Multi-Threaded Dictionary Server

Nico Eka Dinata (770318) — COMP90015 2022 Assignment 1

Problem Context

The deliverable of this assignment is to build a client-server program representing a dictionary system where users are able to query the definitions of words, add and update definitions, and remove words (dictionary operations).

The requirements for the server program include the following:

- The server program has to make explicit use of sockets and threads to be able to service multiple concurrent client requests
- Client-server communication has to be reliable
- Errors must be handled gracefully via exception handling

On the other hand, the client program has the following requirements:

- Communication with the server has to make explicit use of sockets
- An appropriate GUI has to be implemented to allow carrying out the aforementioned dictionary operations
- The user has to be informed of any errors that occur while performing any operations

System Components

The whole system consists of two independent but interacting components: the server subsystem and the client subsystem.

The server subsystem contains the following major modules:

- `DictionaryServer`: the main module of the subsystem, handling the logic of setting up the communication layer and distributing tasks to worker threads in a thread pool architecture
- `RequestHandler`: responsible for receiving and parsing client requests and returning an appropriate response
- `DictionaryHandler`: manages the logic of handling all supported dictionary operations
- `ExceptionHandler`: responsible for formatting the various exceptions that could be thrown into a more consistent and user-friendly format that the server could then send to the client
- `ServerGUI`: specifies how the GUI of the server program looks and behaves

The client subsystem, on the other hand, contains the following modules:

- `DictionaryClient`: main module of this subsystem, handling the logic of setting up communications and representing the interface between the server and the client GUI
- `ClientGUI`: specifies how the GUI of the client program looks and behaves

System Design Diagrams

The following are the two class diagrams of each of the subsystems in the system.

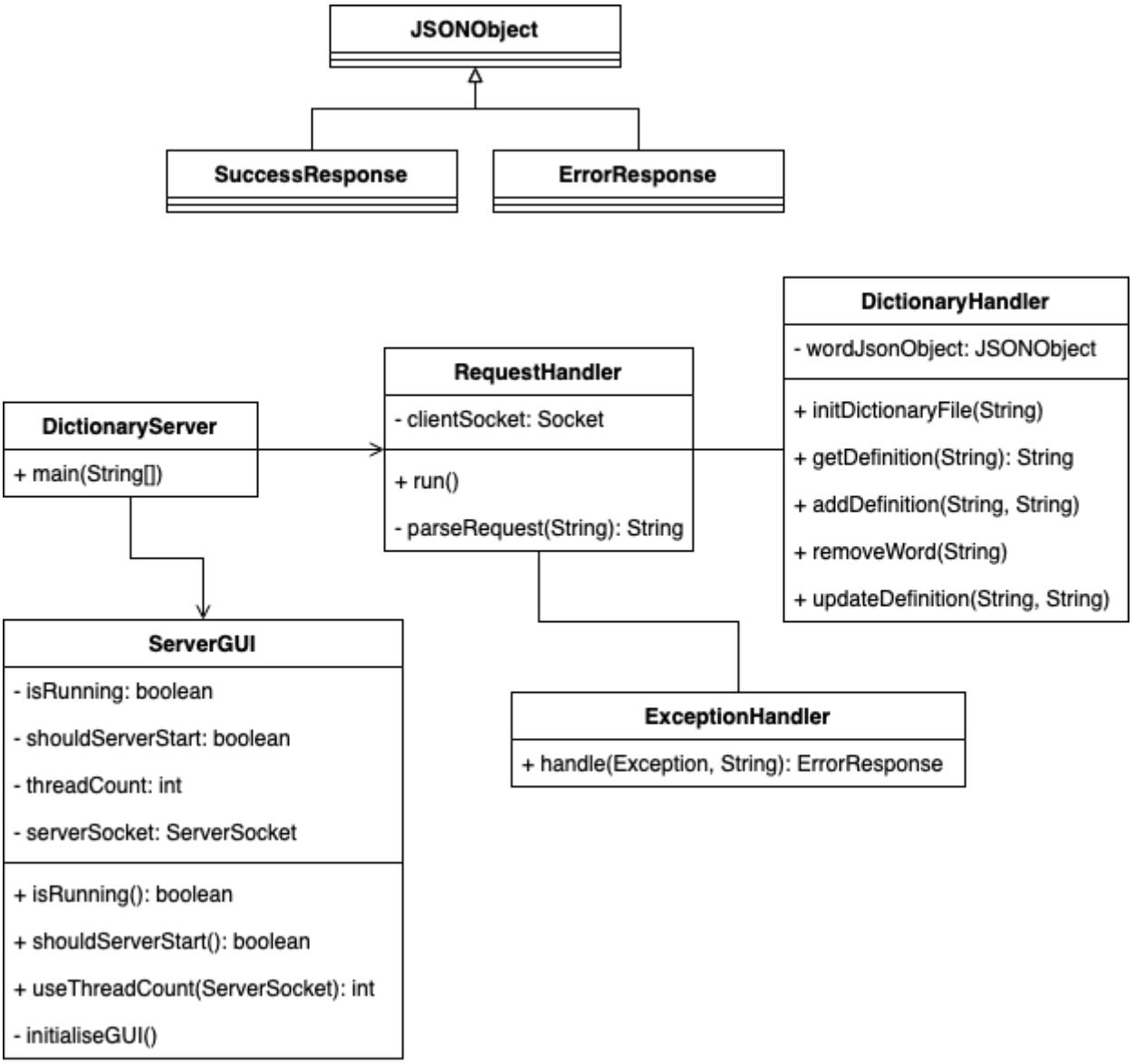

Figure 1: Class diagram of server subsystem

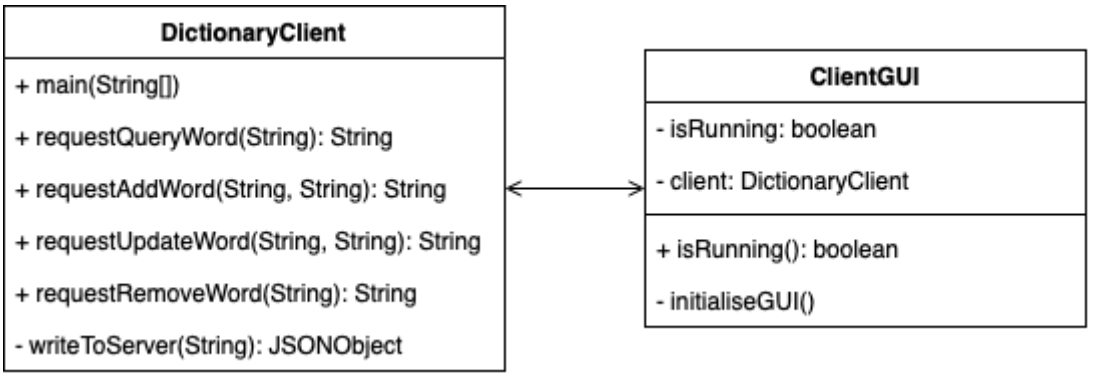

Figure 2: Class diagram of client subsystem

The following are simplified (for readability purposes) sequence diagrams of the four dictionary operations that are supported by the system.

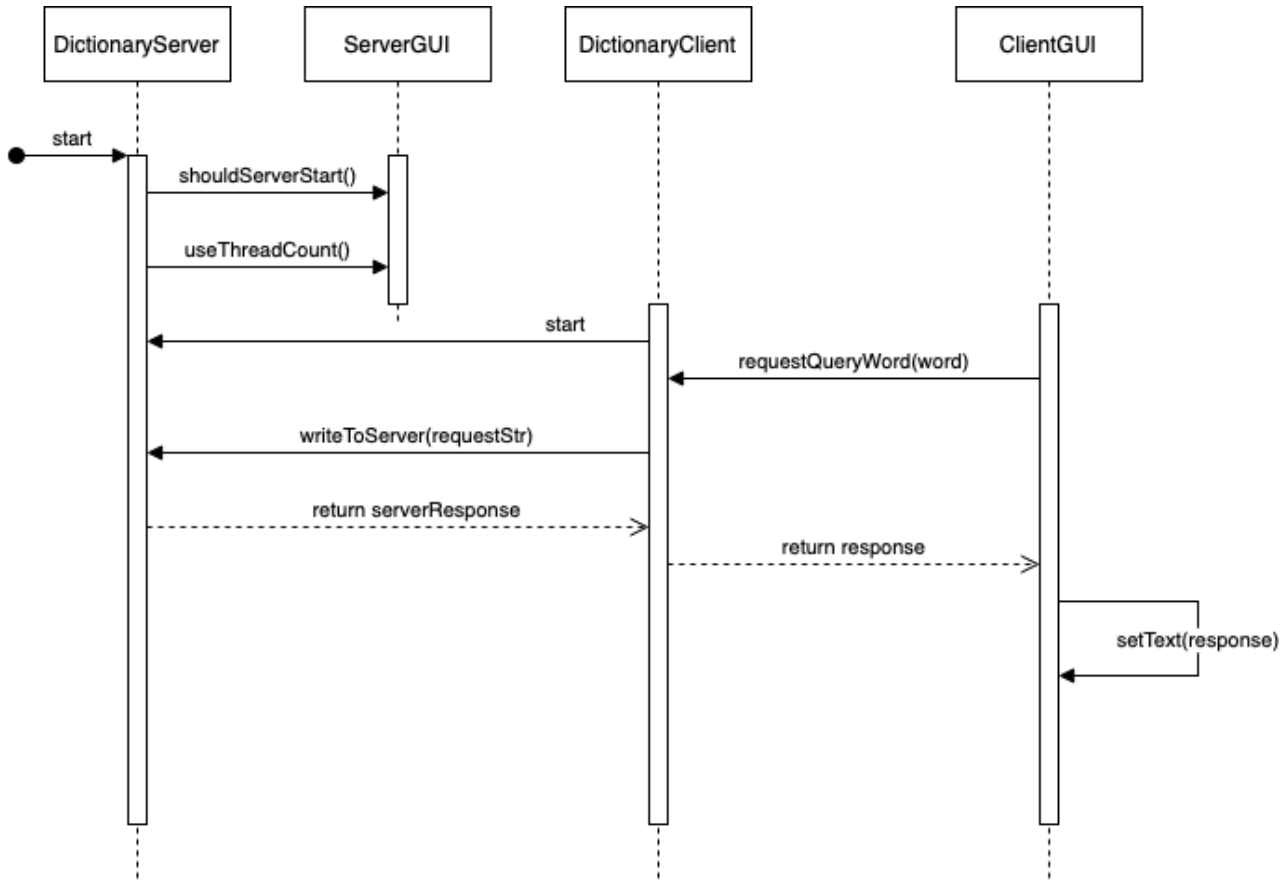

Figure 3: Sequence diagram for querying for a word's definition

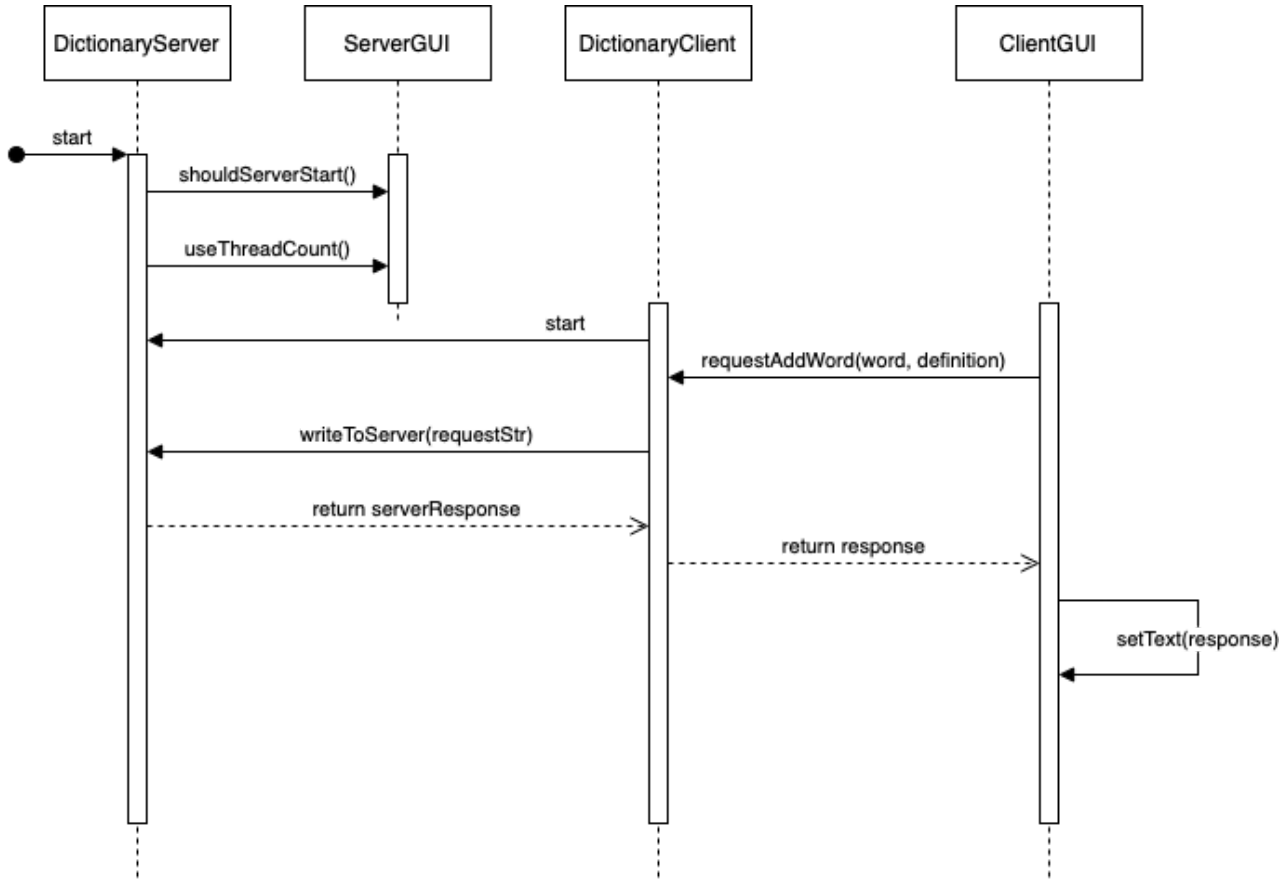

Figure 4: Sequence diagram for adding a new word's definition

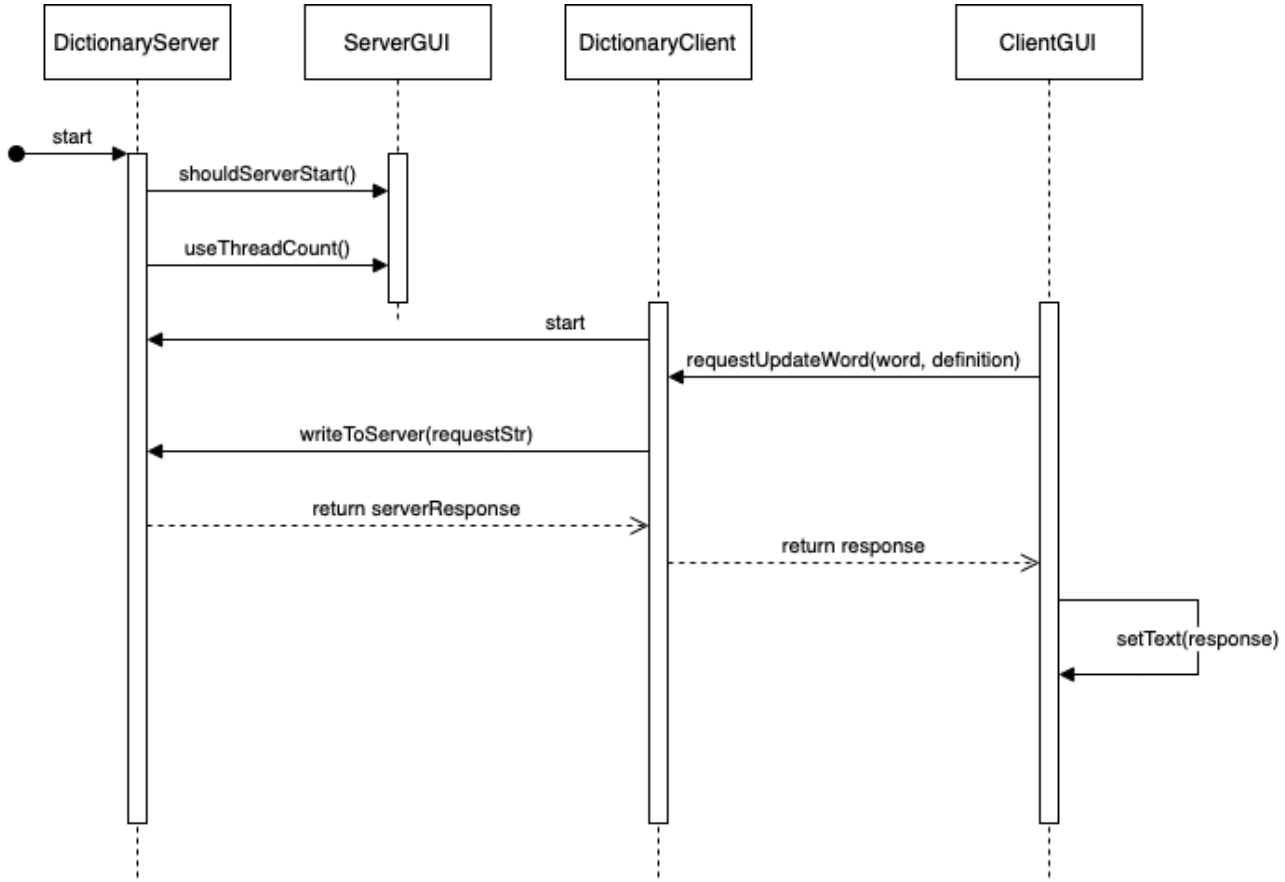

Figure 5: Sequence diagram for updating an existing word's definition

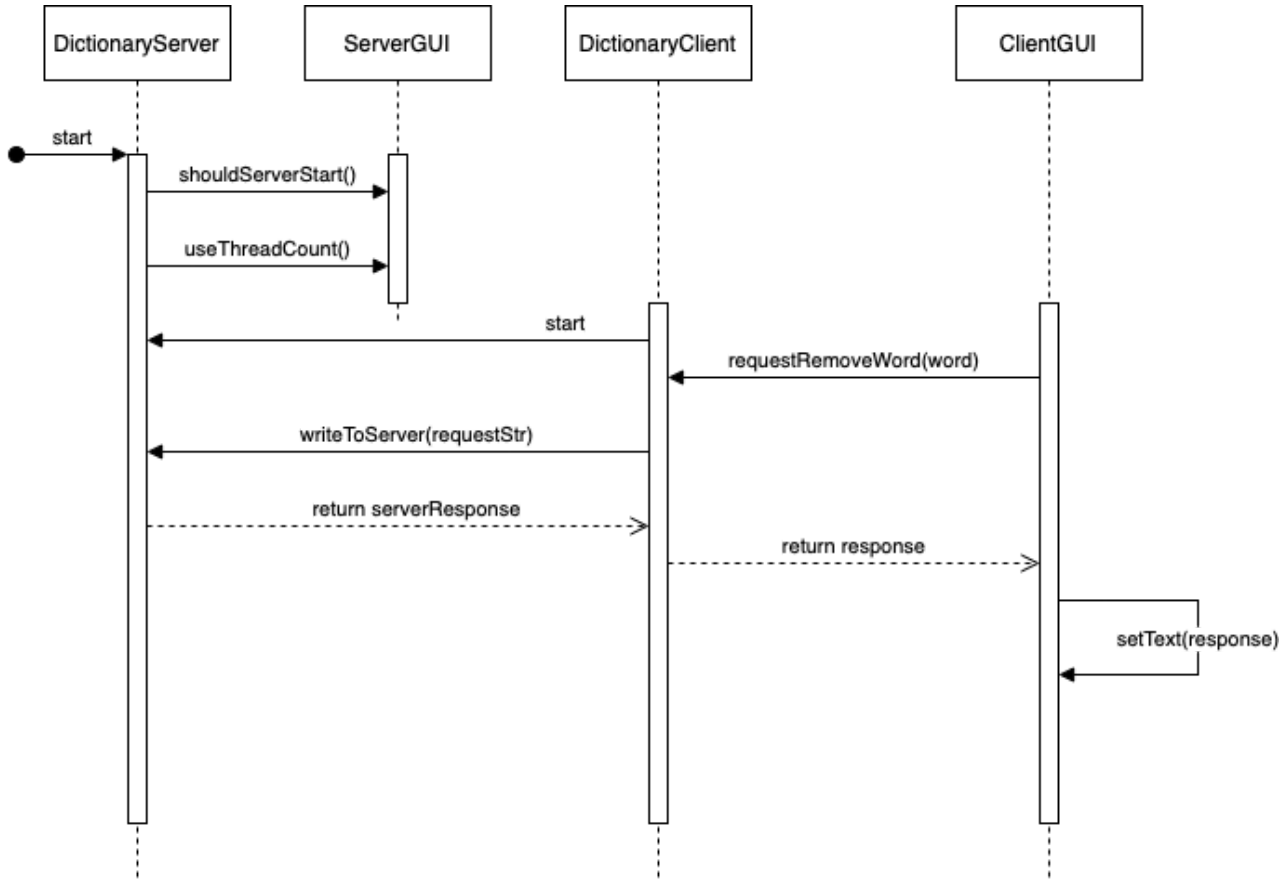

Figure 6: Sequence diagram for removing a word and its associated definition

System Analysis

Protocols

The client-server communication protocol that was chosen for this assignment is TCP. One of the reasons for this is that it provides some reliability guarantees out of the box, which immediately covers one of the requirements of the system having to have reliable communication. The extra overhead that TCP employs to ensure this reliability is arguably a worthy tradeoff, as reliability is more important than latency in a non-time-critical system such as a dictionary server.

The message exchange protocol between the client and the server is chosen to be in JSON format. This is because it is a widely used, established format that lots of real life systems are able to interface with, which means that clients not implemented in Java (heterogeneity) would still be able to communicate with the server. As a result of this, the file used to store the initial list of words is also in JSON.

Architecture and Design

The server employs a worker/thread pool architecture to manage concurrent client requests. This is better than using something like a thread-per-request architecture, as we avoid the overhead associated with the frequent creation and destruction of threads. However, this design choice may incur some performance cost as more clients connect to the server, as there is only a limited number of threads in the pool at all times. This means that once all worker threads in the pool are occupied, subsequent requests would have to be placed in a queue until

a worker becomes available. This is acceptable because arguably a system such as a dictionary server would have short-lived client connections as users search for a couple of words before moving on, meaning that there should always be a healthy number of threads available in the pool most of the time.

The logic of handling dictionary operations is delegated to the dedicated module `DictionaryHandler`, which is effectively a monitor as all of its public methods are synchronised. This helps prevent interference/race conditions by multiple threads trying to read from and write to the dictionary at the same time. This incurs a slight performance cost, as the different `RequestHandler` threads would now have to obtain the lock before being able to access the in-memory dictionary map. However, integrity of the data is arguably a higher priority than performance in this system, as mentioned previously.

Conclusion

The work that has been implemented should cover all the requirements of the system for both subsystems. From the choice of the communication protocol to the implementation of the handling of concurrent client requests, this implemented system prioritises reliability and data integrity over performance, which is arguably reasonable for a dictionary server.

Excellence

All errors have been caught (within reason) and handled so that they are visible and understandable to the humans maintaining and using the system. Server side errors are logged onto the terminal to help with debugging, and client side errors and success statuses are displayed on dedicated sections of the GUI so users are always informed.

The system is designed to follow the single-responsibility principle as much as possible, with each module managing only related pieces of functionality while exposing a consistent set of methods. This means that the `DictionaryHandler` module, for example, could replace its implementation to use a different data structure and logic to store the words and definitions and the rest of the system will not have to change significantly, if at all.

The report has included all of the necessary elements and an analysis of some of the design choices of the system have been conducted with respect to their advantages and disadvantages.

Creativity

A GUI for the server has been implemented, enabling the administrator of the server to easily adjust the number of threads in the worker pool before starting the server.

The communication exchange between the server and the client has been designed to follow a standardised format (`SuccessResponse` and `ErrorResponse`), allowing for both the client and the server to unwrap messages from each other in a consistent and type-safe manner. This also makes the adding of more supported operations in the future relatively straightforward, as their response is always serialised into either of the two response formats.