

# 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

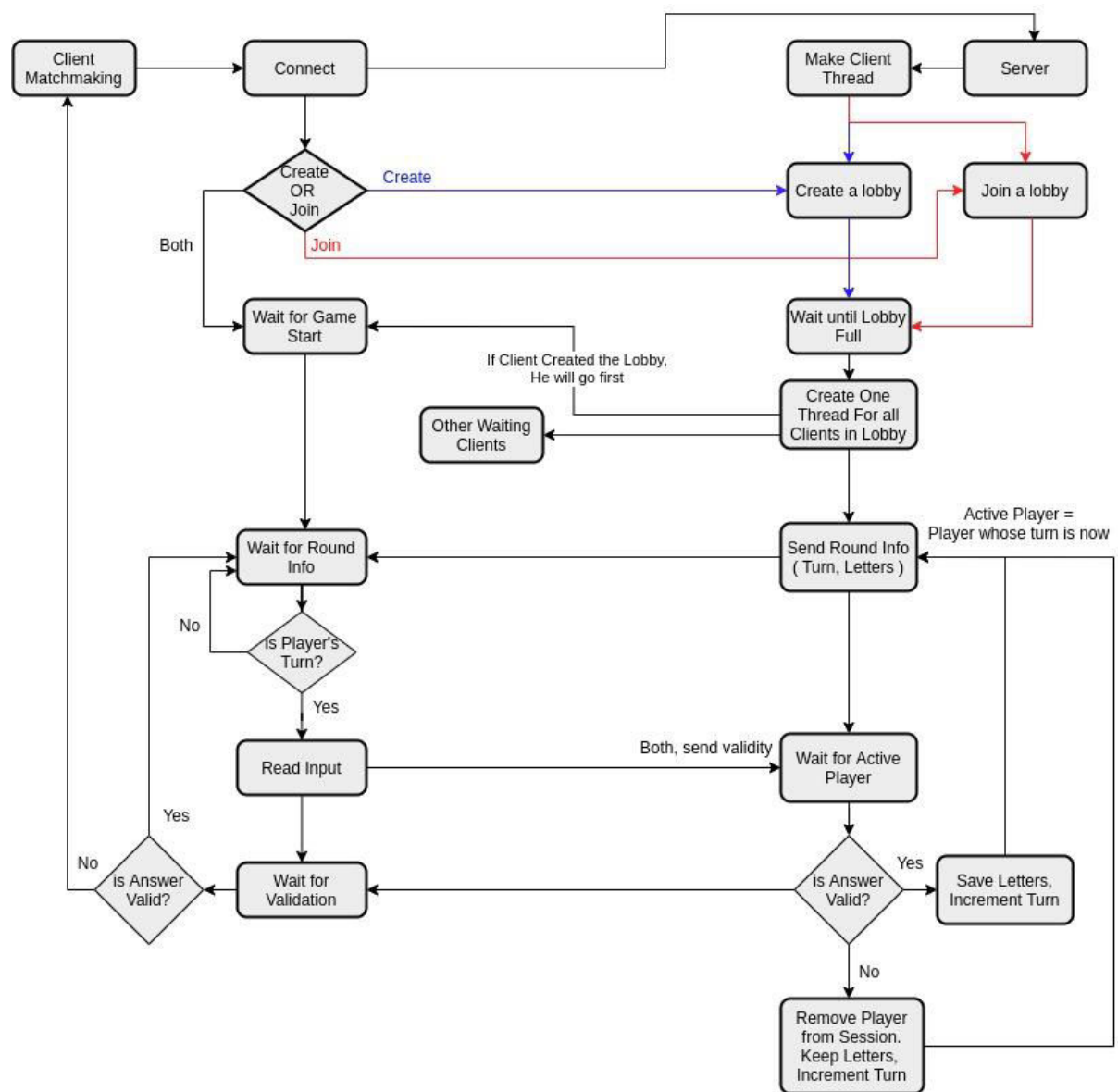
OR

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.

```

graph TD
    ClientMatchmaking[Client Matchmaking] --> Connect[Connect]
    Connect --> CreateOrJoin{Create OR Join}
    CreateOrJoin -- Create --> CreateLobby[Create a lobby]
    CreateOrJoin -- Join --> JoinLobby[Join a lobby]
    CreateLobby --> WaitLobbyFull[Wait until Lobby Full]
    JoinLobby --> WaitLobbyFull
    WaitLobbyFull --> CreateThread[Create One Thread For all Clients in Lobby]
    CreateThread --> SendRoundInfo[Send Round Info ( Turn, Letters )]
    SendRoundInfo --> WaitActivePlayer[Wait for Active Player]
    WaitActivePlayer --> IsAnswerValidServer{Is Answer Valid?}
    IsAnswerValidServer -- Yes --> SaveLetters[Save Letters, Increment Turn]
    IsAnswerValidServer -- No --> RemovePlayer[Remove Player from Session. Keep Letters, Increment Turn]
    SaveLetters --> SendRoundInfo
    RemovePlayer --> SendRoundInfo
    SendRoundInfo --> WaitRoundInfo[Wait for Round Info]
    WaitRoundInfo --> IsPlayersTurn{Is Player's Turn?}
    IsPlayersTurn -- Yes --> ReadInput[Read Input]
    ReadInput --> WaitValidation[Wait for Validation]
    WaitValidation --> IsAnswerValidClient{Is Answer Valid?}
    IsAnswerValidClient -- Yes --> ClientMatchmaking
    IsAnswerValidClient -- No --> IsPlayersTurn
    IsPlayersTurn -- No --> IsPlayersTurn
    
```

The flowchart illustrates the game logic for a word game, showing the interaction between a Client and a Server. The process starts with Client Matchmaking, leading to a Connect step. From Connect, the flow goes to a decision point: Create OR Join. If the client chooses to Create, it leads to Create a lobby. If the client chooses to Join, it leads to Join a lobby. Both paths lead to Wait until Lobby Full. Once the lobby is full, the server creates one thread for all clients in the lobby. The server then sends round info (turn, letters) to the client. The client waits for the active player. The server then checks if the answer is valid. If yes, it saves letters and increments the turn. If no, it removes the player from the session and increments the turn. The server then sends round info back to the client. The client waits for round info. The client then checks if it's their turn. If yes, they read input and wait for validation. The client then checks if their answer is valid. If yes, they return to Client Matchmaking. If no, they return to the Is Player's Turn decision point.



## Source Code

```
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;
}

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 client.c x C commons.h
C client.c > ...
142 Sock createSock ()
143 {
144     Sock s = socket ( AF_INET, SOCK_STREAM, 0 );
145     int activeReuseAddress = 1;
146     setsockopt( s, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, & activeReuseAddress, sizeof ( activeReuseAddress ) );
147
148     return s;
149 }
150
151 Sock createClientSock ( SockAddr sockAddr )
152 {
153     Sock s = createSock();
154
155     if ( connect ( s, ( SockAddrPtr ) & sockAddr, sizeof ( SockAddr ) ) == -1 )
156     {
157         return -1;
158     }
159
160     return s;
161 }
162
163 int main()
164 {
165     SockAddr serverAddress;
166     serverAddress.sin_addr.s_addr = inet_addr(SERVER_ADDRESS);
167     serverAddress.sin_port      = PORT(SERVER_PORT);
168     serverAddress.sin_family    = ADDRESS_IPV4;
169
170     Sock sock = createClientSock( serverAddress );
```

## Macros used

```
#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:

```
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();
    }
}
```

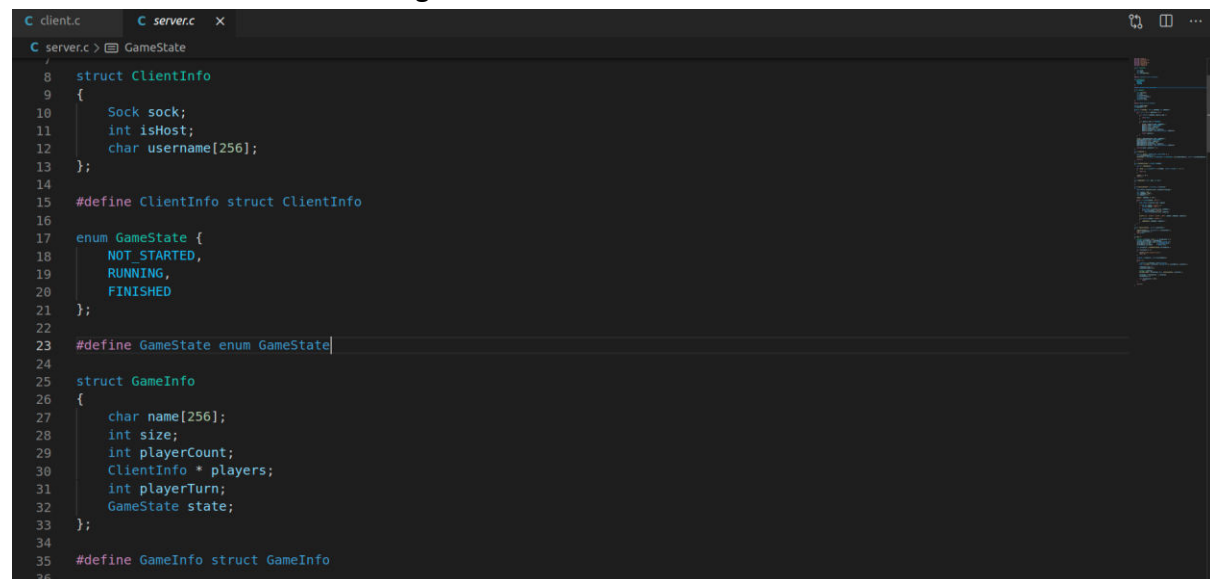
Macros in action

```
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 ); }

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

Structures used in matchmaking



```
C client.c  C server.c x
C server.c > GameState
8 struct ClientInfo
9 {
10     sock_t sock;
11     int isHost;
12     char username[256];
13 };
14
15 #define ClientInfo struct ClientInfo
16
17 enum GameState {
18     NOT_STARTED,
19     RUNNING,
20     FINISHED
21 };
22
23 #define GameState enum GameState
24
25 struct GameInfo
26 {
27     char name[256];
28     int size;
29     int playerCount;
30     ClientInfo * players;
31     int playerTurn;
32     GameState state;
33 };
34
35 #define GameInfo struct GameInfo
36
```

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/Dictionaryes>
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](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