COMP90015 ASSIGNMENT 1 — Online Dictionary

Ruipu Cui Student ID: 1298198

Problem Context

Design and implement a multi-threaded Java dictionary server that uses low-level sockets and threads to support concurrent clients performing dictionary operations. The System should support multiple functions, including query words, add words, remove words, add meanings and update meanings. Clients and the server must communicate through a well-defined message protocol, and a graphical user interface (GUI) is required for client interaction. Robust error handling is expected throughout the system to manage input, network, and I/O exceptions effectively.

System Components

DictionaryClientUI

This class is the GUI of client and its built in **Java Swing.** It allows users to interact with the dictionary **Server** through a clean and intuitive interface. The interface has 2 pages.
The first page is Connection Page. It let the users to enter port number and start connection with **Server**. The second page is the function page. This page present multiple dictionary functions such as Query, Add, Remove, Add Meaning and Update Meaning. Each function has its own input panel and a respond area to display the message from server.
This class interact with **Client** Object to send requests to the back end server.

Client

The **client** class acts like a communication layer between **DictionaryClientUI** and dictionary **Server**. It defines how to establish a TCP socket connection to the server by using provided port number. It also manages the input and output streams for sending and receiving messages by using **MessageProtocol** Class. In summary, this class handles all the network communication logic and allows **DictionaryClientUI** to focus on user interaction only.

MassageProtocol

The **MassageProtocol** class defines the communication format and message structure used between both client to server and server to client. It provides variety of static utility methods to construct and parse JSON messages using the **GSON** library. This class ensure the consistency of communication message. The key responsibilities include creating request messages for client side, generate responses message for server side. It also supports extract partial message of the whole json string. Overall, this class acts as a central protocol handler to make both server and client speak the same language.

ClientHandler

This class is the key component of multi-threading. It is responsible for handing communication with an individual client connected via a socket. This class implements **Runnable**. The work flow of the ClientHandler has 3 steps. First, reading a json message sent by the client. Second, it passing the request to the **DictionaryHandler**. **DictionaryHandler** directly interacts with

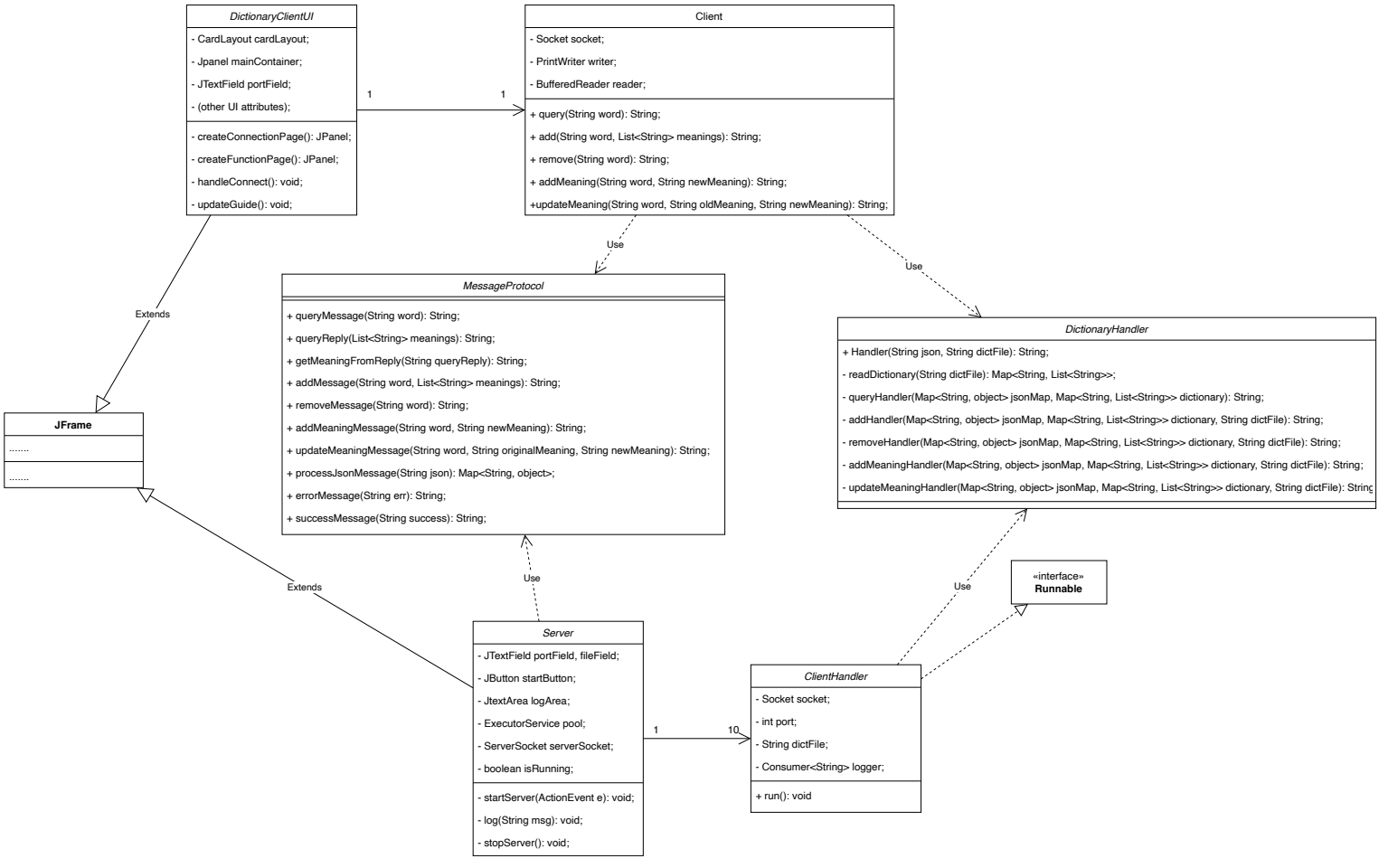
dictionary file and process the request. Finally, the ClientHandler sends back a response to the client (by using **MassageProtocol** for sure). This class ensure **Serve** can deal with multiple clients in the same time.

Server

This class is the GUI of the server side. The GUI is also implemented by using **Java Swing.** Beside the GUI, it also contains a thread pool with many threads in it ready to make connection to the client. The **Server** has 4 key features. 1. Multi-threaded architecture. It uses an **ExecutorServer** to handle concurrency. 2. User-friendly UI. 3. Dynamic Log. All import events for example client connection, receiving request will be displayed in a Log panel in the UI. 4. Connection Handing. When a client connects, it call a **ClientHandler** in the thread pool to manage the communication.

DictionaryHandler

The **DictionaryHandler** class is the core logic of how to read and update the dictionary file directly. The dictionary file is in json format. The class also use **GSON** library to read and write dictionary. It uses the **synchronised** keyword to prevent concurrent access conflicts during dictionary read/write operations.



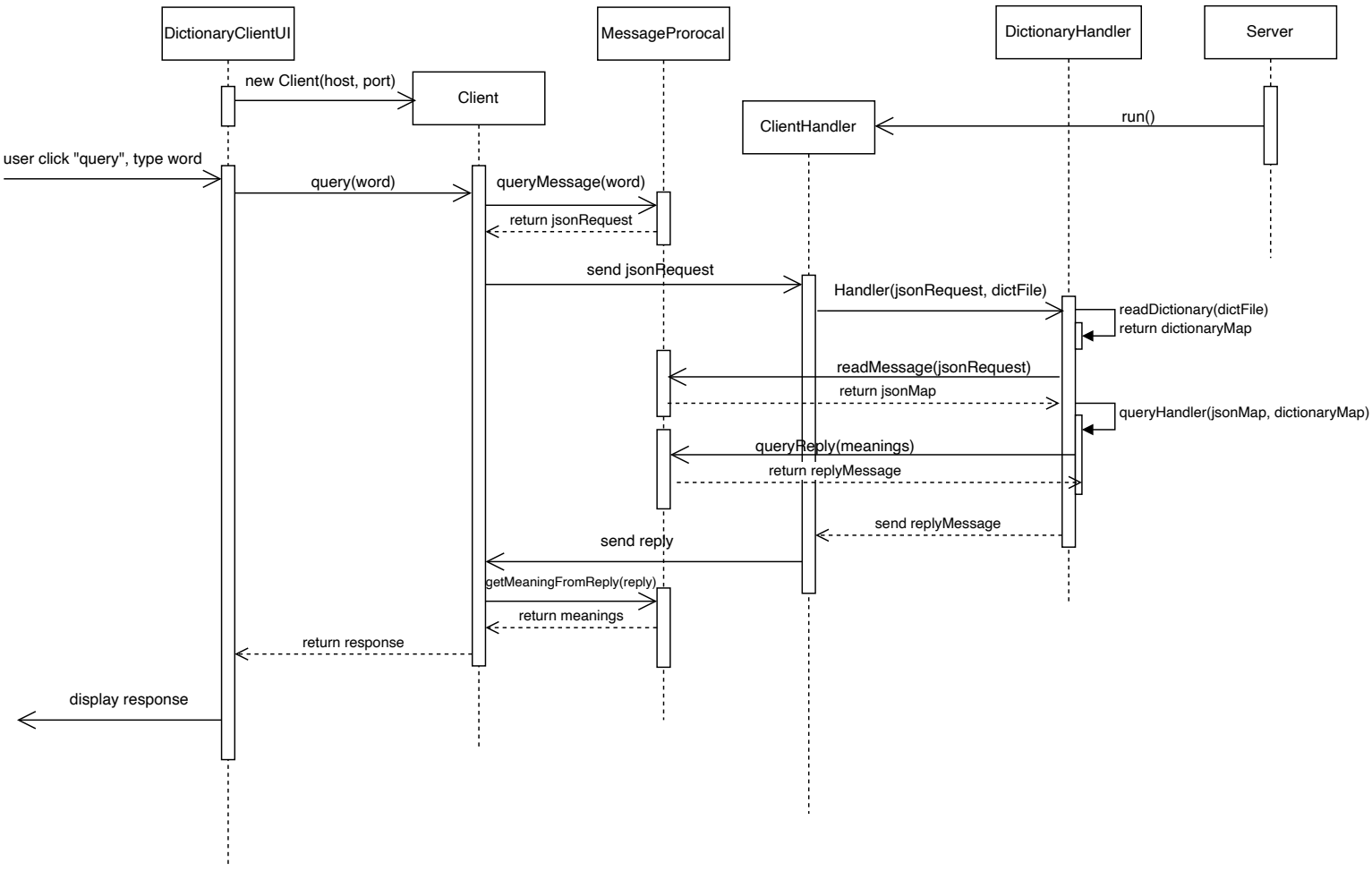
System Design Diagram

System Interactions

As the Server start to run, it creates multiples threads(**ClientHandler**) and add those threads to the worker pool. For every client connects to the server, the server assigns one **ClientHandler** from the worker pool to take the job.

After user choose the type of request and type the request content, **DictionaryClientUI** calls corresponding method in **Client**. **Client** class manages the input and output streams for sending and receiving messages. **Client** then calls helper method from the **MessageProtocol** to generate the request message in **json** format and send it to the other side of **TCP** connection.

The **ClientHandler** receives the request message and then sends the message and the dictionary data file name to the **Handler** method in **DictionaryHandler** class. The **Handler** method first process the dictionary data file and transfers it into **Map** data structure. Then it calls **readMessage** method in **MessageProtocol** class to extract informations from the json format message sent by the client side. After that it calls corresponding helper method depending by the request type to process the request. In the end it calls the method in the **MessageProtocol** again to generate reply message in json format and send the reply message to the client side.



System Interaction Diagram (using “query” function as example)

Critical Analysis

Socket Programming and Multi-Thread Architecture

I choose TCP as my network protocol. TCP as a connect-oriented protocol with flow and congestion control, it can ensure data integrity. It also reorders out-of-order packets before passing them to the application so that application gets data exactly as sent.

The multi-thread server is implemented by worker pool architecture. For thread per request architecture, it creates a thread for every single request causes high overhead under heavy load and risks exhausting system threads or memory. For thread per connection architecture, it creates a thread per client connection which doesn't scale well for thousands of clients.

To prevent conflict between threads, the dictionary file is accessed and updated within synchronised methods in DictionaryHandler.

System Design

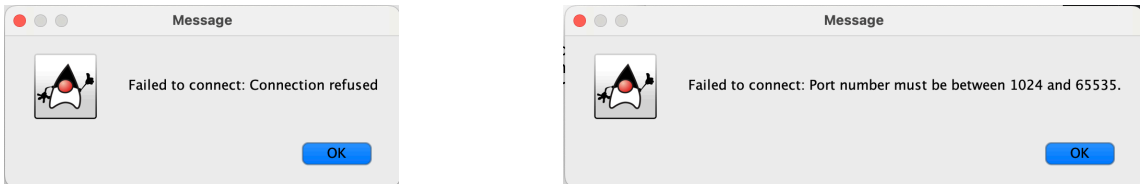
The system designed in its well-structured modular architecture. Each component in the system has a clearly defined and isolated responsibility. DictionaryClientUI manages the user interface and user interactions. Client handles network communication. MessageProtocol standardises request and response formatting. Server and ClientHandler manage concurrent connections and threading. DictionaryHandler encapsulates dictionary-related logic and file I/O.

It not only ensures clean and readable code but also lays a strong foundation for long-term scalability and maintainability. New features can be added by extending the relevant module, without requiring invasive changes to others. For instance, adding a new function like getSynonyms() only involve changes in MessageProtocol, Client, and DictionaryHandler. The server side component doesn't need to change.

Failure Model

I categorised all errors into 2 types, system error and functional error.

System errors are the errors that may cause system crash if they are not handled in a correct way. These include I/O errors like illegal input from users and network communication error like lost connection or fail to connect etc.. When this type of error detected, an error window will immediately pop up on the UI. In the pop up window, it will guide user the next move.



Examples System Error Message

Functional errors are the errors that is already considered during function implementation. For example user may query a word that does not included in the dictionary. I deal with this type of error by directly display error message in the server response panel in the client UI.

Example Functional Error Message

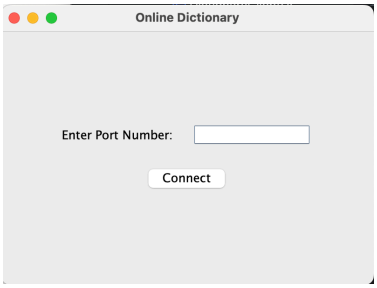
Error Type	Error Description	Error Message / Server Response
System Error	User choose a port number out of range	“Failed to connect: Port number must be between 1024 and 65535."
	Port number is empty when user try to connect with server	"Failed to connect: Port number cannot be empty."
	Invalid port number	"Invalid port number. Please enter a valid number."
	Fail to connect	“Failed to connect: Connection Refuse”
	User didn’t enter the word before make request to server	"Please enter a word."
	User didnt enter meaning when try to make “add” request	"Please enter at least one meaning."
	User didn’t enter the original meaning when make “update meaning” request	"Original meaning cannot be empty.”
	User didn’t enter the new meaning when make “update meaning” request	New meaning cannot be empty."
	Client lost connection to the server	“The connection to the server was lost”
Functional Error	Query/Add/Add meaning/Update/ Remove a word that does not exit in the dictionary	{"ERROR":"the word does not exit"}
	Add a word that already exit in the dictionary	{"ERROR":"the word already exits"}
	Update a meaning that does not exit	{"ERROR":"the meaning does not exits"}
	Add a meaning that already exit	{"ERROR":"the meaning already exits"}

Error Table

Client UI Design

The Online Dictionary Client UI features a clean, user-friendly interface designed in two stages:

Connection Screen:
Users are first prompted to enter a port number to establish a connection with the dictionary server. The layout is minimal, focusing solely on the connection input to ensure simplicity and clarity.



Connection Screen

Main Function Panel:
Once connected, users can access five dictionary operations—Query, Add,

Remove, Add Meaning, and Update Meaning via a sidebar menu.
The main panel contains:

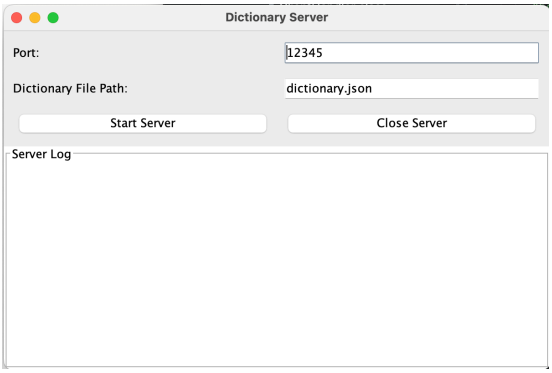
- A word input field and Start button for triggering actions.
- A dynamic guide label that updates based on the selected function.
- Input areas for meanings when needed.
- A scrollable response area to display server feedback clearly.

Main Function Panel

Creativity Elements

Server UI

I built a server UI, making it easy for users to interact with the dictionary server application. It includes the following components:

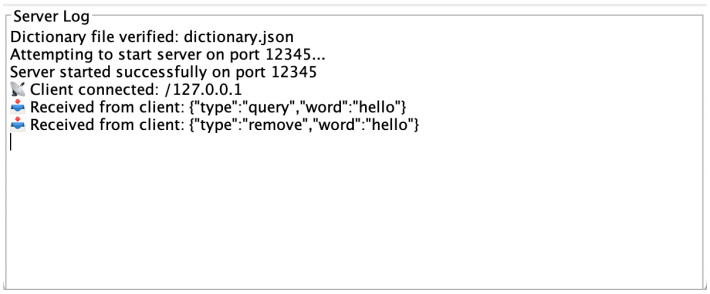


Server UI

Port Input Field: Allows the user to specify the port number for the server.
Dictionary File Path Field: Lets the user input or confirm the path to the dictionary file.
Start Server Button: Start the server with given port number and dictionary file path.
Close Server Button: Stops the server make connections to any new client.
Server Log Text Area: A large log area at the bottom where real-time server activity and messages are displayed for user monitoring.

Server Log

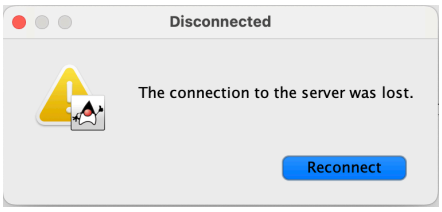
The server log displays real-time updates of server activity, including file verification, startup status, client connections, and received requests. It helps monitor server behaviour, debug issues, and track user interactions for better transparency and maintenance.



Server log

Reconnection Function

If client lost connection during running, user could press “Reconnect” button to redirect to the client connection screen page.



Reconnection Message