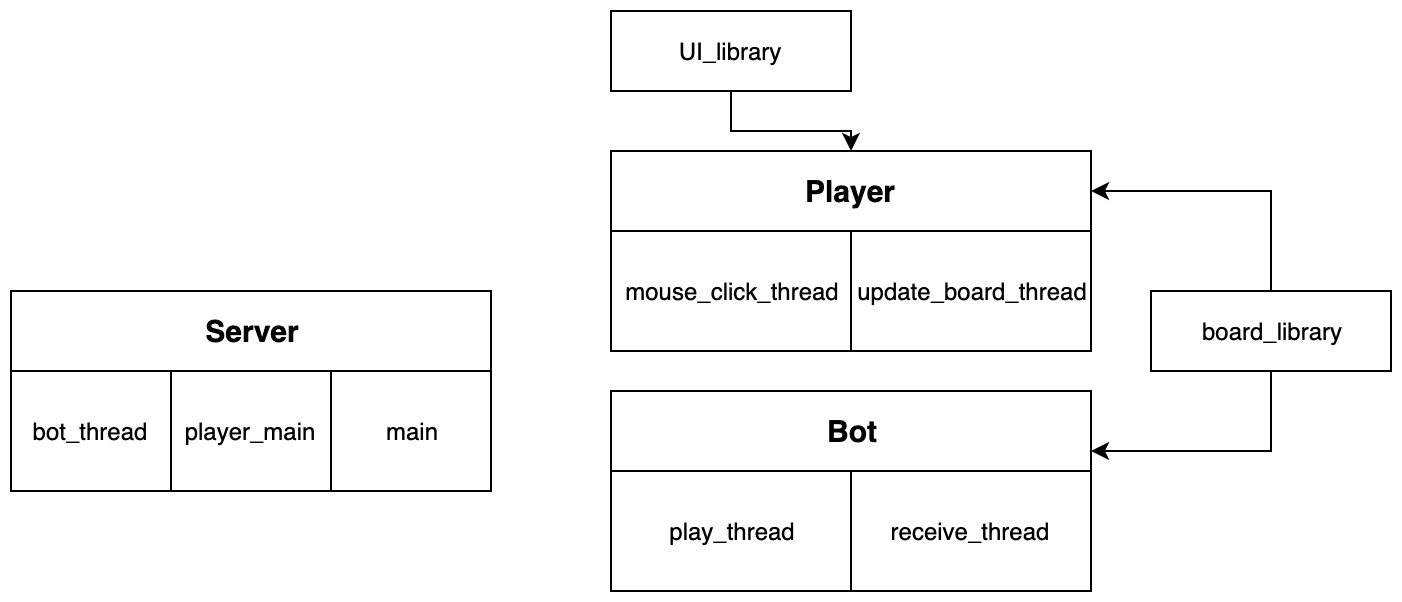
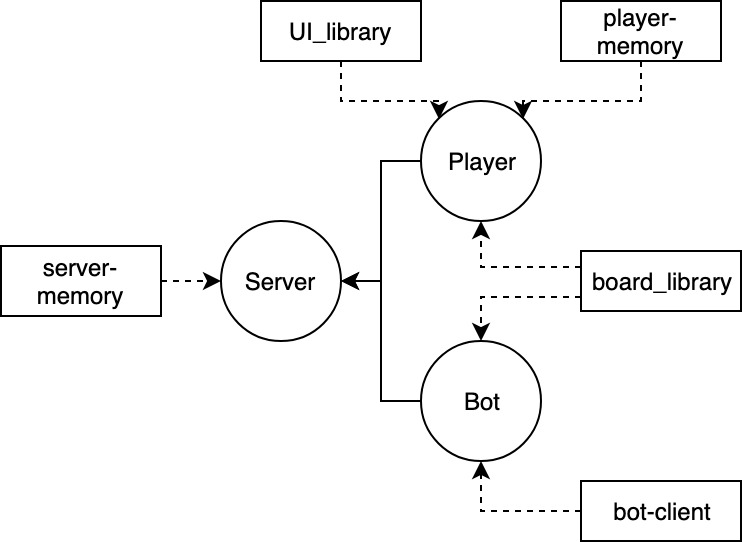
1. Architecture



1. Code Organization

Player-memory.c:

1. *Connect\_to\_server* : create and configure the socket, then connects to the server
2. *Receive\_int* : read a string that represents an integer and convert it before storing it in a buffer
3. *Send\_cards\_coordinates* : receives the x and y coordinates of a card, convert them to string and writes it to the server
4. *mouse\_click\_thread*: Wait for an SDL event (clicking on a cell or closing the window). If the player clicks on a cell, this thread converts the click to a X and Y position that is then communicate to the server.
5. *String\_color\_to\_RGB* : gets a string “RRR-GGG-BBB” where the letters are numbers and converts each of these colors into an integer.
6. *Erase\_board* : paints the whole board in white
7. *Update\_board* :gets a string containing a cell color, the string color, the string and the x and y position of the card. It then use the paint\_card and write\_card function from board\_library.c to update the board.
8. *Parse\_plays* : receives a string containing either one or two cells to modify. If it contains two cells, it splits the string and call the update\_board function with both.
9. *update\_board\_thread*: This thread is waiting to receive a communication from the server to update the board.
10. *Init\_SDL\_TTF*  : initialise SDL and TTF.

Bot-memory.c

1. *Linear\_conv* : receives a X and Y coordinates, and convert them to an index to acces the corresponding element in a data structure.
2. *play\_thread*: One thread generates random X and Y coordinates until the cell is available. Then, it sends the pick to the server, wait and sends the second pick.
3. *Change\_availability :* change the availability of a card.
4. *Connect\_to\_server* : -
5. *Send\_cards\_coordinates* : -
6. *Receive\_int* : -
7. *Get\_coords\_and\_availability* : similar to the function update\_board from the client. It gets a string, parse it to find the coordinates of the cell and the color, and store them in the parameters of the function.
8. *receive\_thread*: The second thread is waiting to receive updates about cells that status has changed.
9. *Erase\_board*: make all cells available
10. *Update\_availability* : change the availability of a cell to unavailable, this function is used when the bot connects to the server and receives the information about the card that are already picked.

Server-memory.c:

1. *Init\_connections* : initialise the socket to receive new connections from clients.
2. *Connect\_to\_player* : add a newly connected player file descriptor to the players\_fd array.
3. *Wait\_2\_seconds\_return* : thread that waits for 2 seconds before returning the cards.
4. *Wait\_5\_sec\_for\_play* : thread that leaves 5 seconds to the player before returning the card
5. *Receive\_card\_coords* : receives a string with the cards coordinates, parse it and converts them to integers.
6. *Update\_info* : creates a string with all the information needed in order for the players to update their board (cell color, string color, string, x, y)
7. *Random\_color* : generates a random color.
8. *Deconnect\_player* : disconnect the player by making him un-active, send a message to close the player listening thread and finally close the socket.
9. *Write\_to\_all* : sends a string to all active connected client.
10. *Send\_current\_board\_dispo* : goes through all the cells, and if the cell is currently picked by another client, sends the information. This function is called whenever a new client connects to the server.
11. *Main* : always waiting for a new connection
12. *player\_main*: One thread created for each new player connection. This thread waits to receive a click, then play the pick and send the cell to update.
13. *bot\_thread*: One thread for each bot connected that waits for a X and Y position, plays the pick and again, sends the correct information to the clients.
14. Data Structures

In the server, there is three data structures (that are global).

1. *int player\_fd[MAX\_PLAYERS][2]* which keeps tracks of the active clients and the file descriptor of every clients that connect. *player\_fd[i][0]* is the file descriptor of the client socket. *player\_fd[i][1]* is a Boolean that represents if the client is still connected (1) or not (0). This 2-dimensional array is initialized with all zeros.
2. *char player\_color[MAX\_PLAYERS][15]* is an array that keeps the players color string (“RRR-GGG-BBB”).
3. *Pthread\_t players\_thread[MAX\_PLAYERS]* is an array that keeps the pthread\_t of every thread that is created when a new client connect.

The client does not need to have data structure.

The board library has 3 data structures:

1. *board\_place \* board* is an array with a size equal to the square of the board dimension. It keeps a string of the symbol present in every cell of the board. It is initialised when the client (or the bot) receive the size of the board.
2. *Cell\_info \* cells\_info* is also an array the size of the square of the board dimension. This array keeps track of every cell status during the game. Cell\_info is a structure containing the string of the cell, the id of the player who *owns* this cell (-1 if nobody owns it) and the color of the string in this cell. When a player picks a cell, this array is updated.
3. *Int play[MAX\_PLAYERS][2]* is an array that keeps track of the pick a player made. Play[id][0] is the first pick a player made (and is reset at -1 after he made his second or waited too long) and play[id][1] is his second pick. With this array, the server can verify if his second pick was right or wrong.
4. Communications Protocols

When a client (player or bot) connects to the server, it immediately sends a string that identifies him as a bot (“b”) or as a client (“cl”).

Then, the server sends the dimension of the board. This way, the player can create his *mouse\_click\_thread* and initialisehis board. The bot thread can initialise his data structures.

Then, the server sends a series of string that are currently taken or selected by the other players, so the newly connected client can update his board and have the same as the other clients. The client receives the updates one at a time and update his board accordingly.

The client and bot threads (in the server) waits to receive a message (coordinates of a pick) from the client node.

On the player node, every time *the mouse\_click\_thread* reads a click, it sends the X and Y of the corresponding cell to the server. The server plays this card and then sends to all connected client the string to update their board.

The clients *update\_board\_thread* are constantly waiting to receive a string formatted this way:

“cell1=cell2” or “cell1”

Where a cell is formatted like this:

“paint\_color:write\_color:string\_to\_write:x\_str:y\_str”

And where the colors are the RGB numbers (“RRR-GGG-BBB”).

Then this thread parses the string and update the board. The *receive\_thread* in the bot node also receives the same string and update his data structures correspondingly.

When a player or a bot wants to disconnect (by sending a CTRL-C signal), it sends the coordinates (-1,-1) to the server. Then the server knows this player (or bot) wants to disconnect and sends to this client (or bot) a string (“over”) so this client (or bot) *update\_board\_thread* (or *receive\_thread*) knows that it’s the last message it will receive and can close the socket and terminate the program.

When the game is over, the server sends to all clients and bots the string “game-finished”. When the client or bot receives this message, it knows the game is finish and a new one will start in 10 seconds.

Every time the server changes a card status, it sends the information to every player and every bot.

1. Validation
2. Critical Regions/Synchronization

The critical region is when the server receive a pick, it must make sure that the pick is played before receiving another pick from another client.

1. Description of the implemented functionalities
   1. Minimum number of players(before, during)

In the server, there is a global variable *nb\_connections* that increase every time a player (or a bot) connects and decrease when a player (or bot) disconnect. This way, before sending the board dimension to a client, the *player\_main* and *bot\_thread* are waiting :

while(nb\_connections < 2){

//wait...

}

Then during the game, the player\_main and bot\_thread in the server node only plays a pick when *nb\_active\_players* is bigger than 2. Otherwise, it receives a pick but doesn’t do anything with it.

if(nb\_active\_players >= 2) {

//some other code…

play\_response resp = board\_play(board\_x, board\_y, id);

This way, the client and the bot don’t need to know if there is enough player, they always send the click information, but thew won’t receive any information to update the board. Therefore, the board will never change when there is only one player connected.

* 1. Distinction from 1st and 2nd pick

The array *play[MAX\_PLAYERS][2]* is initialised with the value -1 in the position *play[id][0]* for each player. This way, when the function

board\_play(board\_x, board\_y, id);

is called, it changes *play[id][0]* to *board\_x*. This way, the next time the player makes a pick, it will verify if *play[id][0] != 1* and if that’s true, it will be his second pick.

* 1. 5 seconds between picks (turn card down and processing 2nd pick)

When the player\_main (or bot\_thread) in the server node receives a first pick, a new thread is created : *wait\_5\_sec\_for\_play* :

void \* wait\_5\_sec\_for\_play (void \* arguments){

    struct thread\_args \* args = (struct thread\_args \*) arguments;

    args->lock = 1;

  sleep(5);

    size\_t len = strlen(args->message);

    write\_to\_all(args->message, len);

    reset\_play(args->id, args->x1, args->y1);

    args->lock = 0;

}

This thread set the value of a lock to 1, sleep 5 seconds, and it it is not canceled before, it sends to all clients (and bots) a string with the information of the first cell that needs to be returned, reset the *play[id][0]* to -1 and set the value of the lock to 0.

When the server receives a second pick, it verifies if the lock is set to 1, if it is, it cancels the thread, and play the second pick.

The function board\_play() in the board\_library always verify first if the card is available :

get\_cell\_status(&status, x, y);

if(status.player\_id == -1){

If it’s not available, the function simply returns a *resp.code* = 0. If the card is available, it verifies if it’s the player’s first pick. If it is, it stores the information’s of the pick in the *resp* structure and change the availability of the card :

printf("FIRST\n");

resp.code = 1;

play[id][0] = x;

play[id][1] = y;

resp.play1[0]= play[id][0];

resp.play1[1]= play[id][1];

strcpy(resp.str\_play1, get\_board\_place\_str(x, y));

change\_cell\_status(x, y, id, resp.str\_play1, GREY);

If it’s the player’s second pick, it gets the player’s first pick card and verifies if the second pick card is the same:

char \* first\_str = get\_board\_place\_str(play[id][0], play[id][1]);

char \* secnd\_str = get\_board\_place\_str(x, y);

if (strcmp(first\_str, secnd\_str) == 0){

printf("CORRECT!!!\n");

change\_cell\_status(play[id][0], play[id][1], id, first\_str, BLACK);

change\_cell\_status(x, y, id, secnd\_str, BLACK);

resp.code =2;

// Other code…

} else {

printf("INCORRECT\n");

resp.code = -2;

change\_cell\_status(play[id][0], play[id][1], id, first\_str, RED);

change\_cell\_status(x, y, id, secnd\_str, RED);

}

No matter if the it’s good or bad, the play[id][0] is set to -1 so the player will be able to perform a first pick later.

* 1. 2 seconds delay if wrong

When the response from the board\_play() function is -2 (which means that the second pick did not match the first), the player\_main creates the thread *wait\_2\_seconds\_return()* :

void \* wait\_2\_seconds\_return(void \* arguments){

struct thread\_args \* args = (struct thread\_args \*) arguments;

\*(args->lock) = 1;

size\_t len = strlen(args->message);

sleep(2);

write\_to\_all(args->message, len);

\*(args->lock) = 0;

     reset\_cell\_status(args->x1, args->y1);

     reset\_cell\_status(args->x2, args->y2);

     pthread\_exit(0);

}

This function sleep for two seconds and send a string to all client. During this time a locked (different than the one used in the function *wait\_5\_sec\_for\_play()* is set to 1 like this we can verify if the player can pick a new card or not.

* 1. End of game (transmission of winner, 10 seconds delay, restart)
  2. Clean up after death/disconnect of a client
  3. Bot

Like explained before, the bot is made of two threads: one that waits to receive the update message and when that simply generates random pick. The *receive\_thread()* constantly waits for a message. When it receives one, it verifies if it’s a signal that the game is over. If it’s not, it verifies if there are two cards to update:

if (strstr(buffer, "=") != NULL)

If there are two cards to update, it separates the play :

char \* play1 = strtok(buffer, updates\_delimiter);

char \* play2 = strtok(NULL, updates\_delimiter);

And finally, gets the cards info with the function *get\_coords\_and\_availability()* and change this cell availability :

get\_coords\_and\_availability(&x,&y,&is\_available, play1);

change\_availability(x,y,is\_available);

The *play\_thread* is pretty simple:

while(1 && !isover){

sleep(1);

do{

card\_x = rand() % dim\_board;

  card\_y = rand() % dim\_board;

     } while(!is\_available(card\_x, card\_y) && n\_correct < (dim\_board\*dim\_board));

     send\_card\_coordinates(card\_x, card\_y);

      sleep(2);

do{

       card\_x = rand() % dim\_board;

  card\_y = rand() % dim\_board;

     } while(!is\_available(card\_x, card\_y) && n\_correct < (dim\_board\*dim\_board));

send\_card\_coordinates(card\_x, card\_y);

sleep(3);

}

The function *is\_aivalble()* verifies the array of int *available\_cells* and return 1 if the cell is available.