eCommerce Systems / Professor H Roumani

PROJECT A

Executive Summary

In this project you will create a number of multithreaded **TCP** services as building blocks of a micro-service architecture. Using a custom (non-HTTP) protocol, these services will expose a number of foundational concepts, such as basic networking, json/xml marshaling, database access, inter-service calls, stateful vs stateless, and cloud computing. You will later use these services to build a shopping-cart web application. In addition, this project will address scalability issues such as multithreading, connection pooling, and throttling, as well as certain security concerns such as vulnerabilities and authentication.

The Environment

You will use the Prism workstations in the lab sessions and during the tests (i.e. laptops not allowed). In your very first lab session, issue the command:

```
4413Go
```

It will install the *4413Go* development environment in your home directory. In it, you will find all the needed libraries under `4413/lib` (so put it in your buildpath for development and on your classpath for deployment); all the needed softwares under `4413/pkg`; and all the needed tools under `4413/bin` (so you may want to add it to your path). For your home machine, you will need to have the *Java JDK*, the *Eclipse IDE for Enterprise Java Developers*, and a *telnet* client (`nc` on Mac) installed. In addition, you will need to download the *4413Go* environment and store it under your user directory.

The Services

This Project asks that you develop and test six micro services with the following functional specifications:

1. **Geo:** Given two points on Earth, this service returns the geodesic distance between them. This service will be used later by a shopping cart application to estimate the drone delivery time. Each point is specified using its latitude (t) and longitude (n) expressed as signed decimal degrees with East of Greenwich being a negative longitude and south of the Equator being a negative latitude. The distance (in km) between two such points is given by:

   ```
   12742 * atan2[sqrt(X), sqrt(1-X)]
   where X = sin²[(t2-t1)/2] + Y * sin²[(n2-n1)/2]
   and Y = cos(t1) * cos(t2)
   ```

   Note that the four coordinates must be in radians so start by multiplying each by `π/180`. Test your code by computing a known distance (you can for example use a map app and pick two points connected by a straight road). Note that we are estimating the *drone* delivery time here, but in a later project, we will estimate the *driving* time by taking the road network and the current traffic volume into account.

2. **Auth:** Given a username and password, this service authenticates these credentials and returns "OK" or "FAILURE" accordingly. This service will be used later by the shopping cart application to authenticate users. Authentication is done by adding a long *salt* to the password; using a cryptographic function to hash the result; and then repeating the process *count* times. More on this in lecture. The CLIENT table in the Sqlite3 database stores the salt, count, and hash of each user. The table adopts PBKDF2 (Password-Based Key Derivation Function 2) to perform the hash, which is the current best practice. An API for computing PBKDF2 is provided in the hr4413 library (in 4413/lib) through the following method in the `g.Util` class:

   ```
   public static String hash(String password, String salt, int count) throws Exception
   ```

   If the username is not found in the table, or if found but the computed hash differs from the stored one, return "FAILURE"; otherwise, return "OK".

3. **Quote:** Given the ID of a product and a return format (json or xml), this service looks up the ID in the Product table in the `hr` schema of the Derby database and returns the product's ID, name, and price in the desired format. If the ID is not found, the return should have "<ID> not found" as ID, an empty name, and 0.0 as price. Use GSON and JAXB to serialize and marshal, and use *"id"*, *"name"*, and *"price"* as identifiers for the json elements and XML nodes. Create a *"Product"* bean to facilitate the transformation (more on this in lecture).

4. **Loc:** Given an address anywhere in the World, this service returns a JSON object representing its specs; most importantly, its latitude and longitude. Use the Google's map API* (https://maps.googleapis.com/maps/api/geocode/json?) and supply the address and your API key to perform this lookup. The URL class in the Java library has a convenient `openConnection` method that creates an HTTP socket (similar to your service's TCP socket) through which you can communicate with the Google API.

5. **Stateful:** This "pedagogical" service delegates to the stateless *Geo* service in a stateful manner to shed light on session management. It receives the coordinates of the first point in one request and the coordinates of the second point in another. It needs to somehow link the two request (despite the multithreading nature of the service) and then supply all four real numbers to *Geo* and return its response.

6. **Gateway:** This "pedagogical" service sheds light on the challenges involved in building an API Gateway. It adopts the HTTP protocol so you can test it with a browser using the URL: http://host:port/SRV?p1=v1&p2=v2... where SRV is either *Geo* or *Auth* or *Quote* or *Loc* and where v1, v2, ... are the parameters of the SRV service. The job of this service is thus to discover the needed service (extract its name from the request and find its IP and port if alive); perform inter-protocol transformation; invoke the service; and then return its response.

Service Implementation

Use the design patterns, methodologies, and hints demonstrated in lecture in order to speed up development and learn best-practice approaches.

- Each of the 6 micro services must have its own class and its own port. This way, they can be deployed in one container, in different containers, in different VMs, or even on different physical hosts.
- Name the service class `Geo`, `Auth`, `Quote`, `Loc`, `Stateful`, or `Gateway`, and put it in the `service` package.
- The service class must extend `Thread` and must have a main method.
- The service class must act as a server with a TCP transport and must bind to either localhost or to the loopback address selected thru the command-line arg: `l` or `p`.
- The server's port is chosen through a second argument. If zero, it indicates that the port can be any available port.
- The server must be multithreaded with a new thread per connection.
- The server (custom) protocol for the first 5 services is text-based, case-sensitive, and line oriented. The request must be a single line (EOL terminated) with its tokens whitespace delimited. The response is also text-based and EOL terminated.
- For the sixth *Gateway* service, the protocol is http. Hence, the request is multi-line and begins with the line:

  ```
  GET /SRV?p1=v1&p2=v2 HTTP/1.x
  ```

  and ends with an empty line. The response begins with the two lines:

  ```
  HTTP/1.1 200 OK
  Content-Type: text/plain
  ```

  followed by an empty line and then the return of the service.
- The service return must be "Do not understand <request>" if the incoming request is malformed.
- Use standard out for logging.
- Log "Connection from <ip:port>" upon receiving a request.
- Log "Closing <ip:port>" upon closing a connection.
- Close the connection with the client after issuing the response.
- Handle exceptions per client by logging the exception message and closing the connection with that client.
- Bundle all six classes in one jar file named `projA.jar` for production deployment. If stored in the 4413 folder then service X can be launched using a command like this:

  ```
  java -cp "projA.jar:lib/*" service.X p 0
  ```

Testing & Deployment

Do all your tests using the loopback network interface. Once all is well, switch to the localhost address and test by deploying your services on one workstation and accessing it (via "`telnet <ip> <port>`") from a different workstation in the lab. (All the lab workstations are on the same LAN.) For production-style deployment, you can use the *red* server by simply issuing "`ssh red.eecs.yorku.ca`" before you launch. In addition, it is highly instructive to deploy on AWS. To that end, create an EC2 instance of the EECS4413 AMI on AWS* and launch your services on it. This way, you can access them from any machine in the lab or at home or from your phone. Note that port 4413 is open in that instance so if one of your services is listening on it then it can be reached from anywhere. All the above applies as-is on your home environment (but for Windows, you would need to activiate the telnet Windows feature in the control panel or use PuTTY).

Testing & Deployment

Do all your tests using the loopback network interface. Once all is well, switch to the localhost address and test by deploying your services on one workstation and accessing it (via "`telnet <ip> <port>`") from a different workstation in the lab. (All the lab workstations are on the same LAN.) For production-style deployment, you can use the *red* server by simply issuing "`ssh red.eecs.yorku.ca`" before you launch. In addition, it is highly instructive to deploy on AWS. To that end, create an EC2 instance of the EECS4413 AMI on AWS* and launch your services on it. This way, you can access them from any machine in the lab or at home or from your phone. Note that port 4413 is open in that instance so if one of your services is listening on it then it can be reached from anywhere. All the above applies as-is on your home environment (but for Windows, you would need to activiate the telnet Windows feature in the control panel or use PuTTY).

Persist Your Work

Follow these steps to persist / backup your work:

- Right-click your project (in the Project Explorer) and select *Export*.
- In the *General* category, select *Archive File*.
- Accept the defaults but uncheck the `lib` folder if present to keep the archive file small (less than 100K). Specify a location/name for the archive file.

Now upload the created zip file to the course cloud so you can use it during tests. (You can also upload it to your Google Drive, DropBox, S3, or some other cloud service). If you later need to restore this backup into a fresh workspace on a different host, do this:

- Download the archive file and store it somewhere.
- Launch Eclipse in a new workspace.
- Right-click anywhere in the Project Explorer and select *Import*.
- In the *General* category, select *Existing Projects into Workspace* (rather than *Archive file*!).
- In the *Select archive* box, point to your archive file, and click Finish.
- Edit the *Build Path* of your project by deleting all the "missing" jars from its *Libraries* tab and adding all the jars in the `4413/libs` directory. (This step is needed because the location of the jars may be different in the new host.)

Try the above procedure end-to-end (by using two machines; by switching to a different workspace on the same machine; or by deleting your workspace after the zip file has been created). Make sure you are comfortable backing up and restoring your course project. Do not wait until the day of the test to learn how to do this. Do not delay the backup until the work is done! Do it often (at least after every release). If you are familiar with `github` then do use it but you still need to upload to the course cloud and to practice the above backup/restore procedure.