Mini Dictionary (Documentation)

Vincent Kurniawan
The University of Melbourne

Problem Context

1.1 Introduction

This report addresses the choice of design, implementation, and analysis of a client-server architecture model of a multi-threaded dictionary server in which multiple clients are able to connect to the server and perform operations concurrently. As outlined in the project specifications, the project is implemented in Java language.

1.2 Functional Requirements

For this project, the server parses a given dictionary file and allows clients to perform the following functional operations on the dictionary:

1. Query the meaning(s) of a given keyword.
2. Add a new keyword with its associated meaning(s).
3. Remove an existing keyword and its associated meaning(s). 4. Update an existing keyword and its associated meaning(s).

1.3 System Requirements

Communication of the implemented client-server architecture should be reliable and must take place via sockets. As concurrency must be upheld across operations from multiple clients, the system uses a Transmission Control Protocol (TCP). The system must also utilize a chosen message exchange protocol. Hence, this system parses incoming requests, outgoing responses, and the dictionary itself in JSON format.

The server must also be able to execute multiple incoming client requests through the use of threads. Hence, the server uses a thread-pool architecture consisting of a fixed number of threads that are ready to execute these requests, where each thread handles a single request one at a time. To determine which request gets executed first, a priority queue is used as the data structure to store incoming requests and prioritized them based on the arrival of the requests to the server.

Furthermore, A graphical User Interface (GUI) is also required for clients to interact with the server. Additionally, the system also provides a GUI for the server which is further explained in the report.

Lastly, any errors and responses associated with a particular request are also required to be visually displayed on the client GUIs. An implementation to display and handle failures or errors that may occur throughout server-client interactions is provided by the system.

System Components

2.1 Project Structure

Figure 1. shows the overview structure of the packages and its associated Java classes that are used in this project. The source folder "java" is divided into 5 separate packages: Server, Client, GUI, Dictionary, and Response. The use of packages provides better encapsulation, management, and reusability of classes. A sample dictionary file "Dictionary.json" is also provided under the resources folder.

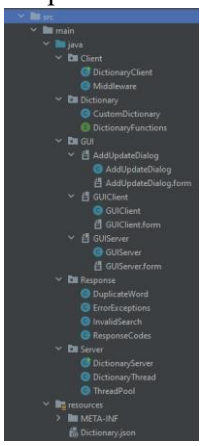

Figure 1: Project structure of system components, including packages and its associated classes

2.2 Server

The Server package contains the classes that are responsible for handling and processing incoming requests from clients. Once the main class "DictionaryServer.java" is instantiated and running, it binds a socket to the specified <port number> and parses the dictionary file given in <dictionary path>. The server also generates a thread-pool with a fixed sample size of 5 and a priority blocking queue with a limit of 10. Once the thread-pool is instantiated, it starts running 5 threads of private class "DictionaryThreads.java" which are ready to execute incoming requests/tasks of type "DictionaryThread.java". Depending on which client request arrives first at the server, they are stored at the priority queue and wait until a thread is available at the thread-pool. The client can interact and perform requests of the operations above by giving input parameter(s) to the client GUI, and the server will query these requests and visually display the output. The server then instantiates the GUI of the server. During the execution of an instance of "DictionaryThread.java", this class parses incoming client requests into the type of the request and the actual client input (i.e., keyword). Once the server has finished processing the request, it sends a response back to the designated client.

2.3 Client

The Client package contains the classes that are responsible for establishing connection between the client and the designated server. Once the main class "DictionaryClient.java' is instantiated and running, it binds a socket to the specified <port number> and <server address>, as well as instantiating the GUI of the client.

The package also contains the "Middleware.java" class, in which the class acts as connection hub between the client and the server to parse, receive, and send messages across endpoints. It is also responsible for displaying the response from server to a particular client's request in the client's GUI, including success, failure, and error responses. Furthermore, in the event where a server's connection to the client has been interrupted, a notification indicating this prompt will appear and then shuts down the GUI.

2.4 GUI

The GUI package has classes that contain the logic of GUI format display for server, client, and pop-up dialog; this information is stored in the forms of each class. The GUI toolkit used for this project is Java Swing and it is chosen due to its convenience and provided layout management system to easily add buttons, labels, and text fields, and update them accordingly based on client's inputs and events (i.e., pressing a button to submit request with an input to server).

The Server GUI as shown in Figure 2. provides user information of (1) total clients that are currently connected to the server, (2) size of the thread-pool, (3) limit of priority queue, (4) the latest request sent by any client to the server, and the (5) option to shut down the server. Furthermore, the GUI is capable of displaying a notification stating which client has just disconnected from the server as shown in Figure 3.

The Client GUI as shown in Figure 4. provides (6) a single text field for clients to input a keyword, along with buttons that correspond to a particular query type: (7) Search, (8) Add, (9) Update, and (10) Remove. Once a button is clicked, it sends the corresponding query type and input to the server. Response from server will then be displayed at the text area below the (11) text field. In the events where an Add button is clicked and the input keyword is not found in the dictionary, or an Update button is clicked and the inputted keyword is found in the dictionary, an AddUpdateDialog GUI will pop-up on top of the client GUI as shown in Figure 5. The pop-up GUI allows clients to (12, 13) add/update meaning(s) of (14) associated keyword to the dictionary, and it will return to the client GUI once such action has been performed or cancelled depending on dialog response as shown in Figure 6.

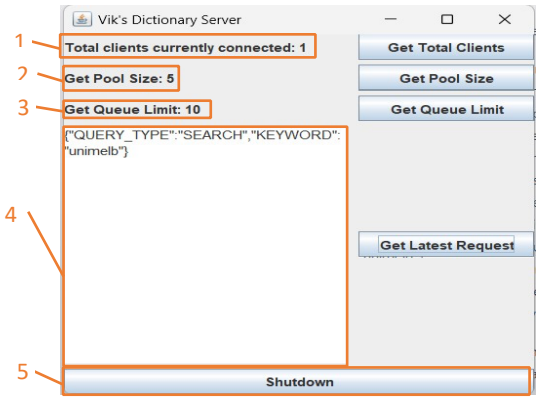

Figure 2: Layout of Server GUI

Figure 3: Dialog notification that a client has disconnected from the server

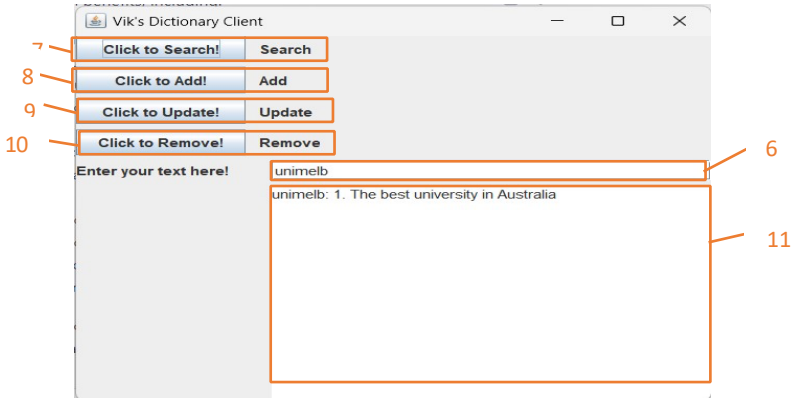

Figure 4: Layout of Client GUI

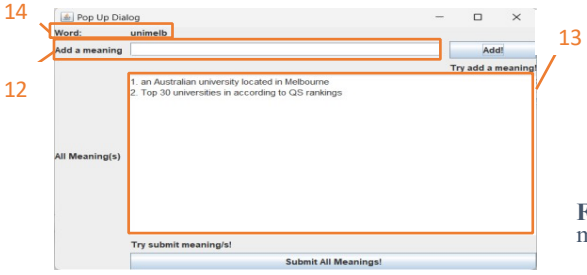

Figure 5: Layout of AddUpdate GUI

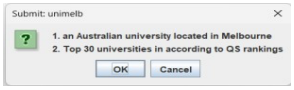

Figure 6: Dialog notification of meaning(s) to be updated/added.

2.5 Dictionary

The Dictionary package contains the "CustomDictionary.java" class that's responsible for parsing the given dictionary and providing the operational capabilities that the server is able to perform with the dictionary. This class implements the "DictionaryFunctions.java" interface that declares these abstract operations. To uphold concurrency between client operations, the dictionary's methods have synchronized access to search, add, update, and remove operations to eliminate data inconsistencies during server-client sessions.

2.6 Response

Lastly, the Response package contains the "ErrorExceptions.java" class that holds custom exceptions that can be thrown by the server and/or client accordingly depending on the errors caught. An exception may either cause the server/client to quit with the "showError" method or display a notification to the server/client with the "notifyServer" method. Each exception is supplied with the error code, description, and message; information of all errors that can be thrown by the exceptions provided is shown in Table 1.

Error Code	*Error Constant*	*Error Description*
400	ARGUMENTS_LENGTH_ERROR	There should only be 2 arguments inputted: <server port> <dictionary path>
401	SERVER_PORT_ERROR	Inputted server port has an invalid format
402	DICTIONARY_PARSE_ERROR	Dictionary file is not in proper JSON format
403	DICTIONARY_PATH_ERROR	Dictionary path could not be found
404	BIND_PORT_ERROR	Port provided is currently in used or bind by another application
405	INVALID_CLIENT_REQUEST	Invalid request format sent by client
406	QUEUE_WAIT_ERROR	Error in waiting for priority queue
407	CLIENT_DISCONNECT	Client disconnected from the server. Sends a notification to the server.
408	INVALID_QUERY_TYPE	Client sent an unknown query that's not recognized/supported by the server
409	THREAD_PROCESS_ERROR	Error in executing a thread
410	PARSE_REQUEST_ERROR	Server fails to parse the request sent by client
411	CONNECT_REFUSED_ERROR	Client fails to connect to the server
412	UNKNOWN_HOST_ERROR	Unable to connect to specified host/domain name
413	SERVER_DISCONNECTED	Server disconnected or shutdown, sends a notification to all clients connected to the server and shuts down the clients.
414	EMPTY_INPUT	Inputted keyword cannot be empty or null
415	TOO_MANY_WORDS	Can't perform query on 2 or more words at once
416	NON_ASCII	Inputted keyword is not in ASCII format

Table 1: List of error codes and its description that can be thrown by an exception at client/server

Furthermore, the package also contains the "ResponseCodes.java" class that holds the constants that specify the query type of a request and the invalid message associated with an invalid input given by client.

2.7 JAR Files

These packages are compiled as two separate Java executables in JAR format, namely DictionaryClient.jar and DictionaryServer.jar. To run these JAR files, Figures 7 and 8 below show the required arguments to be inputted in the command line for each JAR file.

```
> java -jar DictionaryServer.jar <port> <dictionary-file>
```

Figure 7: Command to run DictionaryServer.jar

```
> java -jar DictionaryClient.jar <server-address> <server-port>
```

Figure 8: Command to run DictionaryClient.jar

System Diagrams

3.1 Use Case Diagram

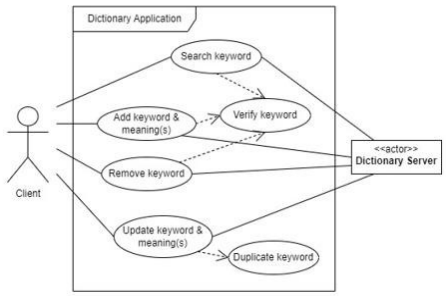

Figure 9 shows a high-level overview of interactions between the client and the server. These interactions include a client sending a request to search, add, update, and/or remove a keyword from the dictionary hosted by the server. The search, add, and remove operations depends on verifying whether the keyword queried exists in the dictionary or not, while the update keyword depends on verifying whether the same keyword exists in the dictionary.

Figure 9: Use case diagram of the dictionary application

3.2 UML Class Diagram

The overall class diagram of the system is shown in Figure 10 below and it is represented using a Unified Modelling Language (UML) model. The model encompasses the instantiation of classes that was previously discussed under the systems components section above.

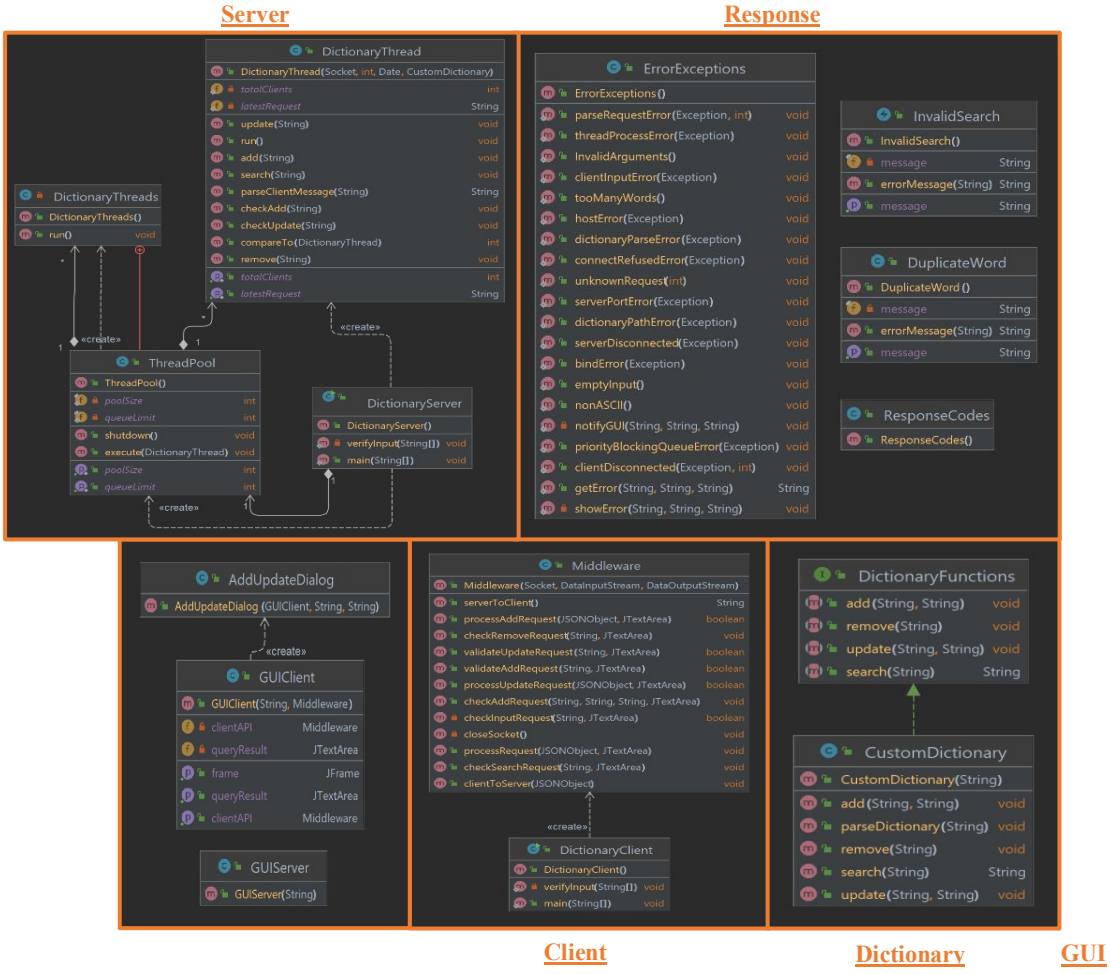

Figure 10: Overall UML class diagram of the system based on the packages organization

3.3 Server GUI Interaction Diagram

Figure 11 below shows an interaction sequence diagram of the server operator in the case where all the buttons are pressed on the server GUI shown in Figure 2 above.

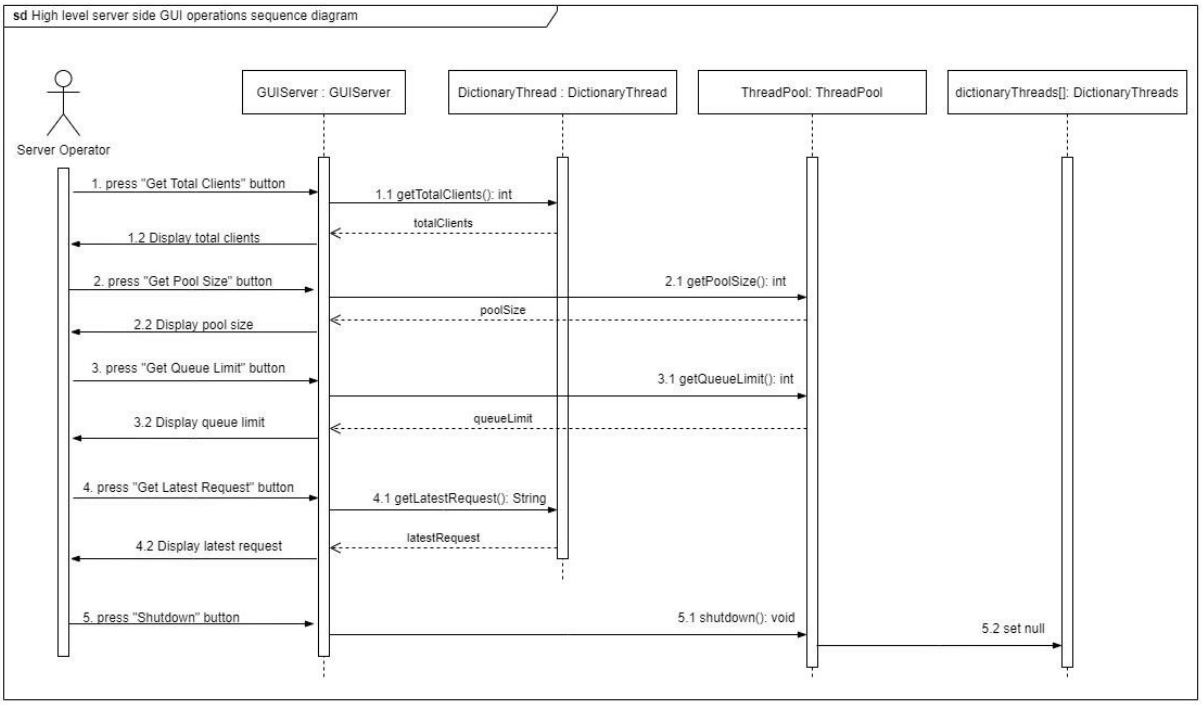

Figure 11: Interaction sequence diagram of a server operator pressing all buttons in order

Critical Analysis

4.1 TCP Protocol

Communication of the implemented client-server architecture is established using the Transmission Control Protocol (TCP), which is a connection-oriented communication protocol that ensures reliable communication and ordered flow of data between the clients and server. Since concurrency must be upheld throughout operations among clients (i.e., changes from one client should be visible to other clients connected to the same server), TCP is a suitable choice protocol for the project as it provides flow and congestion control of data transmission. To support multiple simultaneous connections, TCP uses sockets to bound to a specified port number so that TCP is able to identify the data's designated destination. If the default UDP communication protocol was to be implemented instead, it does not guarantee reliable data transmission, meaning flow of data might not arrive in an orderly manner; this may generate data integrity and inconsistency in retrieving dictionary queries. However, an advantage of implementing UDP is a faster data transmission speed as it does not have to guarantee the order at which data packets arrive.

4.2 Thread Pool & Priority Queue

To execute multiple incoming client requests, the server uses a thread-pool architecture consisting of a fixed number of threads that are ready to execute these requests, where each thread handles a single request one at a time. In contrast to a thread-per request architecture, for each incoming request, a new thread is created

to execute the requests. In the event where multiple concurrent requests are being sent to the server, creation of more threads is necessary to accommodate the supply and demand for these requests, which ultimately leads to higher resource consumption of the system and degrading performance. On the other hand, as a worker pool architecture already has a fixed number of threads readily available, it leads to less resource consumption as it does not create new threads for each incoming request. Hence, this allows more concurrent clients to send requests to the server without overloading the dictionary server.

To determine which request gets executed first, a priority queue is used as the chosen data structure to store incoming requests and prioritized them based on the arrival of the requests to the server. When a thread in the pools is readily available, the task/request with the highest priority is taken out of the queue and gets executed by this thread. The use of a priority queue in this system would allow add/update requests to be prioritized first before other clients may decide to search or remove a particular keyword. However, this system only implements the queue to be prioritized based on a first come first served order, which doesn't support such synchronization issue to be prevented.

4.3 Message Exchange Protocol

An advantage of using JSON as the message exchange data format for this system is the readily available and easy-to-use JSON library named "json-simple" which provides the fundamental operations that the system requires to parse the dictionary file, incoming requests from client, as well as outgoing responses from server. Furthermore, the format of a JSON object and a dictionary are intuitively similar, as both use a key-value data structure, which allows more convenient means of data retrieval. However, since the server needs to support adding or updating a keyword with multiple meanings, more effort and proper data management and display is needed to store this (i.e., parsing value of a key in ArrayList format).

Conclusion

Overall, the system that is discussed in this report provides the fundamental functionalities to design and implement a multi-thread server which allows concurrent clients to search the meaning(s) of a word, add or update a new word, and remove an existing word. This system adopts a client-server architecture that uses TCP to establish reliable client-server communication, as well as a worker pool architecture to handle creation of threads and execution of incoming client requests. These requests along with server responses are parsed and sent in JSON format, and the system also adopts a custom failure model that handles display and handling of errors and exceptions. Furthermore, GUIs of the server and client are also provided to allow client-server interactions.

Excellence Elements

5.1 Error Notifications & Exceptions in GUI

Clear notifications of errors through pop up dialogs in server and client GUIs. Each error is accompanied by a custom error code, description, and message to emphasize further clarity.

5.2 Comprehensive Analysis

Analysis in report highlights the pros and cons of the architecture of the implemented system. Justifications of design and implementation choices are also provided within the analysis.

5.3 Comprehensive Graphs & Figures

Report highlights the overall use case diagram, UML diagram of classes, and a sample interaction sequence diagram between a server operator and the server GUI of the system. Each graph and figure throughout the report is referenced and provided with relevant explanations.

5.4 Comprehensive Report Structure

Report is well structured in headings and sections for better readability and understandability.

Creativity Elements

6.1 Implementation of Server GUI

A GUI for server is implemented to retrieve user information of (1) total clients that are currently connected to the server, (2) size of the thread-pool, (3) limit of priority queue, (4) the latest request sent by any client to the server, and the (5) option to shut down the server.

6.2 Custom Thread Pool Implementation

The thread pool used in the system is self-implemented (not by Java) with a priority queue to store incoming client tasks. A custom pool size and queue limit is able to be set.

6.3 Custom GUI to Add/Update Multiple Meanings

An additional GUI “AddUpdateGUI” is implemented to handle clients inputting multiple meanings to a single keyword. GUI is designed to allow meanings that are entered to be visible to the user. This GUI also ensures that a word being added or updated cannot have its meaning to be empty.

6.4 Custom Handling of Server/Client Disconnections

The server GUI is capable of displaying a pop-up notification stating which client has just got disconnected from the server. While in the client GUI, when the server disconnects, all connected clients will receive an error pop-up once they try to send a request with a valid argument (keyword), hence forcing the clients to close.