Connect4 server client application

Serban Mihai

December 2020

Abstract

Implementing a client/server application for the Connect4 game in c using sockets and the TCP/ip protocol. The server handles all of the game logic and the scoring system while the clients displays the game board (grid) to the player and takes all the user inputs.

1 Introduction

Connect Four (also known as Four Up, Plot Four, Find Four, Four in a Row, Four in a Line, Drop Four, and Gravitrips in the Soviet Union) is a two-player connection board game, in which the players choose a color and then take turns dropping colored discs into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of four of one's own discs. Connect Four is a solved game. The first player can always win by playing the right moves.
A implementation of the classic childhood game Connect4 using socket programming implemented with the c programming language under linux operating system.
The game is implemented using a concurent server in which 2 clients are waited to connect to the server before a game can start. The concurency of the server is obtained by using the fork() function.

2 Used technologies

2.1 TCP

Because Connect4 is a turned based game I choosed TCP over UDP even though UDP is faster at transmiting packages because I send minimal amount of data between the server and the client.(only the messages which have the fixed length five, the board a 2 dimensional array of type char and the move of a player which is a integer)

A TCP protocol is much more reliable and even thought it would stall if a

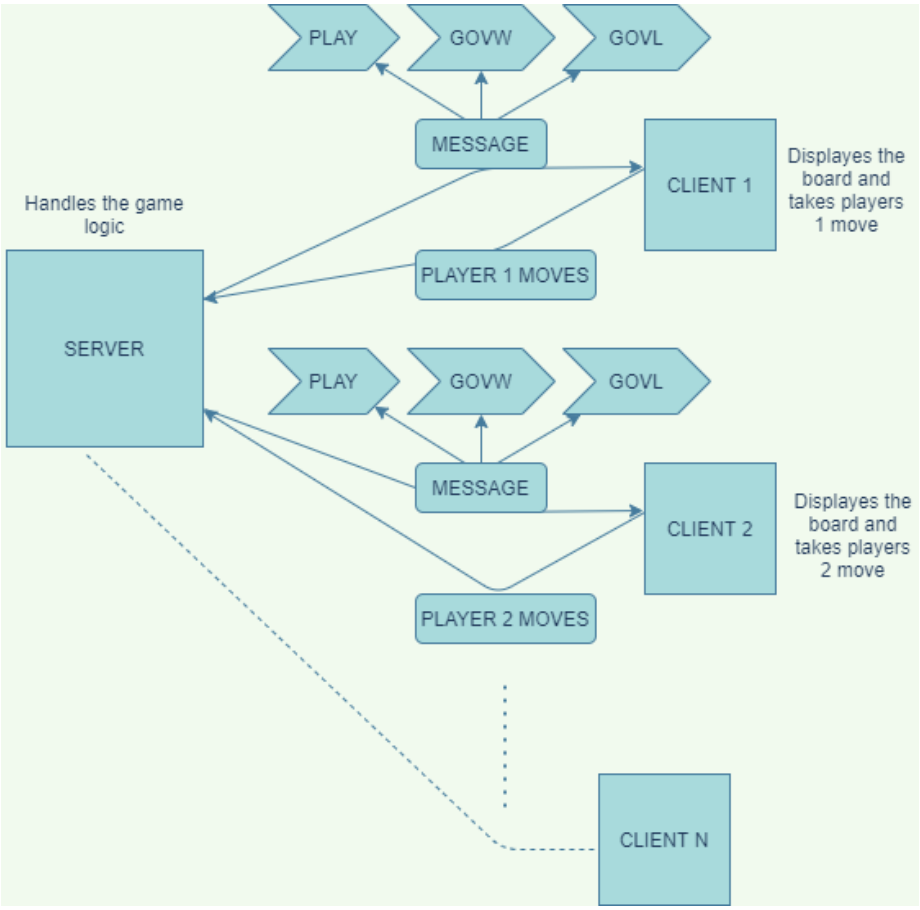
package will be resended I don't have to adress that problem anymore because the protocol handles it for me automatically.

2.2 fork()

The concurency of my server is obtained by using a fork call. That allows me to move the game logic to the child process and to let the parent process handle the accepting of the clients.

3 Application architecture

3.1 Application diagram



3.2 Application protocol

The program waits for to clients to connect in order to start the playing session.

Once we have two players conneted the game starts in a recursive function called servicePlayers that stops when one of the players decides that he doesn't want to play anymore.

First it is checked if the game is over in which case we increment the score for the winner and we send the relevant messages for each client (the govw message to the winner and the govl message to the loser, both of them followed by score), then we aslo ask both players if they want a replay. If both of them asnwer affirmative then we call again the servicePlayer function, otherwise we close the connection to both clients and exit the servicePlayers function.

Then we start the actual game by sending the table to the first player and wait for the position that he wants to take (we do this for both players separately with a sleep() call between players), also we update the game board wiht the position that the player desires.

3.3 Server

The server must start running before any client, and goes into an infinite loop to wait for clients

When the server gets a client, it waits for another client (two players are needed)

When the server gets the other client (now two clients), it forks and, let the child process take care of these two clients (players) in a separate function, called servicePlayers, while the parent process goes back to wait for the next two clients (players). Note that the file descriptors of the two connected sockets (one for each client) are passed to the function servicePlayers.

The server handles the game logic and the score of both players for each session in the servicePlayers function.

3.4 Client

The clients establish the connection with the server and then waits in a infinite loop (that ends when the game over signal it is received) for messages from the server.

The main purpose of the client is to display the board to the player and to take the next move of the player. All the positions that the players what to make would be validated directly in the client before sending them to the server.

The client waits for 3 kind of messages from the server:

1. The play message.
2. The govw message (game over you won).
3. The govl message (game over you lost)

Each meesage triggers different reaction in the client respectively with the game logic.

1) When we recieve the play message follow the steps:

- We read the game board from the server.
- Then we display the table to the player.
- We ask the player for the collum that he wants to move in.
- We transale the collum to the actual possition in the 2D array and then we send it to the server

2) When we recieve the govw message:

- We tell the player that he won.
- We read the score board form the server and we display it to the player.
- We ask the player if he want to replay and send the answer back to the server.

3) When we recieve the govl messgae we follow the same steps as in hte govw but this time we say to the player that he lost.

4 Implementation details

4.1 Server

```c
int main(int argc, char *argv[])
{
    int sd, player1, player2, portNumber;
    socklen_t len;
    struct sockaddr_in servAdd;
    char table[8][15];
    initialize_table(table);

    fprintf(stderr, "Server IP : ");
    //to display the ip address
    system("hostname -I");
    fprintf(stderr, "************************************\n");
    if (argc != 2)
    {
        printf("Call model: %s <Port #>\n", argv[0]);
        exit(0);
    }
    // Creating the TCP Socket
    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        fprintf(stderr, "Cannot create socket\n");
        exit(1);
    }
    //Prepare the sockaddr_in structure
    servAdd.sin_family = AF_INET;                   //setting domain to run over the internet
    servAdd.sin_addr.s_addr = htonl(INADDR_ANY);    //host to network conversion of multibyte integer for long
    sscanf(argv[1], "%d", &portNumber);             //formating the Port Number
    servAdd.sin_port = htons((uint16_t)portNumber); //host to network conversion of multibyte integer for short

    // Binding the IP address with Port Number
    bind(sd, (struct sockaddr *)&servAdd, sizeof(servAdd));
    // Put the server in passive mode and wait for server to accept the player
    listen(sd, 5);

    while (1)
    {
        // block until player 1 approach
        player1 = accept(sd, (struct sockaddr *)NULL, NULL);
        printf("Got Player\n");
        // block until player 2 approach
        player2 = accept(sd, (struct sockaddr *)NULL, NULL);
        printf("Got Player\n");
        //let the fork handle the game
        if (!fork())
        {
            servicePlayers(player1, player2, table);
        }

        close(player1);
        close(player2);
    }
}
```

In the first part we prepare the data structures that we are going to use for the server, client and for the game itself. We initialize the play board to be empty by calling the initialize_table() function
Then we bind the server socket with the ip and the port values
We put the socket to listen to incoming clients

In a infinite loop we will do two consecutive accepts that will be blocking until we will have both players connected to the server
After we obtain the players each one with it's one descriptor we fork and let the child process handle the game session
In the forked process we call the servicePlayers function that takes as arguments the file descriptors for the clients and the game board and the scores for both players. This function handles all the game logic

4.2 Client

```c
int main(int argc, char *argv[])
{
    char message[6]; //to display the message
    char table[8][15];
    int nr;
    int server, portNumber;
    socklen_t len;
    struct sockaddr_in servAdd;
    if (argc != 3)
    {
        printf("Call model:%s <IP> <Port#>\n", argv[0]);
        exit(0);
    }
    //creating and checking if socket is available
    if ((server = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        fprintf(stderr, "Cannot create socket\n");
        exit(1);
    }
    //setting domain to run over the internet
    servAdd.sin_family = AF_INET;
    //formating the Port Number
    sscanf(argv[2], "%d", &portNumber);
    //host to network conversion of multibyte integer
    servAdd.sin_port = htons((uint16_t)portNumber);
    //converts an address from presentation to network format
    if (inet_pton(AF_INET, argv[1], &servAdd.sin_addr) < 0)
    {
        fprintf(stderr, " inet_pton() has failed\n");
        exit(2);
    }
    // Establishing connection with TCP server
    if (connect(server, (struct sockaddr *)&servAdd, sizeof(servAdd)) < 0)
    { //check if connection is created
        fprintf(stderr, "connect() has failed, exiting\n");
        exit(3);
    }
```

```c
while (1)
{

    if (read(server, message, 5) < 0)
    {
        fprintf(stderr, "read() error\n");
        printf("ERROR: I can't read");
        close(server);
        exit(3);
    }
    if (strcmp(message, "play") == 0)
    {
        int colum;
        system("clear");
        printf("It is your turn\n");
        if (read(server, table, 120) < 0)
        {
            printf("ERROR: I can't read the table from the server");
            exit(0);
        }
        display_table(table);
        //I want to get a valid position from the user
        int validPosition = 0;
        while (validPosition == 0)
        {
            char rsp[10];
            printf("Give me the colum: ");
            fgets(rsp, 10, stdin);
            colum = atoi(rsp);
            colum = translate_position(colum);
            fflush(stdout);
            printf("colum postiion translated : %d \n", colum);
            if (table[0][colum] == ' ')
            {
                validPosition = 1;
            }
        }
        write(server, &colum, sizeof(int));
        fflush(stdout);
    }
    else if (strcmp(message, "govw") == 0)
    {
        printf("I won the game\n");
        close(server);
        exit(0);
    }
    else if (strcmp(message, "govl") == 0)
    {
        printf("I lost the game\n");
        close(server);
        exit(0);
    }
}
```

In the first part we establish the connection with the server
Then we enter a infinite loop in which we wait for different messages from the server
Case 1: we get the play message

We wait for the board then we ask the player for the next move, we check if the entered move is available (that collum most not be full already), we send the move to the server
Case 2: we get the govw message
govw message means that we won the game so we let the player know that he won and break the infinite loop and the connection with the server (in the future there will be a option to replay)
Case 3: we get the govl message
govl message means that the player lost the game, the steps are identical as in the case 2.

References

[1] Computer Netwrorks Course page UAIC computer science faculty
`https://profs.info.uaic.ro/~computernetworks/index.php`

[2] Beej's Guide to Unix IPC
`http://beej.us/guide/bgipc/html/multi/index.html`

[3] Beej's Guide to Network Programming
`https://beej.us/guide/bgnet/html//index.html`

[4] System programing - networking
`https://github.com/angrave/SystemProgramming/wiki#8-networking`

[5] Connect four game
`https://en.wikipedia.org/wiki/Connect_Four`

[6] Zombie processes prevention
`https://www.geeksforgeeks.org/zombie-processes-prevention/`