TÉCNICO LISBOA

Concentration card game) Distributed Multiplayer

Systems Programming, 2oSemester 2018/2019

Electrical and Computer Engineering, Instituto Superior Técnico

José Silva, 84109

Miguel D'Ajuda, 84144

June 5, 2019

Summary

This report performs a detailed analysis of the program written in C that implements a memory game, distributed multi-player. This game implements fundamental concepts of Systems Programming, from synchronization between clients and server to the use of*threads*

All topics mentioned in the report suggestion will be explained, also including criticism of major decisions taken throughout the project. The individual functioning of each function is described in the comments of the code created.

Contents

1 Architecture

The program architecture is shown in Figure 1.

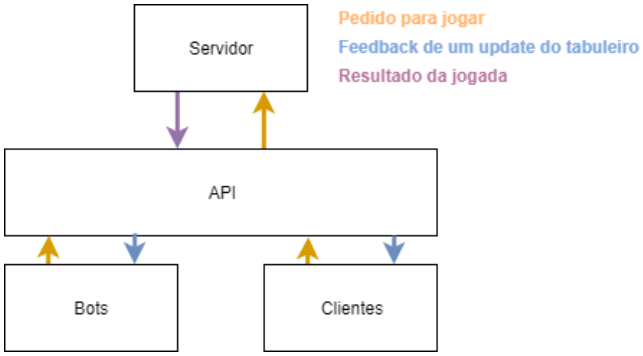

Figure 1: System architecture

As is evident from Figure 1, a Client-Server type of communication architecture was adopted, with the addition of the communication API thus mixing with the N-Tier architecture.

The structure of the solution is presented, with a significant degree of abstraction, in Figure 2.

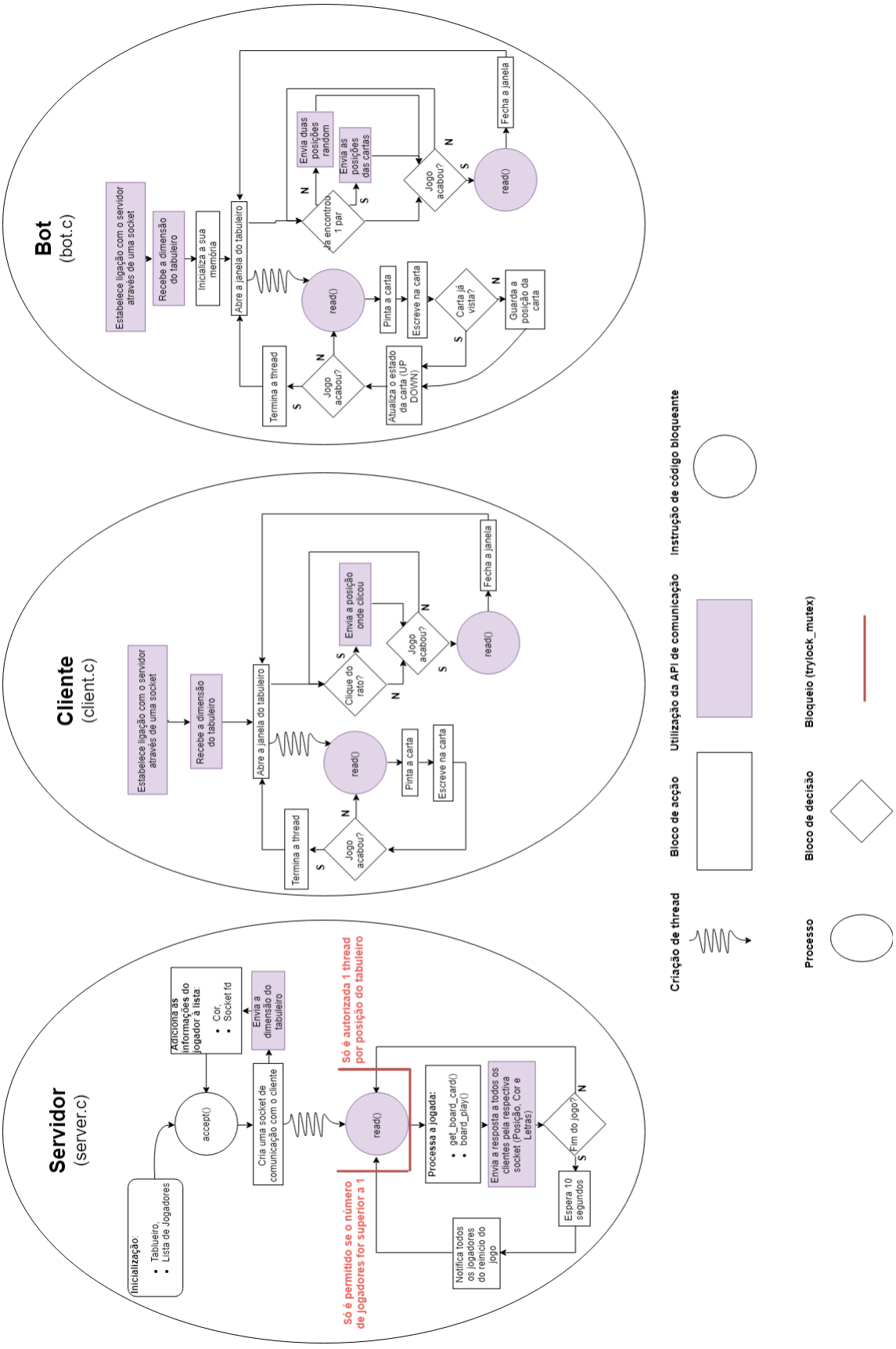

Figure 2: Scheme of the solution.

2 Organization of the code

The code consists of seven .c files, six .h files and one *Makefile* which compiles this set creating the executables.

Starting with the files provided by the faculty:

board_library.c:

In this file, some changes were made to the functions init_board(1) and board_play(4) that allowed storing more information on the structure of the board and dealing with some types of moves that were not defined, for example, loading the card itself. In this way, the structure returned by the second mentioned function, although it is the same, can have more values with different meanings in the return (explained in more detail in the next section).

Furthermore, two functions were added, free_board(0) and get_piece(1), which free the memory of the board vector and return the structure with a specified position on the board, respectively. These functions will be called inside the server file.

board_library.h:

This file remained the same, with the exception of the necessary changes so that the changes made in the .c corresponding to this *header* were possible. In this way, a new library was also included that allows definitions of the type *color*, created by us.

UI_library.c:

In this file, small delays of 10 milliseconds were added for each render (three in total). Due to some out-of-memory bugs in the SDL library, we also created *mutexes* that only allow you to write one letter at a time.

UI_library.h:

No changes were made to this file.

from the file *memory-single.c* provided, much of the code was reused in building the functions that follow. Thus, the next files are related to the servers.

sock_dg.h:

In the light of one of the laboratory works, this simple file was created that defines the location, on the machine, of the *socket* and what is the port number to be used in communications between the server and client/*bot*

our_list.c:

This file serves as an aid to the server and contains all the functions related to the manipulation of the list of clients (from the server). The names of each function make it explicit what they do. The comments in the code describe this and also make its arguments and returns explicit.

our_list.h:

In this module *header* are the declared headers of the functions defined in *our_list.c* This also includes the definition of the player structure used on the server.

server.c:

This is the biggest module of the project and includes six global variables that are responsible for the *you mess up*, by the list and number of players, by the size of the board (received as an argument by the program) and by the state of the end of the game. This file contains two functions ("client_handler" and "main") that form part of the upper layer of the code, being responsible for managing most of the functions of the modules described above.

An individual description of each function can be found by topics below.

color PlayColor(int fd):

The single argument of this function was used only to debug the results of this function. Its functionality is to generate a new random color for each new player as soon as he enters your*thread*

In order not to have card colors that make the letters incomprehensible, three conditions were created that filter out these cases and try a new color. Thus, the critical colors would be gray, red and black, and the new color would have to be at a distance (calculated by the Euclidean norm) greater than 75 rgb units from each of these. This logic can be better interpreted by thinking of a cube with a side of 255 (red, green and blue) where the new color must be at a radius greater than 75 from each of the three points corresponding to the three critical colors, as represented in the figure. 3.

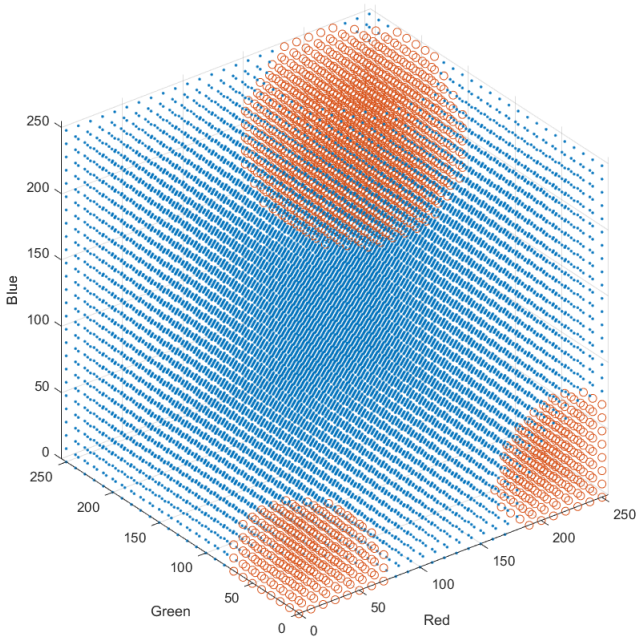

Figure 3: Areas in red - prohibited; blue zones - available

void play(int play[2], char str_play[3], color c_letter, color c_card):

This function fills the structure with the graphical information of the move and goes through the list of players sending the move to each one. Drawing on the server is also performed in this function.

void timeout(struct pollfd *fds, int x, int y, int play1[2], int *locked, color color_xpto):

This function is responsible for making the*timeout*if the player takes more than 5 seconds to make the second move. To implement this functionality we used the function*poll(3)*It is a*mutex*which unlocks the card after waiting.

void *garbage_collector(void *sock_fd):

This function does a non-blocking reading of the customers' moves and is called after the second move, in case it is wrong. So all the garbage that would fill the *buffer* of server reads during the two second wait is absorbed here and ignored. Note that this function is a *thread* separate that is destroyed after the two second delay.

void *wait10(void *sock_fd):

Applying the same logic as in the previous function, a *thread* that absorbs the possible *clicks* of customers during the 10 seconds of waiting until the start of the next game. Furthermore, if an invalid message (with error) is detected, it is assumed that the player has disconnected and the information in the list of players is updated. It is *thread* is destroyed at the end of the 10. seconds.

void new_customer(int fd):

Here is the logic of the graphic part of adding a new player. This function was built for cases where a player joins in the middle of the game.

void *client_handler(void *sock_fd):

This function governs the course of the game, from the point of view of a client/*bot*, as far as the server is concerned. Thus, it is launched in a *thread* when the connection to a new player is accepted.

In order to deal with the 10 second wait between games, a state machine controlled by the global variable "done" was created, which signals the end of the game. If "done=1": the final score is shown (calling the "leaderboard" function); launched the *thread* from the "wait10" function; prepared the board for the next game; scores are reset to zero; it is checked if the player who made the last move left (player of the *thread* concerned - the others are checked against the read they are locked on); the "done" variable stops being zero again and the next game restarts.

For "done=0": the move is received and checked if the player has disconnected, eliminating him from the list in this case; once the position of the play coordinates is obtained, it is checked if there are enough players to continue and if the card is blocked by some *mutex*, otherwise and if the move is on the card itself, the *mutex* is unlocked and the move processed; once the move is validated, it is processed by the "board_play" function; depending on the return of this function, a *switch case* based on the memory-single.c file that performs each hypothesis and the *mutexes* correspondents.

Much of the synchronism logic is in this function but it will only be explained in detail in section 6.

void check_args(int argc, char *argv[]):

This function validates the input arguments and initializes the global variable with the board size. Validation tests will be described in section r5.

int main(int argc, char *argv[]):

Initially, the "check_args" function is called. Then the memory for the *mutexes* of the board is allocated and these are initialized. It is also initialized to

graphical part of the SDL library. Then it creates the *socket* communication with customers/*bots* through the "socket", "bind" and "listen" functions.

Finally, within a cycle, new customers are accepted through the "accept" function and inserted into the customer list. It is still launched thread of the new client ("client_handler" function) and sent the board size to it.

messages.c:

This file contains the four functions common to the client and the *bot*, who are responsible for communication. The description of these functions is in a comment above their declaration.

Note that, although similar, the functions responsible for validating the client's and the client's input arguments were not included. *bot*, as these are likely to be changed in the future. For example, if you want to add an argument to the *bot* which defines the difficulty of it.

messages.h:

In this module, you can find the header of the functions defined in messages.c, as well as the definition of the "message" structure that allows communication in the direction of sending the server. It is also done *define* of the constant "TIMEOUT" that designates the time limit in seconds of the duration of a move.

Next, the "color" structure is also defined, which contains the rgb values of a certain color. Thus, the colors are arranged more clearly. Finally, static constants are created that define 4 base colors, used throughout the course of the program on a regular basis. To prevent the redefinition of these constants, it was necessary to use compilation directives (with "#ifndef" and "#endif").

client.c:

This module consists of three functions that are executed in two *threads* many different. Thus, the "check_args" function validates the program's arguments and is called in "main", followed by initialization of the SDL library, creation of the *socket* and reading the size of the board sent by the server.

Similar to the server.c module, a state machine was built in the "main" function that depends on a global variable "done". If this is equal to 1, the graphics part is restarted and starts sending messages to the server again. If it is equal to 0, a new *thread* for reading the *socket* Then the mouse input is read and sent to the server cyclically.

The "receiver" function is called as the new thread in the "main" and is constantly reading the information sent by the server and drawing cards according to that data. When the game is over, this *thread* is dead.

bot.c:

This file is based on client.c, consisting of the same three functions. However, functionalities are implemented here that save the cards already shown (*bot* with memory), playing based on that information.

The functioning of *bot* is explained in section 7.7, but the information is advanced that the implemented code is prepared to work with different difficulties defined by the "diff" variable, comprised between 0 and 10, defining the time between moves of the same.

bot.h:

This module only includes the declaration of the "bot_memuit" structure that will be used to save the board information in a vector created in the previous file.

As can be seen, whenever possible, a*interface*through files with .h extension that allowed better integration of all the code.

Finally, a Makefile file was also created that creates the server, client and server executables.*bot* If desired, commands have been added to this file that run the program with the*valgrind*, specifically with the*flags*gives*threads* helgrind).

3 Data Structures

In the server.c and message.c modules, a structure already defined in the <netinet/in.h> library of the sockaddr_in type was used, where the data referring to the establishment of a conception between server and client/*bot* This structure was used due to the type of arguments of the client-side "connect" functions/*bot*and "bind" and "accept" on the server side.

Another structure already defined in a C library was used, pollfd from <sys/poll.h>, where the necessary information for calling the "poll" function, responsible for the 5-second wait after the first move, was stored.

As for the two structures provided by the faculty, some changes were made, as can be seen below:

board_place:Two fields have been added to this structure, in order to satisfy the new multiplayer rules. In the board_library.c file, a vector with this structure is used, where the state of the board is stored.

char v[3]: continues to store the two letters of a letter

color owner: identifies the owner of the card by the color code assigned to it

int available: identifies if the card is available

play_response:This structure remained the same, but the values taken by the "code" field were developed. It is used when passing the result of a move on the server.

int code: 0 - completed letter; 1 - first move; 2 - second right move; -2 - second wrong move; -1 - second move on the same card; 3 - end of the game

int play1[2]: position of the first play

int play2[2]: position of the second play

char str_play1[3]: letters of the first play

char str_play2[3]: second play letters

As for the structures created by us, they are present in files with the .h extension and are listed below:

Node:This structure is used as a linked list to store player information.

int fd: player connection file descriptor

int score: player score

struct _Node *next: pointer to the next item in the list

color:This structure holds the three rgb components (*red green blue*) into three integers, r, g and b. It was created to simplify the use of multiple colors throughout the project. It also helped in defining the 4 constant and static colors.

message:This structure is used to send the information from the server to the client/*bot* referring to the graphic part of each move.

int end: marks the end of the game

int pos[2]: move position

char letters[3]: move letters

color color_letter: color of the letters

color color_letter: color of the letter

bot_memunit:This structure is used as a vector in the*bot*and is used to store information on the board

int play[2]: position on the board

char str_play[3]: letters of this position

int available: card availability

4 Communication Protocols

In this section all exchanges of information throughout the course of the game will be described. Although different functions are used, communication is carried out with*socket streams*

4.1 Beginning of the game

At the beginning of the game, as soon as the*thread*of a new player on the server, the board size for that player is also sent.

On the server side, the "write" function is used, and on the client side /*bot*called the function "read_size" which uses "read" to receive this information.

4.2 Card selection

In letter selection, the "write_pos" function (which works with write) is used on the client side/*bot* On the server side, the reading is carried out cyclically at the beginning of the*thread*of the respective user and using the "read" function.

4.3 Updating the board

On the server side, updating the board is done within the*switch case*of the user who made the move. Thus, the list of players is traversed and this information is sent to each one, through a "write".

On the other side of communication, there is a*thread*which is always reading messages from the server with the function "read_msg" which uses a "read".

4.4 Endgame

At the end of the game, the same logic used in updating the board is used, with the difference of using an additional "write" that unlocks the user's "reads" for the next game.

The "recv" function is also used with a *flnow* non-blocking in the 10 second wait between games. This function absorbs all the garbage sent by users and checks if they disconnected.

4.5 Others

Endgame logic is also used in the "garbage_collector" function, using the "recv" function to absorb any *input* of the user during the 2 seconds after a wrong move.

5 Validation

5.1 Between processes

In order to debug the code, several "printfs" were strategically added in critical areas, which allowed us to assess possible errors and their location.

despising the *inputs* of the user, the code will behave relatively predictably. However, program arguments can cause unpredictable situations in case they have an invalid format. In this way, with regard to validation, the focus was directed to the user, creating the "check_args" functions that end the program with *exit(0)* and with an explanatory message for each case. The tests performed are listed below: Common to all files:

argc > 2: too many arguments

argc < 2: less arguments

Only on the server:

sscanf == 0": Failed *cast* converting argument to integer

board_size < 2: board too small

board_size > 26: board too big - not enough letters for pairs not to repeat

board_size%2 == 1: final odd number of cards

Client only/ *bot*

argv[1] > 15: IP address too small

argv[1] < 7: IP address too large

5.2 Return of functions

The return values of the functions that were considered relevant were tested before being returned in the methods themselves. Thus, the only return created that presents a value corresponding to an error is in the "main", in case the *mutexes* fail on startup.

In all creation functions *sockets* and link establishment is checked for return to find on link. If this happens, the program is closed, showing an error message with the "perror" function from the <errno.h> library. This type of validation is also used in all writing and reading functions between the server and the users. Even when using the "recv" function while waiting 10 seconds for the next game, a comparison is made with a *flnow* errno != EAGAIN) to check if the player disconnected.

In the module of the graphic part, it is checked if the creation of the window was successful, and, if not, the program ends with *exit(-1)* and with a suitable "printf".

6 Critical Regions and Synchronization

6.1 Critical Regions

Each board position represents a critical region since it is intended that only 1 player has possession of the card at any instant of time.

For this it will be necessary D_{two} *mutexes* where D corresponds to the dimensions of the game board. In addition these *mutexes* they cannot be blockers since if they were, a player trying to access a card to be used by another would be blocked until the player in possession of the card finishes his turn.

The server, as soon as it receives the valid move request, blocks the *mutex* which corresponds to the position to be played, which is stored in a vector of D_{two} dimensions. If it is the first move, the locked position is saved for later unlocking. As soon as the server receives the second move, if it completes the pair of moves (the player turned over 2 cards), after the move is processed, both *mutexes* are unlocked allowing access to the cards again.

6.2 Synchronization

In the program there are basically 2 events that need to be synchronized, one of them being the restart of the game and the pause of the game due to insufficient players.

The first is done by sending a message indicating the restart of the game. While this is not sent, all clients are blocked reading the socket that connects them to the server.

The second is explained in point 7.1.

7 Description of implemented functionalities

7.1 Minimum number of players

Whenever a player connects to the server, the value of a global variable on the server is incremented. If one of the players disconnects, this is decremented. In Figure 2 it is clear where the blocking takes place if the value of this global variable in the server process is below 2.

Thus, this test is tested before a move is made and resolves the issue of the minimum number of players both at the beginning and during the game.

7.2 Differentiation between first and second play

To differentiate between the first and second move, the existing logic in the code provided by the faculty was applied, in which a vector is used that stores the position of the first move, and this is at -1" if it is the first move. This vector passes by reference from the server module to the board logic module.*bot*applies a similar logic, but in isolation, where it stores each move (first and second) in two separate vectors. These vectors also have the value -1" if there is no move.

7.3 Five-second wait between plays

In this section, the subject of processing and distinction between moves will be revisited.

7.3.1 On the first turn

The 5 second wait is done using the "poll" function, called within the server, whenever a first move is made. After 5 seconds, a move is simulated on the same card (case with code=-1) to turn the card face down again.

7.3.2 On the second turn

The "poll" allows the code to continue normal execution, without adding additional code to cancel the wait (this cancellation is done internally by the function itself).

7.4 Two-second wait after second wrong move

If the player made a mistake on his second move, he will have to wait 2 seconds while the selected cards are highlighted in red. For this, if the second wrong move is verified, the server launches a*thread garbageçselector void strut* which receives a vector with two boxes, the first being the descriptor of the client's communication socket that performed the move and the second a*token* This mechanism of ignoring messages received from the player while he is waiting to play is further explained in point 7.5.2.

7.5 Endgame

In the program there are 2 distinct processes in which their operating cycles are governed by a global variable within each of them called*done* This is set to 1 if the game is over, 0 if the game is in progress. With this, the end of the game is marked as the passage of the value of the variable from 1 to 0 on the server, which is later communicated by the response of the move to the client process via socket.

7.5.1 Announcement with the winner

Before restarting the game, the list of players +is traversed and prints on the server's terminal the score of each player (10 points for each pair found) together with its descriptor (socket fd). Finally, it determines the player with the highest score and announces it, printing the descriptor of the winning player(s).

7.5.2 Ten-second wait between games

At the end of each game, the 10-second wait is ensured by the thread *wait10(void * strut)* This receives a vector whose first position indicates the socket descriptor and the second position corresponds to a *token* While the value of *token* is set to 1, all messages are read from the socket in a non-blocking way and are discarded. If there is a loss of connection with the socket, it means that one of the clients has disconnected. As such, it will be necessary to remove him from the game.

As soon as the game is over, the token is initialized to 1 in the array and the thread is started. Following the start of *thread*, the code remains blocked for 10 seconds using the function *wait()* After 10 seconds, the first position of the vector, the *token*, is updated to 0, ending the *thread*

Finally, the server sends a message to all clients in the game, notifying them of the re-start of the game.

7.5.3 Resuming the game on the client

As soon as the indication that the game ends is received, the client ends the *thread* which handles messages from the server and interrupts the loop that checks for mouse events. Then wait for the message from the server notifying you of the re-start of the game.

Once you receive it, close the board window and reopen it, now being empty. Finally, start the *thread* which handles server messages and resumes the cycle of sending moves.

7.6 Cleaning after customers leave/*bots*

Each player's information (color, fd and score) is stored in a list of structures on the server. The number of active players is also found on the server in the form of a global variable. Whenever a player is added to the game, the structure is created with their data and it is added to the list, increasing the number of players. When exiting, the player is searched for in the list using his descriptor and the respective node is removed from the list, decreasing the number of players.

7.7 *bot*

O *bot* It works very similarly to a normal client. It communicates with the server in the same way as the client and also displays the board on your machine. Additionally, it has a memory capable of storing D_{two} cards, where D is the size of the board.

Thus, whenever it receives a response from the server, it runs through memory checking if the received position is already known, if so it updates the state of the letter. If you have received a letter with black letters, it updates the position as unavailable, otherwise it updates the status of the position as available.

The play of one *bot* is done based on the contents of memory. If two positions are saved with the same *string*, the two positions are added to a queue and these are the next moves to be made. Otherwise, two random moves are added to the queue.

8 known bugs

During the first game the program follows the desired specifications without any indication of problems. However, there are some bugs in the graphics when the program is run on the laboratory's computers, namely the painting of certain cards in black for no apparent reason. The reason behind these errors is unknown.

In the course of the games after the end of the first one, some errors were found.

8.1 Blocked cards

In the restarted game, it is sometimes verified that there are some cards that are impossible to access. This problem comes from the blocking of cards when they are in possession of a certain player. At the end of the game some should not be unlocked which results in the error checked.

8.2 Memory release

Although the code contains the necessary elements to free all the allocated memory, the program never manages to execute them. Initially, the variable *done* no not actually indicated the end of the game but the end of the program. With this, the instructions to release the memory were executed, until the 10 second waiting mechanism was implemented and the game restarted.

References

[1] IBM. *C language examples* url:https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/fg18800_.htm.

[2] Michael Kerrisk. *The Linux Programming Interface* url:http://man7.org/linux/manpages/index.html.

[3] João Nuno Silva and Ricardo Martins. «Theoretical lectures slides and laboratory work code».

[4] *Stack Overflow* url:https://stackoverflow.com.