COMP90015 – Assignment 1

Xiuge Chen 961392

1. Problem Description

In this assignment, students are required to build a simple dictionary that consists of a server and a client application. Clients application should provide a GUI for user to send request to server, such as search for the meanings of words, add new words and remove existing words. Server should maintain a dictionary and take corresponding actions whenever receives request. Server and clients are communicating reliably through socket, our own choice of internet protocol and message exchange protocol, in order to support multiple clients send and receive messages at the same time, server should also handles different request simultaneously through multithreading. Also, user could configure the port number that server is binding to, the dictionary file server is using, and the server ip/port address that a client is connecting to. Moreover, server and clients together should handle some expected errors well without crashing, such as invalid(empty) input from user, network communication failure and IO errors. Besides basic requirements above, students have some degrees of freedom to extend the functionalities of server/client application to build a better application. For example, create GUI for server to help manage it and monitor its interaction with clients, use worker pool architecture to utilize fixed number of threads and avoid creating too many threads, thoroughly error handling and so on.

2. Excellence and Creativity Elements

2.1 Excellence Elements

In this assignment, I did several following things to meet the requirements of excellence elements:

1. Thoroughly error handling and notification to users, please refer to section 6.
2. To better maintain and trace problems. I also create log files for both client and server applications, which dynamically records all the information and errors occurred. For more information please refer to section 5.2.
3. Extra fields in dictionary file and exchanged message to help maintenance and analysis, such as created/last-modified time, indicator of validity. For more information please refer to section 4.2 and 5.1.
4. Detailed analysis design choices of my system and their pros and cons, please refer to section 3, 4 and 5.

2.2 Creativity Elements

In this assignment, I extended several following functionalities to meet the requirement of creativity elements:

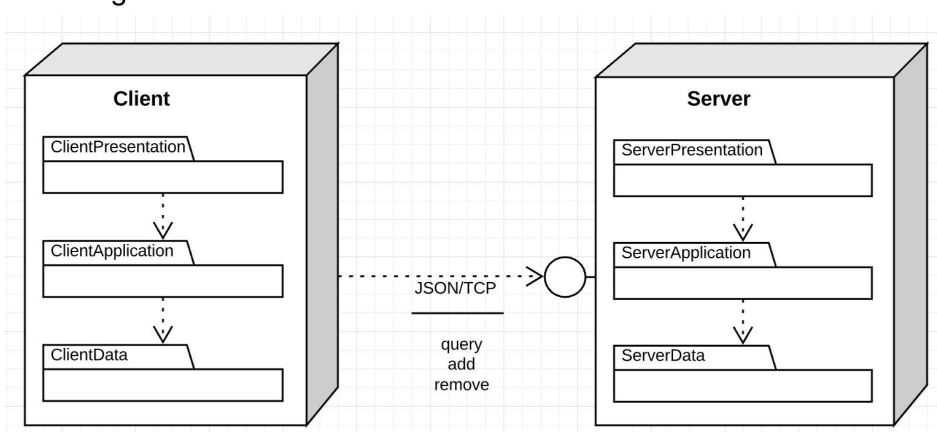
1. Adopt worker pool architecture to handle multithreading in server, for detailed design, implementation and reasons behind this choice, please refer to section 3.2

2. A GUI for server, so that server user can manually close the server rather than terminate it in the terminal, also server user can monitor the client details of all ongoing connection, as well as all historical actions that all clients have taken. Please refer to section 3.1 for detailed explanation.
3. Few additional functions on both GUIs, such as a *Help* menu bar that redirects users to our subject website, background and icon images to make application more pretty, and File menu bar to allow user exit program. For more information please refer to section 3.3.

3. System Design and Components

3.1 Software Architectural

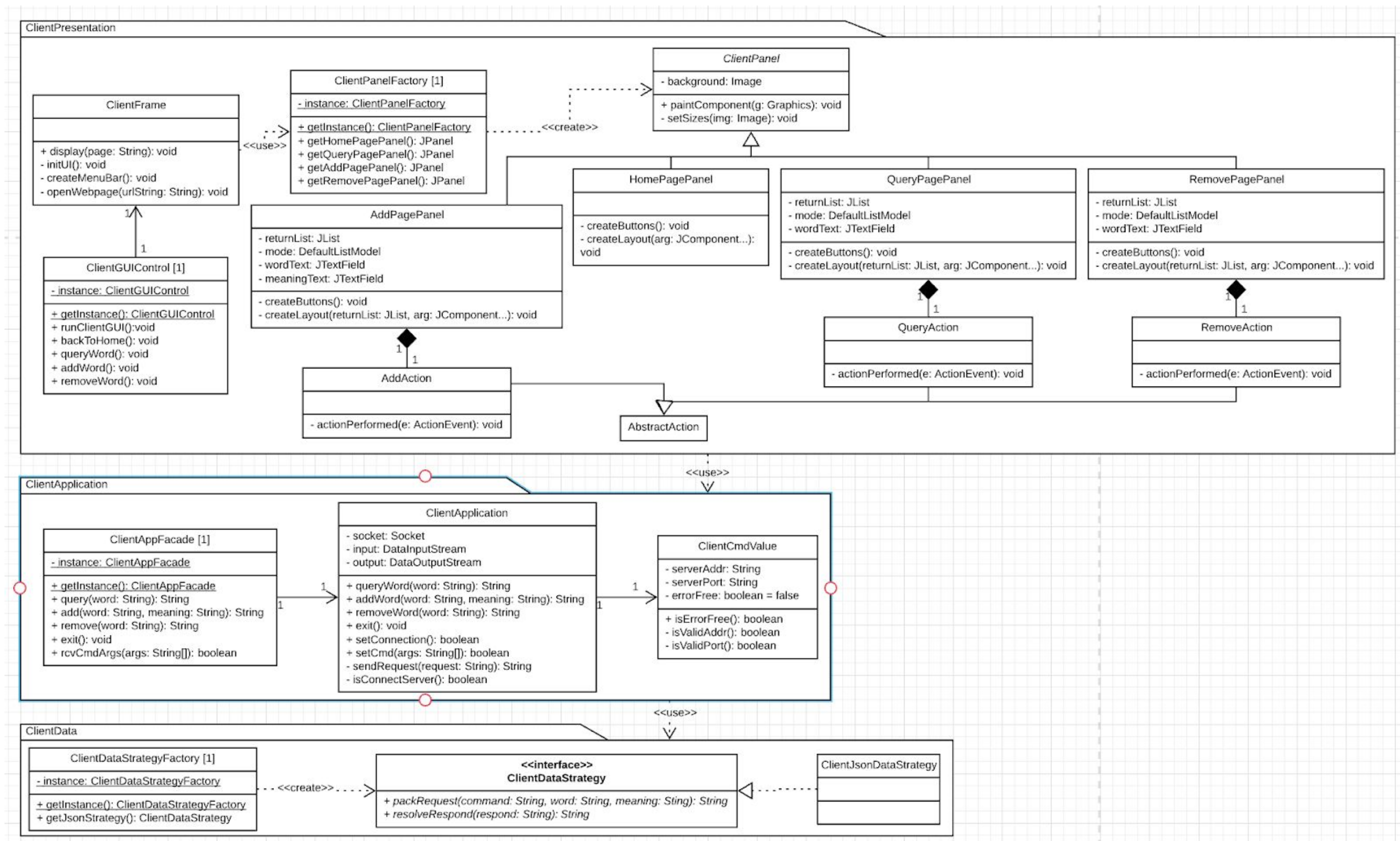
As shown in G-3-1, I adopted client-server architecture, and selected 3-tier architecture as the internal layering model for them. I chose client-server architecture because this system only need one server to serve all clients, the boundary between server and client is pretty clear and they are only responsible to perform one of the server/client functionality. Therefore as mentioned in spec, client-server is the better architecture in this project. For simplicity reasons, I put all data manipulation inside of server as data layer rather than implementing another data server to execute it, which might cause performance problems when the dictionary data becomes fairly large. But my system are also carefully designed that wraps every layer and provides interface to interact with, so that it could be easily extended to add data server or other alterations like change data format.



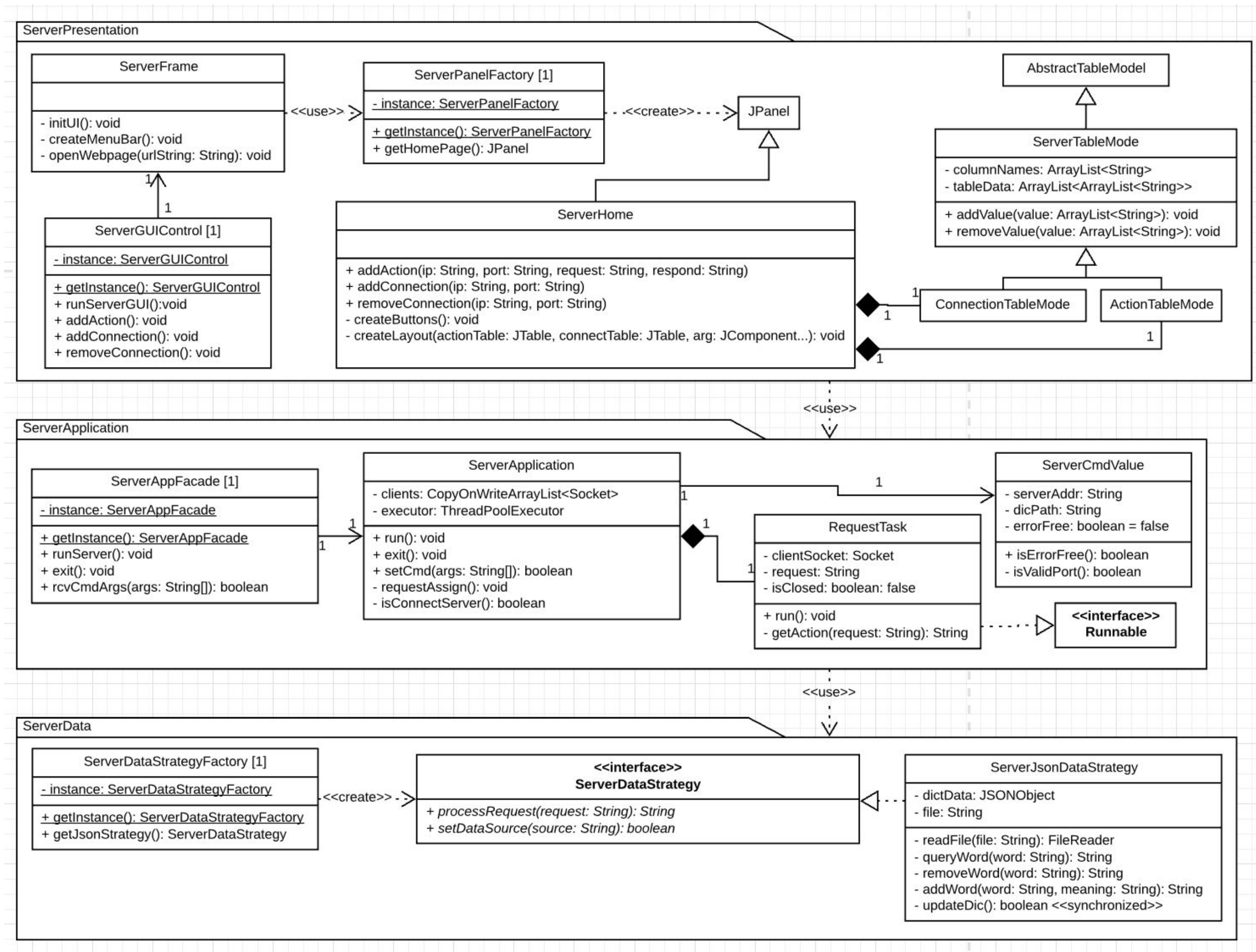
G-3-1 Software Architectural

I selected typical 3-tier architecture as the internal architecture of server and client, as shown in G-3-2 and G-3-3. 3-tier architecture consists of presentation layer, application layer and data layer, this architecture helps separate classes into different layer based on their responsibilities, so that each layer could scale horizontally easily and the system is more manageable. Presentation layer controls all interaction with users, like provide GUI, display certain page and content. Application layer is responsible for the main application logic such as checking command line arguments, establishing connections, managing connections(only server), sending and receiving messages. Data layer contains different strategy classes used for handling different data, such as JSON message parsing and building.

The class design principle I followed is responsibility-driven, that is making each class only responsible for a small amount and highly-related actions, such that the whole design is in both high cohesion and low coupling. Moreover, I also applied few GoF patterns and *Polymorphism* to support better extensibility and maintainability. For example, I implemented a *Singleton Facade*/*Factory* for each layer, classes outside(especially in the upper layer) could easily use the functionalities provided in this layer without worrying about their actual implement detail. Meanwhile, *Strategy* pattern in data layer let developers be able to add other data format strategies easily, or even extend to data server.



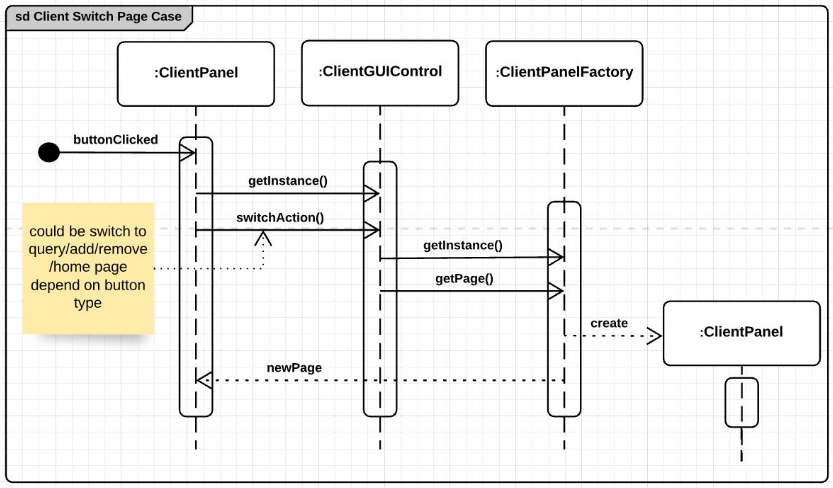
G-3-2 Client Class Diagrams



G-3-3 Server Class Diagrams

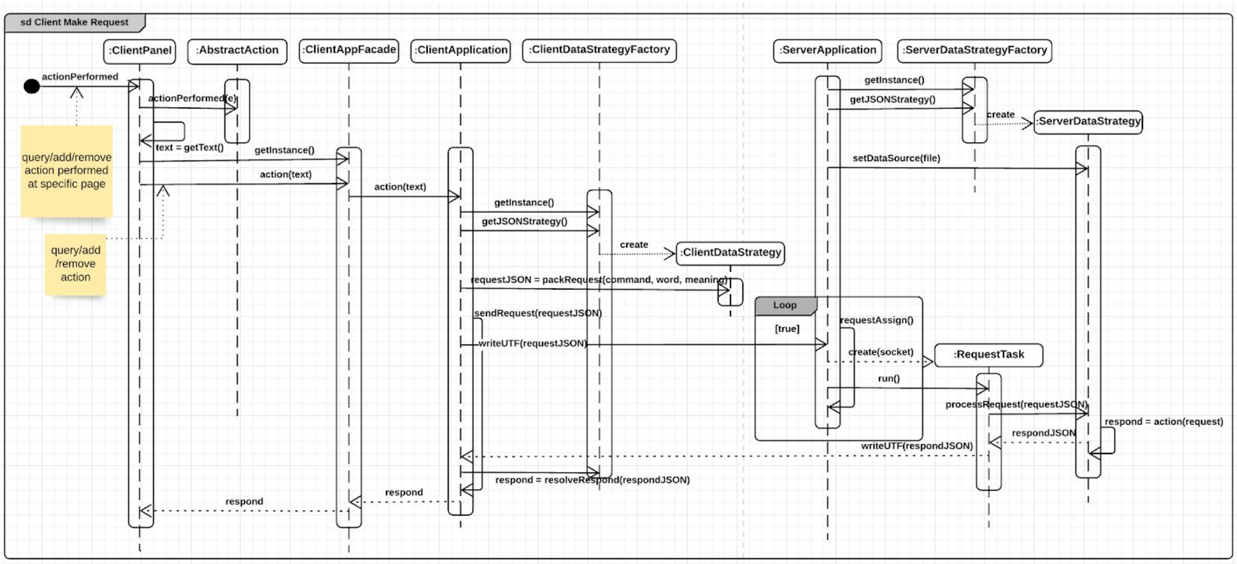
Interaction between user and this system could be concluded as two main scenario, User switch pages and user post request to server. In G-3-4 and G-3-5, I only showed the main generic steps of these two user cases, some error handling, user exit scenario, GUI notifications were left due to page limitation. The interaction flows also follows the requirement of strict 3-tier architecture, that is each class only uses services provided in either the same layer or the layer directly below it. Thus combining with my class design choices, makes this system more modular and scalable, also easier to implement and understand since implementation the lower layer is hidden by interface, but it also might introduce some overheads in communication and performance.

As shown in G-3-4, user could switch among 4 pages of client system while using GUI (home, addWord, QueryWord and RemoveWord pages). Once the user clicked related buttons, ClientGUIControl together with ClientPanelFactory will create new instance of required page, then display it to the user.



G-3-4 Communication Diagram - Client switch page

As user making request (query/add/remove) to server, those requests will first being received by action listener *AbstractAction* at Client *Presentation* layer. Then related services provided at Application layer interface *ClientAppFacade* will be called, so that the action will be transmitted to *ClientApplication* which contains the main application logic. Before sending request to server, *ClientApplication* again will use the services from lower data layer, to transform the request into JSON format, then send it to the server. Server application side contains an infinite loop to hear requests from all clients, once it detects a new request, it will create a new *RequestTask* thread to execute this request. Similarly, RequestTask will use services from lower data layer to resolve request, take corresponding actions within the dictionary to get response, and then pack response in JSON format. JSON format response then will be transmitted from Server *Data* layer to Server *Application* layer, then to Client *Application* layer. Finally, *ClientApplication* use same data strategy to resolve response, then transmit the plain text response back to Client *Presentation* layer and display it to user.



G-3-5 Communication Diagram - Client make request

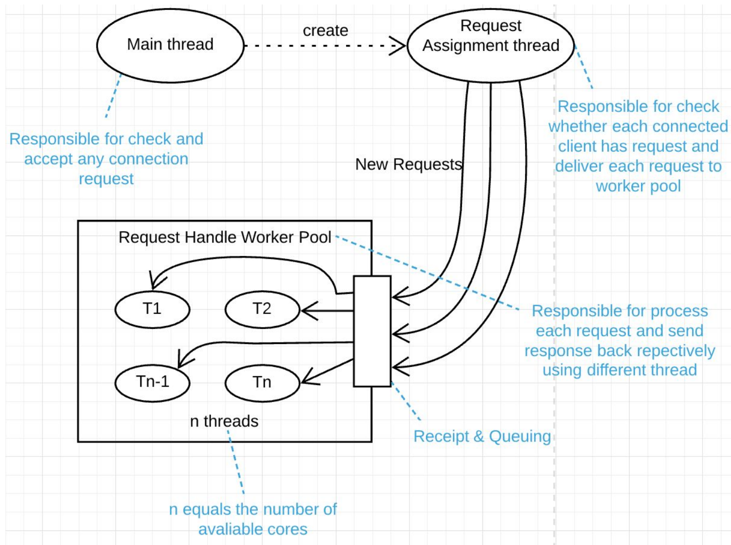
3.2 Server Multithreading Architectural

On server side, I used worker pool architecture (see G-3-6) to handle multithreading more efficiently and under reasonable control.

Firstly, this assignment requires a single server to handle multiple concurrent access and interaction with different clients. Unlike computer program, clients tend to have variant behaviour in terms of the request sending speed, since they won't send request immediately after initializing or receiving responses. Therefore, it is expected that some clients might set up connection and do nothing afterwards. If the server uses thread-per-connection architecture, one thread might be occupied by such client for some time and waste the resources being allocated to it, results in server unavailability to new request. Therefore, using thread-per-request or worker pool architecture could fully utilize server, process each request efficiently and concurrently without make each thread consistently hearing request from only one client.

Moreover, worker pool outperforms thread-per-request due to its ability to control the number of threads and reuse each threads. Since thread-per-request will create one new thread for every new request, if there are too many new requests it will create too many threads that our system could handle, also increase the overhead of our system by adding multiple thread creating cost. Using a worker pool could help reduce these two problems by creating a fixed number of threads at first and recycle it every time after finishing its task. So the server would be more manageable with less overhead.

In total, for the above reasons, I chose worker pool architecture for my server. However, since I used TCP as my internet protocol, if I simply assign each client connection to a single thread and let it responsible for handling all requests from that client, the problem that clients might occupy a thread for a long time might still occur. Therefore, besides the *Main* thread which is responsible for initializing and accepting any connection, I also create a new *Request Assignment* thread from *Main* thread. Once created, *Request Assignment* thread will keep looping through existing connections and check whether there is a request from them (just like polling), if so, *Request Assignment* thread will then read the request and deliver it to *Request Handle Worker Pool*. *Request Handle Worker Pool* contains a fixed number of threads that equals to the number of available cores, requests will be queued and processed whenever one thread is available. Overall, despite the overhead being introduced from looping through connections and the delay of processing request even while some threads are available (especially when server has lots of connections to loop through), this worker pool architecture successfully helps eliminate the possibility of some threads being occupied by some clients for a long time, also prevent from creating too many threads as well as reduce the cost of creating threads.

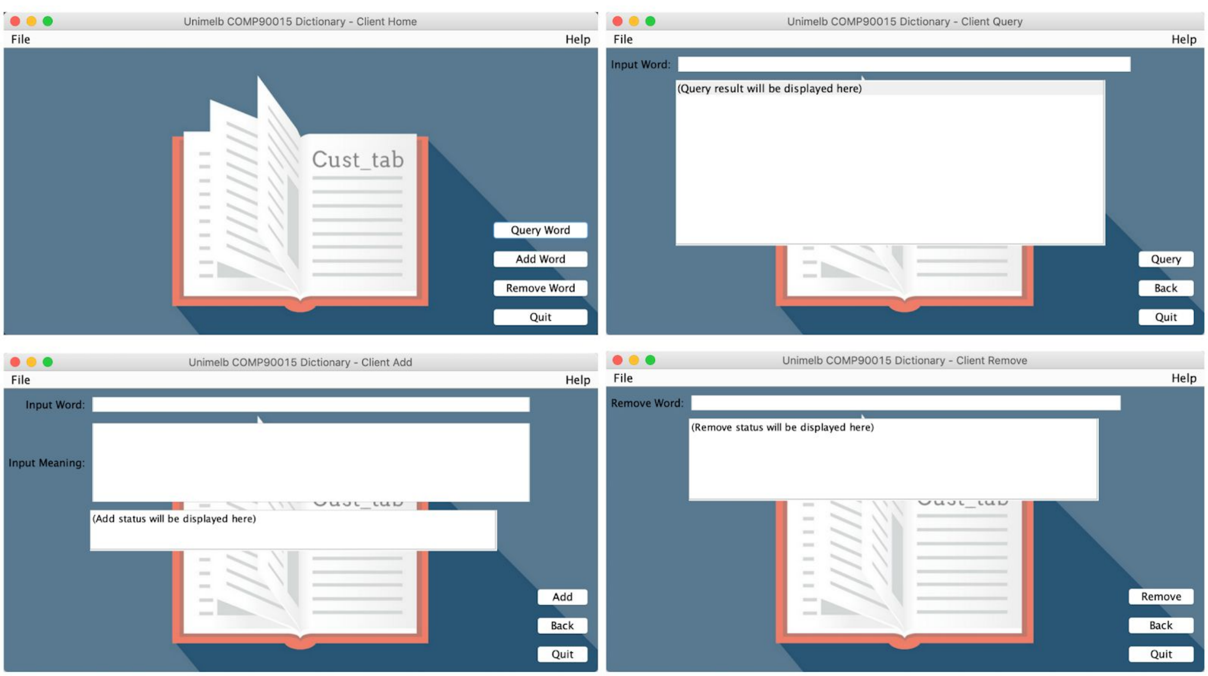


G-3-6 Multithreading architectural

3.3 GUI Design

3.3.1 Client

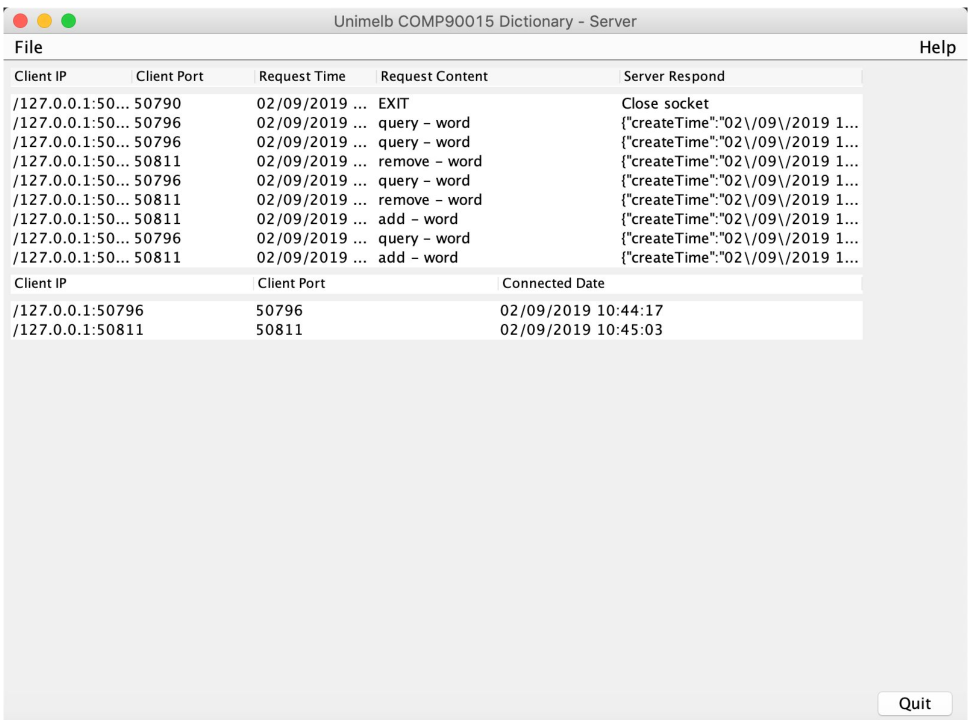
As shown in G-3-7, Client GUI consists of 4 pages, home, query, add and remove page. All four pages contain a background image, a Quit button to exit system, and a menu bar with a *File* section and a *Help* section. *File* section provide a *Quit* item and places for future functions, Help section contains *Subject* item that redirects user to subject website once clicked. *Home* page contains buttons that direct users to other three pages, and other pages contain Back button to go back *Home* page.



G-3-7 Client GUI Design

3.3.2 Server

As shown in G-3-8, server GUI contains only one home page that displays information about all ongoing connections and historical interactions. Server GUI has exactly the same menu bar settings, also a *Quit* button in the right corner. Besides, server GUI contains two display tables that dynamically updated. Once a client connected/disconnected from the server, the lower table will update so that it only display ongoing connections. Whenever server received client request, upper table will be updated with request content and corresponding response, unlike connection table, request table will keep all historical records since the server opened. By using this GUI, server user could not only manually safely quit the program rather than using command line ctrl+c, but also better monitor and analysis server status.



G-3-8 Server GUI Design

4. Interaction and Communication Method

4.1 Internet Protocol

As mentioned in the specification, all communication between server and clients should be reliable, therefore, I decided to use TCP as the internet protocol. Although TCP is connection-oriented and thus will be challenged if I want use thread-per-request or worker pool for multithreading, TCP itself is a reliable protocol so I don't need to add anything above it. UDP of course is easier for request-oriented multithreading architectures, but the work needed to be done to make it reliable is much more than the work to use request-oriented multithreading on TCP connection. Therefore, I decided to use TCP as my internet protocol.

4.2 Message Exchange Protocol

Since server and clients are required to communicate quickly, accurately and efficiently in this assignment, I designed a simple message exchange protocol in JSON data format. JSON is a lightweight format for storing and transporting data, and it is often used when data is sent from server to a webpage. The data being exchanged between the server and client are very structured that suitable for JSON, each request from client should contain the type of request (query/add/remove), related word, sometimes related meaning and other useful information (like date). Also, JSON contains human-readable text, so we could easily modify the content of server dictionary. Lastly, Java provides very useful package *org.json.simple* that I could use to parse and build JSON object easily. So, I adopted JSON as my message exchange protocol. Request JSON includes created time, command, word, meaning (optional), is valid indicator. Respond JSON includes created time, respond, is valid field.

5. Data Storage

5.1 Dictionary File

My dictionary file is stored in JSON format, as the similar reasons in section 4.2 about why I choose JSON as my message exchange protocol. Keep dictionary file and exchange protocol consistent makes the system easier to write and more efficient. Dictionary JSON file should contain all word-meaning pairs, created time and last updated time.

5.2 Log File

Log records is very important for tracking historical information, so that data analyst could use them to analysis questions such as how many unique users visited server so far, also engineers could use them to debug and develop any potential optimization strategies. Therefore, I implemented a detailed logging function on both client and server sides as an extension functionality, log records not only will be output to terminal, but also will be stored into local system files for future use. Log records varies in four levels: *INFO*, *WARN*, *ERROR* and *FATAL*, all logs contain the current time and their level. INFO level logs mainly includes system status and input/output, such as time and content of information exchanged, when clients connected/disconnected to server, how tasks being arranged, when and what update dictionary, and so on. *WARN* level logs notice that there might exist something invalid in system, but the system could still work normally using some backup or recovery strategies. For example, if the dictionary path is invalid, the server could still run using built-in default path, if user input invalid empty word, system will request user to input again. ERROR level logs indicates there exist some errors that interrupt current execution but won't terminate system, like clients send request to server but the server is closed, or parse JSON message failed. *FATAL* level logs record anything fatal to system that causes system exit, such as invalid command line arguments, default dictionary path not reachable.

6. Error Handling

To increase the availability and stability of my system, as well as give informative notification to users and maintainers when errors occur, I implemented lots of error handling and recording function in my system. Besides document all errors occurred, the strategies of handling different errors are consistent with log levels of these errors (please refer to section 5.2). In summary, for *WARN* level errors, I will use some built-in backup/recovery strategies to keep system running. As *ERROR* level errors, the system will terminate only the current operation but not the system itself, and send back error notifications. When encountering *FATAL* errors, I will simply terminate the system. Below I list the major errors my system will handle and the corresponding strategies.
Format: error - level - action/strategy

6.1 Server

1. Invalid command line argument (missing, port in wrong data type, port not in specified range) - FATAL - close all systems, output correct usage to users
2. Parse command line argument failed (wrong format) - FATAL - close all systems, output correct usage to users
3. Dictionary path not reachable or in none json format - WARN - change dictionary file path to built-in default path, try open and read again
4. Default dictionary path not reachable or can't open - FATAL - close all systems
5. Can't find dictionary file under specified path - WARN - try to create file and read it again.
6. Create new dictionary file failed - FATAL - close all systems
7. Parse dictionary JSON data failed - FATAL - close all systems
8. Create listening socket failed - FATAL - close all systems
9. Accept any client connection request failed - FATAL - close all systems
10. Close client socket failed - ERROR - cancel operation, system keep running
11. Read requests from clients failed - ERROR - cancel operation, system keep server running
12. Received empty or invalid request, or unknown command - ERROR - cancel operation, notify users, system keep server running
13. Parse client JSON request failed - ERROR - cancel operation, system keep running
14. Client query/remove word not found - INFO - notify clients
15. Client add words already exist in dictionary - INFO - notify clients
16. Update and write dictionary file failed - ERROR - cancel operation, system keep updated copy and keep running, updated copy will be written to dictionary file at the next update.
17. Send response to clients failed - ERROR - cancel operation, system keep running
18. Create JSON format response failed - ERROR - cancel operation, send notification to users, system keep running
19. Client can not be reached (client closed or internet connection lost) - ERROR - cancel operation, system keep running
20. Read background and icon images failed - WARN - use blank background and icons.

21. Open subject website in *Help* menu failed - ERROR - cancel operation, keep system running.

6.2 Client

1. Invalid command line argument (missing, port in wrong data type, port not in specified range) - FATAL - close all systems, output correct format to users
2. Parse command line argument failed (wrong format) - FATAL - close all systems, output correct usage to users
3. Client input empty word/meaning while querying/adding/removing - WARN - cancel request sending and ask client to input again.
4. Server can not be reached (server closed or internet connection lost) - ERROR - cancel operation, system keep running
5. Read from server failed - ERROR - cancel operation, close socket, system keep running
6. Parse server JSON response failed - ERROR - cancel operation, system keep running
7. Send request to server failed - ERROR - cancel operation, close socket, system keep running
8. Send unknown request (not applicable through GUI but could happen from back-end modification) - INFO - Notify user of unknown request command.
9. Create JSON format request failed - ERROR - cancel operation, send exit command to server, system keep running
10. Close using socket failed - ERROR - cancel operation, system keep running
11. Read background and icon images failed - WARN - use blank background and icons.
12. Open subject website in *Help* menu failed - ERROR - cancel operation, keep system running.