Completed with Lab Partner: Josh Lake

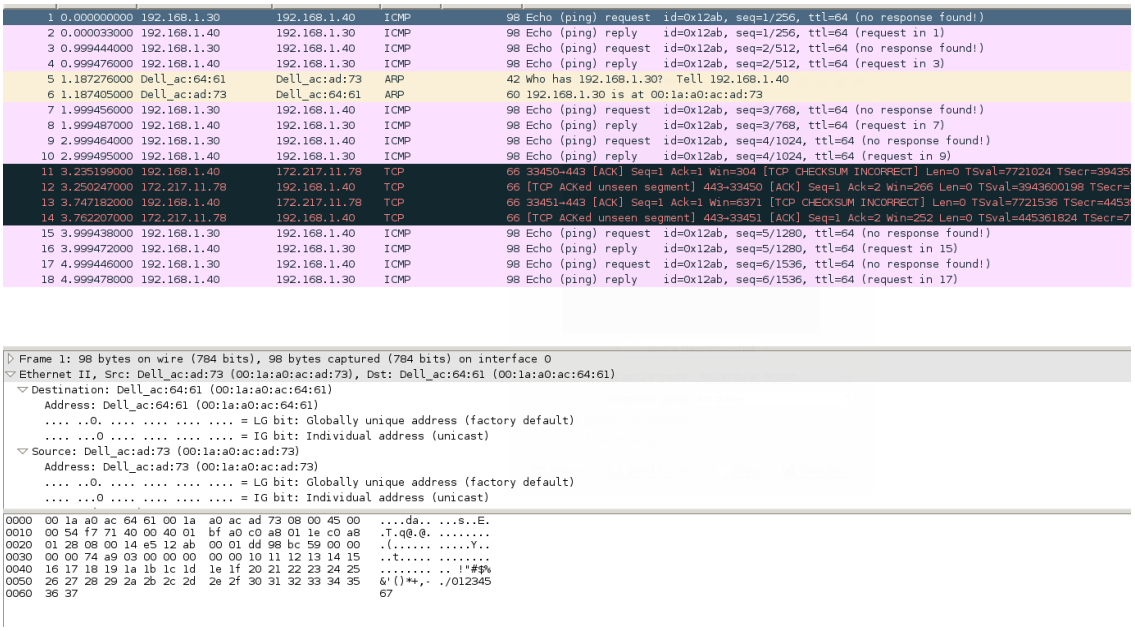
Objective

To gain familiarity with programming Linux, the structure of Ethernet packets, and Wireshark.

Structural Overview

One computer will ping another. The receiving end will run a program developed by the lab participants to dump the first 42 bytes of the packets to the console. These results will be compared with a data dump from Wireshark.

Simulation



```
    1 0.000000000 192.168.1.30   192.168.1.40   ICMP   98 Echo (ping) request  id=0x12ab, seq=1/256, ttl=64 (no response found!)
    2 0.000033000 192.168.1.40   192.168.1.30   ICMP   98 Echo (ping) reply    id=0x12ab, seq=1/256, ttl=64 (request in 1)
    3 0.999444000 192.168.1.30   192.168.1.40   ICMP   98 Echo (ping) request  id=0x12ab, seq=2/512, ttl=64 (no response found!)
    4 0.999476000 192.168.1.40   192.168.1.30   ICMP   98 Echo (ping) reply    id=0x12ab, seq=2/512, ttl=64 (request in 3)
    5 1.187276000 Dell_ac:64:61  Dell_ac:ad:73  ARP    42 Who has 192.168.1.30?  Tell 192.168.1.40
    6 1.187405000 Dell_ac:ad:73  Dell_ac:64:61  ARP    60 192.168.1.30 is at 00:1a:a0:ac:ad:73
    7 1.999456000 192.168.1.30   192.168.1.40   ICMP   98 Echo (ping) request  id=0x12ab, seq=3/768, ttl=64 (no response found!)
    8 1.999487000 192.168.1.40   192.168.1.30   ICMP   98 Echo (ping) reply    id=0x12ab, seq=3/768, ttl=64 (request in 7)
    9 2.999464000 192.168.1.30   192.168.1.40   ICMP   98 Echo (ping) request  id=0x12ab, seq=4/1024, ttl=64 (no response found!)
   10 2.999495000 192.168.1.40   192.168.1.30   ICMP   98 Echo (ping) reply    id=0x12ab, seq=4/1024, ttl=64 (request in 9)
   11 3.235199000 192.168.1.40   172.217.11.78  TCP    66 33450→443 [ACK] Seq=1 Ack=1 Win=304 [TCP CHECKSUM INCORRECT] Len=0 TSval=7721024 TSecr=39435
   12 3.250247000 172.217.11.78  192.168.1.40   TCP    66 [TCP ACKed unseen segment] 443→33450 [ACK] Seq=1 Ack=2 Win=266 Len=0 TSval=3943600198 TSecr=
   13 3.747182000 192.168.1.40   172.217.11.78  TCP    66 33451→443 [ACK] Seq=1 Ack=1 Win=6371 [TCP CHECKSUM INCORRECT] Len=0 TSval=7721536 TSecr=4453
   14 3.762207000 172.217.11.78  192.168.1.40   TCP    66 [TCP ACKed unseen segment] 443→33451 [ACK] Seq=1 Ack=2 Win=252 Len=0 TSval=445361824 TSecr=7
   15 3.999438000 192.168.1.30   192.168.1.40   ICMP   98 Echo (ping) request  id=0x12ab, seq=5/1280, ttl=64 (no response found!)
   16 3.999472000 192.168.1.40   192.168.1.30   ICMP   98 Echo (ping) reply    id=0x12ab, seq=5/1280, ttl=64 (request in 15)
   17 4.999446000 192.168.1.30   192.168.1.40   ICMP   98 Echo (ping) request  id=0x12ab, seq=6/1536, ttl=64 (no response found!)
   18 4.999478000 192.168.1.40   192.168.1.30   ICMP   98 Echo (ping) reply    id=0x12ab, seq=6/1536, ttl=64 (request in 17)

▷ Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▽ Ethernet II, Src: Dell_ac:ad:73 (00:1a:a0:ac:ad:73), Dst: Dell_ac:64:61 (00:1a:a0:ac:64:61)
  ▽ Destination: Dell_ac:64:61 (00:1a:a0:ac:64:61)
      Address: Dell_ac:64:61 (00:1a:a0:ac:64:61)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
  ▽ Source: Dell_ac:ad:73 (00:1a:a0:ac:ad:73)
      Address: Dell_ac:ad:73 (00:1a:a0:ac:ad:73)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)

0000  00 1a a0 ac 64 61 00 1a  a0 ac ad 73 08 00 45 00   ....da.. ...s..E.
0010  00 54 f7 71 40 00 40 01  bf a0 c0 a8 01 1e c0 a8   .T.q@.@. ........
0020  01 28 08 00 14 e5 12 ab  00 01 dd 98 bc 59 00 00   .(...... .....Y..
0030  00 00 74 a9 03 00 00 00  00 00 10 11 12 13 14 15   ..t..... ........
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ........ .. !"#$%
0050  26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35   &'()*+,- ./012345
0060  36 37                                             67
```

Figure 1 Wireshark Output

```
netlab40:/home/A01406538/sample codes # ./lab1
Received Frame:
00 1a a0 ac 64 61 00 1a a0 ac ad 73 08 00 45 00 00 54 bc ca 40 00
40 01 fa 47 c0 a8 01 1e c0 a8 01 28 08 00 97 cd 13 39 00 01

Received Frame:
00 1a a0 ac ad 73 00 1a a0 ac 64 61 08 00 45 00 00 54 d5 83 00 00
40 01 21 8f c0 a8 01 28 c0 a8 01 1e 00 00 9f cd 13 39 00 01

Received Frame:
00 1a a0 ac 64 61 00 1a a0 ac ad 73 08 00 45 00 00 54 be 90 40 00
40 01 f8 81 c0 a8 01 1e c0 a8 01 28 08 00 a1 cc 13 39 00 02

Received Frame:
00 1a a0 ac ad 73 00 1a a0 ac 64 61 08 00 45 00 00 54 d5 99 00 00
40 01 21 79 c0 a8 01 28 c0 a8 01 1e 00 00 a9 cc 13 39 00 02

Received Frame:
00 1a a0 ac 64 61 00 1a a0 ac ad 73 08 00 45 00 00 54 bf 87 40 00
40 01 f7 8a c0 a8 01 1e c0 a8 01 28 08 00 9b cb 13 39 00 03

Received Frame:
00 1a a0 ac ad 73 00 1a a0 ac 64 61 08 00 45 00 00 54 d7 41 00 00
40 01 1f d1 c0 a8 01 28 c0 a8 01 1e 00 00 a3 cb 13 39 00 03

Received Frame:
00 1a a0 ac 64 61 00 1a a0 ac ad 73 08 00 45 00 00 54 c1 0c 40 00
40 01 f6 05 c0 a8 01 1e c0 a8 01 28 08 00 85 ca 13 39 00 04

Received Frame:
00 1a a0 ac ad 73 00 1a a0 ac 64 61 08 00 45 00 00 54 d8 28 00 00
40 01 1e ea c0 a8 01 28 c0 a8 01 1e 00 00 8d ca 13 39 00 04

^C
```

Figure 2 Program Output

Results

It can be observed the Wireshark output and agree on the header. There was difficulty finding a complete match as it is believed there is a counter of some sort in part of the body of the ping request. However, the structure of the packets is the same, and it can be concluded that we are in fact seeing the ping requests and replies in the custom program.

Makefile

```
lab1: lab1main.cpp frameio.o
	g++ lab1main.cpp frameio.o -o lab1

frameio.o: frameio.cpp frameio.h
	g++ frameio.cpp -c -o frameio.o

clean:
	rm *.o
	rm lab1
```

lab1main.cpp

```
#include "frameio.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

frameio net;            // gives us access to the raw network

struct ether_frame      // handy template for 802.3/DIX frames
{
  octet dst_mac[6];     // destination MAC address
  octet src_mac[6];     // source MAC address
  octet prot[2];        // protocol (or length)
  octet data[1500];     // payload
};

int main()
{
  net.open_net("enp3s0");
  ether_frame buf;
  octet* raw = (octet*)(&buf);

  while(1)
  {
    int n = net.recv_frame(&buf,sizeof(buf));
    if ( n < 42 ) continue; // bad frame!
    switch ( buf.prot[0]<<8 | buf.prot[1] )
    {
      case 0x800:
      case 0x806:
        printf(
          "Received Frame: \n"
          "%02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x \n"
          "%02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x \n\n",
          raw[00], raw[01], raw[02], raw[03], raw[04], raw[05], raw[06], raw[07], raw[08], raw[09],
          raw[10], raw[11], raw[12], raw[13], raw[14], raw[15], raw[16], raw[17], raw[18], raw[19],
          raw[20], raw[21], raw[22], raw[23], raw[24], raw[25], raw[26], raw[27], raw[28], raw[29],
          raw[30], raw[31], raw[32], raw[33], raw[34], raw[35], raw[36], raw[37], raw[38], raw[39],
          raw[40], raw[41], raw[42]
        );
    }
  }
  return 0;
}
```

frameio.h (code provided by instructor)

```
//************************************************************************
//
// frameio.h
//
// frameio.h and frameio.cpp provide convenient access to the ethernet
// using raw sockets. This provides the means to read and write frames
// directly to and from the ethernet interface.
//
// Before the frameio object can be used, you must specify which interface
// you are using. This is done via the member function open_net(). For
// example, if you wish to communicate using inteface "eth1", you might
// write the code:
//
// frameio net;
//
// main(int argc, char *argv[])
```

```
// {
//    net.open_net("eth1");
//
// After the net has been opened, the interface's 6-byte MAC address can be
// obtained by calling get_mac(). Note that this function gives you a pointer
// to the object's internal storage - it is not recommended that you change
// the memory referenced by get_mac.
//
// To read from the ethernet interface, call net.recv_frame with a buffer
// address (and maximum size). The function waits for the next frame (unless
// one is already queued up) and copies it into the buffer (except for the
// CRC, which is handled by the interface). The function returns the actual
// number of bytes in the frame, but beware, it may not match the number of
// bytes in the logical frame (although it better not be smaller). Usually,
// you will want to dedicate a thread to reading the frame from the network
// and dispatching them to the protocol stack(s).
//
// To write a frame to the interface, call net.send_frame with the address
// and size of the frame to send. Again, leave off the CRC - the interface
// handles that. send_frame returns the number of bytes actually written,
// but you can usually ignore that.
//
//*************************************************************************
#ifndef FRAMEIO_H
#define FRAMEIO_H

#include <sys/socket.h>
#include <unistd.h>

//
// it is a pain to declare unsigned chars everywhere, so we define
// "octet" to be unsigned char
//
typedef unsigned char octet;

//
// class frameio - see descripion in the file header.
//
class frameio
{
public:
   //
   // send a frame to the open interface, return number of bytes sent
   //
   int send_frame(void *frame, int len)
   {
      return write(sock,frame,len);
   }
   //
   // block, waiting for a frame. When it arrives, copy it into the buffer.
   // return the number of bytes in the wire packet
   //
   int recv_frame(void *frame, int max_len)
   {
      return recvfrom(sock,frame,max_len,0,NULL,NULL);
   }
   //
   // open a socket on the specified interface and load my_mac
   //
   int open_net(const char *device); // e.g. "eth0"
   //
   // return this interface's MAC address
   //
   const octet *get_mac()
   {
      return my_mac;
   }
   //
   // Constructor
   //
   frameio()
   {
      // make sure sock is not valid
      sock = -1;
   }
   //
   // Destructor
   //
   ~frameio()
   {
      // if socket has been opened, close it
      if ( sock >= 0 ) close(sock);
      sock = -1;
   }
private:
   octet my_mac[6]; // this interface's 6-byte MAC address
   int sock;        // socket descriptor
};

#endif
```

frameio.cpp (code provided by instructor)

```
#include "frameio.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <netpacket/packet.h>
#include <net/ethernet.h>
#include <string.h>
#include <netinet/in.h>

//
// Open a raw socket on the interface and get the MAC address
//
int frameio::open_net(const char *device)
{
   const int LEN = 80;
   struct sockaddr_ll sll;
   struct ifreq ifreq;

   //
   // set up the link-layer socket address
   //
   memset(&sll, 0, sizeof sll);
   sll.sll_family = PF_PACKET;
   sll.sll_protocol = htons (ETH_P_ALL);

   //
   // open the socket, tell the OS we want all protocols
   //
   sock = socket (PF_PACKET, SOCK_RAW, htons (ETH_P_ALL));
   if ( sock < 0 ) return sock;

   //
   // if all you want to do is read, we could stop here. But to
   // write a frame, we have to bind this socket to an interface
   // and to do that, we need its interface number (small int)
   //
   strcpy (ifreq.ifr_name, device);
   ioctl (sock, SIOCGIFINDEX, &ifreq);
   sll.sll_ifindex = ifreq.ifr_ifindex;

   //
   // now let us bind...
   //
   bind (sock, (struct sockaddr *) &sll, sizeof sll);

   //
   // get the mac address
   //
   struct ifreq ifr;
   strncpy(&ifr.ifr_name[0], device, IFNAMSIZ);
   if (ioctl(sock, SIOCGIFHWADDR, &ifr) >= 0)
   {
      memcpy(my_mac, &ifr.ifr_hwaddr.sa_data, 6);
      return sock;
   }

   //
   // could have failed three different ways, but failed nontheless...
   //
   close(sock);
   return sock = -1;
}
```