The DictionaryServer : a client-server application

Abdul Rehman Mohammad
Student ID: 569618

Introduction

The scope of the development project was to develop a client-server architecture for a dictionary server, which allows for concurrent access for multiple clients. Clients are expected to query meaning(s) of a given word, add a new word or remove an existing word from a dictionary file - which is hosted at a server.

A multithreaded dictionary server has been developed, which allows the servicing of multiple clients requesting concurrent access. The communication between the client and server takes place via socket, using the TCP protocol. Each client request is passed into a thread, which provides the feature of concurrent access to the server. The software has been developed in Java.

Design

Dictionary Server

The dictionary server was developed with a Worker Pool architecture. The reason for this choice was to limit the overhead associated with starting multiple threads for each individual client request. The interface ExecutorService caters for the implementation of a Worker pool, and the number of threads in the pool is set by method *newFixedThreadPool()* as shown below,

```
protected ExecutorService threadPool = Executors.newFixedThreadPool(10);
```

Figure 1: The creation of a worker pool designed to hold run 10 threads concurrently.

Each client socket request is wrapped in a runnable and placed in the thread pool, which has a fixed number of threads kept in a queue.When a thread becomes available, a runnable is taken from the queue and executed in a thread. The implementation of this is demonstrated by the following code,

```
this.threadPool.execute(new WorkerRunnable(clientSocket, "Thread Pooled Server"));
```

Figure 2: Implementation of the socket from a request, wrapped in a runnable class and passed to the theadPool

The key advantage of the thread pool is the control over the maximum number of threads running in the server, and consequently reducing the chance of resource depletion. Particularly in the case of concurrent access (which would be applicable to the dictionary server), enforcing

a queue of requests in a pool has greater speed performance as compared to many threads working concurrently. The UML class diagram of the dictionary server is shown in Figure 3.

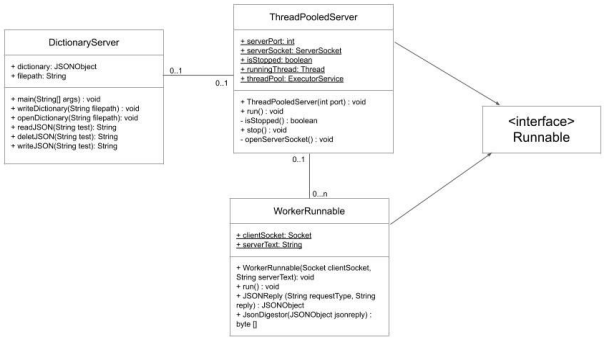

Figure 3: UML class diagram of the dictionary server.

The interaction logic with the client (through the socket) is implemented the ‘run()’ method of the WorkerRunnable class. When the program is run, dictionary file (in a json format) is parsed into a *JSONObject*, derived from the *org.json.simple* library. The advantage of the JSON object is that it is lightweight and the parsing of the object is fast. As JSON objects are written in key-value pairs, they are useful for looking up word requests from the client. The *get()* and *put()* methods from the *org.json.simple* library were utilized for read, write and delete functionalities that were required by the dictionary server.

Dictionary Client

The dictionary client was implemented using the JavaFX library as the graphic user interface (GUI). This was done so by extending the Application class and implementing the Initializable interface. The GUI consists of four scenes: ‘main menu’, the ‘find word’ scene, the ‘add word’ scene and the ‘remove word’ scene.

The ‘main menu’ is shown in Figure 4. It has four buttons: Find a word, Add a word, Delete a word and Exit Program. The ‘Exit Program’ button shuts closes the program, whereas the other three steer the program into request required. Each scene contains a submit button as shown in Figure 5. The submit button, when clicked, leads to the creation of a socket which through which the message is sent to the DictionaryServer.

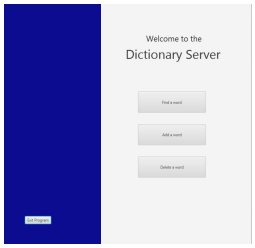

Figure 4: The main menu of the DictionaryClient GUI

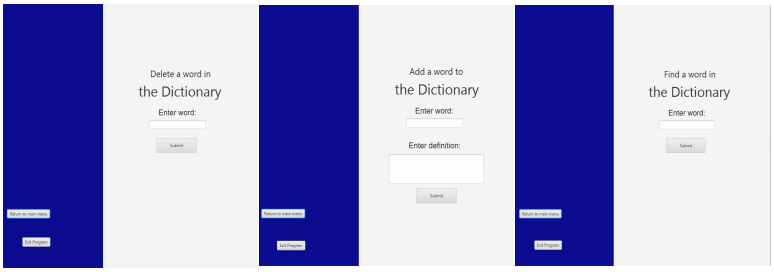

Figure 5: The three scenes: 'Delete a word', 'Add a word' and 'Find a word' shown left to right.

The UML class diagram of the dictionary client is shown in Figure 6 below.

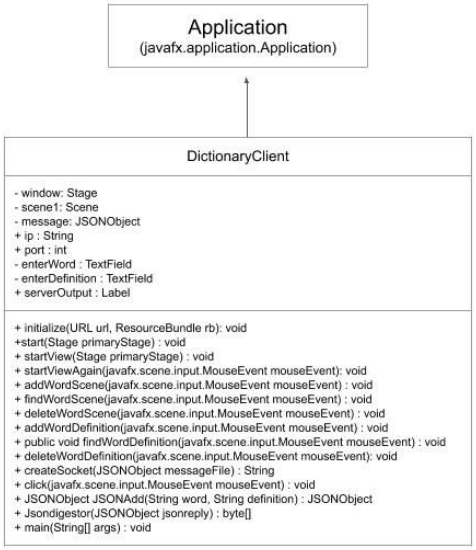

Figure 6: UML class diagram of the dictionary client.

Interaction of Server-Client

The message exchange protocol used for communication between the DictionaryClient and DictionaryServer was JSON (JavaScript Object Notation). The request type, and the content of the request was packaged in the JSON format, retrieving the message through the attribute value pair feature of JSON.

Messages which request to read or delete a word contain two keys, whereas messages which request adding a word contain three keys. The format of the respective JSON messages is shown in Figure 6 below, which are sent from the client to the server.

{"Type":"Read","Word":"champion"}	{"Type":"Delete","Word":"champion"}	{"Type":"Add word","Word":"champion","definition":"the best player"}

Figure 6: Format of the JSON messages sent from the client to the server, based on the nature of the requests.

The format of the message sent from the Dictionary server back to the Dictionary Client, is shown in figure 7. This output is displayed in the Dictionary client GUI by parsing the JSONObject retrieving the value for the attribute “Reply”.

{"Type":"Read","Reply":"1. One who engages in any contest; esp. one who in ancient times contended in single combat in behalf of another's honor or rights; or one who acts or speaks in behalf of a person or a cause; a defender; an advocate; a hero. A stouter champion never handled sword. Shak. Champions of law and liberty. Fisher Ames. 2. One who by defeating all rivals, has obtained an acknowledged supremacy in any branch of athetics or game of skill, and is ready to contend with any rival; as, the champion of England. Note: Champion is used attributively in the sense of surpassing all competitors; overmastering; as, champion pugilist; champion chess player. Syn. -- Leader; chieftain; combatant; hero; warrior; defender; protector."}	{"Type":"Delete","Reply":"Word deleted"}	{"Type":"Add word","Reply":"Word added to dictionary"}

Figure 7: Format of the JSON messages sent from the server to the client, responding to the request of the client

Error handling

The requirement for both DictionaryServer and the DictionaryClient is that exactly two arguments are taken from the command line. This is implemented in the main class for both applications with the following code,

```
if (args.length != 2){
        throw new IllegalArgumentException("Incorrect Number of arguments,"
            + "please only enter two arguments");}
```

Figure 8: Implement of Illegal ArgumentException for incorrect number of arguments passed to the command line.

Since the available ports for use (not reserved) are between 1025 and 65535, any ports specified by the input to be outside this range also prompts errors as shown in the code below,

```
if (port > 65535 && port < 1025){
        throw new IllegalArgumentException("Port number has to be between"
            + "1024 and 65535"); }
```

Figure 9: Implement of Illegal ArgumentException for an incorrect port number passed to the command line.

If the address for connection passed into the command line is not reachable, or is incorrect, then an UnknownHostException is thrown and caught as shown in the code below,

```
catch (UnknownHostException e) {
        System.out.println("The host is not available or could not be"
            + " found. Please re-enter the correct IP address or try"
            + " again later.");
```

Figure 10: Implementation of UnknownHostException for an incorrect hostname passed to the command line, or a bad connection

Error handling has also been implemented for reading and writing the dictionary file. If there is no file to be found a FileNotFoundException is thrown, and caught as shown below.

```
catch (FileNotFoundException e){
        System.out.println("Dictionary file not found."); }
```

Figure 11: Implementation of FileNotFoundException in reading the file, and no file is found

A connect exception is thrown when the client tries to create a socket with the host, when there is no corresponding socket which is open and listening on the server's end. This connection exception is caught with the following code,

```
catch (ConnectException e){
        String response = "Server is not runnning, please try again"
            + " later.";
        return response;  }
```

Figure 12: Implementation of ConnectException caught when a socket is created at the client, but there is not corresponding socket which is listening at the server's end.

Creativity

An additional feature which was implemented was the use of a message digest for the two way communication. This was used to detect any infiltration of the communication channel(s) between the server and client. Along with the JSON object which is transferred through the socket, the hash value of the object (calculated through the SHA-256 hash function), was sent along with every message.

The *MessageDigest* class was imported through the library *java.security*, and the "SHA-256" cryptographic hash function was used. The message digest was performed on the JSON object that was transferred between the client and socket. The method that created the hash value from the JSONObject is shown below.

```
public byte[] JsonDigestor(JSONObject jsonreply){
  try {
     //Message digest added for security
     MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");
     byte[] data1 = jsonreply.toString().getBytes("UTF-8");
     byte[] digest = messageDigest.digest(data1);
     return digest;
  }
  catch (Exception e) {
     System.out.println(e.getMessage()); }
  return "null".getBytes(); }
```

Figure 13: The method used to calculate the hash value of the message, implemented in both the client and server

The hash value is sent in every communication, with the message. The hash value of the message is computed upon receipt at both client and server ends respectively. If there is any manipulation of the message, the hash value of the message computed upon receipt will be different to that sent through the channel. Upon detection of this manipulation, the following message is prompted at both the client and the server, “Security breach in communication, the first message was interrupted and hence the channel is not reliable, please try again later!", after which the socket is closed.

The message digest feature was added to add a degree of authentication between the client-server communication, to verify that the messages received were the original messages sent.

Conclusion

The multithreaded dictionary server, together with the client program, was successfully implemented to query, add or delete certain words which are present in the dictionary file. A worker pool architecture was chosen for the implementation of the multi-threaded server. The TCP protocol for socket communication ensures the reliability offered by the protocol.

A possible improvement to this program could be to use Remote Method Invocation (RMI) between the client and server to invoke remote objects. The server could implement an interface with methods for read, write and delete to the dictionary file - which could be registered in the RMI Name registry. The client would simply lookup the service in the RMI registry and invoke the methods the server has provided access to. This would remove the need for the JSON message exchange protocol, and simplifying the code required to build the application considerably.