COMP 90015: Distributed Systems – Assignment 1
Multi-threaded Dictionary Server

Name: Jiasheng Yang
StudentID: 1464801
Instructor: Dr. Siddharth
Tutor: Mashnoon Islam
Date: April 8, 2024

1. Project Description

The project is designed and developed by a **server-client** model employing the **multi-threaded server** architecture; the server should be capable of handling multiple clients' requests, including **search**, **add**, **update**, and **delete** operations in the dictionary concurrently.

2. Introduction

The program is based on **Sockets** and **Threads** technologies.
The report illustrates the analysis of architecture, class and interaction designs, and implementation, especially focusing on why choose them.
The remaining parts mainly introduce the innovative aspects and shortcomings.

3. Architecture

3.1. Communication Socket

TCP is the communication socket for the multi-thread dictionary.
Analysis (why chose):
Transmission Control Protocol (TCP), a connection-oriented communication mechanism, is the reliable communication protocol, which ensures the integrity, order, and reliability of data transmission. In addition, it also offers data retransmission and error detection features.
On the contrary, *User Datagram Protocol (UDP)*, a connectionless and unreliable protocol, lacks many features present in TCP, such as reliable data transmission, flow control, and error handling.
Hence, TCP is particularly well-suited for applications like multi-threaded dictionary servers that demand reliable data transmission, sequential preservation, and state management.

3.2. Multi-threaded Server

The **worker pool** architecture is used for implementing the multi-threaded dictionary server.
Analysis (why chose):
Worker Pool, the server maintains a pool of threads to handle client requests. When a client connects, the server retrieves an idle worker thread from the pool to handle

requests on that connection. Once processing is complete, the worker thread is returned to the pool.
Thread-per-request, for each incoming client request, the server creates a new thread to handle it. In other words, the server can concurrently handle multiple requests, each running independently in its own thread, which leads to an excessive number of threads and consumes significant system resources when the number of requests is high.
Thread-pre-connection, for each client connection, the server creates a new (same) thread to handle all requests on that connection. This architecture is suitable when there are few connections and each connection's request processing time is relatively long.
Considering that the dictionary may have many users simultaneously performing tasks such as adding, deleting, and searching, I chose the **worker pool** architecture as the framework due to its efficient resource management and scalability. Neither of the latter two is suitable for situations where there may be many connections.

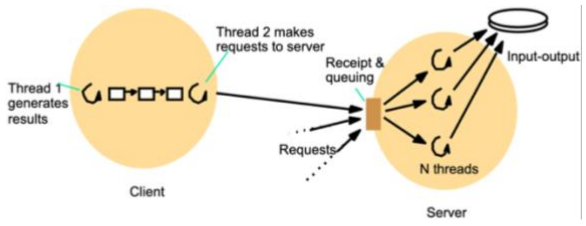


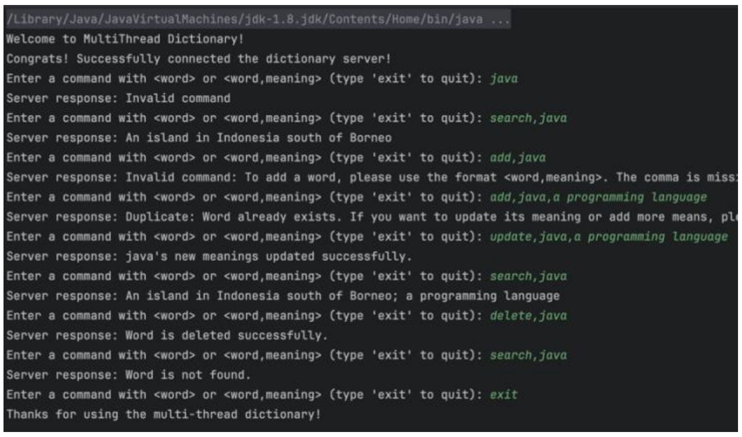
Figure 1 Worker Pool Architecture

4. Class Design & Interaction Design

There are three packages that contain six classes and one configuration class to implement the multi-threaded dictionary.

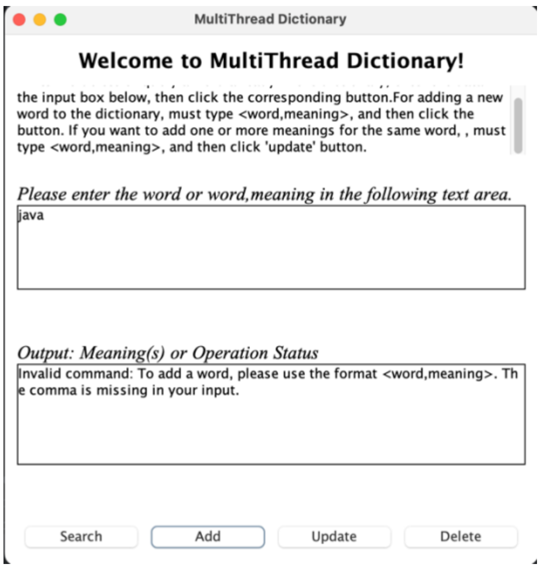
4.1. Client Package

For Client.java: The \`Client\` class contains a constructor used to connect with server; it also contains the \`sendMessage()\` method. This code defines a method to send messages to a socket, allowing users to input <commands,word> or <command,word,meaning>. It continuously prompts for input until "exit"' is entered. Responses from the server are printed, and any socket errors are logged.
For ClientGUI.java: Client GUI is designed and developed by the **JFormDesigner** from IDEA. The \`ClientGUI\` class contains a constructor, where contains three methods, \`**initComponents()**\` for initializing and setting up Client GUI, \`**connect()**\` for establishing the connection with **server(localhost, port 8088)**, \`**closeSocket()**\` for closing socket on Client GUI window close.



```
/Library/Java/JavaVirtualMachines/jdk-1.8.jdk/Contents/Home/bin/java ...
Welcome to MultiThread Dictionary!
Congrats! Successfully connected the dictionary server!
Enter a command with <word> or <word,meaning> (type 'exit' to quit): java
Server response: Invalid command
Enter a command with <word> or <word,meaning> (type 'exit' to quit): search,java
Server response: An island in Indonesia south of Borneo
Enter a command with <word> or <word,meaning> (type 'exit' to quit): add,java
Server response: Invalid command: To add a word, please use the format <word,meaning>. The comma is miss
Enter a command with <word> or <word,meaning> (type 'exit' to quit): add,java,a programming language
Server response: Duplicate: Word already exists. If you want to update its meaning or add more means, pl
Enter a command with <word> or <word,meaning> (type 'exit' to quit): update,java,a programming language
Server response: java's new meanings updated successfully.
Enter a command with <word> or <word,meaning> (type 'exit' to quit): search,java
Server response: An island in Indonesia south of Borneo; a programming language
Enter a command with <word> or <word,meaning> (type 'exit' to quit): delete,java
Server response: Word is deleted successfully.
Enter a command with <word> or <word,meaning> (type 'exit' to quit): search,java
Server response: Word is not found.
Enter a command with <word> or <word,meaning> (type 'exit' to quit): exit
Thanks for using the multi-thread dictionary!
```

Figure 2 Client Response from Console


MultiThread Dictionary

Welcome to MultiThread Dictionary!

the input box below, then click the corresponding button.For adding a new word to the dictionary, must type <word,meaning>, and then click the button. If you want to add one or more meanings for the same word, , must type <word,meaning>, and then click 'update' button.

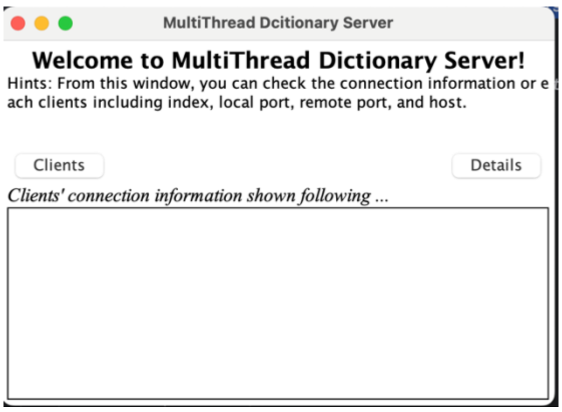
Please enter the word or word,meaning in the following text area.

java

Output: Meaning(s) or Operation Status

Invalid command: To add a word, please use the format <word,meaning>. Th
e comma is missing in your input.

Figure 4 Client GUI Design


MultiThread Dcitionary Server

Welcome to MultiThread Dictionary Server!

Hints: From this window, you can check the connection information or e
ach clients including index, local port, remote port, and host.

Clients' connection information shown following ...

Figure 3 Server GUI Design

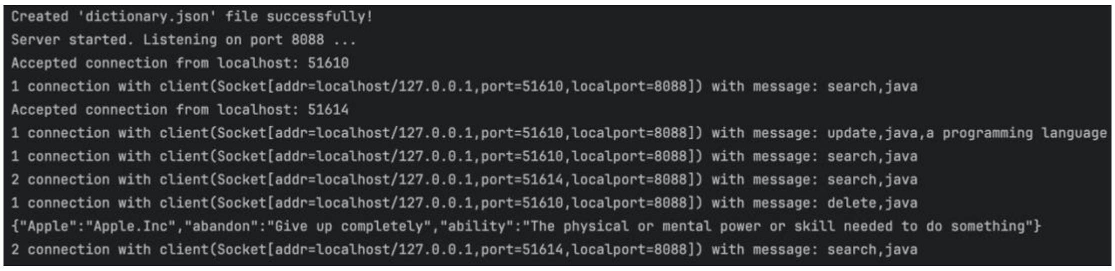

```
Created 'dictionary.json' file successfully!
Server started. Listening on port 8088 ...
Accepted connection from localhost: 51610
1 connection with client(Socket[addr=localhost/127.0.0.1,port=51610,localport=8088]) with message: search,java
Accepted connection from localhost: 51614
1 connection with client(Socket[addr=localhost/127.0.0.1,port=51610,localport=8088]) with message: update,java,a programming language
1 connection with client(Socket[addr=localhost/127.0.0.1,port=51610,localport=8088]) with message: search,java
2 connection with client(Socket[addr=localhost/127.0.0.1,port=51614,localport=8088]) with message: search,java
1 connection with client(Socket[addr=localhost/127.0.0.1,port=51610,localport=8088]) with message: delete,java
{"Apple":"Apple.Inc","abandon":"Give up completely","ability":"The physical or mental power or skill needed to do something"}
2 connection with client(Socket[addr=localhost/127.0.0.1,port=51614,localport=8088]) with message: search,java
```

Figure 5 Server from Console (2 Clients)

4.2. Server Package

***For Server.java*:** The `Server` class contains a constructor, where contains a method, `**start()**` for starting server. It also utilizes the `**Executors**` and `**ExecutorService**` to implement the worker pool architecture for Multi-Threaded server; it calls the `MultiThread` class.

***For ServerGUI.java*:** Server GUI is designed and developed by the **JFormDesigner** from IDEA. The `ServerGUI` class contains a constructor, where contains three methods, `**initComponents()**` for initializing and setting up Server GUI, `**serverStart()**` for initiating server. Additionally, it is same as Server.java, which implements the **worker pool** for the multi-thread. When clicked the

***For Dictionary.java*:** This class contains a constructor, which contains the method `creatDictionaryJSON()` to create the `**dictionary.json**` file. I choose `.json` file due to its flexibility, simplicity, and easy of writing. JSON files use a simple **key-value structure** to represent data, and support n**ested objects and arrays**, making it easy to modify their internal data structures as needed. When encountering one word with multi meanings, I can directly modify the nested **JSONArray**.

4.3. Thread Package

***For MultiThread.java*:** This class also contains a constructor. `MultiThread.java` mainly implements `Runnable` interface and overrides the run() method to create four custom methods (including search, add, update, and delete) to form multi-threaded dictionary. (Details in the Chapter 5.1) There is also the load() method to pass `dictionary.json` to the JSON Object for the next operation.

4.4. Config.class

This class contains several constant fields used to configure parameters for the server (address, port, and the size of the thread pool) and application ('dictionary.json' file path).

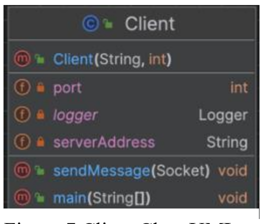


Figure 7 Client Class UML

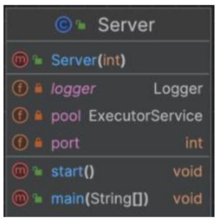


Figure 11 Server Class UML

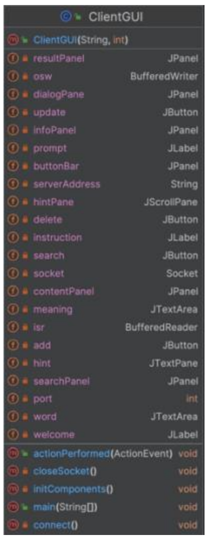


Figure 10 ClientGUI Class

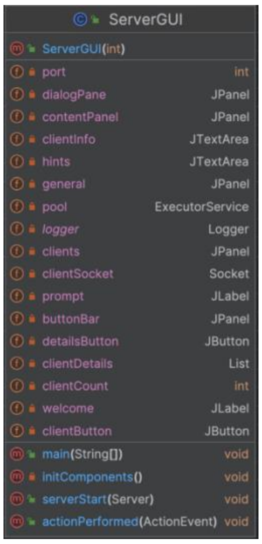


Figure 9 ServerGUI Class UML

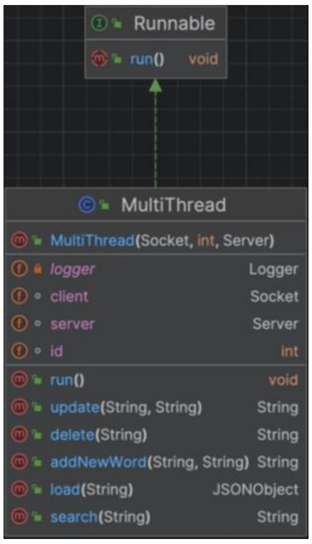


Figure 8 MultiThread Class UML

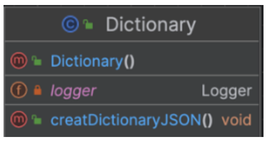


Figure 6 Dictionary Class UML

5. Implementation Details

5.1. Functionality

As mentioned in the last chapter 4.3, the multi-threaded dictionary meets four functions on the client-side (their first step is to load the dictionary), and two functions on the server-side.

For **search()**: If successful in loading, it then attempts to retrieve the object associated with the input word. If the associated object is found, the method checks if this object is an instance of **JSONArray**. If so, it indicates that the word has multiple meanings or definitions. The method iterates through this array, concatenating **all meanings** into a string separated by "**;** ", and returns this string. If the associated object is not a JSONArray, it means the word has only one meaning, so it directly converts this meaning into a string and returns it. If the given word is not found in the dictionary, the method returns "**Word is not found.**"

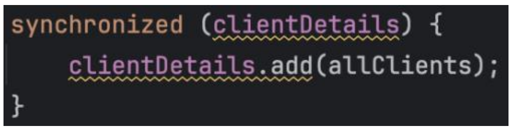
For **addNewWord()**, it is similar to all typical add operations, with the only difference being that it prompts for incorrect commands (but lacks a **comma**). However, when a word appears repeatedly, it prompts for executing **update** operation.

For **update()**, If successful, it checks if the dictionary contains the specified word. If the word exists, it retrieves the corresponding meanings. If the meanings are stored as a JSONArray, it directly appends them; otherwise, it **creates** a new JSONArray and **adds** the existing meaning to it. Next, it checks if the new meaning is already present in the list of meanings. If not, it adds the new meaning to the list, updates the dictionary with the modified list of meanings, and writes the updated dictionary back to the file. Finally, it returns a success message confirming the update. If the word is not found in the dictionary, it returns a message indicating that the word is not found. If there's a **duplicate meaning**, it returns a message indicating the duplicate.

For **delete()**, it is similar to all typical add operations.

5.2. Thread Safety

The shared resources directly affect the thread synchronization safety. For a multi-threaded dictionary, the `dictionary.json` file can be accessed, as well as modified by multiple threads (clients) simultaneously. I added `**synchronized**` to all fields that might endanger thread safety, **except search() method**, which ensures that only one thread can access or modify the value of that field at any given time, thereby preventing race conditions and data inconsistency issues caused by multiple threads modifying data concurrently.



```
synchronized (clientDetails) {
    clientDetails.add(allClients);
}
```

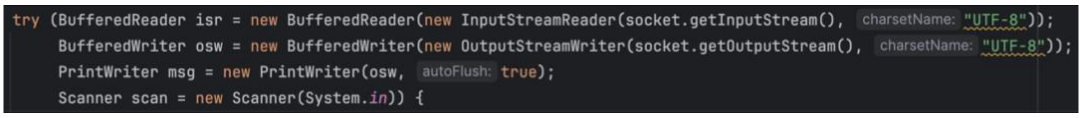
Code 1 Example of `synchronized` for clients information

6. Creativity

6.1. Resource Management

I replaced the traditional `try/catch/finally .close()` statement with `**try-with-resources**` among the client, server, and multi-thread sides. `**try-with-resources**` ensures automatic resource management, guaranteeing that resources are properly closed regardless of whether exceptions occur while preserving original exceptions for better debugging. These features significantly reduce the risk of resource leaks and improve program reliability. What's more, it reduces code redundancy and improves maintainability.

As shown in the following codes, Use `**try-with-resources**` for managing Socket, Stream, and Scanner to ensure they are closed properly.

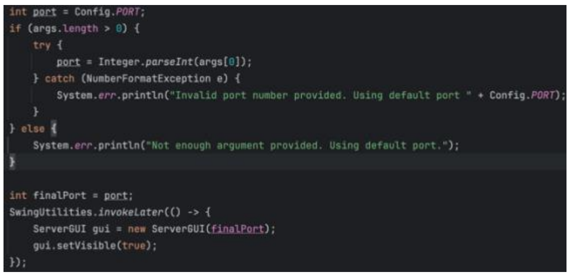


```
try (BufferedReader isr = new BufferedReader(new InputStreamReader(socket.getInputStream(),  charsetName: "UTF-8"));
     BufferedWriter osw = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream(),  charsetName: "UTF-8"));
     PrintWriter msg = new PrintWriter(osw,  autoFlush: true);
     Scanner scan = new Scanner(System.in)) {
```

Code 2 Exampe of `try-with-resources`

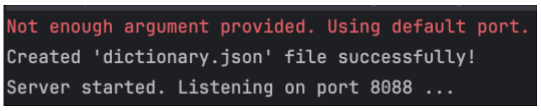
6.2. Custom Port and Server Address

Allow clients to **customize** the **port number** and **server address** to avoid port conflicts. In addition, when no address or port number is provided via the command line, it defaults to binding to **port 8088** and **localhost**.



```
int port = Config.PORT;
if (args.length > 0) {
    try {
        port = Integer.parseInt(args[0]);
    } catch (NumberFormatException e) {
        System.err.println("Invalid port number provided. Using default port " + Config.PORT);
    }
} else {
    System.err.println("Not enough argument provided. Using default port.");
}

int finalPort = port;
SwingUtilities.invokeLater(() -> {
    ServerGUI gui = new ServerGUI(finalPort);
    gui.setVisible(true);
});
```



```
Not enough argument provided. Using default port.
Created 'dictionary.json' file successfully!
Server started. Listening on port 8088 ...
```

Code 3 Custom Port & Server Address

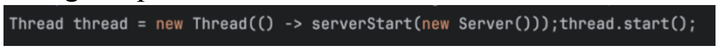
6.3. Thread Safety

Not only do I use `synchronized` to ensure thread synchronization safety, but I also avoid blocking event dispatch threads (EDT) during communication operations.

The multi-threaded dictionary is based on the Swing GUI, and the process of starting server is a time-consuming operation. Additionally, `synchronized` may cause blocking since it suspends threads while waiting for the lock to be released. In other words, it is

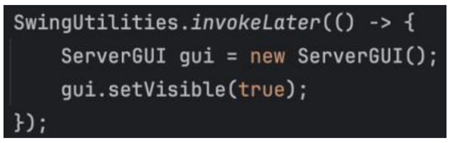
crucial to ensure that the operation of starting server does not execute on the EDT.
I choose two methods to solve these problems to guarantee thread safety:

a. Using a separate `**thread**` to start the server in the backend。



```
Thread thread = new Thread(() -> serverStart(new Server()));thread.start();
```

b. Use `**SwingUtilities.invokeLater**` to ensure that all GUI update operations are executed on the EDT.



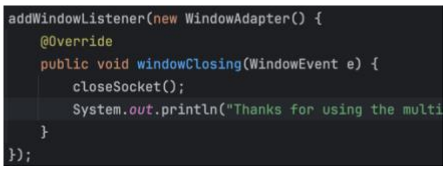
```
SwingUtilities.invokeLater(() -> {
    ServerGUI gui = new ServerGUI();
    gui.setVisible(true);
});
```

Code 4 Example of `SwingUtilities.invokeLater()`

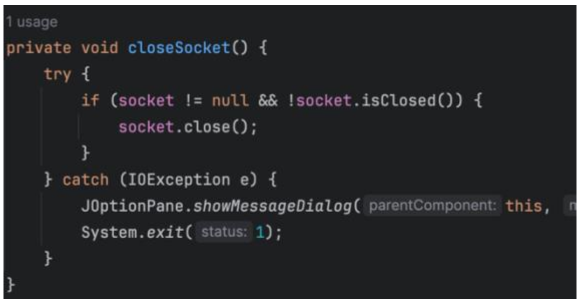
6.4. GUI exit

I overrode the `**windowClosing()**` method so that guarantees the client sockets are closed properly. In other words, it can avoid the connection exception due to lingering data in the input stream while the socket is already disconnected.

Note: This approach is functionally equivalent to `try-with-resources`. Because of variable declaration issues in `clientGUI.java`, this method was chosen to properly manage resources.



```
addWindowListener(new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent e) {
        closeSocket();
        System.out.println("Thanks for using the multi
    }
});
```



```
1 usage
private void closeSocket() {
    try {
        if (socket != null && !socket.isClosed()) {
            socket.close();
        }
    } catch (IOException e) {
        JOptionPane.showMessageDialog( parentComponent: this,
        System.exit( status: 1);
    }
}
```

Code 5 GUI exit for Closing Socket

7. Improvement

It still has the potential to growing into something fancier than I did in the past three weeks.

7.1. Reconnection Mechanism

Provide clients with some strategies when encountering errors connecting to the server, such as **reconnection mechanism**.

7.2. Custom FilePath

Allow clients to import their own 'dictionary.json' file rather than the default one.

7.3. Friendly GUI

Generate another dialog for `add` operation so that users can simplify the input by separating words and meaning into two independent `textArea` without requiring the insertion of additional delimiters.

8. Conclusion

All in all, I gained basic knowledge about socket and multi-thread programming. I will continue to implement the improvements for this project during my spare time. After ending the semester, I will upload the final version to my GitHub. This project will be my "calling card" for my future study and career.

Reference

[1] R. Buyya, S. Selvi, X. Chu, "Object Oriented Programming with Java: Essentials and Applications", McGraw Hill, Delhi, India, 2009.