2019

Multi-threaded dictionary server

COMP90015: ASSIGNMENT 1

AAYUSH MEHTA 1105081

UNIVERSITY OF MELBOURNE | PARKVILLE

Table of content

Problem context

The project requires the development of a dictionary server that can serve multiple clients at the same time. The server should support concurrent client requests which include:

- Find the meaning(s) of a word
- Addition of a new word
- Deletion of a word

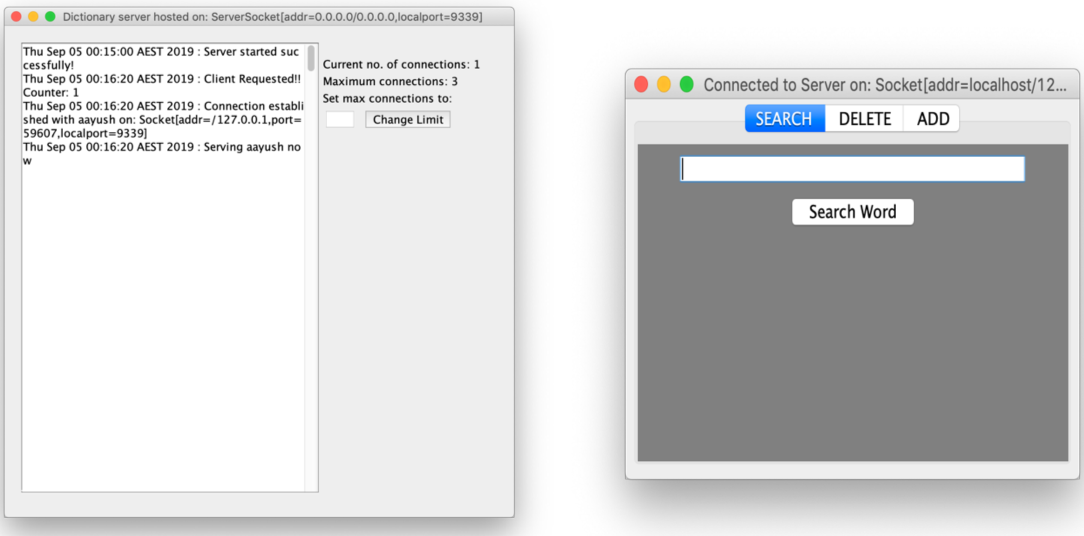
The objective of this project is to demonstrate the implementation of threads and sockets at their lowest level of abstraction for network communication and concurrency.

Solution

The solution of the problem is divided in two parts:

- A server application that is responsible for handling all the operation requests on the dictionary.
- A client application that requests the operations on the dictionary and displays the results

This solution has been implemented using JAVA 1.8, IntelliJ, JAVA Swing, JSON.



Dictionary server hosted on: ServerSocket[addr=0.0.0.0/0.0.0.0,localport=9339]

```
Thu Sep 05 00:15:00 AEST 2019 : Server started suc
cessfully!
Thu Sep 05 00:16:20 AEST 2019 : Client Requested!!
Counter: 1
Thu Sep 05 00:16:20 AEST 2019 : Connection establi
shed with aayush on: Socket[addr=/127.0.0.1,port=
59607,localport=9339]
Thu Sep 05 00:16:20 AEST 2019 : Serving aayush no
w
```

Current no. of connections: 1
Maximum connections: 3
Set max connections to:

Connected to Server on: Socket[addr=localhost/12...

Fig 1. The figure shows server application on the left and the client application on the right.

Architecture and interaction

The solution is implemented using a client-server architecture where a single server can serve requests coming from multiple clients. In this architecture the client(s) send request(s) to the server and the server processes them individually and sends the result back to the respective clients.

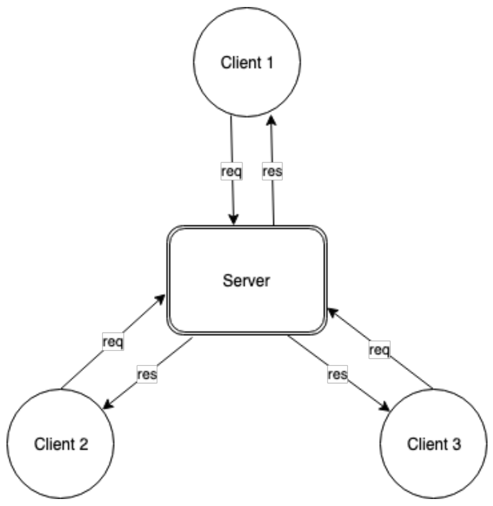


Fig 2. Multi client-server architecture

The architecture need not only satisfy one client but many of them simultaneously. Thus, the concept of threads is used to support multiple clients. The application is developed, based on the assumption that a large number of users will not be using it at the same time. Therefore, in order to process requests from multiple clients, a thread per connection architecture is implemented. A new thread is created for every client that connects to the server. This thread is responsible for all the operations on the dictionary.

The communication in the solution takes place through sockets using TCP (Transmission control protocol). It is a connection-oriented protocol and provides a reliable flow of data between two nodes. It is implemented using sockets that provide and interface for programming networks at the transport layer. The sockets are bound to ports that enable in establishing a path for two-way communication between a server and a client.

Implementation overview

The server side of the application consists of two classes: DictionaryServer and ClientThreadHandler. The DictionaryServer class is used for the creation of the server, loading the dictionary file and creating GUI and, the ClientThreadHandler class is used for client multi-threading for handling the operation requests. The client side of the application contains only one class: DClient that is responsible for the client-side communication and GUI. Figure 3. shows the interaction of the various classes.

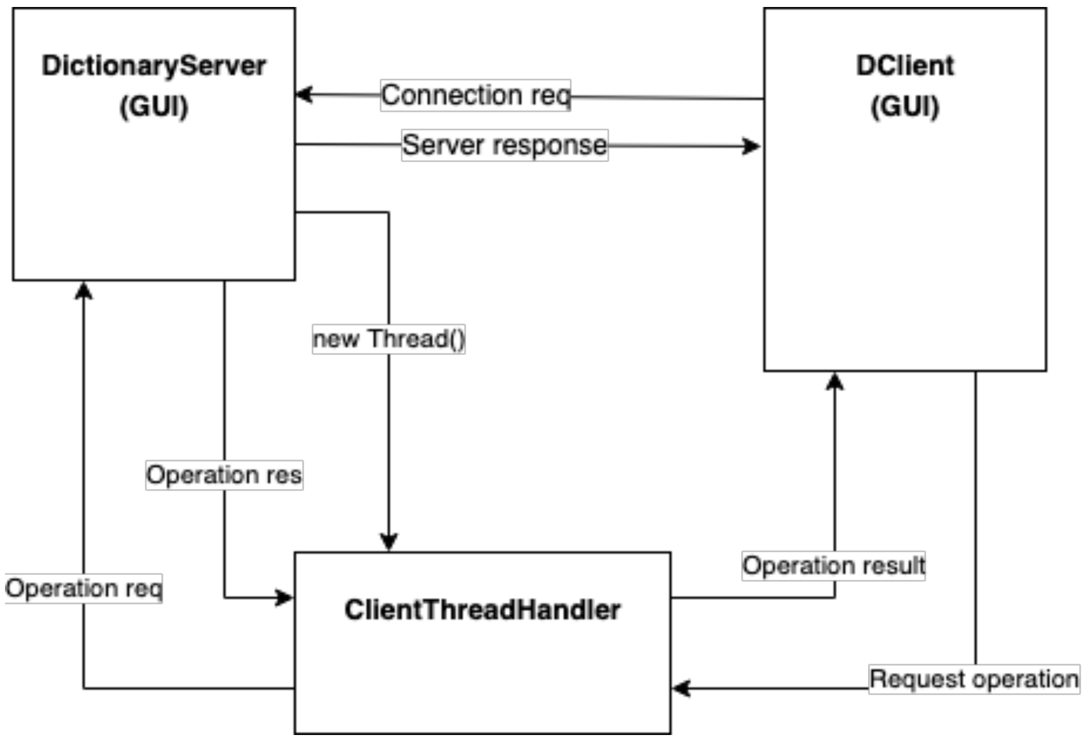


Fig.3 Class interaction diagram

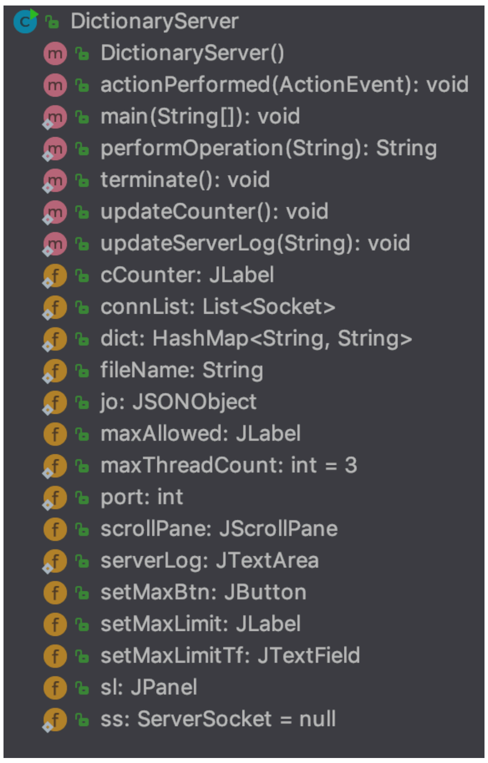
Server-side application

DictionaryServer class: This class is responsible for the following functionalities:

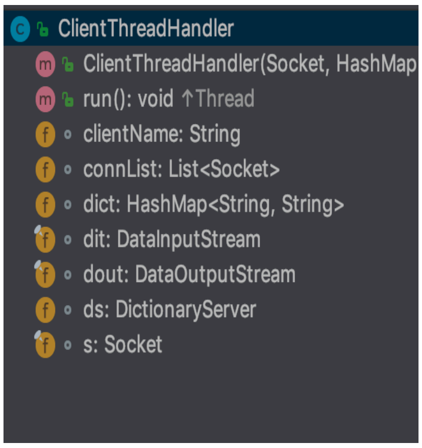
- Loading the dictionary.json file
- Connection establishment on server socket port
- Creating GUI
- Continuous look-out for connection request using a socket object
- Obtaining input and output streams from clients
- Creation of new client handler object on validating the server load
- Dynamically changing the server capacity
- Keep track of the client connections operations happening on the server
- Synchronizing the operations requested by various clients and updating dictionary accordingly
- Log all the session activities on termination of the server
- Fault tolerance

ClientThreadHandler class: This class is responsible for the following functionalities:

- Uniquely identify the client request using a socket, DataInputStream and DataOutputStream
- Check for requests and send a response in a loop
- Call the method of the DictionaryServer class responsible for the operations on the dictionary
- Termination of thread when the client disconnects
- Fault tolerance



```
DictionaryServer
    DictionaryServer()
    actionPerformed(ActionEvent): void
    main(String[]): void
    performOperation(String): String
    terminate(): void
    updateCounter(): void
    updateServerLog(String): void
    cCounter: JLabel
    connList: List<Socket>
    dict: HashMap<String, String>
    fileName: String
    jo: JSONObject
    maxAllowed: JLabel
    maxThreadCount: int = 3
    port: int
    scrollPane: JScrollPane
    serverLog: JTextArea
    setMaxBtn: JButton
    setMaxLimit: JLabel
    setMaxLimitTf: JTextField
    sl: JPanel
    ss: ServerSocket = null
```



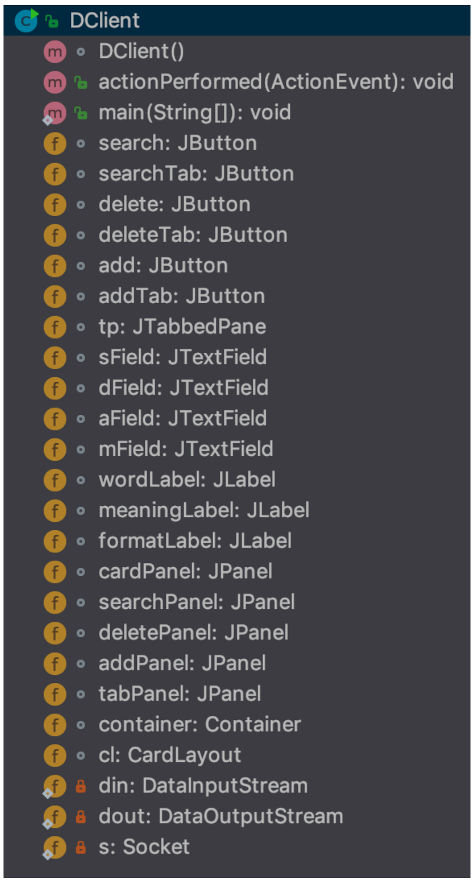
```
ClientThreadHandler
    ClientThreadHandler(Socket, HashMap
    run(): void ↑Thread
    clientName: String
    connList: List<Socket>
    dict: HashMap<String, String>
    dit: DataInputStream
    dout: DataOutputStream
    ds: DictionaryServer
    s: Socket
```

Fig. 4 Class snippet for DictionaryServer.java and ClientThreadHandler.java

Client-side application

DClient class: This class is responsible for the following functionalities:

- Establishing a connection with the server
- Creating DataInput and DataOutput streams to interact with the server
- Creation of the GUI
- Requesting the server for various operations
- Displaying the results
- Fault tolerance



```
DClient
  DClient()
  actionPerformed(ActionEvent): void
  main(String[]): void
  search: JButton
  searchTab: JButton
  delete: JButton
  deleteTab: JButton
  add: JButton
  addTab: JButton
  tp: JTabbedPane
  sField: JTextField
  dField: JTextField
  aField: JTextField
  mField: JTextField
  wordLabel: JLabel
  meaningLabel: JLabel
  formatLabel: JLabel
  cardPanel: JPanel
  searchPanel: JPanel
  deletePanel: JPanel
  addPanel: JPanel
  tabPanel: JPanel
  container: Container
  cl: CardLayout
  din: DataInputStream
  dout: DataOutputStream
  s: Socket
```

Fig 5. Class snippet for DClient.java

Critical analysis

The architecture of this application is designed on an assumption that at any given point of time, there wouldn't be a very large number of active users. Therefore, the application creates a new thread for every connection established and is removed from the connection list when the client is terminated. A functionality to dynamically limit the number of active connections is provided on the server side. This can later be used to scale up or scale down the server by increasing/decreasing the instances of the server application depending on the server load. Creating a new thread for every connection could be a little expensive. Using the worker pool thread is a more efficient way of implementing the server, where we predefine and instantiate a pool of threads which are responsible for all the operations. This way the server does not have to create a new thread for every incoming connection, rather just use the free ones in the pool. But since a big server load isn't expected, a thread per connection will suffice.

A reliable communication is required to communicate between server and the client. The application should make sure that there is no loss of data during communication and, the message is sent and received by both the parties at all times. TCP is chosen over UDP for the implementation of this solution. TCP is a connection-oriented protocol and provides a reliable flow of data. UDP on the other hand is used for broadcasting continuous streams of data without any reliability of successful delivery of messages and hence it wasn't the first choice for the solution. Implementing UDP is possible by adding message delivery guarantee.

JSON file has been chosen as the choice for implementing the dictionary file. JSON has a very simplified notation to store key value pairs, easily-readable and is widely used. The words are stored as the keys and the meanings as values in our file. JSON-simple is package used to parse the contents to and from the dictionary file. JSON is chosen over XML or csv format files. XML has a complex structure as compared to JSON and is not easily comprehended and requires more memory to store a word. CSV on the other hand might take less space to store a word but is not otherwise self-explainable if opened in a text-editor and requires more parsing functionality.

A TreeMap is used as a data structure to store the contents of the dictionary file during the server execution. A treemap is not thread-safe by default which means many threads can operate on treemap at the same instance which can hang the execution of the program. Therefore, all the functions that operate on this treemap are synchronized to maintain the consistency and concurrency. The treemap is used over hashmap because unlike hashmap, the treemap stores the key-value pairs in a sorted order hence making the search faster. Java also provides ConcurrentHasMaps which are thread safe but using it would not have satisfied the requirement to implement threads at the lowest level of abstraction.

Future work

The communication happening between the server and the client is not encrypted. The following work can use some encryption algorithms to encrypt and decrypt the data. Also, the current implication can be improvised to support scaling on a docker platform. On reaching a certain threshold capacity, the docker can deploy a new container containing the server jar and balance the load accordingly.

Excellence

A highly interactive interface has been created at the server and the client side. The design is simple and can be used by anyone without any prior knowledge. A tabbed pane is created at the client side to perform various operations. The server can keep a record of all the session activities. The user is notified of all the errors that might occur during the execution on both client as well as the server application. These errors include:

On the client side -

- Null entries
- Invalid format (number or spaces)
- Pre-existence of words
- Server down
- Server busy
- Invalid hostname/port address

On the server side –

- Invalid/empty file upload
- Port already in use
- Change client limit lower to current no. of connections

Creativity

Following implementations have been done as a part of the excellence component:

- Implemented a server UI that can dynamically set the server capacity for the number of clients that can operate on the dictionary concurrently. The client counter can help scale according to the number of active connections and maximum server capacity.
- A logfile is generated at the server side every time a server shuts down.
- Dictionary file updated every time a delete/add operation occurs, just like it happens on the traditional databases.
- JSON-parser used to read and write to the dictionary files and implementation of a customized parser on the server side to process the requests from the client end.
- Usage of a TreeMap over a regular hashmap to make the search faster.

Deliverables, requirements and deployment

The solution consists of four deliverables:

- DictionaryServer.jar
- DictionaryClient.jar
- dictionary.json
- logs folder

The system should have Java 1.8 installed along with the JRE.

The steps involved in the deployment are:

1. Make sure all files (other than the DictionaryClient.jar) are present in the same directory.
2. Double click on the DictionaryServer.jar and specify the port and the dictionary file in the dialog.
3. Double click on the DictionaryClient.jar and specify the hostname, port and username.

Or

Open terminal in the directory that holds the jar file and type: `java -jar <jar-name>`

Conclusion

A running solution has been implemented that satisfies all the architectural, interaction and functional requirements which are as following:

- Multi-Client server architecture
- Thread and socket implementation using TCP with lowest level of abstraction
- Search, delete and add operations
- Error handling
- Client GUI

Additional features have also been added besides the basic requirements and are described in the previous sections.

Reference:

1. Buyya, R., Selvi, S.T. and Chu, X., 2009. Object-oriented Programming with Java: Essentials and Applications. Tata McGraw-Hill.