COMP90015 Distributed Systems – Project 1

Multi-threading dictionary server report

Name: Kexin Wang
Student ID:847024

The problem context in which the assignment has been given.

The aim of this project is to build a multi-threading dictionary system. Multiple clients should be able to connect to the server and make dictionary requests concurrently. The design of thread architecture can directly affect the performance of the system, such as request handling speed, client connection waiting time. The main operation from the client is to make requests on searching / adding/deleting a word. Since all users are sharing dictionary resources, concurrency needs to be maintained in a way so that information can be accurately displayed or modified upon the requests order.

A brief description of the components of the system.

Server threading architecture:

Choosing the thread architecture for the system can be one of the core decisions that I need to think before starting to implement. Thread pool and thread per connection architectures were mainly considered. Thread pool has a fixed number of threads initiated, and clients will queue up if the number of clients exceeds the pool size. As a result, it could lead to a waiting period until a thread is free. However, it minimizes overhead time due to the thread creation, and also helps to control the use of system resources. On the other hand, using thread per connection architecture will create a new thread each time a client is connected. Thread creation time needs to be considered, as well as the possibility of using up all threads available to the system. After above consideration, thread pool was adopted as the server threading architecture

Fixed number of threads are initiated after the server launches. Each thread will then keep checking on a client task queue (shared between all threads), serving the first client in the queue if there is one. Repeat this process so that no thread is dead.

Socket:

A communication protocol is required to transfer data between endpoints through sockets, that is server and connected clients in this case. Choosing between TCP and UDP was not hard as our system has to be reliable. Therefore, the connection-oriented protocol TCP is used to ensure the reliable flow of data transfer.

IP and port are required when a client is trying to connect to the server. They are passed through as command arguments by the client. TCP 3-way handshaking process will then be performed

Client task queue:

The system handles client tasks using a LinkedBlockingQueue. The client handler object is added into the queue and waiting to be processed by a free thread worker. **Synchronized clientQueue** was adapted to perform wait () and notify () actions. This helps to keep the server and threads agree about the state of the predicate and avoiding infinite waiting time at the thread end.

Dictionary in memory (synchronization issues**)**

ConcurrentHashMap data structure is used to store the Json dictionary file into memory. The main reason that I decided to use it is that it solves the threads synchronization issues and there is no need to use any synchronization block with it. It is also a map which helps to store dictionary data structure (word: meaning pairs).

Dictionary local file

Json format is used to store the dictionary locally. I chose it because it is great for interchanging data. File is read into memory when server launches, and file will be updated when either server or client is closed.

Exception Handling:

- A dialog message will be displayed to the client if there is a connection error, such as an error occurred on the server side or connection is cut by the server due to no request made in a certain time (more detail in the creation part).
- To minimize the chance of Client/Server accidentally closing the window, a dialog message would be displayed when the cross button is clicked and make the Server/Client double check if they actually want to quit.
- In the case where a requested word or meaning is invalid, a corresponding message will be displayed on the client side. Including words is too short or consists of invalid characters, meaning is invalid or empty when adding a word, as well as adding a word which already exists(all in lowercase) or deleting non-existent words in the dictionary etc. There are more constraints set in order to prevent bad data from appearing in the dictionary. With those constraints, clients' requests are more meaningful.
- There is also error handling on the server side where the inputted dictionary file is not found. Error message will be shown on terminal and exit program.
- On the client side, if the command line input address is not reachable, exception message will be displayed in the terminal and exit the program.
- For both client and server command input, if the number of input is not 2, an error message will be printed out and system will exit.

An overall class design and an interaction diagram.

Class Design

There were 6 classes implemented and were separated into 3 packages: Server, Client, Dictionary Request**.** I will now discuss them package by package.
Server:

- Server class:
  - The main class which launches the server. It firstly reads dictionaries into a concurrentHashMap as described in the last section and then launches the GUI and Then it initiates (runs) a fixed number of threads to create a thread pool. After this point, the server will **keep on** listening and accepting client's connection requests. For each client connection, a client handler object is created and added to the queue as well as calling queue.notify() function.

- ThreadWorker class:
    - ThreadWorker extends the Thread class, so that there is a run function. Fixed number of ThreadWorker objects are created and started by the server to form the thread pool as described above. Each thread repeats the process of tracking on the client queue and serves the client when the queue is not empty by calling start() function.
- ClientHandler:
    - ClientHandler is created every time a client is connected and added into the queue, waiting to be served by a thread. run() function is called by the thread worker, handling client requests such as searching, adding and deleting words from/to the dictionary and sending responses to the client.

Client:

- Client class:
    - The client class is the only class on the client side, which tries to connect to the server with provided IP address and port number, launching GUI and making requests. If the server is unconnected, the client will be notified and close the frame.

Dictionary Request:

- ReadWriteFile class:
    - The ReadWriteFile object is created when the server launches. It handles reading the Json file into ConcurrenthashMap structure and writing the map dictionary in the memory into the Json file. Memory data will be updated every time a client is unconnected or a server that is closed without error.
- RequestHandler:
    - This class helps the client handler to handle client' requests. It directly extracts/modifies data in the memory and keeps updating the dictionary stored in a ConcurrentHashMap (details of this data structure have been discussed in the last section). Server class can also get the current map and write into the local file before socket closing.

Class Diagram: **Check the link for a better visibility**

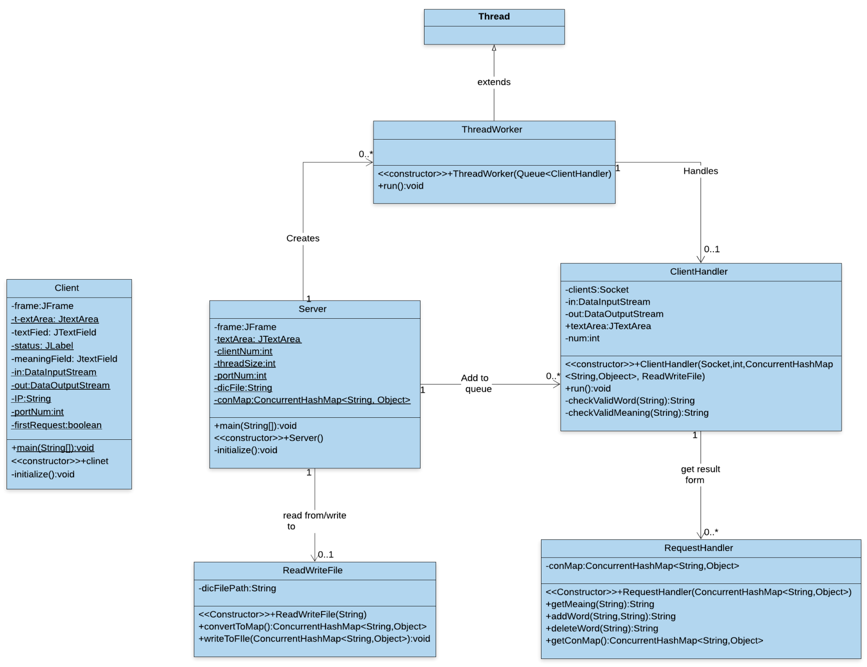


Figure 1： class diagram

Interaction diagram. **Check the link for a better visibility**

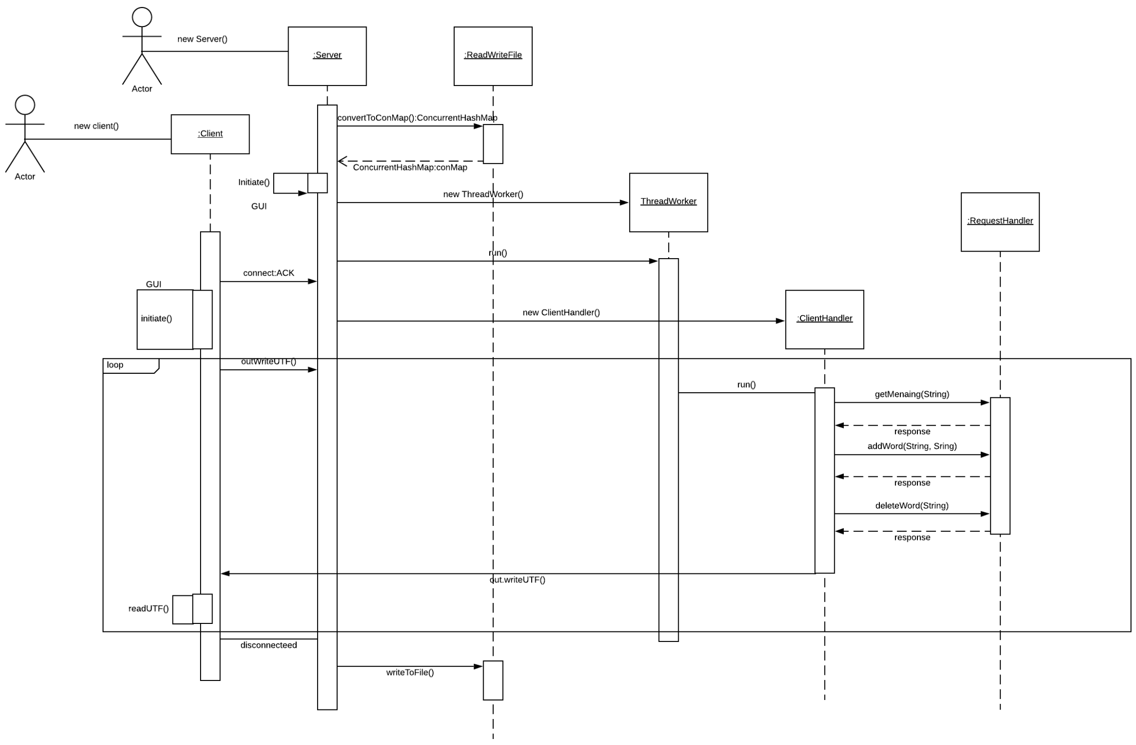


Figure 2: Interaction diagram

Creation

GUI

- **Placeholder:** placeholder text is in the word and meaning field in gray color (Figure 3), which helps the client to know what to enter in the field.
- **radioButton:** Radio buttons were used to show which mode (search, add, delete) the client is on (Figure 3). Comparatively, it works better than a simple button as it is visible when a radio button is selected.

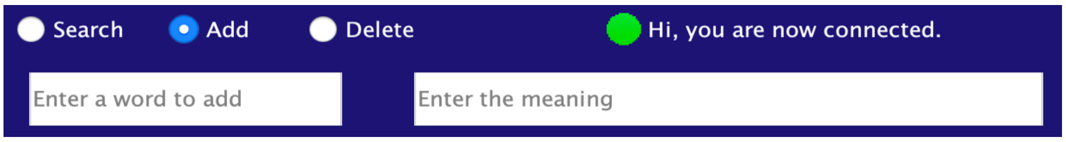
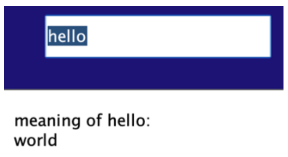


Figure 3: place holder and radio buttons example.

- **Scroll bar:** This function helps to scroll over the text area so that clients can see previous requests and results. newest request and response will always be displayed, scroll bar automatically scrolled.
- **Key listener**: Client can make a request by pressing the enter key. This idea was inspired by several dictionary applications such as EuDic. When a search and delete request is sent, word entered would be selected which is easier to changes on the previous request (Figure 4).


hello

meaning of hello:
world

Figure 4: Word selected example

Implementing Thread pool:

Thread pool was implemented without using high level libraries. Different data structures were attempted in order to achieve similar results.

Status display:

A notification label was implemented to notify the status of the client connection. After launching the client, it asks the client to make a request. The label would tell the client whether there is a thread serving him/her by showing "you are now connected" or "you are in line to be connected" with the corresponding icon set. If the client is waiting in the queue, the label will change as soon as a thread is connected.

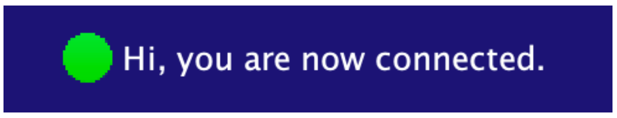

Hi, you are now connected.

Figure 5: Connected status

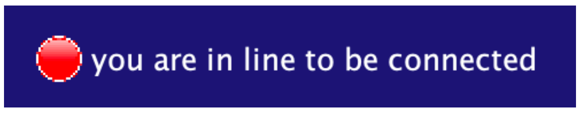

you are in line to be connected

Figure 6: In line to be connected status

Request Queued when waiting:

All requests made by the client are queued up and all will be displayed when a thread is available to serve him/her. The main point is that the data stored will be based on the request time and will be displayed to the user when being serverd.

- For example, client 1 is connected, client 2 is waiting to be served. Client 2 wants to search for a word "Hi" (suppose it exists in the dictionary), client 1 deleted the word "Hi", client 1 is disconnected, and client 2 is now connected and would get the meaning of Hi rather than the non-existing word message.

Writing memory into local file:

Whenever a client is disconnected or the server closes the GUI window, memory data will be written into the local Json file.

Note on testing

- The thread pool size is set to **3** for testing convenience.
- Please include all icon images (search.png, red.png, green.png) and dictionary.json in the same directory when running the jar files. Refer to Figure 6, this would enable the notification icon to be shown.

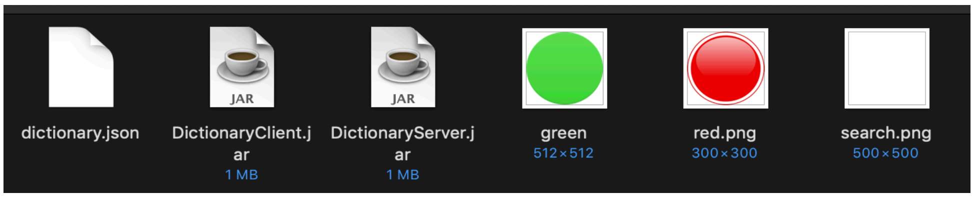


Figure 6: Jar files running requisite

- Client will be disconnected if no request is made for **60 seconds.**

A critical analysis of the work done followed by the conclusions.

In this project, an implementation of a multi-threading dictionary has been attempted without using high level libraries. The system allows clients to make TCP protocol requests and get responses from the server without conflicts. Data structures were used to overcome the concurrency(synchronization) issue of the multi thread structure. A signal label was implemented to inform clients about connection status. Apart from these functionalities, A great amount of time was spent on designing the client-side GUI to provide a user-friendly interface. GUI components such as text fields, placeholders, scroll bars, "enter" key listeners etc, could definitely improve user experience and provide a smooth request making process. However, there are still functions that I would have wanted to implement but did not due to time constraints, such as enabling clients to add word type (i.e. verb, noun, adjective) when adding a word and display it out when any client searches the word, as well as word recommendations for user input.

After going through the process of implementing thread pool structure, I have again a better understanding of the concept of a multithreading system and how it enables multiple users to access a shared resource. I am now having more confidence in both using java and overcoming obstacles when solving difficult problems. Finally, I am really looking forward to the second project of this subject.