COMP90015
Distributed Systems

Assignment #1 - Multi-threaded Dictionary Server

Xing Yang Goh
#1001969
Apr 5, 2023

1 Problem Description

A client-server architecture for a dictionary service will be developed, implementing multi-threading capabilities to handle multiple concurrent client requests with explicit sockets and thread handling. This dictionary server will maintain the data in memory with the following functionalities:

1. Query the meaning(s) of a given word
2. Adding a new word
3. Remove an existing word
4. Update the meaning(s) of an existing word

2 System Design Diagram

Figure 1: Dictionary service UML diagram

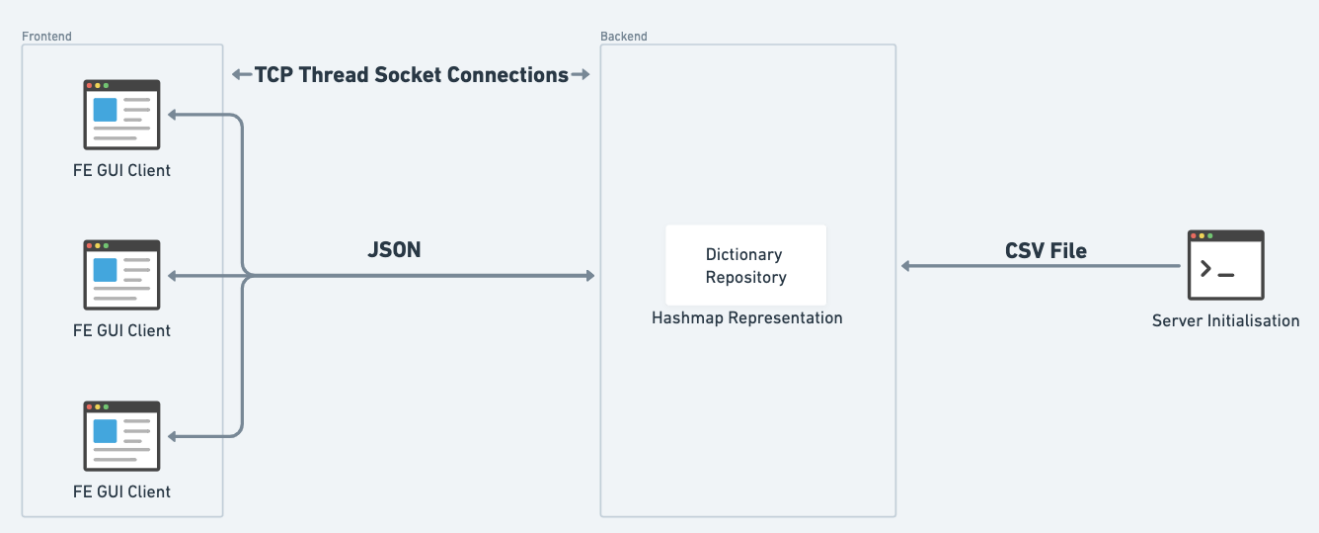

Figure 2: Dictionary service architecture

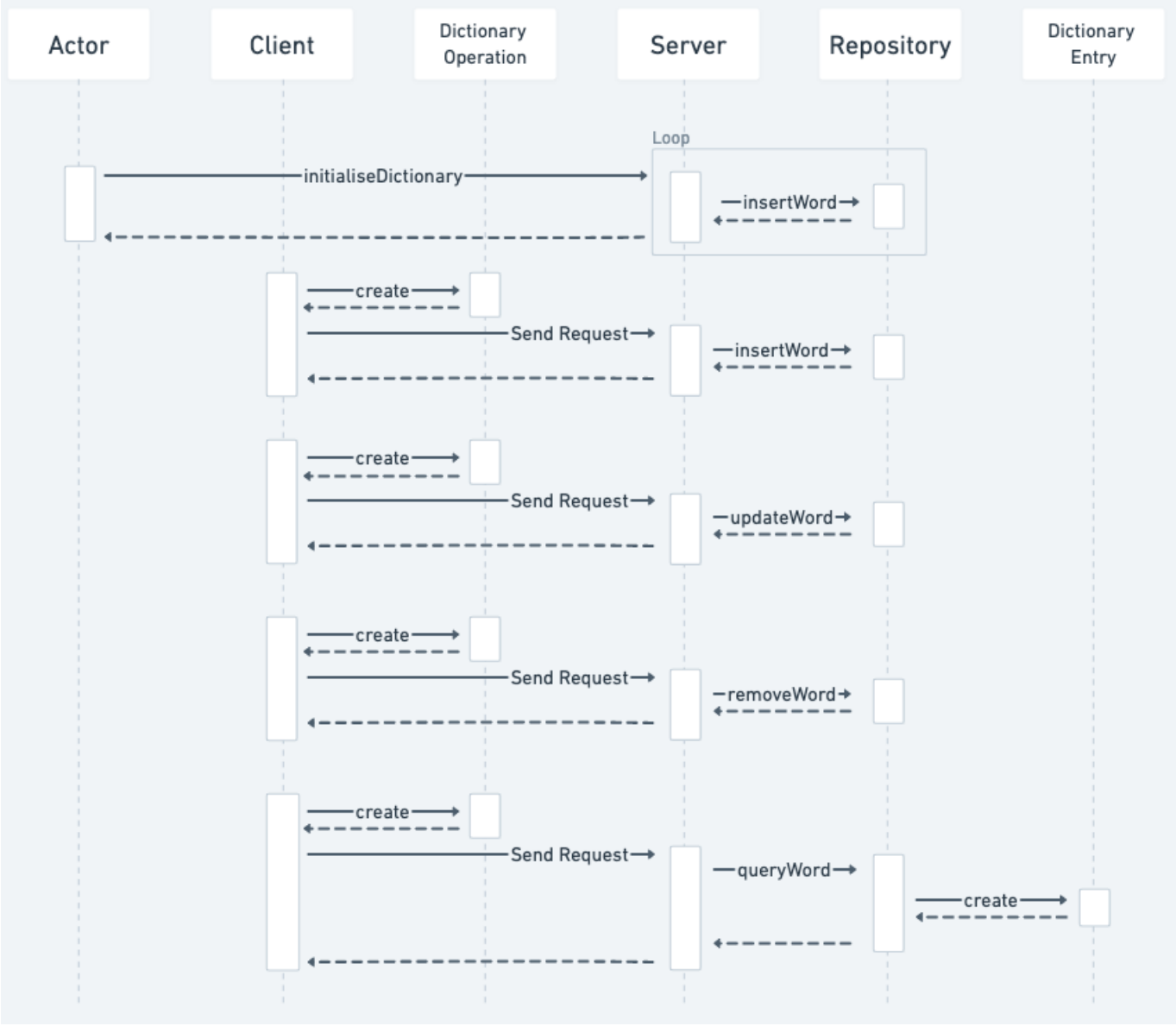

Figure 3: Dictionary service sequence diagram

3 Implementation

3.1 Client-Server Connection

Figure 2 describes the high-level system architecture used to implement this dictionary service. Starting from the Front-end, multiple clients can run simultaneously all interacting with the dictionary server socket running on port 4887 using a GUI client. To form a connection between these clients and the Back-end server, a TCP thread socket connection is established to form a three-way handshake that creates a new client socket on a new port to facilitate the data transfer, while keeping the server socket running on port 4887 to listen for new client connection requests. This is established with the following steps:

1. **SYN**: The client sends a Synchronize Sequence number (SYN) data packet to the server to initiate a connection request.

2. **SYN/ACK**: The server receives this SYN from the client request and returns confirmation with an Acknowledgement Sequence Number (ACK), known as an SYN/ACK packet.

3. **ACK**: The client receives the SYN/ACK sent by the server and responds with an ACK packet. This establishes the connection between the client and server to allow for communication.

3.2 Data Transfer

After the formation of a TCP connection, we can begin sending data between the server and client in a reliable manner. To facilitate this data transfer, a JSON format is used as a language-independent, universal data format, which represents data in a text-based format that uses nested key-value pairs to transfer information which lends itself to describing Java classes. As seen in Figure 1, the first class used is the DictionaryEntry class, which contains a String *word* and an array String [] *meanings* for the server to respond to client queries (as there can be multiple meanings to a word). Additionally, the DictionaryOperation class allows the client to send requests to the server (as shown in Figure 3), which has the same fields as the DictionaryEntry with the extra enum class *operation* to determine if the client wants to perform an update, query, add or remove action. These classes that facilitate communication between the client and server are serialised/deserialised using the Jackson package library in the Codecs class shown in Figure 1.

3.3 Server Initialisation & Repository Structure

Since the data is preserved in memory during execution, a hash-map implementation with the String word as a key and an array of String [] meanings as the value to facilitate $\mathcal{O}(1)$ insertion and lookup. To initialise the server, we can provide a CSV file that contains words and meanings to populate the dictionary. This process can be seen in the server initialisation sequence in Figure 3, where the entries in the initial dictionary CSV file are added into the repository through an insertWord loop that executes based on the number of lines or entries in the initial dictionary CSV file. Note that the words are set to lowercase for the dictionary keys to prevent duplicate words, and capitalised during the response. Additionally, since meanings can contain comma , characters, we use the delimiter with quotations and comma ”,” and contain the meanings in quotations to split the meanings appropriately. These dictionary operations can be referenced in Figure 1 under the class DictionaryRepository.

3.4 Thread Handling

For the multi-threading requirements, a thread-pool executor is used to handle multiple client requests. This architecture uses a core pool size of 10 threads that can increase to a max pool size of 15 which submits the runnable interface ServeRequest to the thread-pool executor, using an ArrayBlockingQueue to sequentially process the client requests. This thread-pool executor is used to control the maximum number of threads the server can create to ensure the server won't be overloaded by making too many

threads, however, it comes at the disadvantage of slower response time during high-volumes of requests as it will be placed in a queue until a thread is available to request. Additionally, these threads operate on a per-request basis, where a connection is formed when a DictionaryOperation request is ready to be sent, and the connection and thread are subsequently closed after a response from the dictionary server. These per-request threads are used as there is no need for a constant communication stream between the server and client, which will allow the threads to be managed much better in comparison to a per-connection thread as each operation should be completed very quickly with the $\mathcal{O}(1)$ hash-map operations on the server-side.

3.5 Concurrency Handling

To handle concurrency for the dictionary server, any client request that alters the repository is handled with the synchronized keyword. These include the insertWord, updateWord and removeWord operations as seen in the DicitonaryRepository class in figure 1. This ensures that only one thread executes at a time for all the synchronized methods by using a lock associated with a monitor that encapsulates the shared resource and allows only one process to alter the dictionary repository at any instance. Note that the queryWord method does not use this monitor lock under the assumption that the response from querying the dictionary is not used for additional important operations and can be eventually consistent (i.e. adding an entry and querying for that entry concurrently can lead to a WORD NOT FOUND error, but eventually, subsequent queries for that entry will return the inserted entry).

3.6 Error Handling

To facilitate these per-request threads, the server will have to send a response to the client regarding the status of the request, which responds with a Boolean true or DictionaryEntry for successful operations, a null response from connection errors and the following list of potential enum class ErrorCodes with the following output mapping for the client as referenced from the ServerError class in Figure 1:

1. **WORD NOT PROVIDED**: No word specified to the operation
2. **WORD NOT FOUND**: The given word does not exist in the dictionary
3. **INVALID WORD FORMAT**: Given word is in an invalid format
4. **MEANING NOT PROVIDED**: No meaning provided for the given word
5. **WORD ALREADY EXISTS**: Given word already exists in the dictionary
6. **UNEXPECTED OPERATION**: Unexpected operation provided

Additionally, methods are captured in try-catch blocks to identify different sorts of errors to log to the server such as various IOExceptions for creating and closing client connections, FileNotFoundExceptions during dictionary initialisation and JsonProcessingException for JSON serialisation/deserialisation errors.

3.7 Load Testing

Testing the server with load testing is achieved by using Fixtures with the Easy Random library to generate 1000+ random dictionary operations to send to the server in a for loop and monitor the thread pool. This demonstrates the server's ability to manage the thread pool using the ArrayBlockingQueue to sequentially handle requests asynchronously, and the threads awaiting a response from the server before closing for a per-request connection.

4 Client GUI

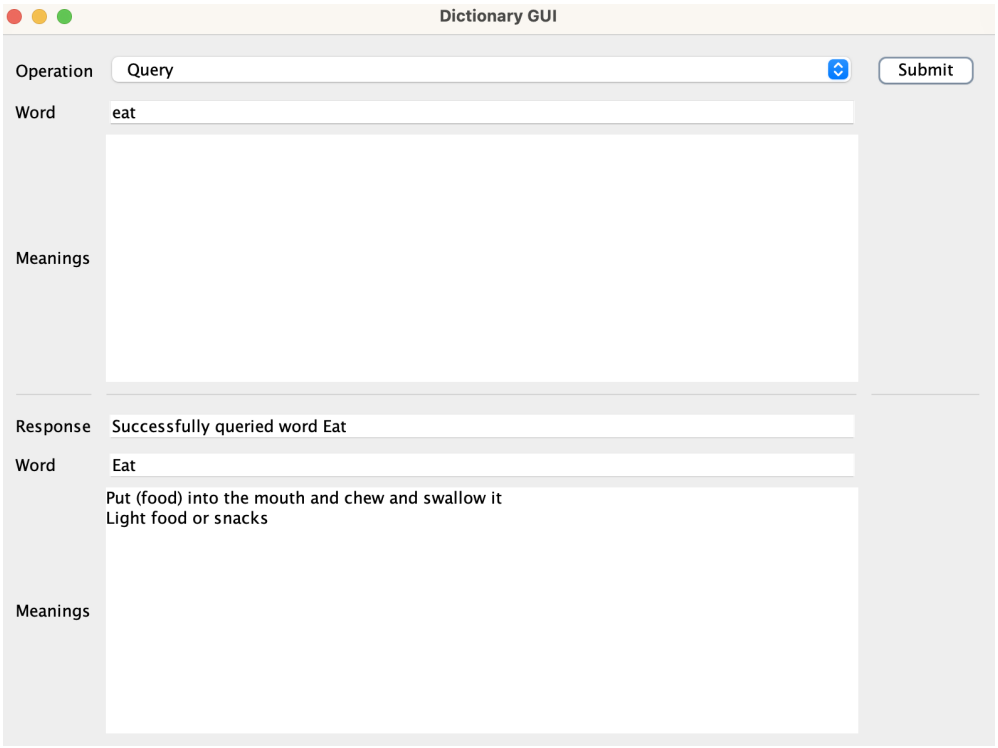

Figure 4: Client GUI performing a query

The Client GUI is designed using the Java Swing framework as seen with the DictionaryClient class in Figure 1. To select operations, there is a drop-down menu to select from Add, Query, Update and Remove operations and a submit button at the side that attempts to form a connection with the server and obtain a response. Note that for the query and remove operations, the meanings text area is cleared and is not editable to convey the functionality in an interactive way. All the operations will return a server response that conveys the result of the action, with errors returning a response as shown in Section 3.6 with an additional popup message to alert the user regarding the error. Additionally, note that the meanings fields can warp the text to subsequent lines as they can be of arbitrary length. Finally, the word and meanings response fields and labels as seen in Figure 4 are only shown on query operations, and will not be visible for other operations to reduce visual clutter from empty fields.

5 Conclusion

The client-server architecture for a dictionary service has been implemented, using a reliable three-way handshake TCP connection between a client and server, with a thread pool executor that performs per-request actions when clients submit an operation. The transfer of data between this client and server uses the language-independent JSON data format using Jackson to serialise and deserialise the data. For the repository, a hash-map is used for in-memory data storage to facilitate $\mathcal{O}(1)$ insertion and lookup, with the capability to initialise the repository using an initial dictionary CSV file. Concurrency for this multi-threaded server is handled using synchronized methods for methods that alter the dictionary repository, which implements a lock associated with a monitor to ensure only one process attempts to alter the repository at a time. Additionally, the server response has comprehensive error messages that are provided to the client to aid the client's understanding of the problem encountered, and additional logging for the server errors in try-catch blocks to aid unexpected errors for troubleshooting. This server is accompanied by a client GUI that uses simple drop-down menus and buttons, along with hiding unnecessary information from the client and making fields uneditable to convey the functionality interactively while presenting the results in a clear manner using responses and popups without unnecessary clutter.