TÉCNICO LISBOA

Systems Programming

MEEC

The Project - Report

Authors:

Miguel Lopes (100036) miguel.angelo.lopes@tecnico.ulisboa.pt
Rúben Cavaco (100082) ruben.neves.cavaco@tecnico.ulisboa.pt

2024/2025 – 1st Semester, P2

1 Introduction

The SpcInvdrs project is a two-part development exercise aimed at implementing and refining a multiplayer version of a simplified Space Invaders game. The game is a multiplayer experience where players control astronauts to navigate outer space, aiming to shoot randomly moving aliens with laser guns while competing to achieve the highest score.

In the first part of the project, a simpler version of the Space Invaders game was implemented, consisting of a server (game-server.c), a client (astronaut-client.c), and a display (outer-space-display.c). The server managed the game environment, where aliens moved randomly, and astronauts, controlled by players via the client, attempted to shoot the aliens using laser guns. The display provided a real-time view of the game, dynamically updating the game state.

In the second part of the project, the solution was refined to address challenges identified in the initial implementation. Threads were incorporated to improve the system's performance, and two additional components were developed. Players used an updated client application (astronaut-display-client.c) to control their astronauts and view the game environment simultaneously. Additionally, a new application (space-highscores) was introduced to maintain and display a database of the highest scores.

2 Implemented functionalities

2.1 Part A

	Not implemented	With faults	Totally Correct
game-server.c			
Suitable sockets			x
Assignment of Astronaut letters			x
Storage of clients Astronauts and Aliens			x
Management of astronauts			x
astronaut-client.c			
Connect			x
Movement			x
Zap			x
Disconnect			x
Display score			x
outer-space-display.c + display on game-server.c			
Correct update of screen			x
Zapps			x
Destruction of aliens			x
Rules			
Movement of Aliens			x
Astronaut zapping			x
Astronaut zapping (fire rate)			x
Astronaut zapping (0.5 second ray)			x
Aliens are destruction (points)			x
Astronauts stunning			x
Misc			
Messages validation			x
Optimization of communication			x
Code organization			x
Functions return values validation			x

Figure 1: Implemented functionalities (PART A)

2.2 Part B

	Not implemented	With faults	Totally Correct
game-server.c			
Suitable sockets			x
Disconnect of clients			x
Communication thread	x		
Aliens thread			x
Synchronization			x
astronaut-display-client.c			
Connect			x
Movement			x
Zap			x
Disconnect			x
Update of display			x
Display score			x
Threads			x
space-high-scores			
Client on other language			x
Protocolbuffer messages			x
Rules			
Movement of Aliens			x
Astronaut zapping			x
Astronaut zapping (fire rate)			x
Astronaut zapping (0.5 second ray)			x
Aliens destruction and points			x
Astronauts stunning			x
Aliens population recovery			
Misc			
Messages validation			x
Optimization of communication			x
Code organization			x
Functions return values validation			x

Figure 2: Implemented functionalities (PART B)

2.3 Description of faulty functionalities

3 PART A - Description of code

3.1 Data Types

In the project was used 3 structures. The first one is the `direction_t` enumeration, which represents the possible movement directions of the characters in the game. It includes the values `UP`, `DOWN`, `LEFT` and `RIGHT`.

The `remote_char_t` is used to encapsulate messages sent by clients to the server. This structure includes the `msg_type`, which indicates the type of action will be performed (0 for joining the game, 1 for moving, 2 for firing, and 3 for leaving). It also contains the `ch` to identify the character, a `ticket` array to store the client unique identifier to prevent cheating and a `direction` to specify the character's movement direction. So this structure ensures clear communication between the client and the server.

The last structure used was the `ch_info_t` to track the state of each character in the game. This structure includes a `ch` to identify the character, `pos_x` and `pos_y` to store the character's

current position, `score` to keep track the client points, `move` and `shoot` flags to indicate if the client can shoot or move, the `ticket` is used to associate the character with a client session and the last ones are `hit_time` and `shoot_time`, which record the last time the character was hit by a laser or fired a laser, respectively, enabling the implementation of features like stunning and rate limiting.

3.2 Function List

Before explaining the functions in the 3 files, we create a file with the common functions we use in the 3 files. This functions are `draw_board` function, where the functions is responsible for drawing the game board within a provided ncurses window. The `send_message` function, where this functions sends a message through a ZeroMQ socket. The `receive_message` function, where it is designed to receive a message through a ZeroMQ socket and the last function is `initialize_zmq_socket` where this function initializes a ZeroMQ socket and connects or binds it to a specified endpoint based on the provided parameters.

3.2.1 game-server.c

- `add_client`: Adds a new client to the game, initializing their position on the board, symbol, starting score, and movement and shooting states. Increments the counter of connected clients.
- `draw_score`: Updates the score window by displaying the scores of all connected players, clearing and redrawing the layout before inserting the data.
- `new_position`: Calculates the new position of an object on the board based on a direction (up, down, left, right), ensuring the coordinates remain within the valid game limits.
- `remove_client`: Removes a specific client from the list of connected clients, reorganizing the remaining elements and decrementing the client counter.
- `Shoot`: Manages the player's shot in a direction (horizontal or vertical), checking if aliens were hit, assigning points, stunning affected players and updating the board with the zap.
- `spawn_aliens`: Adds aliens to the board in random positions within a valid area, ensuring beforehand that the positions are unoccupied.
- `move_alien`: Moves the aliens on the board by choosing random directions and validating that the new positions are within limits and unoccupied before executing the movement.
- `move_player`: Moves a player on the board based on a given direction, ensuring they remain within the limits and updating their position on the board.

3.2.2 astronaut-client.c

In this program was used a `processKeyBoard` function is responsible for interpreting user input from the keyboard and updating the `remote_char_t` structure to reflect the desired action.

3.2.3 outer-space-display.c

The `deserialize_window` function was used in this program to take a serialized representation of a window's content stored in a buffer and apply it to a ncurses window.

3.3 Implementation details

3.3.1 Description of the implementation of the astronaut zapping fire rate:

In the project, the astronaut's fire rate is controlled to prevent consecutive shots. The `update_client_status` function monitors the time elapsed since the astronaut last shot. If the astronaut has recently fired, the system prevents them from shooting again until a specified cooldown period has passed.

This control mechanism relies on the `shoot_time` field in the `ch_info_t` structure. The function checks if the difference between the current time and the last shot time exceeds 3 seconds. If the cooldown has passed, the astronaut is allowed to shoot again, and the `shoot` flag is set to `TRUE`.

The shooting action itself is handled when the server receives a message with `msg_type == 2`. Upon receiving this message, the server checks if the astronaut is permitted to shoot, based on the `shoot` flag. If the astronaut is allowed to shoot (`shoot = TRUE`), the `shoot` flag is set to `FALSE` to prevent immediate subsequent shots, and the time of the shot is recorded with `clients[index].shoot_time = time(NULL)`. This timestamp is then used in the `update_client_status` function, which runs in the main game cycle to manage the shooting cooldown.

3.3.2 Description of the implementation of the astronaut zapping 0.5 second ray display:

When the astronaut shoots, a ray (represented as a '-' or '—') is briefly displayed on the game board to indicate the trajectory of the shot. The direction of the ray—horizontal or vertical—is determined by the astronaut's position and orientation. The ray is drawn by iterating through the rows or columns of the board, checking for empty spaces where it can be placed.

Once the ray is drawn, the game system triggers a delay using `usleep(500000)`, causing a 0.5-second pause. This delay simulates the short duration during which the ray is visible, allowing players to visually track the shot's trajectory.

After the delay, the ray is erased by checking the board again and replacing the temporary ray characters ('-' or '—') with spaces (' '). This ensures that the ray does not remain visible longer than intended, maintaining the board's cleanliness for the next action.

3.3.3 Description of the Implementation of the Astronauts Stunning (Immobility):

In the project, the stunning (immobility) of astronauts occurs when another astronaut's shot hits them. The logic for this is implemented in the `Shoot` function, where, upon detecting that an astronaut has been hit by a shot from another astronaut, their ability to move or shoot is temporarily disabled. This is achieved by setting the astronaut's `move` and `shoot` flags to `false`, effectively "stunning" them. As a result, the astronaut is immobilized and cannot take any actions until a set cooldown period has passed.

The exact logic for stunning is implemented in the part of the `Shoot` function where the positions of the astronaut and the shot are checked. If a shot from an astronaut intersects with another astronaut's position, that astronaut's `move` and `shoot` flags are set to `false`, and their `hit_time` is updated with the current time (using `time(NULL)`). This timestamp indicates the moment when the astronaut was hit.

The `update_client_status` function is responsible for allowing the stunned astronaut to resume their actions after the cooldown period. After 10 seconds, the astronaut is allowed to move and shoot again, ensuring that they are temporarily immobilized when hit, but are free to resume gameplay after the cooldown has elapsed.

4 PART B - Description of code

4.1 Data Types

The data types used in part A were also utilized in part B. Additionally, three new structs were introduced to pass the arguments required by the threads.

- The struct **zap_info** contains variables related to the game windows, the socket used to transmit information for the displays, the coordinates from where the zap was performed, and a boolean variable to indicate whether the shot was made horizontally or vertically. **Server Side**.
- The struct **alien_trial_t** contains variables related to the game windows, the number of aliens remaining in the game, and the socket used to transmit information for the displays. **Server Side**.
- The struct **disp_info** contains the variables related to the game windows. **Client-Display side**

4.2 Function List

4.2.1 game-server.c

remove_bullets, zap_effect, move_alien, update_aliens_alive, o shoot dividiu see em 2 funções que é o zap_effect e o remove_bullets

- `remove_bullets`: Removes the zap from the game board after a certain period. The function scans the area where the zap was (either horizontally or vertically) and clears the positions on the board where the zap or aliens were located.
- `zap_effect`: Applies the zap effect on the board (either horizontal or vertical) and updates the player's score.
- `move_alien`: In addition to the version specified on part A, it updates the number of aliens alive on the board. This function, which was previously a common function, was transformed into a thread to allow the aliens to move.

- `update_aliens_alive`: The function checks if the number of alive aliens has changed; if not, it counts the number of iterations without changes. Once this count reaches 10 (Because the aliens move every 1 second) new aliens are spawned based on 10% of the current number of alive aliens.

4.2.2 astronaut-display-client.c

As funções que estavam separadas em `outer-space-display` e `astronaut-client` agora foram integradas em um único programa, onde a parte do `astronaut-client` continua na função `main` e a parte do `outer-space-display` foi movida para uma thread dedicada, chamada `display`, responsável por atualizar as janelas do jogo e pontuação em tempo real.

4.2.3 space-high-scores.c

Implemented in Python, the program receive messages of type `ScoreUpdates`, which contain information about astronauts' identifiers (letters) and their respective scores. The program processes these messages in real-time using Protocol Buffers for deserialization, updating a dictionary that stores the scores of each astronaut. The scores are then displayed in a formatted table on the terminal, with the display refreshing as new updates are received.

4.3 Implementation details

The astronaut zapping fire rate and the stunning part were implemented the same way as it was done in part A. The astronaut's zapping mechanism, featuring a 0.5-second ray delay, was split into: `shot effect` and `bullet removal`. The shot effect is similar to part A; the bullet appears on the board, and the aliens' hit are removed, awarding points to the player. The second component utilizes a thread to time the interval and erases the shot traces from the board.

The population recovery is a function done in the `Move_Alien` thread. After 10 iterations without the population decreasing, the number of aliens increases by 10%.

4.4 Threads

- **game-server.c** has two threads. One is responsible for the movement of the aliens and the population recovery. The other one is responsible for the astronaut zap, it removes the bullet trails after the 0.5 second window.

- **astronaut-display-client.c** has one thread responsible for handling the messages from the publisher. It receives messages about the information about the board and updates the display accordingly.

4.5 Shared variables

A struct containing the {board window, score window, aliens alive, publisher socket} is shared and accessed by the *alien_thread*. This struct plays a role in the alien thread by updating the windows to reflect alien movements, to send information about the board and score to the displays and also to aid in the alien repopulation.

A struct containing {board window, score window, coords of the player, publisher socket} is shared in the *shoot_thread*. The purpose of this struct is to eliminate the zap trail from the board window after the 0.5s delay from the location where the player executed the shot and to transmit the information to the displays.

4.6 Synchronization

Mutexes were used for the critical regions and were used on the server side, even though a thread was used on the *astronaut-client-display.c* a mutex was not needed because the thread is responsible only for the display part while the main code processes the part about the client.

The server employed a mutex to synchronize the aliens' movements with the zaps. In the absence of this mutex, aliens could occasionally evade the zap and survive, disrupting the count of living aliens.

5 PART A - Communication

5.1 Sockets

Through the use of Zero_MQ communications, a rep-req was used to handle the communication with the clients and a pub-sub for handling the display data.

5.2 Transferred Data

The main message sent through the req-rep socket contains details about the ship sending the message, including its current points, the action it intends to perform, the direction it can move (vertical or horizontal), and its encryption key. For display purposes, we sent a string that contained every *char* on the board, including aliens, players and the zaps.

5.3 Error treatment

While an unreadable or undeliverable message is managed by exit(1). On the client part, exit(1) still applies, it should have been implemented the client disconnect from the server before exiting.

6 PART B - Communication

6.1 Sockets

In this part we send a message with a topic in the publisher socket to send the scores to the program in python with protobuf, we need the topic because if we doesn't had the topic the program in python would also receive other messages not relevant to update the scores.

6.2 Transferred Data

In this system, the C program acts as the publisher, sending real-time score updates to the Python client using ZeroMQ. The C program serializes the score data into Protocol Buffers format (`ScoreUpdates`), which is then sent over ZeroMQ to the Python subscriber. The Python program receives the serialized message, deserializes it using Protocol Buffers, and updates the display with the new scores. The `ScoreUpdates` holds multiple `ScoreUpdate` messages, the same as the number of clients in the game. The `ScoreUpdate` contains the information about the astronaut character and its score.

6.3 Error treatment

The only change in this part since thread is used was error handling when a thread or mutex creation fails is managed by *EXIT_FAILURE*.

7 Conclusion

This project successfully developed a multiplayer game system with real-time score updates, leveraging message-based communication, as well as threads and mutexes for efficient synchronization. The main objective was improve the game experience when compared to the first part using the tools given by the teacher. With this project, we gained an understanding of the importance of these tools and learned how to use them.