Technical Report: Java Dictionary Client-Server Application

Hanjie Liu

1667156

hanjie.liu@student.unimelb.edu.au

Comp90015 Distrubution System

1. Introduction

1.1 Project Overview

This is a Java-based single-server-multi-client system, which supports dictionary operations. Users can use the client to send instructions to the server, such as: add, delete, add, query, and update. The server will return the appropriate action based on the client's requirements. All clients share a single server and dictionary. The client provides a simple GUI, all communications are based on TCP.

1.2 Development Environment

Stack used: Java, Maven, Java Swing (GUI), TCP Sockets, Fasterxml.jackson

IDE： IntelliJ
JDK: Java 19
Project run:

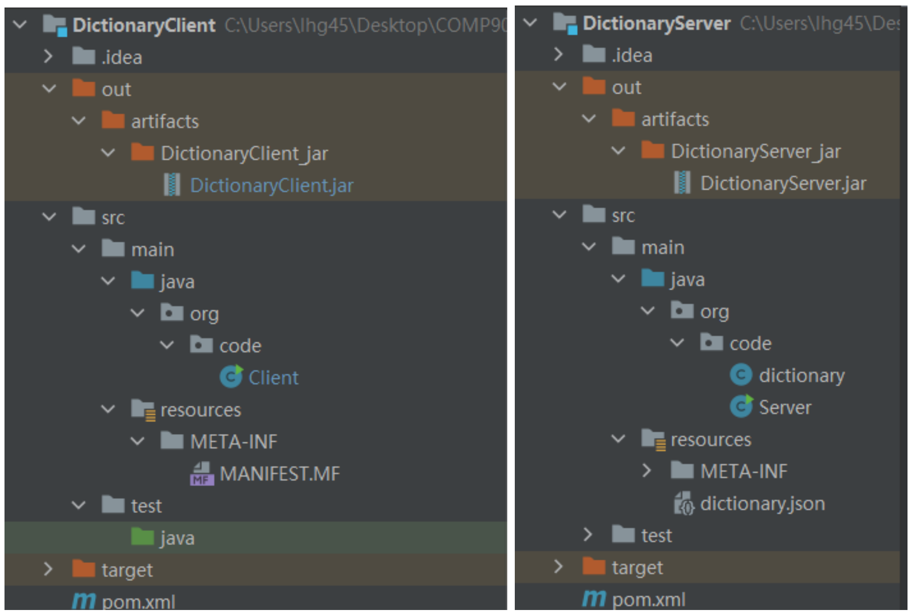
1. java –jar DictionaryServer.jar <port> <dictionary-file>
2. java –jar DictionaryClient.jar <server-address> <server-port>

2. System Architecture Overview

2.1 Codebase Structure

Our code is made up of four files, These are Client.java, Server.java, Dictionary.java, and dictionary.json (called automatically when there are no external resources), where Server.java, Dictionary.java, json were responsible for the server functionality in the same jar, and Clinet.java is responsible for the client functionality as well as the GUI.

We use Maven structure to manage our program. The dependencies are located in pom.xml



```
DictionaryClient C:\Users\lhg45\Desktop\COMP9
  .idea
  out
    artifacts
      DictionaryClient_jar
        DictionaryClient.jar
  src
    main
      java
        org
          code
            Client
      resources
        META-INF
          MANIFEST.MF
    test
      java
  target
  pom.xml
```

```
DictionaryServer C:\Users\lhg45\De
  .idea
  out
    artifacts
      DictionaryServer_jar
        DictionaryServer.jar
  src
    main
      java
        org
          code
            dictionary
            Server
      resources
        META-INF
        dictionary.json
    test
  target
  pom.xml
```

2.2 Class Design

public class dictionary: This class defines the dictionary class, which contains all of the program's dictionary-related operations and data.

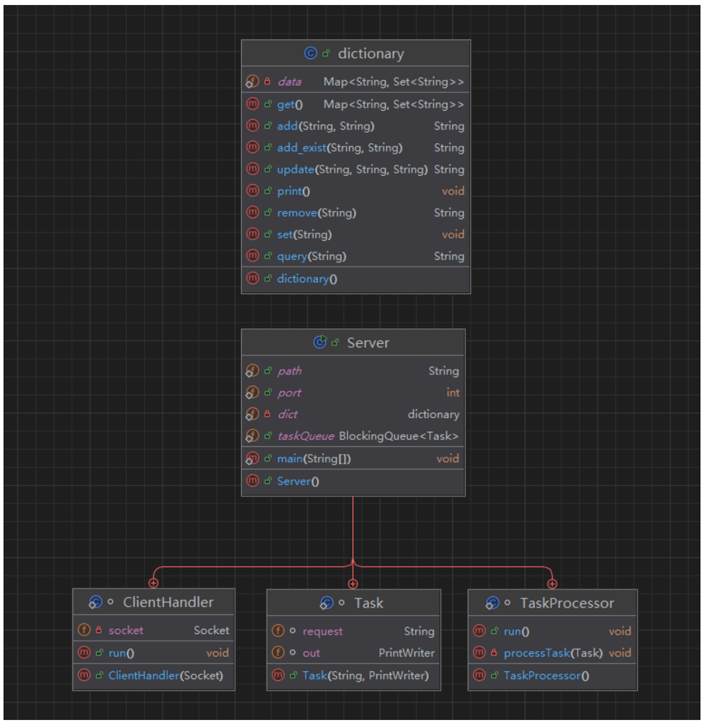
public class Server: This class defines a server class, which will be responsible for defining the Task class, starting a service port, managing sockets, communicating TCP with clients, multithreading tasks, scheduling resources, and error handling.

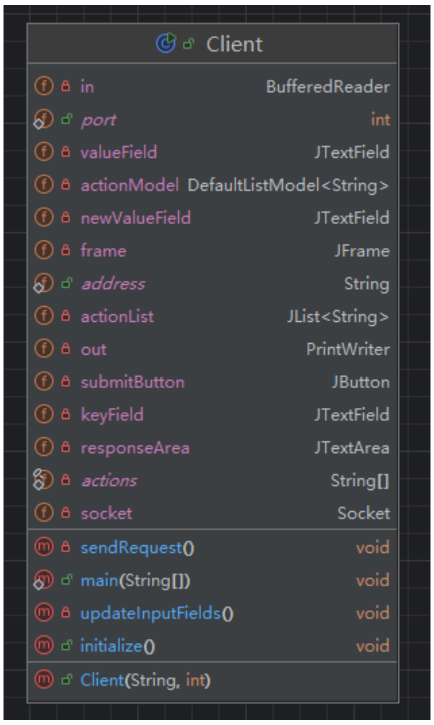
public class Task (inside Server): Define a Task class that contains the client request and output to the client. This task will be the body we use to manage the queued tasks. The Task result will be streaming into the corresponding socket or client.

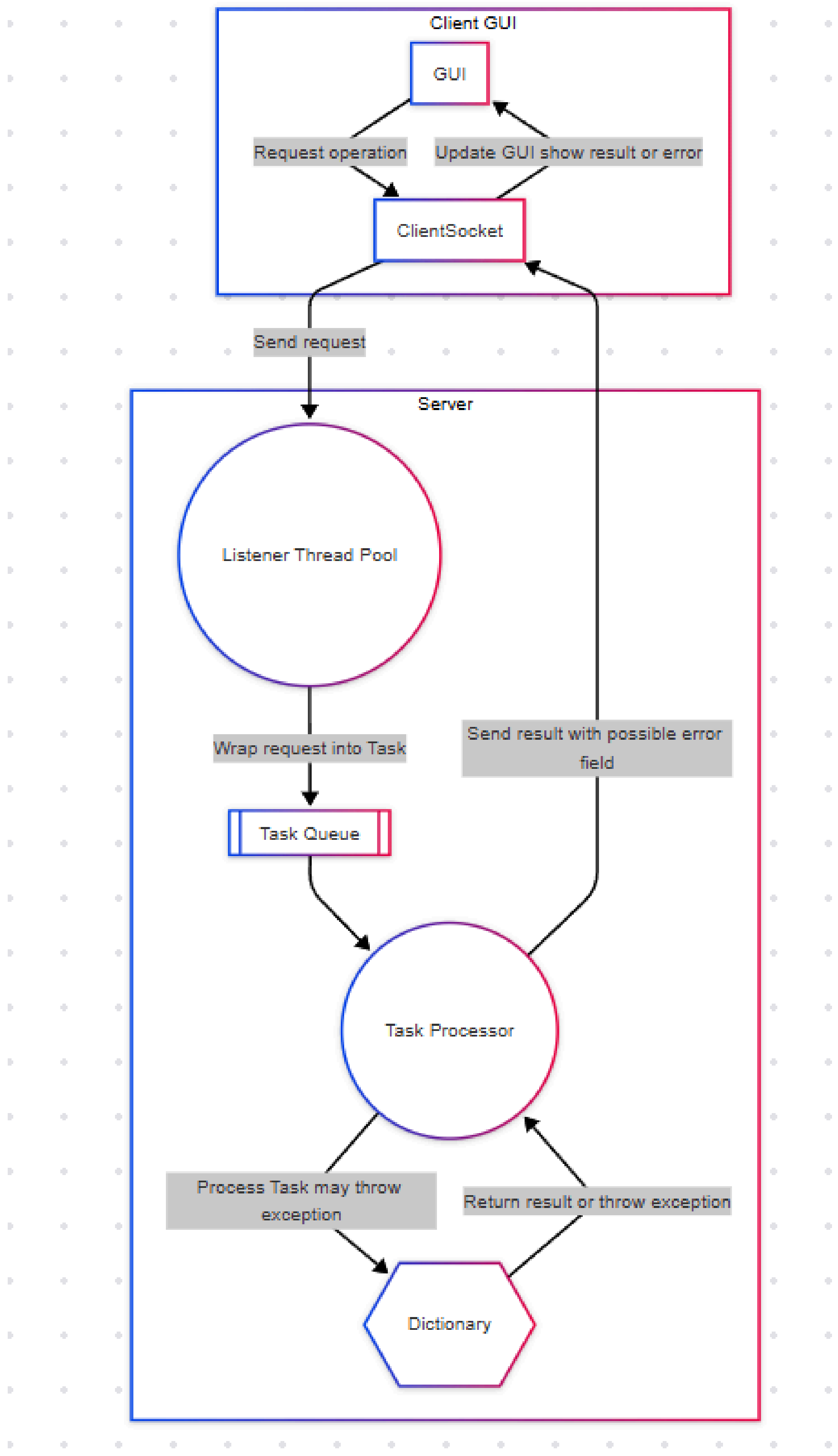
static class TaskProvessor implements Runnable (inside Server): The part that executes tasks by implementing runnable, and the tasks are managed by BlockingQueue, so that our tasks can be queued in a single thread according to the time of the client request, avoiding race conditions. It will do the operations with dictionary, all functions inside are strictly following instructions, including error handling.

static class ClientHandler implements Runnable(inside Server): This class is used to manage connections between clients and multiple servers. We will use multiple threads to manage each connected server, read the information in the socket in real time, put it into the taskqueue, and give it to the taskprocessor for execution. When the client disconnects, close the socket and notify the server. Again, this part will do error handling

public class client: This class contains the entire client, including the client's socket connection, task request management, error handling, GUI, and event listening and occurrence. The validity of the customer input information is tested, and the customer input is also regulated to meet the interpretation of the information on the server.






Client GUI
GUI
Request operation
Update GUI show result or error
ClientSocket
Send request
Server
Listener Thread Pool
Wrap request into Task
Send result with possible error field
Task Queue
Task Processor
Process Task may throw exception
Return result or throw exception
Dictionary

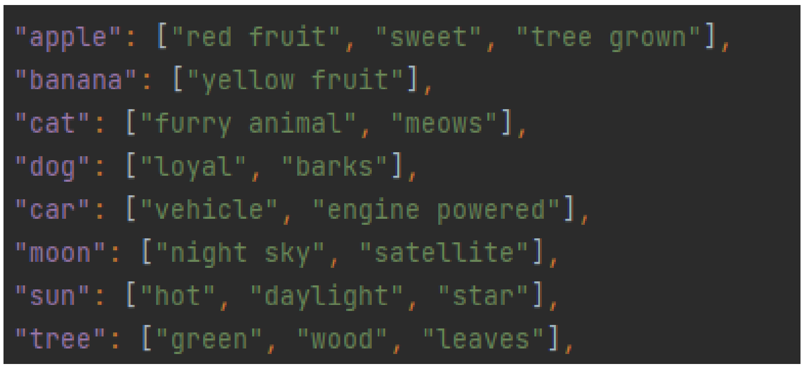
3. Network Communication

3.1 TCP Setup

The default server is *localhost* and port 8888, but you can customize the port at the launch command line. We used ***ServerSocket*** class, ***BufferedReader*** class and ***PrintWriter*** class.

3.2 Communication Protocal

Our propose dictionary format is json format., for example:
"{apple":["red", "fruit"], "banana":["yellow fruit", "sweet"]}



```
"apple": ["red fruit", "sweet", "tree grown"],
"banana": ["yellow fruit"],
"cat": ["furry animal", "meows"],
"dog": ["loyal", "barks"],
"car": ["vehicle", "engine powered"],
"moon": ["night sky", "satellite"],
"sun": ["hot", "daylight", "star"],
"tree": ["green", "wood", "leaves"],
```

Our request and decode format is following:

Query the meaning(s) of a given word: "query:key"
Add a new word: "add:key:value"
Remove an existing word: "remove:key"
Adding additional meaning to an existing word: "add_exist:key:value"
Updating existing meaning of an existing word by new one: "update:key:value:new_value"

<Our decode strategy is splitting value from ":", take them into taskprocess.>
All information should be in String type, the key and value will be given by client input.

4. Threading Model

Server used Thread to manage all connections with clients, each socket will be handled by a single thread. All requests from different client will be send into one task pool, which will execute task one by one. The taskprocessor will also be handled by a thread, which guarantees that no race conditions will happen.
<public static BlockingQueue<Task> taskQueue = new LinkedBlockingQueue<>();>

For the request storm problem, our program was built on task base, which means we used single-thread to take the task from the queue one by one, and return the result to each socket one by one. Our client will also block the submit button once the request sent without respond.

5. Dictionary Functionalities

5.1 Core Operations:

Search: take the key and return all value from list.

If there is no target key, return “notfound”.

Add: add the key and value list, return the result of Add.

If the key was already in dictionary, return “key exist, please use add_exist”.

Delete: take the key and return the result of Delete.

If there is no target key, return “notfound”.

Update: take the key, value, and new_value, and return the result of Update.

If there is no target key, return “notfound”

If there is no target value, return “no target value”.

Add_exist: take the key, value, and return the result of Add_exist.

If there is no target key, return “notfound”.

5.2 Internal Logic:

The dictionary will read json file into a Map<String, Set<String>> dataStructure when it initialized.

We handle case uniformly, so all strings being processed are lowercase to avoid race repetition.

For invalid inputs, such as not enough parameters, the server throws an exception and sends it back to the client.

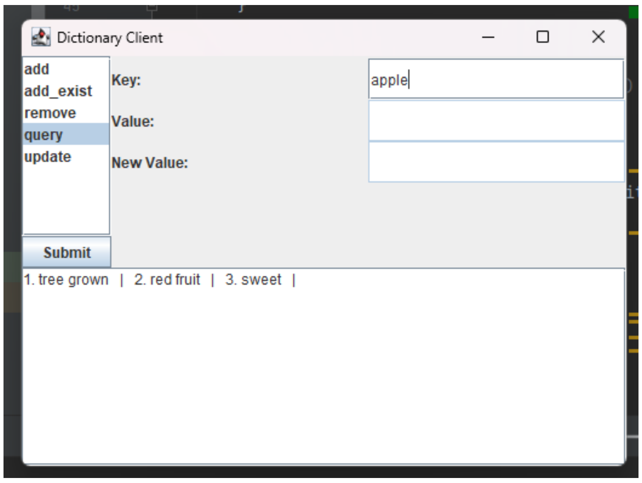
For regular messages, if the message is a query, the value is returned. Any other command will return success, failure, and error messages

For disposing duplicate value, our dataset was based on Set(), which means if all values have been transformed to lowercase, no same means will associate with the key.

6. Client-Side Design

6.1 GUI Overview:

Users can choose one operation on the left sections, and type the string in the right textfields. The result will be shown in the lower textfields.



Key: apple

1. tree grown | 2. red fruit | 3. sweet |

6.2 GUI Event Handling:

Once users input all information inside the textfields, the user can click the submit button and receive the result from the Server. The textfields will be only unlocked since the user chose the corresponding action. Before users receive the result, the submit button will be locked. If there is no connection with the Server, the result textfield will show no connection with the Server.

7. Error handling

7.1 Invalid command in console:

It will throw an exception and print the error information in the console.

7.2 Network communication:

Both disconnections of Server and Client will throw an exception and printan error message in console.

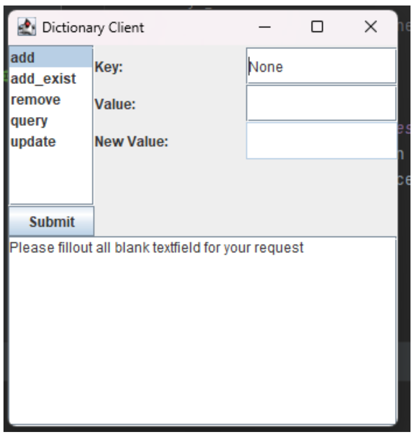
7.2 Dictionary I/O exception:

If there is no external source, it will load the default dictionary inside .jar. If there is an IO error, it will exit the program and throw an exception, print an error message in the console.

7.3 Invalid input from client:

The Invalide request will be handled in the Server without access dictionary. The Server will throw an exception and return an error message back to the client.

Here are some examples:

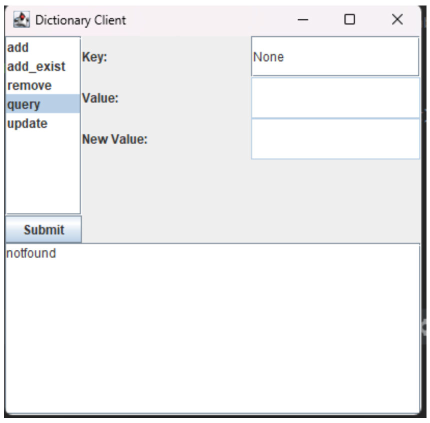


Key: None

Value:

New Value:

Please fillout all blank textfield for your request

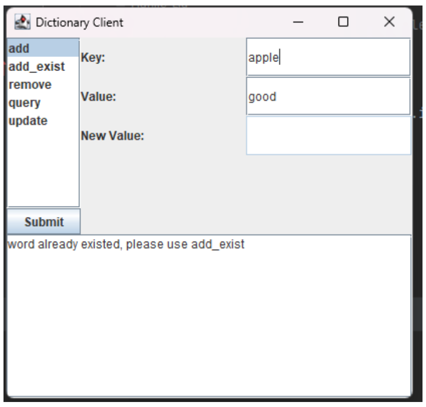


Key: None

Value:

New Value:

notfound

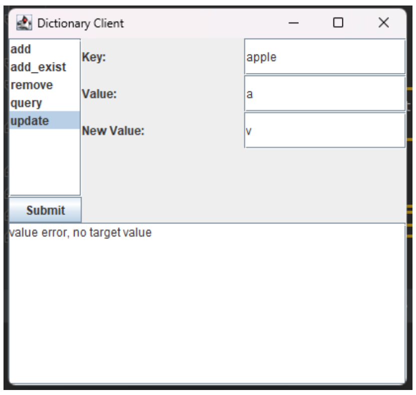


Key: apple

Value: good

New Value:

word already existed, please use add_exist

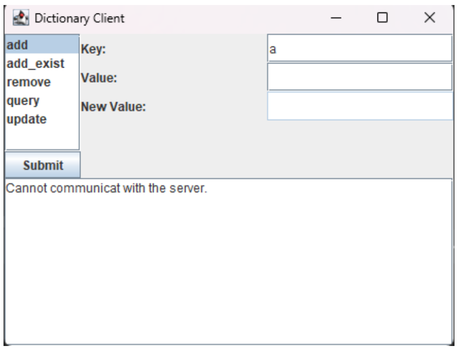


Key: apple

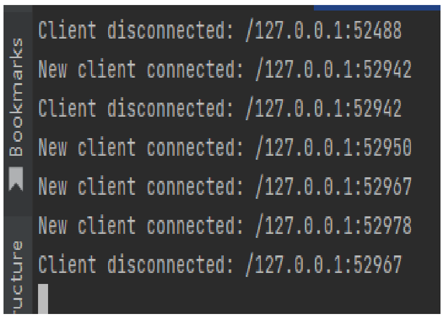
Value: a

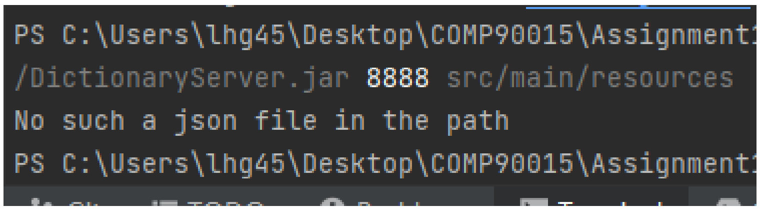
New Value: v

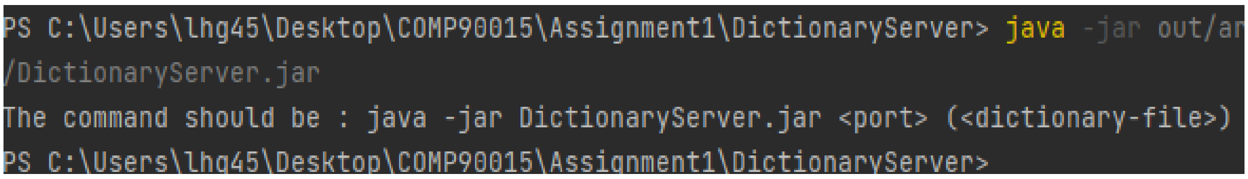
value error, no target value



Cannot communicat with the server.


```
Client disconnected: /127.0.0.1:52488
New client connected: /127.0.0.1:52942
Client disconnected: /127.0.0.1:52942
New client connected: /127.0.0.1:52950
New client connected: /127.0.0.1:52967
New client connected: /127.0.0.1:52978
Client disconnected: /127.0.0.1:52967
```


```
PS C:\Users\lhg45\Desktop\COMP90015\Assignment1
/DictionaryServer.jar 8888 src/main/resources
No such a json file in the path
PS C:\Users\lhg45\Desktop\COMP90015\Assignment1
```


```
PS C:\Users\lhg45\Desktop\COMP90015\Assignment1\DictionaryServer> java -jar out/ar
/DictionaryServer.jar
The command should be : java -jar DictionaryServer.jar <port> (<dictionary-file>)
PS C:\Users\lhg45\Desktop\COMP90015\Assignment1\DictionaryServer>
```

8. Critical Analysis

Client:

Advantage:

The code structure is clear and the module division is reasonable. For example, the initialize() method is responsible for initializing the GUI, while the sendRequest() method focuses on communicating with the server. Using the updateInputFields() method to dynamically enable or disable input fields based on the actions selected by the user is intuitive and effective, avoiding the need for the user to enter invalid data. Some basic handling of common errors is done in the code. For example, a NumberFormatException is caught to handle a port number resolution error and an appropriate error message is displayed to the user.

Drawback:

In the sendRequest() method, exception handling is very basic. If an error occurs while communicating with the server, the response area displays a generic error message ("Cannot communicate with the server."), which doesn't help the user understand the root cause of the problem. The sendRequest() method performs network communication in the event dispatch thread (EDT). If the server takes a long time to respond, this can lead to a sluggish GUI interface and a poor user experience. While WindowAdapter ensures that resources are released when the window is closed (such as closing sockets and input/output streams), these resources may not be released correctly if the application is forced to terminate (such as through the task manager).

Server:

Advantage:

Multi-threading is used to handle the client connection (ClientHandler) and task queue (TaskProcessor), which allows multiple clients to interact with the server at the same time and improves the concurrency performance. The producer-consumer pattern is implemented by using BlockingQueue to manage task queues. This design decouples the client request processing from the network communication, and improves the maintainability and scalability of the code. Dictionary operations such as add, remove, query, and so on are encapsulated in the Dictionary class, which is designed to conform to the single responsibility principle.

Drawback:

taskQueue uses an unbounded queue (LinkedBlockingQueue is unbounded by default). In high concurrency, the task queue may grow indefinitely, causing memory to run out. The TaskProcessor has a single thread that processes tasks in the task queue. If a task takes too long to process, it may cause subsequent tasks to pile up, affecting the overall performance. If there are a large number of unprocessed tasks in the task queue, the client request may be delayed or even timeout. But there is no mechanism in the code to monitor the length of the task queue or the processing latency.