A multithreaded TCP server implementation of QuizzGame

Victor Condurat

1 Introduction

QuizzGame is a popular game in which players are asked questions about different topics and they have to get as many correct answers as possible. The player(s) with the highest score will be the winner(s) of the game. The game has one or more rounds with multiple questions in each round. Players have a limited time for each question to provide an answer and once the answer is submitted, it can‘t be changed anymore. The correct answer is awarded 1 point and failing to provide an answer or providing the wrong one will result in 0 points.

1.1 Implementation

In my implementation of QuizzGame, each player will be required to have a registered account to play the game. There will be 3 types of games available for users. A single player game/training mode, where the user plays alone, a duo game where the user competes with another user, and a free for all game where 10 users compete against eachother. There will be 21 categories of questions for users to choose from. The matchmaking will select users who have chosen the same category and game type. Users will also have access to the history of all games that have been played and there will be a tutorial section that explains the game features to the user and how the game works.

2 Technologies used

The programming language for the application will be **C**. I will use 1 client and 4 servers in my QuizzGame implementation. All servers will be multi-threaded TCP servers. The main server will handle account related requests and the other servers will be game servers that will manage the 3 different game types: solo/practice, duo and ffa. The duo and ffa game servers will use I/O multiplexing to allow creating FIFO type lobbies for users. For storing data, I‘ll use SQLite databases. I chose TCP over UDP because TCP guarantees packet delivery and maintaining of order which is necessary for the correctness of the application. Also, the packets are compact and the game has a small number of send/receive operations between the client and the servers in a session, therefore any reduction in latency by using UDP would be unnoticeable. I chose multithreaded servers over single-threaded because the server should be able to accept multiple users connected at the same time. Due to the large number of possible clients interacting with the server simultaneously, it is more economical in terms of time for the CPU to create and context switch threads than to switch between processes. Since memory is shared between threads, I'll use mutexes in the game servers to prevent thread interference on global data.

3 Application Architecture

The application will be split between 1 client,4 servers and 2 databases.

3.1 Client

The client will display the user the various options and act as an intermediary between the user's input and the servers.

Constants:

- **port**: 2116 - Main server‘s port number
- **solo_game_port**: 2555 - Solo server‘s port number
- **duo_game_port**: 2666 - Duo server‘s port number
- **ffa_game_port**: 2777 - Ffa server‘s port number

Global variables:

- **logged_user**: The username of the logged user.
- **sd**: The socket address.
- **server**: A structure for handling internet addresses.

Functions:

- **Main**: The Client is connected to the main server
- **Menu**: The menu of available options when the user is not logged in.
- **Login**: Login function for authentification. Logs in the user and sends it to **loggedMenu** if successful.
- **Logout**: Logout function. Logs out the user and sends it to **menu**.
- **Leave**: Terminates the Client execution.
- **CreateAcoount**: Creates a loggable account for the user.

- **LoggedMenu**: The menu of available options when the user is logged in.
- **History**: Info about played games.
- **Game**: The menu where the user selects a game to play.
- **Singleplayer**: Starts a single player game.
- **Duo**: Starts a 1v1 (Duo) game.
- **Ffa**: Starts a 10 (Ffa) players game.
- **Tutorial**: The menu for accessing various information about the game
- **HowToPlay**: Tutorial on how the game works.
- **GameTypes**: Info about the different types of games available.
- **QuestionCategories**: Info about the different question categories available.
- **QuestionTypes**: Info about the different question types available.

3.2 Main Server

The main server handles account creation, account management and game history.

Constants:

- **port**: 2116 - Main server‘s port number.

Global variables:

- **current_thread**: The thread id of the last connected Client.
- **thData**: Structure containing the id of the thread and the descriptor of the socket
- **mutex**: Mutex variable
- **db**: Pointer to the database.
- **err_msg**: String used to store the error message of database operations
- **sd**: Socket address.
- **rc**: Variable to store the value of the return call of database operations.

Functions:

- **Main**: Initiates the server and creates a new thread for each connected Client.

- **Treat**: Function used to handle incoming Client requests
- **Answer**: Receives a command from the Client and fulfills the request if the request is valid.
- **Update_history**: Updates the history of games with a new game
- **History**: Provides the Client the game history of a user.
- **Login**: Log ins the Client if the credentials are correct.
- **Logout**: Log outs the Client.
- **CreateAcoount**: Creates a loggable account if the username provided by the Client isn‘t already taken.
- **Conv_addr**: Converts sockaddr_in object to a ‘IP ADDRESS:PORT NUMBER’ formatted string.

3.3 Game Servers

There are 3 game servers, one for Solo (single player/practice), one for Duo (2 players), and one for Ffa (10 players).

3.4 Solo

The server that administrates solo games.

Constants:

- **port**: 2555 - Solo server‘s port number

Global variables:

- **current_thread**: The thread id of the last connected Client.
- **thData**: Structure containing the id of the thread and the descriptor of the socket
- **mutex**: Mutex variable
- **db**: Pointer to the database.
- **err_msg**: String used to store the error message of database operations
- **sd**: Socket address.
- **rc**: Variable to store the value of the return call of database operations.

Functions:

- **Main**: Initiates the server and creates a new thread (game) for each connected Client.

- **Treat**: Starts the game
- **Solo_game**: Manages the game execution
- **Swap2**: Swaps 2 strings.
- **Shuffle**: Shuffles the elements of an array in random order
- **Conv_addr**: Converts sockaddr_in object to a 'IP ADDRESS:PORT NUMBER' formatted string.

3.5 Duo

The server that administrates duo games.

Constants:

- **port**: 2666 - Duo server's port number

Global variables:

- **Client**: Structure containing a username, a socket address and a flag.
- **Game**: Structure containing an array of 2 Client structures, the **nr_of_users** in the Lobby , **category** and the **ThreadId**
- **Users_data**: An array of 21 **Game** structures
- **db**: Pointer to the database.
- **err_msg**: String used to store the error message of database operations
- **rc**: Variable to store the value of the return call of database operations.

Functions:

- **Main**: Initiates the server and creates a new thread (game) when a Lobby has 2 Clients
- **Treat**: Starts the game
- **Game**: Manages the game execution
- **Swap2**: Swaps 2 strings.
- **Shuffle**: Shuffles the elements of an array in random order
- **Conv_addr**: Converts sockaddr_in object to a 'IP ADDRESS:PORT NUMBER' formatted string.

3.6 Ffa

The server that administrates ffa games.

Constants:

- **port**: 2777 - Ffa server‘s port number

Global variables:

- **Client**: Structure containing a username, a socket address and a flag.
- **Game**: Structure containing an array of 10 Client structures, the **nr_of_users** in the Lobby , **category** and the **ThreadId**
- **Users_data**: An array of 21 **Game** structures
- **db**: Pointer to the database.
- **err_msg**: String used to store the error message of database operations
- **rc**: Variable to store the value of the return call of database operations.

Functions:

- **Main**: Initiates the server and creates a new thread (game) when a Lobby has 10 Clients
- **Treat**: Starts the game
- **Game**: Manages the game execution
- **Swap2**: Swaps 2 strings.
- **Shuffle**: Shuffles the elements of an array in random order
- **Conv_addr**: Converts sockaddr_in object to a ‘IP ADDRESS:PORT NUMBER’ formatted string.

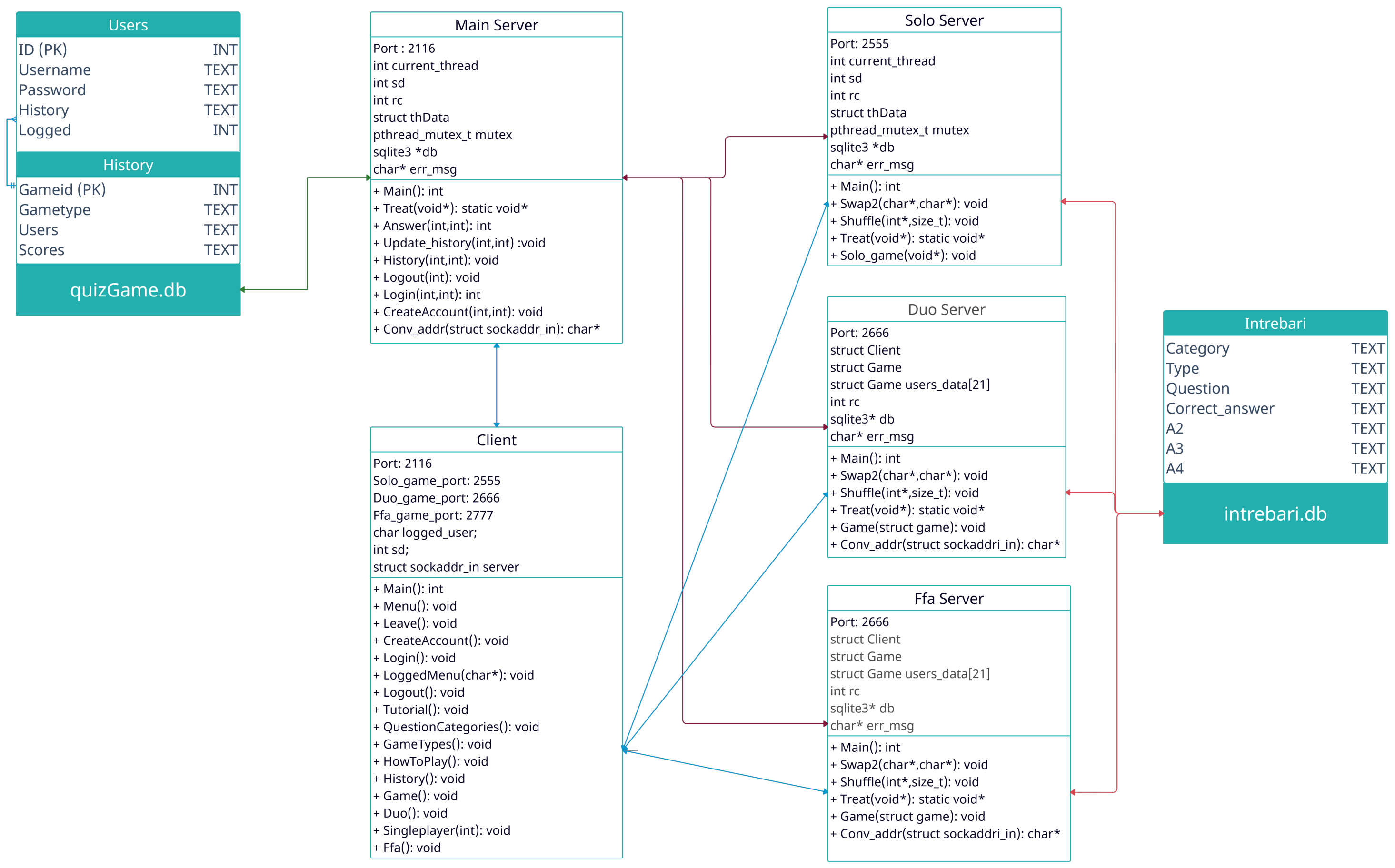

Figure 1: UML Diagram

3.7 Databases

There will be 2 databases. One for users‘ data and their games ,and one that stores all the questions and answers Database **quizGame** will be read/write and store the table **users** and the table **history**. Database **intrebari** will be read only and store the table **intrebari**.

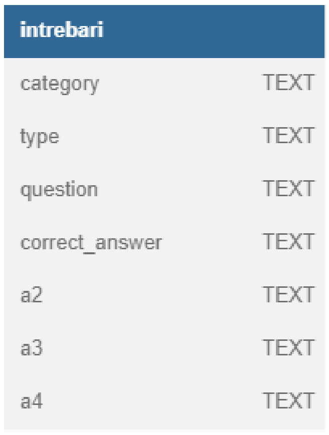

Figure 2: intrebari.db

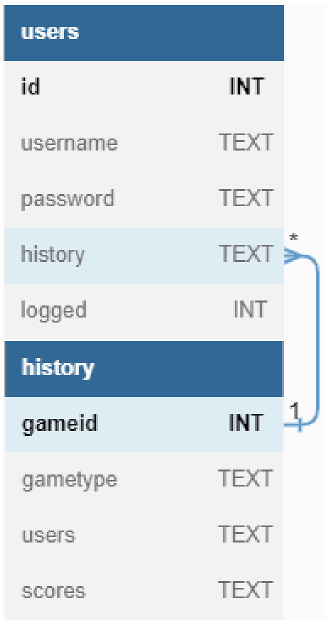

Figure 3: quizGame.db

3.8 Libraries

The C libraries used and required to run the Client and the Servers:

- sys/types.h
- sys/socket.h
- sys/stat.h
- sqlite3.h
- pthread.h
- signal.h
- stdio.h
- stdlib.h
- unistd.h
- errno.h
- string.h
- termios.h

- fcntl.h
- netinet/in.h
- netdb.h
- math.h
- sys/time.h
- ctype.h
- arpa/inet.h

4 Implementation details

4.1 Client

4.1.1 Client is not logged in

- **Menu**: At this stage, the user is currently not logged in. The commands available are **Login**,**Create Account**, and **Quit**. The Client waits for the user's command.

```
void menu()
{
    system("clear");
    tcflush(0, TCIFLUSH);
    char decision = {'\0'};
    int c;

    const char *options = "L to login \nC to create account \
nQ to quit \n \n";
    printf("%s\n", options);

    scanf("%c", &decision);
    while ((c = getchar()) != '\n' && c != EOF);
    decision = tolower(decision);

    switch (decision){
    case 'l': login(); break;
    case 'c': createAccount(0);break;
    case 'q':leave();exit(0);break;
    default: printf("%s\n\n", "Unrecognized");sleep(1);menu();
    }
}

```

- **Login**: The client sends the command **l** to the server and then reads from the user input a username and a password.

```
    char decision = 'l';

    if (write(sd, &decision, sizeof(char)) <= 0){
        perror("Error sending loign request to server\n");
    sleep(1); menu();
    }
    printf("%s", "Username:");
    scanf("%s", username);
    while ((c = getchar()) != '\n' && c != EOF)
        ;
    if (strlen(username) > 0 && username[strlen(username) - 1]
     == '\n') // Removes \n from username
        username[strlen(username) - 1] = '\0';
    printf("%s", "Password:");
    /* changing STDIN display to hide the password while
    typing it*/
    tcgetattr(STDIN_FILENO, &oldt);
    newt = oldt;
    newt.c_lflag &= ~(ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);

    scanf("%s", password);
    while ((c = getchar()) != '\n' && c != EOF);
    /* changing STDIN back to normal display */
    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);

    if (strlen(password) > 0 && password[strlen(password)] ==
    '\n') // Removes \n from password
        password[strlen(password) - 1] = '\0';

```

It sends the encoded string ‘username%password’ to the server and then waits for the server‘s response.

```
    user_size = strlen(username) + strlen(password) + 2;
    sprintf(userdata, "%s%c%s", username, ch, password);
    if (write(sd, &user_size, sizeof(int)) <= 0){ // Sending
    the encoded string length to the server
        perror("Error sending credentials to the server.\n");
    sleep(1);menu();
    }
    if (write(sd, &userdata, user_size) <= 0) {// Sending the
    encoded string to the server
        perror("Error sending credentials to the server.\n");
    sleep(1); menu();
    }

```

The login was successful if the response is **1**. If the response is **0**, the credentials are wrong. If the response is **‘x’**, the user is already logged in. If the response is **‘f’**, the server failed checking the credentials.

```
   if (read(sd, &response, sizeof(char)) <= 0){ // Receving
    response from the server
        perror("Failed to get a response from the server\n");
    sleep(1);menu();
    }
    switch (response){
    case '1':strcpy(logged_user, username);loggedMenu(username
    );break;
    case 'x':printf("Already logged in\n");sleep(1);menu();
    break;
    case 'f':printf("The server has failed checking the
    credentials\n");sleep(1);menu(); break;
    default: printf("Wrong username or password\n");sleep(1);
    menu();
    }

```

After a non-successful login, the user is returned to **menu**. If the login was successful the user is sent to **LoggedMenu**.

- **Create Account**: Restrictions: **username** and **password** must have a length between [6,31] characters and only contain **letters** and **numbers**.

```
const char *allowed = "
    abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
    "; // Allowed characters for username and password

```

The Client reads a **username** from user input. If the **username** doesn't respect the restrictions, the user is warned and asked again for a **username** until the restrictions are met.

```

    const char *username_request = "Enter a username between 6
     and 31 characters:";
    const char *password_request = "Enter a password between 6
     and 31 characters:";
    const char *password_confirmation = "Retype password:";

    printf("%s", username_request);
    scanf("%s", username);
    while ((c = getchar()) != '\n' && c != EOF) // empty STDIN
     buffer
        ;

    while (strlen(username) > 0 && username[strlen(username) -
     1] == '\n') // Removes \n from username
        username[strlen(username) - 1] = '\0';

    if (strlen(username) < 6 || strlen(username) > 31){ //
    Checking if the length restrictions are met
        strlen(username) < 6 ? printf("Username %s is too
    short\n", username) : printf("Username %s is too long\n",
    username);sleep(1);createAccount();
    }

    for (int index = 0; index < strlen(username); index++)
        if (strchr(allowed, username[index]) == NULL){ //
    Checking if the characters restrictions are met
            printf("Username can't contain the character %c
    index %d\n", username[index], index);sleep(1);
    createAccount();
        }

```

If the **username** respects the restrictions it proceeds to read a **password** from user input.

```
    printf("%s", password_request);

   /* changing STDIN display to hide the password while typing it*/

   tcgetattr(STDIN_FILENO, &oldt);
   newt = oldt;
   newt.c_lflag &= ~(ECHO);
   tcsetattr(STDIN_FILENO, TCSANOW, &newt);

   scanf("%s", password);
   while ((c = getchar()) != '\n' && c != EOF); // Empty stdin buffer
   /* changing STDIN back to normal display */
   tcsetattr(STDIN_FILENO, TCSANOW, &oldt);

```

Similarly, if the **password** doesn't respect the restrictions, the user is warned and asked again for a **password** until the restrictions are met. If the **password** respects the restriction it proceeds to ask the user to retype the **password**. If the user correctly retypes the **password**, the Client sends sends the command **c**, the length of the credentials string and the credentials formatted as **'username%password'** to the server and then waits for the server's response.

```
     if (strcmp(password, password2) != 0){ // Checking if passwords match
        printf("Passwords don't match\n");sleep(1) createAccount();
   }
   user_size = strlen(username) + strlen(password) + 2;
   sprintf(userdata, "%s%c%s", username, ch, password);
   if (write(sd, &decision, sizeof(decision)) <= 0){ // Sending account creation request to the server
        perror("Error sending the command create account to server\n"); sleep(1); menu();
   }
   if (write(sd, &user_size, sizeof(int)) <= 0){ // Sending the prefixated encoded string length
        perror("Error sending credentials to the server.\n"); sleep(1);menu();
   }
   if (write(sd, &userdata, user_size) <= 0){ // Sending the encoded string to the server
        perror("Error sending credentials to the server.\n"); sleep(1);menu();
   }

```

If the response is **1**, the operation was successful. If the response is **'x'**, the **username** is already registered. The user is sent back to the **menu**.

```
if (read(sd, &result, sizeof(char)) <= 0) {// Receiving the result of the account creation request
```

```
    perror("Failed to receive an answer from the server)\n
");  sleep(1);menu();
}
switch (result){
case 'l': printf("Account created successfully\n");sleep
(1);break;
case 'x': printf("Username already taken\n");sleep(1);
break;
default: printf("Error creating account\n");sleep(1);
}
menu();

```

- **Leave**: Sends the command **q** to the server and then the Client terminates execution.

4.1.2 Client is logged in

- **LoggedMenu**: At this stage, the user is logged in. The commands available are **Game**, **History**, **Tutorial**, **Logout** and **Quit**.The Client waits for the user's command.
- **Game**: The commands available are **Singleplayer**, **History**, **Duo**, **FFA**, **LoggedMenu** and **Quit**. The Client waits for the user's commands.
  - **Duo & Ffa**: The Client displays the user the available question categories and waits for the user to pick one.

```

const char *correct = "
abcdefghijklmnopqrstuvxABCDEFGHIJKLMNOPQRSTUVX";

printf("%s\n", options);
scanf("%c", &decision);
while ((c = getchar()) != '\n' && c != EOF); // Empty
STDIN buffer
    decision = tolower(decision);

if (strchr(correct, (int)decision) == NULL){ // Check
if the command is not accepted
    printf("\nUnrecognized\n");sleep(1);loggedMenu(''
%%'');
}
switch (decision){
case 'v':close(gd);loggedMenu("%%");break;
case 'x':close(gd); leave();exit(0); break;
}

```

If the user picked a valid category, the Client connects to the duo|ffa server and sends the character **decision** representing the category, the length of the user's **username** and the **username** .

```
if (connect(gd, (struct sockaddr *)&game_server,
sizeof(struct sockaddr)) == -1){ // connecting to the
Ffa|Duo server
```

```
    printf("Server is currently under maintenance. Try again later.\n");sleep(1);loggedMenu("%%");
    }
    if (write(gd, &decision, sizeof(char)) <= 0){ // sending the category of questions to the server
        perror("Error sending category\n");sleep(1);close(gd);loggedMenu("%%");
    }
    if (write(gd, &username_length, sizeof(int)) <= 0){ // sending the username's length to the server
        perror("Error sending username's length\n");sleep(1);close(gd);loggedMenu("%%");
    }
    if (write(gd, &logged_user, username_length) <= 0){ // sending the username to the server
        perror("Error sending username\n");sleep(1);close(gd);loggedMenu("%%");
    }

```

The Client remains in a waiting loop until it receives the value 1 from the server, representing that the game has begun. Additionally, in this loop, when the Client receives any data, it will send (ping) back the data to inform the server that the Client has not disconnected.

```
  printf("Waiting for other users to join\n");

  while (start != 1){ // Game starts when the client receives the ping 1 from the server
    if (read(gd, &start, sizeof(int)) <= 0){ // Receive ping from the server
      printf("Error in queue\n"); sleep(1);close(gd);loggedMenu("%%");
    }
    if (start == 1) // When the game starts we exit the loop without writing to the socket
      break;
    if (write(gd, &start, sizeof(int)) <= 0){ // Send ping back to the server
      printf("Error in queue\n");sleep(1);close(gd);loggedMenu("%%");
    }
  }

```

After the Client exits the loop, the **quizGame** begins and the Client enters a 10 iterations (questions) loop. In each iteration, the Client receives the length of the **type** string, the **type** ("Boolean" for 2 answers questions and "Multiple" for 4 answers questions), the **question's** length and the **question**.

```
 /* The game has started */
 for (int que = 1; que <= 10; que++){ // For each question
   system("clear");        // Clear screen
   tcflush(0, TCIFLUSH); // Clear STDIN buffer

```

```c
    count = 15;             // Used to store the time left to submit an answer
    read_bytes = 0;         // Used to check if the user provided an answer
    ready_for_reading = 0; // Used for timeout read with select

    if (read(gd, &num, sizeof(int)) <= 0){ // Receiving the length of type
      perror("Error reading type length.\n");sleep(1);close(gd);loggedMenu("%%");
    }

    if (read(gd, &type, num) <= 0){ // Receiving the type of the question
      perror("Error reading question type\n");sleep(1);close(gd);loggedMenu("%%");
    }

    if (read(gd, &num, sizeof(int)) <= 0){ // Receiving the question's length
      perror("Error reading question length.\n");sleep(1);close(gd);loggedMenu("%%");
    }

    if (read(gd, &question, num) <= 0){ // Receiving question
      perror("Error reading question.\n");sleep(1);close(gd);loggedMenu("%%");
    }

```

If the **type** is ‘Boolean’ , the Client receives 2 length values and 2 answers.

```c
    if (strcmp(type, "boolean") == 0){ // Boolean type question => only 2 answers
      if (read(gd, &num, sizeof(int)) <= 0){ // Receiving answer1's length
        perror("Error reading answer1 length.\n");sleep(1);close(gd);loggedMenu("%%");
             }
      if (read(gd, &answer1, num) <= 0){ // Receiving answer1
        perror("Error reading answer1 .\n");sleep(1);close(gd);loggedMenu("%%");
       }
      if (read(gd, &num, sizeof(int)) <= 0){ // Receiving answer2's length
        perror("Error reading answer2 length.\n");sleep(1);close(gd);loggedMenu("%%");
      }
      if (read(gd, &answer2, num) <= 0){ // Receiving answer2
        perror("Error reading answer2.\n");sleep(1);close(gd);loggedMenu("%%");
      }
    }
```

```c

```

Otherwise, If the **type** is ‘Multiple’ it receives 4 length values and 4 answers.

```c
  else // Multiple type question => 4 possible answers
  {
    if (read(gd, &num, sizeof(int)) <= 0){ // Receiving answer1's length
      perror("Error reading answer1 length.\n");sleep(1);close(gd);loggedMenu("%%");
    }

    if (read(gd, &answer1, num) <= 0){ // Receiving answer1
      perror("Error reading answer1.\n"); sleep(1);close(gd);loggedMenu("%%");
    }
    if (read(gd, &num, sizeof(int)) <= 0){ // Receiving answer2's length
      perror("Error reading answer2 length.\n");sleep(1);close(gd);loggedMenu("%%");
    }
    if (read(gd, &answer2, num) <= 0){ // Receiving answer2
      perror("Error reading answer2.\n");sleep(1);close(gd);loggedMenu("%%");
    }
    if (read(gd, &num, sizeof(int)) <= 0){ // Receiving answer3's length
      perror("Error reading answer3 length.\n");sleep(1);close(gd);loggedMenu("%%");
    }
    if (read(gd, &answer3, num) <= 0){ // Receiving answer3
      perror("Error reading answer3.\n");sleep(1);close(gd);loggedMenu("%%");
    }
    if (read(gd, &num, sizeof(int)) <= 0){ // Receiving answer4's length
      perror("Error reading answer4 length.\n");sleep(1);close(gd);loggedMenu("%%");
    }
    if (read(gd, &answer4, num) <= 0){ // Receiving answer4
      perror("Error reading answer4.\n"); sleep(1);close(gd);loggedMenu("%%");
    }
  }

```

The Client displays the user the question and the available answers. If the **type** is ‘Boolean’:

```c
system("clear"); // Clear screen
printf("Question %d: %s\n", que, question);
printf("----------------------------------------\n");
printf("a: %s\n", answer1);
```

```
  printf("b: %s\n", answer2);
  printf("Please provide an answer\n\n");

```

Otherwise, if the **type** is ‘Multiple’:

```
 system("clear"); // Clear screen
 printf("Question %d: %s\n", que, question);
 printf("---------------------------------------\n");
 printf("a: %s\n", answer1);
 printf("b: %s\n", answer2);
 printf("c: %s\n", answer3);
 printf("d: %s\n\n", answer4);
 printf("Please provide an answer\n\n");

```

It asigns the variable **count** the value 15, which will represent the time available in seconds for the user to select a valid answer. It beings a loop that will stop when the **count** variable has reached 0 or the user has picked a valid answer. In each iteration we'll use select with a timeout value of 1 second and attempt to read from STDIN after the timeout duration has ‘expired‘ to check if the user provided an answer and decrement **count**. If the user provided a valid answer (a|b for boolean type questions and a|b|c|d for multiple type questions we break out of the loop.

```
  while (((a[0] != 'a' && a[0] != 'b' && strcmp(type, "
    boolean") == 0) || (a[0] != 'a' && a[0] != 'b' && a[0]
     != 'c' && a[0] != 'd' && strcmp(type, "multiple") ==
    0)) && count > 0){
            /* Update the timeout interval for each
    iteration */

    timeout.tv_sec = 1;  // Seconds for each read attempt
    timeout.tv_usec = 0; // Milliseconds for each read
    attempt

    FD_ZERO(&input_set);              // Macro to clear
    the file descriptors set
    FD_SET(STDIN_FILENO, &input_set); // Adding STDIN to
    input_set

    fflush(stdout); // Empy STDOUT buffer

    count == 1 ? printf("(%d second left)", count) :
    printf("(%d seconds left)", count);          printf("\
    n");
    ready_for_reading = select(1, &input_set, NULL, NULL,
    &timeout); // 1 sec timeout read

    if (ready_for_reading == -1){
      printf("Error reading user input\n");close(gd);sleep
    (1);loggedMenu("%");
    }

    if (ready_for_reading){ // 1 second has passed and we
    check if the user wrote something in the STDIN buffer
```

```
      read_bytes = read(0, a, sizeof(a));
      while (strlen(a) > 0 && a[strlen(a) - 1] == '\n') // Removes \n from a
        a[strlen(a) - 1] = '\0';
      a[0] = tolower(a[0]);
      if (read_bytes != 0){// We check if the user provided an answer
        tcflush(0, TCIFLUSH);
        if ((a[0] != 'a' && a[0] != 'b' && (strcmp(type, "boolean") == 0)) && a[0] != '\n') // Answer not accepted
          printf("\nAccepted answers are: a,b \n");
        else if ((a[0] != 'a' && a[0] != 'b' && a[0] != 'c' && a[0] != 'd' && (strcmp(type, "multiple") == 0)) && a[0] != '\n') // Answer not accepted
        printf("\nAccepted answers are: a,b,c,d \n");
      }
    }
    count--;
    /* Answer accepted */
    if (((a[0] == 'a' || a[0] == 'b') && strcmp(type, "boolean") == 0) || ((a[0] == 'a' || a[0] == 'b' || a[0] == 'c' || a[0] == 'd') && strcmp(type, "multiple") == 0)){
    printf("Answer submited: %c\n\n", a[0]);  break;
    }
  }

```

If the variable **count** is > 0, the Client begins another loop that will decrement **count** every second and stop when **count** has reached 0.

```
            while (count > 0) {// If there is any time left after the user provided the answer
            count == 1 ? printf("(%d second left)\n", count) : printf("(%d seconds left)\n", count);
            sleep(1); count--;
        }

```

If the user has picked a valid answer the Client sends the user's answer to the ffa/duo server, otherwise, if the user has not provided a valid answer, or has not provided any answer, the Client sends 'x' to the server.

```
  if (strcmp(type, "boolean") == 0) // Boolean type question
  {
    if (a[0] == 'a' || a[0] == 'b')
    {
      if (write(gd, &a[0], sizeof(char)) <= 0){ // User has submited an accepted answer
        perror("Error sending answer.\n");close(gd);sleep(1);loggedMenu("%%");
      }
    }
    else if (write(gd, &l, sizeof(char)) <= 0) {// User didn't submit an accepted answer
```

```
      perror("Error sending null answer.\n");close(gd);sleep(1);loggedMenu("%%");
    }
  }
  else if (strcmp(type, "multiple") == 0) // Multiple type question
  {
    if (a[0] == 'a' || a[0] == 'b' || a[0] == 'c' || a[0] == 'd')
    {
      if (write(gd, &a[0], sizeof(char)) <= 0){ // User has submited an accepted answer
        perror("Error sending answer.\n");close(gd);sleep(1);loggedMenu("%%");
      }
    }
    else
    if (write(gd, &l, sizeof(char)) <= 0){ // User didn`t submit an accepted answer
      perror("Error sending null answer.\n");close(gd);sleep(1);loggedMenu("%%");
    }
  }

```

After the game has ended, the Client receives a value from the server representing the **score** obtained and displays it to the user.

```
  /* game has finished */
  if (read(gd, &final_score, sizeof(int)) <= 0){ // Receiving the score obtained
    perror("Error reading score.\n");sleep(1);close(gd);loggedMenu("%%")
  }
  printf("\nCongratulations %s! You got %d questions right\n\n", logged_user, final_score);

```

Additionally, the Client receives the **number of winners** (there can be multiple users with the maximum score) of that game, and for each winner, the Client receives his **score**, username length and **username** and displays it to the user.

```
  if (read(gd, &nr_of_winners, sizeof(int)) <= 0){ // Receiving the number of winners
    perror("Error reading number of winners.\n");sleep(1);close(gd);loggedMenu("%%");
  }
  for (int contor = 0; contor < nr_of_winners; contor++){ // For each winner
    if (read(gd, &final_score, sizeof(int)) <= 0){ // Receiving the winner`s score
      perror("Error reading winner score.\n"); sleep(1);close(gd);loggedMenu("%%");
    }
    if (read(gd, &user_length, sizeof(int)) <= 0){ // Receiving the winner`s username length
```

```
      perror("Error reading winner‘s length \n");sleep(1);
     close(gd);loggedMenu("%%");
    }
    if (read(gd, &winner, user_length) <= 0){ // Receiving
     the winner‘s username
      perror("Error reading winner score.\n");sleep(1);
    close(gd);loggedMenu("%%");
    }
    printf("%s : %d points\n", winner, final_score);
  }
```

The Client closes the connection to duo|ffa server and returns to **loggedMenu**.

```
  sleep(10);close(gd);loggedMenu("%%");

```

- **Singleplayer**: The Client displays the user the available question categories and waits for the user to pick one. If the user picked a valid category, the Client connects to the solo server and sends the character **decision** representing the category, the length of the user‘s **username** and the **username** . The **quizGame** begins and the Client enters a 10 iterations (questions) loop. The Client executes the same code as in the Duo/Ffa function. When the game ends, the Client receives a value from the server representing the score obtained and displays it to the user. The Client closes the connection to solo server and returns to **loggedMenu**.

- **History**:The Client waits for a **username** from user input. The Client sends to the server the command **h**, the length of the **username** and the **username**.

```
  scanf("%s", username);
  while ((c = getchar()) != '\n' && c != EOF) // Empty STDIN
    buffer
  if (strlen(username) > 0 && username[strlen(username)] == '\
    n') // Removes \n from      username
  username[strlen(username) - 1] = '\0';
  user_size = strlen(username);
  if (write(sd, &decision, sizeof(decision)) <= 0){ // Sending
     command history to the server
        perror("Error sending command history to server\n");
    sleep(1);loggedMenu("%%");
    }
    if (write(sd, &user_size, sizeof(user_size)) <= 0){ //
    Sending prefixated username length to  the server
        perror("Error sending username to the server.\n");
    sleep(1);loggedMenu("%%");
    }
    if (write(sd, &username, user_size) <= 0){ // Sending
    username to the server
        perror("Error sending username to the server.\n");
    sleep(1);loggedMenu("%%");
    }

```

The Client then receives an answer to the request. If the answer is **-1**, the username is incorrect. If the answer is **0**, the user hasn‘t played any games yet.

```
  if (read(sd, &nr_of_games, sizeof(nr_of_games)) <= 0){ // Receiving the number of games of the user
        perror("Error getting history data\n");sleep(1);loggedMenu("%%");
    }
    if (nr_of_games == -1 || nr_of_games == 0)
    {
    nr_of_games == -1 ? printf("User not found\n\n") : printf("User has not played any games yet\n\n");
          printf("%s\n", options);
          scanf("%c", &decision);
          while ((c = getchar()) != '\n' && c != EOF)
             decision = tolower(decision);
          switch (decision){
             case 'b': loggedMenu("%%");break;
             case 'h': history();break;
             default: printf("%s\n", "Unrecognized"); sleep(1); loggedMenu("%%");
        }
    }

```

If the answer is > 0, the answer represents the number of games and the Client creates a loop with **nr_of_games** iterations. At each iteration the Client receives from the server the **gameid**, **type**, **username** of the winner, **nr_of_users** in the game, and the **username** and **score** for each user and displays it to the user.

```
  for (int i = 0; i < nr_of_games; i++){
    if (read(sd, &gameid, sizeof(int)) <= 0){ // Receiving the id of the game
      perror("Error reading gameid.\n");sleep(1);loggedMenu("%%");
    }

    if (read(sd, &gametype, sizeof(gametype)) <= 0){ // Receiving the type of the game
      perror("Error reading game type.\n");sleep(1);loggedMenu("%%");
    }
    if (read(sd, &winner, sizeof(winner)) <= 0){ // Receiving the username of the winner
      perror("Error reading game winner.\n");sleep(1);loggedMenu("%%");
    }
    printf("Match id:%d \nGame type:%s\n", gameid, gametype);
    if (strlen(gametype) < 4) // No winners in a solo game
      printf("Winner:%s\n\n", winner);
    else
      printf("Usernames                                        Scores\n");
    printf("-----------------------------------------------------------------\n"); // 65 -
    if (read(sd, &nr_of_users, sizeof(int)) <= 0){ // Receiving the nr of users in the game
      perror("Error reading the number of players.\n");sleep(1);loggedMenu("%%");
    }
    for (int k = 0; k < nr_of_users; k++){ // Receiving the username and score for each user
    score = 0;
    /* Receiving the username of the user */
    if (read(sd, &username_game, sizeof(username_game)) <= 0){
      perror("Error reading players.\n");sleep(1);loggedMenu("%%");
    }

    if (read(sd, &score, sizeof(int)) <= 0){ // Receiving the score of the user
      perror("Error reading score.\n");sleep(1);loggedMenu("%%");
    }
    printf("%s%s%d\n", username_game, spaces + strlen(username_game), score);
    }
    printf("—————————————————————————————————\n\n"); // 65
    }
```

- **Tutorial**
  - **HowToPlay**: The client displays a brief summary of how the game works.
  - **Gametypes**: The client displays an overview of all game types.
  - **QuestionCategories**:The client displays an overview of all question categories
- **Logout** The Client sends the command **o** to the server and the user is returned to **Menu**

```
  if (write(sd, &decision, sizeof(char)) <= 0){ // Sending logout request to the server
    perror("Error sending logout request to server.\n");sleep(1);loggedMenu("%%");
      }
    menu();
```

- **Leave** The Client sends the command **q** to the server and terminates execution.

```
  if (write(sd, &decision, 1) <= 0){ // Sending logout request to the server
    printf("Error sending quit command to server\n");sleep(1);menu();
  }
  return;
```

4.2 Servers

4.2.1 Main Server

- **Main**: Logs off all the users (changes the logged flag in the database to 0).

```c
rc = sqlite3_open("quizGame.db", &db); // Opening quizGame
    database
if (rc){
  fprintf(stderr, "Failed opening quizGame database: %s\n",
    sqlite3_errmsg(db));
  return 0;
  }

/* logging off all users when starting the server */

rc = sqlite3_exec(db, "UPDATE users SET logged=0;", 0, 0, &
    err_msg);

if (rc != SQLITE_OK){
  fprintf(stderr, "Failed to log off users.%s\n",
    sqlite3_errmsg(db));
  return 0;
 }

```

  If the accept is successful, a thread is allocated for the Client, the session begins and the function Treat begins execution.

```c
if ((client = accept(sd, (struct sockaddr *)&from, &length)) <
     0){
  printf("Error establishing connection with a client\n");
  continue;
}

td = (struct thData *)malloc(sizeof(struct thData));
td->idThread = current_thread++;
td->cl = client;

pthread_create(&th[current_thread], NULL, &treat, td);
```

- **Treat**: It creates an infinite loop for handling Client requests in which the **answer** function is executed each iteration. When the Client terminates execution, it closes the connection, and detaches the thread . The call of the **answer** function can return 4 values: **-404**, **-999**, **0** or **a valid user ID value** >**0**. **-404** means that the user has logged out, **-999** means that the Client terminated execution, **0** means that no **login**,**logout** or **quit** operation has been made by the Client, and any other value returned means that the user with the returned **id** value has logged in. When the return value of the **answer** function is **-999**, the connection with the Client is closed and the thread is detached.

```c
tdL = *((struct thData *)arg);

```

```c
while (1){
  temp_id = answer(tdL.cl, tdL.idThread);
  if (temp_id != -404 && temp_id != -999 && temp_id != 0)
    id = temp_id;

  if (temp_id == -999){ // Received exit command or the client
     disconnected
    logout(id);

    /* To prevent race condition where two threads would
    decrement the shared variable at the same time */

    pthread_mutex_lock(&mutex);
    current_thread--;
    pthread_mutex_unlock(&mutex);
    close(tdL.cl);
    pthread_detach(pthread_self());
    return (NULL);
    break;
  }
}
```

- **Answer**: The Server thread waits until it receives a request (a character that will be interpreted by the server) from the Client. A user can send the following requests: ' **Login**, **Logout**,**CreateAccount**, **History**, **Quit**. The **update_history** request can only be made by a game server.

```c
if (read(cl, &decision, sizeof(decision)) <= 0) return -999;

decision = tolower(decision);

switch (decision){
  case 'l':{
    logged_user_id = login(cl, idThread);
    if (logged_user_id != -404)
      return logged_user_id;
    else
      return -404;
      }
  case 'c': {createAccount(cl, idThread); return 0;}
  case 'h': {history(cl, idThread);return 0;}
  case 'o': {logout(logged_user_id);return -404;}
  case 'q': { return -999;}
  case '+': {update_history(cl, idThread);return -999;}
  default: return 0; // Should never happen
}
```

- **Login**: The Server receives the length of the encoded string 'username%password', and then the string from the Client.

```c
if (read(cl, &user_size, sizeof(int)) <= 0){ // Receiving the
    length of the username%password string
  fprintf(stderr, "[Thread %d] Failed to receive credentials
    length. %s\n", idThread, err_msg);
  return 0;
}
```

```
if (read(cl, &data, user_size) <= 0){ // Receiving the
    username%password string
    fprintf(stderr, "[Thread %d] Failed to receive credentials
    . %s\n", idThread, err_msg);
    return 0;
}
```

It decodes the string into 2 variables, **username** and **password**.

```
token = strtok(data, "%");
while (token != NULL){
  cont ? strcpy(password, token) : strcpy(username, token);
  cont++;
  token = strtok(NULL, "%");
}
```

It checks the **users** table from the quizGame database for a match.

```
sprintf(query, "SELECT *FROM users WHERE username='%s' and
    password='%s';", username, password);
struct sqlite3_stmt *selectstmt;
result = sqlite3_prepare_v2(db, query, -1, &selectstmt, NULL);
if (result == SQLITE_OK){ // Query is valid
    if (sqlite3_step(selectstmt) == SQLITE_ROW) // Credentials
     are correct
```

If there are no matches, the Server sends to the Client the response '0', representing that the credentials are incorrect.

```
char response = '0';
if (write(cl, &response, sizeof(char)) <= 0)// Sending the
    client the answer '0' (login request denied)
  return 0;
else
  return -404;
```

If there is a match, we verify if the user is not already logged in by checking his **logged** flag.

```
sprintf(query, "SELECT logged FROM users where username='%s';"
    , username);
if (sqlite3_prepare_v2(db, query, 128, &r1, NULL) != SQLITE_OK
    ) // Query is invalid
  return -404;
if (sqlite3_step(r1) == SQLITE_ROW) // User has a valid logged
     field
        if (sqlite3_column_int64(r1, 0) == 1) // A Client is
    already logged on that account
        if (sqlite3_column_int64(r1, 0) == 0) // No Client is
    already logged on that account
```

If someone is already logged in on that account, the Server sends to the Client the response 'x'.

```
char result = 'x';
if (write(cl, &result, sizeof(char)) <= 0) // Sending the
    client the answer 'x' (login request denied)
  return -404;
```

Otherwise, if there is no Client logged on that account, the Server updates his **logged** flag to 1, and finds his **id**.

```
sprintf(query, "UPDATE users SET logged=1 WHERE username='%s';
    ", username);
rc = sqlite3_exec(db, query, 0, 0, &err_msg); // Updating the
    logged flag of the user to 1
if (rc != SQLITE_OK) // Failed to update the logged flag of
    the user
  return -404;
sprintf(query, "SELECT id FROM users where username='%s';",
    username);
if (sqlite3_prepare_v2(db, query, 128, &r1, NULL) != SQLITE_OK
    ) // Query is invalid
  return -404;
```

If the operations are successful, the Server sends the response 'l' to the Client, representing that the login request was successful.

```
sqlite3_step(r1);
id = sqlite3_column_int64(r1, 0);
char result = 'l';
 if (write(cl, &result, sizeof(char)) <= 0) // Sending the
    client the answer 'l' (login request successful)
  return -404;
else // Login request was successful. We save the id of the
    user for logout
  return id;
```

Otherwise, if any operation fails, the Server sends the response 'f' to the Client.

```
char result = 'f';
if (write(cl, &result, sizeof(char)) <= 0){ // Sending the
    client the answer 'f' (login request denied)
  return -404;
```

The function login returns **-404** if the login request is unsuccessful ,and the **id** of the user if the login request is successful. The **id** will be used later by the **logout** function to change the **logged** flag of the user in the database after the Client logged on that account discconects or logs out.

- **CreateAccount**:The Server receives the length of the encoded string 'username%password', and then the string from the Client.

```
if (read(cl, &user_size, sizeof(int)) <= 0){ // Receiving the
    length of the username%password string
  fprintf(stderr, "[Thread %d] Failed to receive credentials
    length. %s\n", idThread, err_msg);
  return 0;
}
if (read(cl, &data, user_size) <= 0){ // Receiving the
    username%password string
    fprintf(stderr, "[Thread %d] Failed to receive credentials
    . %s\n", idThread, err_msg);
    return 0;
}
```

It decodes the string using into 2 variables, **username** and **password**.

```
 token = strtok(data, "%");
while (token != NULL){
  cont ? strcpy(password, token) : strcpy(username, token);
  cont++;
  token = strtok(NULL, "%");
}
```

It checks the **users** table from the quizGame database for a match.

```
sprintf(query, "SELECT *FROM users WHERE username='%s';",
    username);
struct sqlite3_stmt *selectstmt;
result = sqlite3_prepare_v2(db, query, -1, &selectstmt, NULL);
     // Compiling the query
if (result == SQLITE_OK){// Query is valid
    if (sqlite3_step(selectstmt) == SQLITE_ROW) // Username
    already taken
    if (sqlite3_step(selectstmt) != SQLITE_ROW) // Username
    free
}
```

If there is a match, the Server sends the response ‘x’ to the Client (username is already taken).

```
char response = 'x';
if (write(cl, &response, sizeof(char)) <= 0){ // Sending the
    client the answer 'x' (account creation request denied)
  fprintf(stderr, "[Thread %d] Failed to send account creation
     request response to the client.%s\n", idThread, err_msg);
  return;
}
```

If there are no matches, the Server finds an **id** to asign to the username, and inserts the user‘s data in the database.

```
 struct sqlite3_stmt *asignid;
sprintf(query, "SELECT max(id) from users;"); // Finding an id
     for the user
sub_result = sqlite3_prepare_v2(db, query, 256, &asignid, NULL
    );
if (sub_result != SQLITE_OK){ // Query is invalid
  fprintf(stderr, "[Thread %d] Failed to fetch the maximum id.
     %s\n", idThread, sqlite3_errmsg(db));
     sqlite3_free(err_msg);
     return;
}
else if (sqlite3_step(asignid) == SQLITE_ROW){ // Query is
    valid
  id = sqlite3_column_int(asignid, 0);
  id++;
}
sprintf(query, "INSERT INTO users (id,username,password,
    history,logged) values(%d,'%s','%s','',%d);", id, username
    , password, 0);
sub_result = sqlite3_exec(db, query, 0, 0, &err_msg);
if (sub_result != SQLITE_OK){ // Execution failed
```

```
  fprintf(stderr, "[Thread %d] Failed to create account. %s\n"
    , idThread, sqlite3_errmsg(db));
  sqlite3_free(err_msg);
  return;
}
```

If the operations are successful, the Server sends the response ‘l’ to the Client, representing that the account creating request was successful.

```
char response = 'l';
if (write(cl, &response, sizeof(response)) <= 0){ // Sending
    the client the answer 'l' (account created successfuly)
  fprintf(stderr, "[Thread %d]Error sending response. %s\n",
    idThread, err_msg);
  return;
}
```

Otherwise, the Server sends the response ‘f’ to the Client (error creating account).

```
char response = 'f';
if (write(cl, &response, sizeof(char)) <= 0){ // Sending the
    client the answer 'f' (account creation request denied)
  fprintf(stderr, "[Thread %d] Failed to send account creation
     request response to the client.%s\n", idThread, err_msg);
  return;
}
return;
```

- **Logout**: Receives as argument the **id** of the user to logout and changes the **logged** flag of that user in the database to 0.

```
sprintf(query, "UPDATE users SET logged=0 WHERE id='%d';", id)
    ;
int rc = sqlite3_exec(db, query, 0, 0, &err_msg);
if (rc != SQLITE_OK){
  fprintf(stderr, "Failed logging off user with id %d. %s\n",
    id, sqlite3_errmsg(db));
  sqlite3_free(err_msg);
}
return;
```

- **History**: The Server receives a **username**’s length and a username from the Client.

```
if (read(cl, &user_size, sizeof(user_size)) <= 0) {//
    Receiving username length
  printf("[Thread %d] Error receiving username length \n",
    idThread);
  return;
 }
if (read(cl, &username, user_size) <= 0){ // Receiving
    username
  printf("[Thread %d] Error receiving username \n", idThread);
  return;
}
```

The Server checks if the username exists in the database

```
sprintf(query, "SELECT *FROM users WHERE username='%s';", username);
result = sqlite3_prepare_v2(db, query, -1, &selectstmt, NULL);
if (result == SQLITE_OK)
    if (sqlite3_step(selectstmt) != SQLITE_ROW) // Checking if the requested username is not in the database
```

If the username is not in the database, the Server sends the response -1 to the Client.

```
int response = -1;
if (write(cl, &response, sizeof(response)) <= 0){
  printf("[Thread %d] Failed sending response to the client\n", idThread);
  return;
}

```

Otherwise, if the username exists, the Server fetches the history for that username from the database.

```
sprintf(query, "SELECT history FROM users WHERE username='%s';", username); // Getting the id of the games where the username played
if (sqlite3_prepare_v2(db, query, 128, &r1, &t) != SQLITE_OK){
  printf("[Thread %d]Invalid query %s\n", idThread, sqlite3_errmsg(db));
  sqlite3_free(err_msg);
  return;
}
sqlite3_step(r1);

```

It saves the history of the user and saves all the **gameids** in an array.

```
strcpy(history, sqlite3_column_text(r1, 0)); // History string for that user
while ((p = strtok(contor ? NULL : history, ",")) != NULL) {// The game ids are stored as a comma separated string
        history_ids[contor++] = atoi(p); // Saving the game ids in an array
}
```

It sends to the Client the number of the games played by that user.

```
if (write(cl, &contor, sizeof(int)) <= 0){ // Sending to the client the nr of games
  printf("[Thread %d] Failed sending the number of games to the client.\n", idThread);
  return;
}

```

If the user has played any games, the Server iterates over all the **gameids** and fetches the game data (game type, users in the game and the final scores of the users) for each **gameid**.

```
for (i = 0; i < contor; i++) {// iterating over all the games
  sprintf(query, "SELECT *FROM history WHERE gameid='%d';",
    history_ids[i]); // for each game id we will get info from
     the history database
  gameid = sqlite3_column_int64(r2, 0);          // gameid
     strcpy(gametype, sqlite3_column_text(r2, 1)); // type of
    game
     strcpy(users, sqlite3_column_text(r2, 2));    // string
    of users` id
     strcpy(scores, sqlite3_column_text(r2, 3)); // string of
    users`s score
```

The Server saves the **id** of the users and their scores in 2 arrays, and for each user it finds his username.

```
while ((p = strtok(index_strtok ? NULL : users, ",")) != NULL)
     {// Exctracting user ids from comma separated string
    users
  users_id[index_strtok++] = atoi(p); // string to integer
}
while ((p = strtok(index_strtok ? NULL : scores, ",")) != NULL
    ){ // Exctracting scores from comma separated string
    scores
  users_scores[index_strtok++] = atoi(p);
}
```

It finds and saves the username of every user.

```
for (int op = 0; op < index_strtok; op++){ // Getting the
    username for each userid in the game
  sprintf(sub_sub_query, "SELECT username FROM users where id
    ='%d';", users_id[op]);
  while (sqlite3_step(r3) == SQLITE_ROW){
     strcpy(game_usernames[op], sqlite3_column_text(r3, 0));
  }
```

The server sends to the Client ,the **gameid**, the **gametype**, the **username** of the Winner, the **nr_of_users** in the game, and for each user, his **username** and **score**.

```
 if (write(cl, &gameid, sizeof(int)) <= 0){ // Sending to
    client the gameid
  printf("[Thread %d] Failed sending the gameid.\n", idThread)
    ;
  return;
}
if (write(cl, &gametype, sizeof(gametype)) <= 0){ // Sending
    to client the game type
  printf("[Thread %d] Failed sending the game type.\n",
    idThread);
  return;
}
if (write(cl, &game_usernames[max_score_index], sizeof(
    username)) <= 0){ // Sending to client the winner of the
    game
  printf("[Thread %d] Failed sending the winner of the game.\n
    ", idThread);
  return;
}
if (write(cl, &index_strtok, sizeof(index_strtok)) <= 0){ //
    Sending to client the nr of users in the game
  printf("[Thread %d] Failed sending the number of players.\n"
    , idThread);
  return;
}
for (int idk = 0; idk < index_strtok; idk++){ // Iteratinng
    over all users
  if (write(cl, &game_usernames[idk], sizeof(username)) <= 0){
     // Sending to client the usernames
    printf("[Thread %d] Failed sending the players' usernames
    .\n", idThread);
    return;
  }
  if (write(cl, &users_scores[idk], sizeof(int)) <= 0){ //
    Sending to client the scores
    printf("[Thread %d] Failed sending players' scores.\n",
    idThread);
    return;
  }
}
```

- **Update_history**: The Server receives **type** length and **type** from a game server.

```
if (read(cl, &size, sizeof(int)) <= 0){ // Receiving type
    length
    printf("[Thread %d] Solo game server has disconnected \n",
     idThread);
    return;
}
if (read(cl, &game_type, size) <= 0){ // Receiving type
    printf("[Thread %d] Solo game server has disconnected \n",
     idThread);
    return;
}
```

If **type** is 'Solo' , it receives **username** length, **username** and **score** from the solo server.

```
 if (read(cl, &size, sizeof(int)) <= 0){ // Receiving username
     length
      printf("[Thread %d] Solo game server has disconnected \n
    ", idThread);
      return;
}
if (read(cl, &username, size) <= 0){ // Receiving username
      printf("[Thread %d] Solo game server has disconnected \n
    ", idThread);
      return;
}
if (read(cl, &score, sizeof(int)) <= 0){ // Receiving score
      printf("[Thread %d] Solo game server has disconnected \n
    ", idThread);
      return;
}

```

It pings the solo server that all the data has been received and the connection is closed on both sides.

```
int data_received_confirmation = 1;
if (write(cl, &data_received_confirmation, sizeof(int)) <= 0){
	printf("[Thread %d] Solo game server has disconnected \n", idThread);
	return;
}

```

The Server fetches the **id** and the **history** of the user.

```
sprintf(query, "SELECT id,history from users where username='%s';", username);
if (sqlite3_prepare_v2(db, query, 33133, &result, &t) != SQLITE_OK){
	printf("Can't retrieve data: %s\n", sqlite3_errmsg(db));
	sqlite3_free(err_msg);
	return;
}
if (sqlite3_step(result) == SQLITE_ROW){
	user_id = sqlite3_column_int64(result, 0);		// Id of the player
	strcpy(user_history, sqlite3_column_text(result, 1)); // history of the player
}
```

It finds an **gameid** for the game.

```
sprintf(query, "SELECT max(gameid) from history;");
if (sqlite3_step(result2) == SQLITE_ROW){ //
	gameid = sqlite3_column_int64(result2, 0); // Game id
	gameid++;
}
```

It adds the game data to the history and the **gameid** to the history of the user.

```
sprintf(query, "INSERT INTO history values(%d,'Solo',%d,%d);", gameid, user_id, score);
rc = sqlite3_exec(db, query, 0, 0, &err_msg);
sprintf(query, "UPDATE users set history='%s' where username='%s';", user_history, username);
rc = sqlite3_exec(db, query, 0, 0, &err_msg);
```

Otherwise, if **type** is ‘Duo’ or ‘Ffa’ , it receives the **nr_of_users** from the duo|ffa server.

```
if (read(cl, &nr_of_users, sizeof(nr_of_users)) <= 0){ // Receiving the number of users
	printf("[Thread %d] Client disconnected \n", idThread);
	return;
}

```

It finds an **gameid** for the game.

```
sprintf(query, "SELECT max(gameid) from history;");
if (sqlite3_prepare_v2(db, query, 33133, &result, &t) !=
    SQLITE_OK){
  printf("Can't retrieve data: %s\n", sqlite3_errmsg(db));
  return;
}
if (sqlite3_step(result) == SQLITE_ROW){
  gameid = sqlite3_column_int64(result, 0);
  gameid++;
}
```

The string **useridstring** will store the ids of the users, and **scorestring** will store the scores of the users. For each user in the game, the Server receives his **username** length, **username** and **score**,and fetches the **id** and the **history** of the user.

```
if (read(cl, &size, sizeof(int)) <= 0){ // Receiving the
    username length
    ...
    }
if (read(cl, &username, size) <= 0){ // Receiving the username
    ...
        }
if (read(cl, &score, sizeof(int)) <= 0){ // Receiving the
    score
    ...
        }
sprintf(query, "SELECT id,history from users where username='%
    s';", username);
if (sqlite3_prepare_v2(db, query, 33133, &result, &t) !=
    SQLITE_OK){
    ...
        }
if (sqlite3_step(result) == SQLITE_ROW){
        user_id = sqlite3_column_int64(result, 0);
    // User id
        strcpy(user_history, sqlite3_column_text(result, 1));
    // User history
}
```

The server adds **gameid** to the history of every user and appends the user‘s **id** and **score** to **useridstring** and **scorestring**

```
sprintf(query, "UPDATE users set history='%s' where username
    ='%s';", user_history, username);
rc = sqlite3_exec(db, query, 0, 0, &err_msg);
strcat(users_ids, useridstring);
strcat(users_scores, scorestring);
```

After the Server has received the game data, it adds the game to the history , and pings the game server that all the data has been received. The connection is closed on both sides.

```
strcmp(game_type, "Duo") == 0 ? sprintf(query, "INSERT INTO
    history values(%d,'Duo','%s','%s');", gameid, users_ids,
    users_scores) : sprintf(query, "INSERT INTO history values
    (%d,'Ffa','%s','%s');", gameid, users_ids, users_scores);
rc = sqlite3_exec(db, query, 0, 0, &err_msg);
int data_received_confirmation = 1;
if (write(cl, &data_received_confirmation, sizeof(
    data_received_confirmation)) <= 0){ // Sending
    confirmation
  ...
      }
```

4.2.2 Game Servers

4.2.2.1 Duo|Ffa The **client** structure stores the **username**, **socket_address** and **flag** (1-connected 0-disconnected) of the Client.

```
struct client{
  char username[32];
  int socket_address;
  int flag;
};
```

The **game** structure will represent a Lobby. It contains a **users** array of size 2(Duo)|10(Ffa) **client** structures, the **nr_of_users** in the Lobby, the **idThread** and the predefined **category** asigned to this Lobby.

```
struct game{
  struct client users[2];
  int nr_of_users;
  int idThread;
  char category[40];
};
```

The **users_data** array of size 21 game structures will save all the game Lobbies. There is a Lobby for each category of questions.

```
struct game users_data[21];
```

– **Main** Due to the possibility of a Client disconnecting mid-game when attempting to send quiz data, we install a signal handler(SIG_IGN) that ignores the SIGPIPE signal which would terminate the execution of the server.

```
  signal(SIGPIPE, SIG_IGN);

```

When a Client connects, the Server receives a character(**decision**) that will represent the category the user picked, the **username** length and the **username** from the Client.

```
if (read(fd, &decision, sizeof(char)) <= 0){ // Receiving
    category
    ...
    }
```

```
if (read(fd, &username_size, sizeof(int)) <= 0){//
    Receiving username length
    ...
          }
if (read(fd, &username, username_size) <= 0){ // Receiving
     username
    ...
          }

```

Based on the category, the Client is asigned to a Lobby, and his data (username, socket_address , flag=1) is stored in a **client** structure. The **nr_of_users** in that Lobby is incremented and the **socket_address** of the user is removed from active sockets.

```
users_data[index].users[users_data[index].nr_of_users].
    socket_address = fd;
FD_CLR(fd, &active_sockets);
users_data[index].users[users_data[index].nr_of_users].
    flag = 1;
strcpy(users_data[index].users[users_data[index].
    nr_of_users].username, username);
users_data[index].nr_of_users++;
```

The server checks if the Lobby has enough players to start the game, and if so, it pings each Client to check if anyone has disconnected. If the ping fails, the Server removes the Client from the Lobby.

```
if (users_data[index].nr_of_users == 2){ // For duo server
  startgame = 2;
  for (k = 0; k < users_data[index].nr_of_users; k++){
    /* Pings each socket to check if any users
    disconnected before starting the game*/
    if (write(users_data[index].users[k].socket_address, &
    startgame, sizeof(int)) <= 0){ // Pinging the socket
      close(users_data[index].users[k].socket_address);
      users_data[index].nr_of_users--;
      printf("User %s has disconnected.\n", users_data[
    index].users[k].username);
      startgame = 0;
      for (int j = k; j < users_data[index].nr_of_users; j
    ++){ // left shift users
        users_data[index].users[j].flag = users_data[index
    ].users[j + 1].flag;
         users_data[index].users[j].socket_address =
    users_data[index].users[j + 1].socket_address;
        strcpy(users_data[index].users[j].username,
    users_data[index].users[j + 1].username);
      }
}
    if (read(users_data[index].users[k].socket_address, &
    ping_response, sizeof(int)) <= 0){ // Receiving ping
    response
        close(users_data[index].users[k].socket_address);
        users_data[index].nr_of_users--;
        printf("User %s has disconnected.\n", users_data[
    index].users[k].username);
```

```
        startgame = 0;
         /* If any user disconnected ,we won't start the
    game and remove the user's data from the structure by
    left shifting the elements at his position*/
        for (int j = k; j < users_data[index].nr_of_users;
  j++){// left shift users
          users_data[index].users[j].flag = users_data[
    index].users[j + 1].flag;
          users_data[index].users[j].socket_address =
    users_data[index].users[j + 1].socket_address;
          strcpy(users_data[index].users[j].username,
    users_data[index].users[j + 1].username);
          }
    }
}
```

If no client has disconnected, the Server pings each Client to inform them that the game has begun,creates a thread (the game) ,passes the Lobby data to the **treat** function and empties the Lobby.

```
if (startgame == 2){ // We can start the game
  startgame = 1;
  nr_of_games++;
  users_data[index].idThread = nr_of_games;
  for (k = 0; k < users_data[index].nr_of_users; k++)
    write(users_data[index].users[k].socket_address, &
    startgame, sizeof(int)); // Informing the client the
    game is starting
     pthread_create(&th[nr_of_games], NULL, &treat, &
    users_data[index]);
          // /* Waiting for the shared game data to be
    copied in the thread before we empty it in the root.
    Replace with a mutex?*/
  sleep(1);
  /* Emptying 'game lobby' structure for that category */
  for (k = 0; k < users_data[index].nr_of_users; k++){
    users_data[index].users[k].flag = 0;
    users_data[index].users[j].socket_address = 0;
    memset(users_data[index].users[k].username, 0, sizeof(
    users_data[index].users[k].username));
     }
users_data[index].nr_of_users = 0;
}

```

– **Treat**
Calls the **game** function and passes to it as argument, the structure **local_game_gata**, which will contain the details of the game (Lobby data). After the game has finished it attempts to close the connection with all the Clients and detaches the thread.

```
static void *treat(void *data)
{
  struct game global_game = *((struct game *)data);
  struct game local_game_data = global_game;

  game(local_game_data);
```

```c
  pthread_detach(pthread_self());
  for (int i = 0; i < local_game_data.nr_of_users; i++)
    close(local_game_data.users[i].socket_address);
  return (NULL);
}
```

– **Game - Duo|Ffa**
  We create a **user** structure with the **username**, **socket** and **score** fields which stores the username, the score and the socket address of the user.

```c
struct user{
    char username[32];
    int socket;
    int score;
};
```

user_data is an array of size 2 - Duo|10- Ffa **user** structures that will store the game data for all the users.

```c
  struct user user_data[2];

```

The **multiple_order** array of size 4 will store the order in which the answers are sent to the Clients when the question type is multiple.

```c
 int multiple_order[4] = {0, 1, 2, 3};

```

The **boolean_order** array of size 2 will store the order in which the answers are sent to the Clients when the question type is boolean.

```c
 int boolean_order[2] = {0, 1};

```

The server generates the quiz questions.

```c
sprintf(query, "SELECT *FROM intrebari where category='%s'
    ORDER BY RANDOM() LIMIT %d;", local_game.category,
    nr_questions); // 10 questions

```

For each row returned by the query we save the **type** of the question, and the **question**.

```c
while (sqlite3_step(result) == SQLITE_ROW){
  strcpy(type, sqlite3_column_text(result, 1));
  strcpy(question, sqlite3_column_text(result, 2));

```

If the **type** is 'boolean' we save the 2 answers in **a1** and **a2**. Otherwise, if the **type** is 'multiple' we save the 4 answers in **a1**, **a2**,**a3** and **a4**. To randomize the order in which the Server sends the answers to the Clients, we'll use 2 arrays, **multiple]_order**[0,1,2,3] for multiple type questions and **boolean_order**[0,1] for boolean, question which will represent the order of the answers given to the Clients, and will be randomized in each iteration using the **shuffle** function. Characters a,b,c,d will be 'asigned' to the following indexes: a->0,b->1,c->2,d->c. The index of the value 0 in the **multiple_order** and **boolean_order** arrays will represent the correct answer. Some examples:

Indexes	0	**1**	2	3
multiple_order	1	**0**	2	3
Answers	a	**b**	c	d

Table 1: Multiple type, Correct answer: **b**

Indexes	0	1	2	**3**
multiple_order	1	3	2	**0**
Answers	a	b	c	**d**

Table 2: Multiple type , Correct answer: **d**

Indexes	0	**1**
boolean_order	1	**0**
Answers	a	**b**

Table 3: Boolean type, Correct answer: **b**

Indexes	**0**	1
boolean_order	**1**	0
Answers	**a**	b

Table 4: Boolean type, Correct answer: **a**

Boolean questions:

```
if (strcmp(type, "boolean") == 0){
      shuffle(boolean_order, 2);
      switch (boolean_order[0]){
        case 0:{
          strcpy(a1, sqlite3_column_text(result, 3));
          strcpy(a2, sqlite3_column_text(result, 4));
          break;
        }
        case 1:{
          strcpy(a1, sqlite3_column_text(result, 4));
          strcpy(a2, sqlite3_column_text(result, 3));
          }
      }
    }

```

Multiple questions:

```
if (strcmp(type, "multiple") == 0){
      shuffle(multiple_order, 4);
      switch (multiple_order[0]){
        case 0:{strcpy(a1, sqlite3_column_text(result, 3));
    break;}
        case 1:{strcpy(a1, sqlite3_column_text(result, 4));
    break;}
        case 2:{strcpy(a1, sqlite3_column_text(result, 5));
    break;}
```

```
        case 3:{strcpy(a1, sqlite3_column_text(result, 6));}
       }
      switch (multiple_order[1]){
      case 0:{strcpy(a2, sqlite3_column_text(result, 3));break
    ;}
      case 1:{strcpy(a2, sqlite3_column_text(result, 4));break
    ;}
      case 2:{strcpy(a2, sqlite3_column_text(result, 5));break
    ;}
      case 3:{strcpy(a2, sqlite3_column_text(result, 6));}
        }
      ......
  }
```

The Server iterates over the Clients‘ sockets, and if the **flag** of a Client is 1 (the Client has not disconnected during the game), it sends the question **type**, the **question** and 4 answers (**a1,a2,a3,a4**) for multiple type questions or 2 answers (**a1,a2**) for boolean type questions. If any **write** operaton to a Client‘s socket fails, it means that the Client has disconnected and we change his **flag** to 0, decrement the **nr_of_active_clients** variable and close the connection.

```
for (user_path = 0; user_path < local_game.nr_of_users;
    user_path++){
      if (local_game.users[user_path].flag == 1){ // Only
    attempt to send quizz data to users who haven‘t
    disconnected
        buffer_size = (int)strlen(type);
        if (write(local_game.users[user_path].socket_address,
    &buffer_size, sizeof(int)) <= 0){ // Sending type length
          close(local_game.users[user_path].socket_address);
          local_game.users[user_path].flag = 0; // User
    disconnected, we change the flag to 0
          nr_of_active_clients--;
          printf("User %s has left game %d\n", local_game.
    users[user_path].username, local_game.idThread);
          continue;
        }

        if (write(local_game.users[user_path].socket_address,
    &type, buffer_size) <= 0){ // Sending type
          ...
        }

        buffer_size = (int)strlen(question);
        if (write(local_game.users[user_path].socket_address,
    &buffer_size, sizeof(int)) <= 0){ // Sending question
    length
          ...
        }

        if (write(local_game.users[user_path].socket_address,
    &question, buffer_size) <= 0){ // Sending question
      ...
        }
        ......
```

```

```

The server pauses execution for 15 seconds and then attempts to read from each connected Client (**flag**=1) the answer. If any read operaton from a Client‘s socket fails, it means that the Client has disconnected and we change his **flag** to 0, decrement the **nr_of_active_clients** variable and close the connection.

```c
 sleep(15);
   for (user_path = 0; user_path < local_game.nr_of_users; user_path++){
     if (local_game.users[user_path].flag == 1){ // Only attempt to read the answer of users who haven‘t disconnected

       if (read(user_data[user_path].socket, &user_answer, sizeof(char)) <= 0){ // Receiving User answer
           close(local_game.users[user_path].socket_address);
           local_game.users[user_path].flag = 0; // user disconnected, we change the flag to 0
           nr_of_active_clients--;
           printf("User %s has left game %d\n", local_game.users[user_path].username, local_game.idThread);
           continue;
       }
  }

```

The server checks the answers of the Clients. If a Client‘s answer is correct it increments the score variable from the structure corresponding to his index.

```c
  if (strcmp(type, "boolean") == 0)
          switch (user_answer){
          case 'a':{
            if (boolean_order[0] == 0)
              user_data[user_path].score++;
            break;
               }
          case 'b':{
            if (boolean_order[1] == 0)
              user_data[user_path].score++;
                }
          }
        if (strcmp(type, "multiple") == 0)
          switch (user_answer){
          case 'a':{
            if (multiple_order[0] == 0)
              user_data[user_path].score++;
            break;
          }
          case 'b':{
            if (multiple_order[1] == 0)
              user_data[user_path].score++;
            break;
          }
          case 'c':{
            if (multiple_order[2] == 0)
              user_data[user_path].score++;
            break;
            }
          case 'd':{
            if (multiple_order[3] == 0)
              user_data[user_path].score++;
          }
          }

```

If **nr_of_active_clients** becomes 0 during the game, the server terminates the game.

```
    if (nr_of_active_clients == 0)
      break;

```

After the game has ended, the server finds the **nr_of_winners** in the game and for each connected Client, it sends his final score, the number of winners, and the usernames of the winners.

```
int max_score = user_data[0].score;
  for (user_path = 1; user_path < local_game.nr_of_users; user_path++){ // finding the index of the user with the highest score;
    if (user_data[user_path].score > max_score)
      max_score = user_data[user_path].score;

  int nr_of_winners = 0;
  for (user_path = 0; user_path < local_game.nr_of_users; user_path++)
    if (user_data[user_path].score == max_score)
      nr_of_winners++;
  int user_length;
  for (user_path = 0; user_path < local_game.nr_of_users; user_path++)
    if (local_game.users[user_path].flag == 1){ // only attempt to send the quiz results to the users who haven`t disconnected
      if (write(local_game.users[user_path].socket_address, &user_data[user_path].score, sizeof(int)) <= 0){ // Sending score of the user
    ...
      }

      if (write(local_game.users[user_path].socket_address, &nr_of_winners, sizeof(int)) <= 0){ // Sending the number of winners
      ...
      }
      int user_length;
      for (int k = 0; k < local_game.nr_of_users; k++){
        user_length = 0;
        if (user_data[k].score == max_score){
          if (write(local_game.users[user_path].socket_address
    , &user_data[k].score, sizeof(int)) <= 0){ // Sending the
    score of the winner
      ...
          }
          user_length = strlen(user_data[k].username);
          if (write(local_game.users[user_path].socket_address
    , &user_length, sizeof(int)) <= 0){ // Sending username
    length of the winner
      ...
          }
          if (write(local_game.users[user_path].socket_address
    , &user_data[k].username, user_length) <= 0){ // Sending
    username of the winner
      ...
          }
        }
      }

```

The server closes connection with the remaining Clients.

```
  for (user_path = 1; user_path < local_game.nr_of_users;
    user_path++){
    if (local_game.users[user_path].flag == 1)
      close(local_game.users[user_path].socket_address);
    }

```

It connects to the main server and sends the command '+' (**update_history**).

```
if ((dd = socket(AF_INET, SOCK_STREAM, 0)) == -1){
    perror("Error creating socket.\n");sleep(1);return;
  }
  server_management.sin_family = AF_INET;
  server_management.sin_addr.s_addr = inet_addr("
    192.168.174.128");
  server_management.sin_port = htons(main_server_port);
  if (connect(dd, (struct sockaddr *)&server_management,
    sizeof(struct sockaddr)) == -1){
    perror("Error connecting to the main server\n");sleep(1);
    return;
  }
  if (write(dd, &d, sizeof(char)) <= 0){ // Sending command to
     main server
    perror("Error sending command history.\n");sleep(1);close(
    dd);
    return;
  }

```

It sends the **type** of the game, the **number_of_users**, and for each user, his **username** and **score**.

```
if (write(dd, &game_type, size) <= 0) {// Sending type of game
     to the main server
    perror("Error sending game type\n"); sleep(1);close(dd);
    return;
```

```
  }
  if (write(dd, &local_game.nr_of_users, sizeof(local_game.nr_of_users)) <= 0){ // Sending the number of users for flexibility
    perror("Error sending number of users\n");sleep(1);close(dd);
    return;
  }
  for (user_path = 0; user_path < local_game.nr_of_users; user_path++){
    size = strlen(user_data[user_path].username);
    if (write(dd, &size, sizeof(int)) <= 0){ // Sending username length to the main server
      perror("Error sending user data\n");sleep(1);close(dd);
      return;
    }
    if (write(dd, &user_data[user_path].username, size) <= 0){ // Sending username
      perror("Error sending user data\n");sleep(1);close(dd);
      return;
    }
    if (write(dd, &user_data[user_path].score, sizeof(int)) <= 0){ // Sending score
      perror("Error sending user data\n");sleep(1); close(dd);
      return;
    }
  }

```

It waits to for the main server to confirm that all data has been read, and the connection is closed

```
int data_received_confirmation;
  if (read(dd, &data_received_confirmation, sizeof(int)) <= 0){
    printf("Error receiving confirmation from the main server\n");
    close(dd);return;
  }
  printf("The game %d has been added to history\n\n", local_game.idThread);
  close(dd);

```

4.3 Databases

The questions were taken from `https://opentdb.com/api_config.php` using the free api and **python**. We make a series of GET requests to the API‘s endpoint for each category, parse the response as JSON and the resulted data is appended to ‘intrebari’.json.

```
for category in range(9,33):
        url="https://opentdb.com/api.php?amount=100&category={category}".format(category=category)
        r=requests.get(url)
        data.append(r.json())

with open('intrebari.json',"w",encoding='utf-8') as file:
        json.dump(data,file,indent=4, ensure_ascii=False)

```

Then we open the ‘intrebari.json’ file in read mode and read the contents of the file into a variable called **intrebari**.We iterate over the questions and construct a SQL insett statement string with the exctracted information and append the string to the list **queries**. The queries list is then printed to a file and the contents of the file are copy-pasted and ran on the ‘intrebari.db’ database in the linux terminal.

```
with open('intrebari.json',encoding='utf-8') as file:
        intrebari = json.load(file)

for set in intrebari:
    for index in range(len(set['results'])):
        category=set['results'][index]['category']
        type=set['results'][index]['type']
        question=set['results'][index]['question']
        correct_asnwer=set['results'][index]['correct_answer']
        if type=='multiple':
           a2=set['results'][index]['incorrect_answers'][0]
           a3=set['results'][index]['incorrect_answers'][1]
           a4=set['results'][index]['incorrect_answers'][2]
        else:
         a2=a2=set['results'][index]['incorrect_answers'][0]
            a3=''
            a4=''
        query="INSERT INTO intrebari (category,type,question,
    correct_answer,a2,a3,a4) values('{c}','{t}','{q}','{a}','{
    a_2}','{a_3}','{a_4}');".format(c=category,t=type,q=
    question,a=correct_asnwer,a_2=a2,a_3=a3,a_4=a4)

        queries.append(query)
file = open("Inserts.txt", "a",encoding='utf-8')  # append
    mode
for query in queries:
    file.write(query)
    file.write("\n")

```

There will be in total 21 categories(General Knowledge, Books,Film, Music, Television, Video Games, Board Games, Science & Nature, Science: Computers, Science: Mathematics, Mythology, Sports, Geography, History, Politics, Celebrities, Animals, Vehicles, Comics, Japanesse Anime & Manga, Cartoon & Animations) and 50 questions for each category(1050 questions in total). The questions are separated in two groups: **boolean**, which means the answer is either true or false and **multiple**, where each question has 4 possible answers. The correct answer for each question will be stored in the **correct_answer** field. **a2**,**a3**,**a4** will represent the wrong answers. Note: **a3** and **a4** will be a null string for boolean questions. QuizGame.db will contain 2 tables: **users** and **history**. The table users has 5 fields. **id** is the primary key and is an unisgned int value representing the id of the user in the order of their account creation. **history** is a comma seperated string where each gameid of the games that the user played is stored. **logged** is a flag value, which can be 0(user is not logged in) or 1(user is logged in). The table history has 4 fields. **gameid** is the primary key and is an unsigned int value representing the id of the game. **gametype** is the type of the game which can be solo,duo or ffa. **users** and **scores** are comma separated strings which contains the id and the score of the users who played in that game.

5 Conclusion

In conclusion, the implementation of the quizGame with 1 Client and multiple Servers has been a complex and rewarding project. The use of multithreaded TCP servers for all of the servers allowed for efficient handling of multiple connections, improving the performance and scalability of the game, and creating a dynamic and engaging experience for the users. Creating the **intrebari** database with the API from `https://opentdb.com/api_config.php` allowed for the inclusion of a wide range of questions from various categories, ensuring that the game is engaging and informative for players. Overall, this project has demonstrated the potential of using multithreaded servers and databases in developing a quiz game and the importance of effective communication and data management in its implementation.

5.1 Features that could be added in the future

- A method to prevent bad actors from overfilling the database by creating hundreds of thousands of accounts(captcha, email registration and confirmation, ip blacklist, etc)
- Be able to play in teams
- A GUI
- Correct answers/Wrong answers (similar to k/d) and quiz win/lose ratios for each user
- Waiting queue if the number of max threads is exceeded
- Password reset/Forgot password method
- In game chat between players
- Players to be able to rejoin the game if they disconnected

- Different levels of difficulty
- Custom number of rounds per game.
- Automatically add questions to the database with the Trivia API
- Custom games
- Single elimination tournament
- Users to be able to choose a user to play against
- Users to be logged out after a number of minutes of inactivity
- Delete inactive accounts after a number of months

6 Bibliography

- https://dbdiagram.io/d
- https://app.creately.com/d/AeCcAgn4bYo/view
- http://lucamozzo.altervista.org/using-sqlite-in-c-linux/?doing_wp_cron=1669385353.5993719100952148437500
- https://zetcode.com/db/sqlitec/
- https://www.sqlitetutorial.net/sqlite-insert/
- https://www.programiz.com/c-programming/library-function/string.h/strcat
- https://www.techonthenet.com/sqlite/tables/alter_table.php
- https://stackoverflow.com/questions/37751325/how-to-check-if-record-is-present-in-sqli
- https://www.sqlitetutorial.net/sqlite-delete/
- https://pubs.opengroup.org/onlinepubs/009696699/functions/write.html
- https://www.sqlitetutorial.net/sqlite-insert/
- https://www.codingeek.com/tutorials/c-programming/2d-character-array-string-array-decl
- https://www.sqlitetutorial.net/sqlite-update/
- https://stackoverflow.com/questions/8107826/proper-way-to-empty-a-c-string
- https://stackoverflow.com/questions/9146395/reset-c-int-array-to-zero-the-fastest-way
- https://stackoverflow.com/questions/7751342/how-to-execute-make-file

- https://stackoverflow.com/questions/8707857/error-expected-declaration-or-statement-at-
- https://www.linuxtrainingacademy.com/determine-public-ip-address-command-line-curl/
- https://www.sqlite.org/datatype3.html
- https://stackoverflow.com/questions/7012182/retrieve-data-from-sqlite-with-c
- https://discover.cs.ucsb.edu/commonerrors/error/2012.xml
- https://profs.info.uaic.ro/~computernetworks/files/NetEx/S9/servTcpCSel.
  c
- https://man7.org/linux/man-pages/man2/read.2.html
- https://profs.info.uaic.ro/~computernetworks/files/NetEx/S9/Makefile
- https://profs.info.uaic.ro/~computernetworks/files/NetEx/S9/cliTcp.
  c
- https://stackoverflow.com/questions/17271576/clear-screen-in-c-and-c-on-unix-based-sys
- https://stackoverflow.com/questions/18337407/saving-utf-8-texts-with-json-dumps-as-utf
- https://stackoverflow.com/questions/27795767/optional-arguments-in-c-function
- https://www.youtube.com/watch?v=Y6pFtgRdUts
- https://stackoverflow.com/questions/6014055/how-is-socket-connection-being-handled-in-
- https://www.ezinfo.ro/X/sirurichr/functii/strtok.html
- https://stackoverflow.com/questions/26303289/c-socket-consecutive-read-write-what-happ
- https://stackoverflow.com/questions/13538480/sqlite-query-how-do-i-retrieve-multiple-c
- https://stackoverflow.com/questions/4754607/how-would-i-check-if-the-client-is-no-long
- https://www.kodeco.com/6620276-sqlite-with-swift-tutorial-getting-started
- https://stackoverflow.com/questions/283375/detecting-tcp-client-disconnect
- https://profs.info.uaic.ro/~computernetworks/files/NetEx/S12/
  ServerConcThread/cliTcpNr.c
- https://profs.info.uaic.ro/~computernetworks/files/NetEx/S12/
  ServerConcThread/servTcpConcTh2.c
- https://stackoverflow.com/questions/13768182/what-could-cause-bad-file-descriptor-in-m
- https://stackoverflow.com/questions/4360653/bad-file-descriptor-realted-to-threads
- https://www.tutorialspoint.com/c_standard_library/c_function_
  strchr.htm

- https://www.edureka.co/community/164286/what-does-collect2-error-ld-returned-1-exit-st
- https://stackoverflow.com/questions/43847195/how-to-check-certain-chars-in-string-in-c
- https://www.tutorialspoint.com/c_standard_library/c_function_
  sprintf.htm
- https://www.programiz.com/c-programming/library-function/string.
  h/strcmp
- https://stackoverflow.com/questions/64514814/undefined-reference-to-log10
- https://www.javatpoint.com/count-the-number-of-digits-in-c
- https://stackoverflow.com/questions/7780809/is-it-possible-to-print-out-only-a-certain
- https://www.reddit.com/r/C_Programming/comments/5ara6b/using_
  apis_in_c/
- https://curl.se/libcurl/c/libcurl-tutorial.html
- https://stackoverflow.com/questions/16476196/undefined-reference-to-curl-global-init-c
- https://www.youtube.com/watch?v=nbTaHEocCuo
- https://github.com/DaveGamble/cJSON
- https://stackoverflow.com/questions/14011581/how-to-send-a-message-from-server-to-all-
- https://stackoverflow.com/questions/2279706/select-random-row-from-a-sqlite-table
- https://stackoverflow.com/questions/12027708/c-socket-requests-getting-buffered
- https://stackoverflow.com/questions/7226603/timeout-function
- https://stackoverflow.com/questions/21197977/how-can-i-prevent-scanf-to-wait-forever-f
- https://linux.die.net/man/2/select
- https://stackoverflow.com/questions/20812749/why-fflushstdin-does-not-remove-buffer-st
- https://stackoverflow.com/questions/2979209/using-fflushstdin
- https://stackoverflow.com/questions/10938882/how-can-i-flush-unread-data-from-a-tty-in
- https://stackoverflow.com/questions/6127503/shuffle-array-in-c
- https://stackoverflow.com/questions/4060772/sqlite-concurrent-access
- https://stackoverflow.com/questions/8257714/how-to-convert-an-int-to-string-in-c
- https://www.sqlite.org/threadsafe.html
- https://www.geeksforgeeks.org/different-ways-to-initialize-all-members-of-an-array-to-

- https://stackoverflow.com/questions/17096990/why-use-bzero-over-memset
- https://www.programiz.com/c-programming/c-break-continue-statement
- https://stackoverflow.com/questions/73523739/how-do-i-exit-a-loop-without-using-a-brea
- https://www.mkssoftware.com/docs/man3/select.3.asp
- https://stackoverflow.com/questions/3469567/broken-pipe-error
- https://stackoverflow.com/questions/108183/how-to-prevent-sigpipes-or-handle-them-prop
- https://stackoverflow.com/questions/6824265/sigpipe-broken-pipe