COMP90015 Distributed Systems

Assignment 1 - Report

Yixin SHEN 1336242

Problem Context

Assignment 1 aims to implement a multi-threaded dictionary server based on a client-server architecture. This client-server architecture should have four crucial fundamental functionalities, including searching the meaning(s) of a word, adding a new word, removing an existing word, and updating the meaning(s) of a word. The main task of the assignment is to implement the two fundamental technologies that have been discussed during the lectures, which are sockets and threads. In addition, the above two technologies should be applied in explicit use. Here, the explicit use means sockets and threads must be the lowest level of abstraction of network communication and concurrency. Besides, a user-friendly GUI that can properly manage errors (by means of exception handling) should be implemented for this project. Finally, an efficient message exchange protocol should be designed for data transmission.

Description of System Components

Client Side

For the client side, it has a client GUI (shown in Figure 1) implemented by Swing, which contains a textfield for word entering, a textarea for meaning entering and displaying, and five buttons for different features. Apart from the GUI of the client side, a GUI controller is implemented to manage the requests made by clients and receive the responses from the server. When the client sends a request to the server side, the Client_Controller will establish a TCP connection by using a socket and transmit the data using JSON serialization. Moreover, exceptions are managed by the system properly. If there is an error occurs, a prompt will be shown in the client GUI in red color. Conversely, if the request is sent and processed by the server successfully, a green success message will be displayed.

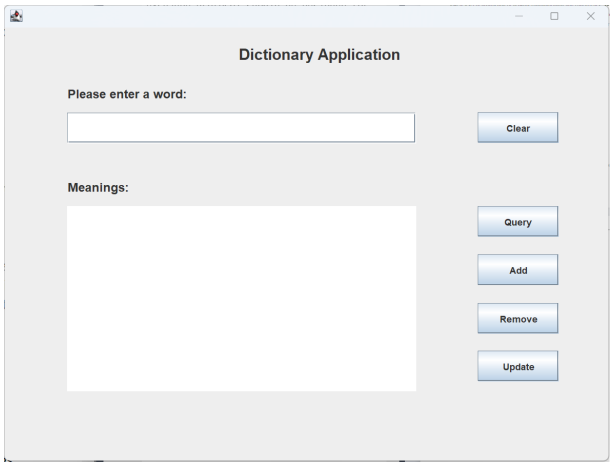
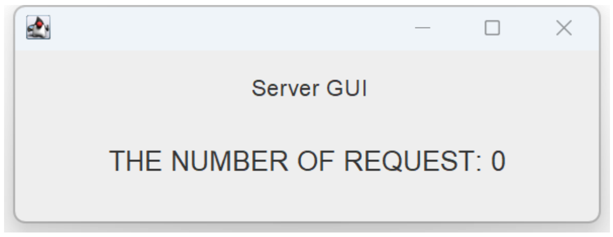


Figure 1: The GUI of Client Side

Server Side

Regarding the server side, a simple GUI is also implemented to illustrate the number of requests received (shown in Figure 2). Except for the server GUI, there are three other parts on the server side, including the server listener, server thread, and server dictionary. The server listener utilizes a server socket to receive requests from the clients. Once the server listener accepts an incoming connection, a server thread will be created to process the data from the client. At the same time, the server thread will also send a response to the client, and such a response always contains the meaning of the entered word, a prompt message, and the status of the request. Besides, the server thread will utilize the function in the server dictionary to operate the dictionary loaded in the current application.



THE NUMBER OF REQUEST: 0

Figure 2: The GUI of Server Side

Class Design and Interaction Diagram

Client Side

The following Figure 3 shows the class diagram of the client side. Overall, there are four classes on the client side. Among them, the *Client* class is the entrance of the client part, and it creates the object instances of the *Client_GUI* and *Client_Controller* classes in the main method. The *Client_GUI* class contains the Swing codes that construct the GUI. As for the *Client_Controller* class, it is designed to control the GUI of the client side, for example, send a request to the server when clicking the buttons of the four basic features (i.e., the button of QUERY, ADD, REMOVE or UPDATE). In this class, the *displayException()* method is responsible for handling the exceptions and prompting the clients, and the *OperateResponse()* method is to decode the JSON response from the server and displays it in the GUI. In addition, there is an inner private class, the *ClientThread* class, which aims to create a client thread, establish a TCP connection between the client and the server, send the request to, and get the response from the server side.

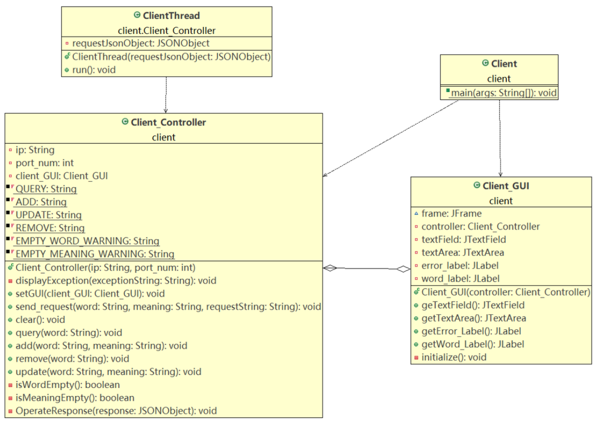

ClientThread
client.Client_Controller
requestJsonObject: JSONObject
ClientThread(requestJsonObject: JSONObject)
run(): void
Client
client
main(args: String[]): void
Client_Controller
client
ip: String
port_num: int
client_GUI: Client_GUI
QUERY: String
ADD: String
UPDATE: String
REMOVE: String
EMPTY_WORD_WARNING: String
EMPTY_MEANING_WARNING: String
Client_Controller(ip: String, port_num: int)
displayException(exceptionString: String): void
setGUI(client_GUI: Client_GUI): void
send_request(word: String, meaning: String, requestString: String): void
clear(): void
query(word: String): void
add(word: String, meaning: String): void
remove(word: String): void
update(word: String, meaning: String): void
isWordEmpty(): boolean
isMeaningEmpty(): boolean
OperateResponse(response: JSONObject): void
Client_GUI
client
frame: JFrame
controller: Client_Controller
textField: JTextField
textArea: JTextArea
error_label: JLabel
word_label: JLabel
Client_GUI(controller: Client_Controller)
geTextField(): JTextField
getTextArea(): JTextArea
getError_Label(): JLabel
getWord_Label(): JLabel
initialize(): void

Figure 3: The class diagram of client side

Server Side

Figure 2 is the class diagram of the server side. Overall, the server side contains four classes. The *Server* class is the entrance of the server part. In this class, it creates and runs the server object in which there is a listening socket to accept the incoming client connection requests. Also, it creates the object instance of *Server_GUI* class, which displays the number of requests received by the listening server socket. The *ServerThread* class is responsible for receiving and processing the request from the client side, generating a meaningful response, and sending it back. Also, there are several static strings which are the prompt messages that will be shown in the client GUI. The *Dictionary* class reads the dictionary file and loads it to the system. In addition, there are four **synchronized** functions in the dictionary, which are designed to realize the **concurrency** of the server. Note that for the dictionary part, it will first read the file according to the input argument when the main function of the server is invoked. If there is no such dictionary in that path, it will read or create the dictionary based on the fixed default dictionary path.

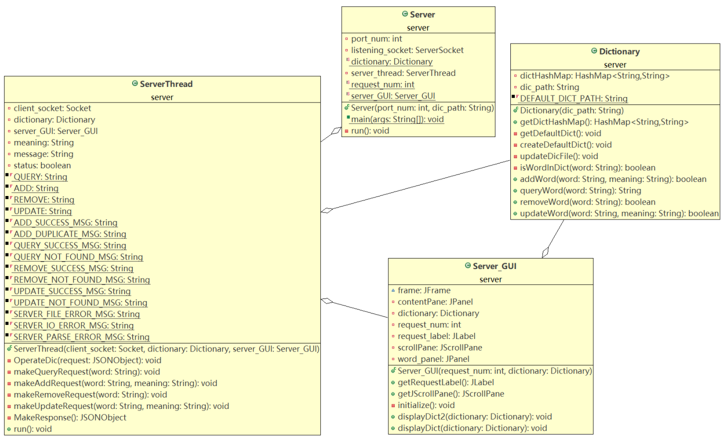

ServerThread
server
client_socket: Socket
dictionary: Dictionary
server_GUI: Server_GUI
meaning: String
message: String
status: boolean
QUERY: String
ADD: String
REMOVE: String
UPDATE: String
ADD_SUCCESS_MSG: String
ADD_DUPLICATE_MSG: String
QUERY_SUCCESS_MSG: String
QUERY_NOT_FOUND_MSG: String
REMOVE_SUCCESS_MSG: String
REMOVE_NOT_FOUND_MSG: String
UPDATE_SUCCESS_MSG: String
UPDATE_NOT_FOUND_MSG: String
SERVER_FILE_ERROR_MSG: String
SERVER_IO_ERROR_MSG: String
SERVER_PARSE_ERROR_MSG: String
ServerThread(client_socket: Socket, dictionary: Dictionary, server_GUI: Server_GUI)
OperateDic(request: JSONObject): void
makeQueryRequest(word: String): void
makeAddRequest(word: String, meaning: String): void
makeRemoveRequest(word: String): void
makeUpdateRequest(word: String, meaning: String): void
MakeResponse(): JSONObject
run(): void
Server
server
port_num: int
listening_socket: ServerSocket
dictionary: Dictionary
server_thread: ServerThread
request_num: int
server_GUI: Server_GUI
Server(port_num: int, dic_path: String)
main(args: String[]): void
run(): void
Dictionary
server
dictHashMap: HashMap<String,String>
dic_path: String
DEFAULT_DICT_PATH: String
Dictionary(dic_path: String)
getDictHashMap(): HashMap<String,String>
getDefaultDict(): void
createDefaultDict(): void
updateDicFile(): void
isWordInDict(word: String): boolean
addWord(word: String, meaning: String): boolean
queryWord(word: String): String
removeWord(word: String): boolean
updateWord(word: String, meaning: String): boolean
Server_GUI
server
frame: JFrame
contentPane: JPanel
dictionary: Dictionary
request_num: int
request_label: JLabel
scrollPane: JScrollPane
word_panel: JPanel
Server_GUI(request_num: int, dictionary: Dictionary)
getRequestLabel(): JLabel
getJScrollPane(): JScrollPane
initialize(): void
displayDict2(dictionary: Dictionary): void
displayDict(dictionary: Dictionary): void

Figure 4: The class diagram of server side

Message Exchange Protocol

Message exchange protocol is a crucial part of the distributed system as it will affect the transmission efficiency. In this project, JSON is used to implement the protocol. The data format of the request is shown below in Table 1. If the request type is 'query' or 'remove', the meaning of the request should be empty. Here is an example: **{word:"add", meaning:"", command:"query"}**

Key	Type of value
word	String
meaning	String
requestString	String ("query", "add", "remove", "update")

Table 1: Request Data Format

The response data format is shown in Table 2. The meaning represents the return meaning of the query word if the word is in the dictionary. The message stands for the prompt string of the client GUI, and the status indicates whether the request is successful. An example of the unsuccessful response will be: **{meaning:“”, message:“Sorry, the word is not found in the dictionary, please add it first.”, status:false}**

Key	Type of value
meaning	String
message	String
status	boolean

Table 2: Response Data Format

Interaction Diagram

Figure 5 is the interaction diagram of the multi-threaded dictionary server system.

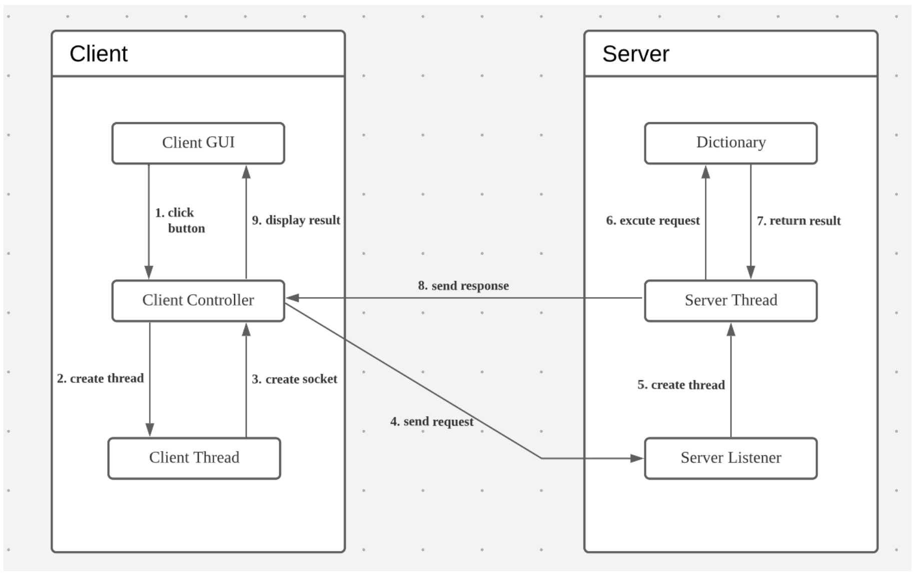


Figure 5: Interaction Diagram of the System

Critical Analysis

Advantages of the system

1. The system utilizes the **thread-per-request architecture**, in which every request/response is processed in its own thread. This architecture has good scalability as new requests come

in, new threads are created to handle them, which allows the server to handle a large number of concurrent requests without becoming overloaded. And since each thread is isolated from the others, if one thread crashes or becomes blocked, it does not affect the other threads, improving the stability.
2. Exceptions, such as *IOException* and *ConnectException* are handled properly. If an error occurs, such information will be shown in the Client_GUI, informing the user about what is wrong.
3. JSON is used for message exchange protocol. Compared with XML, JSON is a more lightweight, flexible, and easier-to-use data interchange format.

Disadvantages of the system

1. The thread-per-request architecture also has several significant disadvantages. Since each thread requires a certain amount of memory overhead, and with a large number of threads, memory usage can quickly become unmanageable.
2. The system only implements some basic functions, and the security issues and high performance are not considered carefully.

Creativity

A GUI for the server side is implemented, which can display the number of requests accepted by the server. Besides, two methods in the *Server_GUI* class are developed for displaying the dictionary. However, due to the time limit, the display effect of these two methods is not good enough, so they are commented out.

Conclusion

In this project, a multi-threaded dictionary server system is developed based on thread-per-request architecture. This system is able to process requests concurrently. Exceptions are handled properly and can prompted to user. However, there are some parts that can be improved further. For example, worker pool architecture can be used since the performance of thread-per-request architecture drops when the number of threads increases.