ECEN 602
Programming Assignment

Team Number 8

1. Garuda Suma Pranavi (UIN 926009146)
2. Dedeepya Venigalla (UIN 726006161)

We have both equally contributed in developing the sbcp chat server.

Design:

1. For server.c

First, we use getaddress() to refer to the struct sockaddr and we have also taken into account both the IPv4 and IPv6 addresses.

We also use getaddrinfo() to get the port, IP etc. from the console input. Then we use a pointer p to loop through all the addresses in the linked list ai.

Then, the socket() call returns the socket descriptor listen_fd. If listen_fd value is less than 0, there is an error and we will keep calling socket() in that case.

The setsockopt() exception is for multicast sockets. If 2 sockets are bound to the same interface and port and are members of the same multicast group, data will be delivered to both sockets, rather than an arbitrarily chosen one.

Bind() then associates the listen_fd socket returned from socket() to a port on the machine.

Then we listen on listen_fd for n clients (specified as 2nd argument). And then this fd is added to the master file descriptor list and fdmax is set to the value of listen_fd.

Each time we get a new fd, it is added to the read_fds set.

Then the select() function is called to check if there is something to read on any of the fd's.

If there are more clients that can be added to the chat session and they are waiting to be connected, the accept() function listens on the listen_fd fd and creates a new_fd which will send() and recv(). We add the new_fd to the master_set and incremenet the client count by 1.

Now we use recv() to see if there is any new data sent form the client. Rcvd_bytes stores the no. of bytes received, if it has a value of 0 the client has closed the connection, hence the count will decrement by 1 and the exit details of the same will also be sent to all other clients in the chat session.

However, if rcvd_bytes>0, we unpack the message and then check if the user wants to join the system. In that case, we first check if the new received username is same as the ones already in the session. If it is, the fd is closed and removed from the master set. However, if the username has not been used before, it is added to the client list and then broadcast to all the users in the chat session.

If the client asks to send information to the other clients, we unpack the message, read it and then print it on the console and then send it to all the other users of the session.

2. For client.c

First, we use getaddress() to refer to the struct sockaddr and we have also taken into account both the IPv4 and IPv6 addresses.

We also use getaddrinfo() to get the port, IP etc. from the console input. Then we use a pointer p to loop through all the addresses in the linked list ai.

Then, the socket() call returns the socket descriptor sock_fd. If sock_fd value is less than 0, there is an error and we will keep calling socket() in that case.

We then use the connect() function to connect to the server.

Now, we pack the message into the network byte order with the version, type and actual payload. Then the message is sent to the server using send().

sock_fd and console input is then added to read_fds . Then we use select() to check if there is something to read from console, once the input is typed, it is packed and sent to the server.

We also have to check if there is data to be read from server, if there is we read the file descriptor and output the message to the screen.

To implement bonus feature 3, we have taken the timeout value to be 60 seconds rather than the specified 10 seconds to facilitate the join of other clients before any one of them expires.

About the program:

This program is a TCP Broadcast chat server and client, where we can see all he clients currently present in a session and then broadcast messages to all the clients through the server.

We have attached the following files: server.c, client.c and makefile.

How to implement the code:

First run the attached makefile

```
all: server.c client.c
	gcc -o server server.c
	gcc -o client1 client.c
	gcc -o client2 client.c
	gcc -o client3 client.c
clean: server.c client.c
	rm server
	rm client1
	rm client2
	rm client3
```

Then run the server, by writing:

./server server_ip server_port max_clients.
(Here we have used 127.0.0.1, loopback address and any port above 1024 for the server, max_clients is the maximum number of clients that can join the ongoing session.)

Then clients can join by writing:

./clientx username server_ip server_port

This program will now allow upto max_clients to connect to the server and x is the client number, if a username repeats itself, we send out a message, chose another username for the client. If the number of clients exceeds max_clients the client has to wait for someone to exit and the server to clear the resources.

Example: ./client1 c1 127.0.0.1 4000

However, if a client manages to successfully join the chat session it receives a list of the client names already in the session and it can then communicate to all the other clients through the server which broadcasts the received message.

Other clients will receive a message as username message. The client can exit unceremoniously at any point of the program and all the other clients get a message saying that the “client username” has exited.

Source Code

Server.c

```
#include<stdio.h>       //printf, fgets, I/O function
#include<stdlib.h>      //size_t, free, exit(), atoi
#include<string.h>      //memset, strcpy, strcmp
#include<sys/types.h>
#include<sys/socket.h> //sockaddr structure (sa_family, sa_data)
#include<netdb.h>       //addrinfo structure (ai_family, ai_addrelen, *ai_next)
#include<netinet/in.h>   //defines sockaddr_in (sin_family, sin_port, sin_addr, sin_zero)
#include<stdarg.h>       //va_start(), va_arg()
#include<stdint.h>       //int8_t, int16_t, uint32_t
#include<arpa/inet.h>    //htons, ntohl
#include<ctype.h>        //isdigit, isalpha
#include<inttypes.h>     //int8_t, int16_t
#include<unistd.h>       //defines miscellaneous symbolic constants and types and declares
miscellaneous functions #include<stdbool.h>
#include <math.h>


struct attribute
{
   int16_t type;                    // username, message, reason, client count
int16_t length;                    //length of sbcp attrinute     char* atr_msg;
//attribute payload

};

struct message
{
   int8_t version;                      //protocol version is 3     int8_t
type;                          //sbcp message type     int16_t length;
//length of the sbcp message     struct attribute *payload;
// 0 or more sbcp attributes };

//storing an integer into a buffer
```

```
void packin16(char *buffer,unsigned int u)
{
   *buffer++ =u>>8;
   *buffer++ = u;
}

//unpacking integer from the buffer unsigned
int unpackin16(char *buffer)
{
   return (buffer[0]<<8)|buffer[1]; }

//packing()-packing the data specified by form variable into buffer int32_t
packing (char *buffer,char *form, ...)
{     va_list ap;     int8_t a;
//8-bit value     int16_t b;
//16-bit value
   char *c;    //string with length not declared
   int32_t total_size=0,str_len; //total size of the buffer after packing in bytes
va_start(ap, form);
   for(; *form!='\0';form++)
   {
      switch(*form)
      {
         case 'a':       total_size+=1;
a=(int8_t)va_arg(ap, int);
*buffer++=(a>>0)&0xff;                          break;

         case 'b':   total_size+=2;
b=(int16_t)va_arg(ap,int);
packin16(buffer,b);
buffer+=2;                      break;

         case 'c':       c=va_arg(ap, char*);
str_len=strlen(c);
                     total_size+=str_len+2;

                     packin16(buffer,str_len);
                     buffer+=2;

                     memcpy(buffer,c,str_len);
                     buffer+=str_len;
                     break;
      }
   }
```

```
va_end(ap); return
total_size;
}

//unpacking()- unpacking the data specified by form variable from the buffer void
unpacking(char *buffer, char *form, ...)
{     va_list
ap;     int8_t
*a;     int16_t
*b;     char
*c;
   int32_t buf_len, counting, strlength=0;
va_start(ap, form);
   for(; *form!='\0'; form++)
   {
switch(*form)
       {
case 'a':
                a=va_arg(ap, int8_t*);
*a=*buffer++;                   break;
case 'b':
                b=va_arg(ap, int16_t*);
*b=unpackin16(buffer);
buffer+=2;                  break;
case 'c':
                c=va_arg(ap, char*);
buf_len=unpackin16(buffer);
                buffer+=2;
                if(strlength>0 && buf_len>strlength)
                {
                   counting=strlength-1;
                }
else
                {
                   counting=buf_len;
                }

                memcpy(c,buffer,counting);
c[counting]='\0';
buffer+=buf_len;                    break;
default:                  if(isdigit(*form))
                {
                   strlength=strlength*10+(*form-'0');
                }
```

```
        }
        if(!isdigit(*form))
strlength=0;
    }
        va_end(ap);

}

void *getaddress (struct sockaddr *sa)
{
    if (sa -> sa_family == AF_INET)
      {
       return & (((struct sockaddr_in*)sa) -> sin_addr);
      }
    return & (((struct sockaddr_in6*)sa) -> sin6_addr);
}

int main (int argc, char* argv[])
{

    char buffer[1000];     char
username[16];     char
client_names[100][16];     char
name[100];     char
buffer_sent_message [500];     char
buffer_rcvd_message[512];     char
exit_details[50];     struct message
message_rcvd;
    struct attribute attribute_rcvd;

    fd_set master;             //add fd to the master file descriptor list
fd_set read_fds;          //temp file descriptor list for select()     int
fdmax;               //maximum file descriptor number

    FD_ZERO (&master);         //like memset for fds, clearing operation
    FD_ZERO (&read_fds);

    int listen_fd;
    int new_fd;

    struct sockaddr_storage remoteaddr;         //for storing client address
socklen_t addrlen;     struct addrinfo hints, *ai, *p;
```

```
   int rcvd_bytes;           // number of bytes read by the recv function
int optval=1;     int i,j=0;     int count=0;     char str_count[5];     char
count_display[100];
   int u,m;
int rcv=0;
FILE *fp;

   if (argc!=4)
     {
       fprintf (stderr, "correct implementation: ./server server_ip server_port max_clients\n");
exit(1);
     }

   int max_clients= atoi(argv[3]);  //maximum clients allowed

   memset(&hints, 0, sizeof hints);
hints.ai_family= AF_UNSPEC;
   hints.ai_socktype= SOCK_STREAM;

   if ((rcv= getaddrinfo(argv[1], argv[2], &hints, &ai))!=0) // fills out the prederfined structs such
as addrinfo
     {
       fprintf(stderr, "%s\n", gai_strerror(rcv));  //gai_strerror for errors specific to getaddrinfo
exit(1);
     }


   for (p=ai; p!=NULL;p=p->ai_next)
    {
       listen_fd= socket (p->ai_family, p->ai_socktype, p->ai_protocol); //get file descriptors for
all sockets in the list

       if (listen < 0) //if there is an error, go through socket call again
        {
          continue;
        }


   setsockopt (listen_fd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof (int)); //used for
multicast

   if (bind (listen_fd, p->ai_addr, p->ai_addrlen) <0) // bind the socket to the server port
    {
```

```
        close (listen_fd);
continue;
    }

    break;
    }

    if(p==NULL)
    {
       fprintf(stderr, "failed to bind\n");
       exit(2);
    }

    freeaddrinfo (ai); //free address space

    if (listen (listen_fd,3) == -1) // listen for any new connections upto 3 clients
    {
       perror ("listen");
       exit(3);
    }

    printf ("%s", "listen works");

    FD_SET (listen_fd, & master);      //adding listen_fd to master list
fdmax= listen_fd;
         for
(;;)
    {
       read_fds= master;    // listen_fd is added to read_fds          if (select (fdmax+1,
&read_fds, NULL, NULL, NULL) == -1)  //helps the server understand which has
messages to forward and who to forward these messages to.
        {
           perror ("select");
           exit(4);
        }

        for (i=0; i<=fdmax; i++)  //looking for data to read
        {
          if (FD_ISSET (i, &read_fds))
            {
              if (i == listen_fd)    // client waiting to connect
                {
                   if (count < max_clients)
                     {
```

```
                    addrlen = sizeof remoteaddr;
                    new_fd= accept (listen_fd, (struct sockaddr*)&remoteaddr, &addrlen);
if (new_fd == -1)
                     {
                        perror ("accept");
                     }
else
                     {
                        FD_SET (new_fd, &master);
                        if (new_fd > fdmax)
                         {
                            fdmax=new_fd;
                         }
                         count ++;
                     }
                  }
}            else
          {
             if ((rcvd_bytes = recv (i, buffer, 1000, 0)) <= 0)
              {
                if (rcvd_bytes == 0)
                {
                   sprintf (exit_details, " %s OFFLINE : client connection has closed",
client_names[i]);
                   client_names[i][0]='\0';

                for (j=0; j <= fdmax; j++)
                {
                   if (FD_ISSET (j, &master))
                   {
                      if (j!= listen_fd && j!=i)
                      {
                       if (send (j, exit_details, sizeof exit_details, 0)==-1)
                        {
                           perror ("send");
                        }
                      }
                   }
                }
}            else
{
                   perror ("recv");
                }
count --;
```

```
            close (i);
            FD_CLR (i, &master);

}
else
            {
                unpacking(buffer,"aabbb", &message_rcvd.version, &message_rcvd.type,
&message_rcvd.length, &attribute_rcvd.type, &attribute_rcvd.length);

                if (message_rcvd.type=='2' && attribute_rcvd.type==2)
                {
                   unpacking (buffer + 8, "c", username);
int j,m;                        int u=1;
                   for (j=1; j<=fdmax;j++)
                    {
                      if (strcmp(username,client_names[j])== 0)
                       {
                          u =0; //username already in use
                          if (send (i, "choose another username for client", 50, 0)== -1)
                             {
                                perror("send");
                             }
count--;
                          close(i);
                          FD_CLR (i, &master);
                         break;
                     }
                }

                    if (u == 1)
                    {
                       sprintf (client_names[i], "%s", username);
                       sprintf (count_display, "%i", count);
                       strcpy (name, "names of clients in the chat session:");

                    for (m=4; m<=fdmax; m++)
                    {
                       strcat (name, client_names[m]);
                       strcat (name, " ");
                    }
                      strcat (name, count_display);
                    if (send (i, name, 100, 0) == -1)
                    {
```

```
                    perror("send");
                }

            }
        }

    if (message_rcvd.type == '4' && attribute_rcvd. type == 4)
    {
        unpacking (buffer +8 , "c", buffer_rcvd_message);
        sprintf (buffer_sent_message, "%s : %s", client_names[i], buffer_rcvd_message);

    for (j=0; j <=fdmax; j++)
    {
        if (FD_ISSET (j, &master))
        {
            if (j!=i && j!=listen_fd)
            {
                if (send (j, buffer_sent_message, rcvd_bytes, 0 )== -1)
                {
                    perror("send");
                }
            }
        }
    }
    }
}
}
}
} }
   return 0;
}
```

Client.c

```
#include<stdio.h>       //printf, fgets, I/O function
#include<stdlib.h>      //size_t, free, exit(), atoi
#include<string.h>      //memset, strcpy, strcmp
#include<sys/types.h>
#include<sys/socket.h> //sockaddr structure (sa_family, sa_data)
#include<netdb.h>       //addrinfo structure (ai_family, ai_addrelen, *ai_next)
#include<netinet/in.h>   //defines sockaddr_in (sin_family, sin_port, sin_addr, sin_zero)
#include<stdarg.h>       //va_start(), va_arg()
#include<stdint.h>       //int8_t, int16_t, uint32_t
#include<arpa/inet.h>    //htons, ntohl
```

```
#include<ctype.h>        //isdigit, isalpha
#include<inttypes.h>     //int8_t, int16_t
#include<unistd.h>       //defines miscellaneous symbolic constants and types and declares
miscellaneous functions
#include <sys/time.h>   //to declare the wait time and has timeval and timespec structures

struct attribute
{
   int16_t type;                    // username, message, reason, client count
int16_t length;                    //length of sbcp attrinute     char* atr_msg;
//attribute payload

};

struct message
{
   int8_t version;                       //protocol version is 3     int8_t
type;                          //sbcp message type     int16_t length;
//length of the sbcp message     struct attribute *payload;
// 0 or more sbcp attributes };

//storing an integer into a buffer
void packin16(char *buffer,unsigned int u)
{
   *buffer++=u>>8;
   *buffer++=u;
}

//unpacking integer from the buffer unsigned
int unpackin16(char *buffer) {
   return (buffer[0]<<8)|buffer[1];
}

//packing()-packing the data specified by form variable into buffer int32_t
packing (char *buffer,char *form, ...)
{     va_list ap;     int8_t a;
//8-bit value     int16_t b;
//16-bit value
   char *c;    //string with length not declared
   int32_t total_size=0,str_len; //total size of the buffer after packing in bytes

   va_start(ap, form);

   for(; *form!='\0';form++)
```

```
   {
      switch(*form)
      {
         case 'a':total_size+=1;
a=(int8_t)va_arg(ap, int);
*buffer++=(a>>0)&0xff;
break;           case 'b': total_size+= 2;
b=(int16_t)va_arg(ap,int);
packin16(buffer,b);
buffer=buffer+2;
                  break;

         case 'c': c=va_arg(ap, char*);
str_len=strlen(c);
total_size+=str_len+2;
packin16(buffer,str_len);                        buffer+=2;
                     memcpy(buffer,c,str_len);
                     buffer+=str_len;
                     break;
      }
   }
va_end(ap); return
total_size;
}

//unpacking()- unpacking the data specified by form variable from the buffer void
unpacking(char *buffer, char *form, ...)
{    va_list
ap;    int8_t
*a;    int16_t
*b;    char
*c;
   int32_t buf_len, counting, strlength=0;
va_start(ap, form);
   for(; *form!='\0'; form++)
   {
      switch(*form)
      {
case 'a':
               a=va_arg(ap, int8_t*);
*a=*buffer++;                  break;
case 'b':
               b=va_arg(ap, int16_t*);
*b=unpackin16(buffer);
```

```
                buffer+=2;
break;            case 'c':
                c=va_arg(ap, char*);
buf_len=unpackin16(buffer);
                buffer=buffer+2;
                if(strlength>0 && buf_len>strlength)
                counting=strlength-1;
else counting=buf_len;
memcpy(c,buffer,counting);
                c[counting]='\0';
buffer+=buf_len;
                break;
default:
if(isdigit(*form))
                {
                   strlength=strlength*10+(*form-'0');
                }
       }
       if(!isdigit(*form)) strlength=0;
   }
       va_end(ap);
}

void *getaddress (struct sockaddr *sa)
{
   if (sa -> sa_family == AF_INET)
     {
      return & (((struct sockaddr_in*)sa) -> sin_addr);
     }
   return & (((struct sockaddr_in6*)sa) -> sin6_addr);
}

int main (int argc, char* argv[])
{     int sock_fd, total_bytes;     int16_t
packet_size;     int rcv,length,i;     struct
addrinfo hints, *serverinfo, *p;     struct
message message_send;     struct
attribute attribute_send;     struct
message;     struct attribute;     char
buffer[1000];     char
username_buffer[16];     char
message_buffer[512];
   fd_set read_fds;  //temp file descriptor list for select()
```

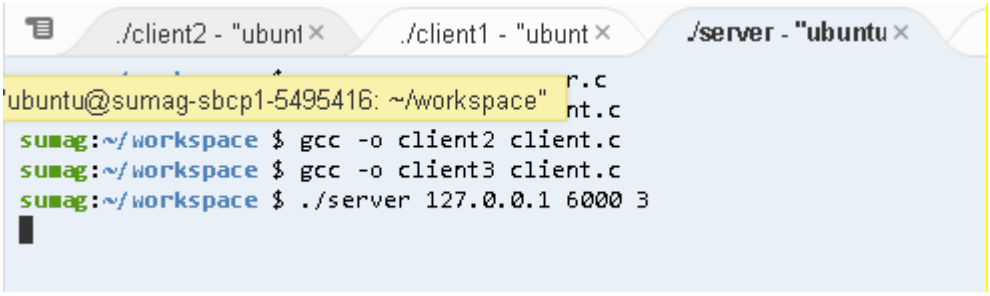
```
   struct timeval tv;
int retval;
   tv.tv_sec=60;  //wait upto 60 seconds
tv.tv_usec=0;

   FD_ZERO(&read_fds);

   if(argc!=4)
   {
       fprintf (stderr, "correct implementation: ./client username server_ip server_port\n");
exit(1);
   }

   memset(&hints, 0, sizeof hints);
hints.ai_family= AF_UNSPEC;
   hints.ai_socktype= SOCK_STREAM;

   if ((rcv= getaddrinfo(argv[2], argv[3], &hints, &serverinfo))!=0)
     {
       fprintf(stderr, "%s\n", gai_strerror(rcv));
return 1;
     }

   for (p=serverinfo; p!=NULL;p=p->ai_next)
    {
       if((sock_fd=socket(p->ai_family, p->ai_socktype, p->ai_protocol))==-1) //get file
descriptors for all sockets in the list
        {
           perror("client socket");
           continue;
        }

       if(connect(sock_fd, p->ai_addr, p->ai_addrlen)==-1)
       {
          close(sock_fd);
perror("client connect");
          continue;
       }
break;
    }
   if(p==NULL)
   {
       fprintf(stderr, "client failed to connect\n");
return 2;
```

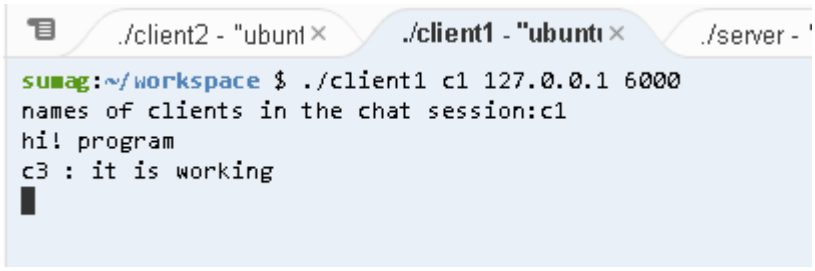
```
    }

    freeaddrinfo(serverinfo); //finished working with serverinfo structure

    strcpy(username_buffer, argv[1]);
attribute_send.atr_msg=username_buffer;
    attribute_send.type=2;
attribute_send.length=20;
message_send.version='3';
message_send.type='2';
message_send.length=24;
    message_send.payload=&attribute_send;

    //packing the message into sbcp message format     packet_size=packing(buffer,
"aabbbc", message_send.version, message_send.type, message_send.length,
attribute_send.type, attribute_send.length, username_buffer);
if(send(sock_fd,buffer,packet_size,0)==-1)
    {
        perror("send");
exit(1);
    }

    attribute_send.atr_msg=message_buffer;
    attribute_send.type=4;
attribute_send.length=516;
message_send.version='3';
message_send.type='4';
message_send.length=520;
    message_send.payload=&attribute_send;

    FD_SET(0,&read_fds); //adding input from the keyboard to read_fds
    FD_SET(sock_fd, &read_fds); //adding sock_fd to read_fd set

    while(1)
    {
        retval=select(sock_fd+1, &read_fds, NULL, NULL,&tv);
if (retval==-1)
        {
            perror("select");
            exit(4);
        }
        else if(retval)
        {
```

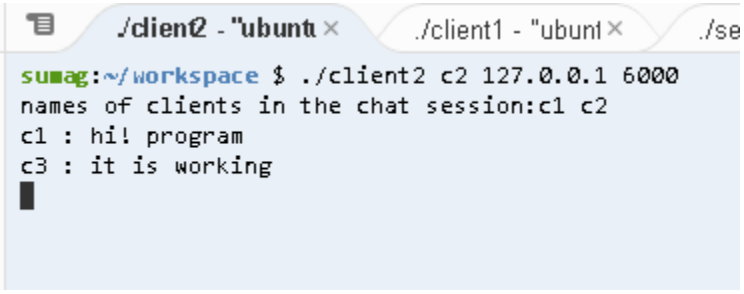
```
      //to check if there is any data to be read
for(i=0; i<=sock_fd;i++)
      {
         if(FD_ISSET(i,&read_fds))
         {
            //data from keyboard
            if(i==0)
            {
             fgets(message_buffer, sizeof(message_buffer), stdin);
length=strlen(message_buffer)-1;
if(message_buffer[length]=='\n')                message_buffer[length]='\0';
              packet_size=packing(buffer, "aabbbc", message_send.version, message_send.type,
message_send. length,attribute_send.type, attribute_send.length, message_buffer);
if(send(sock_fd, buffer,packet_size,0)==-1)
             {
                perror("send");
                exit(1);
             }
            }
            //data to be read from server
if(i==sock_fd)
            {
               if((total_bytes=recv(sock_fd, buffer, 999,0))<=0)
               {
exit(1);
               }
               buffer[total_bytes]='\0';
               printf("%s\n",buffer);
            }
         }
         FD_SET(0,&read_fds);
         FD_SET(sock_fd, &read_fds);
      }
}
else
   {
      printf("Client is idle for 10 seconds hence closing the connection\n");
return 0;

   }
   }
   close(sock_fd);
return 0;
}
```

TEST CASES

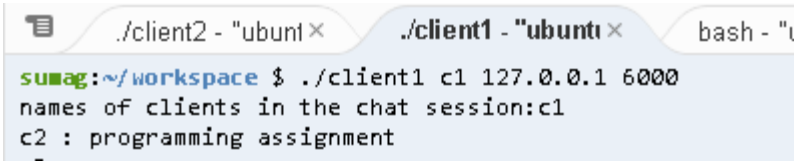
1. Normal operation of the chat server with 3 clients connected:

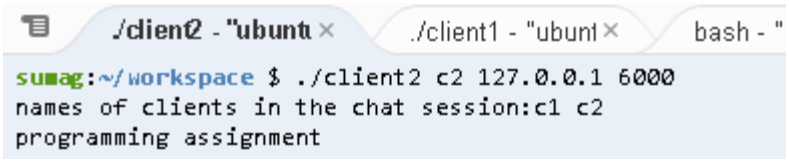


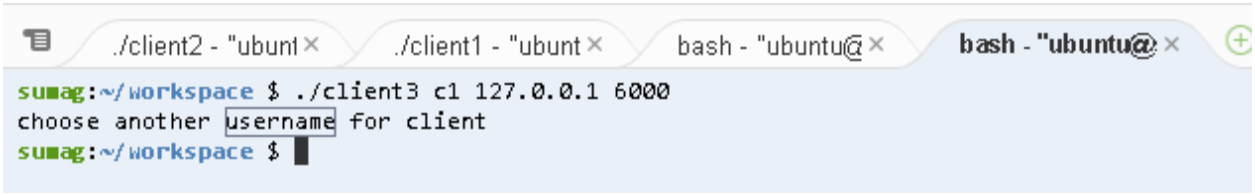




2. server rejects a client with duplicate username

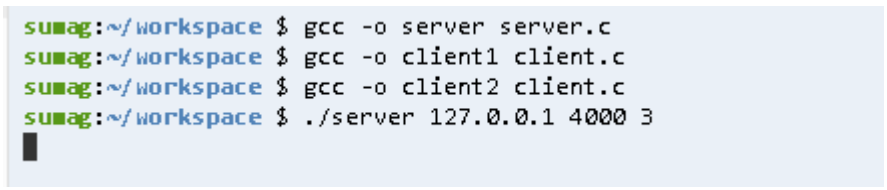


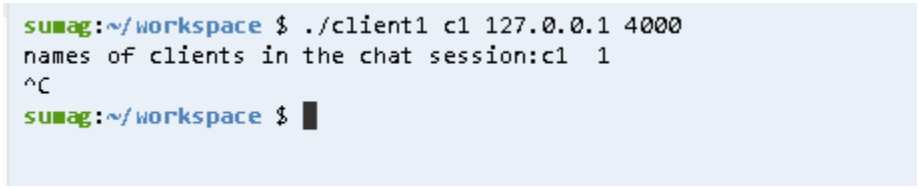




When client 3 tries to give username c1 which has already been used by the 1st client, it throws a message, choose another username for the client and also terminates the client connection.

3. Server allows a previously used username to be used



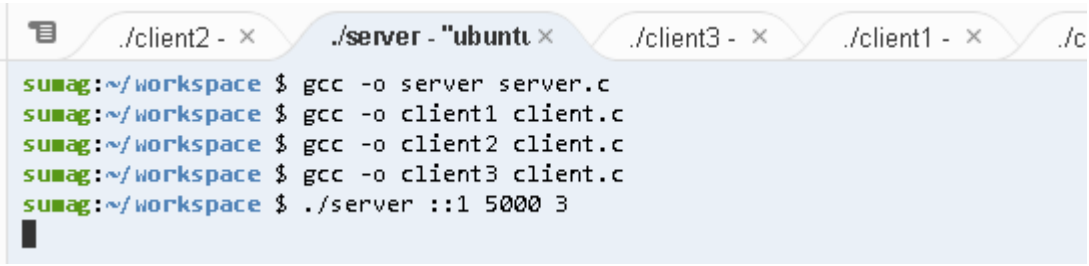


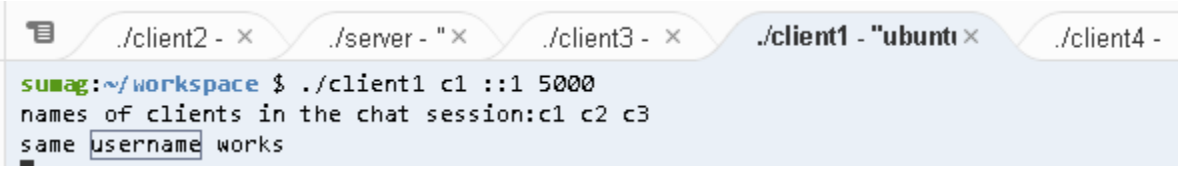
```
sumag:~/workspace $ ./client2 c2 127.0.0.1 4000
names of clients in the chat session:c1 c2 2
 c1 OFFLINE : client connection has closed
Client is idle for 10 seconds hence closing the connection
sumag:~/workspace $
```

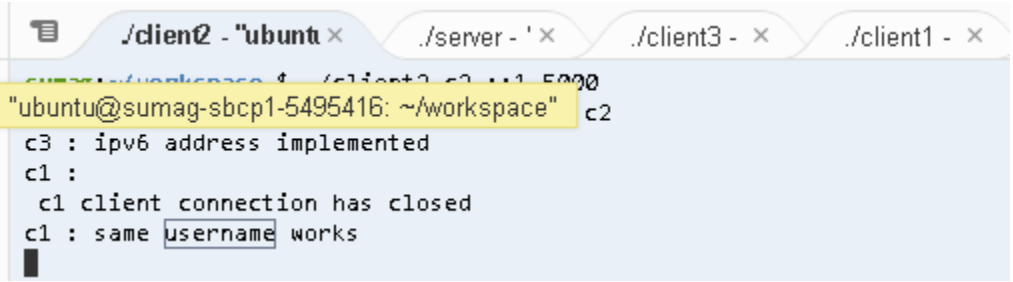
Client3 uses same username as client1 c1, which throws an error message: chose another username for client and once c1 goes offline/exits the chat session, we can see that the username can be reused.

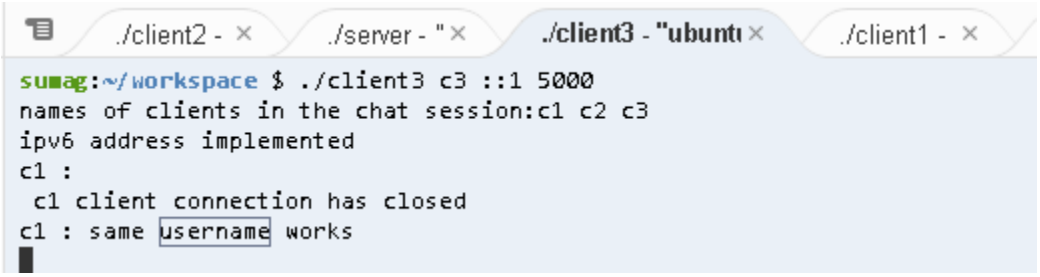
```
sumag:~/workspace $ gcc -o client3 client.c
sumag:~/workspace $ ./client3 c1 127.0.0.1 4000
choose another username for client
sumag:~/workspace $ ./client3 c1 127.0.0.1 4000
names of clients in the chat session:c1 c2  2
 c2 OFFLINE : client connection has closed
Client is idle for 10 seconds hence closing the connection
sumag:~/workspace $
```

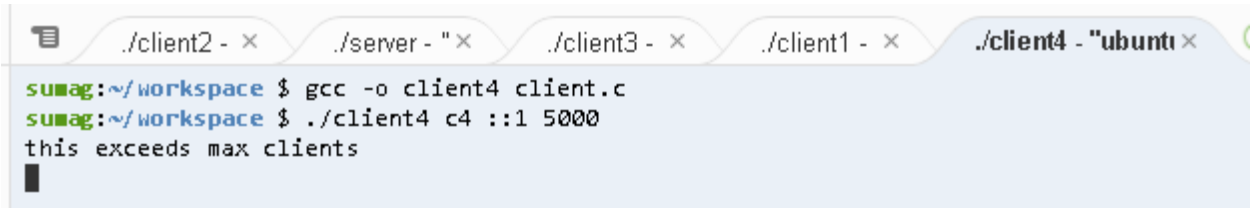
4. Server rejects the client because it exceeds the maximum number of clients allowed







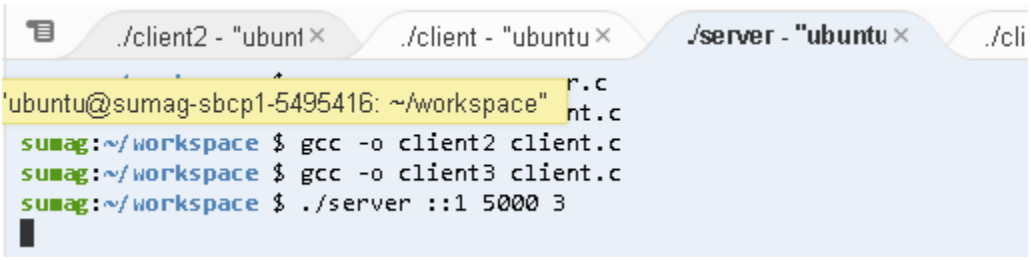


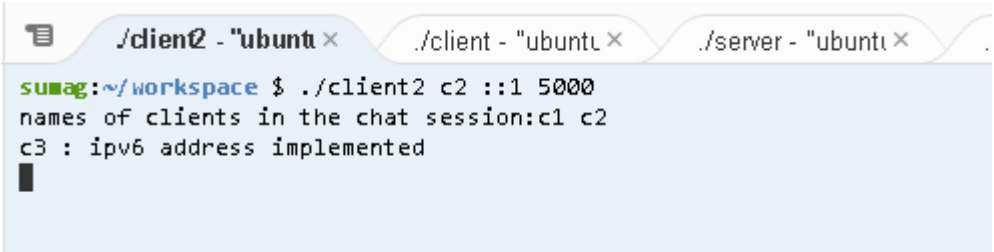


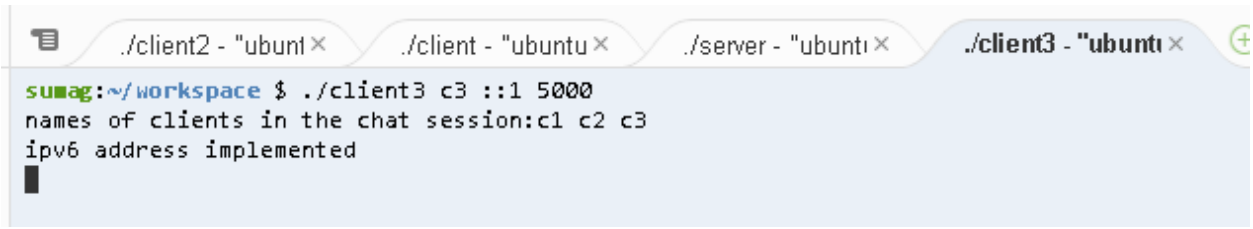
We can see that after it exceeds the maximum number of clients, the message is not sent to all other clients.

5. Bonus feature 1: ipv6 implementation

::1 is the loopback address in ipv6.

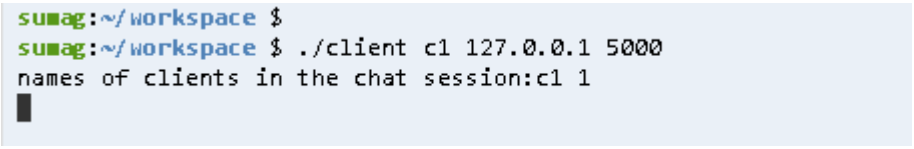


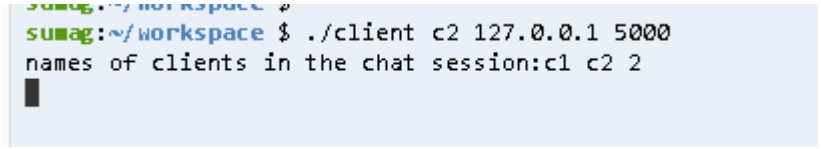




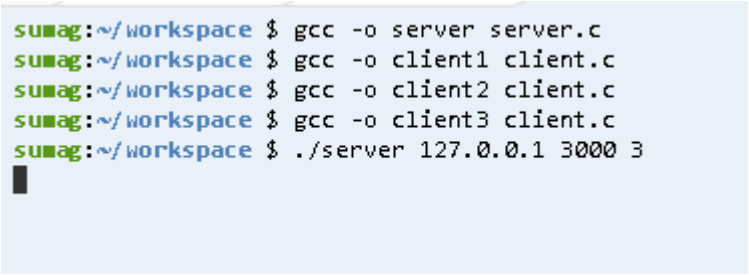
6. Bonus feature 2

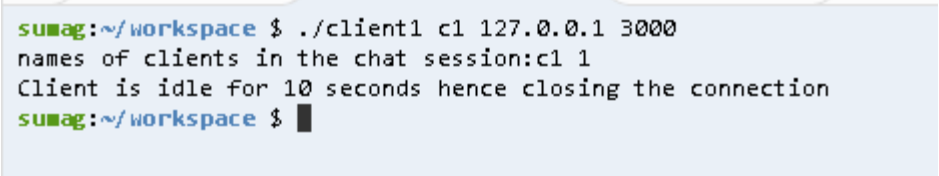
1. ACK message: client list and client count





2. OFFLINE message which is displayed when the client exits the chat session





```
sumag:~/workspace $ ./client2 c2 127.0.0.1 3000
names of clients in the chat session:c1 c2 2
 c1 OFFLINE : client connection has closed
Client is idle for 10 seconds hence closing the connection
sumag:~/workspace $
```

```
sumag:~/workspace $ ./client3 c3 127.0.0.1 3000
names of clients in the chat session:c1 c2 c3 3
 c1 OFFLINE : client connection has closed
 c2 OFFLINE : client connection has closed
Client is idle for 10 seconds hence closing the connection
```

7. Bonus feature 3

Note: We have increased the timeout value to 60 seconds in the code submitted to facilitate creating and joining of the other clients before the first one expires due to timeout.

Connection to the server closed when the client is idle for 10 seconds.

```
sumag:~/workspace $ ./client1 a1 127.0.0.1 2000
names of clients in the chat session:a1 1
a3 : hi
hi
a2 : hi
Client is idle for 10 seconds hence closing the connection
```

Other clients getting a connection close message after the client is idle for 10 seconds.

```
sumag:~/workspace $ ./client2 a2 127.0.0.1 2000
names of clients in the chat session:a1 a2 2
a3 : hi
a1 : hi
hi
 a1 client connection has closed
```

```
sumag:~/workspace $ ./client3 a3 127.0.0.1 2000
names of clients in the chat session:a1 a2 a3 3
hi
a1 : hi
a2 : hi
 a1 client connection has closed
```

Acknowledgements

1. Beej's Guide
2. www.tutorialspoint.com

Multiple references from GeeksforGeeks, Github.