
# *COMP90015 Distributed Dictionary*

# *Assignment 1 Sem 1 2021 Report*

# *Multi-Threaded Dictionary*

***Name*** **:** Chan Jie Ho

***Student ID*** **:** 961948

***Email*** **:** chanjieh@student.unimelb.edu.au

***Word Count*** **:** 1839

---

TABLE OF CONTENTS

# 1. Introduction

Almost every program is built with the intention to handle multiple clients at once in mind. Concurrency when poorly handled can create a lot of problems, e.g. update conflicts, propagating change to clients, etc. This can be easily handled using sockets and threads to synchronise the reading and writing of the data.

This project was to design and implement a multithreaded dictionary server using sockets and threads as the lowest level of abstraction for network communication and concurrency to allow for concurrent clients to interact with a dictionary through a client graphic user interface (GUI). The application, both the server and the client codes, was created using Java while the client GUI was created using Java Swing Designer.

# 2. Requirements

Below is an exhaustive list of requirements as outlined in the assignment specification:

## 2.1. Functional Requirements

- Search up the meaning of a word
- Add a word and its meaning to the dictionary
- Update the meaning of an existing word
- Remove an existing word from the dictionary

## 2.2. Non-functional Requirements

- Concurrent user handling using socket and thread programming
- Appropriate custom message exchange protocol
- Proper error handling
- Creation of client GUI

The implementation of these requirements will be discussed in ***Section 3***.

# 3. Features

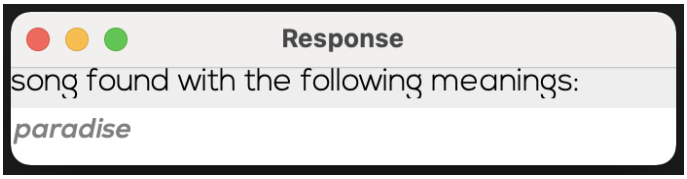
Here I present functional and non-functional features implemented. Extra features implemented have been delegated to or embellished in ***Section 4*** to avoid repetition.

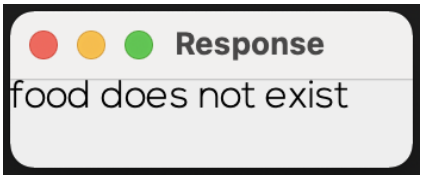
## 3.1. Client GUI

The client GUI was designed with simplicity in mind. The window has a search box in the middle and buttons for each of the functional requirements (search, add, update, and remove) as well as other extra features such as a button to get the full dictionary and a do-not-suggest-word checkbox (see ***Section 4***)

## 3.2. Search word

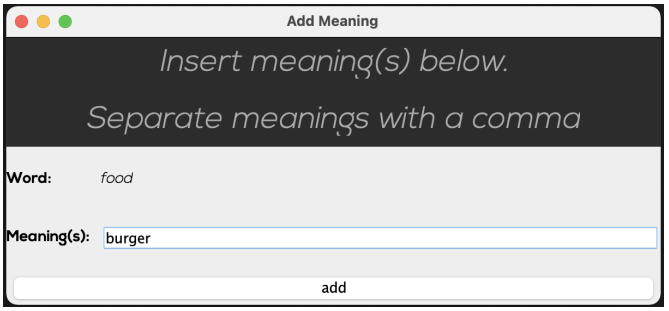
When the user inserts a word and clicks "Search", the server will perform a server-side query and return one of two reponses that the client would display to the client:



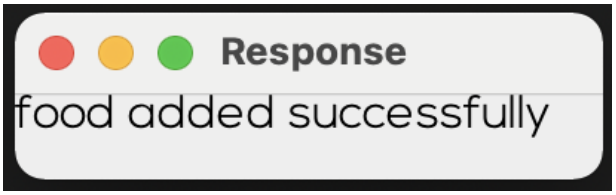



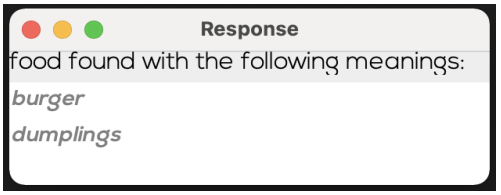

## 3.3. Add word

If the "Add" button was clicked, the client would then generate another window that would prompt the user to key in the meaning(s) of the word. Meanings would be separated with commas (shown in next part).



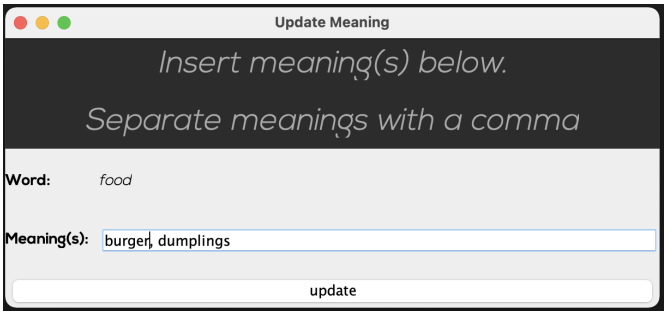

Once the user keys the meaning in and clicks "Add", the client would send the command to the server that would first query the word to check if the word is already a part of the database. If it is not already in the dictionary, the word is then added, else the server would respond with the meaning of the duplicated word. This would be displayed to the client in one of the following manners:



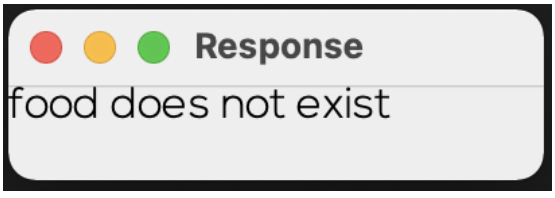



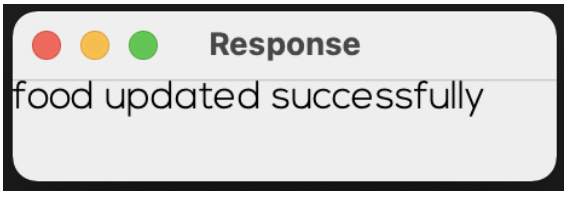

## 3.4. Update word

The "Update" feature functions in the same manner as the "Add" feature would. The only difference is that the server would only respond with a success status and update the word if it already exists in the dictionary, else the server would respond with a failure status.



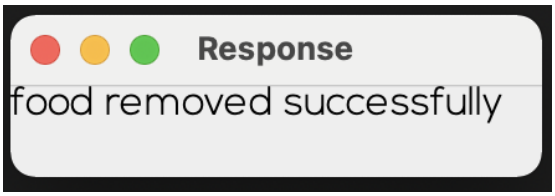

This would be displayed to the client in one of the following manners:

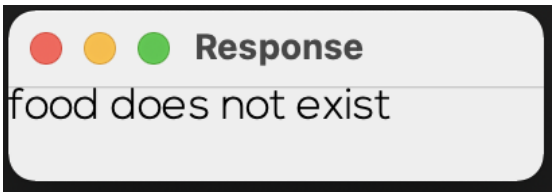


### 3.5. Remove word

Lastly, a user can also remove a word from the dictionary and the server would then respond with one of two responses depending if the word first exists in the dictionary that's displayed to the user as the following:



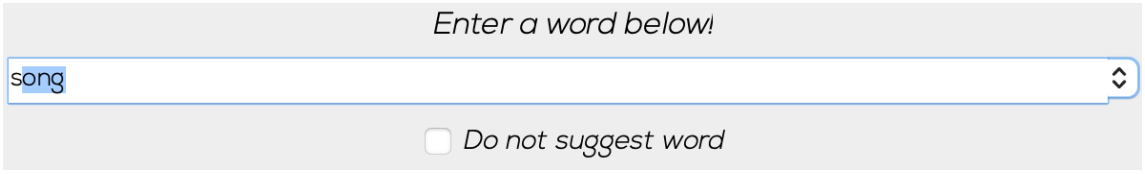

# 4. Creativity Features

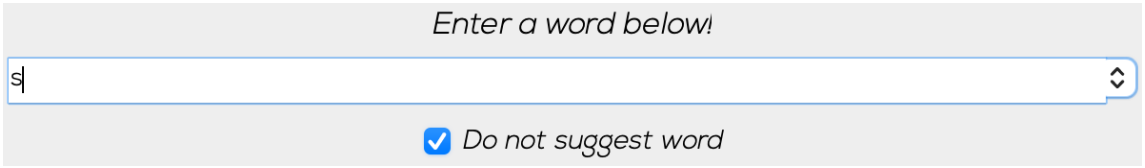
Here I present the extra features implemented that I believe makes my program stand out from the others.

## 4.1. Auto-complete drop-down search box

Using the JComboBox from the Java Swing library, I have managed to implement an auto-complete drop-down search box. The JComboBox is a drop-down box that users can expand and pick from a selection of options. This list of choices is known as the model of the combobox. This model is first filled when the user first starts up the client.

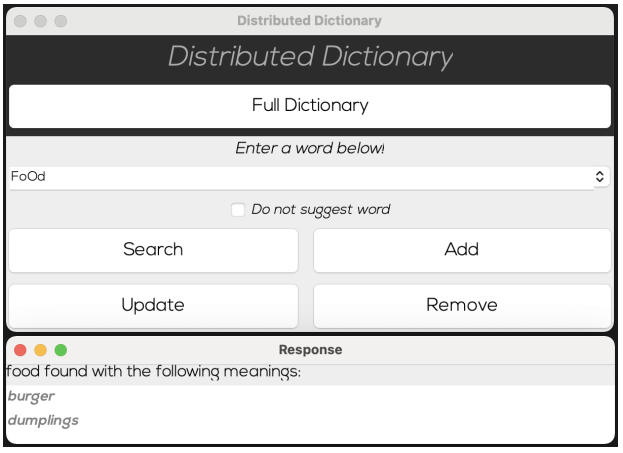
The auto-completion feature is where the user can start typing a word and if the word is already a part of the search model, the combobox would suggest it to the user. This feature can also be disabled by clicking on the do-not-suggest checkbox.



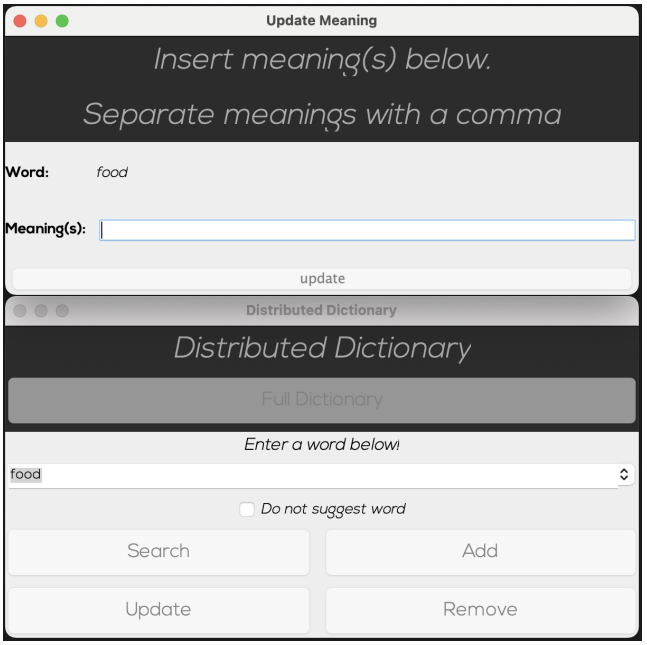

## 4.2. Case insensitivity

I have also implemented the server and client to allow for case insensitivity for all its features. I had initially thought of allowing for case sensitivity as well but scrapped it halfway as it would cause problems when searching for one of the two meanings



### 4.3. Button disabling

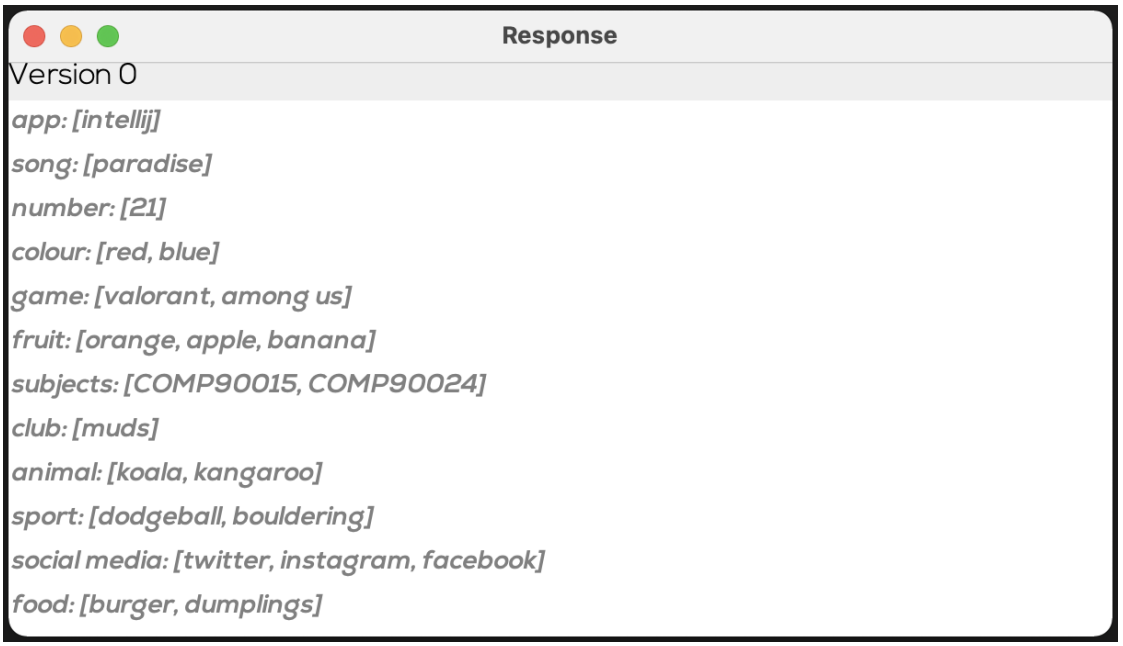
Another extra feature is that when the user has the add/update meaning window open, the buttons on the main client window is then disabled to ensure that the user doesn't accidentally open up multiple windows when adding/updating the same word.




Besides that, within the add/update meaning window itself, the add/update button also only becomes enabled if the textbox has at least 1 non-whitespace character to ensure the user does not send a word to the server without a meaning.

### 4.4. Full dictionary

Lastly, I also added a full dictionary button that the user can click to view the entire list of words and their associated meanings that is currently in the dictionary.




## 5. Message Exchange Protocol

All messages between the client and server are sent as JSON strings that would then be parsed to be JSON objects. The structure differs for most of the request types apart from a few similar keys. Below is a list of the possible requests and messages for the features explained above:

```
Client : {request: "start"}
Server : {words: [...]}

Client : {request: "full", version: xx}
Server : {status: "up-to-date"}
       : {status: "updated", version: xx, keys: [...], values: [[...], ...]}

Client : {request: "add", word: <WORD>, meanings: [...]}
Server : {status: "success", message: "<WORD> added successfully"}
       : {status: "failure", message, "<WORD> found with the following meanings: ",
          meanings: [...]}
       : {status: "error", message: <ERROR MESSAGE>}

Client : {request: "search", word: <WORD>}
Server : {status: "failure", message: "<WORD> does not exist"}
       : {status: "success", message, "<WORD> found with the following meanings: ",
          meanings: [...]}
       : {status: "error", message: <ERROR MESSAGE>}

Client : {request: "remove", word: <WORD>}
Server : {status: "failure", message: "<WORD> does not exist"}
       : {status: "success", message, "<WORD> removed successfully"}
       : {status: "error", message: <ERROR MESSAGE>}

Client : {request: "update", word: <WORD>, meanings: [...]}
Server : {status: "success", message: "<WORD> updated successfully"}
       : {status: "failure", message, "<WORD> does not exist"}
       : {status: "error", message: <ERROR MESSAGE>}
```

# 6. Structure

This section describes the class structure and interactions for both the client and server.

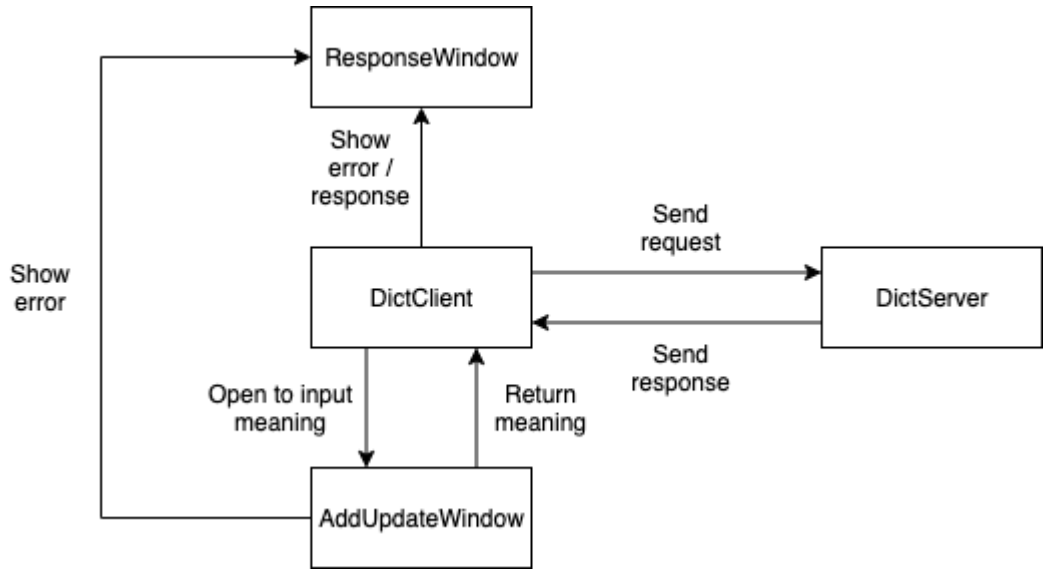
## 6.1. Client

The client is composed of 3 classes: `DictClient`, `AddUpdateWindow`, and `ResponseWindow`. The `DictClient` class handles all the client-server communication and most of the request creation such as the requests for `search`, `remove`, `full`, etc. However, because only the `add` and `update` requests require the user to provide meanings for the associated word, I decided to separate the handling of the meaning to the `AddUpdateWindow` class that creates a new window. This allows the main `DictClient` class to not need to handle missing meanings. The `ResponseWindow` class is for creating a new window for printing out the response received from the server so as to not need to resize the main `DictClient` window to fit in the extra lines.

## 6.2. Server

The server only consists of one `DictServer` class. This class handles all the client-server communication, request handling, and processing of the requests.

### 6.3. Interaction Diagram

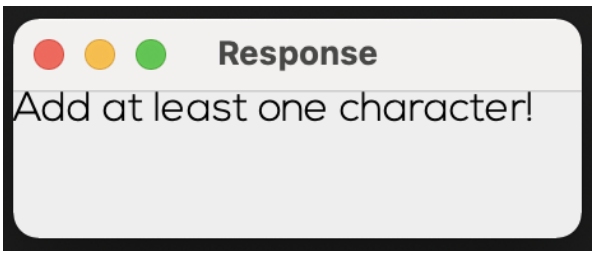



## 7. Excellence Aspects

Below is where I explain the extra aspects past the minimum requirements that I have implemented.

### 7.1. Preliminary Error Handling

Preliminary checks for user input format and ensuring inputs are non-empty on the client side would ease the burden from the server side as servers would not have to waste as much resources sending back error responses due to clients sending out empty strings, etc.




However, this does not mean that the client handles all the error handling – I have also implemented so that both the server and the client would handle their own errors and print out meaningful error messages. I believe that I have managed to cover most, if not all, of the possible errors that could arise.

### 7.2. Cache

After a user chooses to view the full dictionary, the client then stores a cache of the dictionary. This allows us to only need to send the request type along with version number and the server would only have to send out the full dictionary if the client's cache is not up to date. This provides both advantages and disadvantages when it comes to scalability.

One advantage is that it is easier for the server when it has to handle many concurrent clients / requests especially if the dictionary file is big. This is because sending out the full dictionary as a TCP packet would take a lot of time and this would be wasteful if the dictionary has not been updated since the last time the user requested to view it.

However this would be disadvantageous for the client if the dictionary is very big as this would take up a lot of storage space on the client side and would cancel out the advantages of cloud storage. Hence this implementation would be more suitable for small to medium sized scale of usage.

# 8. System Analysis

Lastly, this is where I perform extra critical analysis of the components introduced above that have not already been justified when explaining them.

## 8.1. JSON Parsing

The reason I chose to use JSON as my message exchange protocol structure is that it makes it easier and faster to parse to find the appropriate info needed for each function. That said, I have to ensure that I maintained a list of the structure and that all parts of my code followed that structure as if not written properly, it would result in broken json and/or a lot of errors.

While this may not be much of a disadvantage since it is an individual assignment, this may complicate things when it comes to openness of the system when expanding on this project by adding new features in the future. Expanding this system would require having to take note of the current JSON structure and it would also be hard to change up the current JSON structure. However, this may not be too big of a disadvantage if proper documentation is done.

## 8.2. Server Class

Currently, the `DictServer` class is all written in one class due to the small size of the project as it only has four main commands and a few other extra commands. If the project were to be expanded to have more features, it would be better to introduce a client handler class that handles the creation of all the threads and communication with the client while another class handles the processing of the request.

# 9. Conclusion

In conclusion, this project has managed to open my eyes to how much planning is needed to build a robust and reliable distributed system. It is also imperative to properly plan through the requirements, scalability, and openness of a system so as to be able to choose the right technology and designs as a small-scale system could make do with a simple system but a large-scale system may not be supported by the simplicity.