COMP90015 Distributed Systems

Assignment 1 Multi-threaded Dictionary

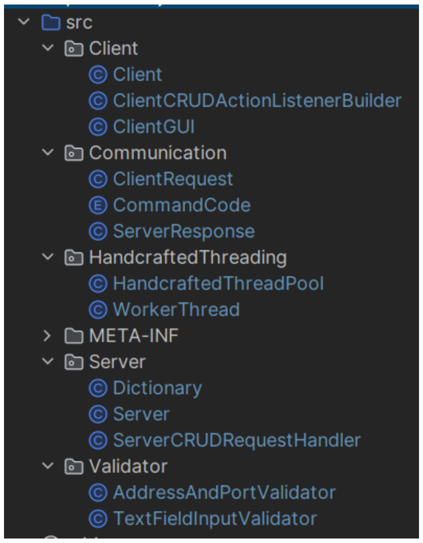
Ziming Wang (1180051)

Contents

1.Discussion of the problem context

The assignment involves creating a java multi-threaded dictionary that follows a client-server architecture. The lowest communication protocol should be at transport layer, e.g., TCP/UDP. The server should have multi-threaded capabilities in order to handle multiple concurrent requests. The dictionary should support word CRUD (CREATE, READ/RETRIEVE, UPDATE, DELETE) operations with reasonable input validations. It should load initial data from a file and modifies the content dynamically while executing. Additionally, an interactive GUI is required for client. Finally, reporting errors properly is essential and the integrity of the client and server should be preserved.

2.Descriptions of the components of system



```
src
  Client
    Client
    ClientCRUDActionListenerBuilder
    ClientGUI
  Communication
    ClientRequest
    CommandCode
    ServerResponse
  HandcraftedThreading
    HandcraftedThreadPool
    WorkerThread
  META-INF
  Server
    Dictionary
    Server
    ServerCRUDRequestHandler
  Validator
    AddressAndPortValidator
    TextFieldInputValidator
```

Figure 1-code directory

Figure 1 demonstrates the components of the systems.

"/Client" and "/Server" package contains corresponding core logic code for client and server.

"/Communication" package contains the communication protocols for client and server. This will be discussed thoroughly in later sections.

"/Validator" package contains the input validator for both server and client to use. For example, both server and client will use the "AddressAndPortValidator.java" to validate the port from command line arguments. They will use "TextFieldInputValidator.java" to validate the user input data including word and meanings.

"/HandcraftedThreading" package contains my handcrafted worker pool multi-threading logics. This will be discussed in later sections.

3.Overall class design and interaction

a. Java Version and Dependency

My java version is corretto-17 (Amazon Corretto Distribution).

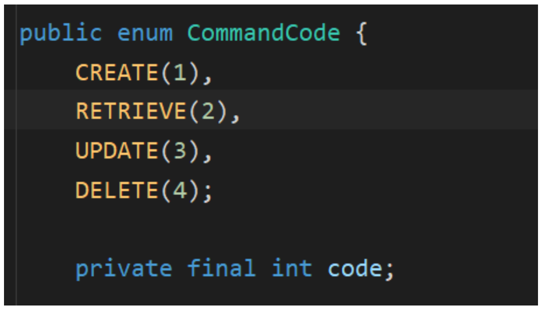
I have an external dependency from org.json to handle json files. The dependency can be downloaded at https://jar-download.com/artifacts/org.json.

b. Transport Layer Protocol

I chose to use TCP socket over UDP as transport layer protocol driven by the need for message reliability. For my application, where reliable message exchange is crucial, the use of TCP ensures that critical messages are handled accurately and efficiently.

c. Message exchange protocol

The server needs to handle CRUD (create, retrieve/read, update, delete) operations. I made an enumeration command code class (Figure 2) for client to indicate the operation type.

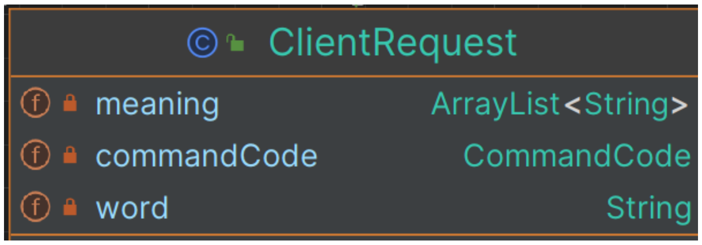


```
public enum CommandCode {
    CREATE(1),
    RETRIEVE(2),
    UPDATE(3),
    DELETE(4);

    private final int code;
```

Figure 2 -ENUM command code (/Communication/CommandCode.java)

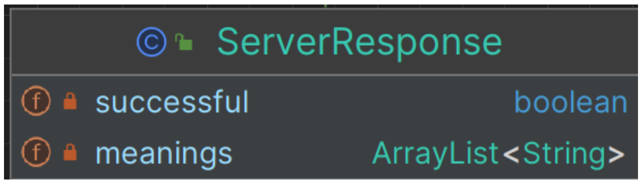
With command code as part of the field, the client can send a ClientRequest object with fields including “commandCode”, “word”, and “meaning” through socket as demonstrated in Figure 3. Java will automatically serialize and deserialize the object. All ClientRequest must have “commandCode” field and “word” field set to indicate what operation they are requesting and the requested word. Additionally, CREATE and UPDATE request must have “meaning” field set. Otherwise, this field should be null.



ClientRequest	
meaning	ArrayList<String>
commandCode	CommandCode
word	String

Figure 3-ClientRequest (/Communication/ClientRequest.java)

Upon receiving the ClientRequest, the server will response with a ServerResponse that contains two fields, “successful” and “meanings” (Figure 4). The “successful” field indicates if the CRUD operation is a success or failure. For example, if the client wants to query a word that doesn’t exist in the dictionary, or wants to create a word that already exists, this field will be set to false to indicate a failure. Furthermore, if the server receives a RETRIEVE request and the operation is performed correctly on the dictionary (a success), then the server will also response with the “meanings” field set as the meaning of the word. Otherwise, the “meanings” field will be null.



ServerResponse	
successful	boolean
meanings	ArrayList<String>

Figure 4-ServerResponse (/Communication/ServerResponse.java)

d. Dictionary and its CRUD workflow

I use json to build the dictionary and perform operations on it supported by an external dependency from org.json. Figure 5 presents the fields and methods of my dictionary. Specifically, I maintain an in-memory JSONObject data structure which is loaded from a json file on disk. Whenever there is CREATE/UPDATE/DELETE action performed, the in-memory JSONObject will be modified accordingly, and the json file on disk will be updated. The RETRIEVE request will only access the in-memory JSONObject to query the meaning of the word, but it won't involve any modification to the json file on disk.

All CRUD methods of the dictionary are synchronized in order to suit the design of concurrency.

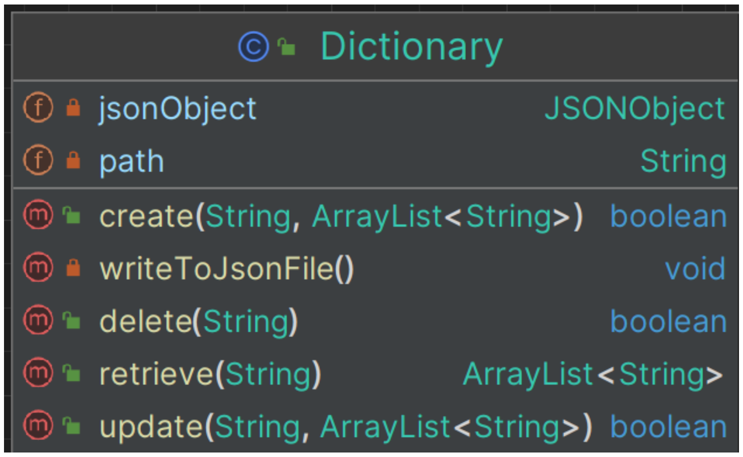

Dictionary
jsonObject JSONObject
path String
create(String, ArrayList<String>) boolean
writeToJsonFile() void
delete(String) boolean
retrieve(String) ArrayList<String>
update(String, ArrayList<String>) boolean

Figure 5-Dictionary (/Server/Dictionary.java)

For example, Figure 6 demonstrates the sequence diagram of a CREATE request workflow for server and dictionary. The "/Server/ServerCRUDRequestHandler.java" receives the request and creates a "/Server/ServerResponse.java" object. Then, it invokes the "handleCreateRequest" method, and performs input validation on word and meanings by invoking the "/Validator/TexFieldInputValidator.java". If all validations are passed, the "create" method of the "/Server/Dictionary.java" will be invoked, and a new entry will be created in the in-memory JSONObject, followed by an update to the json file stored on disk by invoking "writeToJsonFile" method of "/Server/Dictionary.java". The other three type of CRUD requests follow a similar sequence.

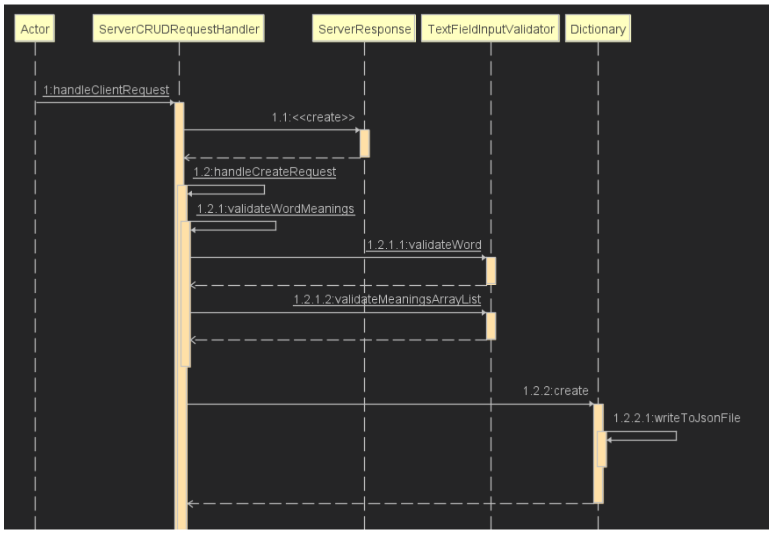


Figure 6-Sequence Diagram, server handles client CREATE requests

4. Highlights and Creativities

a. Own implementation of thread pool for creativity marks

I used my own implementation of thread pool instead of java built-in features. The relevant files can be found in directory "/HandcraftedThreading/HandcraftedThreadPool.java" as shown in Figure 7.

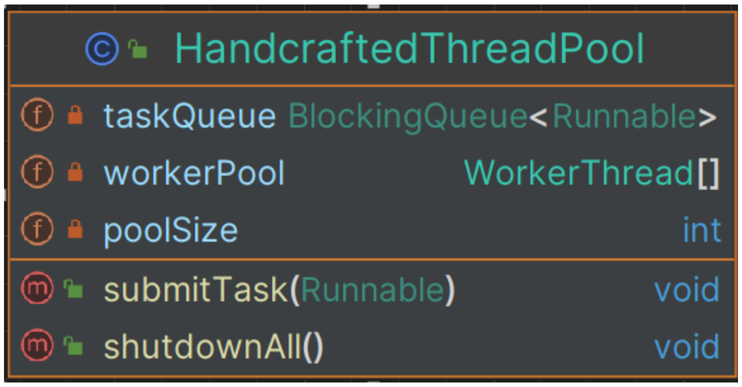

HandcraftedThreadPool
taskQueue BlockingQueue<Runnable>
workerPool WorkerThread[]
poolSize int
submitTask(Runnable) void
shutdownAll() void

Figure 7-(/HandcraftedThreading/HandcraftedThreadPool.java)

It follows a worker pool architecture. Figure 8 demonstrates the sequence of initializing a worker pool. When the "/Server/Server.java" is started, it will create a thread pool (/HandcraftedThreading/HandcraftedThreadPool.java"). The thread pool will create and start a number of worker threads, as indicated by "poolSize". The "taskQueue" field in thread pool is also initialized in order to store all pending tasks. When the client sends a request to the server, the server will set up a new runnable task and "submitTask" to the taskQueue and let it wait for the execution. While the worker threads (WorkerThread.java) will retrieve tasks from this task queue to execute them if available.

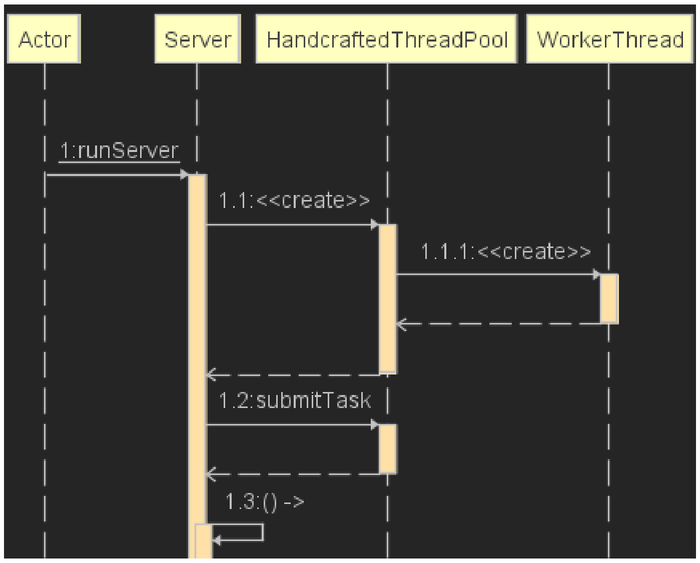


Figure 8-workflow for initializing worker pool

Critical discussion: Using a worker pool architecture for thread management has both advantages and disadvantages. It reduces overhead of creating and destroying threads. It also makes computational resource more controllable and measurable, hence increases scalabilities. Besides, it could prevent the server from being overflowed in a high concurrency situation. However, if the pool size is set to be too small compared with server's capability, then during the time of high traffic, the server's resource may not be fully utilized and the response time may be long. In other words, it lacks of dynamic scaling if we use a fixed pool size. To improve this, a scalable pool size adjustment approach can be used. In addition, it's important to note that for smaller applications with limited number of requests, the administrative overhead of employing a worker pool architecture might outweigh its advantages.

b.Security Highlights

Both client and server should have comprehensive security measures. In general, the ideal use case is that a user uses the client GUI to send a request to the server. Therefore, the client should undoubtfully have comprehensive security measures such as input validations. However, a skillful malicious user could exploit the data package send from the client and write scripts to send request to the server directly. As a result, the security measures in the client GUI are bypassed. Therefore, it is equally important to have same level of security on both the client and server. In my implementation, both client and server use the validators located at "/Validators". For example, for the CREATE request, the client GUI will validate the user's input and deny any invalid format using "TextFieldInputValidator.java". The server will also use it to perform the validation on the ClientRequest payload to double check if the word and meanings are legit. In this way, security is enforced on the whole application.

5. Conclusion

To summarize, this project creates a multi-threaded client-server dictionary with GUI on client side. The server is capable to handle concurrent requests from multiple users supported by my own implementation of worker thread pool. The security is enforced by applying comprehensive validations on both server and client side.