Backend Coding Challenge

1. Introduction

1.1. Rules of Engagement

- The time limit for finishing the coding challenge is 7 days.
- Collibra delivers a JVM-based test client that acts as the final testing framework (downloadable from here).
- Needs to run on a Java SE Runtime Environment 8.
- Within the above boundaries, all libraries or languages can be used.
- The exercise is divided into multiple phases. The test client will test these phases one by one. It is not mandatory (but recommended) to finish all the phases.
- Contact dev-backend-interview@collibra.com in case of questions.

1.2. Test client

The test client can be run by executing:

```
java -jar testclient.jar
```

This will show the different test phases and if they pass or not.
In order to show the debug output for the test client you can use:

```
java -jar testclient.jar --debug
```

In order to start testing from the specific phase you can use `--startFrom <index>` (or `-from <index>`) option. For instance, executing this command:

```
java -jar testclient.jar --startFrom 5
```

will result in client starting from 5th phase.

If you want to run only single, specific phase you can use `--phase <index>` (or `-p <index>`) option. For instance, executing this command:

```
java -jar testclient.jar --phase 4
```

will result in client running only 4th phase.

Note 1: The phases are numbered starting from 1 (which follows the numbering presented in this description).

Note 2: Only phases 1 to 5 can be run independently (that is, using `--phase` argument). In order to check phase 6, you should use `--startFrom` argument, starting from one of the earlier phases, e.g. `--startFrom 5`.

1.3. Deliverable

- The code.
- The build script allowing for compilation of your code, including dependency management.
- The fully built package with all dependencies needed to run on a plain JVM.
- A readme file on how to execute and optional comments.

2. Requirements

2.1. General

- The goal is to implement a simple string protocol communicating over sockets. Below, all messages coming from the server (your code) are presented with **`[SERVER]`** prefix, all client messages are presented with **`[CLIENT]`** prefix.
  Note: The prefixes themselves should not be included in the communication between the client and the server!
- The client will only send and understand standard ASCII characters.
- Every command or response in the protocol is delimited by a newline character, also known as '\n' or ASCII 10.

2.2. Phase 1

For the first phase, setup the TCP socket server application listening on port 50000 and handling the simple commands for greetings and saying goodbye as explained below. Only one session will be active at a certain time.
When the client does not send a message for more than 30 seconds, the connection should timeout and the session should be closed. When closing the session, the server should send the last goodbye message to the client (see later).
Only the current session should be closed. The server should remain ready to accept new sessions.
On connection, the server starts a new session and generates a unique ID for the session. This ID should be a valid UUID.

The server sends a first message back to the client:

```
[SERVER] HI, I AM <session-id>
```

where the <session-id> is replaced by the generated session ID. The client will then respond with:

```
[CLIENT] HI, I AM <name>
```

where <name> can be any string of alphanumeric characters + the dash character (-). The server will respond with:

`[SERVER] HI <name>`

To end the session, the client will send:

`[CLIENT] BYE MATE!`

after which the server will respond with:

`[SERVER] BYE <name>, WE SPOKE FOR <X> MS`

where <name> is the name the client mentioned in its first message and <X> is the number of milliseconds that the session took.
Note: this message should also be sent to the client when the session times out (after not receiving any message from the client for 30 seconds).

> **Example.** If communication lasts for 45 seconds and after that client stops responding (causing timeout) then server should return the session time equal to 75000 (45 seconds of the initial communication + 30 seconds of waiting for timeout, in milliseconds).

When a not supported command is sent, the server should respond with:

`[SERVER] SORRY, I DID NOT UNDERSTAND THAT`

2.3. Phase 2

For the second phase, we will be adding a few commands to be handled by the server. These commands are for building a directed graph. The graph is shared amongst all the sessions during the lifetime of the running server. No persistence on a disk is needed.
These commands can occur 0 or multiple times in any order.

2.3.1. Commands

`[CLIENT] ADD NODE <X>`
Adds a new node with name <X> to the graph.
The response must be:

- If succeeded: `[SERVER] NODE ADDED`.
- If the node already exists: `[SERVER] ERROR: NODE ALREADY EXISTS`.

Note: node names will always contain only alphanumeric characters + the dash character (-).

`[CLIENT] ADD EDGE <X> <Y> <WEIGHT>`
Adds an edge from node <X> to node <Y> with the given <WEIGHT>. The <WEIGHT> is a simple integer indicating the weight of the edge.
The response must be:

- If succeeded: `[SERVER] EDGE ADDED`.
- If a node was not found: `[SERVER] ERROR: NODE NOT FOUND`.

Note 1: edges can occur multiple times (even with the same weight).
Note 2: the weights, will always be non-negative positive integers.

`[CLIENT] REMOVE NODE <X>`
Removes node <X> from the graph.
The response must be:

- If succeeded: `[SERVER] NODE REMOVED`.
- If a node was not found: `[SERVER] ERROR: NODE NOT FOUND`.

Note: this should also cleanup all the edges that are connected to this node.

`[CLIENT] REMOVE EDGE <X> <Y>`
Removes all the edges from node <X> to node<Y>.
The response must be:

- If succeeded: `[SERVER] EDGE REMOVED`.
- If a node was not found: `[SERVER] ERROR: NODE NOT FOUND`.

Note 1: edges are directed and cannot be traversed in the other direction. **Note 1:** no error should be given if there were no edges from node X to node Y. The operation should just succeed.

2.4. Phase 3

For the third phase, we add the calculation of the shortest path between two nodes in the directed graph.

2.4.1. Commands

`[CLIENT] SHORTEST PATH <X> <Y>`

Finds the shortest (weighted) path from node <X> to node <Y>.
The response must be:

- If succeeded: `[SERVER] <WEIGHT>` (where <WEIGHT> is the sum of the weights of the shortest path).
- If a node was not found: `[SERVER] ERROR: NODE NOT FOUND`.

Note: when there is no connection between the two nodes. **Integer.MAX_VALUE** must be returned.

2.5. Phase 4

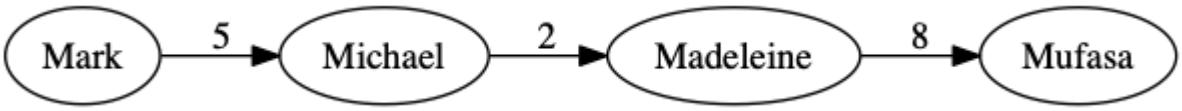
For the fourth phase, we add the calculation to find the nodes which are 'closer' to the given node than the given weight.

2.5.1. Commands

`[CLIENT] CLOSER THAN <WEIGHT> <X>`
Finds all the nodes that are closer to node X than the given weight.
The response must be:

- If succeeded: `[SERVER] <NODES>` (where <NODES> is a comma separated list (no spaces) of the found nodes, sorted alphabetically by name, not including the starting point).
- If a node was not found: `[SERVER] ERROR: NODE NOT FOUND`.

Example: suppose you have this very simple graph:



`[CLIENT] CLOSER THAN 8 Mark`
would return
`[SERVER] Madeleine,Michael`
because Michael is at weight 5 from Mark and Madeleine is at weight 7 (5+2) from Mark.

2.6. Phases 5 & 6

For the 5th and 6th phase, the test client will open multiple sessions at the same time, all adding

and removing nodes and edges.
At the end (6th phase) the client will do some random checks with the 'shortest path' and 'closer than' commands to check the consistency of the graph (also multiple calls in parallel).