SWEN90015 - Distributed Systems
Report for Assignment 1

Lai Wei Hong
ID:1226091

University of Melbourne
e-mail: weihong@student.unimelb.edu.au

April 18, 2021

Contents

1. Introduction

The client-server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. The aim of this assignment is to design and implement a multi-threaded server that stores key-value pairs of words and its respective meaning. A strict requirement for this assignment is to make explicit use of sockets and threads in the design and implementation of the server-client system.

2. System Architecture

The system consists of three distinct components, which are the client(s), server and databases. The interaction between these components are illustrated in the sequence diagram below. Further information on the design and implementation details will be discussed in the following sections.

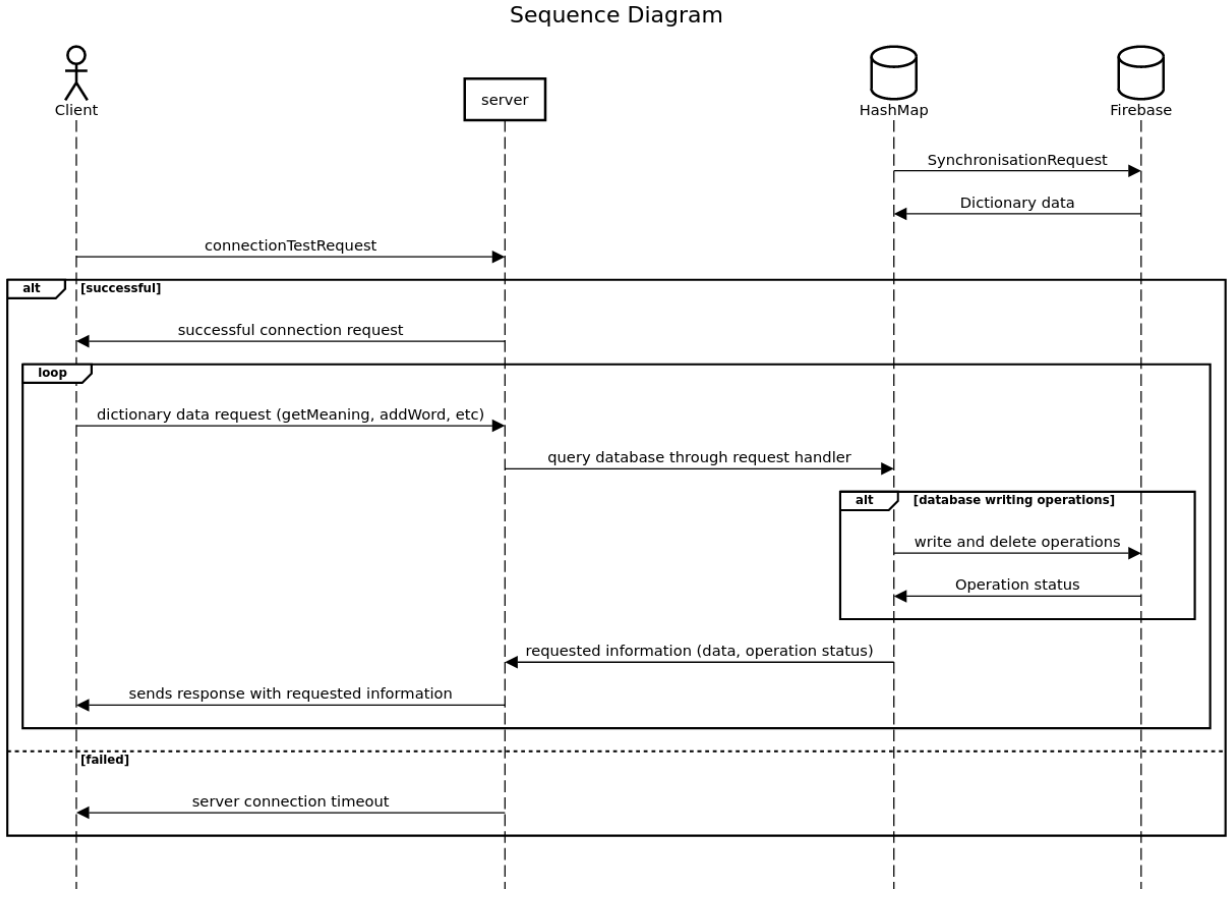

Figure 1: Interactions between components in the system

2.1. Client

The user interface of the client is developed using JavaFX. Through the user interface, the end user is able to perform various actions that are predefined in the assignment brief. These actions are word meaning queries, word addition, word removal and word meaning updates.

After the user performs one of the actions, the ClientFXMLController records the input information and relays it to the Client object which has information on the server address. The Client object packs the inputs into a Packet object, which is then converted into a JSON string using the GSON library. After the request is sent to the server, the Client starts a busy-loop cycle and waits for a reply from the server. If no response is received in 6 seconds, a timeout is called and an error message is shown to the user.

If a response can be received by the end of the timeout, the Client object unpacks the JSON response string and relays the information to the ClientFXMLController. The ClientFXMLController displays the requested output to the user.

A design decision made for the client that is not required in the assignment brief is the inclusion of a welcome page. The welcome page presents an interface that allows the user to input the ip address and port of the target server. This decision has two main benefits.

- Clear error messages
  Any exceptions experienced when inputting incorrect IP addresses can be shown clearly via an alert box instead of a line of text seen in command line implementations. This allows the user to know if any errors has occurred in the system.
- Ability to change servers
  By having a dedicated page to dealing with the server IP address, the user can easily switch back to the welcome page from the regular user interface to change the server that it is connected to. This feature is useful when a server is experience failure and the end user is forced to connect to a different one.

Shown below is the class diagram of the client.

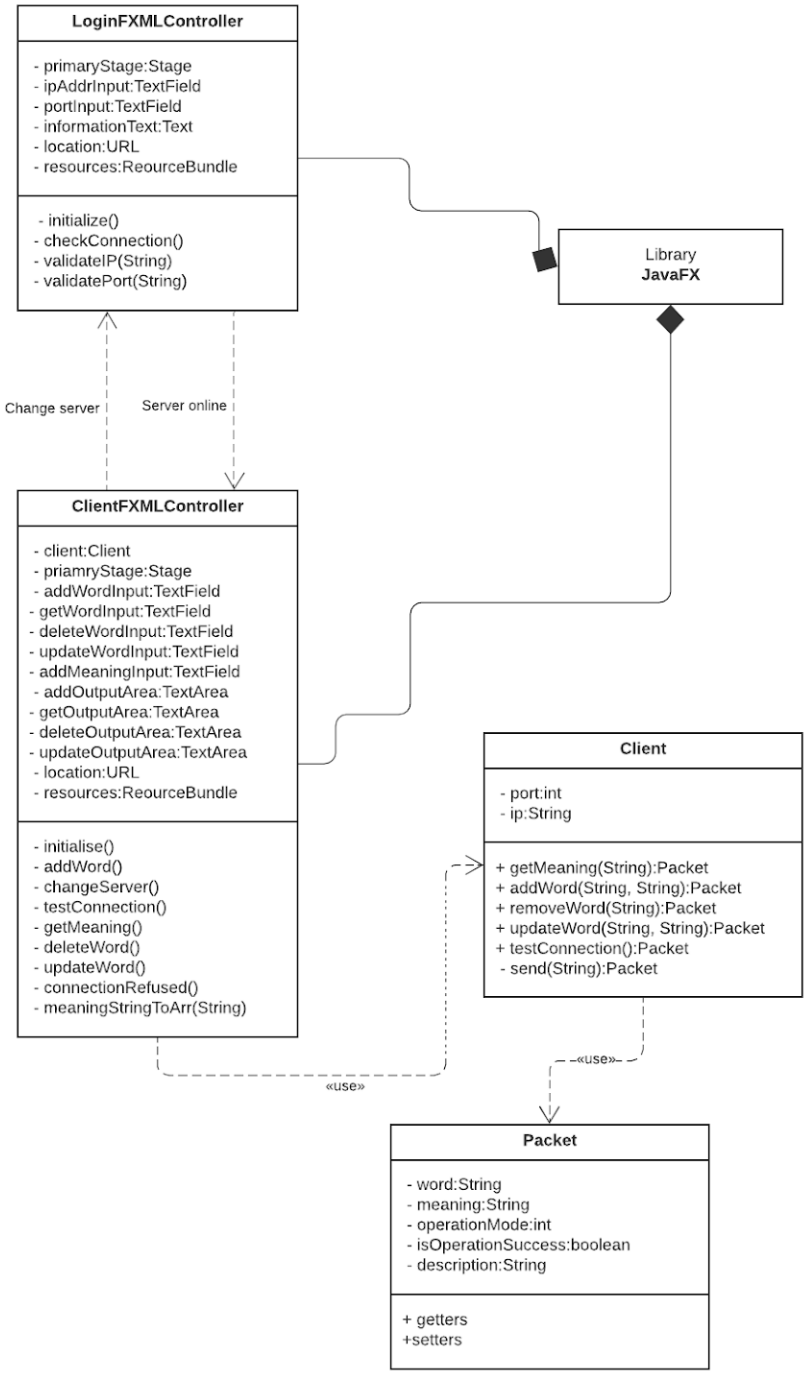

Figure 2: Client class diagram

2.2. Server

The purpose of the server is to receive requests, perform queries on its internal database and send a reply to the client based on its requests. As such, the server is expected to be able to handle instances where there are a number of concurrent requests. To achieve this, a threadpool of size 15 is adopted from the java.util library to achieve concurrency.

When the server receives a request from the client, it assigns a free thread from the threadpool to a request handler. This initial request handler determines the operation that it needs to carry out and passes on the request parameters to the specific request handlers, ie updateRequestHandlers, removalRequesthandlers, etc. These specific request handlers communicate with the database handlers to obtain the requested information. The information is relayed back to the initial request handler where it is packaged into a JSON string to be sent back to the client.

2.3. Database

The database houses the dictionary data that is made up of words and their respective meanings.In this implementation, the database is made up of two components, i.e. a hashmap data structure stored in memory and a Firebase Realtime database stored in the cloud.

The database operations are as follows. During server initialisation, the server syncs its hashmap with the entries in the Firebase dictionary. The specific request handlers mentioned in 2.2 will interface with the hashmap database handler when it is performing read and write operations. At the end of successful write operations(write and delete), the hashmap database handler will spawn a new thread to perform the same operation on the Firebase database.

The motivation for storing the dictionary data in memory is its speed of execution. Compared to querying the Firebase server or reading files, the O(1) complexity for insertion and lookup of a hashmap is multitudes faster than the two alternatives.

However, this design has its flaws. When a server is shut down or crashes, all dictionary data is lost. To counter this, a secondary Firebase dictionary is implemented. The key responsibility of the Firebase database is to act as a persistent data store for the server. In the event of crashes or outages, the server will be able to retrieve lost data from the firebase server. On top of that, by maintaining two means of data storages, the server can decouple itself when the Firebase database malfunctions.

While this architecture ensures that our data is able to persist when faults occur, it is important to note that the intial syncing operation is highly inefficient when the dictionary size is huge.

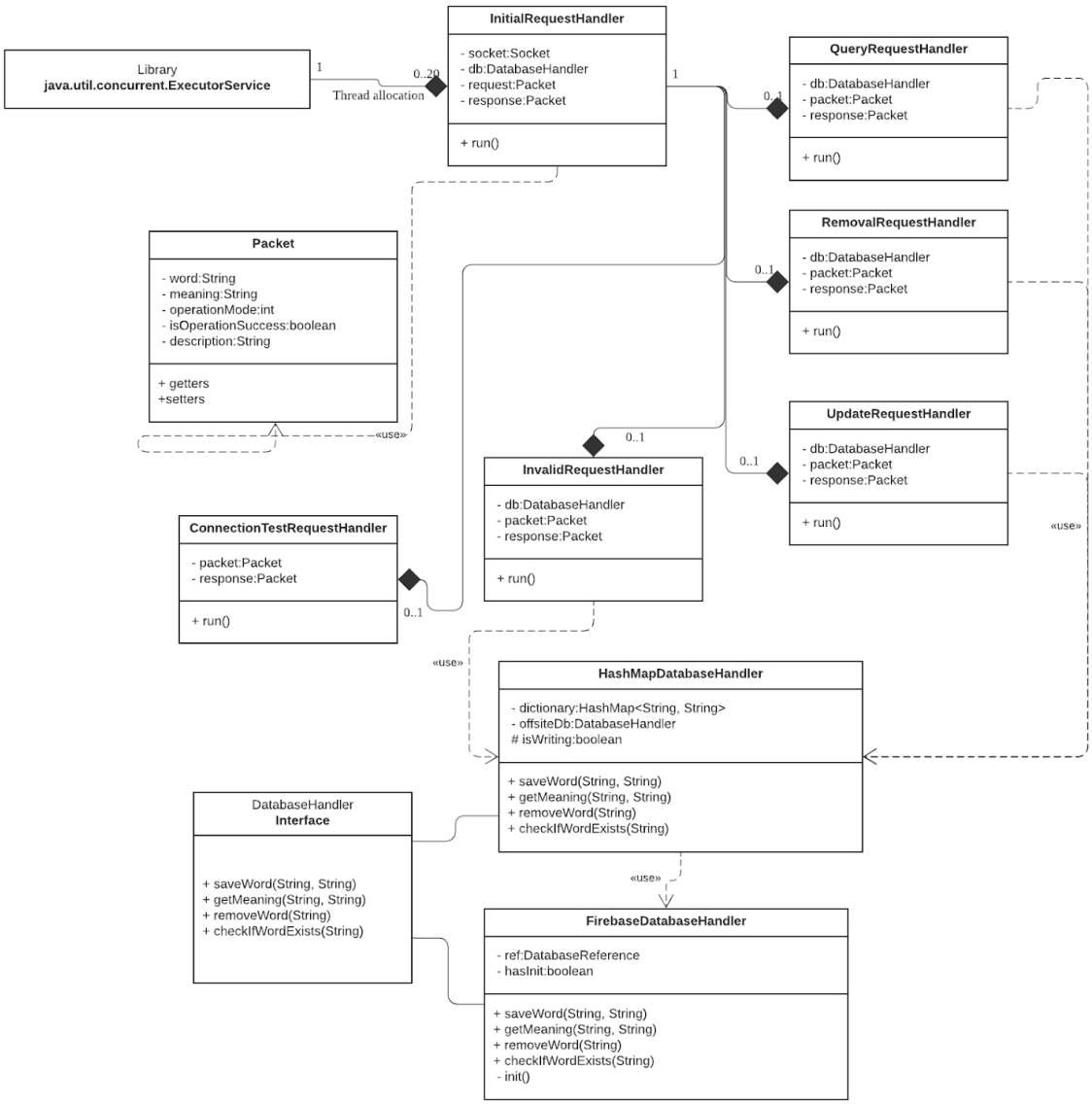

Figure 3: Server class diagram

2.4. Server-client communication

The server-client communication follows a request/response model using the TCP protocol.

The data is transmitted using the JSON data format. JSON is used in favor of XML (a common markup language for encoding documents) and Java Object Serialization because of its simplicity. The data encoded in JSON is much shorter than that of XML, which helps reduce the latency of the transmission. The widespread use of JSON is another factor that went into the decision of adopting JSON as the designated data format. The popularity of JSON would ease the process of extending the server to provide services for clients that did not implement Java specific data formats.

3. Failure Models

In this section, we will discuss about different kinds of faults encountered by the system and actions taken to handle it. As mentioned in the assignment brief, all communications are assumed to be reliable, so these faults will not be considered.

- Data packet loss
- Data packet manipulation/Changes in data packets due to noise

Note, if you find the document is getting too long, feel free to skip this part.The results and conclusion is much more relevant for this assignment. This is here mainly for the sake of completion. :)

3.1. Server

3.1.1. Fault 1: Port number is not passed in the command line

- Overview
  The admin may have forgotten to set the port number for the server during initialisation.

- Error correction
  If no port parameters are passed, port 1234 is selected by default.

3.1.2. Fault 2: Firebase Database Exceptions

- Overview
  The server may not be able to access the Firebase database.

- Error correction
  There are two scenarios at which this error could occur, during the initial synchronisation and update writes. Error correction for these two scenarios will be handled separately.

  If the fault occurred in the initial synchronisation, the server will try to retrieve data two more times. If that is unsuccessful, the server will initialise with an empty dictionary.

  If the fault occurred in the update phase, the server will try to execute the update for 5 minutes before giving up. If that operation fails, the update operation is dropped. Since Firebase boasts a record of 99% monthly service uptime, I believe that the 5 minutes timeout for retrying the update operation is sufficient to ensure that the data is updated in the Firebase database.

  The design decisions made for handling these types of error favours implementation simplicity over the potential loss of a few bytes of data.

3.2. Client

3.2.1. Fault 1: Incorrect IP address

- Overview
  This error can be generalised into two different categories, incorrect format and invalid IP address. An incorrect format error refers to the user inputting an IP address that does not match the predefined format for IP addresses. An invalid IP address error happens when the user inputs an ip address that has the correct format but no response is received.

- Error correction
  Mitigating incorrect format error is fairly straightforward. The IP address input string is matched against a regular expression that reflects the desired input. If the user input is invalid, an alert box will pop out to inform the user that the ip address format is incorrect.

  If the user enters a correct IP address, then the client will attempt to connect to the server at that IP address. If no response is received after 6 seconds, the client terminates the connection process and pops an alert box to inform the user that the connection to the user cannot be established. The same procedure takes place when a response with an invalid format is received.

3.2.2. Fault 2: Server is not in operation when a request is sent

- Overview
  This fault occurs when the server is put out of operation when the client is sending a request to it.

- Error correction
  When connection to the server is ended abruptly, an IOException is thrown. Upon catching the exception, the client displays an alert box to inform the user that connection to the server is interrupted and the operation is unsuccessful. Additionally, if the server remains nonoperational, an option is available for the user to change servers in the menu bar.

4. Performance

The system is stress tested against different numbers of concurrent requests to gauge its performance and reliability.

4.1. Test Setup

There are two parts to the test. The first part is the client request simulation. Here, we will consider the case for n={1, 10, 100, 1000, 5000} simultaneous requests. A threadpool of the size of the n is created with each thread executing a word meaning retrieval process.

The second part of the test intends to compare the performance of the server when its threadpool size is changed. Here, we will consider configurations of {1, 5, 10, 15, 20, 25} threads.

We want to obtain the time taken between the client sending the request to the reception of the reply. The test is repeated for three times and the average time of the three runs is used to compare the results.

4.2. Results

TABLE 1
Response time of the server (milliseconds)

Thread Number \ Packet Number	10	100	1000	5000
1	14	23	162	971
5	7	9	108	412
10	11	14	91	269
15	10	10	64	256
20	10	11	95	357
25	8	9	72	434

4.3. Analysis

From the results, we can see that as the number of threads available to the server increases, the response time decreases, particularly for cases where there are a high number of concurrent requests. However, there are signs of performance degradation as the thread number increases. This could be attributed to the overhead associated with assigning tasks to idle threads in the threadpool. Using the results obtained here, the server's threadpool is set to 15.

Throughout the test, the server did not experience an crash faults. Hence, providing us assurance of the stability of the server when faced with a high processing load.

4.4. Test Notes

While effort has been made to simulate the simultaneous request process as realistic as possible, it is important to note that the results presented here may not be reflective of real world applications due to the following reasons.

- Both the simulation and server are hosted in the same computer, hence sharing the same resources. This could affect the computation speed, especially in cases where there are a huge number of concurrent requests.
- Network-related errors were not considered since both programs are hosted in the same machine.

5. Conclusion

Overall, this assignment has been a success. We have developed a client-server system that makes full use of threads to enable concurrent execution. From our testing results, we can be assured of the performance and stability of the system to perform under load. Despite the successes, there are multiple improvements that can be made to the system to improve the performance of the system.

- Implement a cache to replace the hashmap
- Extend the system to working with multiple servers concurrently
- Better algorithms for database synchronisation