Socket Programming 2

Sharan Narasimhan, CS20MTECH14003
October 4th, 2020

1. Foreword to Evaluators:

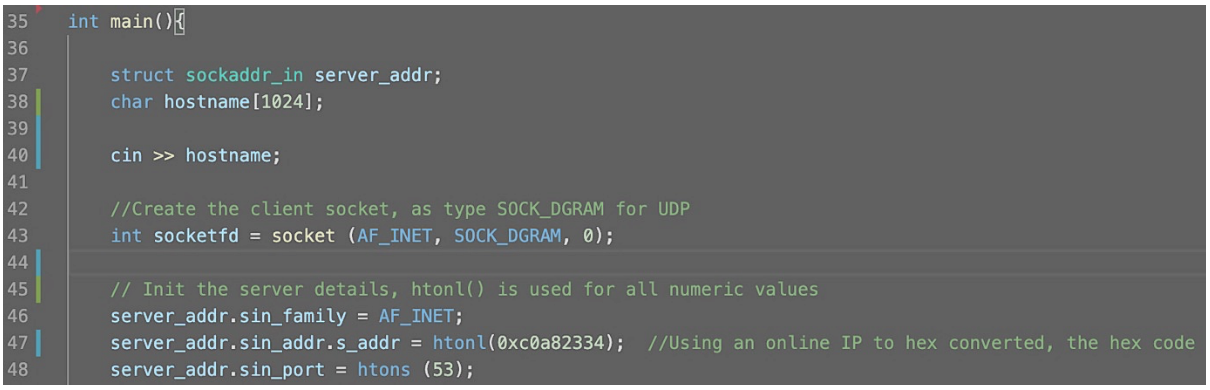
Please refer to the README.txt file for instructions on how to execute. Please ignore the following: *.vscode, .DS_Store, .git.*

For Task 1, I have implemented the source code up to the DNS response processing section. So it is only able the send the DNS packet to server and retrieve the unprocessed response. I had to refer to online resources for the theory and have used the two links the TA provided us. I also had to refer to few DNS tutorials in order to understand certain code segments. All these sources have been referenced wherever needed. I have briefly documented whatever I have learnt from these online tutorials as well.

For Task 2, I have made use of the FD API and the select() function in order facilitate multi socket servicing. All online documentation/tutorials used have been referenced. Code comments have been added for explanation and I have discussed only critical parts of the code in the report.

2. Simple DNS Client using UDP

In *Line 35-49*, I have used the same methods used in the earlier assignments to initialise the UDP Client and configure the DNS Server Address. Using an online IP to Hex tool, I obtained the hex form of the IP for the IITH DNS server (declared in *Line 48*).



```
35   int main(){
36
37       struct sockaddr_in server_addr;
38       char hostname[1024];
39
40       cin >> hostname;
41
42       //Create the client socket, as type SOCK_DGRAM for UDP
43       int socketfd = socket (AF_INET, SOCK_DGRAM, 0);
44
45       // Init the server details, htonl() is used for all numeric values
46       server_addr.sin_family = AF_INET;
47       server_addr.sin_addr.s_addr = htonl(0xc0a82334);  //Using an online IP to hex converted, the hex code
48       server_addr.sin_port = htons (53);
```

Figure 2.1

A DNS packet consists of 5 sections, Header, Question, Answer (also contains ‘Response’), Authority and Additional.

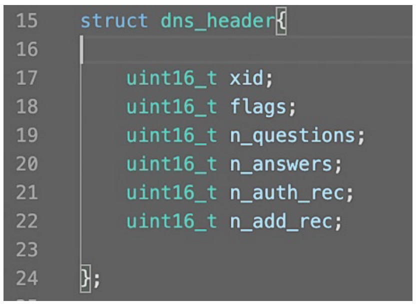
Header:

The header consists of three lines: Random Identifier/ XID (2 Bytes), OPCODE (2 Bytes) and Question Field (2 Bytes).

The XID is a randomly generated 16 bit field created by the client that is used to identify the particular DNS query/response. In the response from DNS Server, the XID should match the sending XID to convey that the response is for the particular query.

The OPCODE is a 16 bit field that consists of various flags/options available for the DNS query. Only the ‘RD’ flag was set to 1, to denote that Recursion method is desired over the iterative method. Thus the corresponding hex code is 0100.

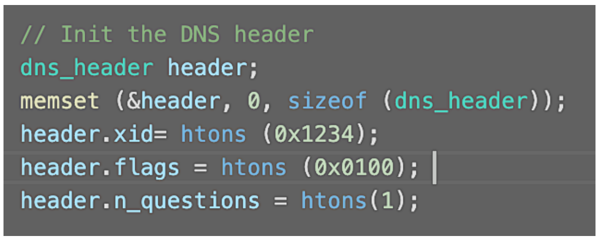
Lastly, the question field conveys how many IPs we want to resolve. So in this case the corresponding 16 bit code is: 0001 0000 0000 0000. The structure *dns_header* was declared to store the header information (Figure 2.2)



```
struct dns_header{

    uint16_t xid;
    uint16_t flags;
    uint16_t n_questions;
    uint16_t n_answers;
    uint16_t n_auth_rec;
    uint16_t n_add_rec;

};
```

Figure 2.2

A header object of type *dns_header* was declared and was initialised appropriately with the *XID, OPCODE/Flags* and *No. of questions*. (Figure 2.3)



```
// Init the DNS header
dns_header header;
memset (&header, 0, sizeof (dns_header));
header.xid= htons (0x1234);
header.flags = htons (0x0100);
header.n_questions = htons(1);
```

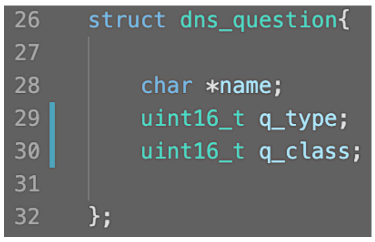
Figure 2.3

Question:

The question consists 2 sections: *QNAME* (variable length), *QCLASS* (1 Byte) and *QTYPE* (1 Byte).

The *QNAME* consists of the name of the domain (e.g. “facebook” in “www.facebook.com” in ASCII form followed by a field with 3 characters corresponding to .COM and continues till the first 0 Byte is read. This *QNAME* field first needs to be pre-processed to the DNS name format (explained later)

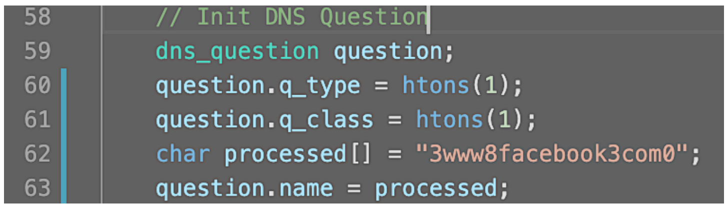
QCLASS is set to IN meaning “Internet” and *QTYPE* corresponds to 1 of the 4 types of record responses requested, ‘A’ in this case for Authoritative. The final 16 bit code is 0001 0001. The structure *dns_question* was defined to store all the information for a DNS question. (Figure 2.4)



```
struct dns_question{

    char *name;
    uint16_t q_type;
    uint16_t q_class;

};
```

Figure 2.4

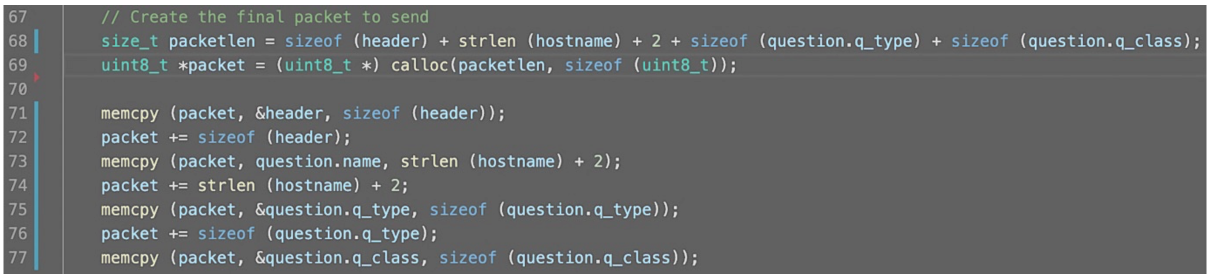
An object question was declared and was initialised appropriately with the domain name, QTYPE and QCLASS. The domain name must first pre-processed to suite the DNS Name format. E.g. www.facebook.com must get transformed to 3www8facebook3com0. I was not able to implement this pre-process function so have directly hardcoded this while declaring the object:



```
    // Init DNS Question
    dns_question question;
    question.q_type = htons(1);
    question.q_class = htons(1);
    char processed[] = "3www8facebook3com0";
    question.name = processed;
```

Figure 2.5

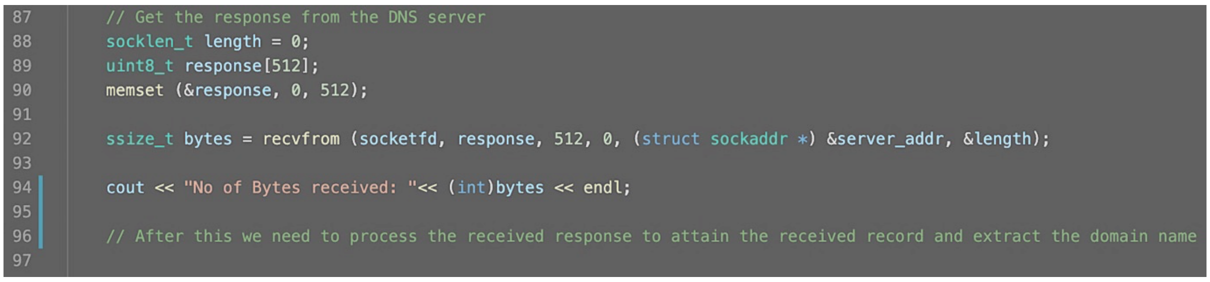
The packet length is first computed in *Line 68*. We need to add an extra 2 Bytes for the first extra byte store ‘3’ and the last null byte storing ‘0’ in the processed hostname. Memory is allocated and the address is stored in the pointer packet. We then proceed to copy the Header and Question data incrementally into the memory space as shown:



```
// Create the final packet to send
size_t packetlen = sizeof (header) + strlen (hostname) + 2 + sizeof (question.q_type) + sizeof (question.q_class);
uint8_t *packet = (uint8_t *) calloc(packetlen, sizeof (uint8_t));

memcpy (packet, &header, sizeof (header));
packet += sizeof (header);
memcpy (packet, question.name, strlen (hostname) + 2);
packet += strlen (hostname) + 2;
memcpy (packet, &question.q_type, sizeof (question.q_type));
packet += sizeof (question.q_type);
memcpy (packet, &question.q_class, sizeof (question.q_class));
```

Figure 2.6

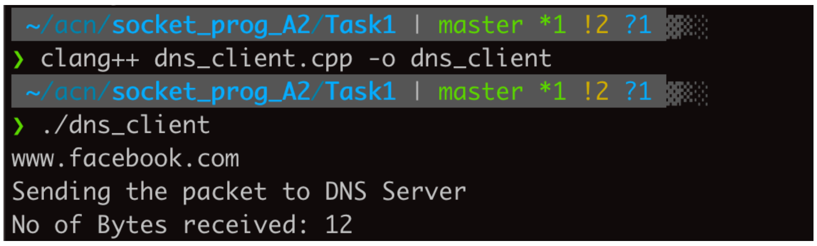
We then initialise an object response and allocation its memory with 0's. The packet is sent and the number of bytes received stored in the variable bytes as shown:



```
// Get the response from the DNS server
socklen_t length = 0;
uint8_t response[512];
memset (&response, 0, 512);

ssize_t bytes = recvfrom (socketfd, response, 512, 0, (struct sockaddr *) &server_addr, &length);

cout << "No of Bytes received: "<< (int)bytes << endl;

// After this we need to process the received response to attain the received record and extract the domain name
```

Figure 2.7

The output obtained after compiling and executing the program is shown below:



```
 ~/acn/socket_prog_A2/Task1 | master *1 !2 ?1
❯ clang++ dns_client.cpp -o dns_client
 ~/acn/socket_prog_A2/Task1 | master *1 !2 ?1
❯ ./dns_client
www.facebook.com
Sending the packet to DNS Server
No of Bytes received: 12
```

Figure 2.2

The online tutorial that was mainly used:
https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/UDPSockets.html.

For more theory on the DNS header and its corresponding flags I used:
https://www2.cs.duke.edu/courses/fall16/compsci356/DNS/DNS-primer.pdf

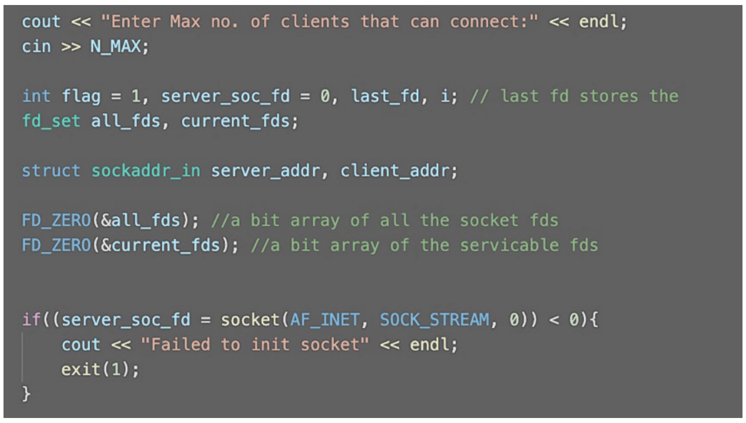
3. Multi-Client Chat Server using select()

Instead of using multi-threading, It seemed more effective to use the select() function. Sockets are treated identically to files (referenced by file descriptors) with respect to the kernel. This enables select() to be used to fetch and push information into sockets whenever an FD 'event' occurs.

General Architecture:

A. Server's Perspective:

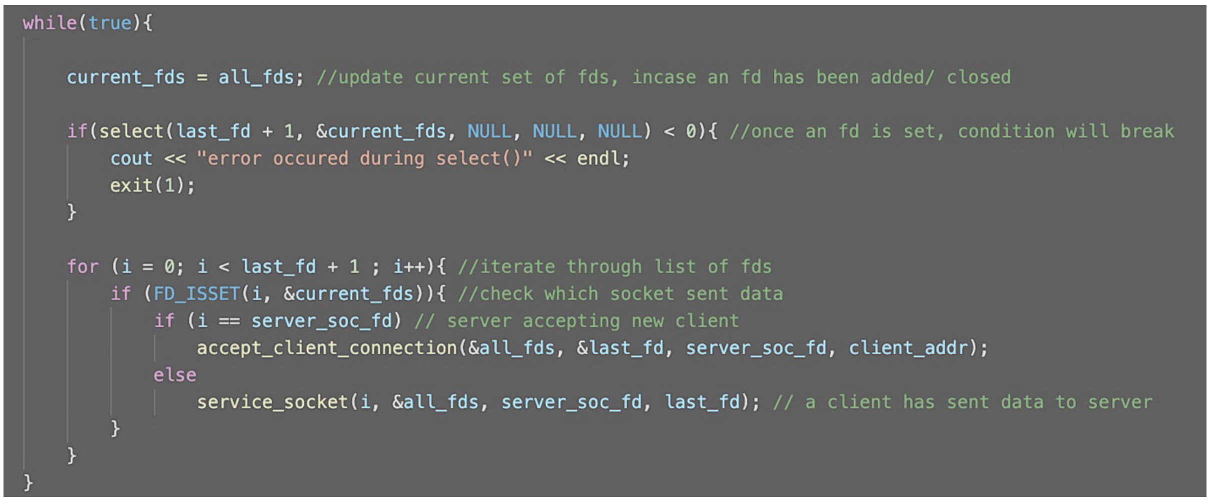
1. The server maintains a list (a bit array) of all the connected sockets.
2. The select() function is a blocking condition, which breaks once one of the fds in the set pushes data to its stream.
3. A for loop iterates through the list of connected sockets and gets the socket FD of the client that pushed the new data.
4. If the FD corresponds to the server itself, this means a new client is trying to connect, so the server tries to accept this new connection.
5. Else the FD is from a client's socket stream, meaning it has sent a message.
6. The server broadcasts this message to all the other connected clients.
7. If the number of clients == N_MAX, the server will not accept any more clients.



```
cout << "Enter Max no. of clients that can connect:" << endl;
cin >> N_MAX;

int flag = 1, server_soc_fd = 0, last_fd, i; // last fd stores the
fd_set all_fds, current_fds;

struct sockaddr_in server_addr, client_addr;

FD_ZERO(&all_fds); //a bit array of all the socket fds
FD_ZERO(&current_fds); //a bit array of the servicable fds


if((server_soc_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
    cout << "Failed to init socket" << endl;
    exit(1);
}
```

Figure 2.3

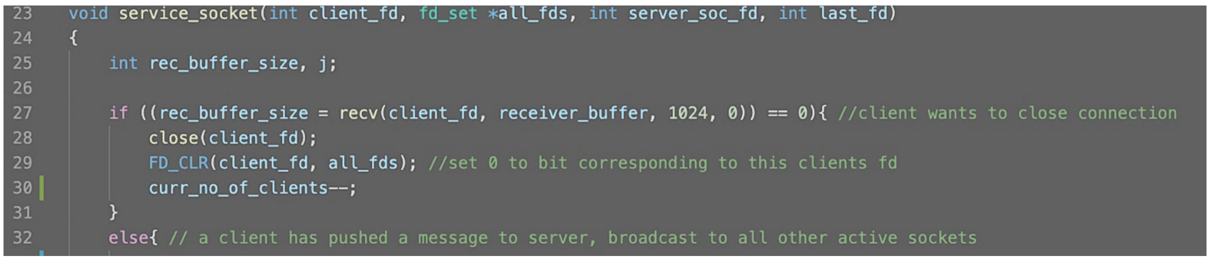
As we can see from the above figure, 2 *fd_set* objects are created and all bits are set to 0. The fd set *all_fds* maintains the set of active sockets/FD that are still participating in the chat. This fd set gets updated once a client enters/leaves the chat. The fd set *current_fds* is passed to the select() function and is responsible for keep track of which socket just pushed data in. After all the necessary sockets are serviced, it is set to *all_fds* in the next round of select() shown below:



```cpp
while(true){

    current_fds = all_fds; //update current set of fds, incase an fd has been added/ closed

    if(select(last_fd + 1, &current_fds, NULL, NULL, NULL) < 0){ //once an fd is set, condition will break
        cout << "error occured during select()" << endl;
        exit(1);
    }

    for (i = 0; i < last_fd + 1 ; i++){ //iterate through list of fds
        if (FD_ISSET(i, &current_fds)){ //check which socket sent data
            if (i == server_soc_fd) // server accepting new client
                accept_client_connection(&all_fds, &last_fd, server_soc_fd, client_addr);
            else
                service_socket(i, &all_fds, server_soc_fd, last_fd); // a client has sent data to server
        }
    }
}
```

Figure 2.4

The below figure shows how the server handles a socket that needs to be serviced. If the received message is empty, it indicates that the client wants to close its connection. The server closes the connections and also updates the *all_fds* fd set, which will update *current_fds* in the next iteration. Else the client wants to send a message, meaning the server needs to capture this message and broadcast it to all other clients.



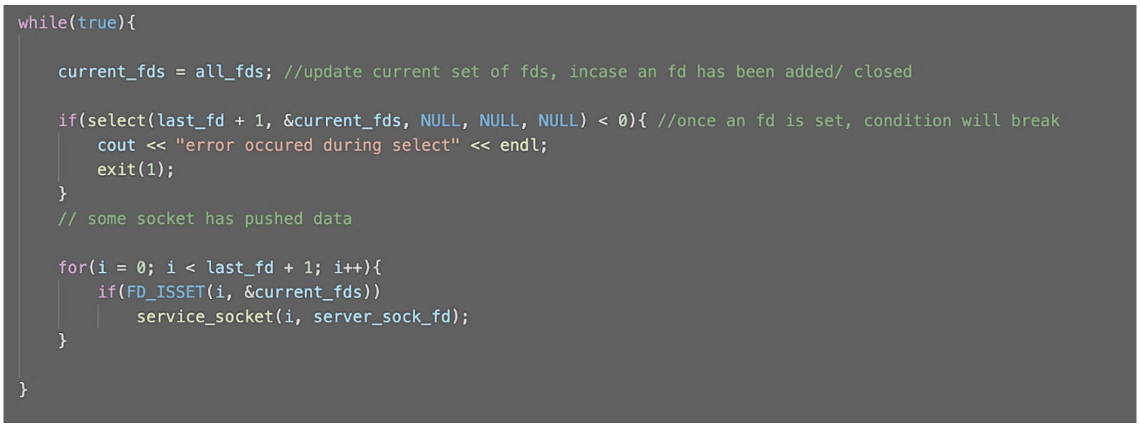
```cpp
void service_socket(int client_fd, fd_set *all_fds, int server_soc_fd, int last_fd)
{
    int rec_buffer_size, j;

    if ((rec_buffer_size = recv(client_fd, receiver_buffer, 1024, 0)) == 0){ //client wants to close connection
        close(client_fd);
        FD_CLR(client_fd, all_fds); //set 0 to bit corresponding to this clients fd
        curr_no_of_clients--;
    }
    else{ // a client has pushed a message to server, broadcast to all other active sockets
```

Figure 2.5

B. Client's Perspective:

1. The client connects to the server and spins in select() waiting for an event.
2. If the server pushes data to the clients stream, the clients select() will pick up this event and break. Or if the client pushes data into its on socket stream it will break.
3. If the FD belong to the server, the client will receive the message.
4. If the FD belongs to the client itself, this message is extracted from stdio and send to the server for broadcasting.

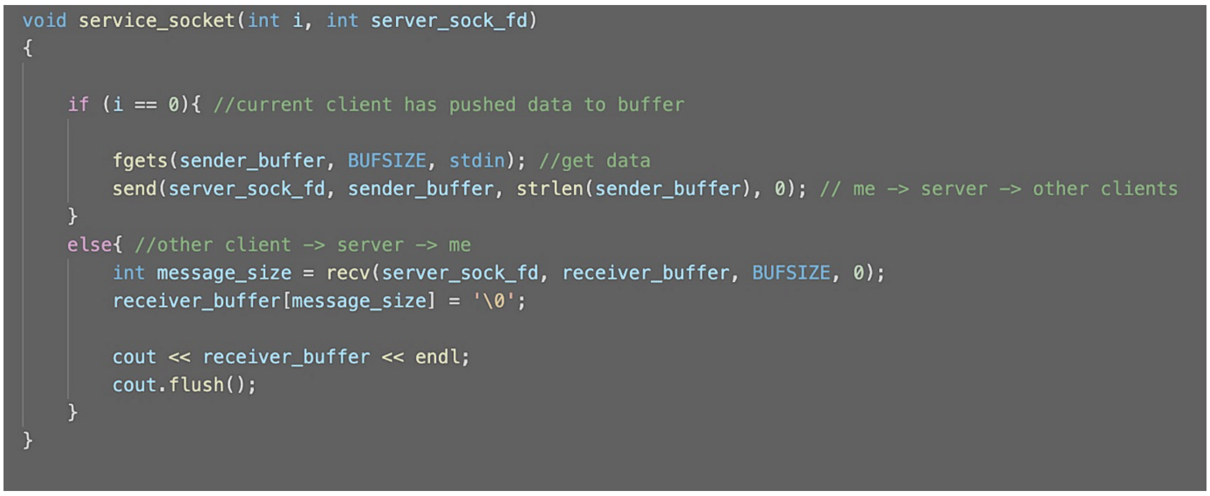
The main() for client.cpp is very similar to server.cpp, as shown below:



```
while(true){

    current_fds = all_fds; //update current set of fds, incase an fd has been added/ closed

    if(select(last_fd + 1, &current_fds, NULL, NULL, NULL) < 0){ //once an fd is set, condition will break
        cout << "error occured during select" << endl;
        exit(1);
    }
    // some socket has pushed data

    for(i = 0; i < last_fd + 1; i++){
        if(FD_ISSET(i, &current_fds))
            service_socket(i, server_sock_fd);
    }

}
```

Figure 2.6

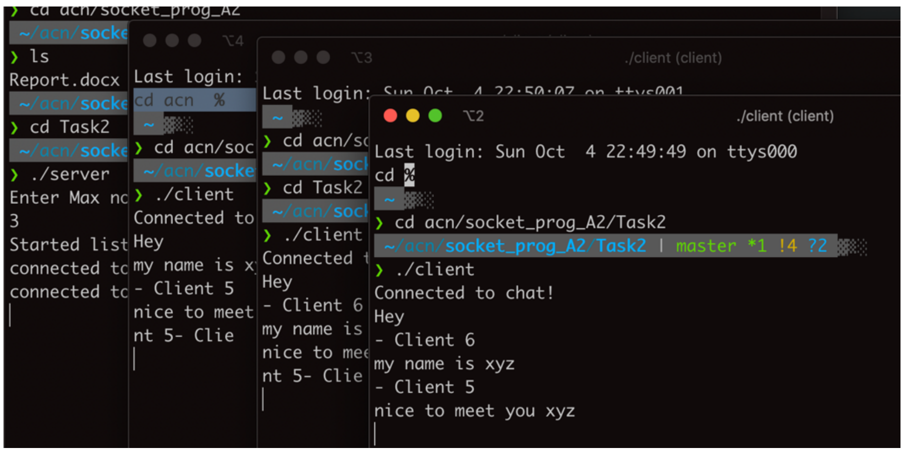
The only difference is how a client services its FD SET array. There are only 2 possibilities for the client, a) either it has seen new data in its own socket, in which case it needs to extract it from stdio and send it to the server. Or b) it has received a new message from the server, in which case it will display it on stdout. The below figure shows the code corresponding to this:



```
void service_socket(int i, int server_sock_fd)
{

    if (i == 0){ //current client has pushed data to buffer

        fgets(sender_buffer, BUFSIZE, stdin); //get data
        send(server_sock_fd, sender_buffer, strlen(sender_buffer), 0); // me -> server -> other clients
    }
    else{ //other client -> server -> me
        int message_size = recv(server_sock_fd, receiver_buffer, BUFSIZE, 0);
        receiver_buffer[message_size] = '\0';

        cout << receiver_buffer << endl;
        cout.flush();
    }
}
```

Figure 2.7

The following output was obtained with 3 clients connected:



```
❯ cd acn/socket_prog_A2
 ~/acn/socke
❯ ls
Report.docx
 ~/acn/socke
❯ cd Task2
 ~/acn/socke
❯ ./server
Enter Max nc
3
Started list
connected tc
connected tc
```

```
⌥4
Last login:
cd acn  %
 ~
❯ cd acn/soc
 ~/acn/socke
❯ ./client
Connected to
Hey
my name is x
- Client 5
nice to meet
nt 5- Clie
```

```
⌥3                                ./client (client)
Last login: Sun Oct  4 22:50:07 on ttys001
 ~
❯ cd acn/sc
 ~/acn/sock
❯ cd Task2
 ~/acn/sock
❯ ./client
Connected t
Hey
- Client 6
my name is
nice to mee
nt 5- Clie
```

```
⌥2                                ./client (client)
Last login: Sun Oct  4 22:49:49 on ttys000
cd %
 ~
❯ cd acn/socket_prog_A2/Task2
 ~/acn/socket_prog_A2/Task2 | master *1 !4 ?2
❯ ./client
Connected to chat!
Hey
- Client 6
my name is xyz
- Client 5
nice to meet you xyz
```

Figure 2.8 (Leftmost is server)

The following tutorial was primarily referenced for Task2:
https://therighttutorial.wordpress.com/2014/06/09/multi-client-server-chat-application-using-socket-programming-tcp/

This concludes my report for the assignment.