Computer Networks Homework 2 - Project Documentation
Pheasant (C) - Tudosa Delia

Introduction
Pheasant is a computer implementation of the popular game of the same name. A game goes the following : A player chooses a letter. The next player will enter a valid word that starts with the letter the first player chose. This player will pass the last two letters onto the next player, who will enter a word starting with these two letters, passing the last two to the next one. If a player makes a mistake (word invalid / doesn’t start with given letters), he is kicked out of the session and the game is continued by the remaining players. The game ends when only one player remains, who will be declared winner.

In the project’s requirements, an advice regarding the player count in a game is given (… to be given in a configuration file in the server). Another method of implementing proper games is to have an intermediate section between connecting to the server and playing the game, represented by a lobbying section (matchmaking). In here, players can create/join lobbies. Once a lobby is ready, the game can start.

The client will be interacted with in a command line interface, allowing users to issue commands before a game’s start so that they can see active lobbies, create / join a lobby, quit a lobby / game. Upon the game’s start, only word inputs (and the first letter input) will be required from the user. Upon a game’s finish/loss, the user will be returned to the matchmaking section.

Technologies Utilized
A series of libraries present in the C Linux compiler collection have been used :
- In order to implement proper synchronous communication, especially required by the game part of the project, TCP-IP sockets have been used, over internet protocol version 4.
- In order to implement parallel sessions of the game, and also to properly implement matchmaking, multithreading has been used, present in the pthread library. Upon initial connection to the server, a client will be allocated a thread. Upon a game start, all clients in the same lobby will use a newly created game thread.
- In order to not corrupt the active lobbies list when a new lobby is created / closed or a game is finished, critical section locking is a must. Mutex locks have been used, present in the pthread library used in multithreading.
- C preprocessor macros were used in maximising code readability and reducing boilerplate from read/write instructions

Architecture and Concepts

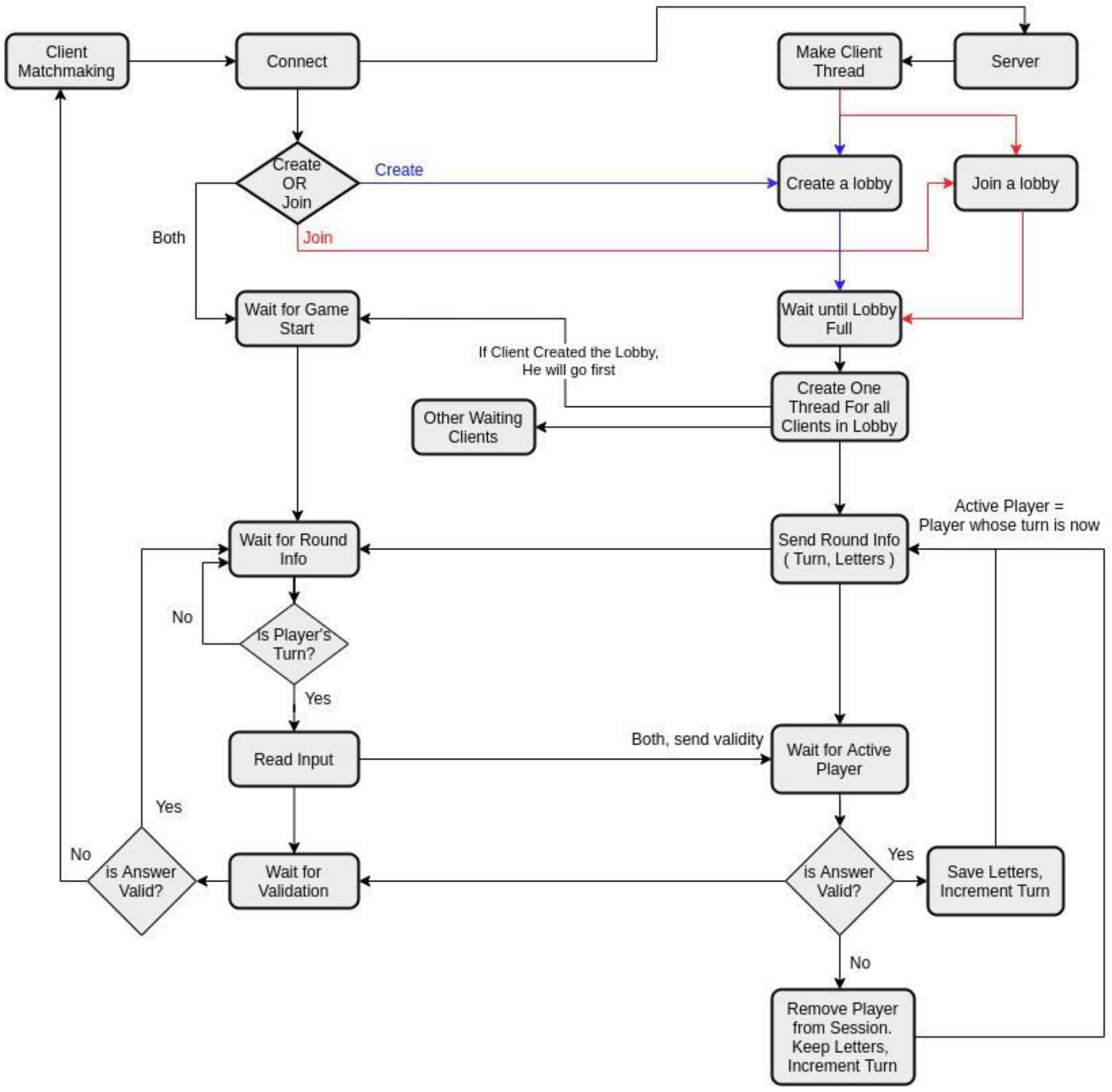
In addition to the game’s architecture, which is iterative through each client, a matchmaking section was implemented, working as follows :

1. An user will create a lobby. The server will allocate a lobby resource
2. The said user will be told to wait
3. The server thread will wait until enough clients have joined the created lobby, at which point it will signal the creator that the game is ready to start.
4. Upon the creator’s start command, all of the users present in the lobby get ‘redirected’ to a new thread, that will act as the game controller, iterating through users

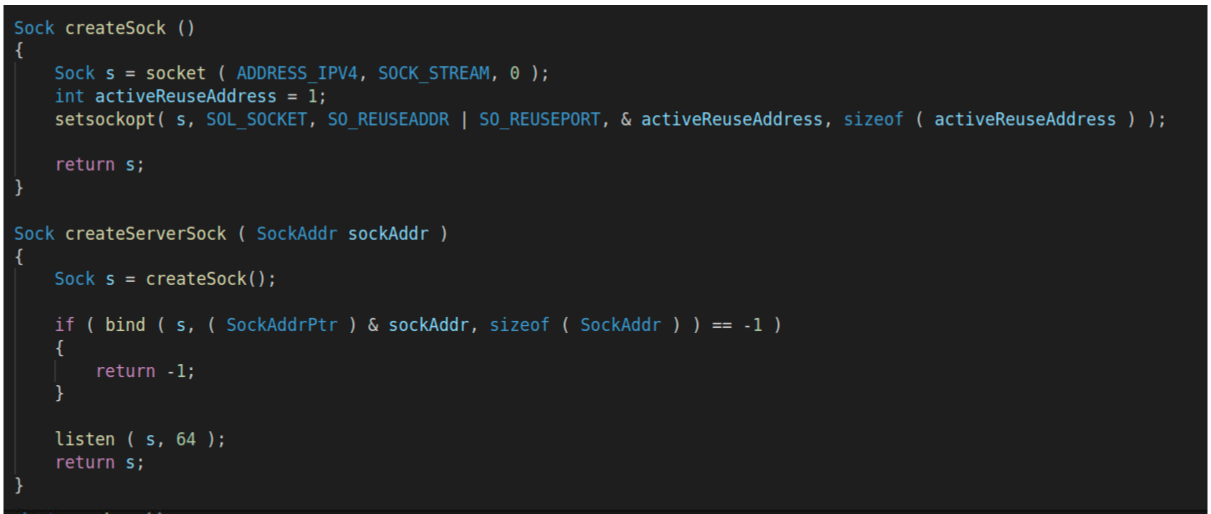
OR

1. An user will join a lobby. The server will add the user to that lobby
2. The said user will be told to wait
3. Upon the lobby creator’s start command, the user will be redirected to the game thread.

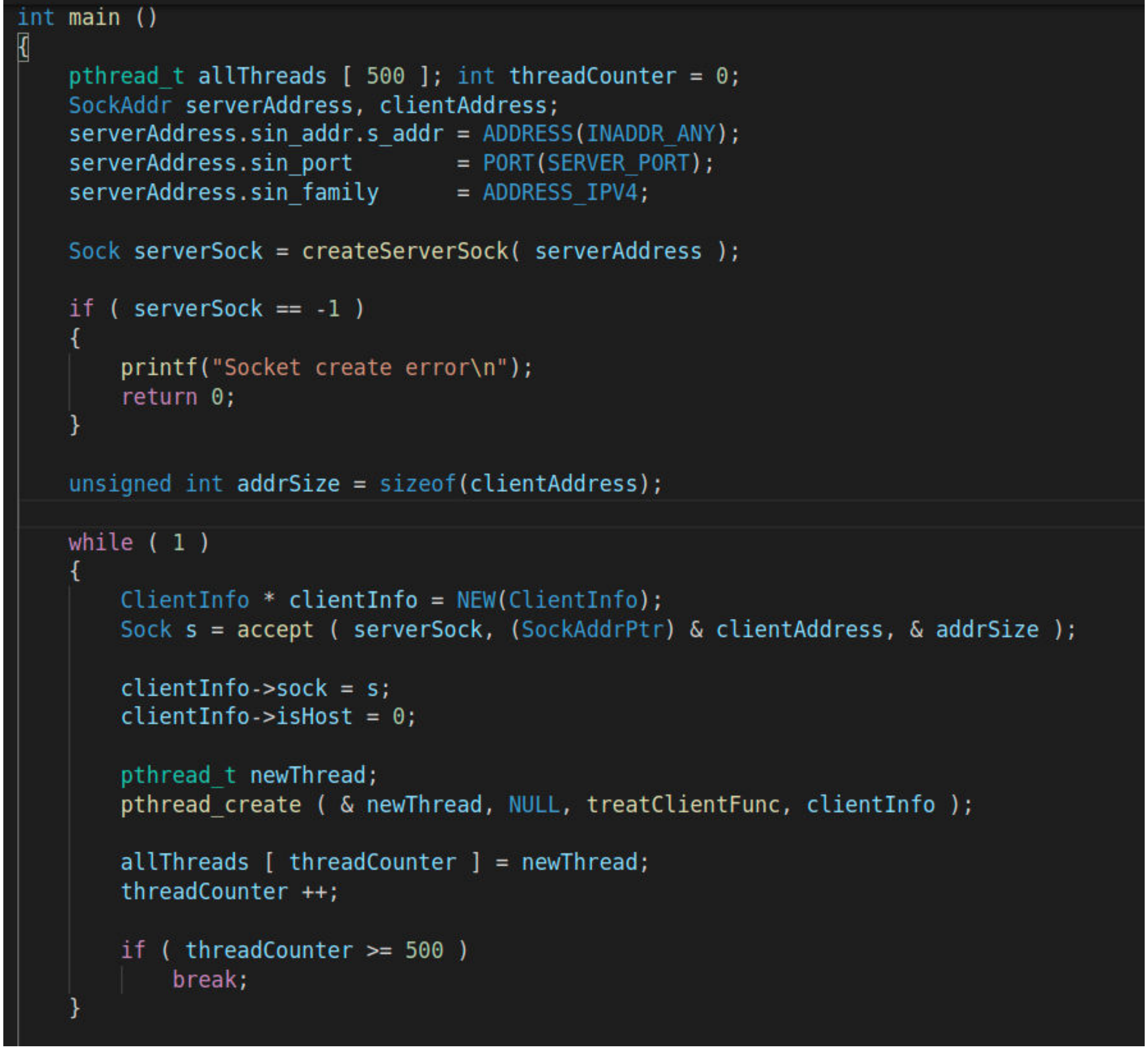
Architecture Diagram :



Source Code



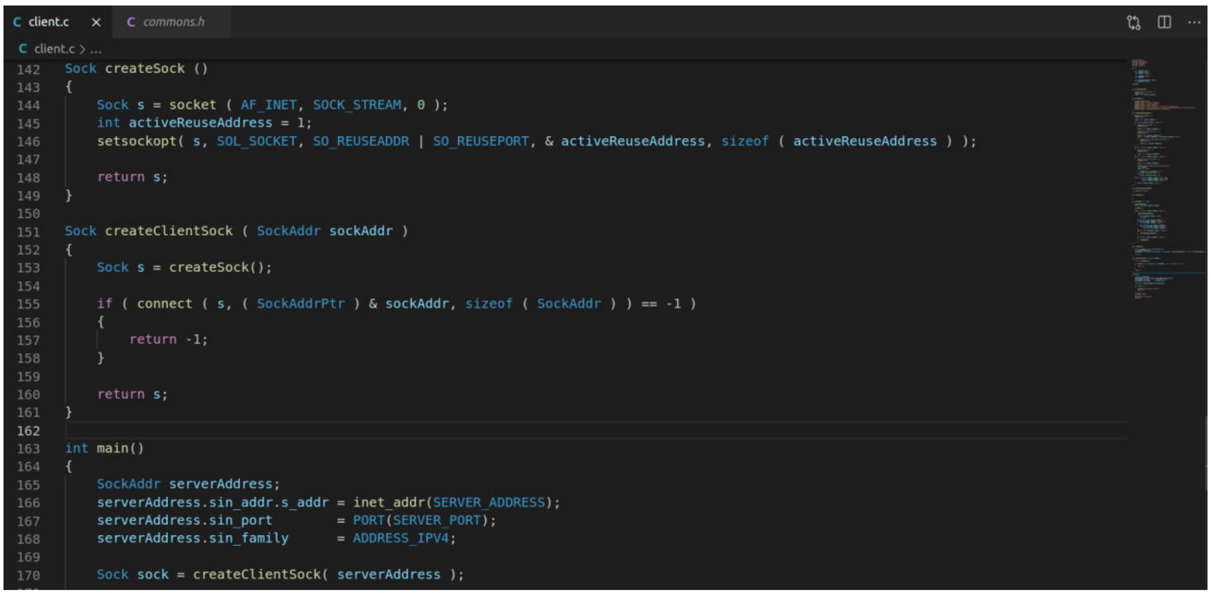
```c
Sock createSock ()
{
    Sock s = socket ( ADDRESS_IPV4, SOCK_STREAM, 0 );
    int activeReuseAddress = 1;
    setsockopt( s, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, & activeReuseAddress, sizeof ( activeReuseAddress ) );

    return s;
}

Sock createServerSock ( SockAddr sockAddr )
{
    Sock s = createSock();

    if ( bind ( s, ( SockAddrPtr ) & sockAddr, sizeof ( SockAddr ) ) == -1 )
    {
        return -1;
    }

    listen ( s, 64 );
    return s;
}
```



```c
int main ()
{
    pthread_t allThreads [ 500 ]; int threadCounter = 0;
    SockAddr serverAddress, clientAddress;
    serverAddress.sin_addr.s_addr = ADDRESS(INADDR_ANY);
    serverAddress.sin_port        = PORT(SERVER_PORT);
    serverAddress.sin_family      = ADDRESS_IPV4;

    Sock serverSock = createServerSock( serverAddress );

    if ( serverSock == -1 )
    {
        printf("Socket create error\n");
        return 0;
    }

    unsigned int addrSize = sizeof(clientAddress);

    while ( 1 )
    {
        ClientInfo * clientInfo = NEW(ClientInfo);
        Sock s = accept ( serverSock, (SockAddrPtr) & clientAddress, & addrSize );

        clientInfo->sock = s;
        clientInfo->isHost = 0;

        pthread_t newThread;
        pthread_create ( & newThread, NULL, treatClientFunc, clientInfo );

        allThreads [ threadCounter ] = newThread;
        threadCounter ++;

        if ( threadCounter >= 500 )
            break;
    }
```

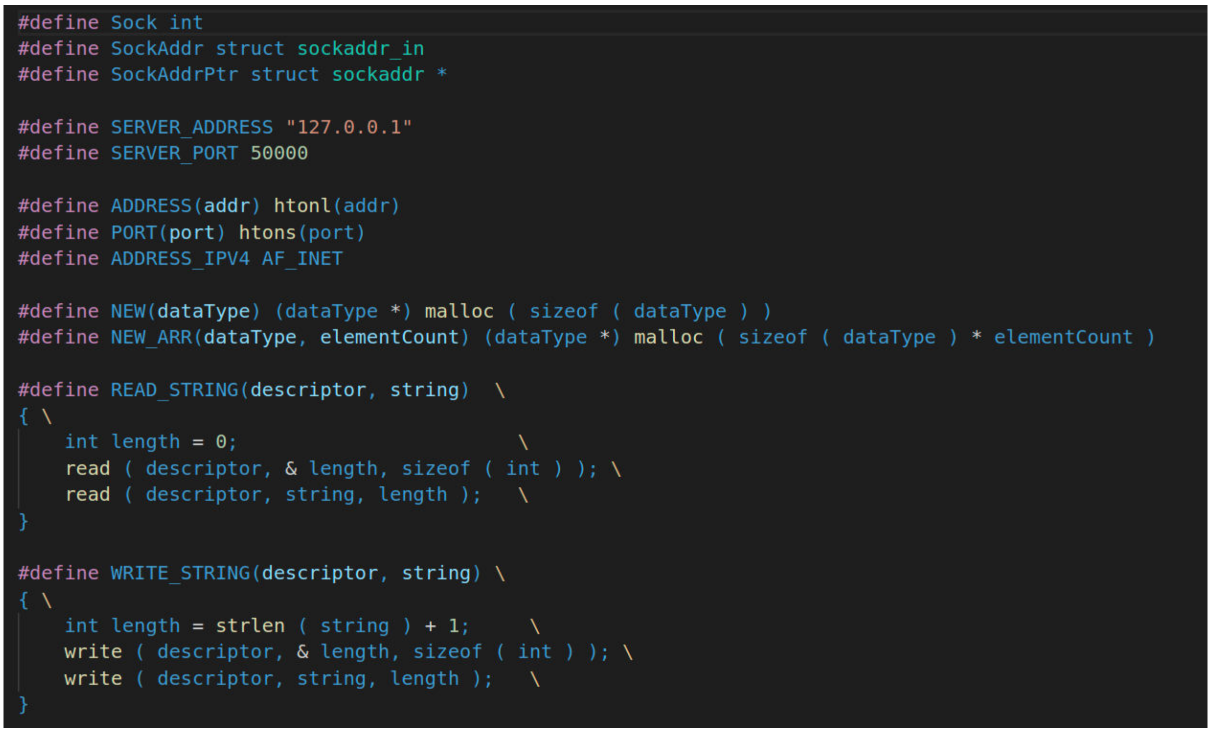
Above, server initialisation

Client Initialization



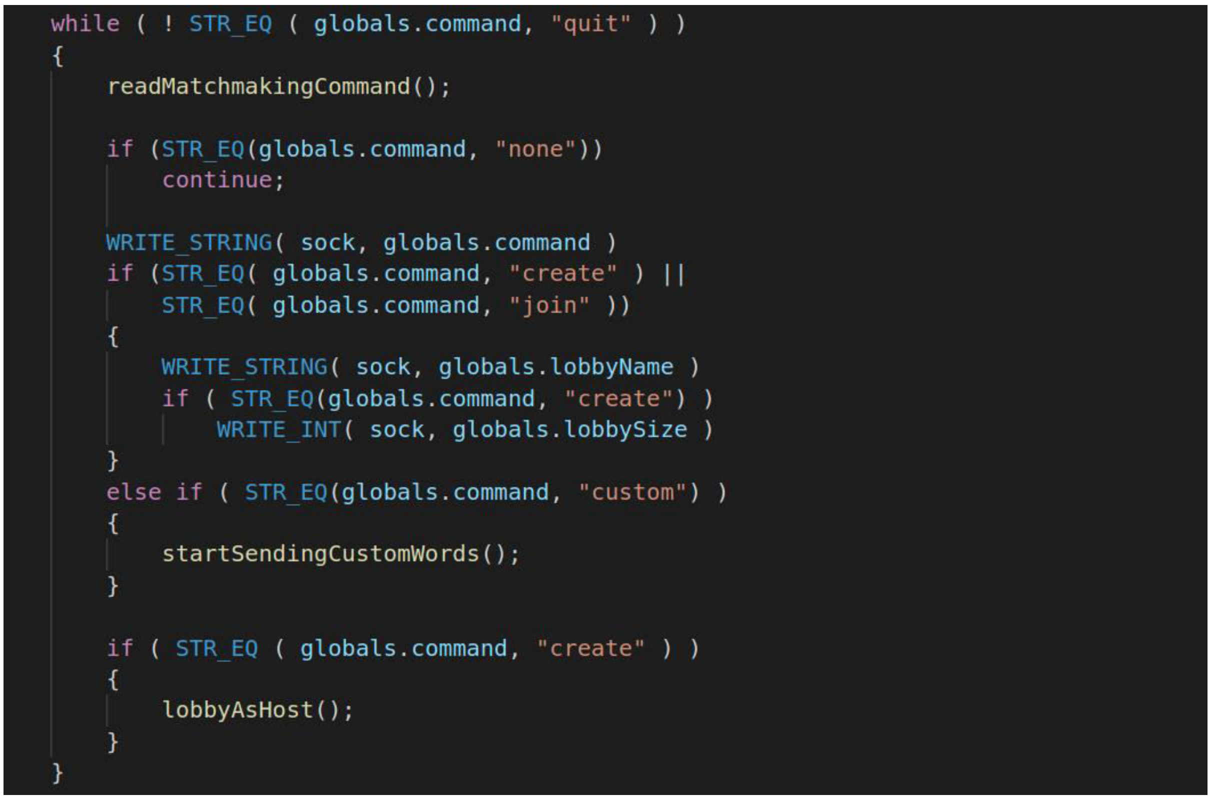
```c
Sock createSock ()
{
    Sock s = socket ( AF_INET, SOCK_STREAM, 0 );
    int activeReuseAddress = 1;
    setsockopt( s, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, & activeReuseAddress, sizeof ( activeReuseAddress ) );

    return s;
}

Sock createClientSock ( SockAddr sockAddr )
{
    Sock s = createSock();

    if ( connect ( s, ( SockAddrPtr ) & sockAddr, sizeof ( SockAddr ) ) == -1 )
    {
        return -1;
    }

    return s;
}

int main()
{
    SockAddr serverAddress;
    serverAddress.sin_addr.s_addr = inet_addr(SERVER_ADDRESS);
    serverAddress.sin_port        = PORT(SERVER_PORT);
    serverAddress.sin_family      = ADDRESS_IPV4;

    Sock sock = createClientSock( serverAddress );
```

Macros used



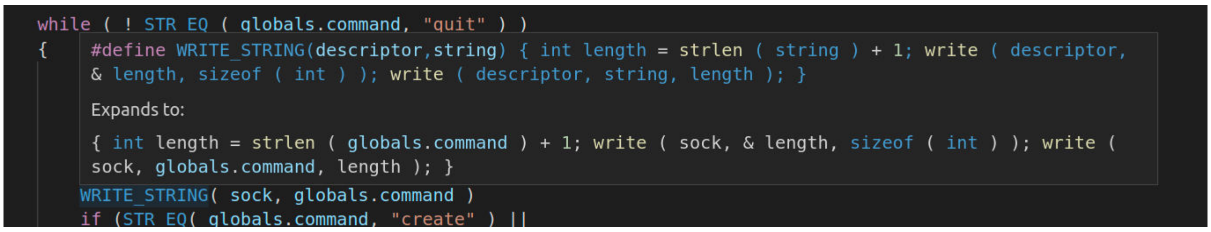
```c
#define Sock int
#define SockAddr struct sockaddr_in
#define SockAddrPtr struct sockaddr *

#define SERVER_ADDRESS "127.0.0.1"
#define SERVER_PORT 50000

#define ADDRESS(addr) htonl(addr)
#define PORT(port) htons(port)
#define ADDRESS_IPV4 AF_INET

#define NEW(dataType) (dataType *) malloc ( sizeof ( dataType ) )
#define NEW_ARR(dataType, elementCount) (dataType *) malloc ( sizeof ( dataType ) * elementCount )

#define READ_STRING(descriptor, string)  \
{ \
    int length = 0;                      \
    read ( descriptor, & length, sizeof ( int ) ); \
    read ( descriptor, string, length );   \
}

#define WRITE_STRING(descriptor, string) \
{ \
    int length = strlen ( string ) + 1;      \
    write ( descriptor, & length, sizeof ( int ) ); \
    write ( descriptor, string, length );    \
}
```

Client Matchmaking section:



```c
while ( ! STR_EQ ( globals.command, "quit" ) )
{
    readMatchmakingCommand();

    if (STR_EQ(globals.command, "none"))
        continue;

    WRITE_STRING( sock, globals.command )
    if (STR_EQ( globals.command, "create" ) ||
        STR_EQ( globals.command, "join" ))
    {
        WRITE_STRING( sock, globals.lobbyName )
        if ( STR_EQ(globals.command, "create") )
            WRITE_INT( sock, globals.lobbySize )
    }
    else if ( STR_EQ(globals.command, "custom") )
    {
        startSendingCustomWords();
    }

    if ( STR_EQ ( globals.command, "create" ) )
    {
        lobbyAsHost();
    }
}
```

Macros in action

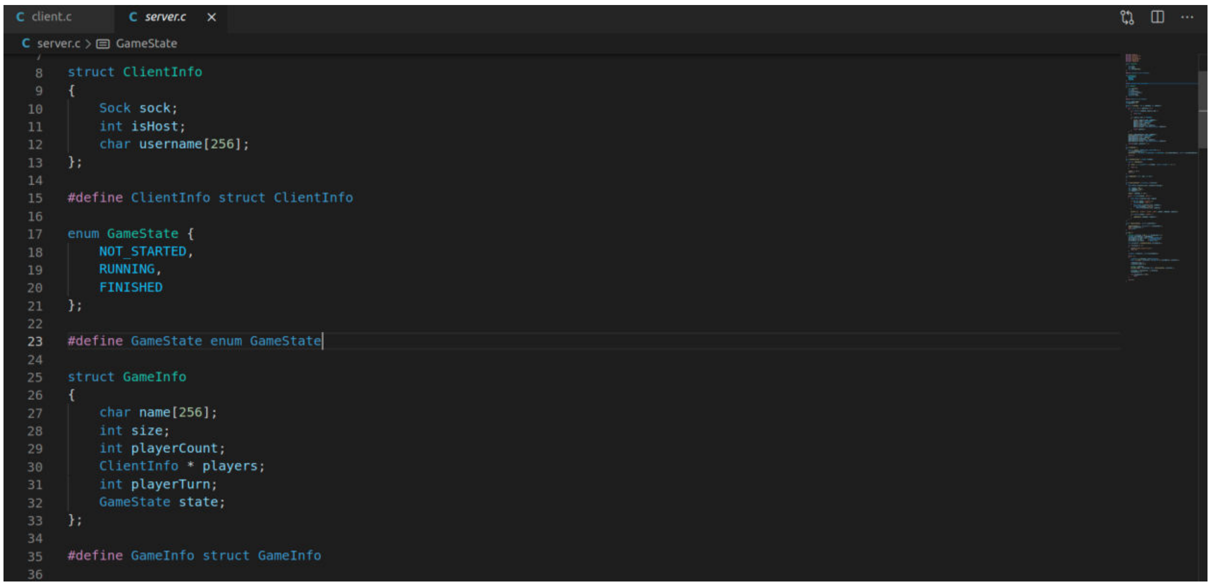


```c
while ( ! STR_EQ ( globals.command, "quit" ) )
{
```

```
#define WRITE_STRING(descriptor,string) { int length = strlen ( string ) + 1; write ( descriptor,
& length, sizeof ( int ) ); write ( descriptor, string, length ); }

Expands to:

{ int length = strlen ( globals.command ) + 1; write ( sock, & length, sizeof ( int ) ); write (
sock, globals.command, length ); }
```

```c
    WRITE_STRING( sock, globals.command )
    if (STR_EQ( globals.command, "create" ) ||
```

Structures used in matchmaking

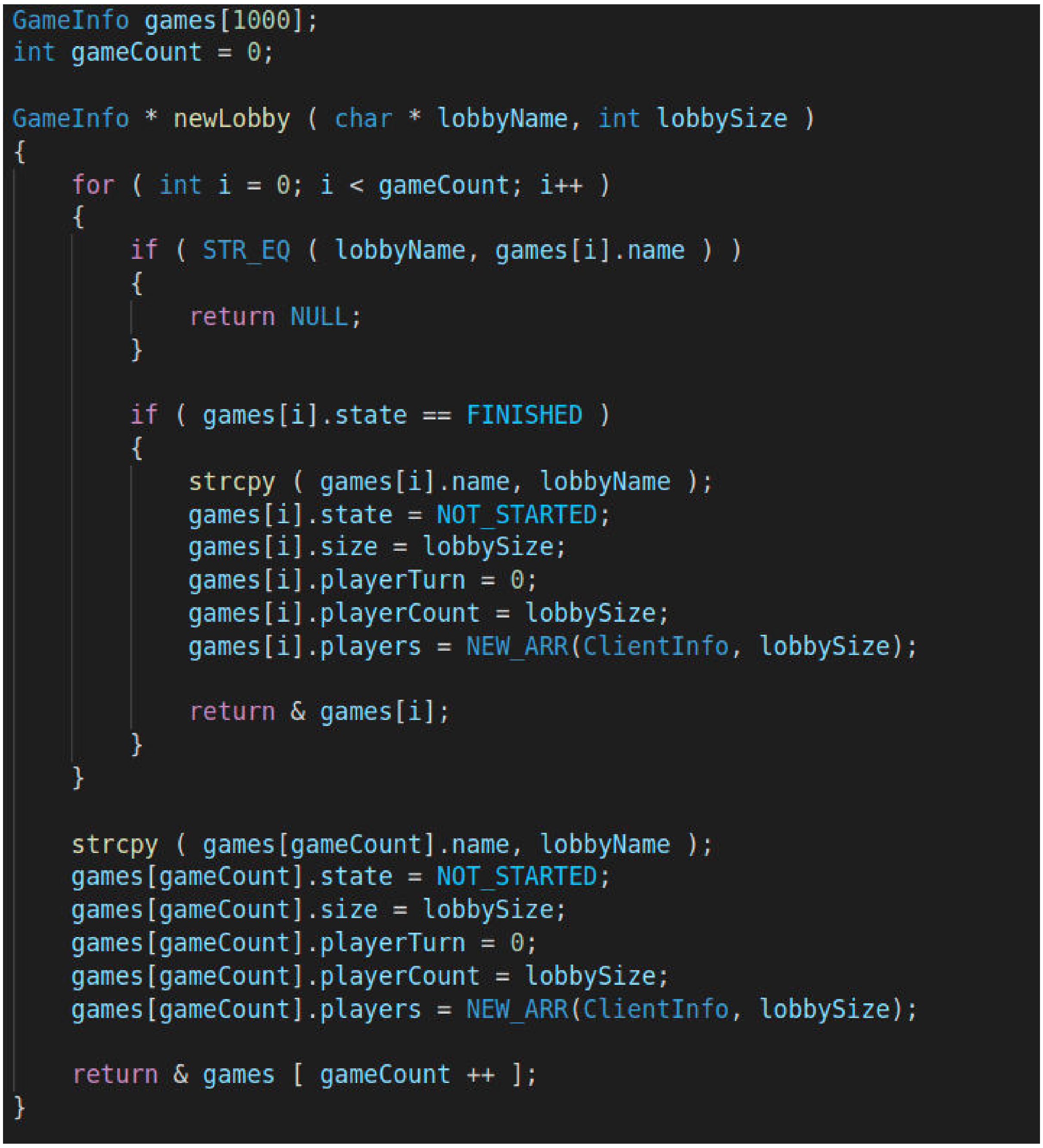


client.c   server.c ×

server.c > GameState

```c
struct ClientInfo
{
    Sock sock;
    int isHost;
    char username[256];
};

#define ClientInfo struct ClientInfo

enum GameState {
    NOT_STARTED,
    RUNNING,
    FINISHED
};

#define GameState enum GameState

struct GameInfo
{
    char name[256];
    int size;
    int playerCount;
    ClientInfo * players;
    int playerTurn;
    GameState state;
};

#define GameInfo struct GameInfo
```

Work In Progress Code section for matchmaking (lobby creation) :
If player attempts to create lobby with the same name as an existing lobby, he will be told that such a lobby already exists (returns NULL here)



```
GameInfo games[1000];
int gameCount = 0;

GameInfo * newLobby ( char * lobbyName, int lobbySize )
{
    for ( int i = 0; i < gameCount; i++ )
    {
        if ( STR_EQ ( lobbyName, games[i].name ) )
        {
            return NULL;
        }

        if ( games[i].state == FINISHED )
        {
            strcpy ( games[i].name, lobbyName );
            games[i].state = NOT_STARTED;
            games[i].size = lobbySize;
            games[i].playerTurn = 0;
            games[i].playerCount = lobbySize;
            games[i].players = NEW_ARR(ClientInfo, lobbySize);

            return & games[i];
        }
    }

    strcpy ( games[gameCount].name, lobbyName );
    games[gameCount].state = NOT_STARTED;
    games[gameCount].size = lobbySize;
    games[gameCount].playerTurn = 0;
    games[gameCount].playerCount = lobbySize;
    games[gameCount].players = NEW_ARR(ClientInfo, lobbySize);

    return & games [ gameCount ++ ];
}
```

Conclusions

While the matchmaking feature is ambitious on its' own, further improvements can be done on the original idea. Another yet undrafted feature is users' custom word dictionary, which will use only words present in that dictionary or the words the server already recognises and the given words.

References

1. https://github.com/titoBouzout/Dictionaries
2. https://gcc.gnu.org/onlinedocs/cpp/Macro-Arguments.html#Macro-Arguments

3. https://www.cs.yale.edu/homes/aspnes/pinewiki/C(2f)Macros.html
4. https://profs.info.uaic.ro/~computernetworks/cursullaboratorul.php - laboratory materials for startup code
5. https://gcc.gnu.org/onlinedocs/gcc/Unnamed-Fields.html - for structured client globals