COMP90015: Distributed System

Assessment1

Student Name:
Zhuoyang Liu

Student ID: 917183

In this assessment, an implementation of a multi-threaded dictionary server facilitating concurrent access by multiple clients is presented. Leveraging multi-threading configurations, the server enables simultaneous client access. The communication protocol adopted herein is Transmission Control Protocol (TCP), utilized to establish connections, wherein sockets facilitate a "three-way handshake" to facilitate client-server connectivity. Despite TCP's inherent drawbacks relative to User Datagram Protocol (UDP), its capacity to ensure connection stability to the utmost degree, thereby guaranteeing reception of every client-sent request by the server, is paramount, particularly in the context of dictionary applications.

Within the present framework, graphical user interfaces (GUIs) have been devised for both the server and client. The server GUI serves the dual purpose of exhibiting the current number of connected clients and allotting a Universally Unique Identifier (UUID) to each client upon connection establishment, thereby facilitating subsequent optimization and administrative management endeavors. Concurrently, diverse client requests are visualized within the server GUI interface, thereby affording administrators the ability to oversee server maintenance.

Conversely, the client GUI assumes a relatively intricate configuration. To ensure compliance with the four requisite functions delineated in this assessment, three text input fields have been incorporated, encompassing target words, corresponding definitions for unselectable target words, and optional new word definitions. Additionally, a selection bar is integrated into the GUI to aid users in specifying their requested queries. Furthermore, the client GUI undertakes validation of user-entered words and definitions, permitting only those meeting stipulated criteria to advance to subsequent stages. Upon initiating the request via the designated button located in the upper-right corner of the client GUI, outcomes relayed by the server are promptly exhibited on the client interface.

Concurrently, meticulous provisions have been made by both the client and server to preempt potential errors stemming from various sources, including network disruptions resulting from server or client disconnection, I/O anomalies during the data retrieval process, and the spectrum of input-related concerns. These comprehensive measures afford users a sense of assurance, mitigating apprehensions regarding potential adversities that may inflict significant detriment.

Subsequently, to augment the server's efficacy in data aggregation and processing, a local JSON file has been instantiated to store the amassed data. This strategic augmentation facilitates streamlined manipulation of the data encompassed within said file within the program. Notably, upon server initialization, a hashmap is instantiated to encapsulate the

data retrieved from the local JSON file. Herein, the word's orthography serves as the hashmap's key, while its associated definition constitutes the corresponding value.

It is pertinent to underscore that in this assessment, I opted for the thread-per-request approach, a method known for its ease of deployment. In this approach, upon receipt of each request, the server instantiates a dedicated thread to handle the incoming task. Each such thread operates autonomously, enjoying full authority over its assigned task. Consequently, this framework renders each thread relatively independent of others, with limited inter-thread communication. Moreover, in contrast to the worker pool paradigm, which entails the establishment of a dedicated pool of threads poised to handle incoming requests, the thread-per-request approach offers a more streamlined and lightweight alternative.

Notably, in the Java environment, the utilization of blocking and asynchronous methodologies akin to those available in GoLang for dynamic work pool instantiation is precluded. Employing such methods would substantially escalate the computational overhead and space complexity of the server infrastructure. This limitation constitutes a primary factor motivating the rejection of the worker pool model in favor of the thread-per-request approach.

It warrants mention that given the utilization of a multi-threaded approach by the dictionary server, inherent risks of thread conflicts and deadlocks may arise. Consequently, synchronization mechanisms and analogous strategies have been implemented within the server framework to meticulously regulate concurrent operations. Specifically, operations such as local JSON file reading and writing are orchestrated to ensure singular thread occupancy, thereby fostering the expeditious resolution of potential thread safety concerns.

Simultaneously, both the client and server have undergone extensive preparations to preemptively address potential errors, ranging from network disruptions induced by server or client disconnection to I/O anomalies encountered during data retrieval, and encompassing a spectrum of input-related contingencies. As a result of these comprehensive measures, users are relieved of concerns regarding potential errors and their attendant adverse ramifications.

Subsequent enhancements aimed at facilitating data collection and processing within the server involved the creation of a corresponding JSON file locally to archive the data amassed by the server. In conjunction, a hashmap has been established to facilitate programmatic access to the data contained within this file upon server initialization. The orthographic representation of words serves as the hashmap's keys, with their associated definitions comprising the corresponding values. Furthermore, in light of the employment of a multi-threaded paradigm by the dictionary server, potential complications arising from thread

conflicts and deadlocks necessitated the implementation of synchronization mechanisms and associated strategies within the server framework. These mechanisms effectively ensure singular thread occupancy during operations such as local JSON file reading and writing, thereby effectively mitigating potential thread safety concerns.

The extant dictionary server encompasses four distinct functionalities: addition, deletion, search, and update.

- Addition: Users are required to furnish the words they wish to append to the dictionary along with their corresponding definitions via the client interface. Subsequently, upon clicking the "send" button, the local database (JSON file) is modified through server-mediated interactions. Prior to transmission, the client conducts requisite validation checks to ensure the completeness of user-provided information. This request is then relayed via the established TCP link to the corresponding server thread, which, upon interpretation, effects modifications within the local database. In cases where the user attempts to add a word already existing within the dictionary, the server will prompt notifications to this effect, which are then displayed on the client interface.

- Deletion: Users endeavor to remove a word from the dictionary through the client interface. Differing from the addition process, deletion necessitates solely the submission of the target word for removal. The server responds by confirming the successful deletion of the specified word or indicating its nonexistence within the dictionary.

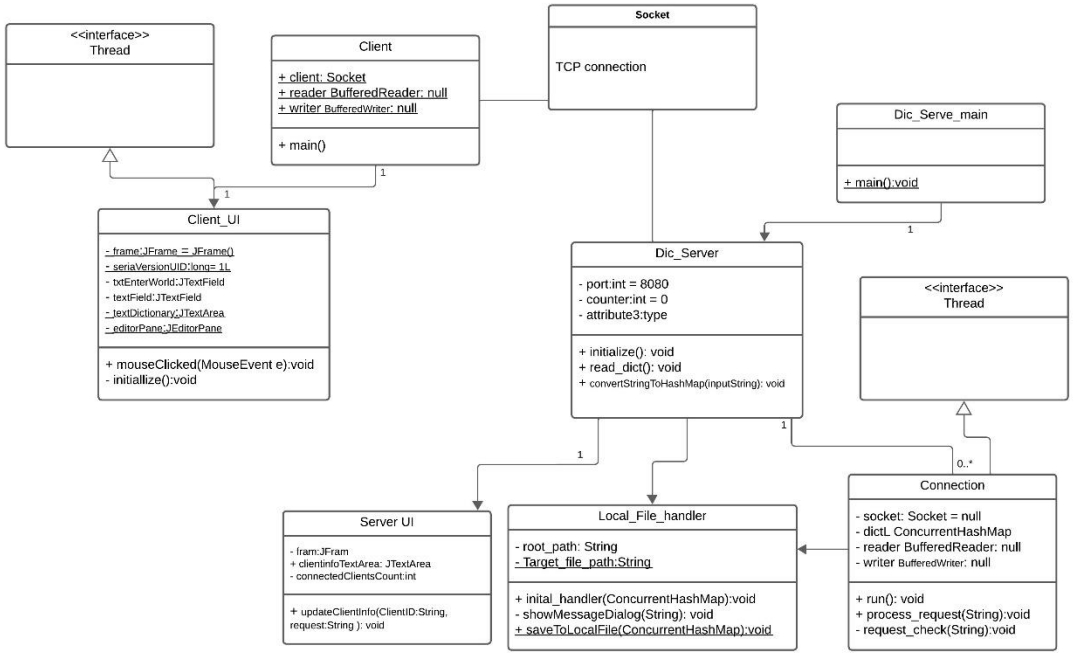
- Search: Users employ the client interface to query the meanings of specified words. The client securely transmits the articulated query to the corresponding server thread via TCP, whereupon the server furnishes corresponding results based on the search criteria.

- Update: Conceptually akin to addition, updating mandates users to provide both the target word and its corresponding definition. However, if the meaning of the word intended for update mirrors that already existing within the dictionary, the update operation is bypassed, and the user is notified via a specialized message relayed by the client interface.

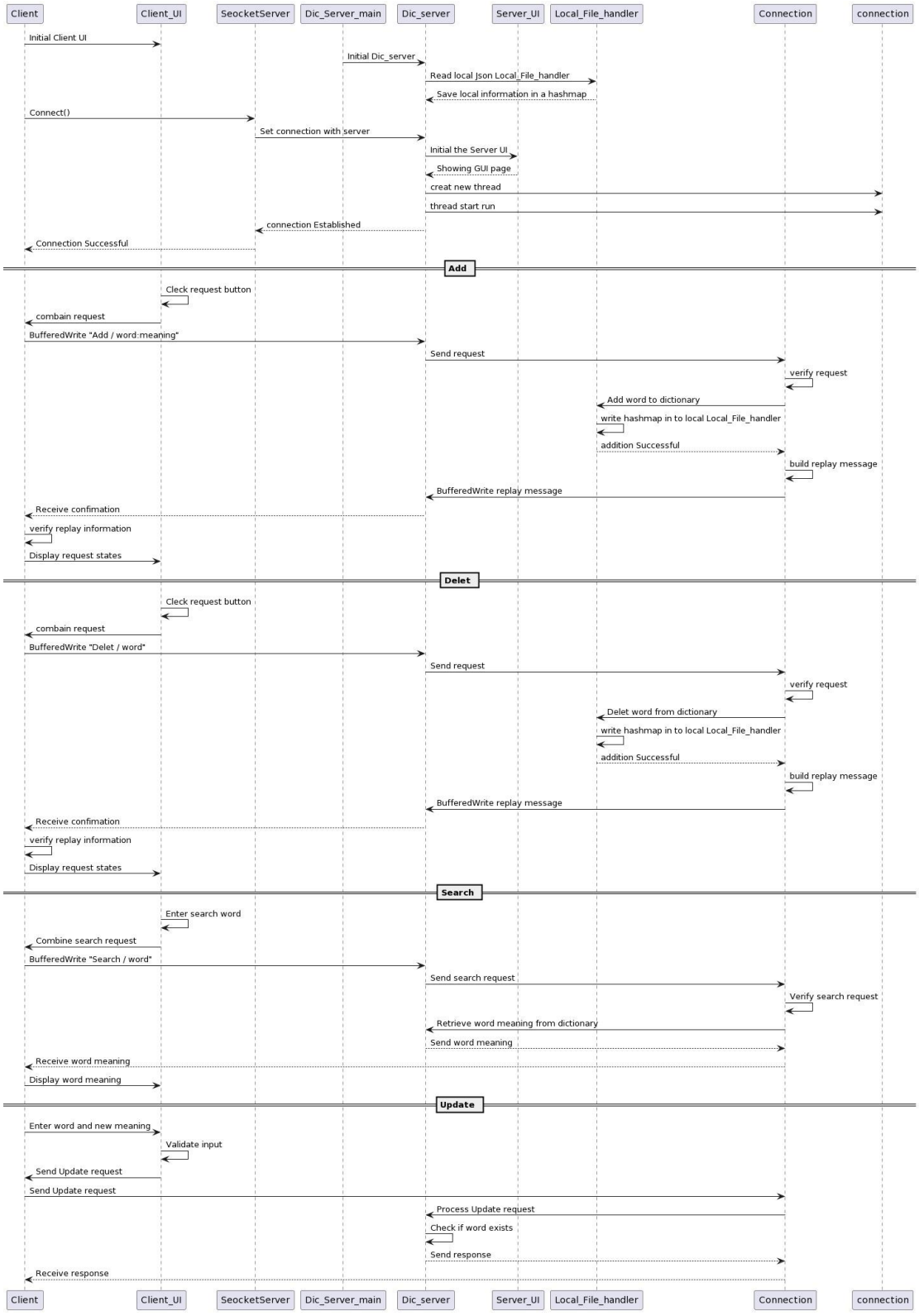
In this section, I will present the Class Design diagram and sequence diagram to illustrate the interaction between the dictionary server and the client comprehensively.

The entire project exhibits a clear division into two distinct components: the client and the server, which communicate via sockets establishing TCP connections. The client aspect encompasses the Client_UI, a graphical user interface constructed using Swing Window Builder, encapsulating interface design and user interaction functionalities. The logic flow and interaction with the server are orchestrated within the Client function. This module not only oversees the establishment of socket connections and transmission of requests to the server but also conducts preliminary validation of user input from the GUI.

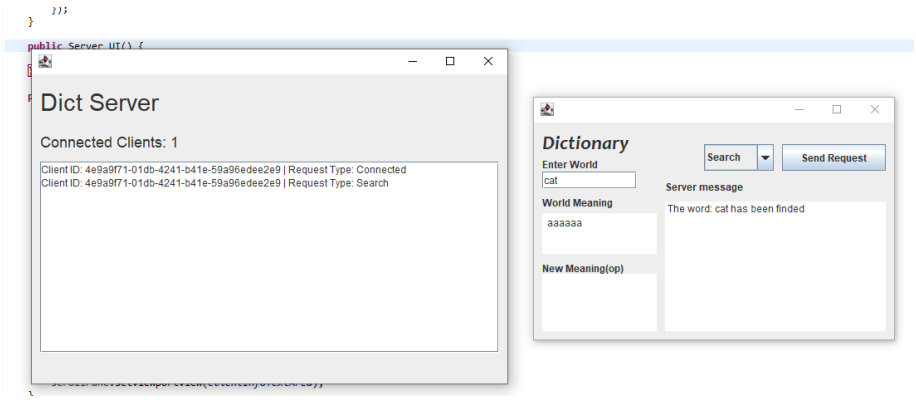
On the server side, the architecture unfolds with Dic_server_main serving as the pivotal entry point. It assumes responsibility for initializing Dic_Server, which, in turn, manages communication with the client. Upon receiving a connection request from a client, Dic_Server orchestrates connection processing by instantiating a new Connection class and corresponding thread. Notably, the server's GUI is instantiated within Dic_server, alongside the invocation of a local_file_handler class tasked with ensuring the seamless operation of the local database (JSON file). Additionally, strategic design patterns such as the adapter design pattern are deployed to facilitate compatibility with diverse types of local databases, including TXT files. Furthermore, a factory design pattern is integrated within the connection module, serving to dynamically produce tailored responses corresponding to different types of requests.

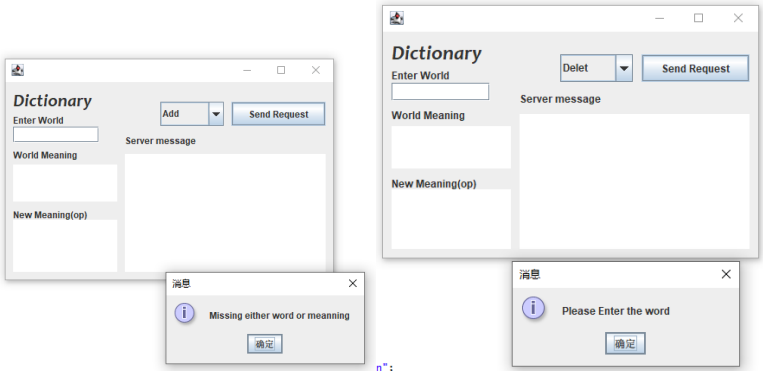


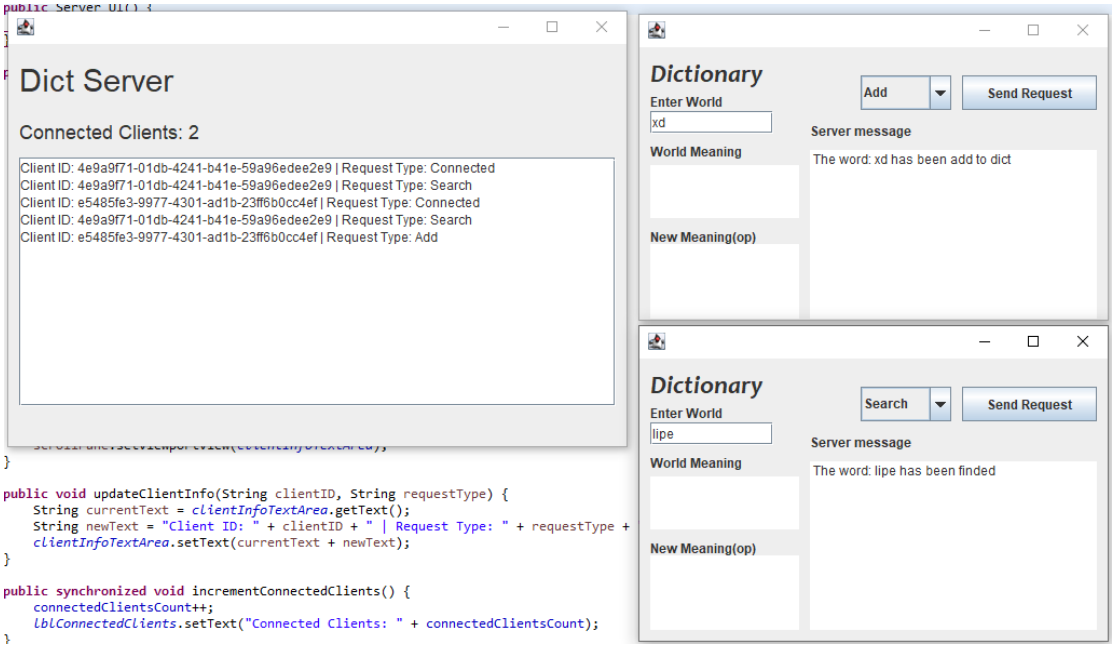
The sequence diagram provides an expanded depiction elucidating the operational workflow of the entire project. It delineates four distinct requests to facilitate a comprehensive understanding of the processThis sequential elucidation elucidates the intricate communication and interaction between the client and server components, elucidating the operational nuances of each request type within the project.



Few GUI details will be shown in the form of screenshots :








```
}

public void updateClientInfo(String clientID, String requestType) {
    String currentText = clientInfoTextArea.getText();
    String newText = "Client ID: " + clientID + " | Request Type: " + requestType +
    clientInfoTextArea.setText(currentText + newText);
}

public synchronized void incrementConnectedClients() {
    connectedClientsCount++;
    lblConnectedClients.setText("Connected Clients: " + connectedClientsCount);
}
```