                cwr:1,
                ece:1,
                urg:1,
                ack:1,
                psh:1,
                rst:1,
                syn:1,
                fin:1;
#else
#error  "Adjust your <asm/byteorder.h> defines"
#endif
        __be16  window;
        __sum16 check;
        __be16  urg_ptr;
};
```

Figure 7: Structure of TCP Header



```
struct udphdr {
        __be16  source;
        __be16  dest;
        __be16  len;
        __sum16 check;
};
```

Figure 8: Structure of UDP Header

The transport layer header

There are various transport layer protocols. Since the underlying header was the IP header, we have various IP or Internet protocols. You can see these protocols in the */etc/protocls file.* The TCP and UDP protocol structures are defined in tcp.h and udp.h respectively. These structures provide the port number of the source and destination. With the help of the port number, the system gives data to a particular application (see Figures 7 and 8).
The size of the IP header varies from 20 bytes to 60 bytes. We can calculate this from the IP header field or IHL. IHL means Internet Header Length (IHL), which is the number of 32-bit words in the header. So we have to multiply the IHL by 4 to get the size of the header in bytes:

```
struct iphdr *ip = (struct iphdr *)( buffer + sizeof(struct ethhdr) );
/* getting actual size of IP header*/
iphdrlen = ip->ihl*4;
/* getting pointer to udp header*/
struct tcphdr *udp=(struct udphdr*)(buffer + iphdrlen + sizeof(struct ethhdr));
```

We now have the pointer to the UDP header. So let's check some of its fields.

Note: *If your machine is little endian, you have to use ntohs because the network uses the big endian scheme.*

```
fprintf(log_txt , “\t|-Source Port : %d\n” , ntohs(udp->source));
fprintf(log_txt , “\t|-Destination Port : %d\n” , ntohs(udp->dest));
fprintf(log_txt , “\t|-UDP Length : %d\n” , ntohs(udp->len));
fprintf(log_txt , “\t|-UDP Checksum : %d\n” , ntohs(udp->check));
```

Similarly, we can access the TCP header field.

Extracting data

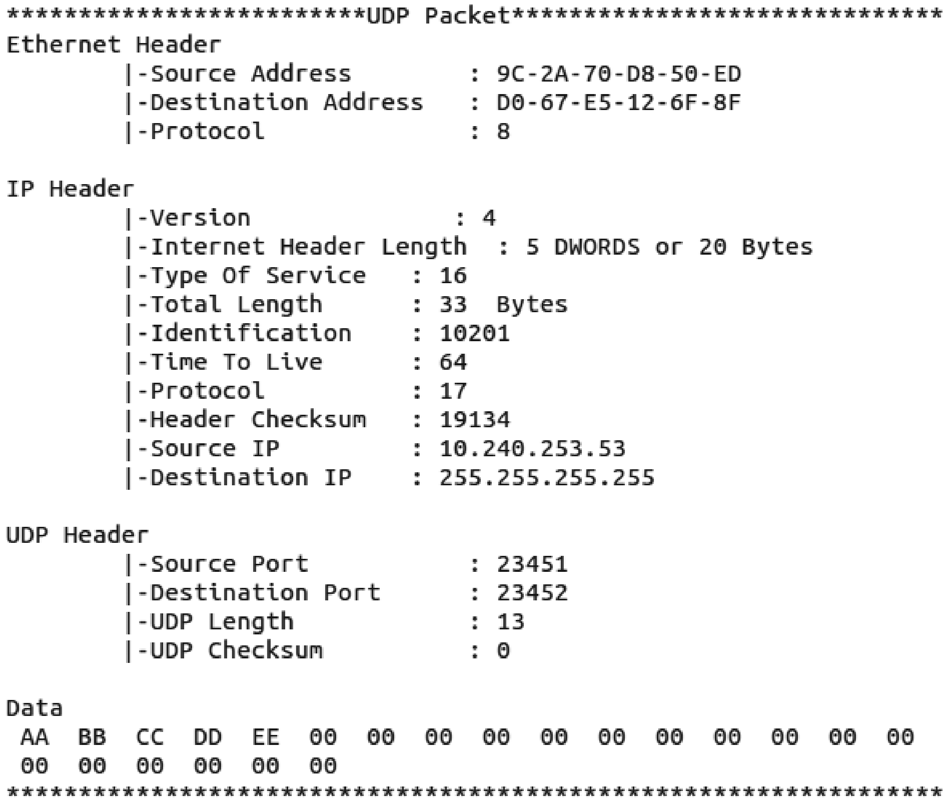
After the transport layer header, there is data payload remaining. For this, we will move the pointer to the data, and then print.

```
unsigned char * data = (buffer + iphdrlen + sizeof(struct ethhdr) + sizeof(struct udphdr));
```

Now, let’s print data, and for better representation, let us print 16 bytes in a line.

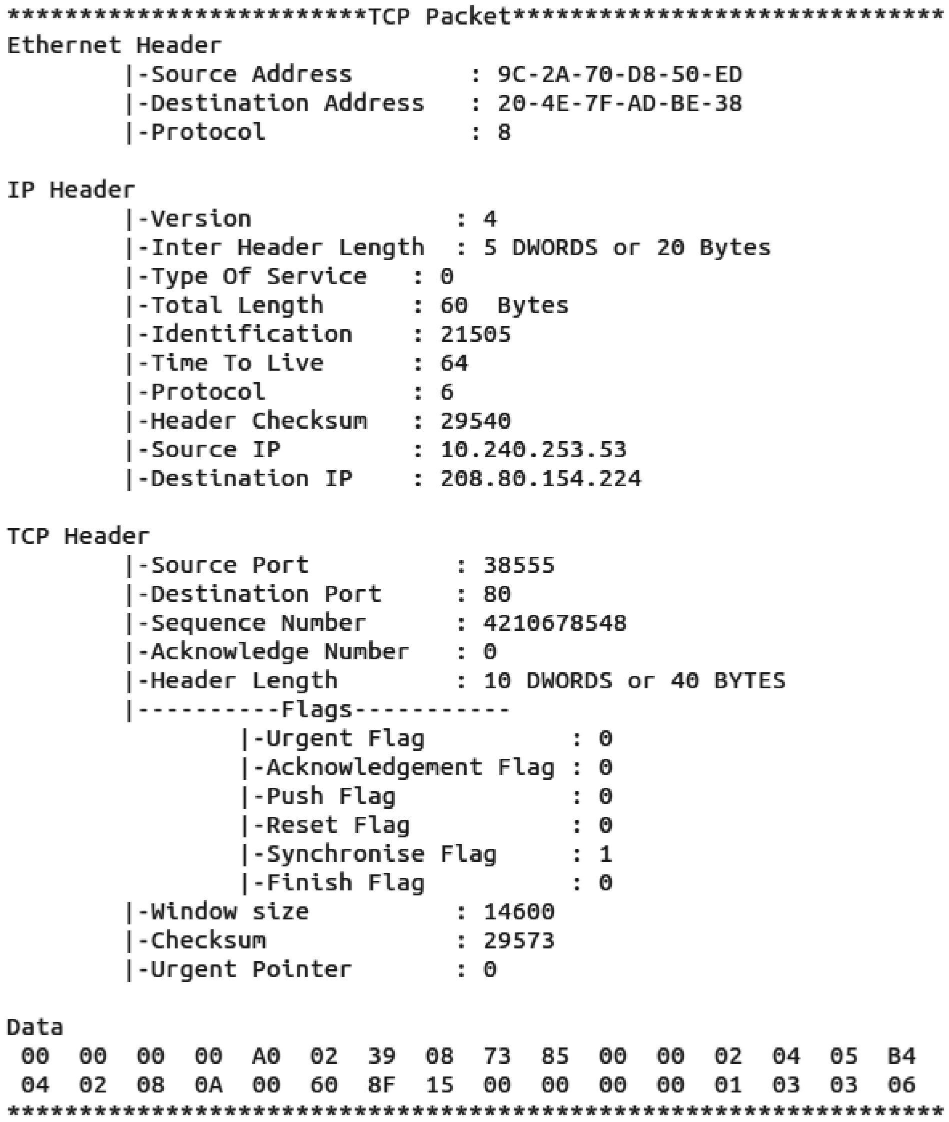
```
int remaining_data = buflen - (iphdrlen + sizeof(struct ethhdr) + sizeof(struct udphdr));

for(i=0;i<remaining_data;i++)
{
if(i!=0 && i%16==0)
fprintf(log_txt,”\n”);
fprintf(log_txt,” %.2X “,data[i]);
}
```

When you receive a packet, it will look like what’s shown is Figures 9 and 10.



```
*************************UDP Packet******************************
Ethernet Header
        |-Source Address         : 9C-2A-70-D8-50-ED
        |-Destination Address    : D0-67-E5-12-6F-8F
        |-Protocol               : 8

IP Header
        |-Version               : 4
        |-Internet Header Length  : 5 DWORDS or 20 Bytes
        |-Type Of Service   : 16
        |-Total Length      : 33  Bytes
        |-Identification    : 10201
        |-Time To Live      : 64
        |-Protocol          : 17
        |-Header Checksum   : 19134
        |-Source IP         : 10.240.253.53
        |-Destination IP    : 255.255.255.255

UDP Header
        |-Source Port            : 23451
        |-Destination Port       : 23452
        |-UDP Length             : 13
        |-UDP Checksum           : 0

Data
 AA  BB  CC  DD  EE  00  00  00  00  00  00  00  00  00  00  00
 00  00  00  00  00  00
*****************************************************************
```

Figure 9: UDP Packet



```
*************************TCP Packet********************************
Ethernet Header
        |-Source Address        : 9C-2A-70-D8-50-ED
        |-Destination Address   : 20-4E-7F-AD-BE-38
        |-Protocol              : 8

IP Header
        |-Version              : 4
        |-Inter Header Length  : 5 DWORDS or 20 Bytes
        |-Type Of Service   : 0
        |-Total Length      : 60  Bytes
        |-Identification    : 21505
        |-Time To Live      : 64
        |-Protocol          : 6
        |-Header Checksum   : 29540
        |-Source IP         : 10.240.253.53
        |-Destination IP    : 208.80.154.224

TCP Header
        |-Source Port          : 38555
        |-Destination Port     : 80
        |-Sequence Number      : 4210678548
        |-Acknowledge Number   : 0
        |-Header Length        : 10 DWORDS or 40 BYTES
        |----------Flags-----------
                |-Urgent Flag          : 0
                |-Acknowledgement Flag : 0
                |-Push Flag            : 0
                |-Reset Flag           : 0
                |-Synchronise Flag     : 1
                |-Finish Flag          : 0
        |-Window size          : 14600
        |-Checksum             : 29573
        |-Urgent Pointer       : 0

Data
 00  00  00  00  A0  02  39  08  73  85  00  00  02  04  05  B4
 04  02  08  0A  00  60  8F  15  00  00  00  00  01  03  03  06
*******************************************************************
```

Figure 10: TCP Packet

Sending packets with a raw socket

To send a packet, we first have to know the source and destination IP addresses as well as the MAC address. Use your friend's *MAC & IP* address as the destination IP and MAC address. There are two ways to find out your IP address and MAC address:

1. Enter *ifconfig* and get the IP and MAC for a particular interface.
2. Enter *ioctl* and get the IP and MAC.

The second way is more efficient and will make your program machine-independent, which means you should not enter *ifconfig* in each machine.

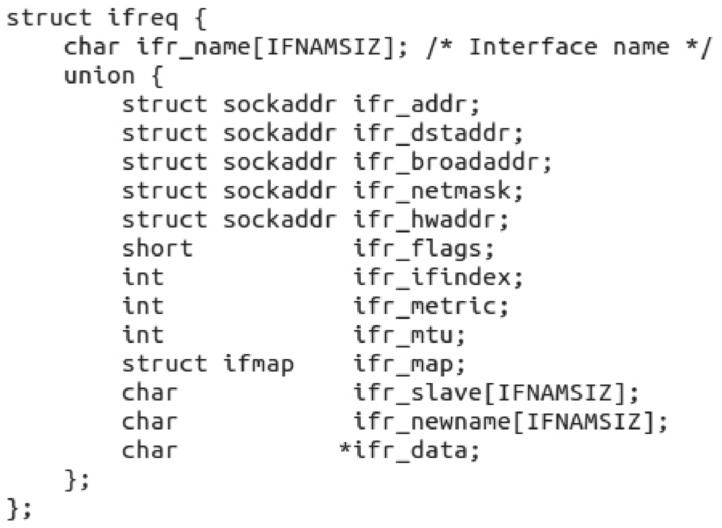
Opening a raw socket

To open a raw socket, you have to know three fields of socket *API — Family- AF_PACKET, Type- SOCK_RAW* and for the protocol, let's use *IPPROTO_RAW* because we are trying to send an IP packet. *IPPROTO_RAW* macro is defined in the in.h header file:

```
sock_raw=socket(AF_PACKET,SOCK_RAW,IPPROTO_RAW);
if(sock_raw == -1)
printf(“error in socket”);
```

What is struct ifreq?

Linux supports some standard ioctls to configure network devices. They can be used on any socket's file descriptor, regardless of the family or type. They pass an ifreq structure, which means that if you want to know some information about the network, like the interface index or interface name, you can use ioctl and it will fill the value of the *ifreq* structure passed as a third argument. In short, the *ifreq* structure is a way to get and set the network configuration. It is defined in the if.h header file or you can check the man page of *netdevice* (see Figure 11).



```
struct ifreq {
    char ifr_name[IFNAMSIZ]; /* Interface name */
    union {
        struct sockaddr ifr_addr;
        struct sockaddr ifr_dstaddr;
        struct sockaddr ifr_broadaddr;
        struct sockaddr ifr_netmask;
        struct sockaddr ifr_hwaddr;
        short           ifr_flags;
        int             ifr_ifindex;
        int             ifr_metric;
        int             ifr_mtu;
        struct ifmap    ifr_map;
        char            ifr_slave[IFNAMSIZ];
        char            ifr_newname[IFNAMSIZ];
        char           *ifr_data;
    };
};
```

Figure 11: Structure of ifreq

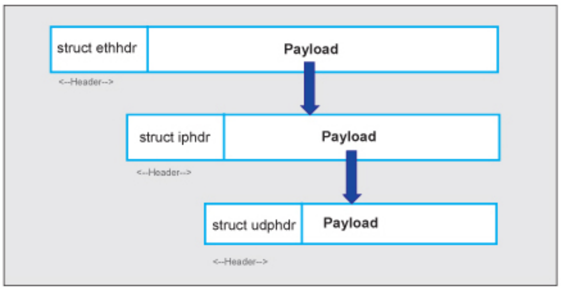


Figure 12: Graphical representation of packets with their structure and payload

Getting the index of the interface to send a packet

There may be various interfaces in your machine like *loopback, wired interface* and *wireless interface*. So you have to decide the interface through which we can send our packet. After deciding on the interface, you have to get the index of that interface. For this, first give the name of the interface by setting the field i*fr_name* of *ifreq* structure, and then use ioctl. Then use the *SIOCGIFINDEX* macro defined in sockios.h and you will receive the index number in the ifreq structure:

```
struct ifreq ifreq_i;
memset(&ifreq_i,0,sizeof(ifreq_i));
strncpy(ifreq_i.ifr_name,"wlan0",IFNAMSIZ-1); //giving name of Interface

if((ioctl(sock_raw,SIOCGIFINDEX,&ifreq_i))<0)
printf("error in index ioctl reading");//getting Index Name

printf("index=%d\n",ifreq_i.ifr_ifindex);
```

Getting the MAC address of the interface

Similarly, you can get the MAC address of the interface, for which you need to use the SIOCGIFHWADDR macro to ioctl:

```
struct ifreq ifreq_c;
memset(&ifreq_c,0,sizeof(ifreq_c));
strncpy(ifreq_c.ifr_name,"wlan0",IFNAMSIZ-1);//giving name of Interface

if((ioctl(sock_raw,SIOCGIFHWADDR,&ifreq_c))<0) //getting MAC Address
printf("error in SIOCGIFHWADDR ioctl reading");
```

Getting the IP address of the interface

For this, use the SIOCGIFADDR macro:

```
struct ifreq ifreq_ip;
memset(&ifreq_ip,0,sizeof(ifreq_ip));
strncpy(ifreq_ip.ifr_name,“wlan0”,IFNAMSIZ-1);//giving name of Interface
if(ioctl(sock_raw,SIOCGIFADDR,&ifreq_ip)<0) //getting IP Address
{
printf(“error in SIOCGIFADDR \n”);
}
```

Constructing the Ethernet header

After getting the index, as well as the MAC and IP addresses of an interface, it's time to construct the Ethernet header. First, take a buffer in which you will place all information like the Ethernet header, IP header, UDP header and data. That buffer will be your packet.

```
sendbuff=(unsigned char*)malloc(64); // increase in case of more data
memset(sendbuff,0,64);
```

To construct the Ethernet header, fill all the fields of the ethhdr structure:

```
struct ethhdr *eth = (struct ethhdr *)(sendbuff);

eth->h_source[0] = (unsigned char)(ifreq_c.ifr_hwaddr.sa_data[0]);
eth->h_source[1] = (unsigned char)(ifreq_c.ifr_hwaddr.sa_data[1]);
eth->h_source[2] = (unsigned char)(ifreq_c.ifr_hwaddr.sa_data[2]);
eth->h_source[3] = (unsigned char)(ifreq_c.ifr_hwaddr.sa_data[3]);
eth->h_source[4] = (unsigned char)(ifreq_c.ifr_hwaddr.sa_data[4]);
eth->h_source[5] = (unsigned char)(ifreq_c.ifr_hwaddr.sa_data[5]);

/* filling destination mac. DESTMAC0 to DESTMAC5 are macro having octets of mac address. */
eth->h_dest[0] = DESTMAC0;
eth->h_dest[1] = DESTMAC1;
eth->h_dest[2] = DESTMAC2;
eth->h_dest[3] = DESTMAC3;
eth->h_dest[4] = DESTMAC4;
eth->h_dest[5] = DESTMAC5;

eth->h_proto = htons(ETH_P_IP); //means next header will be IP header

/* end of ethernet header */
total_len+=sizeof(struct ethhdr);
```

Constructing the IP header

To construct the IP header, increment sendbuff by the size of the Ethernet header and fill each field of the *iphdr* structure. Data after the IP header is called the payload for the IP header and, in the same way, data after the Ethernet header is called the payload for the Ethernet header. In the IP header, there is a field called Total Length, which contains the size of the IP header plus the payload. To know the size of the payload of the IP header, you must know the size of the UDP header and the UDP payload. So, some field of the *iphdr structure* will get the value after filling the UDP header field.

```
struct iphdr *iph = (struct iphdr*)(sendbuff + sizeof(struct ethhdr));
iph->ihl = 5;
iph->version = 4;
iph->tos = 16;
iph->id = htons(10201);
iph->ttl = 64;
iph->protocol = 17;
iph->saddr = inet_addr(inet_ntoa((((struct sockaddr_in *)&(ifreq_ip.ifr_addr))->sin_addr)));
iph->daddr = inet_addr(“destination_ip”); // put destination IP address

total_len += sizeof(struct iphdr);
```

Construct the UDP header

Constructing the UDP header is very similar to constructing the IP header. Assign values to the fields of the *udphdr* structure. For this, increment the *sendbuff* pointer by the size of the Ethernet and the IP headers.

```
struct udphdr *uh = (struct udphdr *)(sendbuff + sizeof(struct iphdr) + sizeof(struc
uh->source = htons(23451);
uh->dest = htons(23452);
uh->check = 0;

total_len+= sizeof(struct udphdr);
```

Like the IP header, the UDP also has the field len, which contains the size of the UDP header and its payload. So, first, you have to know the UDP payload, which is the actual data that will be sent.

Adding data or the UDP payload

We can send any data:

```
sendbuff[total_len++] = 0xAA;
sendbuff[total_len++] = 0xBB;
sendbuff[total_len++] = 0xCC;
sendbuff[total_len++] = 0xDD;
sendbuff[total_len++] = 0xEE;
```

Filling the remaining fields of the IP and UDP headers

We now have the total_len pointer and with the help of this, we can fill the remaining fields of the IP and UDP headers:

```
uh->len = htons((total_len - sizeof(struct iphdr) - sizeof(struct ethhdr)));
//UDP length field
iph->tot_len = htons(total_len - sizeof(struct ethhdr));
//IP length field
```

The IP header checksum

There is one more field remaining in the IP header check, which is used to have a checksum. A checksum is used for error checking of the header.

When the packet arrives at the router, it calculates the checksum, and if the calculated checksum does not match with the checksum field of the header, the router will drop the packet; and if it matches, the router will decrement the time to the live field by one, and forward it.

To calculate the checksum, sum up all the 16-bit words of the IP header and if there is any carry, add it again to get a 16-bit word. After this, find the complement of 1's and that is our checksum. To check whether our checksum is correct, use the above algorithm.

```
unsigned short checksum(unsigned short* buff, int _16bitword)
{
unsigned long sum;
for(sum=0;_16bitword>0;_16bitword--)
sum+=htons(*(buff)++);
sum = ((sum >> 16) + (sum & 0xFFFF));
sum += (sum>>16);
return (unsigned short)(~sum);
}

iph->check = checksum((unsigned short*)(sendbuff + sizeof(struct ethhdr)), (sizeof(struct iphdr)/2));
```

Sending the packet

Now we have our packet but before sending it, let's fill the *sockaddr_ll* structure with the destination MAC address:

```
struct sockaddr_ll sadr_ll;
sadr_ll.sll_ifindex = ifreq_i.ifr_ifindex; // index of interface
sadr_ll.sll_halen = ETH_ALEN; // length of destination mac address
sadr_ll.sll_addr[0] = DESTMAC0;
sadr_ll.sll_addr[1] = DESTMAC1;
sadr_ll.sll_addr[2] = DESTMAC2;
sadr_ll.sll_addr[3] = DESTMAC3;
sadr_ll.sll_addr[4] = DESTMAC4;
sadr_ll.sll_addr[5] = DESTMAC5;
```

And now its time to send it, for which let's use the *sendto api:*

```
send_len = sendto(sock_raw,sendbuff,64,0,(const struct sockaddr*)&sadr_ll,sizeof(struct sockaddr_ll));
if(send_len<0)
{
printf("error in sending....sendlen=%d....errno=%d\n",send_len,errno);
return -1;

}
```

How to run the program

Go to *root user*, then compile and run your program in a machine. And in another machine, or in your destination machine, run the packet sniffer program as the root user and analyse the data that you are sending.

What to do next

We made a packet sniffer as well as a packet sender, but this is a user space task. Now lets try the same things in kernel space. For this, try to understand *struct sk_buff* and make a module that can perform the same things in kernel space.

Subodh Saxena

The author works at Effortsys Technologies in the network device driver space. He is interested in new technologies, and especially open source technologies.