Introduction to Distributed Systems
WT 17/18

Assignment 2 – Part II (programming)

Submission Deadline: Sunday, 19.11.2017, 23:59

- *Submit the solution via ILIAS (only one solution per group).*
- *Respect the submission guidelines (see ILIAS).*

---

4 Java Sockets [14 points]

Implement a Java program for a client (cf. project `CalcSocketClient`) that uses a remote service for calculations in a distributed system. The remote service offers four functions:

```
// Add a value
public void add(int value);

// Subtract a value
public void subtract(int value);

// Multiply by a value
public void multiply(int value);

// Return the result of the current calculation
public int getResult();
```

The specification (cf. `ICalculation`) and implementation (cf. `CalculationImpl`) for this remote service are provided as project stubs within each subtask. Notice that you are not allowed to edit the specification or implementation of this service!

The remote service is made available by a Java server program via socket communication (cf. project `CalcSocketServer`). Every client connecting to the server must be served in an individual session. Use connection-oriented communication and the Multi-Threaded Multi-Session Server concept. The server must handle multiple sessions concurrently and may not place any restriction on the number of concurrently served clients.

The following asynchronous text-based protocol for the communication must be used:

1. All messages are case-insensitive and framed by the two characters '<' and '>'. Bytes outside a message have to be ignored.
2. A valid message starts with the character '<', followed by a two digit integer indicating the total message length "00", a colon ':', an actual message content "...", the ending character '>' (e.g., a valid message is `<07:OK>`)
3. A valid message content is a single integer value, a calculation operator "ADD", "SUB", "MUL", or "RES", or an info operator "RDY", "OK", "ERR', or "FIN"'.
4. Multiple message contents can be concatenated in a valid message content by separating each content by one or more whitespace. Leading and trailing whitespace characters have to be ignored. For instance, a valid message content from a client is `<16:ADD 23 9 -1>`, `<13:  SUB 10 >`, or `<19:MUL 2 SUB 13 >`.

5. After a client has connected to the server, the server sends the content "RDY" to the client, if it is ready to receive requests from this client (e.g., `<08:RDY>`).
6. For each received message, the server sends immediately a response with the content "OK". After the server has processed every message content from a received message, it sends the content "FIN" to the client.
7. If a valid content equals to the operator "ADD", "SUB", or "MUL", the server-side calculation operator is changed to ADDing, SUBtracting, or MULtiplying by values.
8. If a valid content equals to an integer value, the value is – based on the current calculation operator – added to, subtracted from, or multiplied with the result.
9. Each valid content (e.g, "ADD", "SUB", "MUL", or an integer value) is acknowledged to the client with the content "OK" followed by a single whitespace character and the valid content.
10. If a valid content equals to the operator "RES", the current calculation result is sent to the client with "OK" followed by a single whitespace character, the operator "RES", another single whitespace character and the current calculation value.
11. Each invalid content (e.g., "DS1") is acknowledged with "ERR" followed by a single whitespace character and the invalid content. Invalid contents are not processed.
12. A session is only terminated when the client closes the connection.

To implement your solution, download the stubs for projects and class files including the implementation of the `CalculationImpl` service from the course materials website (cf. `a2q4.zip`). Notice that you are not allowed to change the code of any other project than the `CalcSocketServer` or the `CalcSocketClient`! The project `TestCalc` is an exemplary testing project for your implementation.

a) [8 points] First, implement the Java server program to offer this remote service with the above-specified protocol for clients. To this end, implement the `run`-method of the `CalcSocketServer` class. The `run`-method should start for each individual client session a specific client thread.

   Hint: If you want to test your server implementation before implementing the client, you can utilize a program like ***`netcat`****.*

b) [6 points] Second, implement the Java client program that uses the protocol to compute a calculation. To this end, implement the `connectTo`, `disconnect`, and `calculate` method of the given `CalcSocketClient`. All computations during a single run of the program must be performed in a single session. If an error or invalid request occurs during the calculation, the client must report an error.

5 Remote Method Invocation [6 points]

Implement a Java server program (cf. project `CalcRMIServer`) and a Java client program (cf. project `CalcRMIClient`) for the remote calculation service using Java Remote Method Invocation (RMI). The server must serve each client with an individual session *(Hint: Use the factory pattern, for which an interface is provided, cf. `ICalculationFactory`)*. The server program must handle calculation sessions concurrently and may not place any restriction on the number of clients served concurrently.

To implement your code, download the stubs for projects and class files from ILIAS (cf. `a2q5.zip`) including the implementation of the `CalculationImpl` service, the Stub-Project `CalcRMIServer`, and the Stub-Project `CalcRMIClient`. Note the following:

- The RMI client may only have access to the code of the service specification (Interfaces, Project Session, ...), not to the code of the implementation (Classes, Project SessionImpl, ...).
- To use the factory pattern, you must implement `CalculationImplFactory`. This is the only file in projects `CalcRMICalculation` and `CalcRMICalculationImpl` that you may edit.

To become familiar with Java and RMI, read the following information:

- Tutorial presentation on RMI (cf. course materials on ILIAS)
- Java Tutorial: `http://java.sun.com/docs/books/tutorial/index.html`
- Sockets in Java: `http://java.sun.com/docs/books/tutorial/networking/sockets/index.html`
- RMI in Java: `http://java.sun.com/docs/books/tutorial/rmi/index.html`
- Java Doc API: `http://java.sun.com/javase/7/docs/api/`
- Thinking in Java: `http://mindviewinc.com/Books/`

Hint: In case you get a class not found exception take a look at `http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/enhancements-7.html`
A possible solution is to specify the codebase when starting the rmiregistry:

```
rmiregistry -J-Djava.rmi.server.codebase=file:///<path-to-remote-classes>/
```