COMP90015 Distributed Systems

Assignment 1: Multi-threaded Dictionary Server

MUHAMMAD DANIEL FAHMI 197325

The University of Melbourne

April 8, 2024

1. Problem Context

This project involves developing a distributed, multi-threaded dictionary system using client-server architecture to enable multiple concurrent operations such as querying, adding, updating, and deleting dictionary entries. The challenge was to ensure data integrity and consistency in a multi-threaded environment, addressing real-world concerns of network programming and concurrent data access. Java sockets implemented for network communication and threading to manage concurrent requests efficiently. The system is designed to balance technical robustness with user accessibility, offering an intuitive client interface while maintaining a reliable backend. This approach not only hones practical skills in software development but also deepens the understanding of distributed systems' principles, crucial for designing and implementing networked applications effectively. The development of this distributed dictionary system mirrors real-world applications in several ways, particularly in how modern online services handle concurrent user interactions and data management across a network.

1. System Components
   1. Server

For this project, all the codes are implemented using JDK version 20 and IntelliJ IDEA. Maven is used as build and dependency management tool.

The server component is built around the DictionaryServer class, which listens for client connections and manages them using a fixed thread worker pool, illustrating a multi-threaded server architecture. This architecture allows the server to handle multiple client requests simultaneously, improving efficiency and scalability. The ClientHandler threads process these requests by performing operations such as adding, querying, updating, or removing words through interactions with the Dictionary class. This class uses a thread safe ConcurrentHashMap to store dictionary entries, ensuring data integrity during concurrent access. The server's design is focused on robustness, able to manage and process requests from multiple clients concurrently without data corruption or performance bottlenecks.

* 1. Client

The client component is designed for user interaction with the dictionary system. The ClientUserInterface class provides a graphical user interface (GUI) that allows users to request to establish TCP connection with server and perform dictionary operations in an intuitive manner. Users can connect to the server, and then add, query, update, or remove dictionary entries. The DictionaryClient class handles the network communication with the server, sending requests packaged as Message objects and receiving responses. The Message class, along with the Protocol enumeration, defines a standard communication protocol between the client and server, ensuring structured and understandable data exchange.

* 1. Dictionary

The dictionary component, encapsulated within the `Dictionary` class, serves as the fundamental data management layer of the distributed dictionary system, tasked with the secure storage and efficient handling of dictionary entries. Leveraging a ConcurrentHashMap data type for storage, this component ensures thread-safe operations—critical for supporting the concurrent processing of client requests in a multi-threaded server environment. It provides a comprehensive suite of functionalities, including the addition, querying, updating, and removal of words, alongside robust error handling mechanisms to maintain data integrity and handle exceptional scenarios effectively. This component's integration with the server component, particularly through the mediation of ClientHandler threads, underscores its role in executing the core business logic of the application.

1. Design Specification
   1. Client Package

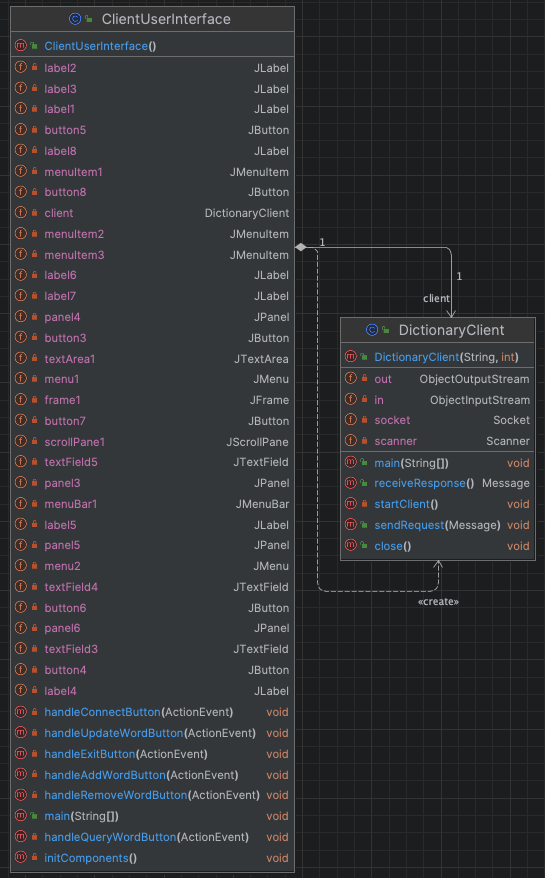


Figure 1: Client Package Class Diagram

Figure 1 shows how classes in the client package defined and interacts with each other. Within the client package, the user initiates an interaction through the ClientUserInterface class by performing actions such as adding, updating, removing, or querying words through the GUI as shown in Figure 2. Based on the user action, ClientUserInterface creates a Message object, setting the operation type to the corresponding action (e.g., ADD\_WORD, UPDATE\_WORD, REMOVE\_WORD, QUERY\_WORD) and including necessary data like the word and possibly its meanings. ClientUserInterface then invokes a method on DictionaryClient, passing the Message object. DictionaryClient serializes this object and sends the request to the server over the network.

A screenshot of a dictionary client

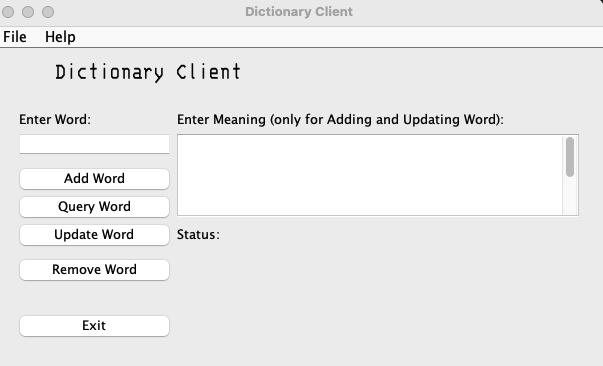
Description automatically generated

Figure 2: Client GUI; (left) login page, (right) dictionary page.

* 1. Common Package

The Message class in the common package plays a crucial role here, providing the structure for request and response messages exchanged between the client and server or simply the Inter-Process Communication, IPC of the system. The Message formatting is as shown in Table 1. It includes methods for serializing and deserializing message content using FasterXML Jackson library.

The Protocol enum, as tabulated in Table 2, also part of the common package, defines constants for different operation types and responses, ensuring both the client and server understand the communication context and content.

|  |  |
| --- | --- |
| Protocol | A set of Protocol enum of allowed operation between client and server |
| Word | Word as a string for operation in HashMap |
| Meaning(s) | Definition of word stored in Array List in HashMap |

Table 1: Inter-Process Communication Format (JSON)

|  |  |
| --- | --- |
| ADD\_WORD | Used in Message request |
| QUERY\_WORD | Used in Message request |
| REMOVE\_WORD | Used in Message request |
| UPDATE\_WORD | Used in Message request |
| RESPONSE\_SUCCESS | Used in Message reply |
| RESPONSE\_ERROR | Used in Message reply |

Table 2: Message Protocol

* 1. Server Package

On the server side, all the server-related classes as shown in Figure 3 such as the DictionaryServer class listens for incoming requests and assigns each to a ClientHandler from its thread pool, showcasing the system's ability to handle concurrent operations. ClientHandler deserializes the Message to interpret the client's request. Depending on the operation type specified in the message, ClientHandler calls the corresponding method on the Dictionary class to execute the operation. The Dictionary class processes the operation—whether it’s adding, updating, removing, or querying words—ensuring thread-safe access and modification of the dictionary entries. After the operation, ClientHandler creates a response Message, indicating the outcome of the operation, and sends it back to the client after serialization.

Back in the client package, DictionaryClient receives and deserializes the response, interpreting the result of the operation. Finally, ClientUserInterface receives the outcome from DictionaryClient and updates the GUI, accordingly, providing the user with feedback on the operation's success or failure. The whole representation of the sequence of operation and how classes interact with each other is shown in the sequence diagram in Figure 4.

A screenshot of a computer

Description automatically generated

Figure 3: Server Package Class Diagram

* 1. Interaction Diagram

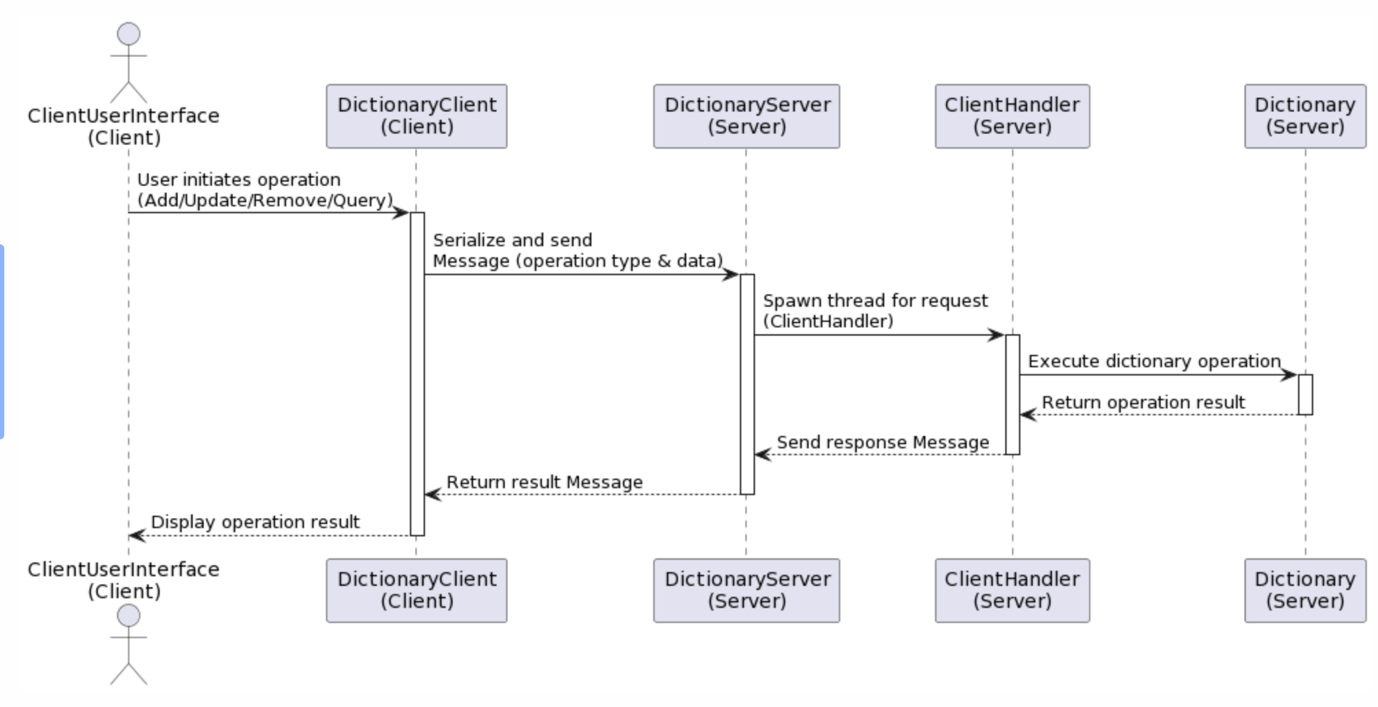


Figure 4: Interaction Diagram

1. Discussion and Retrospection
   1. System and Design Analysis

The design rationales behind the choices in the distributed dictionary application can be grouped into the three distinct layers of the system: the server, client, and dictionary components. Some design choices are made considering the functionality, performance, and the reusability across all layers. Each layer has its unique requirements and challenges, which are addressed by specific design decisions:

* + 1. Server Layer
       1. Fixed Thread Pool Architecture

A fixed thread pool was chosen to manage concurrent client requests for several reasons. It prevents the overhead associated with creating a new thread for each incoming request, thereby conserving system resources.

By reusing a limited number of threads, the system can handle a predictable load efficiently, which is crucial for avoiding potential bottlenecks under high concurrency. However, this fixed size can limit the application's ability to scale dynamically in response to varying workloads.

* + - 1. Use of ExecutorService for Thread Management

ExecutorService provides a more efficient way to reuse threads for multiple tasks, reducing the overhead of thread creation and destruction, which is particularly important for high throughput servers. While ExecutorService simplifies thread management and improves resource utilization, it introduces additional complexity to the codebase.

* + 1. Client Layer
       1. GUI and Dual Entry Points

There are two entry points for client; GUI and terminal. The GUI provides an intuitive interface for end-users, hiding the complexity of network operations and dictionary protocols. The terminal on the other hand, allows for automated or script-based operations, mainly for debugging in this project.

* + 1. Dictionary Layer
       1. Thread-Safe Data Structures

The project utilizes ConcurrentHashMap as the data structure for reasons. It ensures that dictionary data remains consistent and correct, even when accessed by multiple threads simultaneously, crucial for operations like add, remove, update, and query. Although it guarantees data integrity, it can have a higher memory footprint due to the additional metadata and structures they maintain to ensure thread safety. This can be a concern in memory-constrained environments.

* + 1. Multi-Layer
       1. TCP as Transport Protocol

TCP is chosen for ensuring that every request and response between the client and server is accurately transmitted and received, maintaining the integrity of dictionary operations. It facilitates stable and continuous communication sessions between clients and the server, enabling complex interactions like multiple queries or updates within a single connection. However, TCP, being a connection-oriented protocol, requires a handshake process to establish a connection before any data transfer can occur. This adds latency and overhead, particularly noticeable in scenarios requiring low-latency communications or when establishing many short-lived connections.

* + - 1. JSON for IPC

While primarily beneficial for the client for its simplicity and readability, JSON's efficiency in serialization and deserialization also benefits the server and dictionary layers by minimizing the overhead of converting data structures into a network-transmittable format. Jackson external library was chosen for its high-performance serialization/deserialization capabilities and its broad acceptance in the industry, which makes it a reliable choice for JSON processing.

* + - 1. Javadoc Comment Style

All classes and methods in the project are annotated with Javadoc comment. It creates a uniform documentation style throughout the codebase, making it easier for developers to understand and maintain the code.

* 1. Failure Handling and Management

|  |  |
| --- | --- |
| Error Description | Solution |
| Unable to reach server (client side) | Show GUI response and ask for a reachable server. |
| Invalid host (client side) | Show GUI response and ask for a valid host |
| Invalid port number (both sides) | Show GUI response and ask for a valid host |
| Dictionary file not found (server side) | Create a new dictionary file in existing path |
| Bad communication message (both sides) | Catch and notify failure in object serialisation |
| Add existing word to Dictionary | Show word exists, abort operation |
| Query non-existing word | Show word not found |
| Remove non-existing word | Show word not found, abort operation |
| Update non-existing word | Show word not found, abort operation |
| Update same meaning to a word | Show meaning exist |
| Invalid Dictionary operation | Create and bind buttons for each operation |

Table 3: List of errors and solution

* 1. Excellence and Creativity Elements

For excellence, in the implementation of multi-threaded Dictionary server, there are multiple methods utilized to ensure thread-safe system. Thread-safe is needed to allow concurrent access from multiple threads without leading to inconsistent states or other unexpected behaviors. GUI toolkits is usually not thread-safe and need ways to handle concurrent access. Two ways of handling such issue are by implementing Event Dispatch Thread (EDT) and SwingWorker.

EDT is a single-threaded GUI model with a dedicated thread for all GUI-related activities. It is implemented in the main method of the ClientUserInterface. EDT responsible for dispatching and handling events, managing component state change, and performing repaints or revalidations of the GUI.

SwingWorker enable operations that take a long time to complete, such as network communication, file I/O, or intensive computation, can block the EDT if performed directly within an event handler. If the EDT is blocked, the GUI will freeze and become unresponsive. It allows these long-running tasks to be performed in a background thread, keeping the GUI responsive. While the task is running, users can still interact with the GUI, such as clicking buttons, resizing windows, etc. It is implemented in all the operation buttons such as handleAddWord, handleQueryWord, handleUpdateWord, and handleRemoveWord.

For creativity, the program implemented a Server GUI for viewing all the transaction history that is crucial for debugging purposes by developer.

In the program, logging is facilitated through a centralized logging system that hinges on the GlobalLogger class. This class maintains a list of listeners—classes that implement the LogUpdateListener interface. When an event occurs that needs to be logged, the relevant part of the application, such as the ClientHandler, invokes `GlobalLogger.log` with a message detailing the event. The `log` method iterates over all registered listeners and dispatches the log message to them by calling their onLogUpdate method. The ServerUserInterface, acting as one of these listeners, implements the onLogUpdate method to append incoming log messages to a text area within a scroll pane as shown in Figure 5, ensuring updates to the GUI are done on the Event Dispatch Thread to maintain thread safety.

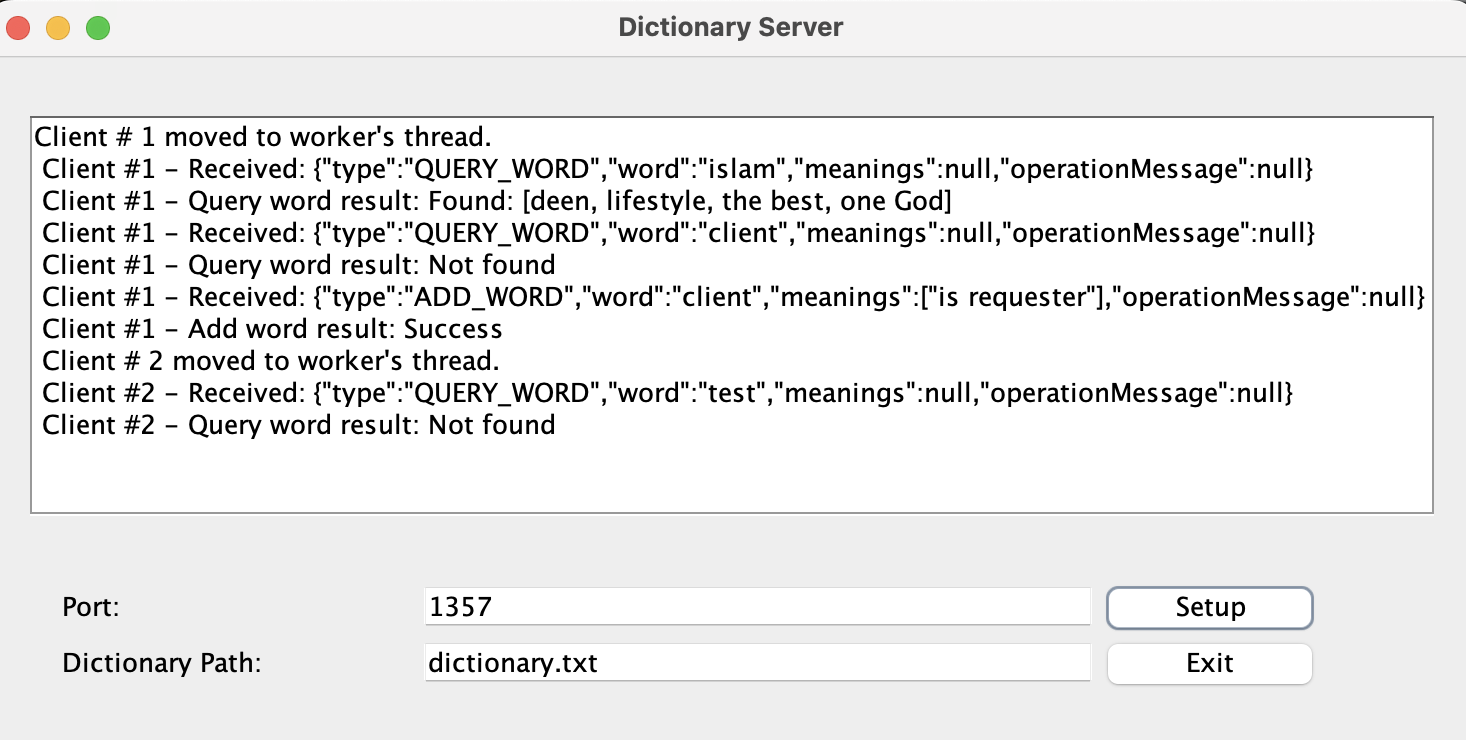


Figure 5. Server GUI