COMP90015 – ASSIGNMENT 1 MULTI-THREADED DICTIONARY SERVER REPORT

Zhuolun Wu, 954465

18th, April 2021

Problem Scope

The dictionary should be built on both server side and client side, where the client side must implement a GUI for interaction. Here are some main problems and requirements for this system.

1. Server and client should be able to communicate through TCP socket.
2. Server and client must use an encoding format which supported by both sides.
3. Server should respond to client with any request, either succeed or fail, word found or not.

System Description

The client side is responsible for sending request to the server through TCP socket, get and display server's reply to users. The server side is responsible for receiving request from the client, create a corresponding thread, accept TCP connection, read, write, and search the dictionary and send the reply to client.

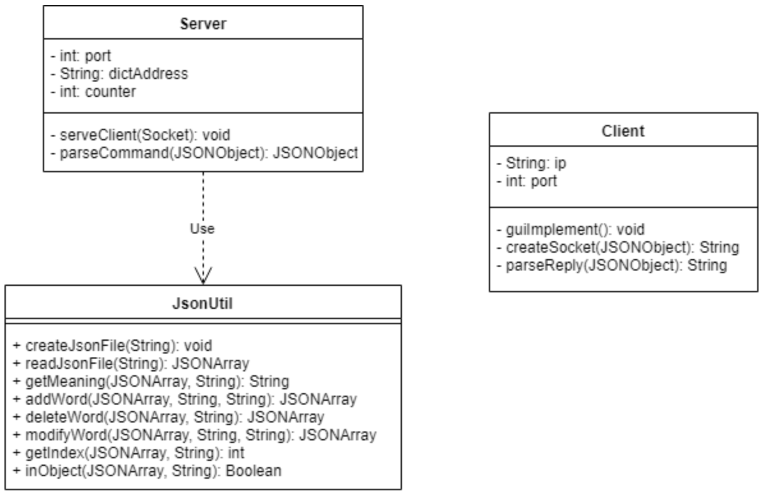
The dictionary is being stored as a JSON file to achieve data consistency on both sides, and an external JSON package for Java has been imported. If the dictionary file does not exist, a new empty dictionary file would be created.

About socket and thread, the architecture here is **thread-per-request**.

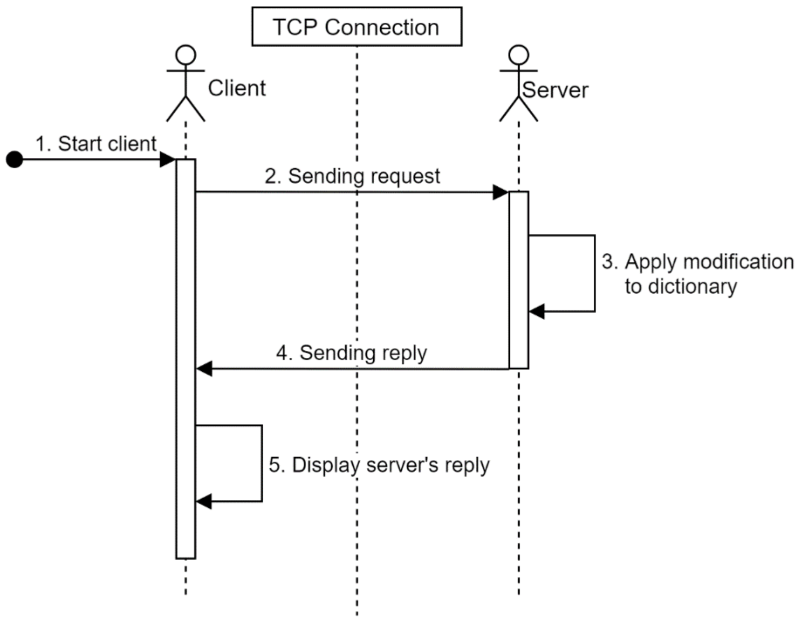
A TCP socket is being used when the client sends the request, and therefore a thread is being created for this communication. The thread would parse client's request, work on the dictionary, and reply to client in the end. In addition to the multithread example showed in tutorial, the thread (server's while-true loop for listening) would be ended when the server replied to client's request. The main reason for using thread-per-request is that it would let the communication being terminated after the reply being sent, therefore server does not need to keep running the thread and load is being reduced.

Overall Class Designing

Source codes consist of two runnable classes – one server, and one client. Both of server and client can send, analysis and reply messages within the class itself, except that the server would need to use external functions in the JsonUtil class to read and write dictionary file.


Server
- int: port
- String: dictAddress
- int: counter
- serveClient(Socket): void
- parseCommand(JSONObject): JSONObject
Use
JsonUtil
+ createJsonFile(String): void
+ readJsonFile(String): JSONArray
+ getMeaning(JSONArray, String): String
+ addWord(JSONArray, String, String): JSONArray
+ deleteWord(JSONArray, String): JSONArray
+ modifyWord(JSONArray, String, String): JSONArray
+ getIndex(JSONArray, String): int
+ inObject(JSONArray, String): Boolean
Client
- String: ip
- int: port
- guiImplement(): void
- createSocket(JSONObject): String
- parseReply(JSONObject): String

And the whole communication process as described in System Description would be:



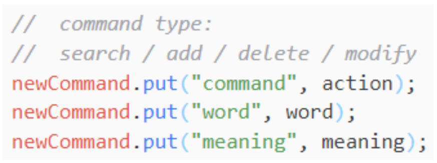
Steps

1. Start running client from command argument line
2. Client collecting word and command from GUI and createSocket()
3. Server serveCommand() and parseCommand()
4. Sending back to client when dictionary operation is finished in parseCommand()
5. Client parseReply() and display through GUI

Failure Analysis

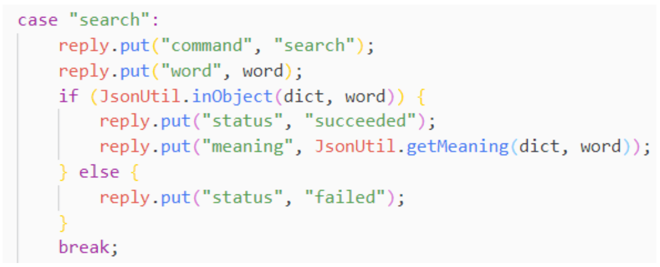
There are two major failures of the system.

The first one is the message for communication has not been properly structured.



```
//  command type:
//  search / add / delete / modify
newCommand.put("command", action);
newCommand.put("word", word);
newCommand.put("meaning", meaning);
```

message generated by client



```
case "search":
    reply.put("command", "search");
    reply.put("word", word);
    if (JsonUtil.inObject(dict, word)) {
        reply.put("status", "succeeded");
        reply.put("meaning", JsonUtil.getMeaning(dict, word));
    } else {
        reply.put("status", "failed");
    }
    break;
```

message generated by server

It would be better to create a separate class storing all utility functions for parsing messages and make message structure consistent as well. This new class should also store all error message, and therefore the communication can include error codes only. This would make the system cleaner and less confusing.

The second one is that the error message for TCP connection has not been developed. It would only show error message from java, instead of showing directly what error occurred during the communication of both sides.

In conclusion, this system has basically implemented all necessary features of a server-client structured dictionary application, with thread-per-request architecture being used. It can successfully search, modify, add, and delete words and their meanings in a dictionary and show important error messages. However, this system still contains small problems like the inconsistency of message structure and hard-understanding TCP connection errors.