TÉCNICO LISBOA

Instituto Superior Técnico

Programação de Sistemas / Systems Programming

MEEC

Project Assignment: lizardsNroachesNwasps, 2022/2023

Group **10**

Students:
Tomás Marques Videira Fonseca | 66325
Afonso Brito Caiado Correia Alemão | 96135
Rui Pedro Canário Daniel | 96317

Teachers:
João Nuno De Oliveira e Silva
Alexandra Sofia Martins de Carvalho

January 7, 2024

1 Introduction

In this project, a variation of the snake game has been implemented where lizards walk in a field and consume roaches. Additionally, there are wasps. The server is responsible for managing the field, while programs/clients control the movements of lizards, roaches, and wasps, sending instructions to the server. These entities can also connect and disconnect from the game. The applications displaying the field's progression throughout the game are integrated with a display app in Part A and with lizard controllers in Part B.

2 The Game

In this section, the implemented functionalities are described, along with the architecture of the systems, the communication protocols (messages and interaction), and a critical analysis of the changes between the two versions. The various modules are presented with their corresponding APIs, message contents, and interaction diagrams between components.

2.1 Implemented Functionalities

Table 1 presents the implemented functionalities in both Part A and Part B of this project. In both cases, we successfully implemented all the required functionalities, and they work correctly. In bold, the differences between both parts are highlighted.

Table 1: Implemented functionalities for Part A and Part B.

Part A	**Part B**
Server - lizardsNroaches	Server - lizardsNroachesNwasps
Send field updates as a publisher to the display app using ZeroMQ TCP sockets.	Send field updates as a publisher to the **lizard clients** using ZeroMQ TCP sockets.
Receive client messages using ZeroMQ TCP sockets.	Receive client messages using ZeroMQ TCP sockets **with threads. 4 threads for handling messages from all the lizard clients. 1 thread for handling messages from the roaches and wasps clients.**
	The interface between the server and the clients is supported by Protocol Buffers.
Connect lizard.	Connect lizard.
Move lizard.	Move lizard.
Disconnect lizard.	Disconnect lizard.
Connect roach client.	Connect roach client.
Move roaches.	Move roaches.
	Disconnect roach client.
	Connect wasp client.
	Move wasps.
	Disconnect wasp client.
Ncurses interface to display field and stats.	Ncurses interface to display the field and stats **(1 thread for this purpose)**.
Maximum number of lizards and roaches as stated in [2].	Maximum number of lizards, roaches, **and wasps** as stated in [2].
Cheating robustness: check if the client connecting has a valid address (unique in the game).	Cheating robustness: check if the client connecting has a valid address (unique in the game).
Read the client port and the display port as input line arguments.	Read the **client port for lizards**, the **client port for roaches and wasps**, and the display port as input line arguments.
The game finishes when a lizard reaches a score greater than or equal to 50.	The game finishes when a lizard reaches a score greater than or equal to 50.
	A lizard loses when it reaches a negative score.
Split health when lizards collide.	Split health when lizards collide.
Lizards eat roaches, eliminating them.	Lizards eat roaches, eliminating them.
Resurrect a roach after being eliminated for 5 seconds.	**1 thread to resurrect roaches that have been eliminated for 5 seconds and to disconnect client after 1 minute of inactivity.**
	Wasps stung lizards.

Lizard-client	Lizard-client
Interact with the server using ZeroMQ TCP sockets.	Interact with the server using ZeroMQ TCP sockets.
	The interface between the server and the lizard client is supported by Protocol Buffers.
Send a ‘connect lizard’ request to the server and receive a response.	Send a ‘connect lizard’ request to the server and receive a response.
Utilize the Ncurses interface to read lizard movements.	Utilize the Ncurses interface to read lizard movements.
Send lizard movements to the server and receive response.	Send lizard movements to the server and receive response.
Send a ‘disconnect lizard’ request to the server and receive a response.	Send a ‘disconnect lizard’ request to the server and receive a response.
Read the client address and client port as input line arguments.	Read the client address, the **client port for lizards**, and the **display port** as input line arguments.
	Receive field data as a subscriber using ZeroMQ TCP sockets.
	Ncurses interface to display the field and stats (1 thread for this purpose).
	Update the field and stats.

Roaches-client	Roaches-client
Interact with the server using ZeroMQ TCP sockets.	Interact with the server using ZeroMQ TCP sockets.
	The interface between the server and the roaches client is supported by Protocol Buffers.
Send a ‘connect roach’ request to the server and receive a response.	Send a ‘connect roach’ request to the server and receive a response.
Generate roach movements randomly.	Generate roach movements randomly.
Send roach movements to the server and receive response.	Send roach movements to the server and receive response.
Read the client address and client port as input line arguments.	Read the client address and **client port for roaches and wasps** as input line arguments.
	Implementation using a different programming language: Python.

Display-app	**Wasps-client**
Receive field data as a subscriber using ZeroMQ TCP sockets.	**Interact with the server using ZeroMQ TCP sockets.**
Ncurses interface to display the field and stats.	**The interface between the server and the wasps client is supported by Protocol Buffers.**
Update the field and stats.	**Send a ‘connect wasp’ request to the server and receive a response.**
Read the display address and display port as input line arguments.	**Generate wasp movements randomly.**
	Send wasp movements to the server and receive response.
	Read the client address and client port for roaches and wasps as input line arguments.
	Implementation using a different programming language: Python.

2.2 Systems Architecture

The **Project - Part A** block diagram in Fig. 1 represents the architecture of the Part-A system.

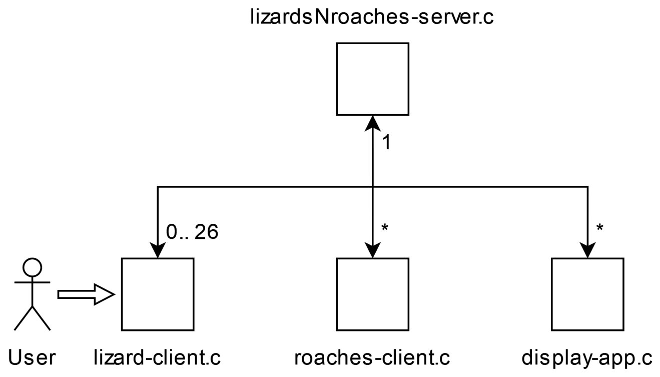
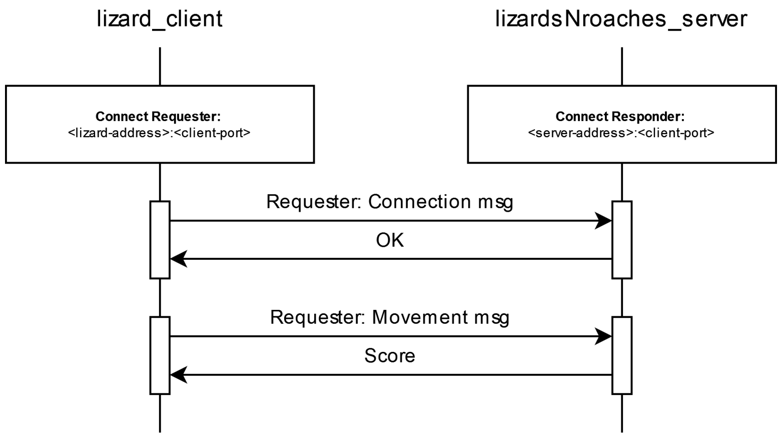


Figure 1: Project - Part A block diagram.

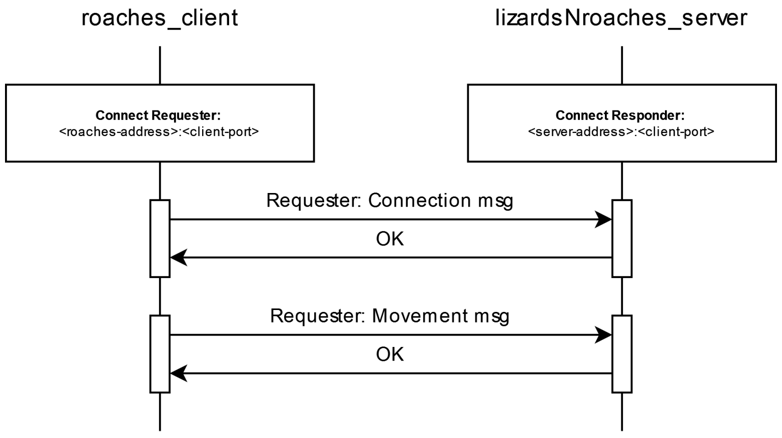
Here, a server communicates with two types of clients, roaches and lizards, using the `REQ/REP` protocol with ZeroMQ TCP sockets. Furthermore, the server also sends messages to a display app using the `PUB/SUB` protocol with ZeroMQ TCP sockets.

The maximum number of connected lizard clients is 26, and the maximum number of roaches is 1/3 of the field size. On the other hand, publishing to the display-app is only available for subscribers with the correct password that the server defines at the beginning of its execution.

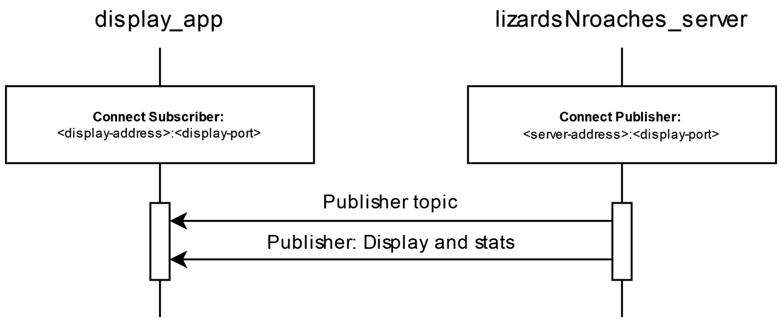
The **Project - Part A** sequence diagram is represented Fig. 2.


lizard_client
lizardsNroaches_server
Connect Requester: <lizard-address>:<client-port>
Connect Responder: <server-address>:<client-port>
Requester: Connection msg
OK
Requester: Movement msg
Score

(a): Lizard-client and server communication.


roaches_client
lizardsNroaches_server
Connect Requester: <roaches-address>:<client-port>
Connect Responder: <server-address>:<client-port>
Requester: Connection msg
OK
Requester: Movement msg
OK

(b): Roaches-client and server communication.


display_app
lizardsNroaches_server
Connect Subscriber: <display-address>:<display-port>
Connect Publisher: <server-address>:<display-port>
Publisher topic
Publisher: Display and stats

(c): Display-app and server communication.

Figure 2: Project - Part A sequence diagrams.

The **Project - Part B** block diagram in Fig. 3 represents the architecture of the Part-B system. Here, the server communicates with three types of clients: roaches, wasps, and lizards. The server uses 1 thread for a REQ-REP protocol to communicate with roaches and wasps, and it uses 4 threads to communicate with lizards using a proxy that intermediates the processing of messages between the REQ-REP protocol. Furthermore, the server also sends messages to lizards using the PUB-SUB protocol. In all these communications, it uses ZeroMQ TCP sockets.

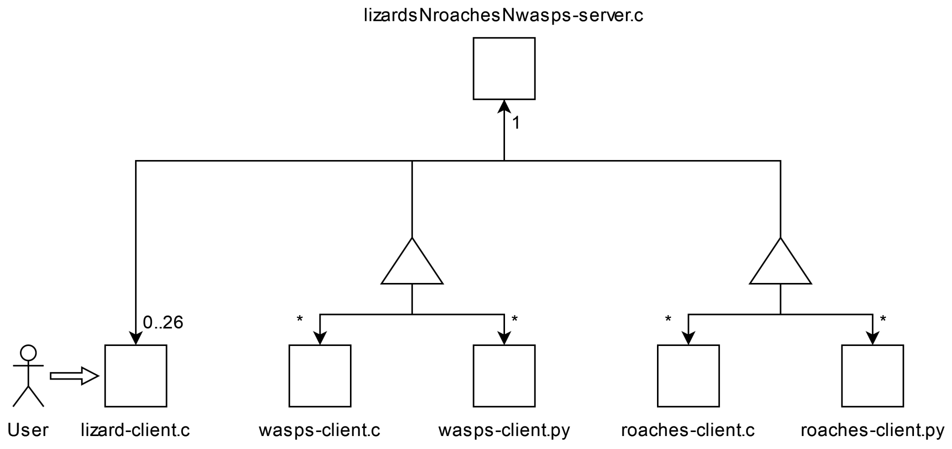
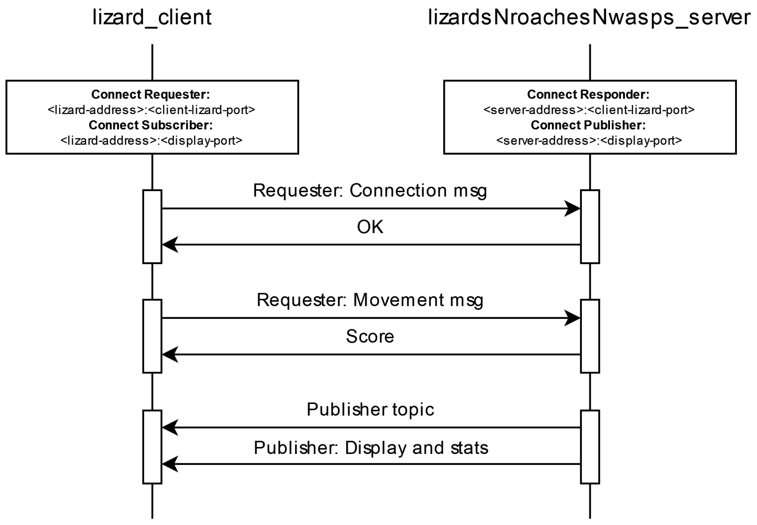


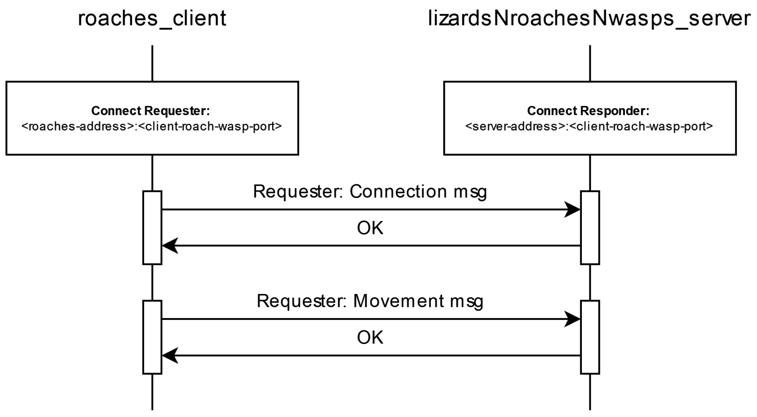
Figure 3: Project - Part B block diagram.

The maximum number of connected lizard clients is 26, and the maximum sum of roaches and wasps is 1/3 of the field size. On the lizard-client, 1 thread is responsible for the Ncurses interface to display the field and stats, but that is only available for subscribers with the correct password. Additionally, there is a version for both the wasps-client and roaches-client using Python.

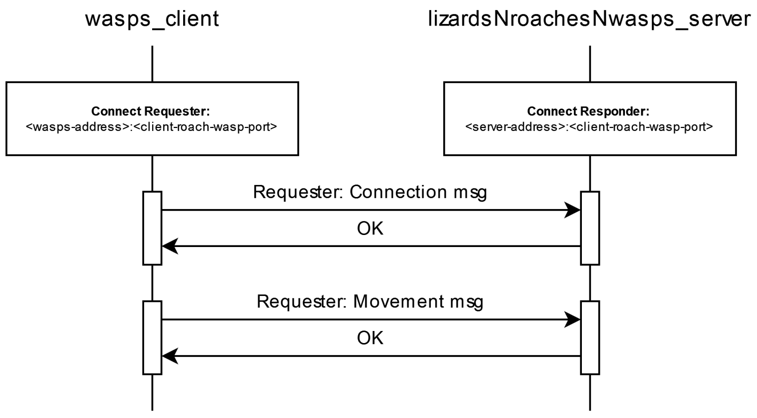
The **Project - Part B** sequence diagram is represented Fig. 4.


lizard_client
lizardsNroachesNwasps_server
Connect Requester:
<lizard-address>:<client-lizard-port>
Connect Subscriber:
<lizard-address>:<display-port>
Connect Responder:
<server-address>:<client-lizard-port>
Connect Publisher:
<server-address>:<display-port>
Requester: Connection msg
OK
Requester: Movement msg
Score
Publisher topic
Publisher: Display and stats

(a): Lizard-client and server communication.


roaches_client
lizardsNroachesNwasps_server
Connect Requester:
<roaches-address>:<client-roach-wasp-port>
Connect Responder:
<server-address>:<client-roach-wasp-port>
Requester: Connection msg
OK
Requester: Movement msg
OK

(b): Roaches-client and server communication.


wasps_client
lizardsNroachesNwasps_server
Connect Requester:
<wasps-address>:<client-roach-wasp-port>
Connect Responder:
<server-address>:<client-roach-wasp-port>
Requester: Connection msg
OK
Requester: Movement msg
OK

(c): Wasp-client and server communication.

Figure 4: Project - Part B sequence diagrams.

Both the server and the lizard-client use multi-threading. The server has 8 threads: the main thread, 4 threads for handling messages from all the lizard clients, 1 thread for handling messages from the roaches and wasps clients, 1 thread for displaying the field, and 1 thread for resurrecting roaches, and managing timeouts of inactive clients. On the other hand, lizard-client has 2 threads: the main thread used for reading inputs from the user, and 1 thread for displaying the field.

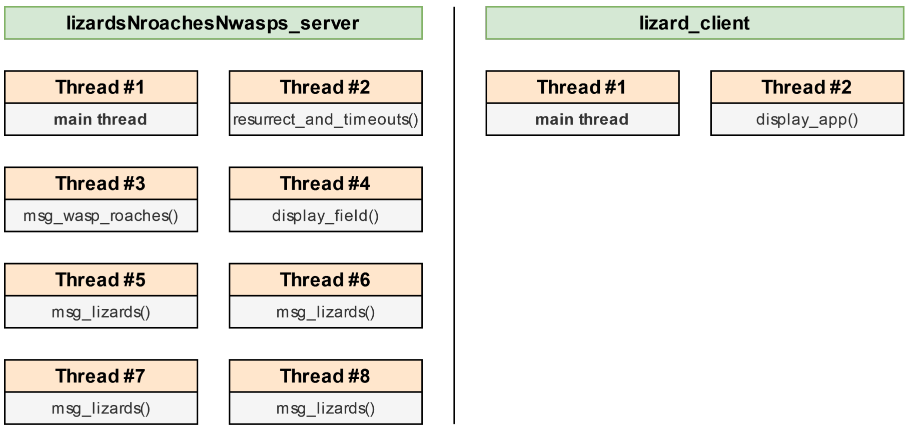


Figure 5: Server and lizard-client thread distribution.

The thread distributions, represented in Fig. 5, are as follows:

- `lizardsNroachesNwasps_server`:
  - **Thread #1:** main thread
    Responsible for the main function, including thread creation and joining, as well as creating a proxy that intermediates the processing of messages between the `REQ/REP` protocol to communicate with lizards.
  - **Thread #2:** `resurrect_and_timeouts()`
    This function is responsible for the resurrection of the roaches after 5 seconds of their death time, and it also handles the disconnection of inactive clients. If the server does not receive a message from a client for more than 60 seconds, the client is disconnected.
  - **Thread #3:** `msg_wasp_roaches()`
    Handles and processes the received messages from roaches and wasps clients.
  - **Thread #4:** `display_field()`
    Displays the game field and the stats, updating in real time.
  - **Thread #5-8:** `msg_lizards()`
    Handles and processes the received messages from lizard clients.

- `lizard_client`:
  - **Thread #1:** main thread
    It is responsible for the main function and thread creation and join. After thread creation, it handles the user input cursor keys. Each key, if valid, is sent to the server. If the user presses the key `q` or `Q`, the lizard-client exits gracefully, sending a disconnect message to the server.
  - **Thread #2:** `display_app()`
    Displays the game field and the stats, updating in real-time.

2.3 Communication Protocols

2.3.1 Part A

The lizardsNroaches-server is a server-side application designed to manage communications with all client entities. It manages the game board or field, which includes both lizards and roaches, and is responsible for sending updates to the display-app.

The server and the lizard client utilize a specific protocol to establish a connection. The server, operating in a more stable environment, binds its socket using a wildcard (*), indicating its willingness to accept connections on the specified port from any valid network interface, and defines a specific port number (<`client-port`>) for the connection. This port number is typically a numeric value above 1024, chosen to avoid conflict with well-known ports used by other services.

On the client side, the lizard client is designed to connect to the server through an address and port number specified by the user. The address can be any valid one where the server is accessible, and the port number must match the one defined by the server (<`client-port`>). This matching of port numbers is crucial for the successful establishment of the connection.

Before discussing the specific communication pattern and the types of messages exchanged, it's important to note that both lizard and roach clients interact with the server utilizing a custom-designed structure known as `remote_char_t`. This structure includes an integer that defines the message type and a unique client identifier of the `uint32_t` type. It also contains an array that holds up to 10 characters, alongside a variable that indicates how many of these character positions are actually filled. This array allows a single client to hold different roach characters. In addition, there is an array comprising 10 elements based on an enumerated type. This array serves a dual purpose: it represents the movement direction of the lizard client, indicated by the first element in the array, and in the case of a roach client, it tracks the various movements of different roaches. This `remote_char_t` structure efficiently bundles together all the essential elements for the lizard's messages, such as type, identity, character content, length, and directional information, in a cohesive and organized manner.

In this communication setup, the lizard-client, also known as the requester, captures the user's key presses and sends them to the server using `zmq_send` within the `REQ/REP` (Request/Respond) pattern of ZeroMQ, represented in Fig. 6. The server (responder) then responds with the user's score via `zmq_recv`. This interaction employs TCP as the transport protocol, enabling communication between server and client processes on different machines over the internet.

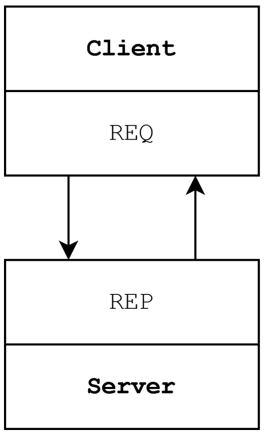


Figure 6: ZeroMQ Client-Server Request-Reply example.

In the interaction between the lizard-client and the server, the lizard-client transmits messages of three distinct types:

- Connection Messages (`message_type = 2`): The lizard_client introduces itself to the server. After assigning an identifier to the client, the server responds with an `‘ok’` message. A successful connection is indicated by the message value being different from the integer `‘?’`. If the message value is `‘?’`, the connection attempt has failed.
- Key Press Messages (`message_type = 3`): The lizard-client sends the keys pressed by the user to the server. Upon receiving this, the server processes the lizard-client's board position, updating both position and score. It then sends the updated score back to the client. A score different from -1 indicates a successful update, whereas a score of -1 signifies an unfulfilled request.
- Disconnection Messages (`message_type = 4`): The lizard_client requests its disconnection from the game to the server by pressing `Q` or `q`. After removing the lizard from the game board, the server responds with an `‘ok’` message.

The interaction between the roaches-client and the server mirrors that of the lizard-clients and server, employing the same communication pattern and transport protocol. Additionally, there are two unique message types involved: `msg_type = 0` and `msg_type = 1`. With `msg_type = 0`, the roaches-client establishes a connection with the server, introducing itself in a manner similar to that previously described for the lizard. For `msg_type = 1`, the roaches-client transmits random movement directions for each active roach to the server. The server then updates its positions on the board and responds with an `‘ok’` message to confirm if the process was successful.

The display-app is an application that visually presents the game board, mirroring the lizardsNroaches-server. It displays all game participants, updating in real-time as the server processes movements and score changes. When a lizard-client makes a move, the server updates the game board, adjusts scores, responds back to the client, and sends a field-update message to the display-app to reflect these changes. Both the lizardsNroaches-server and display-app are responsible for showing the current game status, including the scores of all participating lizards.

In this system, the communication between the server and the display-app operates on a zmq `PUB-SUB` (Publish-Subscribe) protocol. Here, the server acts as the publisher, and the display-app functions as the subscriber. With this setup, each message the server sends (via `ZMQ_PUB` sockets) is assigned a specific topic, essentially functioning as a broadcast password. The display-app needs to connect to the server using a uniquely defined port, different from those used for lizard and roach communications, and select a topic or password to subscribe to.

In the PUB-SUB model, ZeroMQ efficiently filters and delivers messages to clients based on their subscribed topics. Each message from the server, tagged with a topic (password), is distributed to all clients subscribed to that topic. Thus, the display-app client can access the server's board information only if it subscribes using the correct password.

The server communicates with the display-app client using a specific message structure, `msg`, which includes the 2D positions on the board that require updates, the value of each position on the board, and an array. This array contains structures with detailed data, including each lizard-client's score, enabling the display-app to display these scores accurately.

2.3.2 Part B

In this updated configuration, the server has been adapted to a multi-threaded environment, featuring four threads dedicated to processing messages from all the lizard-client's, and an additional thread for managing messages from the roaches-client's or wasps-client's. Moreover, the communication interface between the server and the wasps-client/roaches-client now employs Protocol Buffers for message encoding, with the data structures involved in this communication being defined in a `.proto` file.

The `.proto` file outlines a few data structures using Protocol Buffers (Proto2 syntax). It defines a `Direction` enumeration with movement directions like `UP`, `DOWN`, `LEFT`, `RIGHT`, and `NONE`. There's a `RemoteChar` message, resembling the previously described `remote_char_t`, that includes a message type as an `int32`, a `uint32` id, a string for characters, an `int32` for the number of characters, and a repeated field for multiple direction values. The `'ok'` message is a simple message with an `int32` to signify an acknowledgment. Lastly, the `score_message` carries a double representing a score.

In the multi-threaded server setup, for the threads performing communications with the lizard clients, the server utilizes a `ZMQ_ROUTER` socket to interface with the client's `ZMQ_REQ` sockets and a `ZMQ_DEALER` socket to distribute incoming requests across multiple server threads, as represented in Fig. 7. Each server thread operates with a `ZMQ_REP` socket and connects to the `ZMQ_DEALER` socket via the in-process communication protocol (`inproc`). The server employs a proxy to bridge the `ZMQ_ROUTER` and `ZMQ_DEALER` sockets, ensuring that each client request is evenly and efficiently handled by the available threads. This architecture allows for concurrent processing of messages, maintaining efficient communication flow and response delivery back to the clients.

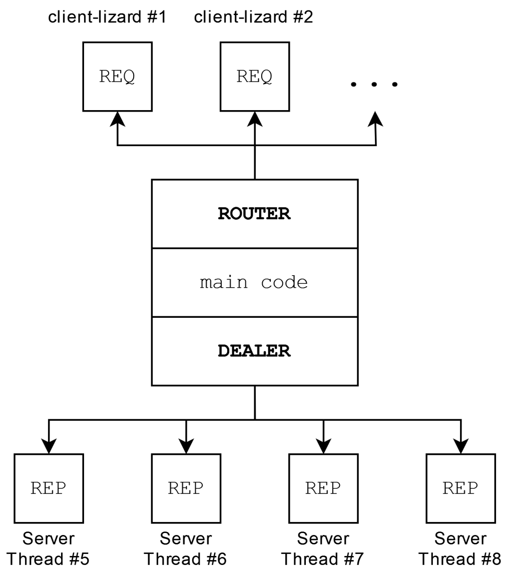


Figure 7: Multi-threaded ZeroMQ Request-Reply broker architecture diagram.

After eliminating the display-app, the responsibility for displaying the game board now falls to each lizard-client, conditional on the correct broadcast password being entered. To implement this, modifications were made on both client and server ends. On the server's end, a specific thread is tasked with sending a `msg` type message to all lizard-clients who have keyed in the correct password, utilizing a `PUB/SUB` pattern where the server publishes and the lizard-clients subscribe. On the client side, two threads are set up: one handles sending movement directions to the server via `REQ/REP` protocol, and the other subscribes to the server's broadcast messages in the `PUB/SUB` setup to visualize the game on the client interface.

In the server configuration, an additional thread (thread #3) is designated to allow communication between the server and wasps clients, and between the server and roaches clients, utilizing the `REQ-REP` protocol with a different socket from the one that threads #5 to #8 use. This communication follows the same method as the one outlined earlier, in Part A, for the connection between the server and roaches.

2.4 Clients Implementation in Python

Leveraging the serialization and deserialization features of Protocol Buffers, communication between the C-based server and the roach/wasp clients implemented in Python was made. Two separate Python implementations were developed—one for the wasp client and another for the roach client—with the intention of replicating the functionality provided by the original C implementation. By using the same `.proto` file across both the C server, Python clients, and C clients, a common communication schema is maintained, ensuring compatibility and enabling the server and clients to interpret the exchanged messages correctly, despite being written in different programming languages. This cross-language communication is facilitated by the language-agnostic nature of Protocol Buffers.

3 Conclusion

All the required functionalities have been implemented using the learned tools in PSis. These include interacting from the clients with the server using ZeroMQ TCP sockets. Furthermore, the interface between the server and the Wasps-client/Roaches-client is supported by Protocol Buffers for message encoding. Both the Wasps-client and Roaches-client have implementations in both C and Python. The server is multi-threaded, with 4 threads for handling messages from all the Lizard clients, 1 thread for handling messages from the Roaches-clients or Wasps-clients, 1 thread for dealing with abrupt exits by computing timeouts, and 1 thread to display the field.

Threaded programming enables parallelism, allowing the server to execute multiple tasks simultaneously on multi-core processors, making better use of available hardware resources. This way, it is possible to process multiple client messages simultaneously instead of one at a time. The asynchronous loading of the field avoids freezing during gameplay, providing the lizard client with the game in real-time. It provides low-latency responses to player actions, ensuring responsiveness. Furthermore, the chosen distribution of work per thread avoids wasting resources. The implementation ensures correctness and thread safety by utilizing mutexes.

References

[1] Systems Programming Slides - Slides of Theoretical Classes 2023/2024, 1st Semester (MEEC);

[2] 1st Semester 2023/2024, Systems Programming (IST, 2023-24), Project Assignment lizardsNroachesNwasps;