Systems Programming Report

Grupo 14

Gonçalo Frazão nº 99945   Gustavo Vitor nº 100311

Implemented functionalities

Our project implements all the functionalities of the assignment.
All communications use **protocol buffers**. The **.proto** file implemented creates 3 types of messages:

- **request messages** - sent from a client to the server to either move the object (player character, wasp or roach), connect or disconnect;
- **replies messages** - sent from the server to the clients that sent request messages earlier;
- **display update messages** - sent from the server to all lizard clients with information needed to update the display.

The **address** and **port** of the server are supplied to the program as a command line argument.

Both the clients (lizards, roaches and wasps) and the server are implemented.

All the **disconnects** are implemented (lizards, roaches and wasps). In the lizard client disconnection is made with **'q'** or **'Q'** or after **one minute of inactivity**, and in the others is made by pressing **Enter**.

Displays are implemented in server and in lizard client with NCurses interfaces. The tail of lizards, the score board and all the move tracking from all clients is displayed.

The **machine clients** (wasps and roaches) use one thread to handle connection and movement of the wasps and roaches, while disconnection is handled in main.
The **lizard client** uses 2 threads. One handles the movement, connection and disconnection of the player, while the other handles the display of the board and scoreboard.
The **server** is also multi-threaded (5 threads to clients, 5 threads to roaches and wasps, 1 thread to control offline users, variable number of threads to control 5 seconds respawn).

All the connection is done with **ZeroMQ sockets**. The communication between threads is made through **inproc sockets** (router-dealer) using a **Proxy** that intermediates the processing of messages. The communication between clients and server is done with **tcp sockets**.

The roaches and wasps clients can control between 1 and 10 roaches and wasps, respectively. Both clients select **random roaches and wasps** to move in **random directions** with **random time intervals** (from 0 to 0.7 seconds) between moves to make the movement more realistic.

All the **movement interactions** are also implemented (lizards can't share positions with other lizards or wasps, wasps can't share positions with roaches, lizards or other wasps, roaches can't move into a position occupied by a lizard or wasp the player/bots can't leave the board).

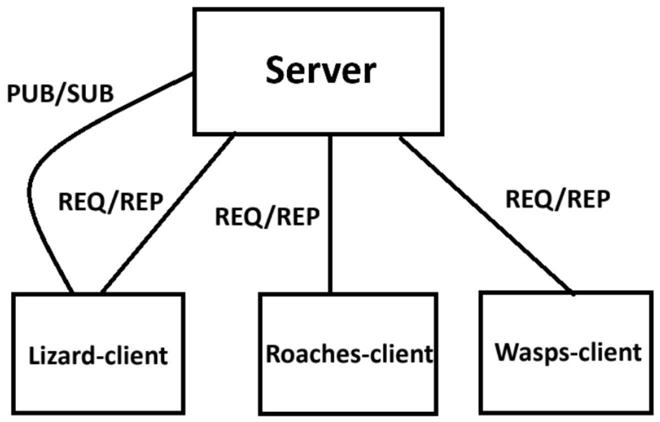
Roaches **disappear when eaten** and the **respawn timer** is implemented with multi threading control. The tail changing to **'*'** when the score of a lizard is higher than 50, and no tail when the score is lower than 0 is also implemented. Points are **added** when lizards eat **roaches**, **removed** when bump into or are bumped by **wasps** and the **average** of their scores is given to the **two players** when they bump into each other.

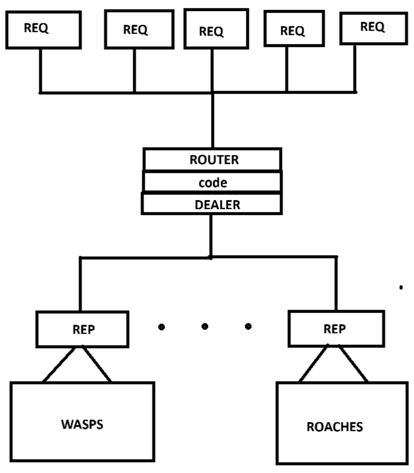
The **number of players** is capped at **26** and if a player tries to connect after that the program terminates with a print saying the server is full.
The **sum of roaches and wasps** in the field can't be higher than **one third** of the field area. There are two cases where the maximum occupation is reached: if the server is almost full and the number of roaches/wasps created by the client is higher than the capacity left; if the server is full and a new roaches/wasps client tries to create roaches/wasps. In the first case the client prints on the terminal a message saying the field is full but continues to run normally but with fewer roaches/wasps than originally intended. In the second case, the clients print the same message but the program terminates.

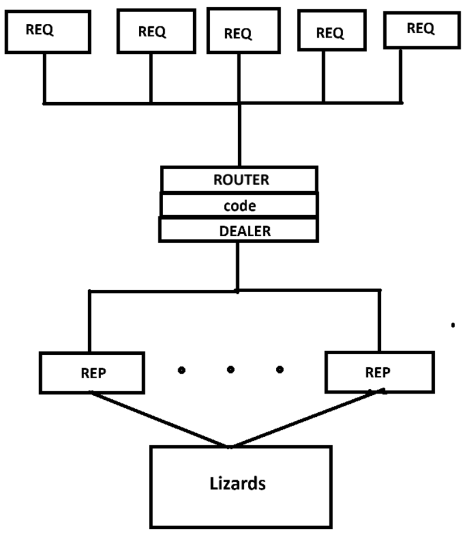
Any non valid messages are ignored.
Cheating is prevented with **passwords** for legit users that connect.

Architecture of the systems

The system is a distributed system composed of 3 clients and 1 server. roaches and wasps clients are bots and lizard clients are controlled by a human.
The server uses two router dealer architectures. One for roaches and wasps and other for the lizards (lizards are up to 26 and roaches + wasps are up to 300 (a third of window size)). Further this, it has a publisher subscriber architecture where the lizard client does one of the connections.
Roaches and wasps have a simple architecture with a request response connected to the router of the dealer.
Lizards have an architecture of a request response connected to the other router of the server, and in parallel a publisher subscriber connected to the publisher of the server.







Communication protocols

All the protocol buffer structures are preceded by an integer defined by a ENUM of protobuf correspondent to the operation

LIZARD_CONNECT = 0;
BOT_CONNECT = 1;
LIZARD_MOVE = 2;
BOT_MOVE = 3;
LIZARD_DISCONNECT = 4;
BOT_DISCONNECT = 5;

Any operation different from this is ignored

After the initial integer, the client sends a protocol buffer structure.
There are 3 types of protocol buffer structures: one is for client requests, other for server responses to the requests, and other to update the display through the publisher socket.

The request type has 3 optional parameter:
The string id is for roaches or wasps to tell the server their id

The password is for movement or disconnect requests
The direction is for movement requests, to tell the direction of the movement

```
    optional string id = 1;
    optional uint32 password = 2;
    optional uint32 direction = 3;
```

The response type has 3 optional and 1 required parameter:
success is required and tels if the request was successful.
The id is the id attributed to a lizard
The password is the password attributed to any client
The score is the score that lizards receive when they make a move request.

```
    required uint32 success = 1;
    optional string id = 2;
    optional uint32 password = 3;
    optional uint32 score = 4;
```

The display update type has 3 required and 1 optional:
String is the char to update at the position x,y of the screen
Score is when it is supposed to update the score of the lizard with id equal to the ch variable.

```
    required string ch = 1;
    required uint32 pos_x = 2;
    required uint32 pos_y = 3;
    optional uint32 score = 4;
```

LIZARD CONNECTION (REQ REP):
Sends the LIZARD_CONNECT enum
Sends the REQUEST_MESSAGE without any parameters
Receives a REPLY_MESSAGE with the id and password

ROACHES AND WASPS CONNECTION (REQ REP):
Client sends the BOT_CONNECT enum
Client sends the REQUEST_MESSAGE with id filled (between 1 and 5 for roaches and # for wasps)
Client receives a REPLY_MESSAGE with new password that will need in all other communication

LIZARD MOVE (REQ REP):
Client sends the LIZARD_MOVE enum
Client sends the REQUEST_MESSAGE with his id, password, and the direction to move
Client receives a REPLY_MESSAGE with success to 1 and score to its actual score

ROACHES AND WASPS MOVE (REQ REP):
Client sends the BOT_MOVE enum
Client sends the REQUEST_MESSAGE with his id, password and the direction to move
Client receives a REPLY_MESSAGE with success to 1.

Any of the requests before gets a REPLY_MESSAGE only with success set to 0 if the protocol buffer struct or the enum are wrong. Protocol buffer struct is wrong if it does not have the needed parameters

DISPLAY UPDATE (PUB SUB)
Client receives a DISPLAY_UPDATE_MESSAGE with the char to put in position x,y of the screen. when the char is between ‘a’ and ‘z’, also have the score to update in the scoreboard for the client with the correspondent char.

Changes between 2 versions

All the structures that were used to communicate between the programs were changed to protocol buffers. To make it easier to implement and reduce the repeated code we created a header file (snd_rcv_proto.h) with macros to pack, unpack, send and receive messages using protocol buffers.

- **Server changes**

The library of lizards and roaches were improved. We used the already made functions (lizard_here, find_lizard, draw_lizard, move_lizard) and the libraries were completely abstracted from main, passing all the responsibility to the individual libraries of each individual. The roach libraries turn into bots library because everything is so similar, except for some movements. We had to add wasp_here function and edit the move_bot function. Still about the wasps, we needed to change the function move_lizard from the lizard library.

For this libraries we created an auxiliar library with common function of the both libraries (lizard and bot): the published_update, new_position, fill_id_and_password and the structure used to store data from roaches, wasps and lizards (in the structure the lizard did not use moved in the first phase which was called eaten, in the second phase it uses everything. Roaches did not use the direction nor the points in the first part, now it does not use the direction, the timer moved, and the points).

To implement the threads we created a thread manager file where all the part of the while inside main was divided into different functions (lizard_handle, bot_handle, run_proxy).

We also added a function to the lizard library that controls the offline time of each lizard.

Now the lizard and the bots are handled in different sockets, in part1 they were handled in the same socket.

all this functions were added to abstract the lizard library

```
void init_lizards();
void *get_lizard(int id);
void *get_lizard_id(void *lizard_);
void fill_lizard_data(void *move, ReplyMessage *send_msg);
int valid_lizard(RequestMessage *recv_msg);
void delete_lizard(void *lizard_);
int stung_lizard(int pos_x, int pos_y);
void *offline_lizards(void *arg);
```

and all this were added to abstract the bot library

```
void init_bots();
int kill_roaches(int pos_x, int pos_y);
int bots_full();
int get_next_free_bot();
void fill_bot_data(int move, ReplyMessage *msg);
int roach_dead(int i);
void delete_bot(int i);
```

- **Lizard client changes**

While the communication with the server through ZeroMQ sockets, the movement with the arrow keys and disconnect with q or Q still hold, they are now handled by a thread (handle_lizard) created in the main function.
Another thread was created to display the board and scoreboard (handle_display). This thread works similarly to the remote display implemented in the first part (receives a display update message with the position and character to display as well as the score changes).

- **Roaches client changes**

Roaches client works in a similar way to the one used in the first part, although now all the process of deciding how many roaches, creating the roaches values and sending the connection and random movement messages is handled by a new thread created, while main checks if Enter is pressed to disconnect.

- **Wasps client**

A new client was created to create a new object in the game (wasps). This client works almost exactly like roaches client with the only difference being the initialization of the character used to represent the bot. In roaches client the character was generated a random value between 1 and 5 corresponding to the points they are worth. In the wasps client the character is simply **“#”**