Distributed Systems
– 2024/25 –
(some practical exercises)

Basic Java Sockets Applications

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 argument `serverPortNumber` 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 when compared to the simple calculator program. The same goes for the server-to-client replies as the returned values have distinct types.

The Poisson Distribution and Time Evolving Processes

4. The Poisson distribution is used to express the probability that a given number of events occur in a time interval assuming a constant average rate. Many real-world processes follow this distribution, e.g., radioactive decay, photons hitting a sensor, etc. The distribution is also important in simulation allowing us to program agent interaction with a known average rate over time. Download the Poisson Generator package provided in the file `poisson.zip` and the simple example of use `UsePoisson.zip`. Try the example and understand how it works. Then adapt this code to write a version of the `peer` example for which each client within a peer randomly (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 also be randomly chosen. What implication does this have on a peer's knowledge of the network? Try your solution using a static fully connected network of 4 peers.

Example of a Remote Procedure Call (RPC) Framework

6. 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/calculator.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.

```
createStartScripts('io.grpc.examples.calculator.CalculatorServer')
createStartScripts('io.grpc.examples.calculator.CalculatorClient')
```

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. After thoroughly understanding how the previous example works, implement a client and a server in gRPC for the interface described in exercise 3.

Warmup for the Practical Assignment

8. Create an application for which two peers exchange a token forever. Run the peers as:

```
$ java Peer localhost 22222  // (first peer)
$ java Peer localhost 33333  // (second peer)
```

The numbers indicate the ports associated with each peer's "token forwarding"service. To start the forwarding write a small program that connects to one of the peers, injects the token, and exits. Now add a calculator server (you can use `CalculatorMultiServer` from the first exercise without changes) and make the peers stop the token exchange when they want to send a calculation to the server. A peer "flips a coin"to decide whether or not it will interact with the server. In this case, you should run the peers as:

```
$ java CalculatorMultiServer localhost 44444
server running at localhost, port 44444
$ java Peer localhost 22222 44444 // (first peer)
$ java Peer localhost 33333 44444 // (second peer)
```

Note that the peer with the token has exclusive access to the server, effectively implementing distributed mutual exclusion. How would you generalize the solution for a ring with `n` peers? Doing this will get you very close to solving the first problem in the practical assignment.

Warmup for the Practical Assignment

9. Create an application for which two peers periodically synchronize their contents. Run the peers as:

```
$ java Peer localhost 22222 33333  // (first peer)
$ java Peer localhost 33333 22222  // (second peer)
```

The numbers are the ports associated with the local and the remote peer's synchronization services, respectively. Each peer keeps a set of numbers with a uniform distribution between 0 and 1. A new number is added to the set using a Poisson process with a frequency of 4 per minute. Each peer contacts the other once per minute (also a Poisson process) to perform the set synchronization (think about how you can implement this operation, e.g., look for a "merge"method in the Java API for sets). Immediately after this operation, their sets of numbers will be the same. Print the set of each peer after a synchronization to visualize this process. Use a thread to handle the synchronization requests and another to generate the numbers and manage the local set. Generalize the solution so that: (a) a given peer may be connected to multiple peers; (b) when a peer initiates a push-pull operation, the target peer is chosen randomly from the set of its neighbors. This small example will make it easier to develop the solution for the second problem of the practical assignment.