# Distributed Programming II

A.Y. 2012/13

## *Assignment n. 6*

### Part 1

All the material needed for this assignment is included in the *.zip* archive where you have found this file. Please extract the archive to an empty directory (that will be called `[root]`) where you will work.

This assignment is about the development of a set of web services for the PJS system. The web services have to fulfill the following specifications:

- Each *execution host* offers a web service that can be used by the *master host* for dispatching and controlling of jobs. This web service implements the interface described by the WSDL document `[root]/wsdl/PJSDispatch/PJSDispatch.wsdl`.
- As it can be seen looking at the description of the PJSDispatch web service, when the *master host* dispatches a job to an *execution host*, the *master host* specifies a callback URI in the request. This URI must be the address of a web service offered by the *master host*. This web service has to be used by the *execution host* when the dispatched job terminates, in order to notify the *master host* about job termination. On notification, the *execution host* gives the exit-status code (an integer) and the standard-output string produced by the dispatched job.
- The web service(s) offered by the *master host* must enable

  (a) read access to information about hosts and jobs in the cluster (for administration clients);

  (b) job submission and subsequent job control (for submission hosts clients).
- The information about hosts and jobs made available by the *master host* for reading is the same that can be retrieved from the `it.polito.dp2.PJS` Java interface (see Assignment 1).
- When a *submission host* submits a new job to the cluster, the submission host has to specify a command-line string and a standard-input string for the job (these parameters will be passed on to an *execution host* by the *master host* on dispatch), and the job group (the default group is used if the group is not specified).
- Job control includes suspending, resuming and killing single jobs or job groups.

Assuming the PJS system has to be developed using standard (SOAP) web services, design the web service(s) provided by the MASTER host, according to the specifications given above. Specify the designed web services in a WSDL document and save it under `[root]/wsdl/PJSMaster/`.

The web services must be designed so as to be robust and so as to make the completion of the specified operations both efficient and easy to be used by clients.

The WSDL document must include comments (`documentation` elements) that explain the semantics of the operations and that briefly explain the main design choices.

### Part 2

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

**A.** Implement a simplified version of the web service(s) designed in Part 1, and write a Java class with a `main` that publishes the implemented web service(s) using the JAX-WS `Endpoint` class.

The simplified version of the web service(s) is required to implement only information retrieval and job submission, while all the other operations can remain unimplemented. Moreover, the management of job groups and job termination have not to be implemented: the service(s) will simply use the default group for all jobs regardless of the group specified on submission and the service for receiving termination notifications must not be implemented. However the job state has to be managed. The web service that includes information retrieval operations must be published at the URL `http://localhost:8182/PJSMaster` and the WSDL document designed in Part 1 must be available at the URL `http://localhost:8182/PJSMaster?wsdl`. If other URLs are necessary, they can be freely chosen but they must use the same host, and the ports immediately following 8182 (i.e. 8183, 8184, ...).

All classes must be developed in package `it.polito.dp2.PJS.sol6.service` and must be stored under `[root]/src/it/polito/dp2/PJS/sol6/service/`. In addition to the Java classes, the solution must include an *ant* script, saved as `[root]/sol_build.xml`. This script must have a target `build-server` that automatically builds the service code starting from the source code and from the WSDL document and a target `run-server` that starts the server. **Important:** the script must define `basedir="."` and all paths used by the script must be under `${basedir}` and must be referenced in a relative way starting from `${basedir}`. If other files are necessary (e.g. for customization), they should be saved in the directory `[root]/custom`.

The service must operate correctly even when accessed by multiple concurrent clients, while it can be assumed that the *PJSDispatch* web services are accessed only by the *master host*. The management of persistency is not required.

The service implementation must get the information about the available execution hosts by reading an XML document that follows the XML schema `[root]/xsd/PJSHostsInfo.xsd`. As it can be seen looking at the schema, the XML document includes, for each host, a URI. It can be assumed that the *PJSDispatch* web service is available at a URI obtained by appending the "/PJSDispatchService" suffix to the URIs found in the XML document. This web service has to be contacted in order to dispatch jobs to the corresponding execution host. A sample XML document written according to the schema is available in the file `[root]/xml/execHosts.xml`. The *ant* script provided with this assignment can be used to start the *PJSDispatch* web services for the hosts listed in `[root]/xml/execHosts.xml`. This can be done by issuing the command

```
ant startPJSDispatch
```

At startup, your web service must be in a state where there are no managed jobs. When dispatching jobs, the web service implementation must try to balance the number of jobs on the managed execution hosts. This means that the service will try to dispatch a new job to one of the hosts that have the minimum number of running jobs at that time. However, if for any reason the dispatch is not possible on the selected host, the job must be dispatched to another host.

**B.** Implement a client similar to the one of assignment 5 that reads the information about hosts, jobs and job groups from the web service(s) implemented in sub-part A and makes this information available through the Java interfaces and abstract classes defined in package `it.polito.dp2.PJS`.

As in assignment 5, the client must take the form of a library. The library must be written entirely in package `it.polito.dp2.PJS.sol6.client1` and must include a factory class named `it.polito.dp2.PJS.sol6.client1.ClusterFactory`, which extends the abstract factory `it.polito.dp2.PJS.ClusterFactory` and, through the method `newCluster()`, creates an instance of your concrete class that implements the `Cluster` interface. On initialization, the instance must contact the service and download all available information, throwing a `ClusterException` if this is not possible. This information has to be cached in the instance. Differently from what done in previous assignments, whenever a method of the `Cluster`

interface is called, the instance must try to contact the web service in order to download and return up to date information, which will also be used to update the local cache. If this is not possible (e.g. service not responding), the instance must return the previously cached information.

All sources must be stored under `[root]/src/it/polito/dp2/PJS/sol6/client1/`. The actual URL used by the library to contact the service must be customizable: if the `it.polito.dp2.PJS.sol6.URL1` system property is set, its value must be used as the actual URL. Otherwise, the library must use the URL specified in the WSDL.

The script `[root]/sol_build.xml` must include a target named `compile-client-1` that compiles the client, including the generation of any necessary artifacts. The target must save all `.class` files under the directory `${basedir}/build`, which is included in the classpath when the tests are executed. Customization files, if necessary, can be stored under `[root]/custom`.

**C.** Implement another client for the designed web service(s) that can submit jobs. This client must be a Java class named `it.polito.dp2.PJS.sol6.client2.Client2` with default constructor, that implements the interface `it.polito.dp2.PJS.lab6.Submit` (provided in source form). This interface includes just one method for submitting a job. When this method is called on the client object, the client object must contact the service and try to submit a job with the specified parameters. The submit method must return after the submitted job has been created and dispatched. If dispatching is impossible or some other error occurs, the submit method must return the error code (-1).

The actual URL used by the client to contact the service must be customizable: if the `it.polito.dp2.PJS.sol6.URL2` system property is set, its value must be used as the actual URL. Otherwise, the client must use the URL specified in the WSDL.

The script `[root]/sol_build.xml` must include a target named `compile-client-2` that compiles the client, including the generation of any necessary artifacts. The target must save all `.class` files under the directory `${basedir}/build`, which is included in the classpath when the tests are executed. As in previous points, source and custom files must be stored under the `[root]/src` and `[root]/custom` directories respectively.

## 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:

1. the submitted WSDL is valid (you can check this requirement by means of the Eclipse WSDL validator);
2. the implemented web service(s) submit jobs correctly to the known hosts;
3. the information about hosts, jobs and job groups is coherent with the operations performed (the hosts are the ones specified in `[root]/xml/executionHosts.xml`, the jobs are the ones that have been submitted since the start of the service and they are all in the default group).

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.

The *.zip* file of this assignment includes a set of tests that check points 2. and 3. They are like the ones that will run on the server after submission. Tests can be run by the ant script included in the *.zip* file, which also compiles your solution by calling your ant script. Before running these tests you must have started the *PJSDispatch* web services and your server (which must be in its initial state). Therefore, the full list of *ant* commands to be issued for testing is:

```
ant startPJSDispatch
ant –f sol_build.xml run-server
ant runFuncTest
```

Note that these commands must be issued in this order but from different shells (or from the *ant* console in Eclipse).

## Submission format

A single *.zip* file must be submitted, including all the files that have been produced. The *.zip* file to be submitted can be produced issuing the following command (from the `[root]` directory, here split over multiple lines for readability):

```
$ jar -cf lab6.zip sol_build.xml wsdl/* custom/*
src/it/polito/dp2/PJS/sol6/service/*.java
src/it/polito/dp2/PJS/sol6/client1/*.java
src/it/polito/dp2/PJS/sol6/client2/*.java
```

Note that the *.zip* file **must not** include the files generated automatically.