COMP90015: Distributed Systems
Assignment 1 Report

Student Name: Yiran Wang
Student ID: 1366272

1. Problem Context

The objective of this project is to leverage the client-server architecture to develop a dictionary service application. This application has two primary components: the server and the client. Both components are required to use sockets and threads for network communication and concurrency. Four functional requirements for this application are querying word definitions, removing existing words, updating definitions of existing words, and adding new words along with their definitions. A non-functional requirement is to implement an intuitive graphical user interface on the client side to facilitate users to interact with the server.

2. System Components

2.1 Overview

The application is composed of two main applications: the DictionaryServer and the DictionaryClient. As seen in Figure 2.1, they interact over a network to facilitate word inquiries, updates, creation and removal operations.

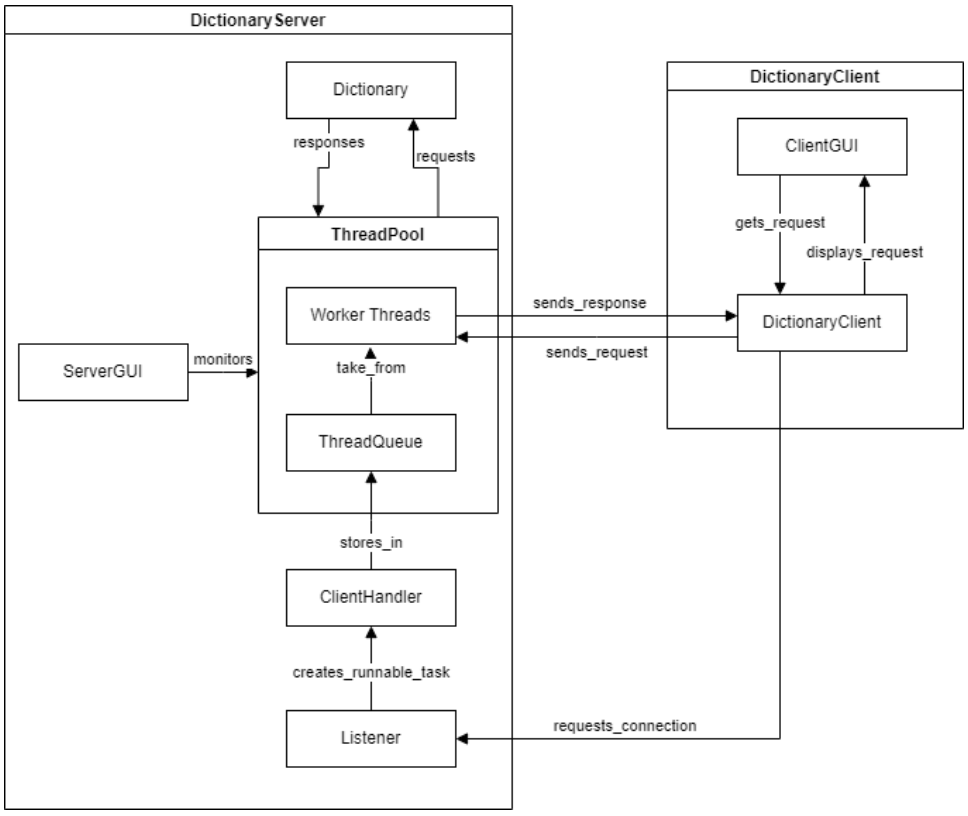

Figure 2.1 Interaction Diagram

2.2 Server

At the heart of the application is the DictionaryServer, a multi-threaded application capable of handling multiple client connections simultaneously (Figure 2.2.1).

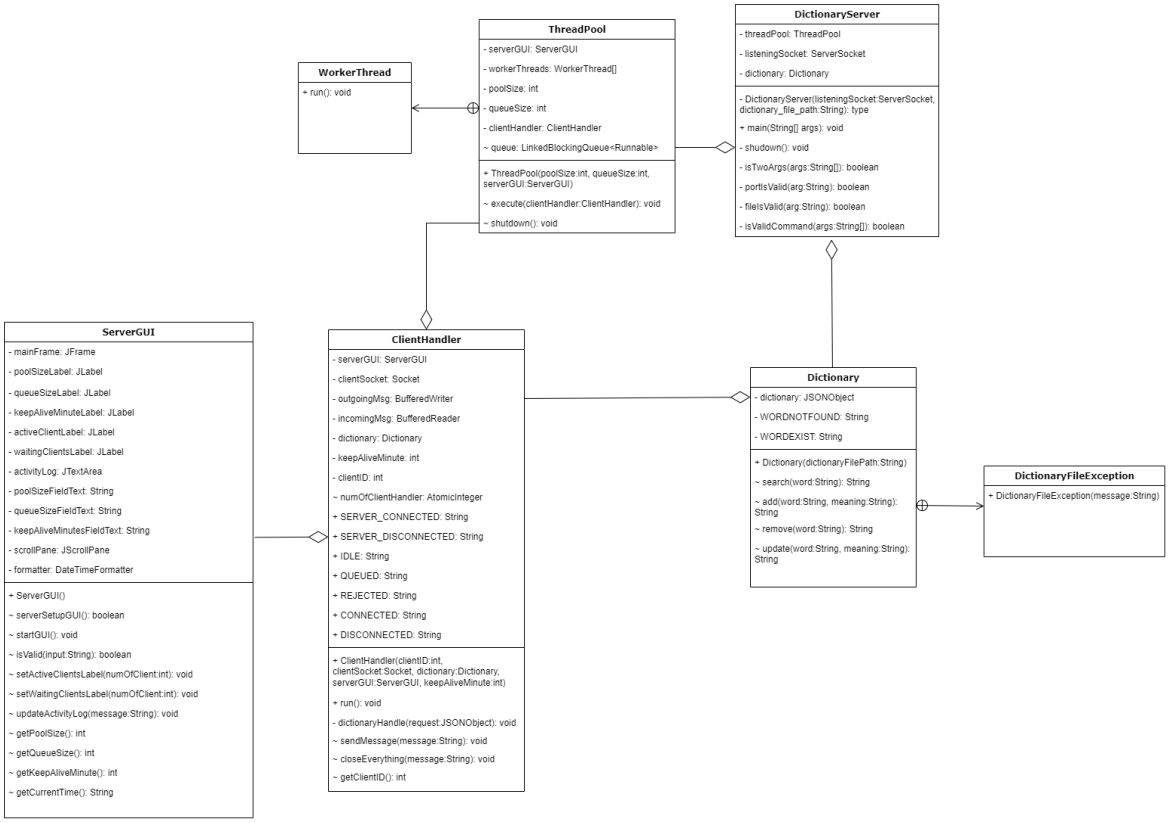

Figure 2.2.1 DictionaryServer Package Diagram

Upon initiating the DictionaryServer, a window (Figure 2.2.2) emerges prompting the server administrator to specify parameters such as pool size, queue size and keep alive time. Once a client connection request is received, the server delegates the request handling to a ClientHandler. This ensures that multiple clients can be served concurrently without waiting for others to complete.

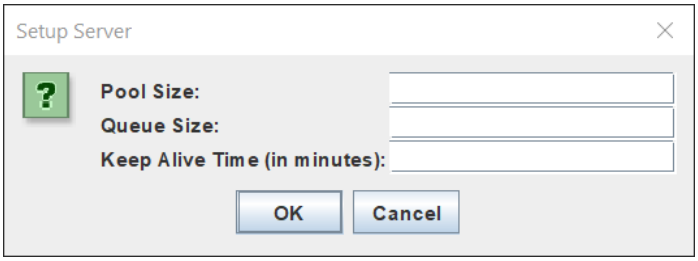

Figure 2.2.2 Server Setup

The server leverages a ThreadPool mechanism, which manages a pool of worker threads. Newly instantiated client handlers are either placed in a First-In-First-Out queue if the queue has capacity, or otherwise rejected with an accompanying notification sent to the client. These worker threads continuously wait for tasks (in this case, ClientHandler instances) to appear in the queue. As a task

becomes available in the queue, it will be picked up by an available worker thread for execution. Additionally, in order to free up worker threads for other awaiting clients, when a client's inactivity exceeds the predefined keep alive time, the connection will be closed and an accompanying notification will be sent to the client. Such an implementation ensures that the server can efficiently handle a large number of client connections without overwhelming system resources or creating excessive threads.

The server's dictionary data is stored in a JSON format (dictionary.json). This data is managed by the Dictionary class, which provides CRUD operations (Create, Read, Update, Delete) on dictionary entries.

Complementing the server's backend mechanics is a graphical interface, the ServerGUI. This GUI offers server administrators a visual platform. It displays server's configuration, active and waiting clients metrics, and an activity log (Figure 2.2.3).

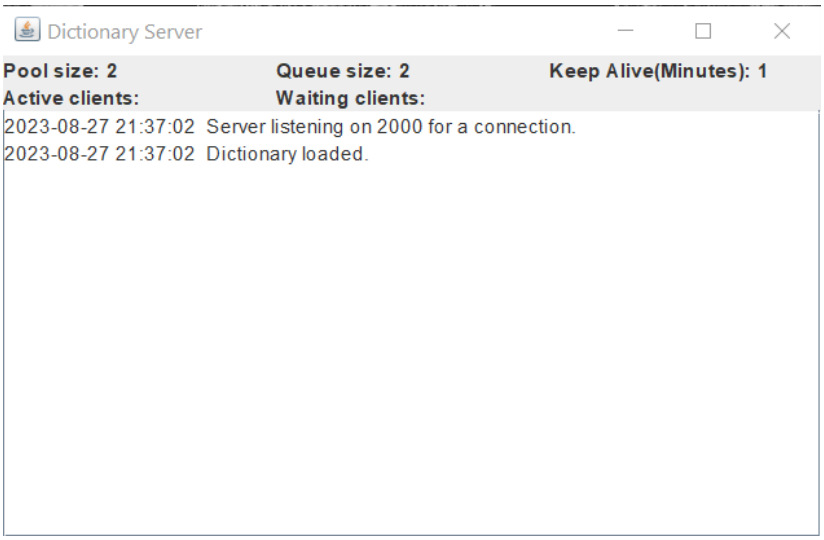

Figure 2.2.3 Server GUI

2.3. Client

On the other side of this application is the DictionaryClient. It allows users to interact with the dictionary server (Figure 2.3.1). This client connects to the server via socket communication, utilizing a designated IP address and port.

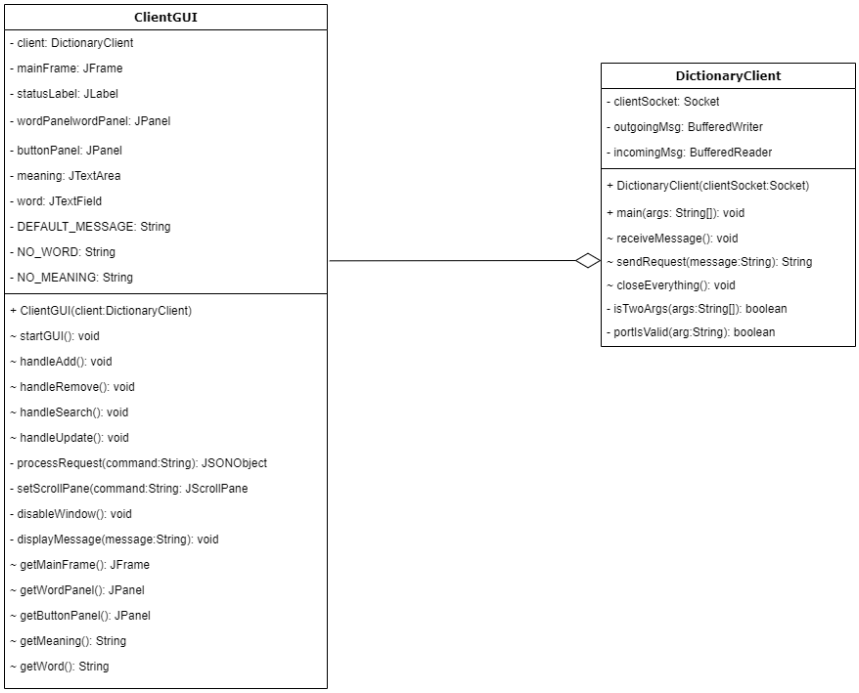

Figure 2.3.1 DictionaryClient Package Diagram

The client application features a ClientGUI, which provides users with an intuitive interface to inquire, create, update and remove words and receive feedback from the server (Figure 2.3.2). Additionally, this interface conducts preliminary validations for missing data (i.e. words, meanings) to lower the load on the server. Behind the scenes, the DictionaryClient class manages the client's communication with the server. It's responsible for sending the requests to the server and displaying responses to the user.

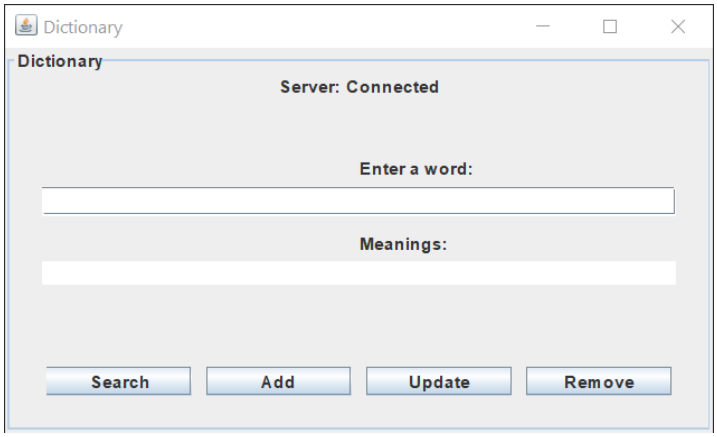

Figure 2.3.2 Client GUI

2.4 Message Exchange Protocol

This system utilizes JSON as its message exchange protocol. JSON offers a key-value structure which aligns well with the functional requirements.

3. Critical Analysis

3.1 Strength

3.1.1 Custom Thread Pool Implementation

One of the most commendable features of the DictionaryServer is its use of a custom thread pool. This approach ensures that a pre-defined number of threads are on standby for incoming client requests. This implementation enhances server performance by minimizing the overhead of thread creation and destruction. By reusing existing threads, the server can efficiently handle a high volume of simultaneous client connections.

3.1.2 Use of TCP Protocol

The system employs the Transmission Control Protocol (TCP) for its client-server communication. As a connection-oriented protocol, TCP ensures reliable, ordered, and error-checked delivery of data. This guarantees the integrity and reliability of data exchanges in the dictionary inquiries and responses.

3.1.3 Error Handling

Both client and server components incorporate comprehensive exception handling mechanisms. This is evident in the way the system handles IOExceptions, SocketException, NumberFormatException, FileNotFoundException, SocketTimeoutException, UnknownHostException, InterruptedException, custom exceptions like DictionaryFileException and so on. Such comprehensive error handling ensures the system's resilience against unexpected situations while providing informative feedback to users or administrators.

3.2 Weakness

3.2.1 Single Thread for Communication

The system adopts a Thread-per-connection architecture. Under this architecture, one single thread is used to handle both inbound and outbound messages for each connection. In the events of substantial data exchange, this design could become a bottleneck, potentially leading to delays in processing client requests. Additionally, this can also lead to unbalanced workload among worker threads especially when one client is sending a surge of requests while others remain relatively quiet.

3.2.2 Custom Thread Pool Design Limitations

While the custom thread pool implementation is a strength, it also possesses limitations. Notably, the thread pool lacks a dedicated thread to monitor client statues within the queue. Consequently, if a queued client opts to exit, it remains undetected in the queue, impeding the release of capacity for new connection requests. Additionally, once the server is configured (Figure 2.2.2), the application offers no flexibility for adjustments. This inflexibility constraints server administrators from optimizing capacity in response to varying demands.

3.3 Creativity

3.3.1 Custom Thread Pool

Dictionary Server has a custom Thread Pool with a FIFO queue implementation.

3.3.2 Server GUI

The ServerGUI provides server administrators with a real-time visual interface for monitoring the server status.

3.3.3 Keep Alive Time

Each client connection has a timer to monitor periods of inactivity. Connections that exceed the predefined keep alive time without sending requests are automatically closed.

3.4 Excellence

3.4.1 User Input Validation at Client side

Dictionary Client handles preliminary validations of user requests (i.e. missing words/meanings). This proactive approach optimizes server performance by lowering unnecessary processing loads.

3.4.2 Thread Pool Setup GUI

Server administrators can tailor the server configuration with an interface.

3.4.3 Reporting

The report provides detailed class design diagrams and an interaction diagram.

4. Conclusion

In conclusion, the project successfully delivers a multi-threaded dictionary application aligned with the stipulated requirements. Its strengths lie in its custom thread pool design for addressing server efficiency, a reliable transmission protocol and comprehensive error handling. However, to improve its robustness and scalability further, it's important to address its limitations, particularly in single thread connection for both inbound and outbound messages and a dynamic thread pool design.