Completed with Lab Partner: Josh Lake

Objective

To understand the ARP mechanism and construct an ARP cache.

Structural Overview

One computer will use the command `arping` to demonstrate regular usage of the address resolution protocol. The program developed in this lab will reply to the ARP request in addition to the reply the operating system generates (these two replies should be the same). The program will, responding to user input, send ARP requests for all IPs not in the cache and ARP replies to all IPs in the cache.

Simulation

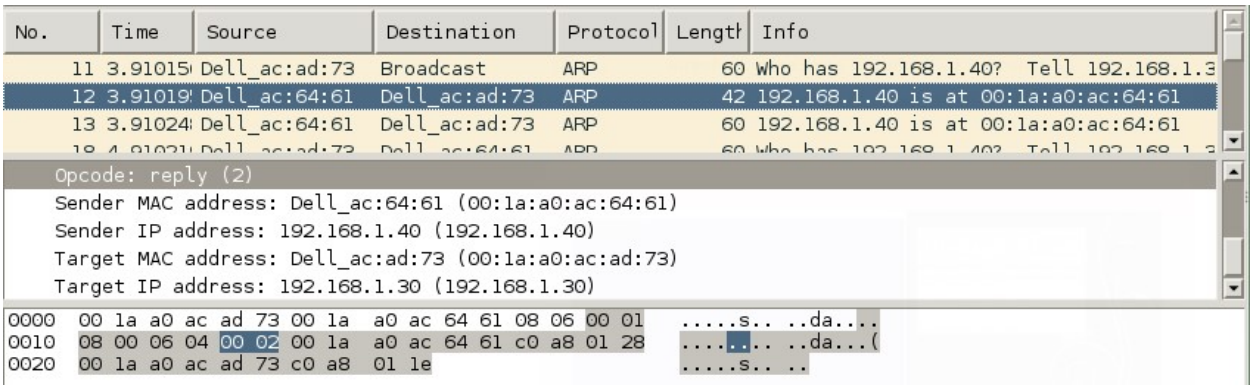

Figure 1 - The Default OS-Generated Reply

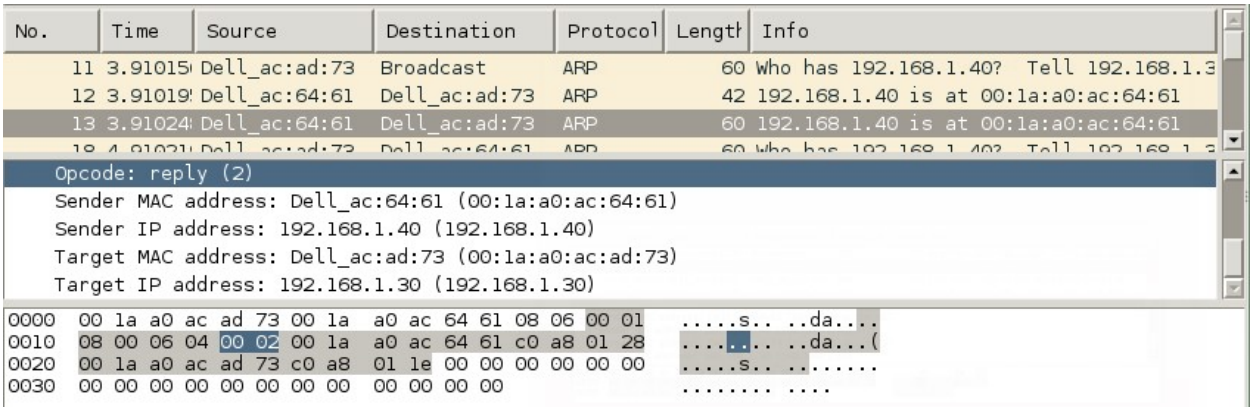

Figure 2 - Our Code-Generated Reply

```
netlab40:~/Documents/lab2_updated # ./lab2
Press enter to send batch ...                                          A
Sending 192.168.1.10: Not Found in cache, sending broadcast request
Sending 192.168.1.15: Not Found in cache, sending broadcast request
Sending 192.168.1.20: Not Found in cache, sending broadcast request
Sending 192.168.1.25: Not Found in cache, sending broadcast request
Sending 192.168.1.30: Found in cache, sending reply
Press enter to send batch ...                                          B
Sending 192.168.1.10: Found in cache, sending reply
Sending 192.168.1.15: Not Found in cache, sending broadcast request
Sending 192.168.1.20: Found in cache, sending reply
Sending 192.168.1.25: Not Found in cache, sending broadcast request
Sending 192.168.1.30: Found in cache, sending reply
Press enter to send batch ...
```

Figure 3 - Console Output

11	2.08780	Dell_ac:64:61	Broadcast	ARP	a 42 Who has 192.168.1.10? Tell 192.168.1.40
12	2.08782	Dell_ac:64:61	Broadcast	ARP	42 Who has 192.168.1.15? Tell 192.168.1.40
13	2.08784	Dell_ac:64:61	Broadcast	ARP	42 Who has 192.168.1.20? Tell 192.168.1.40
14	2.08786	Dell_ac:64:61	Broadcast	ARP	42 Who has 192.168.1.25? Tell 192.168.1.40
15	2.08788	Dell_ac:64:61	Dell_ac:ad:73	ARP	42 192.168.1.40 is at 00:1a:a0:ac:64:61
16	2.08792	Dell_ac:df:57	Dell_ac:64:61	ARP	b 60 192.168.1.10 is at 00:1a:a0:ac:df:57
17	2.08796	Dell_ac:b0:e8	Dell_ac:64:61	ARP	60 192.168.1.20 is at 00:1a:a0:ac:b0:e8
22	6.87983	Dell_ac:64:61	Dell_ac:df:57	ARP	c 42 192.168.1.40 is at 00:1a:a0:ac:64:61
23	6.87986	Dell_ac:64:61	Broadcast	ARP	42 Who has 192.168.1.15? Tell 192.168.1.40
24	6.87988	Dell_ac:64:61	Dell_ac:b0:e8	ARP	42 192.168.1.40 is at 00:1a:a0:ac:64:61
25	6.87990	Dell_ac:64:61	Broadcast	ARP	42 Who has 192.168.1.25? Tell 192.168.1.40
26	6.87992	Dell_ac:64:61	Dell_ac:ad:73	ARP	42 192.168.1.40 is at 00:1a:a0:ac:64:61

Figure 4 - Wireshark Output

Results

Figures 1 and 2 together demonstrate our program responding to an ARP request the same way the OS does. Wireshark omits the padding in OS-Generated messages for reasons undetermined. However, we demonstrated in lab session with the assistance of the TA's that Wireshark on a different machine shows the padding for incoming messages whether OS-Generated or Code-Generated.

Figures 3 and 4 demonstrate the same period showing the following communication

- An IP address is not in the ARP cache, so the program sends a broadcast ARP request (annotated: A, a)
- The computer with that IP sends a reply ARP packet (annotated: b), that IP/MAC pair is added to the cache
- At a later point the program forms an ARP message to the IP address which is now cached (annotated B, c)

Some notes about our code. For our ARP cache we took the liberty of assuming we would only deal with messages in the same subnet, and the subnet was the first three octals; thus we could implement a hash table using the last octal of the IP address as a hash key for the cache (for the purposes of this lab, it was discussed with the TAs that this was acceptable).

Additionally, our code requires hard-coding the ip address of the machine running the code (near the top of main.cpp). The hardware address will be obtained via the `frameio` class provided by the instructor. The `message_queue` class was also provided by the instructor, and is a simplified implementation of the sender/receiver multi-threading pattern.

Makefile

```
lab2: main.cpp util.o frameio.o
    g++ main.cpp util.o frameio.o -lpthread -g -O0 -o lab2

util.o: util.cpp util.h
    g++ util.cpp -c -g -O0 -o util.o

frameio.o: frameio.cpp frameio.h
    g++ frameio.cpp -c -g -O0 -o frameio.o

clean:
    rm *.o
    rm lab2
```

main.cpp

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

frameio net;                    // gives us access to the raw network
message_queue send_queue;       // message queue for the sending ether_frames
message_queue ip_queue;         // message queue for the IP protocol stack
message_queue arp_queue;        // message queue for the ARP protocol stack

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

ipmac me = { 0, 0, 0, 0, 0, 0, 192, 168, 1, 40 };

ipmac* arp_cache[256] = { 0 };

struct ether_header
{
    octet dst_mac[6];           // destination MAC address
    octet src_mac[6];           // source MAC address
    octet prot[2];              // protocol (or length)
};

struct ether_frame              // handy template for 802.3/DIX frames
{
    ether_header header;
    octet data[1500];           // payload
};

#define ETHER_PROT_IP           0x0800
#define ETHER_PROT_ARP          0x0806

#define BUFF_UINT16(buff, i)    (buff[i + 0] << 8 | buff[i + 1] << 0)

void* receive_thread(void* args)
{
    ether_frame buf;

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

            case ETHER_PROT_ARP:
                arp_queue.send(PACKET, buf.data, n - sizeof(ether_header));
                break;
        }
    }
}

void* send_thread(void* args)
{
    int n;
    ether_frame buf;
    event_kind event;
    while(1)
    {
        n = send_queue.recv(&event, &buf, sizeof(buf));
        net.send_frame(&buf, n);
    }
}
```

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

struct arp_header
{
    octet hwtype[2];
    octet prottype[2];
    octet hwlength;
    octet protlength;
    octet opcode[2];
};

struct arp_frame
{
    arp_header header;
    octet data[1500 - sizeof(arp_header)];
};

ipmac* retrieveFromCache(ipmac* value)
{
    return arp_cache[value->ip[3]];
}

void saveToCache(ipmac* value)
{
    if (retrieveFromCache(value) == NULL)
    {
        ipmac* copy = (ipmac*)malloc(sizeof(ipmac));
        memcpy(copy, value, sizeof(ipmac));
        arp_cache[value->ip[3]] = copy;
    }
}

void* arp_protocol(void* args)
{
    int n;
    arp_frame buf;
    event_kind event;

    while (1)
    {
        n = arp_queue.recv(&event, &buf, sizeof(buf));
        switch (BUFF_UINT16(buf.header.opcode, 0))
        {
            case 1: // Request
                saveToCache(((ipmac*)buf.data) + 0);
                if (buf.data[16] == me.ip[0] &&
                    buf.data[17] == me.ip[1] &&
                    buf.data[18] == me.ip[2] &&
                    buf.data[19] == me.ip[3])
                {
                    // Start with a response frame that has a payload exactly matching what we received
                    ether_frame* response = make_frame(buf.data, ETHER_PROT_ARP, (octet*)&buf, n);
                    arp_frame* response_arp = (arp_frame*)((octet*)(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
                saveToCache(((ipmac*)buf.data) + 0);
                saveToCache(((ipmac*)buf.data) + 1);
                break;
        }
    }
}
```

```
// assuming value->mac = { ff, ff, ff, ff, ff, ff }
void sendARP(ipmac* value)
{
    ipmac* found = retrieveFromCache(value);
    arp_frame message = {
        {
            { 0, 1 },
            { 8, 0 },
            6, 4,
            { 0, 0 }
        },
        { 0 },
    };
    if(found == NULL)
    {
        printf("Not Found in cache, sending broadcast request\n");
        message.header.opcode[1] = 1; // request
        memcpy(message.data, &me, sizeof(ipmac));
        memcpy(((ipmac*)(message.data)) + 1, value, sizeof(ipmac));
    }
    else
    {
        printf("Found in cache, sending reply\n");
        message.header.opcode[1] = 2; // reply
        memcpy(message.data, &me, sizeof(ipmac));
        memcpy(((ipmac*)(message.data)) + 1, found, sizeof(ipmac));
    }
    int n = sizeof(arp_header) + (2 * sizeof(ipmac));
    ether_frame* frame = make_frame((octet*)(((ipmac*)(message.data)) + 1), ETHER_PROT_ARP, (octet*)(&message), n);
    send_queue.send(PACKET, frame, n + sizeof(ether_header));
    free(frame);
}

int main()
{
    // Open the shared resource before starting threads
    net.open_net("enp3s0");
    const octet* mymac = net.get_mac();
    me.mac[0] = mymac[0];
    me.mac[1] = mymac[1];
    me.mac[2] = mymac[2];
    me.mac[3] = mymac[3];
    me.mac[4] = mymac[4];
    me.mac[5] = mymac[5];

    arp_cache[me.ip[3]] = &me;

    int err;

    pthread_t rthread, sthread;
    pthread_t arpthread;

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

    err = pthread_create(&arpthread, NULL, arp_protocol, NULL);

    ipmac request = {
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 192, 168, 1, 0
    };

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

        for(int i = 0; i < 5; ++i)
        {
            request.ip[3] = 10 + i * 5;
            printf("Sending 192.168.1.%i: ", request.ip[3]);
            sendARP(&request);
        }
    }

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

    err = pthread_join(arpthread, NULL);

    return 0;
}
```