COMP90015 Distributed System Assignment 1 Report

Zhuoyang Hao 1255309

1 Problem Context

The task involves creating a Java-based dictionary that supports multi-threading and uses a client-server model. The system should communicate over the transport layer, employing TCP protocols. The server needs to handle multiple simultaneous requests efficiently through multi-threading. The dictionary should support CRUD (Create, Retrieve, Update, Delete) and Add-Meaning operations and include necessary input validations. It must also load data from a file at startup and permit changes to content in real time. The client interface should feature an interactive GUI. Additionally, thorough error handling is essential to maintain the integrity of both the client and server components.

2 Components of System

As illustrated in Figure 1, the system comprises the following components:

- `Server` and `Client` package: These contain the core logic code necessary for implementing the functionalities of both the client and server.
- `Communication` package: This includes the protocols that facilitate communication between the client and server.
- `ThreadPool` package: This includes my own implementation of thread pool.

The overall class design is shown in Figure 2.

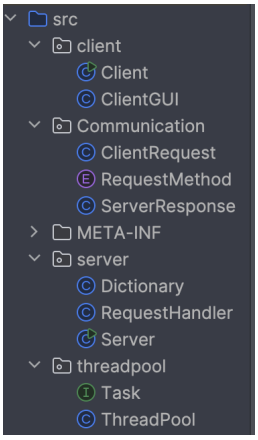

Figure 1: Components of System

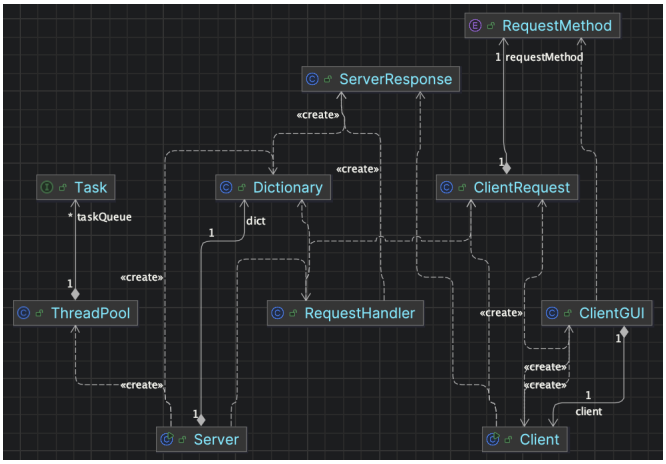

Figure 2: Overall Class Design

```
public enum RequestMethod {
    CREATE,  4 usages
    RETRIEVE,  3 usages
    UPDATE,  3 usages
    DELETE,  3 usages
    UNDO,  3 usages
    ADD  3 usages
}
```

Figure 3: Request Method

3 JDK and Dependencies

For this project, I am utilizing Oracle's JDK 17. Additionally, I am employing the org.json library to manage JSON files.

4 Message Exchange Protocol

4.1 Connection Establishment

The client establishes a TCP connection with the server using the server's IP address and a port. This connection remains open for the duration of the interaction required for a specific request/response cycle.

4.2 Request Type

As shown in Figure 3, the following request are implemented:

- `CREATE`: Adds a new word with its meanings if it doesn't already exist.
- `RETRIEVE`: Fetches meanings of a word if it exists.
- `ADD`: Adds additional meaning to an existing word.
- `UPDATE`: Updates meanings of a word if it exists.
- `DELETE`: Removes a word from the dictionary.

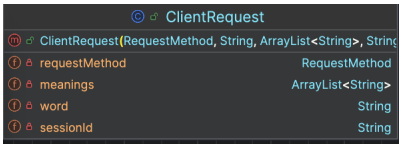

Figure 4: Client Request

Figure 5: Server Response

4.3 Request Sending

`Client Action`: As shown in Figure 4, the client creates a `ClientRequest` object with all necessary details filled in based on user input, including `requestMethod`, `meanings` (only applied when the request method is `CREATE`, `UPDATE` or `ADD`) and `word`. This object is then serialized and sent to the server through the open socket connection.

`Server Reception`: The server listens for incoming connections. Upon accepting a connection, it deserializes the incoming stream back into a `ClientRequest` object to understand and process the request.

4.4 Response Sending

`Server Action`: Once the request is processed, the server serializes the `ServerResponse` object as shown in Figure 5 and returns it to the client using the same connection. The `successful` field indicates the success or failure of the CRUD operation. For example, if the client searches for a non-existent word or tries to add a word that already exists, this field will be marked as false, signaling a failure, with corresponding failure messages in `message` field. Moreover, if the server handles a `Retrieve` request successfully, it will reply with the `meanings` field filled with the word's definition. Otherwise, the `meanings` field will be set to null.

`Client Reception`: The client receives and deserializes the `ServerResponse` object to display the outcome and any relevant data to the user via the GUI.

5 Dictionary Implementation

5.1 Data Structure

The dictionary's framework is constructed using a `JSONObject` (see Figure 6), which serves as a hash map. In this setup, each key corresponds to a word, and the associated value is a list of its definitions. This design allows for efficient searches and straightforward updates. Specifically, a `JSONObject` stored in memory, initially loaded from a JSON file on disk, is used. Actions such as `CREATE`, `UPDATE`, `ADD`, or `DELETE` modify this in-memory `JSONObject`, and these alterations are then synchronized with the JSON file on disk. On the other hand, a `RETRIEVE` operation only accesses the in-memory `JSONObject` to fetch definitions, without modifying the JSON file on disk.

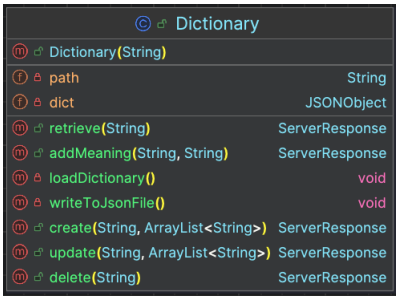

Figure 6: Dictionary

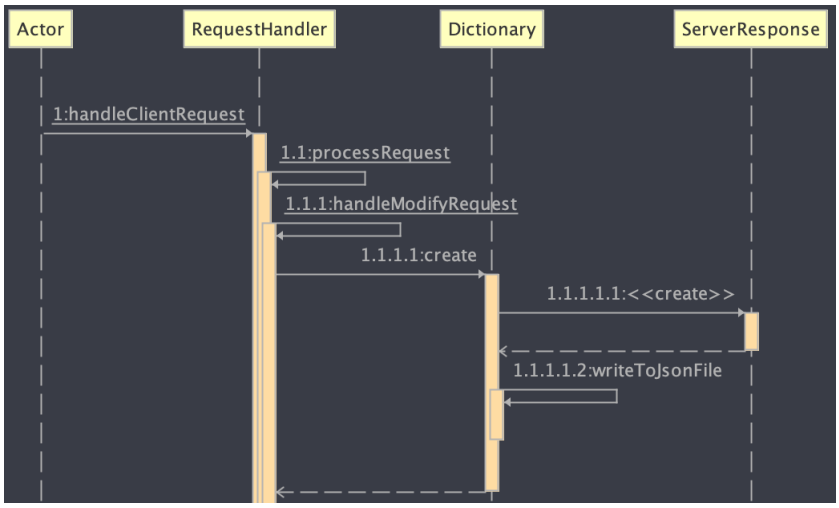

Figure 7: Workflow

5.2 Concurrency Handling

To manage concurrent access by multiple clients, methods that modify the dictionary (create, update and delete) are synchronized. This synchronization ensures that changes to the dictionary are thread-safe and prevent data races and inconsistencies.

5.3 Workflow

Figure 7 illustrates the sequence diagram for a `CREATE` request workflow involving both the server and the dictionary. Upon receiving the request from the client, the `RequestHandler` determines the method of the request. It then delegates the action to the Dictionary, which processes the request and generates a `ServerResponse`. Additionally, it saves the changes from this operation and updates the JSON file stored on disk by invoking the `writeToJsonFile` function. Similar workflows apply to other request types.

6 Additional Feature

6.1 Own Implementation of a Thread Pool

In my implementation of a thread pool (Figure 8), there is a `LinkedList` that serves as a queue for tasks waiting to be executed, ensuring tasks are executed in a first-come, first-served order. Additionally, a list holds the worker threads. Each worker thread continually monitors the task queue for new tasks.

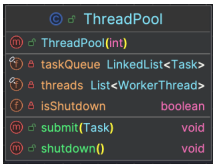

Figure 8: Thread Pool

Upon discovering a task, it removes the task from the queue and executes it. If the task queue is empty, the thread waits until a new task is added or until the thread pool is shut down. This waiting mechanism employs the `wait()` method, which releases the lock (monitor) on the task queue, thereby allowing other threads to add tasks or initiate a shutdown. When notified, a thread will exit if the shutdown has been initiated or otherwise continue processing tasks.

Adding tasks to the queue is synchronized, ensuring that only one thread can add a task to the queue or modify it at any time. After a task is added, the `notify()` method is called to wake up one of the waiting worker threads.

The shutdown of the thread pool sets the `isShutdown` flag to true and interrupts all threads. The threads then check this flag upon completing each task or when they are awakened, and will terminate if the flag is set. This ensures that no new tasks commence once the shutdown process has begun.

6.2 Analysis

Using a worker pool architecture for thread management has its pros and cons. This approach reduces the overhead linked to creating and destroying threads, thereby enhancing control and measurement of computational resources, which improves scalability. It also helps prevent server overload during high concurrency periods. However, if the pool size is too small compared to the server's capacity, it could lead to resource underutilization during peak traffic times, causing longer response times. Essentially, a fixed pool size does not adapt dynamically to changing demands. To address this, implementing a scalable adjustment strategy for the pool size might be advantageous. Additionally, it's important to note that for smaller applications with fewer requests, the overhead of managing a worker pool architecture may not be justified by its benefits.

7 Conclusion

In conclusion, this project has effectively created a multi-threaded client-server dictionary application with a graphical user interface for the client. The server is proficient in managing simultaneous requests from various users, achieved by using a custom worker thread pool. Security is strictly maintained with thorough validations on both client and server sides, providing strong defense against common vulnerabilities and preserving data integrity throughout the application.