## Objective

To understand the IP packet structure and implement ICMP echo.

## Structural Overview

The machine will run code that initiates an ICMP echo request to two IP addresses: one in the same subnet and one outside the subnet. It is expected that the program will determine the destination MAC address via ARP developed in the previous lab, directing IP addresses out of the subnet towards a hard-coded default gateway. Another machine will ping the host running the program, to which it is expected to respond appropriately.

## Simulation

```
IP message received, protocol: 1
ICMP message received
IP message received, protocol: 1
ICMP message received
IP message received, protocol: 1
ICMP message received
IP message received, protocol: 17
IP message received, protocol: 17
IP message received, protocol: 17
IP message received, protocol: 17
IP message received, protocol: 1
ICMP message received
IP message received, protocol: 1
ICMP message received
IP message received, protocol: 17
IP message received, protocol: 6
IP message received, protocol: 6
IP message received, protocol: 6
IP message received, protocol: 6
IP message received, protocol: 17
IP message received, protocol: 17
IP message received, protocol: 17
IP message received, protocol: 17
IP message received, protocol: 6
IP message received, protocol: 17
IP message received, protocol: 17
IP message received, protocol: 6
IP message received, protocol: 6
IP message received, protocol: 1
ICMP message received
IP message received, protocol: 1
ICMP message received
IP message received, protocol: 6
IP message received, protocol: 6
```

Figure 1 - Parsing ICMP Requests

```
netlab30:/home/A01283897/ece5600/lab3 # ./lab3
Press enter to ping ...
Sending 0xdeadbeef 192.168.1.1
Press enter to ping ...
Sending 0xdeadbeef 192.168.1.1
Press enter to ping ...^C
```

Figure 2 - Console output ICMP echo request with ARP

| | | | | |
|---|---|---|---|---|
| Dell_ac:ad:73 | ARP | Broadcast | Who has 192.168.1.1? Tell 192.168.1.30 | |
| Cisco-Li_f5:0c:ac | ARP | Dell_ac:ad:73 | 192.168.1.1 is at 00:1c:10:f5:0c:ac | |
| 192.168.1.30 | ICMP | 192.168.1.1 | Echo (ping) request | id=0x0001, seq=0/0, ttl=64 (no response found!) |
| 192.168.1.1 | ICMP | 192.168.1.30 | Echo (ping) reply | id=0x0001, seq=0/0, ttl=64 (request in 3) |
| 192.168.1.30 | ICMP | 192.168.1.1 | Echo (ping) request | id=0x0002, seq=0/0, ttl=64 (reply in 6) |
| 192.168.1.1 | ICMP | 192.168.1.30 | Echo (ping) reply | id=0x0002, seq=0/0, ttl=64 (request in 5) |

Figure 3 - Wireshark output ICMP echo request with ARP

```
netlab30:/home/A01283897/ece5600/lab3 # ./lab3
Press enter to ping ...
Sending 0xdeadbeef 172.217.6.78
Press enter to ping ...^C
```

Figure 4 - Console output ICMP echo out of subnet

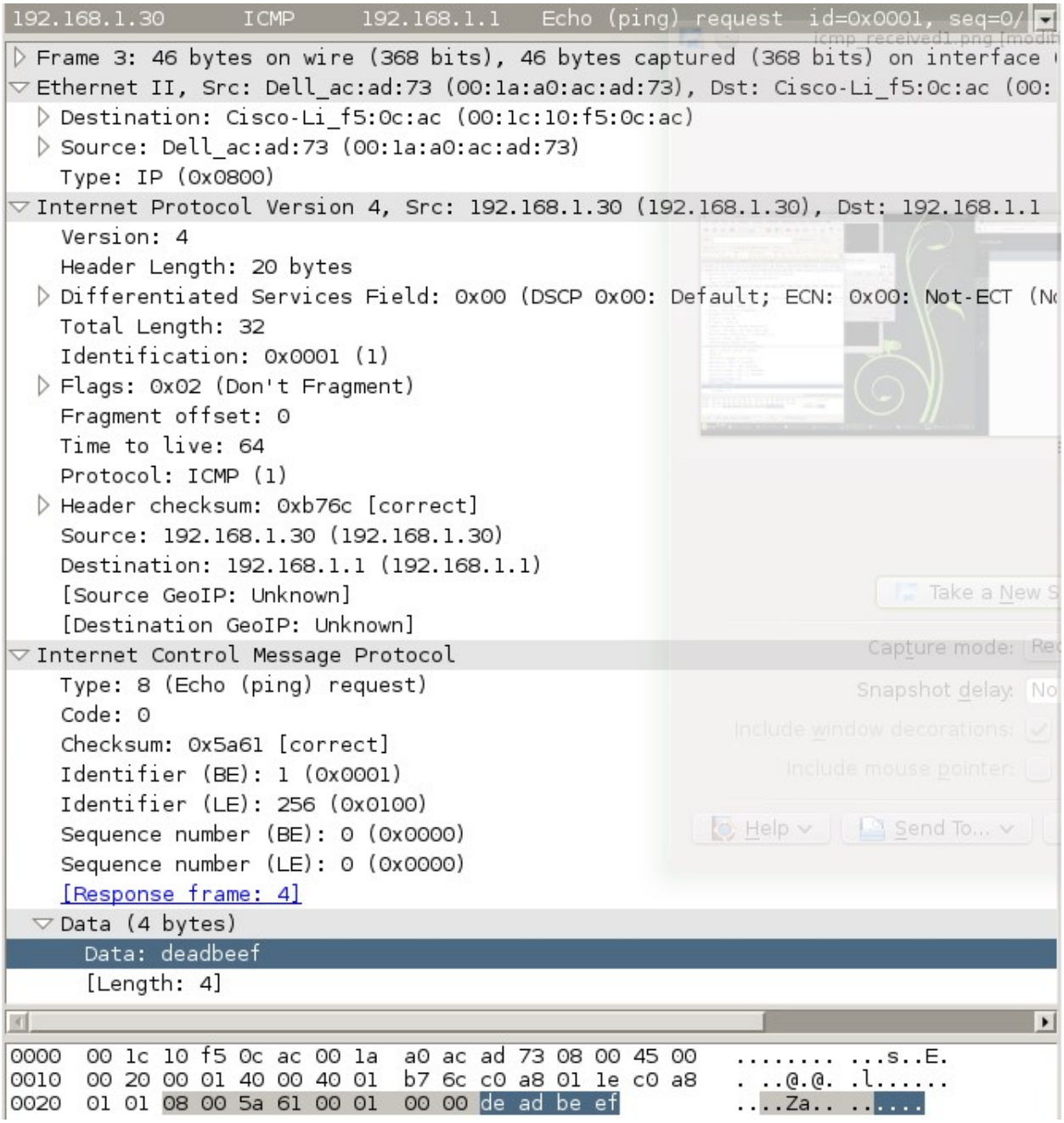


Figure 5 - Custom echo request within the subnet

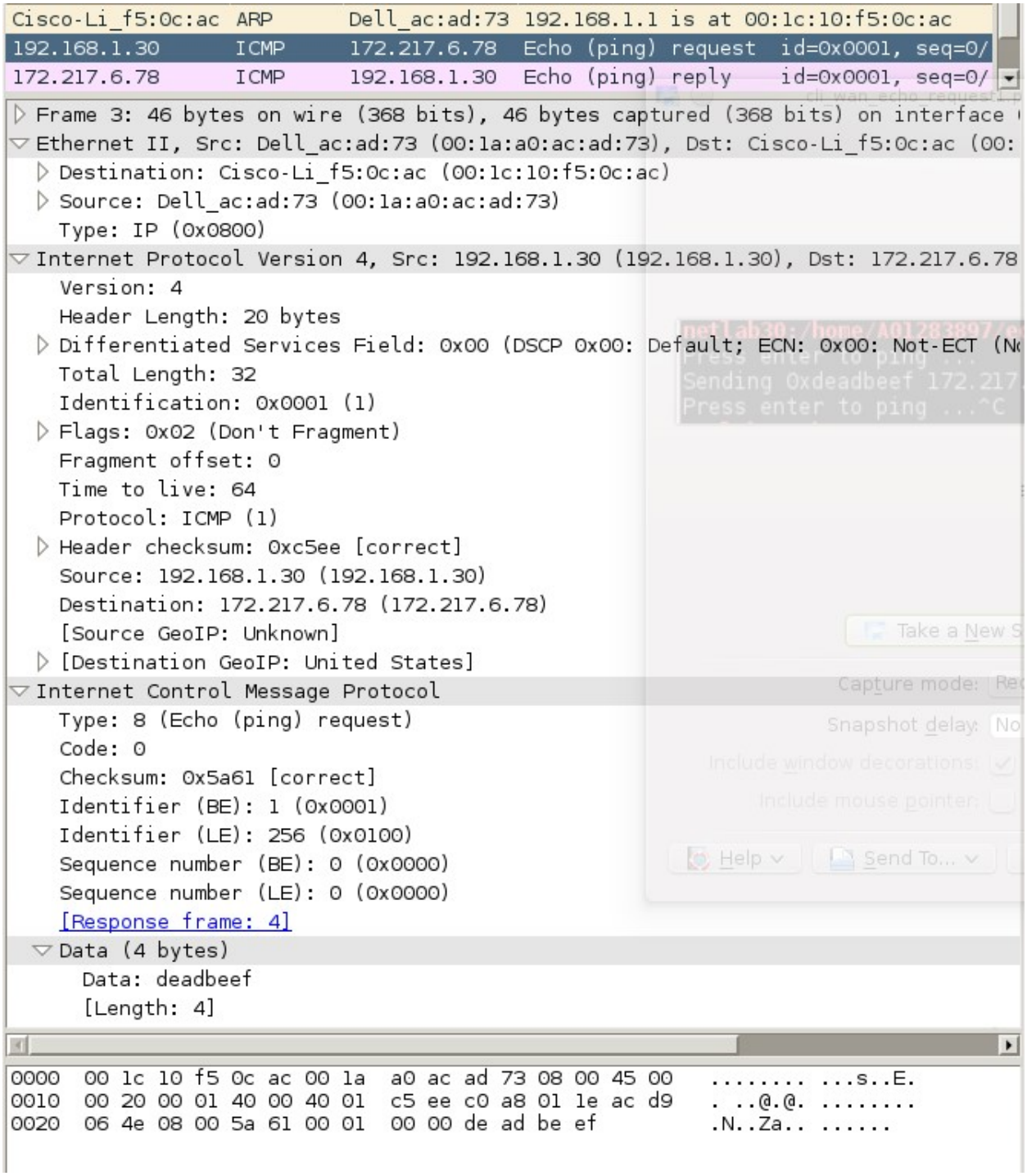


Figure 6 - Custom echo request out of subnet

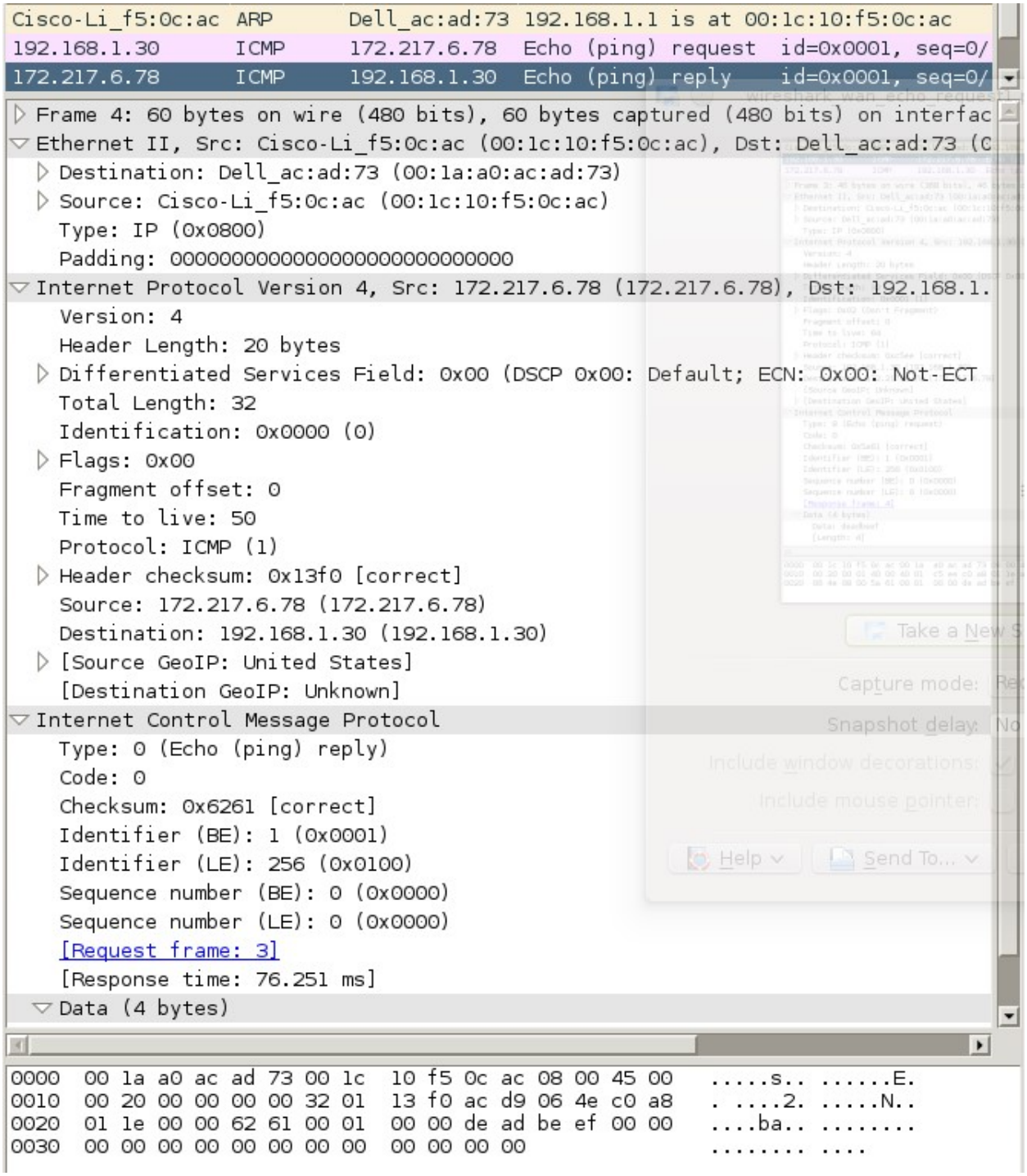


Figure 7 - Google echoes deadbeef

```
netlab40:~ # ping 192.168.1.30
PING 192.168.1.30 (192.168.1.30) 56(84) bytes of data.
64 bytes from 192.168.1.30: icmp_seq=1 ttl=64 time=0.149 ms
64 bytes from 192.168.1.30: icmp_seq=1 ttl=64 time=0.170 ms (DUP!)
64 bytes from 192.168.1.30: icmp_seq=2 ttl=64 time=0.155 ms
64 bytes from 192.168.1.30: icmp_seq=2 ttl=64 time=0.178 ms (DUP!)
64 bytes from 192.168.1.30: icmp_seq=3 ttl=64 time=0.156 ms
64 bytes from 192.168.1.30: icmp_seq=3 ttl=64 time=0.179 ms (DUP!)
64 bytes from 192.168.1.30: icmp_seq=4 ttl=64 time=0.155 ms
64 bytes from 192.168.1.30: icmp_seq=4 ttl=64 time=0.177 ms (DUP!)
^C
```

Figure 8 - Console ping request from remote device

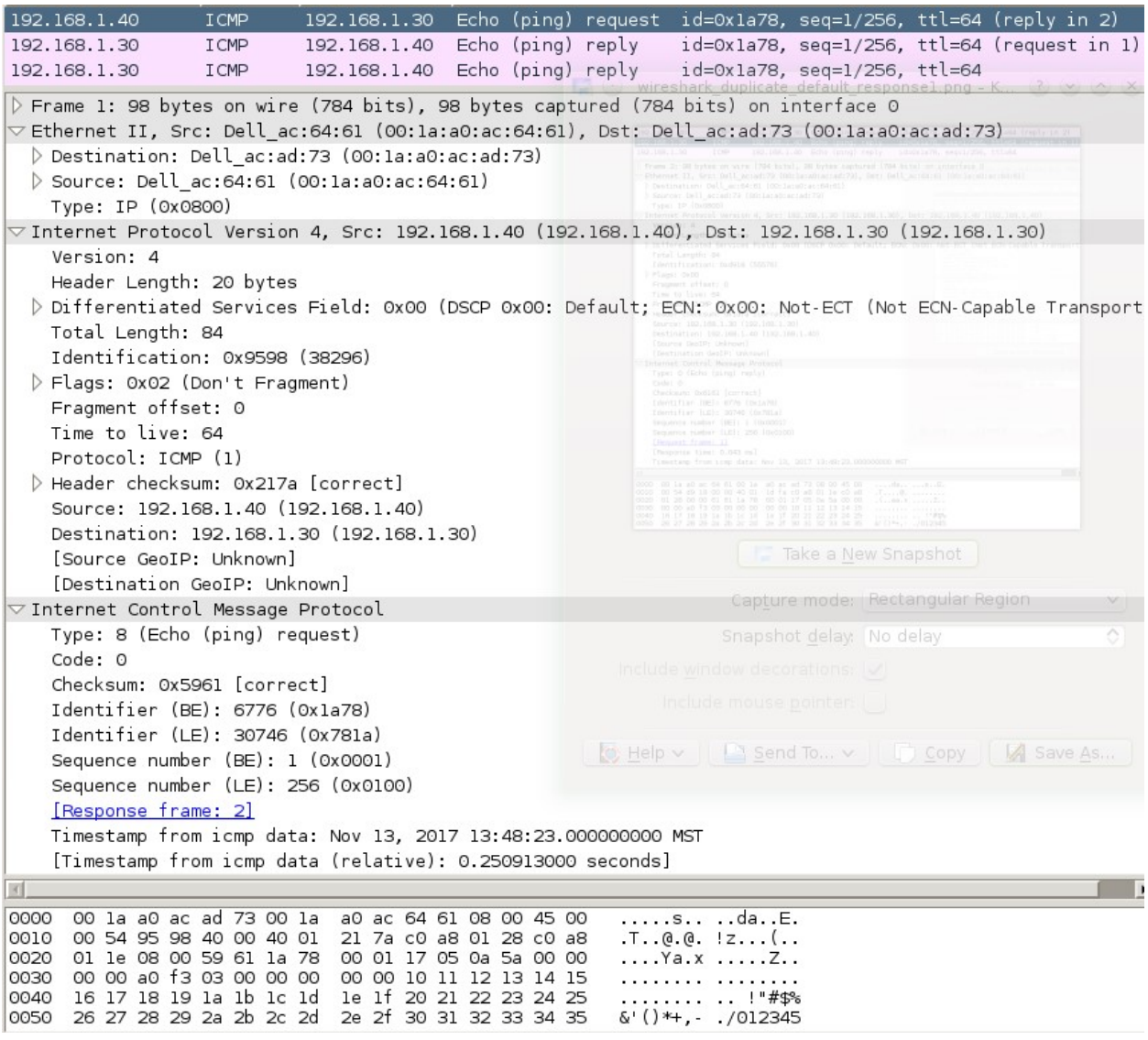


Figure 9 - An incoming ping request

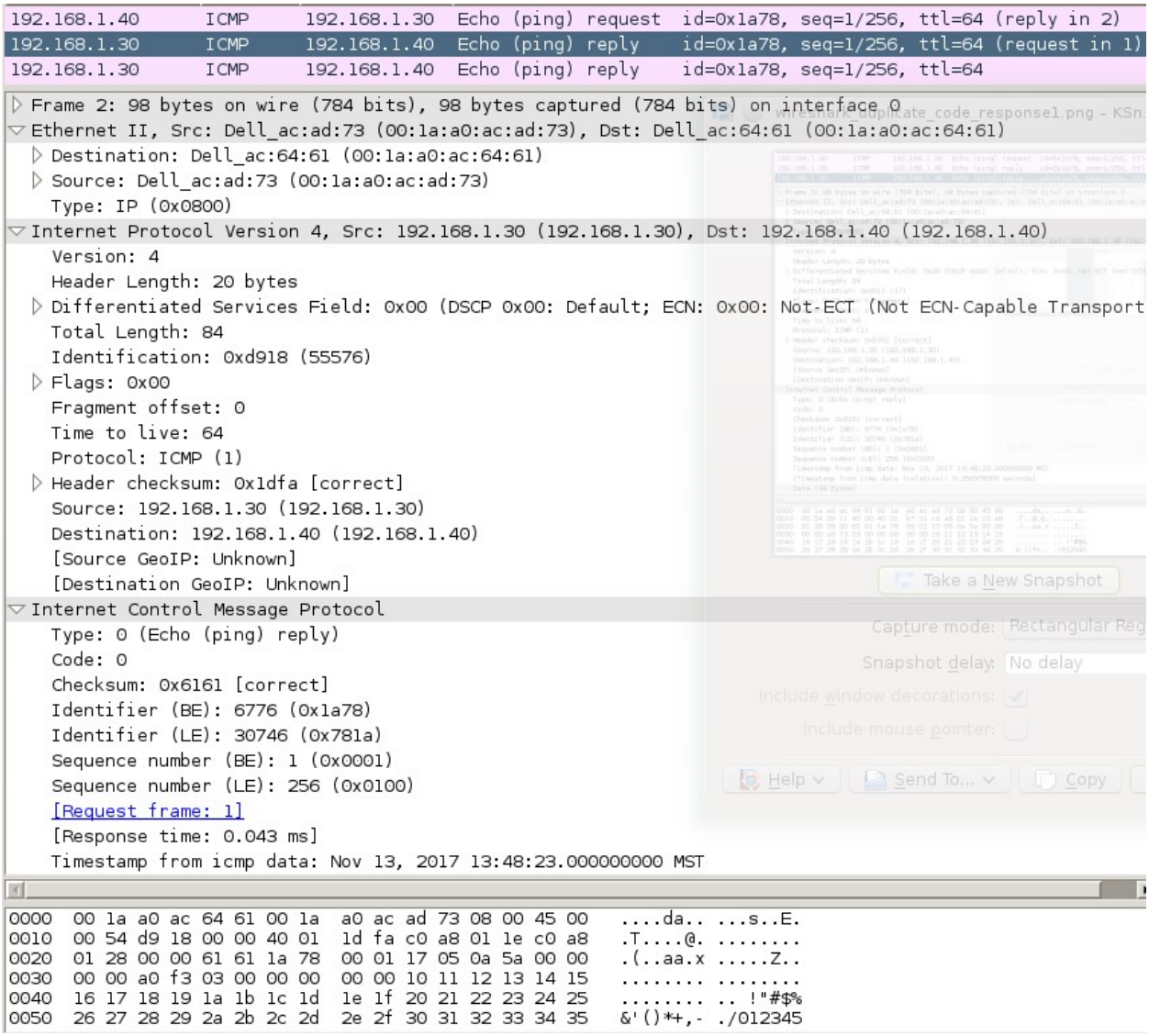


Figure 10 - Automatic echo response from machine

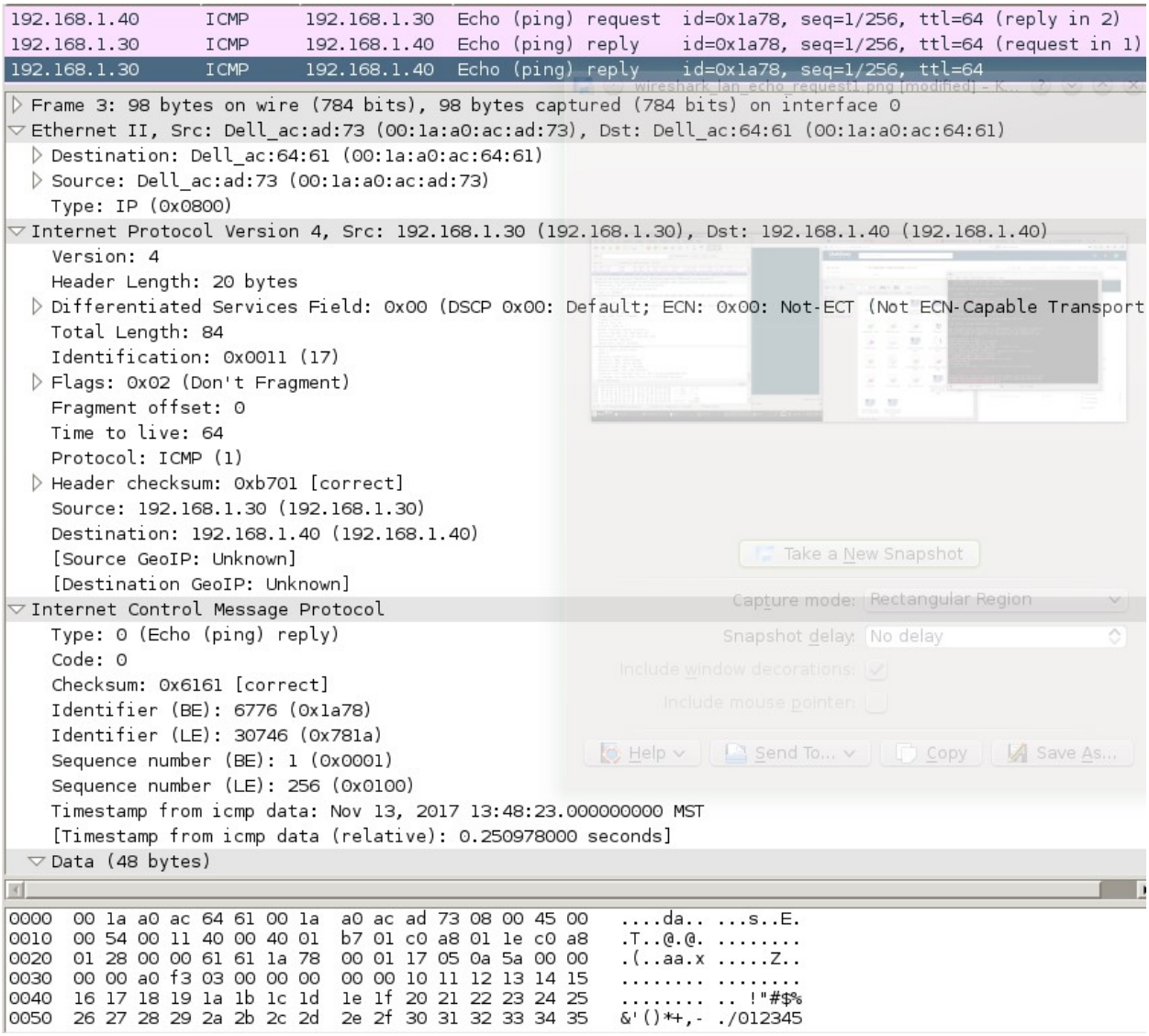


Figure 11 - Custom echo response from program

```
192.168.1.40    ICMP    192.168.1.30    Echo (ping) request  id=0x1a78, seq=2/512, ttl=64 (reply in 5)
192.168.1.30    ICMP    192.168.1.40    Echo (ping) reply    id=0x1a78, seq=2/512, ttl=64 (request in 4)
192.168.1.30    ICMP    192.168.1.40    Echo (ping) reply    id=0x1a78, seq=2/512, ttl=64

▷ Frame 6: 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)
  ▷ Source: Dell_ac:ad:73 (00:1a:a0:ac:ad:73)
    Type: IP (0x0800)
▽ Internet Protocol Version 4, Src: 192.168.1.30 (192.168.1.30), Dst: 192.168.1.40 (192.168.1.40)
    Version: 4
    Header Length: 20 bytes
  ▷ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport
    Total Length: 84
    Identification: 0x0012 (18)
  ▷ Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
  ▷ Header checksum: 0xb700 [correct]
    Source: 192.168.1.30 (192.168.1.30)
    Destination: 192.168.1.40 (192.168.1.40)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
▽ Internet Control Message Protocol
    Type: 0 (Echo (ping) reply)
    Code: 0
    Checksum: 0x9261 [correct]
    Identifier (BE): 6776 (0x1a78)
    Identifier (LE): 30746 (0x781a)
    Sequence number (BE): 2 (0x0002)
    Sequence number (LE): 512 (0x0200)
    Timestamp from icmp data: Nov 13, 2017 13:48:24.000000000 MST
    [Timestamp from icmp data (relative): 0.250678000 seconds]
  ▽ Data (48 bytes)

0000  00 1a a0 ac 64 61 00 1a  a0 ac ad 73 08 00 45 00   ....da.. ...s..E.
0010  00 54 00 12 40 00 40 01  b7 00 c0 a8 01 1e c0 a8   .T..@.@. ........
0020  01 28 00 00 92 61 1a 78  00 02 18 05 0a 5a 00 00   .(...a.x .....Z..
0030  00 00 6e f2 03 00 00 00  00 00 10 11 12 13 14 15   ..n..... ........
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
```

Figure 12 - Custom response for next request in the ICMP sequence

## Results

The results you can see in figure one are a result of commented out code that would print to the console upon reception of any ICMP message. For custom ICMP echo requests used later, the message deadbeef was used.

Figures 2 and 3 correspond to sending an ICMP echo request before and after a cached ARP entry to demonstrate the automatic use of ARP upon requesting an unknown IP within the subnet. Figure 5 shows the packet used for the ARP request within the subnet.

Figures 4 and 6 demonstrate the output for an ICMP message request out of the subnet. As it happens with Figures 2, 3, and 5, I pinged the default gateway instead of another machine in the subnet (I promise it works for other devices as well…I left the lab before realizing this error and am unable to update with corrective screen captures)

Figure 7 is partially for fun, I got Google to say deadbeef. However, the fact it responded demonstrates correct implementation.

Figure 8 shows the output from a machine used in the final procedure for this lab: responding to an ICMP echo request. Note the “(DUP)” appearing as a result of our code responding in addition to the machine’s default response.

Figure 9 shows the packet data for the ping request from the remote machine.

Figure 10 shows the packet data for the machine’s default response.

Figure 11 shows the packet data for our custom code’s response.

Figure 12 demonstrates an increasing sequence number in the echo requests, to which our program correctly responds.

## Notes

I renamed given code samples from the instructor to match their language (.hpp instead of .h) and util to message_queue as a more descriptive name (at some point I was considering adding my own util.hpp). Additionally I renamed the type 'octet' to 'byte'

## makefile

```
lab3: main.cpp net.hpp chksum.o message_queue.o frameio.o
    g++ -std=c++11 main.cpp chksum.o message_queue.o frameio.o -lpthread -g -O0 -o lab3

chksum.o: chksum.c
    g++ chksum.c -c -o chksum.o

message_queue.o: message_queue.cpp message_queue.hpp
    g++ message_queue.cpp -c -o message_queue.o

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

clean:
    rm *.o
    rm lab3
```

## net.hpp

```
#pragma once

typedef unsigned char byte;
extern int chksum(byte* s, int n, int i);

// macro converts byte[] into ushort, uint
#define BUFF_UINT16(buff, i) (buff[i + 0] << 8 | buff[i + 1] << 0)
#define BUFF_UINT32(buff, i) (buff[i + 0] << 24 | buff[i + 1] << 16 | buff[i + 2] << 8 | buff[i + 3] << 0)

struct ipmac
{
    byte mac[6];
    byte ip[4];
};

struct net_device
{
    union
    {
        ipmac arp_cache_self;
        struct
        {
            byte mac[6];
            byte ip[4];
        };
    };
    byte subnet_mask[4];
    byte default_gateway[4];
};

//-------------------------------------------------------------------------+
// Ethernet 802.3/DIX frames                                              |
struct ether_header
{
    byte dst[6];
    byte src[6];
    union
    {
        byte len[2];
        byte prot[2];
    };
};
struct ether_frame
{
    ether_header header;
    byte data[1500];
};

ether_frame* make_frame(byte* dst, unsigned short prot, byte* data, int n);
//-------------------------------------------------------------------------+
```

```
//-------------------------------------------------------------------------------+
// ARP                                                                           |
struct arp_header
{
    byte hwtype[2];
    byte prottype[2];
    byte hwlength;
    byte protlength;
    byte opcode[2];
};

struct arp_frame
{
    arp_header header;
    byte data[1500 - sizeof(arp_header)];
};
ipmac* retrieveArpCache(byte* value);
void saveArpCache(ipmac* value);
void pingARP(byte* ip);
//-------------------------------------------------------------------------------+

//-------------------------------------------------------------------------------+
// IP                                                                            |
struct ip_header
{
    byte ver_ihl;
    byte dscp;

    byte length[2];
    byte ident[2];

    byte frag[2];

    byte ttl;
    byte prot;
    byte crc[2];
    byte src[4];
    byte dst[4];
};

struct ip_frame
{
    ip_header header;
    byte data[1500 - sizeof(ip_header)];
};
void sendIPv4Packet(byte* ip, byte prot, byte* payload, int n);
//-------------------------------------------------------------------------------+

//-------------------------------------------------------------------------------+
// ICMP                                                                          |
struct icmp_header
{
    byte type;
    byte code;
    byte crc[2];
    union
    {
        byte header[4];
        struct
        {
            byte ident[2];
            byte seqno[2];
        } echo;
    };
};

struct icmp_frame
{
    icmp_header header;
    byte data[1500 - sizeof(ip_header) - sizeof(icmp_header)];
};
void pingICMP(byte* ip, byte* data, int n);
//-------------------------------------------------------------------------------+
```

```
#define ETHER_PROT_IPV4         0x0800
#define ETHER_PROT_ARP          0x0806

void arp_handler(byte* packet, int n, ether_header* header);
void ip_handler(byte* packet, int n, ether_header* header);

#define IPV4_PROT_ICMP          0x01
void icmp_handler(byte* packet, int n, ip_header* header);
```

# main.cpp

```
#include "frameio.hpp"
#include "message_queue.hpp"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <unordered_map>

#include "net.hpp"

// device name must be hard-coded
frameio net("enp3s0");

net_device me =
{
    0, 0, 0, 0, 0, 0, // mac copied at start of main()
    192, 168, 1, 30,  // ip              must be hard-coded
    255, 255, 255, 0, // subnet mask     must be hard-coded
    192, 168, 1, 1,   // default gateway must be hard-coded
};

std::unordered_map<int, ipmac*> arp_cache;
inline int hash_ip(byte* ip)
{
    static const int hash_mask = ~BUFF_UINT32(me.subnet_mask, 0);
    int ip4 = BUFF_UINT32(ip, 0);
    int key = ip4 & hash_mask;
    return key;
}
ipmac* retrieveArpCache(byte* ip)
{
    int key = hash_ip(ip);
    auto search = arp_cache.find(key);
    if (search != arp_cache.end()) {
        return search->second;
    }
    return NULL;
}
void saveArpCache(ipmac* value)
{
    ipmac* found = retrieveArpCache(value->ip);
    if (found == NULL)
    {
        // insert
        ipmac* copy = (ipmac*)malloc(sizeof(ipmac));
        memcpy(copy, value, sizeof(ipmac));
        int key = hash_ip(copy->ip);
        arp_cache.insert({key, copy});
    }
    else
    {
        // update
        memcpy(found, value, sizeof(ipmac));
    }
}

// message queue for the sending ether_frames
message_queue send_queue;
void* send_thread(void* args)
{
    int n;
    ether_frame frame;
    event_kind event;
    while(1)
    {
        n = send_queue.recv(&event, &frame, sizeof(ether_frame));
        net.send_frame(&frame, n);
    }
}
```

```
void* receive_thread(void* args)
{
    ether_frame frame;

    while(1)
    {
        int n = net.recv_frame(&frame, sizeof(ether_frame));
        if (n < 42) continue; // bad frame!
        switch (BUFF_UINT16(frame.header.prot, 0))
        {
            case ETHER_PROT_IPV4:
                ip_handler(frame.data, n - sizeof(ether_header), &(frame.header));
                break;

            case ETHER_PROT_ARP:
                arp_handler(frame.data, n - sizeof(ether_header), &(frame.header));
                break;
        }
    }
}

ether_frame* make_frame(byte* dst, unsigned short prot, byte* data, int n)
{
    ether_frame* out = (ether_frame*)malloc(n + sizeof(ether_header));
    memcpy(out->header.dst, dst, 6);
    memcpy(out->header.src, me.mac, 6);
    out->header.prot[0] = (prot & 0xFF00) >> 8;
    out->header.prot[1] = (prot & 0x00FF) >> 0;
    memcpy(out->data, data, n);
    return out;
}

void arp_handler(byte* packet, int n, ether_header* header)
{
    arp_frame* frame = (arp_frame*)packet;

    switch (BUFF_UINT16(frame->header.opcode, 0))
    {
        case 1: // Request
            saveArpCache(((ipmac*)frame->data) + 0);
            if (frame->data[16] == me.ip[0] &&
                frame->data[17] == me.ip[1] &&
                frame->data[18] == me.ip[2] &&
                frame->data[19] == me.ip[3])
            {
                // Start with a response frame that has a payload exactly matching what we received
                ether_frame* response = make_frame(frame->data, ETHER_PROT_ARP, (byte*)&frame, n);
                arp_frame* response_arp = (arp_frame*)((byte*)(response) + sizeof(ether_header));

                // Convert to reply opcode
                response_arp->header.opcode[1] = 2;

                // Move the sender info the the target info
                memcpy(response_arp->data + sizeof(ipmac), response_arp->data + 0, sizeof(ipmac));

                // Fill the sender info with our info
                memcpy(response_arp->data + 0, &me, sizeof(ipmac));

                send_queue.send(PACKET, response, n + sizeof(ether_header));
                free(response);
            }
            break;

        case 2: // Reply
            saveArpCache(((ipmac*)frame->data) + 0);
            saveArpCache(((ipmac*)frame->data) + 1);
            break;
    }
}
```

```
void ip_handler(byte* packet, int n, ether_header* header)
{
    static const int this_ip = BUFF_UINT32(me.ip, 0);
    ip_frame* frame = (ip_frame*)packet;

    // Validate the checksum
    if (chksum(packet, sizeof(ip_header), 0) != 0xffff)
    {
        printf("IP message received with bad checksum\n");
        return;
    }

    // Ignore packets meant for others
    if (BUFF_UINT32(frame->header.dst, 0) != this_ip)
    {
        return;
    }

    // Don't include any padding in ip packet
    int len = BUFF_UINT16(frame->header.length, 0);
    if (n > len) { n = len; }

    // This should be a rare error condition
    if (n < len)
    {
        printf("IP message received with missing data\n");
        return;
    }

    // Find the payload
    byte* payload = frame->data;
    int option_bytes = 4 * ((frame->header.ver_ihl & 0x0f) - 5);
    payload = payload + option_bytes;
    int payload_n = n - option_bytes - sizeof(ip_header);

    //printf("IP message received, protocol: %i\n", frame->header.prot);
    switch (frame->header.prot)
    {
        case IPV4_PROT_ICMP:
            icmp_handler(payload, payload_n, &(frame->header));
            break;
    }
}

void icmp_handler(byte* packet, int n, ip_header* header)
{
    icmp_frame* frame = (icmp_frame*)packet;

    // Validate the checksum
    if (chksum(packet, n, 0) != 0xffff)
    {
        printf("ICMP message received with bad checksum\n");
        return;
    }

    //printf("ICMP message received\n");
    switch (frame->header.type)
    {
        case 0x08: // echo (ping) request
            frame->header.type = 0x00; // echo (ping) reply
            frame->header.crc[0] = 0;
            frame->header.crc[1] = 0;

            int crc = ~chksum((byte*)frame, n, 0);
            frame->header.crc[0] = (crc & 0xff00) >> 8;
            frame->header.crc[1] = (crc & 0x00ff) >> 0;

            sendIPv4Packet(header->src, IPV4_PROT_ICMP, packet, n);
            break;
    }
}
```

```
void pingARP(byte* ip)
{
    static arp_frame message = {
        {
            { 0, 1 },
            { 8, 0 },
            6, 4,
            { 0, 1 },
        },
        { 0 },
    };
    static const int n = sizeof(arp_header) + (2 * sizeof(ipmac));

    if (message.data[0] == 0)
    {
        memcpy(message.data, &me, sizeof(ipmac));
    }

    ipmac* found = retrieveArpCache(ip);
    if(found == NULL)
    {
        ipmac value = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0, 0, 0, 0 };
        memcpy(value.ip, ip, 4);
        memcpy(((ipmac*)(message.data)) + 1, &value, sizeof(ipmac));
    }
    else
    {
        memcpy(((ipmac*)(message.data)) + 1, found, sizeof(ipmac));
    }
    byte* dest_mac = (byte*)(((ipmac*)(message.data)) + 1);
    ether_frame* frame = make_frame(dest_mac, ETHER_PROT_ARP, (byte*)(&message), n);
    send_queue.send(PACKET, frame, n + sizeof(ether_header));
    free(frame);
}

inline byte* hop_ip(byte* ip)
{
    static const int gateway = BUFF_UINT32(me.default_gateway, 0);
    static const int subnet_mask = BUFF_UINT32(me.subnet_mask, 0);
    static const int subnet = subnet_mask & gateway;

    int ip4 = BUFF_UINT32(ip, 0);
    if ((ip4 & subnet_mask) == subnet)
    {
        return ip;
    }
    return me.default_gateway;
}
byte* get_mac(byte* ip)
{
    byte* dst_ip = hop_ip(ip);
    ipmac* dst = retrieveArpCache(dst_ip);

    int attempts = 4;
    while (dst == NULL && --attempts >= 0)
    {
        pingARP(dst_ip);
        sleep(1);
        dst = retrieveArpCache(dst_ip);
    }

    if (dst == NULL)
    {
        printf("Unable to resolve ip address: %i.%i.%i.%i\n", ip[0], ip[1], ip[2], ip[3]);
        return NULL;
    }
    return dst->mac;
}
```

```
void sendIPv4Packet(byte* ip, byte prot, byte* payload, int n)
{
    static unsigned short identifier = 0;
    static ip_frame request = { 0 };
    /*
    {
        {
            { 4, 5 }, // 0x45 // ipv4 optionless header
            { 0, 0 }, // default dscp
            { 0x00, 0x00 }, // length (calculated at each call)
            { 0x00, 0x00 }, // id (calculated at each call)
            { 2, 0 }, // 0x4000, // no fragmentation
            64, // ttl 64 (seems common for a default)
            0, // protocol: (copied at each call)
            { 0 }, // checksum (0 to start)
            { 0 }, // source (0 for now, copied on first call)
            { 0 }, // destination (copied at each call)
        },
        { 0 }, // payload (copied at each call)
    };
    */

    // static initializer for request
    if (request.header.ver_ihl == 0)
    {
        //request.header.version = 4;
        //request.header.ihl = 5; // no options
        request.header.ver_ihl = 0x45;
        //request.header.flags = 2; // no fragmentation
        request.header.frag[0] = 0x40;
        request.header.ttl = 64;
        memcpy(request.header.src, me.ip, 4); // copy source ip
    }

    byte* dst_mac = get_mac(ip);
    if (dst_mac == NULL) { return; }

    ++identifier;

    int N = sizeof(ip_header) + n;
    ether_frame* frame = make_frame(dst_mac, ETHER_PROT_IPV4, (byte*)(&request), N);
    ip_frame* packet = (ip_frame*)(frame->data);
    memcpy(packet->data, payload, n);
    memcpy(packet->header.dst, ip, 4);

    packet->header.length[0] = (N & 0xff00) >> 8;
    packet->header.length[1] = (N & 0x00ff) >> 0;

    packet->header.ident[0] = (identifier & 0xff00) >> 8;
    packet->header.ident[1] = (identifier & 0x00ff) >> 0;

    packet->header.prot = prot;

    int crc = ~chksum((byte*)packet, sizeof(ip_header), 0);
    packet->header.crc[0] = (crc & 0xff00) >> 8;
    packet->header.crc[1] = (crc & 0x00ff) >> 0;

    send_queue.send(PACKET, frame, N + sizeof(ether_header));
    free(frame);
}
```

```
void pingICMP(byte* ip, byte* data, int n)
{
    static unsigned short identifier = 0;
    static icmp_frame request =
    {
        {
            0x08, // echo (ping) request
            0x00, // code
            { 0 }, // checksum (computed every call)
            { 0 }, // header (computed every call)
        },
        { 0 },
    };

    ++identifier;
    unsigned short sequence = 0;

    memcpy(request.data, data, n);
    int N = n + sizeof(icmp_header);

    request.header.crc[0] = 0;
    request.header.crc[1] = 0;

    request.header.echo.ident[0] = (identifier & 0xff00) >> 8;
    request.header.echo.ident[1] = (identifier & 0x00ff) >> 0;

    request.header.echo.seqno[0] = (sequence & 0xff00) >> 8;
    request.header.echo.seqno[1] = (sequence & 0x00ff) >> 0;

    int crc = ~chksum((byte*)(&request), N, 0);
    request.header.crc[0] = (crc & 0xff00) >> 8;
    request.header.crc[1] = (crc & 0x00ff) >> 0;

    sendIPv4Packet(ip, IPV4_PROT_ICMP, (byte*)(&request), N);
}

int main()
{
    memcpy(me.mac, net.get_mac(), 6);
    arp_cache[me.ip[3]] = &(me.arp_cache_self);

    int err;
    pthread_t rthread, sthread;

    // Create the threads
    err = pthread_create(&rthread, NULL, receive_thread, NULL);
    err = pthread_create(&sthread, NULL, send_thread, NULL);

    //------------------------------------------------------------------------+
    // main application routine                                               |

    byte request[4] = { 192, 168, 1, 30 };
    byte payload[4] = { 0xde, 0xad, 0xbe, 0xef };

    while(1) {
        printf("Press enter to ping ...");
        getchar();

        //for(int i = 0; i < 5; ++i)
        {
            //request[3] = 10 + i * 5;
            printf("Sending 0xdeadbeef %i.%i.%i.%i\n", request[0], request[1], request[2], request[3]);
            pingICMP(request, payload, 4);
        }
    }

    // main application routine                                               |
    //------------------------------------------------------------------------+

    // Put main() to sleep until threads exit
    err = pthread_join(rthread, NULL);
    err = pthread_join(sthread, NULL);

    return 0;
}
```