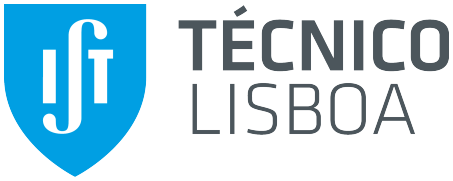

1º Semestre 2021/2022

Mestrado em Engenharia Eletrotécnica e de Computadores

Programação de Sistemas

Relatório do projeto

Nº 93116	Luís Bruno Rafael Verdelho Fava	luis.fava@tecnico.ulisboa.pt
Nº 93906	Diogo José Pereira Araújo	diogoparaujo@tecnico.ulisboa.pt

1 Implemented functionalities

Part 1

	Not implemented	With faults	Totally Correct
Relay-Pong			
Connect of new client to server			X
List of clients on server			X
Paddle/ball movement on client			X
Forward ball movement to all clients			X
Release ball by the client			X
Send ball to another client			X
Update screen (idle clients)			X
Clients disconnect			X
Super-pong			
Connect of new client to server			X
List/Array of clients on server			X
Paddle movement on the client			X
Send paddle movement to server			X
Ball movement			X
Server update board state			X
User points			X
Send board_update to all clients			X
Client screen update			X
Client Disconnect			X

Figure 1: Implemented functionalities (PART 1)

Part 2

	Not implemented	With faults	Totally Correct
Windows writing synchronization			X
New-relay-Pong			
Ball moves independent of paddle			X
Ball movement (1 Hz)			X
Quit with **Q**			X
Player changes every 10 s			X
Synchronization ball/paddle			X
New-super-pong			
Clients connect			X
List/Array of clients on server			X
Send Paddle movement to server			X
Ball movement (1 Hz)			X
Update board			X
Send board_update to all clients			X
Synchronization array/list of clients			X
Synchronization paddles/ball			X
Clients disconnect			X

Figure 2: Implemented functionalities (PART 2)

Description of faulty functionalities

No faulty functionalities were found.

2 Code structure

Functions List

int check_message(message_t msg)

This function is implemented in all programs of this project.

This function is used to check the validity of a message. Its return is 0, for a valid message, or -1, for an invalid one. The workings of this function are explained in more detail in the 'Error treatment' subsection of section 3.

client *add_new_client(client *list, char *addr, int port)

This function is implemented in the Relay-Pong and New-Relay-Pong server programs, and is used to add a new entry to the server's list of clients. The new client is inserted at the end of the list.

client *add_new_client(client *list, char *addr, int port, int * score, paddle_position * paddles, int *id, ball_position_t ball)

This function is implemented in the Super-Pong and New-Super-Pong server programs, with just a small adjustment (the client structure is slightly different, because in New-Super-Pong, each client has its own socket, which needs to be stored).

It is used when a new client tries to connect to the server. If the connection was successful and the maximum number of clients (10) hasn't been reached, then the new client is added to the server's list of clients. The new client is inserted at the head of the list. It is also in this function that the clients get their id.

client *remove_client(client *list, char *addr, int port)

This function is implemented in the New-Relay-Pong server program. It removes the client with the given address and port from the list of clients.

client *next_player(client *list)

This function is implemented in the New-Relay-Pong server program.
This function chooses the next active player.

Threads

New-Relay-Pong

- **Server:**
  - **thread_socket**
    This thread receives messages from the clients.
  - **main**
    This thread initializes the program and then takes care of the player changes that must occur every 10 seconds.
- **Client:**
  - **main**
    This thread initializes the program, sends the **connect** message to the server and creates **thread_recv**. Then it reads keyboard inputs.
  - **thread_ball**
    This thread moves the ball every second. It is only active while the client is in control of the ball. When a **release_ball** message arrives, **thread_recv** cancels this one. That thread is also responsible for creating instances of this one each time the client enters the **play** state.
  - **thread_recv**
    This thread receives messages from the server.

New-Super-Pong

- **Server:**
  - **main**
    The initialization process is carried out in this thread. After creating and binding a socket to listen for new connections and initializing variables, such as **paddles**, **score** and **ball**, it calls **thread_accept** and exits (using the **pthread_exit()** function).
  - **thread_accept**
    This thread accepts new clients and adds them to the list. Then, it sends a **board_update** to all the clients and creates an instance of **thread_client** to listen to the new socket. The socket where new connections are accepted must be ready (using **bind()** and **listen()**) and then passed as an argument to this function (in reality, it's a pointer to the file descriptor that is passed, and not its actual value).
    If a new client connects after the maximum number of clients has been reached, that client is rejected. This is achieved by sending a **board_update** message with **id=-1**.
  - **thread_client**
    This is the thread where client messages are "processed". Since this program uses stream sockets, there must be an instance of this thread for each registered client. If the client disconnects (either normally, by pressing 'q', or abruptly), it is removed from the list, the socket is closed, and that instance of this thread exits. This thread receives a pointer to the file descriptor as an argument.
  - **thread_ball** This thread moves the ball (taking paddles and walls into account and increasing the score if the ball happens to hit a paddle) and then sends a **board_update** to all the registered clients. It does this once every second.

- **Client:**
  - **main**
    The initialization process is carried out in this thread. Then, it is used to read keyboard inputs.
  - **thread_recv**
    This thread receives messages on a given socket. The address of said socket is passed to this thread as an argument. If the **recv()** function returns 0, the whole program is killed (**exit()**) for losing the connection to the server.

Shared variables

New-Relay-Pong

- **Server:**
  - **client_list** This variable holds a pointer to the head of the list of clients held by the server. Since the list is constantly being updated (new clients coming in and disconnected clients being removed) and accessed by all the threads in the program, there is an advantage to making it a global variable.
  - **active_player** The ball can be passed to the next player for two reasons (either 10 s have passed since the last release or the ball-controlling client quit). Since these two situations are detected in different threads, it is helpful to make this a shared variable.
  - **ball** The position of the ball is updated in **thread_socket** and accessed by **main** when switching players (send_ball message).
  - **t_last_snd** This variable is used to keep track of when the last ball release took place. The 10 seconds of play time are counted from the instant stored in this variable.
- **Client:**
  - **state** The current state of the client (**wait** / **play**) is important in all parts of the program.
  - **ball** and **paddle** In order to move the ball, the position of the paddle must be known, and vice-versa.
  - **server_addr** This variable is used when sending messages to the server, which happens in **main**, at the very beginning of the program, and in **thread_ball**. It could have been passed as an argument, but it was simpler to make it a global variable.

New-Super-Pong

- **Server:**
  - **client_list**
    This variable holds a pointer to the head of the list of clients held by the server. Since the list is constantly being updated (new clients coming in and disconnected clients being removed) and accessed by all the threads in the program, there is an advantage to making it a global variable.
  - **score** and **ball**
    Both these variables are updated every second in **thread_ball** and need to be accessed by the other two threads in order to send **board_update** messages. The ball is also needed when moving and creating paddles.
  - **paddles**
    Each instance of **thread_client** is responsible for writing to one position of this vector. This variable is then needed in **thread_ball**, hence why it was made global. However, since each thread is responsible for a different position in the array, it is not protected by a mutex.
- **Client:**
  The client has global variables, but they are not used concurrently by both threads.

Synchronization

The main concern in terms of synchronization in this project was the definition of critical regions and the use of mutual exclusion to guard them. For instance, the lists of clients held by both servers are protected, because searching a list while another thread removes a client can easily lead to a Segmentation Fault. In New-Relay-Pong, active_player is also protected, for good measure.

Most shared variables are protected, especially if multiple threads write to them.

client *client_list

This variable serves the same purpose in both servers. It is accessed by the two versions of **add_new_client()**, by **remove_client()** and by **next_player()**.

client *active_player

This variable holds a pointer to the current ball-controlling client's structure, in order to keep track of who is in control. The next player is chosen in function **next_player()**, which needs to access and update this variable.

Balls and Paddles

All forms of balls and paddles are accessed by the same type of functions. Any function that draws (or erases) a ball from the screen needs to access the **ball** variable. The same goes for paddles and functions that draw or erase paddles from the screen. Likewise, any function used to change the position of a ball or a paddle needs to access both balls and paddles (in order to take collisions into account).

int score[ ]

This variable is exclusive to the Super-Pong "family". It is accessed by **move_ball()**, which is where each player's score is incremented each time the ball hits their paddle.

3 Communication

Transferred data

Relay-Pong & New-Relay-Pong

The data structure exchanged between the server and the client is represented below (message_t).
The messages exchanged between the server and client are the following.
The msg_type, that contains the different messages types exchanged between the server and client:

- type = 'conn' corresponds to the Connect message;
- type = 'rls_ball' corresponds to the Release_ball message;
- type = 'snd_ball' corresponds to the Send_ball message;
- type = 'mv_ball' corresponds to the Move_ball message;
- type = 'disconn' corresponds to the Disconnect message;

The ball position, that contains the coordinates of the ball. This message is only exchange when the type is one of the following three: Release_ball, Send_ball or Move_ball.

```
1 typedef struct message_t{
2     msg_type type;
3     ball_position_t ball_pos;
4 } message_t;
```

```c
typedef enum msg_type{conn, rls_ball, snd_ball, mv_ball, disconn} msg_type;
```

Super-Pong & New-Super-Pong

The data structure exchanged between the server and the client is represented below (message_t). The data structure contains the following elements:

- type corresponds to the type of the message. In the Super-Pong the messages exchanged are either a paddle_move type, or board_update;
- paddle_pos is an array containing the positions of all the paddles.;
- ball_pos is the position of the ball;
- score is an array containing all the players' scores;
- id helps identify the client in question. By using it, clients can identify their own paddle and score;
- key represents the key pressed by the user;

```c
typedef struct message_t{
    msg_type type;
    paddle_position_t paddle_pos[MAX_CLIENTS];
    ball_position_t ball_pos;
    int score[MAX_CLIENTS];
    int id;
    int key;
} message_t;
```

```c
typedef enum msg_type{conn, paddle_move, board_update, disconn} msg_type;
```

The New-Super-Pong **msg_type** does not include **conn** and **disconn**, because the use of stream sockets makes these types of message useless.

Error treatment

In order to prevent the exchange of invalid messages, the number of bytes received must always equal the size of the corresponding **message_t** structure and each "family" has a **check_message()** function. These functions check each field of the corresponding **message_t** structure, to make sure it's acceptable. For instance, in Super-Pong, a **board_update** message must have paddle positions either inside the window or -1, and in any case, the length must be equal to the constant **PADLE_SIZE**.

In New-Super-Pong, the return value of **recv()** was also used to detect when clients disconnect.