Ex. No. 3 **UDP Multi-Client Chat Server** **Karthik D**

2 August 2021 **UCS1511 - Networks Lab** **195001047**

Aim

To develop a chat application between a client and a server using UDP in the C language and then update the program to support multiple clients using fd_set() and select() functions.

Question

Develop a C program to implement a UDP driven multi-client chat server. A client must be able to connect to a specific server, supplying the address of the server. The client should be able to send a text message to the server. The server should be able to detect the arrival of a message, identify the client address and send a reply to the same client.

Algorithms

(a) Server-side

Step 1: Create a network socket with parameters suitable for an end-point of UDP based communication

Step 2: 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 3: Maintain a list of all sockets of the server called *server_sockets* (In this implementation, there is only one socket for the server)

Step 4: Maintain a predefined length of list to store known client addresses called *known_clients*. This list should be empty initially

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:** Populate an *fd_set* instance with all the server sockets in *server_sockets*.
- **ii:** Use the *select()* function to await a message by checking for the readability of the sockets in the set defined in the previous step. Attach a predefined timeout to this wait. If the timeout expires before a message arrives, exit the loop.

iii: When a message arrives before the timeout expires, read the server socket to obtain the message. Use the *recvfrom()* system call to obtain the client address as well.

iv: Check if the client exists in the known_clients list. If so, display the client ID along with the message received. If not, add the new client to the list, display the client ID and the message received. Continue from step-(vii)

v: If the list is already full, send back a reply indicating that the server has reached its client limit and continue from step-(i)

vi: If the client is known and the message sent by the client indicates that the client is exiting, remove this client from the *known_clients* list and acknowledge the same to the client. Continue from step-(i)

vii: Accept a reply from the user. Send this message to the client using the *sendto()* system call, specifying the client address

(Repeat till server times out while waiting for a message)

Step 7: Close the created socket using the *close()* system call and terminate the process

(b) **Client-side**

Step 1: Create a network socket with parameters suitable for an end-point of UDP based communication

Step 2: Accept the target server address as input from the user

Step 3: Using the accepted address and a preset port number agreed upon between the server and client

Step 4: Prepare a memory buffer to read and store messages from the connection.

Step 5: Start an infinite loop to perform the following operations,

i: Accept a multiline string message from the user.

ii: Use the *sendto()* system call to send this message through the client's socket to the intended server, specifying the server address

iii: Wait for the server to send a reply using the *recvfrom()* system call

iv: When a message arrives, check if it was sent from the intended server.

v: If the message indicates that the server has reached its client limit, display a suitable message to the user and exit the loop

vi: If the message is an acknowledgement from the server to indicate that it has accepted the client's request to terminate itself then, exit the loop.

vii: Otherwise, display this message to the user and continue from step-(i)

(Repeat till server is busy or acknowledges connection termination)

Step 6: Close the created socket using the *close()* system call and terminate the process

<u>C Program Code</u>

1. <u>udp_socket.h - UDP connection helper functions</u>

```
#ifndef udp_socket
#define udp_socket

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

#define SERVER_PORT 8088
#define LOCALHOST_IP "127.0.0.1"
#define ADDRESS_FAMILY AF_INET
#define ADDRESS_BUFFER_SIZE 30
#define MSG_BUFFER_SIZE 100
#define IP_STRING_LEN 24
#define TERMINATION_INIT_STRING "FORGETCLIENT"
#define TERMINATION_ACK_STRING "FORGETCLIENT_ACK"
#define SERVER_REJECT_STRING "SERVER_IS_BUSY"
#define MSG_DELIMITER ';'
#define MSG_WAIT_TIMEOUT 20

/*
Use BLOCKING sockets (default configuration)
Alternating send-receive only
```

```
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_DGRAM, IPPROTO_UDP);
   if (sock_fd == -1){
       return -2;    // Could not create socket
   }
   return sock_fd;
}

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
   }
}

struct sockaddr_in wrap_address(char *ip_address, int port){
   struct sockaddr_in address;
   bzero((char*)&address, sizeof(address));
   // Set family to IPv4
   address.sin_family = ADDRESS_FAMILY;
   // Set port in network byte-order to a non-privileged port (>1023)
   address.sin_port = htons(port);
   // Set the ip address in byte format
   address.sin_addr.s_addr = inet_addr(ip_address);
   return address;
}

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

#endif
```

2. <u>msg_io.h - Message transfer helper functions</u>

```

#ifndef msg_io
#define msg_io

#include<sys/types.h>

#include "udp_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 *server_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(*(server_fds+0), &read_fds);
       if (*(server_fds+0) > max_fd){
           max_fd = *(server_fds+0);
       }
   }
   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;
}

ssize_t send_message(int socket, char *msg, struct sockaddr_in
*destn_addr, int destn_addr_len){
   // Using MSG_CONFIRM to prevent ARP reprobing, since this is a reply
message
   int msg_size = sendto(socket, msg, MSG_BUFFER_SIZE, MSG_CONFIRM,
(struct sockaddr*)destn_addr, destn_addr_len);
   if(msg_size == -1){
       return -6;     // Couldn't send message
   }
   return msg_size;
}

ssize_t send_reply(int socket, char *msg, struct sockaddr_in *destn_addr,
int destn_addr_len){
   // Using MSG_CONFIRM to prevent ARP reprobing, since this is a reply
message
```

```
   int msg_size = sendto(socket, msg, MSG_BUFFER_SIZE, MSG_CONFIRM,
(struct sockaddr*)destn_addr, destn_addr_len);
   if(msg_size == -1){
       return -7;     // Couldn't send reply
   }
   return msg_size;
}

#endif
```

3. server.c - Server-side program

```

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

#ifndef udp_socket
   #include "udp_socket.h"
#endif

#ifndef msg_io
   #include "msg_io.h"
#endif

#ifndef client_list
   #include "ClientList.h"
#endif

void main(){

   int 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;
```

```
    }

    // Keep track of all server sockets.
    int num_sockets = 1;
    int *server_sockets = (int*)malloc(sizeof(int)*num_sockets);
    *(server_sockets+0) = self_socket;

    // To keep track of known clients
    ClientList *known_clients = make_empty_client_list();
    // To store sender-client address
    struct sockaddr_in *client_addr = malloc(sizeof(struct sockaddr_in));
    int client_addr_len = sizeof(struct sockaddr_in);
    char *client_addr_ip_str =
(char*)malloc(sizeof(char)*ADDRESS_BUFFER_SIZE);
    int client_addr_port;

    char *msg_buffer = (char*)malloc(sizeof(char)*MSG_BUFFER_SIZE);
    fd_set readable_fds;
    int msg_size = 0;
    int response;
    int read_fd;
    int client_id;
    do{
        // BLOCK till some client sends message

printf("\n----------------------------------------------------------------"
);
        printf("\nServer waiting for client messages from all local
interfaces...\n\n");

        response = wait_for_message(server_sockets, num_sockets,
&readable_fds);
        if(response == -9){
            printf("\nTimed out when waiting for messages\nExiting...\n");
            break;
        }
        else if(response == -8){
            printf("\nError occurred when monitoring socket for
messages\nRetry!\n");
        }
        // Handle all available readable descriptors
        for(int read_idx=0; read_idx<num_sockets; read_idx++){
            read_fd = *(server_sockets+read_idx);

            if (FD_ISSET(read_fd, &readable_fds)==0){
                // This socket is not readable
                continue;
            }

            msg_size = receive_message(self_socket, msg_buffer,
client_addr, &client_addr_len);
            if (msg_size==0){
                printf("\nEmpty message\n");
            }
            else if(client_addr_len == -1){
                printf("\nMessage received from Client.\nCould not read
address\n");
            }
            else{
                // Alternatively, use inet_ntoa
                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("\nMessage received from Client.\nCould not read
address\n");
                    continue;
                }
                else{
                    client_id = find_or_add_client(client_addr,
known_clients);
                    if(client_id==-11){
                        printf("\nNew client rejected and acknowledged.
Client limit reached!\n");
                        msg_size = send_reply(self_socket,
SERVER_REJECT_STRING, client_addr, client_addr_len);
                        continue;
                    }
                    else if(strcmp(msg_buffer,
TERMINATION_INIT_STRING)==0){
                        client_id = remove_client(client_addr,
known_clients);
                        msg_size = send_reply(self_socket,
TERMINATION_ACK_STRING, client_addr, client_addr_len);
                        printf("\nClient-%c (%s:%d) Removed from known
clients\n", (65+client_id), client_addr_ip_str, client_addr_port);
                        continue;
                    }
                    printf("\nMessage received from \nClient-%c (%s:%d): ",
(65+client_id), client_addr_ip_str, client_addr_port);
                }
                printf("%s", msg_buffer);
                // Replying back
                bzero(msg_buffer, MSG_BUFFER_SIZE);
                printf("\nEnter Reply: ");
                scanf(" %[^;]s", msg_buffer);
                // Consume the last newline character from read-buffer
                getchar();
                msg_size = send_reply(self_socket, msg_buffer, client_addr,
client_addr_len);
            }
        }
    }while(1==1);

    destroy_socket(self_socket);
    return;
}
```

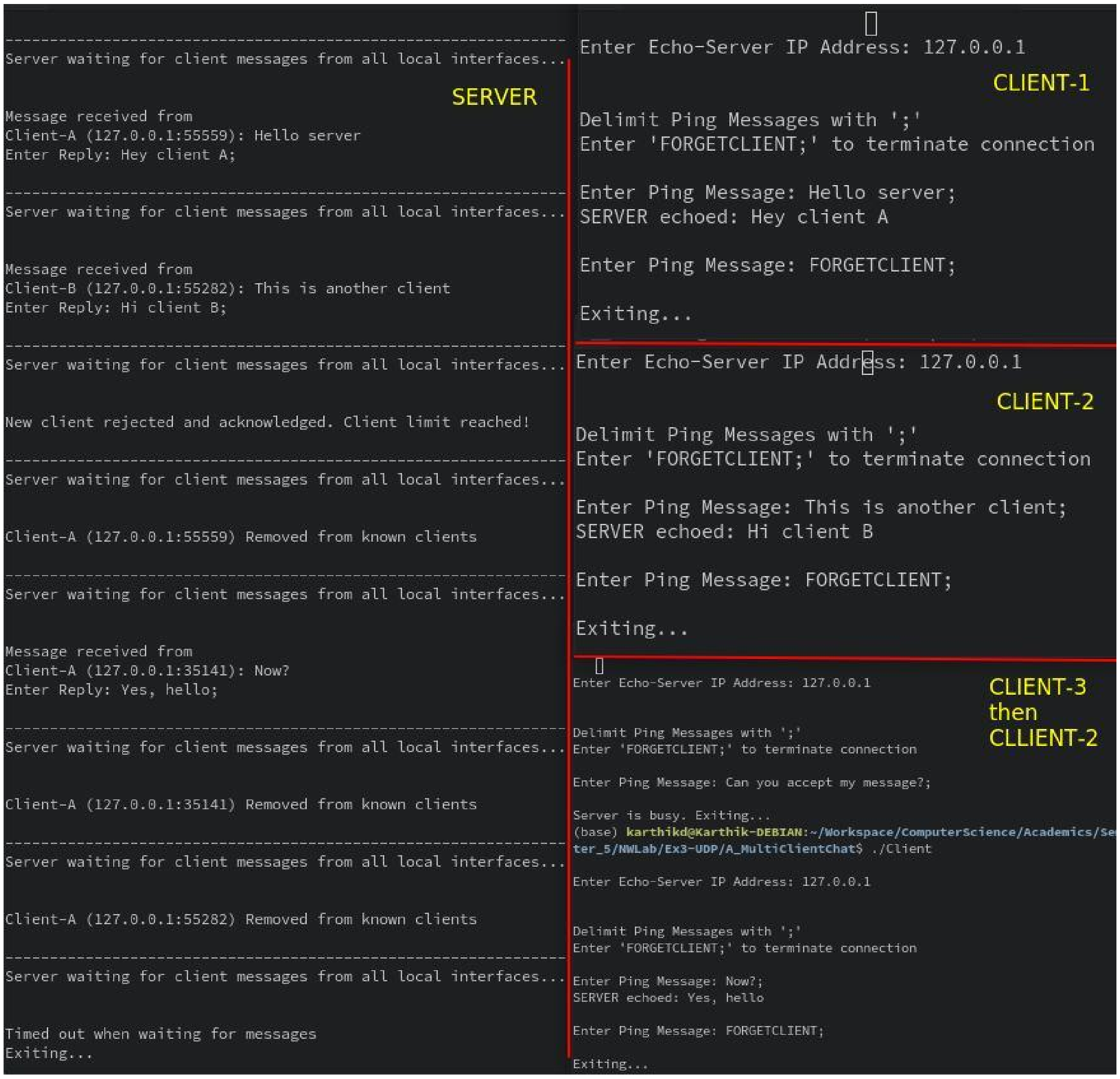
4. <u>client.c - Client-side program</u>

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

#ifndef udp_socket
    #include "udp_socket.h"
#endif
```

```c
#ifndef msg_io
   #include "msg_io.h"
#endif

void main(){

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

   char *server_ip = (char*)malloc(sizeof(char)*IP_STRING_LEN);
   printf("\nEnter Echo-Server IP Address: ");
   scanf(" %s", server_ip);
   struct sockaddr_in server_addr = wrap_address(server_ip, SERVER_PORT);
   int server_addr_len = sizeof(server_addr);

   // Structure to store the message source address
   struct sockaddr_in *source_addr = malloc(sizeof(struct sockaddr_in));
   int source_addr_len = sizeof(struct sockaddr_in);

   char *msg_buffer = (char*)malloc(sizeof(char)*MSG_BUFFER_SIZE);
   int msg_size = 0;
   printf("\n\nDelimit Ping Messages with ';'\nEnter 'FORGETCLIENT;' to
terminate connection\n");
   do {
       bzero(msg_buffer, MSG_BUFFER_SIZE);
       printf("\nEnter Ping Message: ");
       scanf(" %[^;]s", msg_buffer);
       // Consume the last newline character from read-buffer
       getchar();
       msg_size = send_message(self_socket, msg_buffer, &server_addr,
server_addr_len);
       // Reading back
       bzero(msg_buffer, MSG_BUFFER_SIZE);
       msg_size = receive_message(self_socket, msg_buffer, source_addr,
&source_addr_len);
       if(strcmp(msg_buffer, SERVER_REJECT_STRING)==0){
           printf("\nServer is busy. Exiting...\n");
```

```
            break;
        }
        else if(strcmp(msg_buffer, TERMINATION_ACK_STRING)==0){
            printf("\nExiting...\n");
            break;
        }
        printf("SERVER echoed: %s\n", msg_buffer, MSG_BUFFER_SIZE);
    }while(1==1);

    destroy_socket(self_socket);
    return;
}
```

Sample Outputs



```
-----------------------------------------------------------------
Server waiting for client messages from all local interfaces...

Message received from
Client-A (127.0.0.1:55559): Hello server
Enter Reply: Hey client A;

-----------------------------------------------------------------
Server waiting for client messages from all local interfaces...


Message received from
Client-B (127.0.0.1:55282): This is another client
Enter Reply: Hi client B;

-----------------------------------------------------------------
Server waiting for client messages from all local interfaces...


New client rejected and acknowledged. Client limit reached!

-----------------------------------------------------------------
Server waiting for client messages from all local interfaces...


Client-A (127.0.0.1:55559) Removed from known clients

-----------------------------------------------------------------
Server waiting for client messages from all local interfaces...


Message received from
Client-A (127.0.0.1:35141): Now?
Enter Reply: Yes, hello;

-----------------------------------------------------------------
Server waiting for client messages from all local interfaces...


Client-A (127.0.0.1:35141) Removed from known clients

-----------------------------------------------------------------
Server waiting for client messages from all local interfaces...


Client-A (127.0.0.1:55282) Removed from known clients

-----------------------------------------------------------------
Server waiting for client messages from all local interfaces...


Timed out when waiting for messages
Exiting...
```

```
Enter Echo-Server IP Address: 127.0.0.1


Delimit Ping Messages with ';'
Enter 'FORGETCLIENT;' to terminate connection

Enter Ping Message: Hello server;
SERVER echoed: Hey client A

Enter Ping Message: FORGETCLIENT;

Exiting...
```

```
Enter Echo-Server IP Address: 127.0.0.1


Delimit Ping Messages with ';'
Enter 'FORGETCLIENT;' to terminate connection

Enter Ping Message: This is another client;
SERVER echoed: Hi client B

Enter Ping Message: FORGETCLIENT;

Exiting...
```

```
Enter Echo-Server IP Address: 127.0.0.1


Delimit Ping Messages with ';'
Enter 'FORGETCLIENT;' to terminate connection

Enter Ping Message: Can you accept my message?;

Server is busy. Exiting...
(base) karthikd@Karthik-DEBIAN:~/Workspace/ComputerScience/Academics/Se
ter_5/NWLab/Ex3-UDP/A_MultiClientChat$ ./Client

Enter Echo-Server IP Address: 127.0.0.1


Delimit Ping Messages with ';'
Enter 'FORGETCLIENT;' to terminate connection

Enter Ping Message: Now?;
SERVER echoed: Yes, hello

Enter Ping Message: FORGETCLIENT;

Exiting...
```

Result

Implemented a socket program in C language to establish client server communication. A multi-client chat server is developed, wherein a client sends a message to the server and the server in turn, sends back a reply to the client if it is already a known client or can be added to the clients list without exceeding the maximum number of allowed clients. Through this implementation, the following aspects were understood:

1. Basic functioning of the UDP protocol
2. Connectionless communication between client and server
3. Implementation details of socket programming using C language