COMP90015 Distributed Systems
Assignment 1: Multi-threaded Dictionary Server

Renjie Meng 877396
The University of Melbourne

April 15, 2020

1 Problem Context

In this assignment, a multi-threaded dictionary server which allows multiple clients to *add, delete and search* **concurrently** is designed and implemented. The clients and server are communicated via a **TCP socket** while the server handles multiple clients by a **thread pool** architecture. As for interaction, the **JSON object serialization** is adopted to do message exchange between clients and the server. Based on this design, it is possible for the server to provide query, add and delete service to multiple clients concurrently.

For the failure handling of the system, invalid parameters for starting the programs (i.e. illegal port number, unreachable server address, non-exist dictionary file), invalid input for query (i.e. empty input), illegal operation (i.e. adding an word that exists in the dictionary, deleting a non-exist word, querying for a non-exist word) and communication errors can be detected, handled properly as well as notify the user with proper message on the GUI. Furthermore, the connectivity between the client and server is shown on both client and server side's GUI.

2 Components of the System

2.1 Client

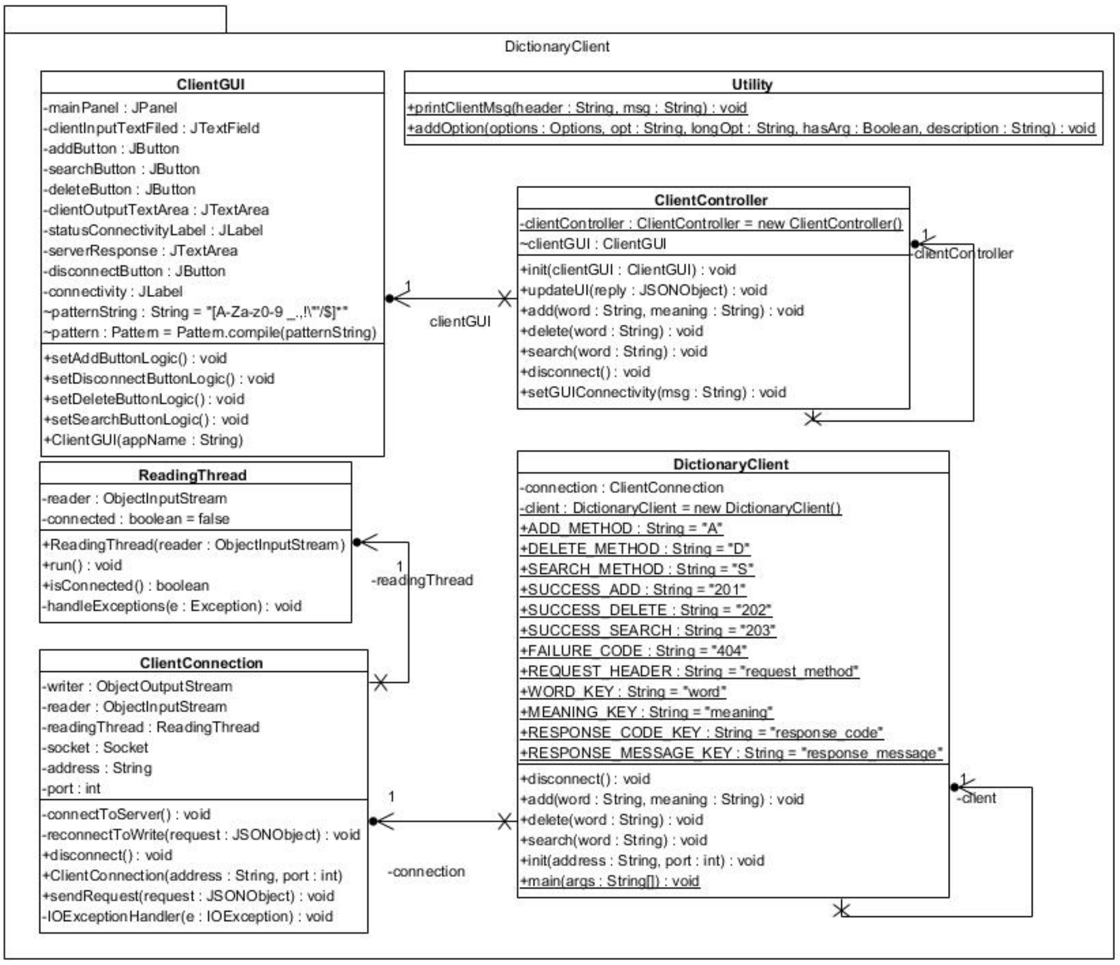
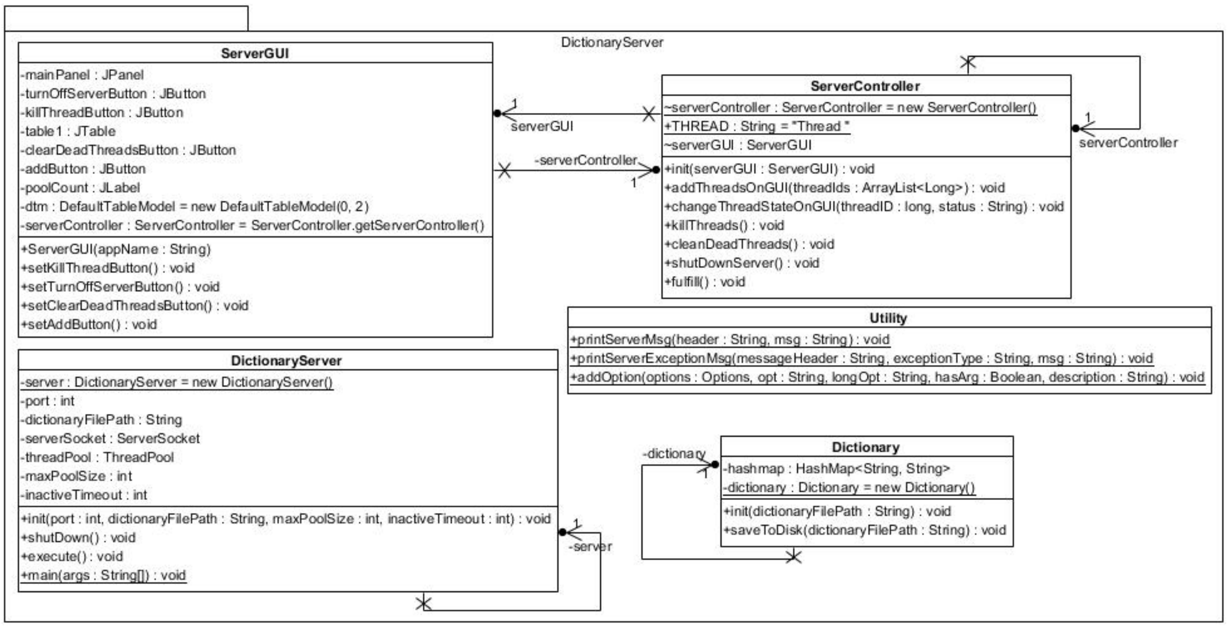

DictionaryClient
ClientGUI
-mainPanel : JPanel
-clientInputTextFiled : JTextField
-addButton : JButton
-searchButton : JButton
-deleteButton : JButton
-clientOutputTextArea : JTextArea
-statusConnectivityLabel : JLabel
-serverResponse : JTextArea
-disconnectButton : JButton
-connectivity : JLabel
~patternString : String = "[A-Za-z0-9 _.,!\"'/$]*"
~pattern : Pattern = Pattern.compile(patternString)
+setAddButtonLogic() : void
+setDisconnectButtonLogic() : void
+setDeleteButtonLogic() : void
+setSearchButtonLogic() : void
+ClientGUI(appName : String)
Utility
+printClientMsg(header : String, msg : String) : void
+addOption(options : Options, opt : String, longOpt : String, hasArg : Boolean, description : String) : void
ClientController
-clientController : ClientController = new ClientController()
~clientGUI : ClientGUI
+init(clientGUI : ClientGUI) : void
+updateUI(reply : JSONObject) : void
+add(word : String, meaning : String) : void
+delete(word : String) : void
+search(word : String) : void
+disconnect() : void
+setGUIConnectivity(msg : String) : void
1
clientGUI
1
-clientController
ReadingThread
-reader : ObjectInputStream
-connected : boolean = false
+ReadingThread(reader : ObjectInputStream)
+run() : void
+isConnected() : boolean
-handleExceptions(e : Exception) : void
1
-readingThread
ClientConnection
-writer : ObjectOutputStream
-reader : ObjectInputStream
-readingThread : ReadingThread
-socket : Socket
-address : String
-port : int
-connectToServer() : void
-reconnectToWrite(request : JSONObject) : void
+disconnect() : void
+ClientConnection(address : String, port : int)
+sendRequest(request : JSONObject) : void
-IOExceptionHandler(e : IOException) : void
1
-connection
DictionaryClient
-connection : ClientConnection
-client : DictionaryClient = new DictionaryClient()
+ADD_METHOD : String = "A"
+DELETE_METHOD : String = "D"
+SEARCH_METHOD : String = "S"
+SUCCESS_ADD : String = "201"
+SUCCESS_DELETE : String = "202"
+SUCCESS_SEARCH : String = "203"
+FAILURE_CODE : String = "404"
+REQUEST_HEADER : String = "request_method"
+WORD_KEY : String = "word"
+MEANING_KEY : String = "meaning"
+RESPONSE_CODE_KEY : String = "response_code"
+RESPONSE_MESSAGE_KEY : String = "response_message"
+disconnect() : void
+add(word : String, meaning : String) : void
+delete(word : String) : void
+search(word : String) : void
+init(address : String, port : int) : void
+main(args : String[]) : void
1
-client

Figure 1: Client Class Diagram

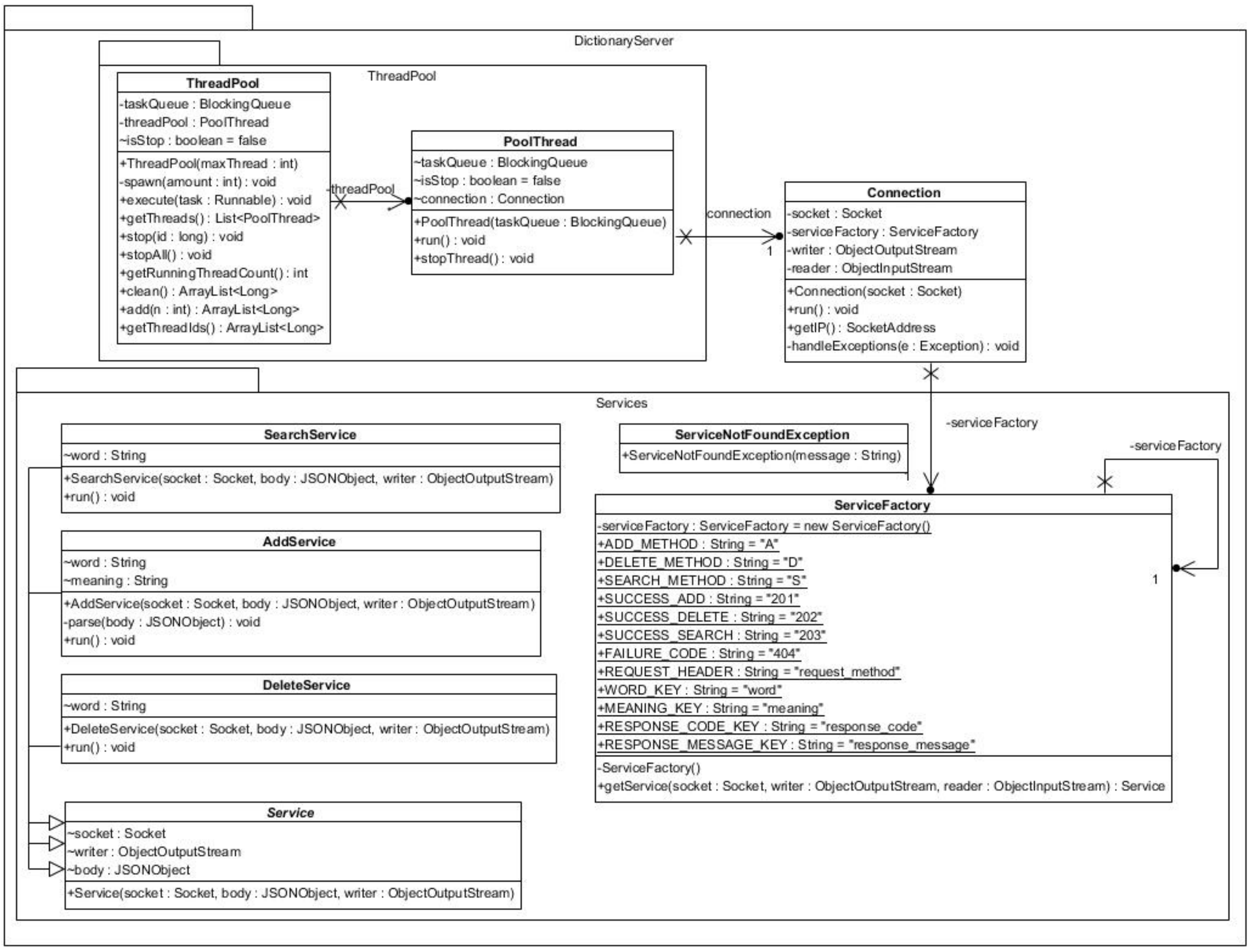
The implementation of the client side follows the MVC architecture (Figure 1). The **ClientGUI** sends user operation to the **ClientController**, then the **ClientController** (singleton) invokes proper method of **DictionaryClient**(singleton) to do the operation. **DictionaryCliet** has a **ClientConnection** which is responsible for writing and reading data. The **ClientConnection** would dedicate a ***second*** thread to run **ReadingThread** which is solely responsible for reading data from the socket.

2.2 Server



DictionaryServer
ServerGUI
-mainPanel : JPanel
-turnOffServerButton : JButton
-killThreadButton : JButton
-table1 : JTable
-clearDeadThreadsButton : JButton
-addButton : JButton
-poolCount : JLabel
-dtm : DefaultTableModel = new DefaultTableModel(0, 2)
-serverController : ServerController = ServerController.getServerController()
+ServerGUI(appName : String)
+setKillThreadButton() : void
+setTurnOffServerButton() : void
+setClearDeadThreadsButton() : void
+setAddButton() : void
serverGUI
1
-serverController
1
ServerController
~serverController : ServerController = new ServerController()
+THREAD : String = "Thread "
~serverGUI : ServerGUI
+init(serverGUI : ServerGUI) : void
+addThreadsOnGUI(threadIds : ArrayList<Long>) : void
+changeThreadStateOnGUI(threadID : long, status : String) : void
+killThreads() : void
+cleanDeadThreads() : void
+shutDownServer() : void
+fulfill() : void
1
serverController
Utility
+printServerMsg(header : String, msg : String) : void
+printServerExceptionMsg(messageHeader : String, exceptionType : String, msg : String) : void
+addOption(options : Options, opt : String, longOpt : String, hasArg : Boolean, description : String) : void
DictionaryServer
-server : DictionaryServer = new DictionaryServer()
-port : int
-dictionaryFilePath : String
-serverSocket : ServerSocket
-threadPool : ThreadPool
-maxPoolSize : int
-inactiveTimeout : int
+init(port : int, dictionaryFilePath : String, maxPoolSize : int, inactiveTimeout : int) : void
+shutDown() : void
+execute() : void
+main(args : String[]) : void
1
-server
-dictionary
1
Dictionary
-hashmap : HashMap<String, String>
-dictionary : Dictionary = new Dictionary()
+init(dictionaryFilePath : String) : void
+saveToDisk(dictionaryFilePath : String) : void

Figure 2: Server Class Diagram 1



DictionaryServer
ThreadPool
ThreadPool
-taskQueue : BlockingQueue
-threadPool : PoolThread
~isStop : boolean = false
+ThreadPool(maxThread : int)
-spawn(amount : int) : void
+execute(task : Runnable) : void
+getThreads() : List<PoolThread>
+stop(id : long) : void
+stopAll() : void
+getRunningThreadCount() : int
+clean() : ArrayList<Long>
+add(n : int) : ArrayList<Long>
+getThreadIds() : ArrayList<Long>
-threadPool
PoolThread
~taskQueue : BlockingQueue
~isStop : boolean = false
~connection : Connection
+PoolThread(taskQueue : BlockingQueue)
+run() : void
+stopThread() : void
connection
1
Connection
-socket : Socket
-serviceFactory : ServiceFactory
-writer : ObjectOutputStream
-reader : ObjectInputStream
+Connection(socket : Socket)
+run() : void
+getIP() : SocketAddress
-handleExceptions(e : Exception) : void
-serviceFactory
-serviceFactory
Services
SearchService
~word : String
+SearchService(socket : Socket, body : JSONObject, writer : ObjectOutputStream)
+run() : void
AddService
~word : String
~meaning : String
+AddService(socket : Socket, body : JSONObject, writer : ObjectOutputStream)
-parse(body : JSONObject) : void
+run() : void
DeleteService
~word : String
+DeleteService(socket : Socket, body : JSONObject, writer : ObjectOutputStream)
+run() : void
Service
~socket : Socket
~writer : ObjectOutputStream
~body : JSONObject
+Service(socket : Socket, body : JSONObject, writer : ObjectOutputStream)
ServiceNotFoundException
+ServiceNotFoundException(message : String)
ServiceFactory
-serviceFactory : ServiceFactory = new ServiceFactory()
+ADD_METHOD : String = "A"
+DELETE_METHOD : String = "D"
+SEARCH_METHOD : String = "S"
+SUCCESS_ADD : String = "201"
+SUCCESS_DELETE : String = "202"
+SUCCESS_SEARCH : String = "203"
+FAILURE_CODE : String = "404"
+REQUEST_HEADER : String = "request_method"
+WORD_KEY : String = "word"
+MEANING_KEY : String = "meaning"
+RESPONSE_CODE_KEY : String = "response_code"
+RESPONSE_MESSAGE_KEY : String = "response_message"
-ServiceFactory()
+getService(socket : Socket, writer : ObjectOutputStream, reader : ObjectInputStream) : Service
1

Figure 3: Server Class Diagram 2

The server implements a thread pool architecture and the classes on the server side are shown in Figure 2 and Figure 3. The sever side also follows the MVC architecture, **ServerGUI** notifies **ServerController** (singleton) to invoke **Dictionary-Server** (singleton). And **DictionaryServer** would ask the **ServerController** to update **ServerGUI**. The **Dictionary** (singleton) represents the words and meanings the server holds, by adopting singleton pattern here, it could provides data consistency.

Inside **DictionaryServer**, there is a **ThreadPool**. The **ThreadPool** would contain a fixed number of **PoolThread** that wait for **Connection** to run. Each time **DictionaryServer** accepts a connection request, it would create a **Connection** object and give it to its thread pool. When the **Connection** is ran by **PoolThread**, it would ask the **ServiceFactory** to read the request and return a proper **Service** object (Strategy pattern), that is the Service may be add, delete or search. Then, the **Service** object would send proper reply to the client, either success response or failure message.

2.3 Socket

The server and client communicate via a **TCP** socket in order to provide reliable data transfer.

2.4 Dictionary File

The dictionary file adopts the **JSON format** since the server uses hash table as its data structure and it is simple to convert between JSON and hash table.

2.5 Message exchange

The message exchanged between the client and server is encapsulated as JSON object. Below are the request and reply format used in the message exchange protocol of this system.

Table 1: Client request format (JSON fields)

request method	S (for Search),A (for Add),D (for Delete)
word	a word for operation
meaning	the definition of a word, only used in the add request type

Table 2: Server response format (JSON fields)

response code	a number indicates the response state (detailed in table 3)
msg	a message (the definition of a word or error messages)

Table 3: Response code reference

404	a general error (the detailed will be describes in the msg field)
201	successfully search a word
202	successfully add a word
203	successfully delete a word

2.6 Overall Interaction of the system

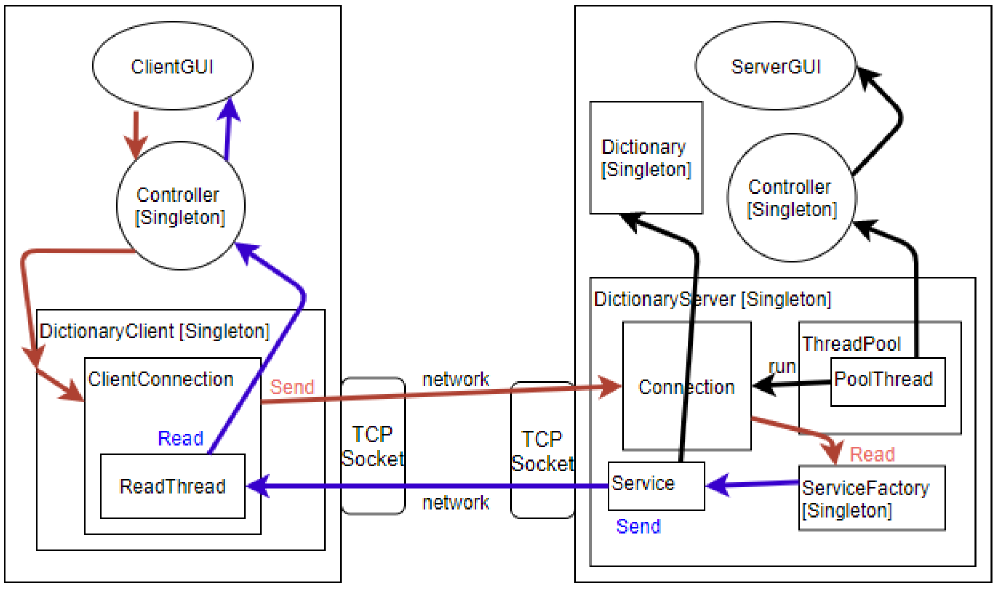


Figure 4: The overall system interaction diagram

As shown in Figure 4, the red arrows represent the message flow from the client to the server. When the **ClientGUI** receives a input, it would notify the **Controller** to do the operation. Then, the **Controller** delegates the operation to the **DictionaryClient**. Then, the **DictionaryClient** would send the request via the **ClientConnection** object via a TCP socket. Then, the **Connection** on the server side would ask the **ServiceFactory** to read the request. Next,the blue arrows represent the message flow from the server to the client. The **Service** object created by the **ServiceFactory** would process the request by using the **Dictionary** singleton. Then, it sends the reply via the TCP socket. The **ReadThread** object in **ClientConnection** would read the server's reply and ask the **Controoler** to update the **ClientGUI**.

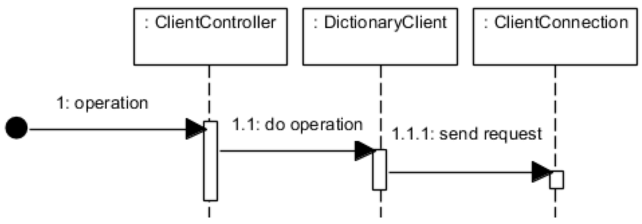


Figure 5: Client send requests.

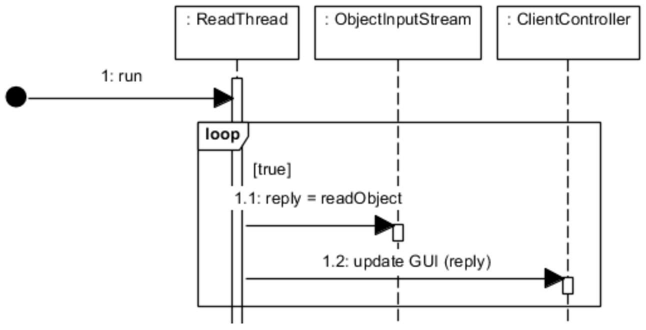


Figure 6: Client reads reply.

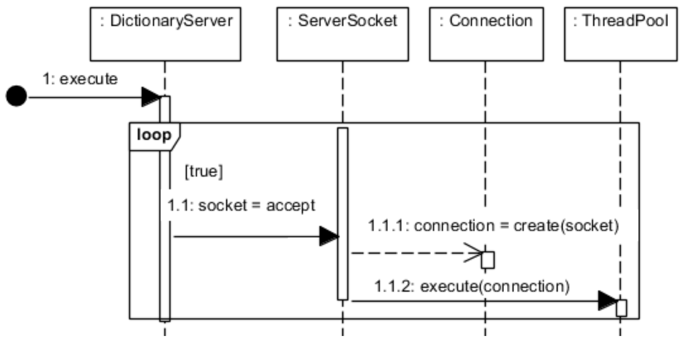


Figure 7: Server execution flow

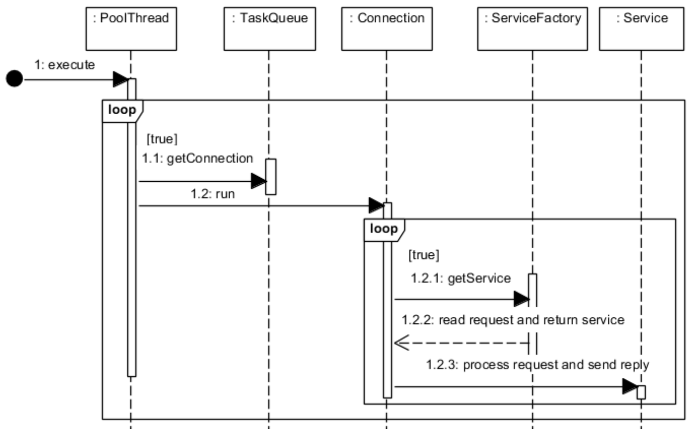


Figure 8: Server read and write

Figure 5 and Figure 6 are the formal interaction diagrams for client side sending and reading. Note that, the client side would dedicates a thread for reading only, that is the **ReadThread**.

Figure 7 shows the execution flow of the **DictionaryServer**. When the **DictionaryServer** is in execution, it would continually listen for new connection request, once a new connection is accepted, it would create a **Connection** object and put it to the **ThreadPool**.

Figure 8 shows how **Connection** is responsible for reading and writing on the server side. Basically, once the **PoolThread** gets and runs the **Connection**, then the **Connection** starts to ask **ServiceFactory** to read request and **Service** to send reply. Note, the **Connection** would close the connection if the client is **inactive** for a certain period of time.

2.7 Command Line Usage

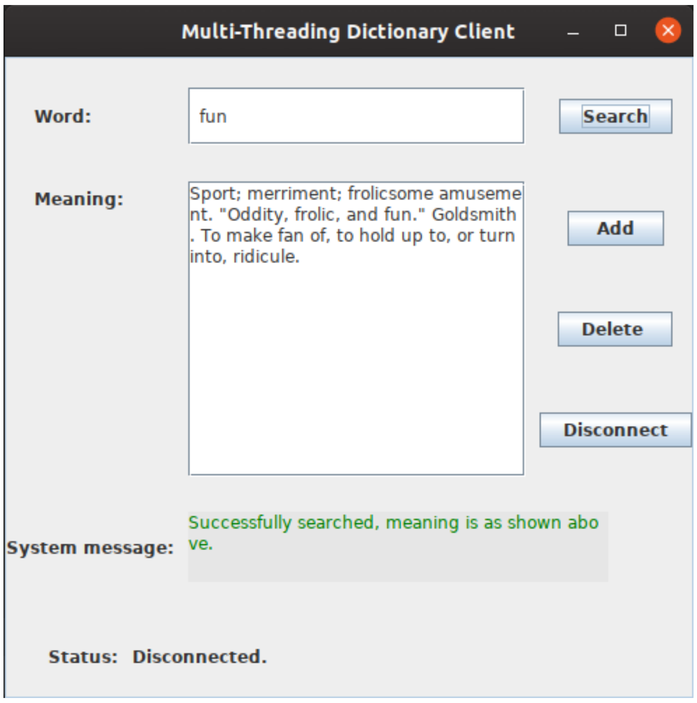
>java -jar DictionaryServer.jar DictionaryServer -p <port> -f <dictionary filepath> -s <pool size> -t <inactive timeout in seconds>

>java -jar DictionaryClient.jar DictionaryClient -a <address> -p <port>

2.8 The GUI

As Figure 9 shown on the next page, the client GUI provides basic functionalities, they are add, delete and search. The system message would be shown in the "System message", the error message would be in red while the success message would be in green. In the bottom, the status shows the connectivity between client and server. Furthermore, the client GUI provides an extra functionality, the user could disconnect from the server actively by pressing the "Disconnect" button.

Figure 10 (next page) shows the server GUI. From this GUI, we could see the number of threads in the pool as well as thread status, a thread could either be idle, interrupted or serving a client. A selected thread could be interrupted from the GUI by clicking the "Kill Thread" button. The interrupted threads are still in the thread pool, by pressing the "Clear Dead Threads", these dead threads would be removed. Furthermore, by pressing the "Fill The Pool" button, it would automatically add threads until the pool is full. Finally, the server could be turned off by click either close or the "Turn off Server" button.



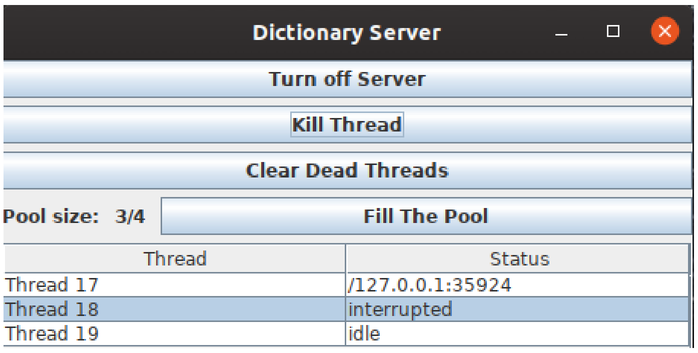
fun

Sport; merriment; frolicsome amusement. "Oddity, frolic, and fun." Goldsmith. To make fan of, to hold up to, or turn into, ridicule.

Successfully searched, meaning is as shown above.

Status: Disconnected.

Figure 9: Client GUI



Pool size: 3/4

Thread	Status
Thread 17	/127.0.0.1:35924
Thread 18	interrupted
Thread 19	idle

Figure 10: Server GUI

3 Critical Analysis of the System

3.1 Challenges

1. *Heterogeneity*

   Due to both the client and server are implemented in Java, the handy JVM provides basic the cross-platform interoperability, such as a client on the Windows could communicate with a server on the Ubuntu successfully.

2. *Openness*

   By following the message exchange protocol defined in section 2.5, a client in different programming language could be implemented. Nevertheless, the client must handle the network connection APIs in that language properly.

3. *Scalability*

   The current design has an obvious limits on scalability, although it offers inactive timeout to disconnect long-time inactive user. As the user amount grows up, the number of connection that a server could serve would be the bottleneck of the system performance. In addition, by following the thread pool architecture, this system removes unnecessary overhead for creating and destroying threads.

4. *Security*

   The current design provides some protection to the threats. Compared to the thread-per-request architecture, if an enemy sends enormous amount of requests to the server, then the server would create huge amount of threads, then the server may run out of memory and crashed in the end. However, with the thread pool architecture, the maximum number of threads are fixed, so the server would not be crashed due to too many threads are created.

3.2 Message Exchange Protocol

The message exchange protocol adopts the format CODE + message, it allows clients and server to exchange message accurately and clearly.

3.3 Synchronization

As leveraging the multi-threading technique in this system, we have to handle concurrency control, particularly synchronization of shared objects.

Firstly, as multiple clients would be able to access and modify the shared Dictionary object simultaneously, proper synchronization is necessary to make sure clients get desired outcome. In this system, the ConcurrentHashMap of java provides thread safe access to a hash table which guarantees the synchronization.

Secondly, in the thread pool, each thread is synchronized on the access to the tasking queue otherwise the same request might be processed multiple times with different responses. The BlockingQueue of java is used as the data structure for holding tasks in this system, it provides thread safe operation on the queue which guarantees synchronization.

3.4 Failure Handling

The current design can handle all failures properly, they are operational and communicational failures.

Table 4: Operation failures

description	note
The dictionary file not exists or not valid (server side)	Load an empty dictionary
Kill the last thread (server side)	not allow and notify the server
invalid input (client side)	notify the client user
already disconnected but still try to disconnect (client side)	notify the client user
invalid operation (client side, i.e. add existing word, delete or search not existing word)	notify the client user

Table 5: Communicational failures

description	note
The server is not available or achievable	notify the user
The connection is closed due to inactive timeout	notify the user
The connection is closed due to thread be killed	notify the user
The connection is closed due to user press disconnect button	notify the server
The connection is closed due to server closed	client would detect it
The connection is closed due to client closed	server would detect it

4 Excellence

1. All information and error messages are shown clearly and concisely on the client GUI.
2. Both client and server follows the MVC architecture pattern, this makes the system easy to understand and easy for further modification, such as add extra functionalities.
3. The description of the system components contains proper diagrams from high-level to detail, which enhance the readability of the report
4. The message exchange protocol and failure handling are clearly presented with tables, which makes the report more concise.
5. Various design patterns are adopted, singleton, factory, strategy are used in the implementation. This makes the system easier to extend and enhance the cohesive of the system. For instance, if a new service is supported, only need to add a new class represents this service and add it to the **ServiceFactory**'s creation logic.

5 Creativity

1. Server would disconnect clients after the inactive timeout is over.
2. Implement my own version of thread pool.
3. The server would save the dictionary file when it is turned off.
4. Lowercasing input at client side.
5. Server side GUI, could view thread status. As well as managing the thread pool such as kill threads and fill thread pool again.
6. The client could actively disconnect from the server.

6 Conclusion

Overall, in this assignment, a simple multi-threading distributed system follows client-server architecture is designed and implemented. Multi-threading is a really powerful technique, but we have to take of synchronization of shared objects, because stochastic result is not desired. Just as *Leslie Lamport* said "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable". The most time-consuming part is failure handling, it needs enormous testing to detecting failures out even in this simple distributed system. Although the distributed system could provides better performance, economics benefits and many other advantages, it is not easy to implement a failure free distributed system.