JPPF Manual

Table of Contents

1 Introduction

1.1 Intended audience

This manual is intended for developers, software engineers and architects who wish to discover, learn or deepen their knowledge of JPPF and how it works. The intent is also to provide enough knowledge to not only write your own applications using JPPF, but also extend it by creating add-ons and connectors to other frameworks.

1.2 Prerequisites

JPPF works on any system that supports Java. There is no operating system requirement, it can be installed on all flavors of Unix, Linux, Windows, Mac OS, and other systems such as zOS or other mainframe systems.

JPPF requires the following installed on your machine:

- Java Standard Edition version 1.5 or later, with the environment variable JAVA_HOME pointing to your Java installation root folder
- Apache Ant, version 1.7.0 or later, with the environment variable ANT_HOME pointing to the Ant installation root folder
- Entries in the default system PATH for JAVA-HOME/bin and ANT_HOME/bin

1.3 Where to download

All JPPF software can be downloaded from the [JPPF downloads page].

We have tried to give each module a name that makes sense. The format is *JPPF-x.y.z-<module-name>.zip*, where:

- x is the major version number
- y is the minor version number
- z is the patch release number - it will not appear if no patch has been released (i.e. if it is equal to 0)
- <module-name> is the name given to the module

1.4 Installation

Each JPPF download is in zip format. To install it, simply unzip it in a directory of your choice.

When unzipped, the content will be under a directory called *JPPF-x.y.z-<module-name>*

1.5 Running the standalone modules

The JPPF distribution includes a number of standalone modules or components, which can be deployed and run independantly from any other on separate machines, and/or from a separate location on each machine

These modules are the following:

- application template: this is the application template to use as starting point for a new JPPF application, file JPPF-x.y.z-application-template.zip
- driver: this is the server component, file JPPF-x.y.z-driver.zip
- node: this is the node component, file JPPF-x.y.z-node.zip
- administration console: this is the management and monitoring user interface, file JPPF-x.y.z .admin-ui.zip
- multiplexer: this is the TCP multipler that routes all traffic through a single port, file JPPF-x.y.z .multiplexer.zip.

These can be run from either a shell script (except for the multiplexer) or an Ant script. The ant script is always called "*build.xml*" and it always has a default target called "*run*". To run any of these modules, simply type "*ant*" or "*ant run*" in a command prompt or shell console. The provided shell scripts are named start<*Component*>.<*ext*> where *Component* is the JPPF component to run (e.g. “Node”, “Driver”, “Console”) and ext is the file extension, “bat” for Windows systems, or “sh” for Linux/Unix-like systems.

2 Tutorial : A first taste of JPPF

2.1 Required software

In this tutorial, we will be writing a sample JPPF application, and we will run it on a small grid. To this effect, we will need to download and install the following JPPF components:

- JPPF application template: this is the JPPF-x.y.z-application-template.zip file
- JPPF driver: this is the JPPF-x.y.z-driver.zip file
- JPPF node: this is the JPPF-x.y.z-node.zip file
- JPPF administration console: this is the JPPF-x.y.z-admin-ui.zip file

---

Note: *"x.y.z" designates the latest version of JPPF (major.minor.update). Generally, "x.y.0" is abbreviated into "x.y".*

These files are all available from the JPPF installer and/or from the JPPF download page.
In addition to this, Java 1.5 or later and Apache Ant 1.7.0 or later should already be installed on your machine.

We will assume the creation of a new folder called "JPPF-Tutorial", in which all these components are unzipped. Thus, we should have the following folder structure:

» JPPF-Tutorial
  » JPPF-x.y.z-admin-ui
  » JPPF-x.y.z-application-template
  » JPPF-x.y.z-driver
  » JPPF-x.y.z-node

2.2 Overview

2.2.1 Tutorial organization

We will base this tutorial on a pre-existing application template, which is one of the components of the JPPF distribution. The advantage is that most of the low-level wiring is already written for us, and we can thus focus on the steps to put together a JPPF application. The template is a very simple, but fully working, JPPF application, and contains fully commented source code, configuration files and scripts to build and run it.

It is organized with the following directory structure:

» **root directory**: contains the scripts to build and run the application
  » **src**: this is where the sources of the application are located
  » **classes**: the location where the Java compiler will place the built sources
  » **config**: contains the JPPF and logging configuration files
  » **lib**: contains the required libraries to build and run the application

2.2.2 Expectations

We will learn how to:

- write a JPPF task
- create a job and execute it
- process the execution results
- manage JPPF jobs
- run a JPPF application

The features of JPPF that we will use:

- JPPF task and job APIs
- local code changes automatically accounted for
- JPPF client APIs
- management and monitoring console
- configuring JPPF

By the end of this tutorial, we will have a full-fledged JPPF application that we can build, run, monitor and manage in a JPPF grid. We will also have gained knowledge of the workings of a typical JPPF application and we will be ready to write real-life, grid-enabled applications.

2.3 Writing a JPPF task

A JPPF task is the smallest unit of code that can be executed on a JPPF grid. From a JPPF perspective, it is thus defined as an *atomic* code unit. A task is always defined as a subclass of the class JPPFTask. JPPFTask is an abstract class that implements the Runnable interface. The part of a task that will be executed on the grid is whatever is written in its `run()` method.

From a design point of view, writing a JPPF task will comprise 2 major steps:

- create a subclass of `JPPFTask`.
- implement the `run()` method.

From the template application root folder, navigate to the folder src/org/jppf/application/template. You will see 2 Java files in this folder: "TemplateApplicationRunner.java" and "TemplateJPPFTask.java". Open the file "TemplateJPPFTask.java" in your favorite text editor.

In the editor you will see a full-fledged JPPF task declared as follows:

```
public class TemplateJPPFTask extends JPPFTask
```

Below this, you will find a run() method declared as:

```
  public void run()
  {
    // write your task code here.
    System.out.println("Hello, this is the node executing a template JPPF task");

    // ...

    // eventually set the execution results
    setResult("the execution was performed successfully");
  }
```

We can guess that this task will first print a "Hello …" message to the console, then set the execution result by calling the `setResult()` method with a string message. The setResult() method actually takes any object, and is provided as a convenience to store the results of the task execution, for later retrieval.

In this method, to show that we have customized the template, let's replace the line "`// …`" with a statement printing a second message, for instance "In fact, this is more than the standard template". The `run()` method becomes:

```
  public void run()
  {
    // write your task code here.
    System.out.println("Hello, this is the node executing a template JPPF task");
    System.out.println("In fact, this is more than the standard template");

    // eventually set the execution results
    setResult("the execution was performed successfully");
  }
```

Do not forget to save the file for this change to be taken into account.

The next step is to create a JPPF job from one or multiple tasks, and execute this job on the grid.

2.4 Creating and executing a job

A job is a grouping of tasks with a common set of characteristics and a common SLA. These characteristics include:

- common data shared between tasks
- a priority
- a maximum number of nodes a job can be executed on
- an execution policy describing which nodes it can run on
- a suspended indicator, that enables submitting a job in suspended state, waiting for an external command to resume or start its execution
- a blocking/non-blocking indicator, specifying whether the job execution is synchronous or asynchronous from the application's point of view

2.4.1 Creating and populating a job

In the JPPF APIs, a job is represented as an instance of the class JPPFJob.
To see how a job is created, let's open the source file "TemplateApplicationRunner.java" in the folder JPPF-x.y.z-application-template/src/org/jppf/application/template. In this file, navigate to the method `createJob()`.

This method is written as follows:

```
public JPPFJob createJob() throws Exception
{
  // create a JPPF job
  JPPFJob job = new JPPFJob();

  // give this job a readable unique id that we can use to
  // monitor and manage it.
  job.setId("Template Job Id");

  // add a task to the job.
  job.addTask(new TemplateJPPFTask());

  // add more tasks here ...

  // there is no guarantee on the order of execution of the tasks,
  // however the results are guaranteed to be returned in the same
  // order as the tasks.
  return job;
}
```

We can see that creating a job is done by calling the default constructor of class `JPPFJob`. The call to the method `job.setId(String)` is used to give the job a meaningful name that we can use later to manage it. If this method is not called, an id is automatically generated, as a string of 32 hexadecimal characters.

Adding a task to the job is done by calling the method `addTask(Object task, Object...args)`. The optional arguments are used when we want to execute other forms of tasks, that are not subclasses of `JPPFTask`. We will see their use in the more advanced sections of the JPPF user manual. As we can see, all the work is already done in the template file, so there is no need to modify the `createJob()` method for now.

2.4.2 Executing a job and processing the results

Now that we have learned how to create a job and populate it with tasks, we still need to execute this job on the grid, and process the results of this execution. Still in the source file "TemplateApplicationRunner.java", let's navigate to the `main(String...args)` method. we will first take a closer look at the `try` block, which contains a very important initialization statement:

```
jppfClient = new JPPFClient();
```

This single statement initializes the JPPF framework in your application. When it is executed JPPF will do several things:

- read the configuration file
- establish a connection with one or multiple servers for job execution
- establish a monitoring and management connection with each connected server
- register listeners to monitor the status of each connection

As you can see, the JPPF client has a non-negligible impact on memory and network resources. This is why we recommend to declare it as a singleton, and always use the same instance throughout your application. This will also ensure a greater scalability, as it is also designed for concurrent use by multiple threads. To this effect, we have declared it as a static variable in TemplateApplicationRunner.java:

```
private static JPPFClient jppfClient =  null;
```

It is also a good practice to release the resources used by the JPPF client when they are not used anymore. We actually recommend to do this by calling its `close()` method within a `finally{}` block:

```
finally
{
  if (jppfClient != null) jppfClient.close();
}
```

Back to the main method, after initializing the JPPF client, the next steps are to initialize our job runner, create a job and execute it:

```
// create a runner instance.
TemplateApplicationRunner runner = new TemplateApplicationRunner();

// Create a job
JPPFJob job = runner.createJob();

// execute a blocking job
runner.executeBlockingJob(job);
```

The call to `runner.createJob()` is exactly what we saw in the previous section 2.4.1 . What remains to do is to execute the job and process the results, which is the intent of the call to `executeBlockingJob(JPPFJob job)`:

```
/**
 * Execute a job in blocking mode. The application will be blocked until the job
 * execution is complete.
 * @param job the JPPF job to execute.
 * @throws Exception if an error occurs while executing the job.
 */
public void executeBlockingJob(JPPFJob job) throws Exception
{
  // set the job in blocking mode.
  job.setBlocking(true);

  // Submit the job and wait until the results are returned.
  // The results are returned as a list of JPPFTask instances,
  // in the same order as the one in which the tasks where initially added the job.
  List<JPPFTask> results = jppfClient.submit(job);

  // process the results
  for (JPPFTask task: results)
  {
    // if the task execution resulted in an exception
    if (task.getException() != null)
    {
      // process the exception here ...
    }
    else
    {
      // process the result here ...
    }
  }
}
```

The first statement of this method ensures that the job will be submitted in blocking mode, meaning that the application will block until the job is executed:

```
job.setBlocking(true);
```

This is, in fact, optional since submission in blocking mode is the default behavior in JPPF.

The second statement is the one that will send the job to the server and wait until it has been executed and the results are returned:

```
List<JPPFTask> results = jppfClient.submit(job);
```

We can see that the results are returned as a list of `JPPFTask` objects. It is guaranteed that each task in this list has the same position as the corresponding task that was added to the job. In other words, the results are always in the same order as the tasks in the the job.

The last step is to interpret and process the results. From the JPPF point of view, there are two possible outcomes of the execution of a task: one that raised a `Throwable`, and one that did not. When an uncaught `Throwable` (i.e. generally an instance of a subclass of `java.lang.Error` or `java.lang.Exception`) is raised, JPPF will catch it and set it as the outcome of the task. To do so, the method `JPPFTask.setException(Exception)` is called. You will note that the parameter is an instance of Exception or of one of its subclasses. Thus, any uncaught `Error` will be wrapped in a `JPPFException`. JPPF considers that exception processing is part of the life cycle of a task and provides the means to capture that information accordingly.

This explains why, in our template code, we have separated the result processing of each task in 2 blocks:

```
if (task.getException() != null)
{
  // process the exception here ...
}
else
{
  // process the result here ...
}
```

The actual results of the computation of a task can be any attribute of the task, or any object accessible from them. The `JPPFTask` API provides two convenience methods to help doing this: `setResult(Object)` and `getResult()`, however it is not mandatory to use them, and you can implement your own result handling scheme, or it could simply be a part of the task's design.

As an example for this tutorial, let's modify this part of the code to display the exception message if an exception was raised, and to display the result otherwise:

```
if (task.getException() != null)
{
  System.out.println("An exception was raised: "
    + task.getException().getMessage());
}
else
{
  System.out.println("Execution result: "
    + task.getResult());
}
```

We can now save the file and close it.

2.5 Running the application

We are now ready to test our JPPF application. To this effect, we will need to first start a JPPF grid, as follows:

Step 1: start a server

Go to the JPPF-x.y.z-driver folder and open a command prompt or shell console. Type "ant".
You should see the following lines printed to the console:

```
run:
     [echo] starting the JPPF driver
     [java] Class Server initialized - listening on port 11111
     [java] Client Server initialized - listening on port 11112
     [java] Tasks Server initialized - listening on port 11113
     [java] JPPF Driver management initialized
     [java] JPPF Driver initialization complete
```

The server is now ready to process job requests.

Step 2: start a node

Go to the JPPF-x.y.z-node folder and open a command prompt or shell console. Type "ant".
You will then see the following lines printed to the console:

```
run:
     [java] JPPFClassLoader.init(): attempting connection to the class server
     [java] JPPFClassLoader.init(): Reconnected to the class server
     [java] PeerNode.init(): Attempting connection to the JPPF driver
     [java] PeerNode.init(): Reconnected to the JPPF driver
     [java] Node successfully initialized
```

Together, this node and the server constitute the smallest JPPF grid that you can have.

Step 3: run the application

Go to the JPPF-x.y.z-application-template folder and open a command prompt or shell console. Type "ant". This time, the Ant script will first compile our application, then run it. You should see these lines printed to the console:

```
  run:
    [java] [client: driver-1 (<ip_address>:11198)] ClassServerDelegate.init(): Attempting
connection to the class server
    [java] [client: driver-1 (<ip_address>:11198)] ClassServerDelegate.init():
Reconnected to the class server
    [java] [client: driver-1 (<ip_address>:11198)] : Attempting connection to the JPPF
task server
    [java] [client: driver-1 (<ip_address>:11198)] : Reconnected to the JPPF task server
    [java] Execution result: the execution was performed successfully
```

where <*ip_address*> corresponds to the IP address of your computer.

You will notice that the last printed line is the same message that we used in our task in the `run()` method, to set the result of the execution in the statement:

```
    setResult("the execution was performed successfully");
```

Now, if you switch back to the node console, you should see that 2 new messages have been printed:

```
     [java] Hello, this is the node executing a template JPPF task
     [java] In fact, this is more than the standard template
```

These 2 lines are those that we actually coded at the beginning of the task's run() method:

```
  System.out.println("Hello, this is the node executing a template JPPF task");
  System.out.println("In fact, this is more than the standard template");
```

From these messages, we can conclude that our application was run successfully. Congratulations!

At this point, there is however one aspect that we have not yet addressed: since the node is a separate process from our application, ***how does it know to execute our task?*** Remember that we have not even attempted to deploy the application classes to any specific location. We have simply compiled them so that we can execute our application locally. This topic is the object of the next section of this tutorial.

2.6 Dynamic deployment

One of the greatest features of JPPF is its ability to dynamically load the code of an application that was deployed only locally. JPPF extends the standard Java class loading mechanism so that, by simply using the JPPF APIs, the classes of an application are loaded to any remote node that needs them. The benefit is that *no deployment of the application is required to have it run on a JPPF grid*, no matter how many nodes or servers are present in the grid. Furthermore, this mechanism is totally transparent to the application developer.

A second major benefit is that code changes are automatically taken into account, without any need to restart the nodes or the server. This means that, when you change any part of the code executed on a node, all you have to do is recompile the code and run the application again, and the changes will take effect immediately, on all the nodes that execute the application.

We will now demonstrate this by making a small, but visible, code change and running it against the server and node we have already started, If you have stopped them already, just perform again all the steps described in the previous section (2.5), before continuing.

Let's open again the source file "TemplateJPPFTask.java" in JPPF-x.y.z-application-template/src/org/jppf/application/template/, and navigate to the `run()` method. Let's replace the first two lines with the following:

```
System.out.println("*** We are now running a modified version of the code ***");
```

The `run()` method should now look like this:

```
public void run()
{
  // write your task code here.
  System.out.println("*** We are now running a modified version of the code ***");

  // eventually set the execution results
  setResult("the execution was performed successfully");
}
```

Save the changes to the file, and open or go back to a command prompt or shell console in the JPPF-2.0-application-template folder. From there, type "ant" to run the application again. You should now see the same messages as in the initial run displayed in the console. This is what we expected. On the other hand, if you switch back to the node console, you should now see a new message displayed:

```
[java] *** We are now running a modified version of the code ***
```

Success! We have successfully executed our new code without any explicit redeployment.

2.7 Job Management

Now that we are able to create, submit and execute a job, we can start thinking about monitoring and eventually controlling its life cycle on the grid. To do that, we will use the JPPF administration and monitoring console. The JPPF console is a standalone graphical tool that provides user-friendly interfaces to:

- obtain statistics on server performance
- define, customize and visualize server performance charts
- monitor and control the status and health of servers and nodes
- monitor and control the execution of the jobs on the grid
- manage the workload and load-balancing behavior

2.7.1 Preparing the job for management

In our application template, the job that we execute on the grid has a single task. As we have seen, this task is very short-live, since it executes in no more than a few milliseconds. This definitely will not allow us us to monitor or manage it with our bare human reaction time. For the purpose of this tutorial, we will now adapt the template to something more realistic from this perspective.

Step 1: make the tasks last longer

What we will do here is add a delay to each task, before it terminates. It will do nothing during this time, only wait for a specified duration. Let's edit again the source file "TemplateJPPFTask.java" in JPPF-x.y.z-application-template/src/org/jppf/application/template/ and modify the `run()` method as follows:

```
public void run()
{
  // write your task code here.
  System.out.println("*** We are now running a modified version of the code ***");

  // simply wait for 3 seconds
  try
  {
    Thread.sleep(3000L);
  }
  catch(InterruptedException e)
  {
    setException(e);
    return;
  }

  // eventually set the execution results
  setResult("the execution was performed successfully");
}
```

Note that here, we make an explicit call to `setException()`, in case an `InterruptedException` is raised. Since the exception would be occurring in the node, capturing it will allow us to know what happened from the application side.

Step 2: add more tasks to the job, submit it as suspended

This time, our job will contain more than one task. In order for us to have the time to manipulate it from the administration console, we will also start it in suspended mode. To this effect, we will modify the method `createJob()` of the application runner "TemplateApplicationRunner.java" as follows:

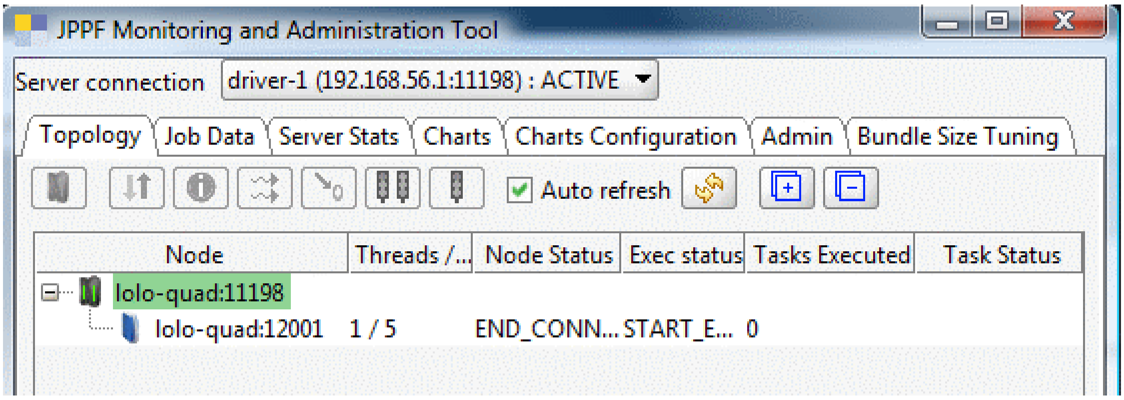
```
public JPPFJob createJob() throws Exception
{
  // create a JPPF job
  JPPFJob job = new JPPFJob();

  // give this job a readable unique id that we can use to monitor and manage it.
  job.setId("Template Job Id");

  // add 10 tasks to the job.
  for (int i=0; i<10; i++) job.addTask(new TemplateJPPFTask());

  // start the job in suspended mode
  job.getJobSLA().setSuspended(true);

  return job;
}
```

Step 3: start the JPPF components

If you have stopped the server and node, simply start them again as described in the first two step of section 2.5 of this tutorial.

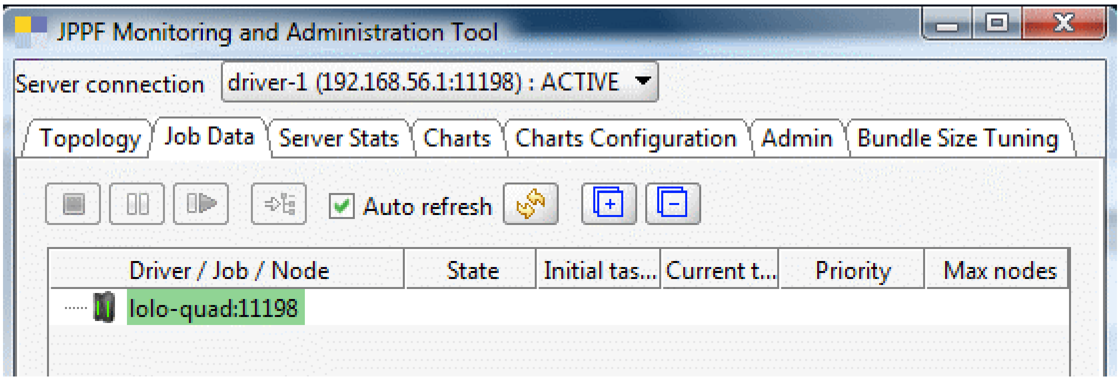
We will also start the administration console:

Go to the JPPF-x.y.z-admin-ui folder and open a command prompt or shell console. Type "ant".
When the console is started, you will see a panel named "Topology" displaying the servers and the nodes attached to them. It should look like this:



We can see here that a server is started on machine "lolo-quad" and that it has a node attached to it. The color for the server is a health indicator, green meaning that it is running normally and red meaning that it is down.

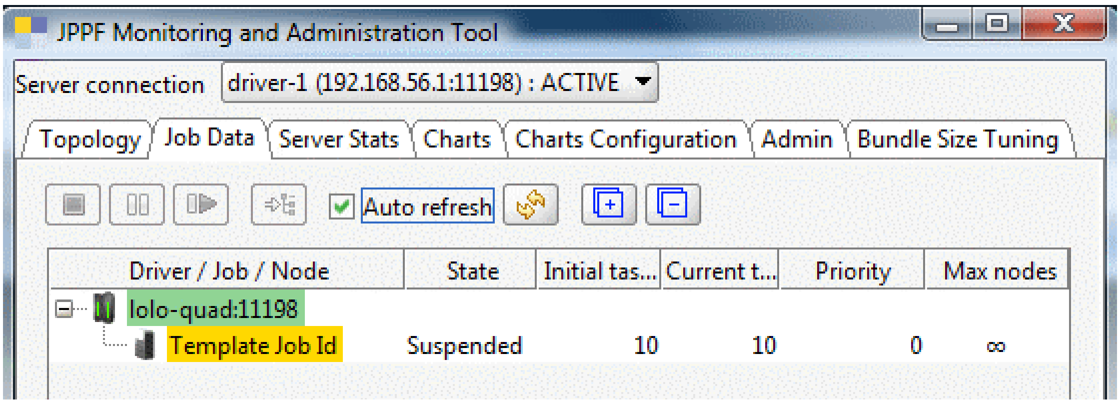
Let's switch to the "Job Data" panel, which should look like this:



We also see the color-coded driver health information in this panel. There is currently no other element displayed, because we haven't submitted a job yet.

Step 4: start a job

We will now start a job by running our application: go to the JPPF-x.y.z-application-template folder and open a command prompt or shell console. Type "ant". Switch back to the administration console. We should now see some change in the display:

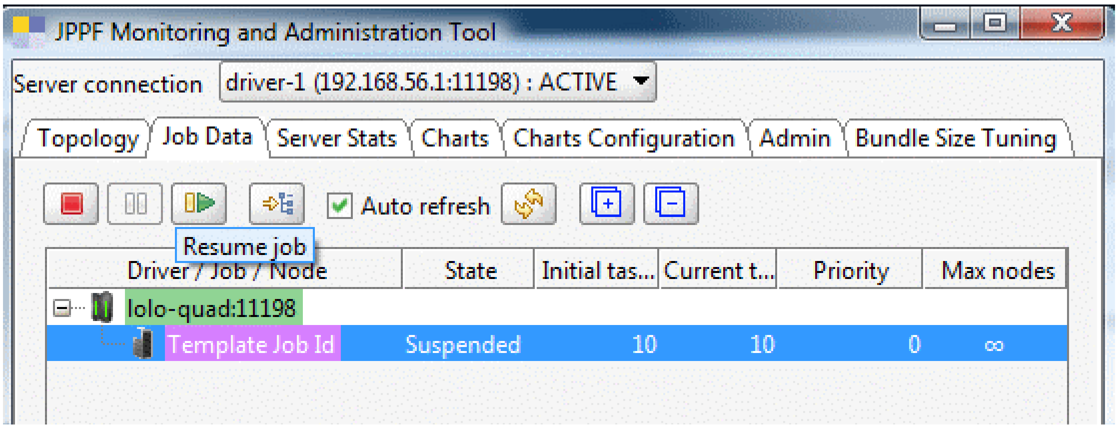


We now see that a job is present in the server's queue, in suspended state (yellow highlighting). Here is an explanation of the columns in the table:

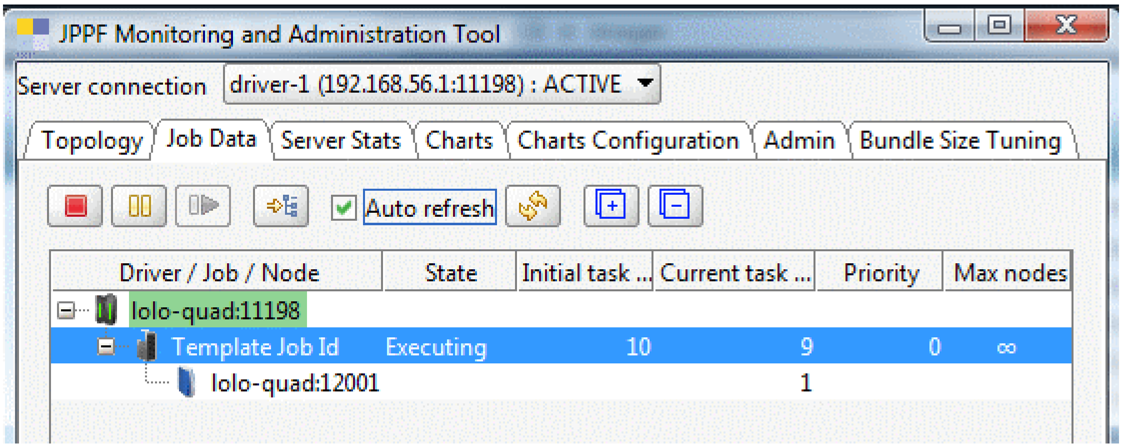
- "Driver / Job / Node" : displays an identifier for a server, for a job submitted to that server, or for a node to which some of the tasks in the job have been dispatched for execution
- "State" : the current state of a job, either "Suspended" or "Executing"
- "Initial task count" : the number of tasks in the job at the time it was submitted by the application
- "Current task count": the number of tasks remaining in the job, that haven't been executed
- "Priority" : this is the priority, of the job, the default value is 0.
- "Max nodes" : the maximum number of nodes a job can be executed on. By default, there is no limit, which is represented as the infinity symbol

Step 5: resuming the job execution

Since the job was submitted in suspended state, we will resume its execution manually from the console. Select the line where the job "Template Job Id" is displayed. You should see that some buttons are now activated. Click on the resume button (marked by the icon) to resume the job execution, as shown below:



As soon as we resume the job, the server starts distributing tasks to the node, and we can see that the current task count starts decreasing accordingly, and the job status has been changed to "Executing":



You are encouraged to experiment with the tool and the code. For example you can add more tasks to the job, make them last longer, suspend, resume or terminate the job while it is executing, etc...

2.8 Conclusion

In this tutorial, we have seen how to write a JPPF-enabled application from end to end. We have also learned the basic APIs that allow us to write an application made of atomic and independent execution units called tasks, and group them into jobs that can be executed on the grid. We have also learned how jobs can be dynamically managed and monitored while executing. Finally, we also learned that, even though an application can be distributed over any number of nodes, there is no need to explicitely deploy the application code, since JPPF implicitely takes care of it.

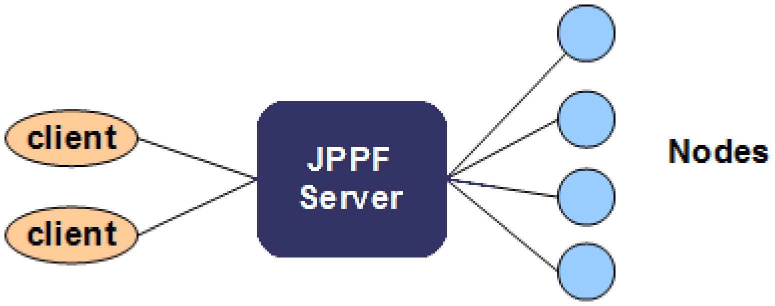
3 JPPF Overview

3.1 Architecture and topology

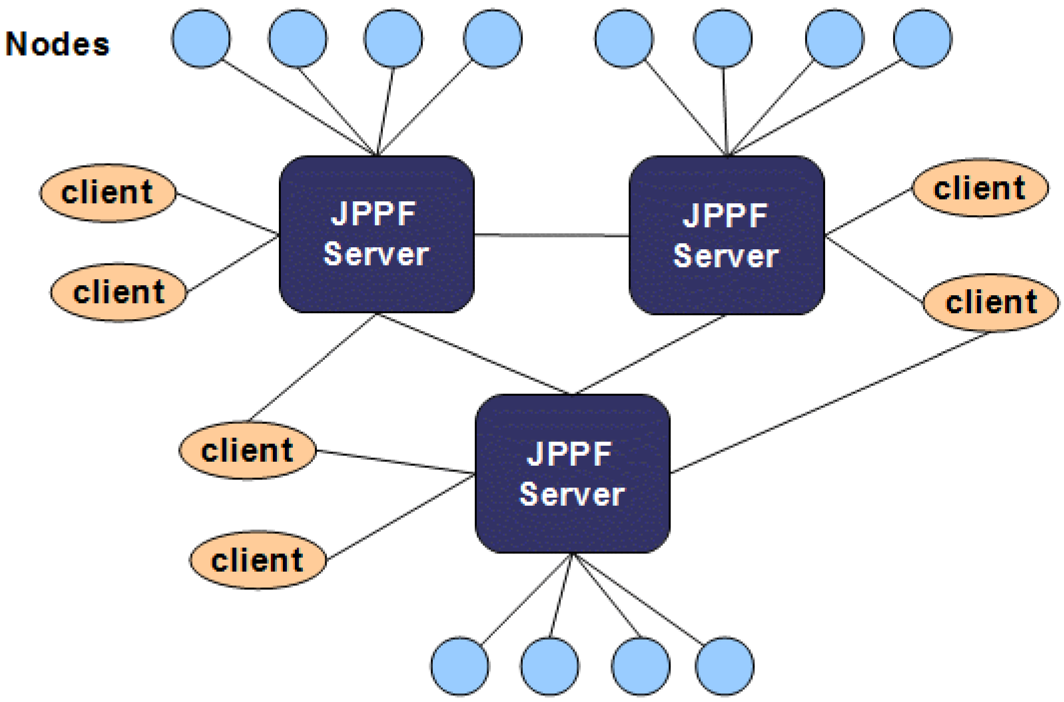
A JPPF grid is made of three different types of components that communicate together:

- **clients** are entry points to the grid and enable developers to submit work via the client APIs
- **servers** are the components that receive work from the clients and dispatch it to the nodes
- **nodes** perform the actual work execution

The figure below shows how all the components are organized together:



From this picture, we can see that the server plays a central role, and its interactions with the nodes define a *master / worker* architecture, where the server (i.e. master) distributes the work to the nodes (i.e. workers). This also represents the most common topology in a JPPF grid, where each client is connected to a single server, and many nodes are attached to the same server. As with any such architecture, this one is facing the risk of a single point of failure.To mitigate this risk, JPPF provides the ability to connect multiple servers together in a peer-to-peer network and additional connectivity options for clients and nodes, as illustrated in this figure:



Note how some of the clients are connected to multiple servers, providing failover as well as load balancing capabilities. In addition, and not visible in the previous figure, the nodes have a failover mechanism that will enable them to attach to a different server, should the one they are attached to fail or die.

The connection between two servers is directional: if server A is connected to server B then A will see B as a client, and B will see A as a node. This relationship can be made bi-directional by also connecting B to A. Note that in this scenario, each server taken locally still acts as a master in a master/worker paradigm.

In short, we can say that the single point of failure issue is addressed by a combination of redundancy and dynamic reconfiguration of the grid topology.

3.2 Work distribution

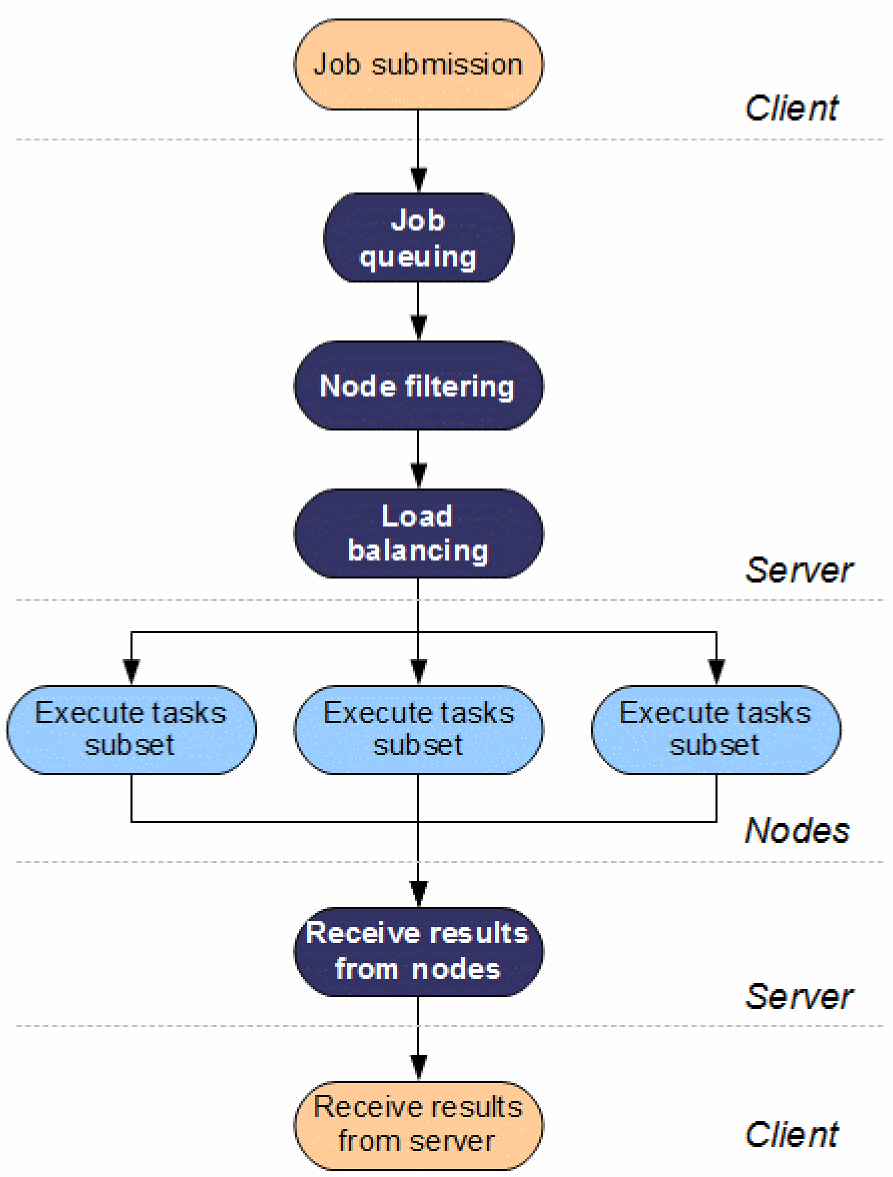
To understand how the work is distributed in a JPPF grid, and what role is played by each component, we will start by defining the two units of work that JPPF handles.

A **task** is the smallest unit of work that can be handled in the grid. From the JPPF perspective, it is considered *atomic*.

A **job** is a logical grouping of tasks that are submitted together, and may define a common service level agreement (SLA) with the JPPF grid.The SLA can have a significant influence on how the job's work will be distributed in the grid, by specifying a number of behavioral characteristics:

- rule-based filtering of nodes, specifying which nodes the work can be distributed to (aka execution policies)
- maximum number of nodes the work can be distributed to
- job priority
- start and expiration schedule
- user-defined metadata which can be used by the load balancer

To illustrate the most common flow of a job's execution, let's take a look at the following flow chart:



This chart shows the different steps involved in the execution of a job, and where each of them takes place with regards to the grid component boundaries.

It also shows that the main source of parallelism is provided by the load balancer, whose role is to split each job into multiple subsets that can be executed on multiple nodes in parallel. There are other sources of parallelism at different levels, and we will describe them in the next sections.

3.3 Networking considerations

3.3.1 Two channels per connection

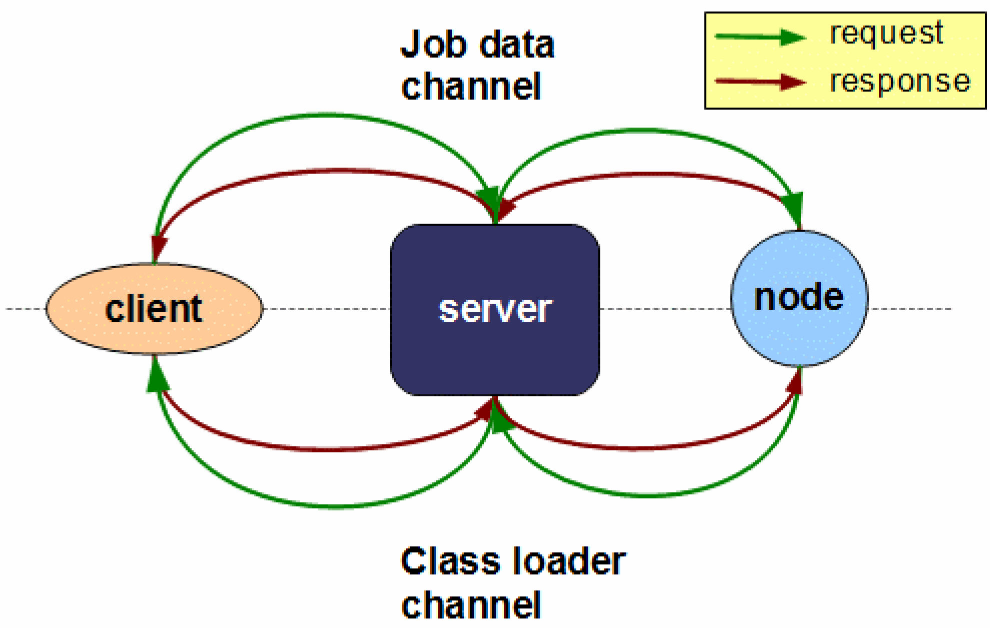
Each connection between a server and any other component is in fact a grouping of two network channels:

- one channel is used to transport job data
- the other channel is used by the JPPF distributed class loader, that allows Java classes to be deployed on-demand where they are needed, completely transparently from a developer's perspective.

3.3.2 Synchronous networking

In JPPF, all network communications are synchronous and follow a protocol based on a request/response paradigm. The attribution of requester vs. responder role depends on which components communicate and through which channel.

We illustrate this in the following picture:



This communication model has a number of important implications:

- nodes can only process one job at a time; however they can execute multiple tasks in parallel
- in the same way, a single client / server connection can only process one job at a time; however, each client can be connected multiple times to the same server, or multiple times to many servers
- in the case of a server-to-server communication, only one job can be processed at a time, since a server attaches to another server in exactly the same way as a node.

3.3.3 Protocol

JPPF components communicate by exchanging messages. As described in the previous section, each JPPF transaction will be made of a request message, followed by a response message.

Messages all have the same structure, and are made of one or more blocks of data (in fact blocks of bytes), each preceded by its block size. Each block of data represents a serialized object graph. Thus, each message can be represented generically as follows:

Size 1	Serialized Object 1	. . . . .	Size n	Serialized Object n

The actual message format is different for each type of communication channel, and may also differ depending on whether it is a request or response message:

Job data channel

A job data request is composed of the following elements:

- a header, which is an object representing information about the job, including the number of tasks in the job, the job SLA, job metadata, and additional information required internally by the JPPF components.
- a data provider, which is a read-only container for data shared among all the tasks
- the tasks themselves

It can be represented as follows:

Header size	Header (nb tasks)	Data provider size	Data provider data	Size $_1$	Task $_1$	.....	Size $_n$	Task $_n$

To read the full message, JPPF has to first read the header and obtain the number of tasks in the job.

The response will be in a very similar format, except that it doesn't have a data provider: being read-only, no change to its content is expected, which removes the need to send it in the response. Thus the response can be represented as:

Header size	Header (nb tasks)	Size $_1$	Task $_1$	.....	Size $_n$	Task $_n$

Class loader channel

A class loader message, either request or response, is always made of a single serialized object. Therefore, the message structure is always as follows:

size	Resource request / response

3.4 Sources of parallelism

3.4.1 At the client level

There are three ways JPPF clients can provide parallel processing, which may be used individually or in any combination:

Single client, multiple concurrent jobs
A single client may submit multiple jobs in parallel. This differs from the single client/single job scenario in that the jobs must be submitted in *non-blocking* mode, and their results are retrieved asynchronously. An other difference is that the client must establish multiple connections to the server to enable parallelism, and not just asynchronous submission. When multiple non-blocking jobs are submitted over a single connection, only one at a time will be submitted, and the others will be queued on the client side. The only parallelism is in the submission of the jobs, but not in their execution. To enable parallel execution of multiple jobs, it is necessary to configure a *pool of connections* for the client.The size of the pool determines the number of jobs that can be processed in parallel by the server.

Multiple clients
In this configuration, the parallelism occurs naturally, by letting the different clients work concurrently.

Mixed local and remote execution
Clients have the ability to execute jobs locally, within the same process, rather than remotely. They may also use both capabilities at the same time, in which case a load-balancing mechanism will provide an additional source of parallelism.

3.4.2 At the server level

The server has a number of factors that determine what can be parallelized and how much:

Number of connected clients
The number of connected clients, or more accurately, client connections, has a direct influence on how many jobs can be processed by the grid at any one time.
The relationship is defined as: *maximum number of parallel jobs = total number of client connections*

Number of attached nodes
This determines the maximum number of jobs that can be executed on the grid nodes. With regards to the previous point, we can redefine it as: *maximum number of parallel jobs* = min(*total number of client connections, total number of nodes)*

Load balancing
This is the mechanism that splits the jobs into multiple subsets of their tasks, and distributes these subsets over the available nodes. Given the synchronous nature of the server to node connectins, a node is available only when it is not already executing a job subset. The load balancing also computes how many tasks will be sent to each node, in a way that can be static, dynamic, or even user-defined.

Job SLA
The job Service Level Agreement is used to filter out nodes in which the user does not want to see a job executed. This can be done by specifying an execution policy (rules-based filtering) for the job, or by configuring the maximum nuumber of nodes a job can run on (grid partitioning).

Parallel I/O
Each server maintains internally a pool of threads dedicated to network I/O. The size of this pool determines how many nodes the server can communicate with in parallel, *at any given time*. Furthermore, as communication with the nodes is non-blocking, this pool of I/O threads is part of a mechanism that achieves a preemptive multitasking of the network I/O. This means that, even if you have a limited number of I/O threads, the overall result will be as if the server were communicating with all nodes in parallel.

3.4.3 At the node level

To execute tasks, each node uses a pool of threads that are called “processing threads”. The size of the pool determines the maximum number of tasks a single node can execute in parallel. The pool size may be adjusted either statically or dynamically to account for the actual number of processors available to the node, and for the tasks' resource usage profile (i.e. I/O-bound tasks versus CPU-bound tasks).

4 Development Guide

4.1 Task objects

In JPPF terms, a task is the smallest unit of execution that can be handled by the framework. We will say that it is an *atomic* execution unit. A JPPF application creates tasks, groups them into a job, and submits the job for execution on the grid.

4.1.1 JPPFTask

JPPFTask is the base super class for any task that is run by JPPF. We will see in the next sections that other forms of tasks, that do not inherit from JPPFTask, are still wrapped by the framework in a subclass of JPPFTask.

JPPFTask is defined as follows:

```
public abstract class JPPFTask implements Runnable, Serializable
{
  ...
}
```

We have outlined three important keywords that characterize JPPFTask:

- abstract: JPPFTask cannot be used directly, it must be extended to construct a real task
- Runnable: when writing a JPPF task, the run() method of `java.lang.Runnable` must be implemented. This is the part of a task that will be executed on a remote node.
- Serializable: tasks are sent to servers and nodes over a network. JPPF uses the Java serialiazation mechanism to transform task objects into a form appropriate for networking

To write a real task in your application, you simply extend JPPFTask to implement your own type:

```
public class MyTask extends JPPFTask
{
  public void run()
  {
    // ... your code here ...
  }
}
```

We will now review the functionalities that are inherited from JPPFTask.

4.1.1.1 Execution results handling

JPPFTask provides 2 convenience methods to store and retrieve the results of the execution:

- `public void setResult(Object result)` : stores the execution result; the argument must be serializable
- `public Object getResult()` : retrieves the execution result

Here is an example using these methods:

```
public class MyTask extends JPPFTask
{
  public void run()
  {
    // ... some code here ...
    setResult("This is the result");
  }
}
```

and later in your application, you would use:

```
String result = (String) myTask.getResult();
```

Using `getResult()` and `setResult()` is not mandatory. As we mentioned earlier, these methods are provided as conveniences with a meaningful semantics attached to them. There are many other ways to store and retrieve execution results, which can be used to the exclusion of others, or in any combination. These include, but are not limited to:

- using custom attributes in the task class and their accessors
- storing and getting data to/from a database
- using a file system
- using third-party applications or libraries
- etc ...

4.1.1.2 Exception handling - task execution

Exception handling is a very important part of processing a task. In effect, exceptions may arise from many places: in the application code, in the JVM, in third-party APIs, etc... To handle this, JPPF provides both a mechanism to process uncaught exceptions and methods to store and retrieve exceptions that arise while executing a task.

JPPFTask provides 2 methods to explicitely handle exceptions:

- `public void setException(Exception e)` : store an exception for later retrieval
- `public Exception getException()` : retrieve an exception that was thrown during execution

Here is an example of explicit exception handling:

```
public class MyTask extends JPPFTask
{
  public void run()
  {
    try
    {
      // ... some code here ...
    }
    catch(Exception e)
    {
      setException(e);
    }
  }
}
```

Later on, you can retrieve the exception as follows:

```
Exception exception = myTask.getException();
```

JPPF also automatically handles uncaught exceptions. In fact, we should say that it handles uncaught *throwables*. Uncaught throwables are never propagated beyond the scope of a task, as this would cause an unpredictable behavior of the node that executes the task. Instead, they are stored within the task using the `setException()` method. This way, it is always possible for the application to know what happened.

We may wonder what happens when the throwable is not an instance of `Exception`, as the method `setException()` only takes an `Exception` object as argument. In this case, the throwable is wrapped into an instance of JPPFException and can be retrieved by calling its `getCause()` method.The following code shows how JPPF handles uncaught exceptions and throwables:

```
JPPFTask task = ...;
try
{
  task.run();
}
catch(Throwable t)
{
  if (t instanceof Exception)
  {
    task.setException((Exception) t);
  }
  else
  {
    task.setException(new JPPFException(t));
  }
}
```

Then in the application, you can retrieve the throwable as follows:

```
JPPFTask task = ...;
if (task.getException() != null)
{
  Throwable cause = null;
  Exception e = task.getException();
  if (e instanceof JPPFException)
  {
    cause = e.getCause();
  }
  else
  {
    cause = e;
  }
}
```

4.1.1.3 Task life cycle

JPPF provides some options to control a task's life cycle once it has been submitted. These include the following functionalities:

- cancel a task
- restart a task
- task timeout settings

Canceling a task applies whether the task has already started or not. Restart and timeout of a task only apply while they are executing. In any case, if a task has already completed its execution, it cannot be canceled, restarted or timed out anymore.

Apart from timeout settings, controlling the life cycle of a task is normally done externally, using the JPPF remote management facilities. We will see those later, in a dedicated chapter of this user manual. It also requires that the task be given an identifier that distinguishes it from the other tasks. To this effect, JPPFTask provides the following accessors:

`public String getId()`: to get the user-defined id for the task
`public void setId(String id)`: to set the user-defined id for the task

It is also possible to perform a specific processing when a task life cycle event occurs. For this, JPPFTask provides a callback method for each type of event:

`public void onCancel()`: invoked when the task is canceled
`public void onRestart()`: invoked when the task is restarted (`onCancel()` is not called in this case)
`public void onTimeout()`: invoked when the task times out

By default, these methods do not do anything. You can, however, override them to implement any application-specific processing, such as releasing resources used by the task, updating the state of the task, etc.

Here is a code sample that illustrates this:

```
public class MyTask extends JPPFTask
{
  public MyTask(String taskId)
  {
    // set the task id
    this.setId(taskId);
  }

  public void run()
  {
    // task processing here ...
  }

  @override
  public void onCancel()
  {
    // process task cancel event ...
  }
```

```
    @override
    public void onRestart()
    {
      // process task restart event ...
    }

    @override
    public void onTimeout()
    {
      // process task timeout event ...
    }
  }
```

A task timeout can be set in two ways:

1) After a specified duration, counting from the time the task is started. JPPFTask provides the following methods for this:

- `public void setTimeout(long duration)`: sets a timeout after *duration* milliseconds
- `public long getTimeout()`: get the timeout duration set for this task

For example:

```
myTask.setTimeout(5000L);
```

will cause myTask to time out if it lasts more than 5 seconds.

2) At a specified date/time, using the following methods:

- `public void setTimeoutDate(String date, String dateFormat)`: specifies the date and time at which the task will time out, as well as the format in which the date is expressed.
- `public String getTimeoutDate()`: get the date/time at which the task will time out
- `public String getTimeoutFormat()`: get the format in which the date/time is expressed; the format pattern follows the specifications for the SimpleDateFormat class.

For example:

```
myTask.setTimeoutDate("09/30/2010 12:08 PM", "MM/dd/yyyy hh:mm a");
```

will cause the task to time out on September 30, 2010 at 12:08 PM.

These two ways of setting a task timeout are mutually exclusive, and setting the value(s) for one will reset the value(s) for the other.

4.1.2 Exception handling - node processing

It is possible that an error occurs while the node is processing a job or a task, before or after its execution.
These error conditions include any instance of Throwable, i.e. any Exception or Error occurring during serialization or deserialization of the tasks, or while sending or receiving data to or from the server.

Depending on when they occur, these errors can be handled in two different ways:

1. **When the node is receiving tasks from the server**: if any error is generated, for any task, then all the tasks will be sent back to the client application as they were initially, with only their exception attribute (see `JPPFTask.getException()`) set to the captured error.
2. **After execution of the tasks**: for each task that generates an error that prevents it from being sent back to the server, the JPPF will substitute an instance of `JPPFExceptionResult`, which is JPPF task used solely to describe the error and the object on which it occurred.

4.1.3 JPPF-annotated tasks

Another way to write a JPPF task is to take an existing class and annotate one of its public methods or constructors using @JPPFRunnable.

Here is an example:

```
public class MyClass implements Serializable
{
  @JPPFRunnable
  public String myMethod(int intArg, String stringArg)
  {
    String s = "int arg = " + intArg + ", string arg = \"" + stringArg + "\"";
    System.out.println(s);
    return s;
  }
}
```

We can see that we are simply using a POJO class, for which we annotated the `myMethod()` method with `@JPPFRunnable`. At runtime, the arguments of the method will be passed when the task is added to a job, as illustrated in the following example:

```
JPPFJob job = new JPPFJob();
JPPFTask task = job.addTask(new MyClass(), 3, "string arg");
```

Here we simply add our annotated class as a task, setting the two arguments of the annotated method in the same call. Note also that a JPPFTask object is returned. It is generated by a mechanism that wraps the annotated class into a JPPFTask, which allows it to use most of the functionalities that come with it.

JPPF-annotated tasks present the following properties and constraints:

- if the annotated element is an *instance* (non-static) method, the annotated class must be serializable
- if the class is already an instance of JPPFTask, the annotation will be ignored
- if an annotated method has a return type (i.e. non void), the return value will be set as the task result
- it is possible to annotate a public method or constructor
- an annotated method can be static or non-static
- if the annotated element is a constructor or a static method, the first argument of JPPFJob.addTask() must be a Class object representing the class that declares it.
- an annotated method or constructor can have any signature, with no limit on the number of arguments
- through the task-wrapping mechanism, a JPPF-annotated class benefits from the JPPFTask facilities described in the previous section 4.1.1 , except for the callback methods onCancel(), onRestart() and onTimeout().

Here is an example using an annotated constructor:

```
public class MyClass implements Serializable
{
  @JPPFRunnable
  public MyClass(int intArg, String stringArg)
  {
    String s = "int arg = " + intArg + ", string arg = \"" + stringArg + "\"";
    System.out.println(s);
  }
}

JPPFJob job = new JPPFJob();
JPPFTask task = job.addTask(MyClass.class, 3, "string arg");
```

Another example using an annotated static method:

```
public class MyClass implements Serializable
{
  @JPPFRunnable
  public static String myStaticMethod(int intArg, String stringArg)
  {
    String s = "int arg = " + intArg + ", string arg = \"" + stringArg + "\"";
    System.out.println(s);
    return s;
  }
}

JPPFJob job = new JPPFJob();
JPPFTask task = job.addTask(MyClass.class, 3, "string arg");
```

Note how, in the last 2 examples, we use `MyClass.class` as the first argument in `JPPFJob.addTask()`.

4.1.4 Runnable tasks

Classes that implement [java.lang.Runnable] can be used as JPPF tasks without any modification. The run() method will then be executed as the task's entry point. Here is an example:

```
public class MyRunnableClass implements Runnable, Serializable
{
  public void run()
  {
    System.out.println("Hello from a Runnable task");
  }
}

JPPFJob job = new JPPFJob();
JPPFTask task = job.add(new MyRunnableClass());
```

The following rules apply to Runnable tasks:

- the class must be serializable
- if the class is already an instance of `JPPFTask`, or annotated with `@JPPFRunnable`, it will be processed as such
- through the task-wrapping mechanism, a Runnable task benefits from the `JPPFTask` facilities described in the previous section 4.1.1 , except for the callback methods onCancel(), onRestart() and onTimeout().

4.1.5 Callable tasks

In the same way as Runnable tasks, classes implementing `java.util.concurrent.Callable<V>` can be directly used as tasks. In this case, the `call()` method will be used as the task's execution entry point. Here's an example:

```
public class MyCallableClass implements Callable<String>, Serializable
{
  public String call() throws Exception
  {
    String s = "Hello from a Callable task";
    System.out.println(s);
    return s;
  }
}

JPPFJob job = new JPPFJob();
JPPFTask task = job.add(new MyCallableClass());
```

The following rules apply to Callable tasks:

- the Callable class must be serializable
- if the class is already an instance of `JPPFTask`, annotated with `@JPPFRunnable` or implements `Runnable`, it will be processed as such and the `call()` method will be ignored
- the return value of the `call()` method will be set as the task result
- through the task-wrapping mechanism, a callable class benefits from the `JPPFTask` facilities described in the previous section 4.1.1 , except for the callback methods onCancel(), onRestart() and onTimeout().

4.1.6 POJO tasks

The most unintrusive way of defining a task is by simply using an existing POJO class without any modification. This will allow you to use existing classes directly even if you don't have the source code. A POJO task offers the same possibilities as a JPPF annotated task (see section Error: Reference source not found), except for the fact that we need to specifiy explicitely which method or constructor to use when adding the task to a job. To this effect, we use a different form of the method `JPPFJob.addTask()`, that takes a method or constructor name as its first argument.

Here is a code example illustrating these possibilities:

```
public class MyClass implements Serializable
{
  public MyClass()
  {
  }

  public MyClass(int intArg, String stringArg)
  {
    String s = "int arg = " + intArg + ", string arg = \"" + stringArg + "\"";
    System.out.println(s);
  }

  public String myMethod(int intArg, String stringArg)
  {
    String s = "int arg = " + intArg + ", string arg = \"" + stringArg + "\"";
    System.out.println(s);
    return s;
  }

  public static String myStaticMethod(int intArg, String stringArg)
  {
    String s = "int arg = " + intArg + ", string arg = \"" + stringArg + "\"";
    System.out.println(s);
    return s;
  }
}

JPPFJob job = new JPPFJob();

// add a task using the constructor as entry point
JPPFTask task1 = job.addTask("MyClass", MyClass.class, 3, "string arg");

// add a task using an instance method as entry point
JPPFTask task2 = job.addTask("myMethod", new MyClass(), 3, "string arg");

// add a task using a static method as entry point
JPPFTask task3 = job.addTask("myStaticMethod", MyClass.class, 3, "string arg");
```

POJO tasks present the following properties and constraints:

- if the entry point is an *instance* (non-static) method, the class must be serializable
- if a method has a return type (i.e. non void), the return value will be set as the task result
- it is possible to use a public method or constructor as entry point
- a method entry point can be static or non-static
- A POJO task is added to a job by calling a `JPPFJob.addTask()` method whose first argument is the method or constructor name.
- if the entry point is a constructor or a static method, the second argument of JPPFJob.addTask() be a Class object representing the class that declares it.
- an annotated method or constructor can have any signature, with no limit on the number of arguments
- through the task-wrapping mechanism, a JPPF-annotated class benefits from the JPPFTask facilities described in the previous section 4.1.1 , except for the callback methods onCancel(), onRestart() and onTimeout().

4.1.7 Running non-Java tasks: CommandLineTask

JPPF has a pre-defined task type that allows you to run an external process from a task. This process can be any executable program (including java), shell script or command. The JPPF API also provides a set of simple classes to access data, whether in-process or outside, local or remote.

The class that will allow you to run a process is [CommandLineTask](). Like [JPPFTask](), it is an abstract class that you must extend and whose `run()` method you must override.

This class provides methods to:

Setup the external process name, path, arguments and environment:

```
// list of commands passed to the shell
List<String> getCommandList()
void setCommandList(List<String> commandList)
void setCommandList(String... commands)

// set of environment variables
Map<String,String> getEnv()
void setEnv(Map<String, String> env)

// directory in which the command is executed
String getStartDir()
void setStartDir(String startDir)
```

You can also use the built-in constructors to do this at task initialization time:

```
CommandLineTask(Map<String, String> env, String startDir, String... commands)
CommandLineTask(String... commands)
```

Launch the process:

The process is launched by calling the following method from the `run()` method of the task:

```
void launchProcess()
```

Setup the capture of the process output:

You can specify and determine whether the process output (either standard or error console output) is or should be captured, and obtain the captured output:

```
boolean isCaptureOutput()

void setCaptureOutput(boolean captureOutput)

// corresponds to what is sent to System.out / stdout
String getErrorOutput()

// corresponds to what is sent to System.err / stderr
String getStandardOutput()
```

Here is a sample command line task that lists the content of a directory in the node's file system:

```
import org.jppf.server.protocol.*;

// This task lists the files in a directory of the node's host
public class ListDirectoryTask extends CommandLineTask
{
  // Execute the script
  public void run()
  {
    try
    {
      // get the name of the node's operating system
      String os = System.getProperty("os.name").toLowerCase();
      // the type of OS determines which command to execute
      if (os.indexOf("linux") >= 0)
      {
        setCommandList("ls", "-a", "/usr/local");
      }
      else if (os.indexOf("windows") >= 0)
      {
        setCommandList("cmd", "/C", "dir", "C:\\Windows");
      }
      // enable the capture of the console output
      setCaptureOutput(true);
      // execute the script/command
      launchProcess();
      // get the resulting console output and set it as a result
      String output = getStandardOutput();
      setResult(output);
    }
    catch(Exception e)
    {
      setException(e);
    }
  }
}
```

4.1.8 The Location API

This API allows developers to easily write data to, or read data from various sources: JVM heap, file system or URL.
It is based on he interface `Location`, which provides the following methods:

```
public interface Location<T>
{
  // Copy the content at this location to another location
  void copyTo(Location location);
  // Obtain an input stream to read from this location
  InputStream getInputStream();
  // Obtain an output stream to write to this location
  OutputStream getOutputStream():
  // Get this location's path
  T getPath();
  // Get the size of the data this location points to
  long size();
  // Get the content at this location as an array of bytes
  byte[] toByteArray() throws Exception;
}
```

Currently, JPPF provides 3 implementations of this interface:

- `FileLocation` represents a path in the file system
- `URLLocation` can be used to get data to and from a URL, including HTTP and FTP URLs
- `MemoryLocation` represents a block of data in memory that can be copied from or sent to another location

To illustrate the use of this API, let's transform our previous ListDirectoryTask in a way such that the output of the command is redirected to a file, instead of the console. We then read the content of this file and set it as the task's result:

```
import org.jppf.server.protocol.*;

// This task lists the files in a directory of the node's host
public class ListDirectoryTask extends CommandLineTask
{
  // Execute the script
  public void run()
  {
    try
    {
      String os = System.getProperty("os.name").toLowerCase();
      if (os.indexOf("linux") >= 0)
        // equivalent to shell command "ls -a /usr/local > output.txt"
        setCommandList("ls", "-a", "/usr/local", ">", "output.txt");
      else if (os.indexOf("windows") >= 0)
        // equivalent to shell command "dir C:\Windows > output.txt"
        setCommandList("cmd", "/C", "dir", "C:\\Windows", ">", "output.txt");
      // disable the capture of the console output
      setCaptureOutput(false);
      // execute the script or command
      launchProcess();
      // copy the resulting file in memory
      FileLocation fileLoc = new FileLocation("output.txt");
      MemoryLocation memoryLoc = new MemoryLocation((int) fileLoc.size());
      fileLoc.copyTo(memoryLoc);
      // set the file content as a result
      setResult(new String(memoryLoc.toByteArray()));
    }
    catch(Exception e)
    {
      setException(e);
    }
  }
}
```

4.2 Dealing with jobs

A job is a grouping of tasks with a common set of characteristics and a common SLA. These characteristics include:

- common data shared between tasks (data provider)
- A common Service Level Agreement (SLA) comprising:
  - the job priority
  - the maximum number of nodes a job can be executed on
  - an optional execution policy describing which nodes it can run on
  - a suspended indicator, that enables submitting a job in suspended state, waiting for an external command to resume or start its execution
  - an execution start date and time
- a blocking/non-blocking indicator, specifying whether the job execution is synchronous or asynchronous from the application's point of view
- a listener to receive notifications of completed tasks when running in non-blocking mode

In addition to these, each job has an identifier, that allows JPPF to manage and monitor the job while distinguishing it from other jobs. If this identifier is not explicitely specified, JPPF will create one as a string of 32 hexadecimal characters.

In the JPPF API, a job is represented by the class `JPPFJob`. In addition to accessors and mutators for the attributes we have seen above, `JPPFJob` provides methods to add tasks and a set of constructors that the make creation of jobs easier.

4.2.1 Creating a job

To create a job, the `JPPFJob` class offers a number of constructors, that can be split in 2 groups:

Constructors for blocking jobs

```
// creates a blocking job with no data provider and default SLA values
public JPPFJob()

// creates a blocking job with the specified data provider and default SLA values
public JPPFJob(DataProvider dataProvider)

// creates a blocking job with the specified data provider and SLA
public JPPFJob(DataProvider dataProvider, JPPFJobSLA jobSLA)
```

Constructors for non-blocking jobs

```
// creates a blocking job with the specified execution results listener,
// no data provider and default SLA values
public JPPFJob(TaskResultListener resultsListener)

// creates a blocking job with the specified execution results listener,
// data provider and default SLA values
public JPPFJob(DataProvider dataProvider, TaskResultListener resultsListener)

// creates a blocking job with the specified execution results listener,
// data provider and SLA
public JPPFJob(DataProvider dataProvider, JPPFJobSLA jobSLA,
               TaskResultListener resultsListener)
```

Basically, the distinction for a non-blocking job is made via the presence of a `TaskResultListener`.

Finally, there is a more generic constructor that embraces everything the other constructors do:

```
// creates a job with the specified data provider, SLA, blocking indicator
// and execution results listener
public JPPFJob(DataProvider dataProvider, JPPFJobSLA jobSLA, boolean blocking,
               TaskResultListener resultsListener)
```

No matter which constructor is used, the job id is automatically generated as a pseudo-random string of 32 hexadecimal characters. It can then be obtained or changed with the job's `getId()` and `setId(String)` methods. This mechanism ensures that a job always has an id, and that developers always have the possibility to change it to a more readable one.

4.2.2 Adding tasks to a job

As we have seen in section 4.1 about the various forms of tasks that we can use in JPPF, JPPFJob provides two methods to add tasks to a job.

Addding a JPPFTask, annotated, Runnable or Callable task

```
public JPPFTask addTask(Object taskObject, Object...args) throws JPPFException
```

The `taskObject` parameter can be one of the following:

- an instance of `JPPFTask`
- an instance of a class with a non-static public method annotated with `@JPPFRunnable`
- a `Class` object representing a class that has a public static method or a constructor annotated with `@JPPFRunnable`
- an instance of a a Runnable class
- an instance of a Callable class

The `args` parameter is optional and is only used to pass the arguments of a method or constructor annotated with `@JPPFRunnable`. It is ignored for all other forms of tasks.

The return value is an instance of (a subclass of) `JPPFTask`, regardless the type of task that is added. In the case of an annotated, Runnable or Callable task, the original task object, wrapped by this JPPFTask, can be retrieved using the method `JPPFTask.getTaskObject()`, as in the following example:

```
JPPFTask task = job.addTask(new MyRunnableTask());
MyRunnableTask runnableTask = (MyRunnableTask) task.getTaskObject();
```

As JPPF is using reflection to properly wrap the task, an eventual exception may be thrown. It will then be wrapped into a `JPPFException`.

Adding a POJO task

```
public JPPFTask addTask(String method, Object taskObject, Object...args)
  throws JPPFException
```

The `method` parameter is the name of the method or of the constructor to execute as the entry point of the task. In the case of a constructor, it must be the same as the name of the class.

The `taskObject` parameter can be one of the following:

- an instance of the POJO class if the entry point is a non-static method
- a `Class` object representing a POJO class that has a public static method or a constructor as entry point

The `args` parameter is optional and is used to pass the arguments of a method or constructor defined as the task's entry point.

As for the other form of this method, the return value is a `JPPFTask`, and the original task object can be retrieved using the method `JPPFTask.getTaskObject()`, as in the following example:

```
JPPFTask task = job.addTask("myMethod", new MyPOJOTask(), 3, "string");
MyPOJOTask pojoTask = (MyPOJOTask) task.getTaskObject();
```

As JPPF is using reflection to properly wrap the task, an eventual exception may be thrown. It will then be wrapped into a `JPPFException`.

4.2.3 Non-blocking jobs

Jobs can be submitted asynchonously from the application's perspective. This means that an asynchronous (or non-blocking) job will not block the application thread from which it is submitted. It also implies that we must have the means to obtain the execution results at a later time. To this effect, it is possible to register a listener with the job, to receive notifications when tasks have been completed and their results were returned to the application.

These listeners are represented by the interface [`TaskResultListener`](). It is defined as follows:

```
public interface TaskResultListener extends EventListener
{
  void resultsReceived(TaskResultEvent event);
}
```

We will thus be listening for events of type [`TaskResultEvent`](). Since JPPF 2.5, the public API for TaskResultEvent has has two non-deprecated methods, as illustrated below:

```
public class TaskResultEvent extends EventObject
{
  // Get the list of tasks whose results have been received from the server
  public List<JPPFTask> getTaskList();

  // Get the throwable eventually raised while receiving the results
  public Throwable getThrowable()
}
```

We can see that each notification is wrapping either a list of `JPPFTask` instances, or a Throwable to may be raised while receiving the results, both being mutually exclusive, mean that there is always one of the two methods that returns null..

Note that there is no guarantee the tasks are in the same order as when they were originally submitted. In fact the list is generally only a subset of the tasks that were submitted, and multiple notifications may be necessary to collect all the results. To restore the original ordering, `JPPFTask.getPosition()` should be used. Each task position is automatically calculated by JPPF at the time the job is submitted, as shown in the example below.

When the event's `Throwable` is non-null, this indicates that there was an issue with the connection to the driver. Using the built-in mechanism in the JPPF client, the client will (attempt to) reconnect and resubmit the job. This implies that you need to reset the state of the `TaskResultListener`, as if it were created anew. This is illustrated in the following example:

```
public class MyResultListener implements TaskResultListener
{
  // Initial count of tasks in the job
  private int initialCount = 0;

  // Count of results not yet received
  private int pendingCount = 0;

  // Sorted map containing the resulting tasks, ordered by ascending position
  private Map<Integer, JPPFTask> resultMap = new TreeMap<Integer, JPPFTask>();

  // Initialize this collector with a specified number of tasks
  public MyResultListener(int count)
  {
    initialCount = count;
    pendingCount = count;
  }

  // Notification that the results of a number of tasks have been received
  public void resultsReceived(TaskResultEvent event)
  {
    if (event.getThrowable() != null)
    {
      // Reset the state of this listener as if for a new job
      resultMap.clear();
      pendingCount = initialCount;
    }
    else
```

```
    {
      // Insert the tasks in the map, in ascending position order
      for (JPPFTask task: event.getTaskList()) resultMap.put(task.getPosition(), task);
      // Update the number of pending tasks accordingly
      pendingCount -= event.getTaskList().size();
    }
  }

  // Get the list of results
  public List<JPPFTask> getResults()
  {
    List<JPPFTask> results = new ArrayList<JPPFTask>();
    // collect all results received so far in ascending position order
    for (Integer n: resultMap.keySet()) results.add(resultMap.get(n));
    return results;
  }

  public boolean isJobComplete()
  {
    return pendingCount <= 0;
  }
}
```

We can then use our result listener as follows:

```
// create a job
JPPFJob myJob  = new JPPFJob();
// add 10 tasks
for (int i=0; i<10; i++) myJob.add(new MyTask(i));
// create a task result listener
MyResultListener myResultListener = new MyResultListener(10);
// register the listener with the job
myJob.setResultListener(myResultListener);
// set the job as non-blocking
myJob.setBlocking(false);

// submit the job
jppfClient.submit(job);

while (!myResultListener.isJobComplete())
{
  // ... do something while the job is executing ...
}

// once the job is complete, process the results
List<JPPFTask> results = myResultListener.getResults();
```

JPPF uses an existing implementation of *`TaskResultListener`*: the class `JPPFResultCollector`. It can be used directly, and its implementation is very similar to that of the code sample above,except that it provides a way to synchronize with the job execution through its `waitForResults()` method, so you don't have to write your own synchronization code.

4.3 Sharing data among tasks : the DataProvider API

After a job is submitted, the server will distribute the tasks in the job among the nodes of the JPPF grid. Generally, more than one task may be sent to each node. Given the communication and serialization protocols implemented in JPPF, objects referenced by multiple tasks at submission time will be deserialized as multiple distinct instances at the time of execution in the node. This means that, if n tasks reference object A at submission time, the node will actually deserialize multiple copies of A, with $Task_1$ referencing A_1, ... , $Task_n$ referencing A_n. We can see that, if the shared object is very large, we will quickly face memory issues.

To resolve this problem, JPPF provides a mechanism called *data provider* that enables sharing common objects among tasks in the same job. A data provider is an instance of a class that implements the interface [`DataProvider`](). Here is the definition of this interface:

```
public interface DataProvider extends Serializable
{
  Object getValue(Object key) throws Exception;

  void setValue(Object key, Object value) throws Exception;
}
```

This is indeed a basic object map interface: you can store objects and associate them with a key, then retrieve these objects using the associated key.

Here is an example of using a data provider:

In the application:

```
MyLargeObject myLargeObject = ...;
// create a data provider backed by a HashMap
DataProvider dataProvider = new MemoryMapDataProvider();
// store the shared object in the data provider
dataProvider.setValue("myKey", myLargeObject);
// associate the dataProvider with the job
JPPFJob = new JPPFJob(dataProvider);
job.add(new MyTask());
```

In the task:

```
public class MyTask extends JPPFTask
{
  public void run()
  {
    // get a reference to the data provider
    DataProvider dataProvider = getDataProvider();
    // retrieve the shared data
    MyLargeObject myLargeObject = (MyLargeObject) dataProvider.getValue("myKey");
    // ... use the data ...
  }
}
```

Note 1: *the association of a data provider to each task is done automatically by JPPF and is totally transparent to the application.*
Note 2: *from each task's perspective,the data provider should be considered read-only. Modifications to the data provider such as adding or modifying values, will NOT be propagated beyond the scope of the node. Hence, a data provider cannot be used as a common data store for the tasks. Its only goal is to avoid exessive memory consumption.*

In the next sub-sections, we will detail the existing implementations of `DataProvider` that exist in the JPPF API.

4.3.1 MemoryMapDataProvider: a map-based data provider

MemoryMapDataProvider is a very simple implementation of the `DataProvider` interface. It is backed by a `java.util.HashMap<Object, Object>`. The `getValue()` method is equivalent to a call to `HashMap.get()`, and the `setValue()` is equivalent to `HashMap.put()`.

4.3.2 ClientDataProvider: computing data in the client application

JPPF provides a way for a task to send a piece of code to be executed on the client and get the resulting data objects. This functionality is available through the use of a new DataProvider: ClientDataProvider.
The class ClientDataProvider extends MemoryMapDataProvider and adds one method:

```
public <V> Object computeValue(Object key, JPPFCallable<V> callable)
```

Here, `callable` is the equivalent of a callback that is sent to the client for execution, and whose result is stored in the DataProvider on the node side. The interface JPPFCallable<V> is defined as follows:

```
public interface JPPFCallable<V> extends Callable<V>, Serializable
{
}
```

Example use:

```
public class DataProviderTestTask extends JPPFTask
{
  public void run()
  {
    System.out.println("this should be on the node side");
    ClientDataProvider dataProvider = (ClientDataProvider) getDataProvider();
    // compute a value on the client side and store in the data provider
    Object o = dataProvider.computeValue("result", new MyCallable());
    System.out.println("Result of client-side execution:\n" + o);
    setResult(o);
    // retrieve the value without re-computing it
    Object o2 = dataProvider.getValue("result");
  }

  /**
   * A callable that simply prints a message on the client side
   * and returns the message to the node.
   */
  public static class MyCallable implements JPPFCallable<String>
  {
    public String call()
    {
      String s = "this should be on the client side";
      System.out.println(s);
      return s;
    }
  }
}
```

Here is the sequence of steps performed when calling the method `ClientDataProvider.computeValue()`:

- the `JPPFCallable` instance is sent to the client application
- the resulting value is computed as the return value of `JPPFCallable.call()`
- the resulting value is sent back to the node
- the value is stored in the data provider by calling `ClientDataProvider.setValue(key, value)`
- the value is returned as the result of `ClientDataProvider.computeValue()`

Once the value has been computed, it can be retrieved, without being computed again, by calling the method `ClientDataProvider.getValue()`. To compute a new value, `ClientDataProvider.computeValue()` should be called again.

4.4 Job Service Level Agreement

A job service level agreement (SLA) defines the terms and conditions in which a job will be processed by the server. The SLA specifies:

- the priority of a job
- whether it is submitted in suspended state
- the maximum number of nodes it can run on
- the characteristics of the nodes it can run on: the node execution policy
- the time at which a job is scheduled to start
- an expiration date for the job
- whether the job is a standard or broadcast job

A job SLA is represented by the class `JPPFJobSLA`. In addition to a default constructor, it also provides a number of convenient constructors to facilitate the creation of an SLA:

```
// Default constructor
public JPPFJobSLA()

// Initialize this job SLA with the specified execution policy
public JPPFJobSLA(ExecutionPolicy policy)

// Initialize this job SLA with the specified execution policy and priority
public JPPFJobSLA(ExecutionPolicy policy, int priority)

// Initialize this job SLA with the specified execution policy,
// priority, max number of nodes and suspended indicator
public JPPFJobSLA(ExecutionPolicy policy, int priority, int maxNodes,
                  boolean suspended)
```

Each attribute of the SLA also has a corresponding setter and getter:

```
// execution policy
public ExecutionPolicy getExecutionPolicy()
public void setExecutionPolicy(ExecutionPolicy executionPolicy)

// job priority
public int getPriority()
public void setPriority(int priority)

// maximum number of nodes
public int getMaxNodes()
public void setMaxNodes(int maxNodes)

// suspended indicator
public boolean isSuspended()
public void setSuspended(boolean suspended)

// job startup schedule
public JPPFSchedule getJobSchedule()
public void setJobSchedule(JPPFSchedule jobSchedule)

// job expiration schedule
public JPPFSchedule getJobExpirationSchedule()
public void setJobExpirationSchedule(JPPFSchedule jobSchedule)

// broadcast indicator
public boolean isBroadcastJob()
public void setBroadcastJob(boolean broadcastJob)
```

4.4.1 Execution policy

An execution policy is an object that determines whether a particular set of JPPF tasks can be executed on a JPPF node. It does so by applying the set of rules (or tests) it is made of, against a set of properties associated with the node.

The properties of the node include:

- JPPF configuration properties
- System properties (including -D*=* properties specified on the JVM command line)
- Environment variables (e.g. PATH, JAVA_HOME, etc.)
- Networking: list of ipv4 and ipv6 addresses with corresponding host name when it can be resolved
- Runtime information such as maximum heap memory, number of available processors, etc...
- Disk space and storage information (JDK 1.6 or later only)

The kind of tests that can be performed apply to the value of a property, and include:

- Binary comparison operators: ==, <, <=, >, >= ; for instance: "property_value <= 15"
- Range operators (intervals): "property_value in" [a,b] , [a,b[,]a,b] ,]a,b[
- "One of" operator (discrete sets): "property_value in { a1, ... , aN }"
- "Contains string" operator: "property_value contains "substring""
- Regular expressions: " property_value matches 'regexp' "
- Custom, user-defined tests

The tests can also be combined into complex expressions using the boolean operators NOT, AND, OR and XOR.

Using this mechanism, it is possible to write an execution policy such as:
"Execute on a node only if the node has at least 256 MB of memory and at least 2 CPUs available"

An execution policy is sent along with the tasks to the JPPF driver, and evaluated by the driver. They do not need to be sent to the nodes.

For a detailed and complete description of all policy elements, operators and available properties, please refer to the chapter ***Appendix B: Execution policy reference***.

4.4.1.1 Creating and using an execution policy

An execution policy is an object whose type is a subclass of `ExecutionPolicy`. It can be built in 2 ways:

By API, using the classes in the `org.jppf.node.policy` package.

Example:

```
// define a policy allowing only nodes with 2 processing threads or more
ExecutionPolicy atLeast2ThreadsPolicy = new AtLeast("processing.threads", 2);
// define a policy allowing only nodes that are part of the "mydomain.com"
// internet domain (case ignored)
ExecutionPolicy myDomainPolicy =
  new Contains("ipv4.addresses", true, "mydomain.com");
// define a policy that requires both of the above to be satisfied
ExecutionPolicy myPolicy = atLeast2ThreadsPolicy.and(myDomainPolicy);
```

Alternatively, this could be written in a single statement:

```
// define the same policy in one statement
ExecutionPolicy myPolicy = new AtLeast("processing.threads", 2).and(
  new Contains("ipv4.addresses", true, "mydomain.com"));
```

Using an XML policy document:

Example XML policy:

```
<ExecutionPolicy>
  <!-- define a policy that requires both rules to be satisfied -->
  <AND>
    <!-- define a policy allowing only nodes with 2 processing threads or more -->
    <AtLeast>
      <Property>processing.threads</Property>
      <Value>2</Value>
    </AtLeast>
    <!-- allow only nodes in the "mydomain.com" internet domain (case ignored) -->
    <Contains ignoreCase="true">
      <Property>ipv4.addresses</Property>
      <Value>mydomain.com</Value>
    </Contains>
  </AND>
</ExecutionPolicy>
```

As you can see, this is the exact equivalent of the policy we constructed programmatically before.

To transform this XML policy into an `ExecutionPolicy` object, we will have to parse it using the `PolicyParser` API, by the means of one of the following methods:

```
static ExecutionPolicy parsePolicy(String)      // parse from a string
static ExecutionPolicy parsePolicyFile(String)  // parse from a file
static ExecutionPolicy parsePolicy(File)        // parse from a file
static ExecutionPolicy parsePolicy(Reader)      // parse from a Reader
static ExecutionPolicy parsePolicy(InputStream) // parse from an InputStream
```

Example use:

```
// parse the specified XML file into an ExecutionPolicy object
ExecutionPolicy myPolicy = PolicyParser.parsePolicyFile("../policies/MyPolicy.xml");
```

It is also possible to validate an XML execution policy against the JPPF Execution Policy schema using one of the `validatePolicy()` methods of `PolicyParser`:

```
static ExecutionPolicy validatePolicy(String)      // validate from a string
static ExecutionPolicy validatePolicyFile(String)  // validate from a file
static ExecutionPolicy validatePolicy(File)        // validate from a file
static ExecutionPolicy validatePolicy(Reader)      // validate from a Reader
static ExecutionPolicy validatePolicy(InputStream) // validate from an InputStream
```

To enable validation, the document's namespace must be specified in the root element:

```
<jppf:ExecutionPolicy xmlns:jppf="http://www.jppf.org/schemas/ExecutionPolicy.xsd">
  ...
</jppf:ExecutionPolicy>
```

Example use:

```
public ExecutionPolicy createPolicy(String policyPath)
{
  try
  {
    // validate the specified XML file
    PolicyParser.validatePolicyFile(policyPath);
  }
  catch(Exception e)
  {
    // the validation and parsing errors are in the exception message
    System.err.println("The execution policy " + policyPath +
      " is not valid: " + e.getMessage());
   return null;
  }
  // the policy is valid, we can parse it safely
  return PolicyParser.parsePolicyFile(policyPath);
}
```

4.4.1.2 Creating custom policies

It is possible to apply user-defined policies. When you do so, a number of constraints must be respected:

- the custom policy class must extend CustomPolicy
- the custom policy class must be deployed in the JPPF server classpath as well as the client's

Here is a sample custom policy code:

```
package my.package;
import org.jppf.management.JPPFSystemInformation;
import org.jppf.node.policy.CustomPolicy;

// define a policy allowing only nodes with 2 processing threads or more
public class MyCustomPolicy extends CustomPolicy
{
  public boolean accepts(JPPFSystemInformation info)
  {
    // get the value of the "processing.threads" property
    String s = this.getProperty(info, "processing.threads");
    int n = -1;
    try { n = Integer.valueOf(s); }
    catch(NumberFormatException e) { // process the exception }
    // node is accepted only if number of threads >= 2
    return n >= 2;
  }
}
```

Now, let's imagine that we want our policy to be more generic, and to accept nodes with at least a parametrized number of threads given as argument to the policy.

Our accepts() method becomes then:

```
public boolean accepts(JPPFSystemInformation info)
{
  // get the value to compare with, passed as the first argument to this policy
  String s1 = getArgs()[0];
  int param = -1;
  try { param = Integer.valueOf(s1); }
  catch(NumberFormatException e) { }
  String s2 = getProperty(info, "processing.thread");
  int n = -1;
  try { n = Integer.valueOf(s2); }
  catch(NumberFormatException e) { }
  // node is accepted only if number of threads >= param
  return n >= param;
}
```

Here we use the `getArgs()` method which returns an array of strings, corresponding to the arguments passed in the XML representation of the policy.

To illustrate how to use a custom policy in an XML policy document, here is an example XML representation of the custom policy we created above:

```
<CustomRule class="my.package.MyCustomPolicy">
  <Arg>3</Arg>
</CustomRule>
```

The "class" attribute is the fully qualified name of the custom policy class. There can be any number of <Arg> elements, these are the parameters that will then be accessible through `CustomPolicy.getArgs()`.

When the XML descriptor is parsed, an execution policy object will be created in a way that is exactly equivalent to this code snippet:

```
MyCustomPolicy policy = new MyCustomPolicy();
policy.setArgs( "3" );
```

4.4.2 Job start and expiration scheduling

It is possible to schedule a job for a later start, and also to set a job for expiration at a specified date/time. The job SLA allows this by providing the following methods:

```
// job start schedule
public JPPFSchedule getJobSchedule()
public void setJobSchedule(JPPFSchedule schedule)

// job expiration schedule
public JPPFSchedule getJobExpirationSchedule()
public void setJobExpirationSchedule(JPPFSchedule schedule)
```

As we can see, this is all about getting and setting an instance of `JPPFSchedule`. A schedule is normally defined through one of its constructors:

As a fixed length of time

```
public JPPFSchedule(long duration)
```

The semantics is that the job will start *duration* milliseconds after the job is received by the server.
Here is an example:

```
JPPFJob myJob = new Job();
// set the job to start 5 seconds after being received
JPPFSchedule mySchedule = new JPPFSchedule(5000L);
myJob.getJobSLA().setJobSchedule(mySchedule);
```

As a specific date/time

```
public JPPFSchedule(String date, String dateFormat)
```

Here the date format is specified as a pattern for a `SimpleDateFormat` instance.

Here is an example use of this constructor:

```
JPPFJob myJob = new Job();
String dateFormat = "MM/dd/yyyy hh:mm a z";
// set the job to expire on September 30, 2010 at 12:08 PM in the CEDT time zone
JPPFSchedule schedule = new JPPFSchedule("09/30/2010 12:08 PM CEDT", dateFormat);
myJob.getJobSLA().setJobExpirationSchedule(mySchedule);
```

4.4.3 Broadcast jobs

A broadcast job is a specific type of job, for which each task will be be executed on all the nodes currently present in the grid. This opens new possibilities for grid applications, such as performing maintenance operations on the nodes or drastically reducing the size of a job that performs identical tasks on each node.

With regards to the job SLA, a job is set in broadcast mode via a boolean indicator, for which the class `JPPFJobSLA` provides the following accessors:

```
public boolean isBroadcastJob()
public void setBroadcastJob(boolean broadcastJob)
```

To set a job in broadcast mode:

```
JPPFJob myJob = new JPPFJob();
myJob.getJobSLA().setBroadcastJob(true);
```

With respect to the dynamic aspect of a JPPF grid, the following behavior is enforced:

- a broadcast job is only executed on the nodes active at the time the job is received by the JPPF driver
- if a node dies or disconnects while the job is executing on it, the job is canceled for this node
- if a new node connects while the job is executing, the broadcast job will not execute on it
- a broadcast job does not return any results, i.e. it returns the tasks in the same state as they were submitted

4.5 Job Metadata

It is possible to attach user-defined metadata to a job, to describe the characteristics of the job and its tasks. This additional data can then be reused by customized load-balancing algorithms, to perform load balancing based on knowledge about the jobs. For instance, the metadata could provide information about the memory footprint of the tasks and about their duration, which can be critical data for the server, in order to determine on which nodes the job or tasks should be executed.

The job metadata is encapsulated in a specific class: `JPPFJobMetadata`, and can be accessed from the job as follows:

```
JPPFJob job = ...;
JPPFJobMetadata metaData = job.getJobMetadata();
```

`JPPFJobMetadata` provides the following methods:

```
// Set a parameter in the metadata
public void setParameter(Object key, Object value)

// Retrieve a parameter in the metadata
public Object getParameter(Object key)

// Retrieve a parameter in the metadata
public Object getParameter(Object key, Object defaultValue)

// Remove a parameter from the metadata
public Object removeParameter(Object key)

// Get a copy of the metadata map
public Map<Object, Object> getAll()
```

Here is an example use:

```
JPPFJob job = ...;
JPPFJobMetadata metaData = job.getJobMetadata();
// set the memory footprint of each task to 10 KB
metaData.setParameter("task.footprint", "" + (10 * 1024));
// set the duration of each task to 80 milliseconds
metaData.setParameter("task.duration", "80");
```

Related sample: "CustomLoadBalancer" in the JPPF samples pack.

4.6 The JPPFClient API

A JPPF client is an object that will handle the communication between the application and the server. Its role is to:

- manage one or multiple connections with the server
- submit jobs and get their results
- handle notifications of job results
- manage each connection's life cycle events
- provide the low-level machinery on the client side for the distributed class loading mechanism
- provide an access point for the management and monitoring of each server

A JPPF client is represented by the class `JPPFClient`. We will detail its functionalities in the next sub-sections.

4.6.1 Creating and closing a JPPFClient

A JPPF client is a Java object, and is created via one of the constructors of the class `JPPFClient`. Each JPPF client has a unique identifier that is always transported along with any job that is submitted by this client. This identifier is what allows JPPF to know from where the classes used in the tasks should be loaded. In effect, each node in the grid will have a map of each client identifier with a unique class loader, creating the class loader when needed. The implication is that, if a new client identifier is specified, the classes used in any job / task submiotted by this client will be dynamically reloaded. This is what enables the immediate dynamic redeployment of code changes in the application. On the other hand, if a previously existing identifier is reused, then no dynamic redeployment occurs, and code changes will be ignored (i.e. the classes already loaded by the node will be reused), even if the application is restarted between 2 job submissions.

There are two forms of constructors for `JPPFClient`, each with a specific corresponding semantics:

Generic constructor with automatic identifier generation

```
public JPPFClient()
```

When using this constructor, JPPF will automatically create a universal unique identifier (uuid) that is guaranteed to be unique on the grid. The first submission of a job will cause the classes it uses to be dynamically loaded by any node that executes the job.

Constructor specifying a user-defined client identifier

```
public JPPFClient(String uuid)
```

In this case, the classes used by a job will be loaded only the first time they are used, including if the application has been restarted in the meantime, or if the JPPF client is created from a separate application. This behavior is more adapted to an application deployed in production, where the client identifier would only change when a new version of the application is deployed on the grid. It is a good practice to include a version number in the identifier.

As a `JPPFClient` uses a number of system and network resources, it is recommended to use it as a singleton. It is designed for concurrent use by multiple threads, which makes it safe for use with a singleton pattern. It is also recommended to release these resources when they are no longer needed, via a call to the `JPPFClient.close()` method. The following code sample illustrates what is considered a best practice for using a `JPPFClient`:

```
public class MyApplication
{
  // singleton instance of the JPPF client
  private static JPPFClient jppfClient = new JPPFClient();

  // allows access to the client from any other class
  public static JPPFClient getJPPFClient()
  {
    return jppfClient;
  }

  public static void main(String...args)
  {
    // enclosed in a try / catch to ensure resources are properly released
    try
    {
      jppfClient = new JPPFClient();

      // ... application-sepcific code here ...
    }
    finally
    {
      // close the client to release its resources
      if (jppfClient != null) jppfClient.close();
    }
  }
}
```

4.6.2 Submitting a job

To submit a job, JPPFClient provides a single method:

```
public List<JPPFTasks> submit(JPPFJob job)
```

This method has two different behaviors, depending on whether the job is blocking or non-blocking:

- blocking job: the `submit()` method blocks until the job execution is complete. The return value is a list of tasks with their results, in the same order as the tasks that were added to the job.
- non-blocking job: `submit()` returns immediately with a null value. It is up to the developer to collect the execution results by the means of a `TaskResultListener` set onto the job (see section 4.2.3).

4.6.3 Exploring the server connections

The JPPF client handles one or more connections to one or multiple servers. Each individual connection is represented as an instance of the interface [`JPPFClientConnection`]. It is possible to explore these connections using the following methods in `JPPFClient`:

```java
// Get all the client connections handled by this JPPFClient
public List<JPPFClientConnection> getAllConnections()

// Get the names of all the client connections handled by this JPPFClient
public List<String> getAllConnectionNames()

// Get a connection given its name
public JPPFClientConnection getClientConnection(String name)
```

4.6.4 Receiving notifications for new server connections

The JPPF client emits an event each time a new connection is established with a server. It is possible to receive these events by registering an implementation of the listener interface `ClientListener` with the client. Since the connections are generally established during the initialization of the client, i.e. when calling its constructor, `JPPFClient` provides a different form of the two constructors we have seen in section 4.6.1 :

```java
// Initialize with the specified listeners and a generated uuid
public JPPFClient(ClientListener...listeners)
// Initialize with the specified listeners and user-defined uuid
public JPPFClient(String uuid,  ClientListener...clientListeners)
```

It is also possible to add and remove listeners using these two more "conventional" methods:

```java
// register a listener with this client
public void addClientListener(ClientListener listener)
// remove a listener from the registered listeners
public synchronized void removeClientListener(ClientListener listener)
```

Here is a sample `ClientListener` implementation:

```java
public class MyClientListener implements ClientListener
{
  public void newConnection(ClientEvent event)
  {
    // the new connection is the source of the event
    JPPFClientConnection connection = event.getConnection();
    System.out.println("New connection with name " + connection.getName());
  }
}

ClientListener  myClientListener =  new MyClientListener();
// initialize the client and register the listener
JPPFClient jppfClient = new JPPFClient(myClientListener);
```

4.6.5 Receiving status notifications for existing server connections

Each individual server connection has a status that depends on the state of its network connection to the server and whether it is executing a job request. A connection status is represented by the enum `JPPFClientConnectionStatus`, and has the following possible values: NEW, DISCONNECTED, CONNECTING, ACTIVE, EXECUTING or FAILED.

JPPFConnection extends the interface ClientConnectionStatusHandler, which provides the following methods to handle the connection status and register or remove listeners:

```java
public interface ClientConnectionStatusHandler
{
  // Get the status of this connection
  JPPFClientConnectionStatus getStatus();
  // Set the status of this connection
  void setStatus(JPPFClientConnectionStatus status);
  // Register a connection status listener with this connection
  void addClientConnectionStatusListener(ClientConnectionStatusListener listener);
  // Remove a connection status listener from the registered listeners
  void removeClientConnectionStatusListener(ClientConnectionStatusListener listener);
}
```

Here is a sample status listener implementation:

```
public class MyStatusListener extends ClientConnectionStatusListener
{
  public void statusChanged(ClientConnectionStatusEvent event)
  {
    // obtain the client connection from the event
    JPPFClientConnection connection =
      (JPPFClientConnection) event.getClientConnectionStatusHandler();
    // get the new status
    JPPFClientConnectionStatus status = connection.getStatus();
    System.out.println("Connection " + connection.getName() + " status changed to "
      + status);
  }
}
```

To put all of this together, let's take the sample listener from the previous section 4.6.4 and modify it to add a status listener to each new connection:

```
public MyClientListener implements ClientListener
{
  public void newConnection(ClientEvent event)
  {
    // the new connection is the source of the event
    JPPFClientConnection connection = event.getConnection();
    System.out.println("New connection with name " + connection.getName());
    // register to receive staus event on the new connection
    connection.addClientConnectionStatusListener(new MyStatusListener());
  }
}
```

4.6.6 Switching local execution on or off

The `JPPFClient` API allows users to dynamicallly turn the local (in the client JVM) execution of jobs on or off, and determine whether it is active or not. This is done via these two methods:

```
// Determine whether local execution is enabled on this client
public boolean isLocalExecutionEnabled()

// Specifiy whether local execution is enabled on this client
public void setLocalExecutionEnabled(boolean localExecutionEnabled)
```

Turning local execution on or off will affect the next job to be executed, but not any that is currently executing.

4.7 JPPF Executor Services

4.7.1 Basic usage

JPPF 2.2 introduced a new API, that serves as an `ExecutorService` facade to the JPPF client API. This API consists in a simple class: `JPPFExecutorService`, implementing the interface `java.util.concurrent.ExecutorService`.

A JPPFExecutorService is obtained via its constructor, to which a JPPFClient must be passed:

```
JPPFClient jppfClient = new JPPFClient();
ExecutorService executor = new JPPFExecutorService(jppfClient);
```

The behavior of the resulting executor will depend largely on the configuration of the JPPFClient and on which ExecutorService method you invoke to submit tasks. In effect, each time you invoke an `invokeAll(...)`, `invokeAny(...)`, `submit(...)` or `execute(...)` method of the executor, a new JPPFJob will be created and sent for execution on the grid. This means that, if the executor method you invoke only takes a single task, then a job with only one task will be sent to the JPPF server.

Here is an example use:

```
JPPFClient jppfClient = new JPPFClient();
ExecutorService executor = new JPPFExecutorService(jppfClient);

try
{
  // submit a single task
  Runnable myTask = new MyRunnable(0);
  Future<?> future = executor.submit(myTask);
  // wait for the results
  future.get();
  // process the results
  ...

  // submit a list of tasks
  List<Runnable> myTaskList = new ArrayList<Runnable>;
  for (int i=0; i<10; i++) myTaskList.add(new MyRunnable(i));
  List<Future<?>> futureList = executor.invokeAll(myTaskList);
  // wait for the results
  for (Future<?> future: futureList) future.get();
  // process the results for the list of tasks
  ...
}
finally
{
  // clean up after use
  executor.shutdown();
  jppfClient.close();
}

// !!! it is important that this task is Serializable !!!
public static class MyRunnable implements Runnable, Serializable
{
  private int id = 0;

  public MyRunnable(int id)
  {
    this.id = id;
  }

  public void run()
  {
    System.out.println("Runing task id " + id);
  }
}
```

4.7.2 Batch modes

The executor's behavior can be modified by using one of the batch modes of the `JPPFExecutorService`. By batch mode, we mean the ability to group tasks into batches, in several different ways. This enables tasks to be sent together, even if they are submitted individually, and allows them to benefit from the parallel features inherent to JPPF. This will also dramatically improve the thorughput of individual tasks sent via an executor service.

Using a batch size: specifying a batch size via the method [`JPPFExecutorService.setBatchSize(int limit)`]() causes the executor to only send tasks when at least that number of tasks have been submitted. When using this mode, you must be cautious as to how many tasks you send via the executor: if you send less than the batch size, these tasks will remain pending and un-executed. Sometimes, the executor will send more than the specified number of tasks in the same batch: this will happen in the case where one of the `JPPFExecutorService.invokeXXX()` method is called with n tasks, such that *current batch size* + *n* > *limit*. The behavior is to send all tasks included in the `invokeXXX()` call together.

Here is an example:

```
JPPFExecutorService executor = new JPPFExecutorService(jppfClient);
// the executor will send jobs with at least 5 tasks each
executor.setBatchSize(5);
List<Future<?>> futures = new ArrayList<Future<?>>();
// we submit 10 = 2 * 5 tasks, this will cause the client to send 2 jobs
for (int i=0; i<10; i++) futures.add(executor.submit(new MyTask(i)));
for (Future<?> f: futures) f.get();
```

Using a batch timeout: this is done via the method [`JPPFExecutorService.setBatchTimeout(long timeout)`]() and causes the executor to send the tasks at regular intervals, specified as the timeout. The timeout value is expressed in milliseconds. Once the timeout has expired, the counter is reset to zero. If no task has been submitted between two timeout expirations, then nothing happens.

Example:

```
JPPFExecutorService executor = new JPPFExecutorService(jppfClient);
// the executor will send a job every second (if any task is submitted)
executor.setBatchTimeout(1000L);
List<Future<?>> futures = new ArrayList<Future<?>>();
// we submit 5 tasks
for (int i=0; i<5; i++) futures.add(executor.submit(new MyTask(i)));
// we wait 1.5 second, during that time a job with 5 tasks will be submitted
Thread.sleep(1500L);
// we submit 6 more tasks, they will be sent in a different job
for (int i=5; i<11; i++) futures.add(executor.submit(new MyTask(i)));
// here we get the results for tasks sent in 2 different jobs!
for (Future<?> f: futures) f.get();
```

Using both batch size and timeout: it is possible to use a combination of batch size and timeout. In this case, a job will be sent whenever the batch limit is reached or the timeout expires, whichever happens first. In any case, the tiemout counter will be reset each time a job is sent. Using a timeout is also an efficient way to deal with the possible blocking behavior of the batch size mode. In this case, just use a timeout that is sufficently large for your needs.

Example:

```
JPPFExecutorService executor = new JPPFExecutorService(jppfClient);
executor.setBatchTimeout(1000L);
executor.setBatchSize(5);
List<Future<?>> futures = new ArrayList<Future<?>>();
// we submit 3 tasks
for (int i=0; i<3; i++) futures.add(executor.submit(new MyTask(i)));
// we wait 1.5 second, during that time a job with 3 tasks will be submitted,
// even though the batch size is set to 5
Thread.sleep(1500L);
for (Future<?> f: futures) f.get();
```

4.8 The JPPF configuration API

The JPPF configuration properties are accessible at runtime, via a static method call: `JPPFConfiguration.getProperties()`. This method returns an object of type `TypedProperties`, which is an extension of `java.util.Properties` with additional methods to handle properties with primitive values: `boolean`, `int`, `long`, `float` and `double`.

Here is a summary of the API provided by `TypedProperties`:

```
// constructors
public TypedProperties()
// initialize with existing key/value pairs from a map
public TypedProperties(Map<Object, Object> map)
// string properties
public String getString(String key)
public String getString(String key, String defValue)
// int properties
public int getInt(String key)
public int getInt(String key, int defValue)
// long properties
public long getLong(String key)
public long getLong(String key, long defValue)
// float properties
public float getFloat(String key)
public float getFloat(String key, float defValue)
// double properties
public double getDouble(String key)
public double getDouble(String key, double defValue)
// boolean properties
public boolean getBoolean(String key)
public boolean getBoolean(String key, boolean defValue)
// properties that are the path to another properties file
public TypedProperties getProperties(String key)
public TypedProperties getProperties(String key, TypedProperties defValue)
```

As you can see, each getXXX() method has a corresponding method that takes a default value, to be returned if the property is not defined for the specified key.

You will also notice the last two methods `getProperties(...)`, which are special in the sense that they do not handle simple value types, but rather specify the path to another properties file, whose content is returned as a `TypedProperties` instance. They are convenience methods that allow an easy navigation into a hierarchy of configuration files. The lookup mechanism for the specified properties file is described in the Javadoc for `TypedProperties.getProperties(java.lang.String)`.

It is possible to alter the JPPF configuration, via a call to the method `setProperty(String, String)` of `java.util.Properties`. Notes that in this case, the value must be specfied as a string. If you wish to programatically change one or more JPPF configuration properties, then it should be done before they are used. For instance, in a client application, it should be done before the JPPF client is initialized, as in this sample code:

```
// get the configuration
TypedProperties props = JPPFConfiguration.getProperties();

// set the connection properties programatically
props.setProperty("jppf.discovery.enabled", "false");
props.setProperty("jppf.drivers", "driver1");
props.setProperty("driver1.jppf.server.host", "www.myhost.com");
props.setProperty("driver1.class.server.port", "11111");
props.setProperty("driver1.app.server.port", "11112");

// now our configuration will be used
JPPFClient client = new JPPFClient();
```

5 Configuration guide

A JPPF grid is composed of many distributed components interacting with each other, often in different environments. While JPPF will work in most environments, the default behavior may not be appropriate or adapted to some situations. Much of the behavior in JPPF components can thus be modified, fine-tuned or sometimes even disabled, via numerous configuration properties. These properties apply to many mechanisms and behaviors in JPPF, including:

- network communication
- management and monitoring
- performance / load-balancing
- failover and recovery

Any configuration property has a default value that is used when the property is not specified, and which should work in most environments. In practice, this means that JPPF can work without any explicitely specified configuration at all.

For a full list of the JPPF configuration properties, do not hesitate to read the chapter ***Appendix A: configuration properties reference*** of this manual.

5.1 Configuration file specification and lookup

All JPPF components work with a set of configuration properties. The format of these propertiesis as specified in the Java Properties class. To enable a JPPF component to retrieve thiese properties file, their source must be specified using one of the two, mutually exclusive, system properties:

– `jppf.config.plugin = class_name`, where *class_name* is the fully qualified name of a class implementing the interface JPPFConfiguration.ConfigurationSource, enabling a configuration source from any origin, such as a URL, a distributed file system, a remote storage facility, a database, etc.
– `jppf.config = path`, where *path* is the location of the configuration file, either on the file system, or relative to the JVM's classpath root. If this system property is not specified, JPPF will look for a default file named "jppf.properties" in the current directory or in the classpath root.

Example use:

```
java -Djppf.config.plugin=my.own.Configuration ...
```

or

```
java -Djppf.config=my/folder/myFile.properties ...
```

The configuration file lookup mechanism is as follows:

1. if `jppf.plugin.config` is specified
   a) instantiate an object of the specified class name and read the properties via the stream provided by this object's `getPropertyStream()` method.
   b) if, for any reason, the stream cannot be obtained or reading the properties from it fails, go to 3.

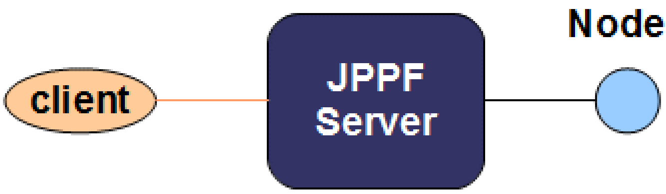
2. else if `jppf.config` is specified
   a) look for the file in the file system
   b) if not found in the file system, look in the classpath
   c) if not found in the classpath use default configuration values

3. if `jppf.config` is not specified
   a) use default file "`jppf.properties`"
   b) look for "jppf.properties" in the file system
   c) if not found in the file system, look for it in the classpath
   d) if not found in the classpath use default configuration values

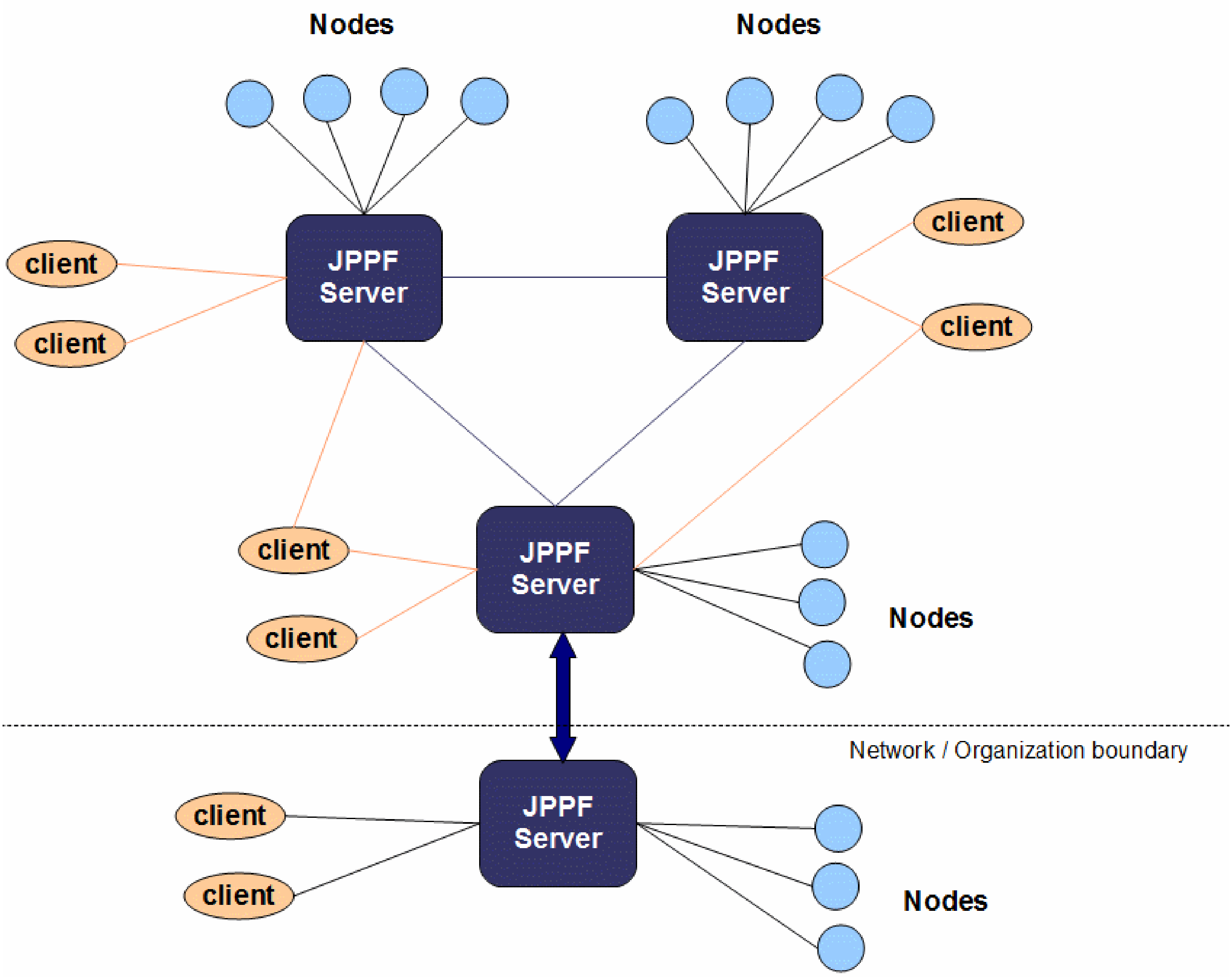
A practical side effect of this mechanism is that it allows us to place a configuration file in the classpath, for instance packaged in a jar file, and override it if needed with an external file, since the the file system is always looked up first.

5.2 Reminder: JPPF topology

Before reviewing the details of each configuration property, it is useful to have the big picture of what we are configuring exactly. In a few words, a JPPF grid is made of clients (instances of your applications), servers and nodes. An application submits jobs to a server, and a server distributes the tasks in each job to its attached nodes. The simplest configuration you can have would be as illustrated in this picture:



We can see here that we have a single client communicating with one server, to which a single node is attached. In practice, there will be many nodes attached to the server, and many clients will be able to communicate with the server concurrently. It is also possible to link servers together, forming a peer-to-peer network of JPPF servers, allowing servers to delegate a part of their workload to other servers.
We could effectively build a much more complex JPPF network, such as the one in this picture:



The role of the configuration will be essentially to determine where each component can find the others, and how their interactions will be processed.

5.3 Configuring a JPPF server

5.3.1 Basic network configuration

The server network communication mechanism uses TCP/IP. To do its basic work of receiving jobs and dispatching them for execution, 3 TCP ports are required:

- one port is reserved for the distributed class loader, and is also used by clients, nodes and other servers
- another port is reserved for the communication with the clients, to receive job requests and send the results
- the third port is reserved to communication with the nodes, and is used to dispatch jobs and tasks to the nodes, and receive the results of their execution. It is also used to delegate jobs to other servers.

In the configuration file, these properties would be defined as follows, with their default values:

```
# class loader port
class.server.port = 11111

# communication with the client applications
app.server.port = 11112

# communication with the nodes
node.server.port = 11113
```

Note that not defining these properties is equivalent to assigning them their default value.

5.3.2 Server discovery

By default, JPPF nodes and clients are configured to automatically discover active servers on the network. This is made possible because, by default, a JPPF server will broadcast the required information (i.e. host address and port numbers) using the UDP multicast mechanism. This mechanism itself is configurable, by setting the following properties:

```
# Enable or disable automatic discovery of JPPF drivers
jppf.discovery.enabled = true

# UDP multicast group to which drivers broadcast their connection parameters
jppf.discovery.group = 230.0.0.1

# UDP multicast port to which drivers broadcast their connection parameters
jppf.discovery.port = 11111

# How long a driver should wait between 2 broadcasts, in milliseconds
jppf.discovery.broadcast.interval = 1000
```

The values indicated above are the default values. Note that, given the nature of the UDP protocol, the broadcast data is transient, and has to be re-sent at regular intervals to allow new nodes or clients to find the information. The broadcast interval property allows some control over the level of network traffic ("chattyness") thus generated.

5.3.3 Connecting to other servers

We have seen in section 5.2 that servers can connect to each other, up to a full-fledged peer-to-peer topology. When a server A connects to another server B, A will act as a node attached to B (from B's perspective). Based on this, there are 4 possible kinds of connectivity between 2 servers:

- A and B are not connected at all
- A is connected to B (i.e. A acts as a node attached to B)
- B is connected to A (i.e. B acts as a node attached to A)
- A and B are connected to each other

There are 2 ways to define a connection from a server to other servers on the network:

Using automatic discovery

In this scenario, we must enable the discovery of peer servers:

```
# Enable or disable auto-discovery of other peer servers (defaults to false)
jppf.peer.discovery.enabled = true
```

For this to work, the server broadcast must be enabled on the peer server(s), and the properties defined in the previous section 5.3.2 will be used, hence they must be set to the same values on the other server(s). A server can discover other servers without having to broadcast its own connection information (i.e. without being "discoverable").

Please note that the default value for the above property is "false". Setting the default to "true" would imply that each server would connect to all other servers accessible on the network, with a high risk of unwanted side effects.

Manual connection to peer servers

This will be best illustrated with an example configuration:

```
# define a space-separated list of peers to connect to
jppf.peers = server_1 server_2

# connection to server_1
jppf.peer.server_1.server.host = host_1
class.peer.server_1.server.port = 11111
node.peer.server_1.server.port = 11113
# connection to server_2
jppf.peer.server_2.server.host = host_2
class.peer.server_2.server.port = 11111
node.peer.server_2.server.port = 11113
```

To connect to each peer, we must define its IP address or host name as well as two port numbers: one for the server's distributed class loader and the other for server to dispatch jobs. Please note that the value we have defined for "`class.peer.server_1.server.port`" must be the same as the one defined for "`class.server.port`" in server_1's configuration, and the value for "`node.peer.server_1.server.port`" must be equal to that of "`node.server.port`" in server_1's configuration.

5.3.4 Socket connections idle timeout

In some environments, a firewall may be configured to automatically close socket connections that have been idle for more than a specified time. This may lead to a situation where a server may be unaware that a node or client was disconnected, and cause one or more jobs to never return. To remedy to that situation, it is possible to configure an idle timeout on either side of the connection, so that the connection can be closed cleanly and grid operations can continue unhindered. This is done via the following property:

```
jppf.socket.max-idle = timeout_in_seconds
```

If the timeout value is less than 10 seconds, then it is considered as no timeout. The default value is -1.

5.3.5 JMX management configuration

JPPF uses JMX to provide remote management capabilities for the servers, and uses the default RMI connector for communication. Each server has its own embedded RMI registry.

The management features are enabled by default; this behavior can be changed by setting the following property:

```
# Enable or disable management of this server
jppf.management.enabled = true
```

When management is enabled, the following properties must be defined:

```
# JMX management host IP address. If not specified (recommended), the first non-local
# IP address (i.e. neither 127.0.0.1 nor localhost) on this machine will be used.
# If no non-local IP is found, localhost will be used.
jppf.management.host = localhost

# JMX management port, used by the remote JMX connector
jppf.management.port = 11198

# Internal RMI port used by JMX management
jppf.management.rmi.port = 12198
```

Let's see in more details the usage of each of these properties:

- **jppf.management.host**: defines the host name or IP address for the remote management and monitoring of the

servers and nodes. It represents the host where an RMI registry is running. When this property is not defined explicitely, JPPF will automatically fetch the first non-local IP address (meaning not the loopback address) it can find on the current host. If none is found, "localhost" will be used. This provides a way to use an identical configuration for all the servers on a network.
• **jppf.management.port**: defines the port number for connecting to the Mbean server. The default value for this property is "'11198'". If 2 nodes, 2 drivers or a driver and a node run on the same host, they must have a different value for this property.
• **jppf.management.rmi.port**: this is the port number used by the embedded RMI registry. It is only used internally by the server, however its value must be distinct from that of any other server or node running on the same machine. The default value for this property is "'12198'".

Note: *if a management port is already in use by another JPPF component or application, JPPF will automatically increment it, until it finds an available port number. This means that you can in fact leave the port numbers to their default values (or not specify them at all), as JPPF will automatically ensure that valid unique port numbers are used.*

5.3.6 Load-balancing

The distribution of the tasks to the nodes is performed by the JPPF driver. This work is actually the main factor of the observed performance of the framework. It consists essentially in determining how many tasks will go to each node for execution, out of a set of tasks sent by the client application. Each set of tasks sent to a node is called a "bundle", and the role of the load balancing (or task scheduling) algorithm is to optimize the performance by adjusting the number of task sent to each node.

The algorithm to use is configured with the following property:

```
jppf.load.balancing.algorithm = <algorithm_name>
```

The algorithm name can be one of those prefefined in JPPF, or a user-defined one. We will see how to define a custom alogrithm in ***Chapter 6 Extending JPPF*** of this manual. JPPF now has 4 predefined load-balancing algorithms to compute the distribution of tasks to the nodes, each with its own configuration parameters. These algorithms are:

- “manual” : each bundle has a fixed number of tasks, meaning that each will receive at most this number of tasks
- “autotuned” : adaptive heuristic algorithm based on the Monte Carlo algorithm
- “proportional” : an adaptive deterministic algorithm based on the contribution of each node to the overall mean task execution time
- “rl” : adaptive algorithm based on an artificial intelligence technique called “reinforcement learning”

The predefined possible values for the property `jppf.load.balancing.algorithm` are thus: `manual`, `autotuned`, `proportional` and `rl`. The default is `manual`, for example:

```
jppf.load.balancing.algorithm = proportional
```

Each algorithm uses its own set of parameters, which define together a *strategy* for the algorithm, It is also called a performance profile or simply profile, and we will use these terms interchangeably. A strategy has a name that serves to identify a group of parameters and their values, using the following pattern:

```
jppf.load.balancing.strategy = <profile_name>

strategy.<profile_name>.<parameter_1> = <value_1>
...
strategy.<profile_name>.<parameter_n> = <value_n>
```

Using this, you can define multiple profiles and easily switch from one to the other, by simple changing the value of `jppf.load.balancing.strategy`. It is also possible to mix, in a single profile, the parameters for multiple algorithms, however it is not recommended, as there may be name collisions.

To illustrate this, we will give a sample profile for each of the predefined algorithms:

“manual” algorithm

```
# algorithm name
jppf.load.balancing.algorithm = manual

# name of the set of parameter values or profile for the algorithm
jppf.load.balancing.strategy = manual_profile

# "manual" profile
strategy.manual_profile.size = 1
```

“autotuned” algorithm

```
# algorithm name
jppf.load.balancing.algorithm = autotuned

# name of the set of parameter values or profile for the algorithm
jppf.load.balancing.strategy = autotuned_profile

# "autotuned" profile
strategy.autotuned_profile.size = 5
strategy.autotuned_profile.minSamplesToAnalyse = 100
strategy.autotuned_profile.minSamplesToCheckConvergence = 50
strategy.autotuned_profile.maxDeviation = 0.2
strategy.autotuned_profile.maxGuessToStable = 50
strategy.autotuned_profile.sizeRatioDeviation = 1.5
strategy.autotuned_profile.decreaseRatio = 0.2
```

“proportional” algorithm

```
# algorithm name
jppf.load.balancing.algorithm = proportional

# name of the set of parameter values or profile for the algorithm
jppf.load.balancing.strategy = proportional_profile

# "proportional" profile
strategy.proportional_profile.size = 5
strategy.proportional_profile.performanceCacheSize = 3000
strategy.proportional_profile.proportionalityFactor = 2
```

“rl” algorithm

```
# algorithm name
jppf.load.balancing.algorithm = rl

# name of the set of parameter values or profile for the algorithm
jppf.load.balancing.strategy = rl_profile

# "rl" profile
strategy.rl_profile.performanceCacheSize = 3000
strategy.rl_profile.performanceVariationThreshold = 0.001
```

5.3.7 Server process configuration

A JPPF server is in fact made of two processes: a “controller” process and a “server” process. The controller launches the server as a separate process and watches its exit code. If the exit code has a pre-defined value of 2, then it will restart the server process, otherwise it will simply terminate. This mechanism allows the remote (eventually delayed) restart of a server using the management APIs or the management console. It is also made such that, if any of the two processes dies unexpectedly, then the other process will die as well, leaving no lingering Java process in the OS.

The server process inherits the following parameters from the controller process:

- location of jppf configuration file (`-Djppf.config`)
- current directory
- environment variables
- Java class path

It is possible to specify additional JVM parameters for the server process, using the configuration property jppf.jvm.options, as in this example:

```
jppf.jvm.options = -Xms64m -Xmx512m
```

Here is another example with remote debugging options:

```
jppf.jvm.options = -Xmx512m -server \
  -Xrunjdwp:transport=dt_socket,address=localhost:8000,server=y,suspend=n
```

It is possible to specify additional class path elements through this property, by adding one or more “-cp” or “-classpath” options (unlike the Java command which only accepts one). For example:

```
jppf.jvm.options = -cp lib/myJar1.jar:lib/myJar1.jar -Xmx512m \
  -classpath lib/external/externalJar.jar
```

5.3.8 Recovery from hardware failures of remote nodes

Network disconnections due to hardware failures are notoriously difficult to detect, let alone recover from. JPPF implements a configurable mechanism that enables detecting such failures, and recover from them, in a reasonable time frame. This mechanism works as follows:

- the node establishes a specific connection to the server, dedicated to failure detection
- at connection time, a handshake protocol takes place, where the node communicates a unique id (UUID) to the server, that can be correlated to other connections for this node (i.e. job server and distributed class loader)
- at regular intervals (heartbeats), the server will send a very short message to the node, which it expects the node to acknowledge by sending a short response of its own
- if the node's response is not received in a specified time frame, and this, a specified number of times in a row, the server will consider the connection to the node broken, will close it cleanly, close the associated connections, and handle the recovery, such as requeing tasks that were being executed by the node for execution on another node

In practice, the polling of the nodes is performed by a “reaper” object that will handle the querying of the nodes, using a pool of dedicated threads rather than one thread per node. This enables a higher scalability with a large number of nodes.

The ability to specify multiple attempts at getting a response from the node is useful to handle situations where the network is slow, or when the node or server is busy with a high CPU utilization level.On the server side, the parameters of this mechanism are configurable via the following properties:

```
# Enable recovery from hardware failures on the nodes.
# Default value is false (disabled).
jppf.recovery.enabled = false
# Maximum number of attempts to get a response form the node before the
# connection is considered broken. Default value is 3.
jppf.recovery.max.retries = 3
# Maximum time in milliseconds allowed for each attempt to get a response
# from the node. Default value is 6000 (6 seconds).
jppf.recovery.read.timeout = 3000
# Dedicated port number for the detection of node failure.
# Default value is 22222.
jppf.recovery.server.port = 22222
# Interval in milliseconds between two runs of the connection reaper.
# Default value is 60000 (1 minute).
jppf.recovery.reaper.run.interval = 60000
# Number of threads allocated to the reaper.
# Default value is the number of available CPUs.
jppf.recovery.reaper.pool.size = 8
```

Note: *if server discovery is active for a node, then the port number specified for the driver will override the one specified in the node's configuration*.

5.3.9 Parallel I/O

The JPPF driver uses 2 pools of threads to perform network I/O with the nodes in parallel. One pool is dedicated to sending and receiving job data, the other is dedicated to the distributed class loader. There is a single configuration property that specifies the size of each of these pools:

```
transition.thread.pool.size = <number_of_io_threads>
```

When left unspecified, this property will take a default value equal to the number of processors available to the JVM (equivalent to `Runtime.getRuntime().availableProcessors()`).

5.3.10 Full server configuration file (default values)

To link all the parts together, here is a sample server configuration file that can be reused as is:

```
# port number for the class server that performs remote class loading
class.server.port = 11111
# port number the clients / applications connect to
app.server.port = 11112
# port number the nodes connect to
node.server.port = 11113

# Enabling JMX features
jppf.management.enabled = true
# JMX management host IP address
#jppf.management.host = localhost
# JMX management port
jppf.management.port = 11198
# Internal RMI port used by JMX management
jppf.management.rmi.port = 12198

# Enable/Disable automatic discovery of JPPF drivers
jppf.discovery.enabled = true
# UDP multicast group to which drivers broadcast their connection parameters
jppf.discovery.group = 230.0.0.1
# UDP multicast port to which drivers broadcast their connection parameters
jppf.discovery.port = 11111
# How long a driver should wait between 2 broadcasts, in milliseconds
jppf.discovery.broadcast.interval = 1000
# Enable/disable auto-discovery for peer-to-peer communication between drivers
jppf.peer.discovery.enabled = false

# Load-balancing algorithm
jppf.load.balancing.algorithm = proportional
# parameters profile name
jppf.load.balancing.strategy = proportional_profile

# "manual" profile
strategy.manual_profile.size = 1

# "autotuned" profile
strategy.autotuned_profile.size = 5
strategy.autotuned_profile.minSamplesToAnalyse = 100
strategy.autotuned_profile.minSamplesToCheckConvergence = 50
strategy.autotuned_profile.maxDeviation = 0.2
strategy.autotuned_profile.maxGuessToStable = 50
strategy.autotuned_profile.sizeRatioDeviation = 1.5
strategy.autotuned_profile.decreaseRatio = 0.2

# "proportional" profile
strategy.proportional_profile.size = 5
strategy.proportional_profile.performanceCacheSize = 3000
strategy.proportional_profile.proportionalityFactor = 2

# "rl" profile
strategy.rl_profile.performanceCacheSize = 3000
strategy.rl_profile.performanceVariationThreshold = 0.001

# Other JVM options added to the java command line when the server is started
# as a subprocess. Multiple options are separated by spaces
jppf.jvm.options = -server -Xmx256m
```

5.4 Node configuration

5.4.1 Server discovery

By default, JPPF nodes are configured to automatically discover active servers on the network. As we have seen in 5.3.2 , this is possible thanks to the UDP broadcast mechanism of the server. On the other end, the node needs to join the same UDP group to subscribe to the bradcasts from the server, which is done by configuring the following properties:

```
# Enable or disable automatic discovery of JPPF drivers
jppf.discovery.enabled = true

# UDP multicast group to which drivers broadcast their connection parameters
jppf.discovery.group = 230.0.0.1

# UDP multicast port to which drivers broadcast their connection parameters
jppf.discovery.port = 11111

# How long in milliseconds the node will attempt to automatically discover a driver
# before falling back to the manual configuration parameters
jppf.discovery.timeout = 5000

# IPv4 address inclusion patterns
jppf.discovery.include.ipv4 =

# IPv4 address exclusion patterns
jppf.discovery.exclude.ipv4 =

# IPv6 address inclusion patterns
jppf.discovery.include.ipv6 =

# IPv6 address exclusion patterns
jppf.discovery.exclude.ipv6 =
```

For the node to actually find a server on the network, the values for the group and port must be the same for a node and at least one server. If multiple servers are found on the network, the node will arbitrarily pick one.

Note the property `jppf.discovery.timeout`: if defines a fall back strategy that wil cause the node to connect to the server defined in the manual configuration parameters (see 5.4.2) after the specified time.

The last four properties define inclusion and exclusion patterns for IPv4 and IPv6 addresses. Each of them defines a list of comma- or semicolumn- separated patterns. For the syntax of the IPv4 patterns, please refer to the Javadoc for the class IPv4AddressPattern, and to IPv6AddressPattern for IPv6 patterns syntax.
This enables filtering out unwanted IP addresses: the discovery mechanism will only allow addresses that are included and not excluded.

Let's take for instance the following pattern specifications:

```
jppf.discovery.include.ipv4 = 192.168.1.
jppf.discovery.exclude.ipv4 = 192.168.1.100-
```

The inclusion pattern only allows IP addresses in the range 192.168.1.0 ... 192.168.1.255
The exclusion pattern filters out IP addresses in the range 192.168.1.100 ... 192.168.1.255
Thus, we actually defined a filter that only accepts addresses in the range 192.168.1.0 ... 192.168.1.100

Instead of these 2 patterns, we could have simply defined the following equivalent inclusion pattern:

```
jppf.discovery.include.ipv4 = 192.168.1.0-99
```

5.4.2 Manual network configuration

If server discovery is disabled, network access to a server must be configured manually. To this effect, the node requires the address of host on which the server is running, and 2 TCP ports, as shown in this example:

```
# IP address or host name of the server
jppf.server.host = my_host
# class loader port
class.server.port = 11111
```

```
# communication between node and server
node.server.port = 11113
```

To not define these properties is equivalent to assigning them their default value (i.e. "localhost" for the host address).

5.4.3 Socket connections idle timeout

In some environments, a firewall may be configured to automatically close socket connections that have been idle for more than a specified time. This may lead to a situation where a server may be unaware that a node or client was disconnected, and cause one or more jobs to never return. To remedy to that situation, it is possible to configure an idle timeout on either side of the connection, so that the connection can be closed cleanly and grid operations can continue unhindered. This is done via the following property:

```
jppf.socket.max-idle = timeout_in_seconds
```

If the timeout value is less than 10 seconds, then it is considered as no timeout. The default value is -1.

5.4.4 JMX management configuration

JPPF uses JMX to provide remote management capabilities for the nodes, and uses the default RMI connector for communication. Each server has its own embedded RMI registry.

The management features are enabled by default; this behavior can be changed by setting the following property:

```
# Enable or disable management of this node
jppf.management.enabled = true
```

When management is enabled, the following properties must be defined:

```
# JMX management host IP address. If not specified (recommended), the first non-local
# IP address (i.e. neither 127.0.0.1 nor localhost) on this machine will be used.
# If no non-local IP is found, localhost will be used.
jppf.management.host = localhost

# JMX management port, used by the remote JMX connector
jppf.management.port = 11198

# Internal RMI port used by JMX management
jppf.management.rmi.port = 12198
```

These properties have the same meaning and usage as for a server, as described in 5.3.5 .

5.4.5 Recovery and failover

When the connection to a server is interrupted, the node will automatically attempt, for a given length of time, and at regular intervals, to reconnect to the same server. These properties are configured as follows, with their default values:

```
# number of seconds before the first reconnection attempt
reconnect.initial.delay = 1

# time after which the system stops trying to reconnect, in seconds
# a value of zero or less means it never stops
reconnect.max.time = 60

# time between two connection attempts, in seconds
reconnect.interval = 1
```

With these values, we have configured the recovery mechanism such that it will attempt to reconnect to the server after a 1 second delay, for 60 seconds and with connection attemps at 1 second intervals.

5.4.6 Interaction of failover and server discovery

When dicovery is enabled for the node (`jppf.dicovery.enabled = true`) and the maximum reconnection time is not infinite (`reconnect.max.time = <strictly_positive_value>`), a sophisticated failover mechanism takes place, following the sequence of steps below:

- the node attempts to reconnect to the driver to which it was previously connected (or attempted to connect), during a maximum time specified by the configuration property "`reconnect.max.time`"
- during this maximum time, it will make multiple attempts to connect to the same driver. This covers the case when the

driver is restarted in the mean time.
• after this maximum time has elapsed, it will attempt to auto-discover another driver, during a maximum time, specified via the configuration property "`jppf.discovery.timeout`" (in milliseconds)
• if the node still fails to reconnect after this timeout has expired, it will fall back to the driver manually specified in the node's configuration file
• the cycle starts again

5.4.7 Recovery from hardware failures

The mechanism to recover from hardwaire failure has its counterpart on each node, which works as follows:

- the node establishes a specific connection to the server, dedicated to failure detection
- at connection time, a handshake protocol takes place, where the node communicates a unique id (UUID) to the server
- the node will then attempt to get a message from the server ("check" message).
- if the message from the server is not received in a specified time frame, and this, a specified number of times in a row, the node will consider the connection to the server broken, will close it cleanly, and let the recovery and failover mechanism take over, as described in the previous section "5.4.6"

The following configuration properties are those required by the nodes' hardware failure recovery mechanism implemented by the server:

```
# Enable recovery from hardware failures on the node. Default is false (disabled).
jppf.recovery.enabled = false

# Dedicated port number for the detection of node failure, must be the same as
# the value specified in the server configuration. Default value is 22222.
jppf.recovery.server.port = 22222

# Maximum number of attempts to get a message from the server before the
# connection is considered broken. Default value is 2.
jppf.recovery.max.retries = 2

# Maximum time in milliseconds allowed for each attempt to get a message
# from the server. Default value is 60000 (1 minute).
jppf.recovery.read.timeout = 60000
```

Note: *if server discovery is active for a node, then the port number specified for the driver will override the one specified in the node's configuration*.

5.4.8 Processing threads

A node can process multiple tasks concurrently, using a pool of threads. The size of this pool is configured as follows:

```
# number of threads running tasks in this node
processing.threads = 4
```

If this property is not defined, its value defaults to the number of processors or cores available to the JVM.

5.4.9 Node process configuration

In the same way as for a server (see 5.3.7 Server process configuration), the node is made of 2 processes. In addition to the properties and environment inherited from the controller process, it is possible to specify other JVM options via the following configuration property:

```
jppf.jvm.options = -Xms64m -Xmx512m
```

As for the server, it is possible to specify additional class path elements through this property, by adding one or more "-cp" or "-classpath" options (unlike the Java command which only accepts one). For example:

```
jppf.jvm.options = -cp lib/myJar1.jar:lib/myJar1.jar -Xmx512m
```

5.4.10 Class loader cache

Each node creates a specific class loader for each new client whose tasks are executed in that node. The cache itself is managed as a bounded queue, and the oldest class loader will be evicted from the cache whenever the maximum size is reached. The evicted class loader then becomes unreachable and can be garbage collected. In most modern JDKs, this also results in the classes being unloaded.

If the class loader cache size is too large, this can lead to an out of memory condition in the node, especially in these 2 scenarios:

- if too many classes are loaded, the space reserved to the class definitions (permanent generation in Oracle JDK) will fill up and cause an “OutOfMemoryError: PermGen space”
- if the classes hold a large amount of static data (via static fields and static initializers), an “OutOfMemoryError: Heap Space” will be thrown

To mitigate this, the size of the class loader cache can be configured in the node as follows:

```
jppf.classloader.cache.size = 50
```

The default value for this property is 50, and the value must be at least equal to 1.

5.4.11 Security policy

To limit what the nodes can do on the machine that hosts them, It is possible to specify what permissions are granted to them on their host. These permissions are based on the Java security policy model.

To implement security, nodes require a security policy file. The syntax of this file is similar to that of Java security policy files, except that it only accepts permission entries (no grant or security context entries).

Some examples of permission entries:

```
// permission to read, write, delete node log file in current directory
permission java.io.FilePermission "${user.dir}/jppf-node.log", "read,write,delete";
// permission to read all log4j system properties
permission java.util.PropertyPermission "log4j.*", "read";
// permission to connect to a MySQL database on the default port on localhost
permission java.net.SocketPermission "localhost:3306", "connect,listen";
```

To enable the security policy, the node configuration file must contain the following property definition:

```
# Path to the security file, relative to the current directory or classpath
jppf.policy.file = jppf.policy
```

When this property is not defined, or the policy file cannot be found, security is disabled.

The policy file does not have to be local to the node. If it is not present locally, the node will download it from the server. In this case it has to be locally accessible by the server, and the path to the policy file will be interpreted as path on the server's file system. This feature, combined with the ablity to remotely restart the nodes, allows to easily update and propagate changes to the security policy for all the nodes.

5.4.12 Full node configuration file (default values)

```
# Host name, or ip address, of the host the JPPF driver is running on
jppf.server.host = localhost

# port number for the class server that performs remote class loading
class.server.port = 11111

# port number the nodes connect to
node.server.port = 11113

# Enabling JMX features
jppf.management.enabled = true

# JMX management host IP address
#jppf.management.host = localhost

# JMX management port
jppf.management.port = 12001

# Internal RMI port used by JMX management
jppf.management.rmi.port = 13001

# path to the JPPF security policy file
#jppf.policy.file = config/jppf.policy

# Enable/Disable automatic discovery of JPPF drivers
jppf.discovery.enabled = true

# UDP multicast group to which drivers broadcast their connection parameters
jppf.discovery.group = 230.0.0.1
# UDP multicast port to which drivers broadcast their connection parameters
jppf.discovery.port = 11111
# How long the  node will attempt to automatically discover a driver before
# falling back to the parameters specified in this configuration file
jppf.discovery.timeout = 5000

# Automatic recovery: number of seconds before the first reconnection attempt
reconnect.initial.delay = 1

# Time after which the system stops trying to reconnect, in seconds
reconnect.max.time = 60

# Automatic recovery: time between two connection attempts, in seconds
reconnect.interval = 1

# Processing Threads: number of threads running tasks in this node
#processing.threads = 1

# Other JVM options added to the java command line when the node is started as
# a subprocess. Multiple options are separated by spaces
jppf.jvm.options = -server -Xmx256m
```

5.5 Client and administration console configuration

5.5.1 Server discovery

By default, JPPF clients are configured to automatically discover active servers on the network. This mechanism works in the same way as for the nodes, and uses the same configuration properties, except for the discovery timeout:

```
# Enable or disable automatic discovery of JPPF drivers
jppf.discovery.enabled = true

# UDP multicast group to which drivers broadcast their connection parameters
jppf.discovery.group = 230.0.0.1

# UDP multicast port to which drivers broadcast their connection parameters
jppf.discovery.port = 11111

# IPv4 address inclusion patterns
jppf.discovery.include.ipv4 =

# IPv4 address exclusion patterns
jppf.discovery.exclude.ipv4 =

# IPv6 address inclusion patterns
jppf.discovery.include.ipv6 =

# IPv6 address exclusion patterns
jppf.discovery.exclude.ipv6 =
```

A major difference is that, when discovery is enabled, the client does not stop attempting to find one or more servers. A client can also connect to multiple servers, and will effectively connect to every server it discovers on the network.

A client can also create mutliple connections to each discovered server, effectively creating a connection pool that can be used for concurrent job submissions. The size of the connection pools is configured with the following property:

```
# connection pool size for each discovered server; defaults to 1 (single connection)
jppf.pool.size = 5
```

Each server connection has an assigned name, following the pattern: “driver-<n>[-<p>]”, where n is a driver number, in order of discovery, and the optional p is the connection number, if the defined connection pool size is greater than 1.

For instance:

- if we defined `jppf.pool.size = 1`, the first discovered driver will have 1 connection named “driver-1”
- if we defined `jppf.pool.size = 2`, the first discovered driver will have 2 connections named “driver-1-1” and “driver-1-2”

The inclusion and exclusion pattern definitions work exactly in the same way as for the node configuration. Please refer to section 5.4.1 for more details.

5.5.2 Manual network configuration

As we have seen, a JPPF client can connect to multiple drivers. The first step will this be to name these drivers:

```
# space-separated list of drivers this client may connect to
# defaults to “default-driver”
jppf.drivers = driver-1 driver-2
```

Then for each driver, we will define the connection and behavior attributes, including:

Connection to the JPPF server

```
# host name, or ip address, of the host the JPPF driver is running on
driver-1.jppf.server.host = localhost
# port number for the class server that performs remote class loading
driver-1.class.server.port = 11111
# port number the clients / applications connect to
```

```
driver-1.app.server.port = 11112
```

Here, `driver-1.class.server.port` and `driver-1.app.server.port` must have the same value as the corresponding properties `class.server.port` and `app.server.port` defined in the server configuration.

Connection pool size

```
# size of the pool of connections to this driver
driver-1.jppf.pool.size = 5
```

This allows the creation of a connection pool with a specific size for each server we connect to, whereas all pools would have the same size when server discovery is enabled.

Priority

```
# assigned driver priority
driver-1.priority = 10
```

The priority assigned to a server connection enables the defintion of a fallback strategy for the client. In effect, the client will always use connections that have the highest priority. If the connection with the server is interrupted, then the client we use connections with the next highest priority in the remaining accessible server connection pools.

Connection to the management server

```
# management host for this driver
driver-1.jppf.management.host = localhost
# management port for this driver
driver-1.jppf.management.port = 11198
```

This will allow direct access to the driver's JMX server using the client APIs, unless the client configuration property `jppf.management.enabled` is set to `false`.

5.5.3 Recovery and failover

As for the nodes, when the connection to a server is interrupted, the client will automatically attempt to reconnect to the same server. This is configured as follows, with the default values:

```
# number of seconds before the first reconnection attempt
reconnect.initial.delay = 1

# time after which the system stops trying to reconnect, in seconds
# a value of zero or less means it never stops
reconnect.max.time = 60

# time between two connection attempts, in seconds
reconnect.interval = 1
```

With these values, we have configured the recovery mechanism such that it will attempt to reconnect to the server after a 1 second delay, for 60 seconds and with connection attemps at 1 second intervals.

5.5.4 Socket connections idle timeout

In some environments, a firewall may be configured to automatically close socket connections that have been idle for more than a specified time. This may lead to a situation where a server may be unaware that a node or client was disconnected, and cause one or more jobs to never return. To remedy to that situation, it is possible to configure an idle timeout on either side of the connection, so that the connection can be closed cleanly and grid operations can continue unhindered. This is done via the following property:

```
jppf.socket.max-idle = timeout_in_seconds
```

If the timeout value is less than 10 seconds, then it is considered as no timeout. The default value is -1.

5.5.5 Local execution

It is possible for a client to execute jobs locally (i.e. in the client JVM) rather than by submitting them to a server. This feature allows taking advantage of muliple CPUs or cores on the client machine, while using the exact same APIs as for a distributed remote execution. I can also be used for local testing and debugging before performing the “real-life” execution of a job.

Local execution is disabled by default. To enable it, set the following configuration property:

```
# enable local job execution; defaults to false
jppf.local.execution.enabled = true
```

Local execution uses a pool of threads, whose size is configured as follows:

```
# number of threads to use for local execution
# the default value is the number of CPUs or cores available to the JVM
jppf.local.execution.threads = 4
```

It is also possible to mix local and remote execution. This will happen whenever the client is connected to a server and has local execution enabled. In this case, the JPPF client uses an adaptive load-balancing alogrithm to balance the workload between local execution and node-side execution.

5.5.6 Full client configuration file (default values)

```
# list of drivers this client may connect to
jppf.drivers = driver-1 driver-2

# host name, or ip address, of the host the JPPF driver is running on
driver-1.jppf.server.host = localhost
# port number for the class server that performs remote class loading
driver-1.class.server.port = 11111
# port number the clients / applications connect to
driver-1.app.server.port = 11112

# priority given to the driver connection
driver-1.priority = 10

# host name for the management server
driver-1.jppf.management.host = localhost
# port number for the management server
driver-1.jppf.management.port = 11198

# configuration for connection driver-2
driver-2.jppf.server.host = my.host.com
driver-2.class.server.port = 11121
driver-2.app.server.port = 11122
driver-2.jppf.management.port = 12003
driver-2.priority = 10

# enable/disable automatic discovery of JPPF drivers
jppf.discovery.enabled = true
# UDP multicast group to which drivers broadcast their connection parameters
jppf.discovery.group = 230.0.0.1
# UDP multicast port to which drivers broadcast their connection parameters
jppf.discovery.port = 11111

# automatic recovery: number of seconds before the first reconnection attempt
reconnect.initial.delay = 1
# time after which the system stops trying to reconnect, in seconds
reconnect.max.time = 60
# automatic recovery: time between two connection attempts, in seconds
reconnect.interval = 1
```

6 Management and monitoring

Management and monitoring are important parts of a grid platform. With these features it is possible to observe the health and status of the grid components, and directly or remotely transform their behavior.

JPPF provides a comprehensive set of monitoring and management functionalities, based on the Java Management Extensions (JMX) standard. In addition to this, a set of APIs enables a simplified access to the management functions, whether locally or remotely.

Management and monitoring functions are available for JPPF servers and nodes and provided as MBeans. We will see these MBeans in detail and then look at the APIs to access them.

All JPPF MBeans are standard MBeans registered with the platform MBean server. This means, among other things, that the can be accessed through external JMX-based applications or APIs, such as VisualVM.

6.1 Node management

Out of the box in JPPF 2.0, each node provides 2 MBeans that can be accessed remotely using an RMI remote connector with the JMX URL "**`service:jmx:rmi:///jndi/rmi://`*`host`*`:`*`port`*`/jppf/node`**", where *host* is the host name or IP address of the machine where the node is running (value of "`jppf.management.host`" in the node configuration file), and *port* is the value of the property "`jppf.management.port`" specified in the node's configuration file.

6.1.1 Node-level management and monitoring MBean

MBean name: "**org.jppf:name=admin,type=node**"
This is also the value of the constant `JPPFAdminMBean.NODE_ADMIN_NAME`.

This MBean's role is to perform management and monitoring at the node level, however we will see that it also has (for historical reasons) some task-level management functions. It exposes the `JPPFNodeAdminMBean` interface, which provides the functionalities described hereafter.

6.1.1.1 Getting a snapshot of the node's state

This is done by invoking the following method on the MBean:

```
/**
 * Get the latest state information from the node.
 * @return a <code>JPPFNodeState</code> object.
 * @throws Exception if any error occurs.
 */
public JPPFNodeState state() throws Exception;
```

This method returns a `JPPFNodeState` object, which provides the following information on the node:

```
// the status of the connection with the server
public String getConnectionStatus()

// the current task execution status
public String getExecutionStatus()

// the cpu time consumed by the node's execution threads
// this includes the tasks cpu time and some JPPF processing overhead
public long getCpuTime()

// the total number of tasks executed
public int getNbTasksExecuted()

// the current size of the pool of threads used for tasks execution
public int getThreadPoolSize()

// the current priority assigned to the execution threads
public int getThreadPriority()

// the set of ids for all currently executing tasks that have an id
public Set<String> getAllTaskIds()
```

6.1.1.2 Updating the execution thread pool properties

```
/**
 * Set the size of the node's execution thread pool.
 * @param size the new size of the thread pool; if 0 or less, this method does nothing.
 * @throws Exception if any error occurs.
 */
public void updateThreadPoolSize(Integer size) throws Exception;

/**
 * Update the priority of all execution threads.
 * @param newPriority the new priority to set.
 * @throws Exception if any error occurs.
 */
public void updateThreadsPriority(Integer newPriority) throws Exception;
```

6.1.1.3 Shutting down and restarting the node

```
/**
 * Restart the node.
 * @throws Exception if any error occurs.
 */
public void restart() throws Exception;

/**
 * Shutdown the node.
 * @throws Exception if any error occurs.
 */
public void shutdown() throws Exception;
```

These two methods should be used with precautions. Please note that, once `shutdown()` has been invoked, it is not possible anymore to restart the node remotely.

When any of these methods is invoked, the tasks that were being executed, if any, are automatically resubmitted to the server queue.

6.1.1.4 Updating the executed tasks counter

```
/**
 * Reset the node's executed tasks counter to zero.
 * @throws Exception if any error occurs.
 */
public void resetTaskCounter() throws Exception;

/**
 * Reset the node's executed tasks counter to the specified value.
 * @param n the new value of the task counter.
 * @throws Exception if any error occurs.
 */
public void setTaskCounter(Integer n) throws Exception;
```

Please note that `resetTaskCounter()` is equivalent to `setTaskCounter(0)`.

6.1.1.5 Getting information about the node's host

```
/**
 * Get detailed information about the node's JVM properties, environment variables
 * and runtime information such as memory usage, available processors and
 * information about available storage space.
 * @return a <code>JPPFSystemInformation</code> instance.
 * @throws Exception if any error occurs.
 */
JPPFSystemInformation systemInformation() throws Exception;
```

This method returns an object of type `JPPFSystemInformation`, which is a snapshot of the environment of the JPPF node, the JVM and the host they run on. The properties defined in this object are also those used by execution policies, as we have seen in section 3.4.1 of this manual.

`JPPFSystemInformation` provides information about 6 different aspects of the environment:

```
// get the system properties
public TypedProperties getSystem()
// get runtime information about JVM memory and available processors
public TypedProperties getRuntime()
// get the host environment variables
public TypedProperties getEnv()
// get IPV4 and IPV6 addresses assigned to the host
public TypedProperties getNetwork()
// get the JPPF configuration properties
public TypedProperties getJppf()
// get information on available disk storage
public TypedProperties getStorage()
```

We encourage the reader to follow the links to the above methods' Javadoc, to obtain details on each set of information, and how the information is formatted and named.

Each of the methods in `JPPFSystemInformation` returns a `TypedProperties` object. `TypedProperties` is a subclass of the standard java.util.Properties that provides convenience methods to read property values as primitive types other than String.

6.1.1.6 Canceling a job

```
/**
 * Cancel the job with the specified id.
 * @param jobId the id of the job to cancel.
 * @param requeue true if the job should be requeued on the server side,
 * false otherwise.
 * @throws Exception if any error occurs.
 */
public void cancelJob(String jobId, Boolean requeue) throws Exception;
```

This MBean method is used to cancel a job currently running in the node. The job is identified by its jobId. The requeue parameter is used to notify the server that the canceled job should be requeued on the server and executed again, possibly on an other node. If requeue is false, the job is simply terminated and any remaining task will not be executed.

This method should normally only be used by the JPPF server, in the case where a user requested that the server terminates a job. In effect, a job can contain several tasks, with each task potentially executed concurrently on a separate node. When the server receives a job termination request, it will handle the termination of “sub-jobs” (i.e. subsets of the tasks in the job) by notifying each corresponding node.

6.1.1.7 Canceling and restarting individual tasks

```
/**
 * Cancel the execution of the tasks with the specified id.
 * @param id the id of the tasks to cancel.
 * @throws Exception if any error occurs.
 */
public void cancelTask(String id) throws Exception;
/**
 * Restart the execution of the tasks with the specified id.
 * The task(s) will be restarted even if their execution has already completed.
 * @param id the id of the task or tasks to restart.
 * @throws Exception if any error occurs.
 */
public void restartTask(String id) throws Exception;
```

Each of these methods applies to currently executing tasks only. They also use a task Id to identifiy the affected task. If multiple tasks have the same Id and are currently executing, all of them will be canceled or restarted. Upon invocation of these methods, the corresponding `onCancel()` and `onRestart()` methods of the tasks will be called.

6.1.1.8 Updating the node's configuration properties

```
/**
 * Update the configuration properties of the node.
 * @param config the set of properties to update.
 * @param reconnect - specifies whether the node should disconnect from,
 * then reconnect to the driver after updating the properties.
 * @throws Exception if any error occurs.
 */
void updateConfiguration(Map<String, String> config, Boolean reconnect)
     throws Exception;
```

This method sends a set of configuration properties to the node, that will override those defined in the node's configuration file. The reconnect parameter will allow the node to take the changes into account, especially in the case where the server connection or discovery properties have been changed, for instance to force the node to connect to another server without having to stop it.

6.1.2 Task-level monitoring

MBean name : "**org.jppf:name=task.monitor,type=node**".
This is also the value of the constant `JPPFNodeTaskMonitorMBean.TASK_MONITOR_MBEAN_NAME`

This MBean monitors the task activity within a node. It exposes the interface `JPPFNodeTaskMonitorMBean` and also emits JMX notifications of type `TaskExecutionNotification`.

6.1.2.1 Snapshot of the tasks activity

The interface `JPPFNodeTaskMonitorMBean` provides access to aggregated statistics on the tasks executed within a node:

```
// The total number of tasks executed by the node
Integer getTotalTasksExecuted();

// The total number of tasks that ended in error
Integer getTotalTasksInError();

// The total number of tasks that executed sucessfully
Integer getTotalTasksSucessfull();

// The total cpu time used by the tasks in milliseconds
Long getTotalTaskCpuTime();

// The total elapsed time used by the tasks in milliseconds
Long getTotalTaskElapsedTime();
```

6.1.2.2 Notification of tasks execution

Each time a task completes its execution in a node, the task monitor MBean will emit a JMX notification of type `TaskExecutionNotification` defined as follows:

```
public class TaskExecutionNotification extends Notification
{
  // Get the object encapsulating information about the task
  public TaskInformation getTaskInformation();
}
```

This notification essentially encapsulates an object of type [TaskInformation](), which provides the following information about each executed task:

```
public class TaskInformation implements Serializable
{
  // Get the task id
  public String getId()

  // Get the id of the job this task belongs to
  public String getJobId()

  // Get the cpu time used by the task
  public long getCpuTime()

  // Get the wall clock time used by the task
  public long getElapsedTime()

  // Determines whether the task had an exception
  public boolean hasError()

  // Get the timestamp for the task completion
  // Caution: this value is related to the node's system time,
  // not to the time of the notification receiver
  public long getTimestamp()
}
```

6.1.3 Accessing and using the node MBeans

JPPF provides an API that simplifies access to the JMX-based management features of a node, by abstracting most of the complexities of JMX programming. This API is represented by the class `JMXNodeConnectionWrapper`, which provides a simplified way of connecting to the node's MBean server, along with a set of convenience methods to easily access the MBeans' exposed methods and attributes.

6.1.3.1 Connecting to an MBean server

Connection to to a node MBean server is done in two steps:

a. Create an instance of JMXNodeConnectionWrapper

To connect to a **local** (same JVM) MBean server, use the no-arg constructor:

```
JMXNodeConnectionWrapper wrapper = new JMXNodeConnectionWrapper();
```

To connect to a **remote** MBean server, use the constructor specifiying the management host and port:

```
JMXNodeConnectionWrapper wrapper = new JMXNodeConnectionWrapper(host, port);
```

Here `host` and `port` represent the node's configuration properties “jppf.management.host” and “jppf.management.port”

b. Initiate the connection to the MBean server and wait until it is established

There are two ways to do this:

Synchronously:

```
// connect and wait for the connection to be established
// choose a reasonable value for the timeout, or 0 for no timeout
wrapper.connectAndWait(timeout);
```

Asynchronously:

```
// initiate the connection; this method returns immediately
wrapper.connect()

// ... do something else ...

// check if we are connected
if (wrapper.isConnected()) ...;
else ...;
```

6.1.3.2 Direct use of the JMX wrapper

JMXNodeConnectionWrapper implements directly the interface JPPFNodeAdminMBean. This means that all the methods of this interface can be used directly from the JMX wrapper. For example:

```
JMXNodeConnectionWrapper wrapper = new JMXNodeConnectionWrapper(host, port);
wrapper.connectAndWait(timeout);

// get the number of tasks executed since the last reset
int nbTasks = wrapper.state().getNbTasksExecuted();
// stop the node
wrapper.shutdown();
```

6.1.3.3 Use of the JMX wrapper's invoke() method

JMXConnectionWrapper.invoke() is a generic method that allows invoking any exposed method of an MBean.

Here is an example:

```
JMXNodeConnectionWrapper wrapper = new JMXNodeConnectionWrapper(host, port);
wrapper.connectAndWait(timeout);

// equivalent to JPPFNodeState state = wrapper.state();
JPPFNodeState state = (JPPFNodeState) wrapper.invoke(
  JPPFAdminMBean.NODE_MBEAN_NAME, "state", (Object[]) null, (String[]) null);
int nbTasks = state.getNbTasksExecuted();
// get the total CPU time used
long cpuTime = (Long) wrapper.invoke(JPPFNodeTaskMonitorMBean.TASK_MONITOR_MBEAN_NAME,
  "getTotalTaskCpuTime", (Object[]) null, (String[]) null);
```

6.1.3.4 Use of an MBean proxy

A proxy is a dynamically created object that implements an interface specified at runtime.
The standard JMX API provides a way to create a proxy to a remote or local MBeans. This is done as follows:

```
JMXNodeConnectionWrapper wrapper = new JMXNodeConnectionWrapper(host, port);
wrapper.connectAndWait(timeout);

// create the proxy instance
JPPFNodeTaskMonitorMBean proxy =
  wrapper.getProxy(JPPFNodeTaskMonitorMBean.TASK_MONITOR_MBEAN_NAME,
                   JPPFNodeTaskMonitorMBean.class);

// get the total CPU time used
long cpuTime = proxy.getTotalTaskCpuTime();
```

6.1.3.5 Subscribing to MBean notifications

We have seen that the task monitoring MBean represented by the JPPFNodeTaskMonitorMBean interface is able to emit notifications of type TaskExecutionNotification. There are 2 ways to subscribe to these notifications:

a. Using a proxy to the MBean

```
JMXNodeConnectionWrapper wrapper = new JMXNodeConnectionWrapper(host, port);
wrapper.connectAndWait(timeout);
JPPFNodeTaskMonitorMBean proxy =
  wrapper.getProxy(JPPFNodeTaskMonitorMBean.TASK_MONITOR_MBEAN_NAME,
                   JPPFNodeTaskMonitorMBean.class);

// subscribe to all notifications from the MBean
proxy.addNotificationListener(myNotificationListener, null, null);
```

b. Using the MBeanServerConnection API

```
JMXNodeConnectionWrapper wrapper = new JMXNodeConnectionWrapper(host, port);
wrapper.connectAndWait(timeout);
MBeanServerConnection mbsc = wrapper.getMbeanConnection();
ObjectName objectName =
  new ObjectName(JPPFNodeTaskMonitorMBean.TASK_MONITOR_MBEAN_NAME);

// subscribe to all notifications from the MBean
mbsc.addNotificationListener(objectName, myNotificationListener, null, null);
```

Here is an example notification listener implementing the `NotificationListener` interface:

```
// this class counts the number of tasks executed, along with
// the total cpu time and wall clock time used by the node
public class MyNotificationListener implements NotificationListener
{
  AtomicInteger taskCount = new AtomicInteger(0);
  AtomicLong cpuTime = new AtomicLong(0L);
  AtomicLong elapsedTime = new AtomicLong(0L);

  // Handle an MBean notification
  public void handleNotification(Notification notification, Object handback)
  {
    TaskExecutionNotification jppfNotif = (TaskExecutionNotification) notification;
    TaskInformation info = jppfNotif.getTaskInformation();
    int n = taskCount.incrementAndGet();
    long cpu = cpuTime.addAndGet(info.getCpuTime());
    long elapsed = elapsedTime.addAndGet(info.getElapsedTime());
    // display the statistics for every 50 tasks executed
    if (n % 50 == 0)
    {
      System.out.println("nb tasks = " + n + ", cpu time = " + cpu
        + " ms, elapsed time = " + elapsed +" ms");
    }
  }
};

NotificationListener myNotificationListener = new MyNotificationListener();
```

6.1.4 Remote logging

It is possible to receive logging messages from a node as JMX notifications. Specific implementation are available for Log4j and JDK logging.

To configure Log4j for emitting JMX notifications, edit the log4j configuration files of the node and add the followoing:

```
### direct messages to the JMX Logger ###
log4j.appender.JMX=org.jppf.logging.log4j.JmxAppender
log4j.appender.JMX.layout=org.apache.log4j.PatternLayout
log4j.appender.JMX.layout.ConversionPattern=%d [%-5p][%c.%M(%L)]: %m\n

### set log levels - for more verbose logging change 'info' to 'debug' ###
log4j.rootLogger=INFO, JPPF, JMX
```

To configure the JDK logging to send JMX notifications, edit the JDK logging configuration file of the node and add the following:

```
# list of handlers
handlers= java.util.logging.FileHandler, org.jppf.logging.jdk.JmxHandler

# Write log messages as JMX notifications.
org.jppf.logging.jdk.JmxHandler.level = FINEST
org.jppf.logging.jdk.JmxHandler.formatter = org.jppf.logging.jdk.JPPFLogFormatter
```

To receive the logging notifications from a remote application, you can use the following code:

```
// get a JMX connection to the node MBean server
JMXNodeConnectionWrapper jmxNode = new JMXNodeConnectionWrapper(host, port);
jmxNode.connectAndWait(5000L);
// get a proxy to the MBean
JmxLogger nodeProxy = jmxNode.getProxy(JmxLogger.DEFAULT_MBEAN_NAME, JmxLogger.class);

// use a handback object so we know where the log messages come from
String source = "node   " + jmxNode.getHost() + ":" + jmxNode.getPort();
// subbscribe to all notifications from the MBean
NotificationListener listener = new MyLoggingHandler();
nodeProxy.addNotificationListener(listener, null, source);

// Logging notification listener that prints remote log messages
// to the console
public class MyLoggingHandler implements NotificationListener
{
  // handle the logging notifications
  public void handleNotification(Notification notification, final Object handback)
  {
    String message = notification.getMessage();
    String toDisplay = handback.toString() + ": " + message;
    System.out.println(toDisplay);
  }
}
```

6.2 Server management

Out of the box in JPPF 2.0, each server provides 2 MBeans that can be accessed remotely using an RMI remote connector with the JMX URL "**service:jmx:rmi:///jndi/rmi://*host*:*port*/jppf/driver**", where *host* is the host name or IP address of the machine where the server is running (value of "jppf.management.host" in the server configuration file), and *port* is the value of the property "jppf.management.port" specified in the server's configuration file.

6.2.1 Server-level management and monitoring

MBean name: "**org.jppf:name=admin,type=driver**"
This is also the value of the constant JPPFAdminMBean.DRIVER_MBEAN_NAME.

This MBean's role is to perform management and monitoring at the server level. It exposes the JPPFAdminMBean interface, which provides the functionalities described hereafter.

6.2.1.1 Server statistics

You can get a snapshot of the server's state by invoking the following method, which porvides statistics on execution performance, network overhead, server queue behavior, number of connected nodes and clients:

```
/**
 * Get the latest statistics snapshot from the JPPF driver.
 * @return a <code>JPPFStats</code> instance.
 * @throws Exception if any error occurs.
 */
public JPPFStats statistics() throws Exception;
```

This method returns an object of type JPPFStats. which exposes the following accessors:

```
// total number of tasks executed
public int getTotalTasksExecuted()
// time statistics for the tasks execution,
// includes network transport and node execution time
public TimeSnapshot getExecution()
// time statistics for execution within the nodes
public TimeSnapshot getNodeExecution()
// time statistics for the network transport between nodes and server
public TimeSnapshot getTransport()
// time statistics for the server overhead
public TimeSnapshot getServer()
// time statistics for the queued tasks
public TimeSnapshot getQueue()
// total number of tasks that have been queued
public int getTotalQueued()
// number of tasks in the queue
public int getQueueSize()
// peak queue size
public int getMaxQueueSize()
// current number of nodes connected to the server
public int getNbNodes()
// peak number of nodes connected to the server
public int getMaxNodes()
// the current number of clients connected to the server
public int getNbClients()
// peak number of clients connected to the server
public int getMaxClients()
```

Some of these methods return an instance of the class TimeSnapshot, that encapsulates multiple aspects of time-related statistics. It exposes the following methods:

```
// total cumulated time
public long getTotalTime()
// latest observed time
public long getLatestTime()
// smallest observed time
public long getMinTime()
```

```
// peak time
public long getMaxTime()
// average time
public double getAvgTime()
```

6.2.1.2 Stopping and restarting the server

```
/**
 * Perform a shutdown or restart of the server.
 * @param shutdownDelay - the delay before shutting down the server,
 * once the command is received.
 * @param restartDelay - the delay before restarting, once the server is shutdown.
 * If the value is negative, no restart occurs, the server simply shuts down.
 * @return an acknowledgement message.
 * @throws Exception if any error occurs.
 */
public String restartShutdown(Long shutdownDelay, Long restartDelay) throws Exception;
```

This method allows you to remotely shut down the server, and eventually to restart it after a specified delay. This can be useful when an upgrade or maintenance of the server must take place within a limited time window.

6.2.1.3 Managing the nodes attached to the server

```
/**
 * Request the JMX connection information for all the nodes attached to the server.
 * @return a collection of <code>JPPFManagementInfo</code> instances.
 * @throws Exception if any error occurs.
 */
public Collection<JPPFManagementInfo> nodesInformation() throws Exception;
```

The `JPPFManagementInfo` objects returned in the resulting collection encapsulate enough information to connect to the corresponding node's MBean server:

```
public class JPPFManagementInfo
  implements Serializable, Comparable<JPPFManagementInfo>
{
  // the host on which the node is running
  public String getHost()

  // the port on which the node's JMX server is listening
  public int getPort()
}
```

For example, based on what we saw in the section about nodes management, we could write code that gathers connection information for each node attached to a server, and then performs some management request on them:

```
// Obtain connection information for all attached nodes
Collection<JPPFManagementInfo> nodesInfo = myDriverMBeanProxy.nodesInformation();
// for each node
for (JPPFManagementInfo info: nodesInfo)
{
  // create a JMX connection wrapper based on the node information
  JMXNodeConnectionWrapper wrapper =
    new JMXNodeConnectionWrapper(info.getHost(), info.getPort());
  // connect to the node's MBean server
  wrapper.connectAndWait(5000);
  // restart the node
  wrapper.restart();
}
```

6.2.1.4 Load-balancing settings

The driver management MBean provides two methods to dynamically obtain and change the server's load balancing settings:

```
/**
 * Obtain the current load-balancing settings.
 * @return an instance of <code>LoadBalancingInformation</code>.
 * @throws Exception if an error occurred while fetching the settings.
 */
public LoadBalancingInformation loadBalancerInformation() throws Exception;
```

This method returns an object of type [LoadBalancingInformation](), defined as follows:

```
public class LoadBalancingInformation implements Serializable
{
  // the name of the algorithm currently used by the server
  public String algorithm = null;
  // the algorithm's parameters
  public TypedProperties parameters = null;
  // the names of all algorithms available to the sever
  public List<String> algorithmNames = null;
}
```

Notes:

– the value of `algorithm` is included in the list of algorithm names
– `parameters` contains a mapping of the algorithm parameters names to their current value. Unlike what we have seen in the configuration guide chapter, the parameter names are expressed without suffix. This means that instead of `strategy.<profile_name>.<parameter_name>`, they will just be named as `<parameter_name>`.

It is also possible to dynamically change the load-balancing algorithm used by the server, and / or its parameters:

```
/**
 * Change the load-balancing settings.
 * @param algorithm - the name opf the load-balancing algorithm to set.
 * @param parameters - the algorithm's parameters.
 * @return an acknowledgement or error message.
 * @throws Exception if an error occurred while updating the settings.
 */
public String changeLoadBalancerSettings(String algorithm, Map parameters)
  throws Exception;
```

Where:

– `algorithm` is the name of the algorithm to use. If it is not known to the server, no change occurs.
– `parameters` is a map of algorithm parameter names to their value. Similarly to what we saw above, the parameter names must be expressed without suffix. Internally, the JPPF server will use a the profile name “jppf”.

6.2.2 Job-level management and monitoring

MBean name: "**org.jppf:name=jobManagement,type=driver**"
This is also the value of the constant `JPPFAdminMBean.DRIVER_JOB_MANAGEMENT_MBEAN_NAME`.

The role of this MBean is to control and monitor the life cycle of all jobs submitted to the server. It exposes the `DriverJobManagementMBean` interface, defined as follows:

```
public interface DriverJobManagementMBean extends NotificationEmitter
{
  // Cancel the job with the specified id
  public void cancelJob(String jobId) throws Exception;
  // Suspend the job with the specified id
  public void suspendJob(String jobId, Boolean requeue) throws Exception;
  // Resume the job with the specified id
  public void resumeJob(String jobId) throws Exception;
  // Update the maximum number of nodes a job can run on
  public void updateMaxNodes(String jobId, Integer maxNodes) throws Exception;
  // Get the set of ids for all the jobs currently queued or executing
  public String[] getAllJobIds() throws Exception;
  // Get an object describing the job with the specified id
  public JobInformation getJobInformation(String jobId) throws Exception;
  // Get a list of objects describing the nodes to which the whole
  // or part of a job was dispatched
  public NodeJobInformation[] getNodeInformation(String jobId) throws Exception;
}
```

Reminder:

A job can be made of multiple tasks. These tasks may not be all executed on the same node. Instead, the set of tasks may be split in several subsets, and these subsets can in turn be disptached to different nodes to allow their execution in parallel. In the remainder of this section we will call each subset a "sub-job", to distinguish them from actual jobs at the server level. Thus a job is associated with a server, whereas a sub-job is associated with a node.

6.2.2.1 Controlling a job's life cycle

It is possible to terminate, suspend and resume a job using the following methods:

```
/**
 * Cancel the job with the specified id.
 * @param jobId the id of the job to cancel.
 * @throws Exception if any error occurs.
 */
public void cancelJob(String jobId) throws Exception;
```

This will terminate the job with the specified `jobId`. Any sub-job running in a node will be terminated as well. If a sub-job was partially executed (i.e. at least one task execution was completed), the results are discarded. If the job was still waiting in the server queue, is simply removed from the queue, and the enclosed tasks are returned in their original state to the client.

```
/**
 * Suspend the job with the specified id.
 * @param jobId the id of the job to suspend.
 * @param requeue true if the sub-jobs running on each node should be canceled
 * and requeued, false if they should be left to execute until completion.
 * @throws Exception if any error occurs.
 */
public void suspendJob(String jobId, Boolean requeue) throws Exception;
```

This method will suspend the job with the specified `jobId`. The `requeue` parameter specifies how the currently running sub-jobs will be processed:

– if **true**, then the sub-job is canceled and inserted back into the server queue, for execution at a later time
– if **false**, JPPF will let the sub-job finish executing in the node, then suspend the rest of the job still in the server queue

If the job is already suspended, then calling this method has no effect.

```
/**
 * Resume the job with the specified id.
 * @param jobId the id of the job to resume.
 * @throws Exception if any error occurs.
 */
public void resumeJob(String jobId) throws Exception;
```

This method resumes the execution of a suspended job with the specified `jobId`. If the job was not suspended, this method has no effect.

6.2.2.2 Number of nodes assigned to a job

```
/**
 * Update the maximum number of nodes a job can run on.
 * @param jobId the id of the job to update.
 * @param maxNodes the new maximum number of nodes for the job.
 * @throws Exception if any error occurs.
 */
public void updateMaxNodes(String jobId, Integer maxNodes) throws Exception;
```

This method specifies the maximum number of nodes a job with the specified `jobId` can run on in parallel. It does not guarantee that this number of nodes will be used: the nodes may already be assigned to other jobs, or the job may not be splitted into that many sub-jobs (depending on the load-balancing algorithm). However it does guarantee that no more than `maxNodes` nodes will be used to execute the job.

6.2.2.3 Job introspection

```
/**
 * Get the set of ids for all the jobs currently queued or executing.
 * @return an array of ids as strings.
 * @throws Exception if any error occurs.
 */
public String[] getAllJobIds() throws Exception;
```

This methods returns the IDs of all the jobs currently handled by the server. These IDs can be directly used with the other methods of the job management MBean.

```
/**
 * Get an object describing the job with the specified id.
 * @param jobId the id of the job to get information about.
 * @return an instance of <code>JobInformation</code>.
 * @throws Exception if any error occurs.
 */
public JobInformation getJobInformation(String jobId) throws Exception;
```

Retrieves information about the state of a job in the server. This method returns an object of type [`JobInformation`], defined as follows:

```
public class JobInformation implements Serializable
{
  // the unique identifier for the job
  public String getJobId()
  // the current number of tasks in the job or sub-job
  public int getTaskCount()
  // the priority of this task bundle
  public int getPriority()
  // the initial task count of the job (at submission time)
  public int getInitialTaskCount()
  // determine whether the job is in suspended state
  public boolean isSuspended()
  // set the maximum number of nodes this job can run on
  public int getMaxNodes()
  // the pending state of the job
  // a job is pending if its scheduled execution date/time has not yet been reached
  public boolean isPending()
}
```

It is also possible to obtain information about all the sub-jobs of a job that are disptached to remote nodes:

```
/**
 * Get a list of objects describing the sub-jobs of a job, and the nodes to which
 * they were dispatched.
 * @param jobId the id of the job for which to find the information.
 * @return an array of <code>NodeJobInformation</code> instances.
 * @throws Exception if any error occurs.
 */
public NodeJobInformation[] getNodeInformation(String jobId) throws Exception;
```

The return value is an array of objects of type `NodeJobInformation`, defined as follows:

```
public class NodeJobInformation implements Serializable
{
  // The JMX connection information for the node
  public final JPPFManagementInfo nodeInfo;

  // The information about the sub-job
  public final JobInformation jobInfo;
}
```

This class is simply a grouping of two objects of type `JobInformation` and `JPPFManagementInfo`, which we have already seen previously. The nodeInfo attribute will allow us to connect to the corresponding node's MBean server and obtain additional job monitoring data.

6.2.2.4 Job notifications

Whenever a job-related event occurs, the job management MBean will emit a notification of type `JobNotification`, defined as follows:

```
public class JobNotification extends Notification
{
  // the information about the job or sub-job
  public JobInformation getJobInformation()

  // the information about the node (for sub-jobs only)
  // null for a job on the server side
  public JPPFManagementInfo getNodeInfo()

  // the creation timestamp for this event
  public long getTimestamp()

  // the type of this job event
  public JobEventType getEventType()
}
```

The value of the job event type (see `JobEventType` type safe enumeration) is one of the following:

- JOB_QUEUED: a new job was submitted to the JPPF driver queue
- JOB_ENDED: a job was completed and sent back to the client
- JOB_DISPATCHED: a sub-job was dispatched to a node
- JOB_RETURNED: a sub job returned from a node
- JOB_UPDATED: one of the job attributes has changed

6.2.3 Accessing and using the server MBeans

As for the nodes, JPPF provides an API that simplifies access to the JMX-based management features of a server, by abstracting most of the complexities of JMX programming. This API is represented by the class JMXDriverConnectionWrapper that provides a simplified way of connecting to the server's MBean server, along with a set of convenience methods to easily access the MBeans' exposed methods and attributes.
Please note that this class implements the `JPPFDriverAdminMBean` interface, as well as all the methods in the `DriverJobManagementMBean` interface (but without implementing the interface itself).

6.2.3.1 Connecting to an MBean server

Connection to to a server MBean server is done in two steps:

a. Create an instance of JMXDriverConnectionWrapper

To connect to a **local** (same JVM) MBean server, use the no-arg constructor:

```
JMXDriverConnectionWrapper wrapper = new JMXDriverConnectionWrapper();
```

To connect to a **remote** MBean server, use the constructor specifiying the management host and port:

```
JMXDriverConnectionWrapper wrapper = new JMXDriverConnectionWrapper(host, port);
```

Here `host` and `port` represent the server's configuration properties “jppf.management.host” and “jppf.management.port”

b. Initiate the connection to the MBean server and wait until it is established

There are two ways to do this:

Synchronously:

```
// connect and wait for the connection to be established
// choose a reasonable value for the timeout, or 0 for no timeout
wrapper.connectAndWait(timeout);
```

Asynchronously:

```
// initiate the connection; this method returns immediately
wrapper.connect()

// ... do something else ...

// check if we are connected
if (wrapper.isConnected()) ...;
else ...;
```

6.2.3.2 Direct use of the JMX wrapper

`JMXDriverConnectionWrapper` implements directly the `JPPFDriverAdminMBean` interface, as well as all the methods in the `DriverJobManagementMBean` interface (but without implementing the interface itself). This means that all the JPPF server's management and monitoring methods can be used directly from the JMX wrapper. For example:

```
JMXDriverConnectionWrapper wrapper = new JMXDriverConnectionWrapper(host, port);
wrapper.connectAndWait(timeout);

// get the ids of all jobs in the server queue
String jobIds = wrapper.getAllJobIds();
// stop the server in 2 seconds (no restart)
wrapper.restartShutdown(2000L, -1L);
```

6.2.3.3 Use of the JMX wrapper's invoke() method

JMXConnectionWrapper.invoke() is a generic method that allows invoking any exposed method of an MBean.

Here is an example:

```
JMXDriverConnectionWrapper wrapper = new JMXDriverConnectionWrapper(host, port);
wrapper.connectAndWait(timeout);

// equivalent to JPPFStats stats = wrapper.statistics();
JPPFStats stats = (JPPFStats) wrapper.invoke(
  JPPFAdminMBean.DRIVER_MBEAN_NAME, "statistics", (Object[]) null, (String[]) null);
int nbNodes = stats.getNbNodes();
// get the total CPU time used
long cpuTime = (Long) wrapper.invoke(JPPFNodeTaskMonitorMBean.TASK_MONITOR_MBEAN_NAME,
  "getTotalTaskCpuTime", (Object[]) null, (String[]) null);
```

6.2.3.4 Use of an MBean proxy

A proxy is a dynamically created object that implements an interface specified at runtime.
The standard JMX API provides a way to create a proxy to a remote or local MBean. This is done as follows:

```
JMXDriverConnectionWrapper wrapper = new JMXDriverConnectionWrapper(host, port);
wrapper.connectAndWait(timeout);

// create the proxy instance
DriverJobManagementMBean proxy =
  wrapper.getProxy(JPPFAdminMBean.DRIVER_JOB_MANAGEMENT_MBEAN_NAME,
                   DriverJobManagementMBean.class);
// get the ids of all jobs in the server queue
String jobIds = proxy.getAllJobIds();
```

6.2.3.5 Subscribing to MBean notifications

We have seen that the task monitoring MBean represented by the `JPPFNodeTaskMonitorMBean` interface is able to emit notifications of type `TaskExecutionNotification`. There are 2 ways to subscribe to these notifications:

a. Using a proxy to the MBean

```
JMXDriverConnectionWrapper wrapper = new JMXNodeConnectionWrapper(host, port);
wrapper.connectAndWait(timeout);
DriverJobManagementMBean proxy =
  wrapper.getProxy(JPPFAdminMBean.DRIVER_JOB_MANAGEMENT_MBEAN_NAME,
                   DriverJobManagementMBean.class);
// subscribe to all notifications from the MBean
proxy.addNotificationListener(myJobNotificationListener, null, null);
```

b. Using the MBeanServerConnection API

```
JMXDriverConnectionWrapper wrapper = new JMXDriverConnectionWrapper(host, port);
wrapper.connectAndWait(timeout);
MBeanServerConnection mbsc = wrapper.getMbeanConnection();
ObjectName objectName =
  new ObjectName(JPPFAdminMBean.DRIVER_JOB_MANAGEMENT_MBEAN_NAME);
// subscribe to all notifications from the MBean
mbsc.addNotificationListener(objectName, myNotificationListener, null, null);
```

Here is an example notification listener implementing the NotificationListener interface:

```
// this class prints a message each time a job is added to the server's queue
public class MyJobNotificationListener implements NotificationListener
{
  // Handle an MBean notification
  public void handleNotification(Notification notification, Object handback)
  {
    JobNotification jobNotif = (JobNotification) notification;
    JobEventType eventType = jobNotif.getEventType();
    // print a message for new jobs only
    if (eventType.equals(JobEventType.JOB_QUEUED))
    {
      String jobId = jobNotif.getJobInformation().getJobId();
      System.out.println("job " + jobId + " was queued at timestamp "
        + jobNotif.getTimestamp());
    }
  }
};
NotificationListener myJobNotificationListener = new MyJobNotificationListener();
```

6.2.4 Remote logging

It is possible to receive logging messages from a driver as JMX notifications. Specific implementations are available for Log4j and JDK logging.

To configure Log4j to send JMX notifications, edit the log4j configuration files of the node and add the following:

```
### direct messages to the JMX Logger ###
log4j.appender.JMX=org.jppf.logging.log4j.JmxAppender
log4j.appender.JMX.layout=org.apache.log4j.PatternLayout
log4j.appender.JMX.layout.ConversionPattern=%d [%-5p][%c.%M(%L)]: %m\n
### set log levels - for more verbose logging change 'info' to 'debug' ###
log4j.rootLogger=INFO, JPPF, JMX
```

To configure the JDK logging to send JMX notifications, edit the JDK logging configuration file of the driver as follows:

```
# list of handlers
handlers= java.util.logging.FileHandler, org.jppf.logging.jdk.JmxHandler
# Write log messages as JMX notifications.
org.jppf.logging.jdk.JmxHandler.level = FINEST
org.jppf.logging.jdk.JmxHandler.formatter = org.jppf.logging.jdk.JPPFLogFormatter
```

To receive the logging notifications from a remote application, you can use the following code:

```
// get a JMX connection to the node MBean server
JMXDriverConnectionWrapper jmxDriver = new JMXDriverConnectionWrapper(host, port);
jmxDriver.connectAndWait(5000L);
// get a proxy to the MBean
JmxLogger driverProxy =
  jmxDriver.getProxy(JmxLogger.DEFAULT_MBEAN_NAME, JmxLogger.class);
// use a handback object so we know where the log messages come from
String source = "driver " + jmxDriver.getHost() + ":" + jmxDriver.getPort();
// subbscribe to all notifications from the MBean
NotificationListener listener = new MyLoggingHandler();
driverProxy.addNotificationListener(listener, null, source);

// Logging notification listener that prints remote log messages to the console
public class MyLoggingHandler implements NotificationListener
{
  // handle the logging notifications
  public void handleNotification(Notification notification, final Object handback)
  {
    String message = notification.getMessage();
    String toDisplay = handback.toString() + ": " + message;
    System.out.println(toDisplay);
  }
}
```

7 Extending and Customizing JPPF

Since version 2.0, JPPF provides the ability to extend the framework without having to learn its source code nor its internal workings. This is done using two kinds of extension or customization mechanisms. One, based on the Service Provider Interface (SPI) APIs, enables the developers to simply drop a jar file in the class path of a server or node for the extension to become active. The other mechanism relies on one or more configuration properties to customize specific features in JPPF. We will detail these mechanisms, along with the areas they apply to, in the next sections.

7.1 Pluggable MBeans

Developers can write their own management beans (MBeans) and register them with the JPPF MBean server for a node or a driver. These MBeans can then be accessed, locally or remotely, as any of the built-in JPPF MBeans. Refer to the chapter on management and monitoring, for details on how to connect to an MBean server and use the registered MBeans.

Note: *all JPPF built-in MBeans are implemented via this mechanism*.

Related sample: “Custom MBeans” sample in the JPPF samples pack.

7.1.1 Elements and constraints common to node and server MBeans

The mechanism for pluggable MBeans is based on the Service Provider Interface, which is a light-weight and standard mechanism to provide extensions to Java applications.

The general workflow for adding a pluggable MBean is as follows:

step 1: implement the MBean: MBean interface + MBean implementation class
step 2: implement the MBean provider interface provided in JPPF
step 3: add or update the corresponding service definition file in the `META-INF/services` folder
step 4: create a jar file containing the above elements and deploy it in the node or server class path

The JPPF MBean handling mechanism relies on standard MBeans that *must* comply with the following constraints:

– the MBean interface name must be of the form *<MyName>*MBean and the MBean implementation class name must be of the form *<MyName>*. For instance, if we want to add a server health monitor, we would create the interface `ServerHealthMonitorMBean` and implement it in a class named `ServerHealthMonitor`.
– the MBean interface and implementation class must be defined in the same package. This is due to the constraints imposed by the JPPF distributed class loading mechanism, which allows nodes to download their custom MBeans from the server. If this constraint is not followed, the default JMX remote connector will be unable to find the MBean implementation class and it will not be possible to use the MBean. The MBean interface and implementation may, however, be in separate jar files or class folders (as long as they are in the same package).
– for custom MBeans that access other MBeans, the order in which the service definition files and their entries are read is important, since it is the order in which the MBeans are instantiated. This means that, if an MBean uses an another, the developer must ensure that the dependant MBean is created *after* the one it depends on.
– the MBean provider interface must have a public no-arg constructor

7.1.2 Writing a custom node MBean

In this section we will follow the workflow described in the previous section and create a simple custom node MBean.

Step 1: create the MBean interface and its implementation

In this example, we will create an MBean that exposes a single method to query the number of processors available to the node's JVM. First we create an interface named `AvailableProcessorsMBean`:

```
package org.jppf.example.mbean;

// Exposes one method that queries the node's JVM
// for the number of available processors
public interface AvailableProcessorsMBean
{
  // return the available processors as an integer value
  Integer queryAvailableProcessors();
}
```

Now we will create an implementation of this interface, in a class named `AvailableProcessors`, defined in the same Java package `org.jppf.example.node.mbean`:

```
package org.jppf.example.mbean;

// Implementation of the AvailableProcessorsMBean interface
public class AvailableProcessors implements AvailableProcessorsMBean
{
  // return the available processors as an integer value
  public Integer queryAvailableProcessors()
  {
    // we use the java.lang.Runtime API
    return Runtime.getRuntime().availableProcessors();
  }
}
```

Step 2: implement the node MBean provider interface

To make our MBean pluggable to the nodes, it must be recognized as a corresponding service instance. To this effect, we will create an implementation of the interface `JPPFNodeMBeanProvider`, which will provide the node with enough information to create the MBean and register it with the MBean server. This interface is defined as follows:

```
// service provider interface for pluggable management beans for JPPF nodes
public interface JPPFNodeMBeanProvider extends JPPFMBeanProvider
{
  // return a concrete MBean instance
  // the class of the returned MBean must implement the interface defined by
  // JPPFMBeanProvider.getMBeanInterfaceName()
  public Object createMBean(MonitoredNode node);
}
```

As we can see, this interface declares a single method whose role is to create an instance of our MBean implementation. There is no obligation to use the node parameter, it is provided here because the JPPF built-in node MBean use it. As stated in the method comment, the class of the created object must implement an MBean interface, whose name is given by the method `getMBeanInterfaceName()` in the super-interface JPPFMBeanProvider, defined as follows:

```
// service provider interface for pluggable management beans
public interface JPPFMBeanProvider
{
  // seturn the fully qualified name of the management interface
  // defined by this provider
  public String getMBeanInterfaceName();

  // return the name of the specified MBean
  // this is the name under which the MBean will be registered with the MBean server
  public String getMBeanName();
}
```

Note that the MBean name must follow the specifications for MBean object names.

We will then write our MBean provider implementation. Generally, the convention is to create it in a separate package, whose name is that of the MBean interface with a ".spi" suffix. We will write it as follows:

```
package org.jppf.example.mbean.node.spi;

import org.jppf.example.mbean.AvailableProcessors;
import org.jppf.management.spi.JPPFNodeMBeanProvider;
import org.jppf.node.MonitoredNode;

// AvailableProcessors MBean provider implementation
public class AvailableProcessorsMBeanProvider implements JPPFNodeMBeanProvider
{
  // return the fully qualified name of the MBean interface defined by this provider
  public String getMBeanInterfaceName()
  {
    return "org.jppf.example.mbean.AvailableProcessorsMBean";
  }

  // create a concrete MBean instance
  public Object createMBean(MonitoredNode node)
  {
    return new AvailableProcessors();
  }

  // return the object name of the specified MBean
  public String getMBeanName()
  {
    return "org.jppf.example.node.mbean:name=AvailableProcessors,type=node";
  }
}
```

Step 3: create the service definition file

If it doesn't already exist, we create, in the source folder, a subfolder named `META-INF/services`. In this folder, we will create a file named `org.jppf.management.spi.JPPFNodeMBeanProvider`, and open it in a text editor. In the editor, we add a single line containing the fully qualified name of our MBean provider class:

```
org.jppf.example.mbean.node.spi.AvailableProcessorsMBeanProvider
```

Step 4: deploy the MBean

First, create a jar that contains all the artifacts we have created: MBean interface, MBean implementation and MBean provider class files, along with the `META-INF/services` folder. We now have two deployment choices: we can either deploy the MBean on a single node, or deploy it on the server side to make it available to all the nodes attached to the server. To do so, we simply add our deployment jar file to the class path of the node or of the server.

Step 5: using the MBean

We can now write a simple class to test our new custom MBean:

```
package org.jppf.example.node.test;

import org.jppf.management.JMXNodeConnectionWrapper;

// simple class to test a custom node MBean
public class AvailableProcessorsMBeanTest
{
  public static void main(String...args) throws Exception
  {
    // we assume the node is running on localhost and uses the management port 12001
    JMXNodeConnectionWrapper wrapper =
      new JMXNodeConnectionWrapper("localhost", 12001);
    wrapper.connectAndWait(5000L);
    // query the node for the available processors
    int n = (Integer) wrapper.invoke(
      "org.jppf.example.mbean:name=AvailableProcessors,type=node",
      "queryAvailableProcessors", (Object[]) null, (String[]) null);
    System.out.println("The node has " + n + " available processors");
  }
}
```

7.1.3 Writing a custom server MBean

The process is almost exactly the same as for adding custom MBeans to a node. In this example, we will reuse the MBean that we wrote in the previous section, as it applies to any JVM, whether node or server.

Step 1: create the MBean interface and its implementation

We will simply reuse the interface `AvailableProcessorsMBean` and its implementation `AvailableProcessors` that we have already created.

Step 2: implement the node MBean provider interface

This time , we will implement the interface [`JPPFDriverMBeanProvider`]:

```
package org.jppf.example.mbean.driver.spi;

import org.jppf.example.mbean.AvailableProcessors;
import org.jppf.management.spi.JPPFDriverMBeanProvider;

// AvailableProcessors MBean provider implementation
public class AvailableProcessorsMBeanProvider implements JPPFDriverMBeanProvider
{
  // return the fully qualified name of the MBean interface defined by this provider
  public String getMBeanInterfaceName()
  {
    return "org.jppf.example.mbean.AvailableProcessorsMBean";
  }

  // create a concrete MBean instance
  public Object createMBean()
  {
    return new AvailableProcessors();
  }

  // return the object name of the specified MBean
  public String getMBeanName()
  {
    return "org.jppf.example.mbean:name=AvailableProcessors,type=driver";
  }
}
```

This looks almost exactly the same as for the node MBean provider, except for the following differences:

- the implemented interface is `JPPFDriverMbeanProvider`, and its `createMBean()` method takes no parameter
- we gave a different object name to our MBean: “…, **`type=driver`**”
- we created the MBean provider in a different package named `org.jppf.example.mbean.driver.spi`.

Step 3: create the service definition file

If it doesn't already exist, we create, in the source folder, a subfolder named `META-INF/services`. In this folder, we will create a file named `org.jppf.management.spi.JPPFDriverMBeanProvider`, and open it in a text editor. In the editor, we add a single line containing the fully qualified name of our MBean provider class:

```
org.jppf.example.mbean.driver.spi.AvailableProcessorsMBeanProvider
```

Step 4: deploy the MBean

Now we just create a jar that contains all the artifacts we have created: MBean interface, MBean implementation and MBean provider class files, along with the `META-INF/services` folder, and add it to the class path of the server.

Step 5: using the MBean

We can write the following simple class to test our new server custom MBean:

```
package org.jppf.example.driver.test;

import org.jppf.management.JMXDriverConnectionWrapper;

// simple class to test a custom node MBean
public class AvailableProcessorsMBeanTest
{
  public static void main(String...args) throws Exception
  {
    // we assume the server is running on localhost and uses the management port 11198
    JMXDriverConnectionWrapper wrapper =
      new JMXDriverConnectionWrapper("localhost", 11198);
    wrapper.connectAndWait(5000L);
    // query the node for the available processors
    int n = (Integer) wrapper.invoke(
      "org.jppf.example.mbean:name=AvailableProcessors,type=driver",
      "queryAvailableProcessors", (Object[]) null, (String[]) null);
    System.out.println("The server has " + n + " available processors");
  }
}
```

7.2 JPPF startup classes

Startup classes allow a piece of code to be executed at startup time of a node or server. They can be used for many purposes, including initialization of resources such as database connections, JMS queues, cache frameworks, authentication, etc ... They permit the creation of any object within the same JVM as the JPPF component they run in.

Startup classes are defined using the Service Provider Interface. The general workflow to create a custom startup class is as follows:

step 1: create a class implementing the startup class provider interface
step 2: add or update the corresponding service definition file in the `META-INF/services` folder
step 3: create a jar file containing the above elements and deploy it in the node or server class path

This mechanism relies on the following rules:

– the provider interface for a node or server startup class extends the interface `JPPFStartup`, which itself extends `java.lang.Runnable`. Thus, writing a startup class consists essentially in writing code in the `run()` method.
– the provider interface implementation must have a no-arg constructor
– startup classes are instantiated and run just after the JPPF and custom MBeans have been initialized. This allows a startup class to subscribe to any notifications that an MBean may emit.

Related sample: “Startup Classes” sample in the JPPF samples pack.

7.2.1 Node startup classes

Step 1: implement the node startup class provider interface

To make our startup class pluggable to the nodes, it must be recognized as a corresponding service instance. To this effect, we will create an implementation of the interface `JPPFNodeStartupSPI`, which will provide the node with enough information to create and run the startup class. This interface is defined as follows:

```
public interface JPPFNodeStartupSPI extends JPPFStartup { }
```

As we can see, this is just a marker interface, used to distinguish between node startup classes and server startup classes. As an example, we will create an implementation that simply prints a message when the node starts:

```java
package org.jppf.example.startup.node;

import org.jppf.startup.JPPFNodeStartupSPI;

// This is a test of a node startup class
public class TestNodeStartup implements JPPFNodeStartupSPI
{
  public void run()
  {
    System.out.println("I'm a node startup class");
  }
}
```

Step 2: create the service definition file

If it doesn't already exist, we create, in the source folder, a subfolder named `META-INF/services`. In this folder, we will create a file named `org.jppf.startup.JPPFNodeStartupSPI`, and open it in a text editor. In the editor, we add a single line containing the fully qualified name of our startup class:

```
org.jppf.example.startup.node.TestNodeStartup
```

Step 3: deploy the startup class

Now we just create a jar that contains all the artifacts we have created: JPPF node startup provider class , along with the `META-INF/services` folder, and add it to the class path of either the server, if we want all nodes attached to the server to use the startup class, or of the node, if we only want one node to use it.

Important note: *when a node startup class is deployed on the server, the objects it creates (for instance as singletons) can be reused from within the tasks executed by the node*.

7.2.2 Server startup classes

Step 1: implement the server startup class provider interface

In the same way as for a node startup class, we need to implement the interface JPPFDriverStartupSPI, defined as follows:

```java
public interface JPPFDriverStartupSPI extends JPPFStartup { }
```

As an example, we will create an implementation that simply prints a message when the server starts:

```java
package org.jppf.example.startup.driver;

import org.jppf.startup.JPPFNodeStartupSPI;

// This is a test of a server startup class
public class TestDriverStartup implements JPPFDriverStartupSPI
{
  public void run()
  {
    System.out.println("I'm a server startup class");
  }
}
```

Step 2: create the service definition file

If it doesn't already exist, we create, in the source folder, a subfolder named `META-INF/services`. In this folder, we will create a file named `org.jppf.startup.JPPFDriverStartupSPI`, and open it in a text editor. In the editor, we add a single line containing the fully qualified name of our startup class:

```
org.jppf.example.startup.driver.TestDriverStartup
```

Step 3: deploy the startup class

Now we just create a jar that contains the JPPF server startup provider class , along with the `META-INF/services` folder, and add it to the class path of the server.

7.3 Transforming and encrypting networked data

In JPPF, most of the network traffic is made of serialized Java objects. By default, these serialized objects are sent over the network without any obfuscation or encryption of any sort. This can be considered risky in highly secured environments. To mitigate this risk, JPPF provides a hook that enables transforming a block of data into another block of data, and transform it back into the orignal data (reverse tranformation).

To better understand how this mechanism works, let's first have a high-level overview of how JPPF components send and receive messages over the network. A message in JPPF is composed of a number of blocks of data, each block representing a serialized object (or object graph) and immediately preceded by its own length. A message would look like this:

L_1	$Block_1$	. . . . .	L_n	$Block_n$

Where:

– $Block_1$, ..., $Block_n$ are separate blocks of data constituting the message
– L_1, ..., L_n are the lengths of each block of data

The data transformation hook allows developers to transform each block of data. The block lengths are always computed by JPPF. For example if the data transformation used is a form of encryption (and decryption for the reverse operation), then everything except the block lengths will be encrypted.

Related sample: “Data Encryption” sample in the JPPF samples pack

The general workflow to implement and deploy a data transformation is as follows:

Step 1: implement the `JPPFDataTransform` interface

This interface is defined as follows:

```
public interface JPPFDataTransform
{
  // Transform a block of data into another, transformed one.
  // This operation must be such that the result of unwrapping the data of the
  // destination must be the equal to the source data
  void wrap(InputStream source, OutputStream destination) throws Exception;

  // Transform a block of data into another, reverse-transformed one
  // This method is the reverse operation with regards to wrap()
  void unwrap(InputStream source, OutputStream destination) throws Exception;
}
```

One very important thing to note is that the sequential application of the `wrap()` and `unwrap()` methods must return exactly the original data.

Also keep in mind that the data transformation is completely stateless. For instance there is no knowledge of where the data comes from or where it is going.

We will now write a data transformation that encrypts data using the DES cryptographic algorithm, based on a 56 bits symetric secret key. This code is available in the related “Data Encryption” sample of the JPPF samples pack. Note that this example is far from totally secure, since the secret key is actually stored with the source code (and in the resulting jar file). It should normally be in a secure location such as a key store. The packaging in the sample is only for demonstration purposes.

Here is our implementation of `JPPFDataTransform`:

```java
// Data transform that uses the DES cyptographic algorithm with a 56 bits secret key
public class SecureKeyCipherTransform implements JPPFDataTransform
{
  // Secret (symetric) key used for encryption and decryption
  private static SecretKey secretKey = getSecretKey();

  // Encrypt the data using streams
  public void wrap(InputStream source, OutputStream dest) throws Exception
  {
    // create a cipher instance
    Cipher cipher = Cipher.getInstance(Helper.getTransformation());
    // initialize the cipher with the key stored in the secured keystore
    cipher.init(Cipher.WRAP_MODE, getSecretKey());
    // generate a new key that we will use to encrypt the data
    SecretKey key = generateKey();
    // encrypt the new key, using the secret key found in the keystore
    byte[] keyBytes = cipher.wrap(key);
    // now we write the encrypted key before the data
    DataOutputStream dos = new DataOutputStream(dest);
    // write the key length
    dos.writeInt(keyBytes.length);
    // write the key content
    dos.write(keyBytes);

    // get a new cipher for the actual encryption
    cipher = Cipher.getInstance(Helper.getTransformation());
    // init the cipher in encryption mode
    cipher.init(Cipher.ENCRYPT_MODE, key);
    // obtain a cipher output stream
    CipherOutputStream cos = new CipherOutputStream(dest, cipher);
    // finally, encrypt the data using the new key
    transform(source, cos);
    cos.close();
  }

  // Decrypt the data
  public void unwrap(InputStream source, OutputStream dest) throws Exception
  {
    // start by reading the secret key to use to decrypt the data
    DataInputStream dis = new DataInputStream(source);
    // read the length of the key
    int keyLength = dis.readInt();
    // read the encrypted key
    byte[] keyBytes = new byte[keyLength];
    dis.read(keyBytes);
    // decrypt the key using the initial key stored in the keystore
    Cipher cipher = Cipher.getInstance(Helper.getTransformation());
    cipher.init(Cipher.UNWRAP_MODE, getSecretKey());
    SecretKey key = (SecretKey) cipher.unwrap(
     keyBytes, Helper.getAlgorithm(), Cipher.SECRET_KEY);

    // get a new cipher for the actual decryption
    cipher = Cipher.getInstance(Helper.getTransformation());
    // init the cipher in decryption mode
    cipher.init(Cipher.DECRYPT_MODE, key);
    // obtain a cipher input stream
    CipherInputStream cis = new CipherInputStream(source, cipher);
    // finally, decrypt the data using the new key
    transform(cis, dest);
    cis.close();
  }

  // Generate a secret key
  private SecretKey generateKey() throws Exception
  {
    KeyGenerator gen = KeyGenerator.getInstance(Helper.getAlgorithm());
    return gen.generateKey();
  }
```

```
  // Transform the specified input source and write it to the specified destination
  private void transform(InputStream source, OutputStream dest) throws Exception
  {
    byte[] buffer = new byte[8192];
    while (true)
    {
      int n = source.read(buffer);
      if (n <= 0) break;
      destination.write(buffer, 0, n);
    }
  }

  // Get the secret key used for encryption/decryption
  private static synchronized SecretKey getSecretKey()
  {
    if (secretKey == null)
    {
      try
      {
        // get the keystore password
        char[] password = Helper.getPassword();
        ClassLoader cl = SecureKeyCipherTransform.class.getClassLoader();
        InputStream is = cl.getResourceAsStream(
          Helper.getKeystoreFolder() + Helper.getKeystoreFilename());
        KeyStore ks = KeyStore.getInstance(Helper.getProvider());
        // load the keystore
        ks.load(is, password);
        // get the secret key from the keystore
        secretKey = (SecretKey) ks.getKey(Helper.getKeyAlias(), password);
      }
      catch(Exception e)
      {
        e.printStackTrace();
      }
    }
    return secretKey;
  }
}
```

Step 2: deploy the data transform implementation

The implementation code and related resources must be deployed in the class path of ***each and every component on the JPPF grid***, including servers, nodes, and client applications. If it is not the case, the results are unpredictable and JPPF will probably stop working altogether. The deployment can be made in the form of a jar file or a class folder, the only constraint being that it must be local to the JVM of each JPPF component.

Step 3: hook the implementation to JPPF

This is done by specifying the property **jppf.data.transform.class** in the JPPF configuration file of each component:

```
jppf.data.transform.class = <fully qualified name of implementation class>
```

In our example it would be:

```
jppf.data.transform.class = org.jppf.example.dataencryption.SecureKeyCipherTransform
```

7.4 *Specifying alternate object streams*

JPPF performs objects transport and associated serialization by the means of object streams, which are instances of `ObjectInputStream` and `ObjectOutputStream` or subclasses of these classes.

It is now possible to specify alternate object stream classes for a JPPF grid, enabling the use of non-serializable classes without any extra coding required for the JPPF task developer. JPPF provides 2 ways to achieve this:

7.4.1 Specifying the object stream implementation classes

This is done in the JPPF configuration file, by adding these 2 properties:

```
# configure the object input stream implementation
jppf.object.input.stream.class = my.package.MyObjectInputStream
# configure the object output stream implementation
jppf.object.output.stream.class = my.package.MyObjectOutputStream
```

Please note that the object stream implementations must have a constructor that takes an `InputStream` parameter for the object input stream class, and an `OutputStream` parameter for the object output stream class.

7.4.2 Implementing an object stream builder

An object stream builder is an object that instantiates input and output object streams.
It is defined as an implementation of the `JPPFObjectStreamBuilder` interface:

```
// Interface for all builders instantiating alternate object input and output streams.
public interface JPPFObjectStreamBuilder
{
  // Obtain an input stream used for deserializing objects.
  public ObjectInputStream newObjectInputStream(InputStream in) throws Exception;

  // Obtain an Output stream used for serializing objects.
  public ObjectOutputStream newObjectOutputStream(OutputStream out) throws Exception;
}
```

Then, configure JPPF to use this object stream builder by specifying the following property in the JPPF configuration file:

```
# configure the object stream builder implementation
jppf.object.stream.builder = my.package.MyObjectStreamBuilder
```

Note: *when alternate object streams are specified, they must be used by all JPPF clients, servers and nodes, otherwise JPPF will not work. The implementation classes must also be present in the classpath of all JPPF components*

7.4.3 Built-in implementations

Out of the box, JPPF provides 3 serialization schemes:

7.4.3.1 Default serialization

This is the default Java serialization mechanism, using the known JDK classes `java.io.ObjectInputStream` and `java.io.ObjectOutputStream`. It is used by default, when no serialization scheme is specified.

7.4.3.2 Generic JPPF serialization

This is a serialization scheme implemented from scratch, which functions pretty much like the standard Java mechanism with one major difference: *it enables the serialization of classes that do not implement `java.io.Serializable` nor `java.io.Externalizable`*. This allows developers to use classes in their tasks that are not normally serializable and for which they cannot access the source code. We understand that it breaks the contract specified in the JDK for serialization, however it provides an effective workaround for dealing with non-serializable classes in JPPF jobs and tasks.

The JPPF implementation relies on an extension of the standard mechanism by defining 2 new classes: `JPPFObjectInputStream` and `JPPFObjectOutputStream`.

Apart from this, it conforms to the specifications for the standard ObjectInputStream and ObjectOutputStream classes, in that it processes transient fields in the same manner, and handles the special cases when a class implements the methods writeObject(ObjectOutputStream) and readObject(ObjectInputStream), and the `java.io.Externalizable` interface.

This implementation is also slower than the default Java one: serialization and deserialization of an object graph takes generally around 50% more time. This overhead will be signifcant essentially for very short-lived tasks (i.e. a few milliseconds). It is thus recommended to use the default Java serialization whenever it is possible.

To specify this scheme in your JPPF configuration:

```
# configure the object stream builder implementation
jppf.object.stream.builder = org.jppf.serialization.GenericObjectStreamBuilder

# configure the object input stream implementation
jppf.object.input.stream.class = org.jppf.serialization.JPPFObjectInputStream
# configure the object output stream implementation
jppf.object.output.stream.class = org.jppf.serialization.JPPFObjectOutputStream
```

7.4.3.3 XStream-based serialization

JPPF has a built-in Object Stream Builder that uses XStream to provide XML serialization: `XstreamObjectStreamBuilder`. To use it, simply specify:

```
# configure the object stream builder implementation
jppf.object.stream.builder = org.jppf.serialization.XstreamObjectStreamBuilder
```

in the JPPF configuration files.

You will also need the XStream 1.3 (or later) jar file and the xpp3 jar file available in the XStream distribution

7.5 Creating a custom load-balancer

Related sample: "CustomLoadBalancer" in the JPPF samples pack.

7.5.1 Overview of JPPF load-balancing

Load-balancing in JPPF relates to the way jobs are split into sub-jobs and how these sub-jobs are disptached to the nodes for execution in parallel. Each sub-job contains a distinct subset of the tasks in the original job.

The distribution of the tasks to the nodes is performed by the JPPF driver. This work is actually the main factor of the observed performance of the framework. It consists essentially in determining how many tasks will go to each node for execution, out of a set of tasks sent by the client application. Each set of tasks sent to a node is called a "bundle", and the role of the load balancing (or task scheduling) algorithm is to optimize the performance by adjusting the number of task sent to each node. In short: it is about computing the optimal bundle size for each node.

Each load-balancing algorithm is encapsulated within a class implementing the interface `Bundler`, defined as follows:

```
public interface Bundler
{
  // Get the latest computed bundle size
  public int getBundleSize();

  // Feed the bundler with the latest execution result for the corresponding node
  public void feedback(int nbTasks, double totalTime);

  // Make a copy of this bundler
  public Bundler copy();

  // Get the timestamp at which this bundler was created
  public long getTimestamp();

  // Release the resources used by this bundler
  public void dispose();

  // Perform context-independant initializations
  public void setup();

  // Get the parameters profile used by this load-balancer
  public LoadBalancingProfile getProfile();
}
```

In practice, it will be more convenient to extend the abstract class `AbstractBundler`, which provides a default implementation for each method of the interface.

The load balancing in JPPF is feedback-driven. The server will create a `Bundler` instance for each node that is attached to it. When a set of tasks returns from a node after execution, the server will call the bundler's `feedback()` method so the bundler can recompute the bundle size with up-to-date data. Whether each bundler computes the bundle size independantly from the other bundlers is entirely up to the implementor. Some of the JPPF built-in algorithms do perform independent computations, others don't.

A bundler's life cycle is as follows:

– when the server starts up, it creates a bundler instance based on the load-balancing algorithm specified in the configuration file
– each time a node connects to the server, the server will make a copy of the initial bundler, using the `copy()` method, call the `setup()` method, and assign the new bundler to the node
– when a node is disconnected, the server will call the `dispose()` method on the corresponding bundler, then discard it
– when the load balancing settings are changed using the management APIs or the administration console, the server will create a new initial Bundler instance, based on the new parameters. Then, each time the server needs to provide feedback data from a node, the server will compare the creation timestamps of the initial bundler and of the node's bundler. If the server determines that the node's bundler is older, it will replace it with a copy of the initial bundler, using the `copy()` method and after calling the `setup()` method on the new bundler

Each bundler has an associated load balancing profile, which encapsulates the parameters of the algorithm. These parameters can be read from the JPPF configuration file, or from any other source. Using a profile is not mandatory, in this case you can just have the `getProfile()` method return a `null` value.

In the following sections, we will see in details how to implement a custom load-balancing algorithm, deploy it, and plug it into the JPPF server. We will do this by example, using the built-in “Fixed Size” algorithm, which is simple enough for our purpose.

Note: *all JPPF built-in load balancing algorithms are implemented and plugged-in as custom algorithms*

7.5.2 Implementing the algorithm and its associated profile

First let's implement our parameters profile. To this effect, we implement the interface `LoadBalancingProfile`:

```
public interface LoadBalancingProfile extends Serializable
{
  // Make a copy of this profile
  public LoadBalancingProfile copy();
}
```

As we can see, this interface has a single method that creates a copy of a profile. Now let's see how it is implemented in the `FixedSizeProfile` class:

```
// Profile for the fixed bundle size load-balancing algorithm
public class FixedSizeProfile implements LoadBalancingProfile
{
  // The bundle size
  private int size = 1;

  // Default constructor
  public FixedSizeProfile()
  {
  }

  // Initialize this profile with values read from the specified configuration
  public FixedSizeProfile(TypedProperties config)
  {
    size = config.getInt("size", 1);
  }

  // Make a copy of this profile
  public LoadBalancingProfile copy()
  {
    FixedSizeProfile other = new FixedSizeProfile();
    other.setSize(size);
    return other;
  }

  // Get the bundle size
  public int getSize()
  {
    return size;
  }

  // Set the bundle size
  public void setSize(int size)
  {
    this.size = size;
  }
}
```

This implementation is fairly trivial, the only notable element being the constructor taking a `TypedProperties` parameter, which will allow us to read the size parameter from the JPPF configuration file.

Now let's take a look at the algorithm implementation itself:

```
public class FixedSizeBundler extends AbstractBundler
{
  // Initialize this bundler
  public FixedSizeBundler(LoadBalancingProfile profile)
  {
    super(profile);
  }

  // This method always returns a statically assigned bundle size
  public int getBundleSize()
  {
    return ((FixedSizeProfile) profile).getSize();
  }

  // Make a copy of this bundler
  public Bundler copy()
  {
    return new FixedSizeBundler(profile.copy());
  }

  // Get the max bundle size that can be used for this bundler
  protected int maxSize()
  {
    return -1;
  }
}
```

The first thing we can notice is that the `feedback()` method is not even implemented! This is due to the fact that our algorithm is independent from the context and involves no computation. Thus, we use the default implementation in `AbstractBundler`, which does nothing. This is visible in the `getBundleSize()` method, where we simply return the value provided in the parameters profile.

We also notice a new method named `maxSize()`. It returns a value representing the maximum bundle size that a bundler can use at a given time. The goal of this is to avoid that a node receives all or most of the tasks, while the other nodes would not receive anything and thus would have nothing to do. This method is declared in the abstract class `AbstractBundler` and doesn't have any default implementation, to avoid any tight coupling between the bundler and the environment in which it runs. This allows the bundler to be used outside of the JPPF server, as is done for instance in the JPPF client when local execution mode is used along with remote execution.

In the context of the server, we have found that an efficient value for `maxSize()` can be computed from the current maximum number of tasks among all the jobs in the server queue. This value is accessible by calling the method `JPPFQueue.getMaxBundleSize()`. We could then rewrite our maxSize() method as follows:

```
protected int maxSize()
{
  return JPPFDriver.getQueue().getMaxBundleSize() / 2;
}
```

The algorithm could then determine that a node should not receive more than half of that value (or 75% or any other function of it, whatever is deemed more efficient), so that other nodes will not be idle and the overall throughput will be optimized.

Tip: *if your algorithm depends on the number of nodes, you can use a bundler instances count as a static variable in your implementation, and use the* `setup()` *and* `dispose()` *methods to increment and decrement the count as needed. For instance*:

```
private static AtomicInteger instanceCount = new AtomicInteger(0);
public void setup()
{
  instanceCount.incrementAndGet();
}
public void dispose()
{
  instanceCount.decrementAndGet();
}
```

7.5.3 Implementing the bundler provider interface

Custom load-balancers are defined and deployed using the Service Provider Interface (SPI) mechanism. For a new load-balancer to be recognized by JPPF, it has to provide an implementation of the `JPPFBundlerProvider` interface, which is defined as:

```
public interface JPPFBundlerProvider
{
  // Get the name of the algorithm defined by this provider
  // Each algorithm must have a name distinct from that of all other algorithms
  public String getAlgorithmName();

  // Create a bundler instance using the specified parameters profile
  public Bundler createBundler(LoadBalancingProfile profile);

  // Create a bundler profile containing the parameters of the algorithm
  public LoadBalancingProfile createProfile(TypedProperties configuration);
}
```

In the case of our fixed size algorithm, the `FixedSizeBundlerProvider` implementation is quite straightforward:

```
public class FixedSizeBundlerProvider implements JPPFBundlerProvider
{
  // Get the name of the algorithm defined by this provider
  public String getAlgorithmName()
  {
    return "manual";
  }

  // Create a bundler instance using the specified parameters profile
  public Bundler createBundler(LoadBalancingProfile profile)
  {
    return new FixedSizeBundler(profile);
  }

  // Create a bundler profile containing the parameters of the algorithm
  public LoadBalancingProfile createProfile(TypedProperties configuration)
  {
    return new FixedSizeProfile(configuration);
  }
}
```

7.5.4 Deploying the custom load-balancer

For our custom load-balancer to be recognized and loaded, we need to create the corresponding service definition file. If it doesn't already exist, we create, in the source folder, a subfolder named `META-INF/services`. In this folder, we will create a file named `org.jppf.server.scheduler.bundle.providers.FixedSizeBundlerProvider`, and open it in a text editor. In the editor, we add a single line containing the fully qualified name of our provider implementation:

```
org.jppf.server.scheduler.bundle.providers.FixedSizeBundlerProvider
```

Now, to actually deploy our implementation, we will create a jar file that contains all the artifacts we have created: the `Bundler`, `LoadBalancingProfile` and `JPPFBundlerProvider` implementation classes, along with the `META-INF/services` folder, and add this jar to the class path of the server.

7.5.5 Node-aware load balancers

Load balancers can be made aware of a node's environment and configuration, and make dynamic decisions based on this information.

To this effect, the Bundler implementation will need to also implement the interface NodeAwareness, defined as follows:

```
// Bundler implementations should implement this interface
// if they wish to have access to a node's configuration
public interface NodeAwareness
{
  // Get the corresponding node's system information
  JPPFSystemInformation getNodeConfiguration();

  // Set the corresponding node's system information
  void setNodeConfiguration(JPPFSystemInformation nodeConfiguration);
}
```

When implementing this interface, the environment and configuration of the node become accessible via an instance of JPPFSystemInformation.

JPPF guarantees that the node information will never be null once the node is connected to the server. You should not assume, however, that it is true when the Bundler is instantiated (for instance in the constructor).

The method `setConfiguration()` can be called in two occasions:

- when the node connects to the server
- when the node's number of processing threads has been updated dynamically (through the admin console or management APIs)

A sample usage of NodeAwareness can be found in the CustomLoadBalancer sample, in the JPPF samples pack.

7.5.6 Job-aware load balancers

Load-balancers can gain access to a job's metadata (see the “Job Metadata” section of the Development Guide). This is done by having the Bundler implement the interface JobAwareness, defined as follows:

```
// Bundler implementations should implement this interface
// if they wish to have access to a job's metadata
public interface JobAwareness
{
  // Get the current job's metadata
  JPPFJobMetadata getJobMetadata();

  // Set the current job's metadata
  void setJobMetadata(JPPFJobMetadata metadata);
}
```

When implementing this interface, the job metadata becomes accessible via an instance of JPPFJobMetadata.

The method `setJobMetadata()` is always called after the execution policy (if any) has been applied to the node, and before the job is dispatched to the node for execution. This allows the load-balancer to use information about the job when computing the number of tasks to send to the node.

A sample usage of JobAwareness can be found in the CustomLoadBalancer sample, in the JPPF samples pack.

7.6 Receiving notifications of node life cycle events

This plugin provides the ability to receive notifications of major events ocurring within a node, including node startup and termination as well as the start and completion of each job processing.

To achieve this, you only need to implement the interface NodeLifeCycleListener, which is defined as follows:

```
public interface NodeLifeCycleListener extends EventListener
{
  // Called when the node has finished initializing,
  // and before it starts processing jobs
  void nodeStarting(NodeLifeCycleEvent event);

  // Called when the node is terminating
  void nodeEnding(NodeLifeCycleEvent event);

  // Called before the node starts processing a job
  void jobStarting(NodeLifeCycleEvent event);

  // Called after the node finishes processing a job
  void jobEnding(NodeLifeCycleEvent event);
}
```

Each method in the listener receives an event of type NodeLifeCycleEvent, which provides the following API:

```
public class NodeLifeCycleEvent extends EventObject
{
  // Get the job currently being executed
  public JPPFDistributedJob getJob();

  // Get the tasks currently being executed
  public List<JPPFTask> getTasks();
}
```

Please note that the two methods getJob() and getTasks() will return null for the events of type “nodeStarting” and may return null for “nodeEnding” events, as the node may not be processing any job at the time these events occur.

JPPFDistributedJob is an interface common to client side jobs (see JPPFJob) and server / node side jobs (see JPPFTaskBundle). It provides the following methods, which can be used in the the NodeLifeCycleListener implementation:

```
public interface JPPFDistributedJob
{
   // Get the user-defined display name for this job
   // This is the name displayed in the administration console
  String getId();

  // Get the universal unique id for this job
  String getJobUuid();

  // Get the service level agreement between the job and the server
  JPPFJobSLA getJobSLA();

  // Get the user-defined metadata associated with this job
  JPPFJobMetadata getJobMetadata();
}
```

Once the implementation is done, the listener is hooked up to JPPF using the service provider interface:

- create a file in META-INF/services named “org.jppf.node.event.NodeLifeCycleListener”
- in this file, add the fully qualified class name of your implementation of the interface
- copy the jar file or class folder containing your implementation and service file to the JPPF driver's class path. Note that this plugin cannot be deployed to individual nodes, due to class loading constraints. It must be deployed onto the server's class path - thus to every node - or not deployed at all

Here is a simple example illustrating the process.

Our implementation of the `NodeLifeCycleListener` interface, which simply prints the events to the node's console:

```
package myPackage;

public class MyNodeListener implements NodeLifeCycleListener
{
  public void nodeStarting(NodeLifeCycleEvent event)
  {
    System.out.println("node ready to process jobs");
  }

  public void nodeEnding(NodeLifeCycleEvent event)
  {
    System.out.println("node ending");
  }

  public void jobStarting(NodeLifeCycleEvent event)
  {
    JPPFDistributedJob job = event.getJob();
    System.out.println("node starting job '" + job.getId() + "' with " +
      event.getTasks().size() + " tasks");
  }

  public void jobEnding(NodeLifeCycleEvent event)
  {
    System.out.println("node finished job '" + event.getJob().getId() + "'");
  }
}
```

Once this is done, we create the file `META-INF/services/org.jppf.node.event.NodeLifeCycleListener` with the following content:

```
myPackage.MyNodeListener
```

Our node listener is now ready to be deployed.

Related JPPF sample: “NodeLifeCycle” in the JPPF samples pack.

8 J2EE Connector

8.1 Overview of the JPPF Resource Adapter

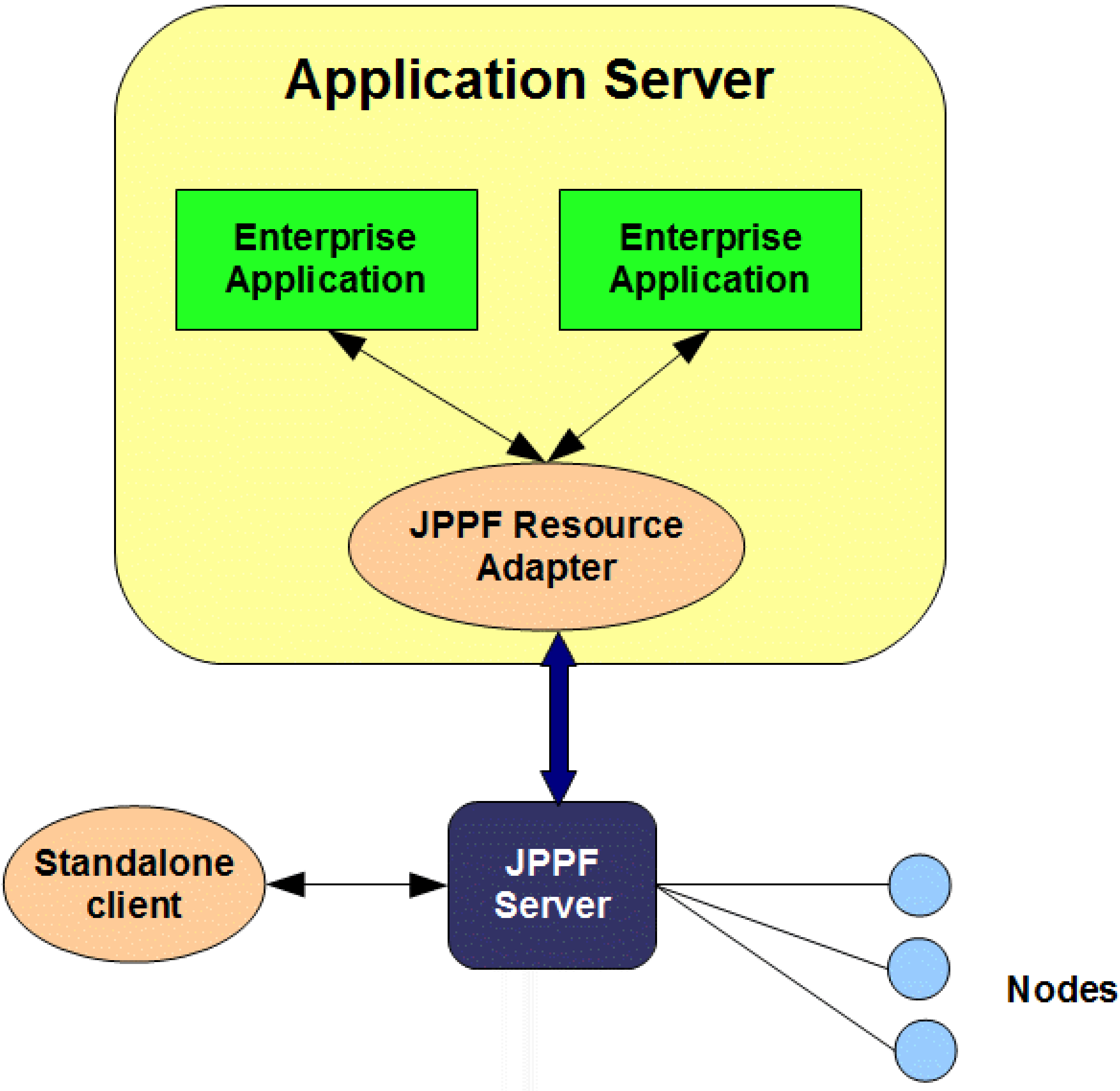
8.1.1 What is it?

The JPPF Resource Adapter is a JCA-compliant resource adapter module that encapsulates a JPPF client. It provides J2EE application servers with an access to JPPF grid services. It is intended for deployment as either a standalone module or embedded within an enterprise application, while preserving the ease of use of JPPF.

8.1.2 Features

- Supports the same configuration properties as a standalone JPPF client, including server discovery, connection to multiple drivers, connection pool per driver
- Supports disconnection from, and reconnection to, any JPPF driver
- Compliant with the JCA 1.5 specifications
- API similar to that of the standard JPPF client (`submit(job)`)
- No transaction support

8.1.3 Architecture



8.2 Supported Platforms

Note: *we have not tested all versions of all platforms, thus it is possible that the integration, with the version of the application server you are using, may not work out-of-the-box. We hope that the existing framework and instructions will allow you to adapt the build scripts and code to do so. If you have issues with a specific port, or if you wish to report one that is not specified in this documentation, we invite you to provide your feedback and comments on the JPPF Forums*.

Technology	Tested Platforms
Operating System	All Windows systems supporting JDK 1.5 or later All Linux and Unix systems supporting JDK 1.5 or later
JVM	Sun JDK 1.5 and later IBM JVM 1.5 and later BEA JRockit 1.5 and later
Application Server	JBoss 4.0.x, 4.2.x, 5.0.x, 5.1.x Oracle OC4J 10.1.3.x Sun AS 9.0.x, Glassfish 2.x IBM Websphere Application Server 6.1.x, 7.0.x Oracle Weblogic 9.x, 10.x Apache Geronimo 2.1.x

8.3 Configuration and build

8.3.1 Requirement

For building, configuring and customizing the JPPF Resource Adapter, your will need the latest version of the JPPF source code distribution. It can be found on the JPPF download page. The name of the file should be '''JPPF-2.x-j2ee-connector.zip'''

8.3.2 Build structure

The J2EE connector has the following folder structure:

Folder	Description
root folder	The root folder, contains the build.xml Ant build script
appserver	contains common and application server-specific configurations for the JPPF resource adapter and the demo application
build	this folder contains all jars, .ear and .rar files resulting from the build
classes	contains the compiled code of the JPPF resource adapter
config	contains application server-specific resources for the deployment of the resource adapter
docroot	contains resources for the build of the demo application on some application servers
src	contains the source code for the resource adapter and demo application.

8.3.3 Building the JPPF resource adapter

To build the resource adapter:

- Open a command prompt
- Go to the **JPPF-2.x-j2ee-connector** folder
- Enter "**ant build**"
- The resulting .rar and .ear files are generated in the "**build**" subfolder.

8.3.4 Configuring the resource adapter and demo application

The configuration files and deployment descriptors are all contained in the '"appserver"' folder. The detailed content of this folder is as follows:

Folder	Description

appserver	root folder, contains:
	
	– the resource adapter's deployment descriptor ra.xml. It is in this file that you set the configuration parameters for the connection to the JPPF drivers, by setting the values of the configuration properties as follows:
	
	<config-property>
	  <config-property-name>ClientConfiguration</config-property-name>
	  <config-property-type>java.lang.String</config-property-type>
	  <config-property-value>
	    # any server on the network will be automatically detected
	    jppf.local.execution.enabled = true
	    # create a pool of 5 connections for each detected server
	    jppf.pool.size = 5
	  </config-property-value>
	</config-property>
	
	– the demo application's deployment desciptor application.xml

appserver/common	contains files common to all application servers, for the demo enterprise application
appserver/<server_name>	root of <server_name>-specific configuration and deployment files. Contains a commons-logging.properties file where you can configure which logging framework will be used (i.e. Log4j, JDK logger, etc...)
appserver/<server_name> /application	contains <server_name>-specific deployment descriptor for the demo application, for example: weblogic-application.xml.
appserver/<server_name> /docroot	contains a <server_name>-specific JSP for the demo application. The specificity is the JNDI name used to look up the JPPF connection factory. It relates to the corresponding resource-ref defined in the web.xml descriptor.
appserver/<server_name> /ra	contains a <server_name>-specific deployment descriptor for the resource adapter. It generally contains the definition of the corresponding JCA connection factory. Not all application servers require one. Example: weblogic-ra.xml.
appserver/<server_name> /WEB-INF	contains the <server_name>-specific deployment descriptors for the demo web application. The specificity is mostly in the resource-ref definition of the JNDI name ofr the JPPF connection factory. For example: web.xml and jboss-web.xml.

8.4 How to use the connector API

To use the JPPF Resource Adapter from your code, follow these steps:

8.4.1 Submitting the tasks

In the following example, a JPPF job is submitted asynchronously. The submission returns an ID that can be used later on to check on the job status and retrieve its results.

```
// Perform a JNDI lookup of the JPPF connection factory
InitialContext ctx = new InitialContext();
Object objref = ctx.lookup("eis/JPPFConnectionFactory");
ConnectionFactory factory =
  (ConnectionFactory) PortableRemoteObject.narrow(objref, ConnectionFactory.class);
// Obtain a JPPF Connection from the connection factory
JPPFConnection connection = (JPPFConnection) factory.getConnection();
// Use the connection to submit your JPPF job
JPPFJob job = new JPPFJob();
job.addTask(new DemoTask());
// Use the connection to submit your JPPF tasks and obtain a submission ID
String submitID = connection.submitNonBlocking(job);
// close the connection
connection.close();
```

8.4.2 Checking the status and obtaining the results

Here, we check on the status of a job and process the execution results or the resulting error:

```
// Obtain a JPPF Connection from the connection factory
JPPFConnection connection = getConnection();
// Use the connection to check the status from the submission ID
JPPFSubmissionResult.Status status = connection.getSubmissionStatus(submitID);
if (status.equals(JPPFSubmissionResult.Status.COMPLETE))
{
  // if successful process the results
  List<JPPFTask> results = connection.getSubmissionResults(submitID);
}
else if (status.equals(JPPFSubmissionResult.Status.FAILED))
{
  // if failed process the errors
}
// close the connection
connection.close();
```

8.4.3 Synchronous execution

It is also possible to execute a job synchronously, without having to code the job submission and status checking in two different methods. The JPPFConnection API provides the method `waitForResults(String submitID)`, which waits until the job has completed and returns the execution results. Here is an example use:

```
// Submit a  job with the specified number of tasks and return the execution results
public List<JPPFTask> submitBlockingJob(int nbTasks) throws Exception
{
  List<JPPFTask> results = null;
  JPPFConnection connection = null;
  try
  {
    // get a connection to the resource adapter
    connection = getConnection();
    // create a new job
    JPPFJob job = new JPPFJob();
    job.setId("test job");
    // add the tasks to the job
    for (int i=0; i<nbTasks; i++) job.addTask(new MyTask(i));
    // submit the job and get the submission id
    String submitID = connection.submitNonBlocking(job);
    // wait until the job has completed
    results = connection.waitForResults(submitID);
  }
  finally
  {
    if (connection != null) connection.close();
  }
  // now return the results
  return results;
}
```

Please note that, when using the synchronous submission mode from within a transaction, you must be careful as to how long the job will take to execute. If the job execution is too long, this may cause the transaction to roll back if the execution time is longer than the transaction timeout.

8.4.4 Using submission status events

Example:

```
JPPFConnection connection = getConnection();
JPPFJob job = new JPPFJob();
JPPFTask task = new DurationTask(duration);
job.addTask(task);

// A status listener can be added at submission time
// using another form of the method JPPFConnection.submitNonBlocking()
String id = connection.submitNonBlocking(job, new SubmissionStatusListener()
{
  public void submissionStatusChanged(SubmissionStatusEvent event)
  {
    String id = event.getSubmissionId();
    SubmissionStatus status = event.getStatus();
    System.out.println("submission [" + id + "] changed to '" + status + "'");
  }
});

// or after the tasks have been submitted
connection.addSubmissionStatusListener(id, new SubmissionStatusListener()
{
  public void submissionStatusChanged(SubmissionStatusEvent event)
  {
    String id = event.getSubmissionId();
    SubmissionStatus status = event.getStatus();
    switch(status)
    {
      case COMPLETE:
        // process successfull completion
        break;
      case FAILED:
        // process failure
        break;
      default:
        System.out.println("submission [" + id + "] changed to '" + status + "'");
        break;
    }
  }
});
connection.close();
```

8.5 Deployment on a J2EE application server

8.5.1 Deployment on JBoss 4.x - 6.x

8.5.1.1 Deploying the JPPF resource adapter

copy the file ""jppf_ra_JBoss.rar"" in this folder: <JBOSS_HOME>/server/<your_server>/deploy, where <JBOSS_HOME> is the root installation folder for JBoss, and <your_server> is the server configuration that you use (JBoss comes with 3 configurations: "default", "minimal" and "all")

8.5.1.2 Creating a connection factory

Create, in the <JBOSS_HOME>/server/<your_server>/deploy folder, a file named jppf-ra-JBoss-ds.xml. Edit this file with a text editor and add this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
  <no-tx-connection-factory>
    <jndi-name>eis/JPPFConnectionFactory</jndi-name>
    <application-managed-security/>
    <rar-name>jppf_ra_JBoss-rar</rar-name>
    <connection-definition>javax.resource.cci.ConnectionFactory</connection-definition>
    <adapter-display-name>JPPF</adapter-display-name>
    <min-pool-size>0</min-pool-size>
    <max-pool-size>10</max-pool-size>
    <blocking-timeout-millis>50000</blocking-timeout-millis>
    <idle-timeout-minutes>15</idle-timeout-minutes>
  </no-tx-connection-factory>
</connection-factories>
```

You can also download this file.

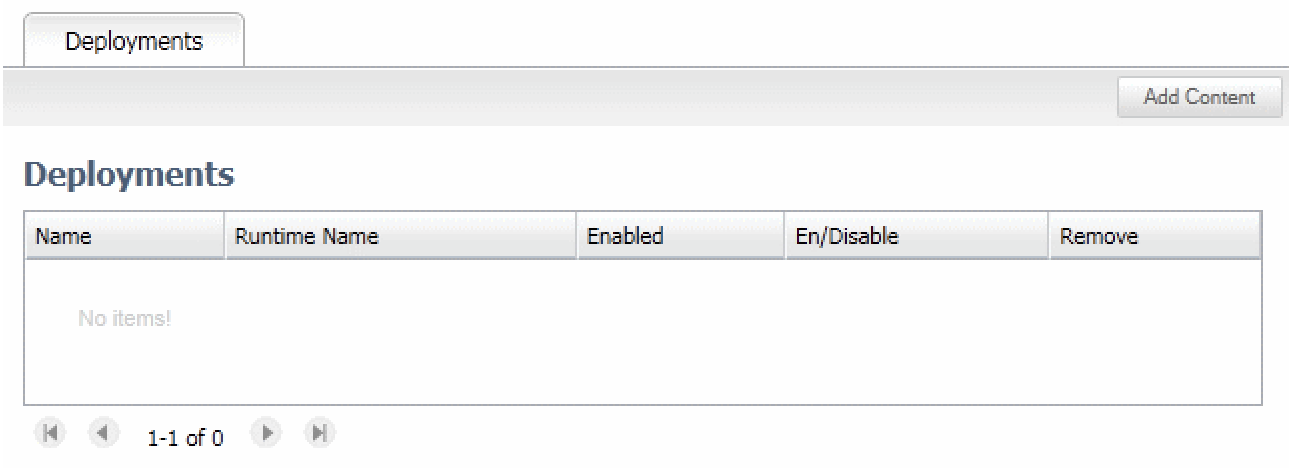
8.5.1.3 Deploying the demo application

Copy the file “JPPF_J2EE_Demo_JBoss-4.0.ear” in the <JBOSS_HOME>/server/<your_server>/deploy folder

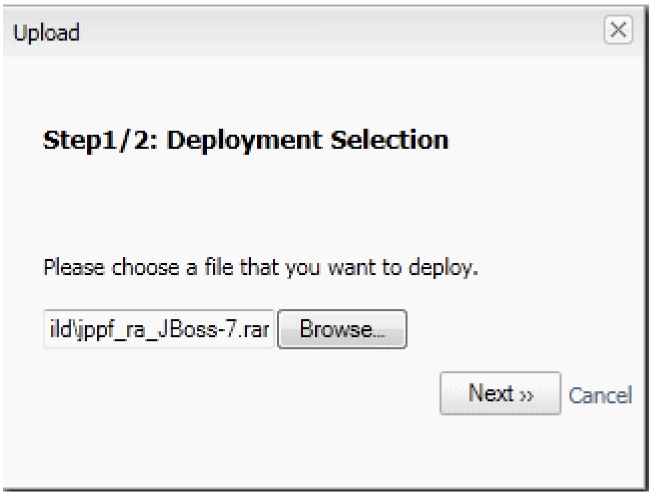
8.5.2 Deployment on JBoss 7+

8.5.2.1 Deploying the resource adapter

- in the JBoss administration console, go to the "Deployments" view
- remove any existing JPPF deployments (.rar and .ear)

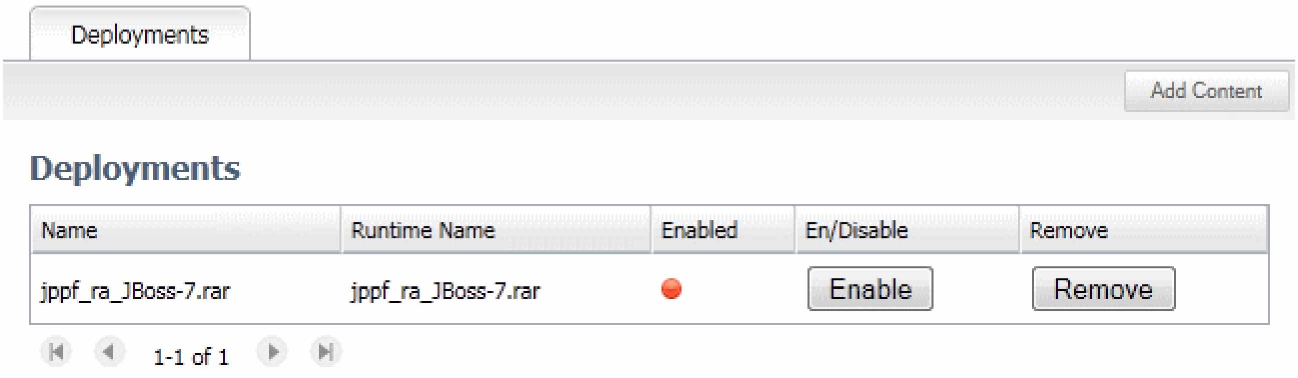


- click on "Add Content"
- browse to the file "jppf_ra_JBoss-7.rar"

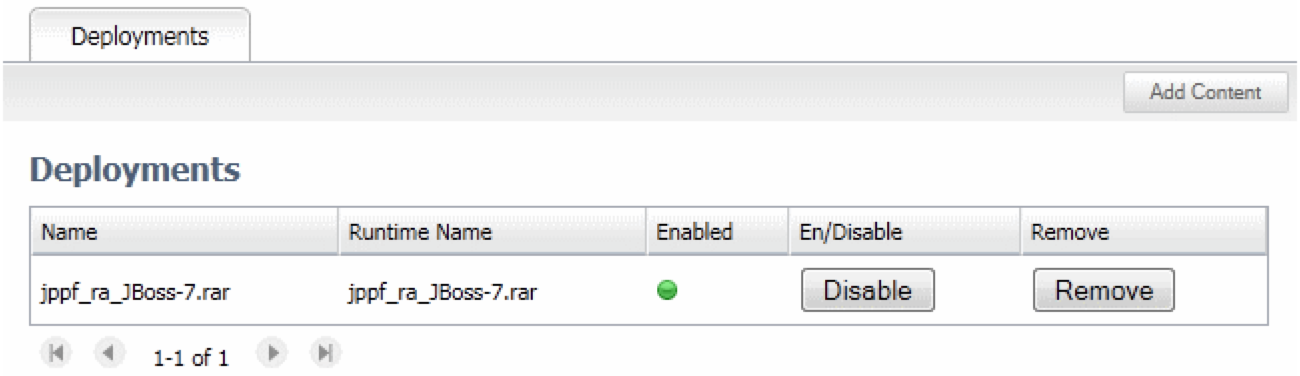


- click "next"

- accept the default names and click "Finish"
- back to the "Deployments" view, you should see the deployed resource adapter:

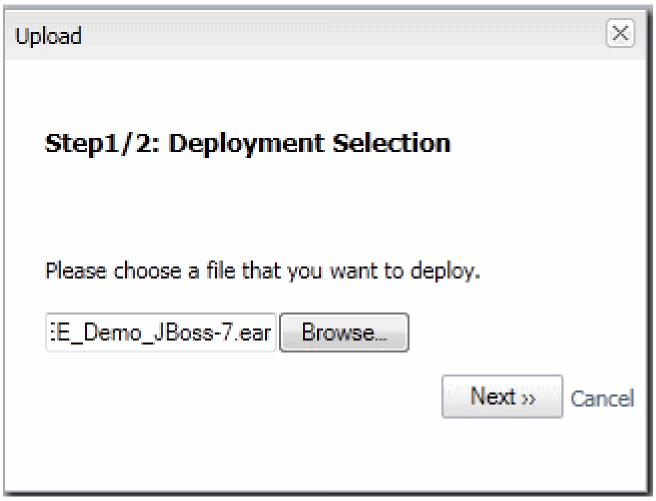


- click on the "Enable" button to start it:

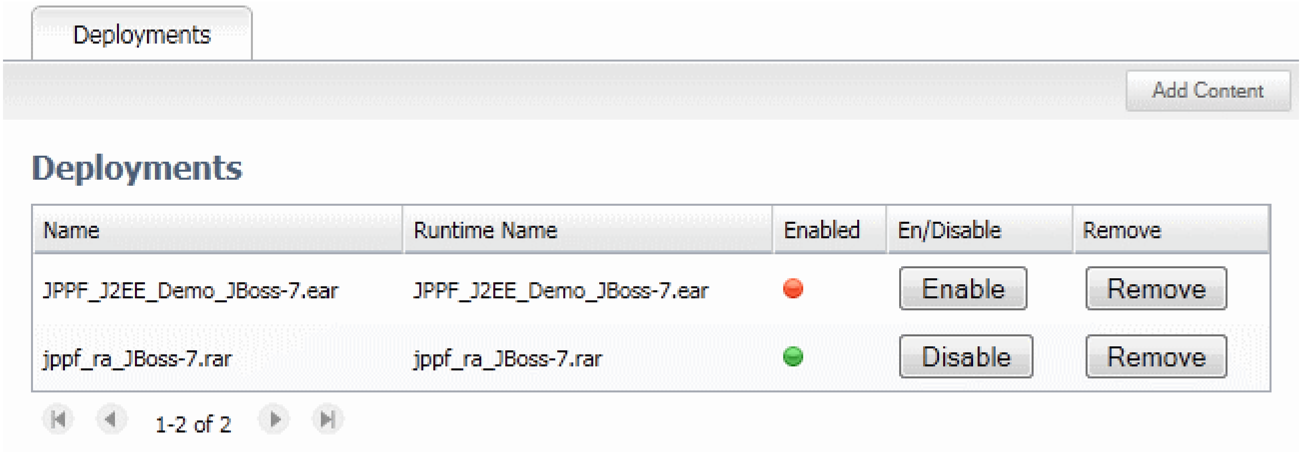


8.5.2.2 Deploying the demo application

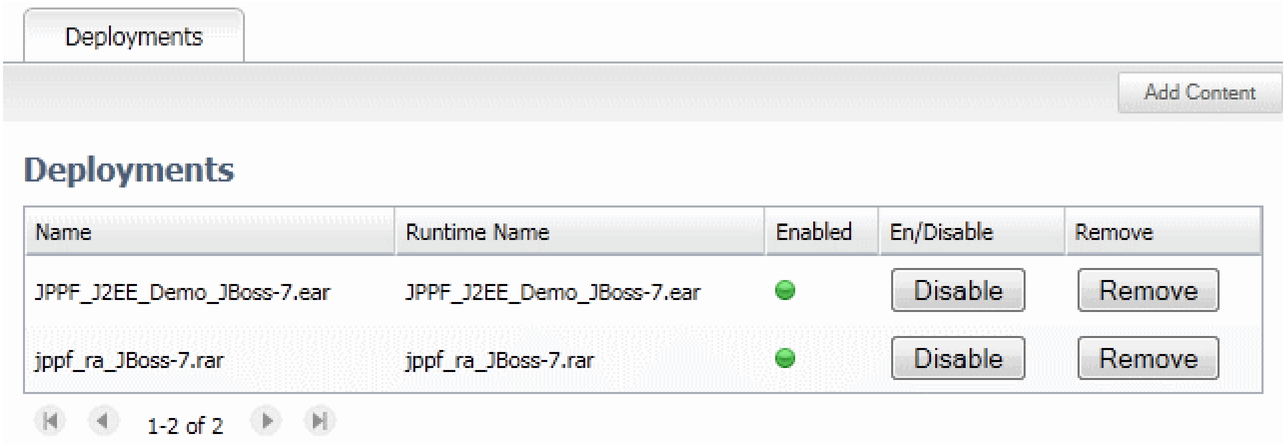
- In the "Deployment" view, click on "Add Content" and browse to "JPPF_J2EE_Demo_JBoss-7.ear"



- click on "Next"
- Accept the default names and click on "Finish"
- You should now see the demo web app in the list of deployments:



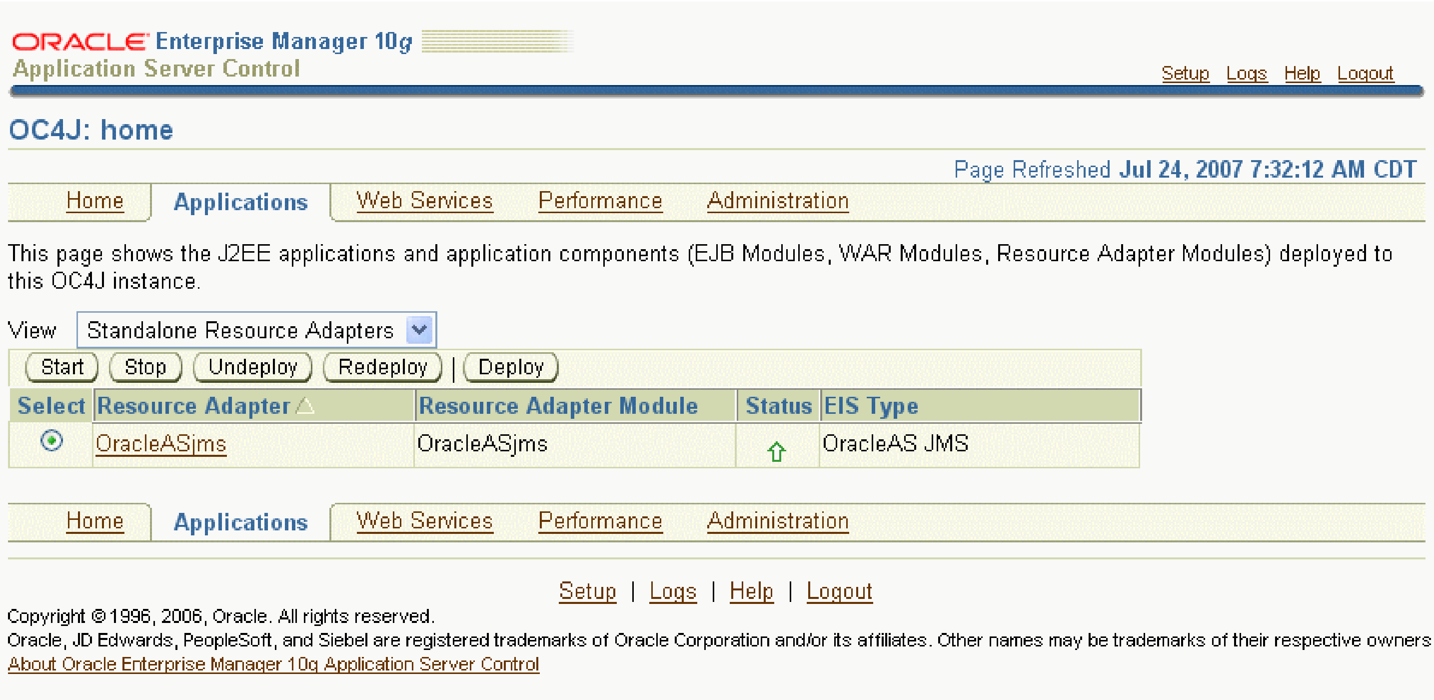
- click on the "Enable" button to start it:



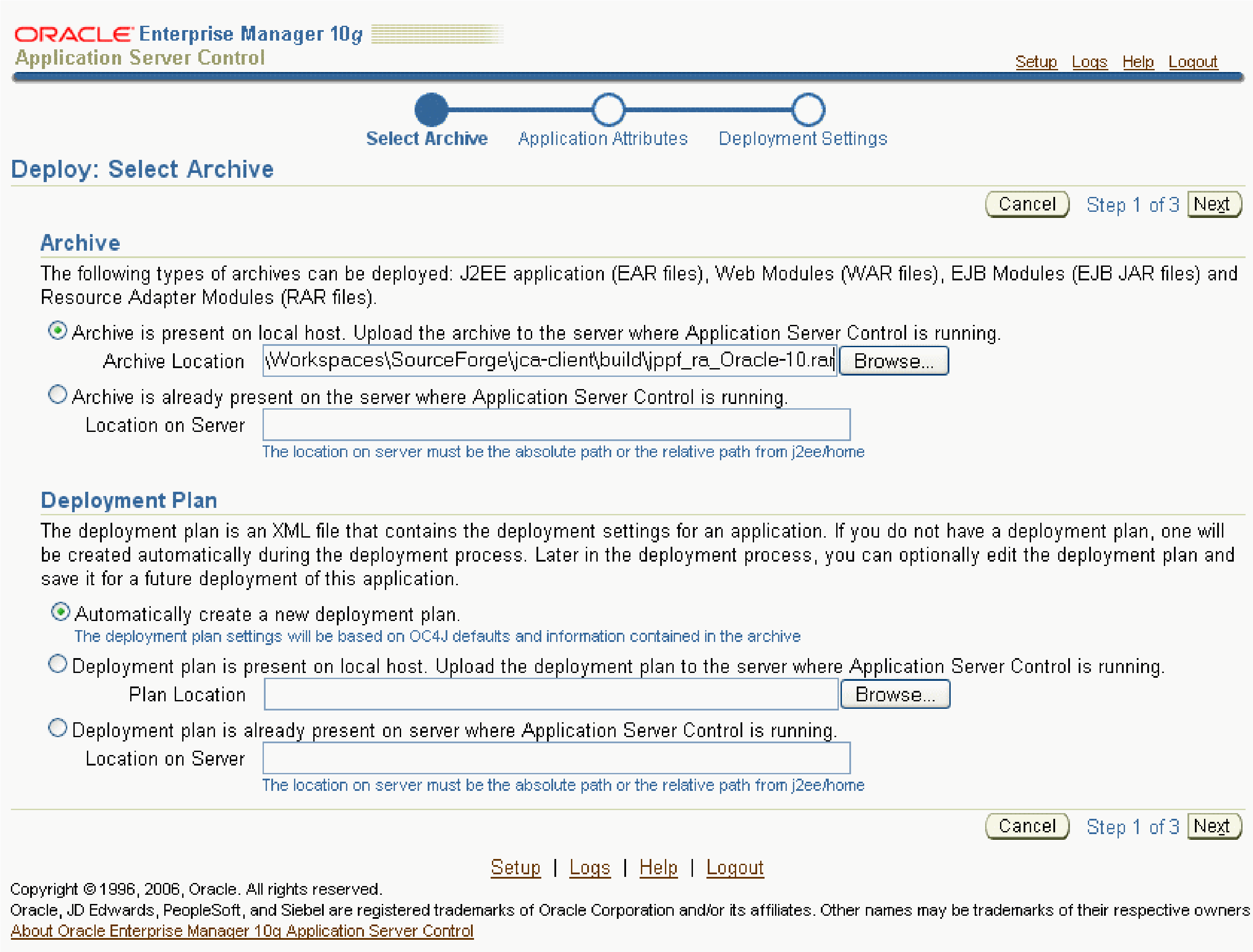
8.5.3 Deployment on Oracle OC4J 10.3.1.x

8.5.3.1 Deploying the resource adapter

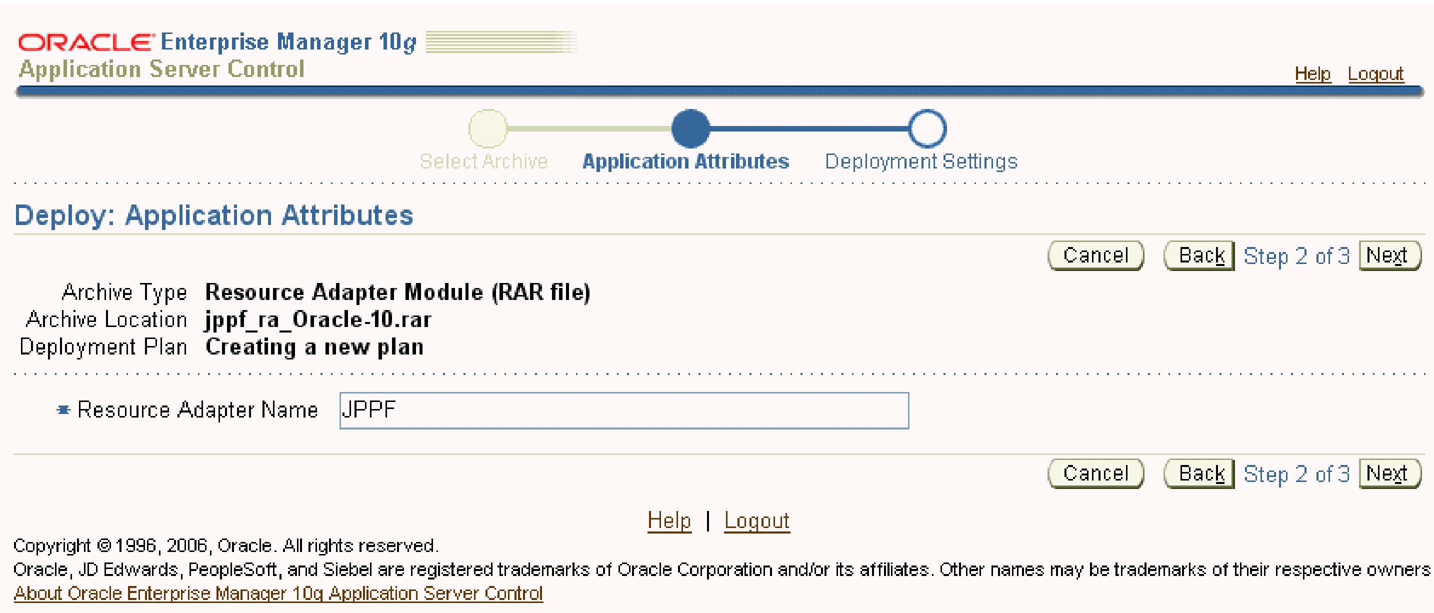
- in the OC4J administration console, go to "Applications"
- select "Standalone Resource Adapters" view



- click on "deploy"
- step 1: browse to the file "jppf_ra_Oracle.rar"



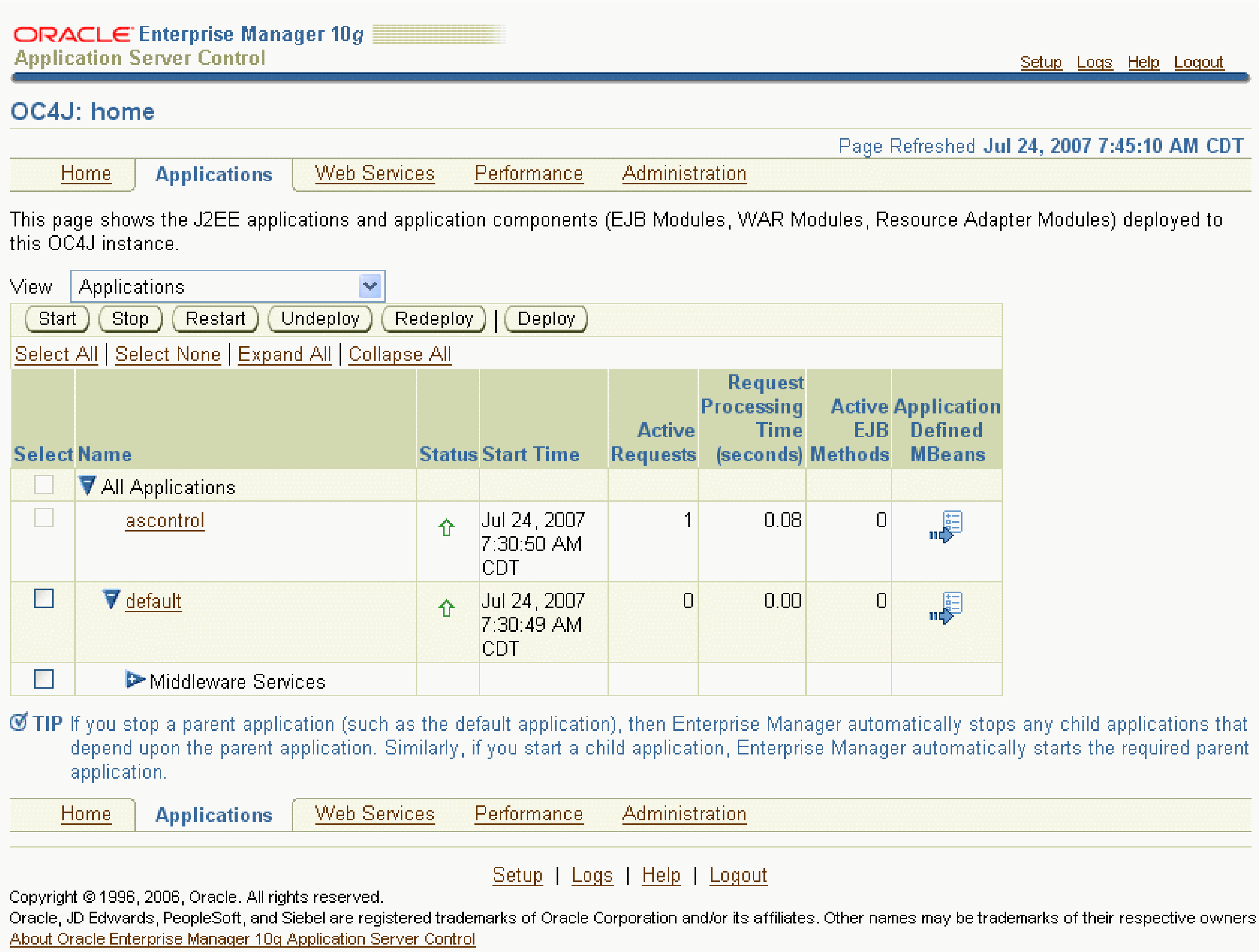
- click "next"
- step 2: enter "JPPF" as the Resource Adapter Name



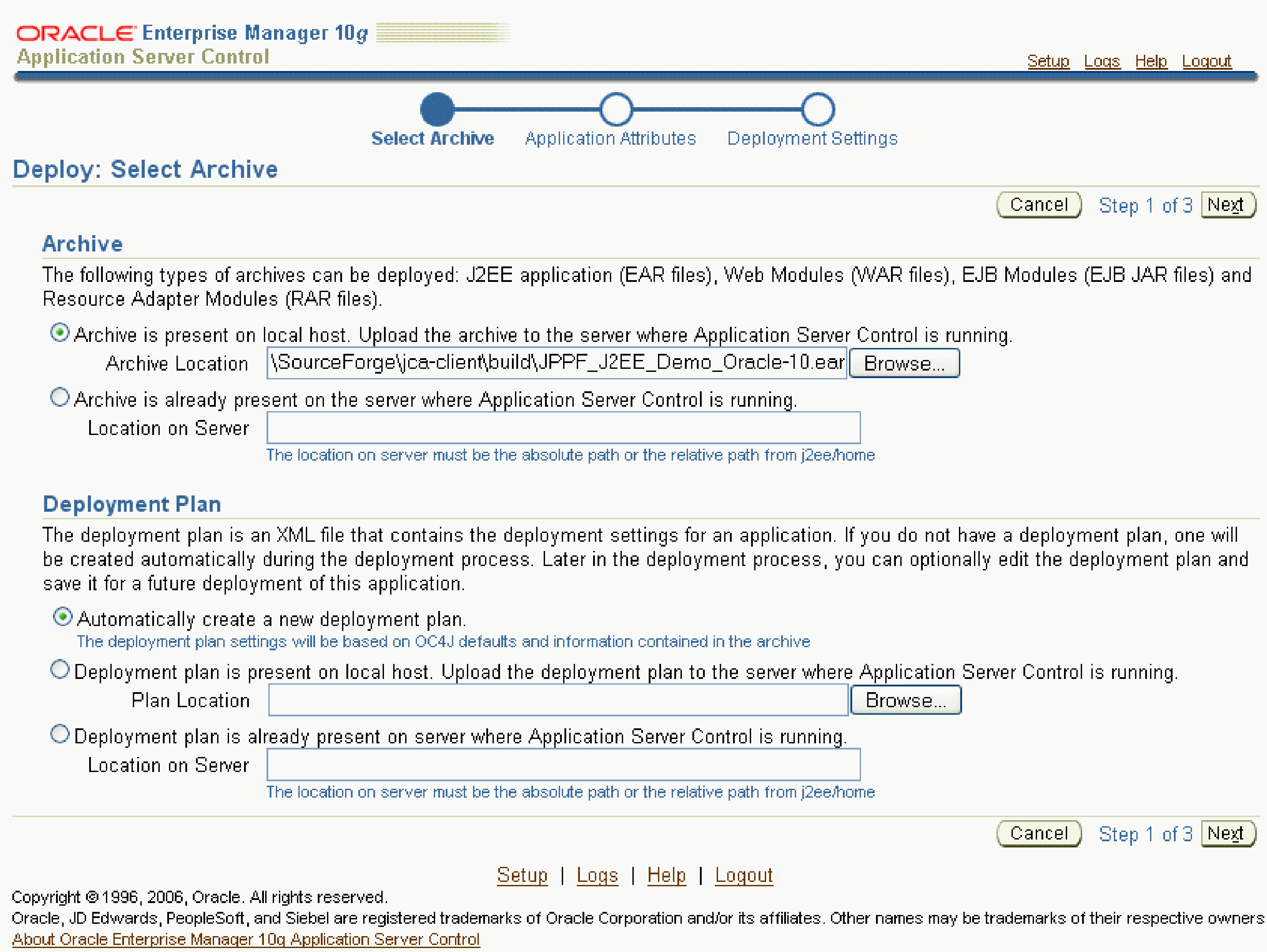
- click "next"
- step 3: click on "deploy"
- after deployment is complete, click on "return"
- restart the application server

8.5.3.2 Deploying the demo application

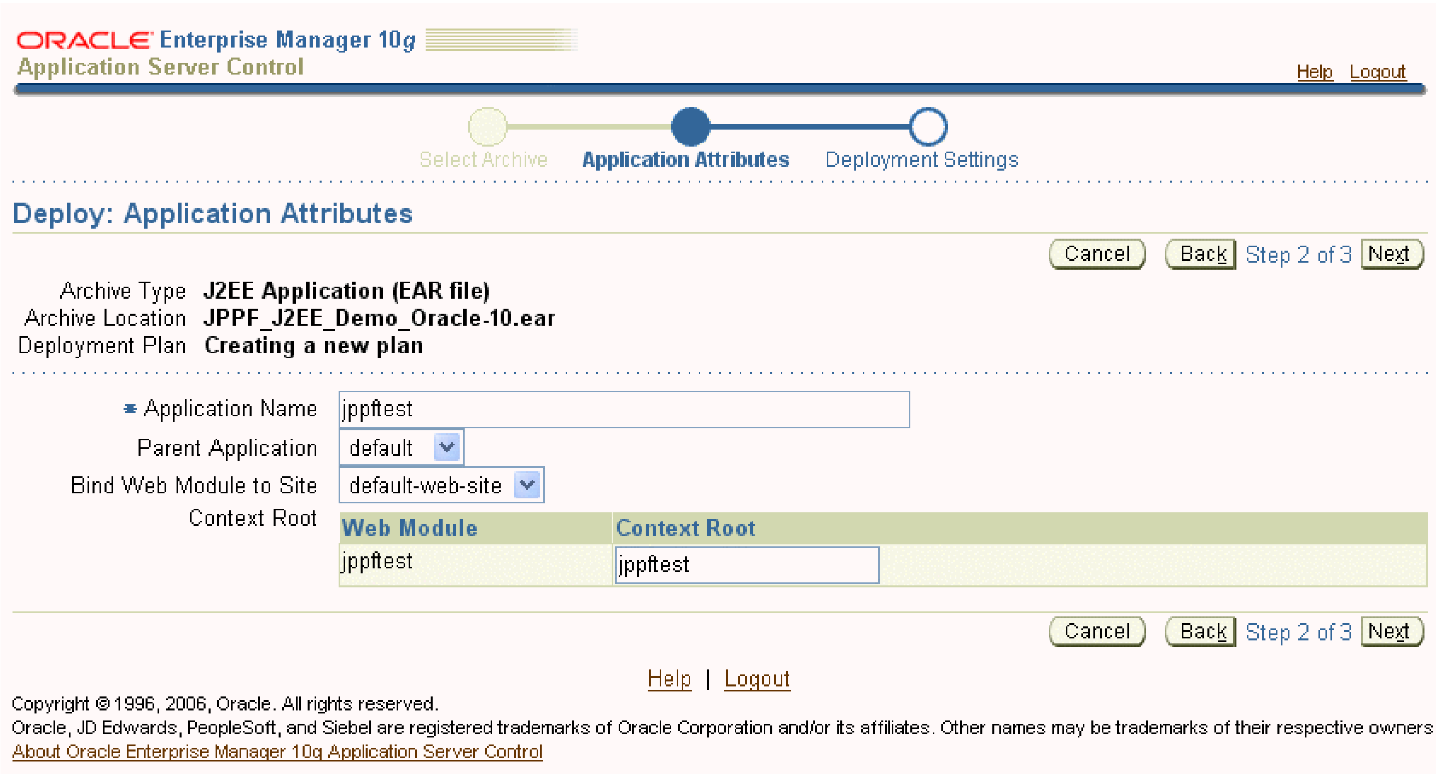
- in OC4J console, go to "Applications"
- select "Applications" view



- click on "deploy"
- step 1: browse to the file "JPPF_J2EE_Demo_Oracle.ear"



- click "next"
- step 2: enter "jppftest" as the Application Name



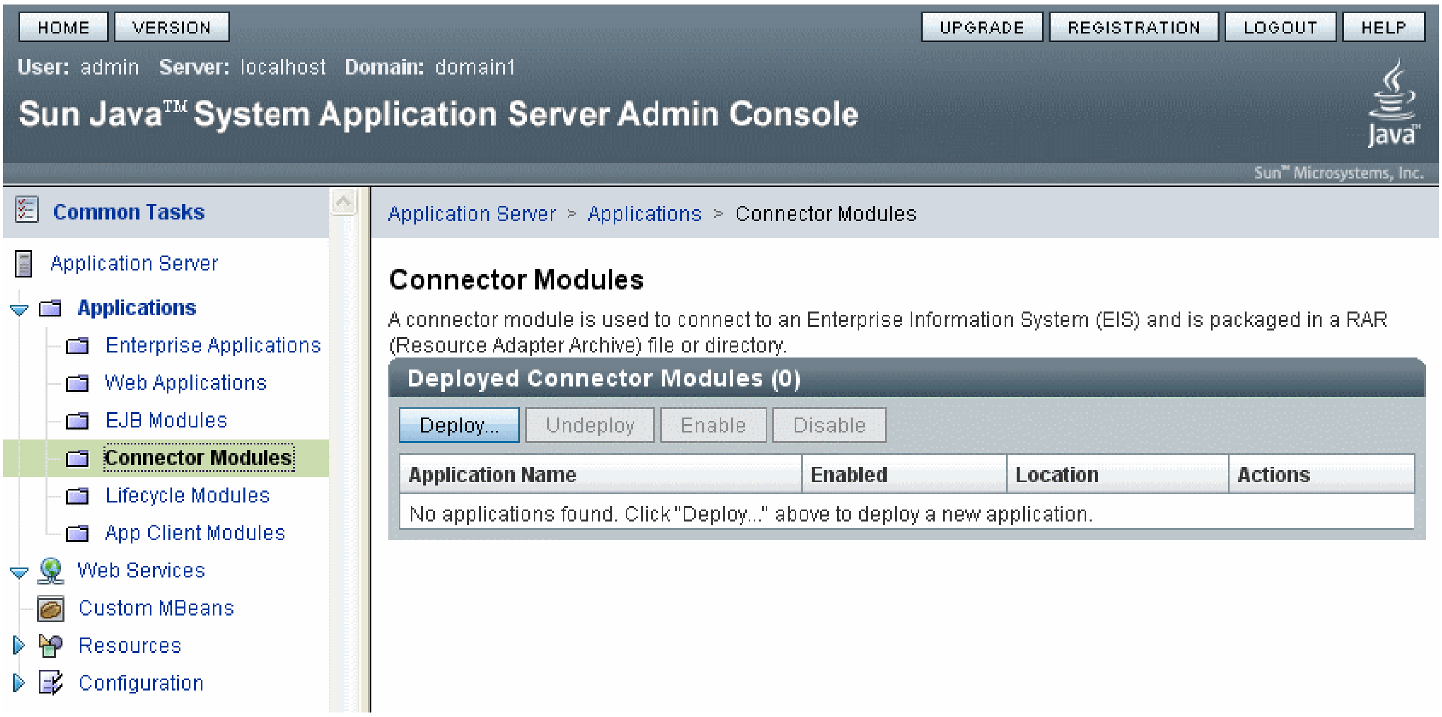
- click "next"
- step 3: click on "deploy"

- after deployment is complete, click on "return"
- restart the application server

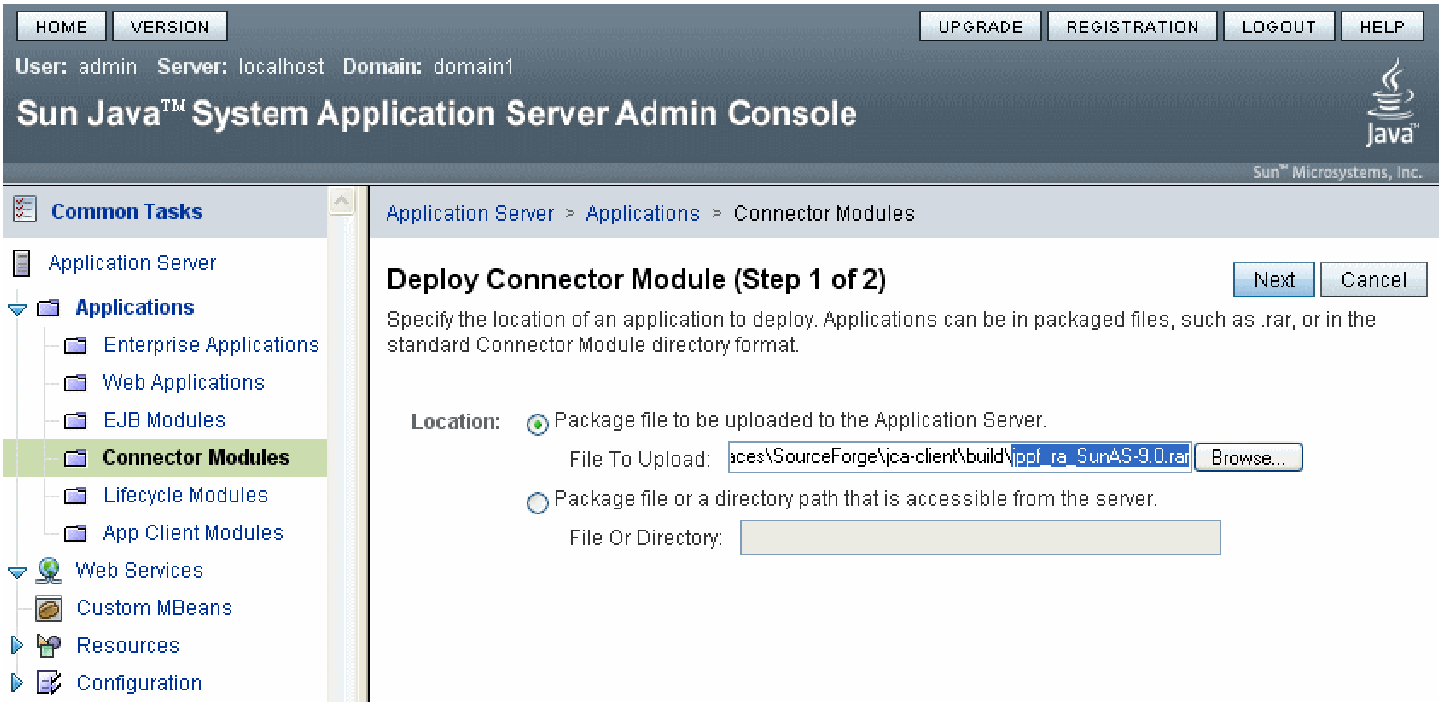
8.5.4 Deployment on SunAS / Glassfish

8.5.4.1 Deploying the Resource Adapter

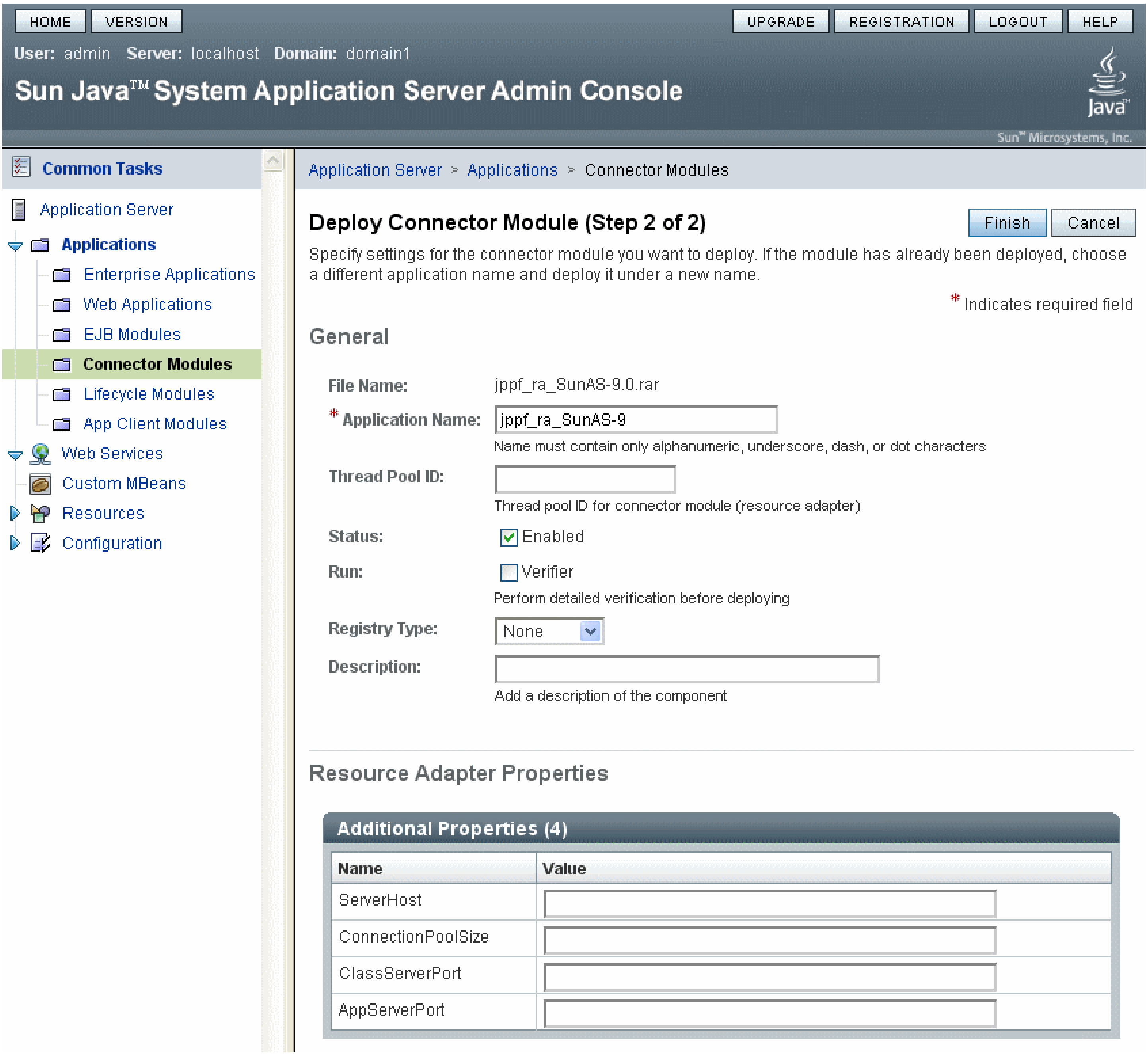
- in Sun AS console, go to "Applications > Connector modules"



- click on "Deploy"
- step 1: browse to the "jppf_ra_Glassfish.rar" file

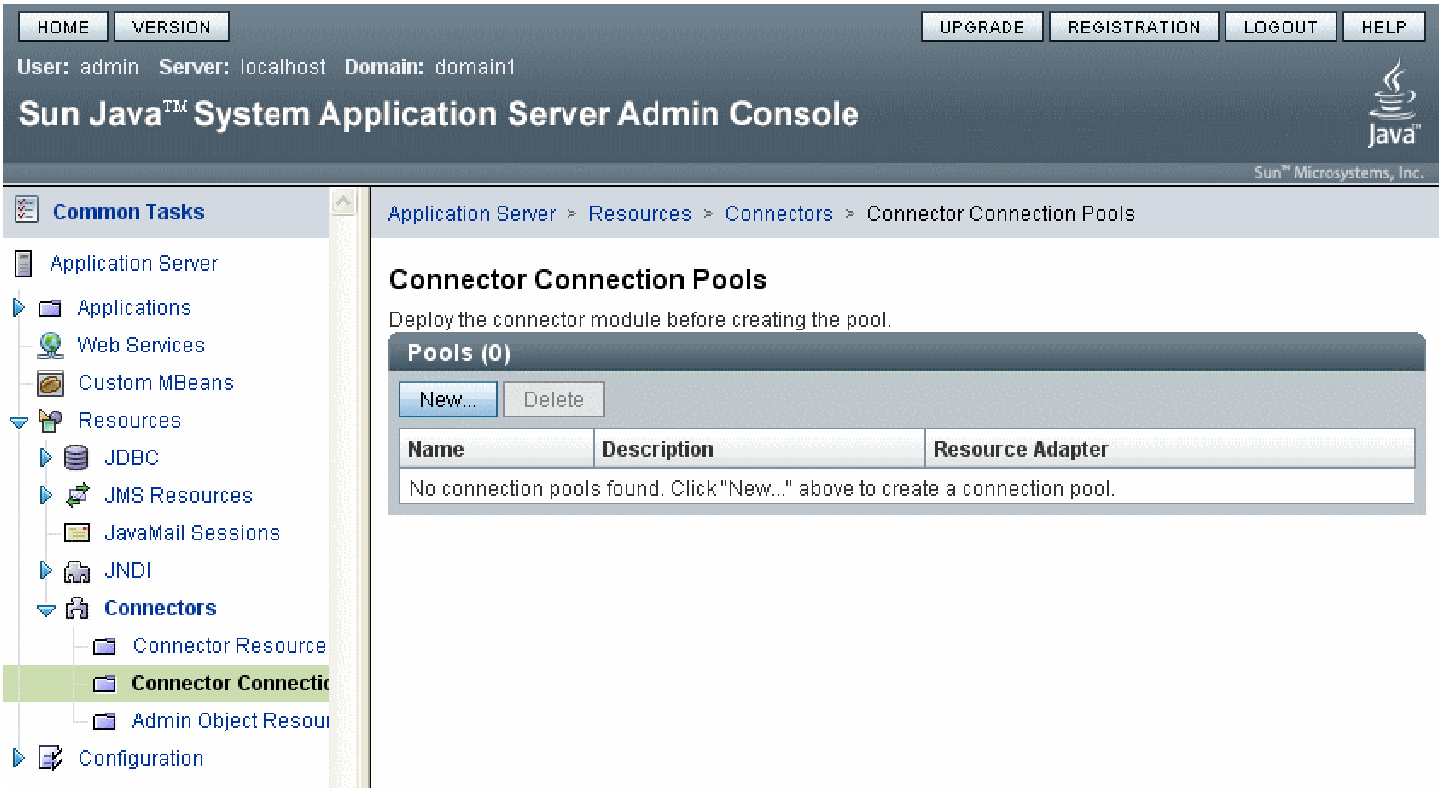


- click "next"
- step 2: leave all default settings and click "Finish"

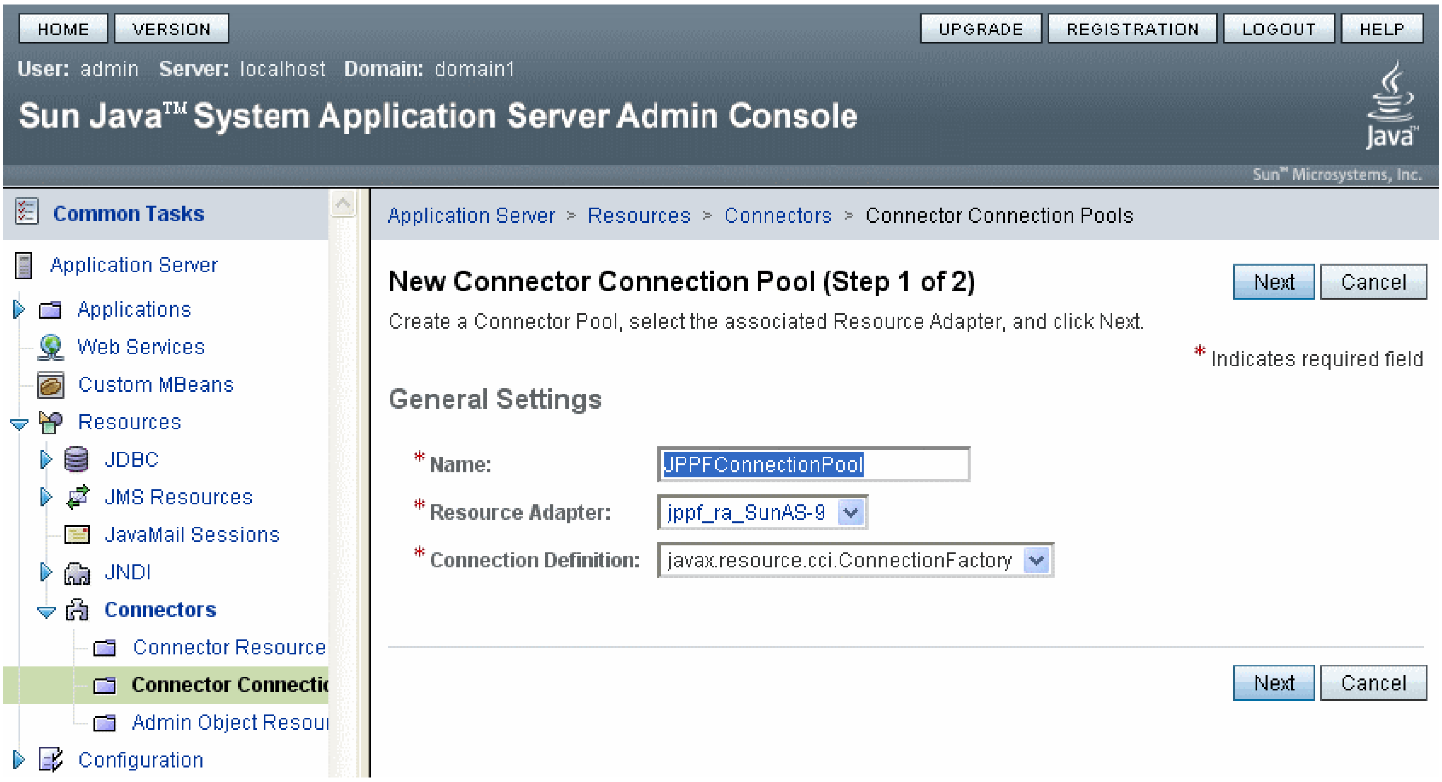

HOME
VERSION
UPGRADE
REGISTRATION
LOGOUT
HELP
User: admin Server: localhost Domain: domain1
Sun Java™ System Application Server Admin Console
Java
Sun™ Microsystems, Inc.
Common Tasks
Application Server
Applications
Enterprise Applications
Web Applications
EJB Modules
Connector Modules
Lifecycle Modules
App Client Modules
Web Services
Custom MBeans
Resources
Configuration
Application Server > Applications > Connector Modules
Deploy Connector Module (Step 2 of 2)
Finish
Cancel
Specify settings for the connector module you want to deploy. If the module has already been deployed, choose a different application name and deploy it under a new name.
* Indicates required field
General
File Name: jppf_ra_SunAS-9.0.rar
* Application Name: jppf_ra_SunAS-9
Name must contain only alphanumeric, underscore, dash, or dot characters
Thread Pool ID:
Thread pool ID for connector module (resource adapter)
Status: Enabled
Run: Verifier
Perform detailed verification before deploying
Registry Type: None
Description:
Add a description of the component
Resource Adapter Properties
Additional Properties (4)
Name
Value
ServerHost
ConnectionPoolSize
ClassServerPort
AppServerPort

8.5.4.2 Creating a connector connection pool

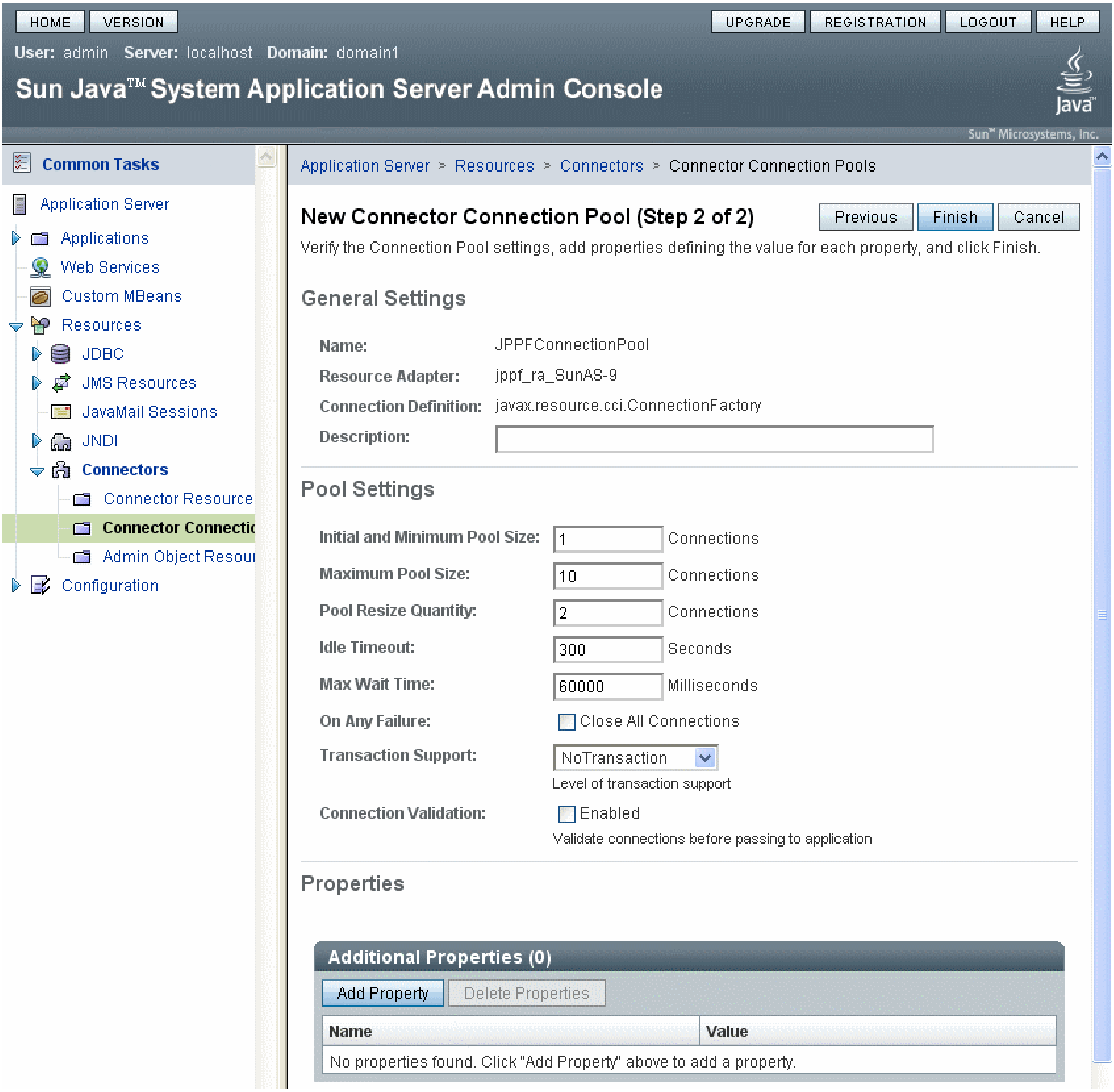
- in the console tree on the left, go to "Resources > Connectors > Connector Connection Pools"



- click on "New"
- step 1: enter "JPPFConnectionPool" as the connection pool name and select "jppf_ra_Glassfish" for the resource adapter



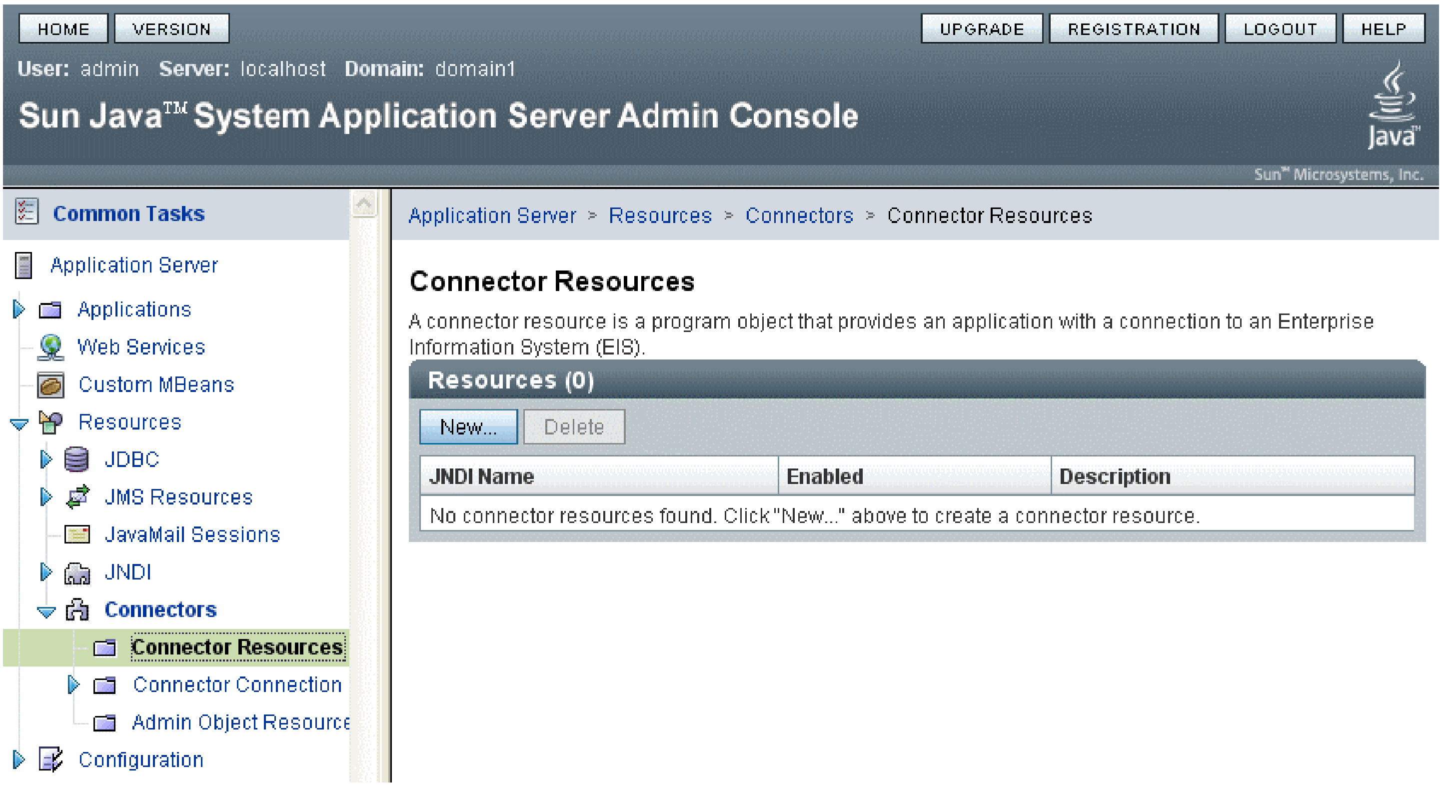
- click "next"
- step 2: set the pool parameters, select "NoTransaction" for transaction support



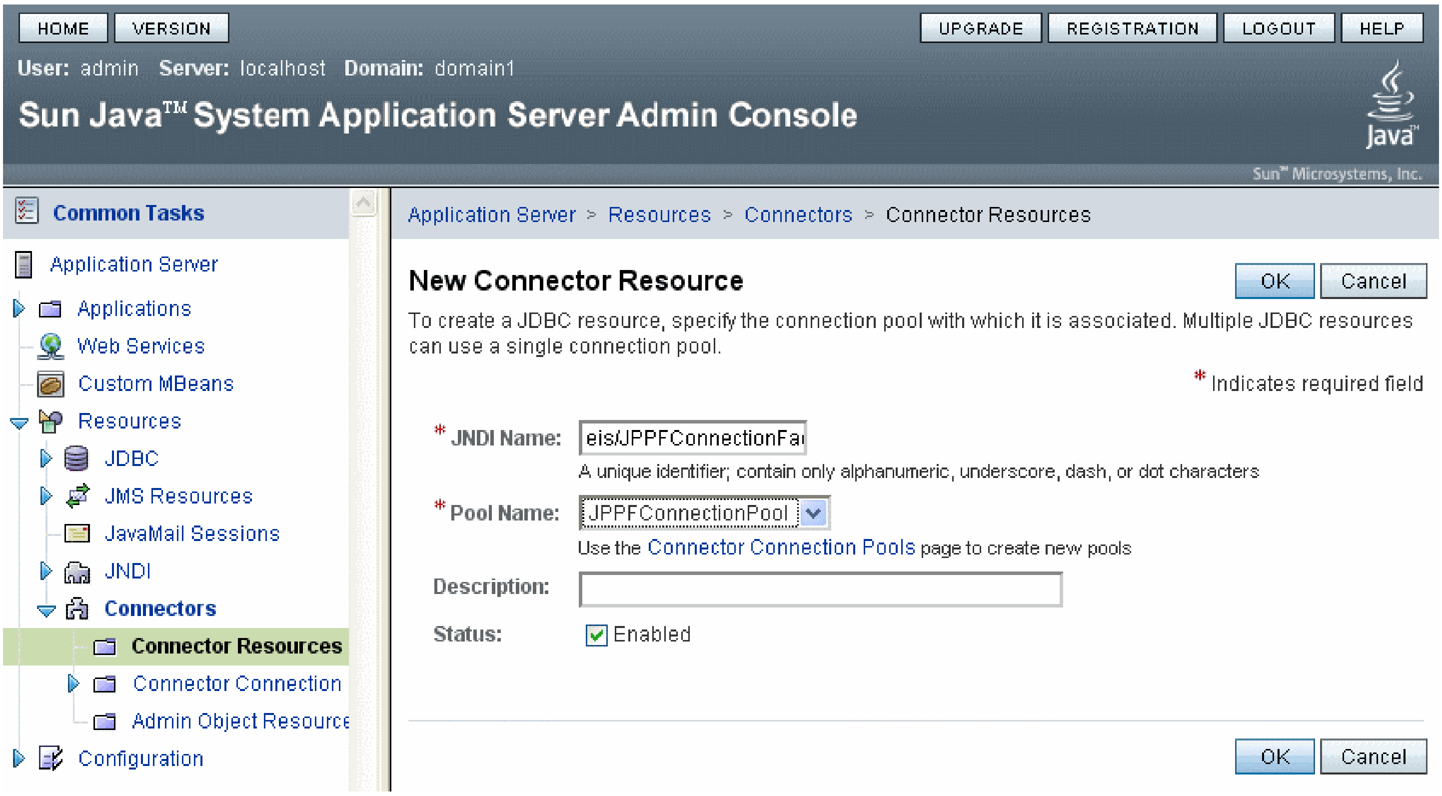
- click "Finish"

8.5.4.3 Creating a connection factory

- in the console tree on the left, go to "Resources > Connectors > Connector Resources"



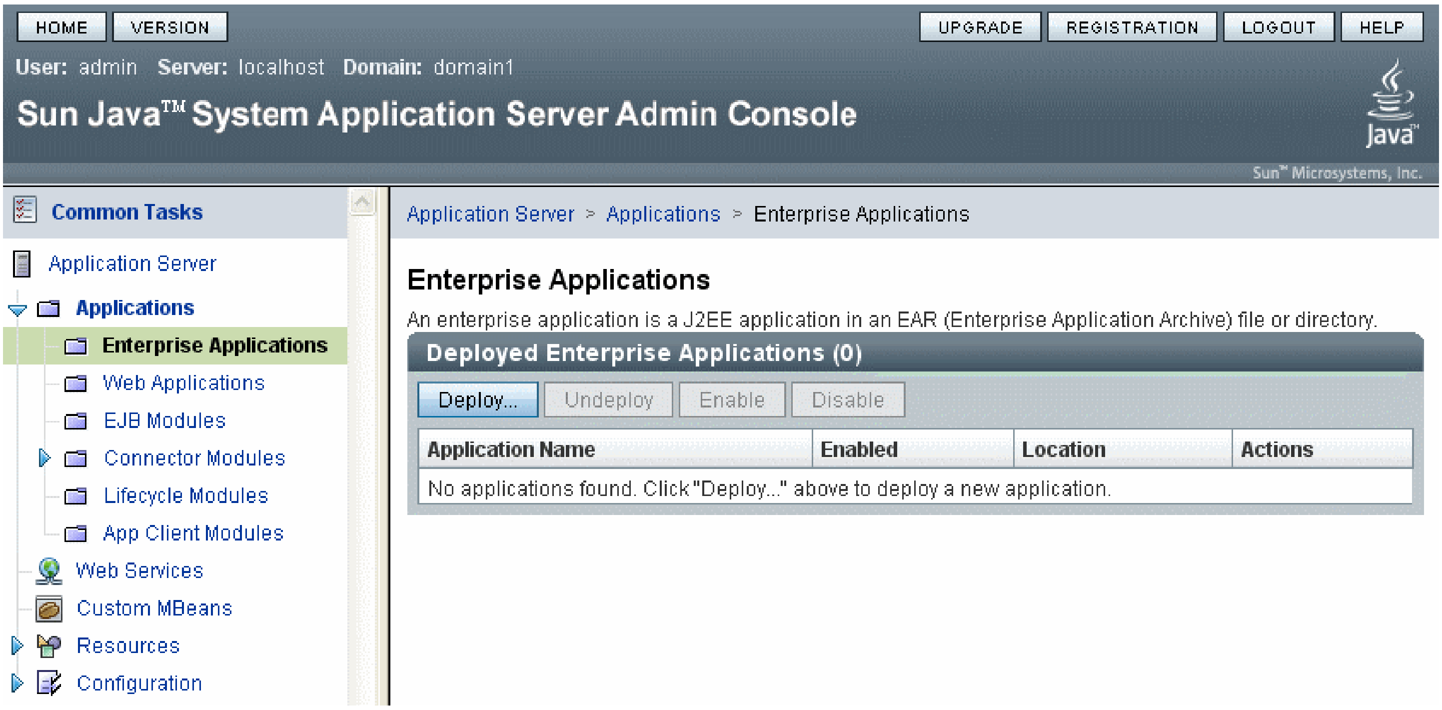
- click on "New"
- for the jndi name, enter "eis/JPPFConnectionFactory"
- for the connection pool, select "JPPFConnectionPool"



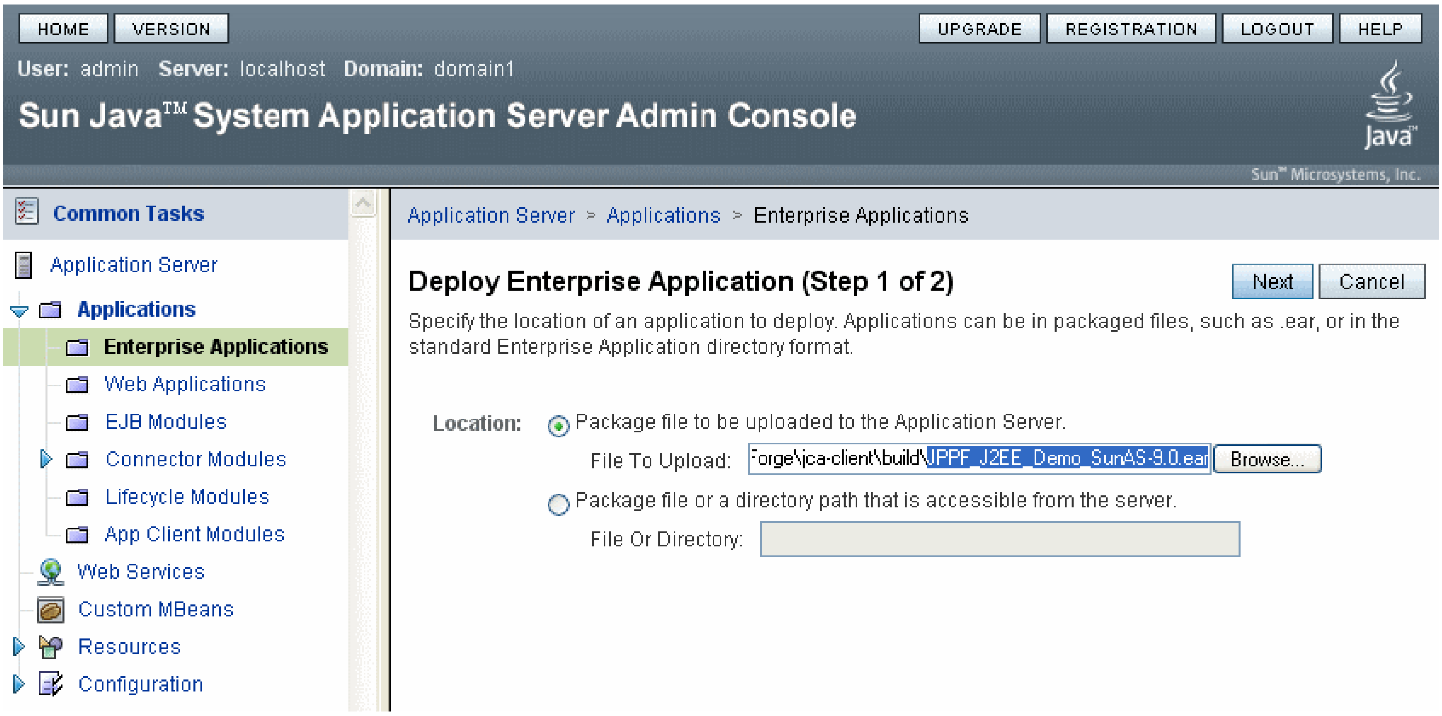
- click "OK"
- restart the application server

8.5.4.4 Deploying the demo application

- in SunAS console, go to "Applications > Enterprise Applications"



- click on "Deploy"
- step 1: browse to the file "JPPF_J2EE_Demo_Glassfish.ear"



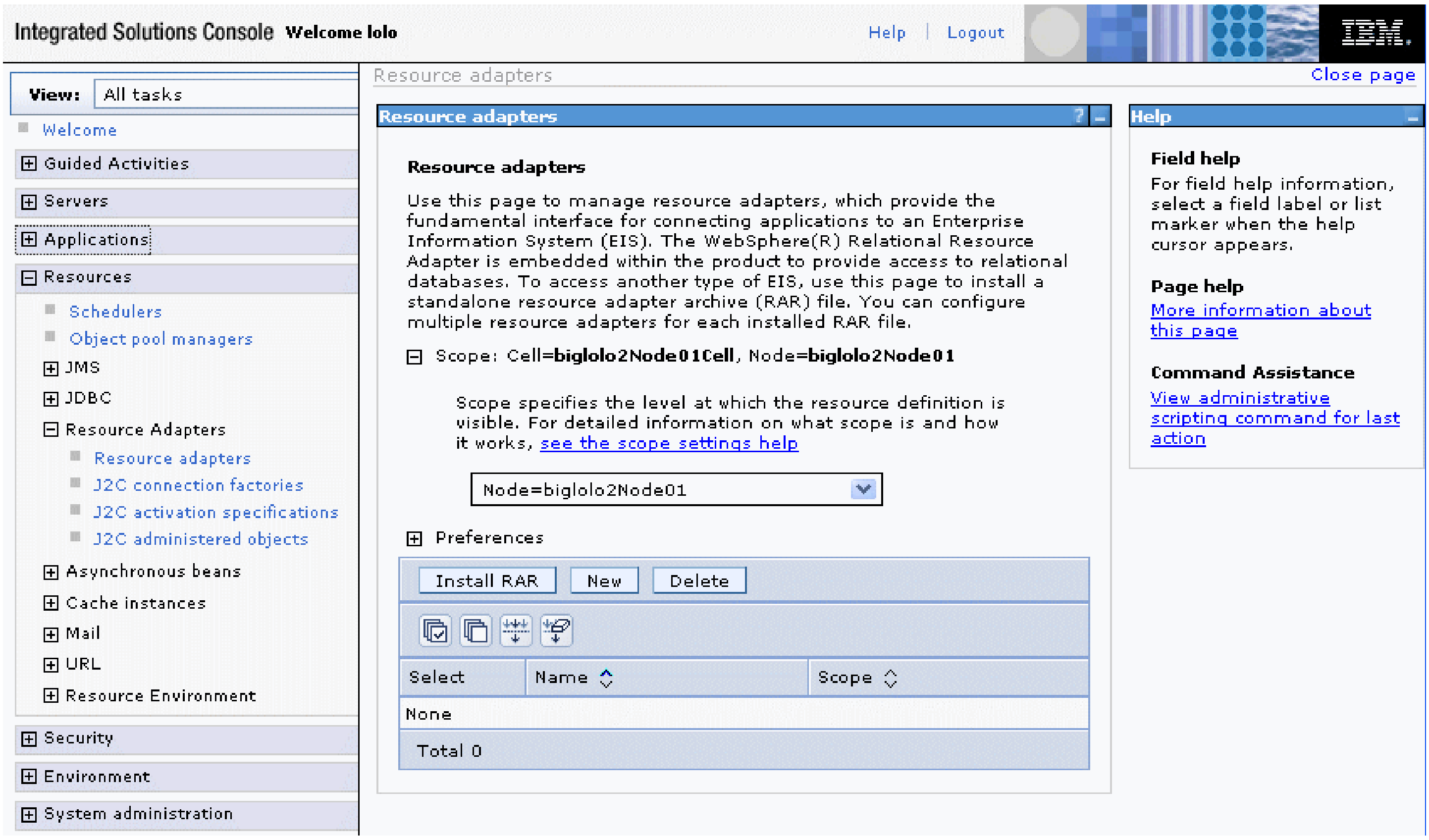
- click "next"
- step 2: leave all default values

- click "Finish"
- restart the application server

8.5.5 Deployment on Websphere

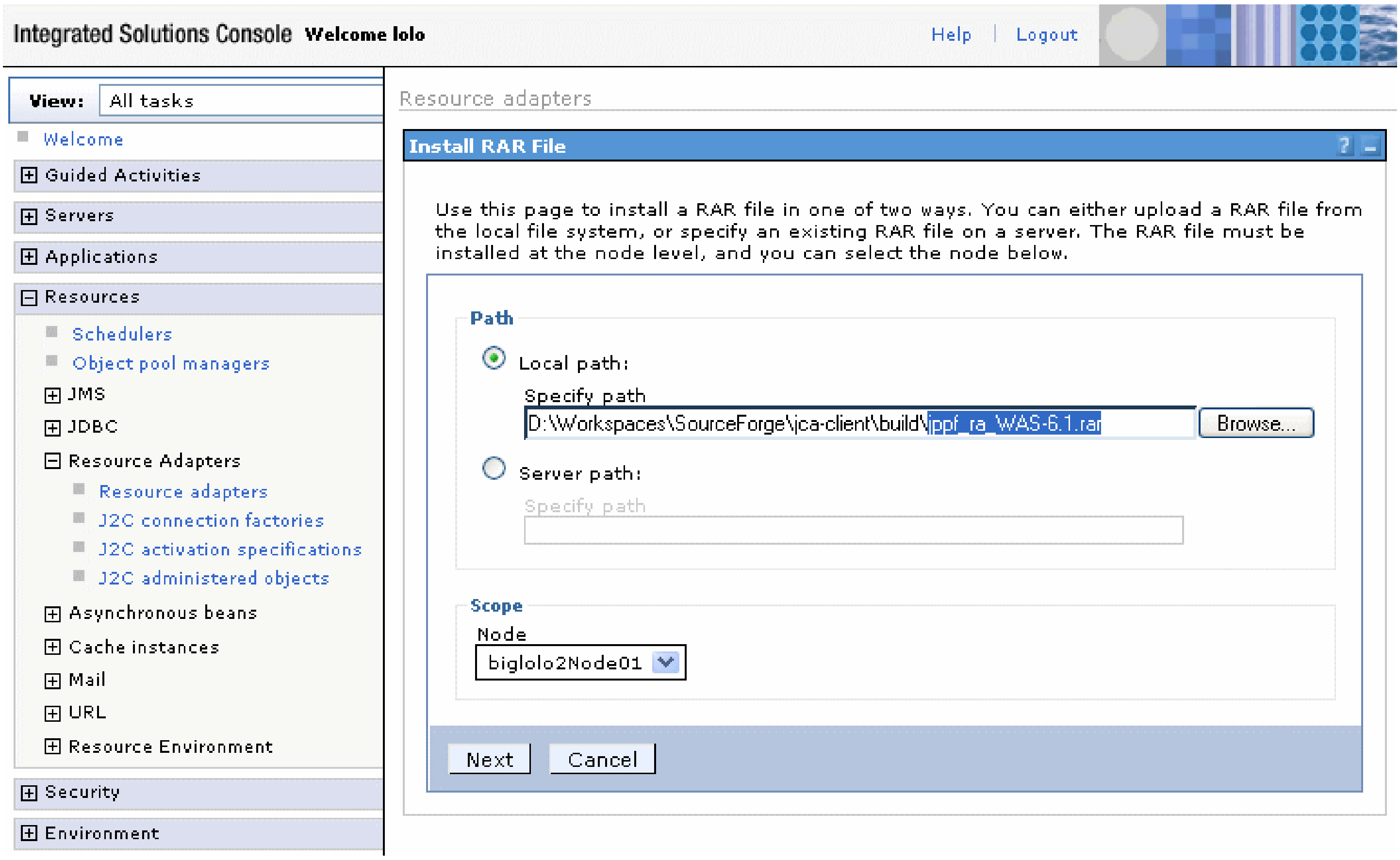
8.5.5.1 Deploying the Resource Adapter

- in WAS console, go to "Resources > Resource Adapters > Resource adapters"
- select scope = "Node"

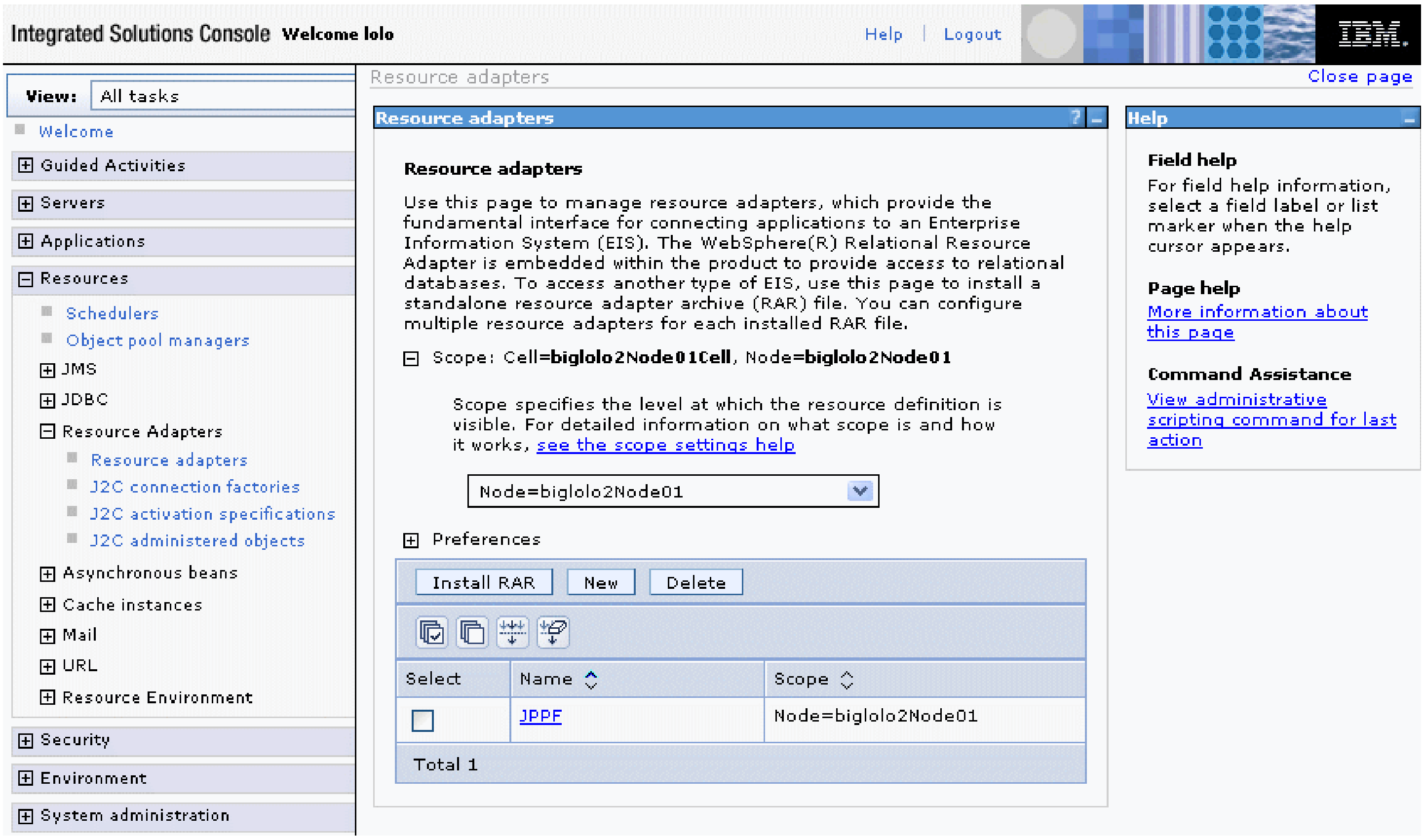


- click "Install RAR"

- in the "Install RAR file" page, browse to the "jppf_ra_Websphere.rar" file

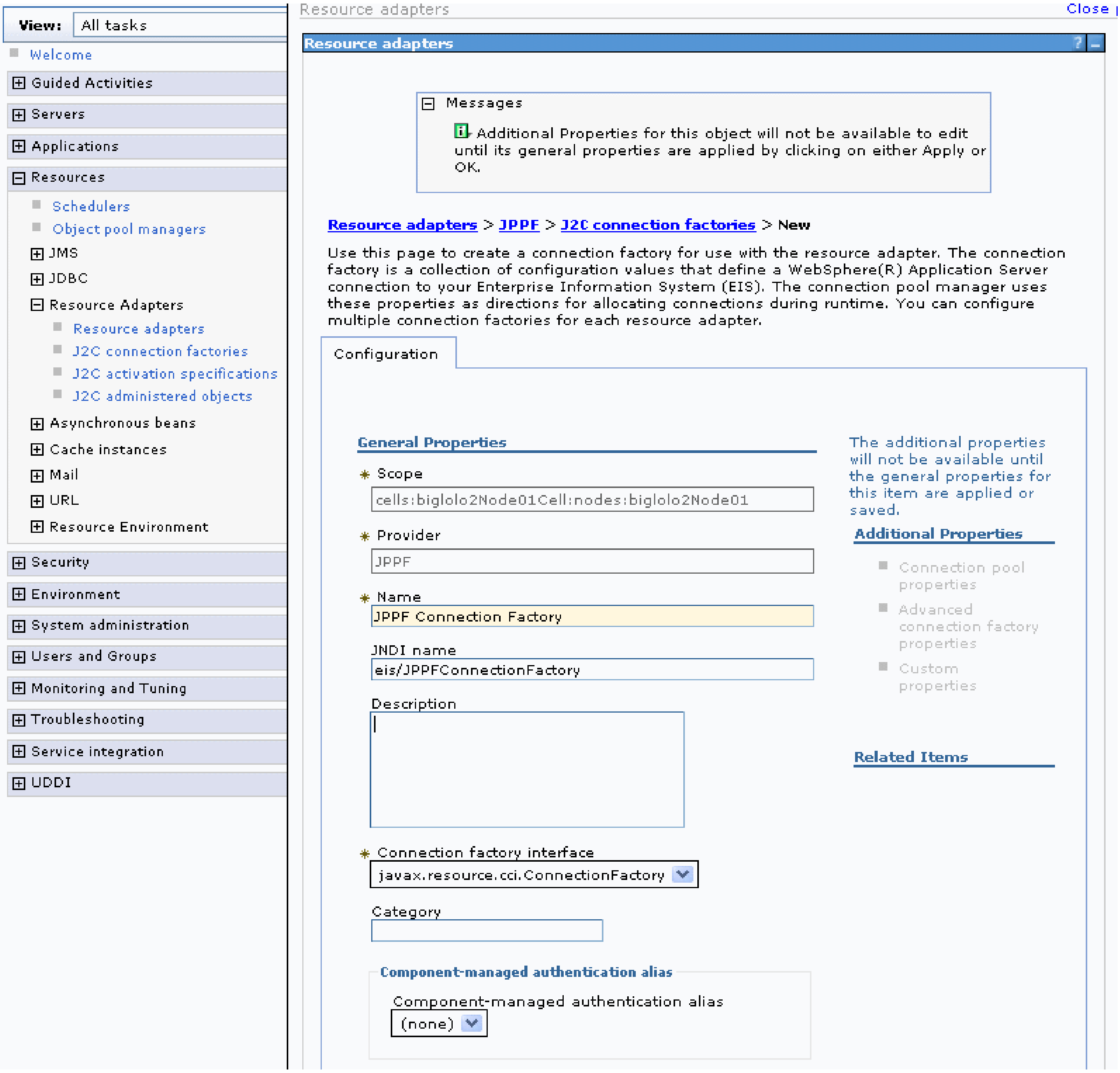


- click "Next"
- click "OK"
- click "Save directly to the master configuration"



8.5.5.2 Creating a connection factory

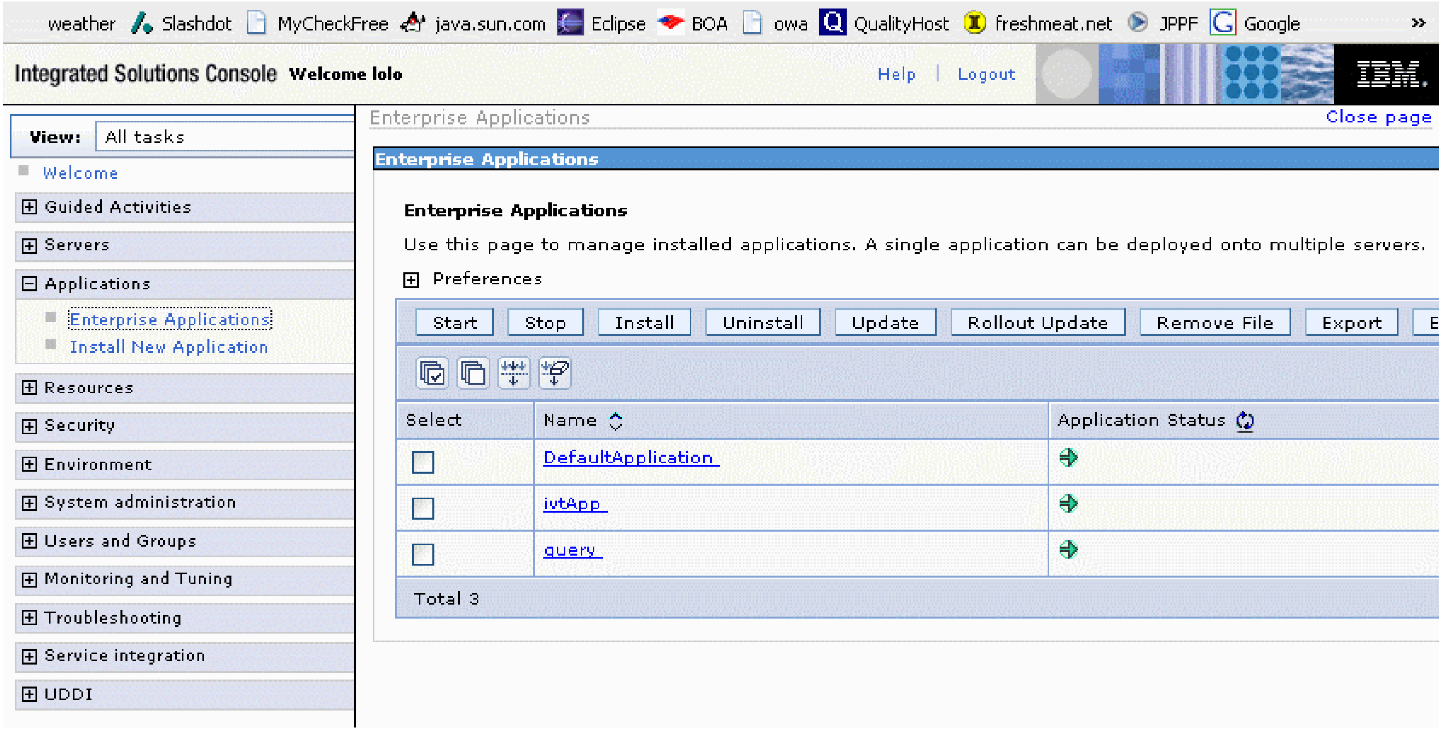
- in the list of resource adapters, click on "JPPF"
- in "Additional Properties", click on "J2C connection factories"
- click on "New"
- enter "JPPF Connection Factory" for the name and "eis/JPPFConnectionFactory" for the JNDI name, leave all other parameters as they are



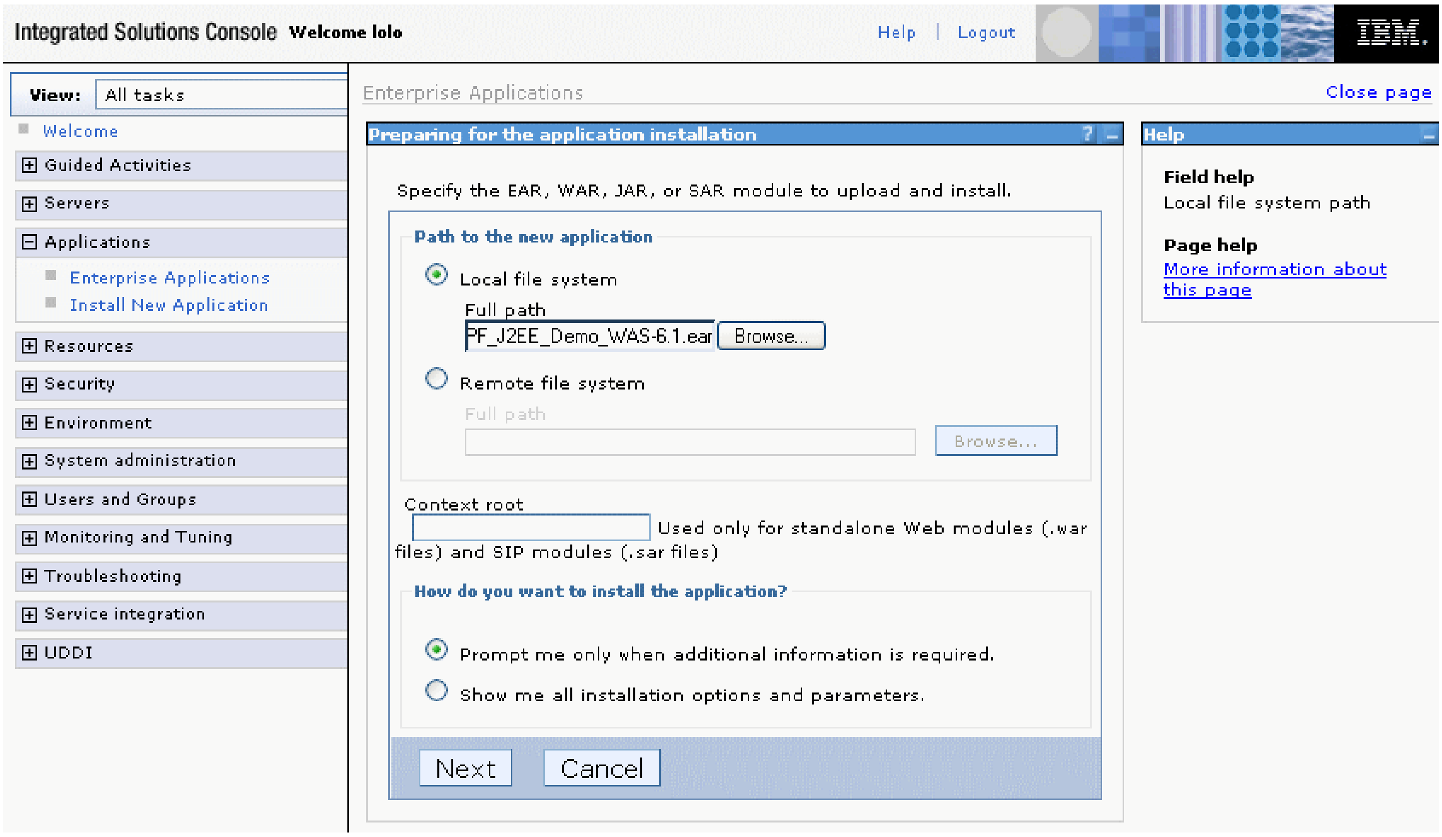
- click "OK"
- click "Save directly to the master configuration"
- Restart the application server

8.5.5.3 Deploying the demo application

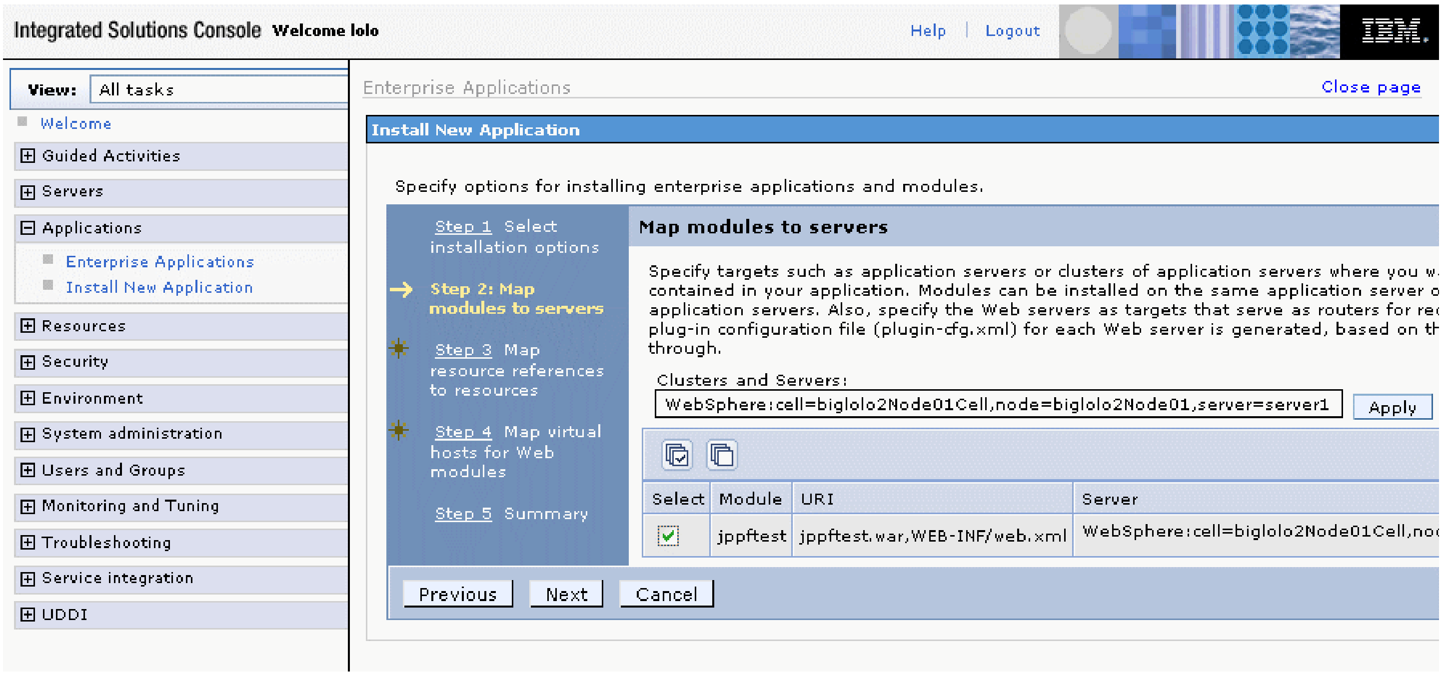
- in Websphere console, go to "Applications > Enterprise Applications"



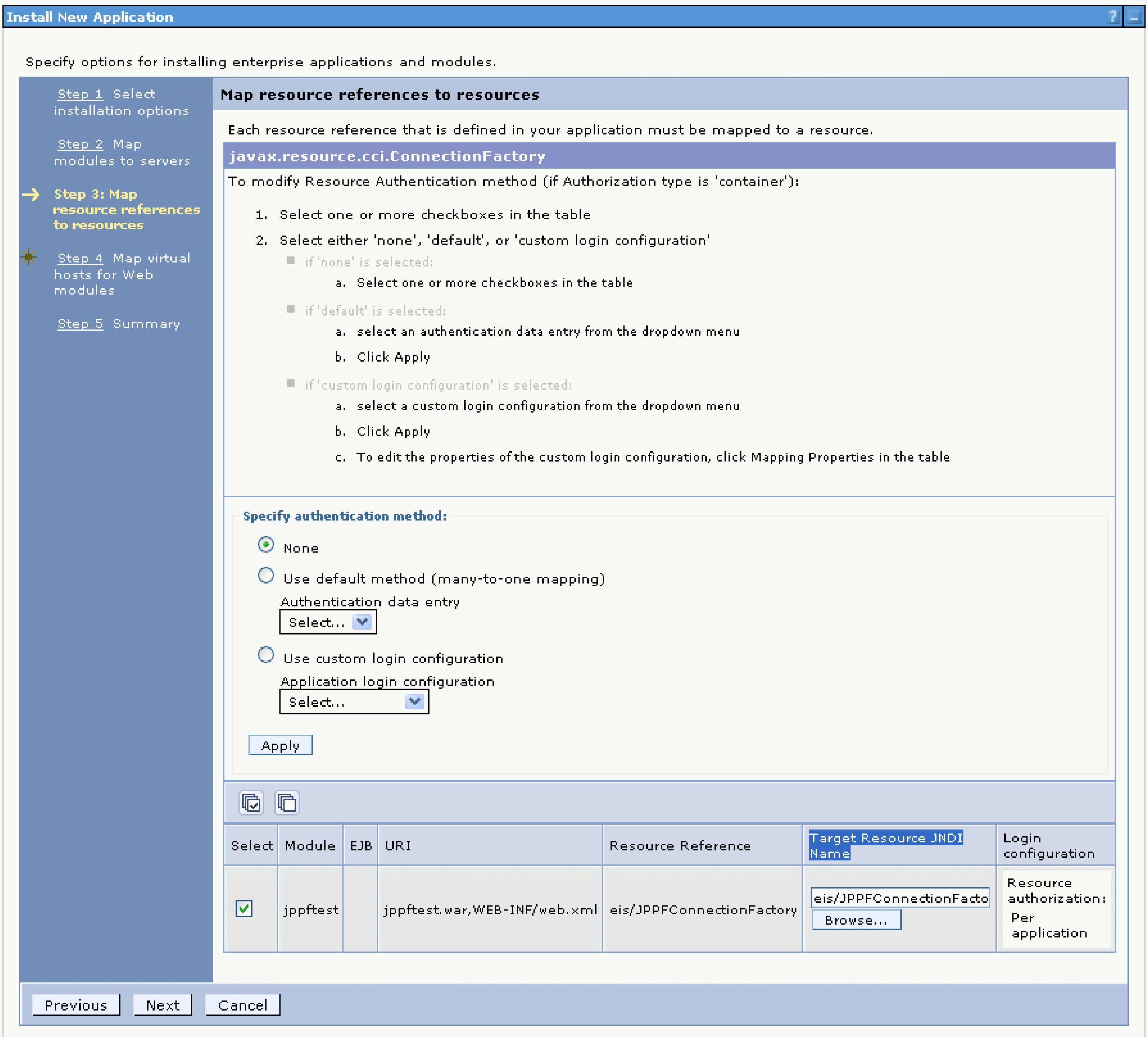
- click on "Install"
- browse to the file "JPPF_J2EE_Demo_Websphere.ear"
- select "Prompt me only when additional information is required".



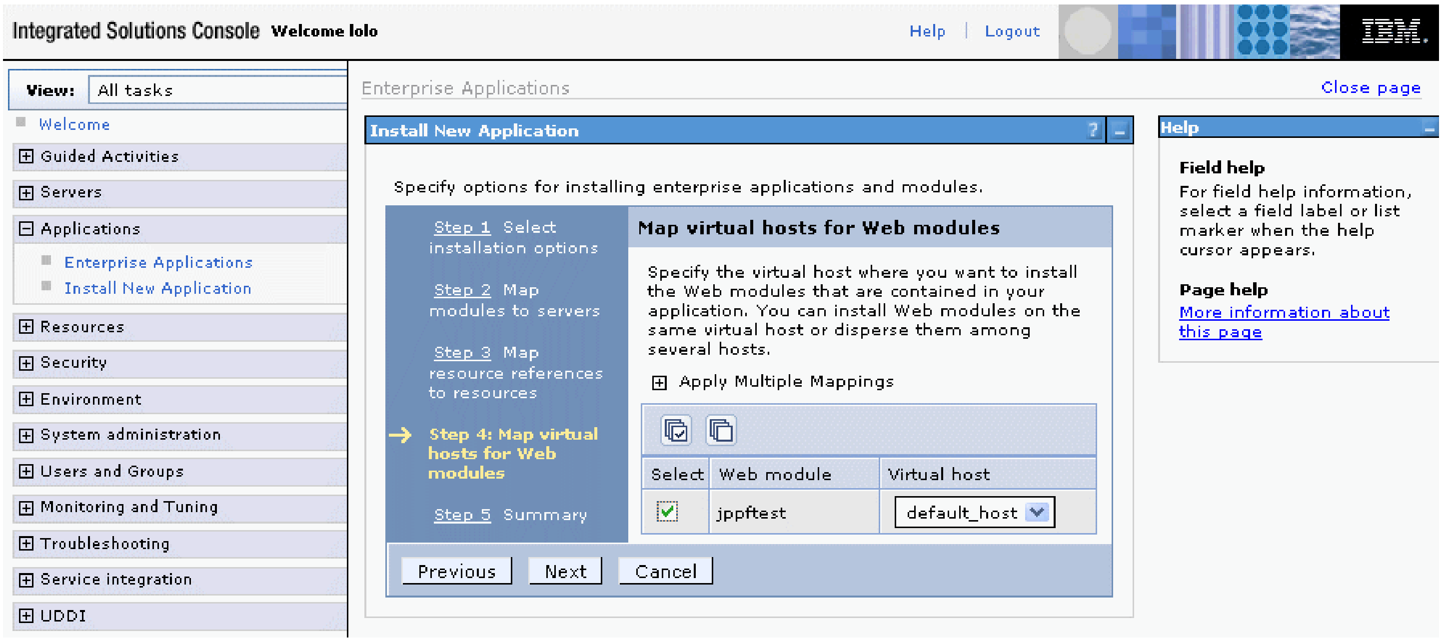
- click "Next"
- step 1: click "Next"
- step 2: check module "jppftest"



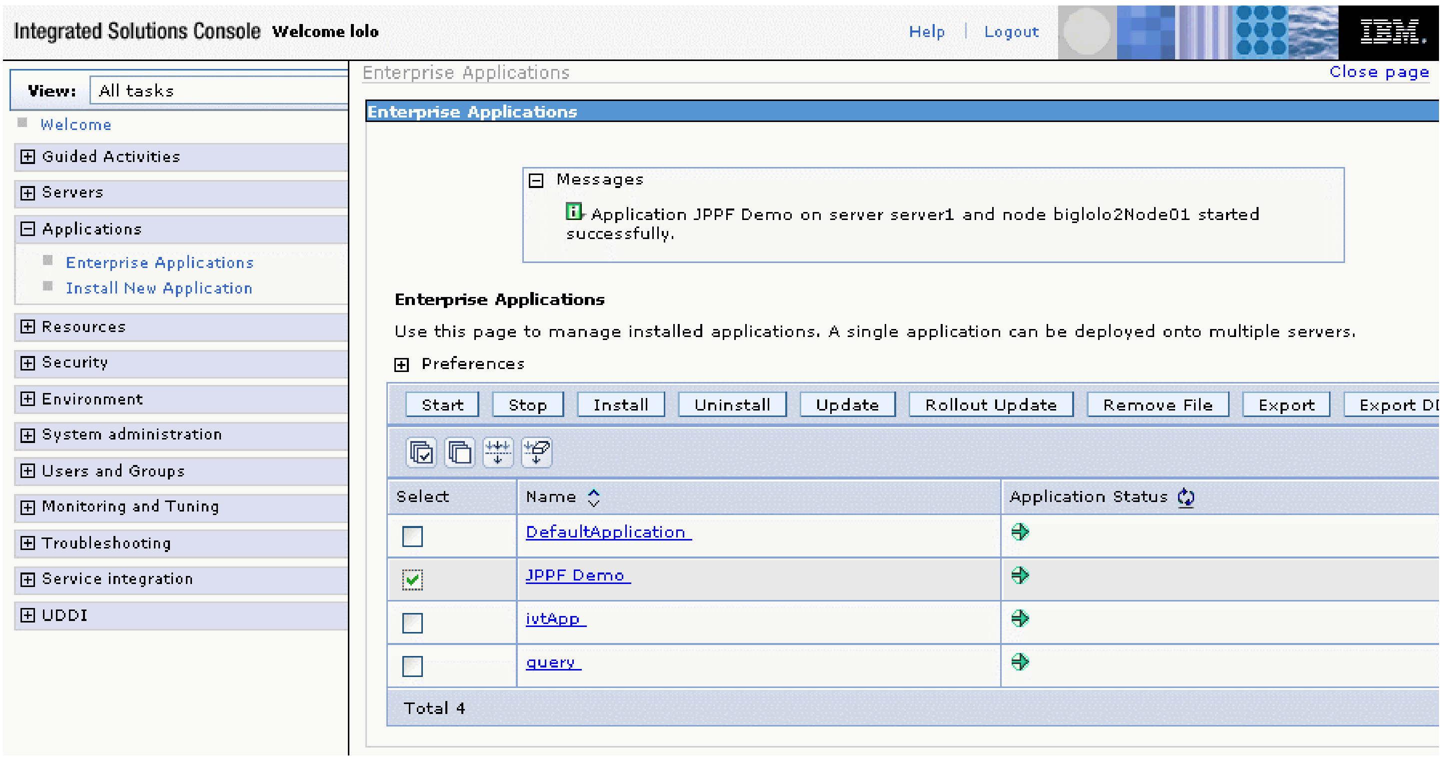
- click "Next"
- step 3: select "None" for the authentication method
- check the "jppftest" module and enter "eis/JPPFConnectionFactory" as Target Resource JNDI Name



- click "Next"
- step 4: check the "jppftest" module and keep "default_host" as the Virtual host



- click "Next"
- step 5: click "Finish"
- click "Save directly to the master configuration"
- in the list of enterprise applications, check "JPPF Demo"



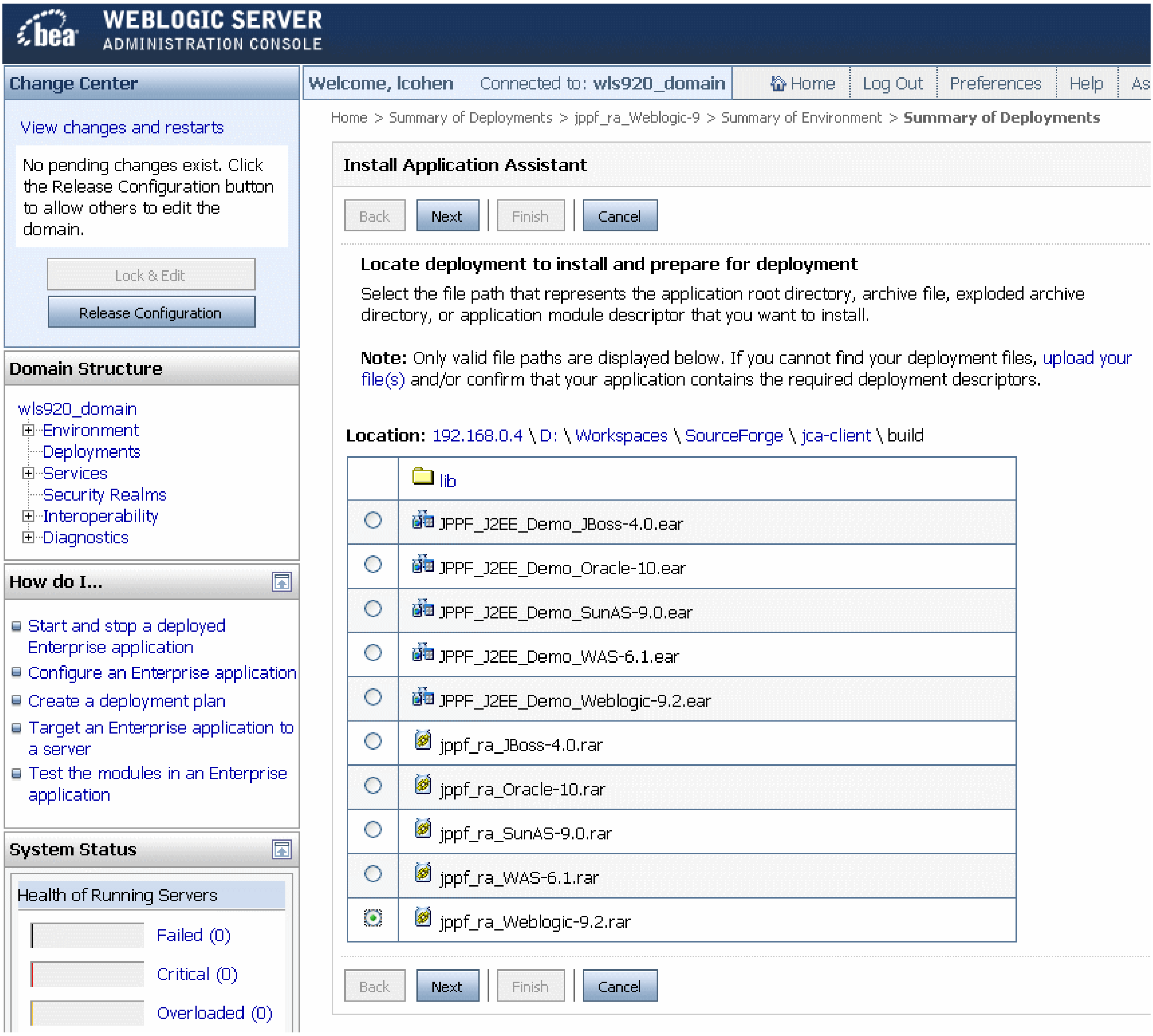
- click on "Start"
- restart the application server

8.5.6 Deployment on Weblogic

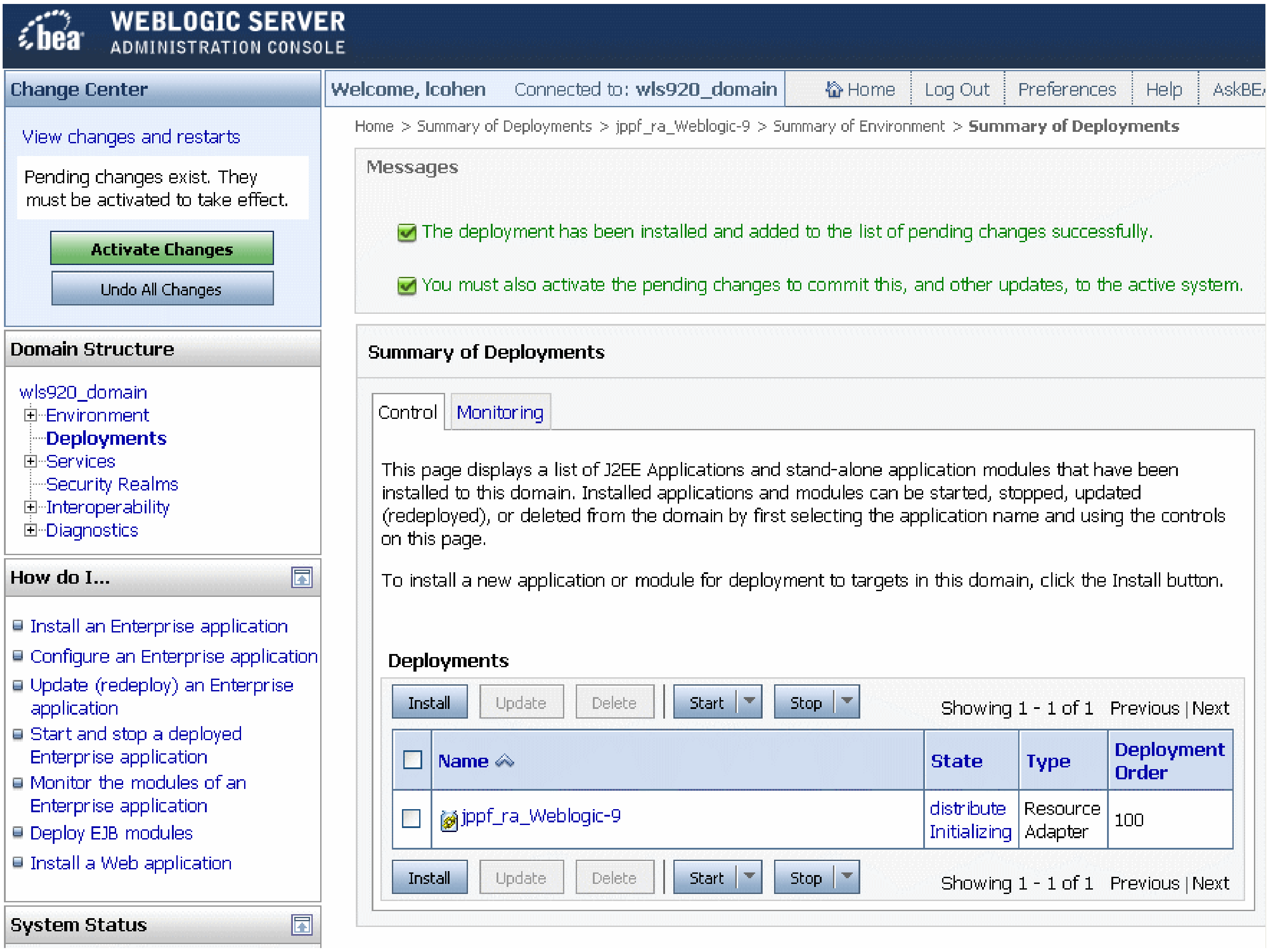
8.5.6.1 Deploying the resource adapter

- in Weblogic console, go to "Deployments"
- click on "Lock & Edit"

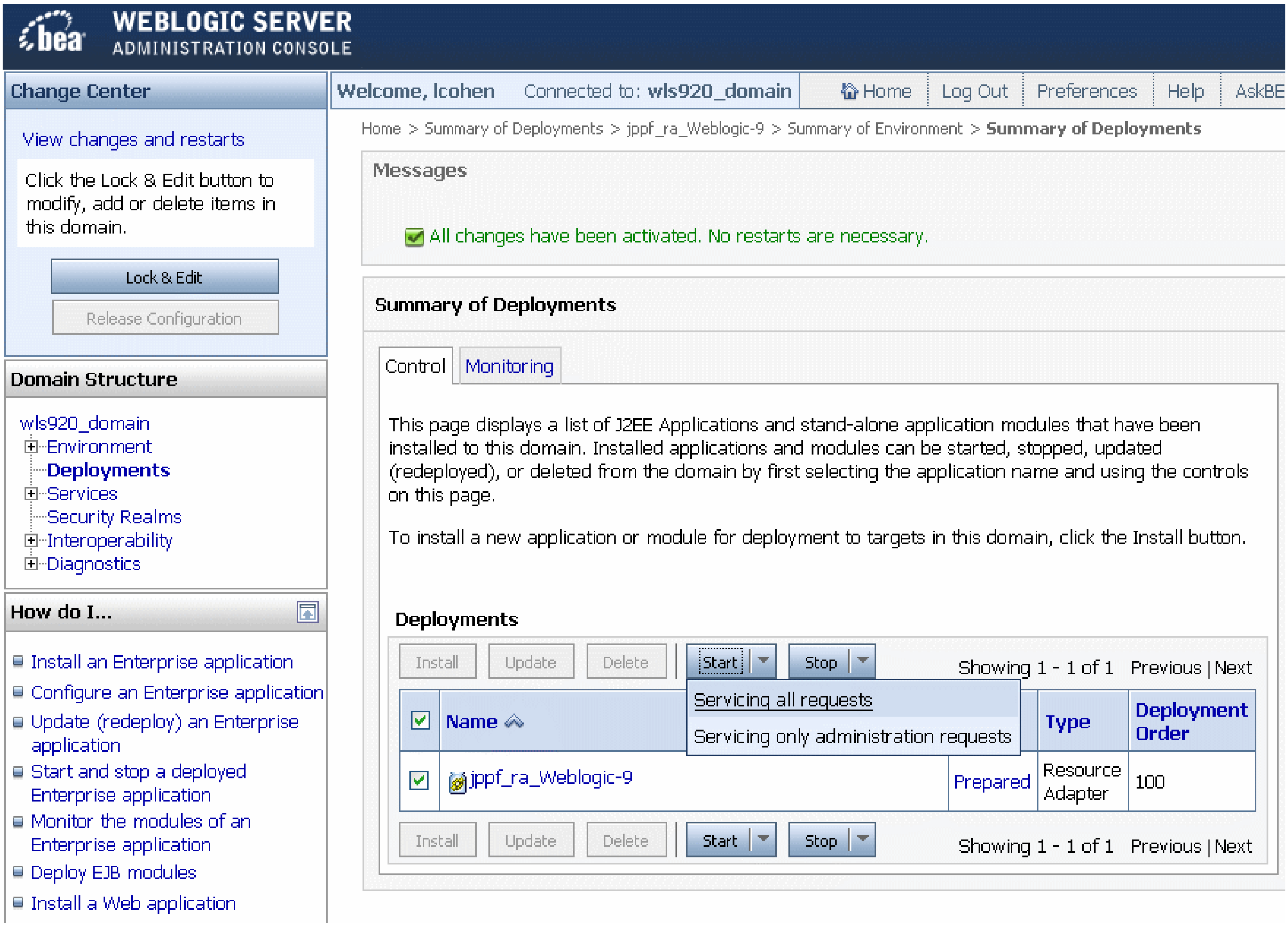
- click "Install"
- navigate to the "jppf_ra_Weblogic.rar" file and select it



- click "Next"
- click "Next"
- click "Finish"



- click "Activate Changes"
- in the list of deployments, check "jppf_ra_Weblogic"
- select "Start > Servicing all requests"



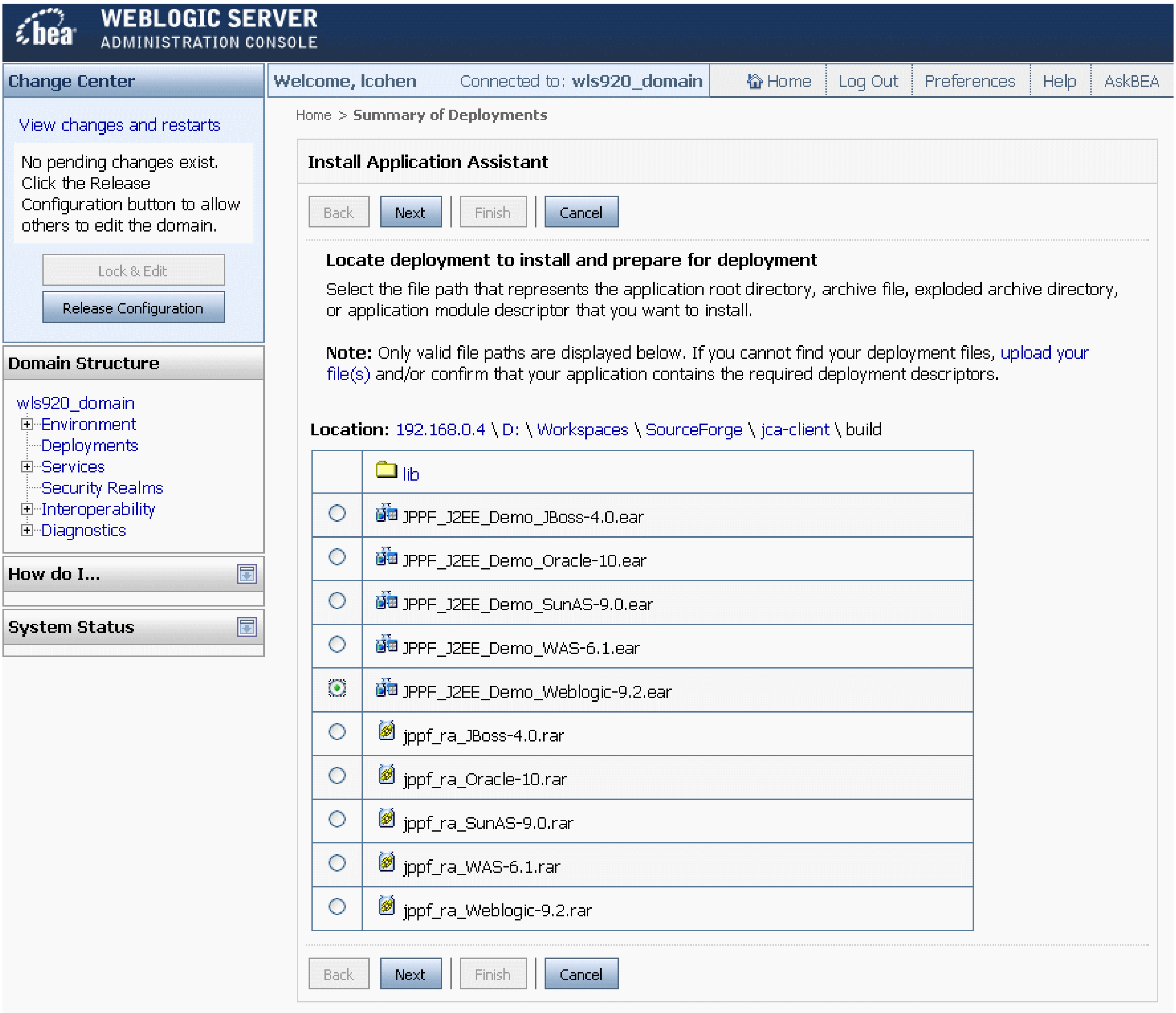
- Click "Yes"
- the state of the resource adapter must now show as "Active"
- restart the application server

Note: *In the Weblogic output console, you will probably see periodic messages saying that 2 threads are stuck. These warnings are harmless. The related threads are required by the JPPF resource adapter and should not be interrupted. The period of these warnings is determined by a setting of the Weblogic instance called "Stuck Thread Timer Interval", set to 60 seconds by default. Consult with your administrator if you need to change that interval*.

8.5.6.2 Deploying the demo application

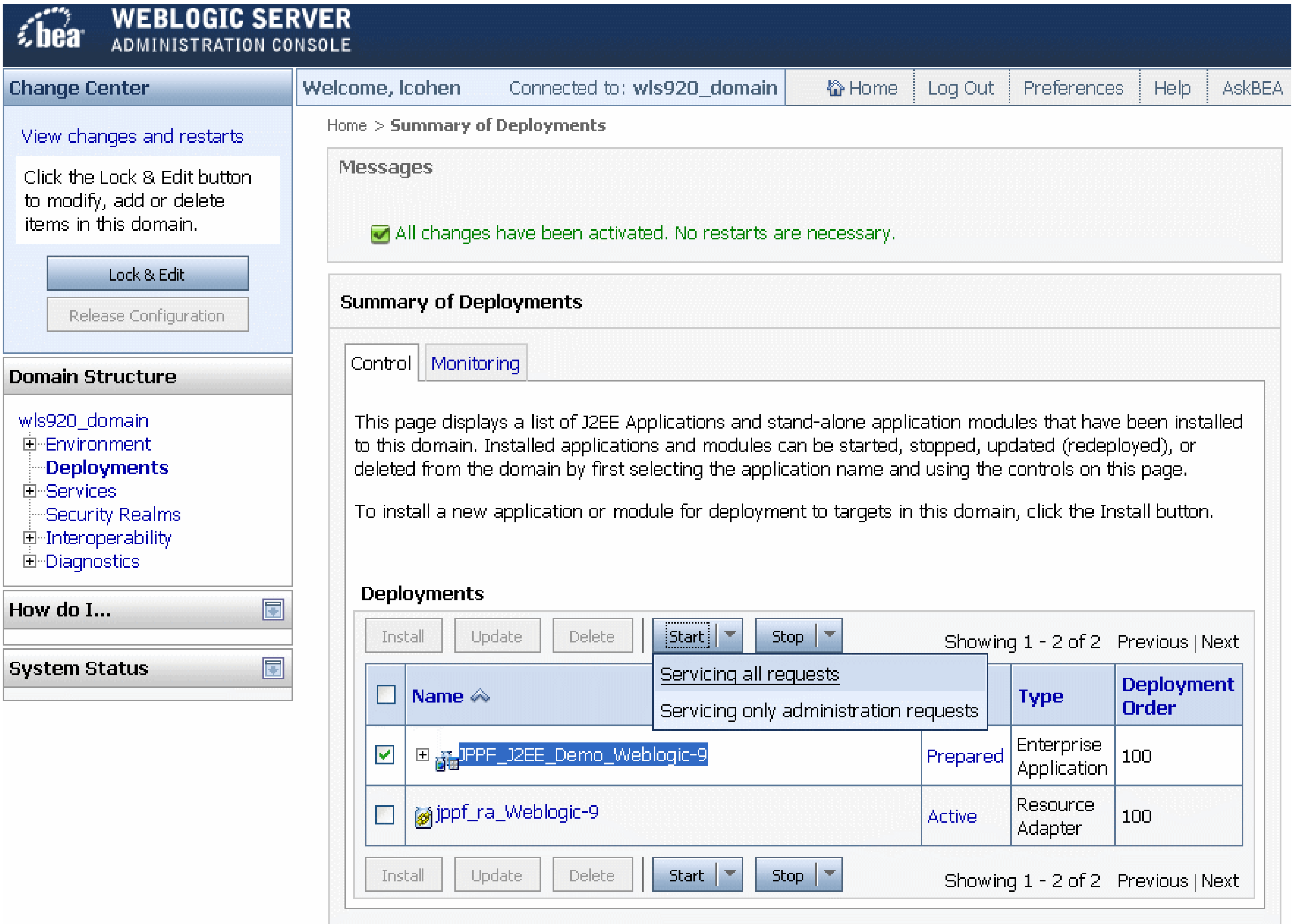
- in Weblogic console, go to "Deployments"
- click on "Lock & Edit"

- click "Install"
- navigate to the "JPPF_J2EE_Demo_Weblogic.ear" file and select it



- click "Next"
- click "Finish"

- click "Activate Changes"
- in the list of deployments, check "JPPF_J2EE_Demo_Weblogic"
- select "Start > Servicing All Requests"

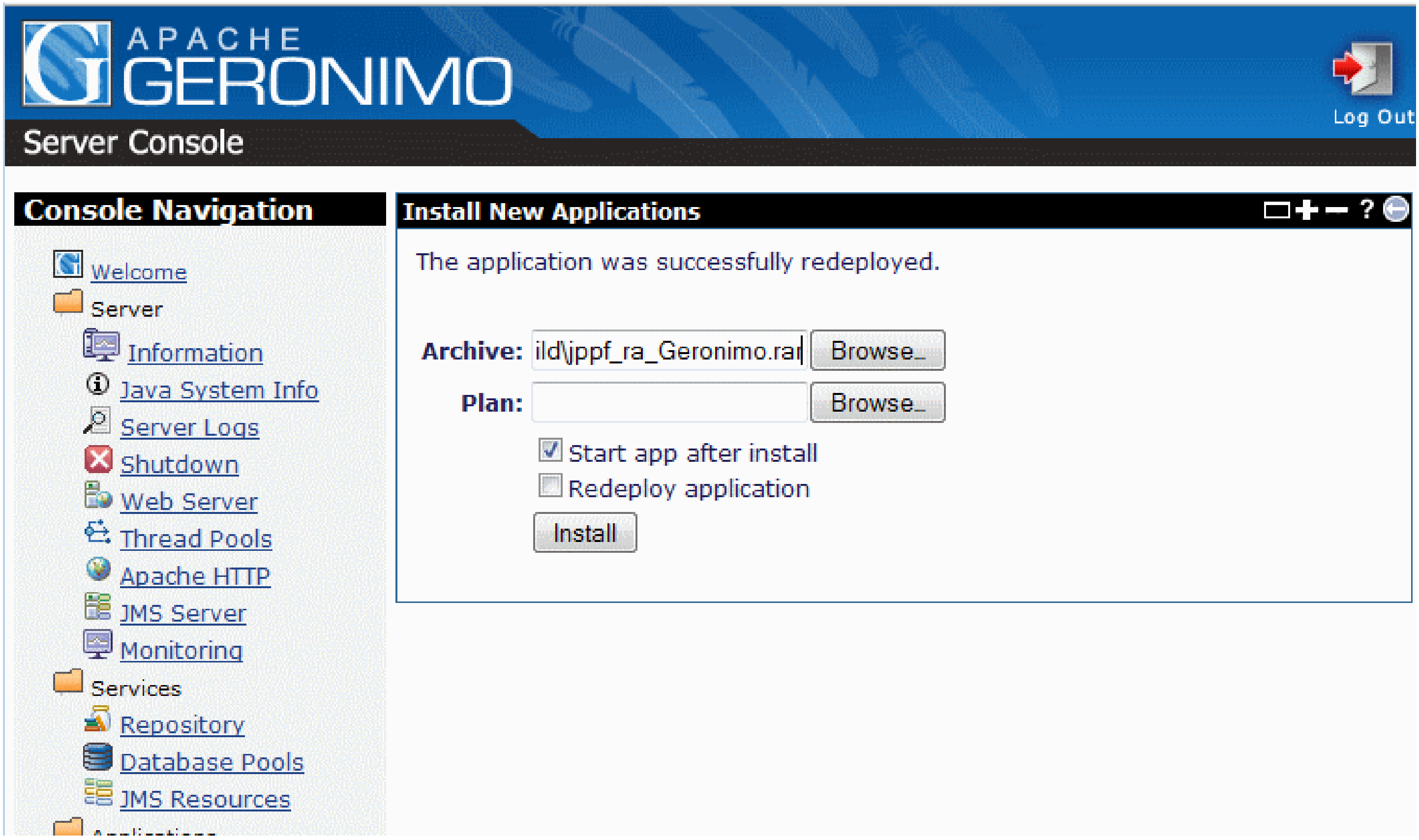


- Click "Yes"
- the state of the demo application must now show as "Active"
- restart the application server

8.5.7 Deployment on Apache Geronimo

8.5.7.1 Deploying the resource adapter

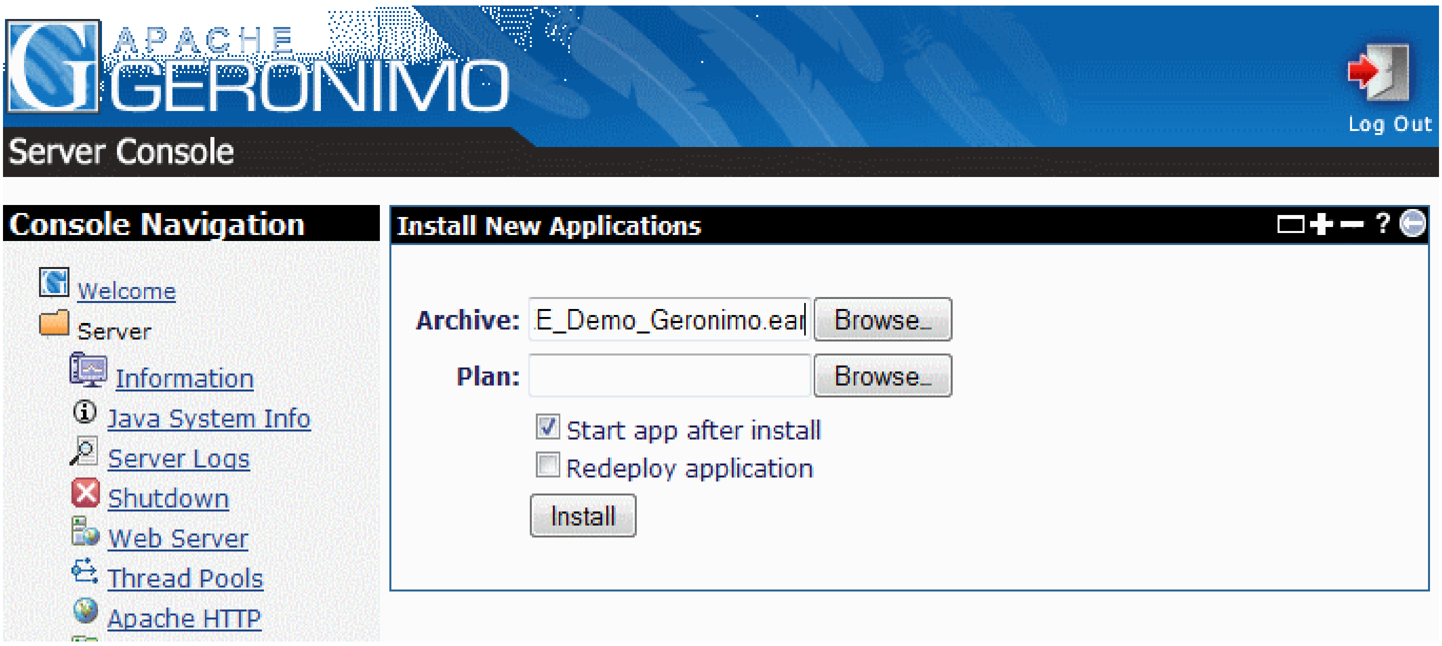
- In the Geronimo administration console, click on “Applications > Deploy new”
- In the “Archive“ field, navigate to the file “jppf_ra_Geronimo.rar”



- click on “Install”

8.5.7.2 Deploying the demo application

- In the Geronimo administration console, click on “Applications > Deploy new”
- In the “Archive“ field, navigate to the file “JPPF_J2EE_Demo_Geronimo.ear”



- click on “Install”

8.6 Packaging your enterprise application

For a J2EE enterprise application to work with the JPPF JCA connector, it is necessary to include a JPPF utility library called jppf-j2ee-client.jar, which can be found in the jca-client/build/lib folder. To ensure that this library can be made visible to all modules in the application, we recommend the following way of packaging it:

* add jppf-j2ee-client.jar in a lib folder under the root of the EAR file
* for each EJB, Web or Resource Adapter module of your application that will use JPPF, add a Class-Path entry in the META-INF/manifest.mf of the module, which will point to the JPPF library, for instance:
  Class-Path: lib/jppf-j2ee-client.jar

In a typical J2EE application, it would look like this:

```
MyApplication.ear/

    lib/
        jppf-j2ee-client.jar

    MyEJBModule.jar/
        ...
        META-INF/
            manifest.mf:
                ...
                Class-Path: lib/jppf-j2ee-client.jar
                ...
        ...

    ... other modules ...

    MyWebApp.war/
        ...
        META-INF/
            manifest.mf:
                ...
                Class-Path: lib/jppf-j2ee-client.jar
                ...
        ...
```

Note: If you only need to use JPPF from a web application or module, then you can simply add jppf-j2ee-client.jar to the WEB-INF/lib folder of the war file.

8.7 Creating an application server port

If the JPPF resource adapter does not include, out-of-the-box, a port for your application server, or your application server version, this section is for you. Here is a sequence of steps to create your own port:

1. copy one of the existing application server-specific folder in `JPPF-2.0-j2ee-connector/appserver` and give it a name that will distinguish it from the others. This name will be used throughout this process, so please make sure it is both unique and meaningful. For the sake of this exercise, we will use a generic name: "`MyServer-1.0`"

2. After creating the `JPPF-2.0-j2ee-connector/appserver/MyServer-1.0` folder, edit the relevant configuration files and deployment descriptors.

3. Open the build.xml build script, in the jca-client folder, with a text editor.

4. At the start of the file, you will see the following section:

```
<!-- ==================================================================== -->
<!-- definition of application server-specific properties                 -->
<!-- the value is used to generate the names of the corresponding EAR and RAR  -->
<!-- ==================================================================== -->
<property name="was"           value="Websphere"/>
<property name="jboss"         value="JBoss"/>
<property name="oracle"        value="Oracle"/>
<property name="sunas"         value="Glassfish"/>
<property name="weblogic"      value="Weblogic"/>
<property name="geronimo"      value="Geronimo"/>
```

You can add your own property here, for instance:

```
<property name="myserver10"      value="MyServer-1.0"/>
```

The property value must be the name of the folder you just created.

5. (optional) navigate to the Ant target "ear.all" and add your own invocation for generating the demo application EAR:

```
<antcall target="ear">
  <param name="appserver" value="${myserver10}"/>
  <param name="include.client.classes" value="true"/>
</antcall>
```

You may also remove or comment out those you do not need.

6. Navigate to the Ant target "ear.all" and add your own invocation for generating the resource adapter RAR:

```
<antcall target="rar"><param name="appserver" value="${myserver10}"/></antcall>
```

You may also remove or comment out those you do not need.

9 GigaSpaces XAP Connector

9.1 About GigaSPaces and JPPF

GigaSpaces is a full-fledged, non-J2EE application server that can handle, among other features, large distributed datasets. Integrating JPPF within GigaSpaces XAP adds the ability to perform intensive computations on those data, seamlessly and without impacting the performance of the applications running within GigaSpaces. It also provides JPPF applications with capabilities that were so far missing form its desing: clustering, replication, transaction management, data access, etc ...

9.2 Installation

9.2.1 Prerequisites

Before installing, please make sure you have the following already installed in your environment:

- Java Runtime Environment (JRE) version 1.5 or later
- Apache Ant version 1.6.2 or later
- GigaSpaces XAP version 6.6 or later

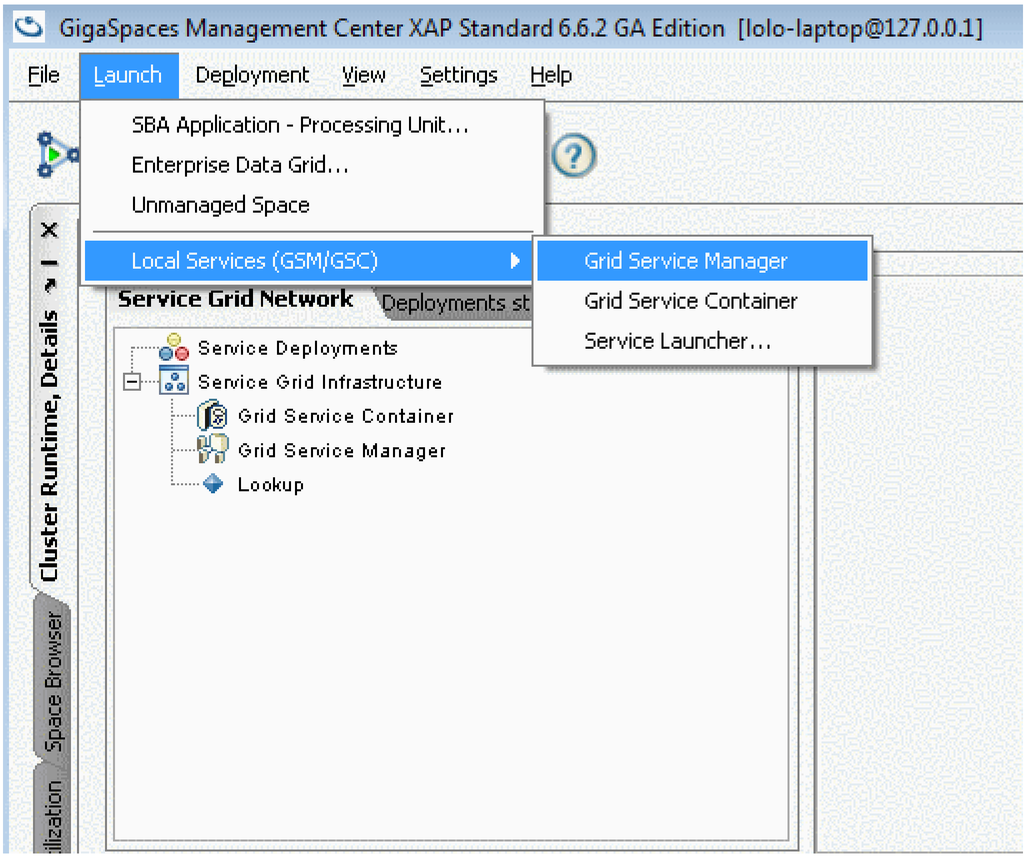
9.2.2 Installing

To install the JPPF / GigaSpaces integration package, perform the following steps:

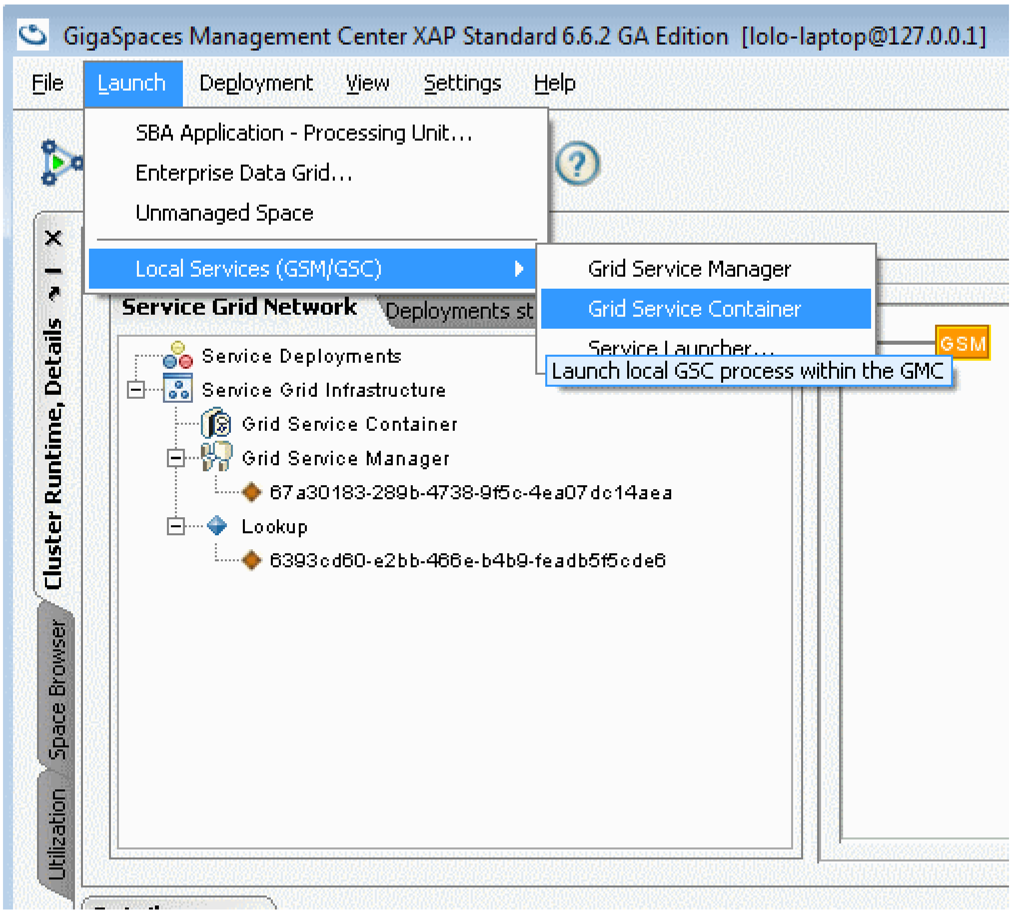
– download the **JPPF-2.x-GigaSpaces-.zip** file from the JPPF download page
– Unzip this file in a location of your choice, this will create a new folder named "`JPPF-GigaSpaces`"
– Edit the file "`JPPF-GigaSpaces/build.properties`" in a text editor and set the appropriate path for the GigaSpaces XAP install root folder
– you are now ready to use the JPPF / GigaSpaces integration

9.3 Deploying the JPPF processing unit

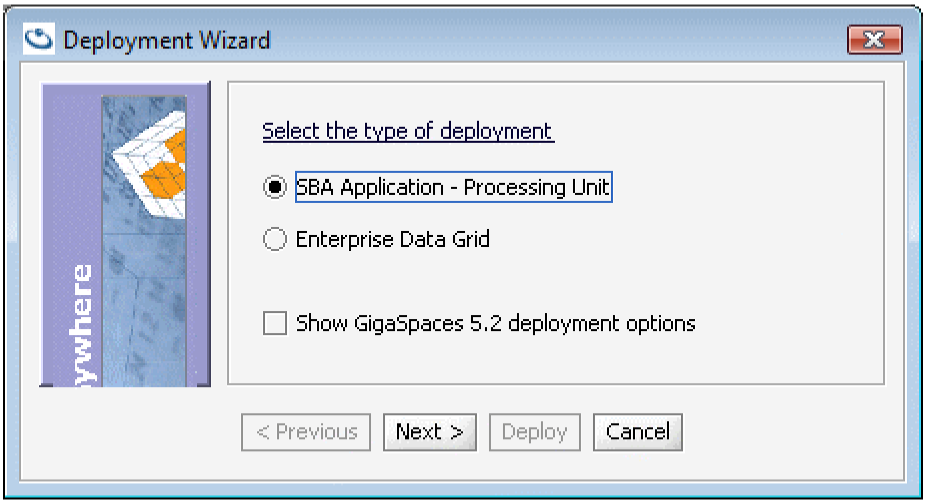
- In the GigaSpaces Management Center, start a Grid Service Manager (GSM):



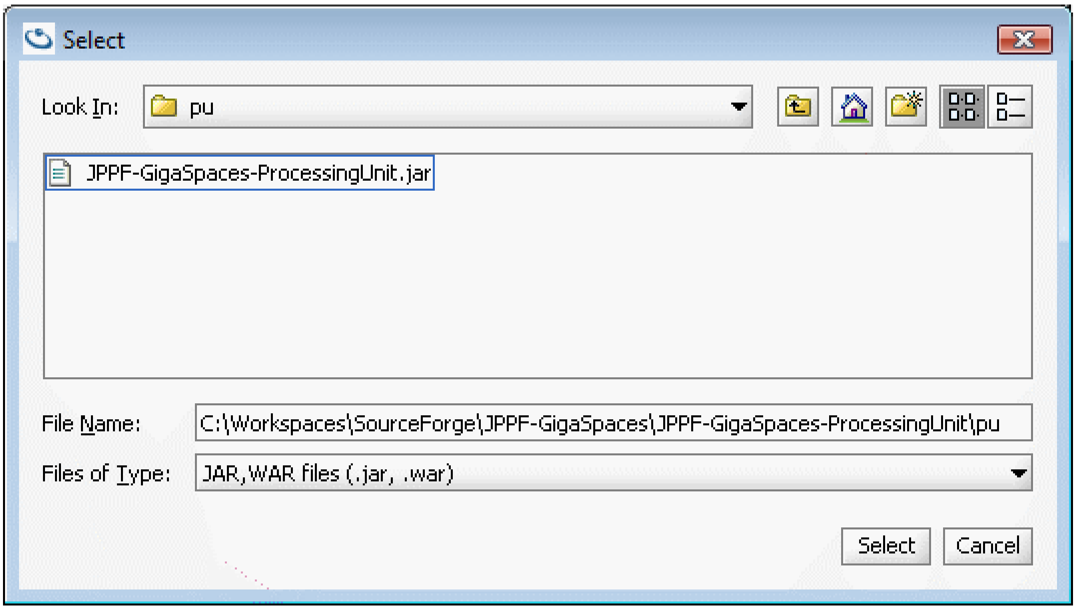
- Next, start a Grid Service Container (GSC):



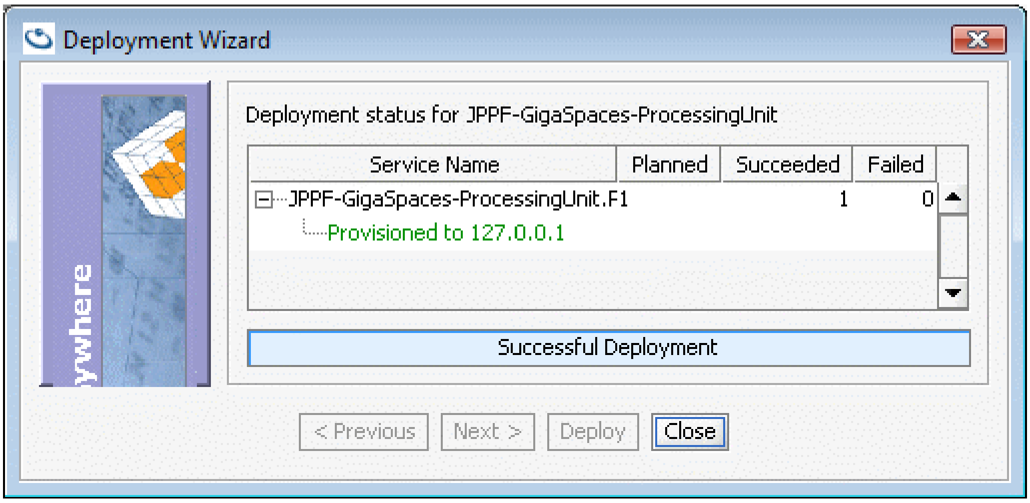
- Next we will deploy a JPPF client as a Processing Unit. To start, open the deployment wizard, as in the screenshot below:



- On the next screen, select the Processing Unit jar file to deploy, located at **`<JPPF-GigaSpaces>/JPPF-GigaSpaces-ProcessingUnit/pu/JPPF-GigaSpaces-ProcessingUnit.jar`**:



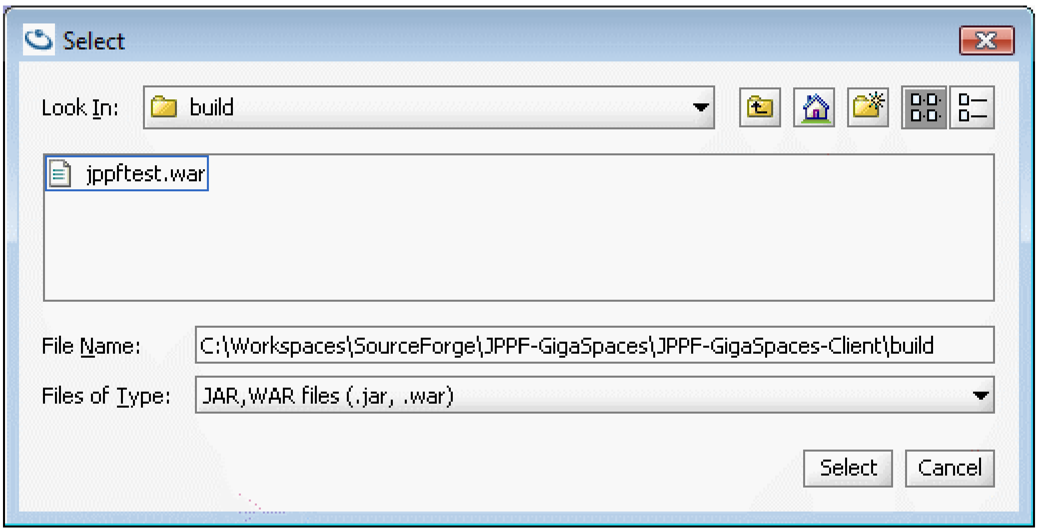
- After selecting the jar file, click "finish" and wait until the processing unit is deployed:



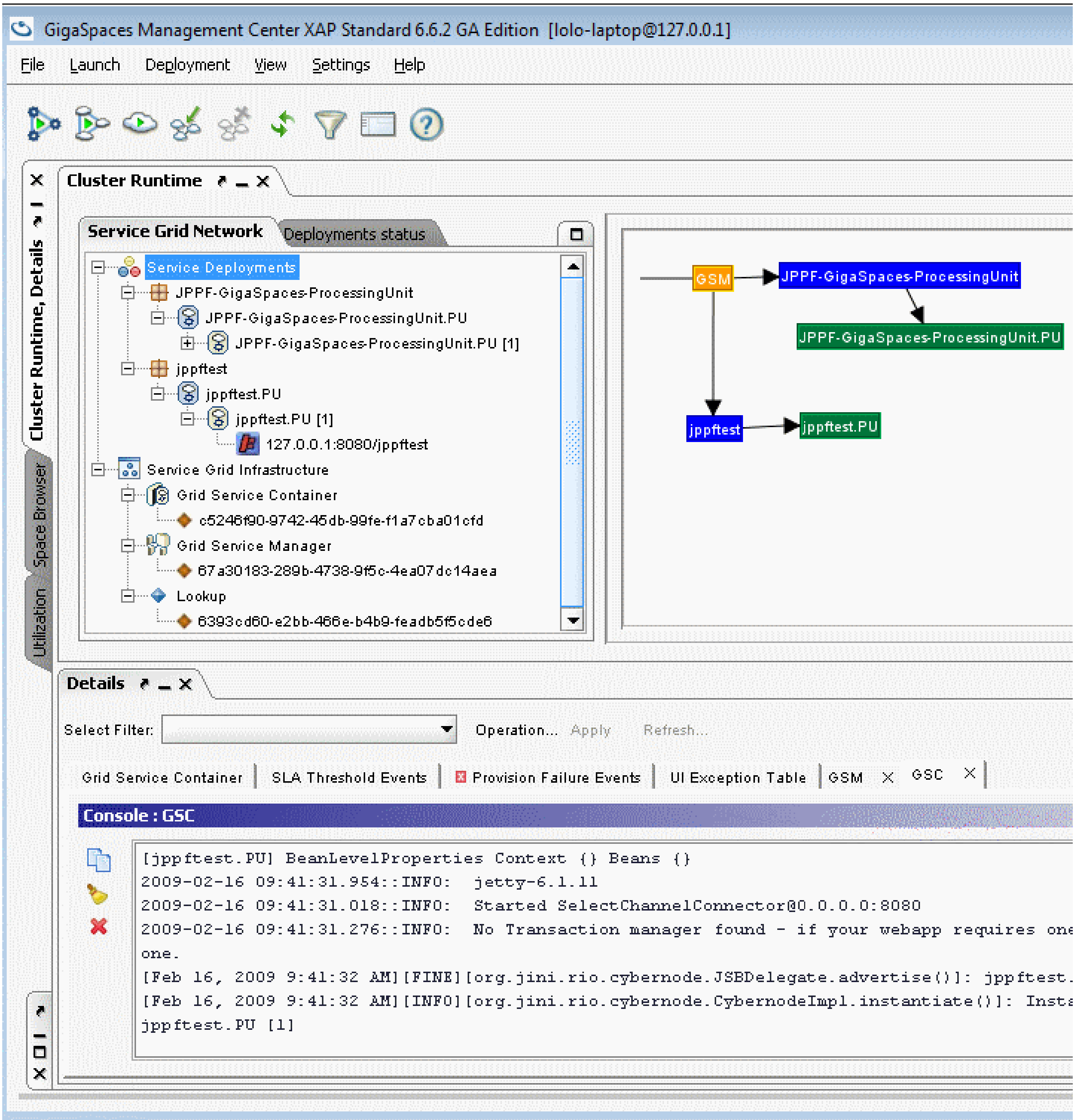
9.4 Deploying the sample web application

We will now deploy a sample web application packaged as a war file

1. First ensure that the JPPF processing unit has been deployed
2. Open the deployment wizard and select the war file to deploy, located at:
`<JPPF-GigaSpaces>/JPPF-GigaSpaces-Client/build/jppftest.war`



3. After selecting the war file, click "finish" and wait until the web application is deployed, the UI should now show both processing unit and web app deployed under the GSM:



4. The web application is now ready to be used at this URL: http://localhost:8080/jppftest

9.5 Considerations for deploying JPPF-enabled applications

9.5.1 Available APIs

The JPPF processing unit publishes all its libraries in its '"shared-lib'" folder, which makes the required JPPF APIs available to other applications deployed within GigaSpaces:

- the job and tasks APIs: `JPPFTask`, `JPPFJob`, `DataProvider`, `ExecutionPolicy`
- the service API used to submit jobs to the JPPF service: `JPPFService`

9.5.2 Spring descriptor

Access to the JPPF service is realized by declaring it as a remoted Spring bean in your application. This is done as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:os-core="http://www.openspaces.org/schema/core"
       xmlns:os-events="http://www.openspaces.org/schema/events"
       xmlns:os-remoting="http://www.openspaces.org/schema/remoting"
       xmlns:os-sla="http://www.openspaces.org/schema/sla"
       xmlns:os-jms="http://www.openspaces.org/schema/jms"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.openspaces.org/schema/core
       http://www.openspaces.org/schema/core/openspaces-core.xsd
       http://www.openspaces.org/schema/events
       http://www.openspaces.org/schema/events/openspaces-events.xsd
       http://www.openspaces.org/schema/remoting
       http://www.openspaces.org/schema/remoting/openspaces-remoting.xsd
       http://www.openspaces.org/schema/jms
       http://www.openspaces.org/schema/jms/openspaces-jms.xsd
       http://www.openspaces.org/schema/sla
       http://www.openspaces.org/schema/sla/openspaces-sla.xsd">

    <!-- Declaration of the JPPF space. -->
    <os-core:space id="space" url="jini://*/*/JPPFSpace" />
    <os-core:giga-space id="gigaSpace" space="space"/>

    <!-- The JPPF service proxy. -->
    <os-remoting:sync-proxy
     id="jppfService" giga-space="gigaSpace"
     interface="org.jppf.gigaspaces.JPPFService"/>

    <!--
      The GSClient bean, uses the proxied service
      without any knowledge of the remoting invocation.
    -->
    <bean id="gsclient" class="org.jppf.gigaspaces.test.GSClient">
        <property name="jppfService" ref="jppfService"/>
    </bean>
</beans>
```

This descriptor can be found in the distribution as:
<JPPF-GigaSpaces>/JPPF-GigaSpaces-Client/src/client.xml

9.5.3 Usage in application code

Obtaining a reference to the JPPF service is done via the Spring application context:

```java
// Parse the application context and instantiate the bean
ClassPathXmlApplicationContext context =
  new ClassPathXmlApplicationContext("classpath:client.xml");
context.start();
// Get a reference to the created bean
GSClient gsc = (GSClient) context.getBean("gsclient");
```

The call to `context.start()` causes the Sprint bean's `afterPropertiesSet()` method to be invoked:

```java
/**
 * Called after the Spring bean init and submits a JPPF job to the JPPF space.
 * @throws Exception if any error occurs.
 * @see org.springframework.beans.factory.InitializingBean#afterPropertiesSet()
 */
public void afterPropertiesSet() throws Exception
{
  // Create a new job and add a task
  JPPFJob newJob = new JPPFJob();
  newJob.addTask(new HelloTask());
  // invoke the JPPF service proxy to submit the job
  this.job = jppfService.submitJob(newJob);
}
```

Putting it all together, here is the code used in our sample web application:

```
package org.jppf.gigaspaces.test;

import org.jppf.client.JPPFJob;
import org.jppf.gigaspaces.*;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * Client class used to invoke a JPPF job submission service
 * deployed as a processing unit.
 */
public class GSClient implements InitializingBean
{
  private JPPFService jppfService = null;
  private JPPFJob job = null;

  // Entry point for execution of this client as a standalone application.
  public static void main(String[] args)
  {
    execute();
  }

  /**
   * Initialize the Spring context, invoke the appropriate bean method,
   * and store the results of the JPPF execution.
   * @return the results as a <code>JPPFJob</code> instance.
   */
  public static JPPFJob execute()
  {
    ClassPathXmlApplicationContext context =
      new ClassPathXmlApplicationContext("classpath:client.xml");
    context.start();
    GSClient gsc = (GSClient) context.getBean("gsclient");
    return gsc.getJob();
  }

  // Called after the Spring bean init and submits a JPPF job to the JPPF space.
  public void afterPropertiesSet() throws Exception
  {
    JPPFJob newJob = new JPPFJob();
    newJob.addTask(new HelloTask());
    this.job = jppfService.submitJob(newJob);
  }

  // Get a proxy to the service deployed in a GS space.
  public JPPFService getJppfService()
  {
    return jppfService;
  }

  // Set a proxy to the service deployed in a GS space.
  public void setJppfService(JPPFService service)
  {
    this.jppfService = service;
  }

  // Get the resulting JPPF job instance.
  public JPPFJob getJob()
  {
    return job;
  }
}
```

10 TCP Multiplexer

We have seen in the ***Configuration Guide*** chapter that JPPF requires at least 3 TCP ports to function. In network environments where a policy imposes the use of a firewall, these ports will generally be blocked, preventing any traffic through unauthorized ports.

This situation is very common, and in the case of JPPF, is very difficult, even nearly impossible, to work around. To address this situation, we have implemented a tool called **TCP port multiplexer**, that enables the routing of network traffic from multiple ports to a single port on one side, and the routing of the same traffic from a single port to multiple ones on the other side.

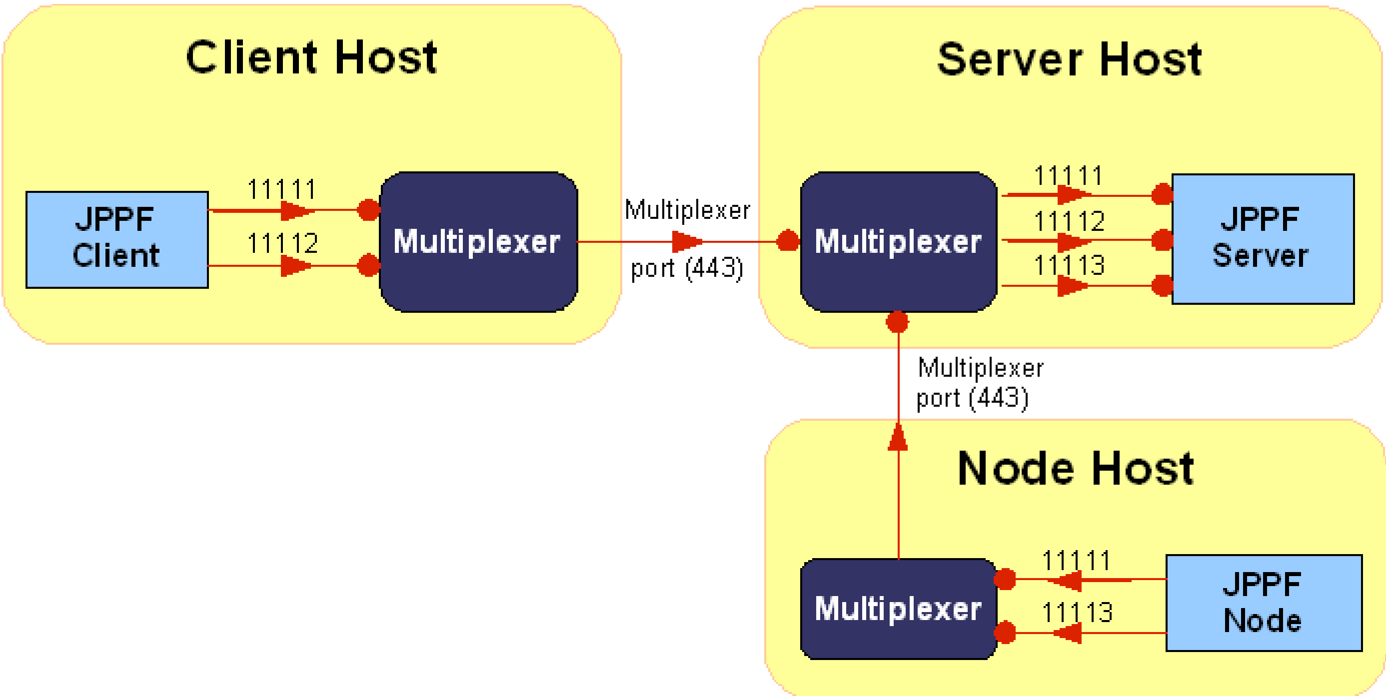
With this tool, it becomes possible to work with JPPF in firewalled environments, without having to deploy a complex hardware/software infrastructure. It also comes as a component separate from JPPF clients, servers and nodes, which means that client applications continue to use and deploy JPPF in exactly the same way as they already do.

10.1 Architecture

To understand how the multiplexer works and how it integrates with a JPPF grid, let's take the example of a typical JPPF configuration:

– the JPPF client, server and node are all on separate machines and use the default ports 11111, 11112 and 11113
– the network environment includes a firewall that will only let traffic through port 443

This results in a configuration illustrated by the following figure:



A directed red arrow represents an incoming connection. The small circle at one end means that the corresponding component is listening for connections on the specified port.

This architecture bears a number of remarks:

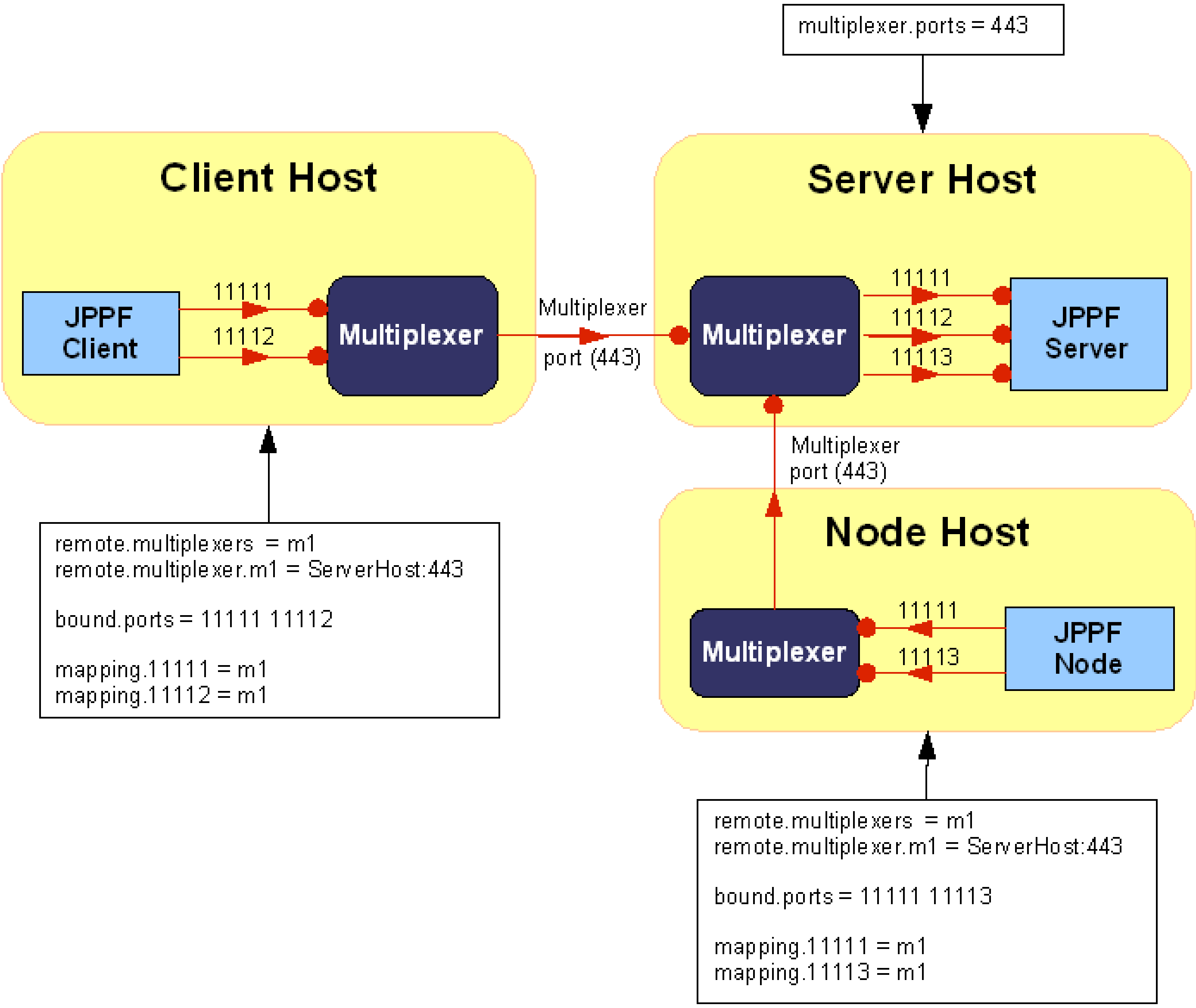
– nodes and clients connect to the multiplexer "as if it were a "'local"' JPPF server"
– the JPPF server accepts connections from the multiplexer "as if they came from "'local"' clients and nodes"
– the multiplexing and demultiplexing of the network traffic is completely transparent to the JPPF components
– the multiplexer implementation is based on the same network communication APIs used in JPPF, and incurs the same benefits in terms of scalability, robustness and reliability
– as additional network routing steps are added, a performance decrease should be expected, compared to a non-multiplexed JPPF grid
– in this configuration, management and monitoring of the nodes and server is not possible, and should be disabled for the nodes and servers behind a firewall

10.2 Configuration

In terms of configuration, we have seen in the previous section that the multiplexer plays a different role, depending on whether it is on the JPPF server machine or not:

– a multiplexer on a JPPF server host only needs to listen to connections on a multiplexing port
– a multiplexer on a JPPF client or node host binds to the JPPF ports, as if it were a server. It needs to know which ports to bind to, and for each bound port which remote multiplexer to route the traffic to.

This is illustrated in the following figure:



Here is a full configuration example for a server host:

```
# Local forwarding port this multiplexer listens to
# The local port number to which to forward messages to is the first message
# sent by any inbound connection
multiplexer.ports = 443
```

And for a non-server host:

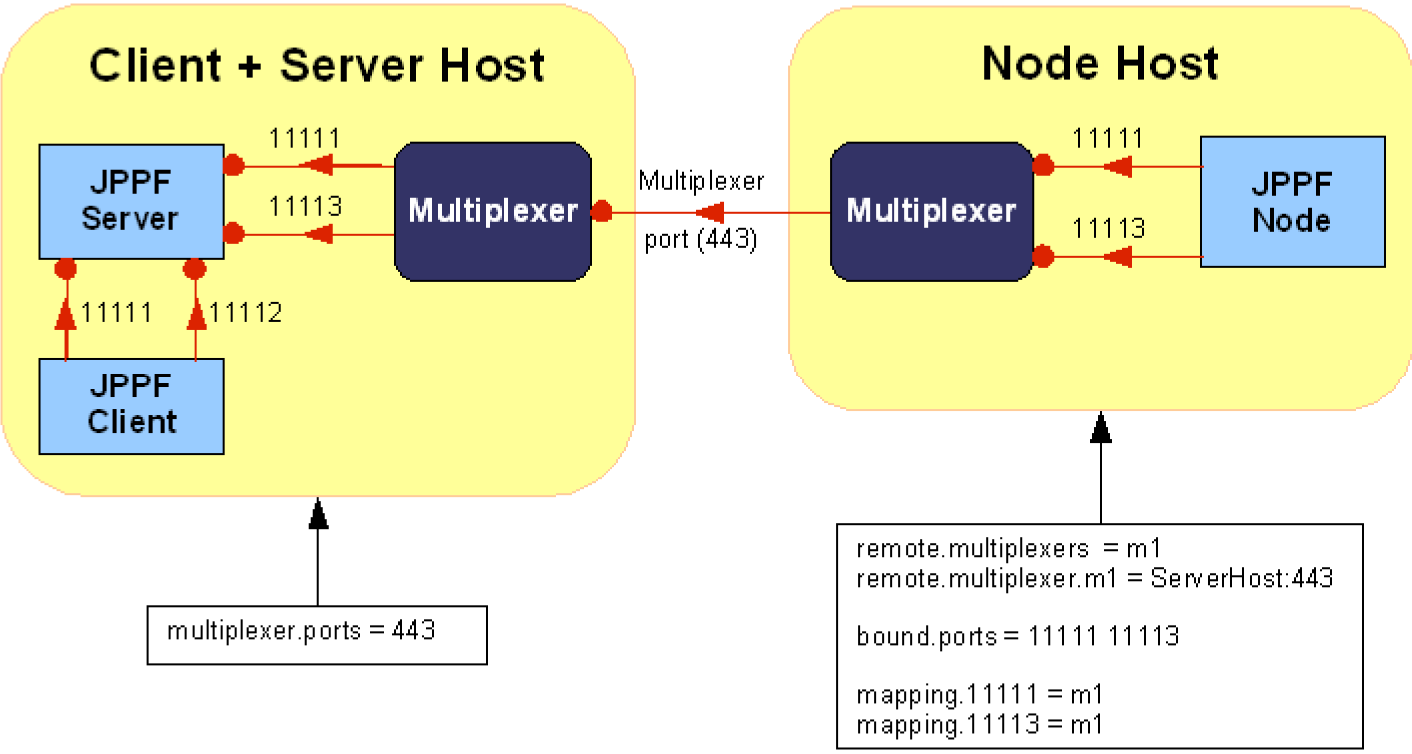
```
# Names of the remote multiplexers to connect to
remote.multiplexers = m1

# Host and port corresponding to the named multiplexer
remote.multiplexer.m1 = server_host:443

# Local ports this multiplexer listens to
bound.ports = 11111 11112 11113
```

```
# Mapping of the bound ports to remote multiplexers (1 per port)
# Communications with the bound ports will be forwarded to and from the
# multiplexer
mapping.11111 = <font color="blue">m1</font>
mapping.11112 = <font color="blue">m1</font>
mapping.11113 = <font color="blue">m1</font>
```

Note:

- *The clients and nodes must use, in their multiplexer configuration, the same port numbers as those specified for the JPPF server. These port numbers are actually sent to the server multiplexer, so that it knows how to re-route the traffic.*
- *The JPPF host specified in the clients and nodes configuration must be set to* `localhost` *or* `127.0.0.1`*, as they connect to the local multiplexer.*

Another use case is when you have not only a JPPF server, but also a client or a node on the same machine. In this case, the node or client does not need a multiplexer, as it will connect **locally** to the JPPF server. This is illustrated in this figure:



10.3 Deployment

Each multiplexer instance is deployed as a separate process. It has its own configuration file and logging configuration file, as for the JPPF components.

The TCP port multiplexer is released in its own package, so it can be easily deployed on any machine. It is the package named JPPF-2.x-multiplexer.zip available on the JPPF download page.

To run the multiplexer:

– unzip the JPPF-2.x-multiplexer.zip anywhere on your hard drive
– for a server host:
  - edit the **config/multiplexer-server.properties** file and set the appropriate values for your environment
  - from a shell or command prompt, run the Ant target "**ant run.multiplexer.server**"

– for a client or node host with no JPPF server:
  - edit the **config/multiplexer.properties** file and set the appropriate values for your environment
  - from a shell or command prompt, run the Ant target "**ant run.multiplexer**"

11 Configuration properties reference

11.1 Server properties

Property name	Default Value	Comments
class.server.port	11111	driver's class loader port
app.server.port	11112	server / client communication port
node.server.port	11113	server / node communication port
jppf.management.enabled	true	enable server management
jppf.management.host	computed	management server host
jppf.management.port	11198	management remote connector port
jppf.management.rmi.port	12198	management server's RMI registry port
jppf.discovery.enabled	true	enable server broadcast and discovery
jppf.discovery.group	230.0.0.1	UPD broadcast group
jppf.discovery.port	11111	UPD broadcast port
jppf.discovery.broadcast.interval	5000	UDP broadcast interval in milliseconds
jppf.peers	null	space separated list of peer server names
jppf.peer.<name>.server.host	localhost	peer server host name
class.peer.<name>.server.port	11111	peer server class loader port
node.peer.<name>.server.port	11113	peer server node communication port
jppf.peer.dicovery.enabled	false	enable peer discovery
jppf.load.balancing.algorithm	proportional	load balancing algorithm name
jppf.load.balancing.strategy	jppf	load balancing parameters profile name
strategy.<profile>.<parameter>	null	parameter for the named parameters profile
jppf.jvm.options	null	JVM options for the server process
jppf.object.stream.builder	null	optional object stream builder
jppf.object.input.stream.class	java.io.ObjectInputStream	optional alternate object input stream
jppf.object.output.stream.class	java.io.ObjectOutputStream	optional alternate object input stream
jppf.data.transform.class	null	optional network data transformation
transition.thread.pool.size	number of available CPUs	number of threads performing network I/O
jppf.local.node.enabled	false	enable a node to run in the same JVM
jppf.recovery.enabled	false	enable recovery from hardware failures on the nodes
jppf.recovery.max.retries	3	maximum number of failed pings to the node before the connection is considered broken
jppf.recovery.read.timeout	6000 (6 seconds)	maximum ping response time from the node
jppf.recovery.server.port	22222	port number for the detection of node failure
jppf.recovery.reaper.run.interval	60000 (1 minute)	interval between connection reaper runs
jppf.recovery.reaper.pool.size	number of available CPUs	number of threads allocated to the reaper
jppf.socket.max-idle	-1	number of seconds a socket connection can remain idle before being closed

11.2 Node properties

Property name	Default Value	Comments
jppf.server.host	localhost	JPPF server host address
class.server.port	11111	driver's class loader port
node.server.port	11113	server / node communication port
jppf.management.enabled	true	enable server management
jppf.management.host	computed	node's management server host
jppf.management.port	11198	management remote connector port
jppf.management.rmi.port	12198	management server's RMI registry port
jppf.discovery.enabled	true	enable server discovery
jppf.discovery.group	230.0.0.1	server discovery: UPD multicast group
jppf.discovery.port	11111	server discovery: UPD multicast port
jppf.discovery.timeout	5000	server discovery timeout in milliseconds
jppf.discovery.ipv4.include	null	IPv4 inclusion patterns for server discovery
jppf.discovery.ipv4.exclude	null	IPv4 exclusion patterns for server discovery
jppf.discovery.ipv6.include	null	IPv6 inclusion patterns for server discovery
jppf.discovery.ipv6.exclude	null	IPv6 exclusion patterns for server discovery
jppf.jvm.options	null	JVM options for the node process
jppf.object.stream.builder	null	optional object stream builder
jppf.object.input.stream.class	java.io.ObjectInputStream	optional alternate object input stream
jppf.object.output.stream.class	java.io.ObjectOutputStream	optional alternate object input stream
jppf.data.transform.class	null	optional network data transformation
reconnect.initial.delay	1	delay in seconds before the first reconnection attempt
reconnect.max.time	60	delay in seconds after which reconnection attemps stop. Negative value = never stop
reconnect.interval	1	frequency in seconds of reconnection attempts
processing.threads	number of available CPUs	number of threads used for tasks execution
jppf.policy.file	null	path to the security policy file, either local to the node or in the server's file system
jppf.idle.mode.enabled	false	enable the idle mode
jppf.idle.timeout	300000 (5 minutes)	the time of keyboard and mouse inactivity before considering the node idle, expressed in milliseconds
jppf.idle.poll.interval	1000 (1 second)	how often the node will check for keyboard and mouse inactivity, in milliseconds
jppf.idle.detector.factory	null	implementation of the idle detector factory
jppf.recovery.enabled	false	enable recovery from hardware failures
jppf.recovery.server.port	22222	port number for the detection of hardware failure
jppf.classloader.cache.size	50	size of the class loader cache for the node
jppf.socket.max-idle	-1	number of seconds a socket connection can remain idle before being closed

11.3 Application client and admin console properties

Property name	**Default Value**	**Comments**
jppf.drivers	default-driver	space-separated list of driver names
<driver_name>.jppf.server.host	localhost	
<driver_name>.class.server.port	11111	named driver's class loader port
<driver_name>.app.server.port	11112	named server / client communication port
<driver_name>.jppf.management.host	localhost	named server's management server host
<driver_name>.jppf.management.port	11198	named server's management remote connector port
<driver_name>.jppf.management.enabled	true	enable remote management of named server
<driver_name>.priority	0	named server priority
<driver_name>.jppf.pool.size	1	named server connection pool size
jppf.client.max.init.time	5000 (5 seconds)	maximum JPPF client initialization wait time
reconnect.initial.delay	1	delay in seconds before the first reconnection attempt
reconnect.max.time	60	delay in seconds after which reconnection attemps stop. Negative value = never stop
reconnect.interval	1	frequency in seconds of reconnection attempts
jppf.local.execution.enabled	false	enable local execution
jppf.local.execution.threads	number of available CPUs	maximum threads to use for local execution
jppf.pool.size	1	connection pool size when discovery is enabled
jppf.discovery.enabled	true	enable server discovery
jppf.discovery.group	230.0.0.1	server discovery: UPD multicast group
jppf.discovery.port	11111	server discovery: UPD multicast port
jppf.discovery.ipv4.include	null	IPv4 inclusion patterns for server discovery
jppf.discovery.ipv4.exclude	null	IPv4 exclusion patterns for server discovery
jppf.discovery.ipv6.include	null	IPv6 inclusion patterns for server discovery
jppf.discovery.ipv6.exclude	null	IPv6 exclusion patterns for server discovery
jppf.object.stream.builder	null	optional object stream builder
jppf.object.input.stream.class	java.io.ObjectInputStream	optional alternate object input stream
jppf.object.output.stream.class	java.io.ObjectOutputStream	optional alternate object input stream
jppf.data.transform.class	null	optional network data transformation
jppf.ui.splash	true	enable display of splash screen at startup
jppf.socket.max-idle	-1	number of seconds a socket connection can remain idle before being closed

12 Execution policy reference

12.1 Execution Policy Elements

12.1.1 NOT

Negates a test

Class name: **org.jppf.node.policy.ExecutionPolicy.Not**
Usage:
```
policy = otherPolicy.not();
```

XML Element: **<NOT>**
Nested element: any other policy element, min = 1, max = 1
Usage:
```
<NOT>
  <Equal ignoreCase="true" valueType="string">
    <Property>some.property</Property>
    <Value>some value here</Value>
  </Equal>
</NOT>
```

12.1.2 AND

Combines multiple tests through a logical AND operator

Class name: **org.jppf.node.policy.ExecutionPolicy.And**
Usage:
```
policy = policy1.and(policy2).and(policy3);
policy = policy1.and(policy2, policy3);
```

XML Element: **<AND>**
Nested element: any other policy element, min = 2, max = unbounded
Usage:

```
<AND>
  <Equal ignoreCase="true" valueType="string">
    <Property>some.property.1</Property>
    <Value>some value here</Value>
  </Equal>
  <LessThan>
    <Property>some.property.2</Property>
    <Value>100</Value>
  </LessThan>
  <Contains ignoreCase="true" valueType="string">
    <Property>some.property.3</Property>
    <Value>substring</Value>
  </Contains>
</AND>
```

12.1.3 OR

Combines multiple tests through a logical OR operator

Class name: **org.jppf.node.policy.ExecutionPolicy.Or**
Usage:
```
policy = policy1.or(policy2).or(policy3);
policy = policy1.or(policy2, policy3);
```

XML Element: <OR>
Nested element: any other policy element, min = 2, max = unbounded
Usage:
```
<OR>
  <Equal ignoreCase="true" valueType="string">
    <Property>some.property.1</Property>
```

```
    <Value>some value here</Value>
  </Equal>
  <LessThan>
    <Property>some.property.2</Property>
    <Value>100</Value>
  </LessThan>
  <Contains ignoreCase="true" valueType="string">
    <Property>some.property.3</Property>
    <Value>substring</Value>
  </Contains>
</OR>
```

12.1.4 XOR

Combines multiple tests through a logical XOR operator

Class name: org.jppf.node.policy.ExecutionPolicy.Xor
Usage:

```
policy = policy1.xor(policy2).xor(policy3);
policy = policy1.xor(policy2, policy3);
```

XML Element: <XOR>
Nested element: any other policy element, min = 2, max = unbounded
Usage:

```
<XOR>
  <Equal ignoreCase="true" valueType="string">
    <Property>some.property.1</Property>
    <Value>some value here</Value>
  </Equal>
  <LessThan>
    <Property>some.property.2</Property>
    <Value>100</Value>
  </LessThan>
  <Contains ignoreCase="true" valueType="string">
    <Property>some.property.3</Property>
    <Value>substring</Value>
  </Contains>
</XOR>
```

12.1.5 Equal

Performs a test of type "property_value = value". The value can be either numeric, boolean or a string.

Class name: **org.jppf.node.policy.Equal**
Constructors:

```
Equal(String propertyName, boolean ignoreCase, String value)
Equal(String propertyName, double value)
Equal(String propertyName, boolean value)
```

Usage:

```
policy = new Equal("some.property", true, "some_value");
policy = new Equal("some.property", 15);
policy = new Equal("some.property", true);
```

XML Element: **<Equal>**
Attributes:
  ignoreCase: one of "true" or "false", optional, defaults to "false"
  valueType: one of "string", "numeric" or "boolean", optional, defaults to "string"
Nested elements:
  <Property> : name of a node property, min = 1, max = 1
  <Value> : value to compare with, min = 1, max = 1
Usage:

```
<Equal ignoreCase="true" valueType="string">
  <Property>some.property</Property>
  <Value>some value here</Value>
</Equal>
```

12.1.6 LessThan

Performs a test of type "property_value < value"

The value can only be numeric.

Class name: **org.jppf.node.policy.LessThan**
Constructor:
```
LessThan(String propertyName, double value)
```
Usage:
```
policy = new LessThan("some.property", 15.50);
```

XML Element: **<LessThan>**
Nested elements:
<Property> : name of a node property, min = 1, max = 1
<Value> : value to compare with, min = 1, max = 1
Usage:
```
<LessThan>
  <Property>some.property</Property>
  <Value>15.50</Value>
</LessThan>
```

12.1.7 AtMost

Performs a test of type "property_value<= value"
The value can only be numeric.

Class name: **org.jppf.node.policy.AtMost**
Constructor:
```
AtMost(String propertyName, double value)
```
Usage:
```
policy = new AtMost("some.property", 15.49);
```

XML Element: **<AtMost>**
Nested elements:
<Property> : name of a node property, min = 1, max = 1
<Value> : value to compare with, min = 1, max = 1
Usage:
```
<AtMost>
  <Property>some.property</Property>
  <Value>15.49</Value>
</AtMost>
```

12.1.8 MoreThan

Performs a test of type "property_value > value"
The value can only be numeric.

Class name: **org.jppf.node.policy.MoreThan**
Constructor:
```
MoreThan(String propertyName, double value)
```
Usage:
```
policy = new MoreThan("some.property", 15.50);
```

XML Element: **<MoreThan>**
Nested elements:
<Property> : name of a node property, min = 1, max = 1
<Value> : value to compare with, min = 1, max = 1
Usage:
```
<MoreThan>
  <Property>some.property</Property>
  <Value>15.50</Value>
</MoreThan>
```

12.1.9 AtLeast

Performs a test of type "property_value >= value"
The value can only be numeric.

Class name: **org.jppf.node.policy.AtLeast**
Constructor:
```
AtLeast(String propertyName, double value)
```
Usage:

```
policy = new AtLeast("some.property", 15.51);
```

XML Element: **<AtLeast>**
Nested elements:
<Property> : name of a node property, min = 1, max = 1
<Value> : value to compare with, min = 1, max = 1
Usage:

```
<AtLeast>
  <Property>some.property</Property>
  <Value>15.51</Value>
</AtLeast>
```

12.1.10 BetweenII

Performs a test of type "property_value in [a, b]" (range interval with lower and upper bounds included)
The values a and b can only be numeric.

Class name: **org.jppf.node.policy.BetweenII**
Constructor:

```
BetweenII(String propertyName, double a, double b)
```

Usage:

```
policy = new BetweenII("some.property", 1.5, 3.0);
```

XML Element: **<BetweenII>**
Nested elements:
<Property> : name of a node property, min = 1, max = 1
<Value> : the bounds of the interval, min = 2, max = 2
Usage:

```
<BetweenII>
  <Property>some.property</Property>
  <Value>1.5</Value>
  <Value>3.0</Value>
</BetweenII>
```

12.1.11 BetweenIE

Performs a test of type “property_value in [a, b[“ (lower bound included, upper bounds excluded)
The values a and b can only be numeric.

Class name: **org.jppf.node.policy.BetweenIE**
Constructor:

```
BetweenIE(String propertyName, double a, double b)
```

Usage:

```
policy = new BetweenIE("some.property", 1.5, 3.0);
```

XML Element: **<BetweenIE>**
Nested elements:
<Property> : name of a node property, min = 1, max = 1
<Value> : the bounds of the interval, min = 2, max = 2
Usage:

```
<BetweenIE>
  <Property>some.property</Property>
  <Value>1.5</Value>
  <Value>3.0</Value>
</BetweenIE>
```

12.1.12 BetweenEI

Performs a test of type “property_value in]a, b]” (lower bound excluded, upper bound included)
The values a and b can only be numeric.

Class name: **org.jppf.node.policy.BetweenEI**
Constructor:

```
BetweenEI(String propertyName, double a, double b)
```

Usage:

```
policy = new BetweenEI("some.property", 1.5, 3.0);
```

XML Element: **<BetweenEI>**
Nested elements:

<Property> : name of a node property, min = 1, max = 1
<Value> : the bounds of the interval, min = 2, max = 2
Usage:
```
<BetweenEI>
  <Property>some.property</Property>
  <Value>1.5</Value>
  <Value>3.0</Value>
</BetweenEI>
```

12.1.13 BetweenEE

Performs a test of type “property_value in]a, b[“ (lower and upper bounds excluded)
The values a and b can only be numeric.

Class name: **org.jppf.node.policy.BetweenEE**
Constructor:
```
BetweenEE(String propertyName, double a, double b)
```
Usage:
```
policy = new BetweenEE("some.property", 1.5, 3.0);
```

XML Element: **<BetweenEE>**
Nested elements:
<Property> : name of a node property, min = 1, max = 1
<Value> : the bounds of the interval, min = 2, max = 2
Usage:
```
<BetweenEE>
  <Property>some.property</Property>
  <Value>1.5</Value>
  <Value>3.0</Value>
</BetweenEE>
```

12.1.14 Contains

Performs a test of type “'property_value contains substring”
The value can be only a string.

Class name: **org.jppf.node.policy.Contains**
Constructor:
```
Contains(String propertyName, boolean ignoreCase, String value)
```
Usage:
```
policy = new Contains("some.property", true, "some_substring");
```

XML Element: **<Contains>**
Attribute: ignoreCase: one of "true" or "false", optional, defaults to "false"
Nested elements:
<Property> : name of a node property, min = 1, max = 1
<Value> : substring to lookup, min = 1, max = 1
Usage:
```
<Contains ignoreCase="true">
  <Property>some.property</Property>
  <Value>some substring</Value>
</Contains>
```

12.1.15 OneOf

Performs a test of type “property_value in { A1, ... , An }” (discrete set).
The values A1 ... An can be either all strings or all numeric.

Class name: **org.jppf.node.policy.OneOf**
Constructor:
```
OneOf(String propertyName, boolean ignoreCase, String...values)
OneOf(String propertyName, double...values)
```
Usage:
```
policy = new OneOf("user.language", true, "en", "en_US", "en_GB");
policy = new OneOf("some.property", 1.2, 5.1, 10.3);
```

XML Element: **<OneOf>**
Attributes:
ignoreCase: one of "true" or "false", optional, defaults to "false"

valueType: one of "string" or "numeric", optional, defaults to "string"
Nested elements:
<Property> : name of a node property, min = 1, max = 1
<Value> : substring to lookup, min = 1, max = unbounded
Usage:
```
<OneOf ignoreCase="true">
  <Property>user.language</Property>
  <Value>en</Value>
  <Value>en_US</Value>
  <Value>en_GB</Value>
</OneOf>
```

12.1.16 RegExp

Performs a test of type "property_value matches regular_expression"
The regular expression must follow the syntax for the Java regular expression patterns.

Class name: **org.jppf.node.policy.RegExp**
Constructor:
```
RegExp(String propertyName, String pattern)
```
Usage:
```
policy = new RegExp("some.property", "a*z");
```

XML Element: **<RegExp>**
Nested elements:
<Property> : name of a node property, min = 1, max = 1
<Value> : regular expression pattern to match against, min = 1, max = 1
Usage:
```
<RegExp>
  <Property>some.property</Property>
  <Value>a*z</Value>
</RegExp>
```

12.1.17 CustomRule

Performs a user-defined test that can be specified in an XML policy document.

Class name: subclass of **org.jppf.node.policy.CustomPolicy**
Constructor:
```
MySubclassOfCustomPolicy(String...args)
```
Usage:
```
policy = new MySubclassOfCustomPolicy("arg 1", "arg 2", "arg 3");
```

XML Element: **<CustomRule>**
Attribute: class: fully qualified name of a policy class, required
Nested element: <Arg> : custom rule parameters, min = 0, max = unbounded
Usage:
```
<CustomRule class="my.sample.MySubclassOfCustomPolicy">
  <Arg>arg 1</Arg>
  <Arg>arg 2</Arg>
  <Arg>arg 3</Arg>
</CustomRule>
```

12.2 Node properties

12.2.1 Related APIs

All node properties can be obtained using the `JPPFSystemInformation` class. This is what is sent to any execution policy object when its [`accepts(JPPFSystemInformation)`]() method is called to evaluate the policy against a specific node. As `JPPFSystemInformation` encapsulates several sets of properties, the `ExecutionPolicy` class provides a method [`getProperty(JPPFSystemInformation, String)`]() that will lookup a specified property in the following order:

4. in JPPFSystemInformation.getJppf() : JPPF configuration properties
5. in JPPFSystemInformation.getSystem() : system properties
6. in JPPFSystemInformation.getEnv() : environment variables
7. in JPPFSystemInformation.getNetwork() : IPV4 and IPV6 addresses assigned to the node
8. in JPPFSystemInformation.getRuntime() : runtime properties
9. in JPPFSystemInformation.getStorage() : storage space properties

12.2.2 JPPF configuration properties

The JPPF properties are all the properties defined in the node's JPPF configuration file.

Related APIs:

JPPFSystemInformation.getJppf()
JPPFConfiguration.getProperties()

12.2.3 System properties

The system properties are all the properties accessible through a call to `System.getProperties()` including all the -`Dproperty=value` definitions in the Java command line.

Related APIs:

JPPFSystemInformation.getSystem()
SystemUtils.getSystemProperties()
java.lang.System.getProperties()

12.2.4 Environment variables

These are the operating system environment variables defined at the time the node's JVM was launched.

Related APIs:

JPPFSystemInformation.getEnv()
SystemUtils.getEnvironment()
java.lang.System.getenv()

12.2.5 Runtime properties

These are properties that can be obtained through a call to the JDK Runtime class.

Related APIs:

JPPFSystemInformation.getRuntime()
SystemUtils.getRuntimeInformation()
java.lang.Runtime

List of properties:

`availableProcessors` : number of processors available to the JVM
`freeMemory` : estimated free JVM heap memory, in bytes
`totalMemory` : estimated total JVM heap memory, in bytes
`maxMemory` : maximum JVM heap memory, in bytes, equivalent to the value defined through the `-Xmx` JVM flag

Note: *`totalMemory` and `freeMemory` are the values taken when the node first connected to the JPPF server. They* *may have changed subsequently and should therefore only be used with appropriate precautions*.

12.2.6 Network properties

These properties enumerate all IPV4 and IPV6 addresses assigned to the JPPF node's host.

Related APIs:

JPPFSystemInformation.getNetwork()
SystemUtils.getNetwork()
java.net.NetworkInterface

List of properties:

`ipv4.addresses` : space-separated list of IPV4 addresses with associated host in the format ipv4_address|host_name
`ipv6.addresses` : space-separated list of IPV6 addresses with associated host in the format ipv6_address|host_name

Example:

```
ipv4.addresses = 192.168.121.3|www.myhost.com 127.0.0.1|localhost 254.169.0.12|
ipv6.addresses = 2001:0db8:85a3:08d3:1319:8a2e:0370:7334|www.myhost.com
```

Note: when a host name cannot be resolved, the right-hand part of the address, on the right of the "|" (pipe character) will be empty

12.2.7 Storage properties

These properties provide storage space information about the node's file system. This is an enumeration of the file system roots with associated information such as root name and storage space informaiton. The storage space information is only available with Java 1.6 or later, as the related APIs did not exist before this version.

Related APIs:

JPPFSystemInformation.getStorage()
SystemUtils.getStorageInformation()
File.getFreeSpace()
File.getTotalSpace()
File.getUsableSpace()

List of properties:

`host.roots.names = root_name_0 ... root_name_n-1` : the names of all accessible file system roots
`host.roots.number = n` : the number of accessible file system roots
For each root i:
`root.i.name = root_name` : for instance "C:\" on Windows or "/" on Unix
`root.i.space.free = space_in_bytes` : current free space for the root (Java 1.6 or later)
`root.i.space.total = space_in_bytes` : total space for the root (Java 1.6 or later)
`root.i.space.usable = space_in_bytes` : space available to the user the JVM is running under (Java 1.6 or later)

Example:

```
host.roots.names = C:\ D:\
host.roots.number = 2
root.0.name = C:\
root.0.space.free = 921802928128
root.0.space.total = 984302772224
root.0.space.usable = 921802928128
root.1.name = D:\
root.1.space.free = 2241486848
root.1.space.total = 15899463680
root.1.space.usable = 2241486848
```

12.3 Execution policy XML schema

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
  JPPF.
   Copyright (C) 2005-2010 JPPF Team.
  http://www.jppf.org

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

     http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
-->

<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:jppf="http://www.jppf.org/schemas/ExecutionPolicy.xsd"
  targetNamespace="http://www.jppf.org/schemas/ExecutionPolicy.xsd"
  elementFormDefault="unqualified"
  attributeFormDefault="unqualified"
>

  <element name="ExecutionPolicy" type="jppf:OneRuleType"/>

  <group name="Rule">
    <choice>
      <element name="NOT" type="jppf:OneRuleType"/>
      <element name="AND" type="jppf:TwoOrMoreRulesType"/>
      <element name="OR" type="jppf:TwoOrMoreRulesType"/>
      <element name="XOR" type="jppf:TwoOrMoreRulesType"/>
      <element name="LessThan" type="jppf:Numeric2Type"/>
      <element name="AtMost" type="jppf:Numeric2Type"/>
      <element name="MoreThan" type="jppf:Numeric2Type"/>
      <element name="AtLeast" type="jppf:Numeric2Type"/>
      <element name="BetweenII" type="jppf:Numeric3Type"/>
      <element name="BetweenIE" type="jppf:Numeric3Type"/>
      <element name="BetweenEI" type="jppf:Numeric3Type"/>
      <element name="BetweenEE" type="jppf:Numeric3Type"/>
      <element name="Equal" type="jppf:EqualType"/>
      <element name="Contains" type="jppf:ContainsType"/>
      <element name="OneOf" type="jppf:OneOfType"/>
      <element name="RegExp" type="jppf:RegExpType"/>
      <element name="CustomRule" type="jppf:CustomRuleType"/>
    </choice>
  </group>

  <complexType name="OneRuleType">
    <sequence>
      <group ref="jppf:Rule"/>
    </sequence>
  </complexType>

  <complexType name="TwoOrMoreRulesType">
    <sequence minOccurs="2" maxOccurs="unbounded">
      <group ref="jppf:Rule"/>
    </sequence>
  </complexType>
```

```xml
<!-- test of type "property_value is less than value" -->
<complexType name="Numeric2Type">
  <sequence>
    <element name="Property" type="string"/>
    <element name="Value" type="double"/>
  </sequence>
</complexType>

<!-- test of type "property_value is in range [a, b]" -->
<complexType name="Numeric3Type">
  <sequence>
    <element name="Property" type="string"/>
    <element name="Value" type="double" minOccurs="2" maxOccurs="2"/>
  </sequence>
</complexType>

<!-- test of type "property_value is equal to value" -->
<complexType name="EqualType">
  <sequence>
    <element name="Property" type="string"/>
    <element name="Value" type="string"/>
  </sequence>
  <attribute name="valueType" use="optional" default="string">
    <simpleType>
      <restriction base="string">
        <enumeration value="string"/>
        <enumeration value="numeric"/>
        <enumeration value="boolean"/>
      </restriction>
    </simpleType>
  </attribute>
  <attribute name="ignoreCase" type="jppf:TrueFalse"
   use="optional" default="false"/>
</complexType>

<!-- test of type "property_value contains substring" -->
<complexType name="ContainsType">
  <sequence>
    <element name="Property" type="string"/>
    <element name="Value" type="string"/>
  </sequence>
  <attribute name="ignoreCase" type="jppf:TrueFalse"
   use="optional" default="false"/>
</complexType>

<!-- test of type "property_value is one of { value1, ... , valueN } " -->
<complexType name="OneOfType">
  <sequence>
    <element name="Property" type="string"/>
    <element name="Value" type="string" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="valueType" use="optional" default="string">
    <simpleType>
      <restriction base="string">
        <enumeration value="string"/>
        <enumeration value="numeric"/>
      </restriction>
    </simpleType>
  </attribute>
  <attribute name="ignoreCase" type="jppf:TrueFalse"
    use="optional" default="false"/>
</complexType>
```

```
  <!-- test of type "property_value matches regular_expression " -->
  <complexType name="RegExpType">
    <sequence>
      <element name="Property" type="string"/>
      <element name="Value" type="string"/>
    </sequence>
  </complexType>

  <simpleType name="TrueFalse">
    <restriction base="string">
      <enumeration value="true"/>
      <enumeration value="false"/>
    </restriction>
  </simpleType>

  <complexType name="CustomRuleType">
    <sequence>
      <element name="Arg" type="string" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="class" type="string"/>
  </complexType>

</schema>
```

13 Deployment and run modes

13.1 JPPF Driver

First you will need to obtain the driver package from the JPPF download page : **JPPF-x.y.z-driver.zip**.
Unzip the file JPPF-x.y.z-driver.zip in a location where you intend to run the JPPF server from.
This zip file contains all the binaries for running the server only.
To run the driver: go to the JPPF-x.y.z-driver folder and type "**ant run**".

13.1.1 JPPF Driver as a Windows Service

A JPPF driver can be run as Windows Service using the Java Service Wrapper available at Tanuki Software.
The JPPF-x.y.z-driver.zip distribution and above are prepared for this installation.

To install:

- download the Java Service Wrapper for your platform and copy the files wrapper.exe, wrapper.dll and wrapper.jar to the JPPF driver install directory
- edit config/wrapper-driver.conf file, check that the setting for wrapper.java.command is valid (either the PATH environment must contain a Java 5 JRE, or the installation directory must be entered here)
- run the **InstallDriverService.bat** file to install the JPPF node service
- run the **UninstallDriverService.bat** file to uninstall the JPPF node service

13.1.2 JPPF Driver as a Linux/Unix daemon

The JPPF driver can be run as a Linux/Unix daemon using the Java Service Wrapper available at Tanuki Software.
The JPPF-x.y.z-driver.zip distribution and above are prepared for this installation.

To install:

- download the Java Service Wrapper for your platform and copy the files wrapper, libwrapper.so and wrapper.jar to the JPPF node install directory
- don't forget to set the executable bit for the JPPFDriver and wrapper script/executable
- edit config/wrapper-driver.conf file, check that the setting for wrapper.java.command is valid (either the PATH environment must contain a Java 5 JRE, or the installation directory must be entered here)
- open a terminal in the JPPF driver root install directory
- to run the driver as a daemon: **./JPPFDriver start**
- to stop the driver: **./JPPFDriver stop**
- to restart the driver: **./JPPFDriver restart**

13.2 JPPF Node

First you will need to obtain the node package from the JPPF download page : **JPPF-x.y.z-node.zip**.
Unzip the file JPPF-x.y.z-node.zip in a location where you intend to run a JPPF node on.
This zip file contains all the binaries for running a node only.
To run the node: go to the JPPF-x.y.z-node folder and type "**ant run**".

13.2.1 JPPF Node as a Windows Service

The JPPF node can be run as Windows Service using the Java Service Wrapper available at http://wrapper.tanukisoftware.org/. The JPPF-x.y.z-node.zip distribution and above are prepared for this installation.

To install:

- download the Java Service Wrapper for your platform and copy the files wrapper.exe, wrapper.dll and wrapper.jar to the JPPF node install directory
- edit config/wrapper-node.conf file, check that the setting for wrapper.java.command is valid (either the PATH environment must contain a Java 5 JRE, or the installation directory must be entered here)
- run the **InstallNodeService.bat** file to install the JPPF node service
- run the **UninstallNodeService.bat** file to uninstall the JPPF node service

13.2.2 JPPF Node as a Linux/Unix daemon

The JPPF node can be run as a Linux/Unix daemon using the Java Service Wrapper available at Tanuki Software.
The JPPF-x.y.z-node.zip distribution and above are prepared for this installation.

To install:

- download the Java Service Wrapper for your platform and copy the files wrapper, libwrapper.so and wrapper.jar to the JPPF node install directory
- don't forget to set the executable bit for the JPPFNode and wrapper script/executable
- edit the config/wrapper-node.conf file, check that the setting for wrapper.java.command is valid (either the PATH environment must contain a Java 5 JRE, or the installation directory must be entered here)
- open a terminal in the JPPF node root install directory
- to run the node as a daemon: **./JPPFNode start**
- to stop the node: **./JPPFNode stop**
- to restart the node: **./JPPFNode restart**

13.2.3 JPPF Node in “Idle Host” mode

A node can be configured to run only when its host is considered idle, i.e. when no keyboard or mouse activity has occurred for a specified time. This requires additional libraries that must be downloaded separately due to licensing compatibility concerns, and used to compile and build a node add-on. Fortunately, we have automated the download and build process, to lae it as easy as possible.

To install and configure a node in idle mode:

- download the JPPF samples pack
- unzip the **JPPF-x.y.z-samples-pack.zip** anywhere on your file system
- open a command prompt or shell console in **JPPF-x.y.z-samples-pack/IdleSystem**
- run the build script: “**ant jar**”, or simply “**ant**”. This will download 2 jar files “**jna.jar**” and “**platform.jar**” and create a third one “**IdleSystem.jar**”, into the **IdleSystem/lib** directory.
- when this is done, copy the 3 jar files IdleSystem/lib into your node's library directory “**JPPF-x.y.z-node/lib**”
- to configure the node to run in idle mode, open the file “**JPPF-x.y.z-node/config/jppf-node.properties**” in a text editor and create or edit the following properties:
  - **jppf.idle.mode.enabled** = true to enable the idle mode
  - **jppf.idle.timeout = 6000** to configure the time of keyboard and mouse inactivity before considering the node idle, expressed in milliseconds
  - **jppf.idle.poll.interval = 1000** to configure how often the node will check for inactivity, in milliseconds
  - **jppf.idle.detector.factory = org.jppf.example.idlesystem.IdleTimeDetectorFactoryImpl** please do not change this!

- when this is all done, you can start the node and it will only run when the system has been idle for the configured time, and stop as soon as any keyboard or mouse input occurs