Project 3

Samuel Bhushan, Partner: Hailey Parkin

November 23, 2015

1 Introduction

The goal of the internet protocol(IP) is to deliver a packet based on the IP address of the desired destination. These IP addresses may or may not be within the current network. If the address is outside of the network, the IP sends the packet to a router connected to a larger network. Many of these networks are interconnected to form the internet.

Also within the network layer is the ICMP protocol which sends messages between devices in a network. These messages have many types and uses, but this lab focuses on the 'echo' type. This type is used when the PING command is called. PING is used to check and see if a computer is connected to the network. This is done by sending out an 'echo request' ICMP message and waiting for an 'echo reply' from the device.

In this lab, our goal is to make a C++ program that sends and responsds to 'echo' messages sent to our computer. This requires us to understand the Ethernet, IP, and ICMP packet structures.

2 Program Results

2.1 Capturing and validating ICMP

The program was able to receive IP packets from the network and filter out valid ICMP packets. We initially forgot to check if the IP packet was an ICMP type before calculating its checksum. This incorrectly led us to believe we were calculating the checksum wrong. The figure below shows our program validating incoming ICMP packets.

```
netlab10:~/Desktop/Project3 # g++ -lpthread *.cpp; ./a.out
Desired IP address:
ICMP Packet received
sequence number: 1
ICMP Packet received
```

2.2 Sending out an ICMP request packet

In order to send an 'echo request', we needed to add headers and calculate checksums. The biggest difficulty in this process was in calcuating the correct sizes of all the parts. We finally settled on a frame with these sizes:

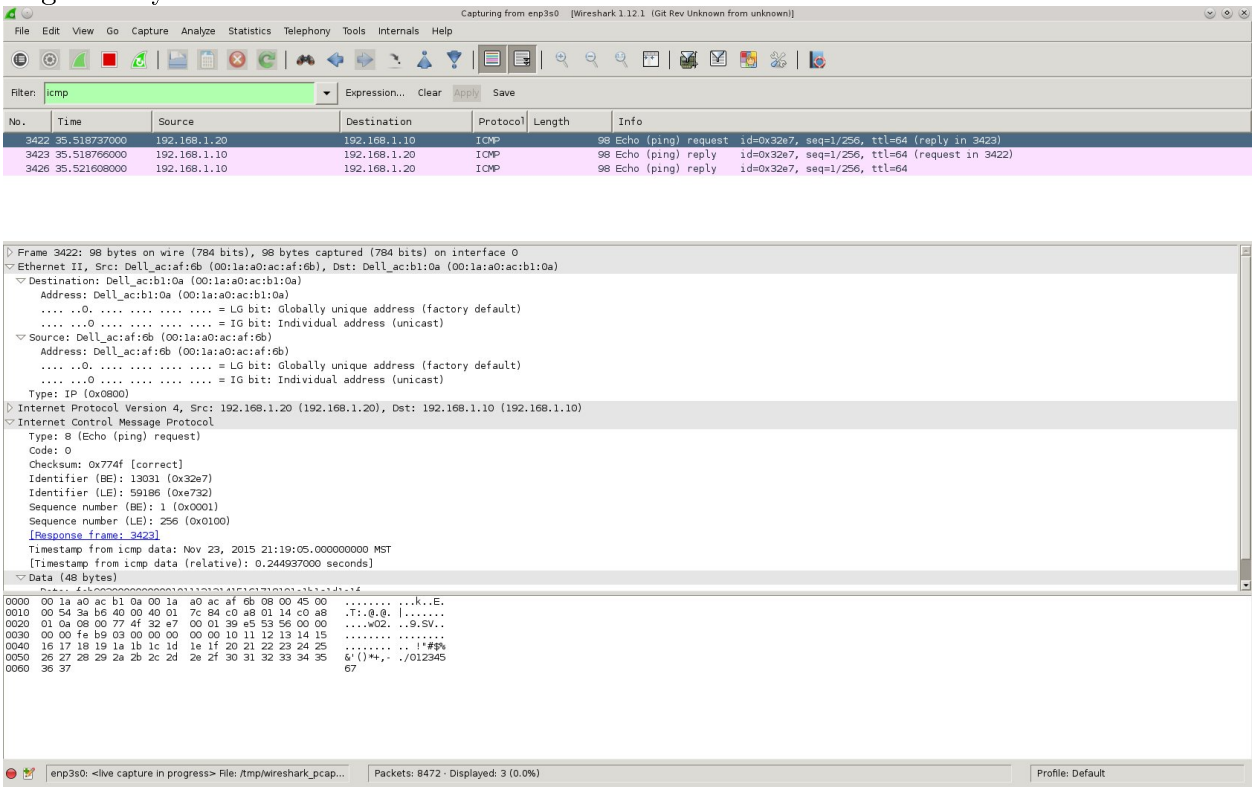
- Ethernet Frame: 98 bytes
- IP Frame: 84 bytes
- ICMP Frame: 64 bytes
- Packet Data 56 bytes

Our program sends a reaquest to the specified address and waits for a reply. This was successfully done, as shown below. In this example we sent a packet with a sequence number of 1.

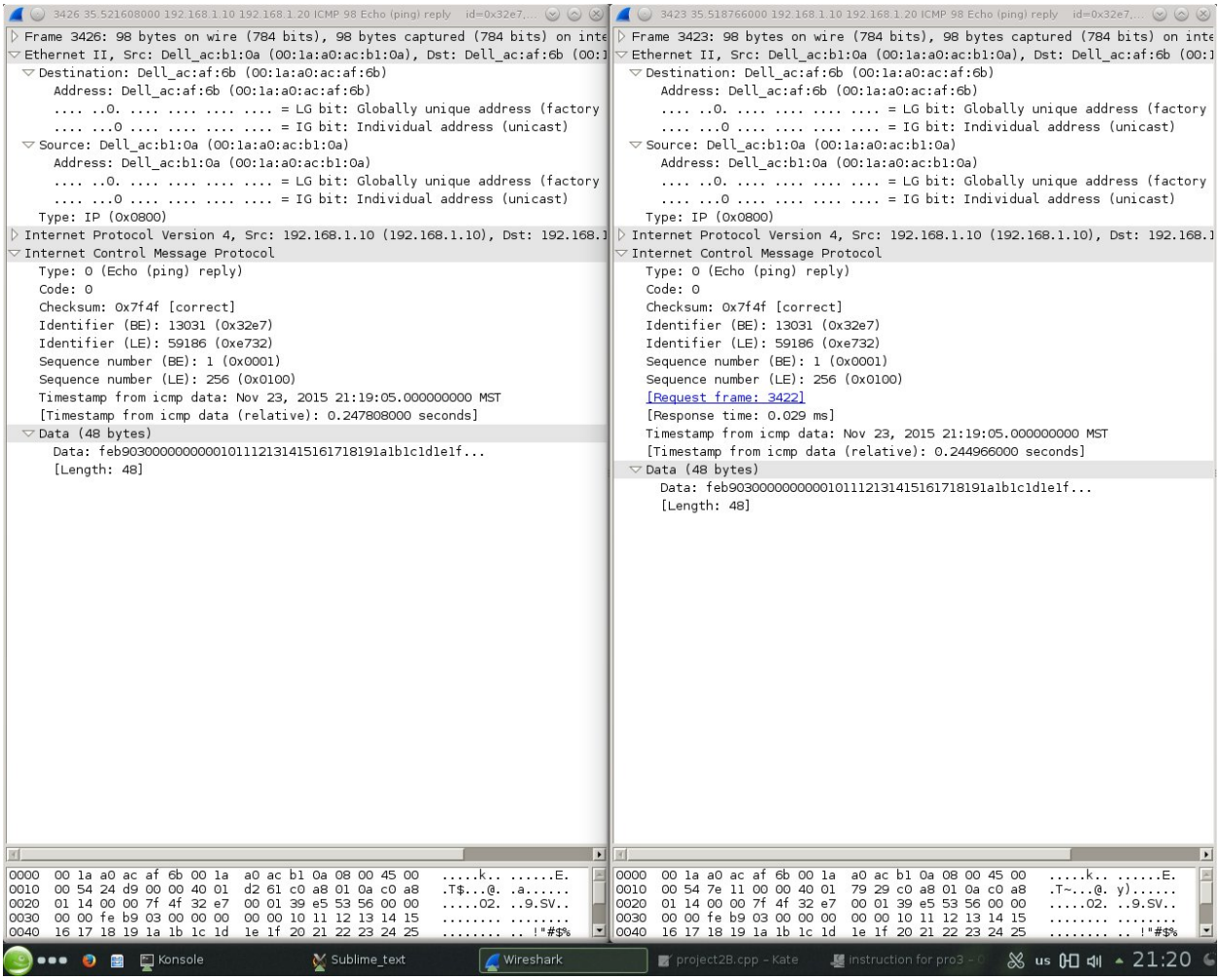
```
^C
netlab10:~/Desktop/Project3 # g++ -lpthread *.cpp; ./a.out
Desired IP address:
192.168.1.20
got icmp reply
sequence number: 1
yay received reply to our request
Desired IP address:
```

2.3 Replying to an ICMP request

Our code also replies to PING requests sent by other computers, we used wireshark to ensure this was working correctly.



In detail below, we can see that our outgoing packet is virtually same as the packet automatically sent by the computer.



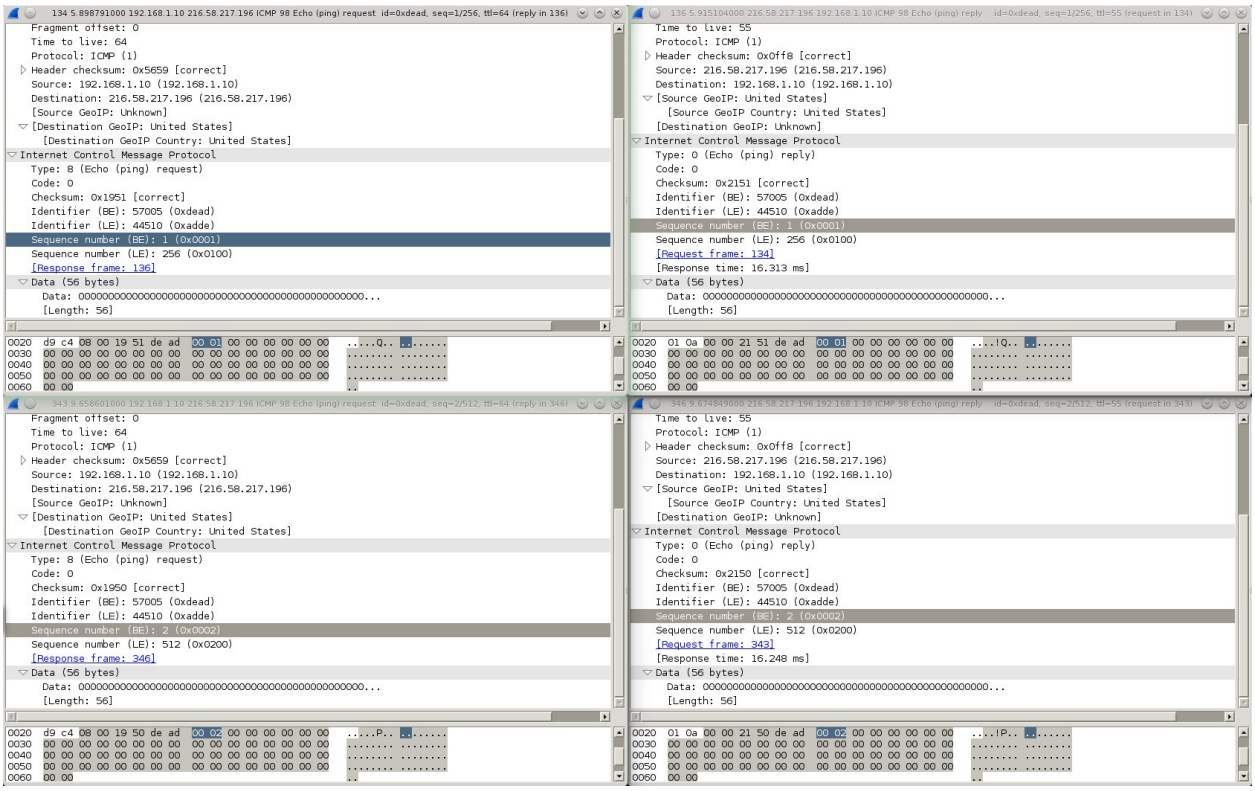
2.4 Seeing a duplicate reply

When we sent a duplicate reply correctly, the requesting computer acknowledged that it received two replies with a DUP identifier. This terminal output is shown below:

```
netlab20:~/Desktop/Parkin/Project3 # ping 192.168.1.10
PING 192.168.1.10 (192.168.1.10) 56(84) bytes of data.
64 bytes from 192.168.1.10: icmp_seq=1 ttl=64 time=0.136 ms
64 bytes from 192.168.1.10: icmp_seq=1 ttl=64 time=7.13 ms (DUP!)
64 bytes from 192.168.1.10: icmp_seq=2 ttl=64 time=0.136 ms
64 bytes from 192.168.1.10: icmp_seq=2 ttl=64 time=0.431 ms (DUP!)
64 bytes from 192.168.1.10: icmp_seq=3 ttl=64 time=0.135 ms
64 bytes from 192.168.1.10: icmp_seq=3 ttl=64 time=0.357 ms (DUP!)
64 bytes from 192.168.1.10: icmp_seq=4 ttl=64 time=0.135 ms
64 bytes from 192.168.1.10: icmp_seq=4 ttl=64 time=0.364 ms (DUP!)
```

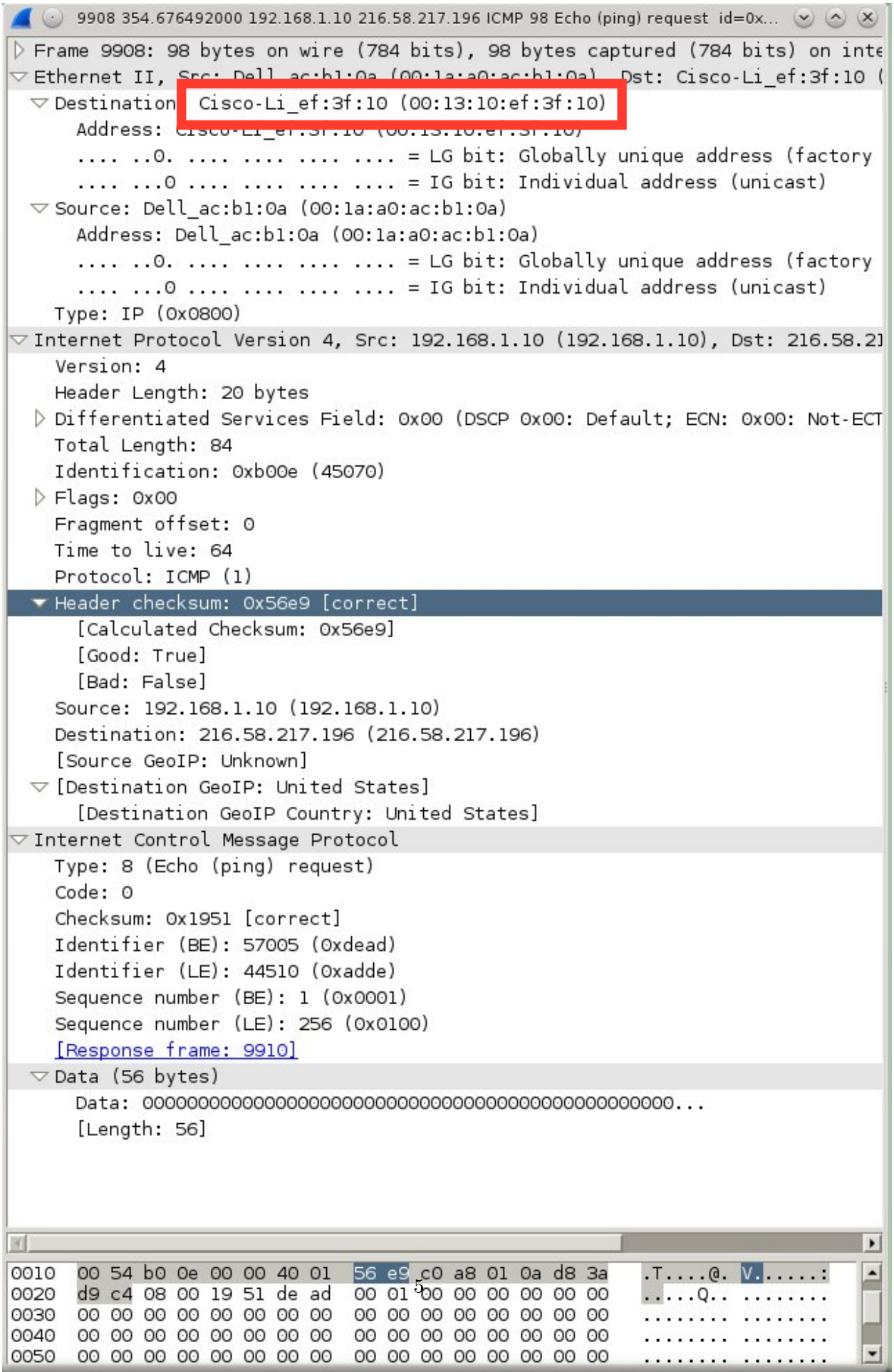
2.5 Sequencing the outgoing requests

When we sent out ICMP echo requests, we incremented the ICMP sequence number field and kept track of which ones we sent out. Once a reply was received, that sequence number was made available to use again. The screenshot below shows the first and second sequence numbers and the corresponding replies.



2.6 Pinging outside of network

We also were able to ping google, by using its ip address. We can see on wireshark, that the MAC address in the ether frame was changed to the cisco router:


9908 354.676492000 192.168.1.10 216.58.217.196 ICMP 98 Echo (ping) request id=0x...
Frame 9908: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on inte
Ethernet II, Src: Dell_ac:b1:0a (00:1a:a0:ac:b1:0a), Dst: Cisco-Li_ef:3f:10 (
Destination: Cisco-Li_ef:3f:10 (00:13:10:ef:3f:10)
Address: Cisco-Li_ef:3f:10 (00:13:10:ef:3f:10)
.... ..0. .... = LG bit: Globally unique address (factory
....0 .... = IG bit: Individual address (unicast)
Source: Dell_ac:b1:0a (00:1a:a0:ac:b1:0a)
Address: Dell_ac:b1:0a (00:1a:a0:ac:b1:0a)
.... ..0. .... = LG bit: Globally unique address (factory
....0 .... = IG bit: Individual address (unicast)
Type: IP (0x0800)
Internet Protocol Version 4, Src: 192.168.1.10 (192.168.1.10), Dst: 216.58.21
Version: 4
Header Length: 20 bytes
Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT
Total Length: 84
Identification: 0xb00e (45070)
Flags: 0x00
Fragment offset: 0
Time to live: 64
Protocol: ICMP (1)
Header checksum: 0x56e9 [correct]
[Calculated Checksum: 0x56e9]
[Good: True]
[Bad: False]
Source: 192.168.1.10 (192.168.1.10)
Destination: 216.58.217.196 (216.58.217.196)
[Source GeoIP: Unknown]
[Destination GeoIP: United States]
[Destination GeoIP Country: United States]
Internet Control Message Protocol
Type: 8 (Echo (ping) request)
Code: 0
Checksum: 0x1951 [correct]
Identifier (BE): 57005 (0xdead)
Identifier (LE): 44510 (0xadde)
Sequence number (BE): 1 (0x0001)
Sequence number (LE): 256 (0x0100)
[Response frame: 9910]
Data (56 bytes)
Data: 000000000000000000000000000000000000000000000000...
[Length: 56]
0010 00 54 b0 0e 00 00 40 01 56 e9 c0 a8 01 0a d8 3a .T....@. V.....:
0020 d9 c4 08 00 19 51 de ad 00 01 00 00 00 00 00 00 .....Q.. ....
0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....

3 Conclusions

In this lab, I was able implement many computer science aspects to achieve this ICMP protocol. It has helped me more fully understand the network layer.

4 The code

4.1 Function and Thread Explainations

1. void* receive_frame(void* arg);
   (a) This is a thread that captures all ethernet frames and filters out the important ones.
   (b) This code gets ARP and IP packets and puts them on a message queue
2. void* ip_checker(void* arg);
   (a) This thread takes IP packets off of the message queue and verifies them.
   (b) Verification is done by validating the checksum field
   (c) After verification, any packets that are ICMP are processed further
3. bool icmp_checker(void* ip_data, int size);
   (a) This function takes an ICMP packet and verifies its checksum
   (b) If it is correct, it returns true
4. void process_icmp(void* ip_data, int size, octet* dest, void* id);
   (a) This receives valid ICMP packets and processes them.
   (b) If the packet was an 'echo reply', then it clears the open sequence number.
   (c) If the packet was an 'echo request', then it sends a reply.
5. void find_mac(void* mac,void* ip);
   (a) This function receives a desired IP address to send a PING to and finds the MAC address of the destination.
   (b) if the destination is outside of the local network, then the router's MAC is used.
6. void* send_icmp(void *arg);
   (a) This is a thread that asks for the user to enter an IP address.
   (b) It then sends an echo request to the entered IP address.
   (c) If it was successful, it prompts for a new IP address.
7. void arp_mac(void* mac, void* ip);
   (a) This function uses an ARP message to find the corresponding MAC address of an IP address.
8. bool send_ip(void* ip_data, int size, octet* dest, octet* idpart);
   (a) This takes data to be sent and adds an IP and Ethernet header.
   (b) It then sends this data out on the network.

4.2 proj3.cpp (All of the code)

```
#include "util.h"
#include "frameio.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <list>
#include <iostream>
#include <arpa/inet.h>
#include <time.h>
#include <vector>

// Project 3 Samuel Bhushan & Hailey Parkin
// Step 1 Identify ICMP Packets

std::vector<unsigned short> sequences;
bool vector_in_use = false;
short seq = 0;

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
};
struct arp_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[32];      // payload
};
struct ip_frame
{
        octet version_ihl;
        octet dscp_ecn;
        octet total_length[2];
        octet ID[2];
        octet flags_frag[2];
        octet ttl;
        octet protocol;
        octet header_checksum[2];
        octet src_ip[4];
        octet dst_ip[4];
        octet data[1480];

};
struct icmp_frame
{
         octet type;
```

```
        octet code;
        octet checksum[2];
        octet restofheader[4];
        octet data[1472];
};
struct arp_frame
{
        octet mac_type[2];
        octet protocol_type[2];
        octet mac_len;
        octet protocol_len;
        octet opcode[2];
        octet sender_mac[6];
        octet sender_ip[4];
        octet target_mac[6];
        octet target_ip[4];
};
frameio net;
message_queue ip_queue;
message_queue arp_queue;
pthread_t threads;
void* receive_frame(void* arg);
void* ip_checker(void* arg);
bool icmp_checker(void* ip_data, int size);
void process_icmp(void* ip_data, int size, octet* dest, void* id);
void find_mac(void* mac,void* ip);
void* send_icmp(void *arg);
void arp_mac(void* mac, void* ip);
bool send_ip(void* ip_data, int size, octet* dest, octet* idpart);

int main()
{
        net.open_net("enp3s0");
        // Thread for putting packets on queues
        pthread_create(&threads,NULL,receive_frame,NULL);
        pthread_create(&threads,NULL,ip_checker,NULL);
        pthread_create(&threads,NULL,send_icmp,NULL);
        for(;;)
                sleep(1);
}

void* receive_frame(void* arg)
{
        //std::cout << "RECIEVE FRAME THREAD STARTED" << std::endl;
        ether_frame buffer;
        while(1)
        {
           int n = net.recv_frame(&buffer,sizeof(buffer));
           if ( n < 42 ) continue; // bad frame!
            switch ( buffer.prot[0]<<8 | buffer.prot[1] )
        {
          case 0x800:
            ip_queue.send(PACKET,buffer.data,n);
                break;
```

```
            case 0x806:
                    arp_queue.send(PACKET, buffer.data,n);
                break;
                    default:
                            break;
        }
        }
}
void* ip_checker(void *arg)
{
        ip_frame received;
        int ip_size;
        int sum;
        octet initial_chksum[2];
        event_kind pck = PACKET;
        //std::cout << "ICMP FILTER THREAD STARTED" << std::endl;
        while(1)
        {
                ip_size = ip_queue.recv(&pck,&received,sizeof(received));

                // VALIDATE IP PACKET
                initial_chksum[0] = received.header_checksum[0];
                initial_chksum[1] = received.header_checksum[1];
                received.header_checksum[0]= 0;
                received.header_checksum[1] = 0;
                sum = chksum((octet*)&received,20,0);
                received.header_checksum[0] = ~sum >> 8;
        received.header_checksum[1] = ~sum & 0xff;
        if(!(initial_chksum[0] == received.header_checksum[0] &&
                initial_chksum[1] == received.header_checksum[1]))
        {
                        //std::cout << "Bad IP checksum " << std::endl;
                        continue;
                }
        // Check if ICMP
                if(!(received.protocol == 1))
                {
                        //std::cout << "Wrong Protocol " << std::endl;
                        continue;
                }
                // Validate the ICMP packet
                if(!icmp_checker(received.data, 64))
                {
                        //std::cout << "Bad ICMP checksum " << std::endl;
                        continue;
                }
                process_icmp(received.data,64,received.src_ip,&received.ID);
                printf("ICMP Packet received \n");
        }
}

bool icmp_checker(void* ip_data, int size)
{
        int sum;
```

```
        octet initial_chksum[2];

        icmp_frame* icmp = new icmp_frame;
        icmp = (icmp_frame*)ip_data;

        initial_chksum[0] = icmp->checksum[0];
        initial_chksum[1] = icmp->checksum[1];
        icmp->checksum[0]= 0x00;
        icmp->checksum[1] = 0x00;
        sum = chksum((octet*)icmp,size,0);
        icmp->checksum[0] = ~sum >> 8;
    icmp->checksum[1] = ~sum & 0xff;
    if(initial_chksum[0] == icmp->checksum[0] &&
        initial_chksum[1] == icmp->checksum[1])
    {
        return true;
    }
    return false;

}
void process_icmp(void* ip_data, int size, octet* dest, void* id)
{
        octet* idpart =(octet*) id;
        icmp_frame* icmp = (icmp_frame*)ip_data;
        icmp_frame icmp_out;
        int sum;
        unsigned short recv_seq;
        if(icmp->type == 0x08)
        {
                //printf("got icmp request\n");
                //respond to request
                icmp_out.type = 0x00; // zero for echo reply(ping)
                icmp_out.code = 0x00;
                for (int i = 0; i < 4; ++i)
                {
                        icmp_out.restofheader[i] = icmp->restofheader[i]; //not using this
                }
                memcpy(&icmp_out.data,&icmp->data,56);

                icmp_out.checksum[0]=0;
                icmp_out.checksum[1]=0;
                sum = chksum((octet*)&icmp_out,64,0);
                icmp_out.checksum[0]=~sum >> 8;
                icmp_out.checksum[1]=~sum & 0xff;

                send_ip(&icmp_out, 64, dest, idpart);
        }
        else if(icmp->type == 0x0)
        {
                //printf("got icmp reply\n");
                recv_seq = icmp->restofheader[2]*256 + icmp->restofheader[3];
                //memcpy(&recv_seq,&icmp->restofheader,2);
                printf("sequence number: %d\n", recv_seq);
                while(vector_in_use)
```

```
                        sleep(1); //wait for vector
                vector_in_use = true;
                for (int i = 0; i < sequences.size(); ++i)
                {
                        if(sequences[i] == recv_seq)
                        {
                                sequences.erase(sequences.begin()+i);
                                std::cout << "yay received reply to our request" << std::e
                                break;
                        }
                }
                vector_in_use = false;

        }

}
bool send_ip(void* ip_data, int size, octet* dest, octet* idpart)
{
        ip_frame packet;
        ether_frame ether;
        int checksum;


        // Make the IP header

        memcpy(&packet.data, ip_data,size);
        packet.version_ihl = 0x45;
        packet.dscp_ecn = 0x00;

        packet.total_length[0] =(size + 20)>> 8;
        packet.total_length[1] = (size+20) & 0xff;

        memcpy(&packet.ID,&idpart,2);
        packet.flags_frag[2] =0x02;
        packet.ttl = 64;
        packet.protocol = 0x01;

        packet.src_ip[0] = 0xC0;
        packet.src_ip[1] = 0xA8;
        packet.src_ip[2] = 0x01;
        packet.src_ip[3] = 0x0A;

        packet.dst_ip[0] = dest[0];
        packet.dst_ip[1] = dest[1];
        packet.dst_ip[2] = dest[2];
        packet.dst_ip[3] = dest[3];


        // Calculate IP Checksum and add it
        packet.header_checksum[0]=0;
        packet.header_checksum[1]=0;
        checksum = chksum((octet*)&packet,20,0);
        packet.header_checksum[0]=~checksum >> 8;
        packet.header_checksum[1]=~checksum & 0xff;
```

```
        // Stuff into etherframe and send
        find_mac(&ether.dst_mac,packet.dst_ip);
        memcpy(ether.src_mac,net.get_mac(),sizeof(ether.src_mac));
        ether.prot[0] = 0x08;
        ether.prot[1] = 0x00;
        memcpy(ether.data,&packet,84); // copy IP into ether data

        net.send_frame(&ether,84+14);
        return 1;
}
void* send_icmp(void *arg)
{
        bool result;

        icmp_frame icmp_out;
        unsigned char* out = new unsigned char[65];
        char user_input[15];
        octet target_ip[4];
        struct sockaddr_in hexip;
        int sum;

        while(1)
        {
                // Get and convert IP
                std::cout << "Desired IP address: " << std::endl;
                std::cin >> user_input;
                inet_pton(AF_INET,user_input,&(hexip.sin_addr));
                for (int i = 0; i < 4; ++i)
                        target_ip[i] = (hexip.sin_addr.s_addr >>(8*i)) & 0xff;

                // Construct ICMP
                icmp_out.type = 0x08; // eight for echo request(ping)
                icmp_out.code = 0x00;

                icmp_out.restofheader[0] = 0xde;
                icmp_out.restofheader[1] = 0xad;

                //add sequence number
                while(vector_in_use)
                        sleep(1);   //wait for vector to be free
                vector_in_use = true;
                seq++;
                sequences.push_back(seq);
                icmp_out.restofheader[2] = seq >> 8;
                icmp_out.restofheader[3] = seq & 0xff;
                vector_in_use = false;

                //calculate checksum
                icmp_out.checksum[0]=0;
                icmp_out.checksum[1]=0;
                sum = chksum((octet*)&icmp_out,64,0);
                icmp_out.checksum[0]=~sum >> 8;
                icmp_out.checksum[1]=~sum & 0xff;
```

```
                // Send out ICMP
                memcpy(out,&icmp_out,64);
                octet id[2] = {0xbe,0xef};
                result = send_ip(out,64,target_ip,id);

                if(result);
                        //std::cout << "ICMP send success" << std::endl;
                else
                        std::cout << "ICMP send failure" << std::endl;
                sleep(1);
        }
}
void find_mac(void* mac,void* ip)
{
        octet* addr = (octet*)ip;
        bool is_lan = true;
        octet myip[4] = {0xc0,0xA8,0x01,0xA};
        octet route_ip[4] = {0xc0,0xA8,0x01,0x1};
        octet target_mac[6] = {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};
        for (int i = 0; i < 3; ++i)
        {
                if(addr[i] != myip[i])
                        is_lan = false;
        }
        if(is_lan)
        {
                // Get the mac
                arp_mac(&target_mac,addr);
        }
        else
        {
                // get the router's mac
                arp_mac(&target_mac,route_ip);
        }
        memcpy(mac,target_mac,6);
}
void arp_mac(void* mac, void* ip)
{
        octet myip[4] =  {0xC0,0xA8,0x01,0xA};
        octet* target_mac = (octet*)mac;
        octet* target_ip = (octet*)ip;
        // Make ARP packet
        arp_frame message;
        arp_ether_frame ether_message;
        message.mac_type[0] = 0x00;
        message.mac_type[1] = 0x01;
        message.protocol_type[0] = 0x08;
        message.protocol_type[1] = 0x00;
        message.mac_len = 6;
        message.protocol_len = 4;
        message.opcode[1] = 0x01; //request ARP

        memcpy(&message.sender_mac,net.get_mac(), sizeof(message.sender_mac));
```

```
        memcpy(&message.sender_ip,myip,sizeof(myip));
        memcpy(&message.target_mac, target_mac, sizeof(message.target_mac));
        memcpy(&message.target_ip,target_ip,sizeof(message.target_ip));

        //Ether Frame Part
        memcpy(ether_message.dst_mac,message.target_mac,sizeof(message.target_mac));
        memcpy(ether_message.src_mac,message.sender_mac,sizeof(message.sender_mac));
        ether_message.prot[0] = 0x08;
        ether_message.prot[1] = 0x06;
        memcpy(ether_message.data, &message, sizeof(message));

        //std::cout << "Sending ARP!" << std::endl;
        net.send_frame(&ether_message,sizeof(ether_message));

        arp_frame received;
        bool found = false;
        bool same;
        int request_size;
        event_kind type = PACKET;
        while(!found)
        {
                request_size = arp_queue.recv(&type,(void*)&received,(int)sizeof(received))
                //printf("received size: %d!\n",request_size);
                same = true;
                for (int i = 0; i < 4; ++i)
                {
                        if(received.sender_ip[i] != *(target_ip+i))
                                same = false;
                }
                if(same)
                {
                        //found the arp reply
                        //std::cout << "Found ARP!" << std::endl;
                        memcpy(mac,&received.sender_mac,sizeof(received.sender_mac));
                        found = true;
                }

        }
}
```