Project 2

Samuel Bhushan, Partner: Hailey Parkin

October 23, 2015

1 Introduction

ARP is a system that allows computers to keep track of which computer (identified by MAC Address) has a given IP Address. The aim of this project was to delve deep into the ARP protocol by implementing some functions manually. In this project we implemented the recieve, reply, and request for ARP communication. We also made a simple ARP cache that stores the MAC/IP Address pairs. This project also forces us to understand how ethernet frames are sent out to the network.

2 The code

We split this project into two programs:

1. This program polls for incoming ARP packets and replies to any directed to us. Requests are pulled inside a thread and put onto a queue for the reply thread. In order to reply, an ARP frame is constructed and then encapsulated by an ethernet frame. Once the frame is ready, it is sent via frameio::send_frame().

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

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

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 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];
};

void* arp_reply(void*);
void* receive_arp(void*);
bool construct_send(arp_frame *request);

pthread_t threads;


int main()
{
        net.open_net("enp3s0");
        pthread_create(&threads,NULL,receive_arp,NULL);
        pthread_create(&threads,NULL,arp_reply,NULL);
        for( ; ;) sleep(1);
}

void* receive_arp(void* arg)
{       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* arp_reply(void *arg)
{
        arp_frame request;
```

```
	octet myip[4] =  {0xC0,0xA8,0x01,0x28};
	bool ip_match = true;
	int request_size;
	event_kind type = PACKET;
	while(1)
	{
		request_size = arp_queue.recv(&type,(void*)&request,(int)sizeof(reques
		//Check if it is a request
		if(request.opcode[1] == 0x01 && request.target_ip[3] == 0x28)
		{
			//Check IP ADDR
			for(int i = 0; i < 4; ++i)
			{
				if(request.target_ip[i] != myip[i])
					ip_match = false;

			}

			if(ip_match)
				construct_send(&request);
		}


	}
}

bool construct_send(arp_frame *request)
{
	arp_frame reply;
	arp_ether_frame ether_reply;
	for(int i=0; i<sizeof(ether_reply.data); i++)
	{
		ether_reply.data[i] = 0;
	}

	std::cout << "FOUND ARP REQUEST TO US!" << std::endl;

	//Arp Reply Part
	reply.mac_type[0] = 0x00;
	reply.mac_type[1] = 0x01;
	reply.protocol_type[0] = 0x08;
	reply.protocol_type[1] = 0x00;
	reply.mac_len = 6;
	reply.protocol_len = 4;
	reply.opcode[1] = 0x02; //switch to a reply ARP
	memcpy(reply.sender_mac,net.get_mac(), sizeof(request->sender_mac));
	memcpy(reply.sender_ip,request->target_ip,sizeof(request->target_ip));
	memcpy(reply.target_mac, request->sender_mac, sizeof(request->sender_mac));
	memcpy(reply.target_ip,request->sender_ip,sizeof(request->sender_ip));
	//Ether Frame Part
	memcpy(ether_reply.dst_mac,reply.target_mac,sizeof(reply.target_mac));
	memcpy(ether_reply.src_mac,reply.sender_mac,sizeof(reply.sender_mac));
	ether_reply.prot[0] = 0x08;
	ether_reply.prot[1] = 0x06;
```

```
        memcpy(ether_reply.data, &reply, sizeof(reply));

for (int i = 0; i < sizeof(ether_reply.dst_mac); ++i)
        {
                printf("%02x",ether_reply.dst_mac[i]);

        }
        std::cout<<std::endl;
        for (int i = 0; i < sizeof(ether_reply.src_mac); ++i)
        {
                printf("%02x",ether_reply.src_mac[i]);

        }
        std::cout<<std::endl;
        for (int i = 0; i < sizeof(ether_reply.prot); ++i)
        {
                printf("%02x",ether_reply.prot[i]);

        }
        std::cout<<std::endl;
                for (int i = 0; i < sizeof(ether_reply.data); ++i)
        {
                printf("%02x",ether_reply.data[i]);

        }
        std::cout<<std::endl;



        std::cout << std::endl;
        std::cout << net.send_frame(&ether_reply,sizeof(ether_reply));
        std::cout << sizeof(ether_reply) << std::endl;
}
```

2. The second program has an ARP cache and tries to send packets to a predefined IP address. An array of structs holding MAC/IP pairs was created as the ARP cache. A global int was iterated to keep track of where the oldest entry is. One thread recieves ARP packets and stores reported pairs in the cache. A second thread intermittently attempts to send a packet to some IP. If the MAC for the IP is not in the cache an ARP request is constructed and sent.

```
#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>

#define SIZE_CACHE 100

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



frameio net;              // gives us access to the raw network
message_queue ip_queue;   // message queue for the IP protocol stack
message_queue arp_queue; // message queue for the ARP protocol stack
pthread_t threads;
arp_pairs cache[SIZE_CACHE];
int cache_index = 0;

void* receive_frame(void* arg);
void* arp_pair_updater(void *arg);
void* arp_requester(void* arg);
bool add_pair(octet* mac,octet* ip);



int main()
{

        net.open_net("enp3s0");

        // Thread for putting packets on queues
        pthread_create(&threads,NULL,receive_frame,NULL);
        // Thread that pulls arp frames and updates the arp cache
        pthread_create(&threads,NULL,arp_pair_updater,NULL);
```

```
        // Thread that continually sends arp requests
        pthread_create(&threads,NULL,arp_requester,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* arp_pair_updater(void *arg)
{
        arp_frame received;
        int request_size;
        event_kind type = PACKET;
        std::cout << "ARP PAIRING THREAD STARTED" << std::endl;
        while(1)
        {
                request_size = arp_queue.recv(&type,(void*)&received,(int)sizeof(recei
                if(received.sender_ip[0] == 192 && received.sender_ip[1] == 168)
                        add_pair(received.sender_mac, received.sender_ip);
        }
}
void* arp_requester(void* arg)
{
        arp_frame message;
        arp_ether_frame ether_message;
        octet myip[4] =  {0xC0,0xA8,0x01,0x14};
        octet target_ip[4] =  {0xC0,0xA8,0x01,0x0A};
        octet target = 10;
        octet target_mac[6] = {0xff,0xff,0xff,0xff,0xff,0xff};
        while(1)
        {
                for (int i = 0; i < SIZE_CACHE; ++i)
                {
                        if(cache[i].ip[3] == target)
                                {
                                        memcpy(target_mac, cache[i].mac, sizeof(target_
```

```
                                    }

                        }

                        //Arp Part
                        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(target_mac));
                        memcpy(message.target_ip,target_ip,sizeof(target_ip));

                        //Ether Frame Part
                        memcpy(ether_message.dst_mac,message.target_mac,sizeof(message.target_m
                        memcpy(ether_message.src_mac,message.sender_mac,sizeof(message.sender_m
                        ether_message.prot[0] = 0x08;
                        ether_message.prot[1] = 0x06;
                        memcpy(ether_message.data, &message, sizeof(message));
                        std::cout << "ARP REQUESTER THREAD STARTED" << std::endl;
                        if(target_mac[0] == 0xff)
                                    std::cout << "not found" << std::endl;
                        else
                              std::cout << "found" << std::endl;
                        //send message
                        std::cout << net.send_frame(&ether_message,sizeof(ether_message));
                        sleep(1);
            }
}
bool add_pair(octet* mac,octet* ip)
{
            /*for (int i = 0; i < 6; ++i)
            {
                        printf("%02x ",*(mac+i));

            }
            printf("\n");
            for (int i = 0; i < 4; ++i)
            {
                        printf("%u ",*(ip+i));

            }
            printf("\n");*/
            for (int i = 0; i < SIZE_CACHE; ++i)
            {
                        if(cache[i].ip[3] == *(ip+3))
                              {
                                    std::cout << "already in cache" <<std::endl;
                                    return 0;
                              }
```

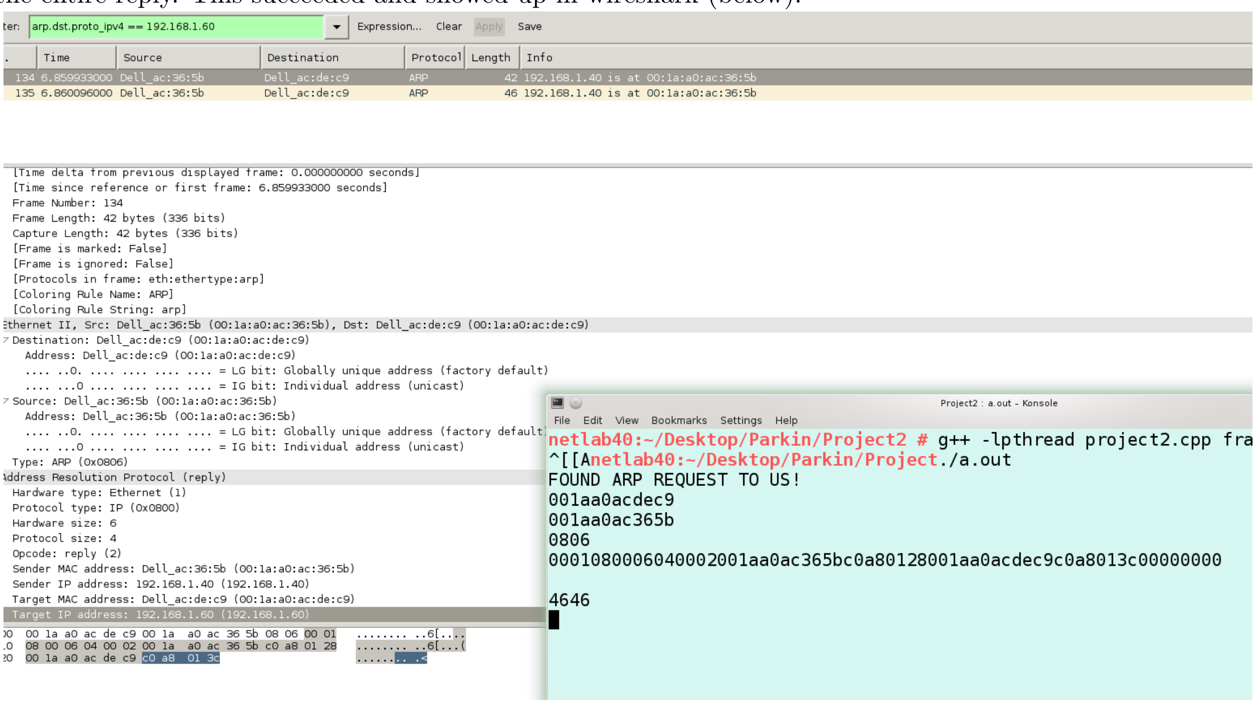
```
        }
        memcpy(&cache[cache_index].mac,mac,sizeof(cache[cache_index].mac));
        memcpy(&cache[cache_index].ip,ip,sizeof(cache[cache_index].ip));
        printf("%02x %u \n", cache[cache_index].mac[5], cache[cache_index].ip[3]);
        cache_index < SIZE_CACHE ? cache_index++: cache_index = 0;
        return 1;

}
```

3 Program Results

3.1 Sending out a reply

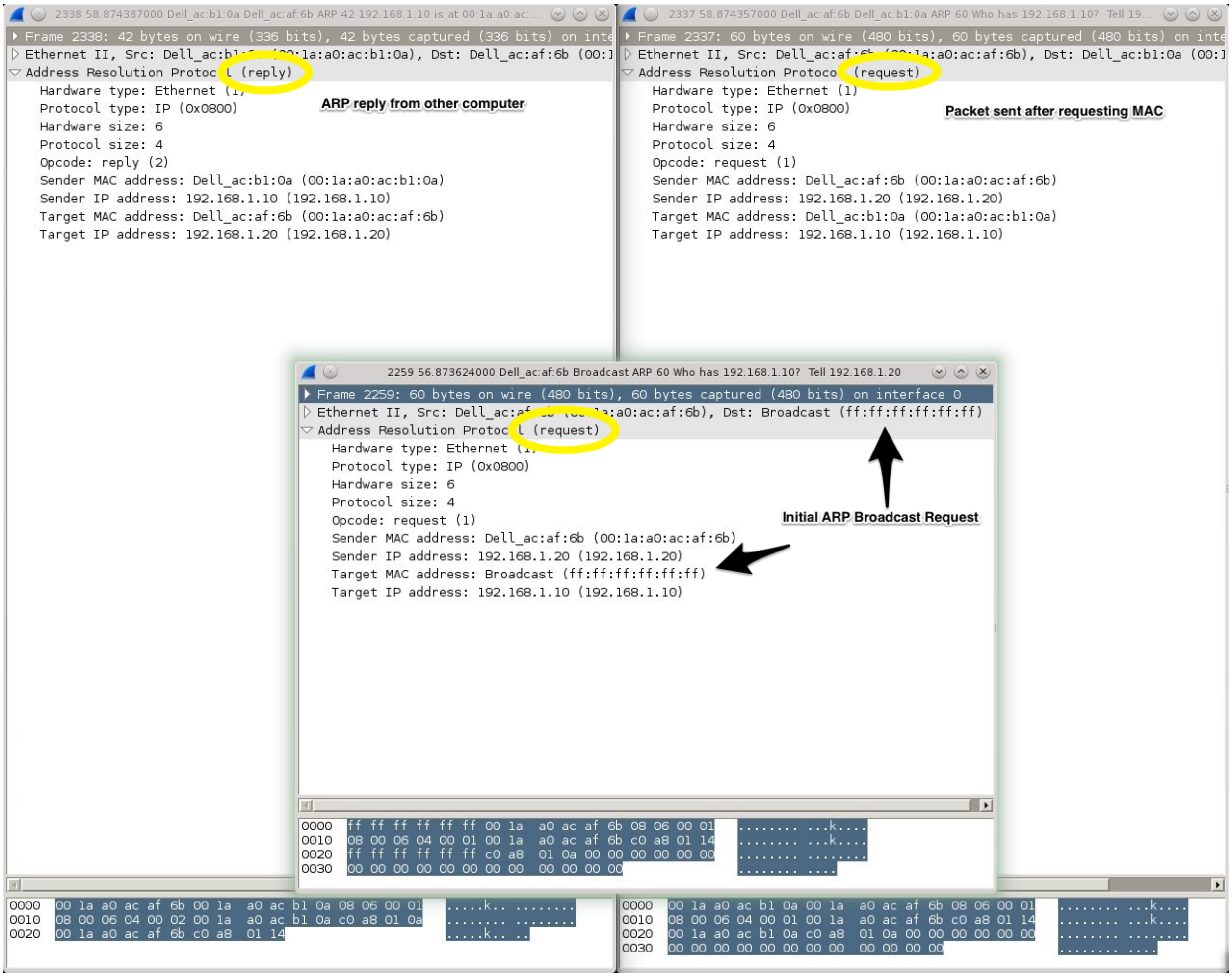
Initially, in order to reply, we decided to copy the request bytes to the new variable used for our reply. We used memcpy() to do this and this completely corrupted the data. We resorted to manually filling each field in the entire reply. This succeeded and showed up in wireshark (below).



3.2 Updating cache and sending a packet

The cache system worked from the start, but some minor issues were preventing successful requests. During debugging, we found that using the %02x parameter with printf() allowed us to print MAC addresses quickly.

In Wireshark we were able to capture the initial ARP broadcast with 0xF's, the ARP reply from the other computer, and our successfully sent packet:



4 Conclusions

This lab educated us in two primary ways that will likely benefit us in the future. Trying to setup ARP routines forced us to learn how all of the included libraries work and how threading is useful. Obviously ARP was understood, but I also saw how layers and headers work to transfer data.

I have always thought of ARP as a very intuitive system; seeing it in operation validated this. Also, debugging with threads is difficult and greatly benefited with many print statements. Again, Wireshark was very useful.