Ex. No. 4	**ARP Simulation**	**Karthik D**
10 August 2021	**UCS1511 - Networks Lab**	**195001047**

Aim

To simulate the working of ARP using socket programming in the C language, with the help of TCP-based multi-client server

Question

Simulate ARP using TCP socket programming, wherein:

The server should perform the following

1. Consider the server as a host or a router
2. Enter hosts/routers' IP address and MAC address
3. Listen for any number of client (for broadcasting purpose)
4. Enter the packet details received from a host or its own packet to send to a destination.
5. The details are
    a. Source IP address
    b. Source MAC address
    c. Destination IP address
    d. 16-bit data
6. Develop an ARP Request packet which is to be broadcasted to all clients.
7. Query packet should contain:
   ARPOperation | SourceMAC | SourceIP | DestinationMAC | DestinationIP

The client should perform the following:

1. When an ARP Reply is received with the Destination MAC address, send the packet to the corresponding destination
2. Also check the validity of IP and MAC address
3. Enter the clients own IP and MAC
4. When an ARP Request packet is received, check whether the Destination IP is its own.
5. If not, no reply
6. If yes, respond with ARP Reply packet formatted as :
   ARPOperation | SourceMAC | SourceIP | DestinationMAC | DestinationIP

Algorithms

(a) Server-side

Step 1: Globally define a dynamic array to store connection socket descriptors of connected clients. This is initially empty

Step 2: Accept an IP address and MAC address for the router device (i.e the server) from the user

Step 3: Create a network socket with parameters suitable for an end-point of TCP based communication

Step 4: Bind the socket to INADDR_ANY which is defined as a *zero* address, allowing the socket to be reachable by all active interfaces on the device. Set the port to a preset value, known to the targeted clients as well

Step 5: Set the socket status to passive i.e initiate listening on the socket to allow it to accept incoming connection requests

Step 6: Wait for a connection request from the first client and accept the first such request. Save the file descriptor of the connection-socket. This will be used to communicate with the client

Step 7: Initiate a separate thread to listen for more incoming client connection requests and accept them when they do. Add every connection socket descriptor to the global connected clients array

Step 8: On the main thread, prepare a memory buffer to read and store messages from the connection.

Step 9: Start an infinite loop to perform the following operations,

- **i:** Accept a 16-bit message and the destination IP for the message from the user
- **ii:** Prepare an ARP-Request packet as per the request format and send it to all the connected clients using the *write()* system call and iterating over all the descriptors for all connected clients
- **iii:** Using the *fd_set* and *select* utilities, wait for one of the client sockets to become readable. If not response is received for a pre-defined time, shut down the server
- **iv:** When a client connection socket is readable, read the message using the *read()* system call and check if it is a valid ARP response packet. Display the ARP-Response packet received.

v: Obtain the message destination client's MAC address from the request packet

vi: Use the *write()* system call to send the 16-bit data to this client by writing to the client connection socket

(Repeat till timeout when waiting for an ARP-Response in step-(iii))

Step 10: Close the created sockets using the *close()* system call and terminate the process

(b) **Client-side**

Step 1: Accept an IP address and MAC address for the host device (i.e the client) from the user

Step 2: Create a network socket with parameters suitable for an end-point of TCP based communication

Step 3: Accept the router's IP address as input from the user

Step 4: Using the accepted IP address and a preset port number agreed upon between the server and client, send a connection request to the server using *connect()* utility of socket programming

Step 5: Prepare a memory buffer to read and store messages from the connection.

Step 6: Start an infinite loop to perform the following operations,

i: Use the *read()* system call to block and read a message sent from the server by reading into own socket, and store the message in the buffer

ii: If a message is read and the size of the message is 0, terminate the loop. This indicates that the server/router has exited

iii: When a message is received, validate the ARP packet format

iv: If the packet is a valid ARP-Request, check if the destination IP in the packet matches with the host's own IP address.

v: If not, display a suitable message to the user and ignore the received packet. Go back to step-(i)

vi: If so, prepare an ARP-Response packet, packing the client's own MAC address and IP address along the with the router's MAC and IP addresses

vii: Use the *write()* system call to send this packet to the server by writing to own socket descriptor

(Repeat till server exits)

C Program Code

1. tcp_socket.h - TCP connection helper functions

```
#ifndef tcp_socket
#define tcp_socket

#include<sys/socket.h>
#include<arpa/inet.h>
#include<unistd.h>
#include<string.h>
#include<errno.h>

#define SERVER_PORT 8080
#define LOCALHOST_IP "127.0.0.1"
#define MAX_CLIENTS 3
#define ADDRESS_FAMILY AF_INET
#define ADDRESS_BUFFER_SIZE 30
#define MSG_BUFFER_SIZE 120
#define TERMINATION_INIT_STRING "ENDSESSION"
#define TERMINATION_ACK_STRING "ENDSESSION_ACK"
#define MSG_DELIMITER ';'
#define IP_ADDRESS_SIZE 20
#define MAC_ADDRESS_SIZE 20
#define MSG_WAIT_TIMEOUT 20

/*
Use BLOCKING sockets (default configuration)
Only one client-connection
And server only echoes messages
No need to initiate messages on the server!
(i.e) Synchronous send/receive
*/
```

```
int make_socket(){
   // AF_INTER specifies IPv4
   // SOCK_STREAM specifies two-way byte-stream
   // 0 selects default protocol
   int sock_fd = socket(ADDRESS_FAMILY, SOCK_STREAM, 0);
   if (sock_fd == -1){
       return -2;    // Could not create socket
   }
   return sock_fd;
}

short check_termination_init(char *msg){
   return (strcmp(msg, TERMINATION_INIT_STRING)==0);
}

short check_termination_ack(char *msg){
   return (strcmp(msg, TERMINATION_ACK_STRING)==0);
}

short bind_server_socket(int sock_fd){
   struct sockaddr_in bind_address;
   // Set family to IPv4
   bind_address.sin_family = ADDRESS_FAMILY;
   // Set port in network byte-order to a non-privileged port (>1023)
   bind_address.sin_port = htons(SERVER_PORT);
   // Set address to 0.0.0.0 to connect to bind to all local interfaces
   bind_address.sin_addr.s_addr = htonl(INADDR_ANY);
   // Convert to generic socket address format and bind
   if (!bind(sock_fd, (struct sockaddr *)&bind_address,
sizeof(bind_address))){
       return 0;    // Success
   }
   else{
       printf("%d", errno);
       return -3;   // Could not bind server-socket
   }
}

short connect_server(int sock_fd, char *server_ip){
   struct sockaddr_in bind_address;
```

```
    bzero((char*)&bind_address, sizeof(bind_address));
    // Set family to IPv4
    bind_address.sin_family = ADDRESS_FAMILY;
    // Set port in network byte-order to a non-privileged port (>1023)
    bind_address.sin_port = htons(SERVER_PORT);
    // Set address to server IP or 127.0.0.1 to loopback to same host
    if (server_ip == NULL){
        bind_address.sin_addr.s_addr = inet_addr(LOCALHOST_IP);
    }
    else{
        bind_address.sin_addr.s_addr = inet_addr(server_ip);
    }
    // Convert to generic socket address format and bind
    if (!connect(sock_fd, (struct sockaddr*)&bind_address,
sizeof(bind_address))){
        return 0;    // Success
    }
    else{
        return -5;   // Could not connect to the local server
    }
}

short initiate_listen(int sock_fd){
    if (!listen(sock_fd, MAX_CLIENTS)){
        return 0;    // Success
    }
    else{
        return -4;   // Could not initiate listening
    }
}

// BLOCKING FUNCTION
int accept_client(int sock_fd, struct sockaddr_in *client_addr, int
*client_addr_len){
    // Address of client is registered
    int client_sock_fd = accept(sock_fd, (struct sockaddr*)client_addr,
client_addr_len);
    if (client_sock_fd == -1){
        return -5;   // Did not connect to a client
    }
```

```
    else if (*client_addr_len > sizeof(struct sockaddr_in)){
        *client_addr_len = -1;    // Non-Fatal Warning: Client address was
truncated to fit in buffer
    }
    return client_sock_fd;
}

void destroy_socket(int sock_fd){
    close(sock_fd);
}

#endif
```

2. msg_io.h - Message transfer helper functions

```
#ifndef msg_io
#define msg_io

#include<sys/types.h>

#include "tcp_socket.h"

struct timeval prepare_time_structure(int duration_sec, int
duration_usec){
    struct timeval time;
    time.tv_sec = duration_sec;
    time.tv_usec = duration_usec;
    return time;
}

int wait_for_message(int *client_fds, int num_fds, fd_set *avl_fds){
    fd_set read_fds;
    FD_ZERO(&read_fds);
    int max_fd = -1;
    for(int i=0;i<num_fds;i++){
        FD_SET(*(client_fds+i), &read_fds);
        if (*(client_fds+i) > max_fd){
            max_fd = *(client_fds+i);
        }
```

```
    }
    struct timeval timeout = prepare_time_structure(MSG_WAIT_TIMEOUT, 0);
    int avl_fds_count = select(max_fd+1, &read_fds, NULL, NULL, &timeout);
    if(avl_fds_count===-1){
        return -8;    // Error when waiting for socket
    }
    else if(avl_fds_count==0){
        return -9;    // Timed-out when waiting for message
    }
    *avl_fds = read_fds;
    return avl_fds_count;
}

ssize_t receive_message(int socket, char *msg_buffer, struct sockaddr_in
*sender_addr, int *sender_addr_len){
    int addr_buffer_size = sizeof(struct sockaddr_in);
    int msg_size = recvfrom(socket, msg_buffer, MSG_BUFFER_SIZE,
MSG_WAITALL, (struct sockaddr*)sender_addr, sender_addr_len);
    if (*sender_addr_len > addr_buffer_size){
        *sender_addr_len = -1;  // Warning: Client address was truncated to
fit in buffer
    }
    return msg_size;
}

#endif
```

3. ARP_Packet.h - ARP Packet ADT

```
#ifndef ARP_Packet_h
#define ARP_Packet_h

#include "tcp_socket.h"

#define REQUEST_OPERATION_ID 1
#define RESPONSE_OPERATION_ID 2
#define ARP_PACKET_STRING_SIZE 120
#define EMPTY_MAC_ADDRESS "00-00-00-00-00-00"
```

```c
#define STRING_INIT "-"


struct arp_packet{
   int operation_id;
   char *source_MAC;
   char *source_IP;
   char *destn_MAC;
   char *destn_IP;
};
typedef struct arp_packet ARP_Packet;

ARP_Packet* make_empty_arp_packet(){
   // Allocate memory
   ARP_Packet *packet = (ARP_Packet*)malloc(sizeof(ARP_Packet));
   packet->source_MAC = (char*)malloc(sizeof(char)*MAC_ADDRESS_SIZE);
   packet->source_IP = (char*)malloc(sizeof(char)*IP_ADDRESS_SIZE);
   packet->destn_MAC = (char*)malloc(sizeof(char)*MAC_ADDRESS_SIZE);
   packet->destn_IP = (char*)malloc(sizeof(char)*IP_ADDRESS_SIZE);
   packet->operation_id = -1;
   memcpy(packet->source_MAC, STRING_INIT, sizeof(STRING_INIT));
   memcpy(packet->source_IP, STRING_INIT, sizeof(STRING_INIT));
   memcpy(packet->destn_MAC, STRING_INIT, sizeof(STRING_INIT));
   memcpy(packet->destn_IP, STRING_INIT, sizeof(STRING_INIT));
   return packet;
}


ARP_Packet* make_arp_packet(int oper_id, char *source_MAC, char
*source_IP, char *destn_MAC, char *destn_IP){
   ARP_Packet* packet = make_empty_arp_packet();
   // Populate the fields
   packet->operation_id = oper_id;
   memcpy(packet->source_MAC, source_MAC, MAC_ADDRESS_SIZE);
   memcpy(packet->source_IP, source_IP, IP_ADDRESS_SIZE);
   memcpy(packet->destn_MAC, destn_MAC, MAC_ADDRESS_SIZE);
   memcpy(packet->destn_IP, destn_IP, IP_ADDRESS_SIZE);
   return packet;
}
```

```
ARP_Packet* retrieve_arp_packet(char *packet_str){
   ARP_Packet *packet = make_empty_arp_packet();
   sscanf(packet_str,
       "%d | %s | %s | %s | %s",
       &packet->operation_id,
       packet->source_MAC,
       packet->source_IP,
       packet->destn_MAC,
       packet->destn_IP
   );
   if(packet->operation_id==-1 ||
       strcmp(packet->source_MAC, STRING_INIT)==0 ||
       strcmp(packet->source_IP, STRING_INIT)==0 ||
       strcmp(packet->destn_MAC, STRING_INIT)==0 ||
       strcmp(packet->destn_IP, STRING_INIT)==0
   ){
       return NULL;
   }
   return packet;
}


short is_destn(ARP_Packet *packet, char *self_ip){
   return (strcmp(packet->destn_IP, self_ip)==0);
}


char* serialize_arp_packet(ARP_Packet *packet){
   char *packet_str = (char*)malloc(sizeof(char)*ARP_PACKET_STRING_SIZE);
   sprintf(packet_str,
       "%d | %s | %s | %s | %s",
       packet->operation_id,
       packet->source_MAC,
       packet->source_IP,
       packet->destn_MAC,
       packet->destn_IP
   );
   return packet_str;
}
```

```
void display_arp_packet(ARP_Packet *packet){
   printf("\n%d | %s | %s | %s | %s",
       packet->operation_id,
       packet->source_MAC,
       packet->source_IP,
       packet->destn_MAC,
       packet->destn_IP
   );
   return;
}

#endif
```

4. server.c - Server-side script

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

#ifndef tcp_socket
   #include "tcp_socket.h"
#endif
#ifndef ARP_Packet_h
   #include "ARP_Packet.h"
#endif
#ifndef msg_io
   #include "msg_io.h"
#endif
#ifndef ClientList
   #include "ClientList.h"
#endif

int num_clients = 0;
int self_socket;
int *client_sockets;

void* accept_connections(){
```

```
   do{
       struct sockaddr_in *client_addr = malloc(sizeof(struct
sockaddr_in));
       int client_addr_len = sizeof(struct sockaddr_in);
       int client_addr_port;
       char *client_addr_ip_str =
(char*)malloc(sizeof(char)*ADDRESS_BUFFER_SIZE);
       int client_socket = accept_client(self_socket, client_addr,
&client_addr_len);
       if (client_socket<0){
           continue;
       }
       else{
           client_addr_ip_str =
(char*)malloc(sizeof(char)*ADDRESS_BUFFER_SIZE);
           inet_ntop(ADDRESS_FAMILY, (void*)&client_addr->sin_addr,
client_addr_ip_str, ADDRESS_BUFFER_SIZE);
           client_addr_port = (int)ntohs(client_addr->sin_port);
           if (client_addr_ip_str == NULL) {
               printf("\n\nNew client connected.\nCould not read
address\n");
           }
           else{
               printf("\n\nNew Client Connected (%s:%d)\n",
client_addr_ip_str, client_addr_port);
           }
           *(client_sockets+num_clients) = client_socket;
           num_clients++;
       }
   }while(1==1);
}


void main(){

   self_socket = make_socket();
   if(self_socket<0){
       printf("\nCould not create socket. Retry!\n");
       return;
   }
```

```
    if (bind_server_socket(self_socket)<0){
        printf("\nCould not bind server socket. Retry!\n");
        destroy_socket(self_socket);
        return;
    }

    // Prepare address storage locations
    char *self_mac = (char*)malloc(sizeof(char)*MAC_ADDRESS_SIZE);
    char *self_ip = (char*)malloc(sizeof(char)*IP_ADDRESS_SIZE);
    // Store own MAC and IP
    printf("\nEnter Own MAC Address: ");
    scanf(" %s", self_mac);
    printf("Enter Own IP Address: ");
    scanf(" %s", self_ip);

    if (initiate_listen(self_socket)<0){
        printf("\nCould not listen on server socket. Retry!\n");
        destroy_socket(self_socket);
        return;
    }
    else{
        printf("\nServer listening for connections from all local
interfaces...\n");
    }

    // Client details storage initialization
    struct sockaddr_in *client_addr = malloc(sizeof(struct sockaddr_in));
    int client_addr_len = sizeof(struct sockaddr_in);
    int client_addr_port;
    char *client_addr_ip_str =
(char*)malloc(sizeof(char)*ADDRESS_BUFFER_SIZE);
    // BLOCKING routine to accept first client
    int client_socket = accept_client(self_socket, client_addr,
&client_addr_len);
    if (client_socket<0){
        printf("\nError when connecting to client. Retry!\n");
        destroy_socket(self_socket);
        return;
    }
```

```
    else if(client_addr_len == -1){
        printf("Client connected.\nCould not read address\n");
    }
    else{
        inet_ntop(ADDRESS_FAMILY, (void*)&client_addr->sin_addr,
client_addr_ip_str, ADDRESS_BUFFER_SIZE);
        client_addr_port = (int)ntohs(client_addr->sin_port);
        if (client_addr_ip_str == NULL) {
            printf("Client connected.\nCould not read address\n");
        }
        else{
            printf("Connected to Client (%s:%d)\n", client_addr_ip_str,
client_addr_port);
        }
    }

    // Keep track of all client sockets.
    client_sockets = (int*)malloc(sizeof(int)*num_clients);
    *(client_sockets+0) = client_socket;
    num_clients++;

    pthread_t listener_thread_id;
    if(pthread_create(&listener_thread_id, NULL,
(void*)(accept_connections), NULL)){
        printf("\nError while creating thread for Client Connections\n");
    }

    // Parent process to handle message transmission
    char *find_mac = (char*)malloc(sizeof(char)*MAC_ADDRESS_SIZE);
    char *find_ip = (char*)malloc(sizeof(char)*IP_ADDRESS_SIZE);
    char *msg_buffer = (char*)malloc(sizeof(char)*MSG_BUFFER_SIZE);
    char *msg_data = (char*)malloc(sizeof(char)*MSG_BUFFER_SIZE);

    // Prepare data transmission loop
    fd_set readable_fds;
    int msg_size = 0;
    int response;
    int read_fd, write_fd;
    char ch = 'n';
    char *arp_packet_string;
```

```
   ARP_Packet *arp_packet;
   do{
       // Accept Destination Address
       printf("\n----------------------------------------------");
       printf("\nEnter Destination IP Address: ");
       scanf(" %s", find_ip);
       // Accept message
       printf("Enter 16-bit Message: ");
       scanf(" %s", msg_data);
       // Prepare request packet
       arp_packet = make_arp_packet(REQUEST_OPERATION_ID, self_mac,
self_ip, EMPTY_MAC_ADDRESS, find_ip);
       arp_packet_string = serialize_arp_packet(arp_packet);
       printf("\n%s", arp_packet_string);
       // Send request packet to all clients
       for(int i=0;i<num_clients;i++){
           write_fd = *(client_sockets+i);
           msg_size = write(write_fd, arp_packet_string,
ARP_PACKET_STRING_SIZE);
       }
       // BLOCK till some client sends a response
       printf("\nARP-Request broadcasted, waiting for response...\n");
       response = wait_for_message(client_sockets, num_clients,
&readable_fds);
       if(response == -9){
           printf("\nTimed out when waiting for responses\nDropping
data...\n");
           continue;
       }
       else if(response == -8){
           printf("\nError occurred when monitoring socket for
responses\nRetry!\n");
       }
       // Handle all available readable descriptors
       for(int read_idx=0; read_idx<num_clients; read_idx++){
           read_fd = *(client_sockets+read_idx);
           if (FD_ISSET(read_fd, &readable_fds)==0){
               // This socket is not readable
               continue;
           }
```

```
            msg_size = receive_message(read_fd, msg_buffer, client_addr,
&client_addr_len);
            arp_packet = retrieve_arp_packet(msg_buffer);
            if(arp_packet==NULL){
                printf("\nUnexpected response received...\n");
                printf("%s\n%s", msg_buffer);
            }
            else{
                printf("\nARP-Response Received\n%s", msg_buffer);
                msg_size = write(read_fd, msg_data, MSG_BUFFER_SIZE);
                printf("\n\nData transmitted");
            }
        }
        printf("\nMore messages to send? (y/n): ");
        scanf(" %c", &ch);
    }while(ch!='n');

    destroy_socket(self_socket);
    return;
}
```

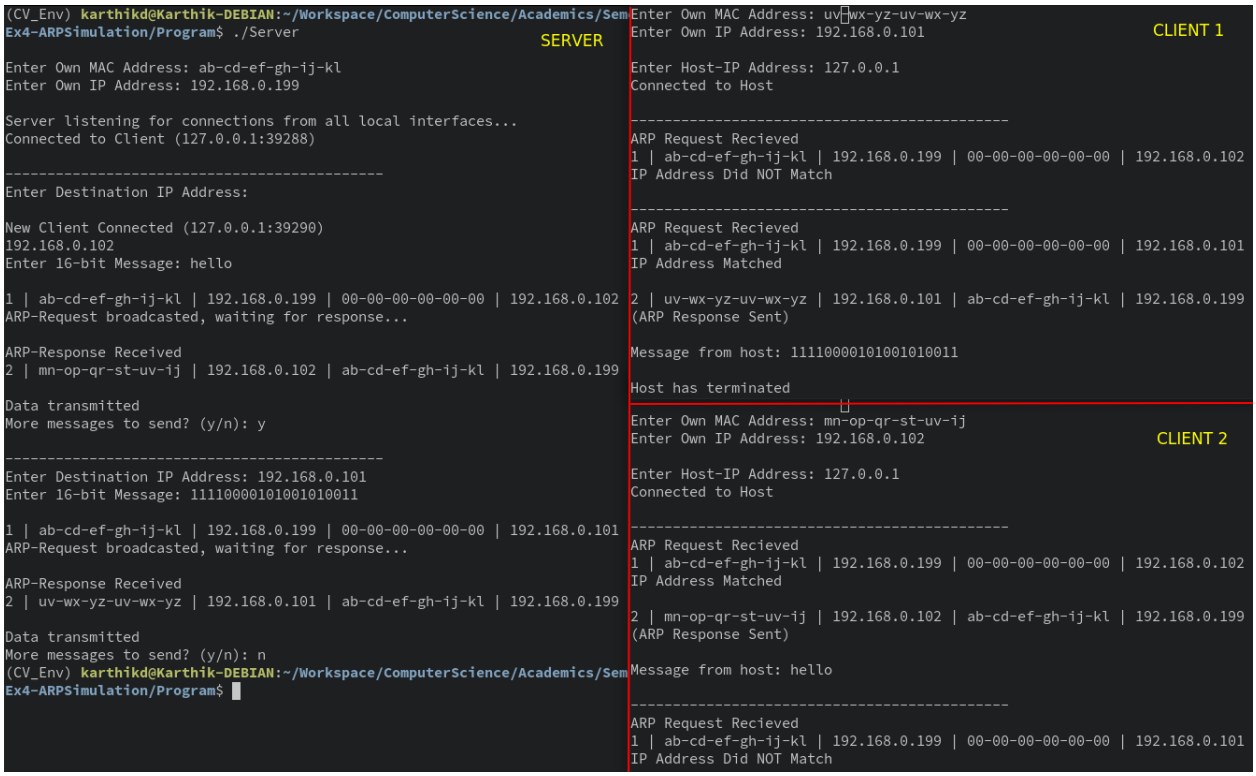
5. client.c - Client-side script

```
#include<stdio.h>
#include<stdlib.h>

#ifndef tcp_socket
    #include "tcp_socket.h"
#endif
#ifndef ARP_Packet_h
    #include "ARP_Packet.h"
#endif

void main(){

    int self_socket = make_socket();
    if(self_socket<0){
        printf("\nCould not create socket. Retry!\n");
        return;
```

```
    }

    char *self_mac = (char*)malloc(sizeof(char)*MAC_ADDRESS_SIZE);
    char *self_ip = (char*)malloc(sizeof(char)*IP_ADDRESS_SIZE);
    // Store own MAC and IP
    printf("\n\nEnter Own MAC Address: ");
    scanf(" %s", self_mac);
    printf("Enter Own IP Address: ");
    scanf(" %s", self_ip);

    char *server_ip = (char*)malloc(sizeof(char)*IP_ADDRESS_SIZE);
    printf("\nEnter router-IP Address: ");
    scanf(" %s", server_ip);
    if (connect_server(self_socket, server_ip) < 0){
        printf("\nCould not connect to router.\nMake sure the server is
running!\n");
        destroy_socket(self_socket);
        return;
    }
    else{
        printf("Connected to router\n");
    }

    // Allocate message memory
    char *msg_buffer = (char*)malloc(sizeof(char)*MSG_BUFFER_SIZE);
    char *arp_packet_string;
    int msg_size = 0;

    do {
        bzero(msg_buffer, MSG_BUFFER_SIZE);
        msg_size = read(self_socket, msg_buffer, MSG_BUFFER_SIZE);
        if(msg_size==0){
            printf("\nrouter has terminated\nExiting...\n");
            break;
        }
        // Attempt packet read, else print message
        ARP_Packet *arp_packet = retrieve_arp_packet(msg_buffer);
        if(arp_packet==NULL){
            printf("\nMessage from router: %s\n", msg_buffer);
        }
```

```
        else{
            printf("\n----------------------------------------------");
            printf("\nARP Request Recieved\n%s", msg_buffer);
            if(is_destn(arp_packet, self_ip)){
                printf("\nIP Address Matched\n");
                arp_packet = make_arp_packet(RESPONSE_OPERATION_ID,
self_mac, self_ip, arp_packet->source_MAC, arp_packet->source_IP);
                arp_packet_string = serialize_arp_packet(arp_packet);
                msg_size = write(self_socket, arp_packet_string,
ARP_PACKET_STRING_SIZE);
                printf("\n%s", arp_packet_string);
                printf("\n(ARP Response Sent)\n");
            }
            else{
                printf("\nIP Address Did NOT Match\n");
            }
        }
        fflush(stdout);
    }while(1==1);
}
```

Sample Output



Result

Implemented a socket program in C language using TCP to simulate ARP in network communication.. A connection-oriented multi-client server is developed, wherein a server broadcasts an ARP request packet to all its clients. The suitable client responds with an ARP response packet. Upon receiving a valid ARP response, the server sends the data message to the client. Through this implementation, the following aspects were understood:

1. Basic functioning of the ARP protocol
2. Order of packet contents in ARP response and request is understood and simulated
3. Implementation details of socket programming using C language for TCP protocol
4. Use of fd_sets and select() method for checking available FDs