Distributed Programming II

A.Y. 2016/17

Assignment n. 3 – part b)

All the material needed for this assignment is included in the *.zip* archive where you have found this file. Please extract the archive to the same working directory where you have already extracted the material for part a) and where you will work. In this way, you should have the xsd and documentation pdf developed in part a) in the `[root]/xsd` and `[root]/doc` folders respectively.

This second part of the assignment consists of three sub-parts:

1. Write a simplified Java implementation (using the JAX-RS framework) of the web service designed in part a). This simplified implementation must implement only functionalities 1, 2 (with the exclusion of removal), 3, and 4 specified in part a). Accordingly, some of the operations that are possible in your design may be disabled in this implementation. The web service will be packaged into a war archive that can be deployed to Tomcat. This will be done by the ant script `[root]/build.xml`, which includes a target named `package-service` that builds and packages the service. The name of the service packaged in this way will be "NffgService", and its base URL will be http://localhost:8080/NffgService/rest (the WADL description will be available at the standard location, i.e. http://localhost:8080/NffgService/rest/application.wadl).

The service does not have to provide data persistency but has to manage concurrency, i.e. several clients can operate concurrently on the same service without restrictions. When deployed, NffgService must be initialized with an empty data set (i.e. no NFFGs and no policies).

The service must exploit the Neo4JXML service (the one already used for Assignment 2; a new war is provided in this assignment with an updated implementation) in the following way: when a new NFFG is added (point 2 of the functionality list), the new NFFG is stored into the Neo4JXML service (if this is not possible for some reason, e.g. service not reachable, the add operation must fail); when a reachability policy has to be verified (point 4 of the functionality list), this is done by exploiting the Neo4JXML service, as done in Assignment 2 (if the operation that Neo4JXML is requested to perform fails, the corresponding operation will fail in NffgService). In order to enable the storage of multiple NFFGs in Neo4JXML, the way NFFGs are mapped to Neo4J nodes will be slightly different from the one adopted in Assignment 2: for each NFFG, in addition to the nodes and relationships used in Assignment2, the service will create another node that represents the NFFG itself, with a property "name" that stores the name of the NFFG and the label "NFFG"; this node will be connected to all the nodes that represent the nodes of that NFFG by means of relationships named "belongs". Note that only the NFFGs are stored in the Neo4JXML service, while policies are not (of course, they are stored by NffgService). NffgService has to read the actual URL of Neo4JXML as the value of the system property `it.polito.dp2.NFFG.lab3.NEO4JURL`. If the property is not set, the default value to be used is http://localhost:8080/Neo4JXML/rest.

The service must be developed entirely in package `it.polito.dp2.NFFG.sol3.service`, and the corresponding source code must be stored under `[root]/src/it/polito/dp2/NFFG/sol3/service`.

Write an ant script that automates the building of your service, including the generation of any necessary artifacts. The script must have a target called `build-service` to build the service. All the class files must be saved under `[root]/build`. Customization files, if necessary, can be stored under `[root]/custom`, while XML schema files can be stored under `[root]/xsd`. Of

The library must load information about NFFGs and policies from the web service at startup. The library must implement all the interfaces and abstract classes defined in package `it.polito.dp2.NFFG`. The classes of the library must be entirely in package `it.polito.dp2.NFFG.sol3.client2` and their sources must be stored in `[root]/src/it/polito/dp2/NFFG/sol3/client2`. The library must include a factory class named `it.polito.dp2.NFFG.sol3.client2.NffgVerifierFactory`, which extends the abstract factory `it.polito.dp2.NFFG.NffgVerifierFactory` and, through the method `newNffgVerifier()`, creates an instance of your concrete class implementing the `NffgVerifier` interface. The actual base URL of the web service to be used for getting the information must be obtained by reading the `it.polito.dp2.NFFG.lab3.URL` system property. If this property is not set, the default URL http://localhost:8080/NffgService/rest/ must be used.

Update your `sol_build.xml` file, so that the target named `build-client` also builds your second client. All the class files must be saved under `[root]/build`. You can assume that the Tomcat server is running with NffgService deployed when the `build-client` target is called (of course, the target may fail if this is not the case). Customization files, if necessary, can be stored under `[root]/custom`.

The client library classes developed in points 2 and 3 must be robust and interoperable, without dependencies on locales. However, these classes are meant for single-thread use only, i.e. the classes will be used by a single thread, which means there cannot be concurrent calls to the methods of these classes.

Correctness verification

Before submitting your solution, you are expected to verify its correctness and adherence to all the specifications given here. In order to be acceptable for examination, your assignment must pass at least all the automatic mandatory tests. Note that these tests check just part of the functional specifications! In particular, they only check that:

- the submitted schema is valid (you can check this requirement by means of the Eclipse XML schema validator);
- the implemented web service and clients behave as expected, in some scenarios: after calling an NNFG add or policy add operation (by means of your client1), the information about NFFGs and policies returned by your service and received by your client2 is consistent with the performed operations.

The same tests that will run on the server can be executed on your computer by issuing the following command

```
ant -Dseed=<seed> run-tests
```

Note that this command also deploys your service and the Neo4JXML service. However, before running this command you have to start Tomcat and Neo4J on your local machine, as explained for Assignment 2.

Other checks and evaluations on the code will be done at exam time (i.e. passing all tests does not guarantee the maximum of marks). Hence, you are advised to test your program with care (e.g. you can also test your solution by running your service and then test it by using your clients in different scenarios and by means of other general-purpose clients).

Submission format

A single *.zip* file must be submitted, including all the files that have been produced. The *.zip* file to be submitted must be produced by issuing the following command (from the `[root]` directory):

```
ant make-final-zip
```

In order to make sure the content of the zip file is as expected by the automatic submission system, do not create the *.zip* file in other ways.