COMP90015: Distributed Systems – Assignment 1
Multi-threaded Dictionary Server

Tianhao Liu 1105160

1. Introduction

This report outlines the Distributed Dictionary System, designed to enable efficient dictionary operations via **TCP sockets** and a **worker-pool architecture**. The system prioritizes scalability, concurrency, and robust error handling.

2. Problem Context

The system meets the growing demand for real-time, multi-user access to a digital dictionary. It addresses key challenges:

- **Concurrent Access**: Ensures conflict-free operations and data consistency.
- **Scalability**: Built to perform well as user numbers rise.
- **Error Handling**: Capable of gracefully managing diverse errors.
- **Real-time Updates**: Immediate visibility of changes to all clients.
- **Efficiency**: Quick data retrieval and modification.

The design leverages TCP sockets and a worker-pool for effective concurrency and low latency.

3. Components of the System

The system includes two core components, DictionaryServer and DictionaryClient, which facilitate a range of dictionary operations.

3.1 DictionaryServer (DictionaryServer.java)

The server acts as the central hub for both the dictionary resource and client interactions.

- **ServerSocket**: Awaits incoming client connections and offloads each to a separate task, ensuring the main server stays receptive.
- **SQLite Database**: Responsible for CRUD (Create, Read, Update, Delete) operations related to words and their meanings in the dictionary.
- **Worker-Pool Architecture**: Utilizes a pool of worker threads to manage and execute incoming client requests concurrently, improving scalability.
- **In-Memory HashMap**: Provides quick, direct access to word entries, populated from the database at startup for efficiency.
- **GUI**: Displays server metrics, logs, and allows manual scaling of the worker pool.

3.2 DictionaryClient (DictionaryClient.java)

Serves as the user's portal to the dictionary resource.

- **TCP Socket**: Manages the underlying TCP connection to the server, facilitating the sending and receiving of data.

- **JSON Protocol**: Standardizes the format for all client-server communication, using JSON for clarity and ease of handling.
- **Operations**: Includes features for querying a word's meaning, adding new words, deleting existing ones, and updating word meanings.
- **Error Handling**: Equipped to catch and display a variety of errors, offering relevant user feedback for each.

4. Class Design

4.1 DictionaryClient

The DictionaryClient class is responsible for establishing a connection to the DictionaryServer, sending requests based on user input, and displaying the server's responses to the user. It provides a simple user interface for users to interact with the dictionary service.

Key Attributes:

- **serverAddress**: The IP address or hostname of the DictionaryServer.
- **serverPort**: The port on which the server is listening..

Key Methods:

- **sendRequestToServer(JSONObject request)**: Sends a JSON-formatted request to the server and receives a response.
- **main(args: String[]) :** The function that accept the address and port number of server from command line, and initialised the GUI and functioning the button for clients.

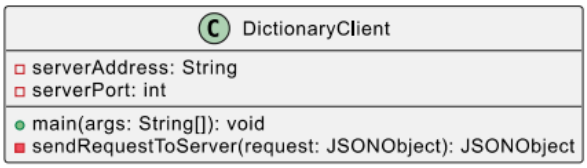

Figure 1: UML diagram for DictionaryClient

4.2 DictionaryServer

This component is responsible for establishing the server-side socket connection, listening for incoming client requests, and handling those requests by delegating them to a pool of worker threads. It also manages the dictionary data by interfacing with an SQLite database.

The following are the key attributes and methods associated with the DictionaryServer class:

Key Attributes:

- **numberOfWorkers**: Represents the number of worker threads in the thread pool.
- **threadPool**: A thread pool to efficiently manage and reuse worker threads.
- **port**: The port on which the server listens for incoming connections.
- **dictionaryFilePath**: Path to the SQLite database file.
- **dictionary**: A HashMap used as a cache to store the dictionary data for faster access.
- **serverSocket**: The server socket for accepting client connections.

Key Methods:

- **connectToDatabase():** Establishes a connection to the SQLite database and ensures the necessary tables exist.
- **loadDictionary():** Loads the dictionary data from the SQLite database into the HashMap for caching purposes.
- **startServer():** Initializes the server socket and listens for incoming client connections.
- **handleClientRequest(Socket clientSocket)**: Handles individual client requests, including searching for words, adding new words, updating word meanings, and removing words.

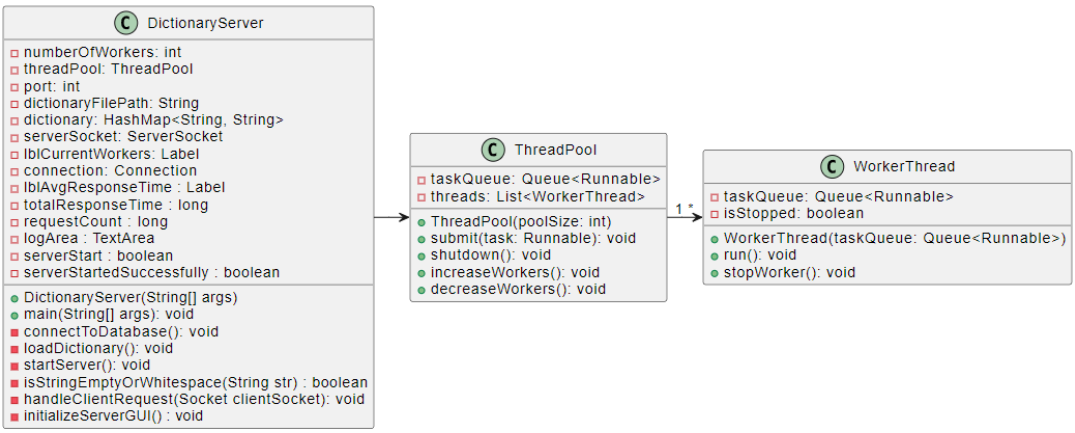

Figure 2: UML diagram for DictionaryServer

4.3 System Interaction

The worker-pool architecture is a design pattern used in the DictionaryServer to efficiently manage the execution of tasks in a multi-threaded environment. It employs a set of worker threads to carry out tasks that are stored in a queue.

Here's a high-level overview of how the worker-pool architecture handles incoming requests:

1. **Task Queuing:** When a new client sends a request to the server, the server encapsulates the handling of that request as a 'task' and places it in the task queue.
2. **Worker Availability:** Worker threads in the ThreadPool are continuously monitoring the task queue for new tasks.
3. **Task Pickup:** As soon as a worker thread finds a task in the queue, it picks it up for execution. This allows multiple threads to work simultaneously, each handling a different client request.
4. **Task Execution:** The worker thread executes the task, which could involve querying, adding, removing, or updating a word in the dictionary. Since each worker thread operates independently of the others, multiple requests can be handled in parallel.
5. **Response:** After completing the task, the worker thread sends the appropriate response back to the client.

6. **Awaiting Next Task:** Once a task is completed, the worker thread goes back to monitoring the task queue, waiting to pick up the next available task.

We also prepare an interaction diagram that visualizing how this system are interacted with each other:

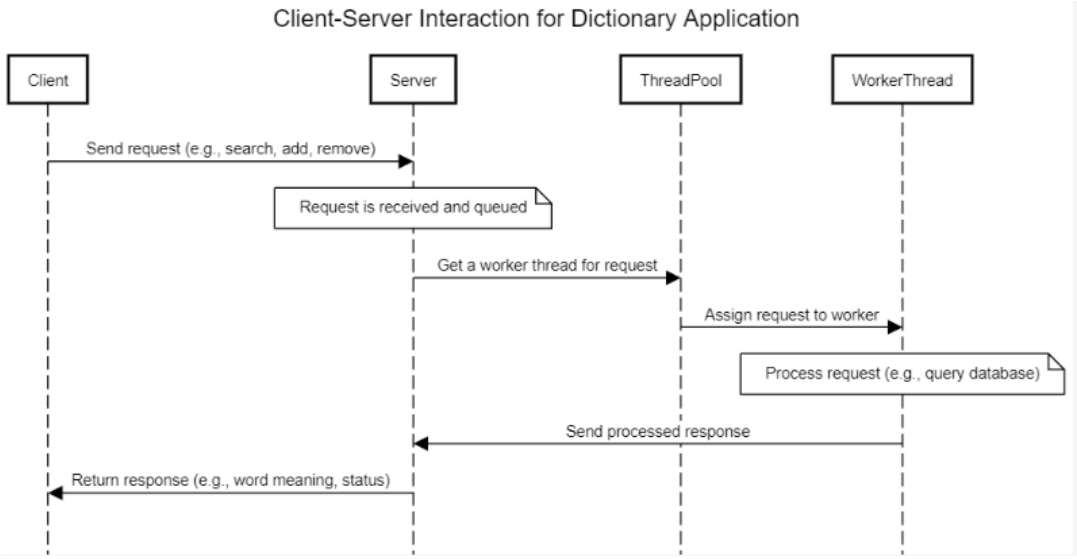

Figure 3: System Interaction diagram

5. Error Handling

5.1 Functional Error Handling

The system was rigorously tested for all functionalities: Querying, Adding, Removing, and Updating words. Here are some highlights:

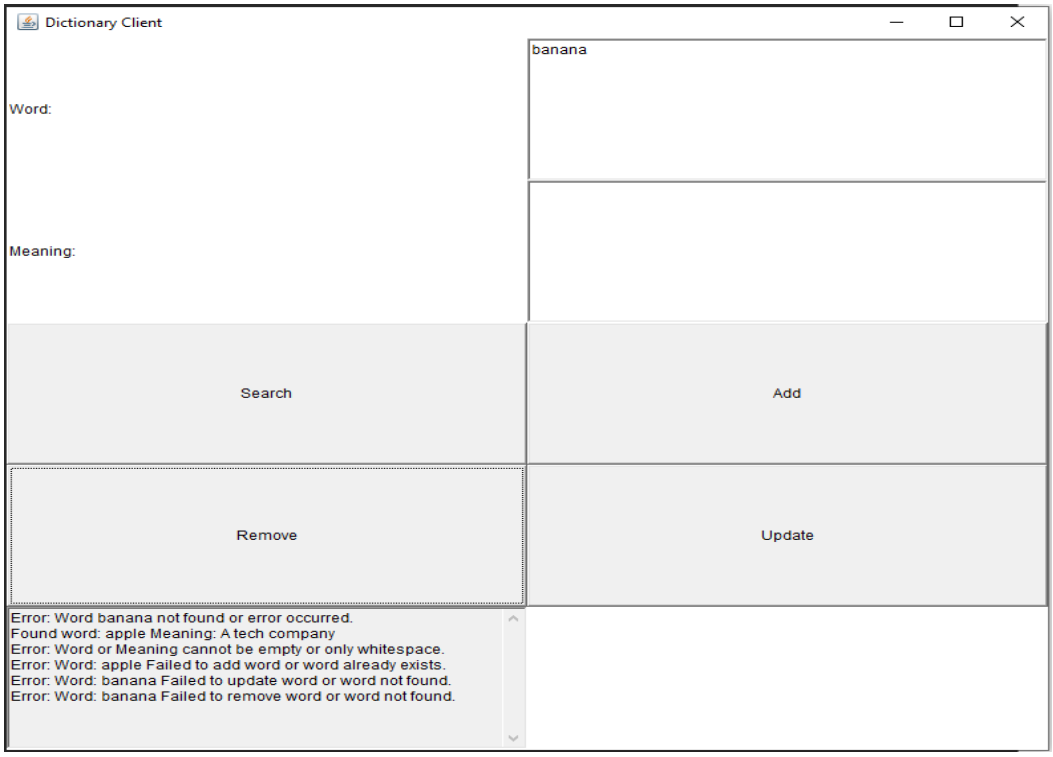

Figure 4: demo of functional error handling

- **Query Word**: The **empty string** will respond to the error messages to clients, and will response an error message when user try to search to a word that are **not exist**.
- **Add a New Word**: The system prevented the addition of **duplicate** words and words **without meanings**.
- **Remove an Existing Word**: The server responded with a "not found" message when trying to remove a word that does not exist.
- **Update Meaning**: Like "Remove", the system efficiently handled the "not found" scenario.

5.2 Database Errors handling

If the SQLite database encounters an issue, such as being deleted or renamed while the server is running, the server responds with appropriate error messages for "add", "remove", and "update" actions. However, "search" actions continue to work as expected because the server also maintains a HashMap of the dictionary.

5.3 Address Binding Error

If the user wants to run the server on a port that is already in use, the log area will display an associated error message to the user.

5.4 Handling incorrect User Input

Both the **DictionaryServer** and **DictionaryClient** classes validate that exactly two command-line parameters are provided. In the server, these parameters are the **port number** and dictionary **file path**, while in the client, they are the **server address** and **port number**. If the number of arguments is not exactly two, the application prints an error message and terminates. In addition, if there is not a running server in the specific port/address, the client GUI will let the user know the **server is not running** when user try to send any request to server. Also, the server will create a new database to store the dictionary if server did **not find a database** file base on the input of dictionary file path.

6. Creativity Elements

6.1 Own Implementation of Worker-Pool Architecture

In the server-side logic, we have implemented our custom version of a worker-pool architecture. This implementation comprises two core classes: **ThreadPool**, and **WorkerThread**.

Instead of relying on Java's built-in **ThreadPoolExecutor**, the class **ThreadPool** employs a combination of **LinkedList** and **Queue** data structure to maintain a queue of tasks. It aims to ensure that tasks should be handled in a **First In, First Out manner**, while the **LinkedList** provides ability of **dynamic sizing**, which make the queue can grow or shrink as tasks are added or remove.

Each **WorkerThread** instance in the **ThreadPool** is stored in a **list** and continuously polls the shared **LinkedList** queue for tasks using **synchronized** methods. By **synchronizing** access to

the shared queue, we ensure that only one thread can access the queue at a given time. This not only prevents multiple threads from picking up the same task but also helps in avoiding deadlock situations. When a task becomes available, it is dequeued and executed by an available worker thread. This behavior closely mimics that of a worker-pool architecture, where multiple threads are on standby to pick up tasks from a shared, common queue.

6.2 Server's GUI

The **DictionaryServer** class also incorporates a graphical user interface (GUI) for server administration. The following features enhance the user experience and system manageability:

1. **Start/Stop Server**: A 'Start' and 'Stop' button is provided for initiating and halting the server, respectively. (Note: the server needs the parameter of initial worker to start the server, and we pre-fill the available processor to initial worker, but the user still could adjust this parameter base on their needs)
2. **Show Dictionary**: A button named 'Show Dictionary' allows users to view the current state of the dictionary, aiding in immediate verification and debugging.
3. **Log Area**: A log area is present that prints out every action requested by a client. Furthermore, the client's address is displayed when a connection is established.
4. **Manual Scaling**: The GUI provides 'Increase Worker' and 'Decrease Worker' buttons, enabling users to manually scale the number of worker threads in real-time.
5. **Mean Response Time**: A display area for the mean response time provides valuable metrics that can help in assessing the system's performance.

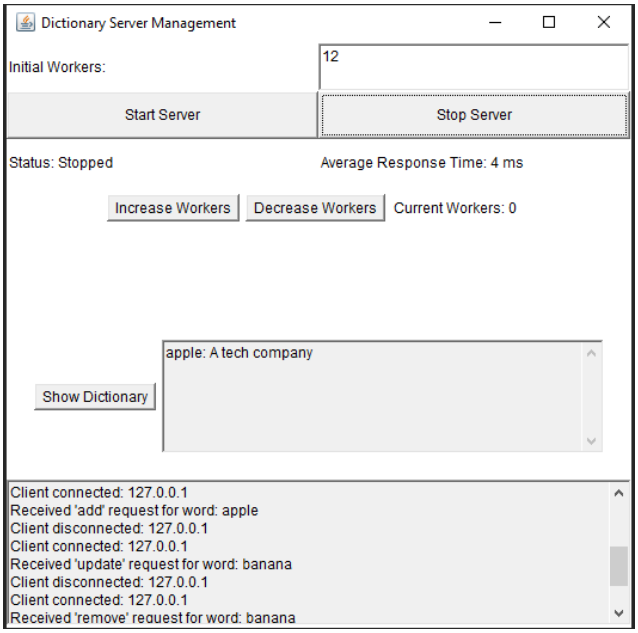

Figure 5: Server's GUI

7. Excellence Elements

7.1 Notification of Error in Edge Cases

For testing the scenarios in which the client sends an invalid or malformed request, we simulate a situation where the request is not sent via the GUI. In this situation, we perform the following test which sends the JSON Object directly to the server:

Send: "invalid{json" , **response**: {"description":"Invalid JSON format.","status":"error"}

Send: {"action": "search"}, **response**: {"description":"Missing 'word' field in search request.","status":"error"}

Send: {"action": "unknownAction", "word": "example"}, **response**: {"description":"Unknown action: unknownAction","status":"error"}

Send: {}, **response**: {"description":"Invalid JSON format.","status":"error"}

7.2 Notification of Dangerous Actions

In the server's GUI, we provide buttons for increasing and decreasing the number of worker threads. If a user attempts to increase the worker count higher than the available processors on the current device, the log area will display a warning message.

7.3 Critical Analysis of Worker-Pool Architecture

Before diving into the high concurrency tests, it's worth mentioning that Wireshark was used to verify that the server-client communication was indeed using the TCP protocol. This double assurance confirms the reliability of the connections.

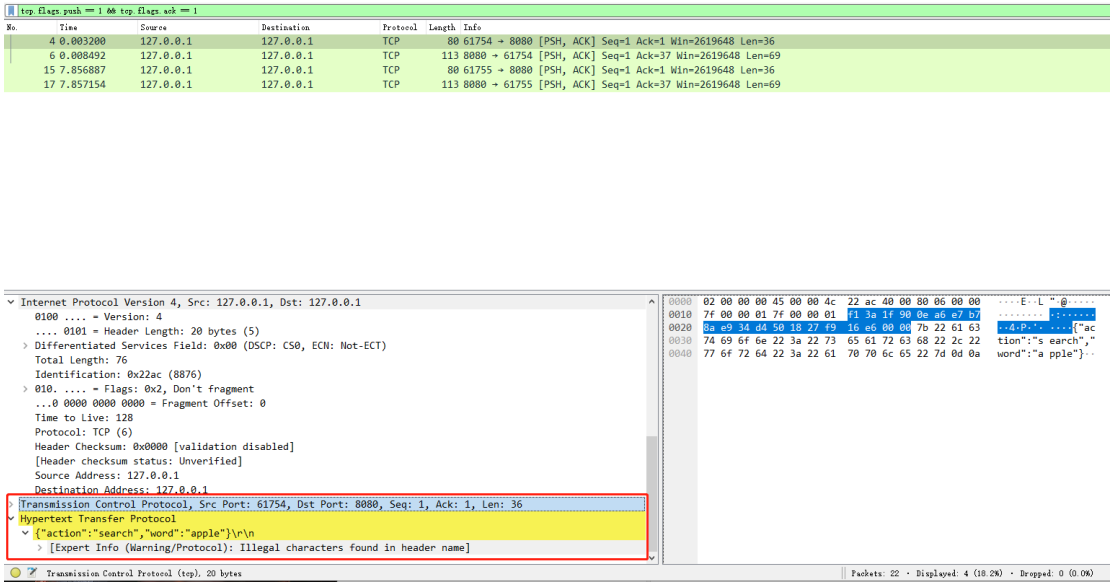

Figure 6: Wireshark network analysis of the Server-Client communication

To test how the system behaves in a more realistic situation, we conduct a high concurrency test for the system by using the JMeter. We initialised 6 TCP samples which performs different action: search, add, remove, update, and setting a random controller to randomly select which these TCP samples. Here is the experiment setup:

1. **400 Concurrency Request, 4 workers**:
    - **Errors**: 2 errors were encountered out of 400 requests. This is a 0.5% error rate.
    - **Average Response Time**: 7ms
    - **Analysis**: With 400 concurrent requests and only 4 workers, it's notable that the system was able to handle the load with a very low error rate and a reasonable response time.

2. **350 concurrency Request, 4 workers**:
    - **Errors**: 0 errors
    - **Average Response Time**: 6ms
    - **Analysis**: Reducing the concurrency by 50 requests resulted in no errors and slightly improved response time.
3. **350 concurrency Request, 3 workers**:
    - **Errors**: 0 errors
    - **Average Response Time**: 6ms
    - **Analysis**: Despite reducing the number of workers by one, the system was able to handle the same load (350 requests) without any errors and with the same average response time as with 4 workers. This suggests that the system is efficiently distributing tasks among the available workers.
4. **350 concurrency Request, 2 workers**:
    - **Errors**: 10 out of 350 requests, approximately a 2.86% error rate.
    - **Average Response Time**: 4ms
    - **Analysis**: Reducing the worker count to half while maintaining the same level of concurrency resulted in a higher error rate. Interestingly, the average response time improved, which might suggest that while the system can process requests faster with fewer workers, it becomes more prone to errors due to overloading.

Based on the experiments conducted, here are some advantages and disadvantages of the worker-pool architecture:

Advantages:

- **Efficiency**: The system can handle a high number of concurrent requests with a relatively low error rate, as evidenced by the 0.5% error rate with 400 concurrent requests and 4 workers.
- **Scalability**: The system can maintain low average response times even when the number of concurrent requests is high, and we could manually adjust the number of worker base on the current load of the server.
- **Resource Utilization**: The experiment with 350 requests and 3 workers showed no errors and consistent response times, indicating that the worker-pool architecture efficiently utilizes resources.

Disadvantages:

- **Error Sensitivity**: Reducing the number of workers while maintaining high concurrency levels can increase the error rate, as seen with 350 requests and 2 workers.
- **Manual Intervention for Worker Configuration**: Although the system suggests a recommended number of workers, manual adjustment is often necessary. Utilizing all available processors on a device may not always be ideal, as it could potentially impact other tasks the user may wish to perform simultaneously. Thus, a basic

understanding of the device's capabilities and the workload is essential for configuring an appropriate number of workers.

- **Optimal Configuration Needed**: The results suggest that there is a trade-off between the number of workers and the level of concurrency. Striking the right balance is crucial for optimal performance, but this will require more experiments to take.

8. Conclusion and future directions

The Distributed Dictionary System successfully addresses the challenges of real-time, multi-user access, scalability, and error handling in a digital dictionary. Utilizing a worker-pool architecture and TCP sockets, the system is capable of efficiently managing high volumes of concurrent requests. Notable features like server GUI and custom error handling add layers of usability and robustness. While the system performs well under different conditions, fine-tuning worker configuration remains an area for future optimization. Additionally, future work could involve conducting experiments with different multithreading architectures (e.g., Thread-per-connection, Thread-per-request…), perhaps exploring combinations with other protocols like UDP, to find the most suitable system design for this specific task.