OPERATING SYSTEM PROJECT

SOCKET PROGRAMMING IN C

Instructor

Ms. Anaum Hamid

Course

CS-2006

Section

BCS-4J

Group Members

Qasim Hasan (21k-3210)
Ahsan Ashraf (21k-3186)
Kantesh kumar (21k-3426)

Features Implemented in Project:

MAIN FEATURE:

1. Group Chat Application (Multithreaded/Multiple Clients)

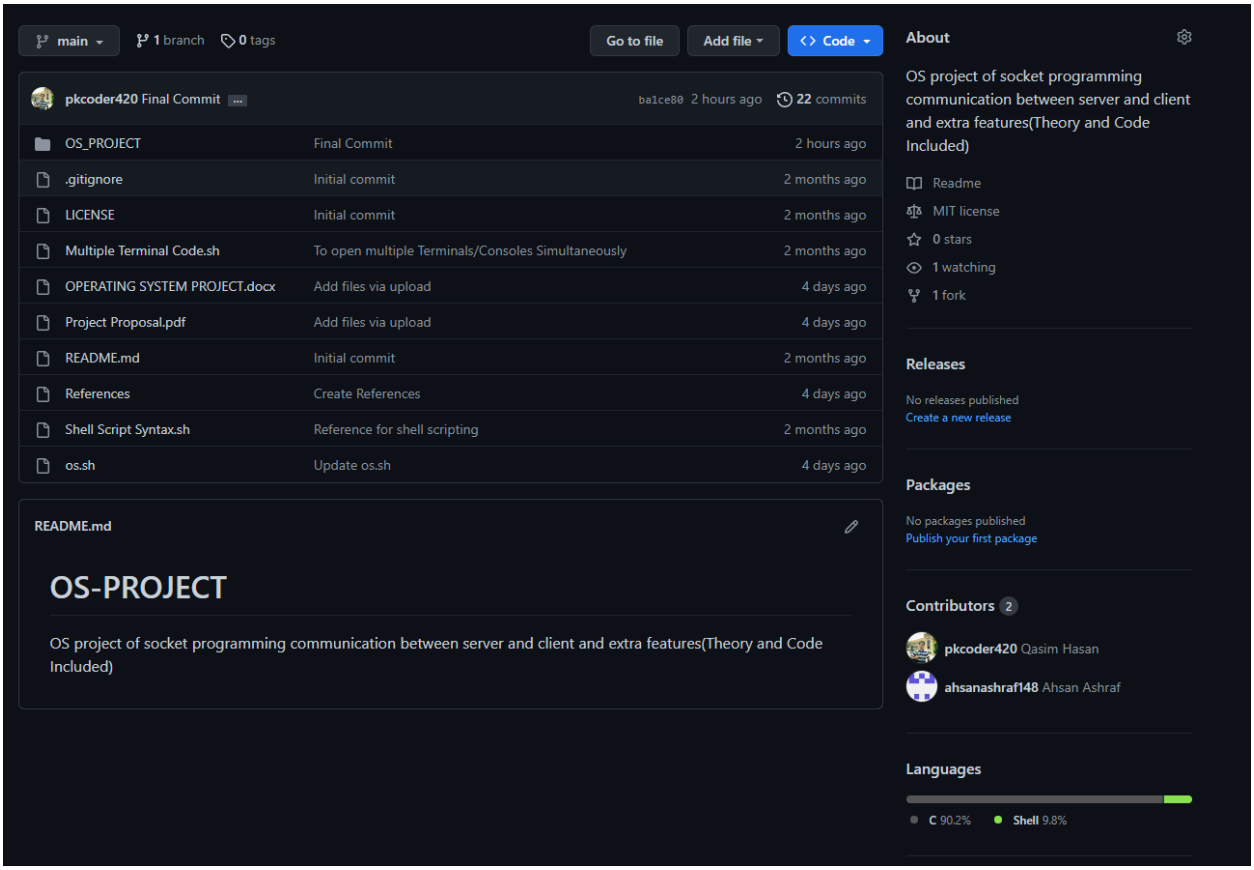
SUB FEATURES:

1. Single client server Interaction
2. Client/Server File Transfer
3. Client/Server Calculator

Tools and Technology, Languages and Utilities:

GitHub Link and QR-Code:

https://github.com/pkcoder420



Objective:

We gave 4 programs each program consisting of two c files one has a server and the other has a client. We are using socket function defined in c using multiple libraries which are explained below.

Our goal was to enable communication between client and user, test various scenarios between them, and enable numerous clients to connect to a server and talk across a socket while it is actively taking requests.

Here we are simulating two pc as two terminals.

Libraries:

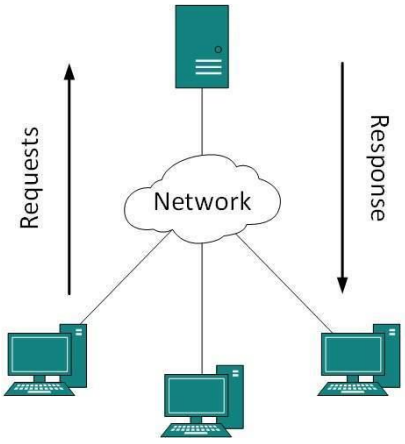
Following are the libraries we used in our C programs.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<arpa/inet.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<pthread.h>
```

stdio.h	This header file contains declarations for input and output in all c programs.
stdlib.h	Defines Variable types, several macros and functions. Ex: int atoi(const char*)
unistd.h	Provide access to Posix API.
string.h	All functionality regarding strings. Len() , concatenate() e.tc
Arpa/inet.h	Contains definitions for internet operations.(For windows winsock)

sys/socket.h	Definitions for structures needed for sockets. Ex: sockaddr
netinet.h	Constants and structures needed for internet domain address ex sockaddr_in
pthread.h	Give us access to make multiple threads so we can give access to multiple users/clients

State Diagram of Server and Client Model:



1. Socket Creation:

int sockfd = socket (domain, type, protocol)

sockfd: socket descriptor, an integer (like a file-handle)

domain: integer, specifies communication domain. We use AF_ LOCAL as defined in the POSIX standard for communication between processes on the same host. For communicating between processes on different hosts connected by IPV4, we use AF_INET and AF_I NET 6 for processes connected by IPV6.

type: communication type

SOCK_STREAM: TCP (reliable, connection oriented)

SOCK_DGRAM: UDP (unreliable, connectionless)

protocol: Protocol value for Internet Protocol(IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet. (man protocols for more details)

2. Bind:

int bind (int sockfd, const struct sockaddr *addr, socklen_t addrlen);

After the creation of the socket, the bind function binds the socket to the address and port number specified in addr (custom data structure). In the example code, we bind the server to the localhost, hence we use INADDR_ANY to specify the IP address.

3. Listen:

int listen (int sockfd, int backlog);

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog, defines the maximum length to which the queue of pending connections for sockfd may grow.

4. Connection:

int connect (int sockfd, const struct sockaddr *addr, socklen_t addrlen);

Socket connection: Exactly same as that of server's socket creation

Connect: The connect () system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server's address and port is specified in addr.

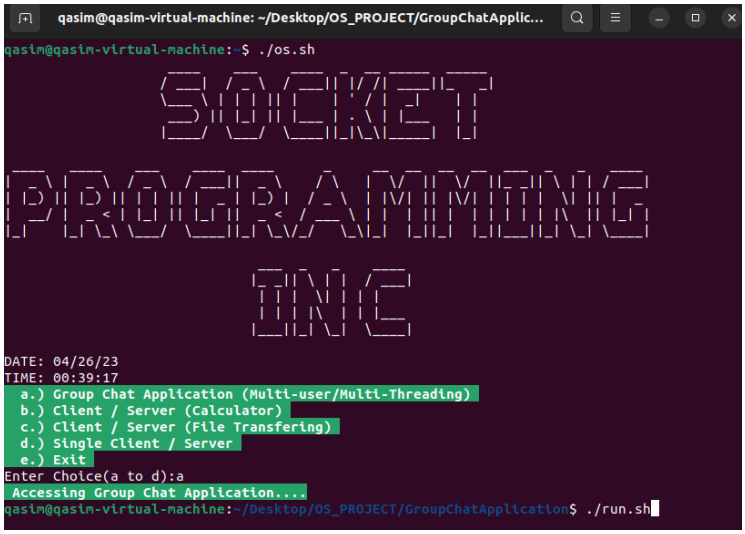
5. Accept:

int new_socket= accept (int sockfd, struct sockaddr *addr, socklen_t *addrlen);

It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, the connection is established between client and server, and they are ready to transfer data.

MANUAL FOR COMPILING AND RUNNING PROGRAMS:

1. RUN THE PROJECT:



2. Compile and Run Multi-Chat Program:

```
konsole --noclose -e ./server 3210 &
sleep 7
konsole --noclose -e ./client 192.168.196.131 3210 qasim &
sleep 9
konsole --noclose -e ./client 192.168.196.131 3210 suffiyan &
exit
```

```
// RUN CLIENT.C TUTORIAL:
//=============================================================================================
// To compile this program:
//      gcc groupchat_linux_client.c -D_REENTRANT -o client -lpthread
//                                    ------------           ---------
//                                         |                     |
//                 drives the compiler to use            adds support for multithreading
//                 thread safe (i.e., re-entrant)        with the pthreads library
//                 versions of several functions
//                 in the C library
//
// To run this program:
//      ./client <serverIP> <port> <name>
//=============================================================================================
```

```
//RUN SERVER.C TUTORIAL:
//=============================================================================================
// To compile this program:
//      gcc groupchat_linux_server.c -D_REENTRANT -o server -lpthread
//                                    ------------           ---------
//                                          |                     |
//                drives the compiler to use            adds support for multithreading
//                thread safe (i.e., re-entrant)        with the pthreads library
//                versions of several functions
//                in the C library
//
// To run this program:
//      ./server <port>
//=============================================================================================
```

3. Compile and Run Single Client/Server Program:

```
konsole --noclose -e ./server 3210 &
sleep 2
konsole --noclose -e ./client 127.0.0.1 3210 &
exit
```

```
//RUN CLIENT.C TUTORIAL:
//=============================================================================================
// To compile this program:
//      gcc client.c -o client
//
// To run this program:
//      ./client <ip> <port>
//=============================================================================================

//RUN SERVER.C TUTORIAL:
//=============================================================================================
// To compile this program:
//      gcc server.c -o server
//
// To run this program:
//      ./client <port>
//=============================================================================================
```

EXPLANATION OF CODE (C language):

MULTI CHAT SERVER AND CLIENT (MAIN CODE)

Server:

```
#define MSG_LEN     2048
#define MAX_CLIENT  512

// function declarations
void* serve_cl(void *arg);
void broadcast_msg(char *msg, int len);
void handle_err(char *err_msg);

// global variables
int cl_cnt = 0;             // counter for connected clients
int cl_socks[MAX_CLIENT];   // array (list) of the connected clients
pthread_mutex_t mutex;
```

The code above defines the global variables **cl_cnt, cl_socks**, and **mutex**, as well as three function prototypes named **serve_cl, broadcast_msg,** and **handle_err.**

- The **cl_cnt** variable is an integer that is used to keep track of the number of connected clients.
- The **cl_socks** variable is an array of integers with a maximum size of **MAX_CLIENT** (which is defined to be 512) that is used to store the socket descriptors of all connected clients.
- The **mutex** variable is a pthread **mutex** that is used to synchronize access to the shared resources of the server, such as the **cl_cnt** and **cl_socks** variables.

```c
int main(int argc, char *argv[]){
    int sv_sock, cl_sock;                   // server/client socket file descriptors
    struct sockaddr_in sv_addr, cl_addr;    // Internet socket address structures
    int cl_addr_sz;                         // size of the client sockaddr_in structure
    pthread_t t_id;                         // thread ID

    // handle invalid number of arguments
    if (argc != 2)
    {
        printf("Usage: %s <port>\n", argv[0]);
        exit(1);
    }

    pthread_mutex_init(&mutex, NULL);

    // create a TCP socket
    sv_sock = socket(PF_INET, SOCK_STREAM, 0);

    // allocate and initialize Internet socket address structure (server info)
    // specify server's IP address and port
    memset(&sv_addr, 0, sizeof(sv_addr));
    sv_addr.sin_family = AF_INET;
    sv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    sv_addr.sin_port = htons(atoi(argv[1]));

    // bind server's address information to the server socket
    if (bind(sv_sock, (struct sockaddr*)&sv_addr, sizeof(sv_addr)) == -1)
        handle_err("ERROR: bind() fail");

    // convert the server socket to listening socket
    if (listen(sv_sock, 5) == -1)
        handle_err("ERROR: listen() fail");
    // clear the screen
    system("clear");
    // print the welcome message to the screen
    printf("Group chat linux server running ... (port: %s)\n", argv[1]);
    while (1)
    {
        // get the size of Internet socket address structure
        cl_addr_sz = sizeof(cl_addr);

        // accept client connection
        cl_sock = accept(sv_sock, (struct sockaddr*)&cl_addr, &cl_addr_sz);

        // add the new client to the list
        pthread_mutex_lock(&mutex);     // enter the critical section
        cl_socks[cl_cnt++] = cl_sock;
        pthread_mutex_unlock(&mutex);   // leave the critical section

        // create a thread to serve the client
        pthread_create(&t_id, NULL, serve_cl, (void*)&cl_sock);

        // detach the thread from the main thread so that it automatically destroys itself
        // upton termination
        pthread_detach(t_id);

        // print the connected client's IP address to the screen
        printf("Connected client IP: %s\n", inet_ntoa(cl_addr.sin_addr));
    }
    close(sv_sock);
    return 0;
} // end of main
```

This is the main function of a Linux server for a group chat application. It sets up the server socket and accepts incoming client connections, adding each connected client to a list of connected clients. The server creates a new thread to serve each connected client and detaches the thread from the main thread so that it can automatically destroy itself upon termination.

The main function first declares the necessary variables, including socket file descriptors, Internet socket address structures, and a thread ID. It then checks the number of command-line arguments and prints an error message if the number is not equal to two.

- The function initializes a mutex for synchronization purposes using pthread_mutex_init(). A TCP socket is then created using socket(). The function then initializes the server's Internet socket address structure by allocating memory and setting its attributes, including the server's IP address and port number. The server's address information is then bound to the server socket using bind(). If this operation fails, the handle_err() function is called to handle the error.
- The function then converts the server socket to a listening socket using listen(). If this operation fails, the handle_err() function is called to handle the error.

```
// receive client's message, and broadcast it to all connected clients
void* serve_cl(void *arg)
{
    int cl_sock = *((int*)arg);
    int str_len = 0, i;
    char msg[MSG_LEN];

    // this while loop won't break until the client terminates the connection
    while ((str_len = read(cl_sock, msg, sizeof(msg))) != 0)
        broadcast_msg(msg, str_len);

    pthread_mutex_lock(&mutex);     // enter the critical section

    // before terminating the thread, remove itself (disconnected client) from the list
    for (i = 0; i < cl_cnt; i++)
    {
        if (cl_sock == cl_socks[i])
        {
            while (i < cl_cnt - 1)
            {
                cl_socks[i] = cl_socks[i + 1];
                i++;
            }

            break;
        }
    }

    cl_cnt--;                       // decrement the client counter
    pthread_mutex_unlock(&mutex);   // leave the critical section

    // terminate the connection
    close(cl_sock);

    return NULL;
} // end of serve_cl
```

The serve_cl function is the function that is executed in a separate thread to handle communication with a single client. Here is a breakdown of the code:
The function takes in a void pointer argument arg, which is cast to an integer and dereferenced to obtain the socket file descriptor of the client.

- The function then declares some variables, including an integer variable str_len to store the length of the message received from the client, an integer variable i for looping through the array of connected clients, and a character array msg to store the message received from the client.
- The function enters a while loop, which reads data from the client using the read function. If the length of the message received is zero, that means the client has terminated the connection, so the loop breaks.
- The message received from the client is then broadcast to all connected clients using the broadcast_msg function.
- After the while loop, the function enters a critical section by acquiring the mutex lock using pthread_mutex_lock. The function then searches for the disconnected client's socket file descriptor in the array of connected clients and removes it from the list. This is done to keep track of the number of connected clients accurately.
- The function then decrements the cl_cnt variable, which keeps track of the number of connected clients.The mutex lock is then released using pthread_mutex_unlock.Finally, the function terminates the connection with the client using the close function and returns NULL.

```
// broadcast message to all connected clients
void broadcast_msg(char *msg, int len)
{
    int i;

    pthread_mutex_lock(&mutex);     // enter the critical section

    // send message to every connected clients
    for (i = 0; i < cl_cnt; i++)
        write(cl_socks[i], msg, len);

    pthread_mutex_unlock(&mutex);   // leave the critical section
} // end of broadcast_msg
```

The function broadcast_msg is responsible for broadcasting a message to all connected clients. The function takes two arguments: msg, a character pointer that holds the message to be sent, and len, an integer representing the length of the message.

- The function first enters a critical section by acquiring the mutex lock, to ensure mutual exclusion and prevent multiple threads from accessing the shared resources (client sockets) at the same time. Then, it loops through all the connected clients' sockets in the cl_socks array and sends the message to each of them using the write function. After sending the

message to all connected clients, the function leaves the critical section by releasing the mutex lock.

- The function assumes that all clients' sockets are valid and connected to the server. If there are any connection issues, the error needs to be handled appropriately in the serve_cl function, which is responsible for removing the disconnected client's socket from the cl_socks array.

Client:

```
#define MSG_LEN     1024
#define NAME_LEN    32

// function declarations
void* send_msg(void *arg);
void* recv_msg(void *arg);
void handle_err(char *err_msg);

// global variables
char name[NAME_LEN] = "noname";
char msg[MSG_LEN];
char info_msg[MSG_LEN];
```

The code above is a set of preprocessor directives and function declarations, as well as global variable declarations.

- #define MSG_LEN 1024 defines a macro constant MSG_LEN with a value of 1024.
- #define NAME_LEN 32 defines a macro constant NAME_LEN with a value of 32.
- void* send_msg(void *arg); is a function declaration for a function called send_msg that takes a void pointer argument and returns a void pointer.
- void* recv_msg(void *arg); is a function declaration for a function called recv_msg that takes a void pointer argument and returns a void pointer.
- void handle_err(char *err_msg); is a function declaration for a function called handle_err that takes a character pointer argument and returns nothing.
- char name[NAME_LEN] = "noname"; is a global variable declaration that creates an array of characters called name with a length of NAME_LEN (32) and initializes it with the string "noname".
- char msg[MSG_LEN]; is a global variable declaration that creates an array of characters called msg with a length of MSG_LEN (1024).
- char info_msg[MSG_LEN]; is a global variable declaration that creates an array of characters called info_msg with a length of MSG_LEN (1024).

```c
int main(int argc, char *argv[])
{
    int sock;                           // client socket file descriptor
    struct sockaddr_in sv_addr;         // Internet socket address structure
    pthread_t send_thread, recv_thread; // two threads of the client process
    void *thread_return;                // pointer to the thread's return value

    // handle invalid number of arguments
    if (argc != 4)
    {
        printf("Usage: %s <serverIP> <port> <name>\n", argv[0]);
        exit(1);
    }

    sprintf(name, "%s", argv[3]);  // setup user name

    // create a TCP socket
    sock = socket(PF_INET, SOCK_STREAM, 0);

    // allocate and initialize the Internet socket address structure (server info)
    // specify server's IP address and port
    memset(&sv_addr, 0, sizeof(sv_addr));
    sv_addr.sin_family = AF_INET;
    sv_addr.sin_addr.s_addr = inet_addr(argv[1]);
    sv_addr.sin_port = htons(atoi(argv[2]));

    // establish connection with server
    if (connect(sock, (struct sockaddr*)&sv_addr, sizeof(sv_addr)) == -1)
        handle_err("ERROR: connect() fail");

    // clear the screen
    system("clear");

    // print the welcome message to the screen
    printf("Welcome to group chat! (Q/q to quit)\n\n");

    // print the join message to the screen
    sprintf(msg, "+++++ %s has joined! +++++\n", name);
    write(sock, msg, strlen(msg));

    // create two threads to carry out send and receive operations, respectively
    pthread_create(&send_thread, NULL, send_msg, (void*)&sock);
    pthread_create(&recv_thread, NULL, recv_msg, (void*)&sock);
    pthread_join(send_thread, &thread_return);
    pthread_join(recv_thread, &thread_return);

    // terminate connection
    close(sock);

    return 0;
} // end of main
```

This is the main function of a simple chat client that connects to a server using TCP sockets and carries out communication through two threads: one for sending messages and another for receiving messages.

- The main function takes three command-line arguments: the IP address of the server, the port number of the server, and the name of the client user. If the number of arguments is not three, it prints the usage message and exits with an error code.
- Then, the client creates a TCP socket using the socket system call, and initializes the Internet socket address structure with the IP address and port number of the server. The client then establishes a connection with the server using the connect system call.
- After connecting to the server, the client clears the screen and prints a welcome message to the user. It also sends a message to the server indicating that the user has joined the chat room.
- The client then creates two threads using the pthread_create system call, one for sending messages and another for receiving messages. The client also waits for the threads to finish using the pthread_join system call.
- Finally, the client terminates the connection with the server using the close system call and exits with a success code.

```
// send message to the server
void* send_msg(void *arg)
{
    int sock = *((int*)arg);
    char name_msg[NAME_LEN + MSG_LEN + 2];  // add '2' for brackets around the name

    while (1)
    {
        fgets(msg, MSG_LEN, stdin);

        // terminate the connection upon user input Q/q
        if (!strcmp(msg, "q\n") || !strcmp(msg, "Q\n"))
        {
            // print the leave message to the screen
            sprintf(msg, "----- %s has left! -----\n", name);
            write(sock, msg, strlen(msg));

            // terminate the connection
            close(sock);
            exit(0);
        }

        // construct the message and send it to the server
        sprintf(name_msg, "[%s] %s", name, msg);
        write(sock, name_msg, strlen(name_msg));
    }

    return NULL;
} // end of send_msg
```

- This function send_msg is responsible for sending messages to the server. It takes a void pointer arg as its argument, which is a pointer to the integer file descriptor of the client

socket. Inside the function, it first initializes a character array name_msg of size NAME_LEN + MSG_LEN + 2, which will be used to store the message that includes the user's name.

- It then enters an infinite loop where it waits for the user to input a message using fgets and stores it in the msg character array of size MSG_LEN. If the user inputs q or Q, it sends a leave message to the server with the user's name and terminates the connection by closing the socket and exiting the program.
- If the user inputs any other message, it constructs the final message by appending the user's name to the input message inside the name_msg character array, enclosed in square brackets. Finally, it sends the message to the server using the write function. The function returns NULL at the end.

```
// receive message and print it to the screen
void* recv_msg(void *arg)
{
    int sock = *((int*)arg);
    char name_msg[NAME_LEN + MSG_LEN + 2];
    int str_len;

    while(1)
    {
        // receive the message broadcasted by the server
        str_len = read(sock, name_msg, NAME_LEN + MSG_LEN + 2 - 1);
            // add '2' for brackets around the name

        // terminate the thread if the client has been disconnected
        if (str_len == -1)
            return (void*)-1;

        // print the received message to the screen
        name_msg[str_len] = 0;
        fputs(name_msg, stdout);
    }

    return NULL;
} // end of recv_msg
```

This code defines a function recv_msg that is intended to be run as a separate thread in a client-server chat application. The function receives messages from the server and prints them to the screen.

- The function starts by extracting the client socket file descriptor from the argument passed to it. Then, in a continuous loop, the function reads messages from the server using the read() function. The read() function reads up to NAME_LEN + MSG_LEN + 2 - 1 bytes of data from the socket (sock) and stores them in the buffer name_msg. The NAME_LEN and MSG_LEN constants define the maximum length of the username and message, respectively, and the 2 is added to account for the square brackets that are added to the message later.

- If the read() function returns -1, it means that the client has been disconnected from the server, and the function returns -1 as well.Otherwise, the received message is printed to the screen using the fputs() function.
- The name_msg buffer is null-terminated by setting the character at str_len to 0 before it is printed. Finally, the function returns NULL.

Calculator: (Client and Server)

```
int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        printf("usage %s hostname port\n", argv[0]);
        exit(1);
    }

    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    char buffer[256];
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    if (sockfd < 0)
    {
        error("ERROR opening socket");
    }

    server = gethostbyname(argv[1]);

    if (server == NULL)
    {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr, server->h_length);
    serv_addr.sin_port = htons(portno);

    if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
    {
        error("ERROR connecting");
    }

    int num1,num2,choice,ans;
    S: bzero(buffer,256);
    n = read(sockfd,buffer,255);
```

```
if (n < 0)
{
    error("Error reading from socket");
}

printf("Server: %s\n",buffer);
scanf("%d",&num1);
write(sockfd,&num1,sizeof(int));

bzero(buffer,256);
n= read(sockfd,buffer,255);
if(n<0)
{
    error("Error reading from socket");
}
printf("Server: %s\n",buffer);
scanf("%d",&num2);
write(sockfd,&num2,sizeof(int));

bzero(buffer,256);
n= read(sockfd,buffer,255);

if(n<0)
{
    error("Error reading from socket");
}
printf("Server: %s\n",buffer);
scanf("%d",&choice);
write(sockfd,&choice,sizeof(int));

if(choice==5)
{
    goto Q;
}
read(sockfd,&ans,sizeof(int));
printf("Server: The answer is %d\n",ans);

if(choice != 5)
{
    goto S;
}
```

This code is a C program that creates a client application using socket programming. It connects to a server using the provided hostname and port number as arguments. The program prompts the user to enter two integers and an arithmetic operation choice. It sends these inputs to the server using write () function and waits for the response from the server. The server processes the input and sends back the result to the client, which then displays it to the user. The program uses a "goto" statement to allow the user to perform multiple calculations before choosing to exit the application.

The program includes error handling to ensure that the socket is opened successfully and that the connection to the server is established without any errors. It also checks for errors when reading from and writing to the socket. The program uses the standard library functions and socket API to create and manage the socket connection.

```c
int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Port No not provided. Program terminated.\n");
        exit(1);
    }

    int sockfd, newsockfd, portno, n;
    char buffer[255];

    struct sockaddr_in serv_addr, cli_addr;
    socklen_t clilen;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        error("Error opening socket");
    }

    bzero((char *) &serv_addr, sizeof(serv_addr));
    portno = atoi(argv[1]);

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);

    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
        error("Binding failed");
    }

    listen(sockfd, 5);
    clilen = sizeof(cli_addr);

    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd < 0) {
        error("Error on accept");
    }

    int num1, num2, ans, choice;

S:
    n = write(newsockfd, "Enter Number 1: ", strlen("Enter Number 1: "));
 if (n < 0) {
    error("Error writing to socket");
 }
 read(newsockfd, &num1, sizeof(int));
 printf("Client - Number 1 is: %d\n", num1);

 n = write(newsockfd, "Enter Number 2: ", strlen("Enter Number 2: "));
 if (n < 0) {
    error("Error writing to socket");
 }
 read(newsockfd, &num2, sizeof(int));
 printf("Client - Number 2 is: %d\n", num2);

 n = write(newsockfd, "Enter your choice:\n1: Addition\n2: Subtraction\n3: Multiplication\n4: Division\n5: Exit\n",
  strlen("Enter your choice:\n1: Addition\n2: Subtraction\n3: Multiplication\n4: Division\n5: Exit\n"));
 if (
                public int __cdecl write (int _Filehandle, const void *_Buf, unsigned int _MaxCharCount)
 }

 read(newsockfd, &choice, sizeof(int));
 printf("Client's choice is: %d\n", choice);

 switch (choice) {
    case 1:
        ans = num1 + num2;
        break;
    case 2:
        ans = num1 - num2;
        break;
    case 3:
        ans = num1 * num2;
        break;
    case 4:
        if (num2 == 0) {
            error("Division by zero error");
        }
        ans = num1 / num2;
        break;
    case 5:
        goto Q;
    default:
        error("Invalid choice");
 }
```

```
    write(newsockfd, &ans, sizeof(int));
    if (choice != 5) {
        goto S;
    }

Q:
    close(newsockfd);
    close(sockfd);
    return 0;
}
```

This code is a server program that listens for incoming connections and provides a simple calculator service over TCP/IP.

It starts by checking if a port number is provided as a command-line argument, then creates a socket and binds it to the provided port. The server then listens for incoming connections and accepts them using the accept function.

Once a connection is established, the server prompts the client to enter two numbers and an operation to perform on them. It reads the user's input from the socket and calculates the result. The result is then written back to the client through the same socket.

The server uses a switch statement to determine which operation to perform based on the user's choice. It also includes some error handling, such as division by zero and invalid user input.

The program runs in an infinite loop until the user chooses to exit. The goto statement is used to jump back to the beginning of the loop after each calculation. Finally, the program closes the sockets and exits.

File Transfer (Client and Server):

Client:

```
FILE *f; // file pointer
int words = 0; //for loop counting depending on number of words that should be sent

char c;
f = fopen("file.txt" , "r"); //opening in read only form
while((c = getc(f))!= EOF)
{

    fscanf(f , "%s", buffer);
    if(isspace(c) || c =='\t')
    {                           // isspace checks whether character is a space because if space

        words++;                //comes that means a word is completed.
    }

}

write(sockfd, &words, sizeof(int)); //printing number of words in server
rewind(f);                           //move file pointer to the beginning

char ch;
while(ch != EOF)
{                                  //writing on server
    fscanf(f, "%s", buffer);
    write(sockfd, buffer, 255);
    ch = fgetc(f);

}

printf("\n\n\t\t\t\tThe file hass been successfully sent.\n");
```

This code reads the contents of a text file and sends them over a socket connection. It begins by defining a file pointer and a variable to count the number of words in the file. It then opens the file in read-only mode and reads through it character by character using getc(), checking if each character is a space or tab using isspace(). If it is, the word counter is incremented. The code then writes the number of words to the server using write() and moves the file pointer back to the beginning of the file using rewind(). It then reads the contents of the file again using fscanf() and writes them to the socket using write(). Finally, it prints a message to indicate that the file has been successfully sent.

```
FILE *fp;
int ch = 0;
fp = fopen("FileReceived.txt", "a"); //if file exists then data will be appended, else it will be made
int words;

read(newsockfd, &words, sizeof(int));

while(ch != words)
{

    read(newsockfd, buffer, 255);
    fprintf(fp, "%s ", buffer);
    ch++;
}

printf("\n\n\n\t\t\t\tFile has been received successfully. It is saved by the name FileReceived.txt\n");
```

This code receives a file from the client via socket programming and saves it on the server-side.

A file pointer fp is declared, and a file named FileReceived.txt is opened in append mode. The number of words in the file is read from the socket using read() and stored in the words variable. The file is then read word by word from the socket, and each word is written to the FileReceived.txt file using fprintf(). Once all the words have been written, the file pointer is closed. Finally, a message is printed to the console, informing that the file has been successfully received and saved.

Limitations:

- The program is not optimized for large file transfers and may not perform well under heavy loads or slow networks.
- The program uses a simple protocol that may not be suitable for more complex applications.
- This project only implements basic client-server communication and file transfer functionality. While it can be useful in certain scenarios, it is not suitable for more complex applications.
- The project does not have any security features such as encryption or authentication, which makes it vulnerable to attacks.

- The project is not designed to handle a large number of clients simultaneously, which limits its ability to scale to meet the demands of a growing user base.
- The code may not work on all platforms due to dependencies on certain libraries or system calls that are specific to a particular operating system.

Conclusion:

The project is a basic implementation of a client-server model using sockets in C. The program allows the client to send a file to the server, perform arithmetic operations on numbers entered by the client, and receive the results from the server. The program provides a simple and efficient way to transfer files and perform arithmetic operations over a network. It basically simulates a chat app that allows doing calculations, file transfers and message transfers.

REFERENCES:

MULTICHAT SERVER/CLIENT:
https://www.youtube.com/watch?v=KEiur5aZnIM&t=1482s
https://www.youtube.com/watch?v=fNerEo6Lstw&t=1160s

SINGLE CLIENT/SERVER:
https://www.youtube.com/watch?v=CMDBF84vSRk&list=PLPyaR5G9aNDvs6TtdpLcVO43_jvxp4emI&index=4

FILE TRANSFER:
https://www.youtube.com/watch?v=9g_nMNJhRVk&list=PLPyaR5G9aNDvs6TtdpLcVO43_jvxp4emI&index=8

CALCULATOR:
https://www.youtube.com/watch?v=cBNabLJH_cw&list=PLPyaR5G9aNDvs6TtdpLcVO43_jvxp4emI&index=7

THEORY:
https://www.youtube.com/watch?v=uHqPzNg_0OE&list=PLAZj-jE2acZKojAI_5Xwt897uzYazWC7w
https://www.youtube.com/watch?v=quH5i50lLOY&list=PLPyaR5G9aNDvs6TtdpLcVO43_jvxp4emI&index=2
https://www.youtube.com/watch?v=b_TUtu3PemQ&list=PLPyaR5G9aNDvs6TtdpLcVO43_jvxp4emI&index=3

THANK YOU for more information contact me at my GitHub given or at my Work Gmail k21321@nu.edu.pk or personal mail qasimhasanbilgrami1@gmail.com