Distributed Systems
– 2023/24 –
(some practical exercises)

---

1. Unzip `jsockets.zip` and inspect the four example applications provided as Java packages under `ds.examples.sockets`:

- `basic` - a very basic client-server, client sends text to server, server writes received text, supports only one client connection;
- `calculator` - another client-server, client sends one of four types of requests for arithmetic calculation, server computes result and replies, client writes received result, supports only one client connection;
- `calculatormulti` - similar to the previous one but multiple client requests are allowed and each is executed in a thread created for that purpose;
- `peer` - a single peer that can act as a client or as a server, networks of these peers are symmetric as all nodes feature the same functionality, the server performes arithmetic operations as in the examples above.

Place yourself at the top directory (`jsockets/`) and compile the `basic` project as:

```
$ javac ds/examples/sockets/basic/*.java
```

run the application with the commands (for the server):

```
$ java ds.examples.sockets.basic.Server localhost
```

and (for the client):

```
$ java ds.examples.sockets.basic.Client localhost serverPortNumber
```

Note that the `serverPortNumber` argument is printed by the Server as it starts. Study the code to understand how the example works. You should start with the `main` functions of the client and the server and follow the flow henceforth.

2. Repeat the above exercise for the remainder of the examples provided.

3. Rewrite the `calculatormulti` example so that it implements a service that performs the following operations on strings:

```
int       length(String);
boolean   equal(String, String);
String    cat(String, String);
String[]  break(String, char);
```

Notice how the client-to-server messages have a different structure and content fields with respect to the original program. The same goes for the server-to-client replies as the returned values have different types.

4. Use the Poisson Generator package provided in the file `poisson.zip` to write a version of the `peer` example for which each client within a peer randomly (according to a Poisson distribution with an average of 5 events per minute) sends a request (`add`, `sub`, `mul` or `div` with uniform random numbers) to another peer.

5. Extend the previous example so that the target peer for each calculator request may itself be randomly chosen. What implication does this have on the knowledge of the network by each peer? Try your solution using a static fully connected network of 4 peers.

6. This exercise shows how to implement a `calculatormulti`-like client-server application using Google's Remote Method Invocation. Start by downloading the gRPC software package from `www.grpc.io/` and install it on your machine. Then download `grpc.zip` from Moodle and unzip it.

```
$ unzip grpc.zip
$ ls
grpc
$ cd grpc
$ ls
calculator.proto
calculatorClient.java
calculatorServer.java
```

The three files in the `grpc/` folder contain the implementation of the application. To build it, you must start by copying `calculator.proto` - that contains the abstract protobuffer specification of the service and the data structures involved - to the `proto` directory in the gRPC installation:

```
$ cp grpc/calcultor.proto  grpc-java/examples/src/main/proto/
```

Now, copy `calculatorClient.java` and `calculatorServer.java` to a new directory `calculator` that you create in the gRPC package:

```
$ mkdir grpc-java/examples/src/main/java/io/grpc/examples/calculator
$ cp grpc/calculatorClient.java
     grpc-java/examples/src/main/java/io/grpc/examples/calculator
$ cp grpc/calculatorServer.java
     grpc-java/examples/src/main/java/io/grpc/examples/calculator
```

Now edit the file:

```
$ nano grpc-java/examples/build.gradle
```

and add the following entries for `CalculatorClient` and `CalculatorServer`. Compare how this is done for other examples in the gRPC package.

```
task calculatorServer(type: CreateStartScripts) {
    mainClass = 'io.grpc.examples.calculator.CalculatorServer'
    applicationName = 'calculator-server'
    outputDir = new File(project.buildDir, 'tmp/scripts/' + name)
    classpath = startScripts.classpath
}

task calculatorClient(type: CreateStartScripts) {
    mainClass = 'io.grpc.examples.calculator.CalculatorClient'
    applicationName = 'calculator-client'
    outputDir = new File(project.buildDir, 'tmp/scripts/' + name)
    classpath = startScripts.classpath
}
```

These entries allow the `gradle` tool used by gRPC to build the client and server applications automatically. Now, you can build the client and the server:

```
$ cd grpc-java/examples
$ ./gradlew clean
$ ./gradlew installDist
```

Finally, while still in the same directory, you can run the server and the client by executing the following commands:

```
$ ./build/install/examples/bin/calculator-server
```

for the server, and:

```
$ ./build/install/examples/bin/calculator-client
```

for the client. Note that the example uses `localhost` as the default location for the client and server and the server's port number is 50051. You can easily set up another port if you wish. Find out how.

7. Warming up exercise for the practical assignment. Create an application in which two peers exchange a token forever. You should run the peers as (first peer) `$ peer m2` and (second peer) `$ peer m1`. Note that this implements a special case of the first exercise of the practical assignment. Now add a calculator server and make the peers stop the token exchange when they want to send a calculation to the server. In this case, you should run the peers as (first peer) `$ peer m2 server server-port` and (second peer) `$ peer m1 server server-port`. If you can do this, you have just solved the first exercise for 2 peers. All you have to do is think about how to generalize the solution for 5 peers, as requested.