Chapter 3

Handling raw network packets

3.1 libpcap

libpcap is a platform-independent interface for capturing raw packets at user level. The low-level system or platform dependent functions are taken care of by **libpcap**, so we do not need to worry about the implementation of multi-platforms raw packets handling. **libpcap** consists of many functions, but by using a sub-set of the functions are sufficient to sniff raw packets. We will learn a few functions that enable us to read raw packets.

```
char *pcap_lookupdev(char * errbuf)
```

`pcap_lookupdev()` function is to get the network device for use in `pcap_open_live()` and other **libpcap** functions. If a device name is found, then it is returned as a character string, and null is return if there is an error. The `pcap.h` header file need to be included for calling any of the **libpcap** functions.

The following codes show how to make a call to `pcap_lookupdev()` and check that a device is detected:

```
#include <stdlib.h>
#include <stdio.h>
#include <pcap.h>

int main()
{
  char *device;
  char errbuf[PCAP_ERRBUF_SIZE];

  device = pcap_lookupdev(errbuf);

  if(device == NULL) {
    printf("%s\n", errbuf);
```

```
      exit(1);
   }

   printf("Device: %s\n", device);

   return 0;
}
```

If you would like to sniff on a device other than what `pcap_lookupdev()` returns, then you need to assign the interface to `device` manually. For example, `device = "eth1"`.

```
pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int to_ms,
                       char *errbuf)
```

`pcap_open_live()` function is for opening the specified device for packet capturing. The first parameter is the device that has been indentified, `snaplen` is the maximum packet size to be captured in bytes, `promisc` is to set/unset the Ethernet card in promiscuous mode (1 to set and 0 to unset), `to_ms` is the time to wait for packets in miliseconds before timeout, `errbuf` is a buffer for holding error message.

The following shows how to pass arguments to `pcap_open_live()`, and check for error:

```
#define BUFFSIZE   1500
pcap_t* descr;

descr = pcap_open_live(device, BUFFSIZE, 0, 0, errbuf);

 if (descr == NULL) {
   printf("pcap_open_live(): %s\n", errbuf);
   exit(1);
 }
```

```
int pcap_loop(pcap_t *, int count, pcap_handler callback, u_char *user)
```

`pcap_loop()` is called to grab `count` packets. If `count` is set to `-1`, then the function loops infinitely and the `callback` function is called whenever a packet is received. The last argument `user` is for passing data to the `callback` function, and is set to `NULL` if not use.

`callback()` function as show below, and it takes three parameters. The first parameter is `u_char *user` for receiving data passed in the `user` argument in `pcap_loop()` function. This parameter is handy for passing data to the `callback` function without using global variable. The second parameter is a structure of `pcap_pkthdr`, which is explained below. The last parameter contains data that has been captured.

```
static void callback(u_char *user, const struct pcap_pkthdr *hdr,
                     const u_char *packet)
```

```
struct pcap_pkthdr {
   struct timeval ts;       /* time stamp */
   bpf_u_int32 caplen;      /* length of portion present */
   bpf_u_int32 len;         /* length this packet (off wire) */
};
```

The first member `ts` in the `pcap_pkthdr` header carries the timestamp of the captured packet. `caplen` is the length of the packet captured, and `len` is the actual length of the packet.

```
void pcap_close(pcap_t *)
```

`pcap_close()` function is for closing the opened pcap descriptor and freeing memory used by pcap.

The `callback()` function is the entry point for manipulating packets. The data presented is in the order of the Internet Protocol stack starting from Data Link layer. When Ethernet is used, then the Ethernet structure defined in `net/ethernet.h` can be used to cast on the packet, then extract the needed information.

The fields in Ethernet that may be of interest are source and destination addresses and the type field, which points to the upper layer protocol. A switch statement can be used to map on the type field then call the respective functions for further parseing the packet.

```
static void callback(u_char *user, const struct pcap_pkthdr *hdr, const u_char *packet)
{
  struct ether_header *eptr;  /* Declared in net/ethernet.h */

  switch ( ntohs(eptr->ether_type) ) {
    case ETHERTYPE_ARP :
      print_arp( (u_char *) (packet + ETHER_HDR_LEN) );
      break;
    case ETHERTYPE_IP :
      print_ipv4_packet( packet + ETHER_HDR_LEN );
      break;
  }
}
```

If the Ethernet type field is Internet Procotol version 4, then a function called `print_ipv4_packet()` can be used to process the packet. The sample function below uses the IP header structure as declared in `netinet/ip.h` to cast on the packet, then use the same method as demonstrated in `callback()` function to call the upper layer protocols. The protocol field in IP header specifies the upper layer protocol.

IP header may vary in size, and this is determined by the Internet Header Length field in the IP header. The pointer to the raw packet should be shifted based on the valude as specified in the IHL field multiply by 4 for four bytes or 32 bits.

```
void print_ipv4_packet( const u_char *packet )
{
  struct ip *iphdr;

  iphdr = (struct ip *) packet;

  /* IPPROTO_??? are defined in netinet/in.h */
  switch ( iphdr->ip_p ) {
  case IPPROTO_ICMP :
    print_icmp( (u_char *) packet + (iphdr->ip_hl * 4) );
    break;
  case IPPROTO_TCP:
    print_tcp( (u_char *) packet + (iphdr->ip_hl * 4) );
    break;
  case IPPROTO_UDP:
    print_udp( (u_char *) packet + (iphdr->ip_hl * 4) );
    break;
  }
}
```